On an Extension of Functional Languages for Use in Prototyping

CIP-Gegevens Koninklijke Bibliotheek, Den Haag

Dorp, Hendrik Dick van

On an extension of functional languages for use in
prototyping / Hendrik Dick van Dorp. - [S.l. : s.n.]. - Ill.
Proefschrift Enschede. - Met lit. opg. - Met samenvatting
in het Nederlands.
ISBN 90-9007594-1
Trefw.: programmeertalen.

Het contact met Twente is deels mogelijk gemaakt door een regeling in het kader van de "Studiefaciliteitenregeling Rijkspersoneel" d.d. 26 maart 1984. Het drukken van dit proefschrift is financieel mogelijk gemaakt door de Stichting BAZIS te Leiden. De geëntameerde vervolgonderzoeken zijn deels gefinancierd door de heer en mevrouw Van Dorp-Bos te Bussum.

On an Extension of Functional Languages for Use in Prototyping

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de Rector Magnificus,
prof. dr. Th.J.A. Popma,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op donderdag 22 december 1994 te 16.45 uur.

door

Hendrik Dick van Dorp

geboren op 8 september 1946
te Hilversum.

Dit proefschrift is goedgekeurd door de promotoren:
prof. dr. ir. A.J.W. Duijvestijn en prof. dr. A. Ollongren,
en de assistent-promotor: dr. G.F. van der Hoeven.

Preface

The seeds of this dissertation originated in the *Vertalergroep* environment in the late sixties. There were long discussions on the rigour of ALGOL and the pleasures of LISP, "the queen of programming languages" (Strachey), on the sturdiness of syntax and the charm of semantics,[1] and on the Backus–Naur Form for fully describing a language's outside, and five small functions for completely describing a language's inner machinations. These discussions set me on a functional, declarative, course: the important thing being what has to be achieved, not how it has to be done, that is the other, orthogonal course, an imperative one.

Other ideas that were set to ripe in those days are: 'the general confusing of efficiency with efficacy', or an effect does not have to be the desired effect, "... our unfortunate obsession with form over content, which Minsky deplored in his Turing Lecture ..."[2] (and in this obsession forgetting to be really formal), and the 'one-architect rule' with its corollary, the 'mythical man month'.

These discussions might also have contributed[3] to the character of the language used in these pages, as is so aptly described by Willard Orman Quine:[4] "Good purposes are often served by not tampering with vagueness. Vagueness is not incompatible with precision. [...] Also, vagueness is an aid in coping with the linearity of discourse. An expositor finds that an understanding of some matter A is necessary preparation for an understanding of B, and yet that A cannot itself be expounded in correct detail without, conversely, noting certain exceptions and distinctions which require prior understanding of B. Vagueness, then, to the rescue. The expositor states A vaguely, proceeds to B, and afterwards touches up A, without ever having to call upon his reader to learn and unlearn any outright falsehood in the preliminary statement of A."

Heeding the wise advice of Professor Max Breedveld just before my graduation, "Being an engineer, you should first do some practical work before thinking of a dissertation", it took quite some time before I started on it. However, as LISP's song ended in day to day business, its melody lingered on. So in the early eighties reading on functional languages and their practical use started, when I found in Professor Arie Duijvestijn a very interested, but busy, audience. The ensuing discussions in the mid-eighties with him and Gerrit van der Hoeven, Stef Joosten and Henk Kroeze, on linking Twentel with the outside world, in particular with a normal-sized

[1] "See the little phrases go, / Watch their funny antics, / The men that make them wriggle so, / Are teachers of semantics." Quoted from memory; source lost (*Comm ACM*, Tennent, ...).

[2] R.W. Floyd, "The Paradigms of Programming", *Comm ACM* **22**(8) (Aug 1979): 455–460, (1978 ACM Turing Award Lecture, 4 Dec 1978), also on pp 131–142 of R.L. Ashenhurst and S. Graham (eds) *ACM Turing Award Lectures — The First Twenty Years, 1966 – 1985.* ACM Pr Anthology Ser, ACM Pr, New York / Addison–Wesley, Reading MA, 1987. In this book one can also find Marvin Minsky's 1969 Turing Award Lecture, on pp 219–242, first published as "Form and Content in Computer Science", *J ACM* **17**(2) (Apr 1970): 197–215.

[3] Another contribution might be a too early exposure to the *Bommel* saga, especially to the vocabulary of *Terpen Tijn*.

[4] W.V.O. Quine, *Word and Object.* MIT Pr, Cambridge MA, 1960 (11^{th} pr, Apr 1979), p 127.

relational database ("Backus meeting Codd"), set the course for the work to be done. I thank them for these stimulating discussions.

For various reasons however, it was Lewis Carroll's *The Hunting of the Snark* rather than his *Alice*, that lurked in the early footnotes: "Then you softly and suddenly vanish away, and never be met with again". However, just before Professor Arie Duijvestijn's Valedictory Lecture on 3 November 1989, it became clear that my Snark did not have to be a Boojum after all. So my debt to a great many people can finally be written.

It is a paradoxical situation described by Duijvestijn in saying: "You have to do it all by yourself." You do, but it can't be done. My own set of helping bystanders (none of them owning a Cretan barbershop) runs as follows, to all of them I must express my gratitude for helping, encouraging, teaching and stimulating me in whatever way they did:

Anticipating the first lines of Section 2.1, Margreet, Eveline and Henriëtte, enduring my bodily presence when I was away on my quest for 'referential transcendency', supplied me with a warm, supporting environment, without that this would not have been possible. They urged me on when I needed so, and this in a very direct way, showing me the relativity of it all. Yet, how much they valued this work shows in the sacrifice of most of their 1994 summer holidays;

As is clear from above Professor Arie Duijvestijn, my *promotor*, has been with this for well over a decennium. I owe him for this patience. He taught me the real meaning of 'efficiency', constraint, practicality and graphs, squares, *&c*, and once remarked on the work of the programmer: "An algorithm must not need oral explanation, it should speak for itself";

Though I appreciate and understand the 'Leyden Prohibition' regarding one's *promotor*, I do not choose to follow it: Professor Alexander Ollongren contributed many a incisive comment on structure, and clarified things by apparently simple questions, in short he added to the quality of my work. As a side effect of the process, he showed me the way to philosophy in *informatica*, marking the start of Chapter 1;

Hans van Berne was in it longer than anyone else, because of his early influence on my education. Later he gave his precise comments on all my drafts, found a spurious bracket and kept the link to language and meaning in general by his witty marginal notes and by his laugh with a comment as follow up. Finally, he saved a large part of this work by proposing a shift in attention;

Gerrit van der Hoeven, always showing the right amount of scientific doubt in a minimal number of words with a very good sense of humour, spotted immediately when 'one doesn't do it like that' in a maze of words;

Wim van Dam, (Slavonic) linguist, supervised my English and my comma's and proved an enthusiastic reader — everything twice —; yet his influence stretched far beyond the English, though not as far as his celestial paraphysics;

Gerard te Meerman asked the question, "Why?", thus urging to take my position, and critically commented on large parts of this work;

Henk Kroeze and Martin van Hintum supplied me with Twentel and were present when I needed advice and error corrections;

Willy Schulte constructed some solutions that are mentioned in Chapter 5, also helped in other ways, besides, he was there to talk programming, birds and other matters;

My parents, realising the importance of being functional, sponsored part of my further endeavours in this area;

Paul Hendriks, though busy, could read early drafts and suggested many improvements;

Vanja Rejger suggested a very good idea, its realisation definitely improved the form;

Stef Joosten, Professors Gerrit Blaauw, Peter Naur and John Lansdown, Arthur Elias, Klaas van den Berg, Peter Asveld and Ahmed Patel provided me with literature, references, background material and supporting comments;

And my colleagues within SPIRIT, especially Laurens Rijneveld who took over during all my holi-(writing)-days, and John Caubo and Arend Halfmouw who let me read their books;

Finally, there are many people belonging to the class that Joost Engelfriet describes as "those whose ideas one uses unwittingly after many, many years", I also acknowledge their contribution. This acknowledgment also holds for the class of people "whose contributions one uses and forgets to mention".

It is of course a *gotspe* to produce a dissertation on a subject related to functions and 'referential transparency' with TEX. For my gratitude to Jan Vanderschoot with respect to this language, I refer to the Colophon. Nevertheless, learning another exotic language — far from the mainstream — (like LISP, Snobol, TRAK and Prolog) with the full power of a Turing machine (as the common, business oriented languages cannot offer you, because they do not have recursion) turned out to be another good reason for procrastination. Using this language is infinitely better than using the normal text processing programs that Dijkstra characterised as: "Their manuals are fatter than a textbook for theoretical physics".

Concluding, we should not forget the ubiquitous note on typing errors and such: these are all mine; some however, are intentional, almost with malice aforethought. I did so in order to avoid using the ™ or © sign on intellectual property, which I fully acknowledge, not by silly signs, but by intense intention.

Summary

In a computerised society the user needs a computer because the solution of some of his problems requires a computer. The problem might be of such a novel or uncommon nature, that he has to engage a programmer to take care of the search for the solution. The first part of this dissertation deals with aspects of the interaction between user and programmer when such a user problem is to be solved.

To solve the user problem the effects of the language mismatch between these two kinds of people must be minimised: this requires a discussion of problem-domain language versus a formal, more abstract language. We state that prototyping is essential to quickly overcome this language mismatch, to clarify the problem and to seek out variations in the way the solution is reached.

The prototype must be constructed, so design principles are discussed. The main principle is *consistency* of design.

Requirements for a language to describe the executable prototype are stated during the discussion of the user-programmer interaction. Among these language requirements are: a high level of abstraction in describing the problem, amenability to formal (mathematical) notation, reduction of complexity, thereby strengthening modularisation in the design process, referential transparency and related mathematical properties facilitating correctness proofs and reuse of functions, and easy communication with existing systems and solutions.

This kind of communication is necessary in order to decrease the turn around time of the prototypes in the problem solving cycle. Efficiency in this process can be increased by using existing solutions (efficient use of resources), by using easy notations for subproblems that are unwieldy to the description facilities of a functional language (efficient use of programmer time) and by using other systems that yield efficient solutions for subproblems (use of efficient resources). There are more ways to gain efficiency, e.g., by parallelisation of computing power or by program transformations.

Thanks to their strong mathematical foundation functional languages turn out to fulfill these language requirements, except for the necessary communication with existing solutions in the outside world. To enlarge the potential use of functional languages this last requirement must be met, which is the goal of the second part of this dissertation.

In the intermediate part we treat some basics of functional languages: the notion of *function*, characteristics and use of these languages, the lambda calculus as their computational model, and combinatory logic and graph rewriting as implementation vehicle of the (untyped) functional language that was chosen as demonstration language, Twentel.

The second part introduces the solution to the outside world communication problem for functional languages, a 'trapdoor'. A variety of ways to use such a solution are treated. The trapdoor

is implemented in Twentel and its functioning is demonstrated for communication with two kinds of computing agents: a graph identificator (a Pascal program) and a SQL server (a Twentel prototype of a relational database). Communication with the outside world computing agents is done by means of an 'Intermediate List Structure', a free format, two dimensional data structure based on list representation. The model for converting data in the outside world to and from the 'Intermediate List Structure' is presented.

Some practical problems are discussed, and a solution is given for the problem resulting from the interaction between 'lazy evaluation', 'bounded buffer size' and 'trapdoor communication'. Finally we present a short discussion of different implementation models (one / two machines; MS-DOS and Unix) of the trapdoor and on the practical problems encountered in the realisation of the trapdoor.

The dissertation ends with some ideas for further research, especially the possible communication of functions as first-class citizens.

Contents

Aan L. Verhoeven,
die erin geloofde,
en zijn (klein)dochter(s),
die het mogelijk maakten.

"Als je durft aanvaarden en beleven dat je meer bent dan je rationele verstand en dat er meer is dan de wetenschap hard *kan maken, blijft het kind in je zich verwonderen."*

— Marten Toonder, 1985.

Chapter 1

Introduction

'Programming is not an art' is the thesis which starts this dissertation. Its value will be elaborated in the first Section, where we prepare for the use of aesthetics in the second Chapter. The main theme of this dissertation unfolds itself in the second Section: language. In this Introduction we will not confine ourselves to programming languages: we start with language — a communication vehicle — as used in, and around, the field of computer science in a very general sense. We hold that the way language is used is one of the main causes of the difficulties encountered in the delivery of programming products to the world. The third Section starts with an analogy to the ordered set of rules which comprises a computer program. Links to other disciplines of science and scholarship are given, links that are closed in the aesthetics part of Chapter 2 where design principles are given. A further elaboration on the theme of this dissertation and an overview of its supporting material ends the Introduction.

1.1 Programming and Art

Programming is not an art. One programs to get things done. Art is for our aesthetic excitable senses, thus creating emotions: programming is for easing the running of our daily lives. In creating a piece of art the artist, apart from his or her creative mind, uses tools and, perhaps, some raw materials; so a poet uses language to evoke emotions with the reader. In programming, the programmer uses language to give orders to the computer to evoke a desired effect. We use the term 'programmer' in the all-encompassing meaning Edsger Dijkstra gave it in 1972 [8]: someone who approaches, in a humble way, the intellectual challenge of "designing classes of computations that will display a desired behaviour", but who is *not* the mere maker of programs. The 'language' was defined by Peter Naur as follows: "A programming language is a set of conventions for the actions a human computer user has to take to make the computer do what he wants in a certain context" [23]. The 'desired effect' then, is the solution of the problem, the things to get done.

Nevertheless, there are aspects of programming or computing, which are able to stir up very strong emotions; we consider these 'holy wars' known. There is only one exception we would like to mention, regarding the way we give these orders to the computer [16,17]. Donald Knuth states that programming is an art, "an aesthetic experience much like composing poetry or music" [16], as opposed to science. "Science is knowledge which we understand so well that we can teach it to a computer; and if we don't fully understand something, it is an art to deal with" [17]. We often do not quite understand the problem we are supposed to write a program,

a solution, for, nor do we always have the exact algorithm needed to solve a problem. So looking at art the way Knuth does, the mere use of prototyping in program development and heuristics in algorithms, incorporates programming with the arts, which we believe, is not the right place for programming.

Knuth also introduced 'literate programming' [18], the idea that programs can and should have elegance and style. He expresses that the structure of a program may be thought of as a web that is made up of many interconnected pieces, program statements together with their documentation, which should be read as a normal, readable text. Such a text then can be subjected to literary criticism, which is a way of looking at art, whatever criteria one applies.

In true dialectic fashion there is no contradiction between Knuth's view and our view. Knuth tends toward the art of creating programs, we tend toward the science of creating programs, though for the time being we will occupy ourselves with the engineering of programming, based on the science of computing. So the contradiction disappears when one looks upon programs as tools with an observable outside, an exterior which must be given form. Apart from characteristics as smooth and accurate functioning, or a user interface, there is the underlying program text which can be given a good structure and style. But we still hold that programming is not an art, as it serves more purposes than mere aesthetic and cultural ones (apart from using pieces of art in showing off prestige, wealth and power); its products, however, can exhibit, apart from their function, a certain beauty, as the beholder of steam engines and the Boulder dam, vintage automobiles and a Citroën DS, Dudok and Le Corbusier, will agree.

In using 'art', the notion from the first sentences, and contrasting it with our view of programming, we exploited the ambiguity of the word, as a language oriented dissertation must be able to deal with. We contend that someone of Knuth' stature, stressed the liberal arts related notion as a pun (on first sight), but started — with others at the same time — the transition from art, the craft related notion, to science with his multiple-volume *The Art of Computer Programming*, as he remarked on the purpose of this Work: "to train the reader in the various skills which go into a programmer's craft" [16]. To be more precise, we think that he must have known of Gower's revision of Fowler's *A Dictionary of Modern English Usage*, where one finds [12]:

> " ... the term science is extended to denote a department of practical work that depends on the knowledge and conscious application of principles; an art, on the other hand, being understood to require merely knowledge of traditional rules and skill acquired by habit."

Both meanings of 'art' will reappear in later Chapters: the creation of form in design will be discussed (liberal arts related), as well as the move from the craft of programming in the older days toward the use of scientific principles in a special kind of programming language today. Having resolved the dialectic, we should not proceed without acknowledging the existence of " 'the aesthetic school of computer science' represented by Dijkstra, Hoare and Gries" [31]. Meticulous care for details, correctness of what has been written, and elegance of algorithms, are some of the characteristics of this school of programming. However, twenty years after the Turing Award Lecture *The Art of Computer Programming* [17], we are still traveling from 'art' to 'science', and we have not arrived yet.

1.2 Computing and Language

Opening with an upbeat in aesthetics, we follow suit with the traditional way of starting an argument: offering a suitable quotation from a respectable source. We use a quote from Alan Perlis' keynote speech at the tenth Anniversary Symposium at the Computer Science Department of Carnegie-Mellon University in 1975 [28]:

> "I think that it's extraordinarily important that we in computer science keep fun in computing. When it started out, it was an awful lot of fun. Of course, the paying customers got shafted every now and then, and after a while we began to take their complaints seriously. We began to feel as if we really were responsible for the successful, error-free, perfect use of these machines. I don't think we are. I think we're responsible for stretching them, setting them off in new directions, and keeping fun in the house. Fun comes in many ways. Fun comes in making a discovery, proving a theorem, writing a program, breaking a code. Whatever form or sense it comes in I hope the field of computer science [...] never loses its sense of fun. Above all, I hope we don't [...] become missionaries. Don't feel as if you're Bible salesmen. The world has too many of those already. What you know about computing other people will learn. Don't feel as if the key to successful computing is only in your hands. What's in your hands, I think and hope, is intelligence: the ability to see the machine as more than when you were first led up to it, that you can make it more."

The theme of this dissertation can be derived from this quotation (though we have to read 'fun' as the generally accepted abbreviation for 'function' [10]): "... fun ... customers ... stretching ... new directions ... and making it more." So the theme contains functions, which, together with the data (to which the functions are applied), comprise the science of computing, or else, computer science. Then we pursue the extension of the current capabilities of the computer, so making it more, and giving directions for this extension. Not for pure scientific reasons, as *l'art pour l'art,* but ultimately for giving the user, or customer, a sense of being well-served. The error-free and perfect use of the computer might not be within our powers. As scientists we must use existing and proven scientific principles in order to produce methods that produce error-free and perfect programs: quality for the user. On the engineering side we should mention that the mere perfection of administrative procedures in the production of software (the use of standard, 'clean room' hardware production methods at Hewlett-Packard) already yields very high quality programs in terms of absence of errors [13]. Of course, we must not forget the possibility of error-free and perfect use of our programs, not so much through the above perfection of processes, but as a result of applying creative science. (However, we should not forget the fun, as it provides the motivation; and we should not forget that humour, in most cases, puts a situation into perspective, and as such it relates to abstraction.)

One of the first purposes of the computer was the routine production of ballistic tables [11], i.e., calculating and printing functions *in extenso*: nowadays we can use the computer to visualise functions by means of symbolic computational programs. Though we might agree that the word 'function' is correctly used in both cases in the above sentence, the involved 'computing', however, — letting the computer perform its ordained tasks — raises some problems.

Back in 1967, Joost Engelfriet, then at the staff of the Chair of Theoretical Computer Science at the *Technische Hogeschool Twente* (University of Twente), made a very pointed remark regarding the problems of a language designer, when dampening the enthusiasm of two students (and other members) of the *Vertalergroep* (Compilergroup: occupied with designing programming languages and their compilers) of this technical University. The latter chortled in their joy, regarding the replacement of the IBM/360-30 by an IBM/360-50, because from then on, everything could be computed much, much, faster. "Maar ik *hèb* helemaal niets uit te rekenen!" Joost, seriously, said.[1]

These language designers (or, rather, compiler constructors) thought it very strange that a theorist, earning his living with building theories on how to look at the computer and its processes, was not involved at all with the very apparatus which was supposed to be the subject of his studies. So the language designers perceived a gulf between the theorists and themselves, not realising that there also existed a gulf between them and the common programmer, who was using their languages and compilers. Not to mention the gulf which existed between these programmers and the users. But, alas, in the last twenty-five years, none of these gulfs has been truly bridged. Islands, separated by these gulfs, can be imagined as a lattice, a kind of network, with the theorists at one end and the users at the other end. The proper terminology of these ends prepares us for the Abstract Machine: top and bottom. But on all these islands, and even on the farthest island, the shore, where 'real world' solutions are brought out from the computer room, one uses the computer to get things done. Everybody 'computes', uses a kind of calculus, a finite set of rules, to get the desired results.

Let us paint this island world in black and white. Admittedly, in reality one might come across colours in this picture, but for the sake of the argument we keep it in black and white. Not only the group of the language designers as applied computing scientists, finds itself between the theorist and the programmer. The area in between contains also other groups of applied scientists, those in the field of databases, networking, artificial intelligence, and operating systems, to mention but a few specialisations. Generally speaking, none of these groups are well versed in the problem domain of the other groups — the Ivory Tower symptom, which befalls every expanding field of science. The database man doesn't know about right, or tail-recursion, the language man doesn't know about collision detection, the network man doesn't know about magic sets (and no one knows about Turing Machines), but, worse even, and applicable to all these groups, they don't know about the everyday problems the programmers who use their products are confronted with. The same, however, applies to these programmers: they don't know about the everyday problems users of their products are confronted with in daily using their products.

During the 1993 Perlis symposium at Yale, Ehud Shapiro said [29]: "The evolutionary construction of layers of abstraction may be the only real [...] output of computer science." This abstraction layering can also be described in the form of the Abstract Machine metaphor. In this metaphor, one distinguishes several layers, called machines, one atop each other, each machine being an abstraction of the one beneath it. One starts at the top, using the topmost machine for one's needs: describing the problem to be solved, and one addresses it in its own language: its directions for use, found in the machine manual. This machine in accomplishing its tasks uses the machine beneath it through *that* machine's directions for use. And so on, until the chain

[1] "But, really, I don't have anything to compute at all!" — transl HD.

of directions reaches the bottom. The bottom machine does all the work, and the intermediate ones are there for simplifying the way of addressing problem descriptions.

In this description of the Abstract Machine metaphor one recognises the way programming started. The first machines, the electronic stored program computers of the Turing-Von Neumann model, were programmed in machinecode, directly written for the hardware. Then came the symbolic assembly language in which one recognised concepts and constructs from the underlying real hardware (every different machine had its own assembler), but one did not have to write all the details, many of them were hidden by symbolisation. These assembler statements were translated into the proper machinecode of their hardware. After that came the machine independent, so called 'high level', languages, in which only a highly abstract notion of the underlying hardware was kept, and where the written high level language was translated into the symbolic assembly language for the hardware the machine independent language program was supposed to run on. Quite another example of such a stack of abstract machines is the OSI layer of networking protocols [14].

Many reasons can be found for the programmer not knowing the users' problems, apart from the very simple explanation like plain 'not asking', or not asking the right questions. If we take two deeper explanations, the programmer not having shared the same experiences as the user, or both not being educated in the same field, these can be reduced to one underlying cause: the 'cultural' difference between user and programmer, manifesting itself in not using the same language. It is common knowledge that a programmer does not communicate very well with the general user. He is inclined to use his own terms, and (if pressed, to be true), to explain them in terms of the user, but in the first place, he does not question the use of his own terms at all. The user is not interested in how his problem is solved, which is implied by the programmer using his own terms. He is only interested in what the programmer has made, the solution of his problem. So if we look upon this situation as having the layered structure of the Abstract Machine metaphor, we see a person in a certain layer, the programmer, talking to persons in lower layers, the users, in his own language, instead of in the language of the persons of the lower layer. The Abstract Machine metaphor is not followed, which might account for many problems encountered in software development in the past years.

Where at one end, users deal with everyday problems, the theorists, at the other end, deal with abstract things, like complexity of algorithms, computability of functions, and structures of languages. In between, the language designers deal with compiler construction, program transformation and partial evaluation. And all these groups suffer to a certain extent from this language mismatch between them, almost from the moment the islands started to drift apart.

The only (new) development in this picture is the growing of a 'software engineering' layer between the applied scientists and the programmer [5, 25]: facilitating the programmer in using the products made by the applied scientists, in order to alleviate problems and satisfying wishes in the user environment. At the same time, it gives the applied scientist these new facilities too, thus facilitating, and accelerating, the development of new products. The word 'engineering' in this notion, not 'science', sets it apart from the other islands, that are more oriented toward the scientific method. Software engineering is still more geared toward the craft of software construction than to the science of it. As such it might be better to change the metaphor, and call software engineering the oil in the machine, a thin layer (again) necessary for smooth functioning of the machine.

Another development, apart from the growth in substance of the different sub-disciplines, is the growing computational awareness of the user, or his rapidly diminishing computer illiteracy. This translates itself into a growing interference of the user with the programmer. We leave it as an open question, whether the user is now talking the programmer's language, or is merely talking about new possibilities in his own domain. As a reminder of the intrinsically strange formality of the programmer's language, the programmer might use words like, 'group', 'ring', 'field', and 'signature', words every user will understand, though he might feel a little wondered about these being used somewhat out of context.

This is an asymmetrical situation. It is not an exchange between equals, like in barter, it is the user coming to the programmer, who holds — as he (with ambiguity aforethought) thinks — all the keys to a successful solution of his problems. One can observe here the High Priest syndrome. In the early days of computing 'the programmer holding the keys to a solution' was transformed, through common psychological and sociological forces, into the High Priest guarding the Computer. The only way the user could communicate with the Computer was through its High Priests. Here it is that we find again the 'higher – lower' from the Abstract Machine metaphor, but now it has acquired real world properties.

So one of the main problems the programmer has to deal with in his relation to the user, is the way the user talks about his problems, using idiom from his own domain. When the programmer specifies means to obtain solutions to the user problems, he uses his own (formal) language, and its idiom. However, in this process of writing the specifications, the user language and the user idiom is forced into the (far more formal [2]) language and idiom the programmer needs to describe his solution. User and programmer cannot have a consensus on this mutual description, because consensus is based upon potentially equally informed people. As educating the user in computer science is not a feasible way of reaching this consensus, the programmer, as we will see, has to learn the user language. Dijkstra voiced both the formal aspect and the communicating with users as follows: "Besides a mathematical inclination, an exceptionally good mastery of one's native tongue is the most vital asset of a competent programmer."

An example of forcing the problems of the programmer with this description into the user domain is the user interface of the classic menu-driven software. Such software used to be built as nested subroutines, because the programmer saw a logical path from start to goal and back again, all in orderly fashion. For a novice user this guided tour is a help, but an experienced user needs shortcuts, not only when starting, also on exit. And as the `goto` was considered harmful, the user had to endure the hard-coded maze. On exit however, he used the `Ctrl-C`. The user was forced into the programmer's problem — but found an escape.

Another example deals with the way codes are used in everyday use of computer systems. In the early days of (administrative) computing, codes were used instead of natural language as space was scarce (in core, on disk, and on the coding (!) forms), and we did not quite understand free text processing. This habit of using codes persisted. However, the use of codes or numbers, instead of proper names of people, or description of items, is acceptable as it reduces input labour, but to declare those codes sacred, so completely blocking any other way of entering data or retrieving data on the subject, is a very common practice still in daily use wherever

[2] 'Tis the voice of the luser; I heard him declare: "From ghoulies and ghosties / and longleggety beasties / and things that go bump in the night, / Good Lord, deliver us!" which is an ancient Cornish litany, as noted on page 208 of The Revised Report [33].

information systems are used: "It's so convenient" or, "It's so efficient". Again the user must use his system in a way that is forced upon him by the programmer's problems, not because of his own free will. Apart from this argument against the proliferation of codes in human communication, there is also a risk of using codes instead of natural language. Codes have only form, no content, so it takes an extra translation to get at the content of a code, which action is often omitted for efficiency reasons. Guarding against error in communication is now extremely difficult, as the content of the message is absent.

One of software engineering's purposes is precisely the opposite, namely to facilitate the use of products of the higher layer. Theory must follow practice, not the other way round ("There is more hidden in practice than is dreamt of in your theory, dear programmer"). The fact that the user problem is written down in a more or less formal manner, as a kind of specification, is no guarantee that it will be solved, as there is no formal way of relating the specification to the 'real world' problem. Through building theories, (and trying to falsify them) we are gradually building our edifice of knowledge, and making theories involves abstraction from the bare facts. But abstraction as such doesn't build a theory; the essence of a good theory is that it has very strong ties with the underlying practice, something which is definitely not the case with abstraction *per se*, which is a one-way process.

Programming language designers have recognised this as a philosophical problem. They started the quest for a universal programming language some thirty-five years ago. One of the products was the General Problem Solver [26], an existing algorithm into which only a formal description of the problem to be solved had to be fed, to solve the problem. This meant mystifying the real problem: as the programming of the computer had already been done, the only thing still to be given was the description of the problem (a software Universal Turing Machine). This quest was based on the fallacious assumption that there are 'natural' ways of coding programs, valid in all cases, which followed from the belief that there was a natural way of describing all aspects of the real world.

Peter Naur gives another reason why the idea of a universal language has faded away, thus corroborating Edsger Dijkstra's description of the programmer's task, with his Theory Building view of programming [24]. It is not the description of the problem in this universal language that matters, but the ultimate solution: "... The task of the programmer is not to write program text, but to form a theory, which is an *understanding* of how a certain data processing problem can be performed by a computer." He asserts then, that "... a running application is a far better tool for communicating this 'theory' than an unsupported program text." Terry Winograd rephrases this by abstaining from the abstract 'class' and 'theory', and emphasising 'working with programs' and 'understanding': "The main activity of programming is not the origination of new independent programs, but the integration, modification and explanation of existing ones. [...] The activity that we think of as 'writing a program' is only part of the overall activity that the system must support, and emphasis should be given to understanding, rather than creating programs" [34].

In the oral communication between programmers one cannot use formal language as a vehicle: we are dealing with human interaction instead of with human-computer interaction. We saw this as one of the principles behind Knuth's literate programming: the combination of program and documentation into one text in such a way that people can read it like any other text. In their interaction programmers use natural language, and as the problem to-be-solved needs to

be understood by the other programmer too, in terms of the problem domain, it is of no use to have recourse to a formal language, as that language does not contain the problem domain terms.

As formal language cannot be used in communication between programmers, it cannot be used at all in communication between user and programmer. In order to solve the user's problem a programmer must be ready to learn the language of the problem. Thus one strives for better understanding of the problem, and so, for a better description of it. But this description will eventually wind up in one of the programmer's own — more formal — languages as it needs to be executed. It is essentially a language conversion. We need a professional to make this conversion, as one has to keep one's distance from the problem. A certain abstraction from the problem domain, or, generalisation, is called for, as generalisation often leads to simpler solutions. Many problems resemble each other when seen from afar, and so generalisation may lead to already known solutions, accelerating the finding of the needed solution. So next to the ability to absorb the essentials of the user domain and to generalise, the mathematical inclination, one needs expert knowledge from all over computer science, to be able to solve the user problem. This characterises the programmer as a (computer) science professional.

The problem description has to be written down as easy as possible, as the programmer needs feedback on his solution from the user, in order to check on his comprehension of the problem matter, and on the way the user wants his problem solved. Writing the description leading to the desired solution, using a formal language, the programmer needs to stay as close to the problem as possible. The problem description needs his attention, especially when the transition from problem language domain to formal language domain is not a trivial one. He should not be distracted by housekeeping problems regarding the way the computer resources are handled, the how, when executing the program yielding the solution. Such tasks should be left to a lower level machine, as it is the problem, the what, he should be dealing with. In the next Chapter we will return to the formal language domain, and find a language which shows this characteristic, and besides, has a few other desirable ones.

We end this Section by turning it into a fugue with Edsger Dijkstra's Marktoberdorf advice: "Hence my urgent advice to all of you to reject the morals of the best seller society and to find, to start with, your reward in your own fun. This is quite feasible, for the challenge of simplification is so fascinating that, if we do our job properly, we shall have the greatest fun in the world. In short: Two [Three] cheers for Elegance" [9], thus setting the second Chapter's overtone: elegance.

The liberal arts theme from the opening Section, can now help in giving another dimension to 'the fun in computer science'. It is more than fun, fun being person bound, it is *enthusiasm*, springing from fun, and radiating, thus reaching others. A ballet performance on a Royal stage by full amateurs, young, ranging from four years up, cannot be termed error-free or perfect, but what these missionaries of Terpsichore did achieve, was "... stretching [their capabilities], ..., their ability to see [their capabilities] as more than when they were first led up to [them]." Transformed through their enthusiasm, this performance, being choreographed or programmed, became a perfect performance in the eyes of the audience, the 'users' watching the program evolve [19]. Being imperfect, missing in techniques, or the exact how-to of the program, is made up for by an overdose of intention, meaning, the what, that had to be conveyed to the audience. This how-versus-what contrast sets the second Chapter's main theme, where we

introduce functional languages to aid in conveying the meaning of the intended program, the prototype, to the user.

1.3 Translating and Society

The mechanistic idea of relating an Abstract Machine metaphor to the user - programmer situation must not be seen as a *panacea*, a cure-all. After all, we deal with people, and also, as there is interaction between people, we might take a look at sociology, or psychology, and, if conflicts are present, law, history or political science may offer solutions. The solutions we seek may not come from computing science at all: that science does not deal with human interaction. In this quest for a solution of the language mismatch in the other Culture we confine ourselves to the only other discipline engaged in writing complex sets of rules to be followed by an organism or organisation ('the Turing Test the other way round'): Law. The scholars in this field are engaged in explaining written language in conjunction with the reasons why something has been written, and with the form in which it is written down. In order to regulate society according to past and current morals and values, and to correct deviant behaviour in a consistent way, one has to write down the rules, and specify precisely what has been written. Limited time, space, and the subject prevent us from looking at sciences that do not deal with compiling large bodies of rules, or orders, like sociology, psychology, and history.

However, we find another language problem in the written rules of law [20]. Unlike programming languages, the language one's *Grondwet* (Constitution) is written in, looks like one's native language. Something we passed by in the previous paragraphs must be taken into account now: meaning. These rules, as every written or spoken text, must be explained or interpreted to acquire a meaning. Every reader must 'translate' the text he looks upon, even when he speaks the same language as the writer. From Shannon's classic information theory, we know that the receiver may not get the meaning of the message a sender intended it to have: a wrong translation. In this sense, every text is multi-interpretable. Part of the whole process of legislation (and in our case it is confined to codification: only law written by an authority) and judiciary actions deals with the precise explanation what is written. It is thus based upon reducing the number of different interpretations that can be given to the written law in order to arrive at the intended meaning [20].

Returning to computing: a computer together with a machine program, yields one, and only one, meaning, or desired behaviour, of the machine program. When we add compilers and interpreters to this picture, we know that the computer, together with a program in a higher language does not yield one, unique, meaning, if we are allowed different compilers. As computer science is engaged in delivering single meanings, what compiler gives the unique intended meaning[3] to the given program? Moreover, having found this compiler, then this unique meaning renders the other meanings incorrect. No matter how, it leaves the question whether this compiler generates the correct, unique, meaning to all other possible programs in its language? This is a problem

[3] Professor Willem van der Poel is often heard to say, that the program is always correct; and if it does not live up to your expectations (the intended meaning), your expectations are wrong and should be adjusted (e.g., at the Farewell Symposium in honour of Professor Alexander Ollongren, 17 December 1993, Leiden). But, as we will have demonstrated at the end of Chapter 3, textual changes to specifications are of the same order of magnitude as changes to programs, so at last, the question is: "Which is to be master?"

we cannot resolve. Nevertheless, all compilers generate meanings for a certain program that are fairly close to each other, spoken in general statistical terms, but not exactly alike.

The juridical situation shows even more similarity to the computing field. Everyone can have his own interpretation of the law, his (empirical) truth being 'the truth, the whole truth, and nothing but the truth' to himself, but with a second person there is another interpretation possible. The ensuing conflicting interpretations of the truth make that one has to go to court, the core of the machine of state, to resolve the dilemma. The judge gives the valid interpretation, he is the interpreter of the machine of state, with written law as his 'principles of operation'. He has to declare the real truce. However, in court one needs a person, a lawyer, well versed in the language of the juridical layer, to communicate with the judge. The judge, the official, authoritative, interpreter of the law, gives *ex officio* the official meaning of the text, that yielded the conflicting meanings or interpretations.

The judge also has some personal influence, yielding slightly different meanings of the legal text, given different judges. He is even compelled to have this own interpretation and to express it, for example when the law text does not apply at all to the context (e.g., classical copyright law with respect to software). In these circumstances it is his official duty to cover such open ground, with reasoned interpretation.[4] In this process we see the creative part of legislation: done by a judge, based on empirical facts and circumstances, yielding a new fabric covering open ground, lexture. His freedom of interpretation is restricted by the intentions of the legislator (which intentions are again open to interpretation). Convergence to one meaning is introduced by the hierarchy of judges, stopping at the *Hoge Raad* (Supreme Court), in our picture yielding the machine language meaning.

There is only one remarkable aspect: the juridical system is capable of modifying itself in a controlled fashion, something which has not been done yet in computer science in a theoretically sound way. The programming language in which it can be done is already present, LISP, but not yet the theoretical underpinning of program self-modification, "The fact that most functions are constants defined by the programmer, and not variables that are modified by the program, is not due to the weakness of the system. On the contrary, it indicates a richness of the system which we do not know how to exploit very well" [22].

Instead of finding a solution for the language mismatch problem in a centuries old field of scholarship, we found a similar situation. A formal language (though at first sight it is quite similar to our native language), users being addressed in that formal language when controlling them, an intermediary between the user and the machine of state in case of conflict, the lawyer, comparable to the programmer, and an elaborator, a semantic organ of the machine of state, the judge, comparable to part of the CPU. The juridical system knows even a phenomenon called 'backlog'. Contrary to the application backlog of the next Chapter, this backlog is caused by the 'hardware' system, as we just described. And so, again, too few people working with perhaps inadequate tools or processes, are a major reason for the backlog. It looks like we have to return to our own discipline to alleviate the consequences of the language mismatch.

[4] [20]: The interpreter is free in his interpretation, and can even deviate from the 'normal' meaning of the text. "Waar bijvoorbeeld in art. 124.2^0 in het Academisch Statuut staat dat de promotieplechtigheid een uur duurt, legt hij uit dat onder één uur drie kwartier wordt verstaan." ("Where, for example, in [...] the Academic Statute is written that the formal graduation ceremony lasts one hour, he expounds that one hour should be read as three quarters of an hour" — transl HD.)

Where Dijkstra remarked that, "... the use of anthropomorphic terms when dealing with computing systems is a symptom of professional immaturity", we, contrariwise, used mechanistic terms for human behaviour. Would this view of society express our personal ideas — not being used as a metaphor —, David Bolter would classify us as a 'Turing man' [4]. Bolter uses the computer as the 'defining technology' of the twentieth century, a 'defining technology' being a technology which redefines man's role in relation to nature. Other centuries have known other defining technologies, e.g., the clock and the steam engine. In our century the computer promises (or threatens) to replace man as information processor. The Turing man, accepting this view, looks upon man as an 'information-processor', and on nature (inclusive of mankind) as 'information to be processed'. He is inclined to categorise a large part of the human intellectual activity as 'programming', believing that most intellectual questions will eventually be solved by the computer.

However, although Bolter demonstrates that defining technologies shape the world of thought, some of his examples do not address this world, but the more or less pathological way this view is instantiated in human beings. Returning to the Abstract Machine metaphor, where we discussed the use of codes as belonging to the programmer's domain, Bolter exemplifies: "Human beings lose the possibility to react to each other in a human way, if each of them is not treated as an individual with a unique history and unique problems, but as a mere identification number with a chain of cold, hard facts attached" [4]. We claim that the transient instance of this view on man in a human being, is similar to the behaviour of the young (technical) student trying to integrate the new, promising, notions he just learned into his outlook on life, showing it off by using the terms in and out of season.[5] Being transient, it will not heavily influence the young student. Nevertheless, as programmers, we should be aware of the possibility of this creeping psychic deformation linked to the basis of our profession, and not only in ourselves, but also in the general public.

Bolter's thesis that the computer, viz a realisation of the Turing Machine Paradigm, is a real paradigm, holds. Ollongren and Van den Herik support the importance of this paradigm, they too postulate that Turing's Machine is the most influential new concept of this century in operationalising computability, spanning many disciplines [27]. Also the extension of this paradigm to a defining technology is a valid extension, but only for the world of thought. If this paradigm gets instantiated in a human being, thus shaping his thought processes and his way of viewing the world, it is not any more a defining technology, with its positive connotations, but it becomes a psychological symptom, to say the least. Programming should be left to professionals, and should be done for computers.

Before ending this Section after having wandered off into the fields of law, and having looked at the threat of humans masquerading as automata, we present some sources in the outside world of Thought for a few program design principles we will encounter in the next Chapter, taken from the table of the Dining Philosophers and their guests.[6]

[5] E.g., the course 'Bicycle Parts' of the late dr W. Holthuis, during the Academic Year 1964–1965, University of Twente. A related example is the Clinical Student's Syndrome of the young medical student. Contrary to the technical student, who is confronted with abstract notions which he can use straight away, the medical student develops the syndrome only after confrontation with the reality of the sick patient; the notions he learns are not abstract, but real: lectures on Pathology in the student's pre-clinical years do not invoke this syndrome, they are too theoretical. (This syndrome, turned pathological, becomes Münchhausen's Syndrome.)

[6] Gerrit van der Hoeven kindly showed the way to Giuseppe Scollo's dissertation (see References Chapter 2),

For an analogy of the programming principle, 'separation of concerns', we must go back to Montesquieu (1689–1755) with his *séparations des pouvoirs* within a state. This *trias politica* recognises three independent functions within a state: the executive, the legislative and the judicial. In order to prevent abuse of power, he separates the legislator from both the executive power and the judiciary power, who are also separated [7]. This separation of functions which should not have anything to do with each other, can be phrased as 'orthogonal' functions of the offices of State.

The early Renaissance gives a link to the intuitive aspects of 'harmony'. These were already known and practised since Ancient Greek architecture and sculpture, but the Italian mathematician fra Luca Pacioli was the first to publish these aspects in his *De Divina Proportione* (1509) [30]: "Dividing a segment into two parts in mean and extreme proportion, so that the smaller part is to the larger part as the larger is to the entire segment, yields the so-called 'Golden Section' " ($u : v = v : (u+v)$, when $u < v$, and by setting the ratio $\phi = v/u$ one derives the relation $\phi = 1/(\phi\ +1)$. So $\phi\ \approx\ 0.618$). Euclid had this notion present in his construction of the pentagon and the dodecagon. Why this ratio is aesthetically pleasing to us is unknown. Surprisingly, it is expressible in mathematical terms, terms that also relate to such diverse subjects as the breeding of rabbits (the Fibonacci sequence) and the arrangement of leaves on a stem and their relative positions. Another mathematical formulation of 'harmony' can be traced back to the Pythagoreans (± 530 B.C.), who discovered that musical consonants are produced by strings with lengths that are proportional to simple integer numbers, e.g., $1:2, 2:3$, or $3:4$.

From the Middle Ages come two contributions for the foundation of our design guidance in Chapter 2. John Duns Scots (± 1266–1308; Latinised as Scotus), a realist, and the forerunner of philosophy independent from theology, was the originator of *"Ex falso sequitur quodlibet"*,[7] an appropriate statement for the logic fabric of a software system, as an inconsistency in a formal system yields it useless for practical work. The realists claim that there are universal concepts present in reality, e.g., 'daughter'. These are based on some 'common nature' of the individuals or of the objects. A newer train of thought was started by the nominalists who claimed that reality only consisted of real objects, these objects perceived by the senses, create in the mind 'notions', which are denominated by words (*nomina*) to be able to talk about them.

Scotus' brother Franciscan, William of Ockham (± 1285–1349; Latinised as Occam), was the founder of the nominalists. He presents us with the basic philosophy behind simplicity in argumentation, or, the law of economy or parsimony, of thought. This argument was used before, but Occam used it so frequently and so wittingly, that it became known as 'Occam's Razor': *"Entia non sunt multiplicanda praeter necessitatem"*.[8] He used the argument to eliminate many entities that had been devised, especially by Scholastic philosophers, to explain reality, that as the Scholastics had it, started as idea in the mind of the Creator. As reality only consisted of objects, the nominalists say that the philosophy of science should keep the number of notions added to reality as small as possible, lest needless or faulty additions to this reality lead to false conclusions. In everyday practice we use the 'razor' mostly in the sense, that if there are two explanations of a certain phenomenon, one more elaborate than the other, then the simpler explanation is nearer to the true explanation than the elaborate one [1, 6].

---

where the quotes of Occam, Duns Scotus, and Herakleitos are mentioned.

[7] "Everything follows from falsity."

[8] "Entities shall not increase beyond necessity."

The Greek philosopher Herakleitos (± 535–± 475 B.C.) provided us with the necessity for thinking ahead, planning for change, not thinking that things tomorrow will not exhibit any change from their current state: "$\pi\alpha\nu\tau\alpha\ \check{\rho}\epsilon\iota$".[9] He is the philosopher of the intrinsic change, the harmony of opposite characteristics within the same object.

The principle behind the modularisation of the systems we build, has an still older analogy. The *divide et impera* has been practised in commanding and controlling large complex bodies or situations from the Ancients until today by military powers. An army is commanded[10] through the hierarchy of units and subunits and so forth until the chain of command reaches the lone soldier. And a conflict in which armies are involved, is nowadays controlled through tactics and strategy, using (specialised) parts of the army. The ideal situation of command *and* control did, however, not exist in the days of the Iliad. Commands were issued to the, e.g., Byzantine, generals and one trusted these commanders to follow their orders. Real control could only be gained by introducing feedback into the chain of command. And in fast changing, complex situations, this is only possible with the advent of communication networks: messengers, semaphores, telegraph, and radio, i.e., electro-magnetic means in general. (And effective feedback can only be gained by not sacrificing the bad-news messenger.)

The other (political) meaning of *divide et impera* was sowing discord, and playing off parts of the population against each other, in order to gain and keep control over the system by an (administrative) power (emperor, usurpator, dictator). This side of the principle hits us, as the power that divides cannot reckon with unforeseen crosslinks between parts of the system (also known as side effects).

1.4 Embedding and Programme

So, for the time being, we have to alleviate the language mismatch within our own discipline as there are no apparent solutions outside it, to go from desired behaviour to the precise rules enforcing or causing that behaviour. The next Chapter elaborates upon the language mismatch due to different abstraction levels of user and programmer. We propose an attention shift in software engineering from the later parts of the system development cycle to the early parts. Therefore we extend these early parts to cover the whole process of problem solving. At the same time, we make clear what the focus of attention should be, by renaming the 'system development cycle' to 'problem solving cycle'.

We propose to alleviate some of the problems caused by the language mismatch by concentrating on prototyping, making a tangible, usable *model* of the eventual solution. In order to create prototypes we use a functional programming language, a kind of language which focuses attention on describing what to do, instead of on how to do it. Prototyping must be used as the language

[9] "Everything flows."

[10] Only in the second half of the nineteenth century the study of the structure of commands was undertaken; structure not based on power or necessity, but on logic. Nicholas Rescher wrote a monograph on *The Logic of Commands* (Routledge & Kegan Paul, London, 1966), commands including orders, directives, injunctions, instructions and prohibitions (negative commands). He builds a pure logical framework for the command, taking his examples from legal contexts and computer programs. Having established the concept of validity in command inference, he studies the logical relationships between commands. (In one of his programming examples (all done with flow-charts) he uses G.H. von Wright's term 'Sisyphus-orders' for commands establishing an infinite loop.)

used in specifications hinders human communication: prototypes can be judged immediately on their effects by the user, in a way language can not (or never) be judged. So prototyping implies user involvement. The programmer will address the user in his own language. At the same time he shows the user what has been accomplished thus far in the problem solving process. Prototyping as an essential part of the extended 'problem solving cycle' is discussed, and qualitative characteristics (guiding principles) to help the programmer design the solution for the user problem — forming the substance of his program — are given.

However, functional languages were not very open to outside communication, as they started out being a thorough, but academic, exercise in an algebraic setting. Isolated since the researchers at first needed to get a feeling for, and then an understanding of, the principles involved in the Functional Programming Paradigm. This process takes some time, so programs in these languages did not really need to communicate with other, already existing solutions, in their early days. This communication lag is but one of the aspects of functional languages which needs attention in order to spread their use in programming.

All through the second Chapter we derive or state requirements for a prototyping language. In the third Chapter we present a fulfillment of these requirements: functional languages. We further give background information on the Functional Programming Paradigm, functional languages and their basics (lambda calculus and functions, combinatory logic), and functional programming itself.

We present this material in order to prepare the reader for Chapters 4 and 5, where the communication side of facilitating the use of functional programming in prototyping is elaborated and demonstrated. This side is one of the stated requirements where functional languages fall short of complete fulfillment. In these Chapters we focus our attention to this one aspect of extending functional languages, and we will demonstrate it, using one particular language, Twentel. In this language we introduce a way of communicating with the outside world, where databases, expedient systems, and existing solutions are present. This communication is possible via a construction in the functional language, called *trapdoor*. Chapter 4 deals with the programmer's view of the trapdoor, how to use it, what to expect, and what possibilities it has. The result is an easy way of linking to the outside world, and consistent with the spirit of the functional language. The next Chapter describes in more detail, and formalises, how the trapdoor operates and shows examples of its application on different software platforms.

We end this dissertation with a discussion whether the effects of the language mismatch are decreased by the use of this trapdoor, and give an overview of the questions we left open, and the new ones we raised.

This ends the programme of this dissertation. The embedding of the research has already been given by the reason for extending functional languages: at the surface of the functional language connecting with software engineering in order to get a better, earlier, and / or cheaper solution for the user problem. Note however, we stay within Computer Science, and remarks on its subdiscipline Applied Informatics will be made with the scientific method in mind. This method is in the case of the technical sciences beautifully paraphrased by the motto of the KIvI, "Scheppend

denken, en schouwend doen."[11]

References

[1] (A.A.Ma.), "Medieval Philosophy", in: *The New Encyclopædia Britannica.* Vol 25 — Macropædia, Encyclopædia Britannica, Chicago IL, 15^{th} ed, 1992, pp 754–757.

[2] Abelson, H., Sussman, G.J., with Sussman, J. *Structure and Interpretation of Computer Programs.* (with foreword by Alan J. Perlis), MIT Electrical Eng and Comp Sci Ser, MIT Pr, Cambridge MA, *and* McGraw-Hill, New York, 1985.

[3] Ashenhurst, R.L., Graham, S. (eds) *ACM Turing Award Lectures — The First Twenty Years, 1966 – 1985.* ACM Pr Anthology Ser, ACM Pr, New York / Addison–Wesley, Reading MA, 1987.

[4] Bolter, J.D., *Turing's Man: Western Culture in the Computer Age.* Univ North Carolina Pr, Chapel Hill NC, 1984.

[5] Buxton, J.N., Randell, B. (eds), *Software Engineering Techniques.* (Rep Conf sponsored by NATO Science Committee, Rome, Oct 27–31, 1969), NATO Scientific Affairs Div, Brussel, Apr 1970, 164 pp.

[6] Delfgaauw, B., Van Peperstraten, F., *Beknopte Geschiedenis der Wijsbegeerte — van Thales tot Lyotard.* 15e ed, Kok Agora / Pelckmans, Kampen / Kapellen, 1993, 280 pp, (in Dutch).

[7] De Secondat, Charles-Louis, baron de la Brède et de Montesquieu, *De l'esprit des loix, ou du rapport que les loix doivent avoir avec la constitution de chaque gouvernement, les moeurs, le climat, la réligion, le commerce, etc..* 1748, (The *trias politica* appears in Book XI, Chapter 6).

[8] Dijkstra, E.W., "The Humble Programmer", *Comm ACM* **15**(10) (Oct 1972): 859–866, (1972 ACM Turing Award Lecture, 14 Aug 1972), also in [3], pp 17–31.

[9] Dijkstra, E.W., "On the Nature of Computing Science", in: *Control Flow and Data Flow: Concepts of Distributed Programming*, M. Broy (ed), (Int'l Summer School, Marktoberdorf, Jul / Aug 1984), NATO ASI Ser, Vol F14, Springer-Verlag, Berlin, 1985, pp 1–3.

[10] Van Dorp, H.D., *Inputreductie — Foutcorrectie en Afkortingen.* MSc Thesis, Vertalergroep, Rep VGP.ISA-R 801, Dept Electrical Eng, Technische Hogeschool Twente, Enschede, *and* Philips I.S.A. Research, CS Group, Eindhoven, 10 Dec 1971, 59 + x pp, (in Dutch).

[11] Fritz, W.B., "ENIAC — A Problem Solver", *IEEE Ann History Computing* **16**(1) (Spring 1994): 25–45.

[12] Gries, D., "A Comment on Parnas' Counterpoint", in [32], pp 359–364.

[13] Head, G.E., "Six-Sigma Software Using Cleanroom Techniques", *Hewlett-Packard J* **45**(3) (Jun 1994): 40–50.

[14] International Organization for Standardization (ISO/TC 97/SC 21). *Information Processing Systems — Open Systems Interconnection — Basic Reference Model.* ISO 7498, Geneva, Oct 1984.

[15] Jones, A.K. (ed), *Perspectives on Computer Science.* (Rec 10th Anniversary Symp Comp Sci Dept, Carnegie-Mellon Univ, 6–8 Oct 1975, Pittsburgh PA), ACM Monograph Ser, Academic Pr, New York, 1977.

[16] Knuth, D.E., *The Art of Computer Programming.* Vols 1–3, Addison–Wesley, Reading MA, 1968–1973, (2^{nd} ed 1981–).

[17] Knuth, D.E., "The Art of Computer Programming", *Comm ACM* **17**(12) (Dec 1974): 667–673, (1974 ACM Turing Award Lecture, 11 Nov 1974), also in [3], pp 33–46.

[18] Knuth, D.E., *Literate Programming.* Center for Study of Language and Information, CSLI Lecture Not No 27, Leland Stanford Jr Univ, Stanford CA, 1992, xv + 368 pp.

[11] Koninklijk Instituut van Ingenieurs, 'Royal Dutch Institute of Engineers'. As the KIvI does not have an official translation of their motto: "Creative thinking, and contemplative acting" — transl HD. Gerard te Meerman touched upon another relevant meaning of 'schouwend': 'visionary'.

[19] Lansdown, J., "Using the Computer to Augment Creativity: Computer Choreography", Report, Centre for Advanced Studies in Computer Aided Art and Design, Middlesex Univ, Barnet, 1994, 15 pp.

[20] Lokin, J.H.A., Zwalve, W.J., *Hoofdstukken uit de Europese Codificatiegeschiedenis.* Wolters-Noordhoff / Forsten, Groningen, 1986, xiv + 402 pp, (in Dutch).

[21] Minsky, M., "Form and Content in Computer Science", *J ACM* **17**(2) (Apr 1970): 197–215, (1969 ACM Turing Award Lecture, Aug 1969), also in [3], pp 219–242.

[22] McCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P., Levin, M.I., Lisp *1.5 Programmer's Manual.* MIT Pr, Cambridge MA, 1962, (2^{nd} ed, 1965, vi + 106 pp).

[23] Naur, P., "Languages for Multi-levelled Computers used as Models, Tools, and Toys", Unpubl Manuscript for Perlis Symposium, 29 Apr 1993 (the presentation itself is discussed in [29]).

[24] Naur, P., "Programming as Theory Building", *Microprocessing and Microprogr* **15** (1985): 253–261, (Invited Keynote Address, 'Euromicro 84', Copenhagen, 28 Aug 1984).

[25] Naur, P., Randell, B. (eds), *Software Engineering.* (Rep Conf sponsored by NATO Science Committee, Garmisch, Oct 7–11, 1968), NATO Scientific Affairs Div, Brussel, Jan 1969, 231 pp.

[26] Newell, A., Simon, H.A., "Computer Science an Empirical Enquiry: Symbols and Search", *Comm ACM* **18**(10) (Oct 1975): 197–215, (1975 ACM Turing Award Lecture, Oct 1975), also in [3], pp 287–317.

[27] Ollongren, A., Van den Herik, H.J., *Filosofie van de Informatica.* Ser Wetenschapsfilosofie, Martinus Nijhoff, Leiden, 1989, (in Dutch).

[28] Perlis, A.J., "The Keynote Speech", in [15], pp 1–6; (A slightly abridged version appears as dedication in [2]. This dedication starts with: "This book is dedicated, in respect and admiration, to the spirit that lives in the computer").

[29] Pfeiffer, P., "Report on the Second Annual Alan J. Perlis Symposium on Programming Languages", *SIGPLAN Not* **28**(9) (Sep 1993): 6–12.

[30] (W.L.S.), "Golden Section", in: *The New Encyclopædia Britannica.* Vol 22 — Macropædia, Encyclopædia Britannica, Chicago IL, 15^{th} ed, 1992, pp 5–6.

[31] Wegner, P., "Introduction to Part II.I – Software Methodology", in [32], pp 203–206.

[32] Wegner, P. (ed), *Research Directions in Software Technology.* MIT Pr, Cambridge MA, 1979.

[33] Van Wijngaarden, A. *et al* (eds), *Revised Report on the Algorithmic Language ALGOL 68.* Springer-Verlag, Berlin, 1976.

[34] Winograd, T., "Beyond Programming Languages", *Comm ACM* **22**(7) (Jul 1979): 391–401.

Chapter 2

Problem in Context: Prototyping

In the Introduction we tacitly introduced the computer for solving user problems, and in the same move, programming and programmers. It is obvious that we will restrict ourselves in this dissertation to solutions in which computers are needed. We mentioned the language mismatch, a language problem which existed in the relation between users and programmers: the problem of addressing the user in the wrong language, not the language of his own layer.

Here we follow this language mismatch thread a little further. We saw that there is no obvious solution for this mismatch outside our discipline, so we must look for tools in the other computer science sub-disciplines to speed up the primary production process of the application programmer, given the interdependent constraints of money, time and quality. We concentrate on the demands on the higher layers arising from the application programmer layer, because of the problems the application programmer is faced with when obtaining solutions for the user. Throughout this Chapter we will state these demands or requirements regarding the principal tool of the programmer: his programming language. Due to the language mismatch we cannot resolve, our main concern is to find something within our own discipline to help this programmer to get at the desired solution as fast as possible: faster delivery of higher quality with lower cost.

In the first Section we discuss the problems the user seeks solutions for when handling information flows or large computations. The first stages of the ensuing user - programmer contact are described. Following up on this contact, the second Section states the purpose of the dissertation: extending functional languages in order to facilitate their use in prototyping, and in doing so, spreading their overall use.

The subsequent Sections offer a view at the programmer's side of the medal, describing his way of getting at the solutions the user wants; this entails software engineering, the engineering of the process of software production. We will highlight a few qualities of software engineering, needed for positioning our (partial) solution of the language mismatch problem. Partial, as a programmer is still needed in applying the tool that is the result of the research this dissertation is based on. One designs the software that solves the user problem before producing the software. We give some principles for good design, leaning heavily on general principles, one of them, elegance, another, consistency. The reason for writing them down is that we needed justification regarding some design decisions raised in Chapters 4 and 5. However, they are of greater significance. Finally, we will discuss prototyping, a technique used to elicit from the user more definite statements regarding his problem, or regarding the programmer's intended solution. Using this technique must then lead to the desired solution. This process is made easier through the use of an extended functional language.

2.1 The Solution for the User

It is odd to start a Chapter in which the problem to be solved will be described, with 'solution'. However, staying in character with the notion of 'lazy evaluation' ('only do it, if it is actually requested', 3.1.3.2), it all starts with 'the desired solution'. Why does the user have to import computers in his environment? We hold that the user, whether he is a human being or an organisation, needs solutions involving computers to solve some of the problems he has. Problems that are connected with information flows, or with long, or complex, computations. Problems in the primary processes that his business depends on, or hamper the goals he sets himself. Well engineered computer based solutions can help the user commanding and controlling these primary processes faster, in a more reliable way, better tuned to the changing circumstances, with higher quality. In other words these solutions help him run his business better, facilitate the fulfilling of his goals. So it is the user who triggers the creation of solutions, and, of course, the programmer is in a similar way a user, so he too, can demand new solutions for the problems he has in his everyday handling of information or computations.

Computer or computing science is the general, catch-all phrase for our discipline, but in dealing with user problems it is mostly the sub-discipline " Applied Informatics", that plays the central role. The subject " Applied Informatics " is not theory for the sake of theory, but is practice-oriented, based on theory. It is the professional production of applications (software, hardware or any mixture) for the general user [67]. Here we encounter the professional as the person who delivers these products as a service to the user, and it is the essence of a professional to use his knowledge and skills in the service to others — his clients — with principal loyalty to his profession after having taken the utmost care to do his job right. And we have seen that this professional is the programmer of the Introduction, in this case, the application programmer. Applied Informatics (with computer science in the background as its support, like mathematics) is a society supporting discipline. Working in this field of Informatics, one must take the *servitude* (a notion based on service, not on servility) of Applied Informatics (and computer science) to society (the assembled users) as lead principle: recall that the programmer should have a humble attitude toward the design and delivery of solutions. Society uses computers with its applications, or, more to the point, society uses applications — that need computers to be able to function.

Everything starts with the user describing his problem to a programmer. We claim that the problem must be described at the highest possible level, as close to the problem as one can get. Apart from the fact that it yields a shorter description, and it is so more easily comprehensible, there is also the matter of time spent. The more details one takes into account, the more time it takes to write them down, while at the same time details do not add clarification to the overall problem structure. Time spent on studying the problem, essential for getting at the problem structure, is time spent wisely. When dealing with complex problems one must investigate the problem very thoroughly until the underlying structure is fully understood, such that it can be written down in a clear and concise way. Shlaer and Mellor remark on this stage of the problem–solution route: " One of the major problems in a systems development project is a form of floundering [...] modern software engineering tools [...] have little impact on [...], since the origin of the problem is a lack of fundamental understanding of the conceptual entities that make up the application problem" [69].

Not only must the programmer sketch the problem, also its environment, its context, must be taken along. Seldom does one have a problem without context, except perhaps, a pure mathematical one. The desired solution must fit in the context of the problem: the interfaces of the outside world must seamlessly merge with the solution, otherwise these interfaces will create new problems to be solved. However, for reasons of simplicity we will henceforth refer to the 'problem to be solved', not to the 'problem within its specific context to be solved'. Following Kuhn [39] we stick to 'problem' as the perfect solution may not exist, otherwise it would be a 'puzzle'. The same holds in mathematics: an equation can have an analytical solution — the perfect fit —, so finding the analytical solution is solving a puzzle, but sometimes only an approximation can be given, e.g., as a result of the application of numerical mathematics, thus finding a solution to a problem.

Of course, there is always the possibility of being confronted with a problem of such a complex or intractable structure that one has to develop a new way of describing such a structure; having found it, the problem can be dealt with — described — in an easier way. Having found an easier description, a better model, e.g., mountain ranges, shore lines, ferns, and moon craters described by fractal geometry, this does not necessarily imply that the related problem can be solved more easily. Perhaps describing that particular aspect of the problem in an easier way does not lead to a solution at all, or the new description itself is intractable.

Now we have a description of the problem; it is a point from which to start generating the solution, from which the solution can be engineered subsequently. The requirements, the problem description with all aspects of the desired solution in its context, are stated at this point. However, they tend to be ambiguous, they can be inconsistent, interacting in a way which even prohibits meeting some of the requirements themselves; they can be non-specific or qualitative; they are likely to be vague and incomplete. These user requirements have already been mentioned in the Introduction: they form part of the language mismatch between user and programmer, with the user talking his own language (and jargon) which is not formal at all, and the programmer talking his (more) formal language and writing the specifications, which, being precise, cannot be considered requirements. However, if there are real requirements — real, so they include the context and are written in the user's language — these requirements can, in theory, generate many solutions.

In order to get at the *one* solution, there must be one product specification. Now we try to close the gap between user (requirements) and programmer (specifications). However, the process from requirements to specifications is not formalised. And in order to become more precise than in the requirements, one needs to elicit the necessary refinements from the user. As the language in which the specifications are written, is the language of the programmer, the user cannot grasp them, let alone in their entirety: it is the wrong language. If there is something tangible present which shows everything in action what is promised by the programmer, the user can decide on what he needs, exactly and not vague, by watching the actions of a tangible object: the *prototype* of the solution. This is not yet the solution: it is coined from the 'first' (Gr. *proto-*) 'embodiment' (Gr. *-typos*) of the solution. In this way the user gets a feeling of the product the programmer is about to construct. Through many change-description and execute-prototype sessions the trial-and-error process of design is supported. This leaves the programmer needing a programming language in which he can write his prototype in a fast and efficient way, written down quickly and yielding an executable prototype which shows the desired functionality on which the user can comment.

Before entering the field of the dissertation proper, we must point at the growing number of standard solutions and user-made solutions based on these standard packages. We understand 'standard solutions' to be off-the-shelf packages, like spreadsheets, database management systems with easily modifiable user interfaces and query languages, and word and text processors (and even symbolic mathematical packages). At the same time this class of solutions contains specialistic packages, ranging from general ledger programs to control modules of digital telephone exchanges, and packages with special purpose languages, e.g., CAD / CAM and SQL.

We have arrived at a situation in which many solutions which formerly had to be programmed by the (central) computer department of the organisation, could now be taken on by the users themselves, whose computer literacy and awareness also grew in these years. This growing awareness is by itself also one of the developments that helped to increase the number of user-made solutions. These solutions do not quite seamlessly fit into their context, but passably, taking into account the (apparently negligible) amount of budget spent on it. However, compare this discussion with self-medication. A glass of wine, or an aspirin, does solve some problems effectively. But sometimes problems are symptoms of larger problems, and then it takes a physician to diagnose the real situation and to solve the larger problem, instead of fixing the smaller symptom.

So the user can devise his own solutions, using or building upon these off-the-shelf packages. But there is a hidden cost of ownership involved. At any time during the prolonged use of the package, the user must be able to determine the cost of fixing the shortcomings of the package due to the changing circumstances. It may well turn out that adapting the package to the changes is too costly. At that moment it is obvious that a more generally designed, more easily adaptable solution to this problem is preferable to a monolithic, off-the-shelf package in the first place. And for creating this kind of solution the user needs a programmer.

The advent of these standard packages was brought about by the following developments. In the past ten to fifteen years there has been a growing decentralisation of computing power, caused by its tremendous increase and falling price: a cost / benefit ratio becoming very small, provided the benefit remains the same. Through this process, standard solutions could emerge. Solutions needing so much parametrisation and extra functionality, that it would slow down an application beyond functioning, but for the ever faster hardware; this hardware technology push was precisely the reason why these standard solutions were constructed. Many of the user problems with a rather simple and fixed structure now came within the solving capabilities of the standard packages. These standard solutions match the immediate needs of the user, and match his expectations on how his problem must be solved.

The first steps taken after introduction of a standard solution, are correcting the erroneous and erratically behaving functions, and adding missing functions from the problem domain. After a complete covering of the problem domain within the standard package — a full grown package —, the next step is adding new, obvious, functions at the rim of the problem domain — an expanding package. If the package is specialistic at a too low level, the danger of mismatch with other standard packages exists. Especially so when the user wants his packages integrated, a natural requirement as in his view his business processes are integrated.

The solutions — enabled by the hardware technology push — of adding more standard packages to cover the integration, or of adding more machine power to keep the package going under

increasing load (in short, 'outrunning' the problems) is not a real way out. The 'push' cannot continue for another forty years in the same way as one gets near the physical limits for increasing computing speed in current computer architectures. To keep up with the everlasting demand for more power, one needs, either a radically novel architecture (or realisation, 2.4.3), or more efficient solutions. An order of magnitude in 'inefficiency' of solutions can easily be absorbed by newer hardware technology, as mentioned. More orders of magnitude require another realisation of the solution, and programmers are needed to deliver that other realisation.

The solution for the integration problem is different. At first it entails modelling the problem at a high, abstract level. Next splitting up the problem in such a way that some parts of the problem can be solved by standard packages, with the inter-package communication, the integration aspect, taken care of by the high level model. And for creating this high level model the user needs a programmer, as this creation cannot be done through the application of a standard package.

Through scaling up of applications, the expanding knowledge of reality, and the increasing number of relations between various parts of reality, the existing solutions will start to fall short of their erstwhile full satisfaction. This means that new solutions must be made, next to, on top of, or instead of the existing ones. Also the growing acceleration of changes in reality triggers the search for new solutions. Most of the day-to-day problems that have been solved have a fixed structure, so, from this point of view, it is explicable that standard solutions could emerge. The traditional imperative languages (FORTRAN, ALGOL, COBOL) with their unchangeable, fixed format, data structures are well suited to these fixed structure problems, once this structure is clear. A language like LISP with its inherent free format data structure, the list, can also handle these problems, even more efficiently than FORTRAN [15], but LISP's main strength lies in handling (non) day-to-day problems with a variable structure ('Artificial Intelligence' related). Today — due to the fast changing knowledge of reality — the problem is not so much the fixed structure, built as such into a solution, but the inherent unknown or intractable structure which has to be taken along in the solution. So, through sheer complexity, intractability, magnitude, minuteness, or number of dimensions, we hold that there will remain problems that are not to be solved by the methods of the standard route. To solve these non-standard problems, we still need programmers.

Another reason to keep the programmer engaged, is the creation of new 'classes of computations', devising new standard solutions from old ones or from non-standard ones through the processes of abstraction and generalisation on the existing solutions.

2.2 The Problem of the Language Designer

We will leave the choice of prototyping for minimising the effect of the language mismatch at this point as an established fact: somewhere in the problem-solution chain prototyping has its own place [11]. The application programmer needs a powerful prototype language in order to deliver his solution through the use of prototyping. And this is where the language designer comes in. We hold that functional programming languages are among the powerful prototyping languages [3,38,75], but they miss something, which brings us to the subject of this dissertation.

If functional programming languages were the philosophers' stone, this dissertation could not have been written. In the next Chapter we will discuss these languages in a wider perspective and in much more detail. At this point we state that functional languages are very promising, but they lack a general applicability, an expansion into the outside world: they are closed, their environment consists of themselves including their data. Of course, being a powerful prototyping language, it is fairly easy to build a prototype of a relational database management system in such a language, for instance in the functional language Twentel [63,70], but linking the self-built system to an existing database system is not trivial. Also, one can easily build a new, fairly complex system in a functional language [38], but using the accomplishments of others within one's self-built system is not easily done. And one needs these accomplishments because they execute much faster than the self-built system.

We are not aware of any functional language in which it is easy to link to other programs or systems in order to obtain — in the true spirit of prototyping — an acceleration in the process of effect generation (using other solutions or, on the efficiency side, using faster solutions). There might be languages, or ways of describing the problem, more closely related to a part of the problem than the main programming language, used for solving the problem. Why squirming under unnatural constructions, contrary to the main language philosophy? Such a situation calls for making a sidestep to the other description regime. An additional advantage of such a sidestep to a 'better' description is the gain in efficiency of the solution of the subproblem. Not only in describing it, but also in executing it: experts have spent so much effort in optimising the elaboration of the particular programming language (e.g., its compiler), that the effect of this effort cannot be expected to be recovered in another environment. A very nice example of such a description change lies in the realm of the integral transformations. If obtaining the solution is not going to be easy in the time domain, a transformation of the problem into the frequency domain might make it almost instantly solvable, after which only the inverse transformation has to be applied to get the desired solution.

Another reason for stepping out of the main programming language lies in the possible existence of proven solutions to subproblems of the original problem. It is also possible that the solution has to refer to data or programs already present in the problem environment, which is one more reason for switching language, or stepping out.

Having chosen functional languages as our main programming language, we shall demonstrate the feasibility of such a sidestep solution by extending a functional language. And in extending it we must stay true to the language philosophy. Both here and in other cases, availability of the resource (hardware, programming language, tools) and its adaptability, was the decisive factor for making a choice. Availability shaped the environment of the demonstration.

What kind of functional language is available for expansion, in what language is it implemented, on what kind of hardware can it be demonstrated. We choose the functional language Twentel [34] to be the demonstration language, because the sources of its implementation, written in a PC-based Pascal, were available. Relating to this availability, and also for economical reasons, we choose an 'IBM-compatible' PC as the hardware platform for this demonstration.

We are now ready to state the purpose of this dissertation, and, subsequently, to define the goal of the demonstration that takes up Chapters 4 and 5. In order to further disseminate the use of functional languages, apart from raising awareness and more education on the subject, one

can enlarge their possibilities, such that the programmer tends toward using them more often, as such a language becomes more desirable. This can be considered to be our purpose. To serve this we set up a demonstration. One should not enlarge possibilities by introducing more constructs in the language, *à la* PL/I, but by facilitating their use in different environments. In this dissertation we demonstrate that the above sketched gap in the applicability of these languages can be filled, enlarging their possibilities. However, as will be clear after reading Chapter 3, the extension of a functional language elaborated there, is only one of the aspects to deal with when facilitating the use of functional languages.

To demonstrate the feasibility of this extension, we use the following scenario. The user and the programmer, speaking the user's language and minimising the language gap, are together in the process of formulating requirements for the user problem. At the same time, the programmer is engaged in creating specifications through repeated (re)formulation of the user requirements with the use of evolving prototypes. In the creation of the prototype and its subsequent modifications he must stay as close to the model of the problem as possible, so the prototype notation must be similar to the model description, as high-level as possible. He uses the prototyping tool for eliciting from the user more definite statements regarding the problem, or regarding his intended solution, which narrowing should eventually lead to the desired unique specification of the solution. In order to keep the user interested, and so involved, he needs a powerful vehicle for fast delivery of the various prototype versions, which must be executable without further time-consuming transformations. For a prototype to be realistic it must execute adequately, and one must be able to execute it in the context which includes the problem to be solved. The first requirement might necessitate faster hardware, or one must be able to delegate execution-intensive parts of the prototype to other processors. The last requirement implies connecting the prototype to already existing programs or systems, which means that we are concerned with a kind of embedding those systems in the prototype. A programmer using a functional language is able to deliver quickly the necessary prototype evolutions because of the characteristics of his programming language.

However, for historical reasons a functional language is not very amenable to communicating with other systems. In extending a functional language with the possibility of linking to other programs or systems, we create the required powerful vehicle. The resulting extended language, the prototype tool, is the new building block for dispatching the production of applications in a more efficient way than before, which is our main concern in delivering solutions to the user. We will show then that connecting to other programs is possible in an easy way, thus making the resulting extended language more useful for prototyping.

2.3 Providing Solutions: Software Engineering

In the Introduction we saw the Software Engineering layer come into existence around 1968, somewhat wedged or folded between the application programmer's layer and the other layers. The definitions of Software Engineering are manifold. They all relate to providing programs to the user, using all sorts of methods and techniques from computer science. Of course, this is not the place to outline a course on software engineering, so we will only touch upon some points in order to embed prototyping, and the use of functional languages in it. The same holds for the

discussion of the Software Crisis: not all points are touched upon, only those that will profit from a language based solution.

2.3.1 Software Engineering as Facilitator

A classic definition of *software engineering* is given by Barry Boehm: "the practical application of scientific knowledge in the design and construction of computer programs and the associated documentation required to develop, operate, and maintain them" [12]. Hans van Vliet adds the difference between programming-in-the-large, the construction of large programs, and programming-in-the-small, which is taught, most notably, and in its purest form, in the aesthetic school of Dijkstra c.s.. He poses that the main focus of attention for a software engineer is the overcoming of complexity in design [74]. David Parnas adds complexity of control: "... multi-person construction of multi-version software ..." [57]. The 'multi-version software' necessary for, as Hans van Vliet also mentioned, the inevitable evolution of programs as they model a part of reality which changes with time (1.3). And according to SION,[1] "Software engineering aims at improving the quality and efficiency of the program development process. Its ultimate goal is the construction of error-less programs for an acceptable price. Based upon a set of informally worded requirements, the question is how to achieve a functioning and maintainable system that satisfies those requirements, and, if possible, also the less stringent requirements, the wishes". We summarise software engineering here as "*facilitating* the use of the computer for ...".[2]

Note however, that none of these definitions take into account the 'classes of computation', or 'theories', from the Introduction. Software engineering concerns itself with making products, not with designing abstractions, like theories or classes. This again is a source of many problems software engineering tries to find a solution for. Perhaps due to the product and process oriented nature of software engineering, it is itself sometimes part of the problem.[3]

As is clear from the definitions, there is much involved in the software engineering process, but we will concentrate on the first stages of this process: requirements, specifications, design and the related prototyping. Nothing will be said about the consequences of the complexity of control (e.g., management of people, resources, versions, changes and processes), documentation,[4] or maintenance. Knowing approximately what to make — the requirements — the programmer must now start giving form to the solution by writing it up in its specification: he designs the solution (in its context), he can be termed its 'architect'. Many times program construction has been compared to building houses, bridges, or large buildings. So using the terms 'architecture' and 'architect' in connection with program construction is in line with this comparison: from the Greek *archi-* (master-) *-tektoon* (-builder; the same stem as technique).

If one observes the production of software, the engineering of it, and the product itself, there are some software engineering aspects worth noting. The first aspects deal with the user expectations

[1] Stichting Informatica Onderzoek Nederland, 'Dutch Foundation for Computer Science Research'; Quote — transl HD.

[2] The notion 'facilitating' is due to Jan van Katwijk. We understand the notion to have, according to D.C. Browning's revision of *Roget's Thesaurus of English Word and Phrases*, the following related notions from 'ease' and 'aid': practicability, help, and, to smooth, ease, lighten, assist.

[3] Around 1968 at the IBM Laboratory in Böblingen there was this saying: "Helfen Sie bei der Lösung, oder gehören Sie zum Problem."

[4] .

of the solutions, the last ones relate to the way the programmer delivers the solutions.

- quality and dependability (correct, available, reliable, robust, stable, secure, safe and sound): characteristics of this kind are essential because the user wants his solutions to function in the way he wants them to, at the moment he wants them to, and as long as he wants them to, with as little maintenance as he is willing to pay for;
- flexibility; easily modifiable: which is essential because of the often (fast) changing circumstances (internal, as well as external). It is also one of software's characteristics, its malleability compared with that of physical constructions: try refitting a Continent bought car for further British use.

  On the other hand, one might point to this malleability of software as just one of the main reasons of the software crisis. As errors can be so quickly patched (if found), why bother to try not to make them. Also, quickly adapting a program to whim or wish is too easily done.
- within the budget (people, time, and money); this entails striving for higher production rates through automation of the process, contracting out to a low-wages country and through the possibility of using (parts of) earlier solutions: *reuse*. Reuse implies that these earlier solutions can be formulated and subsequently found. One should not forget the Bethesda (or NIH) syndrome,[5] which might account for — through psychological processes — less reuse then theoretically possible. A side-issue in this respect is standardisation: fitting other solutions into an existing framework is easier if the interfaces are standardised.

We stay close to the user, so our main concern here will be parts of the first aspects above, and of course, the language related points of them.

2.3.2 Overcoming Emerging Obstacles: Decreasing the Software Crisis

Even with positive factors present, the growing number of programmers, the hardware becoming more powerful and cheaper, and the growing number of technical and theoretical insights from the field of computer science, a 'Software Crisis' developed. In the last two decades much has been said about this software crisis. Already in 1975 it was almost a cliché to say that a software crisis existed [24, 83]. D.A. Fisher, at the 'birth' of Ada, remarks on its appearance "The symptoms appear in the form of software that is non-responsive to user needs, unreliable, excessively expensive, untimely, inflexible, difficult to maintain, and not reusable" [23,24]. Quite contrary to the aspects mentioned in the previous paragraph. And William Wulf remarks on its origins: "The 'Software Crisis' is the result of our human limitations in dealing with complexity. To 'solve' the problem we must reduce the 'apparent complexity' of programs, and this reduction *must* occur in the program text" [83]. On the other hand, we must not neglect the huge contact potential between the user layer and the programmer layer, which gave birth to this software crisis.

We already discussed (2.1) the formal language in which the solution for the user problem is described by the programmer. Now it returns in the form of the building blocks and mortar with

[5] "Not Invented Here", or, "NIH", or, the National Institute of Health, is situated in Bethesda, Maryland. Yourdon uses in [84] the phrase "NIH Syndrome", but in medical nomenclature syndromes, as they have to be pronounced readily, are named after people, places, pathogens, or objects.

which the program text is assembled, from the design laid out in the specification. The language should not only be close to the problem domain, it should also help to reduce complexity. Of course, in the beginning one tried to take recourse to natural language. But even in the plea for a natural language as a programming language, Jean Sammet made the restriction that she "very definitely [wanted to] include mathematics or any other scientific notation" [62]. And she goes on, "I think more time is spent worrying over the format of a DO-statement, or a **begin - end** bracketing, then in whether one should be looping or bracketing at all. [...] If [the programmer] could have stated the problem in the way most natural to him in the first place, he would have saved lots of time." We will return to this statement when discussing the consequences of hardware structure and constraints.

Terry Winograd takes a slightly different tack, but again takes recourse to the notion of 'naturally' describing problems: "[In natural language we] reduce complexity by allowing imprecision when precision is not required. This is not an excuse for avoiding all precision, or a justification for 'natural language programming'. We need to understand the deep psychological properties of how people understand language, not mimic its superficial forms. The justification is not that natural language is 'better' in some abstract sense, but that it is what we people know how to use. The most essential feature of a programming formalism is its understandability by programmers. [...] We cannot turn programmers into native speakers of abstract mathematics, but we can turn out programming formalisms in the direction of natural descriptive forms" [80]. So 'natural' is not 'superficially alike natural language' (compare this with the appearance of, and the motivation behind, the common business oriented language, COBOL), but 'natural' is 'more in character with the problem described', and as such, it is more in character with structured programming.

In describing the problem in a more formal way, the building of its model, one might have to use or invent mathematics — new ways of describing the problem. The language one uses for describing the model, must have a 'natural' way of representing these formal notations and the questions defined on them: Jean Sammet's 'scientific notation'.

Traditionally, system development methods start with requirements, specifications, or the like, and then give much attention to the construction – testing part, the bottom of the waterfall (method). In constructing 'theories' or 'classes of computation', it is essential to stay at the top of the waterfall, because the user is still present there. We use natural language to communicate with the user, and use natural descriptive forms in communicating with the computer. If we use formal forms at the start, and are able to formalise (in essence, to automate them) all the steps in the process of creating the solution from then on, we decrease the amount of programmer time spent on housekeeping details: coding and testing, or 'hunting for bugs' (an anthropomorphic term, by the way, 'finding programmer errors' is the more professional phrase).

The Problem Solving Cycle As has been mentioned in the Introduction, it would be more in line with the solution we seek for the user problem, to have an attention shift from the later parts of the system development cycle to the early parts. The state-of-the-art in computer science is such that the later stages in this cycle can be handled more than adequately, as there are enough knowledgeable people around to get forward, iff these people are committed to this task. So we extend the early parts of the cycle to cover the whole process of problem solving. To get a proper shift of attention we rename the 'system development cycle' to 'problem solving

cycle'.[6]

This extended problem solving cycle can be viewed as a meta-method, steps the programmer has to take, but it is up to him how to take those steps, many different techniques can help him in that. We can characterise these steps as follows:

1. human *conceives* idea concerning a problem;
2. idea *described in* requirements;
3. requirements *quantified into* specifications;
4. specifications *formalised as* formal specifications;
5. formal specifications *executed in context, yield* prototype;
6. prototype *executed, yields* observable result;
7. observations of result *verified against requirements, make* Naur's 'theory' (viz the prototype)
   (a) *go back to* human, idea, requirements, or, (formal) specifications, if the results are not quite up to the user's (or programmer's) expectations: the prototype does not satisfy its requirements,
   or,
   (b) *be transformed or compiled, yielding* a program, now being an (efficient) solution;
8. solution *used in practice, yields* solved problem;
9. observations of solved problem *held against current 'theory', make* the solution
   (a) *go back to* human, idea, requirements, or, (formal) specifications: solution not any more up to the user requirements,
   or,
   (b) *return to* step (8), if problem still within solving capability of solution.

Note that at the beginning of this characterisation we use 'human', not 'user', as that would include the organisation. If an organisation is involved, there is always the problem of interpreting and representing the views of the organisation regarding the problem. Next to this extra complication, there is also a distinction between the user or buyer of the software, and the end user, the person who actually enters into a dialogue with the software: the former user being his manager or director, representing the organisation. We will leave this problem open, its solution lies outside computer science; here we start with a human being, voicing an idea.

There is another *caveat*. As prototyping is essential in this cycle, the only way to achieve results is to keep the user involved. If the user has to wait too long to see the results of his comments on the functioning of the prototype, his involvement will peter out. In this way the essential

[6] The same attention shift should be applied to the term 'Hospital Information System' in, a.o., informative brochures for the patient. The data in these systems consist of 99% patient or patient related data, so the term does not reflect that hospitals exist for patient care: 'Patient Care Systems' would be a more appropriate term, better reflecting the reality the patient expects.

communication between user and programmer on the refinement of the problem description is seriously handicapped.

The first step, obviously, does not involve the programmer (unless he is the subject). The programmer helps in step (2) to get at the requirements. The language mismatch occurs at the intersection of steps (2) and (3), historically accounting for the development of various sub-disciplines of Applied Informatics: e.g., the so-called *alpha-* and *gamma*-informatics and medical and juridical informatics.

We make an essential distinction between requirements and specifications. Requirements being more inclined toward the natural language, and the problem domain, specifications being written in the language of the programmer, more toward the formal descriptive form. We leave open the question, whether requirements can be formalised at all: if they can be formalised, then we can use them as specifications. So requirements might have the inclination to stay in the informal, descriptive, qualitative domain.

There are, however, new fields emerging in computer science called 'Requirements Engineering', and 'Formal Specification Methods'. Requirements engineering has the transition from step (2) to step (3) as domain. The division between (3) and (4) is significant: first quantifying and then formalising is a trusted mathematical method. Turner demonstrates there is a difference between a specification and a formal specification [71]. By mathematical methods he derives a computable set of recursion equations (an actual program) for the Hamming number problem, from a clear, unambiguous, complete, but non-computable specification, as it involved sorting over infinite lists. The formal specification can be written in any formal language, ranging from the so-called formal specification languages, like Z [60, 82] and VDM [8, 37], to the functional languages.

Design is mainly about the modularisation process, the basis of steps (3) and (4). In the next Subsection we will further discuss this process, finding that step (4) needs a language which supports the modularisation process. The power of modularisation of a problem lies in smaller, simpler, and more general modules, through which productivity is improved:

- small modules can be coded quickly and easily;
- general purpose modules can be reused, thus accelerating the construction of subsequent programs;
- modules can be tested independently, thus reducing time spent in debugging.

However, these modules must *not* exhibit unwanted or unexpected effects. Reusing modules with such hidden effects hinders the programmer in functioning at a higher level of abstraction, as time after time he has to deal with these problems and cannot concentrate on the problem to be solved.

Formal specification methods are used in steps (4) and (5), it is a route more formal than the one we have taken. Our route is the practical route of extending a programming language. One of the problems of strict formal approaches is that a formal description must necessarily describe a closed system. "However, it is often the part that is excluded from the model that can cause most modelling difficulties." [68]. We alleviate this weak point by introducing the user in the prototyping process. It is in the spirit of executable specifications that their natural sequel is

the step of (one more) automatic transformation into an efficient solution. We will return to this automatic 'transformation' of (7b) in Section 2.5.

The distinction between steps (5) and (6) is a subtle one. We made it in order to differentiate between readable specifications, and an executable object; if they are the same, we have one step less to take.

In many circumstances, the life cycle does not start *ab initio*, but in the case of an existing system, the requirements or specifications start to change, step (9a), due to internal or external influences, so the cycle starts again at (3) or (4), a process called maintenance. Often in the methods, steps (2) and (4) through (6) are omitted, and one goes straight from the results of step (3) to step (7b) and (8), in which case 'transformed or' must be read as 'coded and', and testing must be taken into account too. These jumps might account for many unwanted systems, as the user was not really involved in the whole process. This omission resulted in serious misunderstandings occurring in the first steps, mistakes that were not found out in time, the results of which on cost of repair of errors are well documented in literature: "It is widely believed that the 'waterfall' software lifecycle often used commercially [...] has served to exacerbate the problems of software development and maintenance by delaying the discovery of incorrect or inappropriate specifications and requirements until the testing phase that follows implementation. It has been estimated that the cost of correcting such an error or bad decision increases by a factor of 10 for every phase of the waterfall lifecycle through which it passes undetected" [26]. A language which facilitates the taking of all the necessary steps from above, especially those involving the user to get consensus on the problem to be solved, alleviates the main problem. The programmer, using the right language, can get the solution to the user as fast as possible through easing the way by not introducing more language shifts than necessary.

Of course, there are many problem solving methods around, e.g., Michael Jackson's JSD [35], James Martin's Information Engineering [43], Sjir Nijssen's NIAM [52], Hatley / Pirbhai [32], and Yourdon / DeMarco [20]. These methods are well-known and easily obtainable. However, these methods have a fixed structure, and are in general, not geared to prototyping as a method, but can accommodate it as a technique. So we could not use one of these methods to demonstrate our point regarding the attention shift, which is a kind of *Gedanken* problem solving cycle. These methods (or parts of them) must be used as a tool (or a technique) in the hands of the programmer in providing solutions, they must not be considered solution producing mechanisms.

Consequences of Hardware Constraints and Possibilities In 1.2 we described the evolution of programming languages based on the Turing-Von Neumann hardware model. As is clear from that description, the 'high level' programming language did still retain a notion, albeit abstract, of the underlying hardware. Since 1977, when John Backus held his Turing lecture, this has been called the 'Von Neumann bottleneck': in the underlying hardware, and so in the programming language, data and instructions are moved one word-at-a-time from memory to CPU and *vice versa* [6]. Because of this bottleneck path, programming in these imperative languages necessitated much more attention for controlling the movements and whereabouts of these data and instructions, than for solving the problem at hand, thus not achieving breakneck speed in delivering the solution. Apart from the data movements, there is another problem attached to the underlying hardware. The programmer has to define the order of the steps to be taken to reach the solution. This organising and comprehending the exact flow of instructions

is a burden placed on the programmer by the Turing-Von Neumann model. Hence the 'tedium' of programming in such a language [58].

It is not expedient today to avoid the hardware Von-Neumann-bottleneck: it became the logical building block for years to come, after it outlived the special purpose hardware, e.g., the LISP-machines designed for this first functional language, in the early eighties. We need to avoid this bottleneck in our software, as it shapes our thought,[7] recall Jean Sammet's remark on time spent worrying over syntactic detail. We must strive for reduction of complexity, first of all in the language, which is what makes up the program text. A simple language avoiding this bottleneck because it does not follow the Turing-Von Neumann computational model, will alleviate this problem too. With this language the programmer can spend his time concentrating on the problem, instead of on things like loop indices.

Mathematical and Logical Properties of Languages We have not yet touched upon the correctness of the solution we aim at. The aesthetic school of computer science teaches us that it is difficult to verify a program after it has been written, it is better to employ a methodology in which correctness accompanies function: program and proof must be developed together. Their methodology is based upon extensions of Dijkstra's 'guarded commands', and Hoare's 'pre- and post-conditions', but it extends in the direction of the Von Neumann languages. We hold that many of these proofs are unnecessary if one really abstracts from the hardware, like John Backus did. This is the direction of the declarative languages, among which are the functional ones. It is not proof of recursion that bothers us here, the programming equivalent of the mathematical proof by induction. What bothers us in these correctness proofs, is the presence of loops, their initial states, loop invariant, and exit condition, and the assignment which accounts for the combinatorial explosion of the imperative language program state space. The reason behind this uneasiness is enclosed in the language. If there is a language which in its computational model follows a powerful mathematical system in which proofs would amount to canonical reduction of the program, such a language would help in developing program and proof together. Thus by avoiding the bottleneck (a.o., no assignment) we get better programs.

The Engineering of Program Production In a previous Section we stated that there will always be a need for programmers as there will always be non-standard problems to be solved. Solving those user problems, even in a large organisation set to solve such problems, and holding to professional staffing of such an organisation, we must not forget that they are non-standard problems. After the fact, i.e., after the problem has been solved, one might categorise it into some equivalence class of problems, but not beforehand, without investigation.

One should not try to look at programming as a process that can be converted to the Scientific Management of Frederick W. Taylor (1856–1915). This is the process of systematically dividing the work to be done by (a) man(-machine combination) into independent, ever-decreasing tasks, which eventually, can be given to 'robots', no intellectual effort being necessary any more. (Taylor might be called a Turing man *avant la lettre*, but for the fact that in his time, the prevailing 'defining technology' was the steam engine, so man was seen as a source of power.) The above sketched process of programming is not to be found in a car factory, where skilled technicians

[7] According to the Sapir-Whorf hypothesis: "Language limits thought".

assemble a car from exact specifications (for quality reasons, the Scientific Management's assembly line in its pure form, is even there not practised so much as before). In this case it is more appropriate to look at the processes in a hospital, where patients, by definition non-standard, are treated for their problems by professionals. After expert diagnosis, which is not jumping at symptoms, the problem can be classified, and the patient treated for his problem. Western society does not want narrowly trained symptom fighters, but well educated physicians to care for its patients.

David Gries, while discussing myths of structured programming — organising one's thoughts in a way that leads, in a reasonable amount of time, to an understandable expression of a computing task; understandable, because the structure of the program text reflects the structure of the computations it evokes —, finds evidence of this attitude [31]. He summarises a book by Philip Kraft, *Programmers and Managers* (Springer-Verlag, New York): "[Kraft] comes to the conclusion that to most of [the programmers and managers], 'structured programming' is an attempt to 'deskill' the industry, to make programming so simple that the task of the programmer is simpler and can be done with less trained, less paid programmers. Thus [Kraft] says: 'Structured programming, in short, has become the software manager's answer to the assembly line, minus the conveyor belt but with all the other essential features of a mass-production workplace; a standardised product made in a standardised way by people who do the same limited tasks over and over without knowing how they fit into a larger undertaking.' "

This attitude makes Gries pessimistic about the practice of using the methods the aesthetic school of computer *science* advocates, at least in the near future. But that was fifteen years ago. At the same occasion, he states: "The static program should be as close to the dynamic aspect of the program — how it gets executed. The conceptual gap between the program as we read it and the program as it gets executed should be as small as possible" [30]. Which is not as we see it now: we must not give attention to the execution of the program, but to the solving of the problem. The conceptual gap between the program as we read it and the structure of problem as it is defined, should be as small as possible. Structured programming, as the sound basis of program construction, should encompass the problem structure, and take its distance from computational structure. So we need a language in which this conceptual gap is as small as possible.

Another point we can see now, is the impracticality of thinking out all details of a program in preparing it for an 'assembly line' software production model. Before everything has been written out in detail and the workers have been instructed, the first change in specifications will announce itself, which in the most optimistic scenario involves only one work unit. Dijkstra once remarked, "It could well be that all kinds of industrial processes cannot be organised, if one does not rely on the professional competence of one's employees." Today, as should have been common practice earlier, it is flexibility in delivering and adopting solutions that counts; and flexibility can be assured by a language that facilitates fast and easy changing of program text under correctness preserving constraints.

2.4 The Shape of Solutions: Design

How to give form to substance, is the main subject of this Section. As the professional programmer has to apply known principles, we discuss here design principles, or guidance in making a prototype, a product, or in changing a prototype or product. As these design principles are valid for new and existing constructions, we only treat the general principles. We do not treat the application of these principles in both cases, one can find examples in [9]. In Chapter 4 we will return to this guidance regarding some design decisions we take there.

2.4.1 Design: An Engineering Problem

Before discussing software design, we must point out some properties of software that permeate the following discussion. These properties are easily forgotten due to the fact that not only users, but programmers too, tend to prefer a tangible object (prototype) to an abstract subject (a class of computations). We follow mainly Peter Wegner in this discussion [78].

Software is not physical, it does not wear down as physical objects tend to do, something which necessitates their physical maintenance. Software is logical, it represents a process, it handles symbols, not objects, it models a part of reality. As reality changes with time, software must also change, lest it loses its function. Hence also maintenance, not due to the physical embodiment of the product wearing down, but because of the logical embodiment of the product not being valid any more. This logical aspect, which implies using logical values, links to other properties: software is not a continuous system, and, progress in software construction is not visible. A physical system is continuous in the sense that a small change in specifications or in input leads to a small change in the product or in the output. In general, this is definitely not the case with a software system. As software is logical, the difference in progress of placing statements, when creating a formal system, compared with when creating a mere invalid one, is not obvious to the untrained eye. Progress is clear, however, even to the untrained eye, when one observes the construction of a bridge. Nevertheless, we take it that a logical system does have substance, apart from the physical form of its carrier: the formal system which describes the solution, and which description is mechanically interpreted. This logical substance, not physical, leads to the above mentioned malleability of software: logic does not resist change (but it can just turn invalid).

There are two ways of using design to shape the substance. The first one is in the creation of a new product, which is creating a solution for a user problem. The other one is less obvious: when designing a change or expansion of an existing product, the process of adapting an old solution to a changing environment. In expanding an existing construction, one has to stay true to the original design principles and decisions, the results of which are present in the existing construction. In trying to get at the architecture of a product when the why and what of an architectural description are absent, and the architecture of a software product in particular, a certain amount of reverse engineering must be used [76]. Many problems in programming developed because of not adhering to the architecture of the programs, their meaning, when changing them. Even worse, when the programs do not have a clear meaning to start with. Again the words of Dijkstra, Naur and Winograd from the Introduction: it is the *meaning* of the program that has to be conveyed to fellow-programmers. However, our problem in Chapter

4 concerns the existing construction built around and into Twentel, of which these decisions are known, so we did not have to use reverse engineering.

The engineering problem we are faced with in this dissertation, is the seamless fitting of a new part into an old construction. Which means, retracing the principles of that construction, and the way it is built, and holding to those principles, finding a way of introducing the new functionality into the old framework in harmony with the existing construction. Though in itself a programming language is a meta-construction, the constructional aspects lie in a plane, different from the one in which the language is used by programmers to build programs.

Creating something from material substances, thus yielding a new artifact, using known theory and technology, is called *engineering*. The previous Section dealt with software engineering. We recognise a part of its essence in Mary Shaw's compilation of common phrases from various definitions of *engineering* itself:

> "Creating cost-effective solutions ... to practical problems ... by applying scientific knowledge ... to building things ... in the service of mankind." [67]

Thinking about the problem to be solved, then conceiving, and subsequently, shaping form of the outside (and inside) of the new artifact created to solve the problem, is called *design*. Design is a classic trial-and-error process, the ability to create a true solution in (apparently) very few steps takes a master designer. Christopher Alexander, an influential scholar in architectural design circles, defines it as: "the process of inventing physical things which display new physical order, organization, form, in response to function. [...] The ultimate object of design is form" [2]. He also demonstrates it to be a trial-and-error process, by presenting in [2] an algorithm to arrive at a certain optimal solution, given the various requirements of the problem; it is in fact, a linear programming algorithm. He elaborates on design — 'looking for goodness of fit' — as follows,

> "[A general way of stating design problems] is based on the idea that every design problem begins with an effort to achieve fitness between two entities: the form in question and its context. The form is the solution to the problem; the context defines the problem. In other words, when we speak of design, the real object of discussion is not the form alone, but the ensemble comprising the form and its context. Good fit is a desired property of this ensemble which relates to some particular division of the ensemble into form and context."

Generally spoken, engineering is the only field, apart from the biological sciences, in which new things are created. In engineering one can practice classic trial-and-error design (if time, cost and practicality permit), in the life sciences it is just waiting for the practice, no preceding design is possible. (Though in modern genetic engineering the life sciences and engineering meet each other.) So design is a typical engineering problem. But how is design done? There is more to it than mere scientific knowledge and economics. Hans van Vliet even admits to the involvement of some kind of magic to arrive at the definite form [74].

In the first Section of this dissertation we admitted artistic criteria for judging the observable form of a software product. Gustave Eiffel, the architect of *la Tour Eiffel*, offers an appropriate quotation, taken from the Paris newspaper *Le Temps* of 1887,[8] when he defends his creation against its critics:

[8] We take it that Bertrand Meyer (the author of [46]) translated this quote, as *Le Temps* is not *The Times* [7].

> "Must it be assumed that because we are engineers beauty is not our concern, and that while we make our constructions robust and durable we do not also strive to make them elegant? Is it not true that the genuine conditions of strength always comply with the secret conditions of harmony? The first principle of architectural aesthetics is that the essential line of a monument must be determined by a perfect adaptation to its purpose." [46]

Eberhardt Rechtin states the position of the engineering side as: "The design of complex systems [a collection of things related in such a way as to produce a result greater than what its parts, separately, could produce] must blend the art of architecture with the science of engineering." [61]. With this view he corroborates our views expressed in the opening Chapter.

Alexander voices a warning about designers — architects — who cannot cope with the necessity of delivering a form which has to fit in a context with ever increasing complexity: "Today functional problems are becoming less simple all the time. But designers rarely confess their inability to solve them. Instead, when a designer does not understand a problem clearly enough to find the order it really calls for, he falls back on some arbitrarily chosen formal order. The problem, because of its complexity, remains unsolved". He strengthens this warning thus: "The modern designer relies more and more on his position as an 'artist', on catchwords, personal idiom, and intuition — for all these relieve him of some of the burden of decision, and make his cognitive problems manageable. Driven on his own resources, unable to cope with the complicated information he is supposed to organize, he hides his incompetence in a frenzy of artistic individuality. As his capacity to invent clearly conceived, well-fitting forms is exhausted further, the emphasis on intuition and individuality only grows wilder" [2].

In program development one tends to hide design indecision, the inability to proceed from first principles, by introducing a development method into the problem solving process as the standard, that only can solve some problems in the later stages of the cycle. One of the dangers of having found a good form, or system development method, already proven usable in some situations, is the '*f*' fallacy: 'dutifully following form infers a fully faultless function'. An essential part of the form is the context, and many a time the context is taken for granted, "If it looks the same, then it *is* the same, (and if it isn't, it should be)". However, we must extenuate these statements. It is a natural human defence mechanism to resort to ritual whenever personal stress reaches, or is expected to reach, a certain level. But with software engineering evolving toward science, this means that we have to leave the individual position of the artist in designing software; resorting to art or ritual (as is manifest in a standard method) is not a defendable position on the science path we take engineering to go. A way of escaping from the trap of "not understand[ing] a problem clearly enough to find the order it really calls for", is to use prototyping, as it can help both user and programmer to discover other problem - context relations.

There exists a whole range of standard design methods incorporated in the standard problem solving methods we mentioned at the end of 2.3.2. The comments on these methods are made in the same vein as we held the standard-solutions discussion at the end of 2.1.

Many problems — even if they necessitate a non-standard solution — exhibit a standard structure for which the standard design methods are applicable. An essential pre-condition of this decision process regarding a design method is the (full) knowledge of the problem context, which

can be brought to light by prototyping. There is nothing amiss with using a standard design method for a standard problem, iff the chosen design method is fully applicable in the context of the problem.

Standard design methods are powerful tools in the hands of a professional, if they are properly used in their respective domains. Here again, it is the professional designer choosing the right design method. A large deal of the outcome of this decision is visible in the end-product. "Although individual performance remains the largest identifiable factor in software design quality, it is not independent of good design practice. Good designers use good practices. [and ...] the effectiveness of a particular practice or heuristic can change over time as the computing environment changes" [16].

This dissertation concerns itself mainly with languages used in prototyping and embeds prototyping in the problem solving cycle. On the fly we stated some remarks on the application of standard packages and problem-solving methods. We contend that standards in this respect should be applied by professionals, which implies a full knowledge of the problem such that the optimal standard method can be chosen, provided that there exists such a method for the occasion. Nevertheless, there will always be the basic design questions which have to be answered.

2.4.2 Common Design Principles

Why does one want to overcome complexity in building systems? As the single user must be able to understand the design — after all, he must use the solution —, it must be clear and understandable. One man must be able to do the design, think it fully through, and master it to such an extent that he can write about it in a clear, cohesive way. Only then, chances for a consistent and comprehensible, a conceptually integrated design having been reached, are better. A committee, designing a system, has serious problems in delivering a clear, cohesive, consistent and comprehensible description of its designed system [13]. Elsewhere this is called the 'one-architect rule', "the notion of a single designer controlling an entire computer architecture is not only practical but in fact *essential* for good design" ([9], emphasis added, HD). And in a very well motivated way in [13] Brooks states: "Conceptual integrity is *the* most important consideration in system design." Apart from this language based argument, there is the more common one, as the things to be constructed have to be manageable by the team, or person, that is set to construct them.

Another angle of structuring a complex system is described by Robin Milner, when he uses algebra to describe concurrency. In the next Chapter we will appreciate his view on algebra in this respect. A structured view of a complex system is essential in understanding it. This view gives a way to interrelate the many parts in a systematic way. An algebra has as one of its prominent features "that its expressions, by their form, either exhibit the structure of the objects which they represent, or exhibit the way in which those objects were built, or could be built, or may be viewed. Often indeed an object does not *possess* structure, but we *impose* structure upon it by our view of it — and thereby understand it better." [48].

In this structuring one encounters the following well-tested design principles in overcoming complexity:

separation of concerns, which allows us to deal with different individual aspects of a problem, is a notion comparable to *divide et impera* and *séparation des pouvoirs* (Parnas [54], Polya [59], Turski [73], and Plato (*Phaedrus,* 265E, "... separation of the Idea into parts, by dividing it at the joints, as nature directs ..."), and Montesquieu);

abstraction, where we concentrate on the important aspects of a phenomenon and ignore its other aspects [66], a kind of 'separation of concerns';

generalisation, where repeated patterns are brought into one pattern with proper parametrisation, a kind of 'abstraction'.

All three principles are used in *modularisation*, the key technical concept as the Encyclopædia Britannica states: "Research has shown that program design should be based on hierarchical structure and functional modularisation, with each module independently working on only certain aspects of the overall task and communicating with other modules via minimal data interfaces" [28]. To be able to build complex systems we have to use modularisation techniques in a top-down fashion: decomposing the problem in a small number of simpler subproblems. This can also be done by the method known as 'stepwise refinement' [81], the process repeated until the level of existing resources is reached. Constructing the building blocks for these systems — apart from the inevitable straight top-down design and coding of them — is done using generalisation on low level modules or constructions in a bottom-up fashion. Furthermore, we will separate the data from the processes operating on those data, and hide the implementation of the required datastructures, introducing the Abstract Data Type. 'Information hiding' [54] implies separating the databases, terminal I/O, and other communications into independent modules in order to describe the interfaces between them.

The interfaces between the modules should be clear, and orthogonal, and minimal: the number of notions in it should not exceed the magical number Seven [47]. Seven being a kind of psychological limit of immediate understanding, seeing, or comprehending a set of elements making up a whole. In an interface this means that this 'whole' consists of, a) the functional description of the interface (one's recollection of it) or the result of executing the function, b) its name in the programming system, and c) the sequence and *raison d'être* of its arguments, which leaves us with d) at most four arguments. With seven arguments it becomes inevitable that one must take recourse to the manual. Anton Nijholt mentions another 'natural' limit of our brain capacity when discussing Victor H. Yngve's Depth Hypothesis in natural language handling [51]: "When generating a sentence with a phrase structure grammar it is necessary to 'remember' nodes of an associated tree. [...] According to the Depth Hypothesis only seven of such nodes can be remembered." This constraint on the number of notions to be remembered and manipulated at the same time, makes reuse of well-known and well-understood systems important. It is easier to insert a minimal set of old notions into the new system than to insert a set of new notions, being a new subsystem built to identical specifications compared with the well-known system. That is because the minimal set is better internalised by the programmer than the new set, of which also the interrelationship must be remembered.

It is not only elegance which calls for these principles: also the robustness and durability (cf Gustave Eiffel), and the safety of a system is served by these concepts. Peter Neumann, the 'Internet'-editor of "Risks of Using Computers", mentions the following notions, when he discusses the design of safe and secure systems — abstraction, encapsulation, information hiding,

(strong) typing, separation of mechanism in design, separation of duties in operation, polymorphism, inheritance, and modularisation [50]. But these are techniques or properties of a language. They do not constitute a design method.

What do we have to take into account when designing a new system, what matters do we take for granted when writing requirements. However, we can use the following guiding principles of good design, when proceeding from requirements to specifications:

- Design for simplicity; Bauer: "Simplicity is not an add-on feature."
- Design for independence; Bauer: "If you don't need it, you shouldn't have to pay for it."
- Design for change, [27], so prepare for Perfective, Adaptive, and Corrective Maintenance. The last one has the programmer-made error as source. The first two are the real sources of change in software. The modifications in the programs are due to a change of the social environment, or due to a change of algorithms, data representation, peripheral devices, and change of the underlying (abstract) machine, which are all programmer related changes.
- Design an appropriate input language, and language in its broadest sense, so it includes the user interface — communicating with the user; it is an integral part of designing the system interface.

These guiding principles, taken from general literature, already show the main aspects of good design: simplicity, independence, change. We already mentioned them in 1.3. But they remain too vague, or become too specific, thus turning into the craftsman's rules of thumb; it is not a system of abstract notions.

2.4.3 Architectural Design Principles

As Eiffel already hinted at, when discussing design, a few strongly related notions spring to mind: 'elegance', 'good taste', and 'aesthetic principles'. A lesser known notion in the outside world is 'orthogonality', the mathematical concept from linear algebra — so abundantly present in [79]. Three Dutchmen, Aad van Wijngaarden, Gerrit Blaauw, and Edsger Dijkstra, have written in one way or another about these notions. It is perhaps not strange, that it is from a mathematical institution where all three were employed at the same time in the mid fifties (the *Mathematisch Centrum* in Amsterdam), that the advocacy of an idea like orthogonality in computer science emerges. Though the notions for judging a mathematical proof 'elegant' are not very well defined, every mathematician knows an elegant proof when he sees one.

There is more support from mathematics regarding orthogonality and aesthetics. In the following quote from Carney's *Introduction to Symbolic Logic*,[9] which we will follow up on in the next Subsection, we see (with 'independence' to be read as 'orthogonality'):

> "Consistency is essential; soundness is needed; completeness is most desirable, though not always obtainable; independence is obtainable, aesthetically pleasing, and can reduce one's labors in metalogic".

[9] Carney, J.D. *Introduction to Symbolic Logic.* Prentice Hall, New York, 1970, Section 8.4, p 198; as cited in A. Ollongren, *Definition of Programming Languages by Interpreting Automata.* Academic Press, London, 1974, p 99.

A quote from G.H. Hardy's *A Mathematician's Apology* (1940)[10] links Knuth's 'art' again into our thread: "The mathematician's patterns, like the painter's or the poet's, must be *beautiful*; the ideas, like the colours or the words, must fit together in a harmonious way. Beauty is the first test: there is no place in the world for ugly mathematics."

Around 1960 Fred Brooks used the term *architecture* for the first time in relation to computer systems, when he described the functional behaviour of the computer the user can see [29]. Later Blaauw and Brooks, two of the architects of the IBM/360, defined the *architecture of a computer system* as that what the user sees and perceives of the behaviour of a system [4, 10]. After having defined architecture, in hardware, they tried to answer the difficult question of how to judge a good hardware design. They stated a few principles against which the design to be judged should be held. As there is not much difference between hardware and software products at the architectural level, we apply their principles to judge a software construction [21, 64]. A software construction has an architecture too, a way of looking at the functional behaviour.

Blaauw and Brooks present in [9] the following threefold stratification of a product: AIR — Architecture, Implementation and Realisation, three independent design domains:

- *Architecture* is the functional appearance and the conceptual structure of the system as seen by its immediate user. Behaviour and structure the user cannot observe without special tools or attention, do *not* belong to the architecture. The architecture describes the function. This architectural description is a kind of user manual, describing how the various (required) functions can be activated, and to what effect.

- Beyond this (as the user cannot see it) lies the *Implementation*,[11] which is the organisation (data flow and controls) of logical 'building blocks' and other 'building materials'. If a system is 'built', or implemented, that way, it shows the behaviour which is specified in the architecture, no more, no less. The implementation yields the method to achieve the function described by the architecture. This method is the specification of the system, and contains, a.o., algorithms, functional structure, main data structures, and interfaces.

- Ultimately, the *Realisation* is formed by the physical structure of the product and the physical objects of which the product is assembled, the product which must exhibit the specified behaviour. The realisation gives a description of the means with which to materialise the method of the implementation. The 'looks' of the product are realised at this stage, one now can see the 'design' materialise. For a classic software product this is the stage of choosing the programming language, and the subsequent coding, using libraries, standard protocols, OS services and the like, and the subsequent testing of the product.

Each of these design domains has its purpose, an end-product, a domain language in which the end-product is described, and a quality aspect. So has the Architecture domain, respectively, function, principles of operation (an IBM term for a sort of user manual), natural and formal language of which the latter is normative, and, consistency.

[10] As cited in P. Naur, "Programming Languages, Natural Languages, and Mathematics", Invited Address to ACM's Second Symposium on Principles of Programming Languages (in Conf Rec 2^{nd} ACM Symp POPL, Palo Alto CA, Jan 1975, ACM, New York, 1975, pp 137–148).

[11] Note that in everyday software engineering practice, we use the term 'implementation' also in another sense: the process in which a software product is made to function correctly in a new environment.

They present a conclusive argument of layering the design in this way, it leaves the designers ample freedom of design to shape their products. Apart from the one architect (2.4.2), some teams of implementation engineers can be set to design systems that show the behaviour of the architecture, albeit *implemented* in a different way, to say nothing of the number of ways these different designs can be realised. This concurs with Dijkstra's 'classes of computation' to solve a problem: there does not have to be only one solution to a problem (not a puzzle), there can be more ways to solve it. Which solution is taken, depends on other requirements.

2.4.4 Evaluation of a Design

Blaauw and Brooks state their principles of quality as being "fundamentally aesthetic principles, describing the nature of beauty in computer architecture. As is always true with lists of aesthetic principles, clean design is a matter of balance, of judgment, of taste" [9]. Their main quality principle in judging good architectural design is *consistency*: a knowledge of the partial system (functions, stimuli and responses) must have a high predictive value for knowing what to expect of the unknown rest of the system. Once we know a system's reaction to certain stimuli, we will know how it will react to a similar new stimulus. Consistency can be described in three of its aspects:

- *Orthogonality*. The designer must keep independent functions separate: a change in one function should not lead automatically to a change in an independent function, or, not cause side effects. A positive way of formulating orthogonality is that the elements of an orthogonal system can be mixed without regard to arbitrary restrictions. A way of accomplishing this is through 'information hiding', and 'separation of concerns' (but both may also be grouped under 'decomposition') (Montesquieu) [56].

- *Propriety*. The designer must not put functions into the system, which are immaterial to the purpose of the system. This principle is the characteristic of a system meeting an essential requirement of its design (Antoine de Saint-Exupéry, well known (*Le Petit Prince*), but not as the aircraft designer he also was, said on engineering elegance: "A designer knows he has achieved perfection, not when there is nothing left to add, but when there is nothing left to take away"; Occam's Razor). Propriety leads to:
  - *parsimony*, "When in doubt, leave it out",[12] do not introduce improper things (bells and whistles), Lambert Meertens calls this 'too-muchness' [45]. Parsimony also prescribes that a function must not be provided in two separate ways;
  - *transparency*, a function is transparent if its *implementation* does not produce side effects.

- *Generality*. The designer must not put unnecessary, uncalled for, restrictions in his design. He must not restrict what is inherent in the function, the user must be able to use the design for whatever he can use it for, not hindered by ideas of the designer, who, in restraining himself, shows modesty in knowing that he cannot foresee the user requirements in the near future (Herakleitos). This aspect leading to:

[12] Even surgeons know this maxim attributed to Augustine: "*In dubio, abstine*". Of course, here we stretch the design part of the surgical profession a bit, much of it is done in maintenance, but certainly there *is* work done in design (reconstructive surgery, prosthetics, plastic surgery).

- *open–endedness*, create ample provision for future expansion;
- *completeness*, all functions of a given class, which do 'logically' belong to the purpose of the system, are supplied (this entails symmetry, and inverse operations: if there is a MUL there must be a DIV, with a LOAD comes a STORE, if there are bytes, halfwords and words, all operations, if applicable, should handle all three data widths);[13]
- *decomposition*, if completeness introduces an excess number of functions, one must decompose these functions into orthogonal subfunctions; the screwdriver set with one handle and a variety of bits, is such a solution.

Mark that Blaauw and Brooks do not express 'quality' and 'dependability' of the designed product in their principles, the aspects from 2.3.1. We assume that their notion of a designer is such, that designing according to their rules, implies designing for quality.[14] Apart from good practice, which means that relevant, measurable things should be measured if measuring can improve the production process and so the quality of the product, we postulate that quality, just like a 'theory' or 'classes of computation', can only be demonstrated ("Never mind the quality, feel the width!"), notwithstanding the existence of measurable aspects of quality [65].

In classic quality measurement one measures products against metrics. Measurable quality can only be measured at the Realisation stage, as this is where the product starts to materialise. (Recall the logic fabric of a software system, so think, before trying to measure a tautology.) Lagging quality can be caused by the processes used in realising the Implementation, in which case improving those processes improves the quality. If, however, the architect caused the low quality to be introduced at the Architecture or Implementation stage, then he failed his design: the form does not fit the context.

Wlad Turski mentions some properties of programs, not systems, but the two are interchangeable in his article. He is more inclined toward the programming-in-the-small style of the 'aesthetic school of programming', as can be seen from his criteria that have an altogether different angle compared with Blaauw and Brooks [73]:

- *correct*, and to facilitate proving correctness he calls for modularity;
- *adaptable*, to cater for changes in specifications;
- *robust*, behaving with a predictable reaction if confronted with unforeseen conditions: errors in input;
- *stable*, permitting 'graceful degradation' [54], if the system is confronted with incomplete and / or partially incorrect input.

Stef Joosten also gives a few general design considerations, tracing back to Blaauw and Brooks. He does not discuss them in depth, apart from mentioning that these notions are not absolute [38]:

[13] Note J.R.R. Tolkien and his creation of Middle Earth and the overwhelming sense of its grandeur through its completeness, from alphabets and languages, through genealogy, mythology and history. In another fictional space — with not so much impact on literature, but more on (Dutch) language in general — there is Marten Toonder with his Rommeldam saga. Also, within the juridical world, there are scholars who regard some law corpora as 'more sound', 'more consistent', than others [42].

[14] "Take care of the sense, and the sounds will take care of themselves," as the Duchess said. [17].

- *orthogonal*, like [9]'s;
- *liberal*, [9]'s generality: "A designer is not allowed to protect a user against himself; a patronising and protective attitude may eventually protect the user from using the system";
- *general*, [9]'s completeness, mixed with their 'propriety' characteristic;
- *robust*, handling errors, such that disaster cannot spread beyond the system.This principle is based on the same grounds as Turski's 'correct', 'robust', and 'stable'. As we contend it is implicit in Blaauw and Brooks' principles.

Recently, Giuseppe Scollo took most of Blaauw and Brooks' principles, but removed the classification, and he left out 'transparency'. He then calls them 'quality design rules', thus putting 'quality' into focus, true to the current period. He elaborates on some principles, differently from Blaauw and Brooks, [64]:

- *parsimony*, a parsimonious design solution goes straight to the point,
- *generality*, aspects that are potentially common to several parts of the design, must share the same description (something which is classified in [9] under 'decomposition'),
- *open-endedness*, Herakleitos' 'Everything flows' becoming the law of continuous change (or programs becoming obsolete),[15]
- *completeness*, a sort of complement of propriety,
- *consistency*, using Duns Scotus' 'Everything follows from falsity' in characterising inconsistency of a formal system as the derivability of every sentence from the given set of sentences. Here, having discarded [9]'s classification from consistency downward, he proposes 'consistency' as a meta principle, so re-entering it through the back-door.

The emergence of a 'correctness' as an independent design notion, might originate from the idea, that 'correctness' too, can be added to a product. Just as with 'simplicity', this cannot be done. This has been the firm conviction of the 'aesthetic school' since long, and from electrical engineering we have Rechtin paraphrase this position: "High-quality, reliable systems are produced by high-quality architecting, engineering, design, and manufacture, not by inspection, test, and rework." [61].

In the next Subsection we propose another design characteristic in the line of Blaauw and Brooks. This one incorporates in our opinion the 'elegance' and 'intuition' of a design, more than the above ones are supposed to do.

2.4.5 Intuitive Evaluation of a Design

Not only the formal, consistent functional system, which is strongly related to the architecture of it, plays a role in good design, also static aspects of the system, like its style, its 'looks', can be used to judge a system's consistency (Lambert Meertens, [45]). Another aspect of the 'look and

[15] "A slow sort of country !' said the Queen. 'Now, *here*, you see, it takes all the running *you* can do, to keep in the same place. If you want to get somewhere else, you must run at least twice as fast as that!" [18].

feel' of a system, is its 'intuitive conformity with reality'. The construction must be intuitive correct in its modelling of reality. The user does have ideas, expectations, and perceptions, whether rational or intuitive, about the way the system accomplishes its functionality. The design must not go against these appreciations, as otherwise the system, though consistent according to the logic of the designer, is inconsistent according to the logic of the user, who eventually judges the design.

Seen from this point of view, the intuition of the user, we propose a new characteristic in the classification of Blaauw and Brooks, a property in line with elegance and symmetry, subjugate to consistency:

- *Harmony*. The purpose (user defined and expected) must not clash with the functions the designer put into his product: the 'intuitive conformity with reality' should be correct in the eyes of the user.

  Following Van de Goor, we understand his additional property to belong to this characteristic [29]:

  - *balance*, the distribution of the functions should be in accordance with the foreseen use of the system (no emphasis on scarcely used functions, and note that the ubiquitous 80–20 rule should also hold).

An example of the inconsistency between program and user expectation can be observed in the problems which arise in the real time programming environment of dealing with multiple inputs on different input channels. In many cases it is essential to handle these inputs on the different channels at the time they arrive at the computer, or in the order in which they arrive. Stef Joosten gives a comprehensive treatment of the solutions of this real time programming problem in functional languages [38].

One of the described solutions uses David Turner's 'hiaton' concept [72]. All possible inputs on the input channel are defined as being events on that channel, and handling inputs is now dispatching the events on their channels. In order to be able to model this input handling process in a functional language, Turner introduced an event called *hiaton*, which is distinct from any other possible event that can happen on a channel. The hiaton is present at the channel when no other event occurs on that channel; it thus bridges the gap, the hiatus, between two subsequent (real) events. That hiaton event emits a special token which is handled by the channel program. However, the architecture of 'waiting' as a process, should not introduce something — for the user to be seen — when there is nothing present. Only when busy doing nothing, one can introduce such a concept.[16] So, in the case of a modem, one has the Carrier-Detect signal. If nothing comes in on the line, only the carrier signal is present, the Carrier-Detect, and so the hiaton, is there, and can be handled. If information comes in, the information itself is the event to be handled, the carrier is 'hidden' by the signal, and so is the hiaton overridden by the event. But when the carrier falls off, an entirely new situation is created. Mark that in this case no system signal remains active on which to base a reaction. Admittedly, the hiaton is used to model the polling process in the real time world, and indeed, when nothing happens

[16] As there exists a process 'busy doing nothing' (Turner modelled this by hiatons), there must be a unit in which to express the psychological weight of 'waiting'. We propose the unit *Godot*. To be able to measure it, we use the following metrics: 1 Godot being the time in sec needed by the geological process 'Continental Drift' to reverse its movement and subsequently, to cover 1 m .

in the outside world, the polling program is really 'busy doing nothing' in a waiting loop. In practice this solution will do, but theoretically some disadvantages remain. By using a hiaton at the first level of description, one introduces an implementation of the 'waiting' process at the architectural level, which is against the propriety criterion. This relates to an old problem: as the architecture is the user manual, we are now looking at an introduction of a programmer problem (it is difficult to model real-time problems in functional programming languages) into the user domain.

The architectural approach of Blaauw and Brooks is not a generally accepted paradigm in these terms. Richard Gabriel discusses two 'software philosophies' of major (academic) importance in the USA: 'the right thing', and 'worse is better' [25]. Both schools have the same characteristics, very much like Blaauw and Brooks' design principles. However, they accentuate these characteristics differently, resulting in two very different styles of programming language or system: Scheme, and Common Lisp (produced by the 'MIT school', at MIT or Stanford) versus C, and Unix (produced by the 'New Jersey school', at the Bell laboratories). Though the characteristics are the same — *simplicity*, *correctness*, *completeness*, and *consistency* —, it is the variation in their partial ordering on importance which makes the difference. Blaauw and Brooks recognise these notions too, albeit with correctness implied in good design.

First, the view of the 'MIT school', or, 'the right thing', then the view of the 'New Jersey school', or, the 'worse is better' approach. These two schools look upon these characteristics as follows (after [25]):

- *simplicity*, the implementation and the interface of the design must be simple. The simplicity of the interface is more important than that of the implementation; 'New Jersey': But the simplicity of the implementation is more important than that of the interface. And simplicity is the most important design consideration.

- *correctness*, the design must be correct in all observable aspects. Incorrectness is forbidden. At 'Bell' it is slightly better to be simple than to be correct.

- *completeness*, the design must cover as many important aspects as practical, and all reasonably expected cases must be covered. Simplicity may not compromise completeness. In 'New Jersey', completeness can be sacrificed for any other quality.

- *consistency*, the design must not be inconsistent. A design is allowed to be slightly less simple and complete to avoid inconsistency. Consistency is as important as correctness. The 'New Jersey school' holds that the design must not be overly inconsistent. Consistency can be sacrificed for simplicity in some cases, but it is better to drop functionality than introduce implementational complexity or inconsistency. Consistency can be sacrificed to achieve completeness if simplicity is retained; consistency of interface is especially worthless.

The difference between these two approaches can be seen in the immediate product as is evident, and can be taken as the survival characteristic of the producer (or the product, as is obviously the case with Pascal). In these approaches we see this survival characteristic, an economic aspect, as being an integral part of the design, which might be introduced into Blaauw and Brooks' categorisation. They see this survival characteristic as a personal trait of an eminent designer (e.g., Seymour Cray), a designer being able to make brilliant trade-offs between the

different design domains of the product [9]. So efficiency is not found in these characteristics either. We hold it as a design principle sub-ordinate to consistency and the implicit correctness.

As the 'How to design' is not yet very well understood, but design is done anyhow, this process is still far from the scientific model. We see that Blaauw and Brooks' main characteristic, consistency, is different from Gabriel's simplicity. And not only from this observation, we conclude that there is as yet no understanding of 'nature', Plato's directive for dividing the whole into parts: these natural laws are still undiscovered. However, as can be seen from the successes of the IBM/360 and the Cray, it is practical to think along the lines sketched in this Section.

2.5 Shaping of Solutions: Prototyping

Computer science is one of the engineering fields where the architect essentially can build his own system by using very powerful tools [67]. If one of these tools is a functional language, and the other tools lie outside the functional language, it is obvious to import the power of those other tools into the language. But many of the fundamental problems in applications development lie precisely in the description of the problem, not so much in the tools. And this is why prototyping is introduced.

In between design (creating something) and prototyping (having something) we find John Lansdown, a British architect, who states that no architect uses a white sheet of memory or paper, when imagining or drawing a new design. Also architects have a kind of *horror vacui*. The architect knows archetypes, examples (and a good architect knows many examples, together with their strong points and weaknesses in different situations), and uses them as malleable prototypes in his new design. This 'Lansdown' approach to prototyping is characterised by controlled modification of prototypes [40, 41].

What is prototyping? In 2.1 we dwelt on the literal meaning of 'prototype', referring to the first embodiment of the eventual solution. In the previous Sections we encountered some properties a prototype language must have. In this Section we present an operational definition of prototyping, covering those properties:

> "Prototyping is the process of constructing software for the purpose of obtaining information about the adequacy and appropriateness of the designer's conception of a software product. Prototyping is usually done as a precursor to writing a production system, and a prototype is distinguished from a production system by typically being more quickly developed, more readily adapted, less efficient and / or complete, and more easily instrumented and monitored. Prototyping is useful to the extent that it enables information to be gained quickly and at low cost" [26].

Another definition runs as follows: "Prototyping is the development of an application which, so much has to be clear, preferably *a priori*, only solves part of the total problem to be tackled, only fulfilling some of the requirements, not all of them. Which part is solved, depends on the situation: e.g., a part of the functionality, of the efficiency, or, of the user interface" [33].

In literature several kinds of prototyping are mentioned, e.g., throwaway, evolutionary, operational, explorative, and experimental prototyping. We will not discuss their respective merits

and faults, as we have described what we expect of prototyping, which function is covered by the above definitions.

Of the three ways a prototype can be brought to action in the 'problem solving cycle' (following [26]):

- as a basis for writing the specification,
- as the specification itself,
- as the initial implementation of the solution,

we tend toward using a prototype in the last two ways. The first way only uses the prototype for providing insight into the problem structure, which then has to be written out again, a step which need not to be taken. In the above definitions of prototyping, the transformational approach to prototyping is missing; however the other two ways of using a prototype merge in our view, as we advocated the automation of step (7b) in 2.3.2.

Imagine the following situation regarding the 'prototype as specification'. The user and programmer agree on a comprehensive description of the problem to be solved, the requirements, and some necessary business details, all in an Anglo-Saxon gentleman's agreement, 'bound' to an executable prototype, which can be constructed through the 'method' described in this Chapter, and which is accepted as 'that's the thing I would like to have' by the user. In a dispute over an aspect of the problem, one consults the prototype. This is an operational definition of the specifications. If the prototype does not give a conclusive answer, the prototype can be quickly adapted, it being an executable specification, written in a language in which rapidly changeable prototypes can be written. Note in this respect, that Blaauw and Brooks already stated that the architecture of a system needs an executable description, which is a sort of prototype. In [9] they illustrate this with the IBM/360's 'Principles of Operation', described and executed in APL.

This situation imposes an obligation on user and programmer alike. Trying to speak the same (user) language, bound by a gentleman's agreement, the user now holds his own responsibility in setting the requirements of his problem right. Causes behind newspaper headlines like 'Computer did such', and 'Computer neglected so', are now fully the responsibility of the user too, the user who helped to state the requirements and uses the solution.

'Prototype as executable specification' merges with the 'prototype as initial implementation' (its fundamental meaning, the first embodiment). The programmer can generate an efficient solution, based on his prototype, and not on some fat sheaf of paper. This is done, using the mechanical transformations the 'knowledgeable people' from 2.3.2 have prepared, which are based on already known 'program transformations'. Many times a program can be calculated by simple equational reasoning from a formal description of what it is supposed to do, the executable specification in the functional language. In doing it with these 'correctness preserving' transformations, the meaning of the program (agreed upon prototype) is preserved. If the programmer did not manipulate the original program into an 'efficient' one, the chances are very good that the original program, automatically transformed, is more efficient than the result of the same sort of transformations on the 'efficient' program. No redundant information has been removed from the original program, so the problem structure is better preserved, and the transformations can be more effective [19].

Among those who are engaged in this systematic derivation of an efficient executable program from its prototype, is Arie Duijvestijn. He is engaged in finding rules for the transformation from a Twentel prototype to Pascal, which is finding imperative constructs for declarative semantics. For most declarative constructs this is a straightforward task. The problems arise when dealing with the evaluation order of functions and with infinite datastructures which result from lazy evaluated streams [22].

This is one school of program transformation: automatic generation of lower level descriptions, or programs. The other school uses the high level description as a written specification for the programmer to be used in the construction of the next (lower level) program.

There are aspects of a solution which cannot be covered by a prototype:

- performance: metrics for system throughput measurements can be defined,
- dataspace: limits can be defined up front,
- considerations about quality of the product: these should be well-defined, and measurable [65].

By extending the prototype language with linking possibilities to the efficient outside world, we can alleviate some efficiency problems. Another important aspect of a solution is its interface with the user — the look-and-feel of the solution. By linking to existing generative user-interface modules, this aspect is factored out from the prototype proper.

Stef Joosten, in his lucid advocacy of functional programming as a software tool for prototyping, mentions two reasons for considering functional languages as the choice for this tool [38]: the language must allow constructive mathematics to be executed in a direct way, and the language must contain no exceptions that force the designer to occupy himself with programming, rather than problem solving. These reasons are respectively covered by a) the language must be amenable to scientific notation, and b) the language must free the programmer from the 'tedium' of programming.

The choice of the language in which to write the prototype has already been made, and will be strengthened in the next Chapter: a functional language. This choice is corroborated in [26] where the committee members recommend a language which allows both imperative and declarative or functional styles of programming.[17]

References

[1] Abelson, H., Sussman, G.J., with Sussman, J. *Structure and Interpretation of Computer Programs.* (with foreword by Alan J. Perlis), MIT Electr Eng and Comp Sci Ser, MIT Pr, Cambridge MA / McGraw-Hill Book Comp, New York, 1985.

[17] However, we think that Ada, an imperative language, as the target language of the developments in the prototype system they describe, enters the discussion because of the sponsoring agency: DARPA. We hold that this strengthens the case for the functional style.

[2] Alexander, C., *Notes on the Synthesis of Form.* Harvard Univ Pr, Boston MA, 1964.
[3] Alexander, H., Jones, V., *Software Design and Prototyping using me too.* Prentice Hall, New York, 1990.
[4] Amdahl, G.M., Blaauw, G.A., Brooks Jr, F.P., "Architecture of the IBM System/360", *IBM J Res Development* **8**(2) (Apr 1964): 87–101.
[5] Ashenhurst, R.L., Graham, S. (eds) *ACM Turing Award Lectures — The First Twenty Years, 1966 – 1985.* ACM Pr Anthology Ser, ACM Pr, New York / Addison–Wesley, Reading MA, 1987.
[6] Backus, J., "Can Programming Be Liberated from the Von Neumann Style? A Functional Style and its Algebra of Programs", *Comm ACM* **21**(8) (Aug 1978): 613–641 (1977 ACM Turing Award Lecture, 17 Oct 1977).
[7] Van Berne, H., *Personal Communication*, 27 June 1993.
[8] Björner, D., Jones, C.B., *The Vienna Development Method: The Definition Language.* Springer-Verlag, New York, 1978.
[9] Blaauw, G.A., Brooks Jr, F.P., *Computer Architecture — Vol 1: Design Decisions.*, Preliminary Draft, Lect Not, Privately Circulated, Univ Twente, 1975–1987, (to be publ by Addison–Wesley).
[10] Blaauw, G.A., " The Persistence of the Classical Computer Architecture" — A Survey from 1950 to the Present —, (Symp Computational Engines, CWI, Amsterdam, Sep 1989), *CWI Quarterly* **3**(4) (Dec 1990): 335–348, (material taken from [9]).
[11] Boehm, B.W., Gray, T.E., Seewaldt, T., "Prototyping Versus Specifying: A Multiproject Experiment", *IEEE Tr Softw Eng* **SE-10**(3) (May 1984): 290–303.
[12] Boehm, B.W., "Software Engineering", *IEEE Tr Comp* **C-25**(12) (Dec 1976): 1226–1241.
[13] Brooks Jr, F.P., *The Mythical Man-month — Essays on Software Engineering.* Addison–Wesley, Reading MA, 1975.
[14] Brooks Jr, F.P., "No Silver Bullet: Essence and Accidents of Software Engineering", *IEEE Computer* **20**(4) (Apr 1987): 10–19; (also appeared in: H.-J. Kugler (ed), *Information Processing '86.* North-Holland, Amsterdam, 1986, pp 1069–1076).
[15] Cann, D.C., "Retire Fortran? — A Debate Rekindled", *Comm ACM* **35**(8) (Aug 1992): 81–89.
[16] Card, D.N., Glass, R.L., *Measuring Software Design Quality.* Prentice Hall, Englewood Cliffs NJ, 1990.
[17] Carroll, Lewis, *Alice's Adventures in Wonderland.* Macmillan, London, 1865.
[18] Carroll, Lewis, *Through the Looking Glass and What Alice Found There.* Macmillan, London, 1871.
[19] Darlington, J., "The Structured Description of Algorithmic Derivations", in: *Algorithmic Languages*, J.W. de Bakker and J.C. van Vliet (eds), North-Holland, Amsterdam, 1982, pp 221–250.
[20] DeMarco, T., *Structured Analysis and System Specification.* Prentice Hall, Englewood Cliffs NJ, 1979.
[21] Van Diemen de Jel, P.P., *Hulpmiddelen bij het Ontwerpen van Grote Programma's.* MSc Thesis, Dept Applied Math, Comp Sci Section, Technische Hogeschool Twente, Enschede, Dec 1976, (in Dutch).
[22] Duijvestijn, A.J.W., Duijvestijn, L.M., *Inclusion of a Point in a Polygon.* Dept Comp Sci, Draft Memorandum, Univ Twente, Enschede, Jul 1994, 43 pp.
[23] Fisher, D.A., "DoD's Common Programming Language Effort", *IEEE Computer* **11**(3) (Mar 1978): 25–33.
[24] Floyd, R.W., "The Paradigms of Programming", *Comm ACM* **22**(8) (Aug 1979): 455–460, (1978 ACM Turing Award Lecture, 4 Dec 1978), also in [5], pp 131–142.
[25] Gabriel, R.P., "Lisp: Good News, Bad News, How to Win Big", *AI Expert* **6**(6) (Jun 1991): 31–39.
[26] Gabriel, R.P. (ed), Balzer, R., Dewar, R., Hudak, P., Guttag, J., Wand, M., "Draft Report on Requirements for a Common Prototyping System", *SIGPLAN Not* **24**(3) (Mar 1989): 93–135.
[27] Ghezzi, C., Jazayeri, M., Mandrioli, D., *Fundamentals of Software Engineering.* Prentice Hall Int'l, London, 1991.
[28] (G.N.G.), "Computer Science", in: *The New Encyclopædia Britannica.* Vol 16 — Macropædia, Encyclopædia Britannica, Chicago IL, 15^{th} ed, 1992, pp 629–637.
[29] Van de Goor, A.J., Spanjersberg, H.A., *Computer Architectuur.* College Notes, Delft Univ of Technology, Delft, 1982, 272 pp, (in Dutch).

[30] Gries, D., "Current Ideas in Programming Methodology", in [77], pp 254–275.
[31] Gries, D., "A Comment on Parnas' Counterpoint", in [77], pp 359–364.
[32] Hatley, D.J., Pirbhai, I., *Strategies for Real–Time System Specification.* Dorset House, London, 1987.
[33] Hendriks, P.W.M., *Personal Communication*, March 1994.
[34] Van der Hoeven, G.F., *Preliminary Report on the Language Twentel.* Memorandum INF-84-5, Dept Comp Sci, Univ Twente, Enschede, Mar 1984, 87 pp.
[35] Jackson, M., *System Development.* Prentice Hall, Englewood Cliffs NJ, 1983.
[36] Jones, A.K. (ed), *Perspectives on Computer Science.* (Rec 10th Anniversary Symp Comp Sci Dept, Carnegie-Mellon Univ, Oct 1975, Pittsburgh PA), ACM Monograph, Acad Pr, New York, 1977.
[37] Jones, C.B., *Software Development: A Rigorous Approach.* Prentice Hall, Englewood Cliffs NJ, 1980.
[38] Joosten, S.M.M., *The Use of Functional Programming in Software Development.* PhD Thesis, Dept Comp Sci, Univ Twente, Enschede, Apr 1989, 140 pp.
[39] Kuhn, T.S., *The Structure of Scientific Revolutions.* Univ Chicago Pr, Chicago IL, 2^{nd} ed, enlarged, 1970, xii + 210 pp.
[40] Lansdown, J., "A Theory of Computer-Aided Design: A Possible Approach", in: *Computers in Art, Design, and Animation*, J. Lansdown and A. Earnshaw (eds), Springer-Verlag, Berlin, 1989, pp 163–172.
[41] Lansdown, J., "Visualising Design Ideas", in: *Interacting with Virtual Environments*, L. MacDonald and J. Vince (eds), Wiley, London, 1994, pp 61–77.
[42] Lokin, J.H.A., Zwalve, W.J., *Hoofdstukken uit de Europese Codificatiegeschiedenis.* Wolters-Noordhoff / Forsten, Groningen, 1986, xiv + 402 pp (in Dutch).
[43] Martin, J., *Information Engineering.* 4 vols, Savant, 1986.
[44] McCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P., Levin, M.I., Lisp *1.5 Programmer's Manual.* MIT Pr, Cambridge MA, 1962 (2^{nd} ed, 1965, vi + 106 pp).
[45] Meertens, L.G.T., "The Design of Elegant Languages", in: *Conference on the History of ALGOL 68.* G. Alberts (ed), (Conf Proc "25 Years ALGOL 68", Amsterdam, 11 Feb 1993), Historical Note AM-HN9301, Centrum Wiskunde & Informatica, Amsterdam, Jan 1993, pp 53–64.
[46] Meyer, B., *Eiffel: The Language.* Prentice Hall, New York, 1991.
[47] Miller, G.A., "The Magical Number Seven, Plus or Minus Two: Some Limits on our Capacity for Processing Information", *Psychol Rev* **63**() (Mar 1956): 81–97.
[48] Milner, A.J.R.G., "Using Algebra for Concurrency", in: *Distributed Computing*, F.B. Chambers, D.A. Duce and G.P. Jones (eds), Academic Pr, London, 1984, pp 291–305.
[49] Naur, P., Randell, B. (eds), *Software Engineering.* (Rep Conf sponsored by NATO Science Committee, Garmisch, Oct 7–11, 1968), NATO Scientific Affairs Div, Brussel, Jan 1969, 231 pp.
[50] Neumann, P.G. (ed), "Risks of Using Computers", *Comm ACM* **36**(5) (May 1993): 114.
[51] Nijholt, A., *Computers and Languages — Theory and Practice.* Studies in Comp Sci and Artificial Intelligence, Vol 4, North-Holland, Amsterdam, 1988.
[52] Nijssen, G.M., Halpin, T.A., *Conceptual Schema and Relational Database Design — A Fact-oriented Approach.* Prentice Hall, Englewood Cliffs NJ, 1989.
[53] Ollongren, A., Van den Herik, H.J., *Filosofie van de Informatica.* Ser Wetenschapsfilosofie, Martinus Nijhoff, Leiden, 1989 (in Dutch).
[54] Parnas, D.L., "On the Criteria to be Used in Decomposing Systems", *Comm ACM* **15**(12) (Dec 1972): 1053–1058.
[55] Parnas, D.L., "Designing Software for Ease of Extension and Contraction", *IEEE Tr Softw Eng* **SE-5**(2) (Mar 1979): 128–137.
[56] Parnas, D.L. *et al*, *On the Design and Development of Program Families*, Technische Hochschule Darmstadt, Fachbereich Informatik, Forschungsbericht BS I 75/2, 1975.
[57] Parnas, D.L., "Some Software Engineering Principles", in: *Structured Analysis and Design,* State of the Art Rep, Infotech Int'l, London, 1978, 237–247.
[58] Perlis, A.J., "The Keynote Speech", in [36], pp 1–6; (A slightly abridged version appears as dedication in [1]. This dedication starts with: "This book is dedicated, in respect and admiration, to the spirit that lives in the computer").
[59] Polya, G., *How To Solve It — A New Aspect of Mathemetical Method —.* Princeton Univ Pr,

Princeton NJ, 1945, (2^{nd} ed, 1957).

[60] Potter, B., Sinclair, J., Till, D., *An Introduction to Formal Specification and Z.* Prentice Hall, Englewood Cliffs NJ, 1991.

[61] Rechtin, E., "The Art of Systems Architecting", *IEEE Spectrum* **29**(10) (Oct 1992): 66–69.

[62] Sammet, J.E., "The Use of English as a Programming Language", *Comm ACM* **9**(3) (Mar 1966): 228, (incl discussion: 228–230).

[63] Schnitger, P.K.H., *Prototyping a Distributed Multi-user RDBMS in a Functional Language.* MSc Thesis, Dept Comp Sci, SETI Section, Univ Twente, Enschede, May 1991, 143 pp.

[64] Scollo, G., *On the Engineering of Logics.* PhD Thesis, Inf Dept, Univ Twente, Enschede, Mar 1993.

[65] SERC, *Het specificeren van software kwaliteit — een praktische handleiding*, Stichting SERC (QUINT project), Kluwer Bedrijfswetenschappen, Deventer, 1992 (in Dutch).

[66] Shaw, M., "Abstraction Techniques in Modern Programming Languages", *IEEE Softw* **1**(6) (Oct 1984): 10–26, (updated and revised version of "The Impact of Abstraction Concerns on Modern Programming Languages", *Proc IEEE* **68**(9) (Sep 1980): 1119–1130).

[67] Shaw, M., "Prospects for an Engineering Discipline of Software", *IEEE Softw* **7**(6) (Nov 1990): 15–24.

[68] Shepperd, M., Ince, D., *Derivation and Validation of Software Metrics.* Clarendon Pr, Oxford, 1993.

[69] Shlaer, S., Mellor, S., *Object-Oriented Systems Analysis: Modelling the World in Data.* Yourdon Pr / Prentice Hall, Englewood Cliffs NJ, 1988.

[70] Steutel, D., Van Zonneveld, E.P., *Prototyping a Relational DBMS in a Functional Language.* MSc Thesis, Dept Comp Sci, SETI Section, Univ Twente, Enschede, May 1988, 53 pp.

[71] Turner, D.A., "Functional Programs as Executable Specifications", *Phil Tr R Soc Lond* **A 312** (1984): 363–388.

[72] Turner, D.A., "Functional Programming and Communicating Processes", in: PARLE '87*, Parallel Architectures and Languages Europe.* J.W. de Bakker *et al* (eds), LNCS # 258, Jun 1987, Springer-Verlag, Berlin, 1987, pp 54–74.

[73] Turski, W.M., "Software Engineering — Some Principles and Problems", in: D. Gries (ed), *Programming Methodology; A Collection of Articles by Members of IFIP WG 2.3*, Springer-Verlag, New York, 1978, pp 29–36.

[74] Van Vliet, J.C., *Software Engineering.* Stenfert Kroese, Leiden, 1984, xiv + 218 pp (in Dutch; the touch of 'magic' is still present in later / other editions of this book, as is confirmed by the author in a personal communication on 14 Oct 1994).

[75] Vree, W.G., *Design Considerations for a Parallel Reduction Machine.* Univ Amsterdam, PhD Thesis, Amsterdam, Dec 1989, x + 195 pp.

[76] Waters, R.C., Chikofsky, E.J. (eds), "Reverse Engineering", *Comm ACM* **37**(5) (May 1994): 22–93.

[77] Wegner, P. (ed), *Research Directions in Software Technology.* MIT Pr, Cambridge MA, 1979.

[78] Wegner, P., "Introduction and Overview", in [77], pp 1–36.

[79] Van Wijngaarden, A. *et al* (eds), *Revised Report on the Algorithmic Language ALGOL 68.* Springer-Verlag, Berlin, 1976.

[80] Winograd, T., "Beyond Programming Languages", *Comm ACM* **22**(7) (Jul 1979): 391–401.

[81] Wirth, N., "Program Development by Stepwise Refinement", *Comm ACM* **14**(4) (Apr 1971): 221–227.

[82] Wordsworth, J.B., *Software Development with Z — A practical approach to formal methods in software engineering.* Addison–Wesley, Wokingham, 1992.

[83] Wulf, W.A., "Some Thoughts on the Next Generation of Programming Languages", in [36], pp 217–234.

[84] Yourdon, E., *Decline & Fall of the American Programmer.* Yourdon Pr, Yourdon Pr Computing Ser, Prentice Hall Bldng, Englewood Cliffs NJ, 1993.

Chapter 3

Solution in Context: Functional Programming

As, eventually, everything that has to be communicated must be formulated in a more or less formalised way, we are confronted with languages again. This Chapter treats functional programming languages in order to show their use in prototyping. Using these languages in prototyping brings the problem description closer to its solution, which means getting the solution faster to the user. The language of the second paragraph of the Introduction was used to give *orders* to the computer, and giving orders means doing it in an imperative way. When writing a dissertation on the use of another kind of programming language, one of the questions to be asked is: "Are there more languages of this kind around?" We will introduce a few different language paradigms, languages not imperative in nature: functional and logical. At the same point we also discuss the object-oriented paradigm. However, logical and object-oriented are only briefly dealt with.

Functional languages are the colours of the Functional Language Paradigm, a notion that will be elaborated in comparison with the other paradigms, in pointing at its roots and in its practical use. Based on the desired prototype language properties, raised and discussed in Chapter 2, we will show in this Chapter why the functional languages have been chosen, why they are indeed, 'strong prototyping languages'.

The use of functional languages in prototyping as a tool for building better solutions, leads to the main, practical, result of this dissertation in later Chapters. We will further illustrate in this Chapter that there is a need for expanding the functional languages, were it only to prove — again — that they can be used in providing solutions to real life problems. As already mentioned, the prototype must be executed in the context of the problem, so linking to other, maybe non-functional, systems implies a kind of embedding of the functionality of those systems through the new construction, discussed in the later Chapters.

Studying the nature and use of 'functional' programming languages, the first thing to rouse one's interest, is the notion of *function*, or function *application*, as these languages are also called 'applicative'. What are the basics of these languages? How does one use a functional language? What makes these languages serve their intended purposes? What makes these languages work, or, how does a functional program yield the desired result? It turns out that the answers to these questions are very strongly related.

In between the fundamentals of functional programming and the elaboration of functional programs, there should be the construction of programs in the functional language. In general, this

Chapter covers the ins and outs of functional languages, and to illustrate some points — when getting practical — we will use Twentel (see 2.2) as a demonstration vehicle.

3.1 Describing Solutions: Functional Language

In 1.2 and 2.3.2 we discussed the development of the traditional imperative languages based on the Turing-Von Neumann computational model, put into reality by the electronic stored-program computer. It is generally accepted that the languages this model spawned are called first through third generation languages. The generation number signifying a rising abstraction level from the underlying hardware. If we should care to define the 'next' language, this 'fourth generation language' should then, by progression, leave the state transition model and should be founded upon another computational model.

As Alan Turing proved in 1937 that his Turing machine and the lambda calculus were equivalent [35],[1] a domain switch (cf Fourier and other integral transformations) in this direction offers probably a good course for further developments.

Two forms of calculus require attention: the first order predicate calculus, and the lambda calculus. These calculi became of practical interest as computational model, because programming languages have been designed around them. Another computational model in use today, is the object-oriented model. The first two models have languages that are accepted as being 'declarative languages', languages that 'declare or state, what has to be calculated'. These languages are more or less static, as these languages do not have a notion of a state that changes during time; the meaning of the expressions in the programs written in these languages does not change over time. The underlying model of functional languages is the function applied to its arguments, and the logical languages are based upon the relation between data.

Another way of looking at the Imperative - Declarative controversy with regard to the solution of the problem is the following. Programs in imperative or operational languages describe step by step how to construct something which is a solution: they describe the solution. Programs in declarative or definitional languages state properties of the desired solution (facts, rules, constraints, transformations, relations) to constrain the solution set, without describing how to compute it. Then the language system derives from these properties a scheme for calculating a solution: these programs confine the solution set.

Object-oriented languages are not readily classified. One finds it siding with the declarative languages [73], with the imperative languages [3, 54], and as belonging to a unifying paradigm [42]. We tend toward incorporating them with the imperative languages, as will be discussed further on.

The imperative languages reigned supreme until the late eighties. They 'matched' the hardware, so they had no competitors in efficiency. Programming languages like Lisp [46], with its pure

[1] A.M. Turing, "Computability and $\lambda-$definability", *J Symbolic Logic*, **2** (1937): 153–163. In this equivalence one finds the answer to a puzzling question: though (one of) the main basic paradigms of computing science is the Turing Machine [52] which is almost only states and I/O, the functional languages are proud of avoiding just these states as they are based on the lambda calculus. There is no difference, but a different point of view, between between algebra or calculus and the Turing Machine.

functional kernel, hid their true nature because of the tremendous push resulting from the software backlog, their environment, and their reputation; no time to ponder for the programmer, as Robert Floyd said: "My message to the serious programmer is: 'Spend a part of your working day examining and refining your own methods. Even though programmers are always struggling to meet some future or past deadline, methodological abstraction is a wise long-term investment.' " [23]. LISP was constructed, and later on expanded, with non-functional properties, thus enabling it to become even more run-time efficient than FORTRAN [14, 35].

The fault of the software backlog is not only with the imperative language. Also, though a little overgeneralising, programmers tend to be very action oriented. Two reasons spring to mind: because of their intimate involvement — obsession — with their products ("Just one more feature to put in; Just one more bug to get rid of."), and because of their managers, who understand the professional production of software by the programmer to result in a set of statements, not in devising a solution. Acquiring a new mindset, a '*Gestalt*-switch' from giving orders to envisaging solutions, is a difficult operation in this setting. Nevertheless it is necessary to do so in order to come to grips with the increasing complexity of the problems to come.

3.1.1 Language Paradigms

Up till now we have used in this dissertation the 'paradigm' notion in an intuitive, conglomerate sense, a sense containing 'example', 'model', and 'pattern'. There is, however, a deeper meaning used in the philosophy of science. We used it in the latter sense in 1.4, where we mentioned the initial seclusion of the functional programming community.

Thomas Kuhn, interested in the processes active in the workings and progress of (natural) sciences, constructed a theory of how science 'operates', a special branch of sociology. In this theory he starts with defining 'normal science' as "research firmly based upon one or more past scientific achievements, achievements that some scientific community acknowledges for a time as supplying the foundation for its further practice" [44]. Next, he defines *paradigm* as such an achievement which must have two essential characteristics: "being sufficiently unprecedented to attract an enduring group of adherents away from competing modes of scientific activity", and "being sufficiently open-ended to leave all sorts of problems for the redefined group of practitioners to resolve". On the 'process' of science he notes: "Men whose research is based on shared paradigms are committed to the same rules and standards for scientific practice. That commitment and the apparent consensus it produces are prerequisites for normal science, i.e., for the genesis and continuation of a particular research tradition." He gives also a less operational definition of 'paradigm' in the following: "A model from which springs a particular coherent tradition of scientific research based on accepted examples of actual scientific practice, being laws, theory, application and instrumentation."

Time prohibits delving deeper in this theory, so we conclude that the Functional Programming Paradigm is a paradigm in the above sense, created around the colours of John Backus' Turing Lecture in 1977 [7], and, since Backus was rather theoretical, David Turner's practical *Software Practice & Experience* article from 1979 [67]. In literature one can find references to functional program(s/ming) earlier than these papers, but these earlier papers did not succeed in attracting a group of scientists. In Turner's article, theory, albeit slightly different from Backus' one, was given a practical foundation for a computational model by introducing the generally almost

unknown combinators.[2] Hudak notes with respect to the 'particular coherent tradition': "Unlike many developments in computer science, functional languages have maintained the principles on which they were founded to surprising degree. Rather than changing or compromising those ideas modern functional languages are best classified as embellishments of a certain set of ideals. It is a distinguishing feature of modern functional languages that they have so effectively held on to pure mathematical principles in a way shared by few other languages" [35]. We think this coherence is brought about by the very substantial number of purists in the functional languages community, "who believe that purely functional languages are not only sufficient for general computing needs but are also better because of their 'purity' " [35]. (Pure) LISP's earlier existence as a functional language was only generally observed after the genesis of the Functional Programming Paradigm.[3] However, one might call LISP a paradigm in its own right, as it does have its own scientific community.

3.1.2 Language Paradigms Based on Logic and Objects

Before entering the main subject of this Chapter, we first sketch logic and object-oriented languages. Being a paradigm holds for these kinds of languages too. Logic programming started rather quietly with Prolog in 1972 in Marseilles (and Edinburgh), its seed sown in 1965: [61]. Where the other paradigms know a respectable number of languages sprouting from the research, logic programming started with, and stays by and large with Prolog. In this sense, Prolog may be viewed as the banner itself. Object-oriented had Smalltalk in its initial banner (with Simula67 as its 'LISP'), the true colours being objects, things that exist in the outside world and interact with each other. This idea is not language bound, so many languages could implement this idea. Now, perhaps, C++ is written on the banner. But the object-oriented colours have also been taken over by industry, obscuring the legitimate research.

Logic Languages The other main stream of declarative languages is formed by the logic (or relational) languages, with the first order predicate calculus as their computational model. The computational process is called resolution, or inference [61]. Its main, uniform, datastructure is the term, based on the Horn clause. From the term, the Prolog program, as well as its data, are constructed. A Prolog program is a set of clauses, where each clause is either a fact with its relationship about the given problem, or a rule how the solution to the problem may relate to (or be inferred from) the given facts. Facts are always true. Rules become true if certain, related subgoals are true. Programming in Prolog amounts to declaring some facts, defining rules, and asking questions, all about objects and their relationships. The evaluation of a Prolog program starts with the question asked, the goal. Then, through trying to satisfy the goal (prove the goal to be true) and all its related sub-goals by using the given facts and rules through a mechanism called backward chaining. The system yields (a set of) facts and relationships which satisfy the goal, and displays it: the solution. The solution set is constructed by backtracking on a partial solution: can more facts be found that satisfy the goal?

[2] However, note the appearance of Van der Poel's [59] in 1973: as usual taking the lead with very interesting subjects.

[3] Already in 1973 LISP was considered a very special language, as Jean Sammet remarked on it: "Programming languages can be divided into two categories. In one category there is Lisp; in the second category all the other programming languages." With the advent of functional languages, LISP is not alone any more in its category.

Prolog started the Logic Language Paradigm, and it stayed its main language. Though its computational model is also a powerful one, a Prolog program is through its language implementation less amenable to mathematical treatment, as the referential transparency property does not hold in general (3.1.3.2). This was caused by the introduction of the 'cut', necessitated for efficiency reasons [17], the same reasons that were used for LISP when incorporating imperative features into it. Another problem caused by the complicated evaluation mechanism is that left recursion is unreliable. Nevertheless, in our opinion logic languages offer the same prospects as functional languages, only, as their community is smaller, progress cannot be measured on the same terms.

Object-Oriented Languages A nice touch of Clocksin and Mellish in the last quoted text [17] is the opening sentence, where they remark on Prolog: "*Prolog* is a computer programming language that is used for solving problems that involve *objects* and the *relationships* between objects." The object-oriented paradigm is based on three essential ingredients: a) *objects* (the basic run-time entities, that respond to messages and have a state), b) encapsulation (the 'abstract data type', complete with access procedures, activated through messages) which is embodied in the *class* notion (a sort of type, of which objects are an instance; all objects of a certain class display a common behaviour), and c) *inheritance* of properties (behaviour) of a certain class by another class, that is sub-ordinate to it (a sub-type in a hierarchical ordering).

Programming in object-oriented languages must be regarded as the interplay of objects, every object representing an object in the outside world or an abstract object created in modelling the problem to be solved. Objects are instances of a certain class. The interplay is implemented by objects sending messages to other objects (function calls) upon which these are requested to act. Within the objects, the functional behaviour is represented by 'methods', the imperative procedure's counterpart. The main structuring effort in object-oriented programming lies in the definition of the class tree: which classes does one need, with which properties, and how must these properties be distributed over the classes.

The object-oriented approach as it exists, is still a state transition computational model. Inside the objects the way of programming the processes is still done in an imperative way. This is in our opinion not the right way of describing the problem solution as has been made plausible in the previous Chapter. The paradigm as such — especially the object notion which is an abstract data type with all its advantages, and which cannot be compromised, and the creation of new objects with the application of inheritance — definitely merits much research. However, objects too, should be liberated from the Von Neumann bottleneck in some way. One cannot deny the mathematical soundness of the functional languages as we will see, and as it proves that mathematical soundness is achievable in a practical way, one should use such a method.[4]

The object-oriented approach promised a few things — just like the structured programming approach did a decade or more before. The latter approach — intrinsically still very valid — is paid lip service as it presupposes higher abstraction capabilities or more mathematical inclination then usually found in a programming shop. The former's main promises are higher programmer output through facilitating reuse of objects, and, having in the use of 'objects' a

[4] This adapted approach must be given an altogether different name: nobody will work with functional object-oriented languages, nor say he is engaged in object-oriented functions; and function-oriented objects are also barred.

natural way of describing problems. The last idea relates to problem solutions as being objects passing messages to each other, and using 'objects' is said to be a more natural way of describing problems — more down-to-earth, as objects represent real world things — than the higher data and function abstraction facilities which are offered by declarative languages. We have seen that 'general problem solvers' are not the ultimate solution.

The real problem in reuse of objects, lies in the construction of the class-tree. Reuse of objects, and the subsequent addition of a new feature to a class, is often said to be a mere adding of a new method to the class. But adding to a substructure without proper attention to the overall structure (the class tree) will eventually lead to maintainability problems of the whole. In constructing this tree one must have the Delphinian Oracle view on the future: the structure of the data and its access procedures is not to be solved by using objects as the magic dividing rule of the problem. Changing something in the class-tree, is like changing the structure of a pyramid.

Many of the advantages of data structuring, which originated in the imperative languages and are present in the object-oriented languages — notably the abstract data type — are also implemented in modern functional languages, with an even stronger foundation (see 3.1.7). So the separation of architecture from implementation and realisation can also be done in these languages. If full 'inheritance' with super- and sub-typing is implemented in functional languages (type is given 'first-class citizenship') the last advantage of object-oriented languages (inheritance) is assimilated. In this respect one should try to distinguish the 'genes' of a class, a 'gene' being an inheritable property, and in inheriting 'genes' one does not automatically inherit all the properties of a certain class (which is 'natural' with inheritance). Checking on 'genes' must in some cases be done dynamically, it cannot be done statically in all cases.

3.1.3 The Functional Programming Paradigm

Having given a meaning to the notion of 'paradigm', we arrive at an important moment in the history of the functional languages, their genesis, which is the right place to give a short historical overview. Paul Hudak [35] gives an interesting and detailed overview of the various sources of the functional languages. In this survey he discusses the lambda calculus, LISP, Peter Landin's work in the sixties (ALGOL60 and the lambda calculus, the SECD machine, and the ISWIM ('If you See What I Mean') language), and Kenneth Iverson's APL. Subsequently he treats the evolution of the functional languages, starting with John Backus' FP, ML and the Hindley-Milner polymorphic type system, David Turner's work on SASL, KRC and Marinda, dataflow languages, and, finally, Haskell.

In our paradigm based view of the programming language history functional languages enter the fray in 1977. And right in the first two years, already two styles in functional programming were present.

3.1.3.1 Two Styles of Functional Programming

One can distinguish two styles of programming in functional languages:

- the strict style of Backus, based on pure function composition, as used in [7]. In this style one programs with functionals, or function 'combining forms'. Backus' main idea is that programs are created by subsequently converting input data through a function into output data to the next function, until the required output data is obtained. These intermediate functions are combined into the ultimate function: the program, through the use of combining forms. There are five main combining forms:

  Composition : $f \circ g$. First apply g to the argument, then f to this result;

  Construction : $[f_1, f_2, \ldots, f_n]$. A sequence of n elements, obtained by subsequently applying f_i to the argument;

  Conditional : $p \rightarrow f; g$. If p is true, then apply f to the argument, else apply g;

  Apply to All : αf. This form yields a sequence of the same length as its argument, with elements obtained by applying f to each element of the argument;

  Insert : $/f$. 'Insert' yields the value obtained by applying f to the first element of the argument and $/f$ applied to the rest of the argument (in other words, inserting f between all the elements).

  This style tends toward the APL-like conglomeration of operators and functions, and is in our opinion not conducive to readable programs; one might call this style 'algebraic'. An example of this style,

  `IP = (/ +)` $\circ$ `(` α `*)` $\circ$ `Trans.`

- the predominant style, 'first' used by Turner in SASL [67]: using declarative, or equational, constructs to describe the needed computations.

  An example of this style (in Twentel),

```
rotateleft n list
    = CASE list IN
              [] -> [],
         (x:xs) -> IF n = 0 -> list,
                            -> rotateleft (n-1) (xs ++ [x]) FI
      ESAC.
```

  On this equational style, Turner makes the following observations [68]:

  – "recursion equations are a convenient notation to both data types and functions over those data types;

  – the equations of the program can be read mathematically as premises from which to deduce other properties of the functions involved;

  – the equations of the program can be used to compute values of the functions by treating them as left-to-right replacement rules (so that computation here is a special case of deduction.) It should be noted that in general more than one replacement can be made at a time, so there is significant opportunity to exploit concurrency in the implementation of a functional language. A basic result of mathematical logic, Kleene's (1952) first recursion theorem, tells us that whenever a system of recursion equations has a unique function for its solution, the function computed by treating the equations as left-to-right replacement rules is the *same* one. This result should be considered as the logical foundation of functional programming."

We confine ourselves to the equational style of functional programming as that one is generally considered more mainstream (e.g., SASL, KRC, Hope, Twentel, ML, Marinda, Gofer, Haskell) than the Backus' style. In our opinion the prevalent use of recursion equations over function composition, is due to the fact that in Backus' style programs the programmer does not have an easy way of recognising (and so comprehending) intermediate results. It is even deemed wrong by Backus, as the freedom of defining one's own 'functionals' (higher order functions, or, combining forms) leads to chaos, and accustomed to this unrestricted freedom one cannot become familiar with the useful properties of the few combining forms that are adequate for all purposes. An unconvincing educational argument, which does not display 'generality' (2.4.4). In programs written in an equational style these intermediate results, being quantities in the model of the problem, can be given meaningful names, thus enhancing comprehensibility. Also, as the distance between model and its description in an equational style language is not too great, it increases its usability. Furthermore, the few syntactic elements to be combined *ad libitum*, not too much parentheses, make a readable text. In this we see programmer friendliness as the main factor of preferring equational style over the strict algebraic style.

It is possible to employ the Backus' style in an equational style language. However, the other way round is cut off by the fact that the handling of equations — viewing them as replacement rules — involves 'pattern matching' on the equations to be rewritten. In the equational example from above, we see the form of the `list` argument for `rotateleft` specified in two different forms: an empty list; or a list which should be taken as an element `x` in front of the rest of the list `xs`. The interpreter / compiler determines at runtime which of the two 'patterns' must be used in evaluating the actual `rotateleft` application. This main aspect of the equational style languages is missing from the Backus' style languages.

3.1.3.2 Aspects of Functional Languages

Functional languages are based upon the lambda calculus (3.2.3). This strong mathematical basis makes it possible to reason about them, because of the *referential transparency* property of the lambda calculus.

Functional programming is expression oriented, it has the advantages of simple algebraic expressions. One is only interested in the meaning, the value of an expression. In 2.4.2 Robin Milner elucidated the use of algebra in structuring. Here we see the foundation for this view. In algebra an expression can be understood by understanding its constituents, its subexpressions, and the meaning, the value, of these subexpressions is *independent* of their context. This algebraic property is called *referential transparency* [74].

We need names, words, to talk about things. To be sure that communication about a thing, represented by a name, always concerns the same thing, it is important in a conversation not to change the definition of the word or name in the meantime, which is a common mathematical practice regarding (mathematical) variables. This is another way of defining 'referential transparency'.

Program variables are very different. In imperative programs, the variables are memory locations with a name attached. Referring to the name yields the value stored in the memory location, which value can be changed by storing another value into the memory location through an

assignment operation; referring anew to the same name yields now another value: a 'referentially opaque' situation [60].

This implies the impossibility of describing state spaces and their changes in functional languages — in imperative languages brought about by assignments — by which no side effects are possible. No surprise effects because of unsuspected changes in program variables deep down the imperative program one writes, or in the modules from others one uses.

There is 'glory' for the functional languages. Making a good design implies, a.o., obeying the transparency design decision rule (2.4.4). Blaauw and Brooks advocate policing certain design decisions [10]. No matter how heavy the policing, short of banning the assignment, it will not guarantee that undesirable side-effects are absent from an imperative language program. So one has to use functional programming languages as in these languages it is be proven that no side-effects are possible.[5]

In literature one finds more aspects which make functional programming languages appealing, all of them inherent to the above mentioned mathematical nature of these languages:

- "Functional programs are clearer expressions of their purpose than conventional programs, and, as such, are easier to maintain, easier to understand and easier to be built correctly in the first place. Programs in this style can be an order of magnitude shorter than programs to perform the same task in conventional languages" [68]. Another observation falls partly in this category: "The simple elegant languages [...] conduce to more orderly, more rigourous, more verifiable and, ultimately more efficient programming" [26];

- functional programs are amenable to formal analysis and manipulation;

- as there is no time dependency in functional languages, parts of the functional program can be evaluated in any order which brings us to parallelisation, made possible through the referential transparency property. So functional programs are naturally amenable to implementation on a parallel machine. However, Vree makes some reservations about 'direct' execution of these programs on a parallel architecture. Before parallel evaluation can be feasible, the program has to be transformed, which is made possible through the mathematical properties of the language [70].

- higher order functions are possible: functions can be used without restrictions as argument, as well as function value;

[5] At this point we could not resist the temptation to use the following quote from [15]:

"[...] There's glory for you!" [Humpty Dumpty said.]
"I don't know what you mean by 'glory'," Alice said.
Humpty Dumpty smiled contemptuously. "Of course you don't — till I tell you. I meant 'there's a nice knock-down argument for you!' "
"But 'glory' doesn't mean 'a nice knock-down argument'," Alice objected.
"When *I* use a word," Humpty Dumpty said in rather a scornful tone, "it means just what I choose it to mean — neither more nor less."
"The question is," said Alice, "whether you *can* make words mean so many different things."
"The question is," said Humpty Dumpty, "which is to be master — that's all."

- through the absence of the Von Neumann bottleneck and the program variables the programmer is relieved of housekeeping details, and the many burdens (the tedium of programming) concerning the organising or comprehending the sequencing of events or statements.

Another main aspect of the equational style is the possibility of 'lazy evaluation'. This property does not trace back to the lambda calculus. With the notion 'lazy evaluation' one defines the process of only computing a value when it is needed in other computations, postponing the work until it becomes inevitable. With this in mind, it is easy to describe an infinite list, or other data structure, and to manipulate it: if infinity is not asked for in the computation, it is not reached, thus it cannot exhaust the resources. It gives the programmer the opportunity to describe a specific data structure without worrying about how it gets to be evaluated. The notion was proposed by Vuillemin in 1974 [71], and was first used in 1976 by Henderson and Morris, and independently, by Friedman and Wise [25, 30]. Backus' style languages do not possess this kind of property (nor do all equational style languages, e.g., ML). If the infinite structure is a possibly infinite (so maybe, finite) list of input or output characters, the structure is called a 'stream'.

John Hughes' position is that lazy evaluation is the most powerful tool for modularisation of the functional programmer. Though in its exposition he enters the domain of implementation, it is indeed a practical problem, solved by the lazy evaluation property. We quote: "If f and g are functional programs, then $(g \cdot f)$ is a program that computes g $(f\ input)$. The program f computes its output, which is used as the input to program g. This might be implemented by storing the output from f in a temporary file. The problem with this is that the temporary file might occupy so much memory that it is impractical to glue the programs together in this way. Functional languages provide a solution to this problem. The two programs f and g are run together in strict synchronisation. Program f is started only when g tries to read some input, and runs only for long enough to deliver the output g is trying to read. Then f is suspended and g is run until it tries to read another input. As an added bonus, if g terminates without reading all of f's output, then f is aborted. Program f can even be a non-terminating program, producing an infinite amount of output, since it will be terminated forcibly as soon as g is finished. This allows termination conditions to be separated from loop bodies — a powerful modularisation. Since this method of evaluation runs f as little as possible, it is called 'lazy evaluation'. It makes it practical to modularise a program as a generator that constructs a large number of possible answers, and a selector that chooses the appropriate one." [37].

We should not proceed without mentioning some important weak points of functional programming languages:

- run time efficiency not excessive; however, Willem Vree mentions some positive results in this respect, so this weak point will turn out to be a myth [70]. Apart from this development, note that in the long run, programmers are more expensive than hardware;[6]

- dataspaces not large (though, with cdr-coding, and other bit-nibbling techniques, something can be gained);

- demands on graph space not predictable (might even be exponential);

[6] Arie Duijvestijn already took this position in the late sixties in his lectures at the University of Twente on Numerical Mathematics and Programming Methods: "The time total to solve the problem by a human being is of the essence, not the time fractional used by the machine." [48].

- there exists a steep initial learning curve for (imperative) programmers, which is *not* the case with novice programmers; education is necessary if one has not been exposed to the set of ideas of declarative programming as opposed to the ideas in imperative programming [8];
- opponents have it that one loses expressivity, as a result of the expression-only model (e.g., no clocks, semaphores, device drivers under program control) [26].

We will not pursue these points in the sequel, but for a partial solution to the first one. The solution for the first three points is purely a matter of implementation of functional languages — outside the scope of this dissertation —, and for the last two points it is a matter of education and perhaps language design — likewise outside our scope.

3.1.4 Why Using a Functional Language

In Chapter 2 we mentioned in the course of discussion the following requirements for a prototyping language (2.3.2 and 2.5):

- the language should be close to the problem domain, and be more in character with the problem described;
- the language should be amenable to formal (mathematical) notation;
- the language should help to reduce complexity in the program text;
- the language should strengthen the modularisation process;
- the language should facilitate fast delivery of prototype versions, thus ensuring the involvement of the user in the problem solving cycle;
- the language should avoid the Von Neumann bottleneck, thus freeing the programmer from the 'tedium' of programming;
- the language should have a computational model based upon a strong mathematical system, thus facilitating the correctness proofs of the program;
- the language should facilitate easy communication with existing systems and solutions.

After the previous Subsection it is clear why the functional programming languages fulfill these requirements, but for the last one.

With functional languages we have recursion equations as a convenient notation to both data types and functions over those data types, so with properly chosen data structures we can stay close to the problem domain.

The recursion equations and list comprehensions offer a good mathematical description, which description is directly executable.

The reduction of complexity is brought about by the fact that the program texts are usually an order of magnitude shorter than imperative ones, and that the number of primitive notions is very low, thus yielding simple semantics.

Fast delivery of prototype versions can be obtained as functional programs are easier to maintain, less complex, closer to the problem domain, and easier to understand than their conventional counterparts. So they are more flexible (malleable, changeable), and as functional programs are a kind of executable specifications, by being easier to maintain, they facilitate faster delivery of the next prototype.

The powerful lambda calculus upon which the functional languages are based, does not only facilitate correctness proofs, also all kinds of automatic transformations with meaning and correctness preserved, are possible. So a clear, understandable, inefficient solution can be transformed to an efficient one [19]

In this summing-up we can come back to the position of John Hughes, as it elucidates some of the above points. He holds that the key to functional programming power lies in the improved modularisation of the problem, which functional languages facilitate through their higher-order functions and lazy evaluation. "The ways in which one can divide up the original problem depend directly on the ways in which one can glue solutions together. Therefore, to increase one's ability to modularise a problem conceptually, one must provide new kinds of glue in the programming language." [37].

Here a few relevant remarks should be made about the application of a programming language to an area it was not supposed to cover well:

- Blaauw, Amdahl and Brooks, did very instructive work of creating a model of hardware (its architecture) in APL. They had a testable 'product', a product in which it was easy to decide on different design possibilities, before any 'soldering iron' could be seen [2];
- Boute used a functional language to model digital telephone exchanges [12, 13];
- Joosten's research shows the following phenomenon. Suppose the language closest to the problem domain has a paradigm different from the paradigm of the functional language in which the solution for the problem has to be given. In other words, the functional language is apparently unsuitable. Nevertheless, a description in the functional language is still possible, and, perhaps more important, natural. Joosten demonstrated this in particular in digital circuit design, which has a finite state machine as the observation paradigm (state space with changes easily described in an imperative language) [39].

  Peyton Jones corroborates the last application when he states: "However, it turns out that [using a functional language for implementing a finite state machine] is even easier than in an imperative language; when one state wishes to make a transition to another, it just recursively invokes a function which implements the new state, which then completes the parse from that point," and he concludes with: "The (slightly non-obvious) conclusion seems to be that it is, if anything, easier to write a finite state machine in a functional than in an imperative style." [57].

Though these lessons are clear, 'it can be done', we should remain practical and not stay in the functional regime, '*because* it can be done.'

One of the intrinsic properties of a functional language is referential transparency. The dilemma is that in the changing, outside world there is no referential transparency. The outside world has never been static, it changes with time. Take a database, a kind of reflection of reality, linked

to a functional program: the database has to change, because reality changes. The functional program cannot forever yield 'Adam and Eve', when asked what persons are in the database. A referential transparent database is a nice academic exercise, but referential transparency in its strict mathematical form does not belong in the outside world, we should not enforce it; remember the reason why we rejected the hiaton solution (2.4.5) in the first place.

3.1.5 Extending the Language with Other Solutions

As there was, as yet, no solution to the last requirement of the previous Subsection, we propose to extend our prototyping language in two dimensions:

- with respect to computational efficiency: delegate certain computing tasks to other programs, more efficient in that particular task;

- with respect to programmer efficiency: delegate certain description tasks (eventually ending up as computing tasks) to another description regime, another language, as the sub-problem can be described much easier and faster in that language.

If we could extend a functional language this way, business applications too, mostly with large databases, could be easily prototyped with a functional language. And prototyping, which involves reuse of existing systems, can now be done. In this way we show the functional language to be a full grown language, which is applicable to a very wide range of applications, due to the fact that we are fully aware of its limitations which can be easily overcome by (re)using other solutions.

However, one should keep in mind in connecting to the outside world, that the basic mathematical properties of functional languages as based on the lambda calculus should be kept. These properties should not be corrupted by introducing (imperative) features for the sake of efficiency: efficiency was sub-ordinate to consistency and correctness (2.4.5). The functional paradigm must remain master of the problem solution.

The Functional Programming paradigm can now be applied in real practice, thus solidifying its *raison d'être*. It should not succumb to the PL/I syndrome, not inventing yet another solution to a sub-problem, but keep the number of independent notions as small as possible, thus following William of Ockham. We cannot create the General Language, so delegate. What has to be demonstrated is the ease of communicating with the outside world solutions.

The danger of not delegating exists, as functional programming is the progeny of formalists. "A major challenge lies in accommodating the large number of 'built-ins' needed in a pragmatically useful language, without succumbing to turgidity, or yielding to referential opaqueness." [45]. We contend with Dave Harrison, who advocates that 'fully functional' is not a panacea, that e.g., "real inter-process communication should be done in CSP or a similar language, but the processes themselves should be written in a functional programming language, if that is the language most suited for the problem" [29]. So we do link to the outside world, to other description regimes, if the problem solution requires it, but on the inter-process communication we have a — temporary — diverging opinion, as is discussed in the next Subsection.

The practice of the Functional Paradigm already shows tremendous progress. Progress: from the theoretical eight-queens problem to prototyping in different problem areas [39], one progressed to ray-tracing, spreadsheets[7] and a North Sea tidal prediction model [70]. But functional programming should also prove itself in the administrative practice, with lots of heterogeneous data, entered, sorted, searched and printed, every other day in a different format.

3.1.6 Cooperating with Other Solutions

We are concerned with different computing processes, processes that communicate with the functional program and we must connect these outside processes to the functional framework. The logical choice for describing communication between processes is to use the formalisms of Communicating Sequential Processes (CSP) of Tony Hoare [33], or Robin Milner's Calculus of Communicating Systems (CCS) [49]. Even Petri nets might have been used [56]. In this Subsection we look exclusively at these three basic formalisms, we do not consider e.g., the more modern approaches of Jan Bergstra, Jan Willem Klop and Jos Baeten on process algebras (ACP).

Hoare's CSP is an elaboration of Dijkstra's guarded commands, by extending it with other primitive commands: the concurrent execution, input and output, assignment, alternative and repetitive command. The resulting process description language is strictly imperative in nature, but for one interesting feature: parallel executing processes may not communicate with each other by updating global variables, thus facilitating correctness proofs as the state space is limited.

CSP has had much influence on the evolution of 'languages' for describing parallel processes, more than the Petri nets had. One of its offspring is David May's Occam, a language developed for the INMOS Transputer. It describes concurrent processes communicating via one-way channels. There are four basic processes: input, output, assignment, and wait. More complex processes are constructed through a sequential execution operator, a parallel execution operator, an operator which associates inputs to processes, and a conditional operator.

We did not pursue this line of research as it accented too much the state transition model (e.g., the assignment), and it takes I/O as a primitive of programming, which should be delegated to other layers in our opinion (4.4.3, 4.4.4).

Milner's CCS is a strong algebraic system, based on expressions, not on commands. His central idea(s) regarding a concurrent system are, observation of the behaviour of a system (the extensional view), and, as concurrent systems are built from independent communicating agents, synchronised communication between agents. This communication is an indivisible action of the composite system: the basic binary operation of the calculus is 'concurrent composition', composing two independent agents, allowing them to communicate. The terms of the calculus

[7] Respectively by P. Kelly, *Functional programming for loosely-coupled multiprocessors*, Ser Res Monogr Parallel Distrib Computing, Pitman / MIT Pr, Cambridge MA, Apr 1989; and by S.C. Wray, *Implementing and programming techniques for functional languages*, PhD Thesis, Tech Rep 92, Univ Cambridge, Cambridge, Jan 1986 [70].

(the structure of the system) are subject to equational laws, thus constituting an algebra. This algebra is used for describing concurrency between computing agents, with basic operations in connecting these agents: atomic action, summation, product, and, encapsulation. This route is much more promising than the previous one, as again, the mathematical properties of this approach are those of a proper algebra, and in our view, would fit better in the context of a functional language than CSP.

Petri nets. Last but not least, we have to mention Petri nets, as they were one of the sources of inspiration of Robin Milner [50], developed in the early sixties by the pioneer of the modelling of discrete concurrent systems, Carl-Adam Petri [56]. A *Petri net* is an abstract formal model of information flow in a system with asynchronous and concurrent operating parts [55]. The relationships between the parts are represented by a graph. This is a directed graph with two kinds of nodes: 'places' (modelling states of the system) and 'transitions' (modelling events or actions in the system). Two nodes of the same type are never connected to each other, only via a node of the other type: places are only connected to transitions, and *vice versa*. By using markers ('tokens') that travel through the graph, activated ('fired') by transition-nodes, that fire if all connected input place-nodes (pre-conditions) have tokens in them: the transition 'consumes' tokens. The firing of a transition causes token(s) to travel to all output place-nodes (post-conditions) of a transition: the transition 'produces' tokens. A complete system can now be described by properly designing the Petri net for it. Though Petri nets can be expressed algebraically, we have the feeling that the mere presence of a drawing which can be traced with a pencil, subjects them in our view to the same objections which befell the flowcharts — a subjective opinion, to say the least. We did not pursue this description mechanism either.

In our research we tried to keep our approach as uniform as possible, especially in the area of applying the rewriting paradigm, so abundantly present in the sphere around functional languages. We did not want to introduce another description regime, which is contrary to our own motivations behind extending the functional language itself. So we pursued 'it can be done', this time 'waiting in rewriting', knowing that it is far from the usual approach. It must be admitted that also time and the scope of the work undertaken, were of the essence in this decision.

3.1.7 Data Structures and Types

The problem of the ubiquitous data types remains: Twentel is non-typed, like Lisp and apl. This permits the use in this dissertation of the general list constructor from Lisp, which means that lists can contain elements of different type. The lists in most typed functional languages do not possess this property (easily), these lists are more like mathematical sequences, unless one has polymorphic typing (ml, Marinda).

Nevertheless we have to discuss 'type' as it is one of programming languages' central notions. Until 1978, *type* has been used as a vague, intuitive concept. "The use of type as a formal programming language term is perhaps due (more than to anything else) to the fact that the syntax of Algol60 was defined rigorously in a formal grammar. [The presence of] the rule

```
<type> ::= real | integer | Boolean
```

simply gave 'type' a more formal structure in programming languages than it previously had." [28]. Yet is was used in ALGOL60 in a non-technical sense — it was simply a word, roughly equivalent to 'kind', used to aid in distinguishing different classes of objects, as Alan Perlis recalled [28]. Here the basic, or atomic, data types were introduced.

After ALGOL60, the advent of programmer-defined types, initiated by Tony Hoare, necessitated the more precise definition of type. The 'type' notion was related to the set in a still imprecise way: "But note how almost all these [definitions] secure their position by using a word (basically, essentially, abstractly) to indicate that there may be more to a type than a set, but that the exact meaning of *type* is not of interest" [28].

Gradually, however, the notion of 'type' centered on the data structure, and on extending the error checking capabilities of compilers — the 'strong typing' notion of Pascal succeeds in limiting this kind of errors at the price of imposing a severe penalty of inflexibility on the programmer, and the language alike. However, types are still static collections of other types, basic and structured.

The next step is 'dynamically' constructing types from other types: the algebraic or concrete data type. Now one can define new data types together with programmer-defined constructors using 'type-' and 'data-constructors' and 'type-variables'. The resulting data structure and its constructors are, however, open to the programmer, which, as always, is a risk.

This evolution culminated in the concept of Abstract Data Types (or encapsulation: the data plus its access functions). This has proven itself a significant step forward in structuring algorithms and improving reliability and readability of programs. The hiding of 'how to do it' from 'what you want to do' or 'want to be done' is also an essential property of functional languages. And these Abstract Data Types are now also an asset of the modern functional languages. Sorting algorithms can be written down in a compact way, independent of the data type of the entities to be sorted, if the $=, \neq, >$ and $<$ relations are an integral part of the data type as an ordering relation. In this way a balance has been found between functions and data in the functional languages, which improved them both very much.

Structuring of program data is Good Programming Practice nowadays. The modern functional languages (e.g., Marinda, Haskell) have implemented data structuring (and function typing), which improved their strength. Function typing is the apogee of ALGOL's **integer procedure**; it is in typed functional languages with functions as 'first class citizens' necessary to state explicitly domain and range of a function (unless derivable); type is not the prerogative of data. Due to the fact that we have chosen Twentel — with no language based structuring capabilities other than the basic data types and lists thereof, and no typing of functions — as the vehicle for our research, this structuring does not play a role in our research. Nevertheless, with respect to structuring of data and typing of functions, increasing the readability and reliability of programs, finding type mismatches (an extra level of parentheses is easily introduced), structuring and typing is definitely not to be neglected.

Roland Backhouse comments on the tension that exists between the typed and the non-typed worlds of programming languages — "two quite distinct and antagonistic parts". In his view, these worlds "will never [...] be completely reconciled. Those of us who advocate typed languages are, in so doing, also advocating a discipline that ensures that the structure of our 'pyramids' is always evident. Discipline means constraint. But there will always be need for

'throw-away' programs, 'organisms' that are used, perhaps quite intensively but for a short period of time and then discarded. [...] the aim must be to bring the two sides closer and closer together. Such certainly is the aim of ML with its introduction of the notion of polymorphic type. Moreover, on the other side, no one would argue against the idea that a clearer structure would facilitate and not hinder the reuse of software." [6].

If we define the semantics of the 'type' notion to determine whether the program with its typing information stays within the purposes of the program, as stated by the programmer himself, it does not matter whether this type system enforced control is static (done at compile-time) or dynamic (done at run-time). We tend toward the hybrid solution of this problem in stating that as much typing information must be checked at the earliest moment possible through type inferencing. We must allow the programmer to supply as much information as is relevant about structures and types without impeding his (and the program's) flexibility, which means automatically deriving as much typing information about his constructions as possible. If not otherwise possible, type checking can take place at run-time. Remember that solutions must evolve, as Alan Perlis said: "It would be difficult to find two languages [Scheme (or LISP) and Pascal, HD] that are the communicating coin of two more different cultures than those gathered around these two languages. Pascal is for building pyramids — imposing, breathtaking, static structures built by armies pushing heavy blocks into place. LISP is for building organisms — imposing, breathtaking, dynamic structures built by squads fitting fluctuating myriads of simpler organisms into place. [...] The discretionary exportable functionality entrusted to the individual LISP programmer is more than an order of magnitude greater than that to be found within Pascal enterprises. [...] The list, LISP's native data structure, is largely responsible for such a growth of utility. [...] As a result the pyramid must stand unchanged for a millennium; the organism must evolve or perish." [1][8].

3.2 Functions

This Section deals briefly with the simple question 'What is a function?' — the corner stone of functional languages — in a historical and operational sense. The historical thread owes to A.P. Youschkevitch, whom we follow in those paragraphs [76]. As will be clear after reading this Section, the question cannot be answered in a theoretically sound manner, but in practice we can work with functions. Extracting 'variables' from an expression in order to use it as a function, can be done through the lambda calculus (3.2.3), and the almost equivalent combinatory logic (3.4.2). Then we treat recursion and the 'meaning' or value of functions. With this knowledge one can understand one of the ways one operates on the programs in a functional language in order to obtain executable programs that yield the desired value (3.4).

[8] This statement is corroborated by Edsger Dijkstra when he discussed the character of LISP: "... the most intelligent way to misuse a computer. I think that description a great compliment because it transmits the full flavour of liberation: it has assisted a number of our most gifted fellow humans in thinking previously impossible thoughts."

3.2.1 Definition of Function

In ancient mathematics the idea of functional dependency of a certain quantity on another quantity was not expressed explicitly; priests, surveyors, and fiscal authorities could do their work without the explicit notion of a function: they used graphs and tables. It was not an independent subject of research, although a wide range of specific functional dependencies existed. In rudimentary form the concept started to appear in works of Middle Age scholars. The rise of the natural sciences meant that geometric, analytic and kinematic ideas were used to specify a function, and gradually the notion of a function as a certain analytic expression, formula or equation, began to prevail. The inter-dependency of coordinates in a plane (points in a plane are always related to some curve) is still present in René Descartes' (1596–1650) *La Géométrie* (1637). But he uses equations expressing this dependency and so presented the analytical method of introducing functions. Independently of Descartes, Pierre de Fermat (1601–1665) expresses the same ideas. In a Latin commentary on *La Géométrie*[9] Frans van Schooten (16xx–16yy) — obtaining the first formulae for transformation of coordinates — had to use the notion of arbitrary points in a plane having no connection with some curve [76], but the independence was still not given a name.

One of the first mathematicians to develop it into an independent concept is found in the seventeenth century: Gottfried Wilhelm Leibniz (1646–1716). Youschkevitch remarks on his studies [76]:

> The word 'function' first appears in Leibniz' manuscripts of August 1673, and in particular in his manuscript entitled *The inverse methods of tangents, or about functions (Methodus tangentium inversa, se de functionibus).* [...] It should be remembered that the Latin verb { *fungor, functus sum, fungi* } means to perform, to fulfill (execute) an obligation [...]: "Leibniz gebraucht allerdings [...] noch nicht das Wort Funktion; aber wie der Anfang der Handschrift beweist, hat er den Funktionsbegriff schon im weitesten Sinne gebildet und benennt ihn mit dem Wort *relatio*. Auch [...] hat das Wort Funktion noch nicht ganz den heutigen mathematischen Sinn."

Youschkevitch further notes that Johann Bernoulli (1667–1748) first uses the word 'function' in an article appended to a letter, dated 5 July 1698. And in the eighteenth century we meet the first explicit definition of a function in precise form in an another article of Johann Bernoulli. This explicit definition of a function as an analytic expression appeared in 1718:[10]

> *Définition.* On appelle **fonction** d'une grandeur variable une quantité composée de quelque manière que ce soit de cette grandeur variable et de constantes.

In this article Bernoulli also proposed the Greek letter ϕ as a notation for a *caractéristique* of a function (the term is due to Leibniz), still writing the argument without brackets: ϕx. So the essential parentheses-free notation of the modern functional languages (apart from indicating

[9] *Geometria à Renato Des Cartes Anno 1637 Gallicè edita; nunc autem ... in linguam Latinam versa et commentariis illustrata, Opera atque studio Francisci à Schooten ... Lugduni Batavorum, 1649.*

[10] Joh. Bernoulli, *Remarques sur ce qu'on a donné jusqu'ici de solutions des problèmes sur les isopérimètres*, published in the *Mémoires de l'Académie royale des sciences à Paris*, 1718.

priority or association) can be traced to Bernoulli.[11]

Brackets, as well as the sign f for function are due to Leonhard Euler (1707–1783), a pupil of Bernoulli, who used them in his article E. 45, communicated in 1734 and published in 1740. He gave the first 'modern', set-theoretical, definition of function, using an idea which is not clearly present in the known methods of describing functions, the general notion of correspondence between pairs of elements, each belonging to its own set of values of variable quantities. In the preface to his *Institutiones calculi differentialis* (published in 1755) he writes [76]:

> If some quantities so depend on other quantities that if the latter are changed the former undergo change, then the former quantities are called functions of the latter. This denomination is of broadest nature and comprises every method by means of which one quantity could be determined by others. If, therefore, x denotes a variable quantity, then all quantities which depend upon x in any way or are determined by it are called functions of it. [translation from the original Latin by Youschkevitch, HD]

We skip a few ages in which functions are gradually better understood, but which still do not yield a strict definition, based on the fundamentals of mathematics: they all remain rather operational in character. We end this very brief history, with Youschkevitch [76]: "... modern mathematical logic discovered essential difficulties inherent in the universal, hence nonalgorithmic definition of a function. Even in 1927, Hermann Weyl (1885–1955) maintained, quite correctly, that:[12]

> Niemand kann erklären, was eine Funktion ist. Aber: Eine Funktion f ist gegeben, wenn auf irgendeine bestimmte gesetzmässige Weise jeder reelen Zahl a eine Zahl b zugeordnet ist [...] Man sagt dann, b sei der Wert der Funktion f für den Argumentwert a."

Alonzo Church (1903–) gives for our purposes an admirably loose intuitive definition of a *function* (operation or transformation) in [16]:

> a function is a rule of correspondence by which when anything is given (as argument) another thing (the value of the function for that argument) may be obtained.

Implicitly he states that a function can have at most one value for each (tuple of) argument(s). Such a correspondence rule does not have to be applicable to everything whatsoever: it has its set of arguments for which the operation makes sense, but there are no constraints on the elements of this set.

3.2.2 Working with Functions

In this practical way returning to everyday mathematics, we contend that a function is a mechanism that yields a value when given a certain value, and every time we give this certain value, the same value results (which is an operational definition of referential transparency). In other mathematical terms, it is a mapping of things, objects, taken out of a set of values, the source

[11] It is an old popular belief in medicine, that the patient's condition first has to deteriorate before he can be cured: LISP.

[12] *Philosophie der Mathematik und Naturwissenschaft*, München / Berlin, 1927.

of the objects, called the *domain*, to objects of another set, the target set, called the *range*, or *codomain*. Another way of obtaining a function value, is to draw it from a set of pairs of values, which set comprises the function *in extenso*. The domain of such a function consists of all first values of these pairs, the other values of the pairs comprising the range of the function. Giving the first value of a pair yields the second value of the pair: a table look-up. A function described by a mechanism that yields the value (a rule how to construct the function value from its arguments, or a mapping from domain to range) is called a function *in intenso*.

As is clear from Church's definition, nothing is specified about the domain (the given values) or the range (the resulting value) of the functions under consideration. So one of the elements of the domain or the range of the function f may be f itself. Usually in a set oriented environment, predefined domains (and ranges) upon which transformations take place, can — and so, do — not include these possibilities (think of Russell's Paradox). In Church's system the function is given first, and its domain and range are determined afterwards with no restrictions. Therefore the function can be included in its own domain, and is a candidate for self-application.

Apart from the normal functions (e.g., SIN, COS, LN), it is obvious that also the operators (e.g., $*$, $+$, $/$) can be considered as functions. So in determining the value of an expression it turns out there is only one underlying operation, the application of a function to its arguments: *functional application*. The 'application' is a kind of operator, that, when given a function f and an argument x, forms the expression $f(x)$ which stands for 'f applied to x'. It should be noted that a function only needs its input argument to deliver its output value.

In functional programming one does not use '$f(x)$', the ordinary (Euler) mathematical notation of the value of a function, or, the application of the function f to its argument x: it would cost too many parentheses. Here one denotes the operation of applying a function to its argument by simple juxtaposition: '(fx)', where outer parentheses may even be omitted, thus 'fx', herein following Johann Bernoulli (3.2.1). This basic operation (application) associates to the left, such that 'fgh' means '$(fg)h$' and not '$f(gh)$'. The latter meaning can be obtained by proper parenthesiation. So the above '$(f(x))(y)$' is written '$(fx)y$' or simply, 'fxy'. We follow this notation in this dissertation.

Now we have the 'White Knight' predicament,[13] how do you call, how do you invoke functions? This means naming conventions of the functions, how do we write down a function, how do we convert an expression into a function of its arguments? The process of obtaining a function from an expression is fundamental to the lambda calculus of the next Subsection. It is called

[13] Named after a discussion in [15], being reminiscent of the **'reference to'** discussions from the Revised Report [75], and the programming errors involving indirections:

> 'You are sad,' the [White] Knight said in an anxious tone: 'let me sing you a song to comfort you.' [...] 'The name of the song is called "*Haddocks' Eyes.*"'
>
> 'Oh, that's the name of the song, is it?' Alice said, trying to feel interested.
>
> 'No, you don't understand,' the Knight said, looking a little vexed. 'That's what the name is *called*. The name really *is* "*The Aged Aged Man.*"'
>
> 'Then I ought to have said "That's what the *song* is called"?' Alice corrected herself.
>
> 'No, you oughtn't: that's quite another thing! The *song* is called "*Ways and Means*": but that's only what it's *called*, you know!'
>
> 'Well, what *is* the song, then?' said Alice, who was by this time completely bewildered.
>
> 'I was coming to that,' the Knight said. 'The song really *is* "*A-sitting on a Gate*": and the tune's my own invention.'

functional abstraction. Now we have: abstraction as an inverse to application. Abstraction converts an expression into a function, whereas application converts a function with arguments into an expression.

3.2.3 Origin and Nature of the Lambda Calculus

Based upon everyday mathematical notions we introduce here in an explanatory way the *Lambda Calculus*. Theoretical thoroughness is far beyond the scope of this Subsection. Of course, 'Barendregt' [9] is the main source for those who want to Study the lambda calculus, otherwise the simple introduction by Hindley and Seldin will do [32]. Many authors of texts on (implementation of) functional languages include in their book a treatment of the lambda calculus, e.g., [20, 27, 35, 53]. We are indebted to all these authors as they provided us with theorems, proofs, and quotes.

Alonzo Church pursued in the early thirties a system of formal mathematics in which an operation of abstracting a function from its unspecified value, the lambda (or λ) operation[14] played the basic role. This operation '$\lambda\, x\,.\, M$' abstracts the function (with respect to variable x) from the expression, M, which follows the dot. From another point of view, the above form stands for the function with formal argument x and body M. If x appears in M, then x is called bound, any occurrence of another variable in M is termed a free variable [16]. Recalling the function definition of Church (3.2.1) it is not strange at all that in the lambda calculus functions can be applied to themselves: this gives the lambda calculus its power, as it is now possible to have recursion without an explicit recursive definition. This self-application does not make the lambda calculus inconsistent as a mathematical system — no paradoxes appear, like the Russell paradox did with set theory.

The lambda calculus concerns itself with unary functions. Frege anticipated in 1893 that one can concentrate on single argument functions while studying functions in general [24]. This unary function producing process was first used by Moses Schönfinkel, and is now called 'currying', after Haskell Curry, who used it extensively in his foundation of combinatory logic. Today one can refer to the related concept of 'partial evaluation'. If one writes a function to add two numbers, one tends to think that it is a function of two arguments. But what happens when the program gets elaborated on a computer to produce the sum $A + B$? First A is brought in from memory. Then we halt the computer. What rests then in the computer is a program — a state of the computer —, to be applied to the forthcoming B, which will then produce the sum of the already fixed A and the to-be-given B: this new function denoted '$\oplus A$', which is precisely Frege's intermediary function.

Actually the lambda calculus is a language, a set of lambda terms, or expressions. These terms are words over the alphabet consisting of variables (like, x, y, M, N), symbols ('(', ')'), an abstractor ('λ' with an optional scope delimiter '.'), the application operation (denoted by juxtaposition, cf Bernoulli), and, in our case, constants (like, $+, /, *$, SIN, 3.14, 7). In the pure (type free) lambda calculus one studies functions and their applicative behaviour, and to

[14] Why it is called the lambda operation, has been described by Rosser [63]: First the operation was depicted by a caret ($\hat{}$) over the variable concerned, (e.g., $\hat{x}$), which caret Church moved down to the front of the variable ($\wedge x$). Later, one added for typographical reasons an appendage (`) to this caret, yielding the form ($\grave{\lambda}$), resembling the Greek lambda (λ).

be more precise, as 'calculus' implies rules for converting expressions, one studies the set of lambda terms modulo convertibility. One is interested in the ultimate value of a lambda term. The lambda calculus gives us the possibility of 'computing with' functions, instead of merely 'computing of' functions. In dissecting a lambda term or expression into all its constituents, and rearranging them under meaning preserving rules, one can deduce other properties of the expression, simplifying it, or rearranging it for other purposes.

As already was hinted, we extended ('enriched') the original untyped lambda calculus with constants, embedded in the *delta*-reduction rules. The essential conversion rules of our extended lambda calculus are the following ones (with '$\Leftrightarrow$' denoting conversion: 'left to right' or 'right to left' rewrite; and '$\rightarrow$' denoting reduction: only 'left to right' rewrite):

β **conversion, or substitution :** The substitution operation, here denoted as a prefix operator, $[N/x]M$, which is M with N substituted for the free occurrences of x in M:

$$(\,\lambda\, x\,.\,M\,)N \Leftrightarrow [N/x]M$$

So we have *beta*-reduction in:

$$(\,\lambda\, x\,.\,*\,x\,x\;\,)\,3 \rightarrow *\,3\,3$$

This *beta*-reduction is of course, the application of a function '$\lambda\, x\,.\,M$' to its argument 'N'. The other way round it is *beta*-abstraction:

$$*\,4\,3 \rightarrow (\,\lambda\, x\,.\,*\,4\,x\,)\,3$$

α **conversion, or renaming :** The renaming of a formal argument in a lambda abstraction, without changing the meaning of the abstraction:

$$\lambda\, x\,.\,M \Leftrightarrow\; \lambda\, y\,.\,[y/x]M$$

This possibility of converting terms is necessary, because if the argument ('N') to a function ('$\lambda\, x\,.\,M$') contains occurrences of the formal argument, one must rename the formal argument, before applying the argument, in order to avoid name clashes.

η **conversion :** $(\,\lambda\, x\,.\,M\,x) \Leftrightarrow M$, provided that x does not occur free in M.

This conversion (used as a reduction) is used to get rid of a redundant lambda abstraction.

δ **reduction :** With this reduction one can reduce lambda terms, constant to the pure lambda calculus. The above constants ($+, /, *$, SIN, 3.14, 7) are introduced in our treatment in order to facilitate working with the lambda calculus as basis for implementing functional languages. Now it is possible to reduce '$+\,3\,4$', as there is now a rule '$+\,3\,4\;\rightarrow 7$'. In this manner all these (arithmetical) notions (addition, division, integers, reals) are contained in the lambda calculus, and can be used in the calculations.

The above difference between 'conversion' and 'reduction' is intentional: converting lambda expressions is essential in 'computing with' functions, reasoning about them, and establishing equality relations. Reducing lambda expressions is essential in 'computing' functions, in getting at their value, their meaning, which are naive terms for its normal form, a form of the lambda expression in which no further *beta, eta*, or *delta* reduction is possible.

3.2.4 Recursion and the Fixpoint Theorem

We first introduce the notion of the 'fixed point' (or, fixpoint) of a function. The fixpoint of a function f is that argument value x for which $fx = x$. A function can have more than one fixpoint. We use this notion in the definition of recursive functions in the lambda calculus. In the lambda calculus functions do not have a name, they are anonymous. So we cannot refer to a function by its name, but must use its definition whenever we use the function. Obviously, this would lead to an infinite regression in the case of recursive functions.

To resolve this problem, we demonstrate it on, e.g.,

$$\text{FAC} = \lambda\ n \cdot \text{IF}\ (=\ n\ 0\ )\ 1\ (*\ n\ (\ \text{FAC}\ (-\ n\ 1)))$$

We can use *beta*-abstraction (after a renaming) on FAC:

$$\text{FAC} = (\lambda\ f\ \cdot\ (\lambda\ n \cdot\ \text{IF}\ (=\ n\ 0\ )\ 1\ (*\ n\ (f(-\ n\ 1))))))\ \text{FAC}$$

which we can write as $\text{FAC} = F\ \text{FAC}$.

But this function F has now FAC as a fixpoint, as defined above, the recursive function we were interested in. So, if there is a function, e.g., Y, which takes a function as argument and yields that function's fixpoint as value, we have solved our problem. Then we can write:

$$\text{FAC} = YF\text{; with } F = (\lambda\ f\ \cdot\ (\lambda\ n \cdot\ \text{IF}\ (=\ n\ 0\ )\ 1\ (*\ n\ (f(-\ n\ 1))))))$$

which is indeed non-recursive in FAC in the body.

To demonstrate the existence of the function Y, we 'prove' the following theorem in the lambda calculus. In the following paragraphs, E, E_1, E_2 are lambda expressions.

Fixpoint Theorem Every lambda expression E has a fixpoint E', such that $(E\,E') \Leftrightarrow E'$.

Take E' to be $(Y\,E)$, with Y (happening to be the name of Curry's paradoxical combinator (3.4.4.1)) defined by

$$Y = \lambda\ f \cdot\ (\lambda\ x \cdot\ f\,(x\ x))(\lambda\ x \cdot\ f\,(x\ x))$$

Then, applying Y to the function whose fixpoint we want to compute, E,

$$\begin{aligned}&(Y\,E) = (\lambda\ f \cdot\ (\lambda\ x \cdot\ f\,(x\,x))(\lambda\ x \cdot\ f\,(x\,x))E\\ &= E\,((\lambda\ x \cdot\ E\,(x\,x))(\lambda\ x \cdot\ E\,(x\,x)))\\ &= E\,(Y\,E)\end{aligned}$$

□

John McCarthy used the lambda-construct to define anonymous functions in LISP [46]. But he did not understand enough of the lambda calculus to do more [47], e.g., he did not use the fixpoint function Y to represent recursion, but he invented the conditional expression to explicitly represent recursion, in which way it is intuitively more clearly represented [35].

3.2.5 Reduction to Normal Form

The canonical form of a lambda term is called *normal form*. The *normal form* of a lambda expression is a lambda expression to which no (further) *beta*-, or *eta*-reductions can be applied. A 'redex' is a reducible expression. As a lambda expression can contain more than one redex, the reduction of lambda expressions to their normal form can take many routes. However, the two *Church-Rosser theorems* prove that if a lambda expression has a normal form, this normal form is unique, and that there is a reduction sequence to reach it.

Church-Rosser Theorem I If $E_1 \Leftrightarrow E_2$, then there exists an E such that $E_1 \rightarrow E$ and $E_2 \rightarrow E$. □

This theorem has as corollary, that no lambda expression can be converted to two different normal forms. The intraconvertibility of E_1 and E_2 makes that there is a third expression, E, possibly equal to E_1 or E_2, to which both can be reduced (Rosser calls this the 'diamond property' [63], and in other formal systems a similar property is called 'confluence').

Church-Rosser Theorem II If $E_1 \rightarrow E_2$, and E_2 is in normal form, then there exists a normal order reduction sequence from E_1 to E_2. □

These two theorems form the basis for the referential transparency property, in itself a fundamental for possible parallelisation of a program based on the lambda calculus.

Reduction Orders The 'normal order reduction' is also one of the possible routes to reduce a lambda expression to normal form, but it is the one which positively gives the normal form of a lambda expression, if it exists. In normal order reduction, an expression is evaluated by first evaluating the leftmost outermost redex, which can be viewed as the function application itself. This reduction strategy will terminate if termination is possible; however, it may not do so in the least number of steps.

Another reduction method is the 'applicative order reduction': an expression is evaluated by first evaluating the leftmost innermost redex, which can be viewed as the argument of a function. This reduction will not terminate if one of the redexes of the expression is non-terminating, even if the expression does not need the redex to be evaluated (the function is non-strict in that argument).

We demonstrate the difference with regard to termination between these two reduction orders as follows, with the following lambda expression:

$$(\lambda x . y) ((\lambda x . xx) (\lambda x . xx))$$

Reducing this expression in applicative order yields:

$$(\lambda x . y) ((\lambda x . xx) (\lambda x . xx)) \rightarrow$$
$$(\lambda x . y) ((\lambda x . xx) (\lambda x . xx)) \rightarrow \ldots$$

and in normal order:

$$(\ \lambda x\,.\ y\ )\ ((\ \lambda x\,.\ xx\ )\ (\ \lambda x\,.\ xx\ )) \rightarrow y.$$

As '$(\ \lambda x\,.\ xx\ )\ (\ \lambda x\,.\ xx\ )$' has no normal form, its reduction is non-terminating.

Normal order reduction can be implemented in a very inefficient way, as is shown here:

$$(\ \lambda x\,.\ +\,x\,x\ )\ (\text{FAC } 723) \rightarrow$$
$$(\ +\ (\text{FAC } 723)\ (\text{FAC } 723)) \rightarrow$$
$$(\ +\ (\ \mathcal{F}\ )\ (\text{FAC } 723)) \rightarrow\ \ldots$$

So (FAC 723) (say, $\mathcal{F}$) will be computed twice, whereas in applicative order it will only be computed once, a tremendous gain in computing time:

$$(\ \lambda x\,.\ +\,x\,x\ )\ (\text{FAC } 723) \rightarrow$$
$$(\ \lambda x\,.\ +\,x\,x\ )\ (\ \mathcal{F}\ ) \rightarrow$$
$$(\ +\ (\ \mathcal{F}\ )\ (\ \mathcal{F}\ )) \rightarrow\ \ldots$$

In 3.4.5 we return to this reduction order problem.

3.3 Programming in a Functional Language

In this section we will give the reader a Bird's eye view of what programming in a functional or applicative language is like. After all, we have to show something of the way programs can be constructed in functional programming languages. Readers who are familiar with functional programming can skip this Section. In giving the examples we use Twentel. Its defining report, [34], and 'Bird-and-Wadler', [11], yield some of the programming comments given with these functions. However, we take it that in many functional programming languages programs can be constructed in this way. We will use 'SIN' for the general (functional programming) functions, and '`sin`' for the Twentel function yielding the 'sinus' of an angle. We will give a short and informal introduction to some of the main functions and constructions to be found in a functional language. Some of these will then be shown at work in the subsequent examples, one from literature, and one from this dissertation. A more expounded treatment of the construction of a function in a functional language is given in 5.1.2.3.

3.3.1 Programming with the Ubiquitous Functions

Many of the following functions belong to every modern functional programming language. We do not treat here classic mathematical functions on integers and reals, like SIN, COS, DIV, MOD. We consider their semantics known from mathematics and the imperative languages. However, due to the currying, customary in functional programming (3.2.3), the process which yields one-argument functions, we have to take care of the following. As the operation of functional application associates to the left, the two argument function 'SUB X Y' must be read as '(SUB X) Y': the function 'SUBTRACT X FROM' applied to 'Y', yielding 'Y-X'. And *not* 'SUB (X Y)' which yields (if naturally defined) 'X-Y'.

As the basic composite data type of functional languages is the list, we discuss here mainly functions operating on list(s), as they are not generally found in imperative languages. We

consider, however, simple operations on lists, like CONCATENATE, APPEND, REVERSE, HEAD and TAIL, TAKE and DROP, and, INITIAL and LAST as self-explanatory.

We start by introducing the very powerful *list comprehension* notation for creating lists. This notation enables the programmer to describe the elements of his list through their properties and their constituents. It is not a basic construction of a functional language, as its effect can be defined by MAP, FILTER and CONCAT. The construct presumably was first used by Jack Schwarz in his SETL language. Turner uses the term ZF-expressions, after the Zermelo-Fränkel-Skolem axioms for set theory. The choice as to whether writing a program using these functions or using the list comprehension, is a matter of style. An advantage of the higher-order functions is that it is easier to state algebraic properties and use these properties in the manipulation of expressions. On the other hand, list comprehensions are clear and simple to understand, and there is no need to invent special names for the subsidiary functions which arises when MAP and FILTER are used.

The following example of the use of a list comprehension gives all the Pythagorean triples ([11]):

```
{ [a, b, c] | a <- [1 ..], b <- [a ..], c <- [b ..]; a^2 + b^2 = c^2 }
```

This form answers to the syntax '`{ body | qualifiers }`'. '`body`' being the description of an element of the set to be constructed (here, a Pythagorean triple), and '`qualifiers`' a list of 'generators' and / or 'filters'. A 'generator' (e.g., '`b <- [a ..]`') generates elements that are used in the construction of the elements of the set. A 'filter' (e.g., '`a^2 + b^2 = c^2`') restricts the use of the generated elements by the generators. Qualifiers come into play from left to right. This means, later generators vary more quickly than their predecessors, and, later generators can depend on variables introduced by earlier ones. Some other examples are the divisors of a positive integer, yielding a set, and the greatest common divisor of two positive integers, yielding one integer:

```
divisors n = { d | d <- [1..n]; n MOD d = 0 }

gcd a b = MAX { d | d <- divisors a; b MOD d = 0 }
```

The use of *higher-order functions* (functional, combining form (3.1.3.1) is demonstrated throughout this Subsection, especially with MAP, FOLD, and FILTER: these functions have a function as one of their arguments.

One of the often used techniques is the *function composition*, denoted by the *dot* operator. This is the composition of Backus' FP (3.1.3.1). Later we will return to it, when discussing the B combinator (3.4.4.1). It is mostly used when minimising the number of parentheses, increasing readability by not introducing distractors. So, if $f\ x = g\ (h\ (i\ x))$, we can write $f = g \cdot h \cdot i$.

A *continuation function* exemplifies another possible use of higher order functions. A 'continuation function' can be used when a function is ready with the first part of processing its argument (e.g., a string to be parsed). This point is mostly reached deep down in recursion, when many characters of the string have been consumed by other functions that, according to the work to be done, have been applied (recursively) to the string. The second part of the processing of the main function must start with the (now changed) argument.

However, there is no way of determining how to return to the proper place in the string we had

when we applied the function to its argument. We need that place in order to finish the work of the function on the rest of its argument. Knowing that would imply the knowledge of how the evaluation of all involved functions proceeds, one of the things we absolutely do not want to know.

The solution is a 'continuation function' to be passed down the recursion as an extra argument. When the 'bottom' of the recursion is reached, the 'continuation function' is applied to the rest of the string, left over at that point. This function continues the work to be done on the (rest of the) main argument, work (or, function) that was postponed, and that was passed down the recursion. With the dot operator we add more 'work to be done' to this continuation function when it is passed down as an argument. The accumulated work will be done in the right order, bottom up, as will be clear from an example, with `h:H` the main argument (suppose `H = (i:j:k:K)`), and `ff` the continuation function:

```
DEF
    fun  (h:H)       =   a   H  first,
     a   (i:I)  ff   =   b   I  (ff . second),
     b   (j:J)  ff   =   c   J  (ff . third),
     c   (k:K)  ff   =   ff  K
FED
```

'Unwinding' from `c` yields: `first.second.third K`, or `first(second(third K)))`.

Local definitions, or definitions with a bounded scope, can also be given; in Twentel by the use of the `WHERE - ENDWH` construction, which can be seen in the following Subsection. Mutual recursive equations can also be used easily. Hardly any language has the 'simultaneous assignment' operator, as used in the predator-prey paradigm: `[W, R] = [f [W, R], g [W, R]]`. "The beginner is correct to believe we should not have to use temporary variables here" [23], and we do not have to in functional languages!

ZIP is the simple merger of equal length lists. Some examples demonstrate its effect:

```
ZIP [[1,2,3], [i,ii,iii], [I,II,III]] = [[1,i,I], [2,ii,II], [3,iii,III]]

ZIP [[1..8], [‘a..‘h]] = [[1,‘a], [2,‘b], ...  [8,‘h]]

ZIP [[1,2], [i,ii], [I,II], [A,B]] = [[1,i,I,A], [2,ii,II,B]]
```

MAP is a kind of iterator on lists: a function f is applied to every element of the argument list L, thus resulting in a new list of the same length as L. MAP can also be defined using list comprehensions: MAP $f\ L = \{f\ l \mid l \leftarrow L\}$. MAP is Backus' 'Apply to All' (3.1.3.1). So,

```
MAP SQR [1..6] = [1,4,9,16,25,36].
```

Using '$\text{MAP}\ f\ L$' instead of $\{f\ l \mid l \leftarrow L\}$ might look hardly an improvement in comprehensibility. However, by using MAP, any function f on a 'simple' argument can be transformed into one which has a list of those 'simple' arguments as argument, which, in its turn, can then be used as argument for another — higher-order — function.

FILTER is a decision function over a list, and like MAP, a kind of iterator. Its application results

in a new list where every element satisfies the filter-condition, the predicate p. FILTER, just like MAP, can be defined with list comprehensions: FILTER $p\ L = \{l \mid l \leftarrow L, p\ l\}$. So,

```
FILTER EVEN (MAP SQR [1..6]) = [4,16,36].
```

FOLD is an important function. It is the true generalisation (in operation and in domain) of the mathematical Σ operator. The generalised function must be associative. It has two forms, right- and left-FOLD. The right-FOLD is Backus' 'Insert' (3.1.3.1), sometimes also called REDUCE. Its informal definition is as follows, with a function f, an initial value a, and a list:

$$\text{FOLDR } (\oplus)\ a\ [x_1, x_2, \ldots, x_n] = x_1 \oplus (x_2 \oplus (\ldots (x_n \oplus\ a) \ldots))$$

'$(\oplus)$' being the notation for the prefix functional form of the associative (infix) operator $\oplus$ (also known as 'section'), (e.g., $(+)$ for ADD), and with parentheses grouping to the right. With parentheses grouping to the left, it becomes:

$$\text{FOLDL } (\oplus)\ a\ [x_1, x_2, \ldots, x_n] = ((\ldots (a\ \oplus\ x_1) \oplus\ x_2) \ldots) \oplus\ x_n))$$

So we have,

SUM L = FOLDR ADD 0 L
PRODUCT L = FOLDR MULT 1 L
CONCAT L = FOLDR APPEND `[]` L
PACK L = FOLDL $(\oplus)$ 0 L, `WHERE` $n \oplus x = 10 * n + x$ `ENDWH`

The last function is

$$\text{PACK } [x_{n-1}, x_{n-2}, \ldots, x_0] = \textstyle\sum_{k=0}^{n-1}\ x_k * 10^k,$$

the coding of a sequence of digits as a single number, with most significant digit first.

We conclude with a sample of the different styles possible in this equational style, assembled by Lex Augusteijn [5]. In these examples we see the use of a.o., higher order functions, list comprehensions, recursion, infinite data structures, and local definitions. We will use ITERATE, which is defined as follows [11]:

ITERATE f x = x : ITERATE f (f x), or mathematically,
$\text{ITERATE } f\ x = [x,\ fx,\ f^2x,\ f^3x,\ \ldots]$

The examples show how to list all powers of x. The '`(x*)`' is the notation for yielding an intermediary higher-order function, which yields the product of x and its argument.

```
powers x = { x^n | N <- [0 ..]  }
         = f 0 WHERE f n = x^n :  f (n+1)
         = APPEND [1] (MAP (x*) (powers x))
         = p WHERE p = APPEND [1] (MAP (x*) p) ENDWH
         = p WHERE p = APPEND [1] { x * y | y <- p } ENDWH
         = ITERATE (x*) 1
```
□

3.3.2 Examples of Programming in Twentel

As we use Twentel throughout this dissertation for demonstration purposes, this is the place to give a view of what (programming in this particular functional language) Twentel is like. The system around Twentel has been built as a classic interpreter, with a top level read-eval-print loop, with the Twentel program text typed in or read from a file.

3.3.2.1 The Murskii Algebra

A problem based on the Murskii algebra,[15] offers a nice introduction to some aspects of programming in a functional language. The problem is taken from Fleck [22]. The Murskii algebra is a three element algebra, M, defined with the following non commutative multiplication (where $row * col$ yields the element $m_{row,col}$ in the matrix M).

M	col	0	1	2
row	0	0	0	0
	1	0	0	1
	2	0	2	2

The problem under consideration is the determination of the elements that constitute the subalgebra M^{27} (27-tuples from M with elementwise multiplication), which is generated by $g^1 = (012)^9$, $g^2 = (0^3 1^3 2^3)^3$, $g^3 = 0^9 1^9 2^9$; so $g^2 = (0^3 1^3 2^3)^3 = (000111222)^3 =$ 000111222000111222000111222.

First, the straight conversion from the Marinda program as given in [22], to Twentel:

```
DEF
    m row col = IF row = 0 -> 0,
                   row = 1 -> IF col = 2 -> 1, -> 0 FI,
                   row = 2 -> IF col = 0 -> 0, -> 2 FI
                FI,
    g1 = [0,1,2,0,1,2,0,1,2,0,1,2,0,1,2,0,1,2,0,1,2,0,1,2,0,1,2],
    g2 = [0,0,0,1,1,1,2,2,2,0,0,0,1,1,1,2,2,2,0,0,0,1,1,1,2,2,2],
    g3 = [0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,2,2,2,2,2,2,2,2,2],

    makelm xs ys = MIP m xs ys,
    stages  = [ [], [g1, g2, g3] ]: stages1,
    stages1 = MAP next stages,
    next [os, ns] = [ os ++ ns, setdiff (allprod os ns) (os ++ ns)],
    setdiff xs ys = unique ((unique xs) ++ ys),
    allprod os ns = prods os ns ++ prods ns os ++ prods ns ns,
    prods xs ys   = unique { makelm x y | x <- xs, y <- ys },
```

[15] The Murskii algebra, the smallest example of an algebraic structure not having a finite equational axiomatisation, is defined in: V.L. Murskii, "The existence in the three-valued logic of a closed class with a finite basis, not having a finite complete system of identities", *Soviet Math Dokl*, **6** (1965), 1020–1024 [22]. Murskii is occupied in automata theory.

```
    unique lst = CASE lst IN
                  [] -> [],
                 l:L -> l: unique { x | x <- L; x<>l }
                 ESAC
FED
DO stages >> "con" >> OD
```

Proper formatting of the output of `stages` yields already a valid solution. But there are too many similar constructions around in this example: see the structure of `g1, g2` and `g3`, and the way `next` and `allprod` are defined. That calls for simplification.

There are three basic elements: `0, 1, 2`. We observe sequences of these elements, one, three, or nine elements long. This translates in a sequence constructor, or copier of elements. Heterogeneous sequences are also copied. Here the copier of elements comes in handy if we replace the element by a list. The basic heterogeneous sequence of basic elements, a part of the complete generator, must also be generated, before it can be passed to the copier of lists. This is handled by a list comprehension. Inside the list comprehension we find the copier of lists operating on the basic elements used as singleton.

```
DEF
    baselms = [0,1,2],
    copyl l n = IF n = 0 -> [],
                         -> l ++ (copyl l (n-1))
                FI,
    genpart n = CAT { copyl [x] n | x <- baselms },
    g1 = copyl (genpart 1) 9,
    g2 = copyl (genpart 3) 3,
    g3 = copyl (genpart 9) 1,
FED
```

The expanded `g1, g2` and `g3` now yield:

```
g1 = [0,1,2,0,1,2,0,1,2,0,1,2,0,1,2,0,1,2,0,1,2,0,1,2,0,1,2]
g2 = [0,0,0,1,1,1,2,2,2,0,0,0,1,1,1,2,2,2,0,0,0,1,1,1,2,2,2]
g3 = [0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,2,2,2,2,2,2,2,2,2]
```

One is tempted to go further, and also do something about the 1-9, 3-3, and 9-1 combinations (divisors of 9). However, as we are not sure about the reason for these numbers, we will not form a 'theory' about them.

One of the most striking omissions in the light of using functional languages is overlooking the presence of a transitive closure (`tc`) of `next` (the functional argument `f`) on `stages` (the set argument `Set`). We can now rewrite the above program, making a few illustrative, beginner's errors. These errors show the 'error-preventing' aspect of typing (3.1.7). Note that no errors could have been made in the control of the 'information flow'. With the introduction of a 'proven' building block, like the transitive closure, thinking is set at a higher level of abstraction, dispensing of control.

```
DEF
    gens         = [g1, g2, g3],
    makelm xs ys = MIP m xs ys,
    solution     = tc fmt gens,
    fmt          = (CPS unique (W makelm)),

    W  f x   = f x x,
    tc f Set = clos [] Set
               WHERE clos found inspect = inspect ++
                     CASE inspect IN
                          [] -> [],
                             -> clos newfound (unique gen)
                     ESAC WHERE gen      = COMPL newfound (f inspect),
                                newfound = found++inspect
                          ENDWH
               ENDWH
FED
```

In this way there is a level too much, which we can see from the following expansion of `fmt`.

```
CPS  unique  ( W  makelm) [ g1, g2, g3 ]
  -> unique  ( W  makelm  [ G ] )
  -> unique  ( makelm  [ G ] [ G ] )
  -> unique  ( MIP  m  [ G ] [ G ] )
  -> unique  ( { m x y  |  [ x, y ]  <-  ZIP  [ [ g1, g2, g3 ] [ g1, g2, g3 ] ] }
                                     <-  [ [ g1, g1 ], [ g1, g2 ], [ g1, g3 ], ...
```

`m` is given a pair of complete generators, instead of the paired elements of a pair of complete generators. Here we will use a `MIP`. `MIP` is defined as a `MAP` with `ZIP` combination:

```
MIP f X Y = { f x y | [x, y] <- ZIP [X, Y] }
```

To set things right, a `MIP` is called for:

```
makesol xs ys     = MAP makelm xs ys
makelm gena genb  = MIP m gena genb
```

On closer inspection of `fmt` we observe a superfluous `unique`. Also another error is manifest now: the `ZIP` at `makesol` level does not yield a Cartesian product but a 'zipped' list, which is quite another thing.

```
makesol makelm ga gb = { makelm g1 g2 | [g1, g2] <- cartpr ga gb }
```

So we come to the following Twentel program:

```
DEF
    origens = [g1, g2, g3],
    makesol ga gb  = MAP makelm (cartpr ga gb),
    makelm [g1,g2] = MIP m g1 g2,
    solution   = tc (W makesol) origens,
```

```
    cartpr x y = { [a,b] | a <- x, b <- y },
FED
```

The complete Twentel Murskii program (with necessary formatting), `DO stages >> "con" >> OD`, finally yields the following (partly given) output. These 64 elements comprise the subalgebra M^{27} of the Murskii algebra consisting of, apart from the three generators g^1, g^2 and g^3 (numbered `1, 2` and `3`), 61 other elements (numbered `4` through `64`):

```
[
    1 [012012012012012012012012],
    2 [000111222000111222000111222],
    3 [000000000111111111222222222],
    4 [010012002010012002010012002],
    5 [010010010012012012002002002],
    6 [000110022000110022000110022],
      ...
   51 [000000000000000000000002022],
   52 [000000000000000000000002002],
   53 [000000000000000002000000002],
   54 [000000000000000000000000002],
   55 [000000000000000012000000012],
      ...
   63 [000000000000000001000000002],
   64 [000000000000000011000000022]]
```

□

3.3.2.2 Token Stream Parsing

The problem can be stated as follows: we have a description (a string of data identifiers, meaning how to interpret the accompanying data: as an integer, a character, a boolean or as a real) of a sequence of data which we want to assemble from a stream of tokens, with a token being a data identifier with its matching data. So we need a kind of merging function on the description of the data with the stream of data: as the tokens come in, they must be matched to the description.

```
DEF
    assembly = merge description tokens
FED
```

Saving the data that answer the description takes place in this merge function. The merge function can be described as a decision function on the head of the description and the head of the data stream, which in true recursive fashion, is followed by the original merge function on the tail of the description and the tail of the data stream. There is one problem: if the head description item does not tally with the head data item, the description item must be saved to test the next token against. So:

```
DEF
    merge (s:S) (t:T) = merge_1 s t : merge statesave T
```

```
          WHERE statesave = nxS s S t,
                nxS s S t = IF tag = s -> S,
                                    -> s:S  FI    ENDWH,
    tag:data  = t,
    merge_1 s t = IF tag = s -> data,
                             -> []   FI
FED
```

A close look at these functions ("retrospect on your work", as Robert Floyd said) shows the same pattern in a decision regarding the tag of the token under consideration. As the description is finite, there must be a test on the end of it (test on the empty list, `[]`). The token stream does not need such a test, as it is infinite.

We consider the function we constructed so far. What was made is a kind of filter on the token stream, and not a merge, as no elements of the description are moved to the assembled record. Finally we get:

```
DEF
    assembly = filter description tokens,
    filter S (t:T) = CASE S IN
                     []     -> [],
                     s:S    -> IF tag = s -> data : filter S  T,
                                          -> filter (s:S)  T  FI
                     WHERE  (tag : data) = t  ENDWH
                     ESAC
FED
```

Here once more we encounter Jackson's idea, that in many cases the instruction sequence is determined by the data structure [21]. Here the data structure is the list, so the equations reflect that structure.

3.4 Elaborating Functional Programs

Executing a functional program means interpreting the program text, or compiling and running it. In demonstrating in a general way how to elaborate functional programs, we use the elaboration of a Twentel program as an example. This process shows enough fundamental steps, (partially) present in other systems too, to give an idea of such an elaboration in general. Paul Hudak and Willem Vree give comprehensive overviews of the various possible mechanisms in this process [35, 70]. Simon Peyton Jones gives a very detailed and well founded treatment of how to implement a functional language [53], essentially this Subsection in 445 pages.

3.4.1 Introduction

What plays a role in elaborating a functional program? The largest role is taken by rewriting. Rewriting is done in order to get at the normal form of the set of equations which comprises the

functional program. The normal form in our case is just the value of the given set of equations, the 'desired behaviour' for which we set up the equations. There are many phases in the process of obtaining the normal form.

We will not discuss the transformation of the Twentel program text to the 'enriched lambda calculus'. A detailed description of this process can be found in 'Peyton Jones' and, for Twentel, in [43].

Implementation problems with the lambda calculus (especially details relating to *beta*-reduction [43]) necessitate another transformation where no abstractions (operations regarding bound variables in a function body) are used. For the target of this transformation we use combinatory logic, naively 'a lambda calculus without abstraction'. Yet it is a system, equivalent to the lambda calculus. Its basic components are combinators, and its basic operation is, again, the application of a function to its argument.

The essence of the translation from Twentel program text to the enriched lambda calculus to combinator constructions is that all occurrences of an identifier are swept into one instance, and by proper introduction and positioning of the combinators introduced in this Section, the evaluation mechanism yields the value the program text was meant to deliver.

This subsequent transformation into the basic program and data format — a graph — of the evaluating engine — a graph-reduction machine — with combinators and built-in functions as elementary machine actions, will be dealt with in two steps. First the combinators. These are a part of the basic 'reduced instruction set' for our evaluating engine, they comprise its 'machine code'. Finally graph rewriting. We will briefly touch upon this subject, as it yields an advantage, not directly present in the normal order reduction essential for lazy evaluation.

The remark from the preamble to this Chapter, "It turns out that the answers to these questions [the how, what and why of functional languages] are very strongly related", can now be placed: everything relates to the lambda calculus.

3.4.2 Origin and Nature of Combinatory Logic

In this Subsection we introduce *Combinatory Logic*, however, to a lesser extent than we did in the case of the lambda calculus. Readers intent on mastering combinatory logic must consult either Stenlund [66], Hindley et al [31], Rosenbloom [62], or, of course, the standard work on this subject by Curry and Feys [18]. For an easy introduction in Dutch one might consult Van der Poel [59], though he deviates from the standard nomenclature due to his interests in number theory (see also his [58]).[16]

Independently of each other, Moses Schönfinkel in 1920 and Haskell B. Curry (1900–1982) in the late twenties started their analysis of the ultimate foundations of mathematical logic. Schönfinkel wanted to remove the variable from the basic notions of logic, as they are only used for linguistic purposes instead of playing an essential part in logic: "Man wird auf dem in ersten Augenblick gewiss äußerst kühn erscheinenden Gedanken geführt [...] auch die noch verbleibenden

[16] A really very nice, and puzzling, book on combinatory logic, is the one "dedicated to the memory of Haskell B. Curry, pioneer in combinatory logic, and enthusiastic birdwatcher": Raymond Smullyan's *To Mock a Mockingbird,* Alfred A. Knopf, New York, 1985.

Grundbegriffe [...] zu beseitigen zu suchen." And he continues: "[...] als die Veränderliche in der logischen Aussage ja nichts weiter als ein Abzeichen ist, um gewisse Argumentstellen und Operatoren als zusammengehörig zu kennzeichen, und somit den Character eines blossen, dem konstanten, 'ewigen' Wesen der logischen Aussage eigentlich unangemessenen Hilfsbegriffes hat." [64].

He created a function calculus based upon a definition of function like the one by Church given above with (implicitly) application as the only operation. Then he showed that, by introducing two very general functions, $\mathsf{C}\,xy = x$ and $\mathsf{S}\,fgx = fx(gx)$, every function in this system can be reduced to a combination of C and S, in which combination the variables that played a role as formal arguments are absent. (Actually he also introduced $\mathsf{I}\,x = x$, $\mathsf{T}\,fxy = fyx$, $\mathsf{U}\,fg = fx|^x gx$, and $\mathsf{Z}\,fgx = f(gx)$, but, as he showed, these were reducible to combinations of C and S).

Curry defined the terms *combinator* and *combinatory logic* in his early work on this subject in 1930. In Curry's 'combinatory logic' there is no distinction between different categories of entities, hence any construct formed from the primitive entities by means of the application of a function to its argument (the only allowed operation) must be significant in the sense that it is admissible as an entity. The system of combinatory logic is combinatorially complete, which means that any function we can define intuitively by means of a variable can be represented formally as an entity of the system. Combinations formed from variables - only by means of the application operation - can be represented by certain operators which are functions of these variables (and perhaps other ones). These operators must be present as entities of the system as it is combinatorially complete. Such an entity, together with the combinations of them formed under the application operation, is called a *combinator*. Combinatory logic then, is the part of mathematical logic which deals with combinators. Curry also set the frame of reference for combinatory logic: that is why Schönfinkel's C, T and Z are henceforth called K, C and B.

Regarding the importance of combinators, Dana Scott remarks: "[The combinators] are interesting enough to merit study [...] Magic may have its place, but it is not here. There is more than sufficient evidence that they have a job to do — and if you do not fully understand how they do it, that is your fault, not theirs. [...] The conceptual basis for the lambda calculus is the notion of function, and the combinators are certain very general functions that can be used to define new functions from old." [65].

The lambda calculus and the combinatory logic are equivalent [9, 35]. We can now give a definition of a combinator in terms of the lambda calculus: A combinator is a lambda expression which contains no occurrences of a free variable [9], it is a closed lambda term.

3.4.3 Functions and Combinators

This Subsection is a sidestep from the main line of discourse. We introduce it here, since by now, both notions, 'function' and 'combinator', have been defined.

What is the difference, or rather correspondence, between a combinator and a function denoted by a function constant in a language? The first one is essentially typeless, as it stems from the untyped lambda calculus. The second one has a type (domain and range). In extending the

functional language we cannot use a function, as we want to be completely free in our choice of arguments for the new construct, wherever we use it. A function that would deviate from the proper definition would do, but then we define a total function over 'everything' to 'everything', which is exactly what a combinator is, a very general function.

3.4.4 Using Combinatory Logic

In this Subsection we introduce the elementary combinators. These general functions will serve as basic instructions of the graph rewriting machine. Definition, intuitive meaning, use and handling them in expressions are the main subjects. We will touch very briefly on the notion supercombinators, as they are 'present' in Twentel. In the Appendix we give a larger exercise in handling combinators.

3.4.4.1 Elementary Combinators

In the definitions (as already tacitly has been done) the equal sign is not used as a Boolean producing function, but as an equivalence in a rewriting rule. The '$\rightarrow$ ' in a rewrite rule must be read as 'reduces to'. There are two basic combinators: K and S. It can be proved that every other combinator can be reduced to a combination of these two [18, 64].

1. **The Elementary Cancellator K (German 'Konstant')**

   The constant function K returns a fixed value (its first argument) as a function of the second argument, whatever it may be. It can express a constant as a function of something. So it can be used to eliminate a variable in a certain position or introducing a variable in a given position with a view to further manipulations. The first use is clear from its definition: $\mathsf{K}\,xy = x$, whatever y is. So $\mathsf{K}\,x(\mathsf{S}\,(\mathsf{B}\;\mathsf{S})\,\mathsf{B}\,x(\mathsf{K}\;\mathsf{B}\;\mathsf{B}\,y)x) \rightarrow x$.

   Its second use is less obvious: if we want to rewrite $\ldots XyZ\ldots$ and for some reasons we need (and can use) an S-reduction there, with $f = X, x = y$ and g to be supplied, $\ldots XyZ\ldots \rightarrow \ldots Xy(\mathsf{K}\,Zy)\ldots \rightarrow \ldots \mathsf{S}\,X(\mathsf{K}\,Z)y\ldots$.

2. **The Elementary Combinator S (Schönfinkel's 'Verschmelzungsfunktion')**

   Its definition $\mathsf{S}\,fgx = fx(gx)$ is rather opaque,[17] as it is not intuitively clear why x has to be doubled in this way, and why the introduction of parentheses must occur at the same time as the doubling of the x. Curry didn't even use it until the forties. Schönfinkel uses it when connecting two functions while both are dependent on one variable, e.g., $fxy = x\log y$ and $gz = 1 + z$, then $\mathsf{S}\,fgx = fx(gx) = x\log(1 + x)$.

   Another use is in the handling of combinations where one needs to reduce the number of times a variable (or even a function) occurs: $\sin x(\cos x) = \mathsf{S}\sin\cos x$.

3. **The Elementary Identificator I**

[17] In 1982 Jan Willem Klop gave an appropriate definition, deeply rooted in history, when on a working visit at the IAC Institute of the C.N.R. in Rome (notably a city with a sort of Church-Rosser property: all roads lead to it). He used in his lecture, as the Romans do, S P Q R = P R (Q R) [41].

The identity function $\mathsf{I}\,x = x$ expresses a variable as a function of itself. It can be used as 'padding' before another combinator can be used. Take $Xxy \to \cdots \to yx$. As C can only be used with three arguments, we rewrite $xy \to \mathsf{I}\,xy \to \mathsf{C}\,\mathsf{I}\,yx$, so $X = \mathsf{C}\,\mathsf{I}$.

4. The Elementary Permutator C

The changing function $\mathsf{C}\,fxy = fyx$ gives the opportunity of altering the sequence of variables. The permutator C can be used to move a variable into a position where it can be tackled by another combinator, e.g., $xy(yx) \to xy(\mathsf{I}\,yx) \to xy(\mathsf{C}\,\mathsf{I}\,xy) \to \mathsf{S}\,x(\mathsf{C}\,\mathsf{I}\,x)y \to \mathsf{S}\,\mathsf{S}\,(\mathsf{C}\,\mathsf{I})xy$.

5. The Elementary Compositor B

With the composition function $\mathsf{B}\,fgx = f(gx)$ parentheses can be (re)moved. Note the distinction between $f(gx)$ and $fgx = (fg)x$, which is due to the left associativity of the application operation. Its intuitive meaning is that of the composition of two functions. Take $f = \log$ and $g = \sin$, then $\mathsf{B}\,fg$ is the logarithmic sine function, or, with $f = g = D$ for differentiation $\mathsf{B}\,DD$ is the twice differentiating function. B has significance as an operation of one argument: it converts a function f into the corresponding operation on functions.

Another way of writing this composite product is the dot expression: $\mathsf{B}\,fg = f \cdot g$. This dot product has a lower precedence than the application operation, so $f \cdot gx = f \cdot (gx)$ and not $(f \cdot g)x$. This is clear from the definition $\mathsf{B}\,fgx = f \cdot gx = f(gx)$. Backus also uses this functionality in his 'composition' combining form (3.1.3.1).

The dot has another nice property: it is associative (unlike the application). So $(f \cdot g) \cdot h = f \cdot (g \cdot h)$. This property is easily demonstrated, using a basic property of Curry's combinatory logic, the 'extension property': if for every x, $Xx = Yx$, then $X = Y$, X and Y being any combination of combinators and constants.

6. The Elementary Duplicator W

The doubling function $\mathsf{W}\,fx = fxx$ applicates a binary function on the same argument: $\mathsf{W}\,{+}x \to {+}xx$ (or $2x$). It is the doubling function, because W is a double U. In the handling of combinations it can be used to reduce the number of times a variable occurs.

7. The Formalising Combinator Φ

The formalising combinator $\Phi\,fabx = f(ax)(bx)$ is used for the same reasons as S, but in more complex cases. This Φ combinator has somewhat the same significance with respect to functions of two arguments, that B had with respect to unary functions. Thus, if A represents addition, $\Phi\,Afg$ represents the sum of two unary functions f and g. 'Formalising' is suggested by the following example: if P represents implication (between propositions) then $\Phi\,P$ is implication between unary predicates, and $\Phi\,(\Phi\,P)$ between relations.

8. The Paradoxical or Fixpoint Combinator Y

The paradoxical combinator Y is a combinator which provides us with the possibility of passing functions as arguments to themselves in recursion. Curry and Feys thought it exhibited properties akin to the Russell paradox, hence 'paradoxical' [18]. They did not link it to the fixpoint theorem of the lambda calculus (3.2.4). Today Y is mostly referred to under its fixpoint name: the fixpoint operator. The fixpoint combinator yields the

fixpoint of its argument, $\mathsf{Y}\,f$. So this combinator is defined as $\mathsf{Y}\,f = f(\mathsf{Y}\,f)$ (see cover logo). With this combinator we can handle recursion in our definitions.

3.4.4.2 Working with Combinators

How combinators work is shown in the following example. Take M as a combination of some constants and the variable x. The constants of the system can be values ($\sqrt{2}$, 3.1415, 2.7181) or constant functions, like the prefix + and $*$, so we take the lambda calculus style $M = +(*xx)(*2x)$ (which is derived from $x^2 + 2x$). One can prove that an F exists that is a combination of K, S and the above constants only (and so does not contain the variable x), such that $Fx = M$. For simplicity, other elementary combinators are admitted here, as they all are reducible to K and S. Then $M \rightarrow +(\mathsf{W}*x)(*2x)$. And with $f = +$, $a = \mathsf{W}*$ and $b = *2$, this reduction of M yields: $+\ (\mathsf{W}*x)(*2x) \rightarrow \Phi +(\mathsf{W}*)(*2)x$, so $F = \Phi +(\mathsf{W}*)(*2)$, which, as it does not contain x, is constant in x. Another example, is the function given by: $\lambda\, x\,.\ (x+1)^2$ which is in combinator form: $\mathsf{B}\,(\mathsf{W}*)(\mathsf{C}+1)$.

Curry remarks on such forms, as they do not resemble '$x^2 + 2x$' or '$(x+1)^2$' any more [18]:

> It will be seen that the equations are less perspicuous than the originals. In fact it is not maintained that the elimination of variables is a practical expedient; its importance is theoretical, in that it shows that variables are a logically unnecessary but practical very useful device.

3.4.4.3 Supercombinators

In this exposition of combinators we also introduce the notion of a 'supercombinator' [53]:

Supercombinators A 'supercombinator', \$S, of 'arity' n is a lambda expression of the form:

$$\lambda\, x_1\,.\ \lambda\, x_2\,.\ \ldots\quad \lambda\, x_n\,.\ E$$

where E is not a lambda abstraction (this just ensures that all the 'leading lambdas' are accounted for by $x_1 \ldots\ x_n$) such that

- \$S has no free variables;
- any lambda abstraction in E is a supercombinator;
- $n \geq 0$; that is, there need be no lambdas at all. □

Based on this definition, Peyton Jones distinguishes two kinds of supercombinators:

- the kind that is introduced by Hughes in his dissertation [36]. This supercombinator is created in 'compiling' or 'transforming' a functional program into an equivalent one where only this kind of lambda expressions occur; the created supercombinators being the units of compilation of the program;

- the other kind is the fixed set of combinators used in the next Subsection as basic 'machine' instructions. Here the term 'super'combinator is incorrectly used (though it corresponds to the above definition), as it precisely covers the 'Curry' combinators introduced above.

3.4.5 Graph Rewriting

In the introduction to this Chapter we described globally how a functional program, e.g., in Twentel, could arrive at its 'desired behaviour'. In this Subsection we will look at some details. A far more detailed treatment can be found in [53], and for Twentel especially in [43].

For evaluating Twentel programs one needs lazy evaluation (3.1.3.2) which can only be implemented by normal order reduction of the lambda expression (3.2.5). Normal order is the surest and the safest way to reduce a lambda expression, and as we have seen, combinations can also be seen as lambda expressions. But it has its disadvantages, in that it does not yield the normal form in the minimal number of steps (not the most efficient way of reducing). There is a solution to this problem, due to a domain switch.

Lambda expressions and their counterparts, the combinations, are normally reduced using string reduction, which precludes any possibility of sharing. Wadsworth suggested in his PhD dissertation in 1971 to use graphs instead of strings as basic reduction medium: which suggestion is essentially a domain switch [72]. The advantage of both string and graph rewriting is the absence of an 'environment', a table with all bound arguments to their values. The disadvantage of string rewriting, viz multiplication of subexpressions in a rewriting, is absent in graph rewriting. Graphs can — what strings cannot — use pointers to common subexpressions. These subexpressions, evaluated once in the canonical reduction order, have their value available at once at all points referring to them. In this way, the normal order 'call-by-name' changes into a 'call-by-need' parameter passing mechanism. It combines the lazy evaluation of the normal order (only evaluate if requested) with the evaluate-only-once of the applicative order (through the sharing mechanism). 'Normal order graph rewriting' arguments are now evaluated 'at-most-once'.

The program and data medium of the graph reduction machine is a tree-like graph with combinators and built-in functions at the end nodes as machine instructions. It is constructed from the basic node, shown in Figure 3.1 on the left, the application node. This node represents a function application, so it has the '`@`' as application operator, and a pointer to the function and its argument; note that the basic application node mirrors the elementary unary function of the lambda calculus.

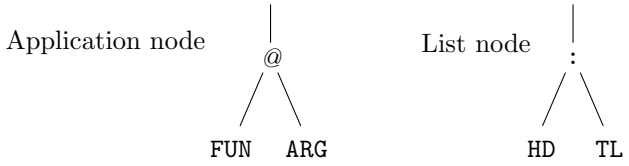

Figure 3.1: *The two basic kinds of nodes in graph reduction*

The basic composite data node for the list (its representation on the right side of Figure 3.1) is elementary, like LISP's dotted pair, with '`:`' the CONS operator, and pointers to the head and tail of the node. We assume the basic, or atomic, data items to be represented accordingly.

A program is now represented in the graph space as follows, with the top application, the 'root', and the continuous path, the 'spine', from the root via function pointers (left branches) to the terminal node, the 'tip'.

Normal order reduction for a lazy regime implies that it is not necessary to reduce *all* redexes of the original expression, as it can contain unnecessary inner redexes. We convert it until there are no top-level redexes left, and then the expression is in 'weak head normal form' [53]:

Weak Head Normal Form A lambda expression is in 'weak head normal form' (WHNF), iff it is of the form:

$$F\ E_1\ E_2\ \ldots\ E_n$$

- with $n\ \geq\ 0$ and,
- either F is a variable or data object,
- or F is a lambda abstraction or built-in function and ($F\ E_1\ E_2\ \ldots\ E_m$) is not a redex for any $m\ \leq\ n$.

An expression has no 'top-level' redex iff it is in WHNF. □

We can now present the simple operational cycle of the reductor, the graph reduction machine:

REPEAT

- SELECT THE TOP-LEVEL REDEX OF THE EXPRESSION, in order to arrive at WHNF for the expression. This selection proceeds as follows: creeping up the spine ('unwinding' it), and at the end of it (at the tip) finding an item. Then we have the situation, shown in Figure 3.2. The item `f` at the 'function position' of the 'deepest' function application, can be:

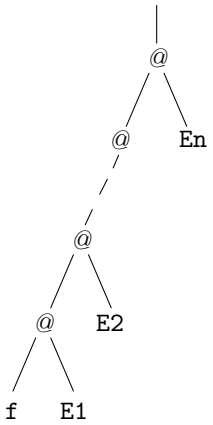

Figure 3.2: *The spine of '$f\ E_1\ E_2\ \ldots\ E_n$' in graph space*

  - a data object (atomic data item or list cell) in which case the expression is in WHNF and the selection can stop, but then $n\ =0$ as otherwise a data object would be at a function position, an error; normally the transformer to the combinator graph will not deliver such a structure to be elaborated, as type checking will find out such an error;

  - a combinator or built-in function, with m arguments.
    If $m \leq n$ then ($f\ E_1\ E_2\ \ldots\ E_m$) is the selected redex, and can be reduced, otherwise (lack of arguments) the expression is in WHNF, and the selection process can stop.

- Rewrite the graph space according to the reduction of the selected redex, with the selected redex being:
  - a combinator: reduce it by rewriting the graph space with the redex' function and argument(s) as left part of the rewrite rule;
  - a built-in function: reduce it by overwriting the root of the redex by its value; this value is obtained by executing it with its argument(s) in normal form; if the argument(s) are not in normal form, call the reductor recursively on the argument(s), E_1 first.

until Expression in WHNF. □

Here we must admit that we cheated on the 'unary function application': as a matter of efficiency (and comprehensibility) combinators and built-in functions are treated as n-ary functions straight away (cf., the second kind of 'supercombinator' in 3.4.4.3). Take as an example in the above situation `f = ADD, E1 = 10, E2 = 7`, which corresponds to the Twentel text '... `(ADD 10 7)`'. Then in true lambda spirit, the following two reductions will occur. First '`ADD 10`' is reduced to the function '`ADD10`', the function that adds 10 to its argument. Then '`ADD10`' is applied to '`7`', yielding '`17`'. This value results after only one reduction step in the algorithm given above, instead of the formalist two.

Rewriting Example Finally, a short example to clarify the process. Take the following equations, taken from a very simple Twentel program:

```
DEF
   x^2 + x WHERE x = 1 * 2 * 3 ENDWH
FED
```

These could be translated into the following lambda expression

($\lambda x\,.\,x^2 + x$) (* (* 1 2) 3)

that, subsequently, is 'compiled' into (unoptimised) combinator code

S ADD SQR (MUL (MUL 1 2) 3)

and represented in graph space as in Figure 3.3 (with '`[a]`' shorthand for the argument of the lambda abstraction (MUL (MUL 1 2) 3))

The first top-level redex is '`S` ADD SQR `[a]`', and reducing it, or rewriting the graph space for a combinator with three arguments ($\mathsf{S}\,P\,Q\,R\ =\ P\,R\,(\,Q\,R\,)$), is shown in Figure 3.4 (note in this rewriting the sharing of the argument `[a]`).

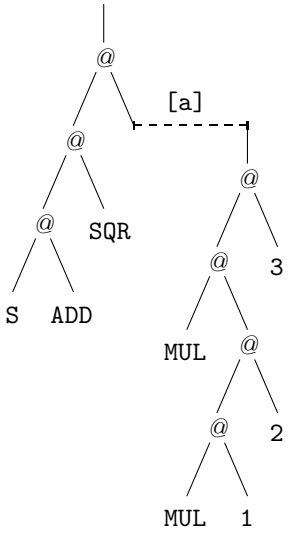

Figure 3.3: *Combinator code for "*S ADD SQR (MUL (MUL 1 2) 3)*"*

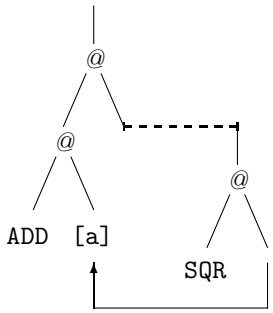

Figure 3.4: *Graph space, one reduction after Figure 3.3*

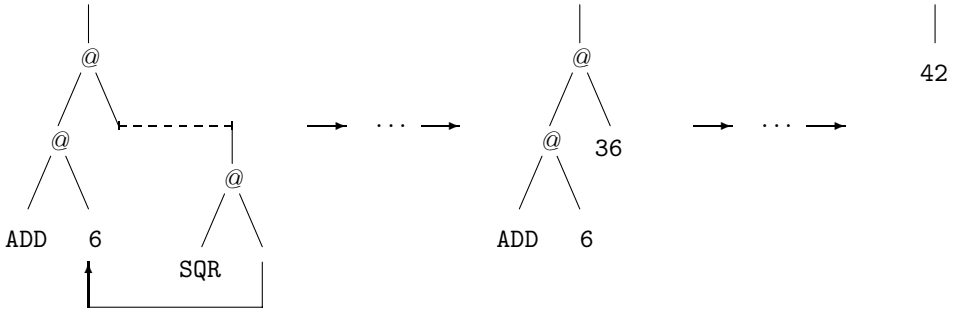

Figure 3.5: *Final stages in the reduction of "*S ADD SQR (MUL (MUL 1 2) 3)*"*

The expression (see Figure 3.5) is still not yet in WHNF, so the next top-level redex is 'ADD `[a]` ...'. This built-in function is strict in both its arguments, so the reductor is recursively applied to, respectively '`[a]`', thus going down the MUL-chain ('a MULberry bush') yielding '6', and then to 'SQR 6', which had its argument, '`[a]`', already evaluated, and can now be reduced straight away yielding '36'. Finally, after having reduced both arguments of ADD to normal form, the ADD redex is reduced, yielding '42', which is the normal form (and WHNF, in this case no other redexes remaining) of the 'program'.

References

[1] Abelson, H., Sussman, G.J., with Sussman, J., *Structure and Interpretation of Computer Programs.* (with foreword by Alan J. Perlis), MIT Electr Eng and Comp Sci Ser, MIT Pr, Cambridge MA / McGraw-Hill Book Comp, New York, 1985.

[2] Amdahl, G.M., Blaauw, G.A., Brooks Jr, F.P., "Architecture of the IBM System/360", *IBM J Res Development* **8**(2) (Apr 1964): 87–101.

[3] Ambler, A.L., Burnett, M.M., Zimmerman, B.A., "Operational Versus Definitional: A Perspective on Programming Paradigms", *IEEE Computer* **25**(9) (Sep 1992): 28–43.

[4] Ashenhurst, R.L., Graham, S. (eds) *ACM Turing Award Lectures — The First Twenty Years, 1966 – 1985.* ACM Pr Anthology Ser, ACM Pr, New York / Addison–Wesley, Reading MA, 1987.

[5] Augusteijn, L., *Functional Programming, Program Transformations and Compiler Construction.* PhD Thesis, Eindhoven Univ Technology, Eindhoven, Oct 1993, 247 pp.

[6] Backhouse, R.C., "Constructive Type Theory: A Perspective from Computing Science", in: *Formal Development of Programs and Proofs.* E.W. Dijkstra (ed), Univ Texas at Austin Year of Programming Ser, Addison–Wesley, Reading MA, 1990, pp 1–32.

[7] Backus, J., "Can Programming Be Liberated from the Von Neumann Style? A Functional Style and its Algebra of Programs", *Comm ACM* **21**(8) (Aug 1978): 613–641 (1977 ACM Turing Award Lecture, 17 Oct 1977); also in [4], pp 69–129.

[8] Bailes, P.A., Salzman, E.J., Rosel, A., "A Proposal for a Bachelor's Degree in Software Engineering ", in: *Software Engineering Education*, N.E. Gibbs (ed), (Proc SEI Conf 1989 -title-, Pittsburgh PA, Jul 1989), Springer LNCS # 376, Springer-Verlag, Berlin, 1989, pp 90–108.

[9] Barendregt, H.P., *The Lambda Calculus, its Syntax and Semantics.* Studies in Logic and the Foundations of Mathematics, Vol 103, North-Holland, Amsterdam, 1981 (2^{nd} revised ed, 1984).

[10] Blaauw, G.A., Brooks Jr, F.P., *Computer Architecture — Vol 1: Design Decisions.*, Preliminary Draft, Lecture Notes, Privately Circulated, Univ Twente, 1975–1987, (to be published by Addison–Wesley).

[11] Bird, R.J., Wadler, P., *Introduction to Functional Programming.* Prentice Hall, New York, 1988, xv + 293 pp.

[12] Boute, R.T., *Functional Languages and their Application to the Description of Digital Systems.* presented at BIRA/ACM Workshop New Generation of Programming Languages and Supporting Architectures, (Antwerpen, Nov 1983), 17 pp.

[13] Boute, R.T., *System Semantics and Formal Circuit Description.* Rep Nr 69, Dept Informatics, Catholic Univ Nijmegen, Nijmegen, Aug 1985.

[14] Cann, D.C., "Retire Fortran? — A Debate Rekindled", *Comm ACM* **35**(8) (Aug 1992): 81–89.

[15] Carroll, Lewis. *Through the Looking Glass and What Alice Found There.* Macmillan, London, 1871.

[16] Church, A., *The Calculi of Lambda Conversion.* Princeton Univ Pr, Princeton NJ, 1941.

[17] Clocksin, W.F., Mellish, C.S., *Programming in Prolog.* Springer-Verlag, Berlin, 1981, xii + 279 pp.

[18] Curry, H.B., Feys, R., *Combinatory Logic.* Vol I, North-Holland, Amsterdam, 1958 (2^{nd} ed, 1974).

[19] Darlington, J., "The Structured Description of Algorithmic Derivations", in: *Algorithmic Languages*,

J.W. de Bakker and J.C. van Vliet (eds), North-Holland, Amsterdam, 1982, pp 221–250.
[20] Diller, A., *Compiling Functional Languages.* John Wiley, Chichester, 1988, 322 pp.
[21] Fairbarn, J., "Making Form Follow Function: An Exercise in Functional Programming Style," *Softw Pract & Exp* **17**(6) (Jun 1987): 379–386.
[22] Fleck, A.C., "A Case Study Comparison of Four Declarative Programming Languages", *Softw Pract & Exp* **20**(1), (Jan 1990): 49–65.
[23] Floyd, R.W., "The Paradigms of Programming", *Comm ACM* **22**(8) (Aug 1979): 455–460, (1978 ACM Turing Award Lecture, 4 Dec 1978), also in [5], pp 131–142.
[24] Frege, G., *Funktion und Begriff*, §36 (in German); as noted by W.V.O. Quine in his Introduction to the English translation of [64] in: J. van Heijenoort (ed), *From Frege to Gödel. — A source book in mathematical logic, 1879–1931.* Harvard Univ Pr, Cambridge MA, 1967, pp 355–357.
[25] Friedman, D.P., Wise, D.S., "CONS Should Not Evaluate its Arguments", in: 3^{rd} *Int'l Coll Automata, Languages and Programming*, S. Michaelson and R. Milner (eds), (Edinburgh, Jul 1976), Edinburgh Univ Pr, Edinburgh, 1976, pp 257–284.
[26] Gelernter, D., "Domesticating Parallelism", *IEEE Computer* **19**(8) (Aug 1986): 12–16.
[27] Glaser, H.W., Hankin, C.L., Till, D., *Principles of Functional Programming.* Prentice Hall Int'l, London, 1984, xi + 251 pp.
[28] Gries, D., "Introduction to Part IV, 'Data Types'", in: D. Gries (ed), *Programming Methodology; A Collection of Articles by Members of IFIP WG 2.3*, Springer-Verlag, New York, 1978, pp 263–267.
[29] Harrison, D., *Functional Real-Time Programming: The Language Ruth and its Semantics.* Univ Stirling, Dept Computing Sci, PhD Thesis, Tech Rep TR.59, Stirling, Sep 1988, 221 pp.
[30] Henderson, P., Morris, J.H., "A Lazy Evaluator", in: *Conf Rec* 3^{rd} *ACM Symposium POPL*, (Atlanta GA, Jan 1976), ACM, New York, 1976, pp 95–103.
[31] Hindley, J.R., Lercher, B., Seldin, J.P., *Introduction to Combinatory Logic.* London Math Soc Lecture Notes, Vol 7, Cambridge Univ Pr, Cambridge, 1972.
[32] Hindley, J.R., Seldin, J.P., *Introduction to Combinators and λ–Calculus.* London Math Soc Student Texts, Vol 1, Cambridge Univ Pr, Cambridge, 1986.
[33] Hoare, C.A.R., "Communicating Sequential Processes", *Comm ACM* **21**(8) (Aug 1978): 666–677.
[34] Van der Hoeven, G.F., *Preliminary Report on the Language Twentel.* Memorandum INF-84-5, Dept Comp Sci, Univ Twente, Enschede, Mar 1984, 87 pp.
[35] Hudak, P., "Conception, Evolution, and Application of Functional Programming Languages", *ACM Computing Surveys* **21**(3) (Sep 1989), 359–411.
[36] Hughes, R.J.M., *The Design and Implementation of Programming Languages.* PhD Thesis, Programming Res Group, Univ Oxford, Rep PRG-40, Oxford, Sep 1984.
[37] Hughes, R.J.M., "Why Functional Programming Matters", *Comp J* **32**(2) (Apr 1989): 98–107; (also in: *Research Topics in Functional Programming.* D.A. Turner (ed), (Proc Meeting Institute of Declarative Programming, Austin TX, Aug 1987), Univ Texas at Austin Year of Programming Ser, Addison–Wesley, Reading MA, 1990, Ch 2, pp 17–42).
[38] Jones, A.K. (ed), *Perspectives on Computer Science.* (Rec 10th Anniversary Symp Comp Sci Dept, Carnegie-Mellon Univ, 6–8 Oct 1975, Pittsburgh PA), ACM Monograph Ser, Academic Pr, New York, 1977.
[39] Joosten, S.M.M., *The Use of Functional Programming in Software Development.* PhD Thesis, Dept Comp Sci, Univ Twente, Enschede, Apr 1989, 140 pp.
[40] Joosten, S.M.M., *Functional Programming, Prototypes and Specifications.* Unpublished Draft, Dept Comp Sci, Univ Twente, Enschede, Feb 1992, 382 pp.
[41] Klop, J.W., *Personal Communication*, 23 March 1994.
[42] Korson, T., McGregor, J.D., "Understanding Object-Oriented: a Unifying Paradigm", *Comm ACM* **33**(9) (Sep 1990): 40–60.
[43] Kroeze, H.J., *The Twentel System, Version I — (1.24–1.99) Available on Various Machines: General Reference Manual & User Guide.* Dept Comp Sci, CAP / Languages Group, Univ Twente, Enschede, 1986–1987, 116 pp.
[44] Kuhn, T.S., *The Structure of Scientific Revolutions.* Univ Chicago Pr, Chicago IL, 2^{nd} ed, enlarged, 1970, xii + 210 pp.

[45] Lindstrom, G. *et al*, "Critical Research Directions in Programming Languages", (Rep Workshop sponsored by ONR), *SIGPLAN Not* **24**(11) (Nov 1989): 10–25.
[46] McCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P., Levin, M.I., Lisp *1.5 Programmer's Manual.* MIT Pr, Cambridge MA, 1962 (2^{nd} ed, 1965, vi + 106 pp).
[47] McCarthy, J., "History of Lisp", in: "Preprints Proc ACM History of Programming Languages Conf (HOPL)", R.L. Wexelblatt (ed), (Proc -title-, Los Angeles CA, Jun 1978), *SIGPLAN Not* **13**(8) (Aug 1978): 217–223, (also in: R.L. Wexelblatt (ed), *History of Programming Languages.* Academic Pr, ACM Monograph Ser, New York, 1981, pp 173–197, (incl transcripts of presentation, remarks and Q&A)).
[48] Te Meerman, G.J., *Personal Communication*, 24 July 1994; The position itself is corroborated four days later by A.J.W. Duijvestijn himself in a personal communication.
[49] Milner, R., *A Calculus of Communicating Systems.* LNCS # 92, Springer-Verlag, Berlin, 1980, vi + 171 pp.
[50] Milner, R., "Elements of Interaction", *Comm ACM* **36**(1) (Jan 1993): 78–89 (1993 ACM Turing Award Lecture).
[51] Nijholt, A., *Computers and Languages — Theory and Practice.* Vol 4, Studies in Comp Sci Artif Intelligence, North-Holland, Amsterdam, 1988, xiii + 482 pp.
[52] Ollongren, A., Herik, H.J. van den, *Filosofie van de Informatica.* Ser Wetenschapsfilosofie, Martinus Nijhoff, Leiden, 1989 (in Dutch).
[53] Peyton Jones, S.L., *The Implementation of Functional Programming Languages.* Prentice-Hall Int'l, Englewood Cliffs NJ, 1987, xviii + 445 pp.
[54] Perrott, R.H., "Development in Parallel Programming Languages", Rep, Dept Comp Sci, Queens's Univ, Belfast, pp 169–189.
[55] Peterson, J.L., "Petri Nets", *ACM Computing Surveys* **9**(3) (Sep 1977): 223–252.
[56] Petri, C.A., *Kommunikation mit Automaten.* Schriften des Rheinisch-Westfalischen Institutes für Instrumentelle Mathematik an der Universität Bonn, Heft 2, PhD Thesis, Bonn, 1962 (in German).
[57] Peyton Jones, S.L., "Yacc in SASL - an Exercise in Functional Programming", *Softw Pract & Exp* **15**(8): (Aug 1985): 807–820.
[58] Van der Poel, W.L., Schaap, C.E., Van der Mey, G., "New Arithmetical Operators in the Theory of Combinators", parts I, II, III; *Proc KNAW Ser A* **83**(3) (Sep 1980): 271–325, (also publ as *Indag Math* **42**(3) (1980).
[59] Van der Poel, W.L., "Combinatorische Logica", *Informatie* **15**(12) (Dec 1973): 667–676 (in Dutch).
[60] Quine, W.V.O., *Word and Object.* MIT Pr, Cambridge MA, 1960 (11^{th} pr, Apr 1979), p 144.
[61] Robinson, J.A., "A Machine-oriented Logic Based on the Resolution Principle", *J ACM* **12**(1) (Jan 1965): 23–41.
[62] Rosenbloom, P.C., *The Elements of Mathematical Logic.* Dover, New York NY, 1950.
[63] Rosser, J.B., "Highlights of the History of the Lambda-Calculus", *Ann History Computing,* **6**(4) (Oct 1984): 337–349.
[64] Schönfinkel, M., "Über die Bausteine der mathematischen Logik", *Math Ann* **92** (1924): 305–316, (in German; nach einem Vortrag vor der Mathematischen Gesellschaft in Göttingen am 7. Dez. 1920; von H. Behmann für Veröffentlichung bearbeitet).
[65] Scott, D., "Some Philosophical Issues Concerning Theories of Combinators", in: C. Böhm (ed), *Lambda Calculus and Computer Science Theory.* (Proc of the Symposium, Rome, March 1975), LNCS # 37, Springer-Verlag, Berlin, 1975, pp 346–366.
[66] Stenlund, S., *Combinators, Lambda-terms and Proof Theory.* Reidel, Dordrecht, 1972.
[67] Turner, D.A., "A New Implementation Technique for Applicative Languages", *Softw Pract & Exp* **9**(1) (Jan 1979): 31–49.
[68] Turner, D.A., "Functional Programs as Executable Specifications", *Phil Tr R Soc Lond* **A 312** (1984): 363–388.
[69] Vinogradov, I.M., Hazewinkel, M., *Encyclopaedia of Mathematics.* (translated, and edited from the Russian: Soviet Mathematical Encyclopaedia), Kluwer, Dordrecht, 1987.
[70] Vree, W.G., *Design Considerations for a Parallel Reduction Machine.* Univ Amsterdam, PhD Thesis, Amsterdam, Dec 1989, x + 195 pp.

[71] Vuillemin, J., "Correct and Optimal Implementations of Recursion in a Simple Programming Language", *J Comp Syst Sci* **9**(3) (Dec 1974): 332–354.

[72] Wadsworth, C.P., *Semantics and Pragmatics of the Lambda Calculus.* Univ Oxford, PhD Thesis, Oxford, 1971.

[73] Wegner, P., "Classification in Object-Oriented Systems", *SIGPLAN Not* **21**(10) (Oct 1986): 173–182.

[74] Whitehead, A.N., Russell, B., *Principia Mathematica.* Vol I, Cambridge Univ Pr, Cambridge, 1913, (2^{nd} ed, p 665).

[75] Van Wijngaarden, A. *et al* (eds), *Revised Report on the Algorithmic Language ALGOL 68.* Springer-Verlag, Berlin, 1976.

[76] Youschkevitch, A.P., "The Concept of Function up to the Middle of the 19^{th} Century," *Arch Hist Exact Sciences* **16**(1) (1976): 37–85.

Chapter 4

Expanding Twentel, the User's View

In this Chapter we describe the solution to the Language Designers problem of Chapter 2 which we introduced in 2.2. The solution contains the insertion of a trapdoor construction in a functional language in order to link functional programs easily to the outside world. This Chapter deals with the architecture of that construction in the demonstration language Twentel. It is the part that the programmer sees and must use: he is the user in this Chapter, according to 2.4.3. Starting with simple applications of the trapdoor, we expand its applicability in other directions. The next Chapter treats the implementation of the trapdoor, and some parts of its realisation.

4.1 Embedding the Solution for User Problems in Twentel

In Chapter 2 we discussed a complex of software engineering problems and their environment. This led from a general language problem to exploring the expansion of a functional language, such that it could interact with the outside world without undue problems arising for the programmer. In order to become a better prototyping language then it is now, the functional language must be able to use other resources. Resources that are not part of the functional programming language environment itself, or that are too expensive to draw on when expressed in the language itself. Thus, the functional language has to be enriched with the possibility of using and communicating with these outside resources.

Now we are to give the solution to such an extension, which opens the way for the alleviation of the software engineering problems of the second Chapter. In order to demonstrate this solution we will expand the functional programming language Twentel in order to achieve a solution with the desired properties. As Twentel is the demonstration language we will use 'Twentel' if we mean a (generic) 'functional language', unless it is clear from the context that Twentel itself is meant.

Resources are not only static, like the run-of-the-mill classic I/O (e.g., terminal, printer and file), but there are also dynamic resources. Introducing dynamics, we had better use 'agent' instead of 'resource'. Dynamic agents can be functions, which are true computing agents or 'outside' programs, and database servers, which are functions defined on a set of data or database. In current terminology we speak about a 'server', with Twentel being the 'client'.

It is obvious that taking an existing language as a demonstration language imposes constraints upon the decisions concerning the design of this extension (2.4.1). Since it was a research

project, it sometimes happened that a solution was present before the problem was crystal clear. However, due to the overall philosophy of functional languages, the spirit of the solution was clear from the onset.

4.2 The Design of a Trapdoor

What kind of function are we looking for? What kind of semantics do we need if we want to expand Twentel with something that creates a gateway to the outside world without unduly sacrificing one of Twentel's main characteristics, its mathematical soundness, present in, a.o., the referential transparency? Twentel's main descriptive component — being descriptive it is a textual one — is the (predefined) function. One must look for a solution in that direction: in doing so, one stays within the framework visible for the user, the architecture of the language [9]. This has been the guiding principle behind our decisions: parsimony, not introducing new constructs if existing constructs can serve the desired purpose equally well.

4.2.1 General Introduction of the Trapdoor Concept

Staying within the architecture of a functional language to create a gateway to the outside world, amounts to creating a new function. Not all the (predefined) functions in Twentel have the same flavour. Some of them are ordinary computing agents (ADD, MUL, DIVIDE, AND, OR), or list-processing agents (NULL, ATOM, REV), all of them having arguments of fixed type. There are also functions which operate on anything, the combinators (`S, K, I, C`; 3.4.4.1). As the outside world cannot be pinned down on a certain description (which is a loose use of the *type* notion (3.1.7)) the function to be used should be a combinator (3.4.3). So Twentel's gateway to the outside world, the *trapdoor*[1] as we call it, is to be a combinator.

From a designer's point of view, this means defining a combinator (or more than one) that exhibits the desired behaviour. We do not want to introduce a different syntactic construction, which is another way of introducing this behaviour in Twentel, as we state that the generalised function call mechanism (a combinator) can implement the desired behaviour adequately.

The essence of the desired *functionality* is something that can be compared to a simple addition, a kind of computational resource. Also list handling can be seen as a functionality: there are a few primitives involved (e.g., CONS, CAR, CDR, EQ, and ATOM) and we can deal with lists without bothering about storage and how lists are represented internally. In this way we can look upon lists and list-handling as a very low-level Abstract Data Type (which encapsulates a garbage collector).

[1] *Trapdoor:* apart from being a pitfall, or hiding 'secret' passages, and having still other connotations (viz in cryptography) this word has a link to the 'trapping' of the application program instruction stream to call on the OS services. It is a command in symbolic assembly languages (e.g., the IBM/360 SVC, the PDP11 TRAP): "Trap instructions provide for calls to emulators, I/O monitors, debugging packages, and user-defined interpreters." (Digital Equipment Corp., *PDP-11 Processor Handbook*, Maynard MA, 1969, p 41). 'Bridge', 'gateway', 'link' might have been better words, but these too, had different meanings and 'trapdoor' already stuck. We used 'TRAP' in favour of the better known 'SVC', as it was pronounceable (see Footnote 5, Chapter 2).

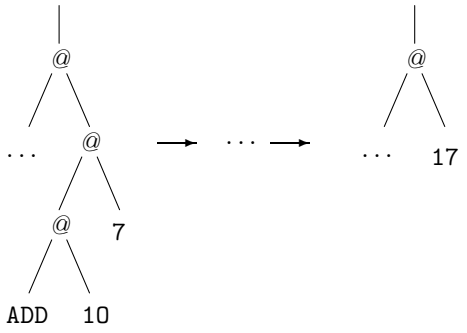

Figure 4.1: *Graph space states in the reduction of* '... (ADD 10 7) ...'

As we have seen (3.4.5), Twentel is defined on a graph rewriting machine, so from the Twentel text '... (ADD 10 7) ...' we get the resulting spine (3.4.5) in the graph space as shown in Figure 4.1, at the left.[2] After rewriting, this spine configuration yields the spine configuration as shown on the right side of Figure 4.1. In this sketch of Twentel's graph transformation we see not only rewriting and reduction at work, or more formally, not only the semantics of the abstract graph rewriting machine below Twentel (3.4.5). In this abstract machine there is also 'something' that, given the command ADD together with its arguments 7 and 10, yields 17 (the example in 3.4.5). We have seen this mechanism in the lambda calculus and also in combinatory logic, where this process, relying on primitive (arithmetic) notions, is known as *delta*-reduction (3.2.3). The supercombinator also displays this behaviour (3.4.4.3). Though the enriched lambda calculus is supposed to have these rewrite rules present — in order to 'close' the theory —, we cannot, being practical, pretend that they are all implemented in this way.

In the hardware deep down, beneath all the abstract machines we care to define, this is ultimately called an adder. The resulting value (17) was not present in the graph beforehand, nor was it present as an explicit rewriting rule of the form 'ADD 10 7 → 17' or something like 'APPLY (APPLY ADD 10) 7 → 17'. The required functional dependency is present, it is never defined, but at the hardware level and its immediate abstract machine above it (range, type of operands, accuracy *&c*), it is a kind of *Ur-primitivum*.[3] Of course we can define the addition operation in rewrite rules (SUCC, PRED, ZERO *&c*), but in practical situations this is never done: one relies on the *Ur-primitiva*. Walters, in his dissertation on implementing algebraic specifications [53], uses the same concept: his *Ur-primitiva* '+' (add) and '∗' (multiply) are expressed in a conventional programming language which is the level under his term rewriting system proper, instead of (with the natural numbers) using the constant z with a successor function s which maps a number on its successor, and the rules for addition and multiplication:

$$a(z,x) \to x, \quad a(s(x),y) \to a(x,s(y))$$
$$m(z,x) \to x, \quad m(s(x),y) \to a(m(x,y),y).$$

We now introduce a new abstract machine below the rewriting machine, called *functionality machine*. This machine incorporates all sorts of functionalities which can be thought of in the rewriting machine taking care of the *delta*-reductions. In this machine a functionality is a device which can be fully defined in a formal way. In this case it could be defined in rewriting rules. Defining it that way, it stays fully within the prevailing description method, or language, of

[2] The '@' denotes, as usual, an application node (3.4.5).

[3] As a construction comparable to the opposite of 'user-friendly': 'userfeindlich'.

the higher abstract machine, the rewriting machine. However, for all practical purposes, the functionality is implemented (realised) on a lower layer, where other description methods (can) prevail. The adder above is an example of such a functionality. It is practical to have only one functionality of each sort, it is not overly economical to have two adders: compare this with the one Arithmetic-and-Logic-Unit (ALU) in each uniprocessor.

Having defined a functionality machine, it is now possible to delegate work to one or more functionalities, work that

- is more efficiently executed in another elaboration regime, and / or
- is difficult to describe in a functional language and is easier to describe in another language.

The same holds for implementing a hardware design, as the isolation of parts of the solution of a problem to another level or medium, is a common design technique (2.4.2). In hardware one has the peripheral devices, each with its own controller. Take e.g., a PC. The CPU is not used to control the screen handling: that is delegated to the video controller. The only thing the CPU does, when the program instructions indicate such, is to specify an '`m`' on $(19, 78)$. How that '`m`' is generated, in what font, which mode (blinking, bold, reversed), how the pixels are switched, which colour, all is left to the video controller; there is no communication with the CPU in case of problems, conflicting video attributes, or an improper functioning controller.

4.2.2 The Architecture of a Trapdoor

Apparently, we can distinguish a number of different functionalities in Twentel's architecture, such as an arithmetical unit, a list handler, and screen, keyboard, and file I/O. We now state that the desired extension can be added to that abstract functionality machine. We use the term '*Outside*' in the following to denote the set of all functions that can be described in Twentel and that we want for some reason to be executed in the outside world. This means that they will be executed without lazy evaluation semantics. *Outside* is the (abstract) machine within the functionality machine, which embodies the desired external functionalities: *Outside* can be viewed as the set of these functionalities. Other sets are of course, the adder *cum suis*, and perhaps — with proper hardware — the primitive list functions (CONS, CAR, CDR, EQ and ATOM, or derivatives).

We need at least something like a combinator `O` (short for the functionality *Outside*, which has some known functional behaviour) which shows the behaviour, shown in Figure 4.2. The

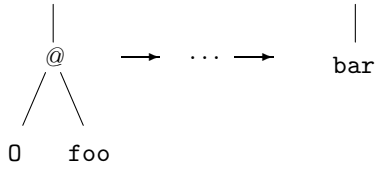

Figure 4.2: *Functional behaviour of the* Outside *combinator* `O`

application of `O` on some Twentel value `foo` can be described as follows. First the functionality `O` in *Outside* (in short *Outside*) receives its argument `foo` from Twentel. Then *Outside* produces in some way (that, for the time being, we assume to be known) its (function) value for `foo`,

say `bar`. This function value is then transferred back to Twentel and put into the graph space, taking into account the rewriting rule regarding the combinator `O` with one argument (3.4.5).

Why do we use a combinator, or rather, why this solution? Are more architectures feasible, and if so, why did we not choose one of those?

We have not chosen to add extra syntactic sugar, e.g., introducing a construction like

⟨ `Outside` ‖ `foo` ‖ `args` ⟩,

which resembles the construction which is used in many functional languages for the list comprehension (a kind of set description, 3.3.1): { ... | ... ; ...}. We rejected this way of solving the problem because it is contrary to our idea of expanding Twentel. The problem must be solved by enlarging the usability of Twentel by redesigning Twentel's inside, not by adding extra visible features that would also entail the difficult problem of making them orthogonal to existing ones. Something elaborate like pattern matching should have to be devised in that case, which next falls victim to Occam's Razor. Actions of this kind only pay off if there is a tremendous gain in combining previously different notions into one.[4]

The essential difference with other Twentel functions is the elaboration of part of the Twentel program outside the Twentel system itself. Thus, using supercombinators (3.4.4.3) within Twentel is not an alternative as supercombinators are generated based on the program. And the program is not present.

A solution of this kind already existed in the old, third generation procedural languages: linking an Assembler-, C-, FORTRAN- or LISP-subroutine with the main COBOL-, ALGOL-, C-, FORTRAN-, or Ada-program. This can only be done with full compatibility at the procedure-call level, and with full machinecode compatibility. However, as we want to abstract from the platform on which the *Outside* functionality is elaborated, so only the Twentel interface remains within the Twentel system, we cannot use this solution. We will use the trapdoor functionality of *Outside* in the functionality machine through a combinator.

Running Example As a running example to demonstrate the features of the trapdoor functionality, we take a non trivial problem: the generation and identification of planar graphs [24].

In the problem described in [13] we have a description of graphs. The idea in this publication is to generate new graphs from a base graph (in [13] this base graph is the Octahedron Graph), using a few easily described operations (a.o., Tutte's). The process of describing these operations can be implemented in an easy way in Twentel (an aspect of its 'easy-prototyping' property), but it is very cumbersome to program it in a third generation language (Pascal). When we automatically generate new graphs from old ones, it is necessary to distinguish the newly generated graphs from each other and from the old ones, in order to be able to determine the set of new graphs.

The nodes or vertices of a planar graph are numbered. Then the nodes in the simple cycles or faces in the graph are written down as lists, taken down when walking anti-clockwise. The last cycle is the one that presents itself when we embed the graph on a sphere (stretch the graph

[4] Apart from these aesthetical reasons, there was also the classic problem of a not quite fully documented Twentel system, especially in the parsing and code generation modules.

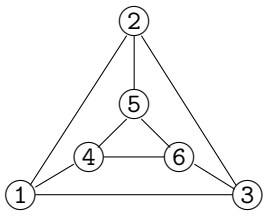

Figure 4.3: *A planar three-regular graph*

over it): it is the cycle at the back of the sphere, when taking an internal node as a point at the front. The graph is now completely described by the list of all its simple cycles, also called the 'code' of the graph.

To identify a graph we use the property that isomorphic graphs have equal identification numbers and non-isomorphic graphs have different identification numbers. The identification is done by a canonical reduction on the code (or representation) of the graph, and subsequently computing the identification number of the canonical representation. The identification number is conveniently represented as an octal number. This number has the property that the graph can be reconstructed from it, as it denotes an adjacency matrix[5] [25].

Take the planar three-regular graph from Figure 4.3 as base graph. In this graph, and with the anti-clockwise traversing the faces to write down the numbers, we distinguish the following four simple cycles: (1,4,5,2), (1,3,6,4), (2,5,6,3), and (4,6,5), with as cycle 'at the back' (1,2,3). This graph is now defined by the code ((1,4,5,2), (1,3,6,4), (2,5,6,3), (4,6,5), (1,2,3)). Therefore a Twentel activation of `graphidnr` to compute the graph's identification number would look like:

```
graphidnr [[1,4,5,2],[1,3,6,4],[2,5,6,3],[4,6,5],[1,2,3]]
```

resulting in the Twentel value `[71217]`.

This octal identification number can be shown to yield a graph, isomorphic with the input graph (thus exhibiting its identification property). Take an $n * n$ matrix, with n the number of vertices[6]: the adjacency matrix. In this case, $n = 6$. Fill the upper triangle (15 elements) sequentially, starting at the leftmost element, with the (binary) identification number (octal "71217" yielding: "111001010001111" (15 bits)).

	1	2	3	4	5	6
1	.	1	1	1	0	0
2	.	.	1	0	1	0
3	.	.	.	0	0	1
4	.	.	.	.	1	1
5	.	.	.	.	.	1
6	.	.	.	.	.	.

[5] From [44]: An adjacency matrix is a matrix representing a graph using the adjacency (two vertices are adjacent if there is an edge between them) of vertices: $a_{ij} = 1$ if (v_i, v_j) is an edge of the graph; note that an adjacency matrix of a graph is based on the ordering chosen for the vertices.

[6] It is not necessary to know the dimensions of the matrix, as a complete (all digits given) identification number, uniquely determines the number of vertices.

We see the following edges exist: (1,2), (1,3), (1,4), (2,3), (2,5), (3,6), (4,5), (4,6), and (5,6). With proper care these edges again result in the above graph.

Duijvestijn wrote this graph-identificator in Twentel as a proof of the algorithm being easily expressible in Twentel [26]. However, before this Twentel program was written, a Pascal version of the graph identification program already existed [25]. The reuse of this Pascal program in an easy way within a Twentel environment, would demonstrate the expandability of Twentel, especially with regard to the software engineering necessity of the reuse of programs.

4.2.3 Syntax and Semantics of the Trapdoor Activation

The syntax of the trapdoor activation, Twentel's interface to the outside world, is simple — we design it to be that way. The look of the interface at the other side of the trapdoor depends on the language chosen. Here we take the language Pascal [34] as demonstration vehicle. The concept of the interface itself, should be as simple as possible.

As we want the trapdoor concept to be simple and generally applicable, we cannot impose special constraints upon an *Outside* computational process. So, for example, we cannot demand a lazy evaluation regime for *Outside*. That would exclude any third generation language to be used for *Outside*, not a very practical suggestion.

On the other hand, *Outside* must not communicate with Twentel if something is amiss. The only communication possibilities for *Outside* have to be the reception of its arguments and the transmission of its function value, no other. At the same time, Twentel can have no contact with *Outside* other than by means of its transferred arguments and the received function value. For if there is a communicating link between Twentel and *Outside*, it must be dealt with in both environments. In the Twentel environment this would mean that the link must be made visible, and, by making it visible, one must be able to address the link, which implies handling its states. This kind of handling the states is one of the things the programmer was freed of by introducing functional programming languages. Furthermore, it would reveal implementation details at the architectural level which is against the 'transparency' principle.

4.2.3.1 The Twentel Side

We propose[7] the trapdoor to be activated by a combinator, called `TRAP`, with two arguments. This combination elaborates to one Twentel value, such that:

```
TRAP  Ofun  Oargument  -->  FVal
```

Outside has to produce a value for the arguments Twentel supplies it with. Therefore *Outside* has to know what functional dependency should be activated. This dependency must be specified at the point of activation of the trapdoor. When this specification is given, the next question deals with the necessary arguments for the external function to be transported to *Outside*. What kind of arguments must be transferred, are there any constraints in choosing these arguments? In the following paragraphs we will discuss these questions.

[7] Design in language amounts to writing.

`Ofun` *Outside* needs a description of the required functional dependency, a 'description' of the function. As obvious, more is to be computed in the outside world than just one function, remember that *Outside* was defined to be the set of all functions which we wanted to compute outside the Twentel system (4.2.2), an Abstract Machine within the Functionality Machine beneath Twentel's Rewriting Machine. We introduce this functional description here as `TRAP`'s first argument, `Ofun`.

A few (theoretical) methods of specifying this functional dependency come to mind:

- We have already seen the lambda calculus, very sound theoretically, but not very practical as such, in the programming environment proper (think of the *delta*-reduction rules).
- The method of specifying it in Twentel is in harmony with the programming environment, but it is not applicable as the essence of the trapdoor is to elaborate something *outside* Twentel.
- Using another programming language as specification vehicle is *the* solution as we have already seen. Specifying the problem to be solved within the Twentel code, in source form, means transporting this dependency description in full to *Outside*, where it has to be elaborated together with the arguments. A nice theoretical solution (*Outside* tends to become a universal Turing machine), but not practical. However, there will turn out to be an escape for this method: if *Outside* is an interpretive environment (a very strong requirement for a production *Outside*), this solution is feasible.
- Another solution is a name which denotes or refers to the required dependency. This dependency must be present in an executable form. Linking the name with the appropriate dependency must be solved by an agent outside Twentel.

For practical reasons we choose the fourth alternative. This name, the first argument denoting the functionality to be executed outside the Twentel program, is a Twentel string, or it must reduce to one.

`Oargument` The to-be-activated *Outside* needs the necessary arguments for computing the function value, and all of the arguments, as *Outside* can only be activated iff it receives all its arguments. *Outside* cannot ask for a missing argument as there is no communication between Twentel and *Outside*, except for arguments and function values (4.2.3). Twentel does not have a vehicle for function calls with an indefinite or variable number of arguments: every function (or combinator) has a fixed number of arguments. However, we do not want to restrict the programmer of an *Outside* functionality in choosing the number of arguments, if that can be helped. If the function that *Outside* embodies needs more than one argument, all these necessary arguments have to be curried into one argument.

We take `TRAP`'s second argument, `Oargument`, to be a list of lists of arguments which Twentel subsequently passes to *Outside*. The currying of arguments implies that if only one argument has to be passed to *Outside*, and there is only one activation of *Outside*, there is apparently one extra level of parentheses present.

As *Outside* needs all its arguments before it can start, it cannot start if (one of) its argument(s) is infinite. Therefore, the Twentel values which are compounded into `Oargument`, must be strict

and finite. There are methods for checking the strictness of arguments [41], but they have not been implemented in Twentel. No other constraints are put upon `Oargument`.

Trapdoor **Activation** In the next few examples of a trapdoor activation, we see the Twentel string, the first argument of `TRAP`, denoting the functionality to be executed outside the Twentel program. The second argument is a list of lists of arguments which Twentel passes to *Outside*, subsequently. Note the currying of arguments in the last example, as only one activation of `"AppendLists"` in *Outside* is to happen, so there is one extra level of parentheses present.

```
TRAP "Sine" [[90],[60],[45],[30],[0]]
TRAP "RootsQuadraticEqn" [[1,-4,4],[1,0,-9]]
TRAP "AppendLists" [[[k,l,m],[s,p,q,r]]]
```
□

A Complicating Factor A classic problem plays a role here too ([8], p 41). In Twentel we can define a function `f` with three arguments. In some cases it turns out in the actual application `f a b c`, that `f` does not depend on `b`: Twentel does not need to evaluate `b` to complete the evaluation of `f a b c`. This non-strict behaviour is allowed by the lazy evaluation mechanism of Twentel.

We cannot, however, migrate this function `f` to *Outside*, as we cannot create a computable *Outside* function `f`. Because, if we want to compute its function value, `TRAP "f" [[a,b,c]]`, Twentel has to give *Outside* all three arguments evaluated, as the use of *Outside* enforces strict behaviour. But in this particular case, Twentel does not know that the value of `b` is irrelevant for the computation and evaluates `b` to $\perp$ (known as *bottom*, a theory-based convention due to Scott [46]: computations that fail to terminate are considered to yield a special value. In cases like this, it signifies an error condition). Twentel raises an error condition, and returns with $\perp$ as result of trying to evaluate the argument list for *Outside* `f` in `TRAP "f" [[a,b,c]]`. As a result of $\perp$ in the Twentel reduction cycle, the system halts. Due to this halting, Twentel can never activate *Outside*, so `f` cannot be computed under Twentel's reduction regime through a trapdoor. *Outside* computed functions are strict functions, so there exist functions which cannot be exported to *Outside* as their arguments must be fully evaluated before passing them on to *Outside*.

A practical problem, not of theoretical interest as the one above, presents itself if a very large argument has to be passed to *Outside*. If the evaluation of this argument in Twentel causes a depletion of system resources (e.g., a stack or heap overflow) it cannot be computed either. However, in such a case, redesigning the Twentel solution (e.g., splitting the argument), and the *Outside* function it was meant for, can solve the problem.

`FVal` `FVal` is the function value, which *Outside* returns to Twentel as a result of processing `Ofun` with the argument(s) taken from `Oargument`. Contrary to the input arguments, this value *can* be infinite (a 'stream', 3.1.3.2) as Twentel can handle infinite values through its lazy evaluation regime. So large amounts of data can be transferred to Twentel, as long as *Outside* can send such a stream of data. The `TRAP` activations from 4.2.3.1 result, with the appropriate `Ofun` in *Outside*, in:

```
TRAP "Sine" [[90],[60],[45],[30],[0]] --> [1, 1/2√3, 1/2√2, 1/2, 0]
TRAP "RootsQuadraticEqn" [[1,-4,4],[1,0,-9]] --> [[2,2], [3,-3]]
TRAP "AppendLists" [[[k,l,m],[s,p,q,r]]] --> [[k,l,m,s,p,q,r]]
```

If *Outside* raises an error condition (e.g., upon having to divide by an argument which is zero) and it can recover from that condition, the `FVal` to be sent to Twentel, is $\perp$. If however, in the course of evaluation Twentel encounters this value at the tip of the spine, it will raise an error condition, and, because of this condition, the system halts. It is not certain that this will happen, as the evaluation regime is lazy.

Running Example If there is an *Outside* process that computes the graph identification number, named `"GraphIdNr"`, the *Outside* Graph Identificator can now be activated as follows:

```
TRAP "GraphIdNr" [[1,4,5,2],[1,3,6,4],[2,5,6,3],[4,6,5],[1,2,3]]
```

This activation of the graph identificator with a description of the graph as a list of lists (the cycles) results in `[71217]`. It is the straightforward way of calling an *Outside* functionality with the same interface as the corresponding Twentel function would have.

As a matter of fact, the actual activation of the graph identificator that is at the root of this example, is different from the one above. This is the result of linking to a Pascal (or other third generation language) implementation of this functionality in *Outside*, which will be explained in Chapter 5. In coding the graph, we complete a cycle by noting the return to its origin, upon which the end of the cycle is denoted by a '0'. The full code ends with an extra '0'. In this way it is a one-dimensional code that can be properly handled by such an implementation:

```
TRAP "GraphIdNr" [[1,4,5,2,1,0,1,3,6,4,1,0,2,5,6,3,2,0,4,6,5,4,0,1,2,3,1,0,0]]
```

and which activation once more results in `[71217]`.

4.2.3.2 The Trapdoor In Between

Seen from an architectural point of view, following Blaauw and Brooks (2.4.3), we do not have to define a process between Twentel and Pascal, as the immediate user cannot observe its behaviour, only its effect at the Twentel and the Pascal side. The agent that finds the right functionality (`Ofun`) and that transfers `Oargument` to *Outside* and `FVal` back to Twentel (4.2.2), can be thought of as residing in the Twentel and / or the Pascal environment.

4.2.3.3 The Pascal Side

At the Twentel side, the trapdoor activation is part of the elaboration of a Twentel program. In *Outside* it is the whole program that has to be activated. The general model is given by the following ALGOL-like description:

```
begin {The Main Outside Program}
      Outsideprocess (Input_Arguments, Output_FunctionValue);
end;
```

Having chosen Pascal as the demonstration language, we introduce the following description model for the main *Outside* program:

```
begin type Inputrecord; {the Pascal representation of the input:
                         Twentel's output}
      type Outputrecord; {the Pascal representation of the output:
                          Twentel's input}
      var Irecord : Inputrecord;
      var Orecord : Outputrecord;
      begin
         Outsideprocess (Irecord, Orecord);
      end;
end.
```

The *Outside* functionality is implemented as a Pascal program with its Pascal environment. The Twentel values transferred by the trapdoor to *Outside* are converted to an (input) record for this Pascal program. Processing this input record by the Pascal program yields a (function) value in the form of another record which leaves the Pascal environment. In leaving the Pascal environment, it is converted to values that are sent to Twentel. Here these are converted to a Twentel value, which is subsequently attached to its appropriate node in the graph space. We define the trapdoor mechanism to cover all these actions, except for the computation of `FVal` on basis of `Oargument` by the functionality `Ofun`.

From the free format list of arguments for *Outside*, which is `Oargument`, and the fixed Pascal `Inputrecord` format for `Irecord`, it is evident that some conversion has to take place. The immediate user can observe this behaviour: he specifies the (list of) argument values in the Twentel program, and he has to write the Pascal program mentioned above with its input record, and these two entities do not match readily. The architectural description of this process — the trapdoor mechanism — is, that the Pascal system that encompasses `Outsideprocess`, contains the necessary intelligence to be able to convert `Oargument` to `Irecord`, and, likewise, `Orecord`, to its Twentel counterpart `FVal`.

4.3 Using the Trapdoor

In the previous Section we have seen the fairly straightforward use of the trapdoor from within Twentel. In this Section, and the next, the need for different views of the trapdoor will become apparent. In the examples of the previous Section, the trapdoor was used as designed: Twentel value to *Outside* (output), and return of computed value to Twentel (input). In this 'duplex' way, more examples will be shown here. Due to the 'generality' principle, one might ask whether `TRAP` must have 'bundled' I/O, or rather O/I. The need for 'unbundling' will be demonstrated in the next Section.

4.3.1 Vectors, and Languages

Suppose in the outside world we have a functionality that inverts matrices, e.g., as part of a complete APL-system, a system handling all kinds of array- and matrix operations. Using such a functionality is clearly a matter of efficiency now. Not so much the notational efficiency, as the matrix inversion can be written down very easily in Twentel too, but the computational efficiency: a $500 * 1000$ matrix operation executes faster within the APL environment, geared to such operations, then in a Twentel environment. The transposition or inversion of a Twentel matrix (a list of equal length lists) proceeds with the following trapdoor activations:

```
transpose matrix = TRAP "transpose" [[matrix]]
invert matrix = TRAP "invert" [[matrix]]
```

The APL-system needs also a conversion process which takes the transferred Twentel list of lists, and converts it into a matrix. Then it is passed to a user-written program that inverts the matrix. The inverted matrix is taken, converted by another process into a list of lists and transferred back to Twentel.

In this example `matrix` comes from Twentel, it is transposed or inverted in the APL system, and the result, `transposedmatrix`, or `invertedmatrix`, is returned to Twentel. APL has been used as Twentel's Matrix Processing Unit through the use of a trapdoor.

However, taking into account Blaauw's generality of design (2.4.4) and the fact that many APL-systems are interpretive, and supposing we have such a system, we might also use the whole APL-system and send it more elaborate operations to be performed on the data. In that case `Ofun` (now to be taken as the whole `"APL-processor"`) denominates a set of executable functions, instead of one simple function call. The requested operation on the data (the APL statement(s)) must be transferred too, as part of `Oargument`.

```
DEF
    transpose matrix = TRAP "APL-processor" [["⍉", matrix]],
    invert matrix    = TRAP "APL-processor" [["⌹", matrix]]
FED
```

The possibility of this construction invalidates to a certain extent the third argument from 4.2.3.1. Because of the interpretive nature of our APL-system, we *do* transfer the functional dependency description to *Outside*. But it would be too stringent a requirement to state: "*Outside* functionalities can only be implemented on interpreter based systems", so we reject it.

4.3.2 Relational Databases

The current trend concerning combining programming languages and databases, is towards unifying the database functions into the languages [16, 17, 47]. We firmly believe that the use of very general, but standard, building blocks (like Stoye propagates in his dissertation [51]) is a more feasible way of implementing useful features in a (functional) programming language.

As regards the modern, relational algebra or calculus based, relational database [21], three possible courses are open for enlarging the possibilities of the systems based upon these databases:

- facilitating prototyping for administrative (and other) purposes, based on the languages used in the relational database systems (e.g., based on SQL);

- expanding a (functional) programming language with the appropriate data types and the operations thereon (which means expanding them with abstract data types of the relational database);

- expanding a functional programming language with a method of establishing a connection to the outside world, that incorporates database servers and other functionalities.

The first course will be too constricted because of the characteristics of the query languages in use: they are not fit for prototyping other applications than pure data based ones.

The second course puts a tremendous burden on the language designer: retrofitting a general purpose language with special purpose abstract data types might very well clash with the principles of orthogonality, generality, and harmony (2.4.4).

In order to demonstrate the third point, as we consider this route the most promising one, we first look at a relational database to see what it is like. Staying close to the subject of this dissertation, we will use a Twentel prototype of such a database.

The reader who wants to skip this Subsection is at liberty to do so. In this Subsection we briefly look at the database literature in connection with functional programming. Next we demonstrate the connection of a Twentel program with a relational database management system through the trapdoor. The (partial) programs will be annotated, but, we are afraid, not to everybody's needs. However, we hope these programs will show, once again, the ease of writing and understanding functional programs. One thing should be made clear, one should *not* try to understand the way the computation of the result takes place. One must understand the various 'equations'.[8] The system then resolves them to the desired result.

4.3.2.1 Functional Languages and Relational Databases

Many publications on the practical side of functional programming deal only with algorithmic and prototyping problems and very few with the integration of databases into a functional programming environment [18,38]. In the mid-seventies the functional data model was proposed as a generalised data system, i.e., a meta-data model able to cover the three dominant data models, relational, network and hierarchical [4, 37, 48]. The functional model is sufficiently abstract (directed graph with sets as vertices (entity and value sets) and edges as total functions) to be modelled on a functional language, but it has no connection to the Functional Paradigm as it originated in 1977. After this and in the early eighties a few papers in the database world emerged, exploring the link between functional languages and databases, the link indicated by the functional data model. The most important studies relevant for our research are those of Peter Buneman *et al* and David Shipman.

[8] Contrary to the most profound imperative language, as Van Wijngaarden was heard to quip: "ALGOL68 is just like the Bible, it is to be interpreted, not to be understood." (cf The Revised Report [55]).

FQL Buneman, Frankel and Nikhil developed the Functional Query Language (FQL) for querying large databases [16, 17, 19, 27]. This language is based on the application of functions to streams of entities (records or tuples, (key) attribute values (basic data types including strings)). The notion 'stream' is taken from functional programming (3.1.3.2), so FQL can handle large databases with small core consumption and does not have to resort to intermediate files if the operations involved do not force the system to do so (e.g., sorting). It also highlighted the fact that databases can be seen as functions *in extenso* (3.2.2).

The 'function' notion in this context has a strong resemblance to the name of an attribute of a relation: one of the properties of a tuple entity. By applying the function to a set of tuples one gets the wanted set of property or attribute values and by applying a function to a set of (key) values one gets the set of tuples that match that set of values.

FQL resembles strongly Backus' FP system ([5]). The authors classify its syntax as, "[...] not ideal for anyone except, perhaps, mathematicians." It is based on database- and built-in functions that are combined by a few operators to form new functions and queries. FQL is a query language and not a full data manipulation language, because, so the authors state, as a query language it is applicative: there is no need for an assignment or an update operation.

An example from [17]: Take the sequence of all EMPLOYEE's and create a sequence of their NAME's. In FQL this is written as:

! EMPLOYEE ∘ ∗ NAME;

A short explanation is called for:

- EMPLOYEE is the name of a relation in the database;
- '!' is an operator over a relation name that creates a function that generates all tuples of the relation (resembling a generator in a list comprehension);
- '∘' is Backus' composition (3.1.3.1);
- NAME is a function (linked to the attribute 'name' of the relation 'employee') that takes a tuple as argument and produces its tuple value (here a character string, the employee's name) (resembling an indexing operation on a tuple);
- '∗' is a higher order operator comparable to LISP's MAPCAR or Backus' apply-to-all (α).
- the 'reverse Polish' notation stems from the deliberate choice to have the order of terms in database queries correspond to the database access path (something which should be left to the 'transformer' of the query).

In Twentel, with appropriate database 'functions' (NAME and EMPLOYEE), this might look like:

{ NAME `emp | emp <- !` EMPLOYEE } □

DAPLEX Shipman in his functional data model ([47]) also uses the notions of an entity and of a function that maps (a set of) entities onto a(nother) set of entities (Shipman does not support the 'stream', so he can not use lazy evaluation). The language that Shipman developed

for manipulating databases (DAPLEX) is, like SQL, based upon the predicate calculus, rather than on algebra as an applicative language should be. Nevertheless, it is also strongly dependent on the composition of functions that produce a set as a result of a query. Shipman combines the power of functional composition with the data description facilities of database languages in which he incorporates the notion of 'semantic net' from artificial intelligence ([56]).

Shipman's goal was to create a 'conceptually natural' database interface language, much like SQL. So the following DAPLEX example needs no further explanation: "What are the names of all students taking electrical engineering courses from assistant professors?"

```
FOR EACH Student
    SUCH THAT FOR SOME Course(Student)
         Name (Dept (Course)) = "EE" AND
         Rank (Instructor (Course)) = "ASSISTANT PROFESSOR"
    PRINT Name (Student)
```

Strictly FP Bossi and Ghezzi ([15]) take the original FP of Backus ([5]) to show that it can be used as a *formal* specification language for queries in a relational database and for formal reasoning about those queries. They did not have the purpose of creating a real user-oriented query language. Within the definition of an FP-system for a relational DBMS they give FP-forms for the following relational operations: union, difference, Cartesian product, selection and projection. They too, use FP only as a query language and thus do not have a data manipulation language. We mention this paper because it is exemplary in its very strict FP-style that is not conducive to adopting, if not in the functional world (3.1.3.1), definitely not in the database world.

For obvious reasons we did not actively cover the late eighties and the nineties in our search, so the tentative remark that these studies have found not much follow-up may be completely off the mark. We base this remark on the programming style used in FQL; 'tentative', because of DAPLEX being sufficiently like SQL, and DAPLEX has offspring: ADAPLEX. However, the seeds for fruitful cooperation are present, especially in the line of Buneman [20]. As the languages mentioned above are more or less database languages, not general purpose languages with full database access, we do not pursue this line further, as the latter is what we are after.

4.3.2.2 A Relational Database in Twentel

The environment of relational databases and their query systems can be understood as a 'stream processor'. A 'stream processor' is a function that maps an n-tuple of sequences (input) to an m-tuple of sequences (output).

The state of a system, `s`, may change with every input event from the input stream, so the state transition function, `next`, maps a state and an input event on a subsequent state. We can define a function `states`, that maps an initial state, s_0, and a sequence of input events (the stream `I`), to a sequence of successive states [35].

```
states (s0, i :  I) = s0 :  states (next (s0, i), I)
```

A model of a relational database with its query processor can now aptly be described by adapting Joosten's 'stream processor equation' with some renaming [35]:

```
sys (rdb, qi :  QI) = out (rdb, qi) :  sys (next (rdb, qi), QI)
```

In this equation, we have:

sys : the complete description of the database environment;

rdb : the database;

(qi : QI) : the input stream of queries, with `qi` the query to be processed at the head of the stream, and `QI` the pending queries in the stream;

next : the state transition function. It is the part of the query processor that based on the input message `qi` and the database or system state `rdb`, yields the (modified) database for the subsequent queries;

out : the generator of an output message in response to the input message, `qi`, in connection with the database, `rdb`, at that moment. It is the other part of the query processor, which yields the answer to the query `qi`.

Based on this general description we can build a relational database in a functional language. Steutel and Van Zonneveld did so, and used Twentel as a prototyping vehicle. They constructed a Relational Database Management System (RDBMS) [50], together with a sufficiently powerful dialect of SQL [33]. The following programs, or sets of Twentel definitions, heavily lean on the original Steutel-Van Zonneveld prototype. First we present the definition of the database used and questions asked in these experiments and subsequently an outline of this stand-alone RDBMS.

The Relational Database in the Experiment We present the Twentel definition of the relations of the simple model relational database to be used in the experiment, together with a few sample tuples from the relations, which comprise the database.

First the definition of the database `datadef`, consisting of three relations: `"s"` the supplier relation, with `r1` a few sample tuples; `"p"` the parts relation, with `r2` giving samples; `"sp"` the supplier / parts relation, and `r3` the sample tuples.

```
DEF
    datadef = [ ["s" , ["snr", "sname", "status", "city"]          ],
                ["p" , ["pnr", "pname", "color",  "weight", "city" ]],
                ["sp", ["snr", "pnr",   "qty"]                     ] ],

    r1 = [ ["s1", "smith", 20, "london"], ["s2", "jones", 10,  "paris"] ],

    r2 = [ ["p1",   "nut",   "red", 12, "london"],
           ["p2",  "bolt", "green", 17,  "paris"] ],

    r3 = [ ["s1", "p1", 300], ["s1", "p2", 200] ]
FED
```

Questions Asked in the Experiment Some of the SQL statements that are used in testing the SQL interpreter stand-alone or through the trapdoor connection, are:

```
SELECT * FROM s WHERE (status > 30 AND city <> 'paris');
SELECT * FROM s WHERE (status < 30) AND (city = 'rome' OR city = 'oslo');
INSERT INTO s VALUES ('sh','terstede',11,'leiden');
UPDATE s SET status = 22 WHERE (city = 'leiden' AND status < 35);
SELECT s.sname,s.city FROM s WHERE city <> 'leiden';
genoeg;
SELECT * FROM sp WHERE pnr > 'p5';
INSERT INTO s VALUES ('sw','pipeline',10,'leiden');
SELECT * FROM s,snr WHERE s.snr=sp.snr AND s.city='leiden';
SELECT * FROM s,p WHERE s.city=p.city AND s.city='leiden';
```

The test suite closes with the '`EXIT;`' command. Simple SQL statements as used in the following paragraphs, are also included in this test suite.

The Stand-Alone Twentel Prototype In this prototype of an RDBMS, normal SQL statements (the input stream of queries) are typed in or read from a file (in the following Twentel definitions, the `INP` argument to `use_db`). They are passed to a syntax checker and, if syntactically correct, a semantic check is performed on them (e.g., all used attribute and relation identifiers correct and consistent?). Then the SQL statement is 'executed', which either results in a changed database (INSERT, UPDATE, DELETE) or in a relation, presented at the keyboard (SELECT) (the two parts of the query processor in the 'stream processor equation' of 4.3.2.2).

The user interface `use_db` for the SQL interpreter is the trigger for Twentel reductions. Activating it results in a triple: text to user on screen, a new — formatted — database, and a file in which the input lines to the system are logged. (`use_db` is comparable to `sys` from 4.3.2.2.)

```
DEF
    use_db INP initDB
    = [ text_to_user           >> KB >> ,
        db_out >> FORMAT >> .. >> "dbfile.out" >> ,
        inputlines             >> "dbfile.inp" >> ]
      WHERE [rstmsg, db_out, rstlin] = Apply_Stream initDB (InLines INP),
            text_to_user = start_mess ++ rstmsg,
            inputlines   = start_mess ++ rstlin,
            start_mess   = "Stand alone SQL Interpreter" ++ [CR,NL,NL,'>,SP]
      ENDWH
FED
```

The main interpreter engine is `Apply_Stream` appearing in the user interface above. This function given a stream of queries, `Stream`, and a database, `DB`, processes a query on the database and results in a response to the user and in a database to be used in the next query. (`Apply_Stream` is comparable to `next` from 4.3.2.2; likewise `Stream` to `QI` and `DB` to `rdb`.) Actually a logging of queries (the third element of `Apply_Stream`'s result list) is present too, but this is not relevant for the functioning, only for bookkeeping purposes. It is not a coincidence that `Apply_Stream` and `use_db` have almost the same function value.

```
DEF
    Apply_Stream DB Stream
    = IF quit_comm
         -> [ end_message , DB , end_message ],
         -> [ indents ++ response ++ prompt ++ next_resp ,
              next_DB ,
              comm ++ prompt ++ next_comm ]
      FI
      WHERE
        [indents, comm:SeqStream]         = ReadCommand Stream,
        [e_syn, call_SemChk]              = parse (scan comm),
        [e_sem, QargL]                    = call_SemChk datadef,
        [exe_resp, resDB]                 = Query_on_DB QargL DB,
        [next_resp, next_DB, next_comm]   = Apply_Stream newDB SeqStream,

        datadef                = HD DB,
        quit_comm              = HD Stream = "EXIT;",
        prompt                 = NL ++ "> ",
        end_message            = "End of Session" : NL,
        [response, newDB]
        = IF INHAB e_syn -> [ListErr "syntax"   e_syn , DB    ],
             INHAB e_sem -> [ListErr "semantic" e_sem , DB    ],
                         -> [exe_resp                 , resDB ] FI
      ENDWH
FED
```

The next function returns a response to the user, a simple "`Ok`", in case of a correct update on the database, or a more elaborate one, in case of a SELECT. The second value in its function value tuple is the new database (the updated database as a result of processing the SQL statement), or the old database (a SELECT has no effect on the database).

```
DEF
    E_Resp_NewDB s new_rel t DB
    = IF s = sel -> [NL ++ form [] 0 rel, DB],
                 -> ["Ok", Replace_in_DB t rel DB]
      FI
      WHERE rel = DelSupPrefix (new_rel stgD) ENDWH,
FED
```

The 'pretty-printing' of the resulting relations is done under `response` by using `form`. `form` is the function for formatting and printing of the transferred relation. One of its duties is to create the ASCII frame of `'-, '|` and `'+` around the relation entries on printout. Note in the following Twentel statements (and in subsequent paragraphs), that many of the '`++`' operations (the infix form of the `append` function) are used in formatting and pretty-printing output.

```
DEF
    form title i r =
         [strL i title ++ line SP (R r)]      ++ [space i ++ separate] ++
```

```
        {space i ++ line '| u | u <- rows r} ++ [space i ++ separate]
    WHERE
        line s u = s  : CAT {strR (m+1) x ++ SP : [s] |
                             [m, x] <- ZIP [lu, u]} ++ NL,
        separate = '+ : CAT {REPTD '- (m+2) ++ "+" | m <- lu} ++ NL,
        lu       = {REDUCE MAX {LEN c | c <- col} | col <- ZIP r}
    ENDWH
FED
```

The basic functions to get at the basic data structure of a relation or table are (needed by `form`):

```
DEF
    R    (attributes: entries) = attributes,
    rows (attributes: entries) = entries
FED
```

Finally, the interpreter is activated in the following `DO ... OD` context:

```
DO use_db << "query" << execdbx OD
```

The activating function is `use_db` which gets its SQL input from the file named `query` and has its database argument delivered by the database (-valued) delivering function `execdbx`.

Exemplary Output from the Experiment We present two examples from the pretty-printed output of the SQL interpreter, through `form`. The first one is printed, when the interpreter is activated by '`SELECT * FROM p;`' (note the columns, properly headed by their attribute names).

```
  pnr   pname    color   weight     city
+-----+-------+--------+--------+--------+
|  p1 |   nut |    red |     12 | london |
|  p2 |  bolt |  green |     17 |  paris |
|  p3 | screw |   blue |     17 |   rome |
|  p4 | screw |    red |     14 | london |
|  p5 |   cam |   blue |     12 |  paris |
|  p6 |   cog |    red |     19 | leiden |
|  p7 |  bolt | yellow |     11 | berlin |
|  p8 |   nut |  white |     18 | leiden |
|  p9 |   cog |   blue |     15 |   rome |
|  pa | crank |    red |     13 |   oslo |
+-----+-------+--------+--------+--------+
```

Another simple SQL statement, `SELECT * FROM s;` results in this printout:

```
  snr   sname   status     city
+-----+-------+--------+--------+
|  s1 | smith |     20 | london |
|  s2 | jones |     50 |  paris |
|  s3 | bloep |     30 | leiden |
|  s4 | clark |     20 | london |
|  s5 | hakon |     40 |   oslo |
```

```
|  s6 | friss |     10 |   oslo |
|  s7 | foret |     30 |  paris |
|  s8 | klaar |     20 | leiden |
|  s9 | verdi |     10 |   rome |
|  sa | weiss |     30 | berlin |
|  sb | janus |     10 |   rome |
|  sc | godel |     20 | berlin |
|  sd | havel |     30 |  praag |
+-----+-------+--------+--------+
```

4.3.2.3 The Experiment

Up till now everything was, in effect, a normal run of the Steutel-Van Zonneveld prototype RDBMS. This being a prototype, we could neglect (abstract from) the fact that the whole database consisted of no more than some forty tuples in the three relations. Which is not quite up to the normal gigabyte database of today's business. Implementing such a database in this way would create a serious problem, as the database has to be present *in toto* in the Twentel graph space, which space is strictly limited. This is an excellent reason for introducing the trapdoor. We can have a link to a normal-sized production database, and yet have a good prototyping language at our disposal.

In the general literature the 'client / server' notion has become the common denominator for function application in its widest sense in a decentralised computing environment. Ranging from standard print-, file-, and database servers (actually full-grown, former OS services, the OS itself becoming leaner) to fully custom-built services, it is the 'services' part which led us — in a natural way — to speak about a SQL server, instead of the more artificial SQL functionality.

In the first trapdoor experiment we make a character connection with an SQL server. This is a connection in which only ASCII characters are moved to and from the *Outside* functionality. Afterwards we will expand on the returned tuples, and show a basic data type connection with the SQL server (4.3.2.5).

In Figure 4.4 we sketch the used model of Twentel and its database server. The Twentel system (left) is the master, and uses the stripped user interface of the prototype above. The SQL server (right) is a slave process with respect to the Twentel system. It contains the full database. In between are two directed channels. The upper one transmits a stream of characters from Twentel to *Outside* SQL server, in other words, sends queries to the SQL server (e.g., `SELECT * FROM partslist WHERE color = "yellow"`). The lower one likewise transmits a stream of characters, from the SQL server to Twentel, bearing the results of the query (e.g., `[ p7, bolt, yellow, 11, berlin ]` (in this example we adapted the tuple transmission adapted for readability)).

4.3.2.4 Character Connection with a SQL Server

The normal SQL statements that are typed in or read from a file at the Twentel side by the master program are transmitted to the *Outside* server. The request — a SQL statement — is handled by the server, and its response in pure ASCII form (the actual response together with

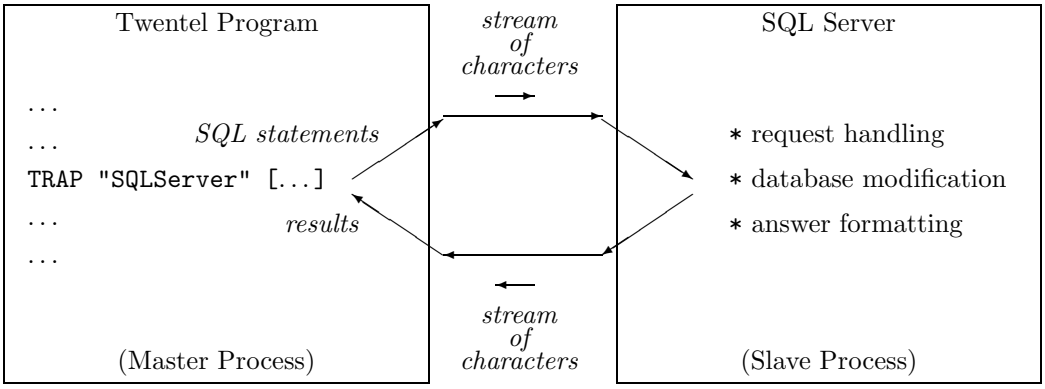

Figure 4.4: *A model of Twentel and its* SQL *database server*

the formatting padding (like `CR`, `NL`, `SP`, `|`, and `-`)) is displayed by the Twentel program without any further formatting.

The user interface, `use_db`, is changed into `MasterSQL`, which calls upon `Apply_Trap` instead of on `Apply_Stream` as in 4.3.2.2.

```
DEF
    MasterSQL INPComm
    = [text_to_user       >> KB >> ,
       inputlines         >> (uniquefilename "Log") >> ]

     WHERE [rstmsg, rstlin]  = Apply_Trap (InLines << INPComm <<),
           text_to_user      = start_mess ++ rstmsg,
           inputlines        = start_mess ++ rstlin,
           start_mess        = NL ++ STIME ++ prompt ++
                               "Master Twentel SQL Interpreter, " ++
                               SDATE ++ " :" ++ NL ++ NL ++ STIME ++ prompt
     ENDWH
FED
```

`Apply_Trap` is the main `TRAP` engine of the master interpreter. It filters commands (or queries) out of the input stream, sends them over the trapdoor and receives a response, which is displayed as is. The main difference between this function and `Apply_Stream` (apart from the `TRAP` of course) is that this function does not have a database as input argument, nor as output argument: the database belongs to the abstract data type residing in the `SQLServer` in *Outside*.

```
DEF
    Apply_Trap Stream =
      IF mend_comm -> [ end_message,
                        comm ++ NL ++ STIME ++ prompt ++ end_message ],
         quit_comm -> [ resprompt ++ end_message, commlog ++ end_message ],
                   -> [ resprompt ++ next_resp,   commlog ++ next_comm   ]
      FI
      WHERE
```

```
        [indents, comm:SeqStream] = ReadCommand Stream,
        trap_in                   = IF quit_comm -> "EXIT;",
                                                 -> comm ++ ";" FI,
        [response]                = TRAP "SQLServer" [trap_in],
        resprompt                 = response ++ NL ++ STIME ++ prompt,
        commlog                   = comm ++ "-  S > " ++ resprompt,
        [next_resp, next_comm]    = Apply_Trap SeqStream,
        quit_comm                 = comm = "QUIT",    (* Both Closing *)
        mend_comm                 = comm = "EXIT",    (* Master End *)
        end_message               = "End of Session,  " ++ SDATE ++ ";"
      ENDWH
FED
```

`ReadCommand (line:L)` is the function that reads a command from its argument character stream, with the command terminated by a semicolon. When the command does not fit on one line, `ReadCommand` also supplies the proper indentation to be used in formatting; `prompt`, and `indnt` are self-explanatory functions.

The master SQL interpreter is activated as in the two following `DO ... OD` contexts, the first one with query input from keyboard, the second one with queries from a file `"qinput"`. Prior to this, the slave side must have become active, as the SQL server waits for queries to be processed on its database.

```
DO MasterSQL KB OD
DO MasterSQL "qinput" OD
```

As can be seen from the above `EXIT` and `QUIT` commands, the master side can suspend its activities, and resume querying at a later moment. If, however, that moment is not to come any more, proper procedure requires closing unused resources. So the slave side, still in query receiving mode, is sent the `EXIT` command through the trapdoor and closes down. Its response is displayed on the keyboard at the master side.

```
DO (TRAP "SQLServer"  [ "EXIT;" ])  >> KB >> OD
```

An example of the output of this experiment can be found in 4.4.2.2.

4.3.2.5 Basic Data Type Connection with a SQL Server

In the character connection case one has to accept the format the SQL server sets for formatting the output on a request. Also, the possibility of subsequent processing of the output is impeded by the pure ASCII format of the response. If the SQL server has the possibility of sending the response in list-like format with the basic data types in 'raw' format (e.g., integers and reals as 'themselves', in binary and not converted to characters) then the above Twentel program can be adjusted accordingly. In this case 'list-like' is an appropriate choice, as a table can easily be represented by a list (the table) of lists (the tuples).

The definitions of the previous master SQL interpreter (4.3.2.4) are the same. The difference lies in the handling of the response from the server in `trap-result`. The server response has to be

handled differently from the previous `response`: the latter was the full, textual, response, and did not have to be formatted for proper printing, which the new response needs in order to be intelligible. In this program section, `ReadCommand (line:L)`, `prompt`, `indnt`, and `form` are the same as before in 4.3.2.2.

```
DEF ...
    [trap_result] = TRAP "SQLServer" [trap_in],
    trap_typ      = HD (HD trap_result),
    trap_out      = HD (TL (HD trap_result)),
    response      = IF trap_typ <> "REL"
                        -> IF size trap_out < 65 -> trap_out,
                                                 -> NL ++ trap_out FI,
                        -> NL ++ form [] 0 trap_out FI,
    ...
FED
```

4.3.2.6 Discussion

In the above demonstration we 'reconciled' a programming language and a database system. It was necessary to do so because the programming language system could not provide for a normal sized database, which is essential if one wants the language to be able to work with the database. When 'reconciling programming languages and databases' Peter Buneman remarked [18]: "Although several attempts have been made to produce a cleanly integrated programming language and database management system, in many cases the interface between a high-level programming language and a database is clumsy, low level and system dependent". Though 'system dependent' is not quite unnatural, we hold that 'raw' list format is not clumsy or low level, but basic, so we succeeded in producing a cleanly integrated programming language and (R)DBMS.

We demonstrated the possibility of linking a Twentel program to a potentially gigabyte database, provided an *Outside* SQL server accommodating the 'raw' format, is present. By the concept of the trapdoor Twentel brings the full power of the lambda calculus into the traditional 'data-processing' shop. So easy prototyping, not impeded by the shortcomings of SQL, becomes possible now: "... most of these [query] languages, while being extremely convenient for end-user interaction, lack the power of conventional programming languages to perform arbitrary computation or calls to the operating system" [18]. The most obvious omission in handling queries in this respect, is the transitive closure [2]. One needs this operator with tables consisting of begin- and end-points of roads, with some roads connected (a description of a planar graph), and then querying for *all* roads leading from London to Edinburgh, or any other place on that route, that are (in some cases not directly) connected, so valid answers must include bypasses.

The theoretical structure of the database management system also is a firm one [21], though it could be made more general, as we saw above. Buneman states: "The relational algebra is unique in presenting a database management system as an abstract data type. The plethora of 'semantic' and 'knowledge based' data models [...] have somehow missed the message of abstract data types that an 'abstract' or 'conceptual' data model can only be formalized by

defining operations upon it" [18]. Together with the likewise mathematical structure of functional languages, it should be possible — the functional language can be the preprocessor of the queries if it has an accurate description of the data model of the relational database — to get an even more powerful combination through the combination of rewriting and optimisation (e.g., [10–12]).

We modularised the system into a 'calculator' (Twentel) and a 'data administrator' (SQL server), so the following problems will not arise in this system (or programming language) as they are fully encapsulated in the DBMS. "The differences between programming languages and databases involve abstraction, integrity and access methods. The database model takes atomicity (serialisation), sharing, integrity (semantic sensibility), and recovery as fundamental requirements. This led to the transaction model which mandates a single granularity for all these responsibilities. Most programming languages have not addressed these requirements. Experience dictates that concurrency and atomicity must be designed into a system (or language design) from the start, not grafted on later" [38]. This also strengthens our case for reuse and extensions to the outside world, as putting these characteristics into a language (thereby only serving the database community, not others, like the real-time programmers) implies a new design, not a redesign. And moreover, the solution is present, so why bother our lazy evaluation regime with the atomicity and integrity of data and operations: how to match these with laziness is an interesting, albeit academic, question.

4.4 A Non-atomic Trapdoor

Reasoning from the notion of function, it is evident that an application of `TRAP` to some arguments, must yield a value: input and output are not divisible. Seeing it as a black box however, a physical metaphor, with an input side and an output side, and ignoring the functional dependency for the time being, one might ask whether input and output side cannot be manipulated, why can't they be switched, why can't the black box be taken apart? So though the concept of the trapdoor is an indivisible (atomic) functionality, after the design has been realised one can split the building blocks from the compound.

Why should one want to split the atom? It is obvious that if *Outside* can be practically any computing agent, a Twentel system can be one of them. But a slave system requires arguments to start its computations, so it waits for these to appear at its input. There are no possibilities in Twentel to connect to the outside world with the functionality of the trapdoor, if one needs just reading or writing: this could be done — with loss of functionality — by means of normal I/O (file, keyboard). The only possibility we created was a write–read (O/I) connection, called `TRAP`. In it we have a read-connection to *Outside*, but it is either the wrong way round, or it carries an unwanted 'write'. The bundling of this O/I, the duplex situation, was discussed in the previous Section, and will be undone in this Section. This results in two new 'simplex' manifestations of the trapdoor mechanism.

4.4.1 Definition of New Models of TRAP

Taking the building blocks of the full trapdoor apart, two distinct features manifest themselves as we have seen: the input side (of `FVal`), and the output side (of `Oargument`). The two 'simplex' definitions of the trapdoor are constructed upon the original 'duplex' definition of the `TRAP` combinator (4.2.3.1). The first one for catching input from the outside world, and importing it into Twentel, as `FVal`. The second one for delivering a Twentel value to the outside world, without waiting for an answer (reducing to the identity combinator, 3.4.4.1):

```
TRAP  "OSWIN"   Oargument --> ... -->  FVal
TRAP  "OSWOUT"  Oargument --> ... -->  I
```

These can be implemented by having two reserved *Outside* functionality names as `Ofun`, i.e., the Twentel strings `"OSWIN"` and `"OSWOUT"`, for 'Outside World Input' and 'Outside World Output'. It is evident that `Oargument` is a dummy argument in the `"OSWIN"` case.

These new models of `TRAP` are not readily usable in Twentel, if we use Twentel as a driving (master) programming system. Using them supposes another Twentel system with an active `TRAP` to *Outside* (or another system linking to Twentel, instead of the other way around). Urged by an uneasy feeling about the design of these functionalities we will reassess these two models of `TRAP` in the next Chapter (5.1.1.3).

4.4.2 A Relational Database Server in Twentel

In order to minimize the problems with implementing different environments and with connecting them afterwards (the 'availability' argument (2.2)), we took the Twentel RDBMS prototype implementation (4.3.2.2), and had it masquerade as a SQL server in the above examples. We now had a known and easily adaptable environment. The remaining problem of interconnecting the RDBMS with the Twentel master program was delegated to the trapdoor. With the two new models of the `TRAP` combinator, we can now easily adapt the Twentel RDBMS to a SQL server role.

4.4.2.1 The Slave Side Twentel Program

The trapdoor interface looks very similar to the user interface of the master (4.3.2.4) or stand-alone version (4.3.2.2). The construction with the list comprehension on the `"OSWIN"` trapdoor however, needs further explanation. It is necessary to give `OSW` a changing second argument. Because of lazy evaluation, a function with arguments, once evaluated, will never be evaluated again; so `OSW` will never get a new value. Though we feel this should be handled differently, in a more intuitive way, we are not quite sure how to do it consistently: *In dubio, abstine*.

```
DEF
    SlaveSQL
    = [ text_to_user         >> KB >>  ,
        db_out     >> FORMAT >> .. >>  (uniquefilename "RDBOut") >> ,
        inputlines                 >>  (uniquefilename "Log"   ) >> ]
```

```
      WHERE
            [rstmsg, db_out, rstlin] = Apply_Trapstream initDB OSW,

            initDB       = ReadSlaveRDB ^ 1,
            OSW          = { TRAP "OSWIN" [t] | t <- [0..123456] },
            text_to_user = start_mess ++ rstmsg,
            inputlines   = start_mess ++ rstlin,
            start_mess   = NL ++ STIME ++ prompt ++
                           "Slave Twentel SQL Interpreter,  " ++
                           SDATE ++ " :" ++ NL ++ NL ++ STIME ++ prompt
      ENDWH
FED
```

Much of the `Apply_Trapstream` is also very similar to the `Apply_Stream` and `Apply_Trap` of the master and stand-alone version.

```
DEF
    Apply_Trapstream DB OSW =
          IF quit_comm
             -> [ TRAP "OSWOUT" [["TXT", end_message]] ++
                  NL ++ STIME ++ prompt ++ end_message,
                  DB ,
                  comm ++ NL ++ STIME ++ prompt ++ end_message ],
             -> [ TRAP "OSWOUT" [response] ++ trap_typ ++
                  NL ++ STIME ++ prompt ++ next_resp,
                  next_DB ,
                  " M > " ++ comm ++ "; --> " ++ trap_typ ++ NL ++
                  loggingtxt ++ NL ++ STIME ++ prompt ++ next_comm ] FI
      WHERE
        ([[trapinput]] : OSWnxt)           = OSW,
        comm                               = ReadCommand trapinput,
        [e_syn, call_SemChk]               = parse (scan comm),
        [e_sem, QargL]                     = call_SemChk datadef,
        [exe_resp, resDB]                  = Query_on_DB QargL DB exe_sem,
        [next_resp, next_DB, next_comm]    = Apply_Trapstream newDB OSWnxt,

        [response, newDB] = IF INHAB e_syn -> [ListErr "syntax"   e_syn,DB],
                               INHAB e_sem -> [ListErr "semantic" e_sem,DB],
                                           -> [exe_resp,             resDB]
                            FI,
        datadef           = HD DB,
        loggingtxt        = IF trap_typ <> "REL" -> trap_out,
                                                 -> slaveform [] 0 trap_out
                            FI,
        trap_typ          = HD response,
        trap_out          = HD (TL response),
```

```
        quit_comm          = comm = "EXIT",
        end_message        = "End of Session,
                             " ++ SDATE ++ " - " ++ STIME ++ " ;"
      ENDWH
FED
```

In these equations, `ReadCommand trapinput` is the function that reads a command from the trapdoor, with the command terminated by a semicolon; `prompt`, and `ListErr` are self-explanatory functions; `form` is a dummy function for formatting of the total relation (which is done at the master side, as there the preparations for presenting the result takes place), and `slaveform` is the original `form` function for internal use in the total relation formatting for logging (see `form` in 4.3.2.2):

```
DEF
    form title i r = ["REL", r],
    slaveform title i r = ...
FED
```

Looking back on the two (three) places where `TRAP` occurs, one must conclude that it is not very consistent: there is no real *Outside* functionality called `"OSWOUT"`, it is just a token, signifying a special treatment from the *Outside* or trapdoor mechanism. In the next Chapter we will find a better solution for this predicament (5.1.1.3).

The slave side has to be active before the master side can send queries. This is done by the following `DO ... OD` context:

```
    DO SlaveSQL OD
```

which starts the slave Twentel system, which, after a few moments, is waiting for arguments to `Apply_Trapstream`.

4.4.2.2 Output from the Slave Side

The following transcript from an exemplary session (logged in the `"Log"`-file) with the relational database, has a three-in-one purpose:

- it shows the full logging of the slave side of the SQL interpreter (4.4.2.1);
- it shows (when leaving out `S >` and `M >`, together with changing `Slave` into `Stand alone`) the functioning of the stand-alone SQL interpreter (4.3.2.2);
- it shows the log of the input SQL statements (when dropping the `'--> REL'` (denoting 'result is a relation') and the `'--> TXT'` (denoting 'result is a message')),

and, *mutatis mutandis*, the functioning of the master SQL interpreter (4.3.2.4, and 4.3.2.5).

The session starts with the opening message (with date and time) and a simple SQL statement, resulting in a relation to be displayed with columns properly headed with their attribute names.

```
20:23:15 S > Slave Twentel SQL Interpreter,  11-Feb-92 :

20:23:15 S >  M > SELECT * FROM s; --> REL
  snr   sname   status     city
+-----+-------+--------+--------+
|  s1 | smith |     20 | london |
|  s3 | bloep |     30 | leiden |
|  s4 | clark |     20 | london |
|  s5 | hakon |     40 |   oslo |
|  s6 | friss |     10 |   oslo |
+-----+-------+--------+--------+
```

After the full relation, we present a selection on the same relation.[9]

```
20:23:54 S >  M > SELECT * FROM s WHERE (status > 30 AND
                                          city <> 'paris'); --> REL
  snr   sname   status     city
+-----+-------+--------+--------+
|  s5 | hakon |     40 |   oslo |
|  se | groot |     42 | leiden |
+-----+-------+--------+--------+
```

A write and an update in the 'suppliers' relation, followed by a selection, which shows the effect of the first two actions.

```
20:25:29 S >  M > INSERT INTO s VALUES ('sh','jetteke',11,'leiden'); --> TXT
Ok

20:26:05 S >  M > UPDATE s SET status = 22 WHERE
                    (city = 'leiden' AND status < 35); --> TXT
Ok

20:26:30 S >  M > SELECT * FROM s WHERE city = 'leiden'; --> REL
  snr       sname   status     city
+-----+-----------+--------+--------+
|  se |     groot |     42 | leiden |
|  s3 |     bloep |     22 | leiden |
|  s8 |     klaar |     22 | leiden |
|  sh |  terstede |     22 | leiden |
+-----+-----------+--------+--------+
```

A selection showing a restriction.

```
20:26:45 S >  M > SELECT s.sname,s.city FROM s WHERE city <> 'leiden'; --> REL
  sname     city
+-------+--------+
| smith | london |
| clark | london |
| hakon |   oslo |
| foret |  paris |
| godel | berlin |
| jones |  paris |
+-------+--------+
```

[9] Note the real time character of the test database; it has changed in the meantime.

An example of an incorrect SQL statement: The transcript shows text as a result of the activation, which text is the subsequent error message.

```
20:32:26 S >  M > genoeg; --> TXT
Err_syntax - Unknown keyword 'genoeg' compilation stopped
```

A join, properly headed.

```
20:35:23 S >  M > SELECT * FROM s,p WHERE s.city=p.city AND
                                          s.city='leiden'; --> REL
  pnr   pname   color   weight   p.city   snr       sname   status   s.city
+-----+-------+-------+--------+--------+-----+-----------+--------+--------+
|  p6 |   cog |   red |     19 | leiden |  se |     groot |     42 | leiden |
|  p6 |   cog |   red |     19 | leiden |  s3 |     bloep |     22 | leiden |
|  p6 |   cog |   red |     19 | leiden |  s8 |     klaar |     22 | leiden |
|  p6 |   cog |   red |     19 | leiden |  sh |  terstede |     22 | leiden |
|  p6 |   cog |   red |     19 | leiden |  sz |     groot |     44 | leiden |
|  p8 |   nut | white |     18 | leiden |  se |     groot |     42 | leiden |
|  p8 |   nut | white |     18 | leiden |  s3 |     bloep |     22 | leiden |
|  p8 |   nut | white |     18 | leiden |  sh |  terstede |     22 | leiden |
+-----+-------+-------+--------+--------+-----+-----------+--------+--------+
```

And the end of the session, brought about by the `EXIT` statement.

```
20:36:29 S > EXIT
20:38:01 S > End of Session,  11-Feb-92 - 20:38:00 ;
```

4.4.2.3 Discussion

We confine ourselves to a technical remark regarding the transferability of functions between platforms. General aspects (query languages, extensions) have been discussed in 4.3.2.6.

In this Subsection we saw the migration of one function (`form`) from one platform (slave) to the other (master). However, on the other hand we had the migration of one 'function' (`SQL`) from stand-alone to the slave side. Migration is easy as long as there is no hidden link behind the argument list, which — through referential transparency —is the case in functional languages.

4.4.3 Visual, or Human I/O

As soon as in the late fifties a trend showed — and is still manifest in the early nineties, and not diminishing — that I/O is not an orthogonal principle. FORTRAN and ALGOL bear witness to the fact that it is something else, something to be handled differently from other functions, like sine and cosine. Here we will show that orthogonality, or adhering to the language paradigm, pays off, without sacrificing functionality. Backus even disposes of I/O when discussing the concept of 'program' [6].

Perhaps this special attention paid to I/O stems from the classic Turing-Von Neumann computational model. The computer, as it is presented in our computing science courses, always has I/O attached, as it is a machine based on the Turing-Von Neumann model. Which brings us to

the observation that most of this concerns input. With respect to output, we distinguish normal output, output which has once been input in the same form (or a permutation thereof). Another kind is the output which is not 'directly' generated by input, output which adds something extra to everything, e.g., management information, accumulated, processed data, or derived output, e.g., based on inferring.

There is no I/O involved in an algebra exercise written down on a piece of paper. The observer sees the solution which is written down, at the same level as the unfolding of the argumentation in his thoughts and on paper. There is no explicit I/O present in this algebra exercise. So there should be no I/O present in our language, the vehicle with which we write algebraically about our problems to be solved. I/O in our language is something which must be pushed away to lower or other layers. There must be hooks, but they must not be 'visible'. Visibility means attention. If it does not require attention, then it must not be visible (see 2.4.4). If there is attention to be dealt with, this attention draws on the same resources as the problem to be solved: the programmer. He should keep himself busy with problem solving, not with housekeeping.

Experience shows that I/O with all its related processes (e.g., error processing), obstructs the view on the main process under consideration. This translates itself in diverting the attention of the programmer, and results in many statements spent on subordinate issues, something the philosophy of functional languages tries to avoid (e.g., trapping input errors, correcting input values, and handling screen layout: maintaining a screen layout is easier to do in a $80 * 25$ grid with the proper description tools, than in a functional language).

As we have now the `"OSWIN"` and `"OSWOUT"` trapdoors, we might dispatch this tedious I/O work to some other description environment. (Recall the large number of formatting and pretty-printing (parts of) statements in the examples of the SQL interpreter.)

As the consequence of such an idea, all basic data types (e.g., `integer, real`, being first class citizens, not only internal, but also external) must be output, and in the same way input again, (cf Marinda and LISP). Such a solution would be a much more elegant way for dealing with Twentel's I/O, without the artificial formatters and filters, especially regarding output. Input, as well as output, can now be described as a trapdoor operating on streams: be it an ASCII or integer stream, basic data types in any case. One can name such a formatter and refer to it in a Twentel trapdoor `Ofun` argument.

Agreeing with our idea of producing software as solving problems with as much reuse of existing solutions as is feasible, I-systems, and O-systems — each with their own description regimes — should be considered producer / consumer processes for the main process, the solution. These processes can be cascaded, the feasibility of which Joosten demonstrated [35]. One can classify these streams as follows:

- consumer: output, as produced by Twentel, cannot have anything but a deterministic value, possibly infinitely long;
- producer: input
  - is indeterministic, in time as well as in value, a kind of 'oracular' function.[10] Its function value at a certain moment cannot, for all practical purposes, be computed, it must be given (*in extenso*),

[10] This 'oracular' terminology is due to Gerrit van der Hoeven.

– has deterministic value, possibly infinitely long. This function is 'computable', or at least, it can be described *in intenso*; it might not halt, but it yields a stream of values.

4.4.4 File I/O and Data Transport

Apart from visual I/O there is also file I/O, and other data transport, e.g., in the ubiquitous networks. Data is transported from one process to another, the processes based in a computer, a fileserver, or data input channels, while during transportation there is no human interference, or no human interest is possible. Data remain in electronic or electro-magnetic form, not to be interpreted by humans. Persistence [3] is the main notion we deal with here: the concept of preserving information in a reusable state within a program.

The difference between visual and file I/O is that in visual I/O humans can inspect or type the data directly — no expedients are necessary for looking at what is going on. On the other hand, humans need the help of a program (or hardware) to inspect files or data traffic: file I/O.

Twentel should use a more natural way of describing these I/O actions through the use of trapdoors as we proposed it in visual I/O, instead of the way it is done now.

If we forego a formal definition of 'self-describing', and rely on its intuitive meaning, we can have file I/O with and without self-describing data. In the case of self-describing data, the description can be persistent (viz stored with the data in the file). The interpretation of the self-describing data file can be done by a program. Non-self-describing data need a human being to look at the data (or its documentation, *&c*). This person then starts the proper program for correct interpretation of the data. This way of treating the data might be classified once again as visual or human I/O, as humans are necessary to interpret the data correctly in order to present them in the right way.

4.5 A Fully Referential Transparent Trapdoor

During the process of implementing and realising the trapdoor, it turned out in retrospect that too much emphasis had been placed upon Twentel's inherent referential transparency property. We not only simply expanded Twentel, but also tried to beautify it in a sense which is theoretically elegant to some, but in practice not very usable. It meant expanding this basic property of Twentel to constructions which are in reality never immutable or fixed (databases, measurement signals), but essentially changing with time.

We now give a short description of this extra construction, a kind of shell around the trapdoor mechanism. This trapdoor shell stores for every *Outside* activation a 'shell triple', consisting of the evaluated argument with the matching returned function value and a Twentel generated tag (system state, denoting the state of the total Twentel system, e.g., the system time, or timestamp). This is done in such a way that, if the Twentel system has the same system tag (e.g., set by the user at system start), an *Outside* activation with the same argument will, through a 'table look-up' on the stored triples, result in the matching function value, without

having to do an actual *Outside* activation. In fact, the extension emulates the 'at-most-once' evaluation from lazy semantics.

So, if we have a very standard function, `f(foo) = bar`, implemented by `TRAP "f" [[foo]]`, then *Outside* is activated. The trapdoor shell registers the tuple (`f`, `foo`, `bar`) such that a subsequent application of `f` on `foo` yields the function value `bar` for *Outside* `f`, thereby satisfying the referential transparency property. All further references to this function value are replaced by the trapdoor shell by means of a look-up and retrieval operation. But as *Outside* was created to be a more efficient way of doing `f`, it is not necessary to save the result in a such a cumbersome manner, as we can execute `TRAP "f" [[foo]]` anew. And then, if for some reason `f(foo)` $\neq$ `bar`, where they should have been equal, there is more amiss in the whole process than a mere referential opaqueness. Let alone the serious implementation problems one will encounter when storing the function values of non-trivial SQL queries on a multi gigabyte database.

Every *Outside* functionality that can be described in Twentel, is by definition, referentially transparent. If time is of the essence in this description, we have to take it along in the model of reality we are building with our programs. So we should not put an *Outside* functionality into a referential transparent strait jacket, since this, by definition, cannot be described in Twentel. It exhibits some kind of oracular behaviour (e.g., an Input function; a database; or `ANY`, the random generator, 4.4.3). Nobody expects a random generator to produce the same value upon every activation. For everyday use it is not necessary — and counter intuitive — to be so strict.

However, the referential transparent behaviour had already been built around the trapdoor mechanism, so in using it together with the trapdoor, it demonstrated some more possibilities for using Twentel in other kinds of applications.

4.5.1 Memo Function, or Library

The shell mechanism is suitable for implementing a memo functionality or library of oft used values. If these values are kept after a session, which is very well possible, one might call this feature 'persistence'. Here we give a very short impression of such a memo function (which is different from the known memoisation techniques [31]).

For this construction we implemented a new Twentel keyword, `UNKNOWN`, which we borrowed from four-valued logic, known from database theory: `true, false, unknown`, and `undefined`.

The memo functionality, presented with a tuple `[id, val]`, first checks whether a triple for `id` exists. In this case the function value part of the triple is returned as value of the application. Otherwise the memo functionality stores `val` for `id`, unless `val = UNKNOWN`, then a request to input a value is displayed at the terminal, upon which the programmer can input a basic Twentel value, which value is subsequently stored for `id`. The stored value is then returned as function value of the memo function.

The following `TRAP` application shows the two uses: the request for the input of a Twentel value for the identifier `valu`(nknown) (and subsequently making it persistent), and making the value of `valk`(nown) (viz `valknown`) persistent.

```
TRAP "MEMO" [[valu, UNKNOWN], [valk, valknown]]
```

The first argument of this application leads to a terminal interaction, as we suppose `valu` not to be known to the memo function:

```
Memofun: valu
Memofun> inputval
```

In the interaction `valu` is displayed first with an indication of the memo functionality, followed by a request for a Twentel value (list, basic data item, string). This value must be given at the terminal (e.g., `inputval`). Then `valu`, `inputval` and the system state are stored as a triple for further reference. The second part of `Oargument` causes the shell to store the triple (`valk`, `valknown`, system state) straight away. The resulting value of the above application is the list `[inputval, valknown]`.

Every subsequent application (with the system in the same state) of

```
TRAP "MEMO" [[valu, UNKNOWN], [valk, otherval]]
```

will straight away result in `[inputval, valknown]`, without any terminal activity (Note that the memo function is referentially transparent, as the memo function searches first for an identifier). One might regard this functionality to be a kind of persistent array with associative indices (cf SNOBOL's `table`, and Concurrent LISP's `xapping`).

4.5.2 Referential Transparent Databases

Returning to the initial environment without trapdoors and remaining fully within the referential transparent regime, we see in the previous examples (starting at 4.3.2.2) a stream of instances of a full sized database. In the subsequent discussions we have ignored referential transparency because of the above discussed reasons: counter intuitive, and not usable.

If we try to implement the above sketched trapdoor shell with queries on databases, it will be evident that quite serious implementation problems will arise. However, there exists a theoretical way of dealing with these implementation problems of the trapdoor shell. In the context of queries on databases, referential transparent means that an answer to a query on 7 October will forever be the same, provided that we manage to set the clock (and the state of the database) back to 7 October, when asking the same query. If one can step through time in this way, an essential prerequisite for trend analysis is present.

Thinking of the system generated timestamp from the shell triple, knowing that it can be manipulated at system initialisation, we know that it can be used for this purpose. By sending it to the database system, with an appropriate action on that side, we can set the 'virtual clock' for the query time. The database system can now provide the correct '7 October' answer again. We have delegated the problem to the database system, and in this process the shell turns out to be unnecessary as the database system is now referentially transparent with respect to time.

Such systems do exist, 'temporal databases' [49]. In these systems one distinguishes 'transaction time', the moment in time when the information was stored in the database, 'valid time', the time when the relationship was valid, and 'user-defined time', used for additional information about time, which is not handled by transaction or valid time. Transaction and valid time are

orthogonal concepts of time. The Twentel sent timestamp must act in *Outside* (the Historical DBMS) on the transaction time of the database. 'Rollback databases' offer this possibility too [49], but these systems do not have 'valid time'. Valid time is essential for historical queries.

4.5.3 Miscellaneous Non-functional Uses

The extension to Twentel can be used to perform more duties than the above mentioned. Note however, that this kind of functions either belong to the realm of 'bells and whistles', or implicate serious philosophical problems regarding Time. This dissertation does not deal with the intricacies of dealing with indeterminacy, when correct functioning not only depends on the input and outputs and their relative ordering, but also upon their absolute ordering in time. The whole area of real-time programming and operating systems ([28]) is not covered. Yet time can play a role in a solution, so we present some aspects of it.

4.5.3.1 Time and its Clock

A 'function constant' `CLOCK` which supplies the time of the day, is *not* a function in the strict sense.

However, a referential transparent solution can be implemented through the introduction of a system constant

```
CLOCK = [currenttime, newclocks]
```

with `currenttime` an obvious `TRAP` application, and `newclocks`, an infinite list of clocks:

```
newclocks = [CLOCK1, CLOCK2, ...]
```

(with all clocks yielding later times than `currenttime`).[11]

With the correct implementation of `CLOCK` the following Twentel program will yield two times the same time, no matter how long the input process (`<< KB <<`) takes.

```
DO HD CLOCK ++ (<< KB <<) ++ HD CLOCK >> KB >> OD
```

4.5.3.2 Random Generator

A random generator is not a function in the strict sense, too. But we can use the trapdoor shell to force a replay of a specific simulation experiment involving random functions, as the system generated tag is open to user manipulation at system initialisation time.

Normal application of the random generator yields, for practical purposes, unpredictable (random) series of numbers (through the operation of a generator function within the random generator).

[11] This idea of a referential transparent solution to the clock functionality is due to Stef Joosten, and was communicated to us by Martin van Hintum (28 Nov 1990). Its implementation is beyond our scope at this point (e.g., non-shared `CLOCK*`'s are essential).

If the initial seed to the random generator is timestamped and stored in a 'shell tuple', then simulations based on a particular series of numbers can be replayed. We reset the initial seed to the value it had when the particular simulation started. As the random generator is a true function (deep down), this can be accomplished. Alas, this will not be the case, if the random generator function deep down was realised with an electron counter based on a radio valve.

4.5.3.3 Resolving Undefined References

A very true 'bell and whistle' is the possibility of resolving undefined references in a Twentel program at execution time. Inside the Twentel environment an undefined reference is 'trapped' to the Twentel `MEMO` trapdoor (4.5.1), which links the Twentel system tag and the name of the undefined reference to a basic Twentel value. This one is supplied by an *Outside* functionality (`"MEMO"`) which converts keyboard input into Twentel values. A forgotten definition can now be given a basic value at the moment its value is required. The same first-class-citizenship problem of functions plays a role here too: lists, integers and other basic data types can be input, but not functions (as yet).

References

[1] Abrahams, P., Larson, B., *Unix for the Impatient.*, Addison–Wesley, Reading MA, 1992.

[2] Aho, A.V., Ullman, J.D., "Universality of Data Retrieval Languages", in: Conf Rec 6^{th} ACM Symp POPL, (San Antonio TX, Jan 1979), ACM, New York, 1979, pp 110–120.

[3] Atkinson, M.P., Buneman, O.P., "Types and Persistence in Database Programming Languages", *ACM Computing Surveys* **9**(2) (Jun 1987), 105–190.

[4] Atkinson, M.P., Kulkarni, K.G., "Experimenting with the Functional Data Model", in: *Databases — Role and Structure. An Advanced Course.* P.M. Stocker, P.M.D. Gray and M.P. Atkinson (eds), (based on papers advanced course Univ East Anglia, Sep 1982), Cambridge Univ Pr, Cambridge, 1984, pp 311–338.

[5] Backus, J., "Can Programming Be Liberated from the Von Neumann Style? A Functional Style and its Algebra of Programs", *Comm ACM* **21**(8) (Aug 1978), 613–641 (1977 ACM Turing Award Lecture, 17 Oct 1977).

[6] Backus, J., "Is Computer Science Based on the Wrong Fundamental Concept of 'Program' ? An Extended Concept", in: *Algorithmic Languages*, J.W. de Bakker and J.C. van Vliet (eds), (Proc Int'l Symp -title-, Amsterdam, Oct 1981), North-Holland, Amsterdam, 1981, pp 133–165.

[7] Barendregt, H.P., *The Lambda Calculus, its Syntax and Semantics.* Studies in Logic and the Foundations of Mathematics, Vol 103, North-Holland, Amsterdam, 1981 (2^{nd} revised ed, 1984).

[8] Bird, R.J., Wadler, P., *Introduction to Functional Programming.* Prentice Hall, New York, 1988, xv + 293 pp.

[9] Blaauw, G.A., Brooks Jr, F.P., *Computer Architecture — Vol 1: Design Decisions.*, Preliminary Draft, College Notes, Privately Circulated, Univ Twente, 1975–1987, (to be published by Addison–Wesley).

[10] Blaauw, G.A., Duijvestijn, A.J.W., Hartmann, R.A.M., "Optimization of Relational Expressions Using a Logical analogon", *IBM J Res Development* **27**(5) (Sep 1983): 497–519.

[11] Blaauw, G.A., Duijvestijn, A.J.W., Hartmann, R.A.M., *Relational Expression Optimization.* IBM Netherlands, Dept INSDC, Doc TR 13.190, Uithoorn, Jul 1984, 20 pp, (with 4 Appendices: [10];

Blaauw, G.A., Duijvestijn, A.J.W., Nieuwerth, F., "Decomposition of Select Expressions", pp A.1–A.9, (also in: *Inf Syst* **10**(3) (1985): 325–330) Blaauw, G.A., Duijvestijn, A.J.W., "The Split, a New Relational Operator", pp A.10–A.22; Anonymous, "Mathematical Expectation", pp A.23–A.26).

[12] Blaauw, G.A., Duijvestijn, A.J.W., "Efficient Calculation of Prime Implicants of a Logical Function. Prototyping Example in a Functional Language", in: *NGI-SION 1985*, A.J.W. Duijvestijn *et al* (eds), (Proc 3rd Conf -title-, Utrecht, Apr 1985), Nederlands Genootschap Informatica, Amsterdam, 1985, pp 177–193.

[13] Broersma, H.J., Duijvestijn, A.J.W., Göbel, F., *Generating all 3-Connected 4-Regular Planar Graphs from the Octahedron Graph.* Dept Applied Math, Memorandum No 854, Univ Twente, Enschede, Apr 1990, 9 pp; (publ in: *J Graph Theory* **17**(5) (1993): 613–620).

[14] Bolour, A., Anderson, T.L., Dekeyser, L.J., Wong, H.K.T., "The Role of Time in Information Processing: a Survey", *SIGART Newsl* #**80** (Apr 1982): 28–48.

[15] Bossi, A., Ghezzi, C., "Using FP as a Query Language for Relational Data-bases", *Comp Lang* **9**(1) (1984): 25–37.

[16] Buneman, O.P., Frankel, R.E., "FQL — A Functional Query Language", in: *Proc ACM-SIGMOD 1979 Int'l Conf Management of Data,* P.A. Bernstein (ed), (Boston MA, May–Jun 1979), ACM, New York, 1979, pp 52–58.

[17] Buneman, O.P., Frankel, R.E., Nikhil, R.S., "An Implementation Technique for Database Query Languages", *ACM Tr Database Syst* **7**(2) (Jun 1982): 164–186.

[18] Buneman, O.P., "Can We Reconcile Programming Languages and Databases?", in: *Databases — Role and Structure. An Advanced Course.* P.M. Stocker, P.M.D. Gray and M.P. Atkinson (eds), (based on papers advanced course Univ East Anglia, Sep 1982), Cambridge Univ Pr, Cambridge, 1984, pp 225–243.

[19] Buneman, O.P., Nikhil, R.S., "The Functional Data Model and its Uses for Interaction with Databases", in: *On Conceptual Modelling — Perspectives from Artificial Intelligence, Databases, and Programming Languages,* M.L. Brodie, J. Mylopoulos and J.W. Schmidt (eds), Springer-Verlag, New York, 1984, pp 359–384.

[20] Buneman, O.P., "Functional Programming and Databases", in: D.A. Turner (ed), *Research Topics in Functional Programming*, (Proc Meeting Institute of Declarative Programming, Austin TX, Aug 1987), Addison–Wesley, Univ Texas at Austin Year of Programming Ser, Reading MA, 1990, Ch 6, pp 155-169.

[21] Codd, E.F., "A Relational Model of Data for Large Shared Data Banks", *Comm ACM* **13**(6) (Jun 1970): 377–387.

[22] Codd, E.F., "Is your DBMS Really Relational?", *Computerworld* **XIX**(41 & 42) (Oct 1985): ID/1–ID/9 & 49–60, (parts 1 and 2).

[23] Curry, H.B., Feys, R., *Combinatory Logic.* Vol I, North-Holland, Amsterdam, 1958, (2^{nd} ed, 1974).

[24] Duijvestijn, A.J.W., *Electronic Computation of Squared Rectangles.* PhD Thesis, Technische Hogeschool Eindhoven, Eindhoven, 1962; (also *Philips Res Rep* **17** (1962): 523–612).

[25] Duijvestijn, A.J.W., *Algorithmic Calculation of the Order of the Automorphism Group of a Graph.* Dept Applied Math, Memorandum No 221, Univ Twente, Enschede, 1978.

[26] Duijvestijn, A.J.W., *e-mail*, Enschede, 5 Feb 1991.

[27] Frankel, R.E., *FQL — The Design and Implementation of a Functional Query Language.* Univ Pennsylvania, Moore School, MSc Thesis, May 1979.

[28] Harrison, D., *Functional Real-time Programming: The Language Ruth and its Semantics.* Univ Stirling, Dept Computing Sci, PhD Thesis, Tech Rep TR.59, Stirling, Sep 1988, 221 pp.

[29] Van der Hoeven, G.F., *Preliminary Report on the Language Twentel.* Memorandum INF-84-5, Dept Comp Sci, Univ Twente, Enschede, Mar 1984, 87 pp.

[30] Hudak, P., Peyton Jones, S.L., Wadler, P., (eds) *et al*, *Report on the Programming Language Haskell, a Non-strict, Purely Functional Language.* Version 1.2; 1 March 1992, xii + 164 pp, in: *SIGPLAN Not* **27**(5) (May 1992), Section R; *also* Technical Report, Yale Univ *and* Univ Glasgow, Aug 1991.

[31] Hughes, R.J.M., *Lazy Memo-Functions.* Chalmers Univ Technology, Prog Methodology Gr, Memo PMG-42, Jan 1985, 22 pp, (also in: J-P. Jouannaud (ed), *Functional Programming Languages and*

Computer Architecture. (Proc Int'l Conf -title-, Nancy, Sep 1985), Springer LNCS # 201, Springer-Verlag, Berlin, 1985, pp 129–146).

[32] International Organization for Standardization (ISO), *ISO Computer Programming Language Pascal.* ISO 7185, Geneva, 1983.

[33] International Organization for Standardization (ISO/TC 97/SC 21/WG 3 & ANSI X3H2)., *ISO Database Language SQL.* ISO 9075, Geneva, 1987.

[34] Jensen, K., Wirth, N., *Pascal User Manual and Report.* Springer-Verlag, Berlin, 1974, (3^{rd} ed, 1985).

[35] Joosten, S.M.M., *The Use of Functional Programming in Software Development.* PhD Thesis, Dept Comp Sci, Univ Twente, Enschede, Apr 1989, 140 pp.

[36] Kroeze, H.J., *The Twentel System, Version I - (1.24–1.99) Available on Various Machines: General Reference Manual & User Guide.* Dept Comp Sci, CAP / Languages Group, Univ Twente, Enschede, 1986–1987, 116 pp.

[37] Kulkarni, K.G., Atkinson, M.P., "EFDM: Extended Functional Data Model", *Comp J* **29**(1) (Jan 1986): 38–46.

[38] Lindstrom, G. *et al*, "Critical Research Directions in Programming Languages", (Rep Workshop sponsored by ONR), *SIGPLAN Not* **24**(11) (Nov 1989): 10–25.

[39] McCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P., Levin, M.I., Lisp *1.5 Programmer's Manual.* MIT Pr, Cambridge MA, 1962 (2^{nd} ed, 1965, vi + 106 pp).

[40] Nikhil, R.S., "Functional Databases, Functional Languages", in: *Data Types and Persistence*, M.P. Atkinson, O.P. Buneman and R. Morrison (eds), (based Proc Appin Workshop -title-, Appin, Aug 1985), Topics in Information Systems Ser, Springer-Verlag, Berlin, 1988, Ch 5, pp 51–67.

[41] Peyton Jones, S.L., *The Implementation of Functional Programming Languages.* Prentice-Hall Int'l, Englewood Cliffs NJ, 1987, xviii + 445 pp.

[42] Poulovassilis, A., King, P., "Extending the Functional Data Model to Computational Completeness", in: F. Bancilhon, C. Thanos, D. Tsichritzis (eds), *Advances in Database Technology — EDBT'90.* (Proc Int'l Conf Extending Database Technology, Mar 1990, Venice), Springer LNCS # 416, Springer-Verlag, Berlin, 1990, pp 75–91.

[43] Raymond, E. (ed), *The On-Line Hacker Jargon File, version 2.9.12.* `esr@snark.thyrsus.com`, May 1993.

[44] Rosen, K.H., *Discrete Mathematics and its Applications.* Random House / Birkhäuser Mathematics Ser, Random House, New York, 1988.

[45] Schnitger, P.K.H., *Prototyping a Distributed Multi-user RDBMS in a Functional Language.* MSc Thesis, Dept Comp Sci, SETI Group, Univ Twente, Enschede, May 1991, 143 pp.

[46] Scott, D.S., *Outline of a Mathematical Theory of Computation.* Univ Oxford, Tech Monograph PRG-2, Oxford, Nov 1970, 24 pp.

[47] Shipman, D.W., "The Functional Data Model and the Data Language DAPLEX", *ACM Tr Database Syst* **6**(1) (Mar 1981): 140–173.

[48] Sibley, E.H., Kerschberg, L., "Data Architecture and Data Model Considerations", in: AFIPS Conf Proc 1977 Nat'l Computer Conf, R.R. Korfhage (ed), (Dallas TX, Jun 1977), AFIPS Pr, Montvale NJ, 1977, pp 85–96.

[49] Snodgrass, R., "The Temporal Query Language TQUEL", *ACM Tr Database Syst* **12**(2) (Jun 1987): 247–298.

[50] Steutel, D., Van Zonneveld, E.P., *Prototyping a Relational DBMS in a Functional Language.* MSc Thesis, Dept Comp Sci, SETI Group, Univ Twente, Enschede, May 1988, 53 pp.

[51] Stoye, W.R., *The Implementation of Functional Languages Using Custom Hardware.* Univ Cambridge, Comp Lab, PhD Thesis, Tech Rep 81, Cambridge, Dec 1985.

[52] Turner, D.A., "A New Implementation Technique for Applicative Languages", *Softw Pract & Exp* **9**(1) (Jan 1979): 31–49.

[53] Walters, H.R., *On Equal Terms – Implementing Algebraic Specifications.* PhD Thesis, Univ Amsterdam, Amsterdam, Jun 1991, 193 pp.

[54] White, J.L., "Lisp: Program is Data — A Historical Perspective on MACLISP" *and* "Lisp: Data is Program — A Tutorial in Lisp", (1977 MACSYMA Users Conf, Berkeley CA, Jul 1977), NASA CP-2012, Jul 1977.

[55] Van Wijngaarden, A. *et al* (eds), *Revised Report on the Algorithmic Language ALGOL 68.* Springer-Verlag, Berlin, 1976.

[56] Woods, W.A., "What's in a Link: Foundations for Semantic Networks", in: *Representation and Understanding — Studies in Cognitive Science.* D.G. Bobrow and A. Collins (eds), Academic Pr, New York, 1975, pp 35–82.

Chapter 5

Expanding Twentel, the Implementor's View

In the first Section of this Chapter we present the Implementation of the trapdoor. By definition the Implementation cannot be seen by the Twentel programmer (2.4.3). The possibility of communicating with outside resources must now be given form, be implemented, and subsequently realised. Implementing, giving form by designing algorithms and structures, calls for a description language. In this vein we observed the necessity of introducing notations related to communicating processes (a.o., CCS and CSP) to describe the interaction between the functional language and the outside resources (3.1.6). In this Chapter we will present another way of describing this interaction, in character with the rewriting paradigm on which many functional languages are founded. It is also in character with the observations we made if we apply a language to an area it is apparently not meant to cover (3.1.4).

In the second Section of this Chapter we will briefly sketch the realisation of the trapdoor-complex. The knowledgeable reader will appreciate the condensation in one Section of the labour involved in our research. Related to this phase we discuss some problems encountered while constructing the trapdoor.

5.1 Implementation of the Trapdoor

Twentel is described by a graph rewriting machine as the first, top most, abstract machine (4.2.1). Within this machine everything can be described in terms of graph rewriting or of primitives of lower abstract machines, among them a functionality machine (e.g., the ALU with the adder (`ADD`), *Outside* with various functionalities (`TRAP`)).

We introduced (4.2.3.1) a new primitive, the `TRAP` combinator, which shows the desired functionality realised in a lower abstract machine. We will now define — in a more formal manner — what happens 'beyond' the `TRAP` combinator. As it is apparent that the construction implementing the trapdoor lies partially outside Twentel, the division between the Twentel system and the *Outside* world, the non-Twentel system, requires some special attention. Gradually we will expand the description of the construction of the trapdoor in rewrite rules until we reach the point where the essential transition to the functionality machine is located.

5.1.1 The Actual Implementation

The architecture of the trapdoor is completely defined at this point. To recapitulate, the trapdoor enables us to send Twentel values (`Oargument`) to a computing agent that is a functionality (`Ofun`) located in *Outside*. This functionality computes the desired function value (`FVal`) and returns this value to Twentel, where it is taken into the graph as the value of the trapdoor evaluation.

5.1.1.1 Twentel — the Master or Client

The *Outside* functionality is represented in rewrite rules of the graph rewriting machine by the following extended rewrite rule from 4.2.3.1:

```
TRAP  Ofun  Oargument  --> ... -->  FVal
```

The special 'arrow', `'--> ... -->'`, denotes the place where the desired *Outside* functionality must be used in the rules of the rewriting machine. In the following further specification of the process we will observe this 'arrow' moving 'down', until we reach the transition to *Outside*.

To keep things simple — separation of concerns — we separate the trapdoor activating activity (write argument from Twentel to *Outside*) from the Twentel reactivating activity (read function value from *Outside* into Twentel). This route is also suggested by the non-atomic trapdoor discussion (4.4).

We introduce a few internal combinators to implement these activities. Internal combinators cannot be used in Twentel source text, but are used by Twentel itself; they are normal Twentel implementation practice [11]. The new combinators are: `TRY` for Trap Recursion,[1] `TWR` for Trap Write, and `TRD` for Trap Read.

Going 'down' one level, we now take the opportunity to 'internalise' `TRAP`'s first argument for efficiency reasons. After some checks (e.g., correct Twentel string, existing trapdoor) `Ofun` is replaced by an internal *Outside* functionality identifier, `Ofid`. The above extended rewriting rule breaks down as follows (still in the rewriting machine), with `a, b, ...` atomic Twentel constructs, like an integer, and with `x, y, ...` arbitrary — and so also composite — Twentel constructs, like a list (see [6]):

```
TRAP  Ofun  Oargument  -->      TRY  Ofid  Oargument
TRY   Ofid  [ ]        -->      [ ]
TRY   Ofid  x : y      -->      TWR  Ofid  x :  ( TRY  Ofid  y )
TWR   Ofid  x      --> ... -->  TRD  Ofid
```

The special 'arrow' `'--> ... -->'` is now situated between write-argument and read-function value: the desired *Outside* functionality will be used in-between.

The trapdoor accepts a list of arguments for subsequent activations of the trapdoor mechanism, all for the same *Outside* functionality. This list of arguments for `Ofun` results in a list of returned

[1] Note that in combinatory logic the 'pattern' $\mathsf{Y}\,f = f\ (\mathsf{Y}\,f)$ exists (3.4.4.1). We reuse this pattern in our design, as is clear from `TRY`'s rewrite rules.

function values (4.2.3.1).

As will be apparent from inspection of the rewrite rules, more than mere input and output of arguments is present at the graph rewriting level. One aspect of the philosophy of functional languages is freeing the programmer from simple, elementary, actions which are, to a certain extent, implicit in new, higher actions. One example is the list comprehension (3.3.1). Twentel has its own examples:

- `CMNLST`, the predefined function `COMMON` operating on a list of lists:
  `CMNLST (x : y : Z) = COMMON x (CMNLST (y : Z));`

- the family of related predefined functions `LREC, LITER, LSTREC, NLREC, NITER` and `REDUCE`.

Keeping this aspect in mind, we extended the use of the trapdoor to contain an implicit `MAP`. This implementation of `TRAP` frees the programmer from having to break up a sequence of *Outside* activations by means of the `MAP` combinator into solitary activations. The subsequent activations of *Outside* are taken care of inside Twentel through the `TRY` combinator. Instead of `MAP (TRAP Ofun) Oargument` we now write `TRAP Ofun Oargument`.[2]

An Example An example will show the functioning of this mechanism on Twentel's side. `Ofun` is an adder of integers in lists (a 'List Adder', named `"Lad"`). This functionality operates on a variable length list of variable length lists of integers, yielding the sum of the basic lists. Every line in the example shows a step in the reduction process, seen from the Twentel side within the extended Twentel graph reduction machine. No rewriting within the functionality — apart from the sum of the integers, manifest in the returned `FVal` — is shown. We see the finite list of arguments to the 'List Adder' functionality, `Oargument`, being examined and dispatched by the `TRY` combinator, every time providing `"Lad"` with a new list of lists through application of `TWR`. Automagically,[3] `TRD` returns the resulting value.

Suppose we have a normal Twentel application of `TRAP` on `"Lad"` with arguments for two subsequent activations of `"Lad"`:

```
TRAP "Lad" [[[1,2,3],[4,5],[7,8,9]],[[11,12,13,14,15],[16,17]]]
```

Inside extended Twentel, applying `TRAP` rewrite and replacing `"Lad"` by an internal number `LadId`:

```
TRY LadId [[[1,2,3],[4,5],[7,8,9]],[[11,12,13,14,15],[16,17]]]
```

A rewrite of `TRY` in order to set up the first activation of the trapdoor, while postponing evaluation of the rest of the arguments:

```
TWR LadId [[1,2,3],[4,5],[7,8,9]] :  TRY LadId [[[11,...,15],[16,17]]]
```

[2] Only after having written down exactly what had been constructed and how this construction formally works, it turned out that there was a `MAP` inside.

[3] [17].

`TWR` activates the trapdoor by sending `[[1,2,3],[4,5],[7,8,9]]`, and the system waits for the return function value in `TRD`, still pending evaluation of the rest of the original list of arguments in `TRY`:

```
TRD LadId :  TRY LadId [[[11,...,15],[16,17]]]
```

`"Lad"` returns its function value, `[6,9,24]`, which is taken up by `TRD`:

```
[6,9,24] :  TRY LadId [[[11,12,13,14,15],[16,17]]]
```

Another `TRY` rewrite inside extended Twentel:

```
[6,9,24] :  TWR LadId [[11,12,13,14,15],[16,17]] :  TRY LadId []
```

`TWR` activates trapdoor by sending `[[11,...,15],[16,17]]`, and the system waits for the return function value in `TRD`, pending evaluation of the rest of the list in `TRY`:

```
[6,9,24] :  TRD LadId :  TRY LadId []
```

Function value returned from *Outside*, `[65,33]`, which is handled by `TRD`:

```
[6,9,24] :  [65,33] :  TRY LadId []
```

`TRY` inside extended Twentel, which returns `[]` on an empty list as second argument:

```
[6,9,24] :  [65,33] :  []
```

And finally, normal Twentel graph rewriting:

```
[[6,9,24] , [65,33]]
```

□

5.1.1.2 The Functionality Machine — the Slave or Server

We now come to the transition from extended Twentel to the functionality machine *Outside*. In this Section, and in the rest of the Chapter, we will often use '*Outside*' where we should have used 'a specific functionality in the *Outside* machine'. Stepping outside Twentel we observe another process to be engaged in computations for the main process within the Twentel system. For the communication of arguments and function values between these processes, with waiting, input and output, we could have used the description methods of 3.1.6. As we choose to stay as close to rewriting as possible, it is apparent that the process of waiting cannot be described in mere rewrite rules in Twentel graph space. Twentel graph space should be kept for rewriting that has something to do with the Twentel elaboration itself (the 'propriety' criterion). We need to augment our rewriting description method, and we take some liberties in applying this addition.

A simple depiction of the mechanisms involved, is found in Figure 5.1. Herein, and thereafter, 'Loek' is the name for parts of the realisation of the trapdoor concept. Loek is a real program. Apart from its main function, transferring values to and from Twentel, it also offers some special facilities to Twentel (e.g., the memo function and the random generator, as discussed in 4.5.1 and 4.5.3.2). In a process 'related' to `TWR` (`deposit`), information regarding the required *Outside*

functionality and the arguments is sent to Loek, outside the Twentel graph space. Loek sends the arguments to the proper *Outside* functionality, where they are received and processed. The resulting function value is sent back to Twentel through Loek. In the Twentel system, by means of a process 'related' to `TRD` (`accept`), the function value is put into the graph space. We

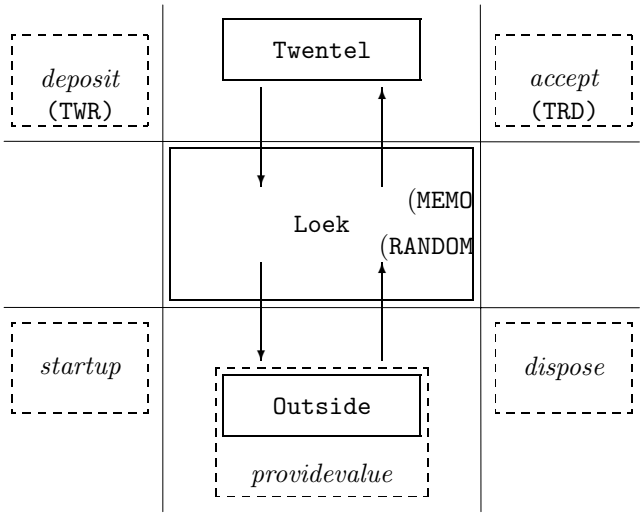

Figure 5.1: *Standard configuration of Twentel with duplex trapdoor to* Outside

will further elaborate on the processes `deposit`, `startup`, `accept`, `providevalue`, `dispose` and `accept` in the subsequent Paragraphs.

The main problem is the waiting of the *Outside* machine. We do not want to complicate our situation with the initiation and subsequent maintenance of a communication session at this level: we assume there is an *Outside* machine functioning and waiting for input from Twentel. Recall the visibility discussion in 4.4.3. The functionality we need simply *is* there. One can distinguish various manifestations of waiting in the *Outside* machine:

- it is not yet ready to accept input from Twentel;
- it cannot send output to Twentel when Twentel requests it;
- it waits for the completion of all its input from Twentel before it can start its computation;
- it is waiting for an outside agent to produce something (user at keyboard, trigger signal).

In the following Paragraphs we will see the way Twentel, Loek and *Outside* handle these problems.

Twentel to *Outside* We start with the first leg of a communication session between Twentel and *Outside*. Within Twentel we define a process 'under' the `TWR` combinator: the `deposit` process. Every combinator with its arguments initiates some activity in the rewriting machine if it is a redex to be reduced in this machine. We introduce the 'process term' ('process' for short), in this case `deposit`. 'Process term' is a dual notion: in the graph space it is a node in the graph with associated rewrite rules. At the same time it is an autonomous process, started by the Twentel graph rewriting machine when it appears at the head of a redex to be reduced,

and kept 'alive' by having its node at the head of a redex to be reduced in the Twentel graph space, at the tip of the spine.

Now we take the liberty to introduce another graph space, down under the Twentel graph space: the 'nether graph space'. The nether graph space is the area where the special 'arrow' `'--> ... -->'` is further elaborated. In this new graph space the other reductions take place, reductions necessary for describing the behaviour of the 'process terms' with rewrite rules. The nether rewrite rules are denoted by '`==>`' to distinguish them from the normal Twentel rewriting arrow '`-->`'.

`deposit` must transfer `Oargument` to the *Outside* functionality designated by `Ofid`, herein aided by Loek. Now we have the possibility of *Outside* still being busy with a previous activation — or not present at all — so we have to wait for an indefinite time. This waiting is modelled by the `WAITETY` loop.[4] [5] As such it is not an altogether strange phenomenon, a waiting loop implemented by rewriting rules. We assume the graph rewriting machine to take an infinitesimal time in every rewrite action, as otherwise the sequence of applied rewrite rules would collapse.

```
TWR  Ofid  x          ==>  deposit  WAITETY  Ofid  x
WAITETY               ==>  p  WAITETY  |  v
deposit  p            ==>  deposit
deposit  v  Ofid  x   -->  TRD  Ofid
```

The 'process term' `deposit` comes into action when `TWR Ofid x` is reduced. Seen from the Twentel graph space `TWR Ofid x` is in the process of being reduced. At the same time `deposit` begins its nether rewriting, initiated by the Twentel graph reduction machine. The last rewrite rule describes the situation wherein the `deposit` process achieves the actual activation of *Outside*. At the same moment it rewrites the normal graph space by finally substituting `TWR Ofid x` by `TRD Ofid`. Only then the reduction

```
TWR   Ofid  x      --> ... -->  TRD  Ofid
```

from 5.1.1.1 is finished.

We can think of the embodiment of this activation as the filling of a kind of tray upon which something can be put (viz `x`, the argument of the function). In another action (by another process) that tray can be emptied again, the argument can be taken away. So `deposit`, in one movement, puts `x` on an empty tray, the argument tray, and finally rewrites the Twentel graph space.

---

[4] The description is taken from the Van Wijngaarden – ALGOL68 two-level grammar style [20]. Though in ALGOL68 the suffix `-ETY` denotes a production rule that can have an empty right hand side, we use it here in the sense that the waiting time in the queue can be zero (signifying an *empty* queue).

[5] Historically, the symbols `p` and `v`, are derived from the Dutch *pak* and *vrij* (or, get and free), which have an altogether different meaning as well; pun might not have been intended.
A less poetic explanation is that `p` stands for *proberen* (or, try) and `v` for *verhogen* (or, increment), according to the classic definition [19]:

$p(S)$: while $S \leq 0$ do no-op; $s := s - 1$;

$v(S)$: $s := s + 1$;.

Yet another explanation is, *passeren* and *vrijgeven*, or pass and yield.

Outside **Itself** *Outside* is a composite of three processes, the initiating process `startup` with `start-of-process`, the main process `providevalue`, and the finalising process `dispose`.

As will be clear from the following discussion, `startup` must be able to activate `providevalue`. To keep the model simple, we do not allow constructions like 'batches' or 'scripts' in the model, so the only possibility to meet this requirement is that both processes belong to the same process in *Outside*. The same holds for `providevalue` and `dispose` and, again, for `dispose` and `start-of-process`.

The `startup` process is eager, a consequence of keeping the *Outside* model simple (5.1.1.2). Its task is to wait actively for the moment that the argument (`x`) for *Outside*'s main process (`providevalue`) is put upon the argument tray (by `deposit`). It takes the argument for `providevalue` from the argument tray (thus emptying it), and passes it in an appropriate form to `providevalue`, upon which it activates the `providevalue` process. Again we introduce a new graph space, the *Outside* graph space, to describe the behaviour of *Outside* in rewrite rules.[6]

```
start-of-process        -->  startup  WAITETY  x
startup  p              -->  startup
startup  v  x           -->  providevalue  INPETY  x
```

At this point we have modelled the possibility of waiting for (the typing of) keyboard input, or the waiting for *Outside* to finish its computations, or the like. Though we want to abstract from the waiting *per se*, these things do take time. This process of the waiting of *Outside* for the supply of its return value is modelled by the `INPETY` loop, an internal waiting loop inside the `providevalue` process. No special graph space is needed here, as everything can be defined in *Outside* graph space, something we could not do by modelling the 'process term' behaviour within Twentel graph space.

When `providevalue` is finished with the computation of its function value, it passes the computed function value (viz `FVal`) to `dispose`.

```
startup  v  x           -->  providevalue  INPETY  x
INPETY                  -->  p  INPETY  |  v
providevalue  p         -->  providevalue
providevalue  v  x      -->  dispose  WAITETY  FVal
```

The last part of the *Outside* process deals with the transition from `providevalue` to `dispose`. `dispose` will put the function value on another tray, the function value tray, which must be empty. If the function value tray is occupied, another `WAITETY` loop will ensue. Once `dispose` has put `FVal` on the tray, it re-activates `start-of-process`, the main entry of the eager-waiting process of *Outside*.

```
providevalue  v  x      -->  dispose  WAITETY  FVal
dispose  p              -->  dispose
dispose  v  FVal        -->  start-of-process
```

□

[6] We should have taken yet another form of rewriting arrow, as this graph space is different again from Twentel and nether graph space. In this case however, there cannot be a mistake in the graph space involved.

Outside **To Twentel** On the return leg of the communication session from *Outside* to Twentel, we have the possibility of a still occupied function value tray. This can be caused by Twentel not yet having taken the function value from the tray. This situation is detected and handled by `dispose`. Another situation can be caused by Twentel not being ready to accept the function value *Outside* provides, in which case the Twentel graph rewriting is not yet ready to reduce the generated `TRD` at the tip of the spine, a situation which is handled by `accept`. In both cases this waiting is once more modelled by a `WAITETY` loop.

The receiving of function values from *Outside* can be described by the following rules. The `WAITETY` loop is necessary for describing the waiting of `accept` for the required result to be put onto the function value tray. If `FVal` is put on the function value tray (by `dispose`), `accept` fetches it, and empties the tray. Next, `accept` converts the returned `FVal` on the function value tray to an internal Twentel format, suitable for inclusion in the Twentel graph space, and inserts it in the Twentel graph space. Again the introduction of nether graph space rewriting is necessary.

```
TRD  Ofid             ==>  accept  WAITETY  Ofid
accept  p             ==>  accept
accept  v  Ofid       -->  FVal
```

And again, the last rule is the place where the actual return of the computed function value from *Outside* takes place and appears in Twentel graph space.

5.1.1.3 Simplex Trapdoor Combinators

We can now address the uneasy feeling we described in 4.4.1, when introducing two special forms of the trapdoor combinator to separate the O- and I- behaviour of `TRAP`:

```
TRAP  "OSWIN"   Oargument --> ... -->  FVal
TRAP  "OSWOUT"  Oargument --> ... -->  I
```

Looking at these two rewrite rules, we observe an offence against the criteria of generality and simplicity (2.4.4): only for `"OSWOUT"` and `"OSWIN"` there is an exception on the dispatch of the first argument of `TRAP`. We introduce two new trapdoor combinators showing the same functional behaviour in order to abolish the idiosyncrasies mentioned above. These new combinators are `TREAD`, for Trapdoor Read, and `TRITE`, for Trapdoor Write.[7] We cannot use a derivation of the `TRD` and `TWR` combinators, as this would expose lower level details, like `Ofid`, at the top, architectural level. Nevertheless, the two new combinators must have the same functional building blocks as `TRAP`.[8] In the same action we also get rid of the superfluous `Oargument` from `TRAP "OSWIN"`.

```
TRITE  Ofun  Oargument  --> ... -->  I
TREAD  Ofun             --> ... -->  FVal
```

[7] The nomenclature of these new combinators is in full accordance with the treatment of this trite and refractory subject in [2]: "Reeling and Writhing, of course, to begin with," the Mock Turtle replied; "and then the different branches of Arithmetic — Ambition, Distraction, Uglification, and Derision."

[8] The derivation of a relation between the complex `TRAP` and the simplex `TREAD` and `TRITE` follows in the Appendix.

On the lower level this means defining the following, additional, rewrite rules together with a new 'process term', `discard`, and a new combinator `TWRI` (or, `TWR` reducing to `I`). Note that the implicit `MAP` present in `TRAP`, has disappeared. `discard` is essentially the same as `deposit`, but for the final rewriting. Instead of reducing to `TRD Ofid`, implying waiting for the function value, `discard` reduces to the identity combinator `I` (3.4.4.1).

```
TRITE  Ofun  Oargument       -->  TWRI   Ofid  Oargument
TWRI   Ofid  Oargument       ==>  discard  WAITETY  Ofid  Oargument
discard  p                   ==>  discard
discard  v  Ofid  Oargument  -->  I
```

On the receiving side, the additional rewrite rule is simple:

```
TREAD  Ofun                  -->  TRD  Ofid
```

The complete, though still global, situation is shown in Figure 5.2, where the 'process term' `discard` is situated 'under' `TRITE`. Looking at the description of `deposit`, `discard` and `accept`,

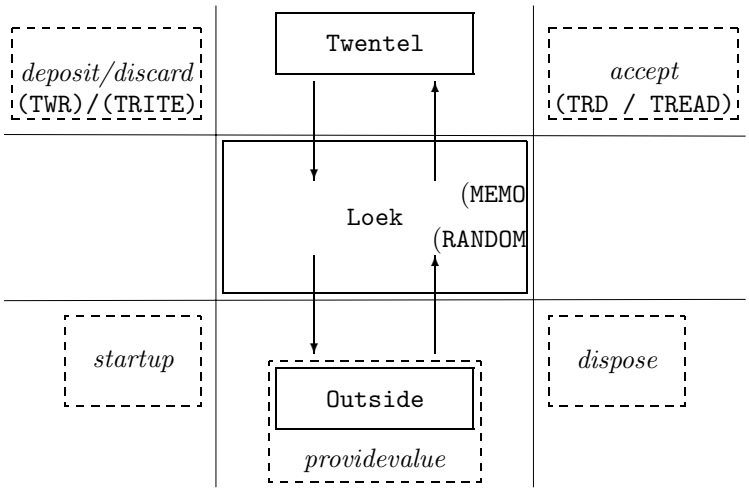

Figure 5.2: *Standard configuration of Twentel with simplex trapdoors to* Outside

it is clear that these processes have been realised in the Twentel system.

5.1.1.4 The Data In Between

The trapdoor is a communication agent: facilitating the transfer of data from one environment to another. Communication involves language: in this kind of communication the language involved is called a protocol, and the protocol we will use here is based upon LISP's main datastructure, the list. We choose a list structure, since this is simple to describe, and easy to implement on top of Twentel. Moreover, it is a very general composite data structure, as every composite data structure can be represented as a list — the representation of a mathematical sequence, made possible through the use of the CONS-operation [14].

In literature one can find a few papers on the subject of transporting data between processes, other than via files, remote procedure calls, message passing and interprocess communication.

POLYLITH [16] aids the programmer to interconnect mixed-language software components for execution in heterogeneous environments. It realises this by introducing a 'software bus' between the components, decoupling the interface requirements from the functional requirements. This 'bus' is comparable to a mix of trapdoor and Loek aspects. In POLYLITH's Module Interconnection Language (MIL) one declares the modules to interconnect and where to find them. We have not implemented MIL, as we required Loek to find the right functionality. The data structuring needed for arbitrary argument passing is hidden in the local stubs connected to the 'software bus' on one side, and to a module on the other side (comparable to `ConvertOutputRecordFor-Twentel` and `GetInputFromTwentel`, only appearing at the slave side). We do not need this system to aid in demonstrating the extension to Twentel, though a MIL subset implementation (where to find what modules) would help in generalising Loek.

The Interface Description Language (IDL, [12]) is the notation for describing structured data needed to control the exchange of these data between different components of a large system (our ILS, 5.1.1.4). The IDL translator generates readers and writers that map between concrete internal descriptions and abstract exchange representations (the work done in `deposit`, `startup`, `dispose` and `accept`). We tend towards the more abstract solution, of supplying a description of the internal description and then have a universal reader / writer absorb or generate the abstract representation.

The IBM effort in enhancing data interchange on one of its platforms is described in [4]. Its 'A Data Language' (ADL) addresses the (byte) representation of the different data types on various platforms (5.1.4). This system also uses translators to generate coding instead of using a universal reader / writer with a description.

The last two systems describe a method, explaining what structures the modules concerned will operate upon, thereby generating the necessary coding for conversion. We make the module carry a description of what structures it will accept and send, a method, that, combined with a universal reader / writer, provides a more flexible approach. Stretching the imagination a bit, we might conclude that the first method is imperative, whereas ours is declarative.

So in the trapdoor system, with one side geared to lists, we stay with the most general composite data structure, the list. We cannot use the internal Twentel format for the data which pass through the trapdoor, comprising the transmitted Twentel values. It would mean low-level Twentel realisation details exported to the outside world. The trapdoor accepts from both sides a full list structure in the form of tokens transmitted over a channel.

In this Subsection we discuss argument and function values passing to and from *Outside* functionalities. The structure of these values — all values are transferred under the same format description — forces a comparison with the message passing from the object-oriented paradigm (3.1.2). With Twentel activating other processes by means of arguments in ILS format, we may consider the Twentel system a giant object, a universal object with much more possibilities than the normal OO-objects. It is clear we favour the more flexible approach offered by the Twentel object as it can connect to other processes in a simple way; objects should profit from this too.

Token Streams The argument and function values are transferred in streams of tokens to and from *Outside*. A token is a composite of tag with (matching) data. What data, how much data and the way the data is represented, depends on the tag (the data type). Its denotation

is the construction "`< tag & ‘data’ >`". A channel is the logical data path between Twentel and *Outside*.

The following tokens are defined:

- basic token, consisting of a basic data item (`integer, real, Boolean`, and `character`) according to the width of the data path of the channel, with matching tag (the *type* of the accompanying basic data item);
- open-, and close-list tokens (left and right parentheses); data are irrelevant here (‘don’t care’), so it is only the ‘(’- or ‘)’-tag that will be meaningful in the transmission of the token;
- bottom token ($\perp$); this is a sensible extension to the basic structure as the required functionality might not deliver. One can think of several reasons for non-delivery of the function value:
  - it takes too much time to compute. Here the aspect of ‘computation time’ plays a role. Computation time depends on the complexity of the computation, and on the size of the data to be handled;
  - it takes too much time to receive a function value. This aspect is based on ‘network transmission errors’. This reason and the previous one are indistinguishable at the receiving side;
  - machine disfunctioning;
  - argument incorrect;
  - *Outside* functionality not present;
  - *Outside* functionality cannot be reached, another ‘network transmission error’.

  A solution for some of these problems is the introduction of a ‘watch dog timer’, to be set in `deposit`, and read in `deposit` or `accept`.

  It will be obvious that a bottom token is relevant in a lazy evaluation regime: if the environment permits so (e.g., in a parallel computational environment), parts of the computation may have been concluded satisfactorily, and if a function is not strict in the arguments yielding $\perp$, it can still have a function value $\neq \perp$. So $\perp$ can be (part of) the returned function value from *Outside*.

Intermediate List Structure The structure of the values in the stream of tokens (Twentel’s `Oargument` and `FVal`) handled by Loek, can be described formally in a syntax. These Twentel values are represented in a list structure, the Intermediate List Structure (ILS).

```
TransferValue ::= TokenList.
TokenList     ::= OpeningToken, TokenCdr.
TokenCar      ::= BasicToken, TokenCdr; TokenList.
TokenCdr      ::= TokenCar; ClosingToken.
BasicToken    ::= TStart, BasicTag, TSep, RealData, TStop; BottomToken.
BasicTag      ::= ‘Char’, ‘Real’, ‘Int’, ‘Bool’.
OpeningToken  ::= TStart, ListOpenTag, TSep, DataPadding, TStop.
```

```
ClosingToken  ::= TStart, ListCloseTag, TSep, DataPadding, TStop.
BottomToken   ::= TStart, BottomTag, TSep, DataPadding, TStop.
RealData      ::= ‘‘datapackage’’.
DataPadding   ::= ‘DC’.
TStart        ::= ‘<’.
TStop         ::= ‘>’.
TSep          ::= ‘&’.
ListOpenTag   ::= ‘(’.
ListCloseTag  ::= ‘)’.
BottomTag     ::= ⊥.
```

This syntax is used in `deposit` and `accept`. The `deposit` process handles sending of values in this structure, which means the conversion from internal Twentel format to ILS. The `accept` process covers the receiving of values in this structure, converting ILS to internal Twentel format.

One can imagine the token stream as two 'synchronous' parallel streams of tokens and data. Every token has its data item, or a 'DON'T CARE'. The tokens are fully self-documenting; the tag depending on the data path width or the bit organisation.[9]

Running Example To have a look at a value in the token stream as it is handled by Loek, we take as an example the first activation of *Outside* in the 'List Adder' example given in 5.1.1.1.

```
TWR LadId [[1,2,3],[4,5],[7,8,9]]
```

We represent the token stream in its token-form, reading from left to right, from top to bottom, ignoring white-space.

```
< ( & DC > < ( & DC > < I & 1 > < I & 2 > < I & 3 > < ) & DC >
          < ( & DC > < I & 4 > < I & 5 >          < ) & DC >
          < ( & DC > < I & 6 > < I & 7 > < I & 8 > < ) & DC >
< ) & DC >
```

However, this representation is 'packet'-based. A better idea of the two-dimensionality gives the following representation, with tag-channel and data-channel indicated, reading from left to right.

```
tag-channel  :   (  (  I  I  I  )  (  I  I  )  (  I  I  I  )  )
data-channel :   DC DC 1  2  3  DC DC 4  5  DC DC 6  7  8  DC DC
```

The linearity of our writing (and speaking) almost automatically causes a serial structure, manifest in the ILS syntax above. In retrospect we can remove some of the constructions from the syntax (viz `TokenStart`, `-Stop`, and `-Sep`). Without these constructions it is easy to see the parallel structure in the token stream, with the now-absent constructions acting as synchronisers between the two channels.

[9] As an example, one can code integers as follows: `I` = '01' byte integer, or `i` = '10' byte integer: the tokens can be considered self-typed data (cf 5.1.4).

Outside **Notation of Intermediate List Structure** The tokens received by *Outside* must be given meaning, be interpreted, in order to be considered an argument (list) for *Outside*. This can only be done if the token sequences adhere to a certain form, which can be given that meaning. We describe in *Outside* the token sequence(s) it can give a meaning to, and check whether the token stream actually consists of sequences in the correct form. Extracting the data from a correct token sequence yields a correctly formed argument for *Outside* in internal format. We will give the syntax for the description of this correct form. As we can give a context-free syntax, using such a notation enables us to automate the conversion of ILS format to and from the format which *Outside* uses internally. The conversion of ILS to and from an *Outside* datastructure is only given as an example for a Pascal record. The conversion from ILS format to a Pascal record is done in `startup`, and, likewise, converting a Pascal record to ILS format is done in `dispose`. Once more we see the currying of arguments in the only argument present at top level: a list.

```
ArgImage         ::= ArgList.
ArgList          ::= '(', ArgForm, ')'.
ArgForm          ::= ArgSequence; Indefinite, Type.
ArgSequence      ::= Arg, ArgSequety.
ArgSequety       ::= Separs, ArgSequence; ArgSequence; .
Arg              ::= Type; RepetitionFactor, Type.
Type             ::= BasicType; ArgList.
BasicType        ::= 'C'; 'R'; 'I'; 'B'.
Indefinite       ::= 'n'.
Separs           ::= ','; ' '.
RepetitionFactor ::= Number.
Number           ::= Digit, Digety.
Digit            ::= '1'; '2'; '3'; '4'; '5'; '6'; '7'; '8'; '9'.
Digety           ::= '0', Digety; Number; .
```

Had we introduced the indeterminacy (the `'n'`) at the `Arg` level, 'undecidable' token streams would have been the result. Let us consider the following situation, where `ArgForm` does not exist and its place in `ArgList` is taken by `ArgSequence`. If we have the description `"n(...a...)(...b...)"` and the situation in which a '`(`' after an `a`-sequence has been received, it is not decidable without full look-ahead (and even with it) whether the '`(`' belongs to a new `a`-sequence or to a `b`-sequence. To exclude this possible ambiguity we introduced the special `ArgForm` construction.

In what way the description relates to a record structure can be seen in the following example. Take the evaluation of the description `"(4C11IBB2R10CB)"` of a correct sequence of tokens coming in from Loek. This evaluation yields — if a correct sequence came in — a string of bytes in memory with an interpretation, an abstraction of this sequence of bytes, called a Pascal record. This physical representation might look like

"CCCCIiIiIiIiIiIiIiIiIiBBRrrrRrrrCCCCCCCCCCCB"

each character in this string occupying one byte in core, with "*C*" (character), and "*B*" (Boolean) occupying 1 byte, "*Ii*" (integer) 2 bytes and "*Rrrr*" (real) 4 bytes. Logically, a possible description of the Pascal record might look like:

```
type DemoPascalRecord =
                record
                   shortext  : packed array [1..4] of char;
                   length    : integer;
                   measures  : array [1..10] of integer;
                   filledup,
                      icdone : Boolean;
                   bottomval,
                      topval : real;
                   longtext  : packed array [1..10] of char;
                   last      : Boolean;
                end {DemoPascalRecord};
```

The description of a token stream can be called 'typing' of the datastructures which are transmitted through the trapdoor. Admittedly a low level typing, of lower level than the `DemoPascalRecord` above, but a typing nevertheless. It is related to the meaning of 'type' in the Pascal sense, where type is a mere description of how to interpret a bitstring in core; it is *not* the modern notion of type in the programming language sense (3.1.7). Such a type might entail sending functions through the trapdoor, for which as yet we have no solution.

With the introduction of an indeterminacy (the `Indefinite` reducing to `n` in the `Arg` production rule) we imposed asymmetric requirements on how the values in different value domains (e.g., Twentel and Pascal) are handled. In Twentel, values have a run time structure, an aspect of the run time type checking mechanism of the language, this free format run time structure being the cause of introducing this indeterminacy. In Pascal such a structure is completely absent: the way bitstrings (values) are to be interpreted at run time, is decided at compile time (unless variant records are used, then one can have a small number of interpretations of the bitstring, all fixed). We can abstain in Twentel from the typing or description of the values that Twentel sends to *Outside* as they are known. We only have to worry at the receiving side in Twentel, where the incoming values must have been tagged in order to maintain the correct run time structure of the imported values with their value tags in the graph space. In *Outside*, written in an imperative, strongly typed language, we have to be more precise: at compile time the exact layout or description of both the argument input record and the function value output record must be known, as there is no way of interpreting within the running functionality the bitstring which comprises the record. So we cannot describe dynamic arrays, neither on input, nor on output. That was the reason why the trapdoor application of the Graph Identificator in 4.2.3.1 was rewritten with 'end-of-cycle' zeros.

Running Example In the above 'List Adder' example (5.1.1.1) the description of the input record would look like `"(n(nI))"`, an indefinite number of lists, each list consisting of an indefinite number of integers. The output record description of the 'List Adder', `"(nI)"`, denotes a list with an indefinite number of integers. The Graph Identificator input record (4.2.3.1) would also have `"(n(nI))"`, and the output record with the graph identification number would have the description `"(3I)"`, a list of just three integers (or octals, masquerading as integers).

5.1.1.5 Twentel, the Functionality Machine, and *Outside*

In Figure 5.3 the cooperation of Twentel and *Outside* is given in a graphical way.

At the left side the domain is given: at the top Twentel with its graph reduction machine, below this the functionality machine that connects through Loek with *Outside*. Reading the figure should start in the top-left corner with the application of `TRAP Ofun [x:y]`. Twentel graph rewriting takes place at the second level. It activates the functionality machine with its related 'process terms' (viz `deposit`, `discard` and `accept`) and 'nether graph space'. The argument, converted to ILS structure, passes through Loek, and is subsequently converted to *Outside* format. Processing it in `providevalue` yields `FVal` which, converted to ILS structure, passes back through Loek, to be taken up by `accept` and subsequently put into the Twentel graph space, yielding the Twentel value, `FVal`. In this description we suppose that both argument and function value trays are initially empty and *Outside* is (eagerly) waiting.

5.1.2 The Model of *Outside*

The illustration of our thesis — Extending a functional language with a trapdoor construction increases the applicability of that language — not only requires a functional language, but also an outside language to serve as demonstration model. The choice of Twentel was based on the criterion of availability (2.2). The choice of Pascal as outside demonstration language is likewise based on availability. For the time being, we use a generic version of Pascal.

The programmer who is engaged in writing a Pascal program that eventually will run as an *Outside* functionality, or is engaged in adapting an existing Pascal program to run in conjunction with a master Twentel program as an *Outside* functionality, must adapt his program to the following Pascal execution model.

5.1.2.1 Interface of *Outside*

The Pascal programmer does not have to know the way the arguments he needs for his `providevalue` (Twentel's `Oargument`) enter his Pascal environment, nor does he have to know the way his results (Twentel's `FVal`) leave his Pascal environment.

Nevertheless, he has to know the input to his *Outside* function. Information regarding the input argument to `providevalue`, Pascal's `Irecord`, must be known in `startup`, such that `Irecord` can be constructed from the transmitted `Oargument` in ILS format. If `startup` and `providevalue` belong to the same process *Outside*, it is easy to convey this information to `startup` when *Outside* is built.

The same holds for the description of the output argument from `providevalue`, Pascal's `Orecord`, which structure must be known in `dispose` in order to convert it to the ILS format for Twentel's `FVal` needed by `accept`.

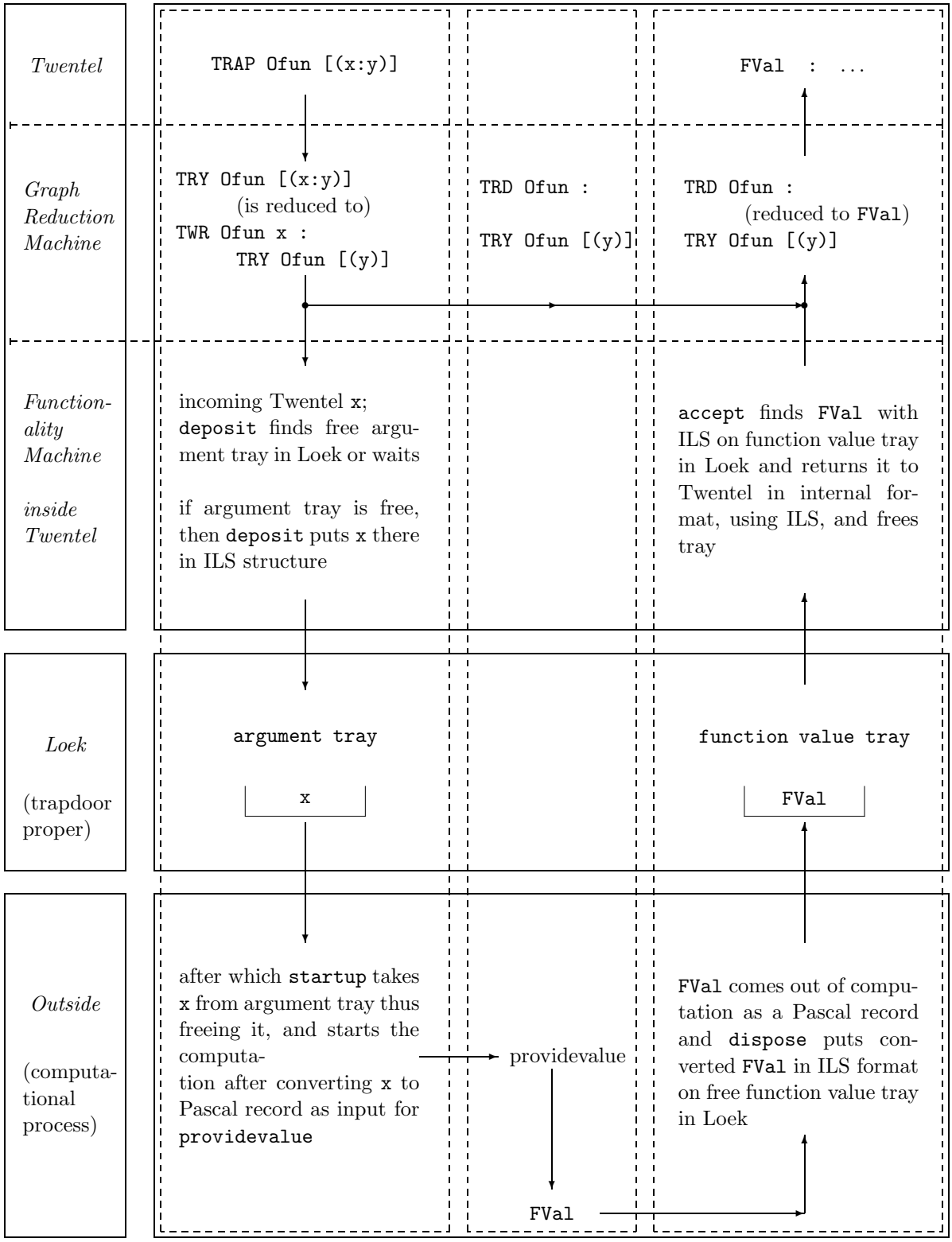

Figure 5.3: *Graphical description of Twentel with trapdoors and* Outside

It is evident that information regarding the interface to a function belongs to that interface — otherwise one cannot specify or even write the function. So extending the function with two interface adapters is a natural thing to do, thus hiding the language specific *Outside* interfaces from Twentel: the knowledge about input argument and output function value stays within *Outside*.

The only items the programmer has to provide are descriptions needed for the conversion from and to Twentel values given in ILS structure, and the input argument and output value of his `providevalue` process:

Idescription : a description of `Oargument`, the input which Twentel sends to *Outside* resulting in a Pascal record that is used as input argument;

Inputrecord : the Pascal type description of `providevalue`'s argument, the Pascal record `Irecord`;

Odescription : a description of the function value of `providevalue`, a Pascal record that is the function value `FVal`;

Outputrecord : the Pascal type description of `providevalue`'s function value, the Pascal record `Orecord`.

5.1.2.2 Construction of *Outside*

The kind of Pascal programming involved in writing the skeleton for `providevalue` is very easy, any programmer can create such a skeleton. It is easy to mold the intended functionality program in a simple '$f(I, O)$'-model, with record I in and record O out. For testing purposes the program f would be embedded in a loop, reading input records from a file (or keyboard) and writing the corresponding output records to another file (or screen).

In this vein a Pascal model for *Outside* can be constructed. This is possible through the use of variant records. Likewise, a model for C is possible, just as it is for APL (shared variables), Basic, FORTRAN (equivalence) (and of course, LISP). An ALGOL68 program cannot be made into an *Outside* functionality, as it is consistently strongly typed.

The Pascal model of an *Outside* functionality can be described by the following statements, a kind of shell around f:

```
begin {mainprogram model}
    type Description;  {the character based description, string type}
    type Inputrecord;  {the Pascal representation of the input}
    type Outputrecord; {the Pascal representation of the output}
    var Irecord : Inputrecord;
    var Orecord : Outputrecord;
    var Idescription, Odescription : Description;
    begin
        while GetInputFromTwentel (Idescription, Irecord) do
           begin
```

```
            providevalue (Irecord, Orecord);
            ConvertOutputRecordForTwentel (Odescription, Orecord);
         end;
end.
```

In this model the `while`-loop takes care of restarting `startup` after `dispose` has sent the function value to Twentel. Both `GetInputFromTwentel` and `ConvertOutputRecordForTwentel` are taken from a standard Pascal library which belongs to this particular implementation of Twentel. The only procedure the programmer himself has to write is `providevalue`, the program f from above.

We skip the description of the realisation of `GetInputFromTwentel`. It might be realised by yielding `false`, when it receives a special (e.g., null) `Irecord`, and thus stopping the *Outside* process, or it can be realised by (eternally) internal waiting for a new `Irecord`, thus effectively idling *Outside*.

Running Example Of course, the programmer does not have to call `providevalue` by that name in the body of his Pascal program: he can see to that as he is the composer of the program. However, the other two procedures — as they are taken from a standard library — *do* have to be called verbatim. We will use the Graph Identificator as an example.

```
begin {mainprogram GraphIdentificator}
    type Description;  {the character based description}
    type Inputrecord   {the Pascal representation of the input}
            = record
                 code  : packed array [1..100] of integer;
              end {Inputrecord};
    type Outputrecord  {the Pascal representation of the output}
            = record
                 idget : packed array [1..3] of integer;
              end {Outputrecord};
    var GraphDescriptionRecord : Inputrecord;
    var IdGetRecord : Outputrecord;
    begin
        while GetInputFromTwentel ('(n(nI))', GraphDescriptionRecord) do
           begin
              ProvideIdentificationNumber (GraphDescriptionRecord,
                                           IdGetRecord);
              ConvertOutputRecordForTwentel ('(3I)', IdGetRecord);
           end;
end.
```

This is the complete Pascal shell of an *Outside* functionality for the Graph Identificator. Similar shells can be produced for C, FORTRAN, *&c.*

5.1.2.3 The ILS to Pascal Conversion Algorithm

In 5.1.1.4 we gave the syntax for values transferred through the token stream. Now we present the algorithm that handles the conversion from ILS structure to Pascal record structure. We do so in Twentel, as we wish to demonstrate once more the naturalness of programming in a functional language.

The conversion algorithm, `ConverttoRecord`, is used in `startup`. The algorithm takes as input a stream of tokens, an outside ('encapsulated') phenomenon, the data of which are moved into a Pascal record, a list of bytes (`Irecord`). This record is formed according to the description (`IDescription`) that is also an argument for the conversion. The description is a string of characters described by the `ArgImage`-syntax (5.1.1.4, e.g., `"(3IB(nI)2(IC))"`). The conversion is aborted and stops with an errormessage when an incorrect description of a token sequence is given, or when an incorrect token appears in the token stream. The algorithm stops with a Pascal record when a correct token sequence has been read.

With the introduction of `ArgList` and `AcceptFun` as extra functions to specify the two underlying processes to be distinguished (thus enhancing comprehensibility, 3.1.3.1), we can informally represent this conversion algorithm as follows:

```
ConverttoRecord  IDescription  TokenStream  =  Irecord
   WHERE
      AcceptFun  =  ArgList    IDescription,
      Irecord    =  AcceptFun  TokenStream
   ENDWH
```

This being only a sketch of how to solve the problem at hand, one has to take into consideration that these functions will turn out to be more elaborate than described here.

The first process (`ArgList`) is the evaluation of the description: what succession of tokens constitutes a correct token sequence. Its function evaluates its argument, a specific description, according to the syntax underlying the description (the `ArgImage` syntax). We can consider this process as being a tree transducer with the tree given in a linear representation made possible by the use of parentheses, viz the description. This tree has as its leaves the tokens to accept in a correct sequence, the sequence encountered when traversing the tree depth-first. So this process starts with converting the description into a kind of token tree adorned with semantic instructions represented as a function.

This token tree is used as a description of the second process (`AcceptFun`) that accepts tokens from the token stream and moves the data to a Pascal record. It is a higher order construction that is a representation of a correct token sequence, so it can be regarded as the recogniser for the specific token sequence given by `IDescription`. This recogniser function is applied to the token stream and produces the byte string in core while visiting the leaves of the tree and at the same time accepting tokens from the stream.

However, before discussing the set of functions, necessary for implementing `ConverttoRecord`, let us, as engineers should do, check whether the dimensions of our computations are right. The natural sciences' 'dimension' is called 'type' in programming (3.1.7).

`ConverttoRecord` is a function of type [tag] → [token] → [byte]. We realise these types as follows:

tag is represented by the Twentel characters: `‘(`, `‘C`, `‘R`, `‘I`, `‘B` and `‘)` (for theoretical reasons we cannot include the bottom tag in our discussion). To increase the legibility we use the Twentel characters `OpenB` and `CloseB` instead of the confusing `‘(` and `‘)`;

token is represented by tuples of type (tag, byte);

byte is represented by the 'bytes' of the 'datapackage' in the token, an ISWIM type (cf `RepFact` below that is an ISWIM function [13]). The `‘C` token occupies one byte, `‘R` four, `‘I` two and `‘B` one byte.

Mark that typing obeys the currying rules: the unary function `ConverttoRecord` applied to `IDescription` of type [tag] yields a function F of type [token] → [byte] and this function F applied to `TokenStream` yields the eventual input record of type [byte].

`ArgList` Process Preliminaries The first process 'consumes' the description. The basic function that consumes characters from the description is `AC`, 'accept a character from the description'. It is of type tag → [tag] → (boolean, accpfun, [tag]). The functions with type accpfun are of type [token] → (boolean, [byte], [token]). This kind of function will be discussed below where the complete converter is given.

```
AC  xch  D          =  IF  cch  =  xch  ->  [ True, AT  xch, rD ],
                                        ->  [ False, I, D ]     FI
                       WHERE
                             ( cch : rD )  =  D
                       ENDWH,
```

To facilitate the construction or description of the first process we introduce the following auxiliary syntax functions of the acceptor builder:

- `alternate`, the decision function, whether the first or second alternative of a syntax rule was used in describing the syntax tree given by the description;
- `sequence`, the combining function that combines two syntactic notions in describing the syntax tree.

These higher order syntax functions are of type syntaxfun → syntaxfun → [tag] → (boolean, accpfun, [tag]). The first order syntax function (denoted by `f` and `g` in the following definitions) is of type [tag] → (boolean, accpfun, [tag]). This kind of basic function will be elaborated below.

```
alternate  f  g  D  =  IF  ftrue  ->  [ True, ffun, rD ],
                                  ->  g  D            FI
                       WHERE
                             [ ftrue, ffun, rD ]  =  f  D
                       ENDWH,
sequence   f  g  D  =  IF  ( ftrue  &  gtrue )
                              ->  [ True, concat ffun gfun, rrD ],
```

```
                    ->  [ False, I, D ]                FI
            WHERE
                  [ ftrue, ffun, rD ]   =  f  D,
                  [ gtrue, gfun, rrD ]  =  g  rD
            ENDWH,
```

To stay in character (ease the construction process) we will also use a `list` function, though this function is not necessary as is clear from the fact that only already known constructions are used in assembling it.

```
list      f     D  =  IF  ( otrue  &  ftrue  &  ctrue )
                      ->  [ True, concat ofun (concat ffun cfun), rrrD ],
                      ->  [ False, I, D ]                              FI
                      WHERE
                            [ otrue, ofun, rD ]    =  AC OpenB D,
                            [ ftrue, ffun, rrD ]   =  f  rD,
                            [ ctrue, cfun, rrrD ]  =  AC CloseB rrD
                      ENDWH,
```

`AcceptFun` Process Preliminaries The higher order building blocks that are used in constructing the description specific acceptor function are `concat`, `repeat` and `oncemore`.[10] The main, higher order, acceptor function (`concat`) is of type accpfun → accpfun → [token] → (boolean, [byte], [token]). The type of the two other higher order functions can be derived from it. The basic acceptor functions (denoted by `f` and `g` in the next set of definitions) are of type [token] → (boolean, [byte], [token]). The function value triple of the acceptor functions consists of:

`boolean`, whether the acceptor function has been successfully applied to its argument;

`[byte]`, the result list of bytes in core that is extracted by the acceptor function from the accepted tokens in the token stream argument;

`[token]`, the rest of the token stream, remaining after the acceptor function has been successfully applied to it, having 'consumed' all tokens the acceptor function was built for, or the original token stream in case of failure.

The main constructor `concat` is used to combine the result byte list of two acceptor functions into a larger byte list, with both functions of course possibly composite. The use of `repeat` and `oncemore` will be discussed when we present the complete converter. `nop` ('no operation') is a basic acceptor function, necessary for the correct handling of the `Empty` rule.

```
concat   f  g  T  =  IF  ftrue  ->  [ gtrue, fbytes++gbytes, rrT ],
                                ->  Terrorexit  "concat"  T      FI
                     WHERE
                           [ ftrue, fbytes, rT ]   =  f  T,
                           [ gtrue, gbytes, rrT ]  =  g  rT
                     ENDWH,
```

[10] The special '`fun f g F`' construction was suggested by Gerrit van der Hoeven.

```
repeat   n  f  T  =  IF  ftrue  ->
                         IF  n  >  1  ->  [ rtrue, fbytes++rbytes, rrT ],
                             n  =  1  ->  [ True, fbytes, rT ],
                                      ->  Terrorexit  "repeat"  T
                         FI,
                                ->  [ False, [], T ]
                     FI
                     WHERE
                           [ ftrue, fbytes, rT ]   =  f  T,
                           [ rtrue, rbytes, rrT ]  =  repeat (n-1)  f  rT
                     ENDWH,

oncemore    f  T  =  IF  ftrue
                     ->  IF  otrue
                         -> [ True, fbytes++obytes, rrT ],
                         -> [ True, fbytes, rT ]       FI,
                     ->  [ False, [], T ]              FI
                     WHERE
                           [ ftrue, fbytes, rT ]   =  f  T,
                           [ otrue, obytes, rrT ]  =  oncemore  f  rT
                     ENDWH,

nop            T  =  [ True, [], T ],
```

We also need an AcceptToken function (`AT`), with `xtag` the tag of the expected token. This function extracts the number of bytes from a token according to the tag of the token ('type', see 5.1.1.4), so `OpenB` and `CloseB` have no data associated.

```
AT  xtag       T  =  IF  xtag  =  tag  ->  [ True, recbytes, rT ],
                                       ->  Terrorexit ["AT",[xtag]]  T  FI
                     WHERE
                           ( token : rT  )    =  T,
                           [ tag, tokbytes ]  =  token,
                           recbytes = IF MEMBER "()" tag -> [],
                                                      -> tokbytes  FI
                     ENDWH,
```

The following observations can be made when looking at the acceptor building blocks and the auxiliary syntactic functions.

- `sequence` generates a `concat` construction, as it is evident that the sequence of syntactic notions will be seen again in the same sequence of tokens in the stream;

- `alternate` does not generate an acceptor building block itself. It cannot start a construction as it does not know what to construct; this building block generation is delegated to the alternates;

- `list` is a compound of `sequence` and `AT` and could also be written as:
  `sequence (AC OpenB) (sequence ArgForm (AC CloseB))`.

The Complete Converter After these preliminaries we are ready to present the converter. In the description of the first process we see the `ArgImage` syntax reappear. The syntactic functions (the `f` and `g` from `alternate` and `sequence` above) embed semantic instructions on what to do with the tokens, instructions that are combined into an acceptor function. These basic syntactic functions are of type [tag] → (boolean, accpfun, [tag]). The function value triple consists of:

`boolean`, whether the syntactic function has been successfully applied to its argument;

`accpfun`, the result function that, when applied to a proper token stream, extracts from it the bytes corresponding to the part of the description that was accepted by the syntactic function. Such a function is a building block of the token stream acceptor function;

`[tag]`, the rest of the description, remaining after the syntactic function has been successfully applied to its argument, or the original argument in case of failure.

The conversion is started by applying the syntactical function `ArgList` to the description. Recursively, according to the syntax rules, the other syntactical functions are applied to the description `D`. While consuming as much of the description as is implied in its syntax rule, the syntactical function yields the specific acceptor or recogniser for that piece of the description. Combining these pieces by `concat` yields the total recogniser. The other syntactical non-terminal functions follow (for the sake of brevity, `ArgSequence`, `BasicType` and `ArgSequety` were abbreviated). Note that we introduced two new non-terminals (`ArgForm2` and `Arg2`) in order to keep the `alternate - sequence` constructions clear and in order to move the special constructions `oncemore` and `repeat` to single purpose functions.

We can now present the full `ConverttoRecord`, as the construction of the underlying processes is clear. We also introduce full error control.

```
DEF
    ConverttoRecord  IDescription  TokenStream  =
       IF  ( ftrue  &  (EMPTY  rD) )  ->  Irecord,
                                      ->  "Error : " ++
                                          IDescription ++ [NL] ++ rD  FI
       WHERE
             [ ftrue, ffun,    rD ]  =  ArgList  IDescription,
             [ bool,  Irecord, rT ]  =  ffun  TokenStream
       ENDWH,
```

Subsequently we have (omitting the handling of the trivial `Separs`):

```
    ArgList  D    =  IF  ftrue  ->  [ True, ffun, rD ],
                                ->  [ False, I, D ]  FI
       WHERE
             [ ftrue, ffun, rD ]  =  list  ArgForm  D,
       ENDWH,
```

```
ArgForm   D   =  alternate  ArgSeq  ArgForm2  D,
ArgSeq    D   =  sequence   Arg     ArgSeqY   D,
Arg       D   =  alternate  Type    Arg2      D,
Type      D   =  alternate  Basic   ArgList   D,
Basic     D   =  alternate  (AC 'C)
                 (alternate (AC 'R)
                 (alternate (AC 'I)
                            (AC 'B)))         D,
ArgSeqY   D   =  alternate  ArgSeq  Empty     D,
Empty     D   =  [ True, nop, D ],
```

Sometimes a 'syntactic unit' from the description must be extracted a specified number of times (as in the example, `'IC'` twice) from the token stream. This is realised by means of `repeat`, `RepFact` being the converter from digit characters to a number. This function presupposes many a thing, it is an ISWIM function, just like `Number`.

```
Arg2      D   =  IF  Number  ch  ->  [ ftrue, repeat  nr  nrfun,  rrD ],
                                 ->  [ False, I, D ]                 FI
                 WHERE
                       ( ch : cD )            =  D,
                       [ ftrue, nrfun, rrD ]  =  Type  rD,
                       [ nr, rD ]             =  RepFact  0  D
                 ENDWH,
```

`oncemore` is instrumental in extracting an undetermined number of times a 'syntactic unit' from the token stream (as in the example, `'I'`):

```
ArgForm2  D   =  IF  ch  =  'n  ->  [ ftrue, oncemore  indefun,  rrD ],
                                ->  [ False, I, D ]                 FI
                 WHERE
                       ( ch : rD )              =  D,
                       [ ftrue, indefun, rrD ]  =  Type  rD
                 ENDWH
FED
```

Two more observations can be added to our list when looking at the complete recogniser generator:

- `Arg2` (in the handling of `RepFact`) generates a `repeat` building block;
- `ArgForm2` (in the handling of `indefinite`) generates a `oncemore` construction.

This concludes the discussion of the construction of a solution. The solution was built with these functions and ran with a simulated token stream and input record under Twentel.

In the Pascal library procedure `GetInputFromTwentel` that uses this algorithm, we 'encapsulate' the token stream; the stream is taken care of by the system.

```
procedure GetInputFromTwentel (IDescription: string; var Irecord);
```

5.1.2.4 The Pascal to ILS Conversion Algorithm

The inverse conversion — from the Pascal record `Orecord` to ILS structure — moves the bytes in the record to tokens in the token stream. This inverse algorithm is used in `dispose`. It has `Orecord` as input, the bytes of which are converted to the data of a stream of tokens ('encapsulated' output) under control of the description, called `ODescription`. This description is given as first argument.

We present, once more in an informal Twentel, the architecture of the conversion of Pascal record structure to ILS structure, to be used in the construction of the other main Pascal procedure `ConvertOutputRecordForTwentel`.

```
ConverttoTokens  ODescription  Orecord  =  TokenStream
   WHERE
      GenFun      =  ArgList   ODescription,
      TokenStream =  GenFun    Orecord
   ENDWH
```

As in `ConverttoRecord`, the description of the bytes in the record (`ODescription`) is converted into a recogniser function, now a token stream generator instead of an acceptor, to operate on the bytes in the output record (`Orecord`) (with `B` as short form (for 'bytes')) yielding the token stream (`T` for short), the function value of `ConverttoTokens`. The auxiliary syntactic functions (e.g., `alternate` and `sequence`) are of course the same as with `ConverttoRecord`, with one exception: `AC`. AcceptCharacter must now generate a Send Token (`ST`) instead of an Accept Token (`AT`).

```
AC  xch  D   =  IF  cch  =  xch  ->  [ True, ST  xch, rD ],
                                 ->  [ False, I, D ]     FI
                WHERE
                      ( cch : rD )  =  D
                ENDWH,
```

As before, the higher order building blocks that are used in constructing the generator function that is specific for the description given as argument, are `concat` and `repeat`, without `oncemore`, and with `ST` (SendToken) instead of `AT`. Of course these functions have the same model as in the previous algorithm. They only need a name (and type) change of their last argument (the byte stream `B` instead of the token stream `T`) and of an internal quantity (the result `ftokens` instead of the result `fbytes`). These functions are of type genfun → genfun → [byte] → (boolean, [token], [byte]), with genfun the same kind of type as accpfun.

The only really different function is the SendToken function, with `stok` the tag of the token to be sent. No check can be performed, as the record byte string does not carry a description of the items in it. `extractbytes` is another ISWIM function, it extracts as many bytes from the record as needed, according to the tag, and returns these bytes together with the rest of the record; an `OpenB` or `CloseB` token to be sent does not generate data, so there will be no adjustment of `B`.

```
ST  stok  B  =  [ True, [token], rB ]
                WHERE
                      [ tokbytes, rB ]  =  extractbytes  stok  B,
```

```
                    token              =  [ stok, tokbytes ]
                ENDWH,
```

With the necessary building blocks in place, we are ready for the converter itself. As the syntax of the underlying conversions is the same, their recogniser function generator is the same. In the case of Pascal this is almost the same. In contrast with `ConverttoRecord`, our task is a little lighter on this side: no indefinite constructions can occur in Pascal (5.1.1.4), which means that the `ArgForm` handling function is simpler. So it suffices to present only the top level function `ConverttoTokens`.

```
DEF
    ConverttoTokens  ODescription  Orecord  =
       IF  ( ftrue  &  (EMPTY  rD) )  ->  TStream,
                                      ->  "Error : " ++
                                          ODescription ++ [NL] ++ rD  FI
       WHERE
             [ ftrue, ffun,    rD ]  =  ArgList  ODescription,
             [ bool,  TStream, rB ]  =  ffun  Orecord
       ENDWH
FED
```

Here ends the discussion of the construction of this part of the solution. We also tested this solution under Twentel with a simulated output record and token stream.

Once more, as we did in `GetInputFromTwentel`, we 'encapsulate' the token stream in the Pascal library function to be used in a Pascal *Outside* functionality.

```
procedure ConvertOutputRecordforTwentel (ODescription: string; var Orecord);
```

5.1.2.5 Discussion of the Conversion Algorithms

Looking back on both converter functions, it is obvious that a large, parameterised function to generate a converter can be made if the whole process would have been used in Twentel itself. The arguments to this giant converter generator function would be mostly the syntax for the description, and the specifics of the acceptor and generator function building blocks like `concat`, `repeat` and `oncemore` with their 'coerced' arguments and internal quantities, and with `AT` and `ST`. The syntax argument yields the specific '`ArgList`' function that, in its turn, generates the acceptor and generator functions (`AcceptFun` and `GenFun`) based on the given descriptions.

However, the given functions are the model, or prototype, of their Pascal (or another language) counterparts, as the conversion has to take place in the Pascal environment. Converting the Twentel algorithms into Pascal will be fairly straightforward (it is mostly recursion while creating a construction in core) but for the higher order function that processes the token stream or the output record. The most simple realisation for this construction is to create a description ('coding') of the functions to be called with their arguments and then to build a description interpreter on this simple 'language' that processes the token stream or output record.

5.1.3 Synchronisation Problems

Waiting problems are a part of communicating processes. Quite another aspect of communicating processes is formed by synchronisation problems. In this Subsection we look at three different kinds of synchronisation-like problems, and where necessary, we present a (theoretical) solution for the problem.

5.1.3.1 Overtaking Trapdoors

Twentel cannot get into synchronisation problems that are due to activating a trapdoor. If a trapdoor has been activated then, by definition of the graph rewriting algorithm that implements the lazy evaluation semantics (3.4.5), that activation occurs within `TRAP` (or more precise, `TWR`) at the tip of the spine. As we do not rearrange the graph above the `TWR` nodes, `TRD Ofid` also appears at the tip of the spine and will wind up as candidate in the straightforward chain of subsequent reduction steps. As this application, `TRD Ofid`, is of the same type as the previous one, which was `TWR Ofid Oargument`, it will be the first one to be reduced after reducing `TWR`. So there is no possibility of one activation of `TRAP` getting the value of the activation of another `TRAP` of the same functionality. This argument must be reconsidered in the case of parallel processing of the program graph.

5.1.3.2 Recursive Trapdoors

Recursion in trapdoor activation sequences can occur, as is shown in the following examples:

```
TRAP "x" [ ... [ TRAP "y" ... ]  ... ]
TRAP "x" [ ... [ TRAP "x" ... ]  ... ]
```

Both constructions are permitted, as an argument to a trapdoor is strict. This means, the inner `TRAP` application is fully evaluated in both cases, before the outer `TRAP` application is evaluated.

5.1.3.3 Unclaimed Resources

The problem we discuss in the following paragraphs is not a theoretical but a very practical one, and it is related to the way the Twentel system is built. It manifests itself in succinct form in the Twentel program we will discuss shortly. After a discussion of the problem, and related problems, we will present an elegant way out of these difficulties, which even leads to a 'call-by-need' Pascal implementation.

Problem Description We created a means for communicating with other environments within the language for (different kinds of) efficiency reasons. Writing a program in a lazy functional language the evaluation order is of no concern. But in Twentel extended with a trapdoor an evaluation order problem arises, due to a realisation issue: the limiting size of the buffers involved; the implementation is not transparent any more. One wants to prevent such problems.

In this example program we use a simplex trapdoor (4.4.1), as the only reason for the existence of this *Outside* functionality is to provide Twentel with a very large array (or rather, list) of integers (say, one million).

```
DEF  somefunc larray = IF larray^10 = 0 -> larray^0,
                                        -> SUM larray FI
FED
DO somefunc ( TREAD "make-array-in-outside" ) ? OD
```

The computation starts with getting a very large number of integers from *Outside*. As the realisation is not equipped with an infinite buffer, it so happens that only the first part of the array will be sent to Twentel. If after some time (here modelled by the inspection of the tenth element) it turns out that the complete array is not needed, what will happen to the part of the array which is not yet sent to Twentel?

The *Outside* part of the trapdoor, `dispose`, is waiting to send the next part of the array to Twentel: `dispose` puts it upon the function value tray and waits for it to be taken away by Twentel. On the other side, Twentel will never activate `accept`, because the part of the graph space which contained the the `accept`-activating `TRD`, does not belong to the spine any more; and so it became garbage, to be collected.

In theory (the architecture) there is no problem: the computation stops and it does not deal with implementation problems of still open resources. The model does not fully tally with reality, because the model is built upon the atomicity of `FVal`.

A more practical version of this problem presents itself in the handling of database queries. If after some time the Twentel program has processed enough of the results of a database query sent through the trapdoor, the graphnode in which this query was activated will become a candidate for garbage collection. This happens when a Twentel program, after having processed 1 Mbyte of a total of 12 Mbyte of query results, and given the results of a computation on parts of the processed records, decides not to bother with the rest of the query results.

Problem Discussion The problem arises because in the initial implementation model `FVal` is undivided. This is a logical consequence of the trapdoor architecture, where nothing is said about delivery of values. Values are 'atomic', an intuitive expectation, it may take time to get or print it because it is large, but it is not naturally divisible. In this model `FVal` is put in atomic state on the function value tray. However, as the function value tray is not infinite, one might regard it as a buffer. So in reality ('realisation'), `FVal` is presented to Twentel in pieces; it amounts to a buffersize problem. This can be modelled in a refinement of the initial implementation (an iterative process). The buffersize problem is in accordance with the way stream input (e.g., from keyboard) is handled in the Twentel system.

This is a typical Pascal problem [9] which also shows up in the architecture of the Twentel system. In this Twentel implementation a stream of characters is not presented to the program on a true character to character basis. Twentel gets control over its input on a character to character basis only then, when either

- there has been a `CR/LF` in the input stream;

- or more than 255 characters without a `CR/LF` are read in from a file;
- or the last character read is a `Ctrl-Z` (which is an MS-DOS End-of-File).

As we do not want to burden the programmer with foreseeing and solving this kind of problems again and again — though, admittedly, it is common practice: letting the programmer solve the problems of the 'other layer' programmers, or programming around the problem — the following system solutions might be considered:

- The most logical way is extending the garbage collection of Twentel graph nodes which refer to external processes, by sending a stop signal to the external process. Upon reception of this signal *Outside* the process discards the not yet dispatched part of `FVal`, and returns to its initial state. This means the construction of a new interrupt channel in the trapdoor. It must be separate because of the strict separation of functions of the two other channels: tag- and data-channel.

  However, we cannot trust the garbage collector to perform the task of freeing those nodes, as it is a fully autonomous process onto which no extra requirements can be put, especially not those which relate to timing. Using a reference counter garbage collector would entail a rewrite of quite a part of the Twentel system which we will not consider;
- Emptying the function value tray if a new argument is presented on the argument tray. Obviously this will lead to a 'deadly embrace';
- The minimal solution is to reset the active trapdoors when reaching the system level again (this measure has already been implemented at the moment) but for our problem this is no solution at all: there may be another reference to the trapdoor under consideration somewhere deep in the graph, or it can be too late;
- Introduction of another combinator does not work, because the action one wants to happen should be hidden from the programmer, and the system does not know which trapdoor to close: a pending trapdoor is not necessarily pending idly.

Looking back at the problem proper, we see that it only exists if `larray[10] = 0`. In that case the initial meaning of the activation of *Outside* was essentially wrong: it should not read 'Get the complete array', but rather 'Get the first 10 elements'. Paraphrasing this, 'Get the complete array' could be worded as " 'Get the first 10 elements' and afterwards, 'Get the rest of the array' ". The trapdoor should not give Twentel access to data it does not require. In a very roundabout way, we are describing here 'lazy semantics' for the output side of *Outside* (5.1.1.2), or 'weak lazy semantics' as it is not quite lazy. Sometimes it still generates (a little) too much data. This situation, however, can be handled by Twentel itself, as all generated data are inserted in the Twentel graph space by `accept`. In the graph space they are subject once more to the normal lazy semantics and the normal garbage collection processes.

The Functionality Machine Revisited We now refine the process of receiving the right function values from *Outside*. We need an identification of computed `FVal`'s, such that the `TRD` is correctly paired with its expected `FVal`, no matter how many interruptions take place.

If we look at the figure in 5.1.1.5 the solution is clear. Apparently the problem exists only between `accept` and `dispose` (or `discard`). But this is deceptive: in 4.2.3 we stated that there

was no communication between Twentel and *Outside*, other than via argument and function value. So there cannot be any communication between `accept` and `dispose` on the necessary identification of the desired `FVal`.

Looking at the figure we see two threads emerging from Twentel at the top left corner: one on top, staying in Twentel (graph space), being Twentel rewrites; the other one going down through Loek, *Outside*, and back again through Loek into Twentel, being the argument and function value passed on. So we identify the needed `FVal` at its inception, viz the `TWR` reduction. We pass the identification as an extra argument down to *Outside*, and straight to `TRD`. In the reduction of `TRD`, (handled by `accept`) the two threads join again, enabling the system to match both identifications.

For the identification we introduce a trapdoor 'activation sequence number', a unique, internal number (e.g., generated by the reduction counter) belonging to the `TWR`-reduction under consideration. It is unchanged during the sequence of reductions from `TWR Ofid x` until `FVal`. By sending it over to *Outside*, `startup` passes it on to `dispose` through a 'communication area', and `dispose` can use it to label its current `FVal`.

The communication area is an area, global to the *Outside* functionality; this is possible as `startup` and `dispose` belong to the same overall program (5.1.1.2). The communication area contains, among other items to be described shortly, the label — 'activation sequence number', `n` — of the currently computed `FVal`.

The process can now be described by changing the rules from 5.1.1.2, such that the identification and the buffer management come into play. We abstract from the identification problem in the following manner. The single function value tray is replaced by as many function value trays as there are `FVal`'s to be computed. `dispose` labels its (partial) function value with its activation sequence number from the communication area. The empty function value tray that `dispose` is waiting for, before it can dispose of `FVal`, is now the empty correctly labeled tray. `accept` searches for its `FVal` at the correctly labeled function value tray. The abstraction from the buffersize problem is taken care of by adapting `accept`.

Another action must be taken into account, an action initiated by the Twentel system when it returns to its initial state modulo static definitions in the graph space. Returning to its initial state, the Twentel system, among other things, clears all garbage, resets open files, and sends a reset signal to active *Outside* functionalities through their trapdoors. This reset signal from the Twentel system clears by definition all active trapdoor processes. This indication is sent to *Outside* as a spurious argument, properly labeled to make sure that `startup` does not activate `providevalue`, and sets the indication in the communication area instead. Also the signal from the garbage collector, that a trapdoor process with activation sequence number `n` can be collected, will be handled by `startup` by setting an indication relating to `FVal` with number `n` in the communication area. The next time `dispose` is started, the necessary cleaning up takes place.

The Duplex Trapdoor The rewrite rules are adapted to the new implementation model, taking into account the extra argument `n`, the activation sequence counter. The definition of `WAITETY` is as above (5.1.1.2). Most processes have the same functional behaviour as before, but

differ in their behaviour regarding the sending or receiving of the new argument. The changed rewrite rules 'under'

```
TWR  Ofid  x            --> ... -->  TRD  Ofid
```

now are the following (unchanged rewrite rules not repeated):

```
TWR  Ofid  x                    ==>  deposit  WAITETY  Ofid  x  n
deposit  v  Ofid  x  n          -->  TRD  Ofid  n

start-of-process                -->  startup  WAITETY  x  n
startup  v  x  n                -->  providevalue  INPETY  x  n
providevalue  v  x  n           -->  dispose  WAITETY  FVal  n
dispose  v  FVal  n             -->  start-of-process

TRD  Ofid   n                   ==>  accept  WAITETY  Ofid  n
accept  v  Ofid  n              -->  FVal
```

We do not have to incorporate the search for the correctly labeled function value tray into the rewrite rules. However, the solution for the buffersize problem must be taken care of. So we adapt

```
accept  v  Ofid  n              -->  FVal
```

with the following remarks on the buffer handling implementation. `accept` tallies the open- and close-parenthesis tokens in the token stream. As every argument is enclosed in at least one pair of parentheses, a complete `FVal` is received iff the parenthesis counts match. So the following rewrite rules apply now:

(1) : if a partial `FVal` is received, and more is to come:

```
accept  v  Ofid  n              -->  FVal : ( TRD  Ofid  n )
```

(2) : if the complete (all of) `FVal` is received, or the rest of `FVal`, which completes `FVal`:

```
accept  v  Ofid  n              -->  FVal
```

These changes in the implementation model imply that `dispose` gets the burden of implementing the 'weak lazy semantics' in the realisation. It has to stock the various parts of computed `FVal`'s which are not yet taken over by `accept`, and it has to check the various trays on whether a tray has been emptied.

The Simplex Trapdoors A few words have to be devoted to the adaptations of processes and rewrite rules underlying the `TRITE` and `TREAD` trapdoor combinators. A solution for these processes was 'tacitly' ignored in 5.1.1.3.

A solution for the `TRITE` process is to transfer a zero as fourth argument to *Outside* (the reduction counter cannot be zero when encountering a trapdoor combinator of this kind). `discard` does not have to change.

```
TWRI  Ofid  Oargument           ==>  discard  WAITETY  Ofid  Oargument  0
```

The `TRITE` and `startup` rewrite rules do not change essentially. The only rule that changes is related to `providevalue`, as the resulting `FVal` must not be transmitted to Twentel. So we have to introduce an `Ostart-of-process` with some related processes in order to distinguish between the duplex model and the simplex model.

```
Ostart-of-process              -->  startup  WAITETY  x  n
startup  v  x  0               -->  Oprovidevalue  INPETY  x
startup  v  x  n               -->  providevalue  INPETY  x  n
Oprovidevalue  p               -->  Oprovidevalue
Oprovidevalue  v  x            -->  Ostart-of-process
```

In case of `TREAD`, no matching of identifications can be carried out as there is no identification generating `TWR` involved. A solution can be obtained by extending the labeling range of function value trays with negative labels (remember the (positive) reduction counter was used as label in the duplex case). `TREAD` uses a zero as 'activation sequence number' argument of `TRD`.

```
TREAD  Ofun                    -->  TRD  Ofid  0
TRD  Ofid   0                  ==>  accept  WAITETY  Ofid  0
```

We adapt `accept` in the following way, changing its behaviour on the zero as fourth argument. Instead of looking at the positive-labelled trays, `accept` now looks at negative labelled trays starting from 0 downwards, and takes the first non-empty one that has not been handled by `accept`. It fetches `FVal` from that tray, and resets the '`accept`-handled' indication. The label of the tray (the negative number) is used in subsequent rewrites as the 'activation sequence number' `n`. The rewrites of the nether graph space are according to the following adapted rewrite rule.

In case a partial `FVal` is received, with `n` taken from the tray label, then:

```
accept  v  Ofid  0              -->  FVal : ( TRD  Ofid  n )
```

In case of the receipt of a complete (all of) `FVal` there is no change in the behaviour of `accept`.

The *Outside* functionality of this Input-kind is described by the following rewrite rules:

```
Istart-of-process               -->  startthrough  n
startthrough  n                 -->  Iprovidevalue  INPETY  n
Iprovidevalue  p                -->  Iprovidevalue
Iprovidevalue  v  n             -->  dispose  WAITETY  FVal  n
dispose  v  FVal  n             -->  startthrough (n-1)
```

When the functionality is started, it gets a negative unique number (e.g., a timestamp) which will act as initial 'activation sequence number'. The range of function value trays is extended with negative numbered trays, and `dispose` handles these trays just like the positive numbered ones. `dispose` does not activate `start-of-process`, as is the case with a positive fourth argument, but activates `startthrough` with decremented argument.

5.1.4 Other Machine Environments

With our ILS solution we take hardware representation problems aboard: viz the fact that an integer in the Digital PDP environment looks different from an integer in the IBM environment, not to mention the integer in the Digital Vax environment. This difference has also been called the big- versus the little endians controversy[11] [3, 10].

In the little endian model, the least significant byte is at the lowest address in memory, that is, the information that is displayed first when reading a memory dump upwards from address 0, from left to right on the screen. In the big endian model, the most significant byte is at the lowest address in memory. As an example of this culture shock we present in the following table the byte order on various well known CPU platforms of short and long (2 and 4 bytes) data types. The least significant byte being 0, and the next least significant byte 1 *&c* [18].

CPU type	Short	Long
IBM/360	1 0	3 2 1 0
PDP-11	0 1	2 3 0 1
80*86	0 1	0 1 2 3
MC68000	1 0	3 2 1 0
VAX	0 1	0 1 2 3

IBM/360 and MC68000 are true big-endians, while VAX and 80*86 are true little-endians.

Low Level Efficiency On the interface of two objects of different substance there exist contact losses, conversion losses. One must try to keep these losses as small as possible. That is why we did not consider converting integers and reals to characters, and back again. To the same category belongs the conversion of database records to full ASCII with field separators and the like, and back again parsing it into a record structure.

As there is no need for human 'interception' of the data, binary ('raw') can be used. Also the lack of a common standard[12] [10] let us consider the use of self-documenting data.

Conversion takes time (a loss) and in our homogeneous environment it was not necessary anyhow. Nevertheless, if conversion is necessary it must be done at the receiving side: all information regarding the target environment will be present and can be taken into account, and as the data come in self-documented, this cannot pose a problem.

Some consideration should be given to measuring of computer time involved, full ASCII conversion compared with basic data items, and the time involved in setting up or choosing a standard (and changing the program when the standard changes) compared with the 'free-format' ILS.

[11] After Jonathan Swift's *Gulliver's Travels*, Everyman's Library, No. 60, Dent, London, 1906 (repr 1966); Part I, Voyage to Lilliput, Chapter IV. The Lilliputians break their eggs at the smaller end, the Blefuscudians eat theirs after breaking them at the larger end: this slight difference in custom is the Cause of many a civil war in the kingdoms.

[12] The real problem is that this construction is not a tautology. Andy Tanenbaum once remarked: "The nice thing about standards is that there are so many of them to choose from."

5.2 A Partial Description of the Realisation of the Trapdoor

The tedium of programming in Pascal has been sufficiently demonstrated in previous Chapters. So this part of the Chapter can be relatively short as we will not describe realisation details of the trapdoor. We assume the reader of this Section to have enough practical experience in computing to possess a basic knowledge of the processes and terminology involved.

After we describe Twentel and its extension, the conversion of Twentel to our own environment with its conversion problems is discussed. The experiments used in Chapter 4 to illustrate some points are described, together with the difficulties encountered in the realisation. We conclude with a discussion on the feasibility of the trapdoor on various platforms.

5.2.1 Twentel's Architecture and the Place of the Trapdoor

5.2.1.1 The Original Twentel Implementation

Twentel was a research vehicle in the Programming Language Group of the section CAP in the Computer Science Department of the University of Twente.[13] The original Twentel development consisted of Twentel implementations for different platforms, e.g., DEC20 (under the TOPS10 operating system), DEC VAX (under the UNIX / ULTRIX operating system), and IBM PC or PC-compatible (with MS-DOS).

Twentel has been built as a classic interpreter, with a top level read-eval-print loop. The classic Lisp system [14] uses two levels of data (external, being symbolic expressions, or program text) and internal (dotted pair, or program / data to be interpreted). Contrary to this classic interpreter model, Twentel elaboration uses three levels of data (3.4.5): external (the normal program text which is typed in or read in from a file — the Twentel program), intermediate (the translation to extended lambda expressions) and internal (translation to combinator forms, or the program / data to be interpreted by the graph reduction machine). More details of the elaboration mechanism can be found in 'Kroeze' [11].

Twentel's main program can be seen as the Twentel system interface to the outside world: initialisation, input of program text through keyboard or file, system messages regarding the status of the translation and evaluation process, and error messages.

The first main function, `compile`, takes care of translating Twentel program text to lambda expressions and further to combinator based program expressions, represented in Twentel graph space.

The second main function, `interprete` (with its work-horse `reduce`), embodies the Twentel elaboration mechanism: reduction of the recently produced program expressions to normal

[13] The observant reader will have noticed the name change from 'Technische Hogeschool Twente' (Chapter 1) to 'University of Twente'. This change is intentional, and law was instrumental to effectuate it (the mere addition of a Medical Faculty would have been a healthier way of realising it). In this transition the subtle similarity (only four characters differing) between the 'Technische School Twente' (located a few hundred meters down the road from the Campus to Enschede) and the 'Technische Hogeschool Twente' was lost. Afterwards the school was renamed to 'Twents MBO College'.

form (actually WHNF, 3.4.5). An important set of building blocks for the function(s) form the 'axioms', separate functions that embody the rewrite rules of (internal) combinators and built-in functions.

One of these 'axioms' needs mentioning, the family of `WR`-related axioms — the function that writes a normalised expression value to screen / file. This 'axiom' family formed the starting point of our work on the `TRAP` combinator 'axiom'.

5.2.1.2 Adaptation of Twentel to Accommodate for the Trapdoor

The introduction of `TRAPP` (with its family members, `TRY`, `TREAD`, `TRITE`, `TRD`, `TWR` and `TWRI`) did not rely on syntactic changes in Twentel. Soon it became clear that the only way to introduce it in a safe way into Twentel was to realise it as a separate 'axiom' in the main 'axioms' library, together with an entry in the main `CASE`-statement in `reduce`. The reduction to normal form, necessary for the strict `Oargument`, was modelled after the `WR`-'axiom'.

The Development Process We could have followed proper procedure in proceeding with the trapdoor construction and its evolution in the following manner:

- proving theoretically that Twentel can accommodate a trapdoor (Chapter 4);
- putting the theory into practice by building the trapdoor into Twentel;
- checking the functionality by transporting a simple basic data item;
- then establishing the feasibility by handling a simple list (of basic data items, or a mix of them). That construction would enable us to model the Graph Identificator;
- subsequent elaboration of the feasibility by handling lists of lists; this would enable us to model the connection to the SQL functionality and I/O with simplex trapdoors;
- then extending the trapdoor with the possibility to work with the undefined value (bottom – $\perp$);
- subsequently the introduction of an interrupt into the system. In this case we need a separate channel, the interrupt channel.
- finally the extension to large function values should be implemented (the 'unclaimed resources' problem, 5.1.3.3).

After dealing with the first two items, we skipped the next two items, as we had gained sufficient expertise in handling full list structure on I/O when implementing different kinds of LISP interpreters. The results of our work on the fifth and sixth item gave sufficient confidence in the functioning and feasibility of the trapdoor, so we did not deem it necessary to tackle the (more academic) last two items (especially where the seventh item would not add functionality to our exposition of the trapdoor, only tedium).

Problems Encountered in Converting the Twentel System The original Twentel system was written by Henk Kroeze in MS-Pascal, version 3.30. We took the Twentel implementation — Version 1.80, March 1988 — as basis for our work [11]. Kroeze's realisation consisted of a MS-Pascal program in various source files, with some OS dependent coding, mainly in I/O and memory handling. Its total size was 782 Kbyte source files, yielding an executable of 231 Kbyte.

As we had no proper version of MS-Pascal available to deal with the features of 3.30 used in Kroeze's realisation, we decided to convert the system to the available TurboPascal, version 5.0. It was a fairly straightforward conversion. However, one interesting technical problem manifested itself, concerned with what we might call the 'dirtyjump' problem. Later we upgraded TurboPascal to version 5.5 without any problems, except the name change of an identifier called 'object'.

In the conversion from MS-Pascal to TurboPascal the following main problems were encountered. We shall discuss them here very briefly.

- in the interface between the formal Pascal and the implementation dependent Pascal with its OS (the PASCOS module), the textfile handling necessitated a different realisation;
- in the module that contains the formalised use of heap and stack (the MALLOC module) we met the following embellishments:
  - due to the use of full, normalised, pointers in TurboPascal we could introduce a more elegant way of dealing with heap pointers;
  - as TurboPascal has a correctly functioning heap manager, the MS-Pascal implemented 'free list' could be discarded;
- care had to be given to the possible differences in the real and integer types;
- in TurboPascal we could not use nested procedures as function type argument: every procedure to be used as the actual value of a function type argument had to be defined at top level;
- `goto` into another (higher) block was not possible in TurboPascal; in the next paragraph we will elaborate on this subject;
- the MS-Pascal 'AND THEN' and 'OR ELSE' constructions were replaced by TurboPascal's run-time shortcut Boolean evaluation.

The first TurboPascal Twentel realisation consisted of 768 Kbyte source files, yielding an executable of 216 Kbyte. Based on this version we undertook the trapdoor construction.

It is not quite fair to compare the original MS-Pascal and the current TurboPascal source volume to estimate the trapdoor effort, as more than the trapdoor was incorporated. In order to get acquainted with the system more features (or bells) were added (a.o., a standard prelude to be supplied by the programmer, and, due to our preference for the Walrus,[14] more timing information). Apart from these features, also the referentially transparent trapdoor with its features (e.g., memo function, random generator, undefined references) is still present.

[14] As in the `tempusfugit` message of old IBM 7090 LISP: " 'The Time has come,' the Walrus said, 'to talk of many thongs: / Of bits — and bytes — and baud-rate shift — / Of `evilquotes` — and `cons`, / And why the garbage can is never full / And if recursion stops."

The current TurboPascal realisation of Twentel and trapdoor consists of 944 Kbyte source files, yielding an executable of 236 Kbyte. Loek must be added to this: the normal version which was used in the SQL-experiment amounted to 6 Kbyte source files (the used functions come from Twentel libraries), yielding an executable of 58 Kbyte.

The 'dirtyjump' Problem In MS-Pascal it is possible to jump into a block from another module (as used by Kroeze in the Twentel system to realise an error exit from somewhere deep down in the compiler or the interpreter). TurboPascal does not support this construction. So we devised a (dirty) contraption that Willy Schulte very kindly realised in Macro Assembler for the 80x86: `fastret`.

In the original MS-Pascal implementation an error-return procedure (`eretproc`) occurred several times within a main procedure (`mainproc`: e.g., being `compile` or `interprete`). `mainproc` had the name of `eretproc` as an actual argument in its call. The essence of the construction was, `eretproc` only consisted of `goto 1000` , with `1000` a label in the block which contained the call to `mainproc`, located right behind the call to `mainproc`. The result of calling `eretproc` somewhere in `mainproc` is to discard all kinds of stack-based values, produced in `mainproc`, and to resume execution as if `mainproc` correctly returned.

In MS-Pascal the construction looked like:

```
Module Dirty                        Module Compile
     ... containing: ...            ... containing ...
     procedure eretproc;                    ...
     goto 1000;                             compile(eretproc);
     end;                             1000: compilefinish;
     ...                                    ...
```

It must be possible to realise a similar construction, and use it in TurboPascal.[15] This construction, say `dirtycall`, saves locally the return address and stackpointer indicating the main program status as it will be right after return from `dirtycall`. `mainproc` is called with `eretproc` as argument in the body of `dirtycall`. This `eretproc`, an external procedure, is also defined in TurboPascal's unit `Dirty`, so it has access to the above local variables: return address and stackpointer from the main program. `eretproc` is passed down in `mainproc` to all its sub-procedures as an error exit function argument. If `eretproc` is called somewhere in `mainproc` , it resets the main program situation right after the call to `dirtycall`. In TurboPascal it should look like:

```
Unit Dirty                                          Unit Compile
     ... containing: ...                                 ... containing ...
     procedure eretproc;                                 ...
     ...                                                 dirtycall(compile);
     procedure dirtycall (mainproc: function);           compilefinish;
     mainproc(eretproc);                                 ...
     end {dirtycall};
     ...
```

[15] At the same time it must be clear that we should not have to resort to this kind of tricks when programming.

Later we found a second type of this kind of procedure, needing two arguments. We solved this problem in the same way.

5.2.2 Implementation Models of the Trapdoor in Twentel

We have defined two mutually independent processes (Twentel and *Outside*), communicating by means of the trapdoor. As is obvious, '640 K' is not sufficient to accommodate these processes in the case of the SQL-experiment: a Twentel system, Loek, and another Twentel system with an in-core database, plus all graph space. So we started with a smaller experiment.

The feasibility of the realisation of the implementation model, based on the second, fourth and fifth point from 5.2.1.2, was tested in a few phases, each with its own purpose:

- one PC, with two memory blocks in which these processes run, and a dispatcher (Loek) between them; can a trapdoor be realised in Twentel and function?
- two PC's, by definition mutually independent, with these running processes, and a communication link (Loek) between them; can Twentel communicate with the outside world by means of the trapdoor?
- a Unix machine with two processes and a dispatcher between them.

However, having demonstrated that the first two realisations did function properly and reached their goals, we hold that the Unix model can be regarded as a special form of the two-machine model. The unavailability of a Unix machine and an incompatible Twentel version (Twentel without trapdoor) made us take this position theoretically.

5.2.2.1 Trapdoor Implementation on MS-DOS Machines

A small uni-processor PC cannot run processes in 'parallel'. Here, small means a PC based on an Intel 286 processor or lower with MS-DOS. The generally accepted method of simulating 'parallel' behaviour with MS-DOS, is to introduce a Terminate-and-Stay-Resident (TSR) program [5].

Such TSR-programs are loaded into core and given control as with any program to be executed. However, when a TSR-program finishes its initialisation it relinquishes control without disappearing from memory, it remains in the background. The TSR-program regains control on a hard- or software interrupt, does its bit and relinquishes control again. (Note that in the single machine configuration we cannot use a hardware interrupt, as that implies independent running processes.) Communication with such a program from another program takes place around an activating software interrupt. Together with the event of the software interrupt, information can be passed to the TSR program in the hardware registers. In our case the TSR program is Loek, the program we introduced in 5.1.1.2.

Loek is the embodiment of the dispatcher of information: on reception of an interrupt it determines what pre-installed functionality must receive the incoming token. Loek contains also the memo functionality (4.5.1) and the random generator (4.5.3.2). Looking at this situation it is

clear in retrospect why the information flow through Loek is done in packets of information, or tokens: there is no information 'flow', information is passed in 'bursts'.

5.2.2.2 Implementation on a Single MS-DOS Machine

The purpose of the Graph Identificator experiments was to see whether a trapdoor could be made to function at all. Not with a simple scalar, but with a sufficiently complex structure. In these experiments the ILS format for transferring values through Loek was fully used. However, *Outside*'s interface with Loek (the Graph Identificator) was hand-coded.

The Graph Identificator experiment progressed through the following stages:

Simple : graph code in; identification number out;

Elaborate : graph code in; identification number, dual graph code, number of nodes, edges and meshes, self-dualism *&c* out;

Referentially Transparent : graph code in; return stored (elaborate) information if previously computed, else compute elaborate information and store it.

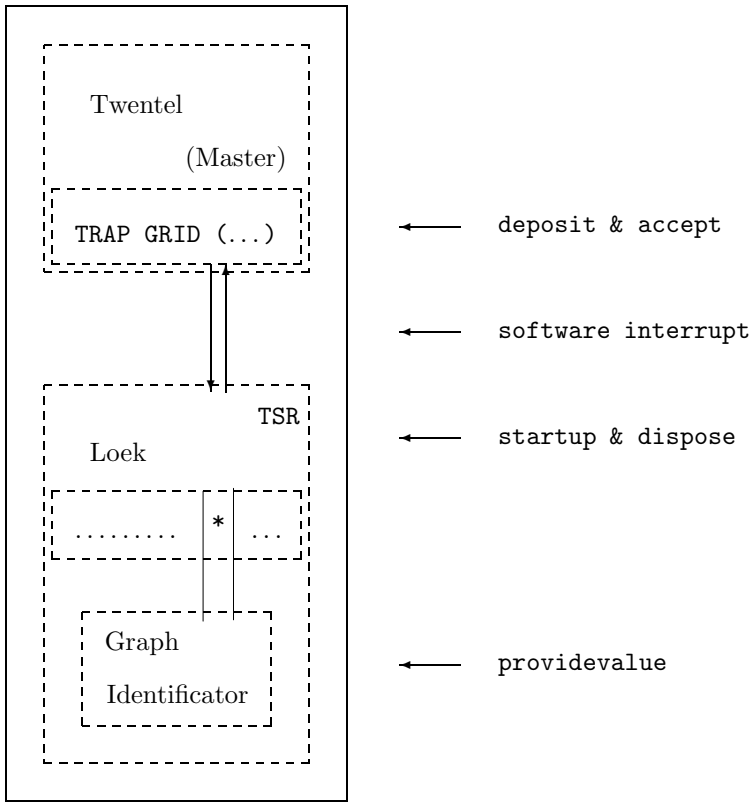

Figure 5.4: *One-machine configuration of Twentel with Loek and* Outside

Most of the realisation of the trapdoor architecture has been discussed in the previous paragraphs. A few remarks are in place.

- Twentel expects an eager *Outside* functionality, that is to be 'started' on the first software interrupt from Twentel. The Loek program must be up and running (viz resident, TSR) when starting Twentel with the trapdoor functionality.[16]
- The Graph Identificator cannot be made into a separate program in this MS-DOS configuration. We made it a subroutine of Loek as one of the resident functionalities.

The configuration in Figure 5.4 (MS-DOS, 640 K) has been used to test the Graph Identificator (and to a lesser extent, also the memo function, the random generator, and the undefined references).

5.2.2.3 Implementation on Two MS-DOS Machines

A two-machine configuration (Figure 5.5, both MS-DOS, 640 K, linked via their 'COM'-ports by means of a null-modem) was used in the next set of experiments. It was set up to test whether Twentel could indeed communicate with the outside world through the trapdoor. In this configuration the relational database experiments (4.3.2.4 and 4.3.2.5) were carried out. Both

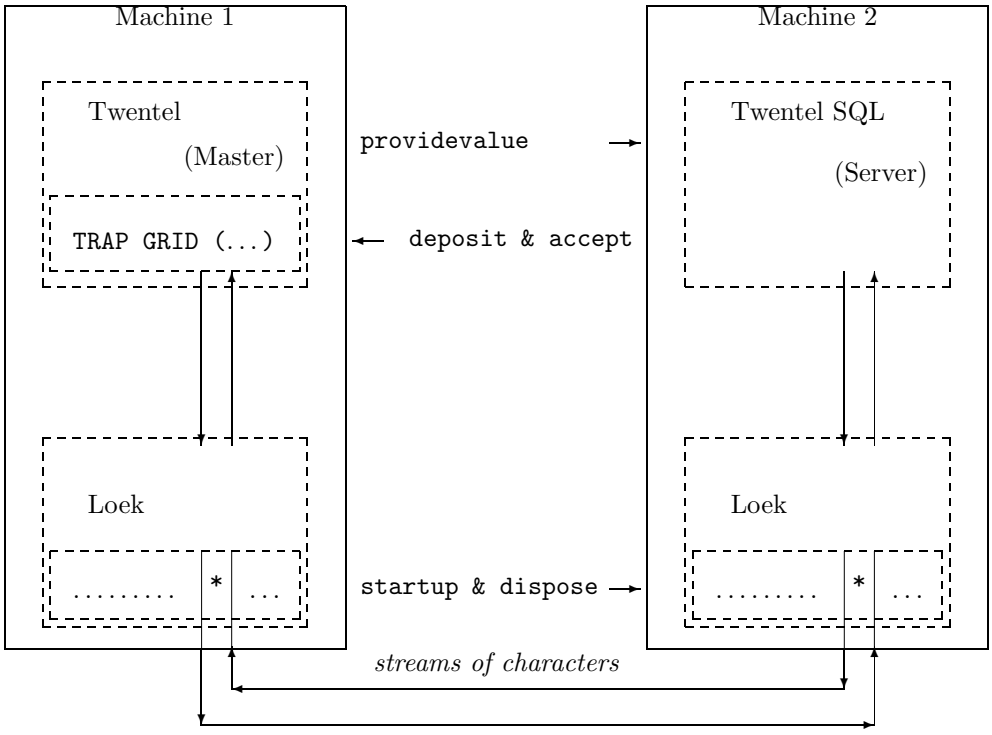

Figure 5.5: *Two-machine configuration of Twentel with Loek and* Outside

Loek programs must be up and running (viz resident) when starting the trapdoor functionalities, as again Twentel expects an eager Outside functionality. So the Slave program (here Twentel) must be up and running too, when starting the trapdoor at the Master side (though waiting is taken care of by the communication link within Loek).

[16] Apart from this 'theoretical' point, there is also the practical point of not being able to start a TSR program when another program is active and uses all available memory.

As is clear from the figure, Loek now embodies a link between the two instances of itself in the sketched configuration: 'streams' of Booleans, characters, integers, reals, and parentheses, all in token form, move between the two machines with Loek as the communicator. Based on RS232-routines from Willy Schulte, we also created this serial communication link of two PC's via the 'COM'-port and incorporated it into Loek.[17]

5.2.2.4 Trapdoor Implementation on a Unix System

As we ourselves did not acquire much experience with programming under Unix, we will only highlight aspects of the implementation of the trapdoor architecture in U(nix)Twentel. When we started our research the unavailability of the Unix machine cut off the practical route of making a Unix implementation. Today, with Linux running on a 486, the Unix implementation would pose no problem. It should take approximately a month of work to adapt UnixTwentel to accommodate the trapdoor, given the modular inclusion of the trapdoor coding in TurboPascal and the equivalent structure of the Twentel system under Unix. In this discussion we assume the notions from Unix to be known (e.g., in [1]).

Loek can be made into a daemon process.

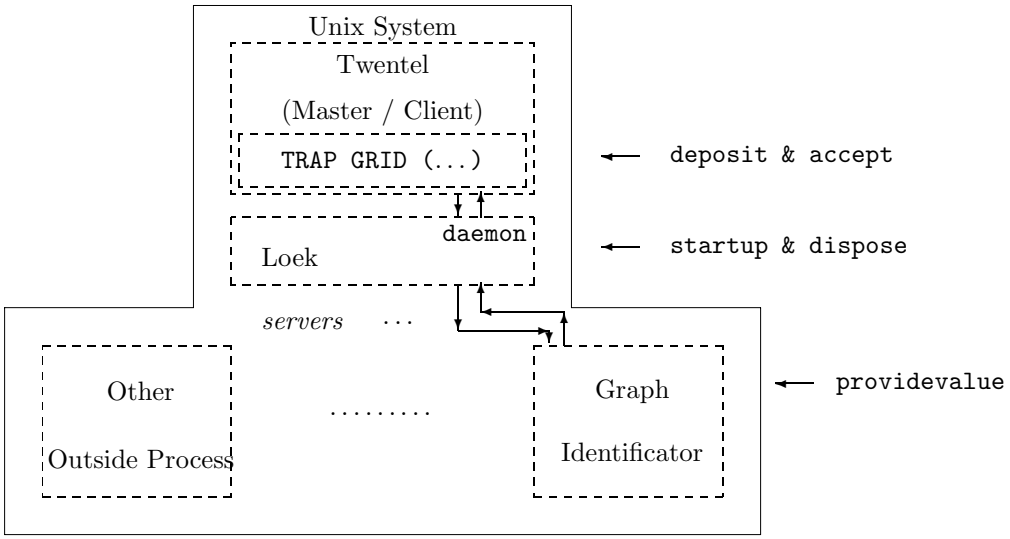

Figure 5.6: *Configuration of Twentel with trapdoors under Unix*

The multi-tasking of Unix necessitates extending ILS with addresses, addressee (more Loek's can be around) and sender (more UTwentel systems can be present). The communication itself should be set up with 'socket communication', enabling the transmission of the binary data in ILS. Further, pipes are uni-directional if considered as belonging to and coming from independently constructed processes. The criteria of propriety and orthogonality strongly oppose implementing simplex trapdoors (uni-directional) in another way then the duplex trapdoor, as the consistency of the design cannot be guaranteed when using two completely different concepts.

[17] Once more establishing the old adagium that mixing electronics and abstractions like bits, raises the weight of the difficulties in the error finding process in the system by an order of magnitude (at least).

The other aspects, the implementation of the trapdoor in UTwentel and the construction of the standard libraries (e.g., for C and Pascal) with `GetInputFromTwentel` and `ConvertOutput-RecordForTwentel`, must be addressed too.

As communication between processes in Unix is not an uncommon feature, we hold that, with the above problems addressed, implementation of UTwentwel under Unix is possible. We give the sketch of such a possible Unix implementation in Figure 5.6.

5.2.3 Migrating the Trapdoor

The trapdoor architecture is independent of the underlying machines. So having demonstrated that it functions on one particular machine, we are justified to conclude that other realisations are likewise possible.

As we used a fairly normal implementation model (viz graph rewriting) that is not adverse to the lambda calculus, we do not see theoretical problems in migrating the trapdoor to functional languages having other implementation models. Some thoughts have to be given to the inclusion of the trapdoor in the typing mechanisms of other functional languages.

The trapdoor is an architecture for communicating with the outside world in order to increase the usability and the use of functional languages. The designer of a functional language not amenable to implementing the trapdoor architecture, should include another communication architecture in his language in order to let the language survive.

References

[1] Abrahams, P., Larson, B., *Unix for the Impatient.* Addison–Wesley, Reading MA, 1992.

[2] Lewis Carroll. *Alice's Adventures in Wonderland.* Macmillan, London, 1865.

[3] Cohen, D., "On Holy Wars and a Plea for Peace", *IEEE Computer* **14**(10) (Oct 1981): 48–54.

[4] Demers, R.A., Yamaguchi, K., "Data Description and Conversion Architecture", *IBM Syst J* **31**(3) (1992): 488–515.

[5] Fichtelman, M., "Don't Worry, Use HLLAPI", *Byte* **15**(11) (Fall 1990): 207–216, (IBM Special Issue).

[6] Van der Hoeven, G.F., *Preliminary Report on the Language Twentel.* Memorandum INF-84-5, Dept Comp Sci, Univ Twente, Enschede, Mar 1984, 87 pp.

[7] Hudak, P., Peyton Jones, S.L., Wadler, P., (eds) *et al*, *Report on the Programming Language Haskell, a Non-strict, Purely Functional Language.* Version 1.2; 1 March 1992, xii + 164 pp, in: *SIGPLAN Not* **27**(5) (May 1992), Section R; *also* Technical Report, Yale Univ *and* Univ Glasgow, Aug 1991.

[8] International Organization for Standardization (ISO), *ISO Computer Programming Language Pascal.* ISO 7185, Geneva, 1983.

[9] Jensen, K., Wirth, N., *Pascal User Manual and Report.* Springer-Verlag, Berlin, 1974, (3^{rd} ed, 1985).

[10] Kirrmann, H.D., "Data Format and Bus Compatibility in Multiprocessors", *IEEE Micro* **3**(4) (Aug 1983): 32–47.

[11] Kroeze, H.J., *The Twentel System, Version I - (1.24–1.99) Available on Various Machines: General Reference Manual & User Guide.* Dept Comp Sci, CAP / Languages Group, Univ Twente, Enschede, 1986–1987, 116 pp.

[12] Lamb, D.A., "IDL: Sharing Intermediate Representations", *ACM Tr TOPLAS* **9**(3) (Jul 1987): 297–318.
[13] Landin, P.J., "The Next 700 Programming Languages", *Comm ACM* **9**(3) (Mar 1966): 157–166.
[14] McCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P., Levin, M.I., Lisp *1.5 Programmer's Manual.* MIT Pr, Cambridge MA, 1962 (2^{nd} ed, 1965, vi + 106 pp).
[15] Peyton Jones, S.L., "Yacc in SASL - an Exercise in Functional Programming", *Softw Pract & Exp* **15**(8) (Aug 1985): 807–820.
[16] Purtilo, J.M., "The POLYLITH Software Bus", *ACM Tr TOPLAS* **16**(1) (Jan 1994): 151–174.
[17] Raymond, E. (ed), *The On-Line Hacker Jargon File, version 2.9.12.* `esr@snark.thyrsus.com`, May 1993.
[18] SCO-UNIX Development System, *Encyclopaedia.* Santa Cruz Operation, version 3.2.4D, Santa Cruz CA, Dec 1991.
[19] Silberschatz, A., Peterson, J.L., Galvin, P.B., *Operating System Concepts.* Addison–Wesley, Reading MA, 3^{rd} ed, 1991.
[20] Van Wijngaarden, A. *et al* (eds), *Revised Report on the Algorithmic Language ALGOL 68.* Springer-Verlag, Berlin, 1976.

Chapter 6

Conclusion

Language is the vehicle of communication. Without communication there can be no cooperation, essential for growing, evolving and improving. Communication is the *conditio sine qua non* for progress. In computer science, for user problems this translates to solutions cooperating with other solutions; in another sense, it means easy communication of the solution from programmer to computer. If a programming language does not facilitate communication, solutions will not be written in that language. However, there are reasons for choosing a programming language other than for its communication potential. Functional languages are very powerful (as are all declarative languages), however, they lack easily applicable communication facilities. The addition of these facilities by 'embedding' them in the language would 'close' the language. Every embedding must obey the syntax and semantics of the host language. In communication open-endedness is needed, so extending without binding is sought after.

We demonstrated the feasibility of the trapdoor construction in functional languages for communicating with the outside world. The two main experiments (the Graph Identificator and the SQL server), together with the model of a conversion algorithm for the free-format transmission structure of values (ILS), support this statement. Functional languages can be used in environments other than the one for pure research, since it is possible now to use existing solutions to help solving the user problems. The power of functional languages — expressive power in program development while staying close to the problem structure, and mathematical power in the possibility of automatic transformations and inherent correctness — can be made available to a larger community, of users and programmers.

The call for more efficiency, as a consequence of the programmer's need for easy development of prototypes to keep the Problem Solving Cycle short, or of the need for faster execution, can now be answered by functional languages too. Not because they contain all these facilities themselves, but they can call upon the outside world for assistance. With the trapdoor we have the possibility of subdividing the problem under consideration into subproblems that for these reasons of efficiency can be delegated to other processes. Also a user request for results of existing processes to be used in the handling of his new problem, can be made available in the functional programming environment through the trapdoor.

Pamela Zave corroborates our argument on extending the language instead of incorporating features into it [3]: "By definition, a paradigm offers a single-minded, cohesive view — this is how the popular paradigms ([...]) help us think clearly, offer substantial analytic capabilities, and achieve their reputations for elegance. The corresponding disadvantage is that each paradigm is too narrowly focused to describe all aspects of a large, complex system. [...] The

purpose of multiparadigm programming is to let us build systems using as many paradigms as we need, each paradigm handling those aspects of the system for which it is best suited." With messages in the format of ILS and extending the Hindley-Milner polymorphic type structure of functional languages with sub- and super-typing ('inheritance') the move towards the object-oriented paradigm will be executed *en passant*, broadening the applicability of the functional languages.

The trapdoor, having its feasibility demonstrated, promotes the idea of the applicability of functional languages in wider domains. Also we expect the expressivity and mathematical aspects of the construction effectuated in this dissertation, to be stimulating for new applications of functional programming in not-evident fields of program development. The trapdoor concept is, however, by no means the only instrument needed for this purpose (3.1.4). More effort has to be put into the task of spreading the power of this kind of languages (which includes the logical languages). We encounter here an educational problem.

The traditional computer science education is based on the Turing-Von Neumann paradigm of computing. In it we are now meeting the boundaries of our understanding. These boundaries are defined by the way we speak about the problems, defined by the language we use in describing these problems. The way out is a paradigm shift, from the Turing machine to the lambda calculus. So the languages in the wake of the paradigm shift based on the lambda calculus, can move once more the boundaries of our comprehension outwards.

Concluding this dissertation, we present some thoughts on various subjects that we raised in connection with this paradigm shift, from 'computing in' to 'computing with' functions.

The Psychology of Programming More thought has to be given to the following observation. Thinking of a solution at the higher level of abstraction functional programming languages endow us with, has a surprising property. It is only possible to write down a functional program when one fully understands the 'theory' behind the problem. Contrary to this observation, it is possible to start writing programs in *imperative* languages without having fully comprehended the problem and its solution.[1] The expressive power of functional languages is such that it forces a greater discipline upon the programmer; this discipline provides a process that yields more robust programs. Difficulties with understanding the 'theory' — the abstraction level involved — might also explain the fact that it is possible to produce in one hour a specification for a program that takes weeks to write, and its opposite, explaining in weeks a program that took an hour to write.

Functions as First-Class Citizens Functions cannot be printed in their 'value' form, a form different from the textual form in which we present them to the compiler. A solution to this problem can be thought of as an extension to the ILS handling procedures, and it is contained in the *delta*-reduction process of the lambda calculus and combinatory logic.

Every computer possesses an arithmetic functionality machine with general consensus about the used functions, e.g., ADD, MUL and SIN — though data formats may differ on these computers. So presenting any computer '`3 + 4`' will always yield `7`. Sending such an arithmetical form to

[1] *In extremo* this amounts to Dijkstra's "debugging of an empty source file".

any computer in ILS format, or sending it a function with its arguments, will also result in 7, if an ILS handler is present on that computer, and the use of the arithmetical operations on the host machine is taken care of in the ILS handler. The mentioned function is an example of a computable function, which can easily be extended to a larger class of functions by introducing a suitable (e.g., Polish reverse) notation. However, this is a very small class of functions.

All computable functions can be expressed using the lambda calculus. As the lambda notation is somewhat unwieldy for transmitting, the equivalent combinator forms can be used instead. They offer an excellent medium of representing functions in a communication format. The other — receiving — side must implement the combinators (`S`, `K`, and the others from 3.4.4.1) and a suitable reduction space in its ILS handler. There cannot be anything but general consensus about the definition of these functions, the combinators.

In this way functions become first-class citizens, also from a communication point of view. Functions can then be printed (and read back in the same format), though the print image will show the consequences of Curry's remark on the appearance of a combinator form we quoted in 3.4.4.2. Reading back functions implies that `TREAD` at a functional position makes sense now. Functions can also be stored in data space. This is useful in data base environments, where constraints and integrity checks now can be part of the data base itself.

Some problems must be solved. We name a few:

- handling infinite (recurrent) structures (through application of the `Y` combinator, or cross-linked data structures);
- handling partially evaluated or shared functions.

A useful tool in this respect might be the use of the `ffpoel` function.[2] Other useful building blocks may be developed using the leads given in the IDL paper [2].

Data as First-Class Citizens The notion of 'functions as first-class citizens' stemmed from the idea that data were the only true first-class citizens. Alas, this is only partly true. Only during the execution of a program, data can be accepted as argument and passed on as function value, by favour of the fixed interpretation given to data by the various functions. Languages with run-time typing structures (e.g., LISP) are the only ones that can hold the position of giving data first-class citizenship.

Data residing on storage media is only data by favour of interpretation (4.4.4). With data in ILS format, data too can be made first-class citizens. In this case systems can use the full range of their possibilities in handling them (e.g., a data compressor can work faster when it does not have to do everything when compressing, some things are already known or implicit in the data type).

It is the automation of things the programmer has to think of now. Processing a long list of integers, once a year, in sequence does not call for an array. The system, based on known

[2] Attributed to W.L. van der Poel. This function checks whether its list argument contains a circular structure. It does so by applying itself recursively to the head of its argument, and applying itself again to the head of that result, then comparing the two results. If there is a loop in following the link, the two results will be equal after a certain number of steps.

information (the self-documenting data) and data on the use of the data, can derive an optimal (time dependent) handling strategy for the data. Not only with simple integers, also complete databases will profit from such an approach. In the database world this phenomenon can result in e.g., automatic rearranging of indices.

In functional programming data and operations on them (yielded by the problem domain), can be considered as first-class citizens. This in turn enables data processing systems to use (in principle) optimal data handling techniques. Incoming data can be handled in different ways but within a unified framework, dependent upon the intended use of them. Typed (or self-documenting) data are not interpreted but treated symbolically — they too are lifted to a higher level of abstraction. Once more we observe here the power of describing the 'what' instead of describing the 'how', now applied upon a data structure.

Epilogue In the Introduction we were already aware of the etymology of 'technics': from the Greek *technè*, meaning 'art' or 'skill'. Taking this as our cue, and returning to the opening phrase, we conclude with a quote from Donald Knuth since we addressed in the meantime in extended functional programming languages the problems of reliability, maintainability and efficiency [1]:

> "All of the major problems associated with computer programming — issues of reliability, portability, learnability, maintainability, and efficiency — are ameliorated when programs and their dialogues with users become more literate."

References

[1] Knuth, D.E., *Literate Programming.* Center for Study of Language and Information, CSLI Lecture Not No 27, Leland Stanford Jr Univ, Stanford CA, 1992, xv + 368 pp.

[2] Lamb, D.A., "IDL: Sharing Intermediate Representations", *ACM Tr TOPLAS* **9**(3) (Jul 1987): 297–318.

[3] Zave, P., "A Compositional Approach to Multiparadigm Programming," *IEEE Softw* **6**(5) (Sep 1989): 15–25.

Appendix

Derivation of a Relation between TRAP, TREAD, and TRITE

We started in 4.2.3.1 with the new combinator TRAP, displaying O/I behaviour. The original definition of TRAP is (4.2.3.1):

$$\texttt{TRAP Ofun Oargument} \rightarrow \texttt{FVal} \tag{1}$$

In 5.1.1.3 two new combinators were introduced to separate the O- and I-behaviour, TRITE and TREAD. For TRITE and TREAD we use the following definitions 5.1.1.3):

$$\texttt{TRITE Ofun Oargument} \rightarrow \mathrm{I}$$
$$\texttt{TREAD Ofun} \rightarrow \texttt{FVal} \ .$$

We will now derive a relation between the original TRAP combinator and the two new TREAD and TRITE combinators.

An instantiation of (1) would appear in a Twentel program (with $f = \texttt{Ofun}$, and $L = \texttt{Oargument}$) as, e.g.,

$$\texttt{TRAP}\ f\ L$$

which must yield the following form after some rewriting:

$$\alpha\ f\ x : (\ \texttt{TRAP}\ f\ X\ ) \tag{2}$$

with α, the relation we want to derive, some function of TRITE and TREAD, and with $f = \texttt{Ofun}$, and $L = \texttt{Oargument}$, written as $L = (\ x : X\ )$.

To obtain the derivation, we use the following combinators from [1] (cf 3.4.4.1):

B	B	$f\ g\ x$	$=$	$f\ (\ g\ x\ )$	or, $f\ .\ g\ x$
B1	B1	$f\ x\ y\ z$	$=$	$f\ x\ (\ y\ z\ )$	
C	C	$f\ x\ y$	$=$	$f\ y\ x$	
CI	CI	$x\ y$	$=$	$y\ x$	
I	I	x	$=$	x	
K	K	$x\ y$	$=$	x	
Φ	Φ	$f\ a\ b\ x$	$=$	$f\ (\ a\ x\ )\ (\ b\ x\ )$	
Ψ	Ψ	$f\ g\ x\ y$	$=$	$f\ (\ g\ x\ )\ (\ g\ y\ )$	
S	S	$f\ g\ x$	$=$	$f\ x\ (\ g\ x\ )$	
W	W	$f\ x$	$=$	$f\ x\ x$	
Y	Y	f	$=$	$f\ (\ \mathrm{Y}\ f\ )$	

Though S is not used in the derivation, it is inserted in the table because it is one of the two basic combinators, the essential S from the SK-system [1].

Apart from the combinators in the above list, we also use a few auxillary combinators, to handle the 'dotted pair' structure [2] in relation (2); with P and U taken from [3]:

P	P	$x\ y$	$=$	$x : y$	(cons)	$x : y \rightarrow$ CI (:) $x\ y \Rightarrow$ P = CI (:)
U	U	f ($x : y$)	$=$	$f\ x\ y$	(uncurry)	
A	A	$x : y$	$=$	x	(car)	$\Rightarrow$ A = U K
	A	(P $x\ y$)	$=$	x	(car)	
D	D	$x : y$	$=$	y	(cdr)	$\Rightarrow$ D = U (B K CI)
	D	(P $x\ y$)	$=$	y	(cdr)	

We are now ready for the derivation. In it, the lines with only parenthesiation changes result from the left associativity of the application operator (3.2.2): '$A\,B\,C = (A\,B)\,C$'.

The effect of TRAP has been defined in 5.1.1.1. The desired function α expressed in TREAD and TRITE, must have the same effect. So the form which (2) must yield, is our starting point:

TRITE $f\ x$ (TREAD f) : (TRAP $f\ X$)

First, remove parentheses in order to manipulate f, by using B three times:

B (TRITE $f\ x$) TREAD f : (TRAP $f\ X$) $\rightarrow$

B B (TRITE f) x TREAD f : (TRAP $f\ X$) $\rightarrow$

B (B B) TRITE $f\ x$ TREAD f : (TRAP $f\ X$) .

Further manipulate both f's in order to obtain, eventually, one f, by using C twice:

(B (B B) TRITE) $f\ x$ TREAD f : (TRAP $f\ X$) $\rightarrow$

C (B (B B) TRITE) $x\ f$ TREAD f : (TRAP $f\ X$) $\rightarrow$

(C (B (B B) TRITE) x) f TREAD f : (TRAP $f\ X$) $\rightarrow$

C (C (B (B B) TRITE) x) TREAD $f\ f$: (TRAP $f\ X$) .

Remove the double f by introducing W, after parenthesiation:

(C (C (B (B B) TRITE) x) TREAD) $f\ f$: (TRAP $f\ X$) $\rightarrow$

W (C (C (B (B B) TRITE) x) TREAD) f : (TRAP $f\ X$) .

Now remove parentheses to manipulate x, again using B, and B1 twice after that:

B W (C (C (B (B B) TRITE) x)) TREAD f : (TRAP $f\ X$) $\rightarrow$

B1 B W C (C (B (B B) TRITE) x) TREAD f : (TRAP $f\ X$) $\rightarrow$

(B1 B W) C (C (B (B B) TRITE) x) TREAD f : (TRAP $f\ X$) $\rightarrow$

B1 (B1 B W) C (C (B (B B) TRITE)) x TREAD f : (TRAP $f\ X$) .

After moving x to the end of the form, correct the sequence of f and x, in order to obtain a form like (2), by using C twice:

(B1 (B1 B W) C (C (B (B B) TRITE))) x TREAD f : (TRAP f X) →

C (B1 (B1 B W) C (C (B (B B) TRITE))) TREAD x f : (TRAP f X) →

(C (B1 (B1 B W) C (C (B (B B) TRITE))) TREAD) x f : (TRAP f X) →

C (C (B1 (B1 B W) C (C (B (B B) TRITE))) TREAD)) f x : (TRAP f X) .

Compared with the original form (2), the above form yields:

α = C (C (B1 (B1 B W) C (C (B (B B) TRITE))) TREAD)) .

In order to get rid of the x and X, we use the initial form (2), and the 'dotted pair' combinators A and D, in order to have only the original Oargument, which is L. (2) can now be cast in the following form:

(α f (A L)) : (TRAP f (D L)) .

Remove the infix ':' (cons) operator in this form by means of P. Then remove the inside parentheses, with Φ and Ψ, in order to manipulate L, moving it to the end:

P (α f (A L)) (TRAP f (D L)) →

P (B1 α f A L) (B1 TRAP f D L) →

Φ P (B1 α f A) (B1 TRAP f D) L →

Ψ (Φ P) B1 (α f A) (TRAP f D) L .

Manipulation of f follows, by using C inside the parentheses twice, in order to get f at the end, afterwards Φ moves f out of the parentheses:

Ψ (Φ P) B1 (C α A f) (C TRAP D f) L →

Φ (Ψ (Φ P) B1) (C α A) (C TRAP D) f L .

Manipulation of TRAP to get it out of the parentheses, by means of Ψ, B, and C:

Ψ (Φ (Ψ (Φ P) B1)) C (α A) (TRAP D) f L →

B (Ψ (Φ (Ψ (Φ P) B1)) C (α A)) TRAP D f L →

C (B (Ψ (Φ (Ψ (Φ P) B1)) C (α A))) D TRAP f L .

Comparing this form with the initial one, (1), we see that:

TRAP = C (B (Ψ (Φ (Ψ (Φ P) B1)) C (α A))) D TRAP

or, shorter,

TRAP = β TRAP,

with

$$\beta = \mathrm{C}\ (\ \mathrm{B}\ (\ \Psi\ (\ \Phi\ (\ \Psi\ (\ \Phi\ \mathrm{P}\ )\ \mathrm{B1}\ )\ )\ \mathrm{C}\ (\ \alpha\ \mathrm{A}\ )\ )\ )\ \mathrm{D},$$

and from above,

$$\alpha = \mathrm{C}\ (\ \mathrm{C}\ (\ \mathrm{B1}\ (\ \mathrm{B1}\ \mathrm{B}\ \mathrm{W}\ )\ \mathrm{C}\ (\ \mathrm{C}\ (\ \mathrm{B}\ (\ \mathrm{B}\ \mathrm{B}\ )\ \texttt{TRITE}\ )\ )\ )\ \texttt{TREAD}\ )\ ) .$$

With the paradoxical combinator Y to express the recursion ($\mathrm{Y}\ \beta = \beta\ (\ \mathrm{Y}\ \beta\ )$), we come to:

$$\texttt{TRAP} = \mathrm{Y}\ \beta\ .$$

So we find `TRAP` expressed in terms of `TRITE`, and `TREAD` as follows:

$$\begin{aligned}\texttt{TRAP} &= \mathrm{Y}\ (\ \mathrm{C}\ (\ \mathrm{B}\ (\ \Psi\ (\ \Phi\ (\ \Psi\ (\ \Phi\ \mathrm{P}\ )\ \mathrm{B1}\ )\ )\ \mathrm{C}\ (\ \alpha\ \mathrm{A}\ )\ )\ )\ \mathrm{D}\ ) \\ &= \mathrm{Y}\ (\ \mathrm{C}\ (\ \mathrm{B}\ (\ \Psi\ (\ \Phi\ (\ \Psi\ (\ \Phi\ \mathrm{P}\ )\ \mathrm{B1}\ )\ )\ ) \\ &\quad \mathrm{C}\ (\ (\ \mathrm{C}\ (\ \mathrm{C}\ (\ \mathrm{B1}\ (\ \mathrm{B1}\ \mathrm{B}\ \mathrm{W}\ )\ \mathrm{C}\ (\ \mathrm{C}\ (\ \mathrm{B}\ (\ \mathrm{B}\ \mathrm{B}\ )\ \texttt{TRITE}\ )\ )\ )\ \texttt{TREAD}\ )\ )\ )\ \mathrm{A}\ )\ )\ )\ \mathrm{D}\ )\end{aligned}$$

□

References

[1] Curry, H.B., Feys, R., *Combinatory Logic.* Vol I, North-Holland, Amsterdam, 1958 (2^{nd} ed, 1974).

[2] McCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P., Levin, M.I., Lisp *1.5 Programmer's Manual.* MIT Pr, Cambridge MA, 1962 (2^{nd} ed, 1965, vi + 106 pp).

[3] Turner, D.A., "A New Implementation Technique for Applicative Languages", *Softw Pract & Exp* **9**(1) (Jan 1979): 31–49.

[4] Kroeze, H.J., *The Twentel System, Version I — (1.24–1.99) Available on Various Machines: General Reference Manual & User Guide.* Dept Comp Sci, CAP / Languages Group, Univ Twente, Enschede, 1986–1987, 116 pp.

Samenvatting

"Over een Uitbreiding van Functionele Talen voor Gebruik in Modellenbouw"

Om een probleem op te lossen met behulp van een computer is het nuttig een *prototype* van de oplossing te maken. De 'Grote Van Dale' (Engels–Nederlands, 1989) vertaalt *prototype* als 'oervorm, oorspronkelijk model, voorbeeld bij uitstek', de werkwoordsvorm kent hij niet. Toch denk ik dat 'modellenbouw' als vertaling van *prototyping* beter weergeeft wat in de informatica onder dit begrip wordt verstaan dan een 'werkwoordsvorm' van een van de bovenstaande begrippen. Het is meer dan het bouwen van een wiskundig model dat het probleem beschrijft: daarin worden bepaalde aspecten van het probleem versimpeld of weggelaten, het eindproduct is een stelsel wiskundige vergelijkingen dat het probleem beschrijft.

In de informatica blijft het ook niet bij een oervorm, of bij het juiste voorbeeld. Hier is *prototyping* het bouwen van een model (een computerprogramma) dat door uitvoering op een computer een deel van de omgeving wordt waarin het probleem voorkomt. Aan dit computerprogramma kan een wiskundig model ten grondslag liggen. De eenvoud waarmee het model aan de gewenste oplossing kan worden aangepast, duidt echter wel op een voorbeeldfunctie: een *prototype* is een voorbeeld van een mogelijke oplossing, het voorbeeld kan echter zeer gemakkelijk aan veranderde inzichten worden aangepast. Is het prototype eenmaal gereed en is men dus tot tot de gewenste oplossing gekomen, dan kan het getransformeerd worden tot een efficiënte oplossing voor gebruik in de practijk. Dit *prototyping* dient meer gebruikt te worden dan tot nu toe het geval is in de Toepassingsgerichte Informatica; het waarom wordt in de eerste hoofdstukken van dit proefschrift behandeld.

In de informatiemaatschappij heeft de gebruiker een programmeur nodig, omdat de oplossing van sommige van zijn problemen een computer vereist. Hij heeft deze programmeur nodig om een oplossing te ontwikkelen voor nieuwe of afwijkende problemen, niet voor problemen waar een standaardoplossing voor bestaat. De eerste hoofdstukken gaan dan ook over de wisselwerking tussen gebruiker en programmeur die ontstaat als zo'n probleem moet worden opgelost. Gebruiker en programmeur spreken ieder een andere taal: de gebruiker spreekt die van zijn werk- of vakgebied waar het probleem ontstond; de programmeur, gewend alles te formaliseren, eenduidig de stappen aan te geven zonder te associëren, gebruikt een formelere taal welke veel dichter bij de computer staat. Deze formele manier van communiceren is niet geschikt voor gebruik tussen mensen.

Programmeren behelst het geven van opdrachten aan de computer om een bepaald effect te bewerkstelligen. Door het hele proefschrift wordt verstaan onder hij of zij[1] die dit doet — de

[1] Voor het 'hij / zij' probleem, zie J. de Jong, "De schrijver en zijn / haar schaamlap", *Onze Taal* **63**(9) (Sept 1994): 193–194. Het voerde te ver om dit aspect in het Engels te vertalen, vandaar dat het niet in de Engelse versie ter sprake wordt gebracht.

'programmeur':[2] hij die verschillende manieren van programmeren van de computer ontwikkelt, teneinde een bepaald effect of gedrag te veroorzaken. Hij is dus niet alleen maar een schrijver van 'programmaatjes'. Bij dit programmeren gebruikt hij een 'programmeertaal', een serie conventies welke gevolgd dient te worden, teneinde de computer dat te laten doen wat in bepaalde omstandigheden gewenst is. En dat wat gewenst is, is uiteindelijk de oplossing van het probleem waar de gebruiker een oplossing voor zoekt.

Prototyping moet nu worden ingezet om het effect van de taalbarrière tussen gebruiker en programmeur te verkleinen. De gebruiker maakt mèt zijn probleem zijn wensen kenbaar. Deze wensen worden door de programmeur 'vertaald' in precieze eisen aan welke de oplossing van het probleem moet voldoen. Dit moet zodanig gebeuren dat de gebruiker in staat is met eigen ogen het effect van zijn wensen te beoordelen: de programmeur moet een prototype, een voorbeeld, maken dat op een computer uit te voeren is en waarin de programmeursinterpretatie van de gebruikerswensen verwerkt is.

De bij het construeren van een prototype te gebruiken ontwerpprincipes worden in hoofdstuk 2 behandeld. Een van de basisprincipes is *consistentie* of *consequentheid*. Teneinde het ontwerp overzichtelijk te houden wordt het in drie lagen verdeeld: de *architectuur*, dat wat de gebruiker van het product ziet, de gebruiksmogelijkheden ervan, de *implementatie*, de wijze waarop (logische) bouwstenen en 'cement' moeten worden gebruikt om de functies welke in de architectuur beschreven zijn gestalte te geven, en de *realisatie* ervan, waarin de fysieke vorm van het product gestalte krijgt. Dit zou men kunnen vergelijken met een impressietekening van een huis en het bijbehorende bestek, de bouwtekeningen ervan, en het huis zelf.

In het tweede hoofdstuk komen ook de eisen ter sprake waaraan zo'n prototype programmeertaal moet voldoen. Een van de belangrijkste eisen is dat de programmeur de mogelijkheid moet hebben op voldoend hoog abstractie niveau te kunnen blijven werken in zijn 'instructie' van de computer. Alles wat hem van de oplossing het probleem afhoudt, moet vermeden worden. Dit zijn vooral de huishoudelijke zaken welke onlosmakelijk verbonden zijn met het 'instrueren' van de computer.

Het overgrote deel van de programmeertalen heeft als onderliggend beeld van de computer de 'ladenkast' waarbij alles in de 'laadjes' wordt opgeborgen. Dit is het Von Neumann model, waarin instructies èn gegevens in dezelfde ruimte worden ondergebracht. In zo'n model zal een groot deel van de tijd van de programmeur moeten worden besteed aan het op juiste wijze achter elkaar laten volgen van de instructies. Deze hebben op hun beurt weer de juiste gegevens nodig en de resultaten dienen op de juiste plaats te worden achtergelaten. In de afgelopen veertig jaar is er natuurlijk geweldige vooruitgang te zien geweest in de hulp die de programmeur kreeg bij het verrichten van deze huishoudelijke taken, maar nog steeds dient de programmeur een groot deel van zijn tijd aan deze zaken te besteden. Tijd die hij beter kan besteden in de interactie met de gebruiker om het probleem scherp te krijgen, waarna het helder te beschrijven valt.

Functionele talen gaan niet uit van precieze instructies hoe een oplossing te verkrijgen, maar van voorwaarden waaraan een oplossing moet voldoen (ook wel, 'vergelijkingen') waarna het systeem achter de taal bepaalt of er oplossingen zijn. Het basisbegrip hierin is de wiskundige functie toegepast op haar argumenten, een functie heeft niets anders nodig. Een functie met

[2] Vele begrippen worden in de Engelse versie van relevante literatuurverwijzingen voorzien. Voor hen die die versie niet willen lezen, moet worden volstaan met deze algemene verwijzing.

argumenten heeft een waarde, en die waarde kan op zich weer een functie zijn. Deze eenvoudige bouwsteen met deze eigenschappen — een functie welke een functie als resultaat kan hebben — blijkt minstens even krachtig te zijn als de gewone, derde generatie, programmeertaal. Op deze wiskundige ondergrond kan een taal worden geconstrueerd die eenvoudig bewijsbaar correcte programma's oplevert, en daardoor bovendien de prettige (algebraïsche) eigenschap bezit dat functies gèèn neveneffecten hebben. De programmeur kan een functie te allen tijde gebruiken zonder rekening te hoeven houden met onverwachte en onvoorziene gevolgen van het gebruik van die functie in andere delen van zijn programma. Zo zal niemand het vreemd vinden dat in een algebra som of goniometrie opgave, als $a = 3$ of $\alpha = 30^0$, deze in de gehele uitwerking onveranderd blijven. In functionele talen is dat eveneens zo, in tegenstelling tot gewone programmeertalen waar dat in het geheel niet vanzelfsprekend is, reden van een zeer grote klasse fouten.

Eén eigenschap die in de implementatie van een functionele taal vaak eraan wordt toegevoegd, tilt haar zelfs ver boven het niveau van de gewone programmeertaal uit: *lazy evaluation*, de waarde van een functie slechts dan, en alleen dan, bepalen als dat strikt noodzakelijk is. Met deze eigenschap kunnen oneindige processen op eenvoudige wijze worden beschreven en kan met deze processen worden gemanipuleerd.

Efficiency is een belangrijk begrip bij het gebruik van computers. Niet alleen in een snel programma (want dat is maar één onderdeel van de totale probleemoplossingscyclus), ook in het tijdsgebruik van een programmeur. Het inzetten van reeds bestaande oplossingen is hierbij een belangrijk middel. Soms verloopt (de ontwikkeling van) een functionele programma sneller op een andere manier dan wanneer men puur functioneel blijft ontwikkelen. In de functionele prototyping taal dient dan communicatie mogelijk te zijn met de buitenwereld. Dit betreft dan:

- communicatie met reeds bestaande oplossingen;
- gebruik van notatiewijzen waarin een deelprobleem sneller kan worden beschreven dan in de functionele taal;
- communicatie met systemen welke een efficiëntere oplossing bieden voor een deelprobleem.

De hierboven geschetste voordelen van de functionele talen zouden deze talen een veel breder toepassingsgebied moeten kunnen geven. Naast het vergroten van de bekendheid van deze talen met hun voordelen, zal aan het communicatieaspect ervan aandacht moeten worden geschonken: niemand zal een geïsoleerde taal in de practijk gebruiken, de ontwikkeling is in de richting van 'open' systemen.

In het derde hoofdstuk worden enkele basisbegrippen behandeld welke in de volgende twee hoofdstukken gebruikt worden, of op een andere wijze relevant zijn voor het onderwerp:

- de plaats van de functionele talen in het scala van programmeertalen;
- het begrip *functie*;
- het werken met functies en het programmeren in een functionele taal;
- het model dat aan de functionele talen ten grondslag ligt: de *lambada calculus*;

- enkele noodzakelijke begrippenapparaten nodig voor de implementatie van sommige functionele talen: *combinatorische logica* en *graaf herschrijf technieken*.

Functionele talen blijken te voldoen aan de eisen welke in hoofdstuk 2 aan een taal voor prototyping werden gesteld, behalve voor wat betreft de communicatie met de buitenwereld. Teneinde het gebruik van functionele talen te bevorderen, laten we in de hoofdstukken 4 en 5 van dit proefschrift dan ook de uitwerking van dit aspect zien. Teneinde het te ontwikkelen communicatie hulpmiddel te demonstreren en te toetsen op practische bruikbaarheid, is gekozen voor de functionele taal Twentel.

Hoofdstuk 4 laat de ontwikkeling zien van een zeer algemene functie welke de gewenste communicatie met de buitenwereld verzorgt: de *trapdoor*. Eerst op het niveau van de *architectuur* van de oplossing: hoe moet de programmeur tegen de 'trapdoor' aankijken, hoe kan hij hem gebruiken, en waar moet hij rekening mee houden bij het gebruik. Ook wordt aandacht besteed aan de andere kant van de 'trapdoor', de buitenwereld die met Twentel communiceert door middel van de 'trapdoor'. Verschillende gebruiksmogelijkheden worden geschetst, onder meer de communicatie met een groot Pascal programma en de verbinding met een SQL *server* in de vorm van een Twentel prototype van een relationele database.

In het volgende hoofdstuk wordt de *implementatie* van de 'trapdoor' beschreven, met de bijbehorende herschijfregels die in de implementatie van Twentel zelf moeten worden opgenomen. De communicatie tussen Twentel en de buitenwereld verloopt door middel van het gebruik van een twee-dimensionale beschrijving van de door de 'trapdoor' naar de buitenwereld over te zenden gegevens (de argumenten van de functie in de buitenwereld), en *vice versa*, het functieresultaat weer terug naar Twentel. Het model van de conversie van gegevens in deze beschrijving naar een Pascal record, en terug, wordt beschreven in een functionele taal (Twentel). Tot slot van het hoofdstuk worden de problemen beschreven die optraden bij de *realisatie* van de 'trapdoor' in Twentel. De omstandigheden waarin de 'trapdoor' aan de tand is gevoeld komen eveneens ter sprake.

Het proefschrift wordt besloten met een korte terugblik op de probleemstelling (verbreding van het gebruik van functionele talen door inbouw van communicatiemogelijkheden), de mate waarin het probleem is opgelost en een schets voor verder werk in deze richting.

Curriculum Vitae

Henk van Dorp (*ref* titlepage) completed in 1964 his secondary education (with H.B.S.-B diploma) from the 'Willem de Zwijger Lyceum' in Bussum.

The same year he started his studies Electrotechnical Engineering at the new campus university, the *Technische Hogeschool Twente*, in Enschede. On 15 August 1967 he was engaged as a student assistant at the Chair of Numerical Mathematics and Programming Methodology of Professor A.J.W. Duijvestijn, with as main activities in the following four years assisting with the Algol and Assembler/360 laboratories and being an active member of the *Vertalergroep*, led in a spiritually non-deterministic way by Mr. H. van Berne, and with Mr. P. Eilers as Honorary Member.

He received his Bachelors degree (B.t.w.) in 1969 (subject: fully deterministic program analysis by means of syntactical subroutines) and his Master's degree (ir) in 1972 (subject: see Ch 1 ref [10]; work on the thesis performed while at Philips, Eindhoven ('I.S.A.-R', the research group of the staff department on information systems and automation)).

After doing his bit in the Air Force, he went in 1974 to the *Academisch Ziekenhuis* in Leiden, and took a position in the *Centrale Dienst Informatie Verwerking* (CDIV). The first years however, he worked on frog's leg nerve potentials and cat's respiratory data (and, in his spare time, on a Lisp interpreter for the IBM 1800) within the *Laboratorium voor Physiologie* of the Medical Faculty of Leiden University, on an exchange basis with the hospital.

Then, in 1976, back in the CDIV at the hospital, he undertook the OPERA project, the registration of administrative surgical, anaesthetic, and personnel data of operations performed within the main operating theatres of the General Surgery Department (OKHK), culminating in a legal link *à titre personel* between OKHK and the CDIV. The next project dealt with registering diagnoses throughout the hospital, for use within the hospital. These projects were a small part of the Dutch NOBIN-ZIS project, which was started in 1972 at the Leiden University Hospital. Its main aim was to acquire knowledge and experience with the development and use of a full scale hospital information system.

In 1981 he became manager of Spirit (*ref* Perlis), a development group within the Application Development Sector (project domains a.o.: financial, laboratory, anti-coagulation dosage, data acquisition, and connecting PC's) of the organisational unit in the hospital that was to become the independent BAZIS Foundation.

In 1992 he joined an AIM project (SEISMED), creating guidelines for the development and design of secure systems in a health care environment. Subsequently, in the spring of 1993 he became a member of CEN/TC251 (Medical Informatics)/WG 6 (Security).

Colophon

This dissertation came into being by adapting the handwritten text — while typing it — to the TEX and LATEX typesetting conventions from Donald E. Knuth [5] and Leslie Lamport [8] respectively. In the process we also used some tips and tricks from the Internet [4] regarding "typesetting beautiful documents" (R. J. Drofnats [5], p 24). In the final stages, sound and critical advice on style and looks, and other things, was given by Jan Vanderschoot.

The main text font is Computer Modern, 11 pt, designed by Donald E. Knuth [6, 7], with additional fonts. The actual typesetting process has been handled by `emTex` 3.0a, by Eberhard Mattes.

The printing of draft versions and the more stable versions was done with `dvihplj` 1.4d, another Eberhard Mattes program. The draft versions appeared on a HP-Laserjet IIId, the more stable versions and the camera ready final version emerged from a HP-Laserjet 4, the jets stationed at the BAZIS premises.

Production of the dissertation was done with a Xerox 5100, printed on Colotech white (100 g/m^2) after a 80% reduction in size from the camera ready copy. The cover (design by the author, more information on p 87) was printed on Lustrulux Colour red (250 g/m^2) by Multicopy, Leiden. The *Copyshop Pre-klinische Laboratoria* in the *Sylvius Laboratorium* of the Medical Faculty of Leiden University handled the production.

There is still room for the answer to a 'last' question: Why has *Alice*[1] been mentioned so many times in this dissertation? The reason is, apart from Perlis' opening quotation and the author's preferences, also a result of Perlis' epigram [9] on *Alice*:

> "The best book on programming for the layman is *Alice in Wonderland*; but that's because it's the best book on anything for the layman."

[1] Generally spoken, 'Alice' means both [1] and [2].

So only one word remains to be written, "Impenetrability[2]!"

References

[1] Lewis Carroll, *Alice's Adventures in Wonderland.* Macmillan, London, 4 Jul 1865.

[2] Lewis Carroll, *Through the Looking Glass and What Alice Found There.* Macmillan, London, 1871.

[3] Lewis Carroll, *The Wasp in a Wig. A "Suppressed" Episode of 'Through the Looking Glass and What Alice Found There'.* with Preface, Introduction and Notes by Martin Gardner, The Lewis Carroll Society of North America, New York, 1977 (tacitly cited in this dissertation).

[4] Brendan P. Kehoe, *Zen and the Art of the Internet — A beginner's guide to the Internet.* First ed, Revision 1.0, `brendan@cs.widener.edu`, Widener Univ, Chester PA, 2 Feb 1992.

[5] Donald E. Knuth, *The TEXbook.* Vol A *'Computers and Typesetting'*, Addison–Wesley, Reading MA, 1986.

[6] Donald E. Knuth, *The* METAFONT *book.* Vol C *'Computers and Typesetting'*, Addison–Wesley, Reading MA, 1986.

[7] Donald E. Knuth, *Computer Modern Typefaces.* Vol E *'Computers and Typesetting'*, Addison–Wesley, Reading MA, 1986.

[8] Leslie Lamport, LATEX *: A Document Preparation System.* Addison–Wesley, Reading MA, 1986.

[9] Alan J. Perlis, "Epigrams on Programming", *SIGPLAN Not* **17**(9) (Sep 1982): 7–13.

[2] Finally we can complete the quote from [2] that was started in Footnote 3, Chapter 3, page 59:

"Alice was too much puzzled to say anything, so after a minute Humpty Dumpty began again. "They've a temper, some of them — particularly verbs, they're the proudest — adjectives you can do anything with, but not verbs — however, *I* can manage the whole of them! Impenetrability! That's what *I* say!"

"Would you tell me, please," said Alice "what that means?"

"Now you talk like a reasonable child," said Humpty Dumpty, looking very much pleased. "I meant by 'impenetrability' that we've had enough of that subject, and it would be just as well if you'd mention what you mean to do next, as I suppose you don't mean to stop here all the rest of your life."

"That's a great deal to make one word mean," Alice said in a thoughtful tone.

"When I make a word do a lot of work like that," said Humpty Dumpty, "I always pay it extra."

"Oh!" said Alice. She was too much puzzled to make any other remark."