Contents

List of Figures

Chapter 1

Introduction

The set of valid inputs to any program can be regarded as a form of computer language – an *input data language* [Pra75, 5]. The set of possible outputs can similarly be regarded as a computer language. Furthermore, a program can be thought of as a *translator* of some input data language to an output data language. The central thesis of this dissertation is that this *language oriented* view of programs is not only valid but that it leads to the establishment of a new declarative style of programming in which program design becomes language design.

The practical way of exploring the implications of the language oriented view of programs and of assessing the potential of the language oriented style of programming is to design and implement a programming language and system that provides support for the new programming paradigm. For this purpose, the design and implementation of a programming language, called META-LISP, is presented here, together with a number of case studies of language oriented programming in META-LISP. In addition to providing evidence for the main thesis of this dissertation, the case studies serve also as points of comparison with the functional and the logic programming paradigms.

1.1 The Language Oriented View

Establishing the claim that any program can be regarded as a translator from an input data language to an output data language is simple enough. It depends solely on providing a broad enough notion of a data language. Formally a language is just a set of sequences of symbols drawn from an alphabet. If we allow this alphabet to include not only characters but key clicks, flashes on a screen or any discrete well defined behaviour that a computer

can exhibit in a sequence,[1] then the claim that the input as well as the output can be regarded as languages will be established. Considering further that a translation is just a mapping, and that a program can be thought of as a mapping from its input to its output, the above claim turns out to be just a variant of the black box model of programs with sequential input and output.

The more interesting question is why it is worthwhile to regard programs as translators and how it leads to the establishment of a new programming paradigm. To answer these questions it is necessary to consider the possible implications of the language oriented view of programs for the program design process itself.

Thinking about programs as translators leads naturally to considering the possibility of programs being designed and built *as* translators. Drawing on the experience of building translators using syntax-directed techniques, we can begin to envisage what would be involved in the design of programs *as* translators: first, an explicit grammatical description of the input language of the program would need to be provided; second, the grammatical structure imposed on the set of valid inputs by this description could then be used as a framework to prescribe the appropriate actions to produce the desired output. From a methodological point of view both steps are valuable. In conventional programming the grammatical specification of the set of valid inputs to a program can only be part of the documentation. It can contribute greatly to the clarity of a programming style if this can be incorporated into, and in fact help to structure, the program itself.

On the face of it, the second step appears to be problematic. In most translator writing systems (TWS), the semantic actions that are used to prescribe the output of the translator are written in the host language (e.g. LISP, C, Pascal or Prolog etc.) of the tool. Needless to say, none of these languages explicitly supports the design of programs *as* translators. When using such tools in practice, all the clarity that characterises the overall design of a translator can be swamped by the complexity and often opacity of the way semantic actions are specified.

Maintaining the clarity and uniformity of the design of programs as translators requires a translator writing tool in which the semantic actions can be specified in a way that preserves the language oriented view of programs. This can be achieved, firstly, by viewing all non-primitive procedures in the semantic actions as programs/translators in their own right, and secondly, by stipulating that these procedures are to be elaborated as translators in their own right. In other words, what is required is that the design of these procedures starts from an explicit grammatical description of their input language and uses the structure thereby

[1] this excludes multiple asynchronous sources of input as well as parallel output.

imposed on the input as the framework for specifying further actions etc. Accordingly, a programming system to support a language oriented style of programming would have to be a translator writing system that allows the elaboration of all the non-primitive procedures that appear in the semantic actions as translators in their own right. The programming language and system, called META-LISP, was designed specifically as such a system. Its design is reviewed in the next section.

1.2 The Design of META-LISP

LISP was chosen as the implementation language for the envisaged language oriented programming system. Given LISP's reputation as an "implicit meta-language" (i.e. a language to be used to define other languages and translators [Ing66, 115-6]) the choice of LISP to implement a meta-language seemed all the more appropriate. The affinity between the intended system and LISP goes beyond considerations of ease of implementation. This is reflected in the name of the programming language. The Meta in META-LISP is intended both as an indication of the meta-linguistic power of the system as well as the fact that it is built on top of LISP.

Bootstrapping was used extensively not only in the construction of a compiler for the language but in its design and documentation in the form of a *denotational* style *meta-circular interpreter*. The primary objective of the development of a meta-circular interpreter for META-LISP has been to provide a complete operational definition of the language in the same way that a meta-circular definition plays a role in the definition of LISP. [All78, 162] The main goal of the development of the compiler has been efficiency. Both the interpreter and the compiler have been tested on a range of sample programs, including all the case studies presented here, with identical results.

The primary design objective for META-LISP was to provide linguistic support for the language oriented paradigm in the form of an appropriate translator writing system. This left many design decisions underdetermined. Although it made specific demands on the Semantic Language it left a free choice for the other main component of the system, the underlying grammatical formalism.

The choice of underlying grammatical formalism is always a critical factor in determining the efficiency of the operations of the translators definable by a TWS. In preference to Context Free Grammars (or their restricted forms) the Top Down Parsing Language (TDPL) described by Aho and Ullman [AU72] proved to be a particularly attractive choice of underlying grammatical formalism. The *definitional power* of context-free grammar can

be regarded as *excessive*, in the sense that it is difficult to envisage the consequences of such definitions. The primary reason for this is that CFG can be ambiguous. Another, related problem is that backtracking may become necessary at any point. This, in itself, can render any attempt at envisaging possible forms of the sentences of a language defined by a CFG inherently incomplete, in the absence of machine support. In contrast, the main advantage of TDPL lies in the fact that, as a language definitional formalism, it is tied to a – transparent and efficient – top down parsing algorithm in which backtracking is limited. Equally important is the fact that it is an unambiguous grammatical formalism. Consequently, it is easy to envisage the possible forms that representative sentences of the language can take, even without machine support. Naturally, TDPL could only be a starting point for the design of a grammatical formalism suitable to be the basis of a general purpose TWS. The most fundamental enhancement of the original language definitional formalism was the addition of capabilities of specifying arbitrary (nested) list structures and facilities to test and select specific components of the input. Further enhancements include the allowance for a limited form of explicit left-recursion.

For the Semantic Language of META-LISP, the second main component of the language, the natural choice was a simple *applicative language* with a limited number of extra features and a non-standard notion of application. The standard notion of application is the application of a function (or procedure) to given arguments. When programs or even procedures are viewed as translators from an input language to an output language, we can talk sensibly only about a single "argument" that they could receive viz. a sequence of symbols which may or may not form a sentence of their input language. Accordingly, in the Semantic Language of META-LISP, the input parameters to the procedures appearing in the semantic action are not given as "arguments" but are first made into a single input list which is then passed to the procedure as its single input. It is then the responsibility of this procedure to determine the structure of this input and produce appropriate output in the process. What this amounts to is that the calling mechanism employed in the Semantic Language is Syntax-Directed Translation. Procedure calls with a given number of specific arguments can be regarded as special cases of this concept of procedure application when the input has a particularly simple structure. In addition to the ability of invoking procedures as described, the Semantic Language of META-LISP contains features that allow the construction of arbitrary list structures from given components (like the backquote of LISP), assignment and reference of attributes, and the use of procedural parameters.

Discounting the addition of a control primitive for forcing backtracking, the model of computation of META-LISP is essentially the same as in syntax-directed translation. The

only significant difference lies in the way procedures in the semantic actions are invoked, and the way they are defined, viz as translators. The result is a new kind of programming language which has, as its distinguishing feature, the use of syntax-directed translation as its parameter passing mechanism.

META-LISP's syntax-directed parameter passing mechanism can be contrasted with the calling mechanism of modern functional languages like ML, [Wik87], which utilises pattern matching. As syntax-directed translation properly subsumes pattern matching, META-LISP can offer capabilities not possessed by functional languages, like ML: these include support for *data abstraction*[ASS85, 72], (see page 42) *representation independent* or *level-wise* programming see [All78, 53-5], as well as support for parser construction. The conflicting requirements of pattern matching and data abstraction have been discussed by Wadler in [Wad87, 307]. "Pattern matching depends on making public a data type representation, while data abstraction depends on hiding the representation." The desirability of facilities for language processing can be judged from the point made by Wikström that a parser generator is a tool that should accompany an ML system for production use [Wik87, 294].

What is common to both ML as a functional language and META-LISP as a language oriented programming language, is that they both make commitments about which quantities are inputs and which are outputs. This can be contrasted to logic programming languages, such as Prolog, that do not make such commitments [Red86, 3]. The multidirectionality of Prolog is a consequence of the fact that Prolog uses unification as its calling mechanism [DFP86, 45]. It gives capabilities to Prolog not possessed by ML, or META-LISP for that matter. Through the use of operator declarations Prolog also has explicit language definitional capabilities. In terms of expressive power, Prolog is clearly superior. In fact Prolog's expressive power can be said to be so great, that it may even be regarded as *excessive*, in the sense, that Prolog programs can give rise to computations that the programmer had never thought of, due to full backtracking and multidirectionality. In most cases, this leads to the need to rely on the impure facilities, such as the *cut*, to systematically cut out the excess. Mode declarations are also used for the purpose of limiting the definitional power of a particular clause. The most important design decisions for META-LISP have in fact been informed by the attempt to avoid excesses of definitional power. The most significant of these decisions have been not to use Context-Free Grammars as the underlying grammatical formalism. It can be argued, that CFG's themselves suffer from an excess of definitional power, that is analogous to that of Prolog, viz unlimited backtracking. In terms of support for data-abstraction Prolog does not fair any better than ML.

In terms of expressive power META-LISP as a programming language occupies a kind

of mid-way point between functional and logic programming languages. META-LISP has greater expressive power for defining the set of valid input to a program than say ML. (META-LISP can define it as a language, whereas ML can only define it as a pattern). Prolog on the other hand can be said to define the set of valid inputs to a program as the set of unifiable terms. This is a far greater, and less comprehendable set than what can be defined using META-LISP. The fact that META-LISP, in terms of expressive power occupies the middle ground between ML and Prolog does not mean that META-LISP was designed with this objective in mind. To some extent it is a consequence of a conscious attempt to avoid *excesses* of definitional power. However, there were other, broader motivations that have prompted its development. As an indication of the aspirations of the current work, the following section discusses some of the motivations for it.

1.3 Motivation

The idea that every program implicitly defines an input data language led K. John Gough to suggest that "the ideas of language processing can and should be applied to the design of almost any program" and then go on to contend that "in all situations the input language of a program should be formally defined, and then implemented by systematic techniques". [Gou88, 2] The language oriented view is very similar to his position, except for requiring language definitions to be given in a form that allows the generation of appropriate language processors by "automated" means. META-LISP provides just such means.

Perhaps the richest source of motivation for the development of language oriented programming is the LISP tradition. Abelson and Sussman in their classic textbook *The Structure and Interpretation of Computer Programs* invite us to regard "almost any program as the evaluator for some language" and suggest that "the technology for coping with large-scale computer systems merges with the technology for building new computer languages, and computer science itself becomes no more (and no less) than the discipline of constructing appropriate descriptive languages." [ASS85, 294-5] Accepting the suggestion that any program can be regarded as an evaluator (or interpreter) for a "special-purpose language" for dealing with a given problem domain, leads naturally to a consideration of the use of translator writing tools to define these special-purpose languages as the language oriented way of writing programs as *interpreters*.

In discussing possible improvements to LISP, John McCarthy at the 1980 LISP Conference considered the question whether syntax-directed translation should be a feature to be added to LISP or whether it should be the basis of a new language. McCarthy's response

to this question was that "both the functional form of computation that LISP has now and syntax directed features are wanted in one language." [McC80, vii]. META-LISP is proposed as just such a language.

J.L. Bentley in the *Programming Pearls* column of the ACM Journal under the title "Little Languages" [Ben86] discusses the implementation of a program for the typesetting of pictures as a compiler for a "little language" using a compiler-compiler. In a companion paper [BK86] the advantages of designing a "little language" and implementing it as a compiler are identified in that it "gives users a concise specification of how to use the program, provides an organising framework for implementation, and often enables the implementor to use tools to build the program."

Similar views are expressed by the developers of the EQN typesetting system for mathematics: "Defining a language, and building a compiler for it with a compiler-compiler system seems like the only sensible way to do business. Our experience with the use of a grammar and a compiler-compiler has been uniformly favourable. If we had written everything into code directly, we would have been blocked into our original design. Furthermore, we would have never been sure where the exceptions and the special cases were. But because we have a grammar, we can change our minds readily and still be sure that if a construction works in one place it will work everywhere." [?] These are admirable software engineering qualities indeed. The real motivation for the development of language oriented programming is to be able to do similarly, in a much wider domain than previously thought possible, by turning the methodology and technology of the "little languages" strategy into a general purpose programming methodology and technology.

To sum up: the LISP tradition invites us to view programs as *interpreters* for special-purpose languages. Experience in the development of scripting languages (e.g. EQN, GRAP, PIC etc) teaches us the value of viewing programs as "compilers" for "little languages." As both interpreters and compilers are but translators, the language oriented view of programs can be seen to encompass, as well as generalise both previous approaches.

1.4 Related Work

1.4.1 Towards Language Oriented Programming

The potential of extending syntax-directed techniques to be used as a general-purpose programming system had been investigated before. In this respect the works of David Sandberg on the LITHE programming language [San82], Stefan Feyock on Syntax Programming [Fey84], Yoshiyuki Yamashita and Ikuo Nakata on Coupled Context Free Grammars [YN88]

deserve special mention.

LITHE, an experimental programming language, combines the ideas of syntax-directed translation and the concept of classes. The LITHE system allows for the formulation of a semantic action "as a string that is translated into a sequence of actions by using other rule action pairs". In this regard LITHE can be said to contain the most important technical contribution that makes possible the extension of syntax-directed techniques into a general purpose technique. This possibility, however, is not exploited fully as the work is aimed primarily at extensibility by allowing the user to "freely choose his own syntax".

Feyock's starting point is "the strong formal similarity of BNF (Backus Normal Form) productions to Horn clauses". He goes on to describe a "new programming technology that is to syntax analysis and parser construction as formal logic is to logic programming (LP)" which is then accordingly named *Syntax Programming.* Feyock could be credited spotting the potential for developing a new programming technology based on the idea of syntax directed translation. However, this objective is not fulfilled since the semantic actions are written, again, in the host language of the parser (LISP or Pascal).

The strong formal similarity of Prolog and BNF rules has received much attention in recent years. It seems that we have come full circle. When Prolog was first introduced, its proof strategy was made plausible by comparison with top down parsing. See [Kow79, Chapter 3]. By the end of the eighties, the new generation of computer scientists were more readily familiar with Prolog, so that parsing technology and attribute grammars are viewed frequently from a logic programming viewpoint. In a recent paper [DM88], entitled *"A Grammatical View of Logic Programming"*, Pierre Deransart and Jan Maluszynski have succeeded in showing that the declarative reading and the procedural reading of pure logic programs can be complemented by a grammatical reading where the clauses are considered to be rewrite rules. Their aim has been to show that "this point of view facilitates transfer of expertise from logic programming to other research on programming languages and vice versa." One particularly interesting example of this "transfer of expertise" is provided by the paper, in the same volume, by Yoshiyuki Yamashita and Ikuo Nakata [YN88] that introduces the idea of a Coupled Context Free Grammar and shows that these are duals of equivalent logic programs. This work is of special interest here as CCFG is put forward explicitly as an "extension of the syntax-directed translation schemes or attribute grammars. Although these schemes have provided excellent tools in the field of program translations, their expressive powers are too small to be used as general purpose programming systems."

I would like to close this subsection by mentioning the work of Stephen Adams at the University of Southampton. In terms of its aspiration, his work is the closest to the present

work. This is immediately evident from the very title of his report: *Towards Language-Oriented Programming.* [Ada90]. In the report Adams describes *language-oriented* programming as the "definition and use of 'designer' programming languages." (page 18) He proposes a *Language oriented* methodology which involves

- Understanding of all different phenomena in terms of language.
- Definition and delineation of programming language fragments.
- Re-use of language components.

Adams also envisages an environment supporting language oriented programming as one that would provide tools for defining language fragments as well as the mechanism of combining them to produce new language fragments. In the report, he goes on to investigate the design options for a module system that can be used to combine language fragments. He also investigates partial evaluation as the means of efficient execution of languages and language tools.

1.4.2 Language Development Tools

The evolution of language development tools has its origin in Backus's invention of a notation for describing the syntax of programming languages. This notation was an overnight success. Backus himself expected at the time that he would have a solution just as neat for dealing with Semantics. [Wex81, 89] Although his hopes were not to be fulfilled, the notation that he invented was soon incorporated into practical compiler-compilers.

One of the earliest example of such a system is Meta-II [Sch64]. It is of interest in that the underlying grammatical formalism that it uses is very similar to the Top-Down Parsing Language.

Jed Marti's Little META Translator Writing System [Mar83] is interesting in that it is written in LISP as part of the Portable Standard LISP Project. There are a number of similarities between this system and META-LISP:

- the translators it produces are modifiable without complete recompilation
- the system was built using bootstrapping
- it compiles into LISP

The most important difference between little META and META-LISP is that the former is aimed exclusively at compiler and parser generation. The possibility of defining some of

the functions used in the semantic actions in terms of little META rules is not even being considered.

One of the most widely used compiler-compilers is YACC [Joh79]. For a practical introduction to compiling techniques using YACC see [Ben90]. YACC is used to generate a LALR(1) parser from context-free grammars specified in the appropriate form. Each rule of the grammar has semantic actions associated with it. These are written in C. There are three improvements that have been recently proposed for YACC. They are of interest in the present context, not only for what they provide, but what they reveal about the 'shortcomings' of YACC.

The first improvement, proposed by Purtilo and Callahan in their NewYacc system, concerns the retention of the parse tree after a sentence has been accepted by YACC. The parse tree can then be traversed to carry out additional actions. These actions are controlled by rewrite rules associated with language productions in the NewYacc grammar extensions. The system "presents the *look* and *feel* of attribute grammar without sacrificing the simplicity of using normal yacc declarations." [PC89].

The second improvement concerns the incremental generation LALR(1) parsers. Horspool describes an incremental parser generator, called ILARL, which permits the user to "modify a grammar one rule at a time and reporting problems to the user as soon as they are apparent" [Hor89, 128-9]. It also allows the user to specify, or even change, the start symbol of the grammar.

The proposed third improvement to YACC was to provide the designer of a YACC grammar a method of tracing a parser as it uses the grammar. See [FSO91].

All these features – the *look* and *feel* of attribute grammars, the incremental construction of parsers as well as tracing facilities – which were absent in compiler-compilers such as YACC are incoporated into the META-LISP system, viewed as a compiler-compiler. In fact, since the parsing strategy supported by META-LISP is top-down, the trace can be more meaningful, than in the case for bottom up parsers. The trace of a bottom up parse gives no indication of the overall structures being explored. It only shows how a successful parse can be built up from the bottom up. In the context of language oriented programming, try to imagine what a bottom up trace of the execution of a program would look like. First the lowest level actions would be seen, and only at a later stage would the trace indicate that in fact, what these lowest level operations all add up to is, say, reading a line of input. Whereas, in a top down trace, we would know right away that the program is trying to read a line of input which then involves sub tasks which lead to further, more elementary tasks, etc.

1.5 Dissertation Outline

Chapter 2 introduces basic background material such as the concepts of formal languages, their definition, parsing and translation. The Top Down Parsing Language, which forms the basis of the underlying grammatical formalism of META-LISP is also introduced in this chapter.

Chapter 3 provides an overview of META-LISP. Following two introductory examples, META-LISP is then introduced in two sections. First the the language definitional formalism is introduced, then the Semantic Language.

Chapter 4 presents a number of case studies designed to illustrate the process of language oriented programming in META-LISP. Section 1 presents simple examples of list processing in ML as well as META-LISP. Section 2 develops a complete program for Symbolic Differentiation. Section 3 presents a language oriented design of the program. Section 4 reuses parts of the differentiation program for approximating the roots of differentiable functions using the Newton-Raphson method.

Chapter 5 contains two further case studies. The first presents solutions to a simple path finding problem given both in Prolog and META-LISP. The final case study is a program for the graphical display of trees. Although this program can be thought of as a "compiler" for a "little language," it is shown that standard compiler-compiler technology would not be adequate as the vehicle of its implementation.

Chapter 6 illustrates how META-LISP can be used to write denotational language definitions. The denotational definition of the language of a simple Calculator will be developed alongside the description of a denotational style interpreter for it in META-LISP. The primary objective of this chapter is to introduce the format of denotational definitions in META-LISP. The same format will be used in Chapter 7 in defining the semantics of META-LISP itself.

Chapter 7 formalises the operational semantics of META-LISP in the form of a denotational style meta-circular interpreter.

Chapter 8 discusses the strategy that was used in the implementation of the META-LISP system, its current status and future developments.

Chapter 9 provides comparisons with other paradigms and programming languages including ML, Prolog and LISP. The conclusion is formulated as much on the basis of a retrospective critique of META-LISP as on the basis of a prospective look at future work aimed at improving the technological and the linguistic support that can be given to language oriented programming.

Chapter 2

Background

The aim of this chapter is to review the basic concepts and terminology of formal languages, their parsing and translation. Much of the material is related to syntactic issues of formal languages, i.e. concerning the rules for determining which sequences of symbols are well formed sentences of a given language and which are not. Issues concerning the semantics or meaning of formal languages will not be dealt with, except for introducing the notion of *translational semantics*. The definitions in this chapter follow the treatment of formal languages in [AU72].

The Chapter is organised as follows. Section 1 introduces the concept of a grammar as the means of specifying the syntax of a language. A grammar, however, does not only define a language, but also imposes a structure on the set of sentences of a language. This structure can be illustrated pictorially in the form of a *syntax* or *derivation trees*. The introduction of the concept of a derivation tree in Subsection 2.1.2, provides the means of examining some of the structural differences that can arise between equivalent grammars. In Subsection 2.1.3 the discussion centers on the difference between *left recursive* and *right-recursive* formulation of equivalent grammars. The process of determining if a sequence of symbols can be generated by a grammar is usually referred to as *parsing* or *syntax analysis*. Parsing methods are discussed briefly in Section 2, followed by the definition of an efficient and transparent formalism for language definition and syntactic analysis, known as the Top-Down parsing Language (TDPL) which forms the core of META-LISP. The last section of this chapter introduces the idea of *syntax-directed translation* and its formalisation in the form of *attribute grammars*.

2.1 Language Definition

From a formal point of view, a language is simply a set of sequences of symbols drawn from an *alphabet*. Sequences of symbols drawn from an alphabet that belong to a given language are usually called sentences.

Definition 2.1 Let Σ be a set of symbols, called an *alphabet*. A sequence $s = t_1, t_2 \ldots t_n$ of symbols drawn from some alphabet Σ is called a string over the alphabet Σ. The empty sequence of symbols is referred to as the *empty string*, denoted $\langle\rangle$. Let Σ^* denote the set of all strings over the alphabet Σ including the *empty string*.

A *language* L (over the alphabet Σ) is a subset of Σ^* i.e. it is a set of strings over an alphabet.

2.1.1 Grammars

The rules that determine the construction of the well formed sentences of a language are usually given in the form of a *grammar*.

Definition 2.2 A *grammar* is a 4-tuple $G = (N, \Sigma, P, S)$ where

1. N is a finite set of *nonterminal symbols* or *syntactic categories*.
2. Σ is a finite set of *terminal symbols*, disjoint from N, called an *alphabet*
3. P is a finite set of *productions* or *rules* of the form $\alpha \rightarrow \beta$, where α is a sequence of terminal and/or nonterminal symbols with at least one non-terminal symbol, and β is a possibly empty sequence of terminal and/or nonterminal symbols.
4. S is a *distinguished symbol* in N called the *sentence* or the *start symbol*.

Definition 2.3 A *sentential form* is a possibly empty sequence of terminal and/or nonterminal symbols that can be formed according to the rules of a grammar.

Definition 2.4 A *terminal string* is a possibly empty sequence of terminal symbols.

Definition 2.5 A *derivation* or (*generation*) step according to a given grammar G is a step by which from a given sentential form another sentential form is obtained by substituting in the sentential form an occurrence of the left-hand side of a rule of the grammar by the right hand side of the rule. Formally, a relation $\underset{G}{\Rightarrow}$ (to be read as *directly derives*) on $(N \cup \Sigma)^*$ can be defined as follows: if $\alpha\beta\gamma$ is a sentential form and $\beta \rightarrow \delta$ is a production in P, then $\alpha\beta\gamma \underset{G}{\Rightarrow} \alpha\delta\gamma$.

Definition 2.6 A *reduction step* according to a given grammar G is a step by which from a given sentential form another one is obtained by substituting in the given sentential form an occurrence of the right hand side of a rule of the grammar G by its left hand side. Formally, a relation $\underset{G}{\Leftarrow}$ (to be read as *directly reduces)* on $(N \cup \Sigma)^*$ can be defined as follows: if $\alpha\delta\gamma$ is a sentential form and $\beta \to \delta$ is a production in P, then $\alpha\delta\gamma \underset{G}{\Leftarrow} \alpha\beta\gamma$.

Clearly a reduction step is the inverse of a derivation step in the sense that if $\alpha \underset{G}{\Rightarrow} \beta$ then $\beta \underset{G}{\Leftarrow} \alpha$.

Definition 2.7 A *derivation* of a sentential form α is a sequence of derivation steps that starts with the sentence symbol of the grammar and leads to the sentential form α. The usual notation for a nontrivial derivation is $S \overset{+}{\underset{G}{\Rightarrow}} \alpha$, where $\overset{+}{\underset{G}{\Rightarrow}}$ denotes the transitive closure of the relation $\underset{G}{\Rightarrow}$.

Definition 2.8 A *reduction* of a sentential form α is a sequence of reduction steps that starts with the given sentential form α and leads to the sentence symbol of the grammar. The notation for a reduction is $\alpha \overset{+}{\underset{G}{\Leftarrow}} S$, where $\overset{+}{\underset{G}{\Leftarrow}}$ denotes the transitive closure of the relation $\underset{G}{\Leftarrow}$.

Definition 2.9 A *sentence generated* or *defined by a grammar* G is a terminal string, w, which is derivable from the start symbol of the grammar G, or equivalently, that is reducible to the start symbol of the grammar G.

Definition 2.10 The *language defined by a grammar* G, denoted $\mathcal{L}(G)$, is the set of sentences generated by G, i.e. $\mathcal{L}(G) = \left\{w | S \overset{+}{\underset{G}{\Rightarrow}} w\right\}$, or equivalently, $\mathcal{L}(G) = \left\{w | w \overset{+}{\underset{G}{\Leftarrow}} S\right\}$,

Grammars can be classified according to the format of their productions. Let $G = (N, \Sigma, P, S)$ be a grammar.

Definition 2.11 G is said to be

1. *Right-linear* if each production in P is of the form $A \to xB$ or $A \to x$, where A and B are in N, i.e. nonterminal symbols, and x is in Σ^*, i.e. is a possibly empty sequence of terminal symbols.

2. *Context-free* if each production in P is of the form $A \to \alpha$, where A is in N, i.e. is a nonterminal symbol, and $\alpha \in (N \cup \Sigma)^*$ i.e. α is a possibly empty sequence of nonterminal and/or terminal symbols.

3. *Context-sensitive* if each production in P is of the form $\alpha \to \beta$, where $|\alpha| \geq |\beta|$.

A grammar with no restrictions as above is called *unrestricted.*

Example 2.1.1 An example of a context-free grammar $G_0 = (\{E, T, F\}, \{a, +, *, (,)\}, P, E)$ where P consists of

$$\begin{array}{lcll} E & \rightarrow & T + E & 1 \\ E & \rightarrow & T & 2 \\ T & \rightarrow & F * T & 3 \\ T & \rightarrow & F & 4 \\ F & \rightarrow & (E) & 5 \\ F & \rightarrow & a & 6 \end{array}$$

$\mathcal{L}(G_0)$, the language defined by the grammar G_0, is the set of arithmetic expressions that can be built up using the symbols $a, +, *, (, and)$.

Example 2.1.2 An example of a *derivation* in G_0 would be

$$\begin{array}{lcll} E & \Rightarrow & T + E & \text{by } 1 \quad E \rightarrow T + E \\ & \Rightarrow & F + E & \text{by } 4 \quad T \rightarrow F \\ & \Rightarrow & a + E & \text{by } 6 \quad F \rightarrow a \\ & \Rightarrow & a + T & \text{by } 2 \quad E \rightarrow T \\ & \Rightarrow & a + F * T & \text{by } 3 \quad T \rightarrow F * T \\ & \Rightarrow & a + a * T & \text{by } 6 \quad F \rightarrow a \\ & \Rightarrow & a + a * F & \text{by } 4 \quad T \rightarrow F \\ & \Rightarrow & a + a * a & \text{by } 6 \quad F \rightarrow a \end{array}$$

In the last step a terminal string is obtained, i.e. a sentence of the language defined by our grammar is derived.

2.1.2 Derivation Trees

A derivation of a terminal string w according to a grammar G exhibits a *proof* that $w \in \mathcal{L}(G)$, i.e. that it is a sentence of the language generated by the grammar G. In that proof the production rules of the grammar play the role of axioms. A derivation begins with the most general syntactic category of the grammar, i.e. the sentence symbol, and by proceeding *down* through more specific sentential forms, ultimately a terminal string is reached. Proofs of this kind are usually referred to as *top down* proofs.

The distinctive feature of this kind of proof that it is *goal-directed* in the sense that the nonterminal symbols introduced in the derived sentential form represent further subgoals for the derivation as a whole.

Example 2.1.3 An example of a reduction in G_0 could be constructed by starting with the terminal string $a+a*a$ and attempting to reduce it to the start symbol of the grammar:

$$\begin{array}{rcll} a+a*a & \Leftarrow & F+a*a & \text{by } 6 \quad F \to a \\ & \Leftarrow & T+a*a & \text{by } 4 \quad T \to F \\ & \Leftarrow & T+F*a & \text{by } 6 \quad F \to a \\ & \Leftarrow & T+T*a & \text{by } 4 \quad T \to F \\ & \Leftarrow & T+T*F & \text{by } 6 \quad F \to a \\ & \Leftarrow & T+T & \text{by } 3 \quad T \to T*F \\ & \Leftarrow & T+E & \text{by } 3 \quad E \to T \\ & \Leftarrow & E & \text{by } 1 \quad E \to T+E \end{array}$$

The reduction of a terminal string w according to a grammar G to the sentence symbol exhibits a *proof* that $w \in \mathcal{L}(G)$. A reduction begins with the most specific sentential forms and proceeds *upwards* towards more general sentential forms, ultimately reaching the sentence symbol of the grammar. Proofs of this kind are usually referred to as *bottom up* proofs. In contrast to top down proofs bottom up proofs are *data directed* in the sense that the availability of specific data provides guidance for the proof process. [Win83, page 91]

In a grammar it is possible to have several derivations that are equivalent, in the sense that all derivations use the same productions at the same places, but in different order.

Example 2.1.4 An example of a second *derivation* in G_0 would be

$$\begin{array}{rcll} E & \Rightarrow & T+E & \text{by } 1 \quad E \to T+E \\ & \Rightarrow & T+T & \text{by } 2 \quad E \to T \\ & \Rightarrow & T+F*T & \text{by } 3 \quad T \to F*T \\ & \Rightarrow & T+F*F & \text{by } 4 \quad T \to F \\ & \Rightarrow & T+F*a & \text{by } 6 \quad F \to a \\ & \Rightarrow & T+a*a & \text{by } 6 \quad F \to a \\ & \Rightarrow & F+a*a & \text{by } 4 \quad T \to F \\ & \Rightarrow & a+a*a & \text{by } 6 \quad F \to a \end{array}$$

The definition of when two derivations are equivalent is a complex matter for unrestricted grammars, but for context-free grammars a convenient graphical representative of an equivalence class of derivations called a *derivation tree* can be defined. [AU72, page 139]

A *derivation tree*, or *parse tree*, for a context-free grammar $G = (N, \Sigma, P, S)$ is a labeled ordered tree in which each node is labeled by a symbol from $N \cup \Sigma \cup \{\langle\rangle\}$. If an interior node

is labeled A and its direct descendants are labeled $X_1, X_2, \ldots, X_n$, then $A \to X_1, X_2, \ldots, X_n$ is a production in P.

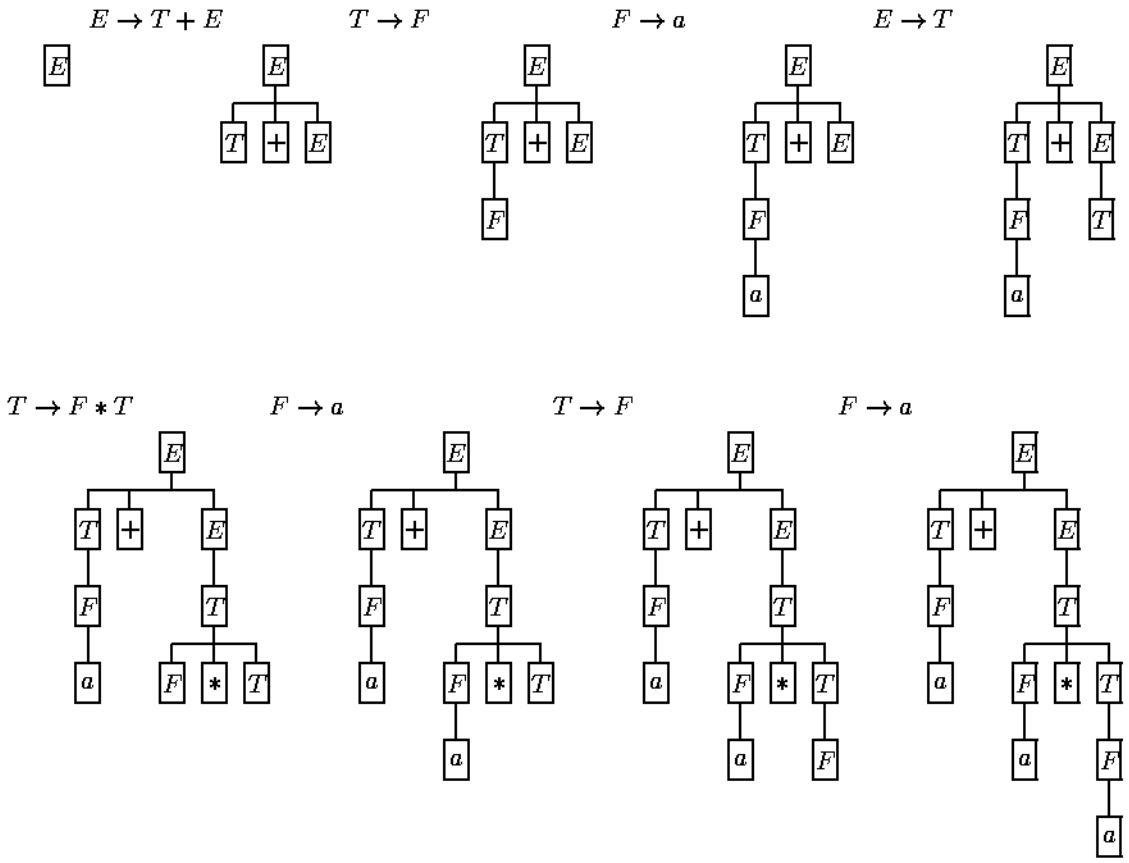


Figure 2.1: Parse Tree for Left to Right Derivation

Definition 2.12 A labeled ordered tree D is a *derivation tree* or (*parse tree*) for a context-free grammar $G(A) = (N, \Sigma, P, A)$ if

1. The root of D is labeled A.

2. If $D_1, ..., D_k$ are the subtrees of the direct descendants of the root and the root of D_i is labeled X_i, then $A \to X_1, X_2, ..., X_n$ is a production in P. D_i must be a derivation tree for $G(X_i) = (N, \Sigma, P, X_i)$ if X_i is a nonterminal symbol, and D_i is a single node labeled X_i if X_i is a terminal symbol.

3. Alternatively, if D_i is the only subtree of the root of D and the root of D_1 is labeled $\langle\rangle$, then $A \to \langle\rangle$ is a production in P.

Example 2.1.5 Let us illustrate how to draw a syntax tree for the derivation of the sentence $a + a * a$ of the grammar G_0 of **Example 2.1.1**. To begin with, the distinguished symbol E of the grammar is designated as the root of the syntax tree. To indicate the first derivation a branch is drawn downward from it. A *branch* is the set of lines together with the *nodes* (the symbols) below the line. Reading from left to right, the nodes of the newly introduced branch form the string corresponding to the right hand side of the production used in the derivation step ($E \to T + E$).

To indicate the second derivation step a branch is drawn downward form the *end node* of the syntax tree labeled T representing the application of the rule $T \to F$. The *end nodes* of a syntax tree are those nodes which have no branches emanating downward from them. Continuing in this manner yields the syntax diagrams shown in Figure 2.1

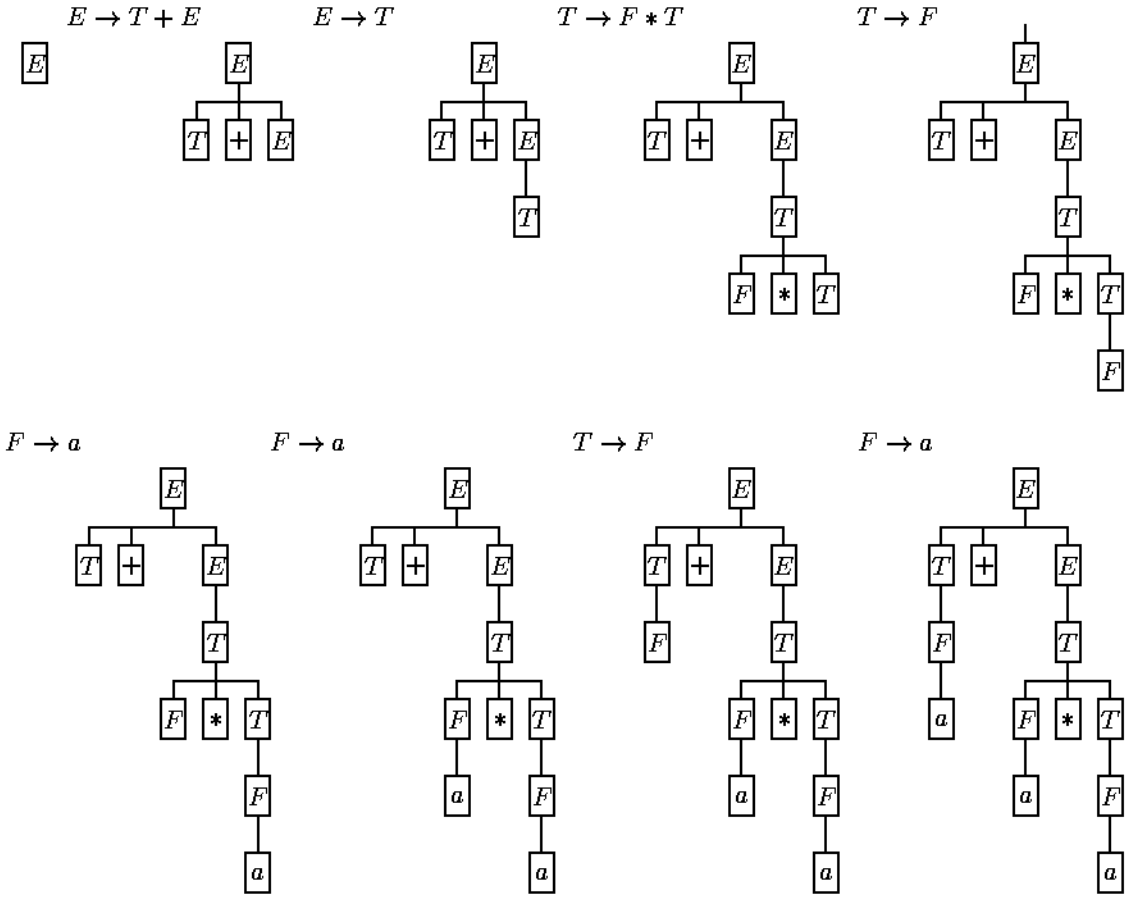


Figure 2.2: Parse Tree for Right to Left Derivation

By drawing syntax diagrams as above the syntactic structure of an input sentence can

be determined from the sequences of productions used to derive that string.

It is most instructive to draw the syntax diagram for the alternative derivation of the sentence $a + a * a$ of $L(G_0)$ to illustrate the notion of equivalence of derivations, shown in Figure 2.2.

Both derivations lead to the construction of the same syntax tree, indicating the equivalence of the two derivations.

The first derivation is an example of a *leftmost* derivation, i.e. the strategy of attempting to expand the leftmost nonterminal in any given sentential from. The second derivation illustrates the strategy of attempting to expand the rightmost nonterminal in any given sentential form, which for this reason is known as a *rightmost* derivation. In general the equivalence of all derivations of a given sentence of a context-free grammar is not guaranteed. Nor indeed is the uniqueness of leftmost or rightmost derivations.

Definition 2.13 A context-free grammar is said to be *ambiguous* if there is at least one sentence w in $\mathcal{L}(G)$ with two or more distinct leftmost (or rightmost) derivations. [AU72, I. p. 143]

2.1.3 Syntax Structures

Definition 2.14 If two grammars generate the same language, the grammars are said to be equivalent.

Example 2.1.6 shows a grammar that generates the same language as the grammar in 2.1.1. The two grammars differ only two rules 1 and 3. Considering rule 1 it is apparent, that in the second grammar, the left-most nonterminal of the rule is the same as the nonterminal on the left-hand side of the arrow. Such a rule, is called *left recursive*. The parse-tree for the derivation of the sentence $a + a * a$ according to the second grammar is shown in Figure 2.3. The parse tree shown in Figure 2.2 can be seen to 'lean' towards the right. Whereas the parse-tree corresponding to the derivation of the same sentence that uses the grammar with the left recursive rules, can be seen to lean towards the left. The structural differences between the syntax structures that these equivalent grammars impose on the sentences of the languages that they define become significant, when interpretations are assigned to them.

Consider the following expression:

$$a - a - a$$

By convention, $a - a - a$ is equivalent to $(a - a) - a = -a$. When an operand has the same operators to its left and right, conventions are needed for deciding which operator takes that operand.

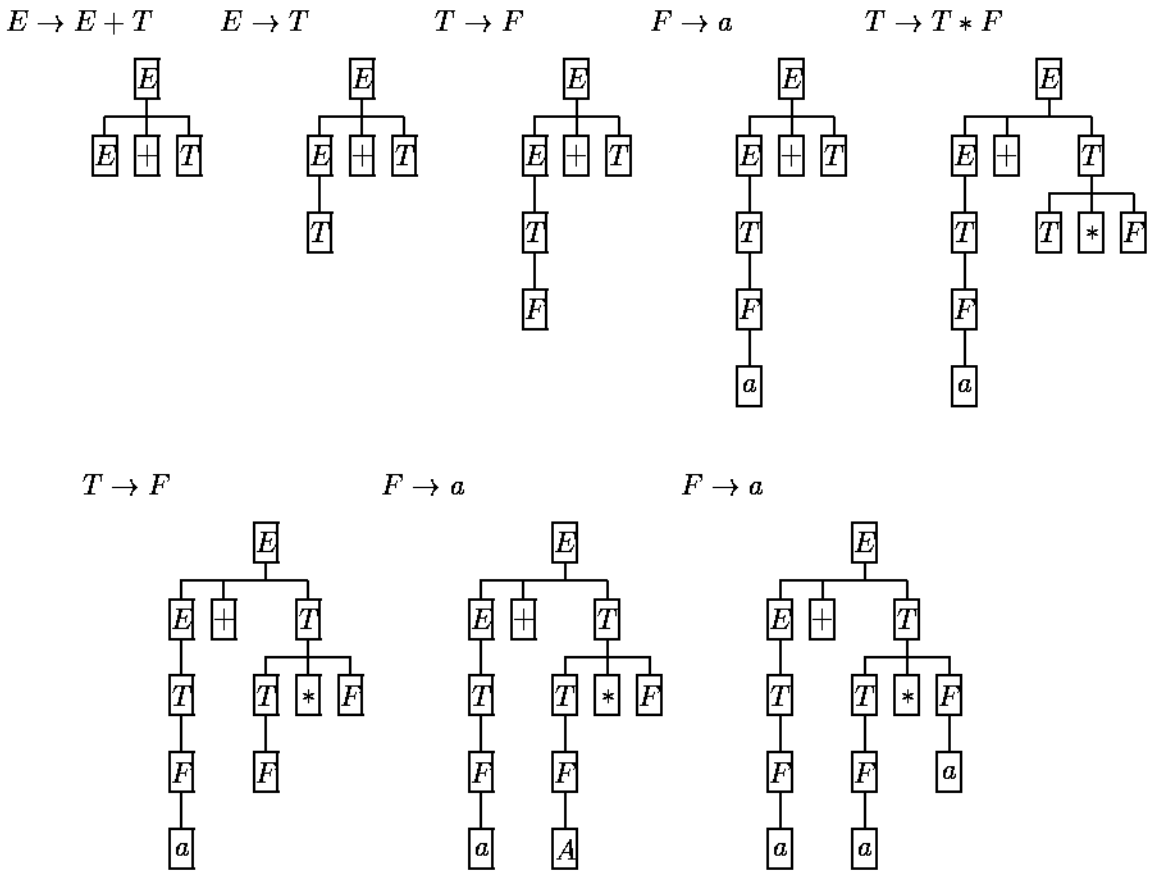


Figure 2.3: Parse Tree for Left Recursive Grammar

Example 2.1.6 An example of an equivalent grammar $G_1 = (\{E, T, F\}, \{a, +, *, (,)\}, P, E)$ where P consists of

$$
\begin{array}{lcll}
E & \rightarrow & E + T & 1 \\
E & \rightarrow & T & 2 \\
T & \rightarrow & T * F & 3 \\
T & \rightarrow & F & 4 \\
F & \rightarrow & (E) & 5 \\
F & \rightarrow & a & 6
\end{array}
$$

$\mathcal{L}(G_1)$, the language defined by the grammar G_0, is the set of arithmetic expressions that can be built up using the symbols $a, +, *, (, and)$.

Definition 2.15 An operator is said to be *left associative* if an operand which has the operator in question on both sides of it is taken by the operator on the left.

For example, the arithmetic operator for subtraction is left associative, whereas exponentiation is right-associative. That is to say, the expression 2^4^3 is treated as (2 ^ (4 ^ 3)).

Left-associative operators are generated by left associative grammars, like G_1. Right-associative operators are generated by right-recursive grammars.

2.2 Parsing

The process used to determine if a string can be generated by a given grammar is called *parsing*. Most parsing methods are either *top-down* or *bottom up*. In top down parsing derivation steps are used in going from the start symbol towards the terminal symbols. In bottom-up parsing reduction steps are used in going from terminal symbols towards the start symbol of the grammar. Parsing methods can also be classified as *deterministic* or *predictive*, in which case choices made in selecting derivation or reduction steps can be guaranteed to be right. Parsing which involves choices between rules that cannot be determined unambiguously, are called non-deterministic. When there are choices, it may be that a wrong route is taken, in which case *backtracking* occurs, to explore other possible choices. Deterministic parsing is to be preferred to non-deterministic methods. There are several methods that can be used to ensure that the parsing can proceed without backtracking. These normally involve reformulations of the grammar. Left-factoring is one of these techniques. It will be introduced next.

2.2.1 Left-Factoring

Left factoring is a grammar transformation technique that is used to make a grammar suitable for predictive parsing.[ASU86, 178-9] It involves the identification of common prefixes in alternative productions for a given nonterminal and rewriting the grammar in a way that will factor out this common prefix and thereby defers the otherwise non-deterministic choice of which alternative to expand first.

For example, consider the productions

$$S \rightarrow \textit{if } E \textit{ then } S \textit{ else } S \tag{2.1}$$
$$\mid \textit{if } E \textit{ then } S \tag{2.2}$$

These two production share a common prefix. The right choice between the alternatives cannot be made by examining the first input symbol. Neither is it possible to determine the number of symbols that need to be examined to make the right choice. The right choice can only be made once the common prefix of the two productions, *if E then S* has been successfully expanded. At that point, if the next input token is *else* than the first rule is applicable, otherwise it is the second. In general, if $A \rightarrow \alpha\beta_1$ and $A \rightarrow \alpha\beta_2$ are two productions for A, and the input begins with a sequence of terminal symbols derivable from α, we cannot tell, in advance which rule is applicable. It is possible to rewrite this grammar in such a way that the decision needs only be made after the common prefix has been expanded. At that point the right rule can be selected deterministically. The *left-factored* form of the example is

$$S \rightarrow \textit{if } E \textit{ then } S' \tag{2.3}$$
$$S' \rightarrow \textit{else } S \tag{2.4}$$
$$\mid S \tag{2.5}$$
$$\tag{2.6}$$

or in general it takes the form:

$$A \rightarrow \alpha A' \tag{2.7}$$
$$A' \rightarrow \beta_1 | \beta_2 \tag{2.8}$$

2.2.2 Limited Backtrack Top-Down Parsing

This section introduces a formalism for language definition and syntactic recognition that is tied to a particular *top down, left to right* parsing algorithm with *limited backtracking.* [1]

In appearance, the formalism will be indistinguishable from a Context-Free Grammar. The difference will be in the way alternatives for a given nonterminal will be treated. Backtracking will be limited by making the *order* in which alternatives for a given nonterminal appear matter for the syntactic recognition process that will take place. That is to say, the alternates for each nonterminal will be tried exhaustively until one alternate has been found

[1] The following account is based on chapter 6 of[AU72]

which derives a prefix of the remaining input. Once such an alternate is found, no other alternates will be tried. Of course, the "wrong" prefix may have been found, in which case the algorithm will not backtrack but will fail. Fortunately, this aspect of the algorithm is rarely a serious problem in practical situations, provided the alternates are ordered so that the longest is tried first.

According to the technique introduced here, nonterminals are treated as string-matching procedures. To illustrate this technique suppose that that $a_1 \ldots a_n$ is the input string and that a partial left parse have been successfully generated matching the first $i-1$ input symbols. If nonterminal A is to be expanded next, then the nonterminal A can be "called" as a procedure, with input $w = a_i \ldots a_n$. If A derives a terminal string that is a prefix of $w = a_i a_{i+1} \ldots a_n$ then A is said to *succeed* on input $w = a_i \ldots a_n$. Otherwise, A fails on input $w = a_i \ldots a_n$.

These procedures call themselves recursively. If A was called in this manner, A itself would call the nonterminals of its first alternate, α_1. If α_1 failed, then A would restore the input string to what it was when A was first called, and then A would call α_2, and so forth. If α_j succeeds in matching $w = a_i a_{i+1} \ldots a_k$, then A returns to the procedure that called it and sets the input string to the unmatched portion $w = a_{k+1} \ldots a_n$.

The difference between the current algorithm and a fully backtracking one is that should the latter fail to find a complete parse in which α_j derives $a_i a_{i+1} \ldots a_k$, then it will backtrack and try derivations beginning with productions $A \rightarrow \alpha_{j+1}$, $A \rightarrow \alpha_{j+2}$, and so forth, possibly deriving a different prefix of $\alpha_i \ldots \alpha_n$ from A. This algorithm will not do so. Once it has found that α_j derives a prefix of the input and that the subsequent derivation fails to match the input, the parsing algorithm returns to the procedure that called A, reporting failure. The algorithm will act as if A can derive no prefix whatsoever of $a_i \ldots a_n$. Thus the algorithm may miss some parses and may not even recognise the same language as its underlying Context Free Grammar defines.

Consider the following example.

Example 2.2.1 If

$$\begin{aligned} S &\rightarrow Ac \\ A &\rightarrow a|ab \end{aligned}$$

are productions and the alternates are taken in the order shown, then the limited backtrack algorithm will not recognise the sentence abc. The nonterminal S is called with input abc will call A with abc. Using the first alternate, A reports success and sets the input string to bc. However, c does not match b, so S reports failure on input abc. Since A reported

success the first time it was called, it will not be called to try the second alternate. Note that this difficulty can be avoided by writing

$$A \rightarrow ab|a$$

The "top-down parsing language", TDPL, introduced in this section can be used to describe parsing procedures of this nature. A *statement* (or *rule*) of TDPL is a string of one of the following forms:

$$A \rightarrow BC/D$$

$$A \rightarrow a$$

where A, B, C and D are nonterminal symbols and a is a terminal symbol, the empty string, or a special symbol f (for failure).

Definition 2.16 A TDPL *program* P is a 4-tuple (N, T, R, S), where

1. N and T are finite disjoint sets of *nonterminals* and *terminals.*
2. R is a sequence of TDPL statements such that for each A in N there is at most one statement with A to the left of the arrow, and
3. S is in N is the *start symbol.*

A TDPL program can be described as a set of procedures (the nonterminals) which are called recursively with certain inputs. The outcome of a call will either be **failure**, (no prefix of the input is matched or recognised) or **success** (some prefix of the input is matched).

The following sequence of procedure calls results from a call of a statement of the form $A \rightarrow BC/D$, with input w:

1. First, A calls B with input w. If $w = xx'$ and B matches x, then B reports **success.** A then calls C with input x'.
   (a) If $x' = yz$ and C matches y, then C reports **success.** A then returns **success** and reports that it has matched the prefix xy of w.
   (b) If C does not match any prefix of x', then C reports **failure.** A then calls D with input w. Note that the success of B is undone in this case.
2. If, when A calls B with input w, and B cannot match any prefix of w, then B reports **failure.** A then calls D with input w.

3. If D has been called with input $w = uv$ and D matches u, a prefix of w, then D reports **success**. A then returns **success** and reports that it has matched the prefix u of w.

4. If D has been called with input w and D cannot match any prefix of w, then D reports **failure**. A then reports **failure**.

Note that D gets called unless both B and C succeed. Note also that if both B and C succeed then the alternate D can never be called.

The special statements $A \to a$, $A \to \langle\rangle$, and $A \to f$ are handled as follows:

1. If $A \to a$, is the rule for A with $a \in T$ and A is called on an input string beginning with a, then A succeeds and matches this a. Otherwise, A fails.

2. If $A \to \langle\rangle$ is the rule for A, then A succeeds whenever it is called and always matches the empty string.

3. If $A \to f$ is the rule, A fails whenever it is called.

The notion of a nonterminal "acting on an input string" can be formalised as follows:

Definition 2.17 Let $P = (N, T, R, S)$ be a TDPL program. A set of relations are defined $\overset{n}{\underset{P}{\Rightarrow}}$ from nonterminals to pairs of the form $(x|y, r)$, where x and y are in T^* and r is either s (for **success**) or f (for **failure**). The metasymbol | is used to indicate the position of the current input symbol. The subscript P will be dropped wherever possible.

1. If $A \to \langle\rangle$ is in R, then $A \overset{1}{\Rightarrow} (|w, s)$ for all $w \in T^*$.

2. If $A \to f$ is in R, then $A \overset{1}{\Rightarrow} (|w, f)$ for all $w \in T^*$.

3. If $A \to a$ is in R, with $a \in T$, then

   (a) $A \overset{1}{\Rightarrow} (a|x, s)$ for all $x \in T^*$.

   (b) $A \overset{1}{\Rightarrow} (|y, f)$ for all those $y \in T^*$ (including $\langle\rangle$) which do not begin with the symbol a.

4. Let $A \to BC/D$ be in R.

   (a) $A \overset{m+n+1}{\Rightarrow} (xy|z, s)$ if

      i. $B \overset{m}{\Rightarrow} (x|yz, s)$ and

      ii. $C \overset{n}{\Rightarrow} (y|z, s)$.

(b) $A \overset{i}{\Rightarrow} (u|v, s)$, with $i = m + n + p + 1$, if

i. $B \overset{m}{\Rightarrow} (x|y, s)$ and

ii. $C \overset{n}{\Rightarrow} (|y, f)$, and

iii. $D \overset{p}{\Rightarrow} (u|v, s)$, where $uv = xy$.

(c) $A \overset{i}{\Rightarrow} (|xy, f)$, with $i = m + n + p + 1$, if

i. $B \overset{m}{\Rightarrow} (x|y, s)$ and

ii. $C \overset{n}{\Rightarrow} (|y, f)$, and

iii. $D \overset{p}{\Rightarrow} (|xy, f)$,

(d) $A \overset{m+n+1}{\Rightarrow} (x|y, s)$, if

i. $B \overset{m}{\Rightarrow} (|xy, f)$ and

ii. $D \overset{n}{\Rightarrow} (x|y, s)$.

(e) $A \overset{m+n+n+1}{\Rightarrow} (|x, f)$, if

i. $B \overset{m}{\Rightarrow} (|xy, f)$ and

ii. $D \overset{n}{\Rightarrow} (|x, f)$.

The relations $\overset{n}{\Rightarrow}$ do not hold except when required by (1)-(4).

Case (4a) takes care of the case in which B and C both succeed. In (4b) and (4c), B succeeds, but C fails. In (4d) and (4e), B fails. In the last four cases, D is called and alternately succeeds and fails. Note that the integer above the arrow indicates the number of "calls" which were made before the outcome is reached. Observe also that if $A \overset{n}{\Rightarrow} (x|y, f)$, then $x = \langle\rangle$. That is, failure always resets the input pointer to where it was at the beginning of the call.

Definition 2.18 Let $A \overset{+}{\underset{P}{\Rightarrow}} (x|y, r)$, if and only if $A \overset{n}{\underset{P}{\Rightarrow}} (x|y, r)$ for some $n \geq 1$.

The *language defined by* P, denoted $\mathcal{L}(\mathcal{P})$, is $\{w | S \overset{+}{\underset{P}{\Rightarrow}} (w|, s)$ and $w \in T^*\}$.

Example 2.2.2 Let P be the TDPL program $(\{S, A, B, C\}, \{a, b\}, R, S)$ where R is the sequence of statements

$$
\begin{aligned}
S &\rightarrow AB/C \\
A &\rightarrow a \\
B &\rightarrow CB/A \\
C &\rightarrow b
\end{aligned}
$$

The action of P on the input string aba using the relations defined above is as follows. To begin, since $S \rightarrow AB/C$ is the rule for S, S calls A with input aba. A recognises the first input symbol and returns success. Using part (3) of the previous definition we can write $A \stackrel{1}{\Rightarrow} (a|ba, s)$. Then, S calls B with input ba. Since $B \rightarrow CB/A$ is the rule for B, the behaviour of C on ba will have to be examined. We find that C matches b and returns success. Using (3) we write $C \stackrel{1}{\Rightarrow} (b|a, s)$.

Then B calls itself recursively with input a. However, C fails on a and so $C \stackrel{1}{\Rightarrow} (|a, f)$. B then calls A with input a. Since A matches a, $A \stackrel{1}{\Rightarrow} (a|, s)$. Since A succeeds, the second call of B succeeds. Using rule (4d) we write $B \stackrel{3}{\Rightarrow} (a|, s)$.

Returning to the first call of B on input ba, both C and B have succeeded. Thus, by using rule (4a) we can write $B \stackrel{5}{\Rightarrow} (ba|, s)$.

Now returning to the call of S, both A and B have succeeded. Thus, S matches aba and returns **success**. Using rule (4a) we can write $S \stackrel{7}{\Rightarrow} (aba|, s)$. Thus, aba is in $\mathcal{L}(\mathcal{P})$.

It is not difficult to show that $\mathcal{L}(P) = ab^* + a$.

An important property of a TDPL is that the outcome of any program on a given input is unique. The interested reader can find the proof in [AU72, 461].

2.2.3 Extensions to TDPL

The notation introduced for a TDPL to this point was designed for ease of presentation. In practical situations it is desirable to use more general rules. For this purpose, *extended* TDPL rules will be introduced. Their meaning will be defined in terms of the basic rules:

Definition 2.19

1. The rule $A \rightarrow BC$ is taken to stand for the pair of rules $A \rightarrow BC/D$ and $D \rightarrow f$, where D is a new symbol.

2. The rule $A \rightarrow B/C$ is taken to stand for the pair of rules $A \rightarrow BD/C$ and $D \rightarrow \langle\rangle$.

3. The rule $A \rightarrow B$ is taken to stand for the rules $A \rightarrow BC$ and $C \rightarrow \langle\rangle$.

4. The rule $A \rightarrow A_1A_2...A_n$, for $n > 2$, is taken to stand for the set of rules $A \rightarrow A_1B_1$, $B_1 \rightarrow A_2B_2, \ldots, B_{n-3} \rightarrow A_{n-2}B_{n-2}$,

5. The rule $A \rightarrow \alpha_1/\alpha_2/\ldots/\alpha_n$, where α_is are strings of nonterminals, is taken to stand for the set of rules $A \rightarrow B_1/C_1, C_1 \rightarrow B_2/C_2, \ldots, C_{n-3} \rightarrow B_{n-2}/C_{n-2}, C_{n-2} \rightarrow B_{n-1}/B_n$, and $B_1 \rightarrow \alpha_1, B_2 \rightarrow \alpha_2, \ldots B_n \rightarrow \alpha_n$. If n=2, these rules reduce to $A \rightarrow B_1/B_2$, $B_1 \rightarrow \alpha_1$, $B_2 \rightarrow \alpha_2$. For $1 \leq i \leq n$ if $|\alpha_i| = 1$, B_i can be let as α_i and the rule $B_i \rightarrow \alpha_i$ can be eliminated .

6. The rule $A \rightarrow \alpha_1/\alpha_2/\ldots/\alpha_n$, where the α's are strings of nonterminals and terminals, is taken to stand for the set of rules, $A \rightarrow \alpha\prime_1/\alpha'_2/\ldots/\alpha'_n$, and $X_a \rightarrow a$ for each terminal a, where α'_i is α_i with each terminal a replaced by X_a

Henceforth extended rules of this type will be allowed in TDPL programs. The definitions above provide a mechanical way of constructing an equivalent TDPL program that meets the original definition.

These extended rules have natural meanings. For example, if A has the rule $A \rightarrow Y_1Y_2...Y_n$, then A succeeds of and only if Y_1 succeeds at the input position where A is called, Y_2 succeeds where Y_1 left off, Y_3 succeeds where Y_2 left off, and so forth.

Likewise, If A has the rule $A \rightarrow \alpha_1/\alpha_2/\ldots/\alpha_n,$, then A succeeds if and only if α_1 succeeds where A is called, or if α_1 fails, and α_2 succeeds where A is called, and so forth.

2.2.3.1 Left Recursion

TDPL can also be extended to handle left recursive rules: Let $A \rightarrow B/AC$, be in R.

(7) (a) i. If $B\overset{m}{\Rightarrow}(x|y,s)$,

ii. $C\overset{n_i}{\Rightarrow}(u_i|v_i,s)$, for some $1 \leq i \leq k$, where $u_1v_1 = y$ and $u_iv_i = v_{i-1}$ and

iii. $C\overset{p}{\Rightarrow}(|v_k,f)$,

then $A\overset{r}{\Rightarrow}(xu_1\cdots u_k|v_k,s)$, with $r = m + n_1...n_k + p + 1$,

(b) If

i. $B\overset{m}{\Rightarrow}(x|y,s)$,

ii. $C\overset{p}{\Rightarrow}(|y,f)$,

then $A\overset{m+p+1}{\Rightarrow}(x|y,s)$.

(c) If

i. $B\overset{m}{\Rightarrow}(|u,f)$,

then $A\overset{m+1}{\Rightarrow}(|u,f)$.

In case (a) B succeeds and then C succeeds k times successively before failing A then returns successfully.

In (4b) and (4c), B succeeds, but C fails. In (4d) and (4e), B fails. In the last four cases, B is called and alternately succeeds and fails. Note that the integer above the arrow indicates the number of "calls" which were made before the outcome is reached.

To allow extended forms of left recursive rules add to the extension rules the following:

(8) The rule $A \rightarrow B/A\alpha_1/\cdots/A\alpha_n$ for $n >= 2$, where α_is are strings of terminals and or nonterminals, is taken to stand for the set of rules $A \rightarrow AC/D$ and $C \rightarrow \alpha_1/\cdots/\alpha_n$.

With this extension the left recursive verison of grammar G_0 for arithmetic expressions on page 21 can be interpreted as an extended TDPL program. Note that other rules for extensions may be needed to translate the definition for C in the above rule for extension.

2.3 Syntax-Directed Translation

This section introduces the concepts and terminology of Syntax-Directed Translations.

Translations will be first explored from an abstract point of view and then extensions to the basic definitional framework to enhance its expressive power will then be considered.

Definition 2.20 A translation is a set of pairs (x, y) of finite-length strings, where x is a string over some finite *input alphabet* and y is a string over some finite *output alphabet.* The strings x and y are called *input string* and *output string*, respectively. y is said to be the *translation* of x. The set of all strings x, for which there is a translation y, is called the *input language.* Analogously, the set of all strings y, for which there is a corresponding input string x, is called the *output language.*

A Translation Scheme is a grammar with a mechanism for producing an output for each sentence generated. A transducer is a recogniser which can emit a finite-length string of output symbols on each move.

2.3.1 Translation and Semantics

It has been pointed out that the formal notion of a language introduced earlier is devoid of any concept of meaning. With the introduction of the notion of translation this restriction can be removed. One way of conceiving of semantics, or meaning, of a language is to associate with each sentence of the language another string which is to be taken to describe the meaning of the original sentence. This way of construing the task of defining the semantics of a formal language can be made to work if, in a given context, we can regard the output language of the translation to be semantically primitive. Such specification of the meaning of a language is known as *translational semantics.*

2.3.2 Syntax-Directed Translation Schemata

The problem of finitely specifying an infinite translation is similar to the problem of specifying an infinite language. A device which given an input string x, calculates an output string y such that (x, y) is in a given translation T, is called a *translator* for T. There are several features which are desirable in the definition of a translation:

1. The definition of the translation should be readable. That is to say, it should be easy to determine what pairs are in the translation.

2. It should be possible to mechanically construct an efficient translator for that translation directly from the definition.

Features which are desirable in translators are

1. Efficient operation. For an input string w of length n, the amount of time required to process w should be linearly proportional to n.

2. Small size.

3. Correctness. It would be desirable to have a small finite test such that if the translator passed this test, this would be a guarantee that the translator works correctly on all inputs.

One formalism for defining translations is the syntax-directed translation schema. [AU72, I pp. 215-216] Intuitively, a syntax-directed translation schema is simply a grammar in which translation elements are attached to each production. Whenever a production is used in the derivation of an input-sentence, the translation element is used to help compute a portion of the output sentence associated with the portion of the input sentence generated by that production.

Example 2.3.1 Consider the following translation schema which defines the translation $\{(x, x^R)|x \in \{0,1\}^*\}$. That is, for each input x, the output is x reversed. The rules defining this translation are

Production	Translation Element
(1) $S \rightarrow 0S$	$S = S0$
(2) $S \rightarrow 1S$	$S = S1$
(3) $S \rightarrow \langle\rangle$	$S = \langle\rangle$

An input-output pair in the translation defined by this schema can be obtained by generating a sequence of pairs of strings (α, β) called *translation forms*, where α is an input sentential form and β and output sentential form. The Translation form (S, S) is considered first. The first rule can then be applied to this form. To do so, first S is expanded first using the production $S \rightarrow 0S$. The output sentential form S is then replaced by $S0$ is in accordance with the translation element $S = S0$. For the time being, the translation element can be thought of simply as a production $S \rightarrow S0$. The translation form $(0S, S0)$ is thus obtained. S can be expanded in this new translation form by using rule (1) again to obtain $(00S, S00)$. Rule (2) can then be applied, to obtain $(001S, S100)$. Applying rule (3) results in the translation form $(001, 100)$. No further rules can be applied to this translation form and thus $(001, 100)$ is in the translation defined by this translation schema.

A translation schema T defines some translation $\tau(\mathcal{T})$. A translator $\tau(\mathcal{T})$ can be built from the translation schema that works as follows. Given an input string x, the translator finds (if possible) some derivation of x from S using the productions in the translation schema. Suppose that $S = \alpha_0 \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow ...\alpha_n = x$ is such a derivation. Then the translator creates a derivation of translation forms

$$(\alpha_0, \beta_0) \Rightarrow (\alpha_1, \beta_1)... \Rightarrow (\alpha_n, \beta_n)$$

such that $(\alpha_0, \beta_0) = (S, S)$, $(\alpha_n, \beta_n) = (x, y)$, and each β_i is obtained by applying to β_{i-1} the translation element corresponding to the production used in going from α_{i-1} to α_i at the "corresponding" place. The string y is an output for x. Often the output sentential form can be created at the time the input is being parsed (as in META-LISP).

Example 2.3.2 Consider the following translation scheme which maps arithmetic expression of $\mathcal{L}(G_0)$ into fully parenthesised prefix notation:

Production	Translation Element
$E \rightarrow T + E$	$E = (+\ T\ E)$
$E \rightarrow T$	$E = T$
$T \rightarrow F * T$	$T = (*\ F\ T)$
$T \rightarrow F$	$T = F$
$F \rightarrow (E)$	$F = E$
$F \rightarrow a$	$F = a$

The translation element $E = (+\ T\ E)$ is associated with the production $E \rightarrow T+E$. The translation element says that the translation associated with E on the left of the production, is first an opening parenthesis, followed by a plus sign, the translation of T, the translation associated with E on the right of the production, and a closing parenthesis.

The output for the input $a + a * a$ can be determined by finding a leftmost derivation of $a + a * a$ from E using the productions of the translation scheme. Then the corresponding sequence of translation forms is computed as shown:

$$\begin{aligned}
(E, E) &\Rightarrow (T + E, (+\ T\ E)) \\
&\Rightarrow (F + E, (+\ F\ E)) \\
&\Rightarrow (a + E, (+\ a\ E)) \\
&\Rightarrow (a + T, (+\ a\ T)) \\
&\Rightarrow (a + F * T), (+\ a\ (*\ F\ T))) \\
&\Rightarrow (a + a * F), (+\ a\ (*\ a\ F))) \\
&\Rightarrow (a + a * a, (+\ a\ (*\ a\ a)))
\end{aligned}$$

A parse tree showing the translations at each node is called an *annotated* parse tree. The annotated parse tree for the above translation is shown in figure 2.3.2.

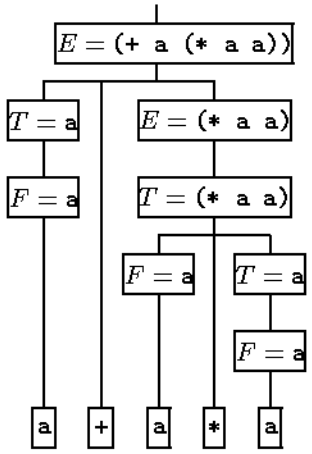


Figure 2.4: Annotated Parse Tree

The translation schemata in Examples 2.3.1 and 2.3.2 are special cases of an important class of translation schemata called *syntax-directed translation schemata.*

Definition 2.21 A *syntax-directed translation schema* (SDTS for short) is a 5-tuple $T = (N, \Sigma, \Delta, R, S)$, where

1. N is a finite set of *nonterminal symbols.*
2. Σ is a finite *input alphabet.*
3. Δ is a finite *output alphabet.*
4. R is a finite set of *rules* of the form $A \to \alpha, \beta$, where $\alpha \in (N \cup \Sigma)^*$, $\beta \in (N \cup \Delta)^*$, and the nonterminals in β are a permutation of the nonterminals in α.
5. S is a distinguished nonterminal in N, the *start symbol.*

Let $A \to \alpha, \beta$ be a rule. To each nonterminal of α there is *associated* an identical nonterminal of β. If a nonterminal B appears only once in α and β, then the association is obvious. If B appears more than once, we use integer superscripts to indicate the association. This association is an intimate part if the rule. For example, in the rule $A \to B^{(1)}CB^{(2)}, B^{(2)}B^{(1)}C$, the three positions in $B^{(1)}CB^{(2)}$ are associated with positions 2, 3, and 1, respectively, in $B^{(2)}B^{(1)}C$.

We define a *translation form* of T as follows:

1. (S, S) is a translation form, and the two S's are said to be *associated.*

2. If $(\alpha A\beta, \alpha' A\beta')$ is a translation form, in which the two explicit instances of A are associated, and if $A \to \gamma, \gamma'$ is a rule in R, then $(\alpha\gamma\beta, \alpha'\gamma'\beta')$ is a translation form. The nonterminals of γ and γ' are associated in the translation form exactly as they are associated in the rule. The nonterminals of α and β are associated with those of α' and β' in the new translation form exactly as in the old. The association will again be indicated by superscripts, when needed, and this association is an essential feature of the form.

If the forms $(\alpha A\beta, \alpha' A\beta')$ and $(\alpha\gamma\beta, \alpha'\gamma'\beta')$, together with their associations, are related as above, then we write $(\alpha A\beta, \alpha' A\beta') \overset{*}{\underset{T}{\Rightarrow}} (\alpha\gamma\beta, \alpha'\gamma'\beta')$. We use $\overset{+}{\underset{T}{\Rightarrow}}$, $\overset{*}{\underset{T}{\Rightarrow}}$, and $\overset{k}{\underset{T}{\Rightarrow}}$ to stand for the transitive closure, reflexive transitive closure, and k-fold product of $\underset{T}{\Rightarrow}$. As is customary we shall drop the subscript T whenever possible.

The *translation defined by* T, denoted $\tau(T)$, is the set of pairs

$$\{(x, y) | (S, S) \overset{*}{\underset{T}{\Rightarrow}} (x, y), x \in \Sigma^* \text{ and } y \in \Delta^*\}$$

This is not very realistic. In terms of expressive power it is rather limited. The translation elements can be generalised to be of arbitrary functions of the translations associated with the nonterminals of the underlying grammar.

2.3.3 Attribute Grammars

An attribute grammar is context-free grammar in which each grammar production $A \to \alpha$ has associated with it a set of *semantic rules* of the form $b := f(c_1, c_2, \ldots, c_k)$ where f is a function and either

1. b is a *synthesised attribute* of A and $c_1, c_2, \ldots, c_k$ are attributes belonging to the grammar symbols of the production, or

2. b is an *inherited attribute* of one of the grammar symbols on the right side of the production, and $c_1, c_2, \ldots, c_k$ are attributes belonging to the grammar symbols of the production

An attribute b is said to *depend* on $c_1, c_2, \ldots, c_k$. The functions in semantic actions cannot have side-effects.

Synthesised attributes are used to pass information from the leaves towards the root. The value of an inherited attribute, on the other hand is defined in term of attributes at the parent and/or siblings of that node.

Chapter 3

Overview of META-LISP

META-LISP is a programming language that combines the functional model of LISP with the syntax-directed model. As in the case of all functional languages, function application is the major computational mechanism. Modern functional languages like HOPE and ML, use *pattern matching* as their parameter-passing mechanism. That is, as the mechanism to select the appropriate statement from the body of the function from a set of statements on the basis of the values of the actual arguments. The main advantage of the resulting *pattern directed invocation* is that it combines a limited form of testing of the appropriateness of the arguments of a function, with the selection of their components, it also selects the transformation appropriate for the arguments.

META-LISP, in contrast, uses *syntax-directed invocation.* A META-LISP function takes a single argument, a list. The set of valid input lists is defined by a grammar. Each rule of this grammar has semantic actions associated with it. These rules are not only used to specify the set of valid inputs to a given function, but also to prescribe arbitrarily complex transformations of components of the input. These components, therefore, can be "preprocessed" by the syntax-directed translation that is used to accept them, even before they are passed to the appropriate semantic action for further transformation. This "preprocessing" of the components of the input is responsible for the main methodological advantage of META-LISP: its support for *data abstraction* and *level wise* programming. The case studies that follow this chapter will elaborate this point fully. Before this could be done it is necessary to develop an intuitive understanding of META-LISP as a programming language.

The aim of this Chapter is then to introduce all the constructs of the language on an informal basis. It assumes familiarity with the concept of syntax-directed translation and the limited backtrack top-down parsing language (TDPL) introduced in Chapter 2. The

Chapter is organised as follows. META-LISP is first introduced by two small examples, in Section 1. These examples illustrate most features of the language. In particular the second example conveys some of the flavour of language oriented programming in META-LISP. Section 2 introduces the translation formalism of META-LISP. Section 3 deals with the Semantic Language. The Chapter closes with a discussion of some of the design issues of META-LISP.

3.1 Introductory Examples

The first example introduced in this Section is a reformulation, in META-LISP of the simple translator presented in Example 2.3.2 on page 33 in the previous Chapter. It serves to illustrate the basic features and workings of the core of META-LISP as a *syntax-directed translation schema.* The second example defines a function to calculate the symbolic derivative of arithmetic expressions involving addition and multiplication. It illustrates some of the more advanced linguistic features of META-LISP. More importantly, it illustrates the support that META-LISP provides for the methodology of *representation independent* or *level-wise* programming.

3.1.1 Simple Translation

Example 2.3.2 in Chapter 2 presented the definition of a translation from infix to fully parenthesised prefix notation of arithmetic expressions involving the operations of addition and multiplication. The specification of this translation in META-LISP is presented in Figure 3.1 alongside the original definition.

The notational differences can be easily identified:

- '$\rightarrow$' is replaced by ':'
- the nonterminal on the left of the arrow is written only once, thus allowing the rules for a given non-terminal to be treated as a single unit.
- the semantic actions which correspond to the translation elements are marked by '=' without repeating the name of the nonterminal
- if semantic actions are not given, as in rules 2, 4 and 6, the default semantic action is to return the translation of the last successfully expanded component
- the presence of a pair of parentheses, (), in the input are indicated by matching square brackets [ ]

Figure 3.1: SDT Schema in META-LISP

Production	Translation Element		META-LISP
			`E`
$E \rightarrow T + E$	$E = (+\ T\ E)$	1	`: T + E = [+ T E]`
$E \rightarrow T$	$E = T$	2	`: T`
			`T`
$T \rightarrow F * T$	$T = (*\ F\ T)$	3	`: F * T = [* F T]`
$T \rightarrow F$	$T = F$	4	`: F`
			`F`
$F \rightarrow (E)$	$F = E$	5	`: [E] = E`
$F \rightarrow a$	$F = a$	6	`: a`
			Elementary Definitions
			`+`
			`: '+`
			`*`
			`: '*`
			`a`
			`: 'a`

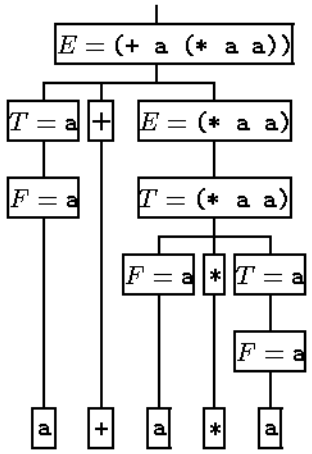
- the introduction of a pair of matching parentheses in the translation element is replaced by a pair of square brackets, indicating the formation of a list of given components
- There are three further rules in the META-LISP formulation of this example. These correspond to the convention of distinguishing between non-terminal and terminal symbols in the original translation scheme. The symbols `+`, `*` and `a` in rules 2, 3 and 6 are not terminal symbols. They are defined by *elementary definition*, which state that they should match terminal symbols.

The translation schema on the left defines a translation independent of any parsing algorithm, and indeed it allows for a complete separation of parsing and the computation of the appropriate translation forms. In contrast, the translation formalism of META-LISP is tied

to a particular parsing algorithm and the translations are computed at the time the input is being parsed. This is achieved by treating the nonterminals of the underlying grammar as translation procedures. As a translation procedure is called with some input, it tries to expand each of its rules in turn, until one rule succeeds in finding a translation of some prefix of the input. The expansion of a rule involves calls to translation procedures corresponding to the nonterminals of the rule with input that was left behind by the preceding successful expansions. Once each component of a rule is expanded successfully, the semantic actions is evaluated to compute the appropriate translation of the successfully parsed prefix of the input. This, otherwise simple, picture is complicated somewhat by the use of *left-factoring* (see Section 2.2.1). As an illustration of the translation process of META-LISP consider the trace of the translation of the input `(a + a * a)` shown in Figure 3.3 In the trace '`>`' indicates the call of a translation procedure with input following the '`:`'; '`<`' marks the return of a call, where the '`:`' is followed by the matched prefix, and the '`=`' sign is followed by the translation produced.

The translation is constructed as the left-most derivation of the input is found. Figure 3.2 shows the parse tree for the input `(a + a * a)` annotated with the translations produced. This example have illustrated the use of META-LISP as a translator writing tool. The next section will illustrate its real power as a general purpose programming language.

Figure 3.2: Decorated Parse Tree



`0> E : (a + a * a)`	E is called with input `(a + a * a)`
`1> T : (a + a * a)`	E calls T, the first component of its first rule
`2> F : (a + a * a)`	T calls F, the first component of its first rule
`3> a : (a + a * a)`	The first rule of F requires the first element of the input to be a list. This is not the case, hence F calls on the single component of its second rule a
`<3 a : a = a`	a matches the first element of the input and returns `a`
`<2 F : a = a`	F returns successfully with `a`
`2> * : (+ a * a)`	T calls $*$, the second component of its first rule
`<2 * :  = fail!`	the call to $*$ fails as `*` does not match `+`
`<1 T : a = a`	the first rule of T fails, but as the single component of the second rule, F, to be considered next, is the same as the first component of the first rule for T, T returns `a`, the outcome of the call to F
`1> + : (+ a * a)`	E now calls the second component of its first rule $+$
`<1 + : + = +`	$+$ matches the first element of the input `+` and returns it
`1> E : (a * a)`	E calls itself recursively, as the third component of its first rule with the remaining input `a * a`
`2> T : (a * a)`	as before E calls T
`3> F : (a * a)`	and T calls F
`4> a : (a * a)`	and F expands its second rule by calling a
`<4 a : a = a`	a matches the first element of the input and returns it
`<3 F : a = a`	F returns successfully
`3> * : (* a)`	T calls $*$, the second component of its first rule
`<3 * : * = *`	$*$ succeeds this time
`3> T : (a)`	T calls itself recursively, as the third component of its first rule with the remaining input `(a)`
`4> F : (a)`	as before, T calls F
`5> a : (a)`	and F calls a
`<5 a : a = a`	a matches the first element of the input and returns it
`<4 F : a = a`	F returns successfully
`4> * : ()`	T calls $*$, the second component of its first rule
`<4 * :  = fail!`	$*$ fails
`<3 T : a = a`	as before, T returns the outcome of F
`<2 T : a * a = (* a a)`	following the successful expansion of the first rule of T the associated semantic actions `[* F T]` is evaluated to return a list of the translations of $*$, F and T
`2> + : ()`	E now calls $+$ the second component of its first rule
`<2 + :  = fail!`	$+$ now fails
`<1 E : a * a = (* a a)`	with left-factoring the recursive call to E returns with T
`<0 E : a + a * a = (+ a (* a a))`	the expansion of the first rule of E now succeeds, having matched the input `(a + a * a)`, the semantic action `[+ T E]` is then evaluated to return a list of the translations of $+$, T and E.

Figure 3.3: The Trace of a Simple Translation

3.1.2 Symbolic Differentiation

The problem of calculating the symbolic derivative of algebraic polynomials is one of the oldest and most widely used example of symbolic manipulation in the literature. [1] John McCarthy, the inventor of LISP, used this problem for one of his examples in [McC60]. In the *LISP 1.5 Primer* by Clark Weissman, [Wei67], a complete program is presented for the differentiation of algebraic polynomial, including input and output. John Allen in the *Anatomy of LISP*, [All78], used this problem as the vehicle of teaching the importance of adopting a *representation independent* style of programming. In the *Structure and Interpretation of Computer Programs*, [ASS85], the example of symbolic differentiation is used to illustrate the idea of *data abstraction*. Programs for symbolic differentiation can be found in the literature on Prolog and ML. The motivation for using this problem, both as an introductory example and as one of the main case studies in this dissertation, is twofold. The first is to allow direct comparison with solutions offered in other languages (LISP, Prolog and ML). Secondly, since it is a problem that exhibits many of the common features characteristic of symbolic computation, its discussion can bring to sharper focus many important methodological issues. This section is devoted to the discussion of methodological issues while constructing a program in META-LISP for calculating the symbolic derivative of arithmetic expression involving the operations of addition and multiplication.

In this section only the following rules of differentiation will be considered:

$$\frac{d}{dx}c = 0 \tag{3.1}$$

$$\frac{d}{dx}u = \begin{cases} 1; & if \quad u = x \\ 0; & else \end{cases} \tag{3.2}$$

$$\frac{d}{dx}(u+v) = \frac{d}{dx}u + \frac{d}{dx}v \tag{3.3}$$

$$\frac{d}{dx}(uv) = v\frac{d}{dx}u + u\frac{d}{dx}v \tag{3.4}$$

$$\tag{3.5}$$

The first rule says that the derivative of a constant is zero; the second rule applies for variables, the third gives the rule for sums and the fourth for products. Given that the program for this problem will be called *deriv*, the following META-LISP grammar rules can be used to define the set of valid input:

[1] the problem of proving theorems in propositional calculus enjoys a similar status

```
deriv
 : Const x
 : Var   x
 : Sum   x
 : Prod  x
```

In the above grammar the categories *Const, Var, Sum and Prod* are intended to define the class of constants, variables, sums and products, respectively. For the first two rules, it is straightforward to formulate semantic actions appropriate for the task. The derivative of a constant is just zero. The first rule is then simply:

```
 : Const x = 0
```

Note, that the role of *Const* here is simply to *recognise* the appropriate component of the input, so that the right semantic action for dealing with it can be selected. Note also, that the value of the semantic action associated with the this rule does not depend on the actual value of the constant in the input.

In the case of differentiating a variable, however, it is not sufficient to recognise the presence of a variable as the first component of the input, but it is also necessary to compare it with the differentiation variable to determine the value of *deriv*. This requires the actual components of the input to be passed to the semantic action. Thus *Var* and x in this case serve not only the purpose of recognition, but also the purpose of *selection* of the appropriate elements of the input. The derivative of a variable is 1, if it is the differentiation variable, otherwise it is zero. Assuming that `Var` refers to a variable recognised in the input, and `x` refers to the variable of differentiation, the appropriate semantic action for the second rule could be written using LISP-like syntax as `(if (same Var x) 1 0)`, giving the rule:

```
 : Var   x = (if (same Var x) 1 0)
```

Every function in a semantic action is required to be be defined in terms of further META-LISP definitions. *if* is a primitive of META-LISP, with the usual meaning. *same* can be defined in terms of the built in LISP function `eq`. The pseudo rule of the form: `: with lisp` ⟨*function*⟩, is used to designate a LISP function to be used as a semantic function. The definition of *same* can be given as:

```
same
 : with lisp eq
```

Unlike variables and constants, both sums and products have internal structure, viz. two operands u and v. Hence, *Sum* and *Prod* will be required not only to *analyse* the input, but to *extract* these operands from it. In META-LISP this can be achieved by the use of *synthesised attributes*. (see page 57) Thus, for example, *Sum* will have two synthesised attributes,

denoted as `u@Sum` and `v@Sum`, representing the addend and the augend of the sum. In terms of these attributes the appropriate semantic action to express the rule of differentiation for addition can be formulated as: `(make-Sum (deriv u@Sum x) (deriv v@Sum x))`, where `make-Sum` is an abstract *constructor* of elements of the domain *Sum*, and `deriv` is a recursive call to the program that is being defined. Again it looks like a LISP function call, but in fact it is an example of a *syntax-directed invocation*. E.g., the first recursive call to *deriv* will take a single argument, a list comprising the extracted components named *u@Sum* and *x*. Then it will be the responsibility of *deriv* (again) to determine whether the first component is a constant, a variable, or again a sum or a product, etc. Note, that without considering the possibility of elaborating the semantic functions themselves in a language oriented way, *deriv* could not even be defined! Clearly, there is not much point in defining the input with a grammar, just to find that it is then required to define deriv in C, LISP or Prolog as in a conventional translator-writing systems.

The handling of products is analogous to the way sums are treated. This gives, as the first level elaboration of the symbolic differentiation program, the following META-LISP definition:

```
deriv
: Const x = 0
: Var   x = (if (same Var x) 1 0)
: Sum   x = (make-Sum (deriv u@Sum x) (deriv v@Sum x))
: Prod  x = (make-Sum
              (make-Prod (deriv u@Prod x) v@Prod)
              (make-Prod u@Prod (deriv v@Prod x)))
```

Points to note about this definition:

- it is *abstract* in the sense that it makes no commitment for the actual representation (in terms of list structures) of the domains of interest.
- all the complexity of relating the abstract properties of the input (that a *Sum* has two components) to concrete representation (whether it be prefix, infix or postfix or anything else) are hidden in the appropriate definitions for all the subfunctions being introduced as components in the grammar rules.
- all the complexity of constructing appropriate representation for elements of abstract domains are hidden in the form of *domain constructors* in the semantic actions, such as *make-sum* and *make-prod* which construct elements of the domain of sums and products, respectively.
- the algorithm (in this case for calculating the derivative) can thus be formulated by "passing off" the job of *recognition*, *selection* and *construction* of appropriate elements

> of the domain of interest to sub-functions. No change in the concrete representation of the data will require any change in the above formulation of the algorithm. This is the essence of *level-wise programming*. [All78, 55]

The general technique of isolating parts of a program that deal with how data objects are represented from those parts that deal with how they are used is called *data abstraction*.[ASS85, 72] It is a powerful design methodology much appreciated and practiced within the LISP tradition. It is instructive to compare the above definition of *deriv* with its definition in Scheme [2] taken from [ASS85, 106]:

```
(define (deriv exp var)
  (cond ((constant? exp) 0)
        ((variable? exp)
         (if (same-variable? exp var) 1 0))
        ((sum? exp)
         (make-sum (deriv (addend exp) var)
                   (deriv (augend exp) var)))
        ((product? exp)
         (make-sum
           (make-product (multiplier exp)
                         (deriv (multiplicand exp) var))
           (make-product (deriv (multiplier exp) var)
                         (multiplicand exp))))))
```

Writing representation-independent programs in LISP is clearly a matter of style and discipline. Note also, that in LISP the recognition of instances of abstract data and their selection is the job of separate functions. (e.g., **sum?** will recognise instances of a sum, but its components have to be selected by two further functions **augend** and **addend**). Not only does this enforce a somewhat artificial division, but it can be a source of inefficiency. The separation of recognition and selection can lead to inefficiency as it necessitates the examination of the same data *twice*: once for the purpose of recognising a given instance; and again to select a desired component of it. META-LISP, in contrast, makes it possible to combine both tasks in one functional unit, called an *abstract analyser*. The explicit support that META-LISP gives for describing the abstract properties of the input means that it supports *data abstraction* and *level-wise programming* explicitly.

To carry on with the example, all the subfunctions introduced in the first-level definition need to be defined.

Constants are numbers. LISP provides a built in function to recognise numbers. The condition that the first element of the input should be a number, as recognised by the built-in function **numberp**, can be expressed in META-LISP using the *pseudo rule* of the form: **:** **is** ⟨*predicate*⟩. With this the definition for *Const* becomes:

[2] Scheme is a modern dialect of LISP. [GLSS75] It was invented by Guy Lewis Steele Jr. and Gerald Jay Sussman of the MIT Artificial Intelligence Laboratory.

```
Const
 : is numberp
```

Variables are single lowercase character symbols. The easiest way to define this class is to enumerate them, using the pseudo rule : **any** $\langle object \rangle_1 \ldots \langle object \rangle_n$

```
Var
 : any a b c d e f g h i j k l m n o p q r s t u v w x y z
```

x is a variable, so it can be defined in terms of *Var*:

```
x
 : Var
```

Assuming a fully parenthesised infix notation as a concrete representation of sums, their structure can be described by writing:

```
Sum
 : [augend + addend]
```

Sum is required to make available the two components *addend* and *augend* as synthesised attributes, called u and v. Assignment of synthesised attributes take the following form in META-LISP: (@ ⟨*attribute name*⟩ <- ⟨*term*⟩). Using this form, and noting that a sequence of *semantic terms* can be given in a semantic action, the definition for *Sum* becomes:

```
Sum
 : [augend + addend]  = (@ u <- augend) (@ v <- addend)
```

Similarly for *Product*

```
Prod
 : [multiplicand * multiplier]  = (@ u <- multiplicand) (@ v <- multiplier)
```

make-Sum may use some rudimentary simplifications – such as carrying out addition if both addend and augend are numbers; returning one of them if the other is zero – before constructing an appropriate internal representation for sums, if all else fails.

```
make-Sum
 : a=number b=number = (add a=number b=number)
 : zero     b        = b
 : a        zero     = a
 : a        b        = [a + b]
```

where *add* is defined as

```
add
 : with lisp +
```

and a and b are defined so as to accept the first element of the current input using the rule `: _ .` The rational for such a permissive description of the input to *make-Sum* is that it presumably will have received input that has been produced by other components of the whole program. In some other circumstances it might be necessary to be more restrictive.

What has been shown so far of design of the program should be sufficient to illustrate the following important points about META-LISP:

- it is integrated with LISP, both in the sense of relying on LISP to supply recognisers for primitive domains, such as numbers, as well as for primitive semantic functions, such as `eq`.
- it supports *data abstraction* and *level-wise programming* by making possible the combination of the functions of *recognition*, and *selection* in a single functional unit. This is due to the increase in expressive power that syntax-directed invocation brings.
- enables the design of every non-primitive functional component of a program to be designed in a language oriented style

Above all, this example has emphasised the role of data abstraction. The rule based form of program formulation of META-LISP together with the ability to designate arbitrary complex transformations to pass parameters for further transformations, has important further methodological implications which goes beyond the support it gives to data abstraction. This point will be taken up in the case studies. The following two sections will introduce all the features of the language.

3.2 Translation Formalism

This section describes the translation formalism of META-LISP. The presentation will follow a top-down sequence. First, the concept of a META-LISP *translation procedure* is introduced, and the grammatical means of defining them in terms of other translation procedure and in terms of *elementary definitions* which do not make reference to other translation procedures.

3.2.1 Non-Elementary Rules

3.2.1.1 Alternates

$$
\begin{array}{lllll}
\langle p \rangle & & & & \\
\quad : \langle p \rangle_{1,1} & \langle p \rangle_{1,2} & \cdots & \langle p \rangle_{1,i} & = \langle \textit{semantic action} \rangle_1 \\
\quad : \langle p \rangle_{2,1} & \langle p \rangle_{2,2} & \cdots & \langle p \rangle_{2,j} & = \langle \textit{semantic action} \rangle_2 \\
& & \vdots & & \\
\quad : \langle p \rangle_{n,1} & \langle p \rangle_{n,2} & \cdots & \langle p \rangle_{n,k} & = \langle \textit{semantic action} \rangle_n
\end{array}
$$

The meta-symbol ':' designates the beginning of a grammar rule. The symbol '=' separates the rule from the semantic action. A grammar rule is composed of *constituents* which are nonterminals of the grammar.

The procedural interpretation of the above definition is as follows: $\langle p \rangle$ first attempts to expand its first alternate, i.e. consecutively calls the constituent procedures of the first alternate. If the translation of the first constituent succeeds, meaning that some prefix of the input has been accepted and some value has been produced by procedure $\langle p \rangle_{1,1}$, then $\langle p \rangle$ calls the procedure $\langle p \rangle_{1,2}$ with the remaining input. If the consecutive calls to all the constituent procedures appearing in the first alternate are successful, then procedure $\langle p \rangle$ evaluates the associated semantic action, and returns its value. If there is no semantic action attached to the rule,(i.e., there is no '=' followed by a semantic action), then $\langle p \rangle$ returns the value produced by the last constituent procedure of the rule.

If any of the consecutive calls to the procedures $\langle p \rangle_{1,1}\langle p \rangle_{1,2} \cdots \langle p \rangle_{1,i}$ fails, then the expansion of the first alternate fails. If that happens, procedure $\langle p \rangle$ backtracks by restoring the input to what it was when $\langle p \rangle$ was called, and then attempts to expand its second alternate, and so forth. If the expansion of all the alternates fails then $\langle p \rangle$ returns failure.

The distinctive feature of the translation algorithm outlined above is that the alternates for each nonterminal are tried exhaustively until one translates a prefix of the input. Once such an alternate is found, the procedure returns successfully. However, the algorithm will not backtrack to try the further alternates of the procedure once it has returned successfully,

as in full backtracking algorithms. This aspect of the algorithm dictates that the alternates are to be ordered so that the one that can translate the longest prefix is presented first.

In formulating a definition for a procedure $\langle p \rangle$ it does not concern us how the constituent procedures are defined. At this stage we simply assume that over the range of valid input that they themselves define, they will produce appropriate results.

Multiple occurrences of the same constituent in a rule are allowed. If the semantic action refers to such a component, its value is taken to be the value of the last occurrence.

The procedural interpretation of alternatives given here does not cover the use of left recursive alternates; i.e., when $\langle p \rangle_{i,1} = \langle p \rangle$ for some $1 \leq i \leq n$. For the treatment of left recursive alternates see Section 3.2.1.3.

3.2.1.2 Nested Structures

$$[\ \langle c \rangle_1 \quad \langle c \rangle_2 \quad \cdots \quad \langle c \rangle_n\ ]$$

A nested structure is a sequence of components enclosed by a pair of matching square brackets. In a grammar rule a nested structure can occur in the place of a single constituent procedure, in which case it matches a nested list from the input. The term *component* is for both constituent procedures and nested structures.

The expansion of a nested structure is carried out as follows: a test is made to see whether the next element of the input is indeed a (non-empty) list. If it is, then the components enclosed by the square brackets are consecutively expanded with this list as input. If this expansion is successful, and the entire list has been exhausted in the process, then the list, that has thus been matched, is removed from the input. The expansion of a nested structure fails, if either the first element of the input is not a (non-empty) list, or the expansion of the components in the nested structure fails, or the nested list is not exhausted after the successful expansion of all the enclosed components.

The combination of nested structures and alternates gives the expressive power needed to write list processing applications.

Example 3.2.1 Consider the procedure that tests list membership:

```
member
% Test if an  item is a member of a given list
 : item []          = []
 : item [head tail] = (if (equal item head) t (member item tail))
```

The reference to `member` in the semantic action is a recursive invocation of the translation procedure being defined, with an input list constructed out of the values of the constituent procedures `item` and `tail` which are assumed to select the appropriate parameters from the input.

Note: The line beginning with the character '%' marks out a line of *comment*. Note also that the pair of square brackets in the grammar rule stands for the empty list in the input. In the associated semantic action, on the other hand, the pair of square brackets designate the empty list as a return value.

3.2.1.3 Left Recursion

$$
\begin{array}{l}
\langle p \rangle \\
\quad : \langle p \rangle_{1,1} \quad \langle p \rangle_{1,2} \quad \cdots \quad \langle p \rangle_{1,i} \quad = \quad \langle \textit{semantic action} \rangle_1 \\
\qquad\qquad\qquad\qquad \vdots \\
\quad : \langle p \rangle_{m,1} \quad \langle p \rangle_{m,2} \quad \cdots \quad \langle p \rangle_{m,j} \quad = \quad \langle \textit{semantic action} \rangle_m \\
\quad : \langle p \rangle \quad \langle p \rangle_{m+1,2} \quad \cdots \quad \langle p \rangle_{m+1,j} \quad = \quad \langle \textit{semantic action} \rangle_{m+1} \\
\qquad\qquad\qquad\qquad \vdots \\
\quad : \langle p \rangle \quad \langle p \rangle_{n,2} \quad \cdots \quad \langle p \rangle_{n,k} \quad = \quad \langle \textit{semantic action} \rangle_n
\end{array}
$$

An alternate in a definition of a procedure $\langle p \rangle$ is called *left recursive* if the left-most constituent procedure appearing in the grammar rule of the alternate is the same as the procedure that is being defined. A procedure $\langle p \rangle$ is called left recursive if it has at least one non left recursive alternate followed by one or more left recursive alternates. [3]

The interpretation of a left recursive procedure is as follows: first the non-left recursive alternates are tried in order. If none of them succeeds then the expansion of $\langle p \rangle$ returns failure. If one of them succeeds and produces value v_1 while reducing the input to x_1, then regard the left recursive call to $\langle p \rangle$ in the left recursive alternate(s) as already successfully expanded with v_1 as its value and x_1 as the input that it left unmatched. With this assumption, the expansion of the left recursive alternates does not call $\langle p \rangle$. If the expansion of one of the left recursive alternates, tried in order, is successful, produces value v_2 and reduces the input to x_2, then the expansion of the left recursive alternates is repeated, with the assumption that the call to $\langle p \rangle$ had been successful with value v_2 and x_2 as the unmatched portion of the input. This process is iterated until it leads to failure, in which case $\langle p \rangle$ returns the last successfully produced value, and restores the input to what it was before the last unsuccessful expansion.

Example 3.2.2 The translation of binary numerals into decimal representation of whole numbers is best described using left recursion:

[3] Presenting the left recursive alternatives before the non left recursive ones is also allowed without changing the meaning of the construct as defined below. The important point is not to mix them.

```
binary
 : bit
 : binary bit     = (+ (* 2 binary) bit)
```

The following is a trace of its execution, where **>** indicates the call of a translation procedure with input following the colon; **<** marks the return of a call, where the colon is followed by the matched prefix, and the equal sign is followed by the return value:

```
1> binary : (1 1 0 1 + 1 1 0)
 2> bit : (1 1 0 1 + 1 1 0)
 <2 bit : 1 = 1
<1 binary : 1 = 1
1> binary : (1 0 1 + 1 1 0)
 2> bit : (1 0 1 + 1 1 0)
 <2 bit : 1 = 1
<1 binary : 1 1 = 3
1> binary : (0 1 + 1 1 0)
 2> bit : (0 1 + 1 1 0)
 <2 bit : 0 = 0
<1 binary : 1 1 0 = 6
1> binary : (1 + 1 1 0)
 2> bit : (1)
 <2 bit : 1 = 1
<1 binary : 1 1 0 1 = 13
1> binary : (+ 1 1 0)
 2> bit : (+ 1 1 0)
 <2 bit :   = fail!
<1 binary : 1 1 0 1 = 13
```

`binary` successfully translates the prefix `1 1 0 1` of the input and leaves behind unmatched the input `(+ 1 1 0)`.

3.2.2 Elementary Components

Elementary components specify immediate tests on the input and specific return values without reference to other translation procedures. If the test succeeds than they remove the successfully matched one or more elements from the input. These are then returned as the value of the elementary component. If the test fails then they report failure leaving the input unchanged.

3.2.2.1 Denotation

$$'\langle object \rangle$$
$$\langle string \rangle$$
$$\langle keyword \rangle$$

There are three different ways to specify a test whether the first element of the input is a given object. Firstly, the object to be matched can be designated by the use of a single quote before the object as a component in a grammar rule. Secondly, a LISP *strings* (e.g. `"this is \" a string"`) can be given as a component in a grammar rule to designate a test for the presence of the given string as the first element of the input. Thirdly, a *keyword* (e.g. :test) can be given as a component in a rule to designate a test for the presence of that keyword as the first element of the input.

3.2.2.2 End of Input Test

$

The expansion of an elementary component of this form succeeds if the input is empty, and leaves the empty input unchanged. Otherwise it fails.

3.2.2.3 Prefix

_

or

_⟨*identifier*⟩

An elementary component of this form always succeeds and matches the first element of the input. The second form is really equivalent to having a definition of the form:

```
_s
  : _
```

An elementary component of this form succeeds even if the input is empty. In that case, the value returned is the empty list. If the first element of the input represents failure, then that value will be returned and will cause the calling procedure to fail.

3.2.2.4 Suffix

._

An elementary component of this form always succeeds, matches the entire input and returns it as its value.

3.2.2.5 Empty

<>

An elementary component of this form always succeeds, without matching any of the input, i.e. leaves the input unchanged. The value it returns is the empty list.

3.2.3 Pseudo Rules

Pseudo rules are so called because they look like ordinary grammar rules, but in fact have special interpretations. Pseudo rules have as their first component the symbol `is`, `any` or `with`. If semantic actions are associated with a pseudo rule then if the match was successful, then the successfully matched and removed object from the input can be referenced as the value of the META-LISP keyword appearing in the rule.

3.2.3.1 Enumeration

$$\langle p \rangle \texttt{ : any } \langle object \rangle_1 \ldots \langle object \rangle_n \texttt{ = } \langle semantic\ action \rangle$$

Test: that the first element of the input equals *any* one of the $\langle object \rangle_i$ given in the rule.

Remove: the matched object.

Example:

```
day : any Mon Tue Wed Thu Fri Sat Sun = (print any)
```

3.2.3.2 Predication

$$\langle p \rangle \texttt{ : is } \langle predicate \rangle \texttt{ = } \langle semantic\ action \rangle$$

Test: that the first element of the input satisfies the LISP $\langle predicate \rangle$ named in the rule.

Remove: first element of the input.

Example: An integer is negative, if it is an integer, tested by the Lisp predicate `integerp`, and if it is greater than 0. If the given integer is not greater than zero return failure.

```
negative_int : is integerp = (if (>  0 is) is fail!)
```

3.3 Semantic Actions

Semantic actions are functions of the values (and possibly attributes) of translation procedures appearing in a grammar rule. Semantic actions comprise a non-empty sequence of *semantic terms*. There are four basic mechanisms used to build up semantic actions: construction of list structures, invocation of functions, assignment of attributes and sequencing of semantic terms.

The function invocations have the same outward form as in Lisp, and if the function is a built-in Lisp function, then this Lisp function is called. Otherwise, it signifies the invocation of a META-LISP translation procedure with a single input list. The presence of translation procedures as *functions* in the semantic actions presents a terminological dilemma. Although they are defined as procedures that are expected to consume some portion of their input in the course of producing a translation, in the context of semantic actions this procedural aspect of their behaviour is completely ignored. Their single role is to produce values. So it is more appropriate to adopt the terminology of referring to translation procedure appearing in semantic actions as *semantic functions*. The term *effective concept* is introduced as a term to refer to META-LISP translation procedures regardless of the context in which they appear. Effective concepts are also allowed as parameters. Since semantic actions are invoked only once the expansion of the grammar rule with which they are associated has succeeded, all the immediate constituent procedures of the rule have returned some values. If these immediate constituents occur in the semantic action as parameters, then their values are referenced.

3.3.1 Packages

The unit of modularity in Meta-Lisp is called a *package*. The development of a program always takes place in the context of a package. Packages are introduced by issuing the following META-LISP *directive* at the top-level of META-LISP:

`|` `?=` `package` ⟨*name*⟩

The semantic functions or effective concepts that are invoked in a semantic action are assumed to belong to the same package as the program in which they are called. effective concepts are imported from other packages using a pseudo rule, called a *with clause*.

⟨*p*⟩ `:` `with` ⟨*package*⟩ ⟨*name*⟩

where, ⟨*p*⟩ is the name of the effective concept that is invoked in a semantic action and ⟨*package*⟩ designates the name of the module from which an effective concept with the given ⟨*name*⟩ is to be imported.

3.3.2 List Construction

The facilities for constructing list structures are analogous to the *backquote macro* in Common Lisp.

List construction takes the form of a non-empty sequence of list elements enclosed in a pair of square brackets. A list element can be either a term or a term preceded by a splicing operator, represented by a full stop.

$$\texttt{[}\ \langle e\rangle_1\ \cdots\ \langle e\rangle_n\ \texttt{]}$$

This feature allows the construction of arbitrary complex list structures. The splicing operator is used to embed all elements of the list that it precedes.

Example 3.3.1 *Suppose that* $A =$ (aaa), *and* $B =$ (bbb), *then:*

$$\begin{aligned}
\texttt{[}\ A\ \texttt{.}\ B\ \texttt{]} &= \texttt{((a a a) b b b)}\\
\texttt{[}\ \texttt{.}\ A\ \texttt{.}\ B\ \texttt{]} &= \texttt{(a a a b b b)}\\
\texttt{[}\ A\ \texttt{.}\ B\ C\ \texttt{]} &= \texttt{((a a a) b b b C)}\\
\texttt{[}\ \texttt{.}\ A\ \texttt{]} &= \texttt{(a a a)}
\end{aligned}$$

3.3.3 Invocation

$$(\langle \textit{function term}\rangle\ \langle t\rangle_1 \ldots \langle t\rangle_n)$$

Invocations are evaluated by the following steps:

1. evaluate the $\langle$*function term*$\rangle$, which should evaluate to the name of an effective concept, call it *ec*, in the current package.
2. evaluate the terms $\langle t\rangle_1 \ldots \langle t\rangle_n$ *from left to right*
3. construct an input list, x, formed of the values of these term (e.g. $x = [\langle t\rangle_1 \ldots \langle t\rangle_n]$
4. find the definition for *ec*, (which may involve importing it from another package), and expand this definition with input x.

The following exceptions are raised:

1. If the evaluation of the $\langle$*function term*$\rangle$ fails, then the value of the invocation is *failure.*
2. If the evaluation of the term $\langle$*function term*$\rangle$ succeeds, but it does not evaluate to the name of an effective concept defined in the current package then
   - if it is an identifier then the exception *undefined effective concept* is raised
   - if it is not an identifier the exception *Inappropriate function term* is raised

Note: The application of list construction to the terms given in an invocation is *implicit.* This convention is adopted mainly to reduce the clutter that the introduction of pairs of square brackets into an invocation would present.

3.3.3.1 Dotted Invocation

In the special case when only one term is given, if it has a list value, then it is desirable to override the implicit construction of a list. To indicate this the *dot* notation is used, (exploiting the equivalence: `[ .` $\langle t \rangle$ `]` $= \langle t \rangle$). Thus,

$$(\langle p \rangle \ . \ \langle t \rangle)$$

means that procedure $\langle p \rangle$ is invoked with the value of $\langle t \rangle$ as input.

3.3.3.2 LISP Functions

The elaboration of an effective concept in a given package normally involves further effective concepts, both as immediate constituents in terms of which the input to them is described, as well as those that appear in the semantic actions. Since these effective concepts, in turn, are expected to be elaborated in the same way, at some point procedures will have to be introduced in the semantic actions that require no definitions from the user. Meta-Lisp depends on Lisp for these primitive procedures. The incorporation of primitive functions from Lisp is achieved by using a variant of the with clause, in which `lisp` is designated as the package name:

$$\langle p \rangle \ : \ \texttt{with lisp} \ \langle name \rangle$$

When the semantic function $\langle p \rangle$ is invoked in a semantic action, the named lisp function is *applied* , in the sense of Lisp, to the list that is constructed in the invocation.

Example 3.3.2

```
plus : with lisp +
```

3.3.3.3 Calling META-LISP from LISP

The user interacts with the META-LISP system by 'talking' to the META-LISP *top level* (see page 54) by issuing *directives* (see page 54) and running META-LISP programs. Effective concepts can be invoked from within LISP with a call to the function `?=`

$$\texttt{(?=} \ \langle ec \rangle \ \langle module \rangle \ \langle input \rangle \texttt{)}$$

The arguments to this function should evaluate to the name of an effective concept to be invoked, the name of the module to which it belongs and a list that is passed to the named concept as its input. The module argument is optional.

3.3.4 Attributes

3.3.4.1 Synthesised Attributes

It is sometimes desirable to compute more then one value. Such a facility is provided in the form of *synthesised attributes*. One can think of the value returned by an effective concept as a distinguished *synthesised* attribute, which is referenced by the name of the concept. Contrast this with attribute grammars, which force the user to declare an attribute even when only a single value needs to be computed.

Synthesised attributes for a given effective concept are assigned by the following form of a semantic term appearing in its definition:

$$\texttt{(@}\ \langle \textit{attribute name} \rangle\ \texttt{<-}\ \langle t \rangle \texttt{)}$$

This will create a binding for an identifier made up of the name of the effective concept the attribute name separator `@` and the given name of the attribute, binding it to the value of the semantic term $\langle t \rangle$ (see page 46).

3.3.4.2 Inherited Attributes

The attribute mechanism also allows the specification of *inherited* attributes, permitting information that arises early in the translation process to affect the course of later expansions, in a form well-known from the literature on attribute grammars. See [Knu68, Pag81]

$$\texttt{(\^{}}\ \langle \textit{attribute name} \rangle\ \texttt{<-}\ \langle t \rangle \texttt{)}$$

This will create a binding for the identifier made up of the symbol `^` and the $\langle \textit{attribute name} \rangle$.

The terminology of 'inherited attributes' can be criticised on the grounds that what the above form really defines is simply a local binding for a variable. The following equivalence indeed holds:

$\texttt{(\^{}}\ \langle \textit{name} \rangle\ \texttt{<-}\ t_1\texttt{)}\quad t_2$

is equivalent to

$\texttt{(let ((\^{}}\langle \textit{name} \rangle\ t_1\ \texttt{))}\ t_2\ \texttt{)}$

Their role is similar to the role of "pass variables" in LISA, see [Kos84, 181]. The term inherited attributes is used to highlight their role in affecting subsequent parses.

Both forms of attribute assignements can be given in a shortened form if the attribute name is the same as the name of an effective concept appearing in the rule. In that case the value of the name effective concept is assigned to the identically named attribute. I.e. the following equivalences hold:

$$(@\ \langle name \rangle) \equiv (@\ \langle name \rangle\ \texttt{<-}\ \langle name \rangle)$$

And similarly for inherited attributes:

$$(\hat{}\ \langle name \rangle) \equiv (\hat{}\ \langle name \rangle\ \texttt{<-}\ \langle name \rangle)$$

3.3.5 Conceptual Values

$$\texttt{<}\ \langle identifier \rangle\ \texttt{>}$$

Effective concepts in META-LISP are first class objects. They can be returned as values, and passed as parameters. The piece of special syntax to indicate that a particular identifier is to be interpreted as the name of an effective concept is to enclose the given *identifier* in a pair of pointed brackets.

3.3.6 Semantic Backtracking

In general a semantic action involves calls to other translation procedures, so that the value of the semantic action itself can be failure. If that happens, or equivalently, if the semantic action evaluates to the special constant `fail!`, then the procedure reports failure without considering further alternates. This feature is used to force backtracking at a higher level, or to specify linguistic constraints via negation.

If the programmer's intention is to consider further alternates on the return of a semantic action with failure, this can be achieved by marking out the semantic action with the symbol '`?`' instead of the usual '`=`' sign. The backtracking that is triggered by such an arrangement is called *semantic backtracking*. This provides the ability to impose *context sensitive constraints* on the input, such as, elements of the input are equal, etc. Semantic backtracking can also be used for obtaining *multiple solutions*, as in Prolog, see Chapter 5.

3.4 Discussion

In terms of the number of its features, META-LISP is a programming language of modest size. The syntax of META-LISP, comprising just over 40 non-trivial productions, is shown in Figures 7.1 and 7.2 on pages 140-141. The main charactersitics of the language can be summarised as follows:

- The underlying grammatical formalism is tied to a particular parsing algorithm

- The language of semantic actions is an *applicative language* for which the order of evaluation is fixed (left-to-right inside out).
- Collection of rules for the same effective concept are treated as a single, named unit
- exception handling has not been worked out fully.

Chapter 4

Programming in META-LISP I

The purpose of the case studies presented in this chapter is twofold. First, it contributes to the development of an intuitive understanding of the constructs and idioms of META-LISP through familiar examples. Secondly, these case studies allow direct comparisons with alternative programming styles. Section 1 presents a number of simple examples of list processing in ML as well as META-LISP. Some of these examples were selected from a list of predefined functions in [Wik87]. Section 2 develops a complete program for Symbolic Differentiation in META-LISP. The design of the program follows the design of a program for symbolic differentiation presented in the *LISP 1.5 Primer* by Clark Weissman (see [Wei67]. The aim of this is to facilitate direct comparison between LISP and META-LISP, both in terms of their performance and the quality of program formulation that they make possible. The following Section presents an alternative design of the program for symbolic differentiation, which takes full advantage of the higher expressive power of META-LISP. Although this second design is a bit more complex, in terms of efficiency it outperforms even the original hand written LISP code. Section 4 reuses parts of the differentiation program for approximating the roots of polynomials using the Newton-Raphson method.

4.1 List Processing

The following section introduces some of the basic "idioms" of programming in META-LISP in the context of developing a number of list processing functions. These include variants of functions to calculate the length of a list, reversing, and mapping a list, as well as functions for splitting, merging and sorting a list. The definitions for some of these functions are compared with their equivalents in ML. The use of ML in this context serves two purposes. The first is to provide a convenient starting point for the development of the functions that

will be discussed. The second is to highlight the differences as much as the similarities between the two languages.

4.1.1 Length

4.1.1.1 Naive Length

```
fun len nil       = 0
  | len (_ :: xs) = 1 + len xs;
```

This definition of length in ML can be read as stating the following two rules for calculating the length of a list:

- the length of the empty list is 0
- the length of a list comprising a head and a tail can be obtained by adding one to the length of the tail of the list

A direct transcription of this definition into META-LISP is possible. This can be expected, given that patterns can be readily described in META-LISP

```
len
 : []      = 0
 : [_ ._]  = (+ 1 (len ._))
```

The language oriented , as opposed to the above, "*pattern oriented*", way of elaborating function definitions in META-LISP leads to a different formulation. The development of list processing functions in the language oriented style starts with the formulation of a grammatical description of *what is* a list. A list is a sequence of elements enclosed in parentheses. Depending on whether lists are allowed as elements we can have flat or nested lists. One possible form that a grammatical description of lists can take is the following:

```
list
 : [seq]

seq
 : $
 : elem seq

elem
 : atom
 : list

atom
 : is atom
```

This description can be read as saying, that

- a list is a sequence of elements enclosed by a pair of parentheses

- a sequence forming a list can be either empty or comprising an element and a sequence
- elements of a sequence that form a list can themselves be lists or atoms

There are a number of possible alternative grammatical descriptions that can be used to define what a list is. The choice of which one to use, depends on the intended application. In fact, designing an appropriate grammar is a major part of the program design process. For the moment try to use the above grammatical description as the basis for defining the function length. Clearly, calculating the length of a list, boils down to calculating the length of a sequence. This can be expressed by writing:

```
length.list
 : [length.seq]
```

The notational convention of writing *length.list*, and *length.seq*, is intended to indicate that the function *length*, that is being defined, is over lists and sequences, respectively. Rewriting the grammar rules for sequence with the same intention of indicating what function is being defined we obtain the following:

```
length.seq
 : length.$
 : length.elem length.seq
```

In exploiting this grammatical structure, we are invited to write down

- how to obtain the length of a sequence, on the assumption, that we know the length of its components,
- and to define how the length of each component is to be obtained.

Considering the first alternative: if the sequence is the empty sequence, and *length.$* tells us what its length is, we have nothing further to do, but to return its value. For the second alternative, we can say, that if we know the length of the element, and the length of a sequence, that make up a sequence, than the length of a composite sequence, comprising both, should be the sum of the lengths of its components:

```
length.seq
 : length.$
 : length.elem length.seq = (+ length.elem length.seq)

+
 : with lisp +
```

All that remains to say is, that the length of the empty sequence is zero, and that the length of an element is one, regardless of the fact whether it be a list or an atom. This last fact, can be reflected in changing the grammatical description of an element of a sequence, by leaving its internal structure unspecified:

```
length.$
 : $  = 0

length.elem
 : _  = 1
```

This example has already illustrated one of the most characteristic features of the language oriented style of programming: that it invites us to

- define the structure of the input with a suitable grammar
- exploiting this structure, specify the value of a function over composite data in terms of the values of its immediate constituents,
- and to give the value of non-composite data directly.

The fact that in a rule every constituent can be assigned the task of arbitrary complex computation, on the portion of the input that it accepts, makes it possible to reflect in the formulation of the algorithm, for a given task, the composition of the input data that is being examined. This *compositionality*, is the key to both writing, and understanding, META-LISP definitions. This is worth keeping in mind, even if the notation used does not make it so plainly and painfully obvious, as in the above example, what structure is operated on by what functions. A terser formulation of *length* can be given as follows:

```
len
 : [len.seq]

len.seq
 : $          = 0
 : _  len.seq = (+ 1 len.seq)
```

The advantage of bearing in mind the full version of the definition is that it can be readily adopted to be used in a great variety of list-processing functions, such as for counting atoms in a nested list structure

```
count
 : [c.seq]

c.seq
 : $            = 0
 : c.elem c.seq = (+ c.elem c.seq)

c.elem
 : atom         = 1
 : count
```

or flattening it.

```
flat
 : [f.seq]

f.seq
 : $            = []
 : atom f.seq   = [atom . f.seq]
 : f.seq1 f.seq = [. f.seq1 . f.seq]

f.seq1
 : f.seq
```

4.1.1.2 A Better Length

All three definitions of the function *length*, in the previous subsection, were naive in the sense of requiring space proportional to the length of the input in maintaining storage for pending recursive calls. In the list of predefined functions in [Wik87, 429], the naive version is only given in the form of a comment followed by a tail-recursive definition:

```
local fun len' n nil       = n
        | len' n (_ :: xs) = len' (n+1) xs
in fun len xs = len' 0 xs end;
```

The same gain in efficiency can be obtained in META-LISP, in this case, by using left-recursion in *len.seq*.

```
length
 : <>                = 0
 : length elem       = (+ 1 length)

elem
 : $                 = fail!
 : _
```

This time *elem* needs to be defined in such a way that it excludes the empty sequence. In the right-recursive definition there was no need for this, as the end of input test took place before *elem* was called.

The technique of using left-recursion is applicable, when the operation that we wish to apply in the semantic action is associative, as in the case for addition, above.

4.1.2 Reversing a List

In defining the function *reverse*, tail-recursion and the use of an accumulator are again desirable:

```
local fun rev' nil h       = h
        | rev' (a :: r)  h = rev' r (a :: h)
in fun rev l = rev'
```

The left-recursive formulation of *reverse* in META-LISP is again simpler:

```
reverse
 : [reverse.seq]       = reverse.seq

reverse.seq
 : reverse.seq elem    = [elem . reverse.seq]
 : elem                = [elem]

elem
 : $                   = fail!
 : _
```

It is tempting to read it "declaratively" as:

- the reversal of a sequence, comprising a sequence and an element appended to it to the right, is obtained by constructing a list whose first element is the given element and its tail is the reversal of the component sequence
- the reversal of a single element is a list containing that element

The fact that sequences are represented as lists in META-LISP accounts for the use of list constructions in the above definition.

4.1.3 List Membership

```
member.a+list
 : sought list       = (^ sought)
                       (member.seq . list)

member.seq
 : $                 = []
 : elem=sought       = t
 : elem member.seq   = member.seq

elem=sought
 : elem = (if (equal ^sought elem) t fail!)

elem : _
```

Recall that the inherited attribute assignment `(^ sought)` is equivalent to

```
(let ((^sought sought)) (?= (member.seq list)))
```

given that `sought` and `list` have the appropriate bindings.

4.1.4 Mapping a List

The practical advantage of higher order functions lies in that they allow for common patterns of computations to be abstracted out. The most familiar example is the function *map* which

applies a given functions to each element of a list in turn, and returns a list with the values of the applications. It can be defined in META-LISP as follows:

```
map
 : list function = (^ function)
                   (map.seq . list)

map.seq
 : $ = []
 : _ map.seq = [(^function _) . map.seq]
```

4.1.5 Splitting a List into two

Again just consider splitting a sequence of elements. Return two values in the form of *synthesised attributes* `p1@split.seq` and `p2@split.seq`.

```
split
 : [split.seq]        =  (@ p1 <- p1@split.seq)
                         (@ p2 <- p2@split.seq)

split.seq
 : $                  = (@ p1 <- [])
                        (@ p2 <- [])
 : e1 $               = (@ p1 <- [e1])
                        (@ p2 <- [])
 : e1 e2 split.seq    = (@ p1 <- [e1 . p1@split.seq])
                        (@ p2 <- [e2 . p2@split.seq])
e1 : _

e2 : _
```

4.1.6 Merging two sorted lists

A function to merge two sorted lists of integers can be defined as shown below:

```
merge.parts
 : [i1 tl1] [i2 tl2] = (if (precedes i1 i2)
                            [ i1 . (merge.parts tl1 [i2 . tl2])]
                           [ i2 . (merge.parts tl2 [i1 . tl1])])
 : [] list           = list

precedes
 : with lisp <


i1 : _
i1 : _
tl1 : ._
tl2 : ._
```

By changing the definition of *precedes* appropriately the same definition can be used to merge other kinds of lists too. One possibility is

```
precedes
 : <> =  ^pred
```

This will work assuming that `^pred` is bound to the name of an appropriate effective concept to be used as a predicate to determine the ordering of two elements.

4.1.7 Sorting

The related problems of sorting and searching are fundamental in computer programming. Volume 2 of Donald Knuth "Art of Computer Programming" is devoted entirely to these problems. The function for sorting presented in this section is one of very many possible sorting algorithms. It is appropriate to form the basis of a "built-in" sorting function because its requires, on average, $nlog(n)$ comparisons, which is very good when there is no *a priori* knowledge about the distribution of items in the list to be sorted. The *divide and conquer* strategy would split the list and sort the parts and then merge the result. If we make the splitting easy, then we carry the burden of sorting in the merging. Alternatively we can do the brunt of the work in the splitting of the list, in which case merging is trivial. The first choice leads to merge sort the second is called quick sort, (or rather split sort?)

Using the functions for splitting and merging, introduced in the previous section merge sort can be defined in META-LISP as follows:

```
ms
 : pred list =  (^ pred) (sort.list list)

sort.list
 : sort.short_list
 : sort.longer_list

sort.short_list
 : []
 : [i1]              = [i1]

sort.longer_list
 : [split.seq]       = (merge.parts
                        (sort.list part1@split.seq)
                        (sort.list part2@split.seq))
```

4.2 Symbolic Differentiation: as in LISP

The subject of both this and the following section will be the design of a program for Symbolic Differentiation. Both sections will describe, in detail, a program for Symbolic differentiation written in META-LISP. The design of the program, presented in this section, mirrors that of a symbolic differentiation program presented in the final chapter of the Lisp 1.5 Primer, see [Wei67]. The purpose of this is to allow direct comparison with LISP, both in terms of performance and quality of program formulation. The next section will present an alternative way of writing a program for symbolic differentiation in META-LISP. For the design of this second program advantage will be taken of the higher expressive powers of META-LISP, not only in the way the program is formulated, but in its overall design. The present section will demonstrate that, for the task of writing a program for symbolic differentiation, the advantages of higher-level program formulation, offered by META-LISP, can be enjoyed without any sacrifice in efficiency. This is in sharp contrast to usual expectations, where gain in expressive power is usually paid for by loss in efficiency. The main result of the following section is, if anything, even more striking. It shows that, in this particular instance, the exploitation of the higher expressive powers of META-LISP can even lead to *gains* in efficiency when compared to the hand coded LISP program (some 40% reduction in runtime).

4.2.1 Program Strategy

The program reads a polynomial constructed from the arithmetic operators and exponentiation in the usual infix notation, and then prints its derivative. The program repeatedly calculates the derivative of polynomials until told to stop (see figure 4.1).

```
THE DERIVATIVE OF-
3(X^3 + X) + 2X^3,
WITH RESPECT TO-
X,

IS-
3 (3 x^2  + 1 ) + 6 x^2
THE DERIVATIVE OF-
B+B)*(A-B),

(POORLY FORMED EXPRESSION)
.

FINIS
```

Figure 4.1: Symbolic Differentiation

The strategy proposed by Weissman for developing this program is to translate the given expression into a fully parenthesised prefix representation, differentiate and simplify that form, and translate the resulting form back into infix form, in accordance with the usual conventions for operator precedence. The only point where the META-LISP program, presented in this subsection, deviates from this overall design is that instead of prefix notation, it translates to and from fully parenthesised infix notation. As will be pointed out, the use of data abstraction enables the formulation of this program in META-LISP in a way that makes it easy to switch from one internal representation to the other.

4.2.2 Top Level Elaboration

META-LISP programs are made to belong on entry to some module. The first step in the development of a program is to *declare* the module to which the program will belong:

```
| ?= package diff
```

In formulating a suitable top-level definition, in META-LISP for the program our main concern is the identification of the immediate constituent effective concepts in terms of which the relevant input can be captured, and the form in which these parameters are to be transformed to obtain the output.

We can break down the problem by assuming that we have two input procedures: one to read and validate an algebraic expression, named *inexpr*; and another, to read and validate a variable, named *invar*. These procedures are combined in a grammar rule prescribing the intended sequence of their call:

```
diff
 : inexpr invar
```

The output can be specified using functional composition, which takes the result of the input procedures as parameters, giving as the top level definition:

```
diff
 : inexpr invar = (show (simplify (deriv inexpr invar)))
```

To express the idea that this process is to be repeated, the left-recursive form of iteration can be used:

```
diff
 : <>
 : diff inexpr invar = (show (simplify (deriv inexpr invar)))
```

This will work, if we can assume, that when the user requests termination of the session with the program, either *inexpr* or *invar* will return failure.

As the effective concepts introduced in the above definition are each elaborated, a common pattern emerges: the identification of further immediate procedures and their combination to achieve the desired result is repeated. The attention is always focused on the immediate constituent level. In this regard, META-LISP strongly supports *top-down* or to use John Allen's phrase, *level-wise* programming.

To obtain a complete program the immediate constituent concepts introduced above need to be elaborated. This will lead to the introduction of further concepts until eventually we reach bedrock: elementary procedures and/or LISP primitives.

4.2.3 Reading and Validating the Input

The overall structure of both input routines is similar. Both involve three main steps:

- Prompting the User
- Reading a line of input
- Validating the input

As can be seen in Figures 4.2 and 4.3, validating the input is responsible for returning failure when the user requests termination of the session by typing a full stop.

```
inexpr
 : prompt1 readl      = (validexpr readl)

prompt1
 : <>                 = (format t "~%THE DERIVATIVE OF-~%")

validexpr
 : finish             = fail!
 : [expr]             = expr
 : error1 readl       = (validexpr readl)

finish
 : :end               = (format t "~&FINIS~%") t

error1
 : <>                 = (format t "~%(POORLY FORMED ~%EXPRESSION)~%")
```

Figure 4.2: Reading and Validating an Expression

The second alternative of *validexpr* calls *expr* which is responsible for translating the input line into internal representation. This will succeed if the entire line of input read forms a valid algebraic expression. If this translation fails, or if not the entire line is deemed to be an algebraic expression, then the third alternate is taken, which issues an appropriate error message, then reads a new line of input, and repeats the validation process.

Reading and validating a variable is analogous, as shown in Figure 4.3

```
invar
 : prompt2 readl      = (validvar readl)

prompt2
 : <>                 = (format t "WITH RESPECT TO-~%")

validvar
 : finish             = fail!
 : [var]              = var
 : error2 readl       = (validvar readl)

error2
 : <>                 = (format t "~&REENTER VARIABLE")
```

Figure 4.3: Reading and Validating a Variable

Reading and validating both an expression and a variable uses *readl* to read a line of input. Its definition is discussed in the next subsection.

4.2.4 Reading a Line of Input

The line reading routine, *readl*, is invoked with no input. The action of the reader is governed by the last character read. Accordingly, *readl* first calls *read-char* to read a character from the current input stream and then calls *readlh* with the first character read to constructs a list of input characters using an accumulator. The steps carried out by *readlh* are as follows:

```
readl
 : <>                 = (readlh (read-char) [])

readlh
 : comma line         = (reverse line)
 : end line           = :end
 : skip line          = (readlh (read-char) line)
 : char2digit line    = (readlh (read-char) [char2digit . line])
 : char2symbol line   = (readlh (read-char) [char2symbol . line])
```

Figure 4.4: Reading a Line of Input

- If the last character read is a *comma*, then that character is read and the line of input, *line*, that has been constructed is *reverse*d and returned.
- If the last character read signifies the end of the session then the keyword `:end` is returned, to be acted upon by *finish* in *validexpr* and *validvar*.

- If the last character read is a character to be skipped, it is ignored.
- If the last character read is a character representing a digit, then it is converted to a digit and the reading of the input is continued with this digit added to the line read so far.
- Otherwise the last character read is converted to a symbol and is added to the line read so far.

In all but the first two alternates the next character is read, using *read-char*, and *readlh* is called again. *readlh* uses tail recursion together with an accumulator.

4.2.5 Translating into Internal Representation

The program specification contains a grammatical description of the class of algebraic expressions to be differentiated. The task of validating and translating algebraic expressions from infix to fully parenthesised notation is a syntax-directed translation task that can be readily formulated in META-LISP. Its definition is shown in Figure 4.5.

It is important to emphasise that the definition of the translation of algebraic expressions into internal representation is formulated in such a way that it makes no specific commitment to their precise form. That is to say, whether it be fully parenthesised infix, prefix, postfix or mixfix form. These details are specified in the form of constructors, which specify how algebraic expressions are to be represented as list structures. Figure 4.6 shows the definition of constructors for fully parenthesised infix notation. Corresponding to these constructors there are *abstract analysers* that are used both to recognise algebraic expressions and to select their components. These are presented in Figure 4.7. Changing the underlying representation can thus be achieved by changing the constructors to construct list representation of algebraic expressions in a form that corresponds to their Abstract Syntax. An earlier version of the program, reported in [Laj90] was formulated without the benefit of synthesised attributes, e.g. explicit support for data-abstraction. Reusability of the program was thus severely limited.

```
expr
 : term
 : expr + term        = (mk-Sum expr term)
 : expr - term        = (mk-Diff expr term)

term
 : secondary / term   = (mk-Quot secondary term)
 : secondary mul term = (mk-Prod secondary term)
 : secondary

mul
 : '*
 : <>

secondary
 : primary ^ constant =  (mk-Power primary constant)
 : primary

primary
 : open expr close    = expr
 : constant
 : var

open
 : '|(|

close
 : '|)|

constant
 : digit
 : constant digit     = (plus (times 10 constant) digit)

digit
 : any 0 1 2 3 4 5 6 7 8 9

var
 : any a b c d e f g h i j k l m n o p q r s t u v w x y z
```

Figure 4.5: Translating into Internal Representation

```
mk-Sum
 : u v                  = [u + v]

mk-Diff
 : u v                  = [u - v]

mk-Quot
 : u v                  = [u / v]

mk-Prod
 : u v                  = [u * v]

mk-Power
 : u v                  = [u ^ v]

mk-Minus
 : a                    = [- a]

mk-Expr
 : u op v               = [u op v]
```

Figure 4.6: Constructors for Algebraic Expressions

```
Const
 : is integerp

Var
 : var

Sum
 : [u + v]              = (@ u) (@ v)

Diff
 : [u - v]              = (@ u) (@ v)

Prod
 : [u * v]              = (@ u) (@ v)

Quot
 : [u / v]              = (@ u) (@ v)

Power
 : [u ^ n]              = (@ u) (@ n)

Minus
 : [- a]                = (@ a) [- a]

Expr
 : [a +/- b]            = [a +/- b]
```

Figure 4.7: Abstract Analysers of Algebraic Expressions

4.2.6 Derivation

The following rules of differentiation are considered:

$$
\begin{aligned}
\frac{d}{dx}u &= 0; \text{ if } u \neq f(x) \\
\frac{d}{dx}u &= 1; \text{ if } u = x \\
\frac{d}{dx}(u+v) &= \frac{d}{dx}u + \frac{d}{dx}v \\
\frac{d}{dx}(u-v) &= \frac{d}{dx}u - \frac{d}{dx}v \\
\frac{d}{dx}(uv) &= v\frac{d}{dx}u + u\frac{d}{dx}v \\
\frac{d}{dx}(u/v) &= (v\frac{d}{dx}u - u\frac{d}{dx}v)/v^2 \\
\frac{d}{dx}(u^n) &= nu^{n-1}\frac{d}{dx}u
\end{aligned}
$$

These rules can be easily transcribed into META-LISP as shown in Figure 4.8. The design of this function have already been discussed in Section 3.1.2. The inclusion of additional rules should pose no difficulty.

```
deriv
 : Const x          = 0
 : Var x            = (if (same Var x) 1 0)
 : Sum x            = (mk-Sum (deriv u@Sum x) (deriv v@Sum x))
 : Diff x           = (mk-Diff (deriv u@Diff x) (deriv v@Diff x))
 : Prod x           = (mk-Sum
                        (mk-Prod v@Prod (deriv u@Prod x))
                        (mk-Prod u@Prod (deriv v@Prod x)))
 : Quot x           = (mk-Quot
                        (mk-Diff
                          (mk-Prod v@Quot (deriv u@Quot x))
                          (mk-Prod u@Quot (deriv v@Quot x)))
                        (mk-Prod v@Quot v@Quot))
 : Power x          = (mk-Prod
                        n@Power
                        (mk-Prod
                          (mk-Power u@Power (1- n@Power))
                          (deriv u@Power x)))

x
 : _
```

Figure 4.8: Differentiation Rules

4.2.7 Simplification

This part of the program handles simplification of algebraic expressions, and uses the same set of rules as the Primer example. *simplify* is a supervisor program that parcels the task of simplification according to the arithmetic operators involved. There are three alternatives to be considered depending on the kind of expression involved: if the expression to be simplified

1. is atomic, then it is already in a simplified form.
2. has unary minus as its outermost operator, then it is simplified accordingly.
3. involves a binary operator, then the operands are first simplified and are then passed to the corresponding simplification routine.

Binary operators are handled in a *data-directed* way. (See [ASS85, 136-142]). As shown in Figure 4.9, *s.expr* is responsible for recognising the presence of binary algebraic expressions to be simplified. It also analyses its input, and extracts the operands u and v. These operands are simplified before they are passed to the appropriate simplification routine returned by *s.op*. This is another illustration of the power of syntax-directed translation as a parameter passing mechanism, in that components of the input are not only selected, as would be the case with pattern matching, but the desired transformations, in this case simplification, are also applied to them.

```
simplify
 : a=atom
 : Minus              = (s.Minus (simplify a@Minus))
 : s.expr             = (s.op@s.expr u@s.expr v@s.expr)

s.expr
 : [u s.op v]         = (@ u <- (simplify u))
                        (@ v <- (simplify v))
                        (@ s.op)

op
 : any + - * /

s.op
 : -                  = < s.- >
 : +                  = < s.+ >
 : *                  = < s.* >
 : /                  = < s./ >
 : ^                  = < s.^ >
```

Figure 4.9: Simplification

The actual rules used for simplifying algebraic expressions are straight-forward. They can produce simpler expressions, but not necessarily the simplest. As an illustration, consider the rules for simplifying *sums*. Figure 4.10 shows the way these rules are presented on page 172 in the LISP 1.5 Primer. The rule based form of META-LISP allows for a

For an expression of the form (PLUS a b) the following simplification rules are used by SPLUS. Higher-numbered rules assume prior rules failed.

	Rule	Value	Line No.
1.	a and b = *constant*	a + b	111
2.	a = 0	b	115
3.	b = 0	a	112
4.	b = *constant*, a ≠ *constant*	(PLUS b a)†	113
5.	a = b	(TIMES 2 a)†	116
6.	a = (MINUS a_1) b = (MINUS b_1)	(MINUS (PLUS a_1 b_1)†	121
7.	a = (MINUS a_1), b = a_1	0	125
8.	a = (MINUS a_1), b ≠ *constant*	(PLUS b a)†	126
9.	b = (MINUS b_1), a = b_1	0	128
10.	b = (MINUS b_1), a ≠ *constant*	(PLUS a b)†	129
11.	all else	(PLUS a b)†	130

† The *expression* is further simplified by the function COLLECT.

Figure 4.10: SPLUS in the LISP 1.5 Primer

particularly straight-forward way of transcribing these rules, as shown in Figure 4.11.

```
s.+
 : a=const b=const    = (plus a=const b=const)
 : a=0 b              = b
 : a b=0              = a
 : a b=const          = (collect [b=const + a])
 : a b                ? (if (equal a b) (collect [2 * a]) fail!)
 : Minusa Minusb      = (mk-Minus (collect [a@Minusa + b@Minusb]))
 : Minusa b           = (if (equal a@Minusa b) 0 (collect [b + Minusa]))
 : a Minusb           = (if (equal a b@Minusb) 0 (collect [a + Minusb]))
 : a b                = (collect [a + b])
```

Figure 4.11: SPLUS in META-LISP

The transcription of rule 5.

```
: a b                ? (if (equal a b) (collect [2 * a]) fail!)
```

is interesting in that it uses the *semantic backtracking* feature of META-LISP to impose the context condition that $a = b$. Rules 7. and 8. are combined in the alternative

```
: Minusa b           = (if (equal a@Minusa b) 0 (collect [b + Minusa]))
```

Similarly rules 9. and 10. are combined in the alternative

```
: a Minusb           = (if (equal a b@Minusb) 0 (collect [a + Minusb]))
```

It is instructive to compare the META-LISP definition with the LISP original, shown in Figure 4.12. The code generated by the META-LISP compiler is very similar to this hand-written code.

```
(DEFUN SPLUS (E)
  (COND ((NUMBERP (CADDR E))
         (COND ((NUMBERP (CADR E)) (EVAL E))
               ((ZEROP (CADDR E)) (CADR E))
               (T (COLLECT (LIST (CAR E) (CADDR E) (CADR E))))))
        ((AND (NUMBERP (CADR E)) (ZEROP (CADR E))) (CADDR E))
        ((EQUAL (CADR E) (CADDR E))
         (COLLECT (LIST 'TIMES 2 (CADR E))))
        ((AND (NOT (ATOM (CADR E))) (EQ (CAADR E) 'MINUS))
         (COND ((AND (NOT (ATOM (CADDR E))) (EQ (CAADDR E) 'MINUS))
                (LIST 'MINUS
                      (COLLECT (LIST (CAR E) (CADADR E) (CADR (CADDR E))))))
               ((EQUAL (CADADR E) (CADDR E)) 0)
               (T (COLLECT (LIST (CAR E) (CADDR E) (CADR E))))))
        ((AND (NOT (ATOM (CADDR E))) (EQ (CAADDR E) 'MINUS))
         (COND ((EQUAL (CADR (CADDR E)) (CADR E)) 0) (T (COLLECT E))))
        (T (COLLECT E))))
```

Figure 4.12: SPLUS in LISP

The other simplification rules can be seen in Figure 4.13. It is worth pointing out, that this part of the program is also written in a representation independent style, although internally, it uses infix representation.

```
s.Minus
 : Const              = (times -1 Const)
 : Minus              = a@Minus
 : a                  = (mk-Minus a)

s.-
 : u v                = (s.+ u (s.Minus v))

s.+
 : a=const b=const    = (plus a=const b=const)
 : a=0 b              = b
 : a b=0              = a
 : a b=const          = (collect [b=const + a])
 : a b                ? (if (equal a b) (collect [2 * a]) fail!)
 : Minusa Minusb      = (mk-Minus (collect [a@Minusa + b@Minusb]))
 : Minusa b           = (if (equal a@Minusa b) 0 (collect [b + Minusa]))
 : a Minusb           = (if (equal a b@Minusb) 0 (collect [a + Minusb]))
 : a b                = (collect [a + b])

s.*
 : a=const b=const    = (times a=const b=const)
 : a=0 b              = 0
 : a=1 b              = b
 : a=const b          = (collect [a=const * b])
 : a b=0              = 0
 : a b=1              = a
 : a b=const          = (collect [b=const * a])
 : a b                ? (if (equal a b) (s.^ a 2) fail!)
 : Minusa Minusb      = (collect [a@Minusa * b@Minusb])
 : Minusa b           = (if (equal a@Minusa b)
                            (mk-Minus (s.^ a@Minusa 2))
                          (collect [b * Minusa]))
 : a Minusb           = (if (equal a b@Minusb)
                            (mk-Minus (s.^ a 2))
                          (collect [a * Minusb]))
 : a b                = (collect [a * b])

s.^
 : a b=0              = 1
 : a b=1              = a
 : a=atom b           = (mk-Power a=atom b)
 : Power c            = (mk-Power u@Power (times n@Power c))
 : Minusa b=even      = (s.^ a@Minusa b=even)
 : Minusa b=odd       = (mk-Minus (s.^ a@Minusa b=odd))
 : a b                = (mk-Power a b)

s./
 : a b                ? (if (equal a b) 1 fail!)
 : a=0 b              = 0
 : a=1 b              = (mk-Quot a b)
 : a b=1              = a
 : a=const b=const    = (quotient a=const b=const)
 : a b=const          = (collect [(quotient 1.0 b=const) * a])
 : a Minusb           = (s.* a (mk-Minus (mk-Quot 1 b@Minusb)))
 : a b                = (mk-Prod a (mk-Quot 1 b))
```

Figure 4.13: Simplification Rules

4.2.7.1 Collect

collect is a function used in simplifying both sums, products and quotients. It provides additional simplification rules by attempting to simplify certain patterns of nested addition and multiplication. These rules are shown in Figure 4.14.

```
collect
 : atom
 : [a=atom op b=atom] = (mk-Expr a=atom op b=atom)
 : [a op b=atom]      = (collect [b=atom op a])
 : [a=const + b+c]    = (mk-Sum (plus a=const b@b+c) c@b+c)
 : [a=const * b*c]    = (mk-Prod (times a=const b@b*c) c@b*c)
 : [a+b + c+d]        = (mk-Sum (plus a@a+b c@c+d) (mk-Sum b@a+b d@c+d))
 : [a*b * c*d]        = (mk-Prod (times a@a*b c@c*d) (mk-Prod b@a*b d@c*d))
 : [u op v]           = (mk-Expr u op v)

a*b
 : Prod               = (if (Const? u@Prod)
                            { (@ a <- u@Prod) (@ b <- v@Prod) }
                          fail!)

a+b
 : Sum                = (if (Const? u@Sum)
                            { (@ a <- u@Sum) (@ b <- v@Sum) }
                          fail!)

b*c
 : Prod               = (if (Const? u@Prod)
                            { (@ b <- u@Prod) (@ c <- v@Prod) }
                          fail!)

b+c
 : Sum                = (if (Const? u@Sum)
                            { (@ b <- u@Sum) (@ c <- v@Sum) }
                          fail!)

c*d
 : Prod               = (if (Const? u@Prod)
                            { (@ c <- u@Prod) (@ d <- v@Prod) }
                          fail!)

c+d
 : Sum                = (if (Const? u@Sum)
                            { (@ c <- u@Sum) (@ d <- v@Sum) }
                          fail!)

Const?
 : Const              = t
 : <>                 = nil
```

Figure 4.14: Collect

These rules can be understood with reference to the definitions of these nested patterns, such as $a*b$, $a+b$, etc. Consider the definition of $b+c$ as a representative example. It refers to the sum of two expressions b and c such that b is a constant. With this, the fourth rule

can be read as saying that the sum of a constant a and the sum of b and c such that b is also a constant, can be simplified as the sum of the result of adding a b together and the the expression c.

4.2.8 Show Result

The result of derivation and simplification is displayed in the usual format for algebraic expressions which takes account of the precedence of the operators involved. The supervisory program for this, *out*, uses the same *data directed* technique to parcel out the task of displaying the result in an infix form, with the usual conventions for operator precedence, as was used in organising the simplification of algebraic expression. As seen on Figure 4.15 the task of displaying constants and variables is passed immediately to the LISP built in function `format`. The task of displaying expressions, that involve unary minus, is just as straight-forward. For expressions involving binary operators, the appropriate display routine is selected by *out_expr*, and is applied to the operands:

```
show
 : msg out              = t

msg
 : <>                   = (format t "~&IS-~%")

out
 : Const                = (format t "~S " Const) t
 : Var                  = (format t "~S" Var) t
 : Minus                = (format t "- ") (out a@Minus)
 : out.expr             = (op@out.expr a@out.expr b@out.expr)

out.expr
 : [a out.op b]         = (@ a) (@ b) (@ op <- out.op)

out.op
 : +                    = < out.Sum >
 : *                    = < out.Prod >
 : /                    = < out.Quot >
 : ^                    = < out.Power >
```

Figure 4.15: Display Result

The original LISP code for the individual display routines were presented in the *Primer* without much discussion, as they were deemed to be simple enough to be followed by the reader. Hence, much of the code for the individual display routines, presented in Figure 4.16, had to be reconstructed on the basis of examining the original LISP code. Some simplifications have been applied to the organisation of the code, such as factoring out repeated pieces of code. This lead to the introduction of a new function *xterm* which does

not appear in the original formulation.

```
out.Sum
 : a=const b          = (out.Sum b a=const)
 : a b<0              = (out a) (blank) (out b<0)
 : a Minus            = (out a) (blank) (out Minus)
 : a b                = (out a) (format t " + ") (out b)

b<0
 : b=const            = (if (> 0 b=const) b=const fail!)

blank
 : <>                 = (format t " ")

out.Prod
 : xterm xterm

xterm
 : atom               = (out atom)
 : Expr               = (wrap Expr)
 : out

+/-
 : any + -

wrap
 : a                  = (princ "(") (out a) (princ ")")

out.Quot
 : xterm pslash xterm = t

pslash
 : <>                 = (format t "/")

out.Power
 : a=atom n           = (format t "~S^^S " a=atom n) t
 : a n                = (wrap a) (format t "^^S " n) t
```

Figure 4.16: Output Routines

The elementary definitions for the program are shown in Figure 4.17

4.2.9 Performance: LISP versus META-LISP

The performance of the program written in META-LISP is identical to the LISP version. Although the code generated by the META-LISP compiler for effective concepts which define their input in terms of patterns, is very similar to the original LISP code, the code generated for effective concepts that are involved in parsing the input look quite different. However, it seems that the LUCID compiler eliminates all the apparent differences.

```
*
 : '*

+
 : '+

-
 : '-

/
 : '/

^
 : '^

1-
 : with lisp 1-

>
 : with lisp >

a
 : _

a=0
 : '0

a=1
 : '1

a=atom
 : is atom

a=const
 : is numberp

atom
 : is atom

b
 : _

b=0
 : '0

b=1
 : '1

b=atom
 : is atom

b=const
 : is numberp

b=even
 : is evenp

b=odd
 : is oddp

c
 : _

c=const
 : is numberp

char
 : _

char2digit
 : '#\0              = 0
 : '#\1              = 1
 : '#\2              = 2
 : '#\3              = 3
 : '#\4              = 4
 : '#\5              = 5
 : '#\6              = 6
 : '#\7              = 7
 : '#\8              = 8
 : '#\9              = 9

char2symbol
 : char              = (intern (string char))

comma
 : '#\,

d
 : _

end
 : '#\.

equal
 : with lisp equal

format
 : with lisp format

if
 : with lisp if

intern
 : with lisp intern

line
 : _

Minusa
 : [- a]             = (@ a) [- a]

Minusb
 : [- b]             = (@ b) [- b]

n
 : is integerp

plus
 : with lisp +

princ
 : with lisp princ

quotient
 : with lisp /
```

```
read-char
 : with lisp read-char

reverse
 : with listp reverse

same
   : with lisp eq

skip
 : any #\Newline #\Space

string
 : with lisp string

times
 : with lisp *

u
 : _

v
 : _
```

Figure 4.17: Elementary Definitions for Symbolic Differentiation

4.3 Symbolic Differentiation: A Language Oriented Design

4.3.1 Program Strategy

In the previous design the task of the program was partitioned in a way that involved several 'passes'. I.e. the input was first examined, and translated into internal form, then this representation was traversed for the purpose of applying the appropriate rules of differentiation, and then it was re-examined to carry out simplification. In the language oriented design all these passes are not required. The rules of differentiation and simplification can be applied to the syntactically correct forms of the input as they are parsed.

4.3.2 Top-Level Elaboration

The top-level is reorganised a bit. *inexpr* will no longer validate the input as this will take place as the input is simultaneously differentiated and the result is simplified while it is parsed and validated. In order to achieve this it is also necessary to know the variable of differentiation so that the derivative of an input variable can be computed as it is recognised in the input. For this reason the variable of differentiation read by *invar* is used to set an inherited attribute `^x` which can then be referenced in the course of differentiation a variable. Assuming that *diff.expr* will return a synthesised attribute `d@diff.expr` which denotes the derivative of the expression read in, the top-level elaboration of the program reads as follows:

```
diff
 : <>
 : diff inexpr invar  = (^ x <- invar)
                        (diff.expr . inexpr)
                        (show d@diff.expr)
```

4.3.3 Revised Input Routines

As has been pointed out *inexpr* in this version of the program will no longer be responsible for the validation of the input. Its new definition, shown in Figure 4.18 reflects this. Reading and validating a variable is unchanged from the previous version. This is indicated by the fact that it is simply imported from the previous version.

4.3.4 Differentiating and Validating an Expression

There is no need for a separate supervisor program to dispatch the appropriate rules. Instead, we need only to state the rules of differentiation that we would like to use, as shown in Figure 4.19.

```
inexpr
 : prompt1 readl      = (if (end? readl) fail! readl)

prompt1
 : <>                 = (format t "~%THE DERIVATIVE OF-~%")

end?
 : :end               = (format t "~&FINIS~%") t
 : <>

readl
 : with diff readl

invar
 : with diff invar
```

Figure 4.18: Revised Input Routines

Note also, that instead of constructing an internal representation for the derivative, calls to the appropriate simplification routines are made as the differentiation rules require the use of particular algebraic operations.

The functions used for simplification and the output procedures are identical to the previous version, as are the constructors and the abstract analysers. Figure 4.20 shows the new design of the program.

4.3.5 Comparison of the two Designs

Undoubtedly the second version is somewhat more difficult to understand at first glance, as it involves the use of attributes. Apart from the difficulties presented by the use of attributes, the second program is just as easy to write and read as the first. In fact, by arranging for the calculation of the derivative and its simplification to take place as the input first examined in one pass, the program becomes a fair bit shorter. More significantly, there is gain in performance as a result of this. By comparing the performance of the two variants of the program in calculating and simplifying the derivative of a line of input, the second version used about 40% less CPU time than the first. And since the performance of the first version of the program written in META-LISP was identical to the performance of the original LISP program, we can say that, in this instance, META-LISP outperforms LISP itself. It should be noted, that a similar change of design could be applied to the original LISP program, as well. Given, the way the parser was written originally, it would not be that simple to incorporate all the extra functionality required. As an illustration of these difficulties consider the original definition of *expression* shown in Figure 4.22.

```
d.Const
 : u                  = 0

d.Var
 : u x                = (if (eq u x) 1 0)

d.Sum
 : du dv              = (s.+ du dv)

d.Diff
 : du dv              = (s.- du dv)

d.Prod
 : u du v dv          = (s.+ (s.* v du) (s.* u dv))

d.Quot
 : u du v dv          = (s./ (s.- (s.* v du) (s.* u dv)) (s.^ v 2))

d.Power
 : n u du             = (s.* n (s.* (s.^ u (1- n)) du))
```

Figure 4.19: Rules of Differentiation

```
diff.expr
 : expr $             = (@ d <- d@expr) expr
 : error1 inexpr      = (diff.expr . inexpr)

expr
 : term               = (@ d <- d@term) term
 : expr + term        = (@ d <- (d.Sum d@expr d@term))
                        (mk-Sum expr term)
 : expr - term        = (@ d <- (d.Diff d@expr d@term))
                        (mk-Sum expr (mk-Minus term))

term
 : secondary / term   = (@ d <- (d.Quot secondary d@secondary term d@term))
                        (mk-Quot secondary term)
 : secondary mul term = (@ d <- (d.Prod secondary d@secondary term d@term))
                        (mk-Prod secondary term)
 : secondary          = (@ d <- d@secondary) secondary

secondary
 : primary ^ constant = (@ d <- (d.Power constant primary d@primary))
                        (mk-Power primary constant)
 : primary            = (@ d <- d@primary) primary

primary
 : open expr close    = (@ d <- d@expr) expr
 : constant           = (@ d <- (d.Const constant)) constant
 : variable           = (@ d <- (d.Var variable ^x)) variable
```

Figure 4.20: Differentiating and Validating an Expression

Let `(2 y ^ 3 + y)` be the input to *expr*

- Since *expr* is left recursive the start-up rule (the first alternate) is expanded first. So *expr* calls *term*.
- Since *term* is right recursive it will attempt to expand its first alternate first, by calling the first component of its first alternate *secondary*.
- In turn *secondary* calls the first component of its first alternate *primary*.
- The first alternate of *primary* fails, since the first element of the input is not an opening parenthesis. Its second alternate however succeeds, since the first element of the input is a *constant*. The derivative of the constant `2` is calculated using the rule *d.Const* giving `0` which is returned as the synthesised attribute of *primary*. The value returned by *primary* is the value of *constant* i.e. `2`.
- Having successfully expanded the first component of its first alternate, *secondary* calls the next component `^`. This fails. Using left-factoring, which eliminates the need for backtracking (see Figure 3.3), *secondary* then returns the derivative of *primary* (`d@primary`) as its synthesised attribute `d@secondary` and the value `2`.
- Having successfully expanded the first component of its first alternate, *term* calls the second component `/` which will fail. The second component, of the second alternate of *term* (i.e. *mul*) is expanded next via left-factoring, which will succeed (c.f. Figure 4.5 on page 74 for its definition).
- *term* is now invoked recursively with input `(y ^ 3 + y)`, calling *secondary* which calls *primary* which succeeds with its third alternate *variable*, returning the derivative of the variable `y` using *d.Var* which is `1` (since `y` equals the value of the inherited attribute `^x`, the variable of differentiation which is `y`, c.f. Section 4.3.2) and the matched variable `y`.
- Having successfully expanded the first component (*primary*) of its first alternate, *secondary* calls the next component `^`, which now succeeds, indicating the presence of exponentiation. *constant* is then called and matches `3`. *d.Power* is then used to compute the derivative of y^3 through appropriate simplifications giving `(3 * (y ^ 2))` as the value of the synthesised attribute `d@secondary` and `(y ^ 3)` – as constructed by *mk-Power* – as the value of *secondary*.
- After an unsuccessful attempt to find further terms as part of an implicit product, the first recursive call to *term* returns with `(3 * (y ^ 2))` as its synthesised attribute and `(y ^ 3)` as its value.
- Given the previously computed value `2` of *secondary* and its synthesised attribute `d@secondary` which was `0`, the semantic action

```
(@ d <- (d.Prod secondary d@secondary term d@term))
(mk-Prod secondary term)
```

  is evaluated to yield `(6 * (y ^ 2))` as the derivative of the product `2 y ^ 3` which is then returned as the synthesised attribute of *term* and `(2 * (y ^ 3))` – as constructed by *mk-Prod* as the value of *term*.
- The synthesised attribute of *term* is then assigned to be the first synthesised attribute of *expr*. The value of *term* then becomes the first value produced by *expr*. Given these values *expr* now attempts to expand its left recursive rules with the remaining input: `(+ y)`.
- The expansion of the first left recursive rule will be successful in finding a new *term* (following the `+` sign) – with value `y` and derivative `1`. Using the values of the synthesised attributes `d@expr` = `(6 * (y ^ 2))` and `d@term` = `1` *d.Sum* produces the derivative for the entire input `((6 * (y ^ 2)) + 1)`.

Figure 4.21: Differentiation in Action

4.3.6 The Workings of the Program

Here, as in the previous version of the symbolic differentiation program the set of valid inputs to the program (the class of algebraic expressions being considered) has been defined explicitly as a language. The grammatical structure thus imposed on the input, however, has not been exploited fully in the previous version. It has only been used to construct a suitable internal representation to be examined in further passes. The current version, in contrast, exploits fully the structure imposed on the input as the means of making explicit the *applicative structure* of the desired computation, viz. the calculation of the derivative. It is this feature of the design of the program that makes it fully *language oriented.*

As an illustration of how the grammatical structure is exploited to carry out the task of differentiation consider the first alternative of the effective concept *secondary* dealing with exponentiation:

```
: primary ^ constant = (@ d <- (d.Power constant primary d@primary))
                       (mk-Power primary constant)
```

The base of exponentiation is selected and recognised by the effective concept *primary*, whereas its power is by *constant.* Our aim is to calculate the derivative of exponentiation of *primary* to *constant* power. Recall that the rule of differentiation for exponentiation states:

$$\frac{d}{dx}(u^n) \quad = \quad nu^{n-1}\frac{d}{dx}u$$

This rule is formulated in META-LISP as:

```
d.Power
 : n u du              = (s.* n (s.* (s.^ u (1- n)) du))
```

Note that the derivative of exponentiation is a function not only of the algebraic expression that is raised to a constant power but of its derivative. Hence to apply this rule it is necessary to make available, in addition to the algebraic expressions involved in the exponentiation, its derivative. The derivative of *primary* is made available as a synthesised attribute (e.g. `d@primary`). And since the algebraic expression corresponding to *secondary* itself may be an operand of another composite algebraic expression, its derivative is made available as a synthesised attribute also, i.e. the attribute assignment:

```
(@ d <- (d.Power constant primary d@primary))
```

makes available at the level of *term* the derivative of *secondary* as a synthesised attribute, and a suitable representation of the algebraic expression in question as its principal value obtained by `(mk-Power primary constant)`. Similarly each effective concept involved in the definition of the input language of the program will return two values: a principal value representing an algebraic (sub)expression and its derivative as a synthesised attribute.

The working of the program can be best illustrated by considering a concrete example and following through the steps of computation as shown in Figure 4.21.

The present language oriented design of the symbolic differentiation program improves on the previous design by making explicit not only the grammatical structure of its input but exploiting it to capture the *applicative structure* of computing the desired output. At the same time, it also supports *data abstraction* and *representation independent* programming (c.f. pages 42-43). The combination of *data-abstraction* and the use of grammatical structures to reflect the *applicative structure*s of intended computations, as illustrated by this case study, is the essence of Language Oriented Programming.

```
(DEFUN EXPRESSION (E)
  (PROG (EXP X Y OP)
    (COND
      ((NULL E) (RETURN NIL))
      ((NULL (SETQ X (TERM E))) (RETURN NIL)))
    (SETQ EXP (CAR X))
    E
    (COND
      ((NULL (CDR X)) (RETURN EXP))
      ((EQ (CADR X) '#\+) (SETQ OP 'PLUS))
      ((EQ (CADR X) '#\-) (SETQ OP 'DIFFERENCE))
      (T (RETURN (CONS EXP (CDR X)))))
    (COND ((NULL (SETQ Y (TERM (CDDR X)))) (RETURN NIL)))
    (SETQ EXP (LIST OP EXP (CAR Y)))
    (SETQ X Y)
    (GO E)))
```

Figure 4.22: The Original Definition of `expression`

4.4 Approximating Roots

This section presents the initial design of a program to calculate the roots of differentiable functions using Newton's method. The program reuses parts of the symbolic differentiation program. Apart from illustrating the method of combining separate modules in META-LISP it also provides an example of a program that exploits the meta-programming capabilities of LISP.

4.4.1 Top-Level Elaboration of *newton*

Newton's method for finding the roots of a differentiable function $y = f(x)$ says that if x_k is an approximation to a root of the differentiable function f, then a better approximation, x_{k+1} can be obtained by the following *iteration*:

$$x_{k+1} = x_x - \Delta$$

where

$$\Delta = \frac{f(x_k)}{f'(x_k)}$$

Figure 4.23 shows the the top-level elaboration of the program. The program reads an expression and the independent variable used in the expression for $f(x)$ and reads in an initial guess. The program then translates the expression read into fully parenthesised prefix notation while constructing simultaneously an expression that represents its derivative. These expressions are then used to construct appropriate LISP functions that can then be used to compute both the value and the derivative of the function at given points. In doing so the program intends to "capitalise on the *pun* that an expression that describes the value of a function may also be interpreted as a means of computing that value" [ASS85, 335]. Given that the inherited attributes `^f` and `^df` denote these functions *calc-root* is then called with the initial guess to calculate the roots.

```
newton
 : inexpr invar inguess   = (^ f <-
                               [lambda
                                 [invar]
                                 (diff.expr . inexpr)])
                            (^ df <- [lambda [invar] d@diff.expr])
                            (calc-roots inguess)
```

Figure 4.23: Top-Level Elaboration of *newton*

4.4.2 The Main Body of the Program

In most cases there may be more than one roots to be calculated. Hence, calculating the roots will be repeated until the user quits the program by typing `q`. The supervisor for the program is *calc-roots* that carries out the iteration. The calculation of the root starts with one step of improving the initial guess. The calculation of successive improvements to the original guess is iterated until the value of Δ becomes sufficiently small.

Improve returns two values: the ratio of the value of the function for a given guess and the value of its derivative computed for the same guess (i.e. Δ); the second value is the improved guess, which is obtained as the difference between the original guess and Δ. Note that these values are computed by *applying* the LISP representation of these functions to appropriate argument.

Figure 4.24 show the META-LISP definitions for these functions. Note that *ratio* handles the error of attempting to divide by zero. The remaining definitions that complete the

```
calc-roots
 : quit
 : guess                  = (improve (float guess))
                            (show-root (iter delta@improve guess@improve))
                            (calc-roots (inguess))

iter
 : delta improved_guess   = (if (> (abs delta) 1.0E-10)
                                { (improve improved_guess)
                                  (iter delta@improve guess@improve) }
                              improved_guess)
improve
 : guess                  = (@ delta <-
                              (ratio
                                (apply ^f [guess])
                                (apply ^df [guess])))
                            (@ guess <- (difference guess delta@improve))

ratio
 : u zero             = (format t "~%Attempt division by zero~%") 0
 : u v                = (quotient u v)
```

Figure 4.24: Calculating Roots

program are shown in Figure 4.25. Note that *diff.expr* is imported from the module `diffpx`, which itself is made up from the components of the second, language oriented design of the symbolic differentiation program and a collection of abstract analysers and constructors that define prefix notation for algebraic expressions.

Issues of *robustness* and *adaptability* [DJ83, 102-106] were not even addressed in the design of the program. As a consequence it is prone to go into an infinite loop. Nevertheless, it has illustrated important points about program design in META-LISP:

- software reuse in META-LISP
- the power and convenience of exploiting the meta-programming capabilities of LISP
- the use of META-LISP in expressing numerical computations

A trace of the execution of the program is shown opposite.

```
inexpr
  : prompt-expr readl       = readl


 > : with lisp >

 abs : with lisp abs

 apply : with lisp apply

 inguess : <>              = (format t "Enter Initial Guess: ") (read)

 improved_guess : _

 delta : _

 difference : with lisp -

 diff.expr
  : with :diffpx diff.expr

 float : with lisp float

 guess : is numberp

 prompt-expr
  : <>                     = (format t " ;; Calculating Roots ~%Enter Formula : ")

 quit : any q quit

 quotient : with lisp /

 read : with lisp read

 root : _

 show-root
  : root                   = (format t "~%The root found is: ~S~%" root)
```

Figure 4.25: Elementary Definitions for *newton*

```
 Type ? for commands
0> newton : nil  ; l
 ;; Calculating Roots
Enter Formula : X^3-9X+4,
WITH RESPECT TO-
X,
Enter Initial Guess: 0
 <1 newton : (<- ^f)
    = (lambda (x) (+ (+ (^ x 3) (- (* 9 x))) 4))  ;
 <1 newton : (<- ^df)
    = (lambda (x) (+ -9 (* 3 (^ x 2))))  ;
 1> iter : (-0.4444444444444444 0.4444444444444444)  ;
  2> iter : (-0.010442160221895892 0.4548866046663403)  ;
   3> iter : (-1.7486507382534088E-5 0.4549040911737228)  ;
    4> iter : (-4.980058755359147E-11 0.45490409122352344)  ;
    <4 iter : (-4.980058755359147E-11 0.45490409122352344)
       = 0.45490409122352344  ;
   <3 iter : (-1.7486507382534088E-5 0.4549040911737228)
      = 0.45490409122352344  ;
  <2 iter : (-0.010442160221895892 0.4548866046663403)
     = 0.45490409122352344  ;
 <1 iter : (-0.4444444444444444 0.4444444444444444)
    = 0.45490409122352344  ;

The root found is: 0.45490409122352344
Enter Initial Guess: 3
 1> iter : (0.2222222222222222 2.7777777777777777)  ;
  2> iter : (0.0306379678107426 2.747139809967035)  ;
   3> iter : (5.713656040427687E-4 2.746568444362992)  ;
    4> iter : (1.9736725112545713E-7 2.746568246995741)  ;
     5> iter : (2.3717925759785976E-14 2.7465682469957176)  ;
     <5 iter : (2.3717925759785976E-14 2.7465682469957176)
        = 2.7465682469957176  ;
    <4 iter : (1.9736725112545713E-7 2.746568246995741)
       = 2.7465682469957176  ;
   <3 iter : (5.713656040427687E-4 2.746568444362992)
      = 2.7465682469957176  ;
  <2 iter : (0.0306379678107426 2.747139809967035)
     = 2.7465682469957176  ;
 <1 iter : (0.2222222222222222 2.7777777777777777)
    = 2.7465682469957176  ;

The root found is: 2.7465682469957176
Enter Initial Guess: -3
 1> iter : (0.2222222222222222 -3.2222222222222223)  ;
  2> iter : (-0.020562368388455415 -3.201659853833767)  ;
   3> iter : (-1.8750008829361548E-4 -3.2014723537454733)  ;
    4> iter : (-1.5526232439317245E-8 -3.201472338219241)  ;
     5> iter : (-3.2671222098523193E-16 -3.2014723382192405)  ;
     <5 iter : (-3.2671222098523193E-16 -3.2014723382192405)
        = -3.2014723382192405  ;
    <4 iter : (-1.5526232439317245E-8 -3.201472338219241)
       = -3.2014723382192405  ;
   <3 iter : (-1.8750008829361548E-4 -3.2014723537454733)
      = -3.2014723382192405  ;
  <2 iter : (-0.020562368388455415 -3.201659853833767)
     = -3.2014723382192405  ;
 <1 iter : (0.2222222222222222 -3.2222222222222223)
    = -3.2014723382192405  ;

The root found is: -3.2014723382192405
Enter Initial Guess: q
Done
<0 newton : nil
```

Figure 4.26: Tracing *newton*

Chapter 5

Programming in META-LISP II

The purpose of this Chapter is to extend further the basis of evaluating the potential of language oriented programming. It also serves the purpose of enabling direct comparison with Prolog.

Section 1 presents a small Prolog program for solving the "Water Container Puzzle" [Kow79, 75]. Section 2 describes a solution to this problem developed in META-LISP. Section 3 discusses the design of a program for the graphical display of parse-trees.

5.1 Path Finding

Any problem can be formulated as a path-finding problem:

> Given an initial state A, a goal state Z, and operators which transform one state into another, the problem is to find a path from A to Z. [Kow79, 75]

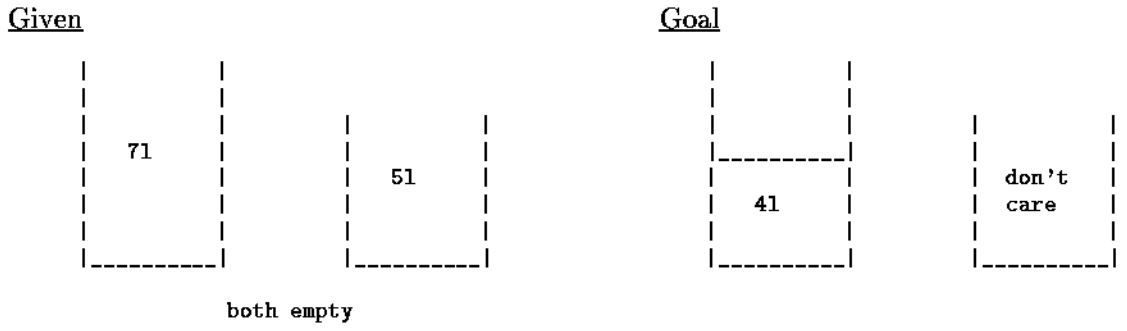


Figure 5.1: The Water-Container Puzzle

The Water-Container Puzzle can be formulated as a path-finding problem:

Given both a seven and a five litre container, initially empty, the goal is to find a sequence of actions which leaves four litres of liquid in the seven litre container. There are three kinds of actions which can alter the state of the containers:

1. A container can be filled.
2. A container can be emptied.
3. liquid can be poured from one container into the other, until the first is empty or the second is full.

Prolog is ideally suited to formulate these kinds of problems. The rules that govern the transition from one possible state to another can naturally be formulated as Prolog clauses. They are shown in Figure 5.2. In addition to defining the admissible state transitions these rules also associate a description of the action that these rules define.

The task of finding a path is simply left to the inference mechanism of Prolog. There are many solutions to the puzzle. In the course of finding solutions Prolog's inference mechanism will face non-deterministic choices. Backtracking to these choice points is automatic in Prolog. Figure 5.3 shows the part of the program that is responsible for the exploration of all possible solutions. The program constructs a list of visited states which is then used to avoid looping.

```
state(_,Y,7,Y," Fill up 7l container ").
state(X,_,X,5," Fill up 5l container ").

state(_,Y,0,Y," Empty 7l container ").
state(X,_,X,0," Empty 5l container ").

state(U,V,0,Y," Empty 7l container into 5l container ")
   :-  Y is U + V, Y =< 5.
state(U,V,X,0," Empty 5l container into 7l container ")
   :-  X is U + V, X =< 7.

state(U,V,7,Y," Pour from 5l container till 7l container is full ")
   :- Z is U + V, Z > 7, Y is Z - 7.
state(U,V,X,5," Pour from 7l container till 5l container is full ")
   :- Z is U + V,  Z > 5, X is Z - 5.
```

Figure 5.2: State Transitions in Prolog

Figure 5.4 shows a small portion of the output of the program.

```
g(X) :- go(0,0,X,_,[[0,0]],[ " Initial State "]).

go(X,_,X,_,S,P)
   :- reverse(S,RS),
      reverse(P,RP),
      show(RS,RP), nl,  write(--------), nl, fail.
go(X,Y,G,_,S,P)
   :- state(X,Y,U,V,W),
   \+(member([U,V],S)),
   \+(X=G),
   go(U,V,G,_,[[U,V]|S],[W|P]).

show([],[]).
show([S|T],[P|Q]) :- print(S), tab(1), printstring(P), nl, show(T,Q).

printstring([]).
printstring([H|T]) :- put(H), printstring(T).
```

Figure 5.3: The Water-Container Puzzle in Prolog

```
[0,0]  Initial State
[7,0]  Fill up 7l container
[7,5]  Fill up 5l container
[0,5]  Empty 7l container
[5,0]  Empty 5l container into 7l container
[5,5]  Fill up 5l container
[7,3]  Pour from 5l container till 7l container is full
[0,3]  Empty 7l container
[3,0]  Empty 5l container into 7l container
[3,5]  Fill up 5l container
[7,1]  Pour from 5l container till 7l container is full
[0,1]  Empty 7l container
[1,0]  Empty 5l container into 7l container
[1,5]  Fill up 5l container
[6,0]  Empty 5l container into 7l container
[6,5]  Fill up 5l container
[7,4]  Pour from 5l container till 7l container is full
[0,4]  Empty 7l container
[4,0]  Empty 5l container into 7l container
--------
[0,0]  Initial State
[7,0]  Fill up 7l container
[2,5]  Pour from 7l container till 5l container is full
[7,5]  Fill up 7l container
[0,5]  Empty 7l container
[5,0]  Empty 5l container into 7l container
[5,5]  Fill up 5l container
[7,3]  Pour from 5l container till 7l container is full
[0,3]  Empty 7l container
[3,0]  Empty 5l container into 7l container
[3,5]  Fill up 5l container
[7,1]  Pour from 5l container till 7l container is full
[0,1]  Empty 7l container
[1,0]  Empty 5l container into 7l container
[1,5]  Fill up 5l container
[6,0]  Empty 5l container into 7l container
[6,5]  Fill up 5l container
[7,4]  Pour from 5l container till 7l container is full
[0,4]  Empty 7l container
[4,0]  Empty 5l container into 7l container
--------
[0,0]  Initial State
[7,0]  Fill up 7l container
[2,5]  Pour from 7l container till 5l container is full
[0,5]  Empty 7l container
[5,0]  Empty 5l container into 7l container
[5,5]  Fill up 5l container
[7,3]  Pour from 5l container till 7l container is full
[0,3]  Empty 7l container
[3,0]  Empty 5l container into 7l container
[3,5]  Fill up 5l container
[7,1]  Pour from 5l container till 7l container is full
[0,1]  Empty 7l container
[1,0]  Empty 5l container into 7l container
[1,5]  Fill up 5l container
[6,0]  Empty 5l container into 7l container
[6,5]  Fill up 5l container
[7,4]  Pour from 5l container till 7l container is full
[0,4]  Empty 7l container
[4,0]  Empty 5l container into 7l container
--------
 ....

[0,0]  Initial State
[7,0]  Fill up 7l container
[2,5]  Pour from 7l container till 5l container is full
[2,0]  Empty 5l container
[0,2]  Empty 7l container into 5l container
[7,2]  Fill up 7l container
[4,5]  Pour from 7l container till 5l container is full
--------
```

Figure 5.4: Solutions to the Water-Container Puzzle

5.2 The Water-Container Puzzle in META-LISP

In contrast to the Prolog solution, in META-LISP, each individual state transition is formulated as an effective concept, i.e. a functional unit on its own right. The state of the containers is represented as a pair of integers x and y. The result of a state transition is formulated in terms of synthesised attributes, corresponding the the next state of the containers and a description of the transition that lead to this new state.

```
f7
 : x y  =  (@ x <- 7) (@ y) (@ step <- "Fill up 7l container")

f5
 : x y  =  (@ x) (@ y <- 5) (@ step <- "Fill up 5l container")

e7
 : x y  =  (@ x <- 0) (@ y) (@ step <- "Empty 7l container")

e5
 : x y  =  (@ x) (@ y <- 0) (@ step <- "Empty 5l container")

pe7
 : x y  = (if (>= 5 (+ x y))
              {(@ x <- 0)
               (@ y <- (+ x y))
               (@ step <- "Empty 7l container into 5l container")}
           fail!)

pe5
 : x y  = (if (>= 7 (+ x y))
              {(@ x <- (+ x y))
               (@ y <- 0)
               (@ step <- "Empty 5l container into 7l container")}
           fail!)

pf7
 : x y  = (if (> (+ x y) 7)
              {(@ x <- 7)
               (@ y <- (- (+ x y) 7))
               (@ step <- "Pour from 5l container till 7l container is full")}
           fail!)

pf5
 : x y  = (if (> (+ x y) 5)
              {(@ x <- (- (+ x y) 5))
               (@ y <- 5)
               (@ step <- "Pour from 7l container till 5l container is full")}
           fail!)
```

Figure 5.5: State Transitions in META-LISP

Given the individual state transition functions, solutions to the Water-Container Puzzle are formulated in META-LISP as follows: The program takes as input some path leading from the initial state to some current state. A path is represented as a list of visited states

and two integers that specify the last visited state. Extending the path involves an attempt to extend a path with a given state transition function. If the current state was such that a new state can be reached by the application of some transition function, then the program attempts to extend further the path from the new state onwards. Eventually the goal state will be reached. At that point the just completed path is printed. After which failure is returned. This failure then causes semantic backtracking at the level of *extend_path*, so that the next available, yet untried state transition is tried. Then a new attempt is made to extend the path from that point onwards. Note that if a newly reached state is not the goal state *go* will test if that state had been visited, and if it had been visited, then *go* will cancels the last state-transition, and again forces backtracking to consider the next available state-transition.

```
wcp
% @(#)wcp   1.5 12/27/91
 : initial_path = (extend_path . initial_path)

initial_path : <>  = [[[0 0 " Initial State"]] 0 0]


extend_path
 : path f7   ? (go path x@f7 y@f7 step@f7)
 : path f5   ? (go path x@f5 y@f5 step@f5)
 : path e7   ? (go path x@e7 y@e7 step@e7)
 : path e5   ? (go path x@e5 y@e5 step@e5)
 : path pe7  ? (go path x@pe7 y@pe7 step@pe7)
 : path pe5  ? (go path x@pe5 y@pe5 step@pe5)
 : path pf7  ? (go path x@pf7 y@pf7 step@pf7)
 : path pf5  ? (go path x@pf5 y@pf5 step@pf5)

go
 : path goal y step  =  (show_path . (reverse [[goal y step] . path]))
                        fail!
 : path x y step     = (if (visited? [x y] path)
                           fail!
                        (extend_path [[x y step] . path] x y))
show_path
 : $                 = (format t "~%~%--------")
 : [x y step] rest   =  (format t "~%[~A,~A] " x y)
                        (princ step)
                        (show_path . rest)
```

Figure 5.6: The Water-Container Puzzle in META-LISP

What this all amounts to is that in the META-LISP formulation of a path-finding problem, choice points for backtracking has to be set up explicitly by the programmer. This represents a great deal more effort. However, in the same way as in Prolog, there is a clearly identifiable programming idiom associated with these kinds of problems. Admittedly, the

```
goal : '4
 visited? : with listp member
 + : with lisp +
 - : with lisp -
 > : with lisp >
 >= : with lisp >=
 elem : _
 equal : with lisp equal
 format : with lisp format
 list : _
 path : _
 princ : with lisp princ
 rest : ._
 reverse : with lisp reverse
 sought : _
 step : _
 x : _
 y : _
```

Figure 5.7: Elementary Definitions in *wcp*

META-LISP idiom is a bit more difficult perhaps to grasp, for the first time, but can be just as effective. In fact, it can be argued, that the procedural interpretation of Prolog requires the programmer to visualise a very similar process, which cannot be said to be any simpler. Needless to say the output of the two programs is identical. Since it is much easier to do these things in META-LISP the output format was biased towards a form that Prolog can handle easier.

It is worth emphasising, in conclusion, that there is a lot to be said in favour of the META-LISP solution to inherently non-deterministic problems, as the Water-Container Puzzle. For such problems, it is required to set up choice points and backtracking explicitly. The advantage of this is that when there is no need for backtracking, there is no need to worry about how undesired backtracking can be pruned, as in Prolog.

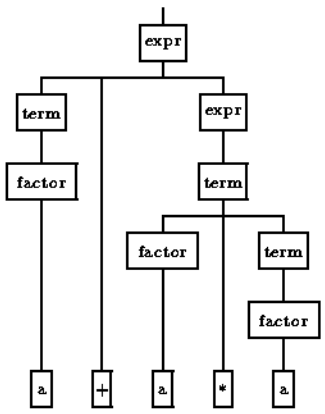
5.3 Parse Tree Printing

From the point of view of programming methodology the graphical display of parse-trees is interesting because it presents a clear example of a programming task that can best be thought of as a compilation task.

The task is simply the following. Given some representation of a parse tree, like below

```
("expr" ("term" ("factor" "a")) +
 ("expr"
  ("term" ("factor" "a") "*" ("term" ("factor" "a"))))))
```

display it in some suitable fashion. like this, say:



The analogy with compilation can be said to go deeper, than the idea that the meaning of a program is given in terms of instructions for a computer to execute. In fact, on reflection the need for some kind of "intermediate" code will become apparent.

The form that this inter-mediate code should take is influenced by the capabilities of the graphics primitives that are available. The parse-tree display program of this section can generate both instructions for the picture environment of LaTeX as well as instructions for GARNET. What really dictates the form of this intermediate representation is the logical dependencies between structural components of what we are trying to display. Furthermore, the requirement of being able to break up a parse tree into smaller ones, that can be fitted into a given display size, dictates many of the design decisions, and in fact makes the construction of an intermediate form unavoidable.

The details of these will not be given. However, Figure 5.8 illustrates the inter-mediate code for the parse-tree shown at the beginning of the section.

Figure 5.9 presents the generated LaTeX code. Figure 5.10 shows the top-level elabora-

```
= (:tree 180 50 (:node "expr" 26 20)
    (:branches
     (:tree 40 50 (:node "term" 27 20)
      (:branches
       (:tree 40 50 (:node "factor" 40 20)
        (:branches (:leaf (:node "a" 11 20))))))
     (:leaf (:node + 10 20))
     (:tree 110 50 (:node "expr" 26 20)
      (:branches
       (:tree 110 50 (:node "term" 27 20)
        (:branches
         (:tree 40 50 (:node "factor" 40 20)
          (:branches (:leaf (:node "a" 11 20))))
         (:leaf (:node "*" 10 20))
         (:tree 40 50 (:node "term" 27 20)
          (:branches
           (:tree 40 50 (:node "factor" 40 20)
            (:branches (:leaf (:node "a" 11 20))))))))))))
```

Figure 5.8: Parse-Tree Decorated with Display Information

tion of the translator into "intermediate" code. The heart of the first phase of the Parse-Tree display routine is the procedure that splits a tree into smaller trees if the whole tree would not fit into the available display size. It is this part of the program that would be difficult to formulate using standard compiler-compilers, say YACC. New YACC would stand a better chance of coping with this problem. But the burden of processing in that case would fall onto the rewrite rules over the parse-tree of the parse-tree. Its only in META-LISP that semantic processing can be specified as a continuum of syntactic elaborations.

```
\begin{tiny}
\setlength{\unitlength}{0.009in}%
\begin{picture}(500,240)(-40,-200)
                %w  h
\thinlines

\put(77.0,0){\framebox(26,20){expr}}
\put(90.0,20){\line(0,1){10}}
\put(20.0,-10){\line(1,0){105.0}} % BAR -
\put(90.0,-10){\line(0,1){10}} % BAR -
\put(6.5,-40){\framebox(27,20){term}}
\put(20.0,-20){\line(0,1){10}}
\put(20.0,-50){\line(1,0){0}} % BAR -
\put(20.0,-50){\line(0,1){10}} % BAR -
\put(0,-80){\framebox(40,20){factor}}
\put(20.0,-60){\line(0,1){10}}
\put(20.0,-90){\line(1,0){0}} % BAR -
\put(20.0,-90){\line(0,1){10}} % BAR -
\put(14.5,-200){\framebox(11,20){a}}
\put(20.0,-180){\line(0,1){10}}
\put(20.0,-180){\line(0,1){90}} % BAR -
\put(50,-200){\framebox(10,20){+}}
\put(55.0,-180){\line(0,1){10}}
\put(55.0,-180){\line(0,1){170}} % BAR -
\put(112.0,-40){\framebox(26,20){expr}}
\put(125.0,-20){\line(0,1){10}}
\put(125.0,-50){\line(1,0){0}} % BAR -
\put(125.0,-50){\line(0,1){10}} % BAR -
\put(111.5,-80){\framebox(27,20){term}}
\put(125.0,-60){\line(0,1){10}}
\put(90.0,-90){\line(1,0){70.0}} % BAR -
\put(125.0,-90){\line(0,1){10}} % BAR -
\put(70.0,-120){\framebox(40,20){factor}}
\put(90.0,-100){\line(0,1){10}}
\put(90.0,-130){\line(1,0){0}} % BAR -
\put(90.0,-130){\line(0,1){10}} % BAR -
\put(84.5,-200){\framebox(11,20){a}}
\put(90.0,-180){\line(0,1){10}}
\put(90.0,-180){\line(0,1){50}} % BAR -
\put(120,-200){\framebox(10,20){*}}
\put(125.0,-180){\line(0,1){10}}
\put(125.0,-180){\line(0,1){90}} % BAR -
\put(146.5,-120){\framebox(27,20){term}}
\put(160.0,-100){\line(0,1){10}}
\put(160.0,-130){\line(1,0){0}} % BAR -
\put(160.0,-130){\line(0,1){10}} % BAR -
\put(140.0,-160){\framebox(40,20){factor}}
\put(160.0,-140){\line(0,1){10}}
\put(160.0,-170){\line(1,0){0}} % BAR -
\put(160.0,-170){\line(0,1){10}} % BAR -
\put(154.5,-200){\framebox(11,20){a}}
\put(160.0,-180){\line(0,1){10}}
\put(160.0,-180){\line(0,1){10}} % BAR -
\end{picture}
\end{tiny}
```

Figure 5.9: LaTeX Code

```
c.ptree
 : c.node
  =  (@ x <- x@c.node)
     (@ y <- y@c.node)
     (@ trees <- [])
     (@ tree <- [:leaf c.node])
 : [c.node c.Branches]
  =  (@ tree <-
       (fit-tree
         (max x@c.node x@c.Branches)
         (+ ^y-label (* 1.5 ^y-sep)  y@c.Branches)
         c.node
         c.Branches ))
     (@ x <- x@fit-tree)
     (@ y <- y@fit-tree)
     (@ trees <- [. trees@fit-tree . trees@c.Branches])
     tree@c.ptree

c.node
 : node-or-label
  =  (@ node-or-label)
     (@ x <- (label-x node-or-label))
     (@ y <- ^y-label)
     [:node node-or-label x@c.node y@c.node]


c.Branches
 : []
  =  (@ x <- (* -1 ^y-sep))
     (@ y <- (* -1 ^y-sep))
     (@ trees <- [])
 : c.ptree c.Branches
  =  (@ x <- (+ x@c.ptree ^y-sep x@c.Branches))
     (@ y <- (max y@c.ptree y@c.ptree))
     (@ trees <- [. trees@c.ptree . trees@c.Branches])
     [tree@c.ptree . c.Branches]
```

Figure 5.10: Calculating the Dimensions of a Tree

```
fit-tree
 : x y DNode _branches
  = (if (fits x )
        { (@ x )
          (@ y )
          (@ trees <- [])
          (@ tree <- [:tree x y DNode [:branches . _branches]]) }
      { (if (pred (elided? (xtract-max . _branches)))
            { (c.ptree [(node-of DNode) ??????])
              (@ x <- x@c.ptree)
              (@ y <- y@c.ptree)
              (@ trees <- [])
              (@ tree <- tree@c.ptree) }
            { (fit-tree
                (+ (- x (x-of max@xtract-max)) (x-of-label (node-of max@xtract-max)))
                (+ ^y-label (* 1.5 ^y-sep) (adjust-y position@xtract-max  _branches))
                DNode
                (subst-nth (mk-stub  n) position@xtract-max _branches)
                )
              (@ x <- x@fit-tree)
              (@ y <- y@fit-tree)
              (@ trees <- [. trees@fit-tree (mark-tree n max@xtract-max)])
              (@ tree <- tree@fit-tree) }) })
```

Figure 5.11: Fitting a Tree into Displays

Chapter 6

Denotational Semantics in Meta-Lisp

Denotational semantics is a methodology for defining the mathematical meaning of programming languages and systems. The essence of denotational definitions is that they allow the specification of the meaning of the phrases of a language in terms of functions defined over the meanings, or *denotations*, of their component phrases. For the purpose of denotational definitions only the phrase structure of the syntax of a language is of interest. The rules that are used to describe the phrase structure of a language constitute the *abstract syntax* of a language. The sets that are used as value spaces in programming language semantics are called *semantic domains*. The meaning, or denotation of abstract syntax structures of a language are drawn from these domains. The mapping of the abstract syntax structures of a language to their denotations are given in terms of *valuation functions*. In specifying the meaning of individual constructs of a language, the valuation functions make use of functions over semantic domains. These functions and their associated domains are normally presented together in the form of *semantic algebras*.

One advantage of denotational definitions is that it is possible to derive language processors, such as compilers or interpreters, directly from their denotational definitions. The approach to the construction of compilers for a language from its denotational definition is known as the *compile-evaluate* method. [Sch86, 217]. According to this method, the valuation functions are used as specification for a translator of source programs into their denotation in terms of complex functional expressions. A separate *evaluation* phase is then used to obtain the output of the program for given actual input. It is also possible to translate a program to its denotation and evaluate it simultaneously with its run time arguments. The result is an interpreter for the language derived from its denotational semantics.

A number of systems have been developed to serve as tools for deriving language processors directly from a denotational definition of their semantics. The first *semantics directed* compiler generating system based on denotational semantics developed was Mosses's Semantic Implementation System (SIS). See [Mos79]. It uses the compile-evaluate method. The "machine-code" is a simple functional language, called LAMB, based on the lambda calculus. SIS has been used to implement its own languages: the Denotational Semantics Language (DSL), and the language GRAM, used for dealing with syntax matters. The mapping from DSL into LAMB is itself described in DSL. SIS is *bootstrapped*, very much the way META-LISP has been implemented.

Wand's Semantic Prototyping System consists of a set of programs for testing and exercising denotational style language specifications. The system is built largely in Scheme (a dialect of LISP), and is used to serve as an efficient lambda-calculus interpreter. The systems parser generator is YACC. The denotational semantic equations, coded in Scheme, are appended to the YACC grammar rules. SPS uses a type checker that validates the domain of definitions and semantic equations for well-definedness.

Paulson's Semantic Processor (PSP) system is a semantics directed compiler generator that generates stack machine code. See [Pau84]. The semantic grammar notation used to define a language is a hybrid of denotational semantics and attribute grammars.

These systems offer distinct advantages. Using them can ensure the correctness of language implementations. They allow experimentation in the design of new languages. These experiments can also help in "debugging" formal language descriptions themselves.

A common feature of these systems is that they enable the specification of both the concrete and the abstract syntax of a language, together with the specification of the evaluation rules in a form that can be interpreted to produce appropriate denotations or actual values. All these tasks can be readily specified in META-LISP. The mapping from concrete syntax of a language to its abstract syntax is a task for which META-LISP is eminently suitable. The mapping of abstract syntax structures to their denotation, using valuation functions, can equally well be described in META-LISP.

The aim of this Chapter is to illustrate how META-LISP can be used as the vehicle of writing denotational language definitions. The denotational definition of the language of a simple Calculator will be developed alongside the description of a denotational style interpreter for it in META-LISP. The purpose of adopting this mode of presentation is to emphasise the close correspondence between the two formulations. It also serves the purpose of introducing the format of denotational definitions in META-LISP. The same format will be used in the next Chapter in defining the semantics of META-LISP itself.

6.1 The Calculator

The following description of a simple calculator and the denotational definition of the semantics of the language that it accepts is based on Chapter 4 of the book on denotational semantics by David A. Schmidt [Sch86].

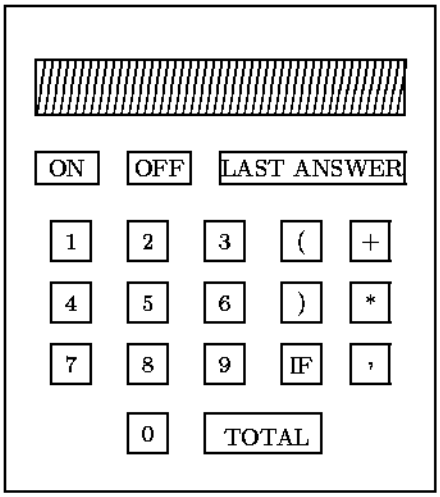


Figure 6.1: The Calculator

Expressions in the language of the calculator can be entered by pressing buttons on the device shown in Figure 6.1. The output appears on a display screen. The calculator can carry out addition and multiplication. It can recall the value of the last calculation. It also allows the user to enter a form of if-then-else expression. A session with the calculator might go:

```
press   ON
press   ( 4 + 1 2 ) * 2
press   TOTAL                       (the calculator prints 32)
press   1 + LAST
press   TOTAL                       (the calculator prints 33)
press   IF LAST + 1 , 0 , 2 + 4     (the second branch of the conditional is taken)
press   TOTAL                       (the calculator prints 6)
press   OFF
```

Figure 6.2: Example Session with the Calculator

The *denotational definition* of the semantics of a language consists of three parts:

- the definition of the *abstract syntax* of the language.
- The specification of the sets and operations used to specify the meaning of the phrases

of the language. These are usually given in the form of *semantic algebras.*

- The specification of the *valuation functions* which map the abstract syntax structures of the language to their meanings drawn from semantic domains.

The above format of denotational definitions will be reflected in the organisation of the remainder of the Chapter. For the purposes of the present discussion the "Calculator" will be identified with its input language.

6.2 Syntax of the Calculator Language

Figure 6.3 gives the abstract syntax of the language of the Calculator.

P ∈ Program
S ∈ Expr-sequence
E ∈ Expression
N ∈ Numeral
D ∈ Digit

P	::=	ON S	(1)
S	::=	E **TOTAL** S	(2)
	|	E **TOTAL OFF**	(3)
E	::=	E_1 + E_2	(4)
	|	E_1 * E_2	(5)
	|	**IF** E_1 , E_2 , E_3	(6)
	|	**LAST**	(7)
	|	(E)	(8)
	|	N	(9)
N	::=	N D	(10)
	|	D	(11)
D	::=	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9	(12)

Figure 6.3: Abstract Syntax of the Calculator

The abstract syntax indicates that a session with the calculator starts by pressing the "ON" key which is followed by entering an expression sequence. An *expression sequence* consists of one or more expressions separated by a single occurrence of pressing the "TOTAL" key. The expression sequence, as well as the session with the calculator, is terminated by pressing the "OFF" key. The syntax for an expression specifies only the use of the operators for addition and multiplication. It also allows a limited form of choice function, parentheses and "recall" of the last answer.

For the purposes of providing a denotational definition of the semantics of a language a mapping from the concrete to the abstract syntax of the language is assumed. Since the purpose of developing a denotational style definition in META-LISP for the language of the Calculator is to be able to derive an interpreter from its denotational definition, it is necessary to supply an appropriate mapping from the concrete to the abstract syntax. This involves the construction of the following programs:

1. a reader and lexical analyser for the language (*calc-lex*)
2. a translator of the concrete syntax of tokens of the language to a suitable internal representation of the abstract syntax of the language (*calc-c2a*)
3. A parser for the internal representation of the abstract syntax (*calc-abst*).

The following subsections will introduce these routines.

6.2.1 Lexical Analysis

The construction of a reader/lexical analyser is a routine task. The aim is to translate the stream of input characters into tokens of the language of the calculator. Tokens will be represented as strings, with the exception of numerals from 0 to 9. Figure 6.4 shows the input and the corresponding output of the lexical analyser on the example session introduced earlier.

The following *lexical* conventions are enforced by the *lexical analysis* routine:

- the keywords of the language (`ON, TOTAL, IF, LAST`) as well as the reserved symbols (`'+', '*', '(', ')', ','`) are represented as strings,
- *Digits* are represented as numbers from 0 to 9,

The LISP reader is used this time, once the special characters such as *comma*, have been read from the input. Straightforward modification of the reader routine of the symbolic differentiation program presented in Chapter 4 (see 4.2.4 on page 72) will do the job.

```
| ?= (calc-lex)

 ON
 ( 4 + 1 2 ) * 2
 TOTAL
 1 + LAST
 TOTAL
 IF LAST + 1 , 0 , 2 + 4
 TOTAL
 OFF

|
|------------
|

calc-lex = ("ON" "(" 4 "+" 1 2 ")" "*" 2 "TOTAL" 1 "+" "LAST" "TOTAL"
            "IF" "LAST" "+" 1 "," 0 "," 2 "+" 4 "TOTAL" "OFF")
```

Figure 6.4: Lexical Analysis of the Example Session

White Spaces will be skipped, Special characters, such as comma and opening and closing parenthesis will be converted into strings. Any other input is read by the LISP `read` function and, with the exception of numerical input, will be converted into tokens represented as strings. The META-LISP code for this is shown in Figure 6.5.

Relying on the LISP reader may result in an error for incorrect input. The lexical analyser could be made robust by supplying a suitable definition of *read-item*, in META-LISP, to handle erroneous input, as has been done in the Symbolic Differentiation Program. However, introduction of these refinements would not be pertinent to the present discussion. The use of the LISP reader to process well-formed input is sufficient for the present purpose.

6.2.2 Concrete to Abstract Syntax

Devising an appropriate description of the concrete syntax of the Calculator Language is fairly straightforward. Care need only be taken to respect the usual rules of precedence and associativity of the arithmetic operators involved. The question of what concrete representation to use for the abstract syntax of the Calculator language is less straightforward. One possible approach is to generate a parse tree and use that as the concrete representation of the abstract syntax. Note, however, that not all productions that are used in deriving a sentence of the language are to be reflected in the parse tree representation, if it is to be used as the concrete representation of the abstract syntax. The advantage of this approach is that in developing the mapping from concrete to abstract syntax the parse tree form can

```
calc-lex
 : <>                 = (readlh (peek) [])

readlh
 : skip line          = (read-char) (readlh (peek) line)
 : special line       =  (read-char)
                         (readlh (peek) [(string special) . line])
 : read line          = (if (equal "OFF" read)
                            (reverse ["OFF" . line])
                          (readlh (peek) [read . line]))

peek : with lisp peek-char

skip : any #\Space #\Newline

line : _

special : any #\, #\( #\)

read : _               = (mk-string (read-item))

read-item : with lisp read

mk-string
 : is fixp
 : _                   = (string _)

string : with lisp string

read-char : with lisp read-char

equal : with lisp equal

reverse : with listp reverse
```

Figure 6.5: Lexical Analyser for the Calculator Language

readily be visualised in a way that makes the structures of interest apparent. The disadvantage of the latter method is that writing a grammar for the internal representation of the abstract syntax will be cluttered with the names of the phrases of the language being represented. An alternative approach is to represent composite phrases as lists formed of their components. In this way the phrase structure of the language will be readily identifiable, with very little clutter (e.g. brackets around composite phrases). The disadvantage of this representation is that it is not as easy to make out the phrase structure from nested list structures as it is from parse-trees.

The strategy adopted here combines the advantages of both approaches, by constructing the definition of the abstract syntax of the language of the calculator in two steps. First a parser is written for the language represented as tokens. The parser is written in such a

way that it can generate either a parse tree or a list structure representation of the phrase structure of the language. The parse tree form is used to validate the parser. Figure 6.7 shows the META-LISP definition of the parser of stage 1. Figure 6.8 shows the definition of the abstraction function *abst* which is used to construct alternatively subtrees or the corresponding list structure representation. The list structure representation of the phrase structure of the language is then used in the second stage. In the second stage, a grammatical description of the list structure representation of the phrase structure of the language is formulated. This, in effect, constitutes a META-LISP specification of *the* abstract syntax of the language. Similarly to the parser for the concrete syntax of the language, the parser for the abstract syntax representation is written in such a way that it can generate either a parse tree (which will be in accordance with the abstract syntax, an abstract parse tree) or again a list structure representation of the abstract syntax. That is to say, it defines an identity transformation on the list structure representation of the abstract syntax of the language. Figure 6.9 shows the concrete derivation tree of the example session, as produced by the parser for the Calculator Language. Figure 6.6 show the corresponding list structure representation of the abstract syntax. Figure 6.11 shows the definition of the abstract syntax of the Calculator Language in META-LISP. Figure 6.10 shows the abstract parse tree for the example session.

```
("ON"
 ((("("
    (4 "+" (1 2)) ")") "*" 2)
  "TOTAL"
  ((1 "+" "LAST")
   "TOTAL"
   (("IF" ("LAST" "+" 1) "," 0 "," (2 "+" 4))
    "TOTAL"
    "OFF"))))
```

Figure 6.6: Internal Representation of the Abstract Syntax

Comparison of the Abstract Parse tree with its concrete counterpart reveals the branching structure of the two trees to be identical. It is in this sense, that the abstract tree can be said to identify its concrete counterpart. In the abstract syntax tree only those rules in the derivation of a sentence of the language that have semantic significance are reflected. Only those rules in the grammar are considered semantically significant for which there are valuation rules.

```
calc-c2a
 : tokens tree                = (^ tree) (P . tokens)

P
 : "ON" S                     = (abst 'P "ON" S)

S
 : E "TOTAL" "OFF"            = (abst 'S E "TOTAL" "OFF")
 : E "TOTAL" S                = (abst 'S E "TOTAL" S)

E
 : E "+" T                    = (abst 'E E "+" T)
 : T                          = (abst 'E T)

T
 : T "*" F                    = (abst 'T T "*" F)
 : F                          = (abst 'T F)

F
 : "LAST"                     = (abst 'F "LAST")
 : "(" E ")"                  = (abst 'F "(" E ")")
 : "IF" E1  "," E2  "," E3    = (abst 'F "IF" E1 "," E2 "," E3 )
 : N                          = (abst 'F N)

E1
 : E                          = (abst 'E1 E)

E2
 : E                          = (abst 'E2 E)

E3
 : E                          = (abst 'E2 E)

N
 : N D                        = (abst 'N N D)
 : D                          = (abst 'N D)

D
 : any 0 1 2 3 4 5 6 7 8 9    = (abst 'D  any)
```

Figure 6.7: Parser for the Calculator Language

```
abst
 : identifier ._              = (if ^tree
                                    [identifier . ._]
                                  (mk-unit ._))

identifier
 : is identifier

mk-unit
 : [_]                      = _
 : _
```

Figure 6.8: Abstraction Function

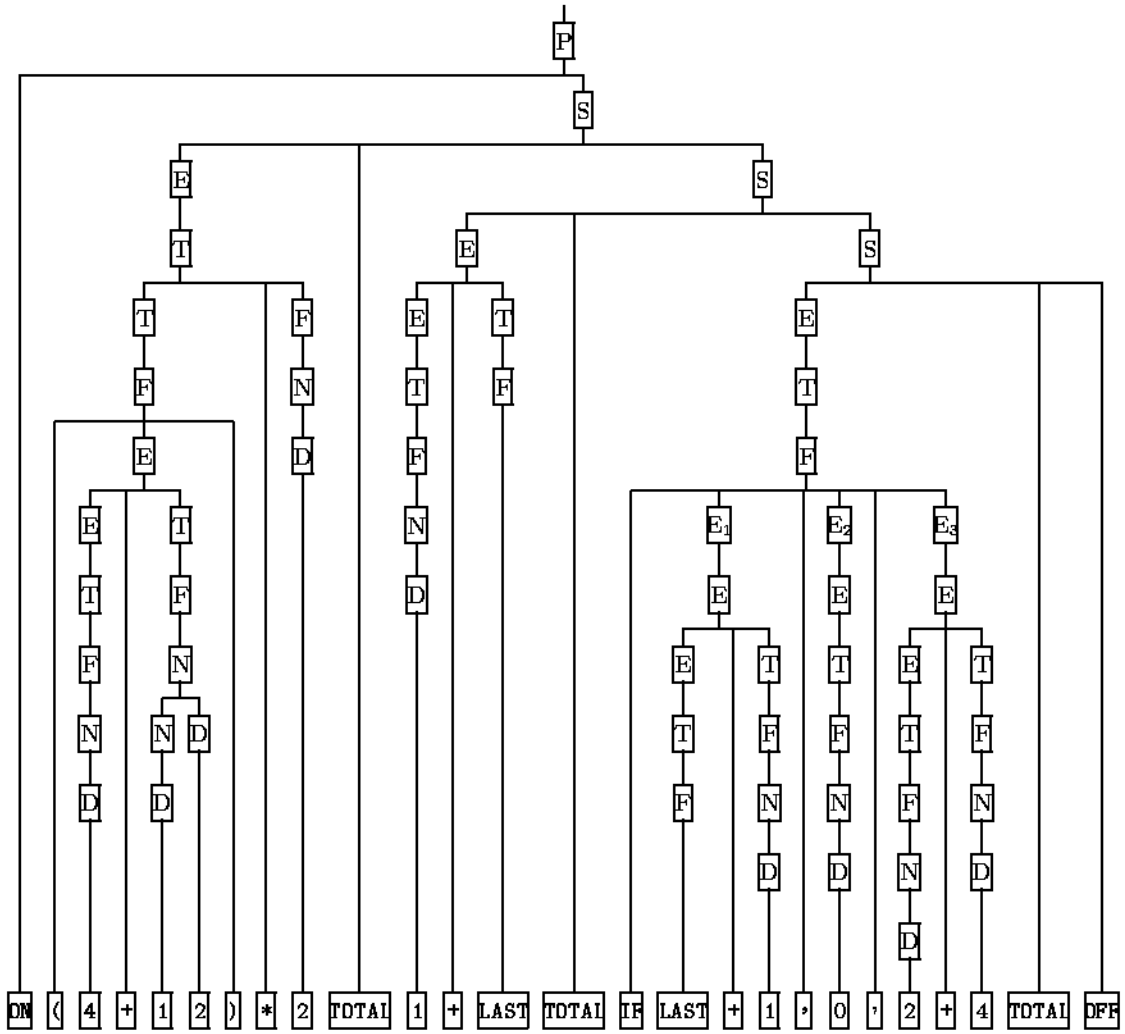


Figure 6.9: A Concrete Derivation Tree

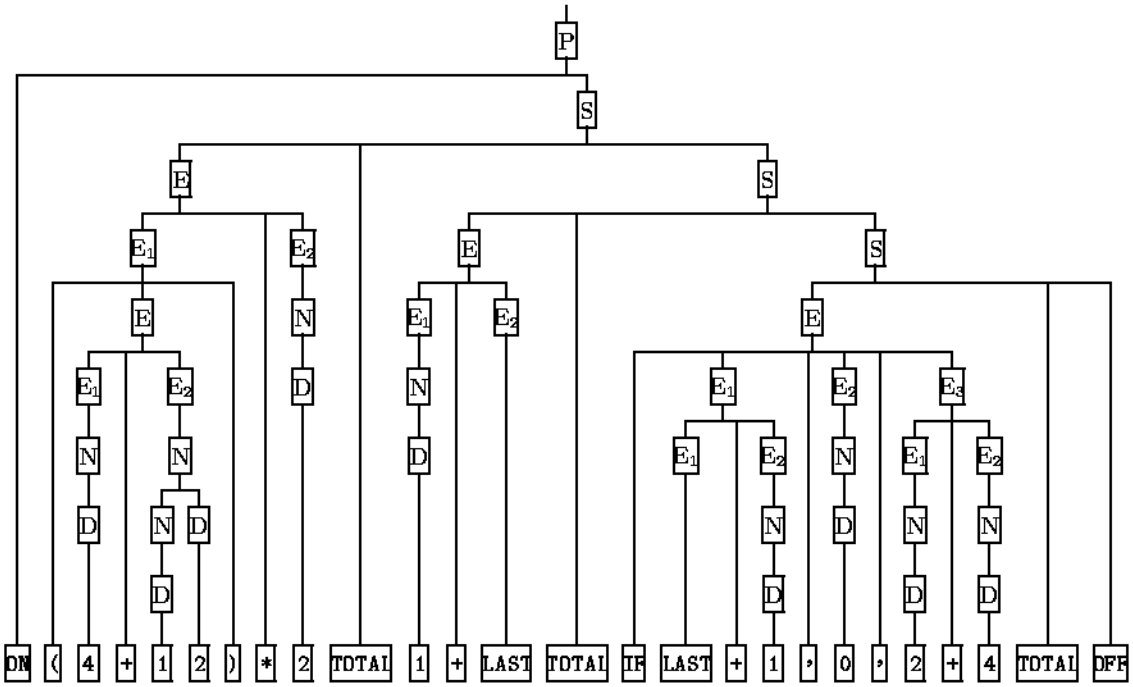


Figure 6.10: An Abstract Derivation Tree

```
calc-abst
  : p tree                    = (^ tree) (P p)

 P
  : ["ON" s]                  = (abst 'P "ON" (S s))

 S
  : [e "TOTAL" "OFF"]         = (abst 'S (E e) "TOTAL" "OFF")
  : [e "TOTAL" s]             = (abst 'S (E e) "TOTAL" (S s))

 E
  : [e1 "+" e2]               = (abst 'E (E e1) "+" (E e2))
  : [e1 "*" e2]               = (abst 'E (E e1) "*" (E e2))
  : ["IF" e1 "," e2 "," e3]   = (abst 'E "IF" (E e1) "," (E e2) "," (E e3))
  : "LAST"                    = (abst 'E "LAST")
  : ["(" e ")"]               = (abst 'E "(" (E e) ")")
  : n                         = (abst 'E (N n))

 N
  : [n d]                     = (abst 'N (N n) (D d))
  : d                         = (abst 'N (D d))

 D
  : any 0 1 2 3 4 5 6 7 8 9   = (abst 'D  any)
```

Figure 6.11: Abstract Syntax in META-LISP

All the effective concepts *p, s, e, e1, e2, e3, n* and *d* are just place-holders, defined to accept input at their position. For ease of comparison the BNF rules for describing the abstract syntax of the Calculator Language are reproduced below:

P	::=	ON S	(1)
S	::=	E `TOTAL` S	(2)
	|	E `TOTAL OFF`	(3)
E	::=	E_1 `+` E_2	(4)
	|	E_1 `*` E_2	(5)
	|	`IF` E_1 `,` E_2 `,` E_3	(6)
	|	`LAST`	(7)
	|	`(` E `)`	(8)
	|	N	(9)
N	::=	N D	(10)
	|	D	(11)
D	::=	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9	(12)

6.3 Semantic Algebras

The sets that are used as value spaces in programming language semantics are called *semantic domains.* Semantic domains are really structured sets, but for most situations their structure, whether it be lattices or topologies, can be ignored. Accompanying a domain is a set of *operations.* These are functions that take arguments from the domain to produce results. An operation is specified in two parts. First, the operation's *functionality* is given describing the domains from which it draws its arguments, and its codomain, which is the domain from which the result of the operation is drawn. For an operation f its functionality $f : D_1 \times D_2 \times \cdots \times D_n \rightarrow A$ says that f takes an argument from domain D_1 and one from D_2, ..., and one from D_n to produce a result in domain A. Second, a description of the operation's mapping is specified, usually in the form of equations. For operations which are considered semantically primitive, the defining equations are usually omitted.

Primitive domains are sets that are fundamental to the application being considered. In defining the semantics of the Calculator two primitive domains are used: booleans and natural numbers. The semantic domains and the functionality of the operations used in the definition of the Calculator are given in Figure 6.12.

I. Truth values
  Domain $t \in Tr = \mathbf{B}$
  Operations
    true, false: Tr
II. Natural Numbers
  Domain $n \in Nat$
  Operations
    $zero,\ one,\ two, \cdots : Nat$
    $plus, times : Nat \times Nat \rightarrow Nat$
    $equals : Nat \times Nat \rightarrow Tr$

Figure 6.12: Semantic Algebras

The valuation functions given in the next section make use of the *choice function* and the operation $cons : A \times A^* \rightarrow A^*$ for constructing lists. The choice function is the usual conditional expression: $e_1 \rightarrow e_2 [] e_3$, which has as its value e_2 if $e_1 = true$ and e_3 if $e_1 = false$.

In the META-LISP formulation of the semantics of a language semantic algebras are replaced by their concrete implementation. Thus, for example, the choice function is a primitive of META-LISP, with the usual semantics as in LISP. For the operations on natural numbers, such as *times* and *sum*, the built in functions of LISP, * and +, will be imported in the usual manner. These will operate on LISP's representation of natural numbers. For more complex semantic algebras appropriate META-LISP definitions may be used.

6.4 Valuation Functions

There are five valuation functions for the language of the Calculator. There is one valuation function corresponding to each abstract syntax domain. These are shown in Figure 6.13. The definition of a valuation function for an abstract syntax domain includes the description of its functionality and a number of rules corresponding to the alternative constructs that appear in the definition of the abstract syntax domain.

P: Program $\rightarrow Nat^*$
P⟦ON S⟧ = **S**⟦S⟧($zero$)

S: Expr-sequence $\rightarrow Nat \rightarrow Nat^*$
S⟦E TOTAL S⟧(m) = let m' = **E**⟦E⟧(m) in m' $cons$ **S**⟦S⟧(m')
S⟦E TOTAL OFF⟧(m) =**E**⟦E⟧(m) $cons$ nil

E: Expression $\rightarrow Nat \rightarrow Nat$
E⟦E_1+E_2⟧(m)=**E**⟦E_1⟧(m) plus **E**⟦E_2⟧(m)
E⟦E_1*E_2⟧(m)=**E**⟦E_1⟧(m) times **E**⟦E_2⟧(m)
E⟦IF E_1,E_2,E_3⟧(m)=**E**⟦E_1⟧(m) $equals$ $zero$ $\rightarrow$ **E**⟦E_2⟧(m) [] **E**⟦E_3⟧(m)
E⟦LAST⟧(m) = m
E⟦(E)⟧(m) =**E**⟦E⟧(m)
E⟦N⟧(m) =**N**⟦N⟧

N: Numeral $\rightarrow Nat$
N⟦N D⟧=(**N**⟦N⟧ $times$ ten) $plus$ **D**⟦D⟧
N⟦D⟧ = **D**⟦D⟧

D: Digit $\rightarrow Nat$
D⟦0⟧= $zero$
D⟦1⟧= one
⋮
D⟦9⟧= $nine$

Figure 6.13: Valuation Functions

Figure 6.14 shows the valuation rules in META-LISP. In what follows each rule will be discussed in some detail, to build up an understanding of both formulations.

```
calc-int
 : p                          = (P p)

P
% Program -> (list Nat)
 : ["ON" s]                   = (S s 0)

S
% Expr-Sequence  -> Nat -> (list Nat)
 : [e "TOTAL" "OFF"]          m = [(E e m)]
 : [e "TOTAL" s]              m = (@ m~ <- (E e m))
                                  [m~@S . (S s m~@S)]

E
% Expr -> Nat -> (list Nat)
 : [e1 "+" e2]                m = (plus (E e1 m) (E e2 m))
 : [e1 "*" e2]                m = (times (E e1 m) (E e2 m))
 : ["IF" e1 "," e2 "," e3]    m = (if (equals 0 (E e1 m))
                                      (E e2 m)
                                    (E e3 m))
 : "LAST"                     m = m
 : ["(" e ")"]                m = (E e m)
 : n                          m = (N m)

N
% Numeral -> Nat
 : [n d]                        = (plus (times 10  (N m)) (D d))
 : d                            = (D d)

D
% Digit -> Nat
 : any 0 1 2 3 4 5 6 7 8 9      = any

plus
% Nat * Nat -> Nat
 : with lisp +

times
% Nat * Nat -> Nat
 : with lisp *

equals
% Nat * Nat -> Bool
 : with lisp equal
```

Figure 6.14: Denotational Semantics of the Calculator in META-LISP

All the effective concepts *p, s, e, e1, e2, e3, n* and *d* are just place-holders, defined to accept input at their position.

6.4.1 Program

There is only one equation to describe the meaning of a Program, as there is only one rule for describing its abstract syntax:

$$\mathbf{P}[\![\text{ON S}]\!] = \mathbf{S}[\![\text{S}]\!](zero)$$

The function **P** maps a program to its meaning, which is a non-empty list of natural numbers. This list represents the sequence of outputs displayed by the calculator during a session. It is obtained as the value of the Expression Sequence, $[\![\text{S}]\!]$, as created by the valuation function **S**. The functionality of **S** states that it is a mapping from an expression sequence, $[\![\text{S}]\!]$, and a natural number, (m), to a non-empty list of numbers. The extra numeric argument is used to make available the value of the most recently evaluated expression which is stored in the calculator's memory. The fact that the initial value of the memory cell is zero is also expressed by the above equation.

The META-LISP formulation of a valuation function associated with a syntax domain defines an *interpreter* for some construct of the language being defined. The formulation of a valuation function for a given abstract syntax domain, in META-LISP, is based on the META-LISP definition of its abstract syntax. Recall the definition in META-LISP of the abstract syntax domain of Programs:

```
P
 : ["ON" s]                    = (S s)
```

Assuming that S will now be a mapping from the abstract syntax representation of Expression Sequence and a *Nat* for the memory cell, to list of natural numbers, we can express in META-LISP the first valuation rule as follows:

```
P
% Program -> (list Nat)
 : ["ON" s]                    = (S s 0)
```

It is easy to see that this defines the same meaning as the equation for **P** given before. The intended functionality of this evaluation rule is given as a comment.

6.4.2 Expression Sequence

There are two rules that describe the meaning of Expression Sequences. The first describes the meaning of a sequence of two or more expressions. The second one applies when there is only one expression left to evaluate before the Calculator is turned off. The functionality of **S** indicates that the value of an Expression Sequence is calculated using the value of

the memory cell. The first equation states that the value of a sequence of two or more expressions can be given by constructing a list which has as its first element the meaning of the first expression, ⟦E⟧, and the rest of the list is obtained as the meaning of the remaining sequence of expressions. The corresponding actions of the calculator can be enumerated as follows:

1. Evaluate ⟦E⟧ using cell m, producing value m'.
2. Print out m' on the display.
3. Place m' into the memory cell.
4. Evaluate the rest of the sequence ⟦S⟧ using the cell.

Note how each of these four steps are represented in the semantic equation:

1. is handled by the expression **E**⟦E⟧(m).
2. is handled by the expression m' *cons* ....
3. and 4. are handled by the expression **S**⟦S⟧(m').

The equation for **S**⟦E TOTAL S⟧ in META-LISP can be given as follows:

```
: [e "TOTAL" s] m          =  (@ m~ <- (E e m))
                              [m~@S . (S s m~@S)]
```

The correspondence between the two formulations is summarised below:

⟦E TOTAL S⟧(m)	[e "TOTAL" s] m
let m' = **E**⟦E⟧(m) in	(@ m~ <- (E e m))
m' *cons* **S**⟦S⟧(m')	[m~@S . (S s m~@S)]

The meaning of **S**⟦E TOTAL OFF⟧ is similar. Since ⟦E⟧ is the last expression to be evaluated, the list of subsequent outputs is just *nil*. The definition of S, then, is as follows:

```
S
% Expr-Sequence  -> Nat -> (list Nat)
 : [e "TOTAL" "OFF"]       m = [(E e m)]
 : [e "TOTAL" s]           m =  (@ n <- (E e m))
                                [m~@S . (S s m~@S)]
```

Note, that the ordering of the rules is important. Reversing it would mean that OFF would be accepted as a Sequence, which is not correct. Alternatively, a new definition of s could be used:

```
s
 : "OFF" = fail!
 : _
```

6.4.3 Expressions

The equations for addition, $\mathbf{E}[\![E_1{+}E_2]\!]$, and multiplication, $\mathbf{E}[\![E_1{*}E_2]\!]$, are straightforward. Their transcription to META-LISP poses no difficulties. All that is needed to be borne in mind is that `plus` and `times` are semantically primitive operations, which are imported from LISP:

```
: [e1 "+" e2]              m = (plus (E e1 m) (E e2 m))
: [e1 "*" e2]              m = (times (E e1 m) (E e2 m))
```

The semantic equation for $[\![\text{IF } E_1,E_2,E_3]\!]$ states that its meaning is given in terms of the conditional. The conditional is a primitive of META-LISP, with analogous semantics to the built in function `if` of LISP. The test for equality is imported from LISP. With these operations as semantic primitives the equation can be written in META-LISP as:

```
: ["IF" e1 "," e2 "," e3]  m = (if (equals 0 (E e1 m))
                                   (E e2 m)
                                 (E e3 m))
```

The remaining three equations describe very simple interpretations: [[LAST]] operator causes a lookup of the value in the memory cell; [[(E)]] specifies that the value of an expression in parentheses is the value of the enclosed expression; and finally, the value of a Numeral is given by its valuation function. Collecting all these rules and the ones previously discussed together gives the following definition of **E** in META-LISP:

```
E
% Expr -> Nat -> (list Nat)
 : [e1 "+" e2]              m = (plus (E e1 m) (E e2 m))
 : [e1 "*" e2]              m = (times (E e1 m) (E e2 m))
 : ["IF" e1 "," e2 "," e3]  m = (if (equals 0 (E e1 m))
                                    (E e2 m)
                                  (E e3 m))
 : ["LAST"]                 m = m
 : ["(" e ")"]              m = (E e m)
 : n                        m = (N n)
```

6.4.4 Numerals

All that remains is to discuss how numerals are mapped into natural numbers. Recall the original equations:

N: Numeral $\rightarrow$ *Nat*
N[[N D]]=(**N**[[N]] *times ten*) *plus* **D**[[D]]
N[[D]] = **D**[[D]]

The first equation for **N** can be understood as saying that, if N is a numeral which denotes the number, $x = \mathbf{N}[\![N]\!]$, and D is a digit which denotes a number $y = \mathbf{D}[\![D]\!]$,then N with

that digit D appended to the right of it denotes the number obtained by multiplying x by ten and adding y to it. The second equation states simply that the meaning of a numeral comprising a single digit is given in terms of the meaning of that digit.

Transcribing these rules into META-LISP is straightforward:

```
N
% Numeral -> Nat
 : [n d]                        = (plus (times 10  (N m)) (D d))
 : d                            = (D d)

D
% Digit -> Nat
 : any 0 1 2 3 4 5 6 7 8 9      = any
```

It is instructive to determine the meaning of the keypad sequence `1 2 3` using these evaluation rules:

N⟦1 2 3⟧ = (**N**⟦1 2⟧ times ten) plus **D**⟦3⟧
= (((**N**⟦1⟧ times ten) plus **D**⟦2⟧) times ten) plus **D**⟦3⟧
= (((**D**⟦1⟧ times ten) plus **D**⟦2⟧) times ten) plus **D**⟦3⟧
= (((one times ten) plus two) times ten) plus three
= one hundred and twenty three

The corresponding trace of the META-LISP program is shown in Figure 6.15. In addition, Figure 6.16 shows a detailed trace of the interpretation of the the example session with the calculator.

6.5 Discussion

This chapter has illustrated the methods that can be used for developing denotational style interpreters for a formal language using META-LISP. To summarise, the main steps involved are as follows:

1. Define the Abstract Syntax of the language
   - Construct a reader and a lexical analyser for the language
   - Construct a Parser for the language which produces a list structure representation of the abstract syntax of the language
   - Construct a parser for the abstract syntax of the language

```
0> N : (((1 2) 3))
 1> n : ((1 2) 3)
 <1 n : (1 2) = (1 2)
 1> d : (3)
 <1 d : 3 = 3
 1> N : ((1 2))
  2> n : (1 2)
  <2 n : 1 = 1
  2> d : (2)
  <2 d : 2 = 2
  2> N : (1)
   3> d : (1)
   <3 d : 1 = 1
   3> D : (1)
   <3 D : (1) = 1
  <2 N : (1) = 1
  2> times : (10 1)
  <2 times : (10 1) = 10
  2> D : (2)
  <2 D : (2) = 2
  2> plus : (10 2)
  <2 plus : (10 2) = 12
 <1 N : ((1 2)) = 12
 1> times : (10 12)
 <1 times : (10 12) = 120
 1> D : (3)
 <1 D : (3) = 3
 1> plus : (120 3)
 <1 plus : (120 3) = 123
<0 N : (((1 2) 3)) = 123
```

Figure 6.15: Trace of Interpreting Numerals

2. Construct or import concrete implementations of the semantic domains and algebras to be used in the specification of the mapping of syntactic constructs of the language into their denotations.

3. Formulate the Evaluation Rules specifying the semantics of the language based on the parser for the abstract syntax of the language.

4. Experiment with the interpreter thus obtained, both as the means of refining the specification and of developing a more thorough understanding of the workings of the language thus defined.

The experiment of writing denotational style interpreters in META-LISP, described in this chapter and the following chapter, has been instructive on several accounts. Although experimenting with executable specification cannot be a substitute for clear thought, it is apparent that a machine readable and executable language definition can greatly assist in 'debugging' language specifications.

```
1> P : (("ON" ((("(" (4 "+" (1 2)) ")") "*" 2) "TOTAL"
              ((1 "+" "LAST") "TOTAL"  (("IF" ("LAST" "+" 1) "," 0 "," (2 "+" 4)) "TOTAL" "OFF")))))
 2> S : ((((("(" (4 "+" (1 2)) ")") "*" 2) "TOTAL" ((1 "+" "LAST") "TOTAL"
        (("IF" ("LAST" "+" 1) "," 0 "," (2 "+" 4)) "TOTAL" "OFF"))) 0)
  3> E : (((("(" (4 "+" (1 2)) ")") "*" 2) 0)
   4> E : ((("(" (4 "+" (1 2)) ")") 0)
    5> E : ((4 "+" (1 2)) 0)
     6> E : (4 0)
      7> N : (4)
      <7 N : (4) = 4
     <6 E : (4) = 4
     6> E : ((1 2) 0)
      7> N : ((1 2))
       8> N : (1)
       <8 N : (1) = 1
      <7 N : ((1 2)) = 12
     <6 E : ((1 2)) = 12
    <5 E : ((4 "+" (1 2)) 0) = 16
   <4 E : ((("(" (4 "+" (1 2)) ")") 0) = 16
   4> E : (2 0)
    5> N : (2)
    <5 N : (2) = 2
   <4 E : (2) = 2
  <3 E : (((("(" (4 "+" (1 2)) ")") "*" 2) 0) = 32
  <3 S : (<- m~@S) = 32
  3> S : (((1 "+" "LAST") "TOTAL" (("IF" ("LAST" "+" 1) "," 0 "," (2 "+" 4)) "TOTAL" "OFF")) 32)
   4> E : ((1 "+" "LAST") 32)
    5> E : (1 32)
     6> N : (1)
     <6 N : (1) = 1
    <5 E : (1) = 1
    5> E : ("LAST" 32)
    <5 E : ("LAST" 32) = 32
   <4 E : ((1 "+" "LAST") 32) = 33
   <4 S : (<- m~@S) = 33
   4> S : ((("IF" ("LAST" "+" 1) "," 0 "," (2 "+" 4)) "TOTAL" "OFF") 33)
    5> E : (("IF" ("LAST" "+" 1) "," 0 "," (2 "+" 4)) 33)
     6> E : (("LAST" "+" 1) 33)
      7> E : ("LAST" 33)
      <7 E : ("LAST" 33) = 33
      7> E : (1 33)
       8> N : (1)
       <8 N : (1) = 1
      <7 E : (1) = 1
     <6 E : (("LAST" "+" 1) 33) = 34
     6> E : ((2 "+" 4) 33)
      7> E : (2 33)
       8> N : (2)
       <8 N : (2) = 2
      <7 E : (2) = 2
      7> E : (4 33)
       8> N : (4)
       <8 N : (4) = 4
      <7 E : (4) = 4
     <6 E : ((2 "+" 4) 33) = 6
    <5 E : (("IF" ("LAST" "+" 1) "," 0 "," (2 "+" 4)) 33) = 6
   <4 S : ((("IF" ("LAST" "+" 1) "," 0 "," (2 "+" 4)) "TOTAL" "OFF") 33)
      = (6)
  <3 S : (((1 "+" "LAST") "TOTAL" (("IF" ("LAST" "+" 1) "," 0 "," (2 "+" 4)) "TOTAL" "OFF")) 32)
     = (33 6)
 <2 S : ((((("(" (4 "+" (1 2)) ")") "*" 2) "TOTAL"
          ((1 "+" "LAST") "TOTAL"  (("IF" ("LAST" "+" 1) "," 0 "," (2 "+" 4)) "TOTAL" "OFF")))
         0)
    = (32 33 6)
<1 P : (("ON"
         ((("(" (4 "+" (1 2)) ")") "*" 2) "TOTAL"
          ((1 "+" "LAST") "TOTAL" (("IF" ("LAST" "+" 1) "," 0 "," (2 "+" 4)) "TOTAL" "OFF")))))
   = (32 33 6)
```

Figure 6.16: Trace of Interpreting Example Session

However, animating the rules cannot, in itself, resolve questions relating to the functionality of the rules used in the specification for the semantics. At best inconsistencies can be indicated. Or, looking at it the other way, specifying the intended functionality of an evaluation rule can help to get the formulation of the rule itself right. Ideally, there should be a machine provable link between the form that evaluation rules take and their prescribed functionality. At the moment, however, information concerning the functionality of the semantic mappings cannot be incorporated into the META-LISP specification. Clearly to do so would require the development of some kind of type discipline or type inference scheme for META-LISP. The role of such type discipline, in the context of writing denotational interpreters will be analogous to the type checker of Wand's Semantic Prototyping System. [Wan84]. Developing a type discipline for META-LISP is the subject of future research. Making the task of writing denotational style interpreters less error-prone is one of the main motivations for such research. Experience, so far, in writing denotational style interpreters in META-LISP indicates not only that a type discipline would be beneficial, but that it may be possible to develop a decidable one, at least for the class of programs that are structured to meet the requirements of developing denotational style interpreters.

Chapter 7

Meta-circular Definition of META-LISP

An interpreter or evaluator for a language is a procedure that, when applied to an expression of the language, performs the actions required to evaluate that expression. An interpreter that is written in the same language that it evaluates is said to be *meta-circular* [ASS85][295]. This chapter presents a denotational style interpreter for META-LISP written in META-LISP. The first section of the chapter discusses the appropriateness and adequacy of meta-circular definitions, in general, and the use of META-LISP as its own meta-language, in particular. It is hoped that the previous chapters have enabled the reader to acquire a reading knowledge of META-LISP so that its meta-circular definition will be understandable. Moreover, that the study of this definition can, in fact, improve one's understanding of the language.

The previous chapter have introduced the format of denotational language definitions. It has also illustrated the use of META-LISP in developing denotational style interpreters. The main steps of providing such a language definition have been identified as follows:

1. Construct a reader and lexical analyser for the language
2. Specify the Abstract Syntax of the language
3. Specify the semantic domains and operations
4. Using the definition of the abstract syntax of the language specify the evaluation rules

The organisation of this Chapter reflects the sequence of these steps. The specification of the syntax of META-LISP is the subject of Section 2. Its aim is to develop a machine readable description of the abstract syntax of META-LISP. Section 3 discusses the semantic domains and operations that are used in the formulation of valuation rules. The semantics

of META-LISP is then formulated in Sections 4 and 5 in terms of *valuation functions*. Section 4 deals largely with the underlying language definitional formalism of META-LISP. Section 5 presents the semantics of the applicative language of the semantic actions. The chapter concludes with a discussion of the lessons learned from writing a meta-circular interpreter for META-LISP.

7.1 Meta-Circular Language Definitions

McCarthy's definition of LISP in terms of the universal function, *eval*, formulated in LISP, was the first example of a meta-circular definition of a programming language. This section looks at some of the arguments for, and against, the adequacy of meta-circular definitions. It also reflects on the status of the meta-circular definition of META-LISP presented in this chapter.

7.1.1 *For* Meta-circular Definitions

John Allen, in the *Anatomy of LISP* argues strongly in favour of the appropriateness of a meta-circular definition of the *operational semantics* [1] of programming languages. Firstly, he points out, that in defining the operational semantics of a programming language, "realistically, the choice is where to stop, not whether to stop."[All78][163]. In the case of LISP, he argues that LISP and its data structures are sufficiently simple to render self-description satisfactory. There are, as Allen points out, compelling reasons for deciding on direct circularity. To understand a meta-circular definition one need only to understand one language, the specification language being the same as the one that it specifies. Understanding the workings of a language then boils down to understanding a single program. Meta-circular definition of programming languages have the added advantage of reducing the task of initial implementation of the language to the task of hand-coding the meta-circular interpreter.[2] Bootstrapping (see 183) can then be used to modify and extend the language by simply modifying a single high level program.

[1] or the *pragmatics* or *procedural semantics* of a programming language concerns itself with the process of interpretation of constructs of a language. It is usually contrasted with *mathematical* or *declarative semantics* which concerns itself with the relation between constructs of the language and the abstract mathematical objects which they denote

[2] It is interesting to note that the original definition of LISP, in the form of a meta-circular interpreter, was put forward purely as the means of defining and illustrating the capabilities of the language as an alternative to Turing machines in the context of the theory of computation. It was only later that S. R. Russell noticed that the meta-circular description of LISP can serve as an interpreter for it, which only needed to be hand-coded to obtain a programming language with an interpreter. [Wex81][179]

In the *The Structure and Interpretation of Computer Programs* Abelson and Sussman point out, that since evaluation is a process, and LISP is used as a tool for describing processes, it is appropriate to describe the evaluation process of LISP using LISP. [ASS85][295].

In his book *On understanding Z*, Spivey presents the formal semantics of the Z notation which itself is written using Z as a meta-language. The spirit of Spivey's main argument, in support of using Z as its own meta-language, is similar to the previous argument for the use of LISP to explicate the process of evaluation in LISP. As Z is put forward as a language to write and reason about formal specifications, the purpose of giving a mathematical model to help us to understand Z specifications and to reason about them, can be served well by a semantics expressed in Z. Spivey also points out that for the design and specification of software tools to assist in the process of writing and refining software tools for Z, it is appropriate that the formal definition of the specification language is already written in a notation designed for expressing software specifications. In addition, writing the semantic definition in Z also provides a useful example of the flexibility of Z as a framework for developing mathematical theories. [Spi88][9-10]

7.1.2 *Against* Meta-circularity

The most familiar objection against of meta-circular definitions is, that if one does not understand the language being defined, then looking at a meta-circular definition of it will not help. It is indeed the case that, for a meta-circular definition to be meaningful, an *independent* understanding is required of at least one program written in the language purportedly defined by it. The need for developing such an understanding is very similar to the need of developing an understanding of, say the language LAMBDA, that is defined and used by Stoy in developing the theory and technique of "standard" denotational semantics. It is tempting to suggest, that many of the difficulties that Stoy had discussed, with reference to meta-circular definitions, if taken literally, can be seen to befall any foundational enterprise. In other words, in defining the semantics of a language, we have to assume the knowledge of at least one, sufficiently rich language.

Stoy calls into doubt whether a meta-circular definition can be thought of as *defining* anything at all.[Sto77][181-182] He acknowledges that if one had a partial, *independent* understanding of a language, examination of a meta-circular definition of it can help to improve this understanding. He then goes on to suggest that the semantics of the language can be thought of as a "fixed point" of the meta-circular interpreter, and warns that it may not be unique. Moreover, he states that the "*minimal* fixed point of the interpreter will be the language in which the value of every expression is undefined: so that the interpreter

cannot really be thought of as *defining* anything at all" [Sto77][182].

The objection against meta-circular definitions, that they can, or are very likely to be ambiguous, if care is not taken, is well founded. For example. Henderson [Hen80][114] discusses two alternative formulations for the interpretation of the conditional (IF e_1 e_2 e_3). Only one of them will assign the property that only one arm of the conditional is evaluated, independently of whether the language, in which the meta-circular interpreter is written, has that property. Similar problems related to assumptions about parameter passing mechanism, (whether it be call by value, or call by name) are discussed by Reynolds [Rey72]. Clearly, care must be taken to make explicit those assumptions about the defining language that are carried over to the defined language. These assumptions include the meanings assigned to the primitive operations of the defining language. The fact that such care needs to be exercised, however, does not, in my view, invalidate the use of meta-circular definitions.

Stoy's objections to meta-circular definitions stems from foundational concerns. For the purpose of developing appropriate mathematical foundations, these concerns are crucial. The point of developing meta-circular definitions is not, however, to address foundational issues, but to develop a model of the semantics of a language, which although it cannot be said to be self-standing, (hardly any theory can ever be) can, nevertheless, enhance our understanding of a language. This purpose is served very well in the case of LISP, or Z for that matter. It is my hope that it will be judged to be the case for META-LISP as well.

7.1.3 META-LISP as its own meta-language

META-LISP is a meta-language to begin with. I.e., it is a language specifically designed for the purpose of defining both the syntax and the semantics of formal languages. From this point of view, it is incidental, that it was designed with a view to write programs *as* language processors for their input data language.

From the standpoint of semantic definitions, META-LISP can be used to specify the semantics of a language, in the form of a *definitional interpreter*, (as demonstrated in the previous Chapter). It can also be used to specify the *translational semantics* of a language by specifying a translation of it into another language, which, for the purpose of the definition of the semantics, can be regarded as a semantically primitive language. Both forms of describing the semantics of a language in META-LISP can be carried out in the denotational style. That is to say, in a form in which the meaning of composite phrases of the defined language are given in terms of the meanings of its components.

Formulating a denotational language specification in META-LISP, whether it be in the

form of a definitional interpreter, or a 'definitional compiler' has many of the advantages usually associated with standard denotational definitions. According to Stoy, formal semantic descriptions have, at least three, well identifiable kinds of benefits:

The first is that such definitions can help to give sufficiently precise description for *implementors* of the language to construct a correct compiler. This purpose have been served very well in the implementation of META-LISP, as the availability of the meta-circular definition helped to clarify many intricate issues of the semantics of the language. The meta-circular definition is not put forward, as a "pedagogic device", [3] but it is being proposed here as the standard of implementation (i.e. how things *should* work). By developing a *partial evaluator* for META-LISP it should be possible to derive a provably correct compiler for META-LISP from its denotational style definition. For further discussion of this issue see the section on Future Work in the concluding Chapter.

Another important benefit of a formal definition of the semantics of a programming language is that it can be used by *programmers* to make rigorous statements about the behaviour of the programs they write. As things stand, at the present moment, reasoning about META-LISP programs, be they language definitions or 'just' programs, can only be informal. Informality, does not, however, exclude rigour. In fact the very structure of META-LISP program encourages the routine use of informal structural induction arguments. Much of the future work envisaged for the further development of the language addresses the objectives of making such arguments more formal, as well as providing machine support for reasoning about META-LISP programs. The planned development of a *type inference* scheme for META-LISP will be the basis for this.

The third, and according to Stoy, probably the most important expected benefit of formal semantic definitions is that they can guide language designers "towards the design of better (cleaner) programming languages, with simpler formal descriptions. And the advantage of that will be that the programs which we concoct by the usual informal methods will be more probably correct, because we will less likely have forgotten about the crucial little exception to some general rule that applies in our particular case". The extent to which the design of META-LISP has benefited from its, denotational style, meta-circular definition can even be said to have surpassed the benefits that could have been gained from the development of a "standard" denotational semantics for it.

The problem with "standard" denotational definitions, as Gougen and Meseguer have

[3] The reader is warned that the definitions of apply and eval given above are pedagogical devices and are not the same functions as those built into the LISP programming system. Appendix B contains the computer implemented version of these functions and should be used to decide questions about how things really work. [MAE+65, 14]

pointed out, is that they " commonly run to thousands of lines of hard-to-digest higher order semantic equations, and are almost certainly wrong in detail, since they have never been mechanically tested." [GM86, 300] The sheer complexity and the number of cases and "crucial little details" that need to be considered makes the need for mechanical testing paramount. This need could, of course have been served, not only by a meta-circular interpreter, but by, say, developing a "denotational semantics interpreter" first, as it has been done, for example, by Nicholson and Foo. [NF89, 665] They have developed and tested a denotational semantics for a core subset of Prolog, with small examples, using a denotational semantics interpreter, written in Prolog. Given the explicit meta-linguistic capabilities of META-LISP, however, there did not seem to be much point in writing a separate denotational semantics interpreter. Instead, the same objective has been equally well served by adhering to an appropriate style of writing language definitions in META-LISP, as in the previous and the present chapter.

7.2 The Syntax of META-LISP

The distinguishing feature of META-LISP programs is that the set of valid input that they are to accept is defined explicitly as a language. The output of a META-LISP program is specified as *translation*(s) of the input language of the program. These translations are specified by attaching *semantic actions* to each rule of the underlying grammar, that defines the set of valid inputs to the program. The fundamental functional units of a META-LISP program are known as *effective concepts* or *translation procedures*. These correspond to the non-terminals of the underlying grammar. The term *effective concept* is used to emphasise their role they play in explicating the *conceptual structure* of both the inputs to the program and the way corresponding output is to be computed.

A META-LISP program consists of a series of *definitions* for effective concepts. These definitions consist of the name of an effective concept and a series of *rules*, terminated by an empty line. The abstract syntax of META-LISP is shown in Figures 7.1 and 7.2. Figure 7.1 describes the abstract syntax of the grammatical means of composition of META-LISP programs. Figure 7.2 describes the abstract syntax of the semantic actions. The following meta-syntactic conventions are being used:

1. non-terminals are shown in *italic*
2. the symbol ::= marks the beginning of grammar rules
3. the symbol | marks an additional alternative
4. keywords are represented in `typewriter` font
5. lexical classes are enclosed in angle-brackets, e.g. ⟨, ⟩

Lexical classes are assumed to denote their corresponding lexical class in LISP.

7.2.1 Lexical Analysis

Lexical matters will largely be ignored in the present discussion. The lexical analyser performs the usual task of translating a stream of characters into tokens of the language. It involves identifying keywords and the words of the language. Its basic design is similar to the lexical analyser for the Symbolic differentiation program. There is only one new feature of its design that needs special mention. This feature concerns the treatment of strings in the input. Since keywords of the language are tokenised in the form of strings, there is a need to be able to tell them apart from strings in the input.

Program	::=	*Ec Rules Program*	(Definitions)
Ec	::=	⟨identifier⟩	(Concept Name)
Rules	::=	*Alts*	(Alternatives)
	|	*Left*	(Left Recursion)
Alts	::=	`:` *Synt* `=` *Sem Alts*	(Committed Alternative)
	|	`:` *Synt* `?` *Sem Alts*	(Backtracking)
	|	`:` *Synt Alts*	(Default Action)
Synt	::=	*Pseudo*	(Pseudo Rule)
	|	*Compos*	(Composition)
Pseudo	::=	**is** *Pred*	(Predication)
	|	**any** *Objects*	(Enumeration)
	|	**with lisp** *Fn*	(LISP Primitive)
	|	**with** *Module Ec*	(Importing)
Pred	::=	⟨identifier⟩	(LISP Predicate)
Objects	::=	⟨object⟩ *Objects*	(LISP Objects)
Fn	::=	⟨indentifier⟩	(LISP Function)
Module	::=	⟨keyword⟩	(LISP Keyword)
Compos	::=	*Comp Compos*	(Composition)
Comp	::=	`_`	(Prefix)
	|	`._`	(Suffix)
	|	`<>`	(Empty)
	|	`$`	(End)
	|	*Denot*	(Denotation)
	|	*Ec*	(Constituent Ec)
	|	□	(nil)
	|	`[` *Compos* `]`	(Nested Composition)
Denot	::=	`'`⟨object⟩	(LISP Object)
	|	⟨string⟩	(String)
	|	⟨keyword⟩	(Keyword)
Left	::=	*Start Rec*	(Left Recursion)
Start	::=	*Alts*	(Starting Alternatives)
Rec	::=	*Alts*	(Iterated Alternatives)

Figure 7.1: Abstract Syntax I

Sem	::=	*Sterms*	(Semantic Action)
Sterms	::=	*Sterm Sterms*	(Semantic Terms)
Sterm	::=	{ *Sterms* }	(Sequencing)
	|	(@ *Id* <- *Sterm*)	(Synthesised Attribute)
	|	(@ *Id*)	(Synthesised Attribute)
	|	(^ *Id* <- *Sterm*)	(Inherited Attribute)
	|	(^ *Id*)	(Inherited Attribute)
	|	(**if** *Bool St1 St2*)	(If Then Else)
	|	(*Sterm Els*)	(Invocation)
	|	[*Els*]	(Construction)
	|	< *Ec* >	(Procedure Designation)
	|	*Denot*	(Denotation)
	|	⟨number⟩	(Number)
	|	*Id*	(Reference)
	|	**fail!**	(Failure)
Id	::=	⟨identifier⟩	(Identifier)
Bool	::=	*Sterm*	(Test)
St1	::=	*Sterm*	(Consequent 1)
St2	::=	*Sterm*	(Consequent 2)
Els	::=	. *Sterm Els*	(Splicing/Appending)
	|	*Sterm Els*	(Cons-ing)

Figure 7.2: Abstract Syntax II

The strategy adopted to distinguish tokens and strings is to represent string in the input as lists formed of the keyword `:string` and the string that has been read.

7.2.2 Mapping from Concrete to Abstract Syntax

The program for mapping from concrete to abstract syntax takes the following input: the name of an effective concept, the representation of the rules, in terms of tokens, that make up its definition, and a boolean value to determine whether a parse tree or the internal representation of the abstract syntax is to be produced. Figure 7.3 shows the top-level elaboration of the program.

The mapping from concrete to abstract syntax is straightforward except for the treatment of left recursive rules. A rule for an effective concept X is said to be left recursive

```
mci-c2a
 : ec tokens _                 = (^ tree <- _) (Defn ec tokens)

Defn
 : ec tokens                   = (^ ec) (Rules . tokens)

Rules
 : Comments Left               ? (if recs@Left
                                     (abst 'Left
                                       (abst 'Start (Alts . start@Left))
                                       (abst 'Rec (Alts . recs@Left)))
                                   fail!)
 : Comments Alts

Comments
 : [:comment rest] Comments    = t
 : <>
```

Figure 7.3: Top Level Elaboration of Concrete to Abstract Mapping

if its first component is X. left recursive rules are identifier by the effective concept *Left*, shown in Figure 7.4.

```
Left
 : rule1  rest               = (if (eq ^ec comp1@rule1)
                                   { (Left . rest)
                                     (@ recs <- [":" . rest-of-rule@rule1 . recs@Left]) }
                                 { (Left . rest)
                                   (@ start <- [. rule1 . start@Left]) })
 : $                         = (@ recs <- []) (@ start <- [])

rule1
 : ":" comp1 rest-of-rule    =  (@ comp1)
                                (@ rest-of-rule)
                                [":" comp1 . rest-of-rule]

rest-of-rule
 : item rest-of-rule         = [item . rest-of-rule]
 : <>

item
 : $                         = fail!
 : ":"                       = fail!
 : _
```

Figure 7.4: Concrete Syntax of Left Recursive Rules

Left examines each rule in turn and decides whether it is left recursive or not. It returns two values, in the form of synthesised attributes, which correspond to the left recursive and the non-left recursive rules found. Rules that are classified as non-left recursive are simply combined to form the *start-up* rules of the definition. left recursive rules are returned as new alternatives which are obtained by removing their first (left recursive) component. If it is found that the rules involve left recursion, then the start-up rules and the left recursive

rules are translated separately as alternatives by *Alts*. If there are no left recursive rules, then the input is translated simply as a sequence of alternatives. Note the use of semantic backtracking (marked by the keyword ?) in the definition of *Rules*, to accomplish this re-examination of the input in the absence of left recursion. Closer examination reveals, that *Left* will separate left recursive rules from non-left recursive ones, but it does not check whether they are consecutive or not. This should be improved upon in future version of the program.

```
expr
 : term
 : expr + term = [+ expr term]
```

```
((":" (term $) $)
 (":" (+ (term $)) "=" (("[" (+ (expr (term $))) "]") $) $)))
```

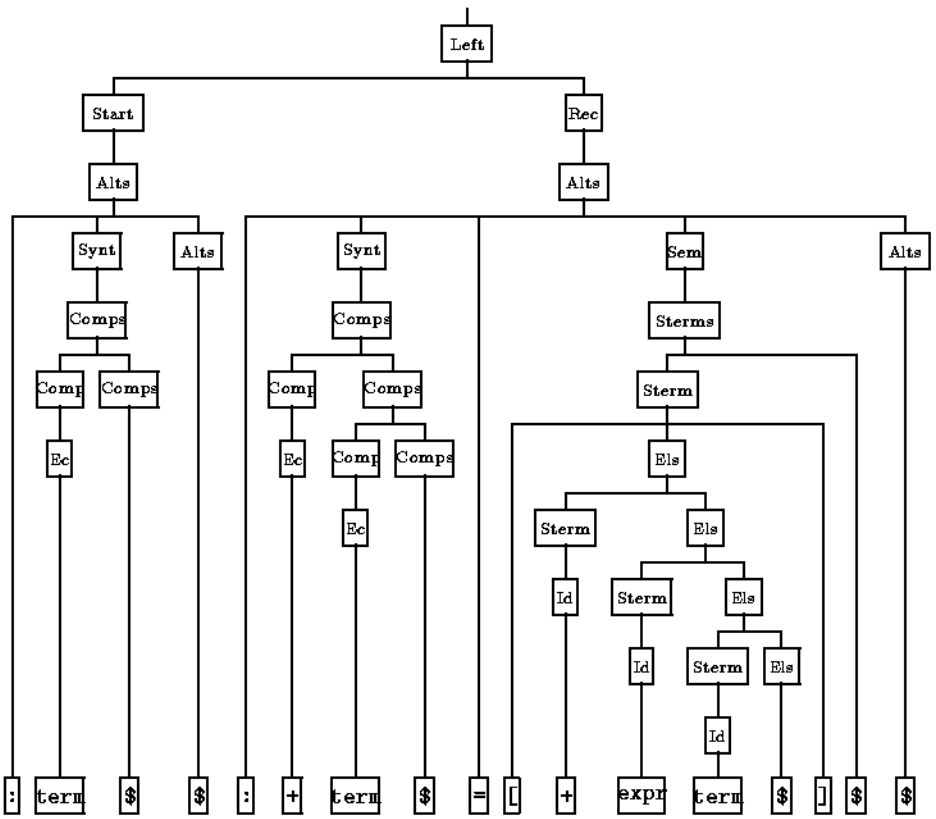


Figure 7.5: Structure of a left recursive Definition

Figure 7.5 shows a left recursive definition, the internal representation of its abstract syntax, and its parse tree. Contrast it with Figure 7.6, that shows a right-recursive (i.e. non-left recursive) variant of *expr*.

```
expr
 : term + expr = [+ term expr]
 : term


(expr
 (":"
  (term (+ (expr $)))
  "="
  (("[" (+ (term (expr $))) "]") $)
  (":" (term $) $)))
```

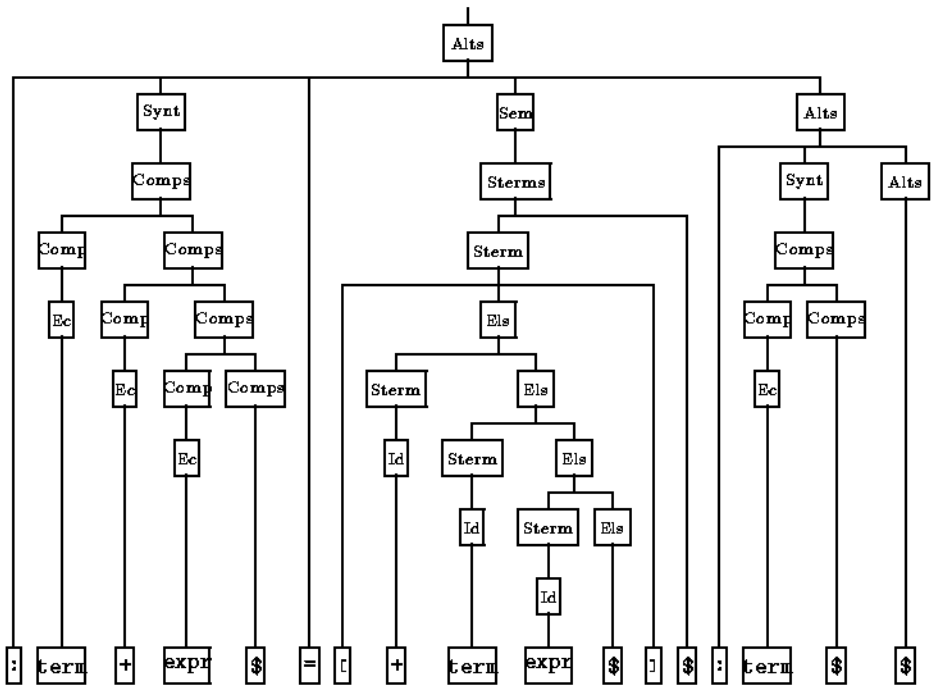


Figure 7.6: Structure of a Right-recursive Definition

The program for mapping alternatives into their abstract syntax representation is shown in Figure 7.7. Figure 7.8 shows the same mapping for semantic actions. Note the use of $ as the means of representing the end of a sequence.

The remaining definition for *mci-c2a* are collected in Figure 7.9.

```
Alts
 : ":" Synt "=" Sem Alts      = (abst 'Alts ":" Synt "=" Sem Alts)
 : ":" Synt "?" Sem Alts      = (abst 'Alts ":" Synt "?" Sem Alts)
 : ":" Synt Alts              = (abst 'Alts ":" Synt Alts)
 : $                          = (abst 'Alts $)

Synt
 : Pseudo                     = (abst 'Synt Pseudo)
 : Compos                     = (abst 'Synt Compos)

Pseudo
 : "is" Pred                  = (abst 'Pseudo "is" Pred)
 : "any" Objects              = (abst 'Pseudo "any" Objects)
 : "with" "lisp" Fn           = (abst 'Pseudo "with" "lisp" Fn)
 : "with" Module Ec           = (abst 'Pseudo "with" Module Ec)

Compos
 : Comp Compos                = (abst 'Compos Comp Compos)
 : <>                         = (abst 'Compos $)

Comp
 : Denot                      = (abst 'Comp Denot)
 : "_"                        = (abst 'Comp "_")
 : "._"                       = (abst 'Comp "._")
 : "<>"                       = (abst 'Comp "<>")
 : "$"                        = (abst 'Comp "$")
 : Ec                         = (abst 'Comp Ec)
 : "[" Compos "]"             = (abst 'Comp "[" Compos "]")

Denot
 : "'" Object                 = (abst 'Denot "'" Object)
 : String                     = (abst 'Denot String)
 : keywordp                   = (abst 'Denot keywordp)

String
 : [:string stringp]          = (abst 'String [:string stringp])

Pred
 : is identifier              = (abst 'Pred is)

Objects
 : objects                    = (abst 'Objects objects)

Fn
 : is identifier              = (abst 'Fn is)

Module
 : is keywordp                = (abst 'Module is)

Ec
 : is identifier               = (abst 'Ec is)

Object
 : _
```

Figure 7.7: Concrete Syntax of Alternatives

```
Sem
 : Sterms                    = (abst 'Sem Sterms)

Sterms
 : Sterm Sterms              = (abst 'Sterms Sterm Sterms)
 : Sterm                     = (abst 'Sterms Sterm $)

Sterm
 : "{" Sterms "}"            = (abst 'Sterm "{" Sterms "}")
 : "(" "@" Id ")"            = (abst 'Sterm "(" "@" Id ")")
 : "(" "@" Id "<-" Sterm ")" = (abst 'Sterm "(" "@" Id "<-" Sterm ")")
 : "(" "^" Id ")"            = (abst 'Sterm "(" "^" Id ")")
 : "(" "^" Id "<-" Sterm ")" = (abst 'Sterm "(" "^" Id "<-" Sterm ")")
 : "(" "if" Bool St1 St2 ")" = (abst 'Sterm "(" "if" Bool St1 St2 ")")
 : "(" Sterm Els ")"         = (abst 'Sterm "(" Sterm Els ")")
 : "[" Els "]"               = (abst 'Sterm "[" Els "]")
 : "<" Ec ">"                = (abst 'Sterm "<" Ec ">")
 : Denot                     = (abst 'Sterm Denot)
 : Id                        = (abst 'Sterm Id)
 : Number                    = (abst 'Sterm Number)
 : Failure                   = (abst 'Sterm Failure)

Id
 : is identifier             = (abst 'Id  is)

Els
 : "." Sterm Els             = (abst 'Els "." Sterm Els)
 : "." Sterm                 = (abst 'Els "." Sterm)
 : Sterm Els                 = (abst 'Els Sterm Els)
 : Sterm                     = (abst 'Els Sterm)
 : <>                        = (abst 'Els $)

Number
 : is numberp                = (abst 'Number is)

Failure
 : "fail!"                   = (abst 'Failure "fail!")

Bool
 : Sterm                     = (abst 'Bool Sterm)

St1
 : Sterm                     = (abst 'St1 Sterm)

St2
 : Sterm                     = (abst 'St1 Sterm)
```

Figure 7.8: Concrete Syntax of Semantic Actions

```
objects
 : object objects              = [object . objects]
 : <>

object
 : any "=" "?" ":"             = fail!
 : $                           = fail!
 : _


comp1          : _
ec             : is identifier
eq             : with lisp eq
rest           : ._
tokens         : _
stringp        : is stringp
```

Figure 7.9: Miscellanous Definition in *mci-c2a*

The definition of *abst* is the same as in Figure 6.8

7.2.3 The Abstract Syntax of META-LISP

Figures 7.10 and 7.11 show the definition of the abstract syntax of META-LISP written in META-LISP. As in the previous chapter, the definition of the abstract syntax in META-LISP specifies the list structure representation of the constructs of the language. The principle is the same, i.e. composite structures are represented as lists formed of their components. The definition can alternatively used to generate abstract parse trees.

The following effective concepts all have definitions of the form X `: _` , i.e. they are all defined as place-holders: `alts, bool, c1, c2, comp, compos, els, fn, id, object, objects, pred, rec, sem, start, sterm, sterms, structure, synt, tree.` The following effective concepts are defined using the *pseudo rule* of the form X `: is` X, i.e. they import the following functions from LISP: `identifier, keywordp, module, stringp.`

```
mci-abs
 : ec structure tree          = (^ tree) (Defn ec structure)

Defn
 : ec Rules                   = Rules

Rules
 : Left
 : Alts

Left
 : [start rec]                = (abst 'Left (Start start) (Rec rec))

Start
 : Alts                       = (abst 'Start Alts)

Rec
 : Alts                       = (abst 'Rec Alts)

Alts
 : [":" synt "=" sem alts]    = (abst 'Alts ":" (Synt synt) "=" (Sem sem) (Alts alts))
 : [":" synt "?" sem alts]    = (abst 'Alts ":" (Synt synt) "?" (Sem sem) (Alts alts))
 : [":" synt alts]            = (abst 'Alts ":" (Synt synt) (Alts alts))
 : '$                         = (abst 'Alts $)

Synt
 : Pseudo                     = (abst 'Synt Pseudo)
 : Compos                     = (abst 'Synt Compos)

Pseudo
 : ["is" pred]                = (abst 'Pseudo "is" pred)
 : ["is!" pred]               = (abst 'Pseudo "is!" pred)
 : ["any" objects]            = (abst 'Pseudo "any" objects)
 : ["with" "lisp" fn]         = (abst 'Pseudo "with" "lisp" fn)
 : ["with" module ec]         = (abst 'Pseudo "with" module ec)

Compos
 : [comp compos]              = (abst 'Compos (Comp comp) (Compos compos))
 : '$                         = (abst 'Compos $)

Comp
 : Denot                      = (abst 'Comp Denot)
 : "_"                        = (abst 'Comp "_")
 : "._"                       = (abst 'Comp "._")
 : "<>"                       = (abst 'Comp "<>")
 : "$"                        = (abst 'Comp "$")
 : Ec                         = (abst 'Comp Ec)
 : String                     = (abst 'Comp String)
 : Keyword                    = (abst 'Comp Keyword)
 : ["[" compos "]"]           = (abst 'Comp "[" (Compos compos) "]")

Denot
 : ["'" object]               = (abst 'Denot "'" object)
 : String                     = (abst 'Denot String)
 : Keyword                    = (abst 'Denot Keyword)

String
 : [:string stringp]          = (abst 'String stringp)

Keyword
 : is keyworp                 = (abst 'Keyword is)

Ec
 : is identifier              = (abst 'Ec is)
```

Figure 7.10: Abstract Syntax I in META-LISP

```
Sem
 : sterms                      = (abst 'Sem (Sterms sterms))

Sterms
 : [sterm sterms]              = (abst 'Sterms (Sterm sterm) (Sterms sterms))
 : '$                          = (abst 'Sterm $)

Sterm
 : ["{" sterms "}"]             = (abst 'Sterm "{" (Sterms sterms) "}")
 : ["(" "@" id ")"]             = (abst 'Sterm "(" "@" id ")")
 : ["(" "@" id "<-" sterm ")"] = (abst 'Sterm "(" "@" id "<-" (Sterm sterm) ")")
 : ["(" "^" id ")"]             = (abst 'Sterm "(" "^" id ")")
 : ["(" "^" id "<-" sterm ")"] = (abst 'Sterm "(" "^" id "<-" (Sterm sterm) ")")
 : ["(" "if" bool c1 c2 ")"]   = (abst 'Sterm "(" "if" bool c1 c2 ")")
 : ["(" sterm els ")"]          = (abst 'Sterm "(" (Sterm sterm) (Els els) ")")
 : ["[" els "]"]                = (abst 'Sterm "[" (Els els) "]")
 : ["<" ec ">"]                 = (abst 'Sterm "<" ec ">")
 : Denot                        = (abst 'Sterm Denot)
 : Id                           = (abst 'Sterm Id)
 : Number                       = (abst 'Sterm Number)
 : Failure                      = (abst 'Sterm Failure)

Id
 : is identifier               = (abst 'Id is)

Number
 : is numberp                  = (abst 'Number is)

Failure
 : "fail!"                     = (abst 'Failure "fail!")

Els
 : ["." sterm els]             = (abst 'Els "." (Sterm sterm) (Els els))
 : [sterm els]                 = (abst 'Els (Sterm sterm) (Els els))
 : '$                          = (abst 'Els $)
```

Figure 7.11: Abstract Syntax II in META-LISP

7.3 Semantic Algebras

7.3.1 Semantic Domains

The following notions will be considered as elementary for the purposes of the present exposition: ⟨*keywords*⟩, ⟨*identifier*⟩, ⟨*atom*⟩ and ⟨*fail*⟩. These are all fundamental constructs of LISP and other programming languages.

`fail!`	= fail | succ(fail!)	Hierarchy of fails
`val`	= ⟨*atom*⟩ | fail! | (list val)	Denotable Value
`input`	= val	Input
`ec`	= id	Name of effective concept
`id`	= ⟨*identifier*⟩	Identifier
`mod`	= ⟨*keyword*⟩	Module Name
`env`	= (list (list id value))	Environment
`loc`	= env	Local Bindings
`glob`	= env	Global Bindings
`iat`	= env	Bindings for Inherited Attributes
`sat`	= env	Bindings for Synthesised Attributes
`init-env`	= (list (list))	Initial Environment

7.3.2 Semantic Functions

The META-LISP implementations of semantic functions for handling environments is shown in Figure 7.12

```
lookup
% id -> iat -> loc -> sat -> val
 : id iat loc sat
  = (if (bound? key (merge-envs iat loc sat)) val@bound? id)

merge-envs
 %  list -> list
 : with lisp append

a-list
 % env
 : _

bound?
 % id -> env -> (list id value) | ()
 : boundh
  = (if boundh { (@ val <- (first (rest boundh))) t } [])

boundh
 : identifier a-list
  = (assoc identifier a-list)

assoc
 : with lisp assoc

add-b
% Add new Binding of ec to val to environment
 : key val env
  = [[key val] . env]

add-bs
 : bindings env
  = (list2set [. bindings . env])
```

Figure 7.12: Semantic functions

7.4 The Semantics of META-LISP: Part I

The notion corresponding to *program execution* in META-LISP is the *invocation* of an *effective concept* with certain input. What is submitted to META-LISP is the name of an effective concept followed by a sequence of objects forming the input. The invocation of an effective concept with certain input can be thought of as a *query* for determining whether some *prefix* of the given input is a sentence of the *input language* of the program. The invocation of an effective concept can succeed or it can fail. If the query has been successful then an 'answer' to the query is produced. This takes the form of a *value*, or *principal translation* for the invocation, and bindings for the possible synthesised attributes of the of the effective concept invoked. The value of an invocation, and its possible attributes represents translation(s) of the matched prefix of the input. An example of a query presented to the META-LISP system is shown below:

```
| ?= (split [a b c d e] alphalessp)
```

The response is shown below:

```
| (a b c d e)
|------------
| alphalessp

p1@split = (a c e)
p2@split = (b d)
split = ((a b c) (b d))
```

The example represents the invocation of the effective concept, *split*, with the input list `((a b c d e) alphalessp)`. As can be seen from the definition of *split* shown in Figure 7.13, *split* has two synthesised attributes `p1@split` and `p2@split`. The answer to the query

```
 split
 : [split.seq]          = [ (@ p1 <- p1@split.seq)
                            (@ p2 <- p2@split.seq) ]

split.seq
 : $                    = (@ p1 <- [])
                          (@ p2 <- [])
 : e1 $                 = (@ p1 <- [e1])
                          (@ p2 <- [])
 : e1 e2 split.seq      = (@ p1 <- [e1 . p1@split.seq])
                          (@ p2 <- [e2 . p2@split.seq])
e1 : _

e2 : _
```

Figure 7.13: Definition of *split*

shows the bindings created for these attributes and lastly, it shows the output value. It also shows, above the horizontal line, the matched portion of the input `(a b c d e)`. The unmatched portion of the input is shown under the horizontal line: `alphalessp`.

Effective concepts can be thought of as list structure matching procedures, that attempt to match some prefix of the input while producing translation(s) of them. The process of attempting to match some prefix of the input while producing translation(s) of it is referred to as the *expansion* of an effective concept with certain input. This terminology is intended to emphasise the continuity between the treatment of non-terminals of a grammar in TDPL, as in Section 2.2.2, as string matching procedures, and the concept of effective concepts in META-LISP.

Effective concepts can also be viewed as pure functions. Let X be an effective concept. Its functionality can be given as

$$X\texttt{:input -> <suf, env, val>}$$

where `suf` is the unmatched portion of the input left behind after the expansion, `env` is a set of bindings for the synthesised attributes of the expanded concept, and `val` is its value or principal translation of the input. The value of an effective concept is used as the means of determining whether a given expansion was successful. The notation `< ... >` is used to represent tuples. In many cases, we are only interested in the synthesised attributes if any, and the value of an effective concept. In these cases, it makes sense to talk about the functionality of a given concept in terms of the type of its synthesised attributes if any, and its value. As an example, consider the informal typing of *split.seq*. Its functionality can be given informally as:

```
split.sq: input -> <p1, p2, list>
```

where `p1` and `p2` are also lists. The full functionality would be:

```
split.sq: input -> <suf, <p1, p2>, list>
```

When effective concepts are viewed as pure functions the informal typing is indicated. Whenever explicit *type* information for the set of valid input can be inferred from the form of a META-LISP definition, the informal type of the input will also be indicated, as it has been done throughout the previous Chapter.

7.4.1 Top-Level Elaboration of the Meta-circular Interpreter

The meta-circular interpreter for META-LISP, presented in this chapter, describes the actions required to produce an answer to a META-LISP query, i.e. the process of expansion

of an effective concept with certain input. The interpreter has been successfully applied to interpret itself interpreting another program (including itself) with some input. Self-application has been made possible by indicating in a call to the meta-circular interpreter the number of *levels* of meta-interpretation involved. See Section 7.6.1 for details of why it was necessary, and how does it work. In discussing the top-level elaboration of the meta-circular interpreter it is sufficient to take a note of the fact that information about the number of levels of meta-interpretation that are involved needs to be made available. This is accomplished in the form of the *inherited attribute* `^level`.

The input to the meta-circular interpreter comprises the following: the name of the effective concept being invoked, the name of the module to which it belongs, an integer argument which signifies the level of meta-interpretation, and the input with which the named effective concept has been invoked. Figure 7.14 shows the top level elaboration of the interpreter.

```
mci
% ec -> mod -> level -> input -> <suf, env, val>
 : ec mod level input
  = (^ level)
    (Xec ec mod input (init-env))
    (@ suf <- suf@Xec)
    (@ env <- env@Xec)
    (@ val <- val@Xec)

Xec
% ec -> mod -> input -> glob -> <suf, env, val>
 : ec mod input glob
  =  (^ ec)
     (^ mod)
     (Rules (get-def ec mod) input glob (init-env) (init-facs))
     (@ suf <- suf@Rules)
     (@ env <- env@Rules)
     (@ val <- val@Rules)

Rules
% rules -> input -> glob -> loc -> facs ->  <suf, env, val>
 : Alts
  = (@ suf <- suf@Alts) (@ env <- env@Alts) (@ val <- val@Alts)
 : Left
  = (@ suf <- suf@Left) (@ env <- env@Left) (@ val <- val@Left)
```

Figure 7.14: Top Level Elaboration of the Meta-circular Interpreter

The workhorse of the interpreter is *Xec* which embodies the notion of the expansion of an effective concept of a given module with some input, in a global environment, initially empty, that holds bindings for *inherited attributes*. *Xec* has three synthesised attributes: `suf`, `env`, and `val`. These convey information concerning the unmatched portion, or suffix,

that the expansion of a concept left unmatched, an environment which holds the bindings for synthesised attributes of the expanded concept, and the value of the expansion, respectively. The values associated with these attributes are assigned to the identically named attributes of *mci*. Considered as a function, *mci* has the following, informal functionality:

```
ec -> mod -> level -> input -> <suf, env, val>
```

The co-domain of *mci* is adequate in that it contains all the information necessary for answering a META-LISP query. Note that the codomain of *Xec* is the same. In fact, most rules will have the same co-domain. Copying the values of synthesised attributes will also be a prominent feature of the definitions. [4]

The expansion of an effective concept belonging to a given module, with an input in some global environment is carried out as follows:

1. The name of the given concept is made available as an inherited attribute: `^ec`.
2. Similarly the name of the module to which it belongs is recorded as `^mod`.
3. The rules making up the definition of the concept are retrieved.
4. These are then interpreted as *Rules*.

META-LISP Rules can be either left recursive or are formed of alternatives without left recursive rules. The expansion of alternatives is the subject of the next subsection.

7.4.2 Alternatives

Alternatives in META-LISP, as their name suggest, allow the description of alternative forms of input and their corresponding translation. The form that these alternatives can take can also influence the way further alternatives are considered. Figure 7.15 shows the definition of alternatives.

The interpretation of alternatives makes reference to a second environment, called `loc` which is used to hold the bindings created in the course of interpreting individual rules. It also involves the maintenance of records of expansions that have been completed at any point. These records are used as a mechanism of *left-factoring* (see Section 2.2.1). Note that left-factoring is achieved not by changing the grammar, but by changing the way grammar rules are expanded. As such, this 'optimisation' forms an integral part of the semantics of META-LISP.

[4] It is arguable, whether there should be rules to govern the copying of attribute values implicitly, in place of the present requirement of making them explicit

```
Alts
% alts -> input -> glob -> loc -> facs -> <suf, env, val>
 : [":" synt "=" sem alts] input glob loc facs
  = (if (success? (Synt synt input glob loc facs) ^level)
        { (Sem sem glob env@Synt val@Synt (init-env))
          (@ suf <- suf@Synt)
          (@ env <- sat@Sem)
          (@ val <- val@Sem) }
      { (Alts alts input glob loc facs@Synt)
        (@ suf <- suf@Alts)
        (@ env <- env@Alts)
        (@ val <- val@Alts) })
 : [":" synt "?" sem alts] input glob loc facs
  = (if (success? (Synt synt input glob loc facs) ^level)
        { (Sem sem glob env@Synt val@Synt (init-env))
          (if (success? val@Sem ^level)
              { (@ suf <- suf@Synt) (@ env <- env@Sem) (@ val <- val@Sem) }
            { (Alts alts input glob loc facs@Synt)
              (@ suf <- suf@Alts)
              (@ env <- env@Alts)
              (@ val <- val@Alts) }) }
      { (Alts alts input glob loc facs@Synt)
        (@ suf <- suf@Alts)
        (@ env <- env@Alts)
        (@ val <- val@Alts) })
 : [":" synt alts] input glob loc facs
  = (if (success? (Synt synt input glob loc facs) ^level)
        { (@ suf <- suf@Synt) (@ env <- (init-env)) (@ val <- val@Synt) }
      { (Alts alts input glob loc facs@Synt)
        (@ suf <- suf@Alts)
        (@ env <- env@Alts)
        (@ val <- val@Alts) })
 : '$ input glob loc facs
  = (@ suf <- input) (@ env <- (init-env)) (@ val <- (mk-fail+ ^level))
```

Figure 7.15: Alternatives

7.4.2.1 Alternatives with Default Action

```
[":" synt alts]
```

The simplest form of alternatives is one which consists of a syntax specification and further alternatives. Its interpretation is as follows:

1. Expand the syntax specification with the given input.
2. If this was successful then return the unmatched portion of the input, the bindings and the value produced by the expansion of the syntax description.
3. If this failed, then the remaining alternatives are tried.

Note that the factors produced, in the course of the (ultimately unsuccessful) expansion of the syntax description, are passed on to enable left-factoring to be carried out in the

course of the expansion of the remaining alternatives.

Note also, that deciding whether a given value represents **success** (i.e. not **failure**) requires the number of levels of meta-interpretation to be known. This is because the representation of *failure* itself depends on the level of meta-interpretation.

7.4.2.2 Backtracking Alternatives

```
[":" synt "?" sem alts]
```

1. The syntax description part of an alternative imposes conditions on the input. If these conditions are not satisfied by the input, the expansion of an alternative fails, which results in backtracking and an attempt to reexamine the input using the remaining alternatives.
2. If the expansion of the syntax description in a rule has been successful then the associated semantic action is evaluated.
3. If the semantic action evaluates to **failure** then the remaining alternatives are tried as if the syntax description had failed.
4. If the semantic action evaluates to any other value then the items returned are
   - the unmatched portion of the input produced by the expansion of the syntax specification
   - the bindings for synthesised attributes produced by the evaluation of the semantic action
   - and the value produced by the evaluation of the semantic action

7.4.2.3 Committed Alternatives

```
[":" synt "=" sem alts]
```

The difference between the treatment of Backtracking Alternatives and Committed Alternatives [5] is that step 3 is omitted and step 4 is carried out regardless of the value of the semantic action. That is to say, even if the value of the semantic action happens to denote failure, this value will be returned, and no further alternatives are tried for the currently expanded concept. Backtracking, will however be caused by this at the level of the caller of the currently expanded concept.

7.4.2.4 Exhausting Alternatives

If all alternatives have been tried and all failed, then failure is returned. This amounts to returning the input as the suffix, an empty environment as bindings for attributes, and the right level representation of **failure**.

This concludes the present discussion of the expansion of alternatives.

[5] The idea of 'commitment' in the consideration of alternatives was first suggested by Mark Tarver.

7.4.3 Syntax Rules

There are two kinds of *syntax rules* in META-LISP. The first comprises a non empty sequence of *syntax components*, forming a *composition*. In its form the second kind of syntax rule looks like a composition, except that its first element is one of the keywords `is`, `any` or `with`. The presence of these keywords signify that these rules are not to be treated as a composition, instead they have some special interpretation. Syntax rules of this kind are known as *pseudo rules*. Figure 7.16 shows clearly the above classification of syntax rules.

```
Synt
% synt -> input -> glob -> loc -> facs -> <suf, env, facs, val>
 : Pseudo
  =  (@ suf <- suf@Pseudo)
     (@ env <- env@Pseudo)
     (@ facs <- facs@Pseudo)
     (@ val <- val@Pseudo)
 : Compos
  =  (@ suf <- suf@Compos)
     (@ env <- env@Compos)
     (@ facs <- facs@Compos)
     (@ val <- val@Compos)
```

Figure 7.16: Syntax Rules

7.4.4 Pseudo Rules

The rationale for *pseudo rules* is that they extend the range of structural constraints that can be imposed on the input by ordinary syntax rules. Specifically, they can be used to

- designate LISP predicates to be used as the means of specifying particular properties of the input,
- specify as an admissible first element of the input any one of a collection of objects,
- specify LISP functions to operate on the input instead of effective concepts,
- import effective concepts from other modules.

The interpretation of these rules is given in Figure 7.17.

7.4.4.1 Predication

```
["is" pred]
```

```
Pseudo
 : ["is" pred] input glob loc facs
 = (if (apply pred [(first input)])
       { (@ suf <- (rest input))
         (@ env <- (add-b "is" (first input) (init-env)))
         (@ facs)
         (@ val <- (first input)) }
     { (@ suf <- input)
       (@ env <- (init-env))
       (@ facs)
       (@ val <- (mk-fail+ ^level)) })
 : ["any" objects] input glob loc facs
 = (if (member (first input) objects)
       { (@ suf <- (rest input))
         (@ env <- (add-b "any" (first input) (init-env)))
         (@ facs)
         (@ val <- (first input)) }
     { (@ suf <- input)
       (@ env <- (init-env))
       (@ facs)
       (@ val <- (mk-fail+ ^level)) })
 : ["with" "lisp" fn] input glob loc facs
 =  (@ suf <- □)
    (@ env <- (init-env))
    (@ facs)
    (@ val <- (apply fn input))
 : ["with" module ec] input glob loc facs
 =  (Xec ec module input glob)
    (@ suf <- suf@Xec)
    (@ env <- env@Xec)
    (@ facs)
    (@ val <- val@Xec)
```

Figure 7.17: Pseudo Rules

Predication allows the use of a named LISP predicate [6] to determine whether the first element of the input is to be deemed grammatical. The interpretation of this rules is as follows:

1. *Apply* (in the sense of LISP) the named LISP predicate to the first element of the input.
2. If the result of the application is a non-nil value then, return
   - the input list without its first element, as suffix
   - a new environment in which the keyword `is` is bound to the first element of the input
   - the factors unchanged
   - the first element of the input as the value of the expansion
3. otherwise report failure, by returning
   - the entire input, as suffix
   - the empty environment

[6] a LISP predicate is a function which may return *nil* as its value

- the original factors unchanged
- failure as the value of the expansion

7.4.4.2 Enumeration

```
["any" objects]
```

Enumeration provides the means of specifying any one of a number of given objects as admissible first element of the input.

1. Test if the first element of the input is included in the list of objects given in the rule
2. If the first element is one of these objects then return
   - the input list without its first element, as suffix
   - a new environment in which the keyword `any` is bound to the first element of the input
   - the factors unchanged
   - the first element of the input as the value of the expansion
3. otherwise report failure, by returning
   - the entire input
   - the empty environment
   - the original factors unchanged
   - failure as the value of the expansion

7.4.4.3 LISP Primitives

```
["with" "lisp" fn]
```

This pseudo rule is special in that it does not impose any conditions on the input. It opens a back door to allow LISP functions to be incorporated into META-LISP programs.

1. Apply the LISP function named in the rule to the entire input, and return
   - the empty suffix (i.e. pretend that the entire input has been matched)
   - a new environment in which the keyword `with` is bound to the first element of the input
   - the factors unchanged
   - the result of the application of the named function to the input as the value of the expansion

7.4.4.4 Importing

The name-space of META-LISP is partitioned into modules. Every effective concept at the point of definition is made to belong to some module. The last pseudo rule provides the means of *importing* the functionality of a named concept from an other module.

```
["with" module ec]
```

1. Expand the effective concept named in the rule from the given module on the input, and return
   - the suffix of the expansion
   - the bindings created by the expansion
   - the factors unchanged
   - the result of the expansion

7.4.5 Composition

The structure of the input is specified in terms of a non-empty sequence of syntax components forming a composition. The process of expansion of a composition provides the means of establishing whether some prefix of the input is grammatical. The expansion of a composition in META-LISP plays the role of a syntax-directed parameter passing mechanism. Figure 7.18 shows that this process is recursive. It also shows that the terminating case is reached when the the last component is to be expanded next.

```
Compos
% compos -> input -> glob -> loc -> facs -> <suf, env, facs, val>
 : [comp '$] input glob loc facs
  = (if (success? (Comp comp input glob facs) ^level)
        { (@ suf <- suf@Comp)
          (@ env <- (add-bs env@Comp loc))
          (@ facs <- facs@Comp)
          (@ val <- val@Comp) }
      { (@ facs <- facs@Comp) (@ val <- (mk-fail+ ^level)) })
 : [comp compos] input glob loc facs
  = (if (success? (Comp comp input glob facs) ^level)
        { (Compos compos suf@Comp glob (add-bs env@Comp loc) facs@Comp)
          (@ suf <- suf@Compos)
          (@ env <- env@Compos)
          (@ facs <- facs@Compos)
          (@ val <- val@Compos) }
      { (@ facs <- facs@Comp) (@ val <- (mk-fail+ ^level)) })
```

Figure 7.18: Composition

7.4.5.1 Composition: General Case

```
[comp compos]
```

The expansion of a composition comprising more then one syntax components proceeds as follows:

1. The first component in the composition is expanded
2. If this was successful, then
   - Expand the remaining components in the composition with
     - the suffix of the successful expansion of the first component as the input.
     - the global bindings unchanged
     - local environment extended to include the bindings created in the course of the successful expansion of the first component
     - new, possibly extended set of factors produced in the course of the expansion of the first component
   - then return the suffix, the extended local environment, the factors and the value of the expansion of the remaining components
3. If the expansion of the first component was not successful, then return
   - the factors returned by the unsuccessful expansion of the first component (this will include a record of this failure)
   - failure as the value of the expansion of the composition

7.4.5.2 Composition: Terminating Case

```
[comp '$]
```

The expansion of a composition comprising a single syntax component proceeds as follows:

1. Expand the given component with the given input
2. If this was successful, then return the suffix, the extended local environment, the factors and the value of the expansion of the component as suffix etc. of the composition
3. if the expansion of the single component was not successful, then return
   - the factors returned by the unsuccessful expansion of the component
   - failure as the value of the expansion of the composition

7.4.6 Syntax Component

Components can be characterised as one of two kinds: *elementary* and *non-elementary* components. Elementary components are so called because their action on the input can be defined without reference to other effective concepts. Non-elementary components, in contrast depend for their definition on other effective concepts. Figure 7.19 shows the definition of components. Elementary components offer grammatical means which go beyond the matching of terminal symbols. They offer extra language definitional capabilities which have been found invaluable in language oriented programming. The inclusion of *nested structures* into the grammatical formalism has been motivated also by their usefulness and convenience in writing language oriented programs.

7.4.6.1 Prefix

"_"

The limiting case of syntax-directed parameter passing is pattern matching. This feature provides the mechanism for unconditional acceptance of the first element of the input. In effect, its role can be likened to that of pattern variables. Note that if the input is empty then the empty list is returned.

1. Always succeeds, and returns
    - the rest of the input as suffix
    - a new environment in which the keyword "_" is bound to the first element of the input
    - the factors unchanged
    - the first element of the input as the value of the expansion

7.4.6.2 Suffix

"._"

This form of syntax component provides the means of passing the entire input as a parameter.

1. Always succeeds, and returns
    - the empty input as suffix
    - a new environment in which the keyword "._" is bound to the entire input
    - the factors unchanged
    - the entire input as the value of the expansion

7.4.6.3 Empty

"<>"

This feature of the grammatical formalism of META-LISP corresponds to the empty production of standard grammatical formalism.

1. Always succeeds, and returns
   - the entire input as suffix
   - an empty, new environment
   - the factors unchanged
   - the empty list as its value

7.4.6.4 End of Input Test

"$"

This feature of the grammatical formalism of META-LISP corresponds to the use of *endmark* in parsing.

1. Tests if the input is empty
2. If it is, then returns
   - the empty input as suffix
   - a new, empty environment
   - the factors unchanged
   - the empty list as its value

```
Comp
% comp -> input -> glob -> facs -> <suf, env, facs, val>
 : "_" input glob facs
  =  (@ suf <- (if input (rest input) [])
     (@ env <- (add-b "_" (if input (first input) []) (init-env)))
     (@ facs)
     (@ val <- (first input))
 : "._" input glob facs
  =  (@ suf <- [])
     (@ env <- (add-b "._" input (init-env)))
     (@ facs)
     (@ val <- input)
 : "<>" input glob facs
  = (@ suf <- input) (@ env <- (init-store)) (@ facs) (@ val <- [])
 : "$" input glob facs
  = (if (null? input)
        { (@ suf <- []) (@ env <- (init-env)) (@ facs) (@ val <- []) }
      { (@ suf <- []) (@ env <- (init-env)) (@ facs) (@ val <- (mk-fail+ ^level)) })
 : Denot input glob facs
  = (if (equal Denot (first input))
        { (@ suf <- (rest input)) (@ env <- (init-env)) (@ facs) (@ val <- (first input)) }
      { (@ suf <- input) (@ env <- (init-env)) (@ facs) (@ val <- (mk-fail+ ^level)) })
 : Ec input glob facs
  = (if (present? (get-factor Ec facs) input)
        { (@ suf <- suf@present?)
          (@ env <- (add-b Ec val@present? env@present?))
          (@ facs)
          (@ val <- val@present?) }
      { (Xec Ec ^mod input glob)
        (@ suf <- suf@Xec)
        (@ env <- (add-b Ec val@Xec env@Xec))
        (@ facs <- (add-factor (mk-factor Ec suf@Xec env@Xec val@Xec (first input)) facs))
        (@ val <- val@Xec) })
 : ["[" '$ "]"] input glob facs
  = (if (null? (first input))
        { (@ suf <- (rest input)) (@ env <- (init-env)) (@ facs) (@ val <- (first input)) }
      { (@ suf <- input) (@ env <- (init-env)) (@ facs) (@ val <- (mk-fail+ ^level)) })
 : ["[" compos "]"] input glob facs
  = (if (atom? (first input))
        { (@ suf <- input) (@ env <- (init-env)) (@ facs) (@ val <- (mk-fail+ ^level)) }
      (if (success? (Compos compos (first input) glob (init-env) facs) ^level)
          (if (null? suf@Compos)
              { (@ suf <- (rest input))
                (@ env <- env@Compos)
                (@ facs <- facs@Compos)
                (@ val <- val@Compos) }
            { (@ suf <- input)
              (@ env <- (init-env))
              (@ facs <- facs@Compos)
              (@ val <- (mk-fail+ ^level)) })
        { (@ suf <- [])
          (@ env <- (init-env))
          (@ facs <- facs@Compos)
          (@ val <- (mk-fail+ ^level)) }))
```

Figure 7.19: Syntax Component

7.4.6.5 Denotation

The concept of terminal productions is made more machine oriented in META-LISP by the introduction of the notion of *denotation* which covers not only the matching of arbitrary objects, but strings and keywords as denoting themselves. The forms that denotations can take is defined in Figure 7.20. The interpretation of denotations is the same regardless of their specific form:

```
Denot
 : ["'" object]
  = object
 : String
 : Keyword

String
 : [:string stringp]
  = stringp

Keyword
 : is keywordp

Ec
 : identifier
```

Figure 7.20: Denotation

1. Test if the first element of the input is the same as the given denotation.
2. If it is the case, then return
   - the input list without its first element, as suffix
   - a new, empty environment
   - the factors unchanged
   - the first element of the input as the value of the expansion
3. If the first element of the input is not the same as the given denotation, then return
   - the entire input, as suffix
   - the empty environment
   - the original factors unchanged
   - failure as the value of the expansion

7.4.6.6 Constituent Effective Concept

Ec

The real definitional power of composition derives from the fact that it can have effective concepts as components. The invocation of *constituent effective concepts* in a composition

in a rule for a given effective concept clearly involves a recursive call to the workhouse of the meta-circular interpreter *Xec*. It is at this point that a record of the expansion of a concept is created and used. The expansion of a constituent concept proceeds as follows:

1. Test if the given concept has been expanded earlier with the same input.
2. If it is the case then return
   - the recorded suffix
   - extend the bindings for synthesised attributes retrieved from the record of the previous expansion of the concept with a binding for the name of the concept to its recorded value
   - the factors unchanged
   - the value of the recorded expansion
3. If there is no record of a previous expansion of the given concept with the same input as the current input, then
   - expand the concept with the current input, and
   - return the suffix of the expansion
   - extend the bindings for synthesised attributes produced by the expansion with a binding for the given concept to the value of the expansion
   - add to the record of previous expansions a new record comprising all the necessary information about the expansion of the concept. The information recorded include, the suffix, the bindings, the value returned by the expansion. In addition to these the first element of the input is also recorded.
   - finally, the value of the recorded expansion is returned

The maintenance of appropriate records of previous expansions is a form of *memoisation*. Recoring information about the input is equivalent to recording the arguments to a function when it is memoised. It is an open question whether it is sufficient to record only the first element of the input for this purpose. It may be that more is needed. Or even, perhaps that the entire input needs to be examined. The latter, in some circumstances, may introduce intolerable overheads. This requires further investigation.

7.4.6.7 Nested Composition

```
["[" compos "]"]
```

META-LISP allows the grammatical description of arbitrary nested list structures.

1. If the first element of the input is not a list, then report failure in the usual manner
2. If the first element of the input was indeed a list, then
   - expand the composition enclosed in square brackets with the first element of the input as input
   - if the expansion of the composition was successful,then

  - if the expansion exhausted its input, then
    * return as the suffix of the expansion of the nested composition the original input less its first element, since, as the test shows, the first element has been matched completely.
    * return the bindings, the factors and the value produced by the expansion of the nested composition in the usual manner
  - if the expansion did not exhaust its input, then return
    * the entire original input as suffix
    * the empty environment
    * the factors returned by the expansion of the composition
    * failure as the value of the expansion of nested composition
- if the expansion of the composition failed, then report failure, but pass on the factors produced by the expansion as well

7.4.7 Left Recursion

The form of left recursion provided in META-LISP is limited to *direct left recursion.* It is useful both as the means of specifying iteration as well as the definition of language constructs that involve left associativity. Figure 7.21 shows the meta-circular definition of this construct.

```
Left
: [start rec] input glob loc facs
 =  (Start start input glob loc facs)
    (if (success? value@Start ^level)
        { (Rec rec suf@Start glob (add-b ^ec val@Start env@Start) val@Start)
          (@ suf <- suf@Rec)
          (@ env <- env@Rec)
          (@ val <- val@Rec) }
      { (@ suf <- input) (@ env <- loc) (@ val <- (mk-fail+ ^level)) })

Start
 : Alts
  = (@ suf <- suf@Alts) (@ env <- env@Alts) (@ val <- val@Alts)

Rec
 : alts input glob loc val
  =  (Alts alts input glob loc (init-facs))
    (if (success? val@Alts ^level)
        { (Rec alts suf@Alts glob (add-b ^ec val@Alts env@Alts) val@Alts)
          (@ suf <- suf@Rec)
          (@ env <- env@Rec)
          (@ val <- val@Rec) }
      { (@ suf <- input) (@ env <- loc) (@ val) })
```

Figure 7.21: Left Recursion

The abstract syntax of left recursive rules was designed to facilitate their interpretation. The distinction drawn between left recursive and non-left recursive rules is crucial. Recall

that the abstract syntax representation of left recursive alternatives omits the left recursive calls from the rules (see Section 7.2.2). The non-left recursive alternatives, are used to produce a start-up value for the left recursive effective concept. For this reason they are referred to as *start-up* rules. The suitably transformed left recursive alternatives are then expanded repeatedly in an environment in which there are bindings for the left recursive concept, – initially using the values produced by the expansion of the startup rules – to the value of the previous iteration.

1. The start-up rules are expanded as alternatives
2. If this expansion was successful, then
   - the recursive alternatives are expanded
     - with input that was left unmatched by the expansion of the startup rules
     - in a local environment that consists of the bindings produced by the expansion of the start-up rules, extended to include the value of the start-up rules bound to the name of left recursive concept being expanded
     - the value of the expansion of the start-up rules is also passed as a parameter
   - the suffix, the environment and the value produced by the expansion of recursive alternatives are then returned
3. if the expansion of the start-up rules failed, then report failure

7.4.7.1 Left Recursive Alternatives

1. The alternatives are expanded
2. If the expansion was successful then the expansion of left recursive alternatives is repeated with
   - input left unmatched by the expansion of the alternatives
   - in a local environment in which name of the left recursive concept being expanded is bound to the value of the alternatives
   - the value of the of the expansion of the alternatives
3. if the expansion of the alternatives fails, then return
   - the input as the unmatched prefix
   - the local environment, which holds the bindings created in the previous successful expansion of the left recursive alternatives
   - and the value of the previous expansion

7.5 The Semantics of META-LISP: Part II

This Section presents the semantics of the language of Semantic Actions. An important property of the language of semantic actions is that the value of an expression of the language is determined solely in terms of its constituent parts. That is to say, it is an applicative language [Hen80, 7].

7.5.1 Semantic Actions

Figure 7.22 show the top-level elaboration of the interpreter for semantic actions. It maps semantic actions into three values: bindings for inherited attributes and synthesised attributes and the value of the evaluation of the semantic action. The valuation of semantic actions takes place in the context of environments that contain bindings created in the course of the expansion of the grammar rule with which the semantic action is associated, inherited and synthesised attributes. The latter two can also be added to in the course of the evaluation. Semantic actions comprise a non-empty sequence of semantic terms. The meaning of semantic actions is therefore given in terms of the meaning of these terms.

```
Sem
% sem -> iat -> loc -> val -> sat -> <iat, sat, val>
 : sterms iat loc val sat
  =  (Sterms sterms iat loc val sat)
     (@ iat <- iat@Sterms)
     (@ sat <- sat@Sterms)
     (@ val <- val@Sterms)
```

Figure 7.22: Semantic Action

7.5.2 Semantic Terms

Semantic terms are evaluated in a sequence. The bindings created by the evaluation of one semantic term can be referenced in the course of the evaluation of subsequent terms.

7.5.3 Semantic Terms: General Case

`[sterm sterms]`

1. Evaluate the given semantic term
2. Evaluate the remining semantic terms in the context of new environments returned by the evaluation of the semantic term
3. Return the environments and the value resulting from the evaluation of semantic terms.

```
Sterms
% sterms -> iat -> loc -> val -> sat -> <iat, loc, sat, val>
 : '$ iat loc val sat
  = (@ iat) (@ loc) (@ sat) (@ val)
 : [sterm sterms] iat loc val sat
  =  (Sterm sterm iat loc val sat)
     (Sterms sterms iat@Sterm loc@Sterm val@Sterm sat@Sterm)
     (@ iat <- iat@Sterms)
     (@ loc <- loc@Sterms)
     (@ sat <- sat@Sterms)
     (@ val <- val@Sterms)
```

Figure 7.23: Semantic Terms

7.5.4 Semantic Terms: Terminating Case

```
'$
```

If there are no more semantic terms in the sequence then return the current values of the

1. environments for
   - inherited attributes
   - local bindings
   - synthesised attributes
2. end the value of the last semantic term

7.5.5 Semantic Term

There are four basic mechanisms used to build up semantic terms. These are

1. sequencing
2. attribute assignments
3. invocation of semantic functions (effective concepts or the choice function `if`)
4. construction of list structures

Figure 7.24 shows the appropriate evaluation rules. These will be considered, one by one in the following subsections.

7.5.5.1 Sequencing

```
["{" sterms "}"]
```

A non-empty sequence of semantic terms enclosed in a pair of curly brackets is a semantic term. It is evaluated as a semantic action.

7.5.5.2 Synthesised Attributes

```
["(" "@" id "<-" sterm ")"]
```

Synthesised attributes can be assigned the value of a semantic term.

1. The semantic term given in the attribute assignment is evaluated
2. Bindings returned by the evaluation of the semantic term are returned unchanged with the exception of bindings for synthesised attributes
3. a new binding is added to the set of bindings for synthesised attributes which binds the value of the semantic term to an identifier constructed out of the given attribute name and the name of the effective concept being expanded

7.5.5.3 Default Synthesised Attributes

```
["(" "@" id ")"]
```

The above form of synthesised attribute assignment is a shorthand for the equivalent semantic term

```
["(" "@" id <- id ")"]
```

which can then be interpreted as a synthesised attribution.

7.5.5.4 Inherited Attributes

```
["(" "^" id "<-" sterm ")"]
```

Inherited attributes can be assigned the value of a semantic term.

1. The semantic term given in the attribute assignment is evaluated
2. Bindings returned by the evaluation of the semantic term are returned unchanged with the exception of bindings for inherited attributes
3. a new binding is added to the set of bindings for inherited attributes which binds the value of the semantic term to an identifier constructed out of the a *caret* ^ and the given attribute name.

The real difference between synthesised and inherited attributes is that the latter are passed on as global bindings for in subsequent invocations of effective concepts, whereas synthesised attributes represent information flow in the opposite direction.

7.5.5.5 Default Inherited Attributes

```
["(" "^" id ")"]
```

The above form of synthesised attribute assignment is a shorthand for the equivalent semantic term

```
["(" "^" id <- id ")"]
```

which can then be interpreted as an inherited attribution.

7.5.5.6 Choice function

```
["(" "if" bool c1 c2 ")"]
```

The semantics of the *choice function* is the standard one. That is to say only one arm of the conditional will be evaluated. Care has been taken to write the evaluation rule in such a way that it is independent of whether META-LISP has this property or not. What it assumes, however, is e left-to-right evaluation order. The trick used is to have two conditionals doing the job of one.

1. The boolean term is evaluated first
2. If it evaluates to a non-nil value then the first arm of the conditional is evaluated.
3. If it evaluates to nil then the first arm is not evaluated, only the second
4. the bindings and the value created in the course of the evaluation of one of the appropriate branch of the conditional is returned.

Note that the bindings created by the evaluation of the boolean term are passed to the evaluation of the branches of the choice function.

7.5.5.7 Invocation

```
["(" sterm els ")"]
```

The ability of invoking effective concepts in the semantic actions is the most important feature of META-LISP. This feature is responsible for META-LISP's ability to provide linguistic support for the language oriented paradigm. The way it is being defined relies on the left-to-right evaluation order of META-LISP. It seems hard to avoid this. It should be regarded as one of the fundamental, irreducible properties of the language.

```
Sterm
 : ["{" sterms "}"] iat loc val sat
  =  (Sterms sterms iat loc val sat)
     (@ iat <- iat@Sterms)
     (@ loc <- loc@Sterms)
     (@ sat <- sat@Sterms)
     (@ val <- val@Sterms)
 : ["(" "@" id "<-" sterm ")"] iat loc val sat
  =  (Sterm sterm iat loc val sat)
     (@ iat <- iat@Sterm)
     (@ loc <- loc@Sterm)
     (@ sat <- (add-b (mk-sattr-name id ^ec) val@Sterm sat@Sterm))
     (@ val <- val@Sterm)
 : ["(" "@" id ")"] iat loc val sat
  =  (Sterm ["(" "@" id "<-" id ")"] iat loc val sat)
     (@ iat <- iat@Sterm)
     (@ loc <- loc@Sterm)
     (@ sat <- sat@Sterm)
     (@ val <- val@Sterm)
 : ["(" "^" id "<-" sterm ")"] iat loc val sat
  =  (Sterm sterm iat loc val sat)
     (@ iat <- (add-b (mk-iattr-name id) val@Sterm iat@Sterm))
     (@ loc <- loc@Sterm)
     (@ sat <- sat@Sterm)
     (@ val <- val@Sterm)
 : ["(" "^" id ")"] iat loc val sat
  =  (Sterm ["(" "^" id "<-" id ")"] iat loc val sat)
     (@ iat <- iat@Sterm)
     (@ loc <- loc@Sterm)
     (@ sat <- sat@Sterm)
     (@ val <- val@Sterm)
 : ["(" "if" bool c1 c2 ")"] iat loc val sat
  =  (Sterm bool iat loc val sat)
     (if val@Sterm (Sterm c1 iat@Sterm loc@Sterm val@Sterm sat@Sterm) [])
     (if val@Sterm []  (Sterm c2 iat@Sterm loc@Sterm val@Sterm sat@Sterm))
     (@ iat <- iat@Sterm)
     (@ loc <- loc@Sterm)
     (@ sat <- sat@Sterm)
     (@ val <- val@Sterm)
 : ["(" sterm els ")"] iat loc val sat
  =  (Xec
       (Sterm sterm iat loc val sat)
       ^mod
       (Els els iat@Sterm loc@Sterm val@Sterm sat@Sterm)
       iat@Els)
     (@ iat <- iat@Els)
     (@ loc <- (add-bs env@Xec loc@Els))
     (@ sat <- sat@Els)
     (@ val <- val@Xec)
 : ["[" els "]"] iat loc val sat
  =  (Els els iat loc val sat)
     (@ iat <- iat@Els)
     (@ loc <- loc@Els)
     (@ sat <- sat@Els)
     (@ val <- val@Els)
 : ["<" ec ">"] iat loc val sat
  = (@ iat) (@ sat) (@ loc) (@ val <- ec)
 : Denot iat loc val sat
  = Denot (@ iat) (@ sat) (@ loc) (@ val <- Denot)
 : Id iat loc val sat
  = (@ iat) (@ sat) (@ loc) (@ val <- (lookup Id iat loc sat))
 : Number iat loc val sat
  = (@ iat) (@ sat) (@ loc) (@ val <- Number)
 : failure? iat loc val sat
  = (@ iat) (@ sat) (@ loc) (@ val <- (mk-fail+ ^level))
```

Figure 7.24: Semantic Term

1. The semantic term given as the *function term* is evaluated first. It is assumed to evaluate to the name of a currently defined concept belonging to the current module. Exception handling could be introduced in the definition of *Xec* if desired.
2. The bindings created are passed on to the evaluation of the input elements.
3. Then the concept – named by the value of the function term – is expanded in the current module with input computed before.
4. Note that the inherited attributes returned by the evaluation of the input elements are passed as a global environment to the expansion of the name effective concept.

Note that the interpretation of input elements, given on page 178 will give the right interpretation of dotted invocation, see page 56 without the need for special rules.

7.5.5.8 List Construction

```
["[" els "]"]
```

In the invocation of effective concept as semantic functions, described above, the input elements were made to form the input list with the implicit use of list constructions. Elements enclosed by a pair of square brackets designate list-construction. The interpretation of elements is given on page 178.

7.5.5.9 Conceptual Value

```
["<" ec ">"]
```

META-LISP treats effective concepts as first class objects. This feature is used to indicate that a return value, an identifier is taken to be the name of an effective concept belonging to the current module.

7.5.5.10 Denotation

```
Denot
```

Denotations *denote* themselves.

7.5.5.11 Identifiers

```
Id
```

Identifiers may have bindings in any one of the current environments. If they are not bound then they are assumed to denote themselves. This feature corresponds to *auto-quote* in some dialects of LISP. The semantic function *lookup* is used to test if a given identifier is bound, if it is than that binding is returned. If there is no binding the identifier itself is returned as a value.

7.5.5.12 Number

Number

Numbers denote themselves.

7.5.5.13 Failure

`failure?`

The treatment of failure as a semantic value is problematic. Care is needed to distinguish failure at the meta-level from failure at the object level. Failure as a value of a semantic actions needs to be distinguished from, say *Sterm* returning failure, because, say the input to be interpreted was not grammatical.

As the means of distinguishing between object level and meta level notions of failure, a whole hierarchy of failures have been introduced. The need for this has become apparent in meta-meta-interpretation, i.e. when the interpreter was used to interpret itself interpreting another program, (which could itself be the interpreter interpreting another program), etc. The definitions used to make this scheme work are shown in Figure 7.25. The solution offered at the present to the problems related to failure requires further investigations.

7.5.6 Elements

A sequence of elements can be combined to form a list. The construction of this list allows *splicing* specified elements into the list being constructed.

7.5.6.1 *Cons*-ing an Element into a List

`[sterm els]`

The basic method of constructing a list of elements is to evaluate an element and constructing a list with that element as its first element and the rest of the list formed of the values of the remaining elements.

1. Evaluate the element.
2. Evaluate the remaining elements in an environment which holds the bindings created in the course of evaluating the first element.
3. Return the bindings created in the course of evaluating the remaining elements
4. return as the value a list with the value of the first element and the list value of the remaining elements as its tail.

7.5.6.2 Splicing an element into a List

```
["." sterm els]
```

An elment is *spliced* into a list of elements by *appending* its value to the list of the values of the remaining elements.

1. Evaluate the element.
2. Evaluate the remaining elements in an environment which holds the bindings created in the course of evaluating the first element.
3. Return the bindings created in the course of evaluating the remaining elements
4. return as the value a list with the value of the first element *appended* to the list value of the remaining elements.

7.5.6.3 Elements: Terminating Case

```
'$
```

The empty sequence of elements evaluates to the empty list.

This completes the present account of the meta-circular interpreter for META-LISP. Figure 7.27 shows the elementary definitions that were used.

```
    failure?
     : stringp
      = (if (equal stringp (mk-fail ^level)) yes (mk-fail+ ^level))

    success?
     : with lisp success?

    mk-fail
     : with lisp mk-fail

    mk-fail+
     : with lisp mk-fail+

    level
     : is integerp
     : <>                       = 0


(defun success? (result level)
  (not (failed? result  (1+ level))))

(defun failed? (result level)
  (equal result (mk-fail level)))

(defun mk-fail (level)
 (format nil "fail!~A" level))

(defun mk-fail+ (level) (mk-fail (1+ level)))
```

Figure 7.25: Dealing with Failure

```
    Els
    % els -> iat -> loc -> val -> sat -> <iat, loc, sat, val>
     : '$ iat loc val sat
      = (@ iat) (@ loc) (@ sat) (@ val <- [])
     : ["." sterm els] iat loc val sat
      =  (Sterm sterm iat loc val sat)
         (Els els iat@Sterm loc@Sterm val@Sterm sat@Sterm)
         (@ iat <- iat@Els)
         (@ loc <- loc@Els)
         (@ sat <- sat@Els)
         (@ val <- (append val@Sterm val@Els))
     : [sterm els] iat loc val sat
      =  (Sterm sterm iat loc val sat)
         (Els els iat@Sterm loc@Sterm val@Sterm sat@Sterm)
         (@ iat <- iat@Els)
         (@ loc <- loc@Els)
         (@ sat <- sat@Els)
         (@ val <- (cons val@Sterm val@Els))
```

Figure 7.26: Semantic Elements

```
add-factor : with lisp cons
all-up-case : with lisp all-up-case
alts : _
aname : _
anything : _
append : with lisp append
apply : with lisp apply
atom? : with lisp atom
bindings : _
bool : _
c1 : _
c2 : _
comp : _
compos : _
concat : with lisp concat
cons : with lisp cons
ec : identifier
els : _
env : _
equal : with lisp equal
explode : with lisp explode
facs : _
first : with lisp first
fn : _
get-def : with lisp get
get-factor : with lisp assoc
glob : _
head : _
iat : _
id : _
identifier : is identifier
if : with lisp if
implode : with lisp implode
init-env : <>
init-facs : <>
init-store : <>
input : _
intern : with lisp intern
Id : is identifier

item
 : $
  = fail!
 : anything

key
 : identifier : stringp
list : ._
list2set : with lisp list2set
loc : _
member : with lisp memeql
mk-factor : list

mk-iattr-name
 : ec = (implode (cons '#\^ (explode ec)))

mk-sattr-name
 : aname ec
  = (if (all-up-case (append (explode aname) (explode ec)))
        (intern (concat (string aname) "@" (string ec)))
      (intern (concat (my-symbol-name aname) "@" (my-symbol-name ec))))
```

```
mod : is keywordp
module : is keywordp
my-symbol-name : with lisp my-symbol-name
null : is null

null?
 : null
  = t
 : <>

object : item

objects
 : object objects
  = [object . objects]
 : object
  = [object]

pred : _

present?
 : null
 : [ec suf env val head] input
  = (if (equal (first input) head) { (@ suf) (@ env) (@ val) (@ head) ec } [])

rec : _
rest : with lisp cdr
sat : _
sem : _
start : _
sterm : _
sterms : _
string : with lisp string
stringp : is stringp
suf : _
synt : _
val : _
```

Figure 7.27: Elementary Definitions

7.6 Discussion

This section presents some of the experiments with the meta-circular interpreter carried out to date. It concludes with a brief discussion of some of the lessons learned.

7.6.1 Meta-Interpreting the Meta-circular Interpreter

One of the advantages of developing a meta-circular interpreter is that self-application provides an extensive test of its capabilities. The programs in Chapter 6 (including the lexical analyser, the abstractor, and the interpreter, were successfully interpreted by the meta-circular interpreter. In fact, the program was run by running the meta-interpreter, running

the meta-interpreter running the calculator program. Each level of meta-interpretation has increased the overall runtime by two orders of magnitude. Since three levels of interpretation involves 100×100, i.e. four orders of magnitude increase in run-time only a very simple program have been run to illustrate meta-interpretation[3]. The approximate runtimes were 0.2, 20 and 2000 seconds respectively.

```
| ?= (mci mci :mci 1 (mci :mci 2 (expr :expr 3 (1 * 2 + 4))))

| mci :mci 1 (mci :mci 2 (expr :expr 3 (1 * 2 + 4))))
|------------
|

suf@mci = ()
env@mci = ()
val@mci = (+ (* 1 2) 4)

mci = (+ (* 1 2) 4)
```

7.6.2 Reflections

Reflecting on the experience of writing a denotational style interpreter in META-LISP I would like to make the following observations:

- Writing any description of the semantics of a non-trivial language is difficult.
- working out the details in full is not only the object of the exercise but it is also the primary means of arriving at a satisfactory definition of the language
- The warning given by Tofte is highly appropriate: "Hacking a machine readable language definition must never be seen as a substitute for thinking seriously about the semantics of the language."[Tof90, page 111]
- Writing denotational style interpreters in META-LISP clearly indicates the desirability of type checking.
- The meta-circular interpreter will be the starting point for the development of a type-discipline for META-LISP.

I would like to conclude this chapter by quoting Tofte's apt admonition:

> The best one can do when writing proofs (or compilers or compiler generators, for that matter) is to care. Only from feeling of having been careful can some level of confidence emerge. [Tof90, page 111]

Chapter 8

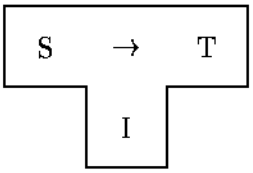
Implementation

This chapter describes the implementation of the programming language META-LISP and its associated programming environment. The chapter is divided into three sections. The first Section outlines the stages in the development of the implementation of the language. Section two discusses the implementation of the programming environment. Section three discusses directions for further development.

8.1 Implementing META-LISP

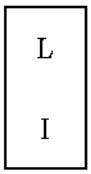
The implementation strategy that relies on the facilities provided by a language to compile itself is called *bootstrapping.* [ASU86, 725]. Bootstrapping is a particularly attractive strategy for the initial implementation of a new programming language. To begin with, a compiler is developed for only a subset of the intended new language. This minimal language is implemented with minimum sophistication in a short order. The addition of further constructs and capabilities can then be carried out in a piecemeal fashion, using bootstrapping. A further advantage of bootstrapping follows from the fact that a compiler for a programming language, itself is a complex program. Having to write such a program in its own language, can help the designer of the language to refine the facilities provided by the language. In addition, the compilation of a compiler written in its own language provides a useful benchmark for (regressive) testing of the compiler itself.

A compiler for a programming language is characterised by three languages: the source language S that it compiles, the target language T that it generates, and the implementation language I that it is written in. These three languages that characterise a compiler are usually represented in a diagram forming a T, called a *T-diagram.*[Bra61]



The notation used to refer to a compiler with the three languages S,T and I is to write $S \underset{I}{\rightarrow} T$.

Bootstrapping poses the problem: how to obtain the first compiler, to begin with. One possible approach is to hand-compile a compiler for a subset of the language into the implementation language to obtain the first compiler for the language. An alternative approach, the one that was adopted here, is to construct an interpreter for a subset of the language first, and to use that interpreter as the means of running the compiler. An interpreter is characterised by two languages: the language that it interprets L, and the language that it is implemented in I. It is represented diagrammatically as



For the above scheme to work, the interpreter itself had to be written in a language that is already implemented on a machine. The diagrammatical notation for representing a machine which executes a (machine) language is show below:

These boxes can be composed to form description of systems involving compilers, interpreters, any kinds of translators, in fact, and machines capable of executing programs in a given language. Placing these boxes on top of each other denotes interpretation. Placing them adjacent to each other horizontally denotes translation.

In implementing META-LISP, LISP was used both as the target language of its compiler and the language in which the first interpreter for a subset of the language was written. [1] The subset of META-LISP, for which the first compiler was constructed, is referred to as MtL^0. A compiler for it for was constructed as follows:

- First, an interpreter was written in LISP for MtL^0.

[1] Given LISP's reputation as the "machine language" for Artificial Intelligence, this choice is very natural. [All78, 243]

- Then a compiler (${MtL^0}_{\vec{LISP}}LISP$) was written for this subset in itself designed to generate LISP code
- a compiler for MtL^0 to LISP in LISP, ${MtL^0}_{\vec{LISP}}LISP$, was finally obtained by running the compiler for MtL^0 written in itself, ${MtL^0}_{\vec{MtL^0}}LISP$, using the interpreter for MtL^0 written in LISP, i.e. ${MtL^0}_{\vec{LISP}}LISP$

This process is illustrated in Figure 8.1. It shows a run of the translator ${MtL^0}_{\vec{MtL^0}}LISP$ with the definition of itself ${MtL^0}_{\vec{MtL^0}}LISP$ as input producing a LISP implementation of the translator for MtL^0 into LISP.

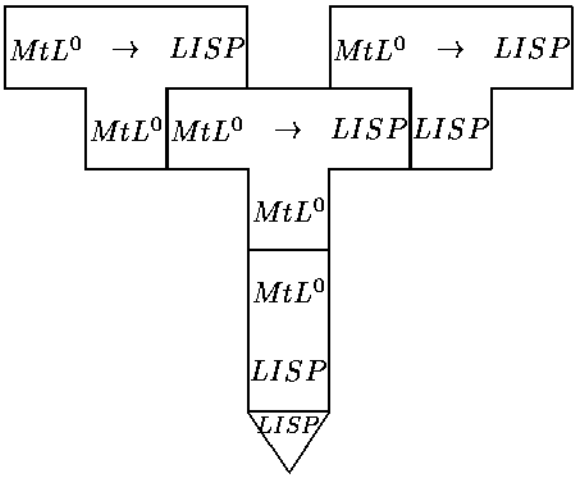


Figure 8.1: The Construction of the First Compiler

MtL^0 included the following features of META-LISP:

- top-down limited backtrack translation
- description of nested list structures
- lisp forms as semantic actions with the addition of the feature of invoking MtL^0 translation procedures as semantic functions

The compiler generated by this process made the original interpreter superfluous. Although the performance of the generated code was an improvement over the interpreter, it was still hopelessly inefficient. The expressive power of MtL^0 was also painfully restrictive at that point. Bootstrapping was used both as the means of improving the performance of the implementation as well as introducing new features into the language.

8.1.1 Extensions

The introduction of a set of new language features using bootstrapping involves the following steps:

- write a new compiler for the extended language (MtL') in the currently implemented form of the language (MtL).
- compile it with the existing compiler to obtain an implementation of the described new extensions of the language
- rewrite the new compiler in the new, extended language MtL'
- compile it with the new compiler

As the result of the last step a the implementation reaches a new *meta-stable state*, i.e. when the compiler obtained in the last step compiles itself it produces a copy of itself. This also implies that the language implementation becomes independent of any previous compilers. This process is illustrated by Figure 8.2:

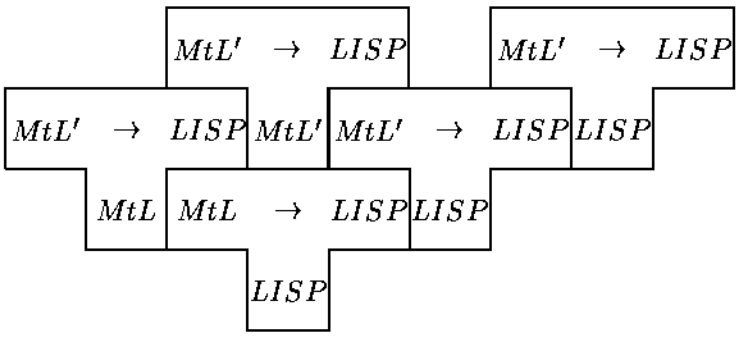


Figure 8.2: Extending the Language

The above process was repeated four times over the development of META-LISP to introduce the following features:

1. left-recursion and left-factoring
2. list-construction
3. attributes
4. 'conceptual' parameters and importing effective concepts from other packages

8.1.2 Optimisation

Improving the performance of the generated LISP code involved the following steps:

- write a new compiler that generates more efficient code ($LISP'$) in the currently implemented form of the language (MtL).
- compile it with the existing compiler to obtain a new optimised compiler, which itself is still implemented in a less optimal way.
- compile the definition of the new optimised compiler again, this time with the previously obtained "hybrid" compiler, to obtain an optimised compiler which is also implemented in a more efficient way then before.

The result of the last step, again, is a meta-stable implementation. The procedure for obtaining a new, optimised form of the META-LISP compiler is illustrated in Figure 8.3:

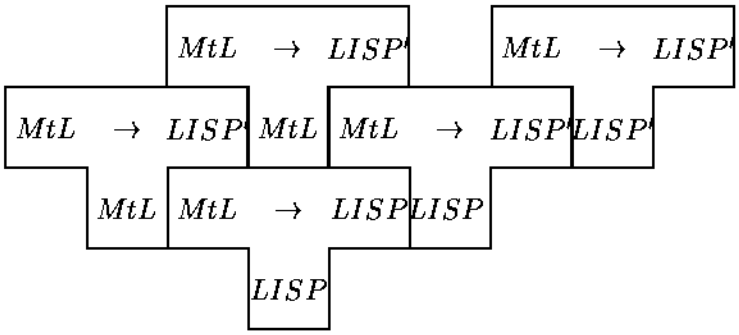


Figure 8.3: Optimising the Implementation

8.1.2.1 Left-factorisation

The idea of left-factorisation was introduced in Chapter 2 Section 2.2.1 as a grammar transformation technique used to make a grammar suitable for predictive parsing. In the implementation of META-LISP, left-factorisation is applied not to the grammar but affects the procedural interpretation of the translation rules. The compiler identifies those rules that share a common prefix, and arranges for code to be generated such that the common prefix is expanded only once, and only when that expansion is successful will the remainders of the rules be expanded as usual for alternatives. The bindings created in the course of the expansion of the common prefix are passed to the semantic actions associated with all the rules that had a common prefix.

8.1.2.2 Parameterisation

Much of the power and convenience of LISP derives from the fact that its fundamental data structure is lists. Doing everything in lists can be elegant, but is costly. Unfortunately there are no automatic means of providing cheaper alternative storage structure to replace lists invisibly. The Elisp of Emacs [KLL+] and the Window Object Oriented Lisp of the General Window Manager [Nah91] are notable attempts in this direction. META-LISP, in comparison, can be said to provide an even more expensive data structure than LISP: list structures definable by grammar rules. The manipulation of these structures can easily be more expensive than the manipulation of list structures. Fortunately, there is a way of providing all the extra expressive power of defining list structures through grammatical means without loss of efficiency. The key to this is that it is possible to distinguish between definitions that really require parsing to take place, and those that require only pattern matching. The meta-circular compiler for META-LISP applies such an analysis. This analysis is referred to a parameterisation.

The performance of the compiler is approximately 2000 lines per minute including compilation by the LISP compiler,(or about 4000 lines per minute excluding compilation by LISP), on an IPC workstation running Lucid Sun Common Lisp version 4.0.1.

8.2 The META-LISP Programming Environment

The Programming Environment for META-LISP provides most of the usual facilities that are expected of languages for symbolic computations. These include

- Incremental Entry of Programs
- Separate Compilation
- Debugging Facilities
- Browsing Facilities
- Integration with Emacs

```
<n>          : do <n> steps
a[bort]      : abort execution
b[reak]      : enter a new break level
c[reep]      : switch to creep mode
d[evelop]    : enter development tool
e[dit]       : edit current concept
f[ail]       : fail the current expansion
h[elp]       : print this help
l[eap]       : leap to next spy point
L[eap]       : leap and show the trace
n[odebug]    : switch off debugging and continue execution
r[eturn]     : return current concept
s[kip]       : skip spypoints until current call returns
S[kip]       : skip and show the trace
t[race]      : Show trace on spypoints without stopping
+            : add spypoint to the current concept
-            : remove spypoint from the current concept
;            : backtrack one step
; <int>      : backtrack  <int> steps
< n          : set print depth to <n>
```

Figure 8.4: Trace Commands

A *User Manual* for META-LISP is under preparation. [Laj93]

Chapter 9

Conclusion

This chapter summarises the main contributions of this dissertation. It also indicates directions for future work. The Chapter concludes with a brief discussion of the place of META-LISP in relation to other languages for symbolic computations.

9.1 Contributions

The starting point of this dissertation was the observation that the set of valid inputs to a program can be regarded as a form of computer language – an *input data language*. It was then proposed that programs can be viewed as *interpreters* or *compilers* of their input data language. The central thesis of this dissertation has been that the adoption of this *language oriented* view of programs leads to the establishment of a new programming style, according to which programs are designed and specified *as* translators of their input language.

As the means of exploring the potential of *language oriented programming* the design and implementation of a new programming language, called META-LISP, have been presented. META-LISP combines the syntax-directed model of computation with the functional model in one language. The language meets the following design objectives:

- It provides linguistic support for the design of programs *as* translators of their input language. That is to say, it supports programming in the *language oriented* style.
- META-LISP integrates well with LISP, which means
  - inter-operability, i.e. META-LISP programs can incorporate LISP functions and vice versa
  - the gain in expressive power over LISP has been achieved without loss in efficiency

- The definitional power of the language is not *excessive*. This has been achieved by
  - tying the language-definitional formalism to a particular – transparent – parsing algorithm.
  - fixing the order of evaluation in the semantic actions.

From the standpoint of *programming methodology* the main contribution of the present work is to offer a uniform design methodology. It is uniform, since at every level the programmer faces the following tasks:

- Define the set of valid inputs to the program explicitly as a language.
- the guiding principle for such definition is that the structure thereby imposed on the input should reflect the *conceptual structure* of the problem domain
- the semantic actions that are to be associated with each rule of the grammar that define the set of valid input to the program are to be formulated to reflect the *applicative structure* of solutions that the program is to offer to problems describable in terms of the input language of the program.

The case studies presented in this thesis served to demonstrate the success of the language orinted methodology. The META-LISP sytems contains a great number of programs written in META-LISP. Nearly everything in the system is written in META-LISP.

The best words that I can find to capture the essence of the methodology of language oriented programming in META-LISP were written twenty years ago by Dijkstra in his Turing award lecture speculating on the form of "future" programming languages. META-LISP can be said to invite us "to reflect in the structure of what we write down all the abstractions needed to cope conceptually with the complexity of what we are designing" [Dij72, 865]

9.2 Future Work

The future work discussed in this section encompasses improvements to, and enhancement of META-LISP, the language, and its associated programming system.

9.2.1 Improvements

The areas of planned improvements include the incorporation of 'copying' rules for synthesised attributes into the language, improvements to the module system, exception handling and support for meta-programming.

Looking at the code for the meta-circular interpreter makes a convincing case for the need of rules governing the copying of synthesised attributes. Attribute grammars have been criticised in the literature for the same prevalence of copying rules. It appears, that easy access to "long-distance"' [Wai90, 261] relationships between attributes could be provided in META-LISP by a suitable modification of its semantics. This needs further investigation.

The facility for importing the functionality of effective concepts from other modules is rudimentary. It constitutes a single, unstructured name space of modules. Work needs to be done to make this more sophisticated.

A not unrelated problem concerns the integration of META-LISP with LISP. The semantics of META-LISP as defined by the meta-circular interpreter, exclude the importation of LISP macros. The compiler allows this. It seems desirable that this discrepancy be resolved in favour of the compiler; i.e. to allow macros to be incorporated into META-LISP programs.

Exceptional situations and error handling, in general, have been rather neglected. These issues will need to be addressed in the future.

The price of META-LISP's gain in expressive power, when it is compared to LISP, has been the loss of one of LISP's main asset: the uniformity of representation of programs and data. META-LISP being a meta-language, *per se*, allows the routine construction of *meta-programs*, i.e. programs that treat another program as data. However, it is nothing like as straight-forward as in LISP, or PROLOG for that matter. [1] Work needs to be done to develop the mechanisms for disciplined access to the meta-programs in the META-LISP system (the interpreter, the compiler, reader, printer, partial evaluator, type checker, etc) and their components. There is also a need to relate the present work to the results of research on meta-programming in logic programming [HL89] and reflection in LISP [Smi84].

9.2.2 Enhancements

There is plenty of scope for future enhancements. Some of these concern the design of META-LISP itself, others concern the environmental support. There is a great deal of overlap between the two, too.

9.2.2.1 Type Checking

META-LISP is an untyped language. As in the case of LISP, this can be a great asset for the purpose of exploratory programming. In particular, it makes it possible to interleave testing and development to a much greater extent than in typed languages. In the case

[1] This may turn out to be more of an asset, than a liability.

of LISP, this makes "debugging the nil program" a serious candidate for a model of the software development process. Similarly, in META-LISP, even an incomplete elaboration of a program will work and may produce meaningful output, if the input on which it is tested belongs to the (partially defined) input language of the program. Even if the input is not acceptable (yet) the program can be run and examined. This can even help to debug the design of the input language itself. There are certain circumstances, however, where the typeless character of the language is a disadvantage. Experience in writing denotational style language definitions in META-LISP has clearly demonstrated this. The ideal solution appears to be to develop a *type inference* scheme for META-LISP – a way of finding out the type of an effective concept from its definition – to be incorporated into a *type checker* which could be enabled or disabled depending on the current requirements of program development.

9.2.2.2 Partial Evaluation

A related area of future enhancements, this time of the programming system, is the development of a partial evaluator for META-LISP. Partial evaluation is a program transformation technique which specialises programs with respect to given incomplete data. The development of partial evaluators for all kinds of languages (Scheme, PROLOG, the lambda calculus, etc) has recently become a very active research area. Partial evaluation is being used in program transformation, semantics directed compiler generation, generation of compiler-compilers, etc. For references see [GJ91].

The motivation for the development of a partial evaluator for META-LISP is twofold. First, it could be used to generate a compiler for META-LISP from its denotational style meta-circular definition. Secondly, it could be used to generate compilers for other languages from their denotational style interpreter written in META-LISP. The expectation is that this would open up another avenue for semantics directed compiler generation, including the implementation of "designer" languages, in accordance with the basic philosophy of language oriented programming.

9.2.2.3 Program Inversion

Most of the benefits of language oriented programming in META-LISP derive from the fact that the design of every program, and every non-elementary procedure in every program, is based on an explicit definition of their inputs as a language. Although the output of a program is also regarded as a language, according to the language oriented view of programs, META-LISP does not provide linguistic support for this. Instead, the set of valid outputs

are 'defined' only, implicitly. Under certain conditions, it is apparent that an appropriate definition of the outputs of a program as a language can be inferred from the definition of the program. Such a definition could, in principle, be used as the basis for generating inverse programs, as a generalisation of the idea of 'unparsers', c.f. [All78, 422].

9.2.2.4 Automatic Generation of Test Data

By the generative use of the explicit grammatical description of the set of valid inputs to a program, suites of test data can be produced automatically. This facility can be used not only in testing, but as a means of evaluating the 'competence' of the program in its intended field of application.

9.3 Discussion

The intended application area of META-LISP is symbolic computation. This section offers brief comparisons of META-LISP with four representatives of established programming languages for symbolic computation. The four languages that will be discussed are SNOBOL4, ML, Prolog and LISP.

SNOBOL4 is programming language for string manipulation. The input to a SNOBOL4 program is a string. The output is also a string. A SNOBOL4 program applies string matching and manipulation of its input to produce its output. In its perspective on programming it can be said to show some resemblance to the language oriented view. A major source of difference is that the basic data-structure of SNOBOL4 is the string, whereas it is list structures in META-LISP. Whereas the structure of the input, in META-LISP is defined using a grammar, in SNOBOL4 the input is defined implicitly through a range of pattern matching operations. These operations are quite powerful, and in certain cases can even be said to resemble the style of META-LISP definitions. Grammatical structures, however, can only be described in terms of low level string matching operations which tend to be opaque. Compare a SNOBOL4 program for translating arithmetic expressions from infix to prefix notation [GPP68, 104-108].

ML is a strongly typed functional language. It uses pattern matching as its parameter passing mechanism. In contrast, META-LISP uses syntax-directed translation as its parameter passing mechanism. As syntax-directed translation properly subsumes pattern matching, META-LISP can offer capabilities not possessed by functional languages: these include support for *data abstraction*, *representation independent* or *level-wise* programming (see page 37), as well as support for parser construction. META-LISP's syntax-directed

parameter passing mechanism encourages the use of *abstract analysers* as an efficient form of data-abstraction. (See chapter 3). The usefulness of META-LISP support for automatic parser-construction even on its own, can be judged from the point made by Wikström that a parser generator is a tool that should accompany an ML system for production use [Wik87, 294]. An example of this is the *grammar* feature of CAML [WAL+90].

ML also provides facilities for abstract data-types. META-LISP does not provide explicit support for this. It is left as a matter of style of writing programs. Incorporating the ideas of abstract data-types into META-LISP will be considered in the future. The planned development of a type-inference scheme for META-LISP will further reduce the differences between ML and META-LISP.

What is common to both ML as a functional language and META-LISP as a language oriented programming language, is that they both make commitments about which quantities are inputs and which are outputs. This can be contrasted to logic programming languages, such as Prolog, that do not make such commitments [Red86, 3]. The multidirectionality of Prolog is a consequence of the fact that Prolog uses unification as its calling mechanism [DFP86, 45]. It gives capabilities to Prolog not possessed by ML, or META-LISP for that matter. Through the use of operator declarations Prolog also has explicit language definitional capabilities. In terms of expressive power, Prolog is clearly superior. However, this extra expressive power of Prolog, can be said to be a mixed blessing. For example, the multidirectionality of Prolog means in practice that "programmers have to devote as much time to think about the different tasks a relation might do, as they would in writing a set of functions for these tasks in any other language" [McD80].

The semantic backtracking mechanism of META-LISP can be used to provide multiple solutions through backtracking. See page 103. In META-LISP backtracking has to be explicitly requested, and even then it is limited. It seems preferable to explicitly request backtracking rather than prune automatic backtracking with judicious use of cut, as in Prolog.

The greatest price that is paid for Prolog's extra expressive power is that it is difficult to envisage the possible computations that a Prolog clause may give rise to under varying circumstances. META-LISP's emphasis on transparent operational semantics was motivated by the desire to avoid the problem associated with excessive definitional power.

The relationship between LISP and META-LISP can be likened to the relationship that exists between a high-level language and a machine language in which it is implemented. For a long time, LISP has been regarded as "the 'machine language' for Artificial Intelligence" [All78, 243]. Even earlier, LISP have been referred to as an implicit meta-language.[Ing66,

115-6] META-LISP can be considered as an extension of LISP, which makes the meta-language character of LISP explicit. In doing so, it inherits a lore of methodological insights which have always been part of the LISP tradition. The present work, ultimately, is dedicated to the LISP programmer who may now pen his thoughts within fewer parentheses and yet let his ever present linguistic insights master the complexity of his task with greater ease.

Bibliography

[Ada90] Stephen Adams. Towards language-oriented programming. Technical Report CSTR 90-5, Department of Electronics & Computer Science, University of Southampton, February 1990.

[All78] John Allen. *Anatomy of Lisp.* McGraw-Hill, 1978.

[ASS85] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* The MIT Press, Cambridge, Massachusetts, 1985.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 1986.

[AU72] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling.* Prentice-Hall, Inc., Englewood Cliffs, N.J., 1972.

[Ben86] Jon Bentley. Programming pearls – little languages. *Communications of the ACM*, 29(8):711–721, August 1986.

[Ben90] J.P. Bennet. *Introduction to Compiling Techniques – A first Course using ANSI C, LEX and YACC.* McGraw-Hill Book Company, 1990.

[BK86] J.L. Bentley and B.W. Kernighan. Grap – a language for typesetting graphs. *Communications of the ACM*, 29(8):782–792, 1986.

[Bra61] H. Bratman. An alternate from of the 'Uncol diagram'. *Comm. ACM*, 4(3):142, 1961.

[DFP86] John Darlington, A. J. Field, and H. Pull. The unification of functional and logic languages. In Doug DeGroot and Gary Lindstrom, editors, *Logic Programming, Functions, Relations, and Equations*, pages 37–70. Prentice-Hall, 1986.

[Dij72] Edsger W. Dijkstra. The Humble Programmer. *CACM*, 15(10):859–866, October 1972.

[DJ83] P. M. Dew and K. R. James. *Introduction to Numerical Computation in Pascal.* MacMillan, 1983.

[DM88] Pierre Deransart and Jan Maluszynski. A grammatical view of logic programming. In P. Deransart and J. Malunszynski, editors, *Programming Languages Implementation and Logic Programming*, Lecture Notes in Computer Science 138, pages 219–252. Springer-Verlag, 1988.

[Fey84] Stefan Feyock. Syntax programming. In *AAAI-84: Proceedings – Sixth National Conference on Artificial Intelligence*, pages 110–115, Austin, Texas, August 1984.

[FSO91] R. Furuta, P. D. Stotts, and J. Ogata. Ytracc: A parse browser for yacc grammars. *Software – Practice and Experience*, 21(2):119–132, February 1991.

[GJ91] Carsten K. Gomard and Neil D. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(1):21–69, January 1991.

[GLSS75] Jr. Guy Lewis Steele and Gerald Jay Sussman. Scheme: An interpreter for the extended lambda calculus. Technical Report 349, MIT Artificial Intelligence Laboratory, 1975.

[GM86] Joseph A. Goguen and José Meseguer. Eqlog: Equality, types and generic modules for logic programming. In Doug DeGroot and Gary Lindstrom, editors, *Logic Programming: Functions, Relations, and Equations*, pages 295–363. Prentice-Hall, 1986.

[Gou88] K. John Gough. *Syntax Analysis and Software Tools.* Addison-Wiesley Publishing Company, 1988.

[GPP68] R. E. Griswold, J. F. Poage, and I. P. Polonsky. *The SNOBOL4 Programming Language.* Prentice-Hall Inc., 1968.

[Hen80] Peter Henderson. *Functional Programming – Application and Implementation.* Prentice-hall International Series in Computer Science. Prentice-Hall International, 1980.

[HL89] P.M. Hill and J.W. Lloyd. Analysis of meta-programs. In Harver Abramson and M.H. Rogers, editors, *Meta-Programming in Logic Programming*, Logic Programming. The MIT Press, 1989.

[Hor89] R. Nigel Horspool. ILALR: An Incremental Generator of LALR(1) Parsers. In D. Hammer, editor, *Compiler Compilers and High Speed Compilation*, volume 371 of *LNCS*, 2nd CCHSC Workshop, Berlin, GDR, October 10-14, 1988, 1989.

[Ing66] Peter Zilahy Ingerman. *A Syntax-Oriented Translator.* Academic Press, 1966.

[Joh79] S. Johnson. *YACC: Yet Another Compiler-Compiler.* Bell Laboratories, 1979.

[KLL+] Robert Krawitz, Bil Lewis, Dan LaLiberte, Richard M. Stallman, and Chris Welty. *GNU Emacs Lisp Reference Manual.* Free Software Foundation.

[Knu68] Donald E. Knuth. Semantics of context-free languages. *Math. Syst. Theory 2*, 1968.

[Kos84] Kai Koskimies. A specification language for one-pass semantic analysis. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, volume 19 of *SIGPLAN Notices*, pages 179–189, June 1984.

[Kow79] Robert Kowalski. *Logic for Problem Solving.* North-Holland, 1979.

[Laj90] G. Lajos. Language-directed programming in meta-lisp. In *EUROPAL'90: The First European Conference on the Practical Applications of Lisp*, Churchill College Cambridge, UK,, 1990.

[Laj93] G. Lajos. META-LISP *Users's Manual.* School of Computer Studies, The University of Leeds, Leeds LS2 9JT, 1993.

[MAE+65] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. *LISP 1.5 Programmer's Manual.* The Computation Center and Research Laboratory of Electronics, Massachusetts Institute of Technology, The M.I.T. Press, 1965.

[Mar83] Jed Marti. The Little META Translator Writing System. *Software – Practice and Experience*, 13:941–959, 1983.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, April 1960.

[McC80] John McCarthy. Lisp - notes on its past and future. In *The 1980 LISP Conference*, Stanford, 1980.

[McD80] D. McDermott. The PROLOG phenomena. *SIGART Newsletter 72*, pages 16–20, 1980.

[Mos79] P. D. Mosses. SIS – semantics implementation system, reference manual and user guide. Technical Report DAIMI MD-30, Aarhus University, 1979.

[Nah91] Colas Nahaboo. *GWM Manual: The X11 Generic Window Manager.* KOALA Project – BULL Research, 1989-1991.

[NF89] Tim Nicholson and Norman Foo. A denotational semantics for prolog. *ACM Transaction on Programming Languages and Systems*, 11(4):651–665, October 1989.

[Pag81] Frank G. Pagan. *Formal Specification of Programming Languages, A Panoramic Primer.* Prentice-Hall Inc., 1981.

[Pau84] Lawrence Paulson. Compiler generation from denotational semantics. In B. Lorho, editor, *Methods and Tools for Compiler Construction*, pages 219–250. Cambridge University Press, 1984.

[PC89] James J. Purtilo and John R. Callahan. Parse-tree annotations. *CACM*, 32(12):1467–1477, 1989.

[Pra75] Terrence W. Pratt. *Programming Languages: Design and Implementation.* Prentice-Hall, 1st edition, 1975.

[Red86] Uday S. Reddy. On the relationship between logic and functional languages. In Doug DeGroot and Gary Lindstrom, editors, *Logic Programming, Functions, Relations, and Equations*, pages 3–36. Prentice-Hall, 1986.

[Rey72] J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the 25th ACM National Conference*, pages 717–740, Boston, 1972.

[San82] David Sandberg. LITHE: A language combining a flexible syntax and classes. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 142–145, January 1982.

[Sch64] D. V. Schorre. Meta-II: a syntax-oriented compiler writing language. In *Proc. 19th ACM National Conf.*, pages D1.3–1–D1.3–11, 1964.

[Sch86] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, Inc., 1986.

[Smi84] Brian Cantwell Smith. Reflection and semantics in Lisp. In *Eleventh Annual ACM Symposium on the Principles of Programming Languages*, volume 11, pages 23–35, Salt Lake City, Utah, January 1984.

[Spi88] J. M. Spivey. *Understanding Z – A Specification language and its formal semantics.* Number 3 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1988.

[Sto77] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics*, volume 1. of *The MIT Press Series in Computer Science.* The MIT Press, 1977.

[Tof90] Mads Tofte. *Compiler Generators: What they Can Do, What They might Do, and What They Will Probably Never Do*, volume 19 of *ETACS Monographs on Theoretical Computer Science.* Springer-Verlag, 1990.

[Wad87] Philip Wadler. A way for pattern matching to cohabit with data abstraction. In *Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 307–313, 1987.

[Wai90] W. M. Waite. Use of Attribute Grammars in Compiler Construction. In P. Deransart and M. Jourdan, editors, *Attribute Grammars and their Applications*, volume 461 of *LNCS*, International Conference WAGA Paris, France, September 1990. Springer-Verlag.

[WAL+90] Pierre Weis, Mária-Virginia Aponte, Alain Laville, Michel Mauny, and Ascánder Suárez. The CAML Reference Manual. Technical Report Version 2.6, Projet Formal, INRIA-ENS, 1990.

[Wan84] M. Wand. A Semantic Prototyping System. In *Proceedings of the ACM SIGPLAN '84 Symp. on Compiler Construction*, volume 19 of *SIGPLAN Notices*, pages 213–221, Montreal, June 1984.

[Wei67] Clark Weissman. *LISP 1.5 PRIMER.* Dickenson Publishing Company, Inc., Belmont, California, 1967.

[Wex81] R. L. Wexelblat. *History of Programming Languages.* Academic Press, 1981.

[Wik87] Ake Wikström. *Functional Programming Using Standard ML.* Prentice Hall, London, 1987.

[Win83] Terry Winograd. *Language as a Cognitive Process: Syntax*, volume 1. Addison-Wesley, 1983.

[YN88] Yoshiyuki Yamashita and Ikuo Nakata. Coupled context-free grammar as a programming paradigm. In P. Deransart and J. Malunszynski, editors, *Programming Languages Implementation and Logic Programming*, Lecture Notes in Computer Science 138, pages 132–145. Springer-Verlag, 1988.

Bibliography

[Ada90] Stephen Adams. Towards language-oriented programming. Technical Report CSTR 90-5, Department of Electronics & Computer Science, University of Southampton, February 1990.

[All78] John Allen. *Anatomy of Lisp.* McGraw-Hill, 1978.

[ASS85] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* The MIT Press, Cambridge, Massachusetts, 1985.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 1986.

[AU72] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling.* Prentice-Hall, Inc., Englewood Cliffs, N.J., 1972.

[Ben86] Jon Bentley. Programming pearls – little languages. *Communications of the ACM*, 29(8):711–721, August 1986.

[Ben90] J.P. Bennet. *Introduction to Compiling Techniques – A first Course using ANSI C, LEX and YACC.* McGraw-Hill Book Company, 1990.

[BK86] J.L. Bentley and B.W. Kernighan. Grap – a language for typesetting graphs. *Communications of the ACM*, 29(8):782–792, 1986.

[Bra61] H. Bratman. An alternate from of the 'Uncol diagram'. *Comm. ACM*, 4(3):142, 1961.

[DFP86] John Darlington, A. J. Field, and H. Pull. The unification of functional and logic languages. In Doug DeGroot and Gary Lindstrom, editors, *Logic Programming, Functions, Relations, and Equations*, pages 37–70. Prentice-Hall, 1986.

[Dij72] Edsger W. Dijkstra. The Humble Programmer. *CACM*, 15(10):859–866, October 1972.

[DJ83] P. M. Dew and K. R. James. *Introduction to Numerical Computation in Pascal.* MacMillan, 1983.

[DM88] Pierre Deransart and Jan Maluszynski. A grammatical view of logic programming. In P. Deransart and J. Malunszynski, editors, *Programming Languages Implementation and Logic Programming*, Lecture Notes in Computer Science 138, pages 219–252. Springer-Verlag, 1988.

[Fey84] Stefan Feyock. Syntax programming. In *AAAI-84: Proceedings – Sixth National Conference on Artificial Intelligence*, pages 110–115, Austin, Texas, August 1984.

[FSO91] R. Furuta, P. D. Stotts, and J. Ogata. Ytracc: A parse browser for yacc grammars. *Software – Practice and Experience*, 21(2):119–132, February 1991.

[GJ91] Carsten K. Gomard and Neil D. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(1):21–69, January 1991.

[GLSS75] Jr. Guy Lewis Steele and Gerald Jay Sussman. Scheme: An interpreter for the extended lambda calculus. Technical Report 349, MIT Artificial Intelligence Laboratory, 1975.

[GM86] Joseph A. Goguen and José Meseguer. Eqlog: Equality, types and generic modules for logic programming. In Doug DeGroot and Gary Lindstrom, editors, *Logic Programming: Functions, Relations, and Equations*, pages 295–363. Prentice-Hall, 1986.

[Gou88] K. John Gough. *Syntax Analysis and Software Tools.* Addison-Wiesley Publishing Company, 1988.

[GPP68] R. E. Griswold, J. F. Poage, and I. P. Polonsky. *The SNOBOL4 Programming Language.* Prentice-Hall Inc., 1968.

[Hen80] Peter Henderson. *Functional Programming – Application and Implementation.* Prentice-hall International Series in Computer Science. Prentice-Hall International, 1980.

[HL89] P.M. Hill and J.W. Lloyd. Analysis of meta-programs. In Harver Abramson and M.H. Rogers, editors, *Meta-Programming in Logic Programming*, Logic Programming. The MIT Press, 1989.

[Hor89] R. Nigel Horspool. ILALR: An Incremental Generator of LALR(1) Parsers. In D. Hammer, editor, *Compiler Compilers and High Speed Compilation*, volume 371 of *LNCS*, 2nd CCHSC Workshop, Berlin, GDR, October 10-14, 1988, 1989.

[Ing66] Peter Zilahy Ingerman. *A Syntax-Oriented Translator.* Academic Press, 1966.

[Joh79] S. Johnson. *YACC: Yet Another Compiler-Compiler.* Bell Laboratories, 1979.

[KLL+] Robert Krawitz, Bil Lewis, Dan LaLiberte, Richard M. Stallman, and Chris Welty. *GNU Emacs Lisp Reference Manual.* Free Software Foundation.

[Knu68] Donald E. Knuth. Semantics of context-free languages. *Math. Syst. Theory 2*, 1968.

[Kos84] Kai Koskimies. A specification language for one-pass semantic analysis. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, volume 19 of *SIGPLAN Notices*, pages 179–189, June 1984.

[Kow79] Robert Kowalski. *Logic for Problem Solving.* North-Holland, 1979.

[Laj90] G. Lajos. Language-directed programming in meta-lisp. In *EUROPAL'90: The First European Conference on the Practical Applications of Lisp*, Churchill College Cambridge, UK,, 1990.

[Laj93] G. Lajos. META-LISP *Users's Manual.* School of Computer Studies, The University of Leeds, Leeds LS2 9JT, 1993.

[MAE+65] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. *LISP 1.5 Programmer's Manual.* The Computation Center and Research Laboratory of Electronics, Massachusetts Institute of Technology, The M.I.T. Press, 1965.

[Mar83] Jed Marti. The Little META Translator Writing System. *Software – Practice and Experience*, 13:941–959, 1983.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, April 1960.

[McC80] John McCarthy. Lisp - notes on its past and future. In *The 1980 LISP Conference*, Stanford, 1980.

[McD80] D. McDermott. The PROLOG phenomena. *SIGART Newsletter 72*, pages 16–20, 1980.

[Mos79] P. D. Mosses. SIS – semantics implementation system, reference manual and user guide. Technical Report DAIMI MD-30, Aarhus University, 1979.

[Nah91] Colas Nahaboo. *GWM Manual: The X11 Generic Window Manager.* KOALA Project – BULL Research, 1989-1991.

[NF89] Tim Nicholson and Norman Foo. A denotational semantics for prolog. *ACM Transaction on Programming Languages and Systems*, 11(4):651–665, October 1989.

[Pag81] Frank G. Pagan. *Formal Specification of Programming Languages, A Panoramic Primer.* Prentice-Hall Inc., 1981.

[Pau84] Lawrence Paulson. Compiler generation from denotational semantics. In B. Lorho, editor, *Methods and Tools for Compiler Construction*, pages 219–250. Cambridge University Press, 1984.

[PC89] James J. Purtilo and John R. Callahan. Parse-tree annotations. *CACM*, 32(12):1467–1477, 1989.

[Pra75] Terrence W. Pratt. *Programming Languages: Design and Implementation.* Prentice-Hall, 1st edition, 1975.

[Red86] Uday S. Reddy. On the relationship between logic and functional languages. In Doug DeGroot and Gary Lindstrom, editors, *Logic Programming, Functions, Relations, and Equations*, pages 3–36. Prentice-Hall, 1986.

[Rey72] J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the 25th ACM National Conference*, pages 717–740, Boston, 1972.

[San82] David Sandberg. LITHE: A language combining a flexible syntax and classes. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 142–145, January 1982.

[Sch64] D. V. Schorre. Meta-II: a syntax-oriented compiler writing language. In *Proc. 19th ACM National Conf.*, pages D1.3–1–D1.3–11, 1964.

[Sch86] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, Inc., 1986.

[Smi84] Brian Cantwell Smith. Reflection and semantics in Lisp. In *Eleventh Annual ACM Symposium on the Principles of Programming Languages*, volume 11, pages 23–35, Salt Lake City, Utah, January 1984.

[Spi88] J. M. Spivey. *Understanding Z – A Specification language and its formal semantics.* Number 3 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1988.

[Sto77] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics*, volume 1. of *The MIT Press Series in Computer Science.* The MIT Press, 1977.

[Tof90] Mads Tofte. *Compiler Generators: What they Can Do, What They might Do, and What They Will Probably Never Do*, volume 19 of *ETACS Monographs on Theoretical Computer Science.* Springer-Verlag, 1990.

[Wad87] Philip Wadler. A way for pattern matching to cohabit with data abstraction. In *Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 307–313, 1987.

[Wai90] W. M. Waite. Use of Attribute Grammars in Compiler Construction. In P. Deransart and M. Jourdan, editors, *Attribute Grammars and their Applications*, volume 461 of *LNCS*, International Conference WAGA Paris, France, September 1990. Springer-Verlag.

[WAL+90] Pierre Weis, Mária-Virginia Aponte, Alain Laville, Michel Mauny, and Ascánder Suárez. The CAML Reference Manual. Technical Report Version 2.6, Projet Formal, INRIA-ENS, 1990.

[Wan84] M. Wand. A Semantic Prototyping System. In *Proceedings of the ACM SIGPLAN '84 Symp. on Compiler Construction*, volume 19 of *SIGPLAN Notices*, pages 213–221, Montreal, June 1984.

[Wei67] Clark Weissman. *LISP 1.5 PRIMER.* Dickenson Publishing Company, Inc., Belmont, California, 1967.

[Wex81] R. L. Wexelblat. *History of Programming Languages.* Academic Press, 1981.

[Wik87] Ake Wikström. *Functional Programming Using Standard ML.* Prentice Hall, London, 1987.

[Win83] Terry Winograd. *Language as a Cognitive Process: Syntax*, volume 1. Addison-Wesley, 1983.

[YN88] Yoshiyuki Yamashita and Ikuo Nakata. Coupled context-free grammar as a programming paradigm. In P. Deransart and J. Malunszynski, editors, *Programming Languages Implementation and Logic Programming*, Lecture Notes in Computer Science 138, pages 132–145. Springer-Verlag, 1988.