Interpreter prototypes from formal language definitions

Gregory John Michaelson

Thesis submitted for the degree of
Doctor of Philosophy

Heriot-Watt University

Department of Computing and Electrical Engineering

April 1993

Declaration

I hereby declare that the work presented in this thesis was carried out by myself at Heriot-Watt University, except where due acknowledgement is made, and has not been submitted for any other degree.

Gregory John Michaelson

Acknowledgements

I would like to thank:

- my supervisers Tony Davie, David Watt and Howard Williams for their support, encouragement and patience.

- Ken Barclay, Bill Findlay, John Patterson, Stewart Anderson and Paul Chisholm for much fruitful disputation.

- David Turner for reconstructing the history of SASL for me and for general information about the genesis of functional languages.

- Howard Williams for checking thoroughly the first draft of this thesis and for making many useful comments and suggestions.

- Peter King for mitigating the worst excesses of TROFF.

- the many people who, over the years, have supplied or helped me to track down references.

Abstract

Denotational semantics is now used widely for the formal definition of programming languages but there is a lack of appropriate tools to support language development. General purpose language implementation systems are oriented to syntax with poor support for semantics. Specialised denotational semantics based systems correspond closely to the formalism but are rendered inflexible for language experimentation by their monolithic multiple stages

Exploratory language development with formal definitions is better served by a unitary notation, encompassing syntax and semantics, which is close to but simpler than denotational semantics. Interactive implementation of the notation then facilitates language investigation through the direct execution of a formal definition as an interpreter for the defined language.

This thesis presents Navel, a run-time typed, applicative order, pure functional programming language with integrated context free grammar rules. Navel has been used to develop prototype implementations from substantial formal language definitions, including for Navel itself. The Navel implementation achieves a performance which enables interactive language experimentation and compares well with that of contemporaneous declarative language implementations.

Denotational semantics does not address concrete syntax issues and denotational semantics based systems either ignore or have ad-hoc provision for context sensitivity. In Navel, rules are full values. Abstraction over rules enables the concise expression of context sensitivity in a style similar to dynamic syntax.

Chapter 1

Introduction

1.1. Background

Programming languages are central to Computing as a means of expressing and animating computations. Formal definitions of programming languages from sound mathematical bases are essential for Computing rigour. In principle they provide standards for ensuring consistency in language implementations, for enabling the systematic construction of correct programs, for proving the correctness of programs relative to formal specifications and for demonstrating that program transformations preserve meaning.

At present, the most widely used formal approach to language definition is Denotational Semantics(DS), based on Strachey's and Scott's pioneering work [142, 134]. A DS is a system of equations which associates syntactic constructs with meaning functions over abstract domains. Formal demonstrations of properties of languages and programs are then carried out through mathematical manipulation of these defining equations, constrained by domain properties. DS definitions have been developed retrospectively for a variety of extant languages [7, 14, 61, 66, 107], and the use of DS in language design is growing [43, 122]. However, the wider use of formal approaches in general and of DS in particular is hindered both by the culture of contemporary Computing and by the absence of appropriate support tools.

1.2. Formality in Computing

Until recently, Computing has been viewed as a practical activity; more of a craft than a science. Most practitioners have relatively weak mathematical backgrounds and view formality with trepidation. Formal techniques are indeed often inaccessible, being based on apparently inscrutable notations which seem to bear little relation to "real" programming languages. With a few exceptions, formality is still seen as an unwarranted overhead which detracts from the purposeful construction of useful code, best left to academe.

Changes in curricula are increasing slowly the number of people with training in formal methods. What were once presented as advanced topics are now being moved into the earlier years of courses. In particular, discrete mathematics and formal specification techniques are now taught as complementary to practical program development, and further integration of formality as a basis for programming seems likely [50]. In the longer term, this may increase the industrial and commercial use of formal techniques as a new generation moves through the hierarchies of businesses.

However, the formal definition of languages is still perceived as a specialist area and its presentation is often delayed to form an option in the final year, if it is covered at all. Ironically, compiling techniques, once also seen as specialist, are taught increasingly in the middle years of courses. Alas, formality in compiling does not usually extend beyond the comparatively well understood area of grammars and translators, with semantic concerns tackled by ad-hoc approaches. Thus, the number of people with DS skills remains small and semantic aspects of language implementations still rest on craft techniques for most commercial production.

1.3. Language implementation from DS

Since the original Brooker-Morris compiler-compiler [21], a number of tools have been developed to aid language implementation, of which LEX [88] and YACC [72] for UNIX are probably the most widely used. However, such tools tend only to offer support for lexical and concrete syntactic aspects of language definitions, providing notations for dictionaries and grammars from which lexical and syntax analysers are generated automatically. Semantic processors must be constructed explicitly by hand. None the less, such tools aid greatly the construction of robust and efficient language implementations.

However, contemporary general purpose language implementation tools are ill suited for DS oriented language development. DS definitions are based on associations between abstract syntax constructs and semantic functions, with no reference to concrete syntax. Concrete syntax defines surface representations whereas abstract syntax reflects meaningful structure. Hence in DS, concrete syntax is deemed to contain details which are irrelevant for semantics and it is assumed implicitly that abstract syntax reflects successful concrete syntax recognition. Thus, DS fails to address those aspects of language definitions for which language implementation tools are intended and such tools offer little support for semantics, the primary emphasis of DS.

The original proponents of DS were hostile philosophically to viewing DS as having operational readings and opposed the idea of treating DS definitions as executable specifications of languages [142, 141, 134, 132, 139, 57, 140]. However, others have taken a more pragmatic view arguing that there are many similarities between operational and denotational definitions [20, 6, 95], that constructing a DS is akin to programming [130, 112, 28], and that executing a DS may implement a language [131, 76, 156, 153].

Subsequently, a number of language implementation systems based on DS have been developed [105, 113, 153, 10, 144, 87]. Such systems adhere closely to the structure of DS which, while consistent, results in inflexibility. DS definitions are multi staged, consisting of descriptions of syntactic domains, abstract syntax, semantic domains and the functionality and details of semantic equations. Each stage also has distinct notations. Developing a DS for a language involves moving repeatedly between stages as the language design evolves, because a change in one stage often implies changes in other stages. Furthermore, practical systems based on DS must support lexical and concrete syntax details and provide linkage between concrete and abstract syntax, adding more notations and stages to language definitions. Thus, systems based directly on DS tend to be large and unwieldy. Such systems, like general purpose tools, are often batch oriented and accept an entire DS at once. While they may be appropriate for assessing a final DS, and may form eventually the basis of true compiler-compilers, they are inflexible for interactive or incremental DS based language experimentation and development.

The DS meta-language is based on syntactic extensions to the λ calculus and is similar stylistically to functional programming languages. Developing a DS is akin to writing an interpreter in a functional language [131, 76, 112, 28, 156, 153] and functional languages have been used to implement prototype interpreters from DS [78, 143, 28, 156]. However, functional languages lack constructs for lexical and syntactic processing: functional language based DS implementation requires the explicit construction of lexical analysers and parsers, or the use of a proprietary parser generator as a front end.

1.4. Research summary

The research discussed here is based on the premises that DS based language experimentation and development are best served by a simple, unitary notation encompassing syntax and semantics to minimise definitional stages. Such a notation should be based on a functional language with a close correspondence to the DS meta-language. Implementation of the notation would then enable the animation of definitions as prototype interpreters.

This thesis presents Navel, an applicative order functional language with integral grammar rules. Navel is a simplification of DS. In particular, definitions are based on the direct association of concrete syntax and semantic functions, without intervening abstract syntax. However, Navel is still extremely close to the DS semantic meta-language and converting a traditional DS to Navel is a straightforward activity. Furthermore, grammar rules are full values and rule abstraction enables the specification of context sensitive aspects of syntax without additional notation.

Navel has been implemented interpretively within an interactive environment and runs on a variety of hardware platforms with speed comparable to other languages of the same vintage. Navel definitions have been developed from DS for substantial demonstration languages and run as prototype interpreters, in particular for Navel itself. Navel is used currently with undergraduate students to teach DS and functional programming.

1.5. Thesis structure

Chapter 2 surveys the background to DS and its use for language implementation. DS is introduced and the operational and axiomatic approaches to formal semantics are discussed briefly. The objections of the DS founders to operational readings of DS are then considered and alternative views of DS as operational are presented. Approaches to language implementation from DS are surveyed briefly and five language implementation systems based on DS are examined. The premises for Navel’s development are elaborated.

Chapter 3 introduces the Navel core language, that is Navel without grammar rules. The relationship between the DS meta-language and functional languages is discussed briefly and a chronological survey of functional languages is presented. The Navel core language is presented informally through examples and formally through a full DS.

Chapter 4 discusses extensions to core Navel for syntactic processing. A chronological survey of programming language extensions and constructs for handling syntax is presented. Navel constructs for grammar rules, parse trees, and associating syntactic structures and semantic functions are presented informally through examples and formally through extensions to the core Navel DS.

Chapter 5 considers the Navel implementation. After a brief survey of functional language implementation techniques, overviews of the Navel implementation and its performance are given. The Navel interactive environment and imperative extensions for I/O and evaluation termination are then presented.

Chapter 6 demonstrates the implementation of a substantial ALGOL-like language fragment in Navel from its DS. The language provides integer and integer array variable declarations, arithmetic and boolean expressions, assignment, conditional and iterative statements, blocks, local definitions, and recursive procedures with call by value and call by reference parameter passing. The DS and equivalent Navel constructs are presented. The performance of the resulting interpreter is considered and used to compare Navel with two DS based systems.

Chapter 7 investigates the implementation of Navel in Navel, highlights apparent limitations to Navel for the implementation of substantial languages and considers how they may be overcome. The performance of Navel in Navel is considered. The implementation of continuation based DS in Navel is presented briefly.

Chapter 8 examines rule abstraction and context sensitive processing in Navel. Context free generalisation through rule abstraction is presented. The 2-level grammar, attribute grammar and dynamic syntax approaches to context sensitivity are surveyed briefly. The implementation of dynamic syntax in Navel is then illustrated with simple examples from imperative language syntax.

Chapter 9 summarises the dissertation. The research is reviewed and suggestions for the development of a future DS language implementation system are presented.

1.6. Previous publications

Some of the material in this thesis has been published elsewhere.

Overviews of Navel, of the interpretive implementation of formal semantics and of the use of higher order context free grammars for context sensitivity are presented in:

> G.Michaelson, ‘Interpreters from functions & grammars’, Computer Languages, Vol. 11, No. 2, pp. 85-104, 1986

The use of Navel for general prototyping is discussed in:

> G.Michaelson, ‘Interpreter prototypes from language definition style specifications’, Information & Software Technology, Vol. 30, No. 1, pp. 23-31, 1988

The semantics of a language similar to core Navel are presented through the stepwise addition of new syntactic constructs to pure type-free λ calculus in:

> G.Michaelson, ‘An introduction to functional programming through lambda calculus’, Addison-Wesley, ISBN 0-201-17812-5, 320 pages, 1989

The generation of text from Navel grammars is discussed in:

> G.Michaelson, ‘Text generation from grammars’, Information and Software Technology, Vol. 32, No. 8, pp 566-568, October 1990

Issues in the relationship between grammars and data structures are considered in:

> G.Michaelson, ‘Grammars and implementation independent structure representation’, in proceedings of "3rd International Workshop on Persistent Object Systems, Newcastle, Australia", J. Rosenberg & D. Koch(eds), Springer-Verlag, pp. 19-28, August 1990

Chapter 2

Denotational semantics and its implementation

2.1. Introduction

Denotational semantics(DS) is based on Strachey's [142, 134] characterisation of the meaning of syntactic constructs as abstract functions over domains. Subsequently DS was given a firm theoretical basis by Scott†. [133]. The meta-language for DS is based on syntactic extensions to a notation which is strongly related to Church's λ calculus [31]. As the λ calculus is referentially transparent, this makes DS well suited for proof of properties of language definitions and for the expression of time ordered control and evaluation sequences in languages in a time order independent manner. DS was the first fully formalised approach to language definition and has been extremely influential in Computer Science.

DS has been used widely to provide formal definitions of extant programming languages for example ALGOL 60 [66], Icon expressions [61], Pascal [7], a sizable fragment of PL/1 [14] and Prolog [107]. It is also used for the design of new languages, for example the proposed European LISP standard EuLISP [43], and Haskell [122], the new international functional language.

DS has been used in the development of compilers. For example, Allison [5], Bjorner [17], Chao and Bryant [28] Ganzinger [53], C. Jones [74], N. Jones and Schmidt [76], Jouvelot [78], Klaeren and Petzsch [81], Rakovsky and Collier [120], Royer [128], Sethi [135], Slonneger [136], Stepney et al [138], Stoy [139] and Takeichi [143] discuss techniques for designing and developing conventional compiler or interpreter based implementations from DS definitions. DS has also been used as the basis of compiler-compiler systems, for example SIS [105], Paulson's [113], Wand's [153], PSG [10], MESS [87] and CERES [144]. Some of these are considered in greater detail below.

DS also forms the basis of the Vienna Development Method [75] for system specification which is currently undergoing international standardisation.

2.2. The structure of DS definitions

In general, a DS definition consists of syntactic categories, syntax, semantic domains and semantic functions.

The syntactic categories identify the sets of syntactic constructs to which meaning is to be given. Each category is named and has associated with it the name of a syntactic rule for the corresponding constructs. Separate categories are identified for semantically significant constructs.

The syntax consists of rules for the structure of syntactic constructs. It is usual to distinguish between the concrete syntax of a language, which is concerned with its representation, and the abstract syntax, which is concerned with its meaningful structure. Concrete syntax contains details which are irrelevant for meaning, for example reserved words and operators which ease layout, and precedence. Indeed, several concrete syntaxes may have the same abstract syntax: that is, there may be different concrete representations with the same meaning. DS are usually specified for abstract syntax constructs, defined by context free grammar rules. The relationship between concrete and abstract syntax is considered in more detail below.

The semantic domains are those in which meanings are defined. They are named and their functionalities in terms of other domains are specified.

The semantic functions specify mappings from syntactic categories to semantic domains. They are defined using notations related to the λ notation. In principle, pure uncurried λ notation could be used. In practice, various syntactic extensions with pure λ notation equivalents are introduced to ease definitions, for example for boolean logic, integer

---

† Scott's theoretical work will not be discussed further here. Accessible accounts may be found in Stoy's [139] and Schmidt's [129] books on DS.

arithmetic, conditional expressions, curried functions, recursion, and fixed and variable size data structures. The status of the meta-language is considered in substantially more detail below.

In McCarthy's approach to abstract syntax [94], used in early operational semantics, a set of predicate, selector and constructor functions are defined for each abstract syntax construct and used to identify and manipulate constructs explicitly. In DS, however, pattern matching on what are effectively abstract syntax trees is used and the meaning of each abstract syntax construct is given by a function with cases for each sub-construct.

2.3. Example - rudimentary imperative language

Consider a rudimentary imperative language, with non-negative integers, addition, input, output, assignment and statement sequences. A program to output double the value of an input is:

```
read a;
b:=a+a;
write b
```

A BNF concrete syntax for this language is:

```
<statements> ::= <statement> ; <statements> | <statement>
<statement> ::= <assign> | <input> | <output>
<assign> ::= <identifier> := <expression>
<input> ::= read <identifier>
<output> ::= write <expression>
<expression> ::= <base> + <expression> | <base>
<base> ::= <identifier> | <number>
<identifier> ::= <alpha> <identifier> | <alpha>
<alpha> ::= a | b | ... | z
<number> ::= <digit> <number> | <digit>
<digit> ::= 0 | 1
```

Informally, the language allows for the association of `<identifier>`s and integers, either from the input or from `<expression>s,` and for the output of integers from `<expression>`s and hence from associations with `<identifier>`s.

The semantically significant syntactic categories are `<number>`s, which denote the base integer values, `<identifier>`s, which are associated with integers, `<statements>` which modify the input, output and associations between `<identifier>`s and integers, and `<expression>`s which retrieve integers associated with `<identifier>`s:

```
Syntactic domains

n ∈ <number>
i ∈ <identifier>
s ∈ <statements>
e ∈ <expression>
```

The abstract syntax for these domains is found by folding together rules to remove redundant unit productions and by dropping redundant terminal symbols:

```
Abstract syntax

s -> s s | READ i | WRITE e | i := e
e -> i | n | e + e
i -> i i | a | b | ... | z
n -> 0 | 1 | n 0 | n 1
```

The semantic domains are the non-negative integers, inputs and outputs, which are both sequences of integers, and states, which associate `<identifier>`s and integers:

```
Semantic domains

N == non-negative integers
I = N* == inputs
O = N* == outputs
S = <identifier> -> N == states
```

Standard operators, predicates and value denotations for these domains are assumed.

Semantic functions are defined for the syntactic categories. Abstract syntax constructs are stropped with the brackets `[...]` and identified by pattern matching. Non-terminals in the constructs are treated as selecting the corresponding sub-constructs on the right hand sides of equations.

For a `<number>` the equivalent integer value is found:

```
mi: <number> -> N

mi [0] = 0
mi [1] = 1
mi [n 0] = 2*(mi [n])
mi [n 1] = 2*(mi [n])+1
```

Note the assumption of arithmetic in the meta-language.

The meaning of an `<expression>` is an integer which is found using the state:

```
me: <expression> -> S -> N
```

When an `<expression>` is a `<number>`, its meaning is the corresponding value:

```
me [n] s = mi [n]
```

When an `<expression>` is an `<identifier>`, its meaning is the associated value in the state:

```
me [i] s = s [i]
```

When an `<expression>` is an addition, its meaning is the sum of the meanings of the operands in the state:

```
me [e1+e2] s = (me [e1] s)+(me [e2] s)
```

The meaning of a `<statements>` is the transformation of the initial state, input and output to a new state, input and output:

```
ms: <statements> -> S * I * O -> S * I * O
```

For a sequence of statements, the meaning of the first statement is found using the current state, input and output. The resulting state, input and output are then used to find the meaning of the rest of the sequence:

```
ms [s1 s2] (s,i,o) = ms [s2] (ms [s1] (s,i,o))
```

It is assumed that the meta-language provides tuples, for uncurried function arguments and for functions to return composite values, and pattern matching on tuples.

An `<assign>` updates the state with an association between the `<identifier>` and the meaning of the `<expression>` using the state, leaving the input and output unchanged:

```
ms [i:=e] (s,i,o) = (new s [i] (me [e] s),i,o)
```

Here function "updating" is used to model the state. This might be defined by:

```
new old lv rv = λ l.if lv=l
                    then rv
```

```
                    else old l
```

`new` is called with an old function, `old,` a left value, `lv,` and right value, `rv.` It returns a new function of an unknown left value `l` within which `lv` is associated with `rv.` When the new function is called, `l` is compared with `lv.` If they match then the associated right value `rv` is returned. Otherwise the old function is called with the unknown left value.

Note that it is assumed that the meta-language allows the use of a conditional construct.

An `<input>` updates the state with an association between the `<identifier>` and the first value in the input which is removed from the input, leaving the output unchanged:

```
ms [READ i] (s,i,o) = (new s i (hd i),(tl i),o)
```

The provision of lists for sequences in the meta-language, with constructor `:` and selectors `hd` and `tl,` is assumed.

An `<output>` updates the output with the meaning of the `<expression>` in the state leaving the state and input unchanged:

```
ms [WRITE e] (s,i,o) = (s,i,(me [e] s):o)
```

Explicit error handling, for example for an empty input or for the absence of an `<identifier>` in an state, has been omitted. This might be accommodated by adding error values to domains, testing explicitly for error conditions and passing error values back. Alternatively, the meta-language might be defined so that constructs pass back errors implicitly.

2.4. Operational semantics

DS is often contrasted with operational semantics(OS) where the meanings of syntactic constructs are defined in terms of an abstract interpreter or reduction rules. OS has had a somewhat chequered history. Its roots lie in McCarthy's definitional meta-interpreter for LISP [93] and was brought to prominence by Landin's definition of the semantics of the λ calculus in terms of the SECD machine [85]. The SECD machine was itself defined in a purely functional manner and was later extended with imperative primitives [83, 84] to define ALGOL 60. Subsequently, this approach formed the basis of the Vienna Definition Language(VDL) [158] which was used to define PL/1 [91]. Reynolds [125] contains a useful overview of early OS.

The abstract machine approach to OS was roundly criticised by proponents of DS on philosophical and practical grounds, which are considered in more detail below. The upshot was that this first approach to OS fell into disuse for language definition.

OS's fortunes were revived by Plotkin's development of structural operational semantics(SOS) [118] in the mid 1980's. SOS is based on transition systems written as axioms and rules. Schmidt [129] suggests that SOS and the earlier abstract interpreter OS are equivalent.

The use of SOS and the related natural semantics is now growing. For example, SOS has been used to define Standard ML [102] and CCS [101]. For example, SOS forms the basis of the Typol specification language used with the Mentor general syntax directed editor [38] and of Berry's program animator [15]. Nielson and Nielson [108] discuss proofs of equivalence between SOS and DS. Gordon [56] proves the equivalence of OS and DS for LISP and uses "operational" reasoning to prove properties of the DS.

The relationship between OS and DS will be considered in more detail below.

2.5. Axiomatic semantics

The third major approach to formal language definition is that of axiomatic semantics where a formal system, typically predicate calculus, is extended with axioms and rules of inference for program constructs. The intention is that the proof system encapsulates the meaning of programs by providing syntactic manipulations which are consistent with a model for a language. The roots of axiomatic semantics lie in Floyd's [46] system for flowchart programs, in particular his introduction of loop invariants to characterise cycles in flowcharts. Subsequently, Hoare [68] developed a system for Algol-like languages where iterative constructs were characterised by inductive assertions, similar in concept to loop invariants. Axiomatic semantics reached its apogee in Dijkstra’s [39] weakest precondition approach. This is now used widely as a basis for program verification [58] and derivation [41, 79], in particular for teaching.

Hoare and Lauer [67] discuss the relationship between different formal semantic approaches, in particular what they term interpretive (VDL operational), deductive (axiomatic) and relational/functional (denotational) semantics, and show that these approaches are consistent for a simple language. They suggest that a language might be defined in a number of different ways which are oriented to particular needs: thus, the deductive approach might be most suitable for proving properties of programs and the interpretive approach might be most suitable for guiding implementations. This view is supported by Nielson and Nielson [108].

Stoy [139], however, disagrees with such egalitarian complementarity and argues that DS is primary. Similarly, Donahue [40] argues that while denotational and axiomatic semantics are complementary, DS provides a model for axiomatic semantics.

Axiomatic semantics will not be considered further here.

2.6. The status of denotational semantics

Opinions differ widely as to the status of DS. It’s original proponents come from a Platonic tradition [115] which accords a privileged position to mathematics as the study of ideal, abstract entities, for example Strachey [141]. In their view, DS is to do with establishing correspondences between syntactic constructs and ideal, abstract functions; most certainly not to do with purely syntactic theories of reduction or evaluation. Thus, Scott [132] at the 1972 Courant Computer Science Symposium denied that reduction rules were enough to specify the semantics of the λ calculus. In discussion with Wegner, he claimed that his sense of meaning was more basic, supporting his view, in part, by summarising a proof by Bohm [19] that there are different λ calculus normal forms which are not reducible to each other but which compute the same function.

Central to this view is the distinction between abstract mathematical objects and the symbols that denote them. In discussing the difference between numbers and numerals Scott and Strachey [134] state that numbers are the abstract objects which provide interpretations for syntactic expressions constructed from numerals. Symbols for numerals should not be mistaken for numbers because there are many different syntactic forms for denoting the same abstract concepts.

Stoy, a close collaborator of Scott and Strachey, expands on this view in his book [139]. While agreeing that much progress can be made by defining a language with syntactic rule, he argues that this involves working with uninterpreted symbols. Associating symbols with interpretations requires proof that the symbol system is consistent with the model. He suggests that in giving an (old style) operational definition in terms of an abstract machine we also need to give a rigorous definition of that machine, and that the problem has been merely pushed back a level. Here, he echoes earlier comments by Strachey [142] that there is an infinite regress in attempting to define syntactic systems in terms of other syntactic systems. Hence, by implication, the primacy given to mathematics as the "fixed point" of this regression.

Stoy [139] suggests that properties of a function associated with a program, such as its existence and well-definedness, cannot be resolved through an operational definition. He states that the elucidation of such properties is a fundamental requirement for a mathematical semantics.

Gordon [57] is at pains to distinguish mathematical functions from function constructs in programming languages. The former are sets of pairs whereas the latter are rules. He also deploys a weaker variant of Scott’s argument that the same mathematical function can be denoted by many different function constructs.

This is reflected in the DS proponents’ lack of interest in concrete syntax on the grounds that many concrete forms may have the same underlying meaning and so should be related through a common abstract syntax which does away with irrelevant representational detail. Gordon echoes Scott and Strachey, stating that only structural details are needed for semantics: details like precedence are irrelevant.

To summarise, Platonist DS supporters think that there are ideal, abstract mathematical objects and that purely syntactic systems, in particular reduction rules, do not capture all their semantic properties.

This philosophical viewpoint led to many DS enthusiasts rejecting or downgrading the early OS approach to language definition on the grounds that even an abstract machine contained irrelevant concrete implementation details. Thus, Strachey [142] claims that as a result of his formulation of imperative constructs as applicative expressions there is no need to refer to an evaluation mechanism in semantics. This comment is in the context of Landin's addition of explicit imperative features to the SECD machine to enable the definition of ALGOL 60. Strachey [141] noted later that his approach had something in common with Landin's but differs in that the ultimate defining machine was in no way specialised and should only be able to evaluate pure λ expressions. He also says that it does this by making the computer store explicit in an abstract form. Ironically, more recent critics of OS give this as a reason for rejecting OS for containing unnecessary implementation details. Thus, Stoy [139] says that OS contain unnecessary implementation details which add unwarranted complication to semantics.

Both Gordon and Stoy concede that a Platonic view of DS is not essential but come to different conclusions about the resultant status of DS. Stoy [140] notes that a DS is like a functional program and that a formal definition could be regarded as an interpreter. DS would then be similar to Landin's OS. However, he identifies a problem with this view in that meta-linguistic assumptions may be reflected implicitly in the defined language. Expanding on this in his book [139], he says that a normal order or applicative order meta-language would give normal order or applicative order semantics to the defined language. Someone who does not know such properties of the meta-language would not be aware that the defined language had them. However, he acknowledges that Reynolds [125] shows how the order of evaluation can be defined explicitly in a definitional interpreter.

In [140] he also reiterates Strachey's concern that the functional language still needs to be defined and that the approach would reduce to another OS method. A DS should define relationships in a domain rather than mechanical evaluation rules and the ability to manipulate DS according to such rules is "an uncovenanted extra". He goes on to question the terming of DS as a "normal order" language, saying that conversion rules are irrelevant syntactic matters. Ironically, as an example in his book he evaluates a little program in detail using its DS.

Gordon [57] is quite candid about the need to discuss abstract objects with concrete notations, and suggests, albeit tongue in cheek, that perhaps notations are the only things that really exist. He then states that he intends to talk about abstract entities as if they do exist and are different from the notations that describe them, claiming that this is a standard approach which enables clarity of conceptualisation. He goes on to say that DS shorn of Platonism specifies compilation. Like Stoy, he uses "mechanical evaluation" to illustrate DS.

It is interesting to note that Milner [101] distinguishes denotational, operational and translational formal approaches to semantics.

Hayes and Jones [63] take the Platonic position to its logical conclusion and argue against the executability of specifications in general. Like the DS proponents, they say that a specification written in an executable notation will contain unnecessary implementation details. Furthermore, they argue that specification languages should be powerful enough to describe non-computable functions. Thus, a specification might contain non-computable clauses whose conjunction is computable. Alternatively, a non-computable specification might be transformed to make it implementable. The tenor of their argument is that a mathematical specification is more than an implementation. Here they concur with Scott and Strachey's claim [134] that implementations must be judged against mathematical standards.

The purist view of DS as the standard for semantics has not gone unchallenged. In the next section, various objections and counter positions are presented.

2.7. Responses to DS purism

A number of people have questioned the distinction between DS and OS. Brady [20] says that the arguments to a denotational function are very close to those of an operational language with static storage. He suggests that, here, DS effectively postulates an abstract machine in much the same way as OS. He argues that the evidence opposes strongly Stracheys' conception of meaning and the distinction between DS and OS. Implementation questions should be seen as central to the meaning of programs, at an appropriate level of abstraction: the dispute is really over the level of abstraction at which implementation details are tackled.

Similarly, Anderson et al [6] argue that DS definitions, which they call the Oxford Definition Method(ODM), have "underlying machine like aspects" which are basic to DS. In presenting their OS approach (SEMANOL) they argue that notions like "abstract machine", "mechanistic" and "mathematical" are inadequate to characterise the difference between DS and OS. Instead, they say that the difference lies in levels of abstraction and the choice of base concepts, and quote Scott to contend that DS practitioners are aware of this. However, they agree that DS succeeds in minimising implementation details and go on to argue that OS and DS are complementary. One advantage they see for SEMANOL is its executability.

McGettrick [95] also suggests that DS contain implementation details. He notes that DS introduces concepts like state, environment and store: thus DS shares problems of overspecification with OS.

A number of authors have suggested that writing DS is akin to programming. Schmidt [130] argues that a DS definition could be used directly as an implementation, hence treating DS as an implementable programming language. In a later report [131] he says that DS enables operational treatment as well as mathematical rigour. The semantic equations are rewrite rules which transform a program and input to an output. He goes on to suggest that valuation equations can be viewed either as a syntax directed translation scheme for compilation or as the control of an interpreter.

N. Jones and Schmidt [76] point out that a denotational definition may be regarded as a single λ expression which when applied to a parse tree and a set of inputs returns a set of outputs. These outputs may be found by reducing the application as far as possible, thus treating a DS directly as an interpreter.

Similarly, Paulson [112] notes that "tricks" for writing DS are general functional programming techniques. The meta-language is a functional programming language and a DS is a functional compiler or interpreter implementation. Chao and Bryant [28] also argue that as DS is written in a λ calculus variant then if the variant is machine readable then it can be regarded as a functional language. Watt [156] also notes that DS is expressed in what is effectively a functional language. As that language is executable a DS may be viewed as an interpreter.

Wand [153] states bluntly that DS is based on characterising meaning as translations from syntactic constructs to functions, conveniently written in the λ calculus. Thus, a λ calculus implementation enables the execution of translated program to reduce it to normal form.

As the old saw has it, the proof of the pudding is in the eating. A major motivation for DS was to enable implementations of languages which were demonstrably consistent with formal definitions. In the following sections, approaches to language implementation from DS are considered.

2.8. Implementations from denotational semantics

Even the DS purists recognise that techniques are needed for deriving implementations from DS definitions. Stoy [139], after Milne and Strachey [100], elaborates a rigorous four stage compilation process from so-called Standard Semantics (ie DS), through Store, Stack and Consecution Semantics to Pointer Semantics, which results in machine code. Each stage introduces more implementation detail and proofs of congruence between the stages are required. The last two stages involve operational definitions to specify the behaviour of an abstract machine and to describe compilation to its machine code.

Bjorner [17] and C. Jones [74] both discuss a similar approach using VDM to formulate DS for the production of compilers and interpreters, which also involves congruence proofs. In principle, a VDM formulation of a DS could form the basis of a language implementation through an approach to the direct implementation of VDM, as in, for example, Cottam’s [34] hand translation to Prolog or O’Neill’s [110] automatic translation to Standard ML.

Stepney et al [138] describe a related approach where a compiler is proved correct by using DS for both the defined language and the target machine. They use code templates to associate defined language constructs with target machine constructs and then use the DS’s to prove that compilation via templates preserves meaning.

Royer [128] criticises the general approach of congruence proofs on the grounds that correctness can only be established once the source and target semantics have been specified. Instead, she considers the constructive derivation of target semantics from DS using correctness preserving transformations, drawing on algebraic semantic techniques.

N. Jones and Schmidt [76] describe a methodology for generating provably correct compilers from DS. They view a DS as a mapping from a language to λ calculus. They then show how to compile λ calculus into state transition machines, a low-level flowchart-like program suitable for conversion to machine code. A DS, combined with a λ calculus to state transition machine compiler is then used to compile from the language to state transition machines.

There are approaches based on translation from DS to other intermediate formalisms. Ganzinger [53] discusses the transformation of DS to attribute grammars as a basis for compiler or interpreter implementation. Klaeren and Petzsch [81] discuss interpreter development by converting DS to algebraic semantics for direct interpretation or through subsequent translation to Pascal. Rakovsky and Collier [120] modify a Standard DS(SDS) to form an Implementation DS(IDS) embodying relatively low level implementation details. The congruence between the SDS and IDS is proved. They then generate a compiler written in BCPL by automatic translation from the IDS.

DS has also been used directly as the basis of a number of automatic language implementation systems using both compilation and interpretation. Five of these are now considered in more detail. Tofte [144] provides summary overviews of other related systems.

2.9. SIS

Mosses' Semantics Implementation System(SIS) [105], built in the late 1970's, was the first compiler generator based on DS. Definitions written in a DS style are used to implement a compiler from the defined language to LAMB, a variant of the Scott-Strachey λ calculus based LAMBDA semantic notation. LAMB output from the compiler is subsequently interpreted through call by need reduction.

LAMB is an extended λ notation, providing non-negative integers, quotations, truth values, tuples, parse trees and functions. In form, LAMB is, like many DS notations, akin to a functional language with constructs including infix and prefix arithmetic, logical and comparison operators, operators for tuple and parse tree manipulation, conditional expressions, and pattern matching.

SIS definitions have two stages. Syntax is written in GRAM, a context-free notation related to BNF with constructs for iteration (Kleene star) and ranges for ordered sequences of base lexical items, like digits or letters. A GRAM definition, in turn, consists of LEXIS, for defining lexical items, and SYNTAX, for syntax proper. GRAM is used to generate SLR(1) parse tables which are then used to generate LAMB parse trees from program texts. Internally, LEXIS is used to generate LAMB tuples to represent lexemes which are then parsed via the SYNTAX parse tables. SYNTAX includes notation for describing the form of parse trees, hence enabling the specification of the concrete/abstract syntax correspondence.

Semantics is specified in the Denotational Semantics Language(DSL), which is LAMB augmented with domain and function definitions, a case construct, function updating and ranges for nodes.

SIS has no provision for context sensitive aspects of syntax. These may either be ignored, or implemented as an explicit additional stage or pushed into the semantics.

SIS is written in BCPL and has been ported to a number of platforms. It is a multi-pass batch system.

Bodwin et al [18] made an extensive study of SIS in a teaching environment. They found that full SIS based language implementations are very large, even for small languages. They highlighted a number of annoying restrictions, in particular the adherence to SLR(1) syntax, limited semantic domains and the handling of separated sums of domains through parse trees. They also comment on the poor provision for error handling, in particular the lack of semantic equation type checking, and on the SIS implementation's general inefficiency. They conclude that while SIS is only suitable for experimentation with small DS definitions, it is, none the less, an important first step.

Peter Mosses himself has stopped working on SIS as he sees inherent problems with manipulating large DS†.

2.10. PSP

Paulson's PSP system [113, 112] is based on semantic grammars which are DS definitions written as attribute grammars. Generated compilers produce SECD machine code for subsequent interpretation using call by value reduction.

The semantic grammar notation enables function and type definitions, and rules specifying context free and context sensitive syntax, and semantics. As with SIS, the base semantic notation is similar to a functional language, providing boolean, integer string, tuple and union types, λ abstraction, function updating, and conditional and case control constructs.

The system consists of three stages. The Grammar Analyser produces parse and compilation tables from a semantic grammar. The Universal Translator then reads the tables and generates SECD machine code from a program. Finally, the Stack Machine optimises and interprets the SECD code. PSP is another batch system.

Pleban, a member of the group which investigated SIS, has also used PSP extensively. He reports [116] that PSP

† These comments were made in unpublished electronic mail to me on 29/8/90.

enables considerable experimentation, that attribute grammar and DS definitions may be readily implemented and that type checking enables many bugs to be located quickly. He also comments that testing a DS helps establish confidence in its correctness. Overall, he thinks that PSP is flexible enough to enable experimentation with a wide variety of definitional styles. However, he draws attention to the lack of an interactive interface, the absence of good error handling, the limited number of built in domains and the lack of support for the modular form of DS definitions.

2.11. PSG

Bahlke and Snelting's Programming System Generator(PSG) [10, 11] generates interactive programming environments from DS definitions augmented with attribute grammars. These environments include structure editors and what are effectively interpreters for the defined language, which enable experimentation with language fragments as well as with entire languages. A library system supports previously developed language fragments for incorporation in new languages. The system generates a compiler from a DS which, in turn, generates functional code from a program for interpretation in SECD machine style using call by need reduction.

Definitions consist of syntax, context conditions and semantics. The syntax consists of lexical structure, concrete syntax augmented with format syntax, abstract syntax, and titles and menus for the interactive environment. The concrete syntax is LL(1). The context conditions consist of scope and visibility rules and a data attribute grammar, augmented with format syntax for attributes. The semantics consist of domain definitions, auxiliary functions, semantic functions for the whole language and meanings for executable fragments.

The semantic notation is a functional language based on an applicative subset of META-IV [16]. It provides integer, real, boolean and string domains, lists, tuples and map data types, higher order functions, definitions and conditional expressions.

The authors comment [10] that the use of a modular approach to language design improves readability and reliability, and that a system like PSG eases rapid prototyping of new languages.

2.12. Wand's semantic prototyping system

Wand's semantic prototyping system [153] is based on Scheme, a dialect of LISP with simplified constructs for handling functions as values. Definitions are based on transducers, which are DS style definitions written in Scheme as associations between syntactic constructs and semantic equations. Transducers are processed to produce a parser for the language. Programs are then parsed to generate λ calculus in Scheme form for call by value reduction by the Scheme interpreter. The notation includes domain equations which are used for what is effectively type checking. Like Pleban, Wand notes that type checking greatly eases error detection in definitions. There is no abstract syntax stage. Wand comments that its use is cumbersome, preferring the direct use of concrete syntax.

2.13. MESS

Lee and Pleban [87, 117] criticise the use of λ calculus based notations for all levels of semantic description. They argue for the separation of underlying semantic models from semantic descriptions of particular languages. Macrosemantics, concerned with compile time issues, should refer to action based operators whose microsemantic details are hidden. These operators might realise run-time actions in arbitrary paradigms, for example declarative or imperative. Language definitions should be modularised, to separate out distinct components for semantically distinct constructs, thus enabling incremental language definition.

Their Modular Extensible Separable Semantics(MESS) system reflects these concerns. It generates abstract syntax trees from which the macrosemantics produces prefix-form operator expressions(POEs). The microsemantics then specify the meanings of the operators.

To begin with, macro- and micro-semantic definitions were written in a case based functional style accompanied by semantic domain signatures for semantic functions. Subsequently, an extension to a pure functional subset of SML has been used. It is not clear how concrete syntax is defined or whether there is any provision for context sensitivity.

The MESS syntax analyser generator is written in and produces Pascal syntax analysers which in turn generate abstract syntax trees represented as Scheme S-expressions. Abstract syntax descriptions are generated automatically by the analyser generator. The semantic analyser is written in Scheme. The microsemantics is processed to generate an interface file containing the names and signatures of operators. An implementation of the operators in Scheme is also generated. The macrosemantics is processed with the interface file to produce a compiler core in Scheme.

A program is translated into an abstract syntax tree from which POEs are generated in Scheme by the compiler core. These may be executed directly as Scheme programs together with the Scheme for the operators. Alternatively, code may be generated from the POEs. Initially, code generators were written in Prolog. MESS has now been extended to allow different styles of microsemantics, for example to produce λ calculus for interpretation or machine code for direct execution.

Lee and Pleban show that MESS generated code runs faster than that from PSP, which, they state, generated the fastest code prior to MESS. The performance of MESS and PSP is compared with that of Navel in chapter 6.

2.14. Limitations to DS based systems

The above systems demonstrate that it is possible to generate automatically language implementations from DS but share a number of disadvantages as aids to language design and experimentation. Language development involves moving repeatedly from abstract syntax to semantic equations and back as changes in one often have implications for the other. In turn, these may lead to changes in domain descriptions and domain equations for semantic functions. This suggests that an incremental approach is most appropriate, based on the manipulation of DS fragments, which may not constitute a full consistent DS, or of isolated definitional stages.

These systems all have the apparent virtue of reflecting closely the multi-stage structure of a DS but vary in their support of incremental language development. Paulson’s and Wand’s systems and SIS accept a DS en-masse. MESS accepts an entire DS for separate language modules with interface information for consistency checks with other modules. PSG accepts an entire DS for a language fragment: when an undefined construct is encountered during test program execution, the system prompts for details. However, none of the above systems offers specific support for isolated experimentation at each stage.

As discussed above, DS does not address the concrete representations of programs in languages. However, concrete representations are fundamental to practical programming and language implementations must provide some support for lexical and syntactic processing. The systems considered above all provide additional stages and notations for lexicons and concrete syntax. At minimum, these enable the definition of context free syntax. Those systems that retain abstract syntax as a basis for semantic equations also provide notations and stages for associating concrete and abstract syntax. Some also provide stages and notations for context sensitive concrete syntax. These all add to system inflexibility.

2.15. Programming language based DS implementation

An alternative to using a generic DS based system is to craft an implementation for a particular language from its DS in an extant programming language. As programming languages are unitary notations, they are simpler to work with than multi-stage notations like DS but their use for DS usually involves the omission or conflation of DS stages. The underlying type system is often used instead of explicit domain equations. Implementations may be based directly on concrete syntax or the concrete/abstract syntax correspondence may be hidden in abstract syntax tree construction during concrete syntax parsing. These simplifications need not lead to a loss of rigour provided the relationship between the DS and the mode of implementation is made explicit, and, ideally, is formalised.

There have been a number of proposals for the use of imperative languages to implement DS. For example, Pagan [111] discusses the use of ALGOL 68 as a meta-language for DS. Allison [5] translates DS into Pascal interpreters for abstract syntax trees. As noted above, Klaeren and Petzsch [81] also use Pascal. However, imperative languages are maximally distant from DS, lacking support for syntax and abstract syntax constructs, and having little if any correspondence with the semantic meta-language. This complicates substantially the construction of DS implementations and demonstrations of their correctness.

As discussed above, the close correspondence between the DS meta-language and functional languages has been noted widely, and functional languages languages have been used to implement DS. For example, Jouvelot [78], Takeichi [143], Chao and Bryant [28] and Watt [156] discuss the use of ML and SML for interpreter implementation from DS. (S)ML has applicative order semantics and is more efficient than normal order languages. However, semantic equations which depend on normal order must be modified through function abstraction or through the use of data structures to delay and force evaluation.

While contemporary functional languages are much closer than imperative languages to the DS meta-language, in particular because of the presence of multi-case recursive functions based on pattern matching, they lack support for syntax. Thus, lexical and syntactic processors must be constructed explicitly or an extant parser generator must be employed. For example, Jouvelot [78], considers the use of an embedded YACC parser generator with ML. Once again, this is a separate en-masse stage rather than an integral part of an extended ML. CAML [35], which also embeds YACC within ML, is more flexible. Parsing may be initiated at arbitrary points in functions. Sethi [135] presents the use of LEX and YACC for the syntactic front end with semantic actions written in PLUMB, a functional notation related to Mosses' DSL discussed above.

The logic language Prolog has also been used to implement DS. For example, Stepney et al [138] use an extended Prolog to implement prototypes for defined languages through direct DS translation. Slonneger [136] also discusses the translation of DS to Prolog. Prolog, while not as close to the DS meta-language as functional languages, provides pattern matching with multi-case recursive rules and the Definite Clause Grammar(DCG) notation for context free parsing over lists. However, lexical analysers must still be constructed to translate character sequences into symbols based on atoms or facts and DCG's must be augmented explicitly to construct abstract syntax trees.

Contemporary declarative languages are closer to DS than imperative languages and are supported by interactive implementations which are more flexible and better suited to incremental development than en-masse DS systems. However, the absence of or limitations to constructs for handling syntax complicates the use of extant languages for the direct implementation of DS.

2.16. Interpreters from functions and grammars

The overall objective of the research discussed in the rest of this thesis is to investigate the close integration of syntax processing constructs within a functional language. The intention is to enable dynamic experimentation with language designs based on DS by running them directly as interpreters.

The core language is modeled on applicative order SASL [150] as a basis for semantic equations with a lazy list constructor to facilitate normal order semantic constructs. This core is extended with integral constructs for defining context free grammar rules, which coerce automatically strings to parse trees, and with a pattern matching construct to identify parse trees. To simplify DS implementation, Navel does not support abstract syntax, domain descriptions or domain equations.

Concrete and abstract syntax are necessarily close. Concrete syntax is certainly larger and contains details which are irrelevant for semantics. Thus, the use of concrete syntax results in slightly larger syntactic construct descriptions and in redundant semantic rules for unit productions. However, the loss of abstract syntax greatly simplifies definition manipulation. In practice, concrete and abstract syntax are not truly uncoupled: changes in abstract syntax often lead to corresponding changes in concrete syntax and vice versa. As noted above, this is the approach taken by Wand.

Domain equations correspond to strong typing in programming languages. However, DS is based on untyped λ calculus although domain equations correspond to type signatures in contemporary strongly typed functional languages. These are based on first order polymorphic type checking [45] which is decidable but excludes the direct definition of a number of important higher order λ calculus constructs, for example the Y combinator:

```
λ G.(λ s.(G (s s)) λ s.(G (s s)))
```

which is used for defining recursion. Decidable type checking for higher order types is still subject to research. Wand, Pleban, and Lee and Pleban comment that domain checks ease the propagation and detection of definition errors. However, explicit domains add additional notational and hence implementation stages. The alternative adopted here is weak typing with run time type checking. While this is less efficient than strict type checking as type checks are repeated throughout evaluation, it simplifies substantially the implementation and avoids costly type checking at compile time. Incidentally, SIS does not actually use the domain equations to check definitions.

Navel does not contain any additional constructs for context sensitive processing. Instead, grammar rules are full values and functional abstraction over rules enables the definition of context sensitive aspects of concrete syntax.

The design of core Navel is discussed in chapter 3. Chapter 4 considers extensions for syntactic processing.

Chapter 3

The Navel core language

3.1. Introduction

The design of the semantic meta-language is central to the provision of an integrated language for DS implementation. This chapter considers the development of core Navel for semantic equations.

Navel is situated through discussion of the relationship between DS and functional languages, and a chronological presentation of the development of functional languages. Next, core Navel is introduced informally through examples. Finally, a DS for core Navel is elaborated. The following chapter considers the addition of constructs to the core language for handling syntax.

3.2. DS notations

As discussed in chapter 1, DS is based on the λ calculus. In principle pure λ calculus could be used for semantic equations. In practice, however, the λ calculus rapidly becomes cumbersome for describing even relatively trivial semantics. Instead, higher level notations are used, based on the addition of useful, generic constructs as "syntactic sugar" to the λ calculus. As discussed above, these usually include basic types like booleans and integers, constructed types such as lists and tuples, notations for type manipulation, global and local definitions, conditional expressions and recursion.

Alas, while there are substantial similarities, there is no one standard DS notation. Thus, Stoy uses single Greek letters for naming semantic functions, Gordon uses single Roman letters whereas Tenant does not name functions at all, using instead an abstract syntax construct to imply directly the corresponding semantic function. Stoy, Gordon and Tenant all use distinct typefaces to distinguish different components of DS but they do not use the same sets of distinguishing typefaces. For conditional constructs, Stoy and Gordon use a LISP meta-expression form:

```
<condition> -> <expression1> , <expression2>
```

whereas Tenant uses a more traditional mathematical form:

```
<expression1> if <condition> = true
<expression2> if <condition> = false
```

This lack of a canonical DS notation has the dubious benefit of allowing a certain amount of freedom in the form of concrete implementations of DS, as implied above in discussion of DS based systems. In particular, as discussed above, several authors have commented on the similarities between DS notations and functional languages, and have suggested that DS notations might be implemented as functional languages, the course followed here. Thus, functional languages will be reviewed next, before the detailed design of Navel is considered.

3.3. Characterising functional languages

A distinction is often made between declarative languages, which supposedly specify what is to be computed, and imperative or procedural languages, which supposedly specify how something is to be computed. This distinction, asserted by some declarative language enthusiasts, carries the implication that declarative languages are somehow more high level or fundamental than imperative languages. However, this classification, which is reminiscent of that made between DS and OS discussed above, is simplistic and misleading: declarative programs have operational readings which suggest that they are as much to do with how something is computed as their imperative siblings.

The roots of this distinction lie in the presence or absence of the Church-Rosser property of evaluation order independence [139], which is a more fruitful starting point. Declarative languages are usually sub-divided into logic languages, based on propositional and predicate calculus, and functional languages, based on the λ calculus and recursive function theory [82]. In both cases name/value associations cannot be modified subsequently and hence both are Church-Rosser. By comparison, imperative languages are closely related to evaluation order dependent Turing machines [146] and are based on name/value association modification through assignment.

Note that concrete implementations of even declarative languages must embody some definite evaluation strategy or strategies. Indeed, declarative languages usually have clearly defined evaluation orders: a useful basis for language comparison. Note, also, that some declarative languages contain imperative features, for example Prolog's [33] is, assert and retract, LISP's [137] SETQ and SML's [102] ref type.

Programs in declarative languages are, indeed, often more succinct than their imperative equivalents and this may give the impression that they are somehow less concerned with operational niceties. However, this is to do with orthogonality of abstraction mechanisms and conciseness of notations; issues which are not paradigm specific.

For example, functional languages usually allow full abstraction over functions. This enables generalisation through higher order functions which may be applied to function arguments to return function results. However, several contemporary imperative languages also provide full functional abstraction, for example PS-algol [27] and Napier [104].

Similarly, contemporary declarative languages often provide notations for the direct association of functions or predicates and data through pattern matching. This allows a closer structural correspondence between programs and data than in older imperative languages where data are manipulated indirectly through explicit conditional and selection constructs. However, imperative languages may also provide pattern matching, for example POP-11 [12] and Icon [59], albeit through explicit operators.

Here, "functional language" will refer to Church-Rosser languages which allow full functional abstraction, and will stretch to non-Church-Rosser languages with pure functional subsets.

3.4. Evaluation orders

As noted above, functional languages are usually defined with specific evaluation orders. In an expression, a reducible sub-expression is called a redex and consists of a function applied to an argument. Here, λ notation is used for functions:

```
λ <name>.<body expression>
```

<name> is called the bound variable and identifies the points of abstraction in the body <body expression>, an arbitrary expression. A redex has the form:

```
λ <name>.<body expression> <argument expression>
```

where <argument expression> is an arbitrary expression. To reduce a redex, some form of the argument expression is substituted for all occurrences of the bound variable in the body expression. Evaluation then consists of repeatedly reducing redexes until there are none left. The expression is then said to be in normal form. Alternatively, evaluation may stop when some acceptable final form still containing redexes is reached. Two common final forms [45] are weak head normal form, where the final expression is a function whose body may still be a redex, and head normal form, where again the final expression is a function whose body may not be a redex but may contain redexes.

The evaluation order depends on how the next redex in an expression is chosen. Consider the function application:

```
λ <name>.<body expression> <argument expression>
```

For normal order evaluation, the leftmost outermost redex is always chosen. Thus the next redex is:

```
λ <name>.<body expression> <argument expression>
```

itself, and is reduced by substituting the unevaluated <argument expression> for all occurrences of <name> in <body expression>.

Note that normal order results in multiple copies of unevaluated argument expressions appearing in the body, for potential subsequent repeated evaluation. Normal order is also known as call by name.

For applicative order evaluation, the leftmost innermost redex is always chosen. Thus, in:

```
λ <name>.<body> <argument expression>
```

if <argument expression> is a redex or contains redexes then it or they are further reduced giving say <reduced argument expression> which contains no more redexes or is in the acceptable final form. The leftmost innermost redex is now:

```
λ <name>.<body expression> <reduced argument result>
```

which is reduced through the substitution of <reduced argument expression> for all occurrences of <name> in <body expression>.

Note that in applicative order an argument expression is only evaluated once. However, all arguments to a function are evaluated before substitution which may lead to unnecessary or indeed non-terminating evaluation of subsequently unused argument expressions. Thus even pure applicative order languages still require what are effectively normal order conditional constructs to avoid evaluation of actually unselected condition options prior to selection. Applicative order is also known as call by value and languages which are based on applicative order are termed strict.

The 2nd Church-Rosser theorem [139] shows that if reduction of an expression in any evaluation order will terminate then normal order will: that is, there are expressions whose reductions will terminate in normal order but not in, say, applicative order. This suggests, naively, that if termination is a priority then normal order should be used even though it may be less efficient than applicative order.

3.5. Lazy evaluation

Lazy evaluation is an implementation strategy which is intended to combine the benefits of normal and applicative order evaluation. Like normal order, argument evaluation is delayed, but, a sharing mechanism ensures that there are references to the same single copy of an argument wherever multiple copies might have occurred. When an argument is finally evaluated all such references subsequently correspond to a single copy of the result of evaluation. Thus, like normal order, unnecessary evaluation of unselected arguments is avoided and, like applicative order, repeated evaluation of the same redex is prevented. There is, however, an overhead in keeping track of and updating shared references. Lazy evaluation is also known as call by need.

Lazy evaluation was first proposed in the context of LISP lists [48, 64]. Hughes [70] promotes lazy evaluation, in particular lazy infinite structures, as a major advantage of functional programming. However, Traub [145], while supporting lazy evaluation, argues that lazy structures are not particularly useful and introduce unacceptable evaluation overheads. Instead, the use of lenient evaluation is proposed, where expressions are delayed until all their arguments have been evaluated, in contrast to lazy evaluation, where expressions are delayed until no further progress can be made without them.

As we shall see, functional languages are not always purely lazy/normal order or applicative order. Various permutations of lazy/normal or applicative order argument evaluation and lazy/normal order or applicative order structures have been implemented or proposed.

A language will be referred to here as fully lazy if it is normal order and intended for lazy implementation.

3.6. Functional languages

3.6.1. LISP

The earliest language with recognisable functional components was LISP(LISt Processing) [93], developed by McCarthy in the late 1950's for the Advice Taker project but now a major language in its own right. LISP is an imperative language but has a pure functional subset. McCarthy's early work uses the unimplemented meta-expression (M-expression) form which has a relatively rich syntax, reminiscent of contemporary functional languages, for example having a λ calculus like notation for functions and the `...->...,...` form of conditional expressions.

LISP as actually implemented is based on the syntactically minimal symbolic expression (S-expression) form, consisting solely of nested bracketed function applications. This syntactic brevity is belied by the huge range of special functions normally provided, giving LISP a positively baroque semantics.

In particular, the manipulation of functions as values involves additional operators for identifying explicitly functions as returned values, `FUNCTION,` and for applying them subsequently to arguments `FUNCALL.` This was due to the so-called FUNARG problem which was concerned with the binding of free variables in the context of early LISP's dynamic scope rules. Contemporary LISPs, like Common Lisp [137] are lexically scoped but retain explicit function value identification. Scheme [2] is a recent LISP dialect which does not require explicit function value identification. LISP functions are uncurried: the operators mentioned above may be used for currying but there is no simple notational support for curried functions.

LISP is an applicative order language which is weakly typed with run-time type checking. An important LISP innovation was the provision of the list as the sole structured data type. As LISP is weakly typed, functions and lists may be polymorphic enabling the construction and manipulation of data structures containing arbitrary types. LISP has no structure or pattern matching: list selection requires the explicit CAR and CDR operators.

3.6.2. ISWIM

Landin's ISWIM(If you See What I Mean) notation [86], developed in the mid-1960's, also foreshadowed contemporary functional language designs. Landin comments that ISWIM was intended to overcome LISP's over reliance on lists, poor storage allocation, excess bracketing and divergences from traditional notations. He also notes that ISWIM is closer to the LISP M-expressions. ISWIM was intended as a family of languages with no specific problem orientation.

ISWIM is based on syntactic extensions to the λ calculus. Conditional expressions use a form of the `...->...,...` notation. Both the `let...in...` and `...where...` forms of local definitions are provided and `rec` is used to introduce recursive definitions, intimating the need for a fixed point functional. ISWIM provides infix operators and indentation to indicate program structure. Landin refers in passing to an experimental implementation. Burge's [22] notation is strongly influenced by ISWIM.

3.6.3. CPL

CPL [13], developed in the early 1960's, is another imperative language with a pure functional subset. Within an ALGOL60-like imperative framework, CPL also provides the `...where...` form of local definitions, which, its authors say, gives it equivalent expressive power to the λ calculus. CPL conditional expressions use a form of the `if...then...else...` notation. CPL is strongly typed but has a `general` type which enables a weak form of polymorphism. CPL has constructs for defining functions to be either normal or applicative order. CPL provides typed array and polymorphic list structures. List selection is through structure matching.

3.6.4. PAL

Evan's PAL [44] was developed in the mid 1960's for teaching programming linguistics and was highly influential for the development of subsequent languages. PAL has applicative(functional) and imperative subsets. PAL is applicative ordered and weakly typed with run-time typing. The applicative subset is based on the λ calculus and provides functions as values with currying. Both `let...in...` and `...where...` are used for local definitions. The `...->...,...` form of conditional expressions is used. Recursive functions are nominated by `rec.` Data structures are based on arbitrary length tuples which may be nested. Tuple selection is through structure matching.

A curious effect of the sharing semantics given to assignable variables is that the value of a "bound" variable may change after function application if a function is applied to an assignable variable which is subsequently assigned. However, the values of expressions using built in basic operations, for example for arithmetic and logic, with assignable variables do not change if those variables are subsequently changed.

Evans refers to a BCPL implementation on the IBM 7094 using an abstract machine similar to Landin's SECD machine. Evans acknowledges ISWIM as a source and says that the first PAL implementation, in LISP by Landin and Morris, was closer to ISWIM than more recent versions. Perhaps this is the ISWIM implementation that Landin refers to?

3.6.5. POP-2

POP-2 [23] was developed for Artificial Intelligence applications in the late 1960's. POP-2 is another imperative language with a pure functional subset. Its designers acknowledge LISP, CPL and ISWIM as sources. POP-2 improves substantially on LISP's brackets, with a richer syntax replacing special functions. In particular, POP-2 allows functions as full values. While functions are uncurried, POP-2 introduces partial application as a means of binding some of a functions arguments, albeit in a clumsy form with explicit nomination of partially applied arguments. The `if...then...else...` form of conditional expressions is used. POP-2 retains lists but provides a wider range of data structures including records and arrays. Like LISP, POP-2 is applicative order and weakly typed with run-time typing. POP-2 has been developed into POP-11 [12] which runs in the POPLOG environment. As well as POP-2's structure selection operators, POP-11 also provides constructs for explicit pattern matching.

3.6.6. GEDANKEN

Reynolds' GEDANKEN [126] is yet another imperative language with a pure functional subset and was developed in the late 1960's. GEDANKEN incorporates fully the λ calculus and allows curried functions. It is applicative order and weakly typed. In GEDANKEN there are no distinct structure types. Instead, structures are implemented as functions though some additional syntax is provided for common structures like vectors. `if...then ...else...` is used for conditional expressions. Reynolds cites LISP, PAL, ISWIM and POP-2 as sources.

3.6.7. SASL and KRC

Turner's SASL(St Andrews Standard Language), a highly influential pure functional language, is also based on syntactic additions to the λ calculus, providing functions as full values. SASL is weakly typed with run-time typing. The sole data structure is the list with both explicit selection notation and structure matching. In the 1975 manual [150] Turner says that SASL corresponds to the applicative subset of PAL except for the use of lists instead of tuples.

The first version was developed in the early 1970's [149] and was applicative order. Here there are explicit denotations for function values and functions may be curried. Definitions are based on the `let...in...` notation extended to allow simultaneous definitions. Conditional expressions are based on the `...->...,...` form. Recursive functions are nominated by `rec.` List selection is through the `hd` and `tl` operators. This version underwent minor syntactic changes in 1974/75 [150]. In particular, all definitions were assumed to be recursive unless nominated as non-recursive by `new.`

A second version was developed in 1976 [151], heralding a break with earlier functional languages. The principle change is that the 1976 SASL is normal order with lazy constructors. There are no longer explicit denotations for function values: all functions are created through definitions which are based on the `...where...` notation. Functions are defined by cases using pattern matching, an extension to structure matching allowing the presence of constants as well as structures of variables in bound variable expressions. This replaced partially the earlier SASL's explicit argument testing in conditional expressions, though conditional expressions are necessarily retained for constraint checking on arguments. List elements are selected by applying the list to the appropriate integer index. The 1976 SASL was freely distributed and used widely for teaching and research.

Turner's KRC(Kent Recursive Calculator) [147] is similar to but simpler than the 1976 SASL. In particular, the `...where...` notation enabling local definitions is dropped so only top-level definitions are permitted. Once again, functions are defined by cases through pattern matching. An innovation is the introduction of notation for set abstraction through what Turner called ZF(Zermelo Frankel) expressions, now known as list comprehension. This enables the definition of lists using generators and guards to qualify properties of general list members. Notation for lists of integer sequences is also introduced. Guards are also used to qualify expressions in place of explicit conditional expressions.

3.6.8. Hope

Hope [24] is a pure functional language developed in the late 1970's. At first sight, it is yet another variant on the languages discussed above, providing function values, `let...in..` and `...where...` local definitions, uncurried functions, case definitions with pattern matching, applicative order function call evaluation and the `if...then...else...` form of conditional expressions. However, Hope embodies a number of important innovations in functional language design.

Hope provides strong typing based on Hindley-Milner [103] polymorphic type checking enabling substantially more static program checking than earlier languages. However, this is compromised by the need to give explicit type definitions for functions, unlike more recent languages where function types may sometimes be deduced from their definitions and uses. Hope also enables the use of type variables in type specifications, enabling the definition of polymorphic higher order functions within a strongly typed framework.

Rather than polytyped lists as the sole data structure, Hope provides both tuples as fixed length sequences of different types and lists as variable length sequences of the same type. Furthermore, both strict and lazy list constructors are provided. Hope also provides data constructors for the definition of discriminated union types. The use of type variables in data constructor definitions enables the definitions of polymorphic structures.

Hope provides unparameterised modules as a simple way of encapsulating data abstraction. Modules may contain statements indicating which of their contents are private and which are public, enabling control over access to implementation details.

The authors acknowledge LISP and ISWIM as major influences, and comment on similarities to Prolog, ML, SASL and OBJ.

Hope is primarily a teaching and research language. In particular, it used for the investigation of parallel architectures and language implementations at Imperial College.

3.6.9. Standard ML

ML was developed in the late 1970's as the meta language for Milner's LCF program proof system. ML has now matured into Standard ML(SML) [102] and is one of the first languages to have a full formal definition based on SOS.

SML is an applicative order, polymorphic typed language. Function definitions are based on cases with pattern matching. Functions are full values and may be curried. Local definitions use the `let...in...` and conditional expressions use the `if...then...else..` notations. Data structures include lists, tuples, records and concrete types for disjoint unions. SML provides both parameterised abstract data types for local information hiding and modules for structuring large systems. Unlike Hope, type specifications are not needed where the types of objects can be deduced from their context of use.

SML is actually an imperative language supporting assignable `ref` type values, statement sequences and iteration. However, the pure functional subset of SML is used widely.

SML is a major teaching and research vehicle and its use is increasing for industrial applications, all due to the general availability of free, robust, efficient implementations on a variety of platforms.

There are a number of SML variants. For example, Concurrent ML [123] augments SML with threads and channel based message passing. Lazy ML(LML) [8] is a pure functional lazy version of SML with minor syntactic variations. CAML [35] provides lazy as well as applicative order lists.

3.6.10. Miranda

Turner's Miranda† [148] developed in the early 1980's, has its roots in the earlier SASL and KRC, discussed above, and like them is a pure functional language. As in Hope and SML, Miranda has polymorphic types. Unlike SML and Hope, Miranda is fully lazy.

Miranda has case structured function definitions using pattern matching which may be curried. There are no function value denotations. Data structures include lists, tuples and concrete types. Parameterised abstract data types and modules are also provided. List comprehension and notation for lists of number sequences are provided. Conditional expressions are generalised to guarded expressions. Local definitions are based on `...where...`.

Early Miranda also provided laws [37] for constraining the values of concrete types. Laws are based on re-write rules and are applied when values of a type are created. Laws are not supported in later versions of Miranda.

---

† "Miranda" is a trademark of Research Software Ltd.

Miranda is a commercial product and is used widely for teaching and research.

3.6.11. Haskell

Haskell [122] is a relatively new functional language being developed cooperatively by groups at several institutions, primarily in Europe and North America. Haskell is a pure functional, polymorphic, fully lazy language. It provides case structured function definitions with pattern matching and guards. Function values have denotations and may be curried. Data structures include lists, tuples and concrete types. Modules subsume abstract data types. List comprehension and conditional expressions are also provided. Both the `...where...` and `let...in...` forms of local definitions are included. Haskell also provides a functional form of arrays, a feature often added on in SML implementations.

Haskell is in many ways the PL/1 of functional languages, containing a portmanteau of features found in other languages and hence providing a variety of ways of achieving the same effects. For example, recursive functions may be defined using pattern matching or guards or explicit conditional expressions.

Haskell is still under development and, at present, is used mainly for research and teaching.

3.6.12. Related languages

Backus' 1977 Turing Award lecture [9] proposed functional programming as a way of avoiding the theoretical and practical difficulties associated with von Neumann (ie non-Church-Rosser imperative) languages. However, Backus' approach differs substantially from that discussed above. He criticises λ calculus as a basis for functional programming for giving users too much freedom to invent new combinatory forms rather than becoming familiar with a few generic forms. Instead he proposes FP systems based on a set of objects, a set of functions from objects to objects, an operation for function application and a set of functional forms which are used to combine functions and objects to form new functions. His criterion for selecting particular functional forms is that they should provide powerful programming constructs with properties characterised by algebraic laws with maximal strength and utility in relating them to other forms.

In his example FP system, objects are either atomic numbers and booleans or sequences of objects. Primitive functions are provided for atom and sequence predication, arithmetic and sequence reversal, distribution, transposition, appendage and length. Basic functional forms are for function composition, construction, conditional expressions, constant functions, insertion (ie folding a function over a sequence) and applying to all (ie mapping a function over a sequence).

Individual FP systems are fixed. Backus also proposes the more general FFP systems (Formal Systems for Functional Programming) which enable the definition of new functional forms.

Backus' lecture has been extremely influential in promoting interest in functional programming. However, his FP approach has found less favour, perhaps in part because the resulting notation is succinct to the point of incomprehensibility. This may reflect the influence of APL [71], an imperative language developed in the early 1960's based on a small number of powerful operators for array and vector manipulation. A number of FP implementations are available.

A variant of functional languages are the single assignment languages, developed for use on data flow parallel architectures, for example SISAL [4] and Id Nouveau. [109] Such languages are a strange mix of functional and imperative notations, including what are apparently iteration and sequencing for assignment. However, assignments are read as initialising new instantiations of a variable rather than changing an existing variable: hence "single assignment" akin to bound variable/value associations. Thus, iteration, for example, provides a convenient notation for array access.

The OBJ specification language [55] is also related in practice, if not in original intention, to functional languages. In OBJ, specifications are written as equational definitions of abstract data types, in a similar manner to case-based recursive function definitions with pattern matching, with guarded case option expressions. Indeed, the developers of OBJ2 [51], the successor to OBJ, say explicitly that it is a functional programming language. Implementations are available of executable subsets of OBJ, for example UMIST OBJ [52].

3.7. Core Navel

Core Navel is modeled on the 1975 SASL. When this work was started, in 1975 at St Andrews University, SASL was actively under development as a basis for teaching and research into formal aspects of programming languages. SASL has a close correspondence to the λ calculus and is similar in many ways to DS notations. At that time, plausible alternatives were LISP or POP2: these both have functional features but are distant notationally from DS. Also, SASL is a small language making it relatively easy to implement: an important consideration as the main focus of this research was experimentation with syntax/semantics linkage in DS implementations rather than solely with semantic notation. However, Navel is not the same as SASL: there are a number of minor syntactic variations and semantic extensions which are intended to ease DS implementation. Nonetheless, Navel reflects many aspects of SASL: a now somewhat dated language. This is discussed further in chapter 9.

Navel, like SASL, is applicative order. As noted above, DS notations assume normal order. Indeed, there are a number of constructs which cannot be used directly in the usual form with applicative order semantics, for example the fixed point finder Y:

```
λ G.(λ s.(G (s s)) λ s.(G (s s)))
```

which does not terminate in applicative order. However, applicative order is a pragmatic choice to optimise ease of implementation and run-time speed. Experimentation with non-lazy normal order evaluation in predecessors of Navel suggested that it was several times slower than applicative order. Furthermore, terminating equivalents may often be found for constructs which do not terminate in applicative order. For example, the construction of delayed evaluation conditional expression options with applicative order is discussed in Michaelson [99]. However, for some constructs normal order is essential and so lazy normal order lists are introduced. The need for these is discussed further in chapter 7.

Rather than evaluating expressions fully to full normal form, a weaker variant of head normal form is used. In Navel, evaluation stops when a function is reached without any further evaluation of the body. While this does not guarantee a unique final form, it retains information which is useful for interpretation of results.

Like SASL, Navel is weakly typed with run-time typing. As discussed in chapter 2, DS is based on untyped λ calculus and there are a number of constructs which cannot be typed with first order polymorphic type checking. To simplify both the language and the implementation, DS domain definitions are not supported: hence, weak typing is appropriate. However, while the absence of strong type checking simplifies and speeds up compilation, run-time type checking leads to a loss of interpretive speed.

Navel inherits from SASL the boolean, character and integer base types. These seem adequate for initial experimentation with formal definitions. In particular, real numbers are not provided. This simplifies implementation and evades awkward questions of operator overloading and type coercion. Navel also inherits polytyped lists as the sole initial data structuring mechanism. Lists may be used to implement arbitrary non-circular data structures. Providing only lists also greatly simplifies store management.

Perhaps the greatest weakness Navel inherits from SASL for DS implementation is the absence of case structured function definitions. Flat function definitions without pattern matching greatly simplifies implementation but leads to a lack of correspondence with the DS definitional style. This is discussed in more detail, and a compromise extension presented, in chapter 4.

The next section gives an informal overview of core Navel, that is Navel without syntax constructs. The following section discusses the formal semantics of core Navel.

3.8. Core Navel informal overview

Navel's base types are the booleans, integers and characters. Booleans are the values:

```
true false
```

with the binary infix operators:

```
|   - disjunction
&   - conjunction
```

and unary prefix:

```
not - negation
```

in increasing order of precedence. Boolean expressions using these operators may be composed with the brackets:

```
( )
```

For example:

```
true & false ==> false
not true and false ==> false
not (true and false) ==> true
```

Integers are negative, zero and positive whole numbers. The integer operators are the infix binary:

```
+ - addition
- - subtraction
* - multiplication
/ - division
% - modulo
```

and the prefix unary:

```
- - negation
```

with precedence:

```
+ - (binary)
* / %
- (unary)
```

in increasing order.

Integer expressions using the above operators may be composed with the brackets:

```
( )
```

For example:

```
6*7 ==> 42
294/(3+4) ==> 42
87%33+3*7 ==> 42
```

Characters have the form:

```
'<character>'
```

for example:

```
'a' '9' '='
```

UNIX-like escape sequences are used for non-printing and context sensitive characters, for example:

```
'\n' - newline
'\"' - "
'\;' - ;
```

Characters may be used in integer contexts with the corresponding ASCII values, for example:

```
'9'-'0' ==> 9
```

The unary prefix operator:

```
char
```

converts an integer to a character, for example:

```
char 65 ==> 'A'
```

Like SASL, untyped lists are the sole data structure. The empty list is:

```
()
```

The infix binary list concatenation operator is:

```
:
```

For example, a list of cubes:

```
1:8:27:64
```

`()` may be omitted from the end of null terminated lists. For example, a list of squares:

```
1:4:9:16:
```

and a list of lists of integers and their squares:

```
(1:1):(2:4):(3:9):(4:16):
```

The prefix unary list head selector is:

```
hd
```

and the prefix unary list tail selector is:

```
tl
```

for example:

```
hd (1:2:3:4:) ==> 1
tl (1:2:3:4:) ==> 2:3:4:
```

Nested applications of `hd` and `tl` need not be bracketed, for example:

```
hd tl tl (1:2:3:4:) ==> 3
```

Lists of characters have a simplified quoted representation:

```
"hello" == 'h':'e':'l':'l':'o':
```

The infix binary operator:

```
@
```

is used for indexed list selection, for example:

```
"hello"@2 ==> 'e'
```

Note that all of the above operators are built in to the language and are not available as function denotations in their own rights. This minor loss of orthogonality again eases implementation.

Identifiers or names are used for global and local definitions and for bound variables. A name is a sequence of alpha-numeric and _ characters starting with an alphabetic character, for example:

```
banana catch22 Route_66
```

Function values are based on λ notation. A function value has the form:

```
lam <bound variable expression> . <expression>
```

where <expression> is an arbitrary expression. At simplest, the <bound variable expression> is a single name, as for example, in the identity function:

```
lam x.x
```

and the doubling function:

```
lam v.2*v
```

Arbitrarily complex bound variable lists may be used for uncurried functions. For example, to select the head of a list:

```
lam h:t.h
```

For curried functions, intervening lams and .s may be dropped:

```
lam <bound variable1>.lam <bound variable2>.<expression> ==
lam <bound variable1> <bound variable2>.<expression>
```

For example the sum of squares function:

```
lam x.lam y.x*x+y*y == lam x y.x*x+y*y
```

A function application has the form:

```
( <function expression> <argument expression> )
```

<function expression> is evaluated to return a function value. <argument expression> is evaluated. <function expression>s <bound variable expression> is then matched against the value of <argument expression> and individual bound variables are associated with the corresponding argument values. The <function expression>s body expression is then evaluated. For example, to double an integer:

```
(lam x.2*x 21) ==> 42
```

and to reverse the elements of a three element list:

```
(lam a:b:c:.c:b:a: "but") ==> "tub"
```

Strict bracketing of function applications is not mandatory: in an application, the function expression is always applied to the argument expression immediately to its right:

```
((<func expression1> <arg expression1>) <arg expression2>) ==
<func expression1> <arg expression1> <arg expression2>
```

For example:

```
lam x y.x*x+y*y 3 4 ==> 25
```

Prefix unary type testers are provided:

```
isbool == true if argument is a boolean
ischar == true if argument is a character
isnumb == true if argument is an integer
islist == true if argument is a list
isstring == true if argument is a string
isfunc == true if argument is a function
```

Comparison expressions are based on binary infix operators. The equality and inequality operators are:

```
= <>
```

Equality applies to all types. For functions, equality requires full structural identity. The comparison of values of different types returns `false.` For example:

```
7<>7 ==> false
(1:2:3:) = tl (0:1:2:3:) ==> true
'a'="a" ==> false
(lam x.x) = (lam x.x) ==> true
```

The ordering operators are:

```
< <= > >=
```

They apply to integers and also characters, where alphabetic ordering is tested. For example:

```
42>43 ==> false
'a'<'b' ==> true
```

Local definitions have the form:

```
let <name> = <expression1>
in <expression2>
```

`<expression2>` is evaluated with `<name>` associated with the value of `<expression1>`.

For example, to define a local square function

```
let sq = lam x.x*x
in (sq 3)+(sq 4) ==> 25
```

For the association of names and function values in local definitions, the `lam` may be dropped and the `.` replaced by the =:

```
let <name> = lam <bound variable>.<expression> in ... ==
let <name> <bound variable> = <expression> in ...
```

The previous example simplifies to:

```
let sq x = x*x
in (sq 3)+(sq 4) ==> 25
```

For example, to define a local sum of squares function:

```
let sum_sq x y = x*x+y*y
in sum_sq 3 4 ==> 25
```

Multiple local names may be defined through structure matching, for example:

```
let (given:family):age = ("Minnie":"Minx"):9
in given ==> "Minnie"
```

Conditional expressions have the form:

```
if <expression1>
then <expression2>
else <expression3>
```

`<expression1>` is evaluated to return a boolean. If it is `true` then `<expression2>` is evaluated. Otherwise `<expression3>` is evaluated.

For example, to find the absolute value of an integer:

```
let abs x =
 if x>=0
 then x
 else -x
in ...
```

and to find the larger of two values:

```
let max x y =
 if x>y
 then x
 else y
in ...
```

The case expression generalises the conditional expression. It has the form:

```
case <expression> of
<expression1> -> <expression2>,
<expression3> -> <expression4>,
 ...
<expressionN>
```

and is equivalent to:

```
if <expression>=<expression1>
then <expression2>
else
 if <expression>=<expression3>
 then <expression4>
 else
  ...
  else <expressionN>
```

For example, to determine the type of a letter:

```
let type l =
 case true of
 ’a’<=l & l<=’z’ -> "lower",
 ’A’<=l & l<=’Z’ -> "upper",
 ’0’<=l & l<=’9’ -> "digit",
 "other"
in ...
```

For function definitions, the function name may appear in the function body enabling recursion. For example, the following two functions are of unknown origin or purpose but are used widely to illustrate recursion:

```
let fac n =
 if n=0
 then 1
 else n*(fac n-1)
in ...

let fib n =
 case n of
 0 -> 1,
 1 -> 1,
 (fib n-1)+(fib n-2)
in ...
```

Lists are normal order and implemented through lazy evaluation. Thus, function calls in lists are not evaluated until the corresponding element is selected. This enables the use of recursion to define infinite structures with finite representations. For example, consider an infinite list of squares:

```
let sq x = x*x
in
 let sqs n = (sq n):(sqs n+1)
 in
  let sqlist = sqs 1
  in ..
```

Initially, `sqlist` is:

```
(sq 1):(sqs 2)
```

After selecting say:

```
sqlist@3 ==> 9
```

`sqlist` is:

```
(sq 1):(sq 2):9:(sqlist 4)
```

Note that Navel operators are not functions. Hence, expression list elements which are not top-level function calls are evaluated.

The local definition form is extended to enable simultaneous function definitions:

```
let <name1> = <expression1>
and <name2> = <expression2>
 ...
in <expression>
```

Here, if the `<expression>`s are functions then they may mutually recurse through reference to the `<name>`s. For example, to define a hierarchy of arithmetic functions through recursion:

```
let succ x = x+1
and pred x = x-1
and add x y =
 if y=0
 then x
 else add (succ x) (pred y)
and mult x y =
 if y=0
 then 0
 else add x (mult x (pred y))
and pow x y =
 if y=0
 then 1
 else mult x (pow x (pred y))
in ...
```

This form may also be used for non-mutually-recursive non-function definitions.

In principle, a Navel program is simply an expression, for example a simultaneous local definition defining several functions followed by an expression which uses them. For interactive use, however, this is extremely inflexible. In particular, it is useful to be able to enter global definitions, which persist through an interactive session, interspersed with expressions to exercise them. Hence, for interactive use, a program is a sequence of global definitions and expressions separated by `;`.

A global definition takes the form:

```
def <name> = <expression>
```

with the same simplified function definition notation as for local expressions. For example:

```
def reverse1 l2 l1 =
 if l1=()
 then l2
 else reverse (hd l1):l2 (tl l1);
def reverse = reverse1 ();
def palindrome p = p=(reverse p);
palindrome "able was I ere I saw elba"
```

3.9. Core Navel formal definition

In the following sections, a formal DS definition of Navel is presented. Note that Navel was designed to be reducible to pure untyped λ calculus through the finite application of rewrite rules. This approach is discussed for a language which is effectively core Navel, in detail, in Michaelson [99].

Mosses [106] gives a formal definition of a language called MSL(Mathematical Semantics Language), which is similar to the notation for DS semantic equations, in MSL itself. The definition is smaller than that which follows for Navel. The use of MSL to define MSL enables the elision of details which are made explicit in the DS for Navel as MSL constructs are usually defined by similar MSL constructs. For example, function values and applications are implicit whereas in the DS for Navel an explicit closure mechanism is used. Also, the MSL definition contains no type checking or error handling. Finally, MSL is a simpler language than Navel.

Mosses does not provide any semantics for syntactic processing or for associating syntactic constructs with semantic actions. The semantics of syntax in Navel is given in chapter 4.

3.9.1. Concrete Syntax

The concrete syntax of core Navel is described using a BNF variant:

```
 [...] ==> optional

 {...} ==> choose one

 # ... # ==> ... is an English description instead of B.N.F.

 ’s over-ride notation e.g. ’[’, ’]’, ’{’, ’}’, ’|’
```

It is intended that the concrete syntax be self explanatory: only salient points are highlighted. The concrete syntax is:

```
<program> ::= [<definitions> ;] <expressions> [; <program>]

<definitions> ::= <definition> [; <definitions>]

<definition> ::= def <name> [<nameseq>] = <expression>

<nameseq> ::= <namelist> [<nameseq>]

<namelist> ::= <namebase> [: [<namelist>]]
```

This construct is space sensitive so that the `<nameseq>`:

```
<namebase1>:<namebase2>
```

consisting of a single list of two `<namebase>`s may be distinguished from:

```
<namebase1>: <namebase2>
```

consisting of a null terminated list of one `<namebase>` followed by another `<namebase>`.

```
<namebase> ::= <name> | ( <namelist> )
```

```
<expressions> ::= <expression> [; <expressions>]
```

```
<expression> ::= <if> | <case>  | <application> | <let>
```

```
<if> ::= if <expression> then <expression> else <expression>
```

```
<case> ::= case <expression> of <cases>
```

```
<cases> ::= <expression> [-> <expression> , <cases>]
```

```
<application> ::= <logexp> [<application>]
```

Function applications are of lower precedence than operator expressions. Thus, operator expression arguments need not be bracketed.

```
<let> ::= let <defs> in <expression>
```

```
<defs> ::= <name> <nameseq> = <expression> [and <defs>] |
           <namelist> = <expression> [and <defs>]
```

```
<logexp> ::= <logterm> ['|' <logexp>] | <lambda>
```

```
<lambda> ::= lam <nameseq> . <logexp>
```

A function application as the body of a function must be bracketed.

```
<logterm> ::= <logfactor> [& <logterm>]
```

```
<logfactor> ::= [not] <logbase>
```

```
<logbase> ::= <list> [{< <= = >= > <>} <list>] |
              {isnumb ischar isbool islist isstring isfunc} <aexp>
```

```
<list> ::= <aexp> [: [<list>]]
```

As with `<namelist>` above, there is an implicit space after a null terminated list so that the single argument:

```
<aexp>:<list>
```

may be distinguished from the two arguments:

```
<aexp>: <list>
```

where the first is a null terminated list.

```
<aexp> ::= <term> [{+ -} <aexp>] | true | false | <string>
```

```
<term> ::= <factor> [{* / %} <term>]
```

```
<factor>::= [-] <base> | <base> [<select>]
```

```
<base> ::= <name> | <number> | ( <expression> ) | () |
           {char hd tl} <base> |
           ''' <character> '''
```

```
<string> ::= "" | " <characters> "
```

```
<characters> ::= <character> | <character> <characters>
```

```
<select> ::= @ <number> | @ ( <expression> )

<name> ::= <alpha> [<rest of name>]

<rest of name> ::= <alpha> [<rest of name>] | <digit> [<rest of name>] |
                   _ [<rest of name>]

<alpha> ::= 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 |
            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

<number> ::= <digit> | <digit> <number>

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<character> ::= #any printable ASCII# | \ #any printable ASCII#

<infix operator> ::= ’|’ | & | <= | < | = | >= | > | <> | + | - | * |
                     / | % | @

<prefix operator> ::= not | isnumb | ischar | isbool | islist |
                      isstring | isfunc | - | hd | tl | char
```

3.9.2. Abstract syntax

The syntactic domains for the abstract syntax are:

```
p ∈ <program>
d ∈ <definitions>
nl ∈ <namelist>
n ∈ <name>
e ∈ <expression>
ca ∈ <cases>
ld ∈ <defs>
i ∈ <integer>
dg ∈ <digit>
s ∈ <characters>
c ∈ <character>
pop ∈ <prefix operator>
iop ∈ <infix operator>
```

The abstract syntax is:

```
p -> e | d e | p p

d -> def n = e | d d

e -> if e then e else e |
     case e of ca |
     let ld in e |
     e e |
     e iop e | pop e |
     e : e |
     lam nl . e |
     true | false | c | i | n | ()

ca -> e -> e , ca | e

ld -> nl = e | ld and ld
```

```
nl -> n | nl: | nl : nl

iop -> '|' | & | <= | < | = | >= | > | <> | + | - | * | / | % | @

pop -> not | isnumb | ischar | isbool | islist | isstring | isfunc |
       - | hd | tl | char

i -> dg | i dg

dg -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Note that a number of assumptions are made about the transition from concrete to abstract syntax. Simplified function definitions are turned into explicit name/function associations:

```
def <name> <nameseq> = <expression> ==>
 def <name> = lam <nameseq>.<expression>

let <name> <nameseq> = <expression> in ... ==>
 let <name> = lam <nameseq>.<expression> in ...
```

Similarly, simplified curried functions are turned into explicit sequences of single bound variable functions:

```
lam <namelist> <nameseq>.<expression> ==>
 lam <namelist>.lam <nameseq>.<expression>
```

Null terminated lists are ended with explicit empty lists:

```
<expression>: ==> <expression>:()
```

Strings are converted back to explicit empty list terminated character lists:

```
"<character> <characters>" ==> '<character>':"<characters>"
```

and the empty string is replaced by the empty list:

```
"" ==> ()
```

3.9.3. Semantic domains

The semantics of core Navel is based on the notation used for the example in chapter 2. First of all, the semantic domains are presented. Note that all domains have associated membership predicates.

The value domain `V` defined below is the union of the following domains.

The error domains consists of a single error object:

```
error == error value
 with iserror : V -> B == predicate
```

`iserror` returns a boolean indicating whether or not its argument is the `error` value.

The boolean domain consists of the boolean constants with operators for disjunction, conjunction, negation, equality and inequality:

```
B == booleans == true false
 with or : B * B -> B == disjunction
      and : B * B -> B == conjunction
      not : B -> B == negation
      = : B * B -> B == equality
      <> : B * B -> B == inequality
      isbool: V -> B == predicate
```

The integer domain consists of the negative and positive integers with operators for addition, subtraction, multiplication, division, modulo, negation, equality, inequality and orderings:

```
N == integers == -MAXINT .. MAXINT
 with + : N * N -> N == addition
      - : N * N -> N == subtraction
      * : N * N -> N == multiplication
      / : N * N -> N == division
      mod : N * N -> N == modulo
      = : N * N -> B == equality
      <> : N * N -> B == inequality
      < : N * N -> B == less than
      <= : N * N -> B == less than or equal
      >= : N * N -> B == greater than or equal
      > : N * N -> B == greater than
      isint: V -> B == predicate
```

The character domain consists of the ASCII character set with operators for integer value, character construction from integer, equality, inequality and orderings:

```
C == characters == ASCII values
 with chartoint : C -> N == integer value of character
      inttochar : N -> C == construct character from integer
      = : C * C -> B == equality
      <> : C * C -> B == inequality
      < : C * C -> B == less than
      <= : C * C -> B == less than or equal
      >= : C * C -> B == greater than or equal
      > : C * C -> B == greater than
      ischar: V -> B == predicate
```

The list domain consists of pairs of values or suspensions (defined below), with operators for selection, construction, equality and inequality:

```
L == lists == (V + Su) * (V + Su)
  - () == empty list
 with : : V * V -> L == construction
      hd : L -> V + error == select head
      tl : L -> V + error == select tail
      = : L * L -> B == equality
      <> : L * L -> B == inequality
      islist: V -> B == predicate
```

The domain of states consists of associations between the syntactic domain of `<name>`s and values. It is used to model bound variable/value associations:

```
S == states == <name> -> V
  - nullstate == empty state - returns error
```

Function values are modelled by closures which pair function meanings with the defining states. A function meaning is a λ function from an argument value and a state to a result value, defined below. Thus, the function domain consists of pairs of states and function meanings, with operators for construction, selection, equality and inequality:

```
F == functions == S * (V -> S -> V + error)
 with makefun : S * (V -> S -> V + error) -> F == constructor
      funstate : F -> S == select state
      funmean : F -> (V -> S -> V + error) == select meaning
      = : F * F -> B == equality
      <> : F * F -> B == inequality
      isfunc: V -> B == predicate
```

When a function is applied to an argument, value the meaning mapping is invoked with that value and the defining state.

Suspensions are used to represent unevaluated function calls in normal order evaluated lists. The suspension domain consists of pairs of function and argument values, with operators for construction and selection:

```
Su == suspensions == F * V
 with makesusp : F -> V -> Su == construction
      suspfun : Su -> F == select first of suspension pair
      susparg : Su -> V == select second of suspension pair
      issusp : V -> B == predicate
```

When a suspension is selected from a list, the function is applied to the argument.

The domain of values is the union of the above domains apart from states and suspensions:

```
V == values == B + N + C + L + F + error
```

Note that in principle pure λ calculus suffices as a basis for semantics. However, the use of distinct domains substantially simplifies presentation. Michaelson [99] describes the construction of the error, boolean, integer, character and list domains in the λ calculus.

Note the overloading of the equality, inequality and ordering operators to simplify presentation.

3.9.4. Program semantics

A semantic equation is given for each abstract syntax construct. Normal order is assumed.

A program computes a final state and value from an initial state:

```
mp : <program> -> S -> S * V
```

If a program is an expression then the initial state and the value of the expression in that state are returned:

```
mp [e] s = s:(me [e] s)
```

If the program is a definition followed by an expression then a new state is found from the initial state and definitions, and is returned along with the value of the expression in that new state:

```
mp [d e] s = let news = (md [d] s news)
             in news:(me [e] news)
```

Note that the construction of the new state is curiously circular. This device, after Gordon [57], makes explicit the finding of the fixed point for mutually recursive definitions, usually elided by vague constructs employing `fix` and `...`s to indicate an arbitrary number of mutually recursive components. For more details, see the clauses for definitions below. As discussed in chapter 7, this device cannot be employed directly in a pure applicative order language.

If a program is a sequence of programs, then the first is evaluated to find an intermediate state and value. That state is then used to evaluate the second program, returning a final state and the concatenation of the values from both programs:

```
mp [p1 p2] s = let news1:v1 = mp [p1] s
               in
                let news2:v2 = mp [p2] news1
                in news2:(v1:v2)
```

3.9.5. Global definition semantics

A definition computes a final state from an initial state and a final state:

```
md : <definitions> -> S -> S -> S
```

If a definition is single then the expression is evaluated. If it returns an error then the initial state is returned. If the expression returns a function then the initial state is returned, updated with an association between the name and the function, with the state in the function's closure replaced by the final state. Note that this use of a final state found by normal order evaluation enables recursive references to other defined names whose definitions are yet to be evaluated. Otherwise, the initial state is returned, updated with an association between the name and the expression value:

```
md [def n = e] s news = let e1 = me [e] s
                         in
                          if iserror e1
                          then s
                          else
                           if isfunc e1
                           then new s [n] (makefun news (funmean e1))
                           else new s [n] e1
```

States are updated by a functional which wraps a layer of `if...then...else...` round the old state function to test for the new name:

```
new old lv rv l = if l=lv
                  then rv
                  else old l;
```

If a definition is a sequence of definitions, then the state from the second is found using the final state and the state from the first, found in turn using the final state and the initial state. Note that the sharing of the common final state enables mutual recursion:

```
md [d1 d2] s news =  md [d2] (md [d1] s news) news
```

3.9.6. Constants and lists semantics

For an expression, a value is found using the state:

```
me : <expression> -> S -> V
```

For booleans, the corresponding value is returned:

```
me [true] s = true
me [false] s = false
```

For characters, a character is constructed:

```
me [c] s = inttochar <ASCII value of [c]>
```

For integers, the integer value is constructed:

```
me [i] s = ival [i]

ival : <integer> -> N
ival [0] = 0
 ...
ival [9] = 9
ival [i dg] = 10*(ival [i])+(ival [dg])
```

For names, the associated value from the state is found:

```
me [n] s = s n
```

For an empty list, the empty list is returned:

```
me [()] s = ()
```

For list construction, both expressions are evaluated lazily and a pair is constructed:

```
me [e1:e2] s = let h = mlazy [e1] s
               in
                if iserror h
                then error
                else
                 let t = mlazy [e2] s
                 in
                  if iserror t
                  then error
                  else h:t
```

For lazy evaluation, function evaluation components are evaluated lazily and a suspension is formed. Other components are evaluated fully:

```
mlazy : <expression> -> S -> V + Su
mlazy [e1 e2] s =  let ev1 = mlazy [e1] s
                   in
                    if iserror ev1
                    then error
                    else
                     let ev2 = mlazy [e2] s
                     in
                      if iserror ev2
                      then error
                      else makesusp ev1 ev2
mlazy [e] s = me [e] s
```

Note this cheap form of "strictness analysis": any operator is assumed to be strict.

3.9.7. Function and function applications semantics

For a function, a closure pair is formed from the state and the function meaning. The function meaning is a function of a state and an argument. When the function is called, the defined bound variable list is bound to the argument, extending the state from the function value, and the function body is then evaluated in that new state:

```
me [lam nl.e] s -> makefun s λ a.λ ss.let news = bind [nl] a ss
                                      in
                                       if iserror news
                                       then error
                                       else me [e] news
```

For a function application, the function and argument expressions are evaluated and the function’s meaning is applied to the argument in the function’s state:

```
me [e1 e2] s -> let f = me [e1] s
                in
                 if not isfunc f
                 then error
                 else
                  let a = me [e2] s
                  in
                   if iserror a
                   then error
```

```
                        else (funmean f) a (funstate f)
```

The test `isfunc` will catch an error value returned from the evaluation of `[e1]`.

Bound variable lists are bound recursively to the argument list. Components of the argument list may be suspensions so they are evaluated:

```
bind : <namelist> -> V -> S -> S
bind [n] a s = new s [n] a
bind [nl:] a s = if not islist a
                 then error
                 else
                  if (msusp (tl a))<>()
                  then error
                  else
                   let hv = msusp (hd a)
                   in
                    if iserror hv
                    then error
                    else bind [nl] hv s
bind [nl1:nl2] a s = if not islist a
                     then error
                     else
                      let hv = msusp (hd a)
                      in
                       if iserror hv
                       then error
                       else
                        let tv = msusp (tl a)
                        in
                         if iserror tv
                         then error
                         else bind [nl1] hv (bind [nl2] tv s)
```

3.9.8. Suspension evaluation semantics

For evaluation of a potential suspension, if it is indeed a suspension then its components are evaluated and the function component applied to the argument component. For a non-suspension, the value is returned:

```
msusp : V -> V
msusp s = if issusp s
          then
           let f = me (suspfun s) nullstate
           in
            if not isfunc f
            then error
            else
             let a = me (susparg s) nullstate
             in
              if iserror a
              then error
              else (funmean f) a (funstate f)
          else s
```

3.9.9. Conditional and case expression semantics

For a conditional expression, the value of the condition determines which option is evaluated:

```
me [if e1 then e2 else e3] s = let b = me [e1] s
```

```
                                    in
                                     if isbool b
                                     then
                                      if b
                                      then me [e2] s
                                      else me [e3] s
                                     else error
```

For a case expression, the initial expression is compared with each case option selection expression in turn. If a match is found then the corresponding case option value expression is evaluated. If no match is found then the final expression is evaluated:

```
me [case e of ca] s = let ev = me [e] s
                      in
                       if iserror ev
                       then error
                       else mcase ev [ca] s

mcase : V -> <cases> -> S -> V
mcase ev [e1 -> e2 , ca] s = let ev1 = me [e1] s
                             in
                              if iserror ev1
                              then error
                              else
                               if ev=ev1
                               then me [e2] s
                               else mcase ev [ca] s
mcase ev [e] s = me [e] s
```

3.9.10. Local definition semantics

For a let expression, the definitions are evaluated in the initial and final state as for global definitions above. The body expression is then evaluated in the resultant state:

```
me [let ld in e] s = let news = mld [ld] s news
                     in
                      if iserror news
                      then error
                      else me [e] news
```

For a single definition, the argument is evaluated and bound to the name(s) in the initial and final states. For simultaneous definitions, the first is carried out in the initial and final state, producing an intermediate state for carrying out the second, with the final state:

```
mld : <defs> -> S -> S -> S
mld [nl = e] s news = let a = me [e] s
                      in
                       if iserror a
                       then error
                       else letbind [nl] a s news
mld [ld1 and ld2] s news = let s1 = mld [ld1] s news
                           in
                            if iserror s1
                            then error
                            else mld [ld2] s1 news
```

To bind a value to a defined name or names:

```
letbind : <namelist> -> V -> S -> S -> S
```

if the definition is of a single name then if a function is being defined then a new closure referring to the final state is associated with the name in the initial state. For a non-function definition, the name is associated with the value in the initial state:

```
letbind [n] a s news = if isfunc a
                       then  new s [n] (makefun news (funmean a))
                       else new s [n] a
```

For structure matching, value components are bound recursively to names in name structures:

```
letbind [nl:] a s news = if not islist a
                         then error
                         else
                          if (msusp (tl a))<>()
                          then error
                          else
                           let av = msusp (hd a)
                           in
                            if iserror av
                            then error
                            else letbind [nl] av s news
letbind [nl1:nl2] a s news = if not islist a
                             then error
                             else
                              let a1 = msusp (hd a)
                              in
                               if iserror a1
                               then error
                               else
                                let a2 = msusp (tl a)
                                in
                                 if iserror a2
                                 then error
                                 else letbind [nl2] a2
                                       (letbind [nl1] a1 s news) news
```

3.9.11. Prefix operator semantics

For prefix expressions, the argument is evaluated and the operator is then applied:

```
me [pop e] s = let a = me [e] s
               in
                if iserror a
                then error
                else applypop [pop] a

applypop : <prefix operator> -> V -> V
```

For numeric negation, the value, if integer or character, is negated:

```
applypop [-] v = if isint v
                 then -v
                 else
                  if ischar v
                  then - (chartoint v)
                  else error
```

For boolean negation, the value, if boolean, is negated:

```
applypop [not] v = if isbool v
                   then not b
```

```
                       else error
```

For type predicates, the appropriate predicate is applied:

```
applypop [isnumb] v = isint v
applypop [ischar] v = ischar v
applypop [isbool] v = isbool v
applypop [islist] v = islist v
applypop [isstring] v = if islist v
                        then checkstring v
                        else error
applypop [isfunc] v = isfunc v

checkstring : L -> B
checkstring v = if v=()
                then true
                else
                 if not islist v
                 then false
                 else (ischar (msusp (hd v))) and
                       (checkstring (msusp (tl v)))
```

For integer to character conversion, the value, if an integer, is converted:

```
applypop [char] v = if isint v
                    then inttochar v
                    else error
```

For head and tail selection from lists, the component is selected and evaluated if a suspension:

```
applypop [hd] v = if islist v
                  then msusp (hd v)
                  else error
applypop [tl] v = if islist v
                  then msusp (tl v)
                  else error
```

Note that this does not define lazy lists as evaluated function calls are not updated in lists. Laziness is an implementation technique which supports efficiently normal order evaluation. Defining explicitly lazy evaluation involves introducing a heap domain of functions which associate addresses and values. List elements are then replaced with heap addresses associated with suspensions. When an element is selected, the heap is updated with a new association between the address and the evaluated suspension.

3.9.12. Infix operator semantics

For infix expressions, both operands are evaluated and the operator is applied:

```
me [e1 iop e2] s = let ev1 = me [e1] s
                   in
                    if iserror ev1
                    then error
                    else
                     let ev2 = me [e2] s
                     in
                      if iserror ev2
                      then error
                      else applyiop [iop] ev1 ev2
```

For infix arithmetic operators, the appropriate operation is performed, after domain checks and coercions where necessary:

```
applyiop : <infix operator> -> V -> V -> V
applyiop [+] e1 e2 = marith λ x.λ y.x+y e1 e2
applyiop [-] e1 e2 = marith λ x.λ y.x-y e1 e2
applyiop [*] e1 e2 = marith λ x.λ y.x*y e1 e2
applyiop [/] e1 e2 = marith λ x.λ y.x/y e1 e2
applyiop [%] e1 e2 = marith λ x.λ y.x mod y e1 e2

marith: V -> V -> N + error
marith f x y = if isint x and isint y
               then f x y
               else
                if isint x and ischar y
                then f x (chartoint y)
                else
                 if ischar x and isint y
                 then f (chartoint x) y
                 else
                  if ischar x and ischar y
                  then f (chartoint x) (chartoint y)
                  else error
```

For infix boolean operations, the appropriate operation is carried out after domain checks:

```
applyiop [&] e1 e2 = if not isbool e1 or not isbool e2
                     then error
                     else e1 and e2
applyiop [|] e1 e2 = if not isbool e1 or not isbool e2
                     then error
                     else e1 or e2
```

For comparisons and orderings, the appropriate operations are carried out after domain checks:

```
applyiop [=] e1 e2 = mcomp λ x.λ y.x=y e1 e2
applyiop [<>] e1 e2 = mcomp λ x.λ y.x<>y e1 e2
applyiop [<=] e1 e2 = mord λ x.λ y.x<=y e1 e2
applyiop [<] e1 e2 = mord λ x.λ y.x<y e1 e2
applyiop [>=] e1 e2 = mord λ x.λ y.x>=y e1 e2
applyiop [>] e1 e2 = mord  λ x.λ y.x>y e1 e2

mcomp : (V -> V -> B) -> V -> V -> B
mcomp c x y = if islist x and islist y or
                 isfunc x and isfunc y or
                 isbool x and isbool y or
                 isint x and isint y or
                 ischar x and ischar y
              then c x y
              else
               if isint x and ischar y
               then c x (chartoint y)
               else
                if ischar x and isint y
                then c (chartoint x) y
                else false

mord : (V -> V -> B) -> V -> V -> B
mord c x y = if isint x and isint y or
                ischar x and ischar y
             then c x y
             else
              if isint x and ischar y
              then c x (chartoint y)
               if ischar x and isint y
```

```
                    then c (chartoint x) y
                    else false
```

Note that for mixed integer/character operations, characters are coerced to integers and integer operations carried out.

For list indexing, after appropriate domain checks the corresponding element is selected:

```
applyiop [@] e1 e2 = if islist e1 and isint e2
                     then select e1 e2
                     else error

select : L -> N -> V
select l n = if not islist l or l=()
             then error
             else
              if n=1
              then msusp (hd l)
              else select (msusp (tl l)) (n-1)
```

Chapter 4

From syntax to semantics in Navel

4.1. Introduction

In DS, semantic functions are associated with abstract syntax constructs, typically through case structured pattern matching on abstract syntax trees. As noted above, DS is not concerned with questions of concrete syntax and it is assumed that there is some predetermined correspondence between concrete and abstract syntax. However, language implementation depends crucially on associations of some form between concrete representations and semantic actions, whether or not mediated by abstract syntax. In particular, DS based implementation systems must have some means of expressing concrete syntax and of transforming concrete representations into forms suitable for semantic associations.

As discussed in chapter 2, most DS based systems are multi-stage and provide notations for concrete syntax and the association of concrete and abstract syntax, as well as for abstract syntax and associating abstract syntax with semantics. The aim of this research was to build a system based on the direct association of concrete syntax constructs and semantics, utilising a unitary notation informed by functional languages. Chapter 3 discussed the genesis of the core Navel language for defining semantics. This chapter considers the development of Navel constructs for defining concrete syntax and associating concrete syntax and semantics. First of all, constructs for syntax manipulation in other languages are discussed briefly. Next, syntax handling in Navel is motivated and presented informally. Finally, the formal definition of Navel from chapter 3 is extended with the new syntax manipulation constructs.

4.2. Syntax manipulation constructs in programming languages

Grammars for describing programming language syntax was one of the earliest areas in Computer Science to be fully formalised, and is widely applicable to problem solving in general as well as to language definition and implementation. It is, thus, surprising how few languages provide facilities for syntactic manipulation.

4.2.1. BCL

One of the earliest languages for syntactic processing was BCL [65], developed in the 1960’s. BCL is based on groups, a conflation of subprograms and data structures. A group consists of alternatives which are in turn sequences of constants, variables with known type properties, names of groups, in line groups and commands. Groups may be used for pattern matching on input data: alternatives are matched until one succeeds; constants match input constants; variables are set to input values of appropriate types; names associated with groups result in the invocation of the corresponding group bodies, potentially recursively; commands are executed in sequence during matching. BCL was intended as a compiler-compiler language with groups combining syntactic recognition and semantic actions, in particular parse tree construction or code emission. Indeed, BCL was used to define and implement the Bell Laboratories low level linked list language L6 on the Atlas computer [69]. BCL has no structure matching and so structures must be manipulated explicitly.

4.2.2. SNOBOL

SNOBOL [60] was developed in the 1960’s as an imperative string manipulation language. Like BCL, at the heart of SNOBOL lies pattern matching. Patterns consist of alternatives which are sequences of constants, nested explicit patterns, references to other patterns through names, and function calls. Patterns may have variables associated with components which are set to the appropriate match values. Semantic actions may be attached to pattern matching through success and failure tests. SNOBOL provides structured user defined data types which may be used to represent parse trees. In the absence of structure matching, structures are manipulated explicitly.

4.2.3. Icon

Icon [59] is a direct descendant of SNOBOL, developed in the late 1970's. Unlike SNOBOL, Icon's syntax has a traditional ALGOLic flavour, extended with constructs for pattern matching on strings. Once again, patterns consist of alternative sequences of constants, nested explicit patterns, pattern references through names and procedure calls. Semantic actions may be invoked through success and failure tests after pattern matching. Icon provides a variety of data structures including lists. Usually, pattern expressions return the matched strings but components of pattern expressions may be enclosed in lists to return list representations of parse trees. Icon lacks structure matching and structures must be manipulated explicitly.

4.2.4. Extensions to ALGOL 60 and ALGOL 68

There have also been a number of proposals for extensions to extant languages for syntactic processing. In the mid 1970's, Maurer and Stucky [92] suggested augmenting ALGOL 60 with a new `chartree` type, based on BNF-like productions and a string type. Assignment or input of a string to a `chartree` variable would parse the string and set the variable to the corresponding structure. Semantic actions are then based on explicit tree inspection and manipulation. A prototype implementation was constructed but the work was discontinued. A pleasing feature of this approach is the automatic coercion of a string to the corresponding parse tree. The authors give an example of the generation of machine code for a simple imperative language.

In the late 1970's, Linneman [90] proposed an extension to ALGOL 68 for generation from rules which may also be used for parsing. Rules are defined in a BNF style. Each rule has an associated data type whose values are derivation trees, corresponding to the result of parsing with the rule. Initially, derivation trees are formed by reading a string into a variable of derivation tree type which parses the string and associates the variable with the corresponding derivation tree. Tree components are selected by root inspection based on structure matching expressions. These consist of alternative variable structures corresponding to different top level tree structures. A successful match binds the variables to the corresponding tree components and returns an integer to indicate which option succeeded. Trees are also formed by generating expressions which check whether or not a string is derivable from a rule. Generating expressions may contain references to trees which are effectively replaced by the corresponding strings, found by inorder traversal and leaf string concatenation. This enables the construction of new trees from the components of existing trees. Printing a derivation tree outputs a concatenated string of leaf strings found by inorder traversal. While Linneman discusses ALGOL 68 equivalents for his constructs, it is not clear whether this proposal was implemented.

4.2.5. Prolog DCGs

Perhaps the most successful language extension for syntactic processing has been that for Definite Clause Grammars(DCGs) in Prolog [33]. At simplest, DCGs are a form of BNF grammar which are translated into Prolog rules and applied to lists of lexemes represented as Prolog atoms. As Prolog is fully backtracking, this provides full context free parsing. The association of variables with DCG rule components enables information to be passed around during the parse, in particular for parse tree construction and for context sensitive processing. Parse trees are built from Prolog structures for subsequent inspection and manipulation through pattern matching. DCGs may also be augmented with Prolog semantic actions. As DCGs operate on lists of atoms, lexical analysers must be constructed explicitly to convert character based representations of source programs to symbol lists. Furthermore, DCGs define parsers but not parse tree construction. Thus, DCGs must be augmented with explicit rules to build parse trees as parsing progresses. Nonetheless, DCGs are now used widely in the construction of front ends for Prolog programs and, as noted in chapter 2, Prolog has been proposed and used for implementing formal language definitions.

4.2.6. Lazy ML conctypes

In the late 1980's, Aasa et al [1] extended Lazy ML with a new data type for manipulating the concrete syntax of data objects. They view current inductive data type representations as the prefix forms of parse trees for the underlying values. Instead, they propose the direct use of linear surface representations of such values which are automatically coerced to the corresponding parse trees. Their `conctype`s specify the surface syntax of data type values and are defined using BNF-like rules. A `conctype` consists of alternative options, where each option is a stropped sequence of characters, for terminal symbols, and `conctype` names in `<...>`s for non-terminals. `conctype` values are usually stropped character sequences. They may also contain quoted LML expressions which must evaluate to `conctype` values. Functions are then defined with cases for each alternative option. In `conctype` patterns, non-terminals are replaced by quoted bound variables. When such a function is called with a `conctype` value as argument, the argument is parsed using the corresponding `conctype` definition, and then pattern matched to determine which case applies and to select argument components. This potentially powerful tool is limited by the absence of any provision for the direct construction of `conctype` values from strings.

4.3. Syntax in Navel

The development of syntax processing facilities for Navel was influenced by the decision that DS implementations in Navel would be based directly on concrete syntax. Thus, Navel constructs were needed to define concrete syntax, to represent the surface forms of defined languages, to use concrete syntax definitions to parse surface representations to create appropriate intermediate representations of syntactic structures, and to associate semantic functions with syntactic constructs through the manipulation of intermediate representations. These considerations are closely inter-related. Here, an objective was to use or extend existing Navel constructs as far as possible and to try and minimise the inevitable introduction of new constructs.

4.3.1. Syntax rules

Syntax definitions in Navel are based on BNF. However, a key concept of functional languages is that of functions as values without any necessary association with defining names. Consequently, rule right hand sides are uncoupled from any necessary associations with defining non-terminal names and rule right hand sides become values in their own rights. Consider the arbitrary BNF rule:

```
<non-terminal> ::= <option1> | <option2> | <option3> | ...
```

where each `<option>` is a sequence of terminal and non-terminal symbols. When the non-terminal `<non-terminal>` is invoked to parse a surface representation, the associated `<option>`s are invoked in turn until one succeeds. Here, the association of the left and right hand sides is contingent. The left hand side non-terminal `<non-terminal>` is effectively the current name of the right hand side `<option>`s. It is, of course significant within those `<option>`s where its presence results in the invocation of those `<option>`s once more. However, this is directly analogous to a function definition:

```
def <name> = <function>
```

where each reference to `<name>` in an expression, including in `<function>`, results in the effective replacement of `<name>` with `<function>`. Hence, in Navel, rule right hand sides, subsequently referred to as rules, are treated as special sorts of function values which when applied to surface representations return intermediate representations.

Rules are identified by stropping their structures in `{...}`s so a rule definition has the form:

```
def <name> = { <rule details> }
```

Strings are used for surface representations and, hence, for terminal symbols. As implied above, Navel identifiers are used for non-terminal symbols. Thus, at simplest, a rule option is a sequence of strings and names, and a rule is a sequence of options separated by `|`.

For example, the BNF for binary numbers:

```
<digit> ::= 0 | 1
<binary> ::= <digit> <binary> | <digit>
```

becomes:

```
def digit = {"0" | "1"};
def binary = {digit binary | digit};
```

Note that there are no separate lexical definitions. Lexemes are implicit in rules.

Nested rules are allowed within rules to enable sub-options to simplify rules. For example, the BNF for binary expressions:

```
<expression> ::= <term> + <expression> | <term> - <expression> | <term>
<term> ::= <factor> * <term> | <factor> / <term> | <factor>
<factor> ::= - <base> | <base>
<base> ::= <binary> | ( <expression> )
```

becomes:

```
def expression = {term {"+" | "-"} expression | term};
def term = {factor {"*" | "/"} term | factor};
def factor = {"-" base | base};
def base = {binary | "(" expression ")"};
```

The empty rule:

```
{}
```

always succeeds when applied to a string. This may be used to recognise 0 or one occurrences of a rule:

```
{<non-terminal> | {}}
```

and hence to factor rules. For example, to simplify the rule associated with `expression` above:

```
def expression = {term {{"+" | "-"} expression | {}}};
```

Here, if:

```
{"+" | "-"} expression
```

fails after recognition of a `term`, then `{}` will always succeed resulting in the recognition of a single `term`.

Arbitrary Navel expressions may appear in rules, enclosed in brackets `(...)` to indicate that they must be evaluated. Such expressions should return strings or rules. Note that bracketing a name results in its effective replacement with its associated value: the significance of this will become clearer when intermediate representations are considered.

A number of standard rules are provided to recognise common lexical classes:

```
word == 1 or more letters
number == 1 or more digits
identifier == 1 or more letters and digits starting with a letter
character == 1 character
```

The rule:

```
fail
```

always causes the enclosing rule to fail. The use of this rule enables the scope of other rules to be restricted.

The predicate `isrule` is introduced to test whether or not an arbitrary value is a rule.

In keeping with the principle of data type completeness [27], rules, like functions, are full values and may be abstracted over. The significance of this is discussed in chapter 8 where context sensitive parsing is considered.

4.3.2. Parse tree representations

The intermediate representation for parse trees is based on lists. In principle, rules could be applied to strings to create nested lists corresponding to the parse structure, with terminal strings at the leaves. Such parse trees could then be inspected explicitly to test for the presence of known terminals in particular branch positions, to identify the appropriate semantic actions. However, for rules whose options lack terminals the corresponding top level nodes will also lack terminals at the top level and so there would be no means of identifying uniquely such nodes. Thus, the representation should reflect how the parse tree was formed, that is which non-terminals were responsible for which parse tree nodes. In principle, this could be achieved within core Navel by pairing a node with the string corresponding to the non-terminal identifier. However, this introduces identifier/string coercions into the language.

To solve this problem, a new construct is introduced to represent non-terminal tagged nodes. The Navel field is an association of an identifier and a value bracketed by `[...]`. The value is selected using the infix operator ˆ and the field identifier. For example:

```
def crop = [field "barley"];
cropˆfield ==> "barley"
```

Field value selection is overloaded to enable selection from lists of fields. For example:

```
def crops = [field "barley"]:[orchard "apples"]:[field "oats"]:;
cropsˆorchard ==> "apples"
```

Indexed list selection is combined with field selection to enable the selection of field values where several fields in a list share the same identifier:

```
def crops = [field "barley"]:[orchard "apples"]:[field "oats"]:;
cropsˆfield@2 ==> "oats"
```

The predicate `isfield` is used to test whether or not an arbitrary value is a field.

To return to rules: applying a rule to a string builds a list representation of the parse tree. The nodes resulting from components of a successful rule option are joined together into a list. Where an option contains an identifier acting as a non-terminal, the resulting node forms the value of a field tagged with that identifier. Finally, the tree is paired with the remains of the string and returned.

For example:

```
digit "10+11" ==> "1":"0+11"
```

Here, `digit` is replaced by its associated rule and the:

```
"1"
```

option succeeds, leaving:

```
"0+11"
```

unmatched.

Consider also:

```
{digit} "10+11" ==> [digit "1"]:"0+11"
```

Here as before the

```
"1"
```

option for the rule associated with `digit` succeeds but is tagged with `digit` in a field because an explicit rule was used.

More generally, consider:

```
{binary} "10+11" ==> [binary [digit "1":[binary [digit "0"]]]]:"+11"
```

Here, `1` is recognised as a `digit` and a further `binary` is sought. `0` is then recognised as a `digit` but a subsequent `binary` is not found. On backtracking, `0` is recognised as a `digit` which is also a `binary.` Thus the whole `binary` is:

```
[binary [digit "1":[binary [digit "0"]]]]
```

and the rest of the string is:

```
"+11"
```

Note that spaces and newlines in strings are ignored unless specified explicitly in rules.

4.3.3. Associating rules with semantic actions

For associating semantic actions with syntactic structures, in principle pattern matching with cases should be used. However, this seemed against the spirit of Navel's SASL roots and would have caused implementation complications, in particular to do with combining pattern matching and curried functions. Instead, a rule matching construct is introduced:

```
rule <expression> of
<case1> -> <expression1>,
<case2> -> <expression2>,
 ...
<expressionN>
```

Each `<caseI>` must be a flat rule, that is it must not contain options and must be composed only of terminal and non-terminal symbols. For matching, the intention is that `<expression>` is a parse tree which could have been constructed by an application of one of the `<caseI>`, that is they must have corresponding top level structures. Fields in parse trees must be in the same position and have the same tag as non-terminals in rules. Strings in parse trees must be in the same position as and the same as terminals in rules.

To evaluate a rule matching expression, if `<expression>` matches `<caseI>` then `<expressionI>` is evaluated. If no matches succeed then the default `<expressionN>` is evaluated.

Note that this is a matching but not a selection process. Tree components must be chosen explicitly by field selection.

For example, consider defining the meaning of binary integers: A `<digit>` is replaced by the corresponding integer value:

```
def mdigit d =
 rule d of
 {"0"} -> 0,
 {"1"} -> 1,
 "bad digit":d;
```

The tree for a digit without the field tag is either:

```
"0"
```

or:

```
"1"
```

When `mdigit` is applied to a digit tree, if the tree is:

```
"0"
```

then `0` is returned and if it is:

```
"1"
```

then `1` is returned.

The default case returns a list error message. This point should never be reached. In chapter 5, an exit mechanism is introduced which is used subsequently as the default case.

For the meaning of a `<binary>` given a value so far, if the `<binary>` is a `<digit>` then the value so far is doubled and has the value of the `<digit>` added to it. Otherwise, the value of the `<binary>` part of the `<binary>` is found with the value so far doubled and added to the value of the `<digit>`:

```
def mbinary b v =
 rule b of
 {digit} -> 2*v+(mdigit b^digit),
 {digit binary} -> mbinary b^binary 2*v+(mdigit b^digit),
 "bad binary":b;
```

The tree for a binary without the enclosing field is either:

```
[digit <digit tree>]
```

for a single `digit` or:

```
[digit <digit tree>]:[binary <binary tree>]
```

for a `digit` followed by a `binary.` In the first case, the `<digit tree>` is extracted using the field selector `^digit` and passed to `mdigit` to find its meaning. In the second case, the `<binary tree>` is also extracted using the field selector `^binary` and passed recursively to `mbinary` to find its meaning. For example:

```
mbinary [digit "1"]:[binary [digit "0"]:[binary [digit "1"]]] 0 ==
mbinary [digit "0"]:[binary [digit "1"]] 2*0+(mdigit "1") ==
mbinary [digit "0"]:[binary [digit "1"]] 1 ==
mbinary [digit "1"] 2*1+(mdigit "0") ==
mbinary [digit "1"] 2 ==
2*2+(mdigit "1") == 2*2+1 == 5
```

To evaluate a string represented binary number, the rule for binary numbers is applied to the string returning a tree and the rest of the string. If the tree is empty or the rest of the string is not empty then the parse failed. Otherwise, the tree is evaluated:

```
def eval s =
 let tree:rest = binary s
 in
  if tree=() | rest<>()
  then "syntax error":rest
  else mbinary tree 0;
```

For example, for:

```
eval "111"
```

the tree is:

```
[digit "1"]:[binary [digit "1"]:[binary [digit "1"]]
```

and the rest of the string is:

```
()
```

The tree is non-empty and the rest of the string is empty. Thus, evaluation continues with:

```
mbinary [digit "1"]:[binary [digit "1"]:[binary [digit "1"]] 0 ==
mbinary [digit "1"]:[binary [digit "1"]] 1 ==
mbinary [digit "1"] 3 ==
7
```

Consider:

```
eval "121"
```

The tree is:

```
[digit "1"]
```

and the rest of the string is:

```
"21"
```

The rest of the string is not empty so the message:

```
"syntax error":"21"
```

is returned.

4.3.4. Example - rudimentary imperative language

Consider the example imperative language from chapter 2:

```
Concrete syntax

<statements> ::= <statement> ; <statements> | <statement>
<statement> ::= <assign> | <input> | <output>
<assign> ::= <identifier> := <expression>
<input> ::= read <identifier>
<output> ::= write <expression>
<expression> ::= <base> + <expression> | <base>
<base> ::= <identifier> | <number>
<identifier> ::= <alpha> <identifier> | <alpha>
<alpha> ::= a | b | ... | z
<number> ::= <digit> <number> | <digit>
<digit> ::= 0 | 1

Syntactic domains

s ∈ <statements>
e ∈ <expression>
n ∈ <number>
i ∈ <identifier>

Abstract syntax

s -> s s | READ i | WRITE e | i := e
e -> i | n | e + e
n -> 0 | 1 | n 0 | n 1

Semantic domains

N == non-negative integers
I = N* == inputs
O = N* == outputs
Env = <identifier> -> N == environments

mi: <number> -> N
mi [0] = 0
mi [1] = 1
mi [n 0] = 2*(mi [n])
mi [n 1] = 2*(mi [n])+1

me: <expression> -> Env -> N
```

```
me [n] e = mi [n]
me [i] e = e [i]
me [e1+e2] e = (me [e1] e)+(me [e2] e)

ms: <statements> -> Env * I * O -> Env * I * O
ms [s1 s2] (e,i,o) = ms [s2] (ms [s1] (e,i,o))
ms [i:=e] (e,i,o) = ((new e [i] (me [e] e)),i,o)
ms [READ i] (e,i,o) = ((new e i (hd i)),(tl i),o)
ms [WRITE e] (e,i,o) = (e,i,(me [e] e)::o)

new old lv rv l = if lv=l
                  then rv
                  else old l
```

The syntax can be written in Navel as:

```
def statements = {statement ";" statements | statement};
def statement = {assign | input | output};
def assign = {word ":=" expression};
def input = {"read" word};
def output = {"write" expression};
def expression = {base "+" expression | base};
def base = {word | binary};
```

The semantics is written in Navel as follows. For an `<expression>` there are cases for numbers, names and additions:

```
def me e s =
 rule e of
 {binary} -> mbinary e^binary 0,
 {word} -> s e^word,
 {base} -> me e^base s,
 {base "+" expression} -> (me e^base s)+(me e^expression s),
 "bad expression";
```

For a `<statement>`, there are cases for statement sequences, assignment, input and output:

```
def ms st s:i:o =
 rule st of
 {statement ";" statements} -> ms st^statements (ms st^statement s:i:o),
 {statement} -> ms st^statement s:i:o,
 {assign} -> ms st^assign s:i:o,
 {word ":=" expression} -> (new s st^word (me st^expression s)):i:o,
 {input} -> ms st^input s:i:o,
 {"read" word} -> (new s st^word (hd i)):(tl i):o,
 {output} -> ms st^output s:i:o,
 {"write" expression} ->
  let ov = me st^expression
  in s:i:(ov:o),
 "bad statement";

def new old lv rv l =
 if l=lv
 then rv
 else old l;
```

The syntax and semantics are sown together with a function which parses a string, checks for a successful parse, calls the semantic function for statements and returns the final output:

```
def run text input =
 let tree:rest = statements text
 in
```

```
if tree=() | rest<>()
then "syntax error":rest
else
 let s:i:o = ms tree ():input:
 in o;
```

4.3.5. Semantic redundancy through concrete syntax

In the above example, note the redundant semantic actions in `me` for the singleton production `{base}` and in `ms` for the singleton productions `{assign}`, `{input}`, `{output}`, and `{statement}`.

Recall that bracketing an expression in a rule results in the evaluation of the expression. In particular, a bracketed name is replaced with the associated value. Hence, if sites of singleton productions are bracketed then no field is formed for them. For example, the above rules may be rewritten as:

```
def statements = {statement ";" statements | (statement)};
def statement = {(assign) | (input) | (output)};
def assign = {word ":=" expression};
def input = {"read" word};
def output = {"write" expression};
def expression = {base "+" expression | (base)};
def base = {word | binary};
```

Before, the tree for say:

```
write 1+1
```

was:

```
[statement
 [output
  "write":
  [expression
   [base
    [binary [digit "1"]]]:
   "+":
   [expression
    [base
     [binary [digit "1"]]]]]]]
```

Now it is:

```
"write":
[expression
 [base
  [binary [digit "1"]]]:
 "+":
 [expression
  [binary [digit "1"]]]]
```

In `me` the case for `{base}` may be dropped. In `ms` the cases for `{assign}`, `{input}`, `{output}` and `{statement}` may be dropped.

Note that bracketing singletons does not combine well with factoring. For example, consider the simplification of:

```
def statements = {statement ";" statements | statement};
```

to:

```
def statements = {statement {";" statements | {}}};
```

Consider now attempting to avoid a singleton node for `statement` through bracketing:

```
def statements = {(statement) {";" statements | {}}};
```

Alas, now a field for `statement` will never be constructed and so the semantic function must have cases for all possible substitutes for `statement` before `;`. Hence, one is faced with the choice between full rules with minimised parse trees but backtracking parsing or factored rules which will not backtrack but have unnecessary singleton nodes.

4.4. Formal definition of syntax in Navel

The following sections discuss the formal definition of syntax in Navel. The DS from chapter 3 is augmented and extended with the new constructs for processing syntax. In summary, rules are treated similarly to functions. In particular, rule values are represented by closures.

4.4.1. Concrete syntax extensions

The Navel concrete syntax is extended to enable rules:

```
<aexp> ::= ... | <non-terminal> | <field>

<rule> ::= '{' <rulebody> '}' | '{''}' | <sysselector> | fail

<rulebody> ::= <ruleexp> ['|' <rulebody>]

<ruleexp> ::= <ruleterm> [<ruleexp>]

<ruleterm> ::= <name> | <string> | (<expression>) | <rule>

<sysselector> ::= number | word | identifier | character
```

fields:

```
<field> ::= '[' {<name> <sysselector>} <expression> ']'
```

rule and field predicates:

```
<log base> ::= ... | {isfield isrule} <aexp>

<prefix operator> ::= ... | isfield | isrule
```

field selection:

```
<factor>::= ... | <base> [<indexes>]

<indexes> ::= <index> [<indexes>]

<index> ::= ^ {<name> <sysselector>} [<select>] | <select>

<select> ::= @<number> | @<name> | @(<expression>)

<infix operator> ::= ... | ^
```

and rule matching:

```
<expression> ::= ... | <rule match>

<rule match> ::= rule <expression> of <rulecases>
<rulecases> ::= <rulecase> -> <expression> , <rulecases> |
                <expression>
```

```
<rulecase> ::= <name> | <string> | <sysselector> |
               <name> <rulecase> | <string> <rulecase> |
               <sysselector> <rulecase>
```

4.4.2. Abstract syntax extensions

The abstract syntax is extended to reflect the above concrete syntax extensions:

```
 ...
r ∈ <rulebody>
s ∈ <sysselector>
 ...
e -> ... | { r } | s | {} | fail | [n e] | rule e of ca

r ->  r ’|’ r | ( e ) | e

s ->  number | word | identifier | character

iop -> ... | ˆ
```

4.4.3. Semantic domains

New semantic domains are introduced for rules and fields.

A rule, like a function, is a closure consisting of a pairing of a state and a rule meaning. A rule meaning is a λ function from a list, for the string to be parsed, and a state to a pair consisting of a value, for the tree, and a list, for the rest of the parsed string. When a rule is applied to a string, the meaning is invoked to parse the string in the state from the closure to return the tree and rest of string. The rule domain includes appropriate constructor, selector, equality, inequality and predicate operators. The value `ofail` indicates option failure, enabling another option to be tried, whereas `rfail` indicates rule failure without further options:

```
R == rules == S * (L -> S -> V * L)
  - ofail == option failure
  - rfail == rule failure
 with makerule : S * (L -> S -> V * L) -> R == constructor
      rulestate : R -> S == select state
      rulemean : R -> (L -> S -> V * L) == select meaning
      = : R * R -> B == equality
      <>: R * R -> B == inequality
      isrule : V -> B == predicate
```

A field is a pair of a name and a value, with operators for construction, selection, equality, inequality and predication:

```
Fi == fields == <name> * V
 with makefield : <name> * V -> Fi == constructor
      fieldtag : Fi -> <name> == select tag
      fieldval : Fi -> V == select value
      = : Fi * Fi -> B == equality
      <> : Fi * Fi -> B == inequality
      isfield : V -> B == predicate
```

The domain of values is extended with rules and fields:

```
V == values == ... + R + Fi
```

4.4.4. Rule semantics

For a rule, a closure pair consisting of the current state and rule meaning is returned:

```
  ...
me [{ r }] s = makerule s (parse [{ r }])
```

Details of `parse` are given below.

For system rules, the meaning involves appropriate matching functions:

```
me [character] s = makerule s λ str.λ s.(charmatch str)

me [word] s = makerule s λ str.λ s.(wordmatch str)

me [identifier] s = makerule s λ str.λ s.(idmatch str)

me [number] s = makerule s λ str.λ s.(numbmatch str)
```

Details of `charmatch`, `wordmatch`, `idmatch` and `numbmatch` are given below.

The empty rule always succeeds:

```
me [{}] s = makerule s λ str.λ s.(():str)
```

The failure rule always fails:

```
me [fail] s = makerule s λ str.λ s.(rfail:str))
```

The rules for global and local definitions must be extended to cater for rules:

```
md [def n = e] s news = let e1 = me [e] s
                         in
                          if iserror e1
                          then s
                          else
                           if isfunc e1
                           then new s [n] (makefun news (funmean e1))
                           else
                            if isrule e1
                            then new s [n]
                                  (makerule news (rulemean e1))
                            else new s [n] e1

letbind [n] a s news = if isfunc a
                       then new s [n] (makefun news (funmean a))
                       else
                        if isrule a
                        then new s [n] (makerule news (rulemean a))
                        else new s [n] a
```

4.4.5. Field semantics

For a field, a field value is constructed:

```
me [[n e]] s = let fv = me [e] s
               in
                if iserror fv
                then error
                else makefield [n] fv
```

For field selection, if the left operand is a field and the tag corresponds to the selector tag then the value is returned. If the left operand is a list then the value of the first field with that tag is returned:

```
me [e ˆ n] s = let fv = me [e] s
                 in
                  if iserror fv
                  then error
                  else
                   if isfield fv
                   then
                    if (fieldtag fv)=[n]
                    then fieldval fv
                    else error
                   else
                    if not islist fv
                    then error
                    else getfield fv [n] 1
```

For indexed field selection, if the left operand is a list and the right operand is an integer `index` then the value of the `index`th field with the tag is returned:

```
me [e1 ˆ n @ e2] s = let fv = me [e1] s
                     in
                      if not islist fv
                      then error
                      else
                       let index = me [e2] s
                       in
                        if not isint index
                        then error
                        else getfield fv [n] index
```

To select an indexed field from a list, the list is examined element by element and the index count decremented until the required field is reached. the field value is then returned:

```
getfield l [n] i = if not islist l
                   then error
                   else
                    let h = msusp (hd l)
                    in
                     if iserror h
                     then error
                     else
                      if isfield h
                      then
                       if [n]=fieldtag h
                       then
                        if i=1
                        then fieldval h
                        else getfield (msusp (tl l)) [n] i-1
                       else getfield (msusp (tl l)) [n] i
                      else getfield (msusp (tl l)) [n] i
```

4.4.6. Field and rule predicates and comparison

For field and rule predicates, the appropriate predicate is called:

```
 ...
applypop [isfield] v = isfield v
applypop [isrule] v = isrule v
```

For field and rule equality and inequality operations, the appropriate operators are called after type checking:

```
mcomp c x y = if ... or
                 isfield x and isfield y or
                 isrule x and isrule y
              then c x y
              else ...
```

4.4.7. Rule application semantics

To apply a rule to a string, the rule's meaning is applied to the string in the rule's state:

```
me [e1 e2] -> let f = me [e1] s
              in
               if not isfunc f and not isrule f
               then error
               else
                let a = me [e2] s
                in
                 if iserror a
                 then error
                 else
                  if isfunc f
                  then (funmean f) a (funstate f)
                  else
                   if not isstring a
                   then error
                   else
                    let t:r = (rulemean f) a (rulestate f)
                    in
                     if t=ofail or t=rfail
                     then ():a
                     else t:r
```

Thus, the rule for list selection must be extended to allow lazy rule applications:

```
msusp s = if issusp s
          then
           let f = me (suspfun s) nullstate
           in
            if not isfunc f and not isrule f
            then error
            else
             let a = me (susparg s) nullstate
             in
              if iserror a
              then error
              else
               if isfunc f
               then (funmean f) a (funstate f)
               else
                if not isstring a
                then error
                else (rulemean f) a (rulestate f)
          else s
```

In principle, full context-free parse semantics for rule application should be defined here. In practice, this is somewhat fiddley especially as parsing is combined with tree construction. Context free parsing is fully backtracking and backtracking is potentially arbitrary. Thus, parse trees may be altered at arbitrary points due to backtracking as parsing progresses. The expression of this is relatively straightforward if the use of destructive assignment operators is permitted as trees may be modified in situ. However, in a declarative semantic framework, this involves maintaining an explicit backtracking structure which is traversed to extract the tree once parsing has finished. Alternatively, continuation passing and lazy constructs might be employed.

Navel rules are actually implemented as AND-committed. That is, for a rule option each component is matched in turn. If a component match fails then the whole option fails and another is tried: there is no backtracking within options. Formally, Navel rules are related to LL(k) grammars. However, they are not truly LL(k) because they may contain arbitrary expressions whose values are determined at run time. Hence, static checks for LL(k) cannot be applied. The semantics below reflects these considerations.

Rule failure is used in place of error handling in the following semantic equations to simplify presentation.

A rule is applied to a string in a state to return a tree and the rest of the string:

```
parse : <rule body> -> L -> S -> V * L
```

For rule options, the options are tried in turn:

```
parse [r1 | r2] str s = optmatch [r1 | r2] str s

optmatch : <expression> -> L -> S -> V * L
optmatch [r1 | r2] str s = let t:r = parse [r1] str s
                           in
                            if t=ofail
                            then parse [r2] str s
                            else t:r
```

Note that if `fail` is invoked at the top level in the first option then `rfail` will be returned and so the second option will not be tried. Thus, the whole rule containing these options fails.

For a sequence of rule components, each is tried sequentially. If one fails then the whole sequence fails. Otherwise, the results from each component are joined into a single list. `{}` may return an empty tree which must be discarded. Note that list representations of trees are not null terminated:

```
parse [r1 r2] str s = seqmatch [r1 r2] str s

seqmatch : <expression> -> L -> S -> V * L
seqmatch [r1 r2] str s = let t1:r1 = parse [r1] str s
                          in
                           if t1=ofail or t1=rfail
                           then t1:str
                           else
                            if r1=()
                            then ofail:str
                            else
                             let t2:r2 = parse [r2] r1 s
                             in
                              if t2=ofail or t2=rfail
                              then t2:str
                              else (seqjoin t1 t2):rest2;

seqjoin : L -> L -> L
seqjoin t1 t2 = if t1=()
                then t2
                else
                 if not islist t1
                 then
                  if t2=()
                  then t1
                  else t1:t2
                 else
                  let t = seqjoin (tl t1) t2
                  in
                   if (hd t1)=()
                   then t
                   else
```

```
                          if t=()
                          then hd t1
                          else (hd t1):t
```

For a name non-terminal, its associated rule is tried:

```
parse [n] str s = let nv = s [n]
                  in
                   if iserror nv
                   then ofail:str
                   else nontermmatch [n] nv str
```

If the match succeeds then an appropriate field is returned:

```
nontermmatch : <name> -> <expression> -> L -> V * L
nontermmatch [n] r str = let t:rest = expmatch r str
                         in
                          if t=ofail or t=rfail
                          then t:str
                          else (makefield [n] t):rest
```

The auxiliary function `expmatch` distinguishes between rule and string application:

```
expmatch : <expression> -> L
expmatch r str = if isrule r
                 then (rulemean r) str (rulestate r)
                 else
                  if isstring r
                  then strmatch r str
                  else ofail:str
```

The empty rule always succeeds, returning an empty tree:

```
parse [{}] str s = ():s
```

The failure rule always fails:

```
parse [fail] str s = rfail:str
```

Some generic matching functions are introduced for system rules. To match a `first` followed by an optional `next`, ignoring leading spaces and newlines:

```
match : (C -> B) -> (C -> B) -> L -> V * L
match first next str = if str=()
                       then ofail:str
                       else
                        let h:t = (msusp (hd str)):(msusp (tl str))
                        in
                         if first h
                         then
                          let tree:rest = rest next t
                          in
                           if tree=ofail
                           then (h:()):t
                           else (h:tree):rest
                         else
                          if h=’ ’ or h=’\n’
                          then match first next t
                          else ofail:(h:t)
```

To recognise one or more `next`s:

```
rest : (C -> B) -> L -> V * L
rest next str = if str=()
                then ofail:()
                else
                 let h:t = (msusp (hd str)):(msusp (tl str))
                 in
                  if next h
                  then
                   let tree:rest = rest next t
                   in
                    if tree=ofail
                    then (h:()):t
                    else (h:tree):rest
                  else ofail:(h:t)
```

To apply a system rule `sysrule` and construct a field tagged with `tag:`

```
sysmatch : (L -> V * R) -> <name> -> L -> V * V
sysmatch sysrule tag str = if str=()
                           then ofail:()
                           else
                            let n:r = sysrule str
                            in
                             if n=ofail
                             then ofail:str
                             else (makefield tag n):r
```

For system rules, the corresponding match is attempted and a field constructed. For `character` a single character is found:

```
parse [character] str s = sysmatch charmatch [character] str

charmatch : L -> V * L
charmatch str = if str=()
                then ofail:()
                else (musp (hd str)):(msusp (tl str))
```

For `number` a digit sequence is found:

```
parse [number] str s = sysmatch (match numbchar numbchar) [number] str

numbchar : C -> B
numbchar c = c>='0' and c<='9';
```

For `word` a sequence of letters is found:

```
parse [word] str s = sysmatch (match wordchar wordchar) [word] str

wordchar : C -> B
wordchar c = c>='A' and c<='Z' or c>='a' and c<='z';
```

For `identifier` a sequence of letters and digits starting with a letter is found:

```
parse [identifier] str s =
 sysmatch (match wordchar idchar) [identifier] str

idchar : C -> B
idchar c = (wordchar c) or (numbchar c);
```

For a bracketed expression, the expression is evaluated and the resultant rule is tried:

```
parse [( e )] str s = let r = mexp [e] s
                      in
                       if iserror r
                       then ofail:s
                       else expmatch r str s
```

For a nested rule, that rule is invoked. If the whole rule has failed due to an invocation of `fail` then the option failure value is returned to indicate that another option may be tried at a higher level:

```
parse [{ r }] str s =  let t:r = parse [r] str s
                       in
                        if t=rfail
                        then ofail:r
                        else t:r
```

For a string terminal, it is matched against the start of the argument string, after skipping leading spaces and newlines:

```
parse [e] str s = let sv = me [e] s
                  in
                   if iserror sv
                   then ofail:str
                   else
                    if isstring sv
                    then strmatch sv str
                    else ofail:str

strmatch : L -> L -> V * L
strmatch s str = if s=() or str=()
                 then ofail:str
                 else
                  let h:t = (msusp (hd str)):(msusp (tl r))
                  in
                   if hd s=h
                   then
                    let tree:rest = reststrmatch (tl s) t
                    in
                     if tree=ofail
                     then ofail:(h:t)
                     else s:rest
                   else
                    if h=’ ’ or h=’\n’
                    then strmatch s t
                    else ofail:(h:t)

restsrtmatch : L -> L -> V * L
reststrmatch s str = if s=()
                     then ():str
                     else
                      if str=()
                      then ofail:()
                      else
                       let h:t = (msusp (hd str)):(msusp (tl str))
                       in
                        if hd s=h
                        then
                         let tree:rest = reststrmatch (tl s) t
                         in
                          if tree=ofail
                          then ofail:(h:t)
                          else s:r
```

```
                              else ofail:(h:t)
```

4.4.8. Rule/tree matching semantics

To match a rule against a parse tree representation, the structure of each rule case is matched against that of the tree:

```
me [rule e of ca] s = let t = me [e] s
                      in
                       if not islist t
                       then error
                       else mrulecase t [ca] s

mrulecase : L -> <cases> -> S -> V
mrulecase t [e1 -> e2 , ca] s = let rm = rulematch t [e1]
                                in
                                 if iserror rm
                                 then error
                                 else
                                  if rm
                                  then me [e2] s
                                  else mrulecase t [ca] s
mrulecase rv [e] s = mexp [e] s
```

For a sequence of rules, each is matched in turn and all must succeed:

```
rulematch : L -> R -> B
rulematch t [r1 r2] = if islist t and not isstring t
                      then
                       let rm1 = rulematch (hd t) [r1]
                       in
                        if iserror rm1
                        then error
                        else
                         if rm1
                         then
                          let rm2 = rulematch (tl t) [r2]
                          in
                           if iserror rm2
                           then error
                           else rm2
                         else false
                       else false
```

Non-terminals and system rules match a tag for the corresponding identifier in a tree consisting of a single field:

```
rulematch t [n] = if isfield t
                  then [n]=(fieldtag t)
                  else false
rulematch t [character] = if isfield t
                          then [character]=(fieldtag t)
                          else false
rulematch t [identifier] = if isfield t
                           then [identifier]=(fieldtag t)
                           else false
rulematch t [number] = if isfield t
                       then [number]=(fieldtag t)
                       else false
rulematch t [word] = if isfield t
                     then [word]=(fieldtag t)
                     else false
```

A string matches a tree consisting of the same string:

```
rulematch t [e] = let sv = me [e] nullstate
                  in
                   if iserror sv
                   then error
                   else
                    if isstring t and isstring sv
                    then t=sv
                    else false
```

Chapter 5

Navel implementation, environment and extensions

5.1. Introduction

This chapter discusses the implementation of Navel. It also presents the interactive environment within which Navel is implemented. Finally, a number of extensions to Navel, mainly imperative, which ease language prototyping in the interactive environment are considered.

5.2. Language implementation

In a sense, all language implementations are based on interpretation. That is, an implementation needs to enable the transformation of a program into some final form. Transformation is guided by rules, based ideally on the formal definition, which may be conceived of as the instructions for some underlying machine. Compilation itself may be seen as a form of interpretation which involves transforming the surface representation of a program into a form with the same meaning in a different notation, usually for subsequent interpretation.

For maximum efficiency, a program might be compiled into the machine code of some specific digital computer for hardware interpretation. While this enables extremely efficient implementations it bears the cost of compiler construction, which in turn depends on the degree of correspondence between the source and target language. This latter consideration also effects the speed of compilation. Compilation to machine code leads to a loss of portability as the code for one digital computer will not usually run on another. This approach is used widely for final implementations of programming languages.

A common starting point for language implementation is to compile programs to some intermediate form which is then interpreted by a program acting as an abstract machine for that form. The intermediate form consists effectively of instructions for the abstract machine, and is chosen as an optimal compromise between ease of its own implementation and its ability to express the concepts of the source language. Language portability is enabled by reimplementing the abstract machine on different hardware architectures. This is often straightforward if the abstract machine is itself written in a machine independent language. While this approach eases initial implementation it also leads to a loss of efficiency as a program being interpreted in intermediate form by an abstract machine itself running on a digital computer is usually considerably slower than the equivalent machine code for the program. However, the intermediate form may be used subsequently as a basis for compilation into the machine code of specific computers.

Explicit abstract machines have been used in implementations of a wide variety of languages, for example OCODE and INTCODE for BCPL [127], PCODE for UCSD Pascal [152] and the Warren Abstract Machine (WAM) [154] for Prolog. They are also widely used as the basis of functional language implementations as discussed below. However, many language implementations based on interpreters lack an explicitly enunciated abstract machine although an abstract machine may be inferred from the implementation.

5.3. Functional language implementation models

The central problem in implementing functional languages lies in the association of functions' bound variables and arguments. The two main implementation models for functional languages are based on SECD machines, where such associations are held explicitly, and on graph reduction, where such associations are implicit in the program representation during evaluation. The brief account below is after Field and Harrison [45].

Landin's [85] SECD machine is essentially a multi-stack von Neumann architecture for λ expression evaluation. It consists of a stack, to hold values which are to be used later in evaluation, an environment, which holds identifier/value associations, a control, which is the program fragment currently being evaluated, and a dump, which holds the stack, environment and control on function entry for restoration on function exit. Evaluation starts with an empty stack, environment and dump, and the control consisting of the initial λ expression in some suitable representation. The top of the control is repeatedly inspected. A constant is removed and pushed onto the stack. A variable is removed, the value found from the environment and pushed onto the stack. A function is removed, turned into a closure consisting of the bound variable, body and environment, and pushed onto the stack. For an application, the function and argument expressions are extracted and replaced on the top of control following an application symbol. For an application symbol, a closure and argument are removed from the stack, the stack, environment and control are pushed onto the dump, the environment from the closure is extended with an association between the bound variable from the closure and the argument to form a new environment, and the function body from the closure forms the new control, with an empty stack. For an empty control, the stack, environment and control are restored from the top of the dump. For an empty control and dump, the final result is on the top of the stack.

Graph reduction is based on a graph representation of a program which is traversed and manipulated depending on the operations at the nodes. Such manipulations consist typically of overwriting the current node with a new graph formed from some or all of the sub-nodes of the current node. For a function application, the graph for the body is copied and sub-nodes for the bound variable in the body copy are replaced with the argument graph. Note that all occurrences of a bound variable may be replaced with a reference to a shared single copy of an argument. However, function bodies cannot be shared as free variables may have different instantiations at different points in the program. Function body copying can result in unacceptable overheads. This may be avoided by compilation to combinators, simple variable free atomic entities, but this leads to a very fine granularity of operation. Alternatively, prior to graph reduction, lambda lifting may be used to treat all free variables as additional arguments to functions or super combinators may be formed by abstracting over expressions containing free variables. The disadvantage here is the additional costs of such compilation stages.

Peyton Jones [77] comments that SECD machine implementations are most suitable for strict languages and that graph reduction is most suitable for lazy languages. Indeed, the Functional Programming Machine(FPM) implementation of Hope [45] and the Functional Abstract Machine(FAM) implementation of ML [26] are based on SECD machines, whereas graph reduction is used in implementations of Miranda and Haskell, and as the basis of the abstract G-machine for Lazy ML [77].

5.4. Navel implementation design considerations

Navel has been implemented within an interactive environment to enable incremental development and testing of interpreter prototypes from formal language definitions. In principle, incremental compilation to machine code would have offered maximal run time speed. However, there are several disadvantages to this approach. First of all, the development of an incremental compiler is a substantial undertaking in its own right. While implementing Navel was crucial, its design and use were the main foci of interest. Hence an objective was to implement quickly a system as a basis for experimentation with Navel use. Secondly, compilation to machine code loses the ability to reconstruct directly the original program, an important facility for the provision of tracing facilities. This could be circumvented by holding program text as well as code and linking execution tracing to text tracing. However, this complicates the implementation, increases its size and loses both compilation and run time speed. Thirdly, in an interactive environment response time is psychologically important. For incremental program development the intention is to work with relatively small program units which may be changed and retried frequently. For such small units, compilation time is a significant portion of overall test time and the pause between unit entry and execution may become noticeable.

As suggested above, an alternative would have been to design an explicit lowish level abstract machine, compile Navel to abstract machine code and then execute it on the abstract machine implemented as an interpreter. However, writing a compiler for abstract machine code is almost as much work as writing a concrete machine code compiler, there is the additional time to implement the abstract machine, and, subsequently, the generation of abstract machine code is almost as costly computationally as concrete machine code generation. Furthermore, abstract machine code generation again loses the ability to reconstruct original program texts.

The Navel system is based on an interpreter for parse trees, a widely used technique for developing quickly an initial language implementation. While parse tree interpreters are relatively slow compared with concrete machine code and lower-level abstract machines, the construction of a compiler to parse trees and of a parse tree interpreter are relatively straightforward, compilation to parse trees is much faster than to concrete or abstract machine code, and original program texts may be reconstructed from parse trees through simple traversal.

5.5. Navel implementation

Navel is compiled from source text to parse trees for subsequent interpretation. Compilation takes place in three stages. Source text is analysed lexically and symbol sequences are generated. Symbol sequences are then analysed syntactically and initial parse trees are built. Finally, parse trees are executed symbolically to carry out context sensitive checks and to plant address information for run time access to values associated with names. Interpretation is based on tree traversal driving what is effectively an SECD machine. These stages are now considered in more but brief detail.

5.5.1. Lexical analysis

The lexical analyser is based on that from Richard's BCPL compiler [127]. Symbols are unique integer values. The lexical analyser consumes the character sequence for a Navel source text inspecting characters and returning symbols along with values for characters and integers. Names are held in an ordered binary tree. Each entry consists of a symbol to indicate a specific reserved word or a name for a variable, the string for the name held as an array of characters, and pointers to sub-trees for alphabetically preceding and following names. Initially, the tree is formed from reserved words which are ordered to give balance. Subsequently, when a potential name is recognised the name tree is searched. If the name is not found then a new entry is made and the name symbol and a pointer to the entry are returned. If the name is found then for a reserved word the symbol from the entry is returned and for a variable name the name symbol and a pointer to the entry are returned.

5.5.2. Object management

Navel object management is based on a heap of tagged two element cells. The tag is a symbol with the top bit used to indicate marking during garbage collection. The elements are termed the head and tail. For structured objects the elements are pointers to other objects. For atomic values, the tail holds the value of characters and integers or a pointer to the string in the name tree for names.

When the system is initialised, the whole heap is claimed from the operating system as a single array and the cells are sown together through the heads to form the free list. When a cell is requested, the next cell from the free list is returned. When the free list is empty garbage collection is initiated. Garbage collection is mark and sweep [45]. All system structures are traversed and marked recursively, as are the arguments to the failed new cell request. The whole heap is then swept: marked cells are unmarked and unmarked cells are added to the free list.

One of the system structures which is garbage collected is the temporary stack. This is used to hold pointers to components of partially created objects to ensure that they are marked should garbage collection occur during the construction of other components.

5.5.3. Syntax analysis

The syntax analyser is recursive descent [36]. It consumes symbols and builds parse trees from heap cells, with the tag holding a symbol to indicate the corresponding construct and the elements being either pointers to subtrees or values for atoms. The temporary stack is used during syntax analysis to hold the left subtree while the right subtree for a construct is being built. No context checks are performed during syntax analysis: initially leaves for names in parse trees have the tag set to the symbol for a name and the tail set to a pointer to the name's string.

5.5.4. Implementation model, name analysis and context checks

At run-time, the interpreter traverses the parse tree carrying out operations indicated by the tag on a node. For names the associated value must be found. Name/value associations may be made by global definitions, by binding bound variables to arguments through function calls or by local definitions.

Global definitions are held in a linked list where each entry contains a pointer to the name string and the associated values. The definitions list is garbage collected.

At run time, values associated with names through function calls or local definitions are held on the stack. All objects on the stack are garbage collected. Note that the names are not held on the stack at run time.

At compile time, during the name pass, the parse tree is evaluated symbolically and the stack is manipulated to reflect name/value associations formed by function calls or local definitions. This enables the prediction of the stack position at run-time of values associated with names. Name nodes are changed during the name pass so that for references to non-global names, the head of the node holds the stack offset for the associated value. Thus, at run time, values for names are found directly on the stack rather than through searching an environment list holding explicit name/value associations.

For functions, rules and local definitions, closures are formed containing details for free variables. At compile time, an uninstantiated closure is formed containing a list of free variables and the positions of their values on the stack at run time. At run time, closures are instantiated with each free variable associated with its value from the stack. Closures hold details only for free variables that are actually used rather than holding the entire defining environment. No run-time searching for names is required as values are found directly in the current stack frame.

During the name pass, when a function, rule or local definition is first encountered, a new stack frame is started and the bound variables or local names are pushed onto the stack. For structured variable lists in functions and on the left of local definitions, checks are made for repeated definition of the same variable. For simultaneous local definitions, all defined names are pushed onto the stack before symbolic evaluation of right hand sides. This ensures that mutual recursion is detected.

Subsequently, when a name is encountered in an expression, the stack is searched top down. If the name is found in the top stack frame, then it was defined by an immediately enclosing function or local definition, so the name node is extended with the relative stack address in the head element. If the name is found in a lower stack frame then it must be a free variable. It is pushed into the top frame along with a level number indicating how many frames down it was defined and its relative address in that frame. Once again, the name node is extended with the relative stack address for the name in the current frame.

If the name is not found on the stack then the global definitions list is searched. If the name is found then its node is replaced with a pointer to the corresponding definitions list entry. Otherwise, it is assumed that the name is a forward reference to a subsequent global definition. A new entry is made associating that name with the empty list in global definitions list, the name node is replaced with a pointer to the new entry and a message is printed warning that an undefined name has been encountered. This approach is used in POP2 implementations and avoids the need for re-compilation following changes to global definitions. As parse trees share global definition entries, changes to such entries will be referenced when global names are subsequently encountered in parse trees.

At the end of a function, rule or local definition, the top stack frame is removed and the level number for each free variable entry is decremented. Any entry with a level value greater than 1 must be for a free variable. If the level value is now 1 then the variable was first introduced in the now current top frame. Otherwise, the name, decremented level number and original stack frame relative address are pushed onto the now current top frame. An uninstantiated closure is formed consisting of a free variable list and the function, rule or local definition. The free variable list consists of the name string and the relative stack address in the now current top frame for each free variable from the old top frame.

At run time, when a function, rule or local definition is first encountered the closure is instantiated, that is the free variable values are picked up from the current top stack frame, using the relative stack address from the old free list, to form a new free list. When a function or rule call or local definition is evaluated, the arguments or right hand side values are evaluated and pushed into a new stack frame along with the values from the instantiated free list. The order of free variables in the list ensures that their associated values are pushed into the anticipated positions in the new frame.

For example, consider:

```
lam x.lam y.lam z.x+y+z
```

This consists of three nested functions. At compile time, the bound variables `x`, `y` and `z` are pushed onto the stack in new frames:

```
|z   ()| offset 0
+------+ frame 3
|y   ()| offset 0
+------+ frame 2
|x   ()| offset 0
+------+ frame 1
```

For the final body:

```
x+y+z
```

`x` is 2 frames down so it is pushed into the current frame with level number 3 and its offset 0 from the lower frame:

```
|x  3 0| offset 1
|z   ()| offset 0
+------+ frame 3
|y   ()| offset 0
+------+ frame 2
|x   ()| offset 0
+------+ frame 1
```

`x` is now at offset 1 in the current frame.

`y` is one frame down so it is pushed into the current frame with level number 2 and its offset 0 from the lower frame:

```
|y  2 0| offset 2
|x  3 0| offset 1
|z   ()| offset 0
+------+ frame 3
|y   ()| offset 0
+------+ frame 2
|x   ()| offset 0
+------+ frame 1
```

`y` is now at offset 2 in the current frame. `z` is at offset 0 in the current frame.

The tree for the inner expression is now:

```
<x,1>+<y,2>+<z,0>
```

with `<...>` indicating the offset in the current frame for a name.

On leaving the inner function a closure is formed. `y` is at level 2 so it is in the immediately preceding frame. `x` is at level 3 so it is pushed onto the preceding frame with a decremented level:

```
|x  2 0| offset 1
|y   ()| offset 0
+------+ frame 2
|x   ()| offset 0
+------+ frame 1
```

and the closure free list contains their addresses in the preceding frame:

```
lam z.{<x,1>,<y,0>} <x,1>+<y,2>+<z,0>
```

with `{...}` indicating the uninstantiated free list.

On leaving the middle function a closure is formed. `x` is at level 2 so it is in the preceding stack frame:

```
|x   ()| offset 0
+------+ frame 1
```

Thus, the free list for the closure contains `x` and its offset:

```
lam y.{<x,0>} lam z.{<x,1>,<y,0>} <x,1>+<y,2>+<z,0>
```

Finally, the outer function has no free variables so its free list is empty:

```
lam x.{} lam y.{<x,0>} lam z.{<x,1>,<y,0>} <x,1>+<y,2>+<z,0>
```

Now consider evaluating:

```
lam x.lam y.lam z.x+y+z 11 22 33
```

at run time. First of all, an instantiated closure for the outer function is formed with an empty free list:

```
lam x.[] lam y.{<x,0>} lam z.{<x,1>,<y,0>} <x,1>+<y,2>+<z,0>
```

where `[...]` indicates an instantiated free list.

Next, the argument `11` is pushed onto the stack for `x`:

```
|    11| offset 0
+------+
```

and a new closure is formed for the middle function with an instantiated free list:

```
lam y.[<x,11>] lam z.{<x,1>,<y,0>} <x,1>+<y,2>+<z,0>
```

Next, the argument `22` is pushed onto the stack for `y`, `11` is pushed on for `x` from the instantiated free list:

```
|    11| offset 1
|    22| offset 0
+------+
```

and an instantiated closure is formed for the inner function:

```
lam z.[<x,11>,<y,22>] <x,1>+<y,2>+<z,0>
```

Finally, the argument `33` is pushed onto the stack for `z` and `11` is pushed on for `x` and `22` for `y` from the instantiated free list:

```
|    22| offset 2
|    11| offset 1
|    33| offset 0
+------+
```

Now:

```
<x,1>+<y,2>+<z,0>
```

is evaluated. The value for `x` is found at offset 1, the value for `y` is at offset 2 and that for `z` is at offset 0.

Note that instantiated closures are always formed for functions even when a curried function is fully applied, as in the above example. In such cases, closure instantiation is unnecessary as free variables' values are always on the stack. However, determining the degree of application of curried functions at compile time is complex. Furthermore, this approach simplifies tail recursion optimisation: the current stack frame can always be cleared after a function call as a function closure always contains its free variables' values in the instantiated free list. None the less, in Navel uncurried functions are faster than their curried equivalents as less closure manipulation for free variables is required.

5.5.5. Interpretation

The interpreter description that follows is very like that of the Navel semantics in chapters 3 and 4. As already discussed, the interpreter traverses a parse tree carrying out appropriate operations determined by the tag at each node, using results from sub-node evaluation. Initially, the global definitions list and stack are empty.

A separate stack is used to hold stack frame base addresses on function entry. While moderately wasteful of space, this was found to be faster than combining return information and variable values in a single stack.

For a boolean, integer or character value, that value is returned.

For a global name, the node is the corresponding entry in the global definitions list and the value from the entry is returned.

For a non-global name, the value is found directly from the current stack frame using the frame offset.

For an arithmetic operator, the operands are evaluated. If the operands are not integer or character then a run-time error occurs. Otherwise the arithmetic operation is carried out and an integer value returned. For arithmetic with characters they are converted to integer ASCII values.

For a comparison operator, the operands are evaluated. For equality and inequality operators, if the operands are of different types then the comparison returns `false.` Otherwise, the comparison is carried out and a boolean value is returned. For functions, equality requires structural identity. For ordering operators, if the operands are not of the same or coercible types then a run-time error occurs. Otherwise, the comparison is carried out and a boolean value is returned. For ordered comparison of integers and characters, characters are converted to integer ASCII values.

For a logical operator, the operands are evaluated. If they do not return boolean values then a run-time error occurs. Otherwise, the appropriate operation is carried out and a boolean value is returned.

For a conditional expression, the condition is evaluated. If it returns `true` then the `then` option is evaluated. If it returns `false` then the `else` option is evaluated.

For a case expression, the selection expression is evaluated. Each case expression is evaluated until a match for the selection expression value is found. The corresponding result expression is then evaluated. If no case expressions match the selection expression value then the default expression value is returned.

For list construction, both operands are evaluated lazily. That is function or rule calls and local definitions have all enclosed uninstantiated closures instantiated but are not evaluated further. Other forms of operand are evaluated fully. A cell is claimed from the object free list, tagged as a list and has its head and tail set to the lazily evaluated operands. The cell is returned.

For list selection, the appropriate element is evaluated and the value overwrites the element. This is an inexpensive resolution of lazily evaluated elements. For indexed list selection, successive elements are evaluated and overwritten until the required element is reached.

For field construction, a new cell is claimed, tagged as a field and has the head set to the identifying name and the tail set to the evaluated expression. The cell is returned.

For field selection, the required name is matched against the tag name in the field. If they match then the value is returned. Otherwise a run-time error occurs. For indexed field selection from lists, successive list elements are evaluated and overwritten. For elements which are fields, the tag name is compared with the required name. If they match and the required index has been reached then the field value is returned. If they match but the required index has not been reached then indexed field selection continues with a decremented index. If they do not match then indexed field selection continues.

For an uninstantiated function or rule closure, the closure is instantiated as described in the previous section.

For a function call, a new stack frame is started. The argument is evaluated. For structured bound variables, a list argument is evaluated element by element until an appropriate structure is formed. If the argument is not a list of the required shape then a run-time error occurs. Otherwise, argument elements corresponding to bound variables are pushed onto the stack and the instantiated free list entries are pushed onto the stack. The body is evaluated lazily, the stack frame is removed and the suspended body is evaluated.

For a local definition, a new stack frame is started, a dummy entry is pushed onto the stack for each defined name and the free variable values are pushed onto the stack from the instantiated free variable list. Each left hand side is then inspected. For structured names, a right hand side list value is evaluated element by element until an appropriate structure is formed. If the right hand side expression does not evaluate to a list of appropriate shape then a run-time error occurs. Otherwise, the dummy entries are replaced with the corresponding list elements. For a single name, if the right hand side is not a rule or function then it is evaluated and the value replaces the dummy value. A rule or function is not evaluated. Instead, a copy of the uninstantiated closure replaces the dummy value. When all left hand sides have been considered, the stack frame is inspected. All uninstantiated closures are changed to instantiated closures with the uninstantiated free lists replaced by instantiated free lists. Thus, initially an instantiated free list may contain a pointer to an uninstantiated closure for a subsequent definition. When that latter closure is finally instantiated, it is overwritten rather than replaced. Thus, pointers in other instantiated free lists will now refer to the same but overwritten entity. Finally, the local definition body is evaluated lazily, the stack frame is removed and the suspended body is then evaluated.

For a global definition, the global definitions list is searched for the name. If it is not found then a new entry is made at the end with the name associated with the value of the defining expression. Otherwise, the existing entry is modified to reflect the new value associated with the name.

For a rule application, a new stack frame is started, the instantiated free list values are pushed onto the stack, the rule is traversed and matching actions are performed. Each option is tried in turn until one succeeds: the tree and the rest of the string argument are then returned. If all fail then an empty list and the whole argument string are returned. For each option, the components are tried from left to right in turn. If one fails then the whole option fails and an empty list and the whole argument string are returned. If they all succeed, then the corresponding sub-trees are formed into a list which is returned along with the rest of the argument string. For a string component, it is matched against the start of the argument string. If the match succeeds then the string and the rest of the argument string are returned. For a name component, the corresponding rule value is used. If it succeeds then the sub-tree forms the value of a field tagged with the name which is returned with the rest of the argument string. For a nested rule, it is applied. For a bracketed expression, it is evaluated and the resultant rule is used. For system rules, appropriate matching actions are invoked.

For rule matching against trees, the tree expression is evaluated to a list. Each case expression is evaluated to a flat rule consisting of a single sequence of names and strings. The tree expression list is matched against the case expression rule. For a successful match, strings in both must be the same and in the same position and a field in the tree must be in the same position as and have the same name tag as a name in the rule. After a successful match, the corresponding result expression is evaluated. If no case succeeds then the default expression is evaluated.

5.5.6. Construction and performance

The Navel system consists of 4701 lines of C and compiles to 90688 bytes of machine code on a DECstation 5000. The construction is modular. While originally implemented under UNIX, it has also been ported to run on the IBM PC architecture under MSDOS, and to run on the Apple Macintosh architecture by a group at Griffiths University, Australia.

The following simple benchmarks were carried out on a VAX 11/750 to compare Navel with Franz LISP [47], C Prolog [114], Cardelli's ML [25] and 1976 SASL [151]. The ML implementation is based on an abstract machine code compiler; all the others are based on interpreters.

The benchmarks used are Ackermann's function:

```
def ack m:n =
 case true of
 m=0 -> n+1,
 n=0 -> ack (m-1):1,
 ack (m-1):(ack m:(n-1));
```

to test recursion depth and naive insertion sort:

```
def sort l =
 if l=()
 then ()
 else insert (hd l):(sort (tl l));

def insert a:l =
 case true of
 l=()->a:,
 a<=(hd l)->a:l,
 (hd l):insert a:(tl l):
```

to test recursion depth and list concatenation.

Each benchmark was run 100 times and the CPU times from the system timer `time` accumulated in a file. All times are in seconds. The times were then totaled to find an overall mean. Each system was then timed just starting up and starting up, loading and compiling but not running the tests.

The times may then be normalised for startup time by subtracting the startup time from the total time giving load, compile and run times:

```
Ackermann's function load, compile and run - ack(3,3)

language |  mean  | % of Navel
---------+--------+-----------
 Navel   |  8.706 | 100.0
 LISP    |  5.025 |  57.7
 ML      |  1.01  |  11.6
 Prolog  |  6.856 |  78.7
 SASL    | 23.285 | 267.4

Insertion sort load, compile and run - 100 elements in descending order

language |  mean  | % of Navel
---------+--------+-----------
 Navel   | 30.356 | 100.0
 LISP    | 32.971 | 108.6
 ML      |  6.011 |  19.8
 Prolog  | 15.278 |  50.3
```

SASL had inadequate space for this test.

```
Insertion sort load, compile and run - 40 elements in descending order

language |  mean  | % of Navel
---------+--------+-----------
 Navel   |  5.104 | 100.0
 LISP    |  5.075 |  99.4
 ML      |  2.962 |  58.0
 Prolog  |  2.769 |  54.2
 SASL    | 12.386 | 242.6
```

The run times for the tests are then be found by subtracting the corresponding start, load and compile times from the total times:

```
Ackermann's function run time - ack(3,3)

language |  mean  | % of Navel
---------+--------+-----------
 Navel   |  8.643 | 100.0
 LISP    |  4.919 |  56.9
 ML      |  0.118 |   1.3
 Prolog  |  6.631 |  76.7
 SASL    | 22.663 | 262.2

Insertion sort run time - 100 elements in descending order

language |  mean  | % of Navel
---------+--------+-----------
 Navel   | 30.244 | 100.0
 LISP    | 32.82  | 108.5
 ML      |  3.865 |  12.7
 Prolog  | 14.95  |  49.4
```

Insertion sort run time - 40 elements in descending order

language	mean	% of Navel
Navel	4.992	100.0
LISP	4.924	98.6
ML	0.816	16.3
Prolog	2.441	48.8
SASL	11.684	234.0

For Ackermann Navel is more than twice as fast as 1976 SASL, somewhat slower than C Prolog and around half the speed of Franz LISP. For insertion sort, Navel is twice the speed of SASL, around the same speed as Franz LISP and around half the speed of C Prolog. ML is substantially faster than all other languages. SASL's low speed may reflect its combinator based implementation and that it is lazy: all the other languages are applicative order.

The load and compile times may be found by taking the start times from the start, load and compile times:

Ackermann's function - load and compile time

language	mean	% of Navel
Navel	0.063	100.0
LISP	0.106	168.2
ML	0.892	1415.8
Prolog	0.234	371.4
SASL	0.622	987.3

Insertion sort - load and compile time

language	mean	% of Navel
Navel	0.112	100.0
LISP	0.151	134.8
ML	2.146	1916.0
Prolog	0.328	292.8
SASL	0.702	626.7

Here, Navel is faster than the other languages. For Navel, LISP and Prolog, the load times are small enough to not be significant psychologically. For ML, however, the load time is becoming significant. This reflects the longer time taken for type checking and abstract machine code generation by the ML compiler. However, compilation is often only a small component of overall processing time.

Navel garbage collection, on a DECStation 5000 under ULTRIX, for a heap of 256,000 cells takes around 0.4 seconds. Navel will run small language definitions with a heap as little as 4096 cells, with frequent but fast garbage collections.

5.5.7. Pretty printer

The Navel pretty printer is used to reconstruct program text from parse trees in a canonical form. Parse trees are traversed recursively. A precedence table coded as nested case statements is used to control bracketing of nested expressions.

For local definitions and conditional, rule match and case expressions, a standard indented layout is used. For a local definition, the `let` starts a new line, indented one space to the right, and the `and`s and `in` lie below it on new lines. For a conditional expression, the `if` starts a new line. indented one space to the right, and the `then` and `else` lie below it on new lines. For a rule match or case expression, the `rule` or `case` starts a new line, indented one space to the right, and the case options lie below it on new lines.

Free variable bindings in instantiated closures are recreated as local definitions. Thus, given:

```
def add x y = x+y;
def inc = add 1;
```

the value of `inc` is pretty printed as:

```
let x = 1
in lam y.x+y;
```

For simultaneous local definitions, the pretty printer attempts to avoid repetitive display of shared free variables by checking to see if a free variable/value association has already been printed at the current level of definition. The check is very crude and may be foiled by nested mutual references.

For lazily evaluated lists, the list is not fully evaluated prior to printing. Instead, unevaluated function calls are printed enabling a finite printed representation of infinite structures.

5.6. Environment

As suggested above, Navel was intended for interactive use. To ease implementation, a simple environment was provided and underlying system facilities were used as far as possible. The interface is based on a command line model which prompts repeatedly for and inputs from the keyboard expressions for evaluation, global definitions for subsequent use and system commands.

The Navel prompt for input is:

```
ok
```

on a line on its own. Input may be multi-line, without prompts for each line, and is ended with a: `;`

Commands are based on meaningful identifiers and, where appropriate, are followed by parameters without bracketing or separators.

The command to leave Navel is:

```
end
```

As a minimum, commands were needed for global definitions, input from a file, output to a file, display of all global definitions, deletion of nominated global definitions, deletion of all global definitions and editing of nominated global definitions.

File names are based on UNIX paths. In the following: `<file name>` stands for an arbitrary UNIX path.

Loading from files is generalised to enable files to contain arbitrary global definitions, expressions and commands including other load commands. The command to load from a file is:

```
load <file name>
```

and to load from the last used file:

```
load
```

The command to save all global definitions as text in a file is:

```
save <file name>
```

and to save to the last used file:

```
save
```

The command to display all global definitions is:

```
defs
```

The command to delete a global definition is:

```
delete <name>
```

The command to delete all global definitions and re-initialise the system is:

```
reset
```

The command to edit a global definition is:

```
edit <name>
```

The definition for name is recreated as text, placed in a temporary file and the `vi` editor is called. On leaving `vi` the temporary file is loaded.

A shell escape of the form:

```
!<UNIX command>
```

is provided where `<UNIX command>` is an arbitrary UNIX command.

Simple forward tracing of expression evaluation is provided. The command:

```
trace
```

toggles tracing on and off. In trace mode, all expressions and their values are printed with values indented to the right of expressions. Rule match and case expressions are truncated.

Timing of expression evaluation is provided. The command:

```
time
```

toggles timing on and off.

For experimental purposes, either strict or lazy evaluation of lists may be nominated. Often, it is frustrating to have to write an extra layer of local definition to evaluate function calls explicitly in order to see their values after list construction. The command:

```
lazy
```

toggles laziness on and off.

The command:

```
traces
```

displays the current status of the above three togglable states.

5.7. Extensions

A number of extensions are made to Navel which ease testing large programs. While useful, they are not central to Navel. Furthermore, several are imperative and so their inclusion in core Navel would complicate the formal definition.

As seen in the formal definition of Navel, error handling complicates DS. An exit expression with the form:

```
<expression> ::= ... | exit <aexp>
```

is provided to terminate evaluation and return the value of `<aexp>` identified as an exit. Exits could be added to the semantics as a new domain with explicit value passing and testing, as for errors. Alternatively continuation passing might be used to define them. Exits are extremely useful for termination after run-time errors and as defaults in rule match expressions.

I/O primitives are added to Navel to enable the construction of interactive and file handling Navel programs. In principle, stream or continuation passing models might be used. However, I/O is essentially imperative and so explicitly imperative constructs are implemented. For single character input:

```
<expression> ::= ... | read
```

returns the next character from the standard input.

For file input:

```
<expression> ::= ... | fread <aexp>
```

is slightly different. `<aexp>` is evaluated to a string file name and the file is opened as a lazy list of characters. This has proved useful for testing formal definition implementations with large example programs.

For output:

```
<expression> ::= ... | write <aexp>
```

displays the value of `<aexp>` on the standard output. For a string argument, the enclosing quotes are not displayed.

```
<expression> ::= ... | writeln <aexp>
```

follows the value of `<aexp>` with a newline.

For file output:

```
<expression> ::=  ... | fwrite <aexp> <aexp>
```

writes the value of the second `<aexp>` to the file whose name is the string from the first `<aexp>` which is then closed.

Other Navel extensions include:

```
<expression> ::= ... | eval <aexp>
```

which evaluates the Navel text string returned by `<aexp>`. This may not contain global definitions or system commands. Thus, Navel programs cannot modify the global context.

The object:

```
<base> ::= ... | ?
```

generates a random integer.

`?` used as a function will select an arbitrary element from a list argument or generate an arbitrary string text from a rule argument. The generation of texts from rules is discussed further in Michaelson [98]. Such generation is useful both for checking grammars and for the production of test cases.

System commands may be initiated from Navel programs by:

```
<expression> ::= ... | system <aexp>
```

`<aexp>` is evaluated to a string system command which is then executed. The result is the integer return code.

5.8. Use

Navel has been used at Heriot-Watt University for teaching a Functional Programming course to 2nd year BSc Computer Science students for 4 years and for projects for a course on Language Specification and Implementation for 4th year BSc Computer Science students for 2 years. Student feedback suggests that the system is robust and that its speed is acceptable.

Navel has been shipped to around 20 sites worldwide. In particular, it has been used to teach functional programming to undergraduates at Griffiths University, Australia and to teach language implementation from formal definitions to undergraduates at Tsinghua University, China. Once again, feedback has been favourable.

Chapter 6

Language implementation from DS in Navel

6.1. Introduction

This chapter looks in detail at the use of Navel in implementing a simple vintage imperative language from a DS definition.

6.2. Implementing a block structured imperative language in Navel

Consider an ALGOL-like language which provides integers and arithmetic, comparison and logical expressions, integer variable declarations, one dimensional integer array declarations, assignment, I/O, statement sequences, conditional and iterative statements, procedures with call by value and by reference, blocks and local declarations. For example, a program to read into an array until a 0 is encountered, counting elements, sort the array with naive bubble sort and print it, is:

```
proc aread(var a[100],var n)
begin
     var i
     n:=0
     read i
     while i<>0 and n<>100 do
     begin
          n:=n+1
          a[n]:=i
          read i
     end
end
proc awrite(var b[100],n)
begin
     var i
     i:=1
     while i<=n do
     begin
          write b[i]
          i:=i+1
     end
end
proc asort(var c[100],n)
begin
     var i
     i:=n-1
     while i>0 do
     begin
          var j
          j:=1
          while j<=i do
          begin
               if c[j]>c[j+1] then
               begin
                    var t
                    t:=c[j]
                    c[j]:=c[j+1]
```

```
                    c[j+1]:=t
                end
                j:=j+1
            end
            i:=i-1
        end
    end
    var x[100]
    var no
    aread(x,no)
    awrite(x,no)
    asort(x,no)
    awrite(x,no)
```

6.3. Syntax

The concrete syntax for this language is:

```
<program> ::= [ <declarations> ] <statements>
<declarations> ::= <declaration> <declarations> | <declaration>
<declaration> ::= var <identifier> [ ’[’ <number> ’]’ ] | <procedure>
<procedure> ::= proc <identifier> ’(’ [ <fparams> ] ’)’ <statement>
<fparams> ::= <fparam> , <fparams> | <fparam>
<fparam> ::= var <identifier> [ ’[’ <number> ’]’ ] | <identifier>
<statements> ::= <statement> <statements> | <statement>
<statement> ::= <assign> | <input> | <output> | <sif> | <while> |
                <block> | <call>
<assign> ::= <identifier> [ ’[’ <expression> ’]’ ] := <expression>
<input> ::= read <identifier> [ ’[’ <expression> ’]’ ]
<output> ::= write <expression>
<expression> ::= <term> {+ -} <term> | <term>
<term> ::= <factor> {* /} <factor> | <factor>
<factor> ::= [-] base
<base> ::= <identifier> [ ’[’ <expression> ’]’ ] |
           <number> | ( <expression> )
<identifier> ::= <letter> [ <rest of id> ]
<rest of id> ::= <letter> [ <rest of id> ] | <digit> [ <rest of id> ]
<letter> ::= a | ... | z | A | ... | Z
<number> ::= <digit> | <number> <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<sif> ::= if <logexp> then <statement> [ else <statement> ]
<logexp> ::= <logterm> and <logterm> | <logterm>
<logterm> ::= <logfactor> or <logfactor> | <logfactor>
<logfactor> ::= [ not ] <logbase>
<logbase> ::= <expression> <compop> <expression> | ( <logexp> )
<compop> ::= <> | <= | < | = | >= | >
<while> ::= while <logexp> do <statement>
<block> ::= begin <program> end
<call> ::= <identifier> ’(’ [ <aparams> ] ’)’
<aparams> ::= <expression> , <aparams> | <expression>
<arithop> ::= + | - | * | /
<empty> ::=
```

The rules for logical and arithmetic expressions, ie `<logexp>`, `<logterm>`, `<expression>` and `<term>`, are non-recursive. LL(k) parsing excludes left recursion but the equivalent right recursion results in right associative parse trees: logical and arithmetic operators are usually left associative. To simplify the presentation, strict bracketing is required for sequences of operators of the same precedence. This issue is considered in more detail in the next chapter.

The rule for conditional statements, ie `<sif>`, is ambiguous, exemplifying the classic "dangling else" problem. In strict left to right parsing, an `else` will actually be associated with the first `then` on its left.

In Navel, the above grammar may be written as:

```
def program = {{declarations | {}} statements};
def declarations = {declaration declarations | (declaration)};
def declaration = {"var" identifier {"[" number "]" | {}} | procedure};
def procedure = {"proc" identifier "(" {fparams | {}} ")" statement};
def fparams = {fparam "," fparams | (fparam)};
def fparam = {"var" identifier {"[" number "]" | {}} | identifier};
def statements = {statement statements | (statement)};
def statement = {(assign) | (input) | (output) | (sif) | (while) |
                 (block) | (call)};
def assign = {identifier {"[" expression "]" | {}} ":=" expression};
def expression = {term {"+" | "-"} term | (term)};
def term = {factor {"*" | "/"} factor | (factor)};
def factor = {"-" base | (base)};
def base = {identifier{"[" expression "]" | {}} |
            number | "(" expression ")"};
def input = {"read" identifier {"[" expression "]" | {}}};
def output = {"write" expression};
def sif = {"if" logexpression "then" statement {"else" statement | {}}};
def logexpression = {logterm "and" logterm | (logterm)};
def logterm = {logfactor "or" logfactor | (logfactor)};
def logfactor = {"not" logbase | (logbase)};
def logbase = {expression (compop) expression | "(" logexpression ")"};
def compop = {"<>" | "<=" | "<" | "=" | ">=" | ">"};
def while = {"while" logexpression "do" statement};
def block = {"begin" program "end"};
def call = {identifier "(" {aparams | {}} ")" };
def aparams = {expression "," aparams | (expression)};
```

The form:

```
{ <construct> | {}}
```

is used for 0 or one occurrences of `<construct>`.

These rules are not factored. Singleton non-terminals are bracketed with `(...)` to prevent tree nodes being created for them.

An abstract syntax for this language is:

```
d ∈ <declarations>
p ∈ <procedure>
fp ∈ <fparams> + <empty>
s ∈ <statements>
ap ∈ <aparams> + <empty>
l ∈ <logexp>
cop ∈ <compop>
e ∈ <expression>
aop ∈ <arithop>
i ∈ <identifier>
ll ∈ <letter>
n ∈ <number>
d ∈ <digit>
ε ∈ <empty>

d -> d d | var i | var i [ n ] | p
p ->  proc i (fp) s
fp -> fp , fp | i | var i [ n ] | var i | ε
s -> s s | i := e | i [ e ] := e | read i | read i [ e ] | write e |
     if l then s else s | if l then s | while l do s | begin p end |
     i (ap)
```

```
ap -> ap , ap | e | ε
l -> l or l | l and l | not l | e cop e
cop -> = | <> | <= | < | >= | >
e -> i | i [ e ] | n | e aop e | - e
aop -> + | - | * | /
i -> ll | i d | i ll
ll -> a | ... | z | A | ... | Z
n -> n n | d
d -> 0 | ... | 9
```

6.4. Semantic domains

The semantic domains for integers, `N`, and booleans, `B`, are assumed. An environment/store model is used for the semantics. The environment associates identifiers with either locations in the store, or dope vectors for arrays consisting of a list pair for the start location in store and the upper bound. Environment entries are tagged to identify the type of entry:

```
T == tags == VAR | PROC | ARRAY
E == environments == <identifier> -> T * (N + N * N)
```

The store associates addresses, and integer values or procedure texts:

```
S == stores == N -> N + <procedure>
```

Formally, this DS is non-compositional. However, this simplifies greatly the presentation. Compositionality may be restored by folding procedure texts into the semantic function for procedure calls.

Inputs and outputs are integer sequences:

```
I == inputs == N*

O == outputs == N*
```

In Navel, the integer and boolean domains are modeled by the integer and boolean types. Environments and stores are modeled by functions. Store addresses are modeled by integers. Environment tags are modelled by strings. Dope vectors are modeled by integer pair lists. Inputs and outputs are modeled by linear integer lists.

6.5. Program semantics

The semantics now follows. Note that type testing and error passing have been omitted to simplify the presentation. Each formal definition is followed by the Navel equivalent.

For a program with declarations followed by statements, a new environment, store and free location are found from the declarations, using the current environment, store and free location, and used to find the final store, input and output:

```
mprog : <program> -> E * S * I * O * N -> S * I * O
mprog [d s] e:s:i:o:n  =
 let newe:news:newn = md [d] e:s:n
 in mstate [s] newe:news:i:o:newn
```

For a single statement the final store, input and output are found:

```
mprog [s] e:s:i:o:n = mstate [s] e:s:i:o:n
```

The Navel is:

```
def mprog p e:s:i:o:n =
 rule p of
```

```
{decls states} ->
 let newe:news:newn = mdecl p^decls e:s:n
 in mstate p^states newe:news:i:o:newn,
{states} -> mstate p^states e:s:i:o:n,
exit (p:"not a program");
```

6.6. Declaration semantics

For a declaration sequence, the first is evaluated to produce a new environment, store and free address using the current environment, store and free address. The rest of the sequence is then evaluated using the new environment, store and free address to return a final environment, store and free address:

```
mdecl : <declarations> -> E * S * N -> E * S * N
mdecl [d1 d2] e:s:n = mdecl [d2] (mdecl [d1] e:s:n)
```

For a simple variable, the environment is updated with an association between the current free location and the identifier, and the free location is incremented:

```
mdecl [var i] e:s:n = (new e [i] VAR:n):s:(n+1)
```

For an array, the environment is updated with an association between the current free location and upper bound, and the identifier, and the current free location is incremented with the upper bound:

```
mdecl [var i [ n ] ] e:s:n =
 let bound = val [n]
 in (new e [i] ARRAY:bound:n):s:(n+bound)
```

For a procedure declaration, the identifier is associated with the next free address in the environment and the next free address is associated with the body in the store. The free address is incremented:

```
mdecl [p] e:s:n = mproc [p] e:s:n,

mproc : <procedure> -> E * S * N -> E * S * N
mproc [proc i ( fp ) s] e:s:n =
 let newe = new e [i] PROC:n
 and news = new s n [proc i ( fp ) s]
 in newe:news:(n+1)
```

The Navel for declarations is:

```
def mdecl d e:s:n =
 rule d of
 {decl decls} -> mdecl d^decls (mdecl d^decl e:s:n),
 {"var" identifier} -> (new e d^identifier "var":n):s:(n+1),
 {"var" identifier "[" number "]"} ->
  let bound = val d^number
  in (new e d^identifier "array":bound:n):s:(n+bound),
 {procedure} -> mproc d^proc e:s:n,
 exit (d:"not a decl");

def mproc p e:s:n =
 let newe = new e p^identifier "proc":n
 and news = new s n p
 in newe:news:(n+1);
```

6.7. Statement semantics

For statements, each DS fragment is followed by the equivalent Navel fragment.

For a statement sequence, the first statement is evaluated with the initial environment, store, inputs, outputs and free address to produce new store, inputs and outputs. The rest of the sequence is then evaluated using the initial environment and free address, and the store, inputs and outputs from the first statement:

```
mstate : <statements> -> E * S * I * O * N -> S * I * O
mstate [s1 s2] e:s:i:o:n =
 let news:newi:newo = mstate [s1] e:s:i:o:n
 in mstate [s2] e:news:newi:newo:n

def mstate st e:s:i:o:n =
 rule st of
 {state states} ->
  let news:newi:newo = mstate st^state e:s:i:o:n
  in mstate st^states e:news:newi:newo:n,
```

A block is treated like a program:

```
mstate [begin p end] e:s:i:o:n = mprog [p] e:s:i:o:n

 ...
 {"begin" program "end"} -> mprog st^program e:s:i:o:n,
 ...
```

An iteration is evaluated recursively:

```
mstate [while l do s] e:s:i:o:n = mwhile [l]:[s] e:s:i:o:n

mwhile : <logexp> * <statement> -> E * S * I * O * N -> S * I * O
mwhile [l]:[s] e:s:i:o:n =
 if mlogexp [l] e:s
 then
  let news:newi:newo = mstate [s] e:s:i:o:n
  in mwhile [l]:[s] e:news:newi:newo:n
 else s:i:o

 ...
 {"while" logexp "do" state} -> mwhile st^logexp:st^state e:s:i:o:n,
 ...

def mwhile le:ls e:s:i:o:n =
 if mlogexp le e:s
 then
  let news:newi:newo = mstate ls e:s:i:o:n
  in mwhile le:ls e:news:newi:newo:n
 else s:i:o;
```

For a conditional statement, if the logical expression is true then the first statement is executed. Otherwise the second statement is executed. If there is no false option then the initial state, input and output are returned:

```
mstate [if l then s1 else s2] e:s:i:o:n =
 if mlogexp [l] e:s
 then mstate [s1] e:s:i:o:n
 else mstate [s2] e:s:i:o:n

mstate [if l then s] e:s:i:o:n =
 if mlogexp [l] e:s
 then mstate [s] e:s:i:o:n
 else s:i:o

 ...
 {"if" logexp "then" state "else" state} ->
  if mlogexp st^logexp e:s
```

```
  then mstate stˆstate e:s:i:o:n
  else mstate stˆstate@2 e:s:i:o:n,

 {"if" logexp "then" state} ->
  if mlogexp stˆlogexp e:s
  then mstate stˆstate e:s:i:o:n
  else s:i:o,
 ...
```

An output extends the output sequence with the value of the expression:

```
mstate [write e] e:s:i:o:n = s:i:(atend o (mexp [e] e:s))

atend : L -> N -> L
atend l i =
 if l=()
 then i:()
 else
  let t = atend (tl l) i
  in (hd l):t

 ...
 {"write" exp} ->
  let newo = atend o (mexp stˆexp e:s)
  in s:i:newo,
 ...

def atend l i =
 if l=()
 then i:
 else
  let t = atend (tl l) i
  in (hd l):t;
```

For a simple assignment, if the identifier is of a simple variable then the store is updated at its associated address with the value of the expression:

```
mstate [i := e] e:s:i:o:n =
 let t:addr = e [i]
 in
  if t<>VAR
  then error
  else new s addr (mexp [e] e:s):i:o,

 ...
 {identifier ":=" exp} ->
  let t:addr = e stˆidentifier
  in
   if t<>"var"
   then exit (st:"not a simple variable but":t)
   else
    let news = new s addr (mexp stˆexp e:s)
    in news:i:o,
 ...
```

For an array assignment, if the identifier is that of an array then the index is checked against the bounds and the appropriate store address is updated:

```
mstate [i [ e1 ] := e2] e:s:i:o:n =
 let t:dv = e [i]
 in
  if t<>ARRAY
```

```
  then error
  else
   let bound:addr = dv
   and subs = mexp [e1] e:s
   in
    if subs<1 | subs>bound
    then error
    else (new s addr+subs-1 (mexp [e2] e:s)):i:o,

 ...
 {identifier "[" exp "]" ":=" exp} ->
  let t:dv = e st^identifier
  in
   if t<>"array"
   then exit (st:"not array but":t)
   else
    let bound:addr = dv
    and subs = mexp st^exp e:s
    in
     if subs<1 | subs>bound
     then exit ("array bound error in":st)
     else
      let news = new s addr+subs-1 (mexp st^exp@2 e:s)
      in news:i:o,
 ...
```

For a simple input, if the identifier is of a simple variable then, provided the input is not empty, the associated store address is updated with the next input value:

```
mstate [read i] e:s:i:o:n =
 let t:addr = e [i]
 in
  if t<>VAR
  then error
  else
   if i=()
   then error
   else (new s addr (hd i)):(tl i):o

 ...
 {"read" identifier} ->
  let t:addr = e st^identifier
  in
   if t<>"var"
   then exit (st:"not a simple variable but":t)
   else
    if i=()
    then exit ("end of input in":st)
    else
     let news:newi = (new s addr (hd i)):(tl i)
     in news:newi:o,
 ...
```

For input to an array element, if the identifier is of an array then the index is checked against the bounds and, if the input is not empty, then the appropriate store address is updated with the next input value:

```
mstate [read i [ e ] ] e:s:i:o:n =
 let t:dv = e [i]
 in
  if t<>ARRAY
  then error
  else
```

```
    let bound:addr = dv
    and subs = mexp [e] e:s
    in
     if subs<1 | subs>bound
     then error
     else
      if i=()
      then error
      else (new s addr+subs-1 (hd i)):(tl i):o
```

```
...
{"read" identifier "[" exp "]"} ->
 let t:dv = e st^identifier
 in
  if t<>"array"
  then exit (st:"not array but":t)
  else
   let bound:addr = dv
   and subs = mexp st^exp e:s
   in
    if subs<1 | subs>bound
    then exit ("array bound error in":st)
    else
     if i=()
     then exit ("end of input in":st)
      else
       let news:newi = (new s addr+subs-1 (hd i)):(tl i)
       in news:newi:o,
...
```

6.8. Procedure call semantics

For a procedure call, the procedure is executed:

```
mstate [i ( ap )] e:s:i:o:n =
 let t:addr = e [i]
 in
  if t<>PROC
  then error
  else mcall (s addr) [ap] e:s:i:o:n
```

The Navel is ordered slightly differently to identify initially calls with and without actual parameters:

```
...
{identifier "(" aparams ")"} -> mcall st e:s:i:o:n,
{identifier "(" ")"} -> mcall st e:s:i:o:n,
...
```

First of all, the procedure and call are checked for the presence of compatible parameters:

```
mcall : <proc> -> <call> -> E * S * I * O * N -> S * I * O
mcall [proc i ( ε ) s] [ε] e:s:i:o:n = mstate [s] e:s:i:o:n
mcall [proc i ( ε ) s]  [ap] e:s:i:o:n = error
mcall [proc i ( fp ) s] [ε] e:s:i:o:n  = error
mcall [proc i ( fp ) s] [ap] e:s:i:o:n =
 let newe:news:newn = enter [fp] [ap] e:s:n
 in mstate p^state newe:news:i:o:newn
```

Again, the Navel is ordered slightly differently. The test for the procedure type is followed by nested rule matching to check for parameter compatibility:

```
def mcall c e:s:i:o:n =
 let t:addr = e c^identifier
 in
  if t<>"proc"
  then exit (t:"not a proc")
  else
   let p = s addr
   in
    rule p of
    {"proc" identifier "(" ")" state} ->
     rule c of
     {identifier "(" ")"} -> mstate p^state e:s:i:o:n,
     {identifier "(" aparams ")"} ->
      exit (p^identifier:"has no parameters"),
     (),
    {"proc" identifier "(" fparams ")" state} ->
     rule c of
     {identifier "(" ")"} -> exit (p^identifier:"has parameters"),
     {identifier "(" aparams ")"} ->
      let newe:news:newn = enter p^fparams c^aparams e:s:n
      in mstate p^state newe:news:i:o:newn,
     (),
    ();
```

Next, the parameter structure is compared:

```
enter : <fparams> -> <aparams> -> E * S * N -> E * S * N
enter [fp1 , fp2] [ap1 , ap2] e:s:n =
 enter [fp2] [ap2] (match [fp1] [ap1] e:s:n)
enter [fp1 , fp2] [e] e:s:n = error
enter [fp] [ap1 , ap2] e:s:n = error
enter [fp] [e] e:s:i:o:n = match [fp] [e] e:s:n
```

In the Navel, nested rule matching is used instead of explicit cases:

```
def enter f a e:s:n =
 rule f of
 {fparam "," fparams} ->
   rule a of
   {exp "," aparams} -> enter f^fparams
                              a^aparams
                              (match f^fparam a^exp e:s:n),
   exit "too few actual parameters",
  rule a of
  {exp "," aparams} -> exit "too many actual parameters",
  match f a e:s:n;
```

Individual formal and actual parameters are matched for type compatibility and the environment and store are updated accordingly:

```
match : <fparam> -> <aparam> -> E * S * N -> E * S * N
```

A value formal parameter is bound to the value of the actual parameter:

```
match [i] [e] e:s:n = (new e [i] VAR:n):(new s n (mexp [e] e:s)):(n+1),

def match f a e:s:n =
 rule f of
 {identifier} ->
  (new e f^identifier "var":n):(new s n (mexp a e:s)):(n+1),
 ...
```

A reference formal parameter is bound to the address of a variable or array element actual parameter:

```
match [var i1] [i2] e:s:n =
 let t:addr = e [i2]
 in
  if t<>VAR
  then error
  else (new e [i1] VAR:addr):s:n,
match [var i1] [i2 [ e ]] e:s:n =
 let t:dv = e [i2]
 in
  if t<>ARRAY
  then error
  else
   let bound:addr = dv
   and subs = mexp [e] e:s
   in
    if subs<1 | subs>bound
    then error
    else (new e [i1] VAR:(addr+subs-1)):s:n,
match [var i] a e:s:n = error
```

The Navel for reference parameters is:

```
def match f a e:s:n =
 rule f of
 ...
 {"var" identifier} ->
  rule a of
  {identifier} ->
   let t:addr = e a^identifier
   in
    if t<>"var"
    then exit (f:"wants a simple var argument, not":a)
    else (new e f^identifier "var":addr):s:n,
 {identifier "[" exp "]"} ->
  let t:dv = e a^identifier
  in
   if t<>"array"
   then exit (a^identifier:"not an array")
   else
    let bound:addr = dv
    and subs = mexp a^exp e:s
    in
     if subs<1 | subs>bound
     then exit (a:"array bound error")
     else (new e f^identifier "var":(addr+subs-1)):s:n,
 exit (a:"illegal reference actual parameter"),
 ...
```

An array reference formal parameter has its name associated with the dope vector for an array actual parameter:

```
match [var i1 [ n ] ] [i2] e:s:n =
 let t:dv = e [i2]
 in
  if t<>ARRAY
  then error
  else
   let bound:addr = dv
   in
    let abound = val [n]
    in
```

```
      if abound<>bound
      then error
      else (new e [i1] ARRAY:bound:addr):s:n
 match [var i [ n ] ] a e:s:n = error
```

The Navel for array reference parameters is:

```
def match f a e:s:n =
 rule f of
 ...
 {"var" identifier "[" number "]"} ->
  rule a of
  {identifier} ->
   let t:dv = e a^identifier
   in
    if t<>"array"
    then exit (a:"not an array actual parameter")
    else
     let bound:addr = dv
     in
      let abound = val f^number
      in
       if abound<>bound
       then exit (f:"has different bounds to":a)
       else (new e f^identifier "array":bound:addr):s:n,
  exit (a:"not an array actual parameter"),
 ();
```

The Navel for statements ends with:

```
 exit (st:"not a statement");
```

as a default case.

6.9. Expression semantics

For logical expressions, the operands are evaluated and the operator applied:

```
mlogexp : <logexpression> -> E * S -> B
mlogexp [l1 and l2] e:s = (mlogexp [l1] e:s) and (mlogexp [l2] e:s)
mlogexp [l1 or l2] e:s = (mlogexp [l1] e:s) or (mlogexp [l2] e:s)
mlogexp [not l] e:s = not (mlogbase [l] e:s)
mlogexp [( l )] e:s = mlogexp [l] e:s
mlogexp [e1 "<=" e2] e:s = (mexp [e1] e:s)<=(mexp [e2] e:s)
mlogexp [e1 "<" e2] e:s = (mexp [e1] e:s)<(mexp [e2] e:s)
mlogexp [e1 "=" e2] e:s = (mexp [e1] e:s)=(mexp [e2] e:s)
mlogexp [e1 ">=" e2] e:s = (mexp [e1] e:s)>=(mexp [e2] e:s)
mlogexp [e1 ">" e2] e:s = (mexp [e1] e:s)>(mexp [e2] e:s)
mlogexp [e1 "<>" e2] e:s = (mexp [e1] e:s)<>(mexp [e2] e:s)
```

The Navel for logical expressions is:

```
def mlogexp le e:s =
 rule le of
 {logterm "and" logterm} -> (mlogexp le^logterm e:s) &
                            (mlogexp le^logterm@2 e:s),
 {logexp "or" logexp} -> (mlogexp le^logexp e:s) |
                         (mlogexp le^logexp@2 e:s),
 {"not" logbase} -> not(mlogbase le^logbase e:s),
 {"(" logexp ")"} -> mlogexp le^logexp e:s,
 {exp "<=" exp} -> (mexp le^exp e:s)<=(mexp le^exp@2 e:s),
```

```
 {exp "<" exp} -> (mexp le^exp e:s)<(mexp le^exp@2 e:s),
 {exp "=" exp} -> (mexp le^exp e:s)=(mexp le^exp@2 e:s),
 {exp ">=" exp} -> (mexp le^exp e:s)>=(mexp le^exp@2 e:s),
 {exp ">" exp} -> (mexp le^exp e:s)>(mexp le^exp@2 e:s),
 {exp "<>" exp} -> (mexp le^exp e:s)<>(mexp le^exp@2 e:s),
 exit (le:"not a logexp");
```

For arithmetic expressions, each DS fragment is followed by the equivalent Navel fragment.

For binary and unary operators, the operands are evaluated and the appropriate operator applied:

```
mexp : <expression> -> E * S -> N
mexp [e1 + e2] e:s = (mexp [e1] e:s)+(mexp [e2] e:s)
mexp [e1 - e2] e:s = (mexp [e1] e:s)-(mexp [e2] e:s)
mexp [e1 * e2] e:s = (mexp [e1] e:s)*(mexp [e2] e:s)
mexp [e1 / e2] e:s =
  let o2 = mexp [e2] e:s
  in
   if o2 = 0
   then error
   else (mexp [e1] e:s)/o2
mexp [- e] e:s = - (mexp [e] e:s)

def mexp ee e:s =
 rule ee of
 {term "+" term} -> (mexp ee^term e:s)+(mexp ee^term@2 e:s),
 {term "-" term} -> (mexp ee^term e:s)-(mexp ee^term@2 e:s),
 {factor "*" factor} -> (mexp ee^factor e:s)*(mexp ee^factor@2 e:s),
 {factor "/" factor} ->
  let o2 = mexp ee^factor@2 e:s
  in
   if o2 = 0
   then exit (ee:"divide by 0")
   else (mexp ee^factor e:s)/o2,
 {"-" base} -> - (mexp ee^base e:s),
 ...
```

For a bracketed expression , the value of the expression is found:

```
mexp ["(" exp ")"] e:s = mexp ee^exp e:s

 ...
 {"(" exp ")"} -> mexp ee^exp e:s,
 ...
```

For a number the value is found:

```
mexp [n] e:s = val [n]

val : <number> -> N
val [0] = 0
 ...
val [9] = 9
val[n d] = 10*(val [n])+(val [d])

 ...
{number} -> val ee^number,
 ...

def val n = value n 0;

def value n v =
```

```
if n=()
then v
else value (tl n) (hd n)-'0'+10*v;
```

For an identifier, if it is that of a simple variable then the value is found from the store:

```
mexp [i] e:s =
  let t:a = e [i]
  in
   if t<>VAR
   then error
   else s a

 ...
 {identifier} ->
  let t:a = e eeˆidentifier
  in
   if t<>"var"
   then exit (b:"not a simple variable but":t)
   else s a,
 ...
```

For an array element, if the identifier is that of an array then the index is checked against the bounds and the value from the corresponding store address is found:

```
mexp [i [ e ]] e:s =
  let t:i = e [i]
  in
   if t<>ARRAY
   then error
   else
    let bound:addr = i
    and subs = mexp [e] e:s
    in
     if subs<1 | subs>bound
     then error
     else s addr+subs-1

  ...
  {identifier "[" exp "]"} ->
   let t:i = e eeˆidentifier
   in
    if t<>"array"
    then exit (ee:"not an array but":t)
    else
     let bound:addr = i
     and subs = mexp eeˆexp e:s
     in
      if subs<1 | subs>bound
      then exit (ee:"subscript error")
      else s addr+subs-1,
  ...
```

The Navel for arithmetic expressions ends with the default:

```
exit (ee:"not an exp");
```

6.10. Navel interpreter

Finally, the rules and functions are sewn together to form a top level Navel interpreter:

```
def intienv n = exit ("name not declared":n);

def initstore a = exit ("address not assigned":a);

def run text ins =
 let tree:rest = program text
 in
  if tree=() | rest<>()
  then "syntax error":rest
  else
   let s:i:o = mprog tree initenv:initstore:ins:():0
   in o;
```

The Navel corresponds closely to the original DS. Note that error handling is either through explicit `exit`s or implicit in type checking for underlying Navel operations.

6.11. Interpreter performance

The Navel for the DS is 292 lines long. Loading, compiling and instantiating the global definitions for the Navel definition takes around 0.4 seconds, total elapsed time, on a DECStation 5000 under ULTRIX.

The following table shows times in seconds for compiling and running the sort example above, inputing between 5 and 80 numbers in descending order, on a DECStation 5000 under ULTRIX, with 256,000 heap cells. Compiling the example took 0.1 seconds.

```
et == total elapsed time
rt == compile & run time
gc == number of garbage collections
```

N	et	rt	gc	rt-0.4*gc	N^2/2	(rt-0.4*gc)/N^2/2
5	0.5	0.5		0.5	12.5	.04
10	1.4	1.4		1.4	50	.028
15	3.2	3.0		3.0	112.5	.026
20	6.1	5.8	1	5.4	200	.027
25	10.0	9.6	2	8.8	312.5	.028
30	14.6	14.2	2	13.4	450	.029
35	21.7	20.6	3	19.4	612.5	.031
40	29.6	28.5	4	26.9	800	.033
45	39.9	38.3	6	35.9	1012.5	.035
50	51.7	49.7	7	46.9	1250	.037
55	66.7	63.8	9	60.2	1512.5	.039
60	86.0	80.4	11	76.0	1800	.042
65	104.0	99.1	14	93.5	2112.5	.044
70	128.7	120.7	17	113.9	2450	.046
75	164.6	147.3	21	138.9	2812.5	.049
80	188.9	175.1	25	165.1	3200	.051

The first column shows the number of inputs, `N.`

The second column shows the total elapsed time perceived by the user, `et,` that is the total user and system time from `time.`

The third column shows the actual compile and run times, `rt,` including garbage collection time, that is the user time only from `time.`

The fourth column shows the number of garbage collections, `gc.`

The fifth column normalises the times for garbage collections, at 0.4 seconds per garbage collection.

Worst case naive bubble sort has O(Nˆ2/2) comparisons: the sixth column shows Nˆ2/2 for each case and the seventh column divides the normalised time by Nˆ2/2 for each case, to show an average time relative to the number of comparison.

Note that the seventh column is non-constant but close to linear: this is because the representation of stores is through the linear functional:

```
new old lv rv l = if l=lv
                  then rv
                  else old l
```

and hence the time to access an arbitrary value in the store grow linearly. Similarly, the number of garbage collections grows steadily as the store grows, freeing less and less heap space.

Overall, the Navel implementation is somewhat slow. However, these figures suggest that plausible test programs in the defined language may be tried out, with small numbers of inputs, in psychologically acceptable times.

6.12. Comparison with other systems

Lee and Pleban [87] have tested MESS and PSP on a 10 MHz Intel 8086 based IBM PC. Both MESS and PSP have been used to implement HypoPL, an imperative language with arrays, nonrecursive procedures and Pascal-like control structures. Thus HypoPL is similar to the language discussed above. They provide performance figures for an 82 line HypoPL bubble sort program. This enables comparison with the 50 line naive bubble sort program discussed above on the assumption that a DECStation 5000 is around 40 times faster than an 8086 ie 8086 ˜= 0.7 MIP; DEC ˜= 27 MIP. The HypoPL definition, in direct semantics style, is 900 lines in MESS and 1100 lines in PSP. The Navel is 292 lines long. The times for definition compilation were:

	lines	8086 secs	8086/40	DEC secs	lines/sec	other/N%
Navel	292			0.4	730	100
MESS	900	69	1.7		529	72
PSP	1100	233	5.8		190	26

The third column normalises the 8086 times to make them comparable with the DEC time. Thus, the lines per second column is for normalised times for MESS and PSP. The last column shows the ratio of PSP and MESS lines per second to Navel as percentages.

The times to compile the sort program were:

	lines	8086 secs	8086/40	DEC secs	lines/sec	other/N%
Navel	50			0.1	500	100
MESS	82	105	2.6		32	6
PSP	82	19	0.5		164	33

Again, the third column normalises the 8086 times. Thus, the lines per second column is for normalised times for MESS and PSP. The last column shows the ratio of PSP and MESS lines per second to Navel as percentages.

The times to sort 10 integers were:

	8086 secs	8086/40	DEC secs	other/N%
Navel			1.4	100
MESS	11.9	0.3		21
PSP	30.1	0.8		57

The last column show the ratio of normalised PSP and MESS times to Navel as percentages.

Here, PSP generated SECD machine code and MESS generated Scheme, both for subsequent interpretation. Machine code from MESS is significantly faster, taking 19 seconds on an 8086 to sort 250 integers.

To summarise, this crude comparison suggests that PSP is almost twice and MESS almost five times the speed of Navel in final performance, but Navel has shorter perceived response times for definition and test program compilation. It also suggests that Navel definitions are smaller than those for PSP and MESS.

Chapter 7

Navel in Navel and continuation passing semantics in Navel

7.1. Introduction

In this chapter, the implementation of Navel in Navel is considered. Here, some problems with using Navel to animate large DS definitions are addressed. Thereafter, continuation passing semantics in Navel are discussed briefly.

7.2. Navel in Navel

Core Navel with syntax extensions has been implemented in Navel. The implementation corresponds closely to the Navel DS in chapters 3 and 4 above, with a number of minor differences. In particular, a Navel in Navel program is a sequence of definitions followed by a sequence of expressions, rather than interleaved definitions and expressions as in Navel proper.

Full definitions for Navel in Navel may be found in Appendix 1: here, only salient features will be considered. These highlight a number of issues raised by the implementation of substantial DS in Navel.

7.2.1. Parsing Navel in Navel

A difficulty arises with the Navel syntax in Navel because Navel rules are based on string matching without prior lexical analysis. Recall that Navel function applications have the minimal syntax:

```
<expression1> <expression2>
```

and that:

```
<expression> => ... => <name>
```

Consider the valid Navel:

```
let f x = x in f 1
```

Because `in` cannot be distinguished as a lexeme as distinct from an arbitrary name, direct Navel rules for the Navel syntax will recognise:

```
x in f 1
```

as the function application:

```
(((x in) f) 1)
```

defining the body of `f.` Thus, the parse will fail as input appears to end without an `in` to match the `let.`

As a first attempt, this might be overcome by rules for names of the form:

```
def lexeme = {"if" | "then" | "else" ... };
def name = {lexeme fail | identifier};
```

which exclude explicitly lexemes. Here, if a `lexeme` is recognised then the subsequent `fail` ensures that the whole rule for `name` fails. However, this is too strong as it excludes valid names which start with lexemes. Thus, the rule for lexemes must be followed by a rule to recognises their followers:

```
def lexeme = {"if" | "then" | "else" ... };
def follower = {" " | "/n" | "        " | ... };
def name = {lexeme follower fail | identifier};
```

The actual rule used is:

```
def name = {{"if" | "then" | "else" | "def" | "let" | "in" | "and" |
             "lam" | "rule" | "case" | "of" | "number" | "word" |
             "identifier" | "character" | "isnumb" | "ischar" |
             "isbool" | "isfield" | "islist" | "isstring" | "isfunc" |
             "isrule" | "true" | "false" | "fail" | "hd" | "tl" |
             "char" | "not"}
            {" " | " " |  "\n" | "\;" | "}" | ")" | "]" | "," | "->" |
             ":" | "(" | "[" | "{"}
            fail | identifier};
```

A classic problem arises with right recursive rules for left associative operators in that the tree structure reflects supposed right associativity. For example:

```
def base = {identifier | number};
def term = {base "*" term | base "/" term | base};
def exp = {term "+" exp | term "-" exp | term};
{exp} "1-2+3"
```

gives:

```
[exp
 [term [base [number "1"]]]:
 "-":
 [exp
  [term [base [number "2"]]]:
  "+":
  [exp [term [base [number "3"]]]]]]
```

which is equivalent to:

```
1-(2+3)
```

instead of

```
(1-2)+3
```

In Navel, right recursion gives incorrect right associativity for logical and arithmetic expressions, and for function applications.

The Navel syntax for Navel is factored: the syntax is sufficiently convoluted for unfactored rules to take unacceptable times to parse programs. Thus, unnecessary singleton tree nodes are present in initial parse trees.

To reverse inappropriate right associativity and to strip out singleton nodes, an intermediate pass is introduced between the rules and the semantic functions. The latter transformation creates effectively abstract syntax trees from concrete syntax trees.

The implicit tree transformations between concrete and abstract syntax, discussed in chapter 3, are also carried out here. Global and local function definitions are converted to explicit associations between names and λ functions. Strings are converted to null terminated character lists.

No context sensitive checking is carried out at this stage. Such checks are already both implicit and explicit in the semantic functions but could be added here.

7.2.2. Value representation

The values in semantic domains for Navel are defined as fields:

```
boolean == [b <boolean value>]
 eg true == [b true]

number == [n <integer value>]
 eg 42 == [n 42]

character == [chr <character value>]
 eg ’a’ == [chr ’a’]

list == [cons <head>:<tail>]
() == [nil ()]
 eg 1:2:3: == [cons [n 1]:[cons [n 2]:[cons [n:3]:[nil ()]]]]
 eg "at" == [cons [chr ’a’]:[cons [chr ’t’]:[nil ()]]]

field == [field <string tag>:<value>]
eg [article "a"] == [field "article":[cons [chr ’a’]:[nil ()]]

function == [fc <state>:<function meaning>]

rule == [rc <state>:<rule meaning>]
```

Subsequently, rule matching is used to identify the types of values. The intermediate pass also converts syntax tree nodes for base values to these representations.

States are represented as functions. No special representation is used for suspensions.

7.2.3. Normal order mutual references in applicative order

As noted above, the semantic functions follow closely those of the original DS. However, `exit`s are used instead of explicit error passing. The main point of divergence from the DS is in the treatment of mutually recursive global and local definitions of functions and rules. Recall from chapters 3 and 4 that the DS equations utilise a circular definition to find explicitly the fixed point for the final state:

```
mp [d e] s = let news = (md [d] s news)
             in news:(me [e] news)
```

This construction cannot be used directly as Navel is applicative order. Instead, abstraction and forced evaluation are used to delay applicative order evaluation:

```
def mprog p s =
 rule p of
 {d e} ->
  let news x = mdecls p^d s news
  in mexp p^e (news()),
 ...
```

Note that there is no construct to return the final state, `news`, in the Navel in Navel. All definitions precede all expressions in a program. Hence there is no need to pass states from one sequence of definitions to another across intervening expression sequences.

For definitions, the DS:

```
md [def n = e] s news = let e1 = me [e] s
                         in
                          if iserror e1
                          then s
```

```
                              else
                               if isfunc e1
                               then new s [n] (makefun news (funmean e1))
                               else
                                if isrule e1
                                then new s [n] (makerule news (rulemean e1))
                                else new s [n] e1
```

becomes, in Navel in Navel:

```
def mdecls d s news =
 rule d of
 ...
 {"def" id "=" e} ->
  let e1 = mexp d^e s
  in
   if not (isfield e1)
   then new s d^id e1
   else
    rule e1 of
    {fc} -> new s d^id [fc (news()):(tl (e1^fc))],
    {rc} -> new s d^id [rc (news()):(tl (e1^rc))],
    new s d^id e1,
  ...
```

Note that the closure contents:

```
(news()):(tl (ev^fc))
```

with a lazy list in `mdecls` means that the final state from `news()` is only evaluated once for each closure. However, the same final state will be constructed repetitively in different closures. This might be avoided through a further layer of lazy list construction with an attendant further layer of de-referencing to access states in closures.

Similar constructions are used for local definitions. Thus, the DS:

```
me [let ld in e] s = let news = mld [ld] s news
                     in
                      if iserror news
                      then error
                      else me [e] news

mld [nl = e] s news = let a = me [e] s
                      in
                       if iserror a
                       then error
                       else letbind [nl] a s news

letbind [n] a s news = if isfunc a
                       then new s [n] (makefun news (funmean a))
                       else
                        if isrule a
                        then new s [n] (makerule news (rulemean a))
                        else new s [n] a
```

becomes, in Navel in Navel:

```
def mexp exp s =
 rule exp of
 ...
 {"let" d "in" e} ->
  let news x = mletdefs exp^d s news
  in mexp exp^e (news()),
```

```
 ...

def mletdefs dd s news =
 rule dd of
 {id "=" e} -> letbind [id dd^id] (mexp dd^e s) s news,
 ...

def letbind n a s news =
 rule n of
 {id} ->
  if not (isfield a)
  then new s n^id a
  else
   rule a of
   {fc} -> new s n^id [fc (news()):(tl (a^fc))],
   {rc} -> new s n^id [rc (news()):(tl (a^rc))],
  new s n^id a,
  ...
```

7.2.4. Comparing Navel in Navel with Navel in C

It is amusing though potentially misleading to compare the size of Navel in Navel with Navel in C. First of all, to compare the sizes of equivalent compilation processes:

```
 Navel in C       | lines | Navel in Navel    | lines
------------------+-------+-------------------+------
 lexical analyser |   288 |
 syntax analyser  |   510 | rules             |   52
------------------+-------+-------------------+------
                  |       | intermediate pass |  226
------------------+-------+-------------------+------
 name pass        |   335 |                   |
------------------+-------+-------------------+------
 total compilation|  1133 | total compilation |  278
```

Here, the Navel in Navel intermediate pass corresponds to Navel in C syntax analyser actions. The Navel in C name pass has no equivalent. Navel in Navel compilation is 25% of the size of Navel in C.

Next, to compare the sizes of equivalent interpretation processes:

```
 Navel in C       | lines | Navel in Navel    | lines
------------------+-------+-------------------+------
 global defs      |   213 |                   |
 local/fun bind   |   246 |                   |
 exp interpreter  |   670 | core semantics    |  329
 rule interpreter |   437 | rule semantics    |  200
------------------+-------+-------------------+------
 object manager   |   291 |                   |
------------------+-------+-------------------+------
 total interpreter|  1857 | total interpreter |  529
```

Here, sizes of the Navel in C sub-processes for global definitions, binding local definitions and function applications and the expression interpreter are shown. These are broadly equivalent to the Navel in Navel core semantics though the global definition processes include minor pretty printing details. Navel in Navel has no object management. Navel in Navel interpretation is 28% of the size of Navel in C.

Finally, the overall sizes are compared:

```
 Navel in C       | lines | Navel in Navel    | lines
------------------+-------+-------------------+------
 overall total    |  2990 | overall total     |  807
```

The Navel in Navel is 27% of Navel in C.

Of course, such comparisons should be treated cautiously. Navel's apparent compactness could be an artifact of layout styles. Alternatively, the C size might be underestimated due to the use of both a functional programming style and nested commands in expressions.

Details of some simple timing tests now follow. All times are from Navel in C on a DECStation 5000 running ULTRIX, with 256,000 heap cells.

Loading, compiling and instantiating the global definitions for Navel in Navel through Navel in C takes around 0.4 seconds elapsed time, of which around 0.3 seconds is the compilation time.

The simple imperative language from chapter 2 was extended to allow conditional and iterative statements, and blocks. For example, a program to find the product of two numbers by repeatedly adding `1` is:

```
read x;
read y;
prod:=0;
while x>0 do
begin
     c:=y;
     while c>0 do
     begin
          prod:=prod+1;
          write prod;
          c:=c-1
     end;
     x:=x-1
end
```

The Navel for this language is 74 lines long. Details are in Appendix 2.

Navel in C and Navel in Navel were timed loading, compiling and instantiating the global definitions for the Navel for this language. Navel in C took under 0.1 seconds. Navel in Navel took around 1.3 seconds.

The above program was then executed to multiply `3` by `2.` The version compiled under Navel in C took around 0.1 seconds. The version compiled under Navel in Navel took around 15.5 seconds elapsed time, of which around 14.8 seconds was processing time, including 3 garbage collections.

These crude tests suggest that Navel in Navel is 2 orders of magnitude slower than Navel in C.

Appendix 1 includes further examples of Navel programs and their results, running on Navel in Navel in turn running on Navel in C.

7.3. Continuations in Navel

The language presented in the previous chapter was defined in a direct DS style. Standard DS is based on a continuation passing model [57] where the meaning of each construct involves making explicit the meaning of the next construct. Continuations are particularly useful for defining languages with explicit breaks in the control flow, for example through jumps or break constructs for iteration. They also simplify greatly error handling: after an error, evaluation terminates directly instead of proceeding with explicit handling of error objects as used in the semantics of Navel in chapters 3 and 4. However, continuations also complicate language definitions where there are no explicit breaks in control flow: hence, they were not used for Navel or in the previous chapter.

The following language is reminiscent of a BASIC subset with jumps. Here, continuations are used to explicate control transfers. The language enables programs like the following to calculate the factorial of an integer, say `3:`

```
  LET F = 1
  LET N = 3
1 IF N=0 THEN 2
  LET F = F*I
```

```
  LET N = N-1
  GOTO 1
2 LET FAC = F
```

The syntax of the language is:

```
<statements> ::= <statement> <statements> | <statement>
<statement> ::= <label> <command> | <command>
<command> ::= LET <name> = <expression> |
              GOTO <label> |
              IF <logical> THEN <label>
<label> ::= <number>
<logical> ::= <expression> <comp> <expression>
<comp> ::= < | <= | = | >= | > | <>
<expression> := <term> + <term> | <term> - <term> | <term>
<term> ::= <factor> * <factor> |<factor> / <factor> | <factor>
<factor> ::= - <base> | <base>
<base> ::= <number> | <name> | ( <expression> )
<arith> ::= + | - | * | /
```

To simplify presentation, arithmetic expressions must be bracketed strictly to avoid explicit right associative to left associative tree transformation. Furthermore, full details of `<name>` and `<number>` have been omitted.

In Navel this may be rendered as:

```
def statements = {statement statements | (statement)};
def statement = {number command | (command)};
def command = {(assign) | (sif) | (goto)};
def assign = { "LET" identifier "=" exp};
def sif = {"IF" logexp "THEN" number};
def goto = {"GOTO" number};
def logexp = {exp (comp) exp};
def comp = {"=" | "<>" | "<=" | "<" | ">=" | ">"};
def exp = {term "+" term | term "-" term | (term)};
def term = {base "*" base | base "/" base | (base)};
def base = {identifier | number | "(" exp ")"};
```

The abstract syntax for this language is:

```
s ∈ <statement>
l ∈ <logical>
e ∈ <expression>
n ∈ <name>
i ∈ <number>
cop ∈ <comp>
aop ∈ <arith>

s -> s s | i s | LET n = e | GOTO i | IF l THEN i
e -> - e | e aop e | i | n
l -> e cop e
cop -> < | <= | = | >= | > | <>
aop -> + | - | * | /
i -> d i | d
```

Assuming the integer domain `N` and boolean domain `B`, the domain of stores maps names to integers, to model variables:

```
S == stores == <name> -> N
```

and the domain of environments maps labels to continuations:

```
E == environments == <label> -> C
```

The domain of continuations maps states to states:

```
C == continuations == S -> S
```

In Navel, `N` and `B` are integers and booleans respectively, and stores, environments and continuations are all functions.

The environment must hold all the label/continuation associations before evaluation of the program. The environment is constructed by:

```
mprog : <statements> -> S
mprog [s] = let finalenv = getenv [s] initenv finalenv λ c.c
            in mstate [s] finalenv λ c.c initstore
```

Note that the final environment `finalenv` is passed as a parameter to the function that finds it. In Navel, this becomes:

```
def initenv num = exit (num:"not a label")

def initstore id = exit (id:"not assigned");

def mprog tree =
 let ee l = getenv tree initenv ee lam c.c
 in mstate tree ee lam c.c initstore;
```

Note that in `mprog` the final environment variable `ee` is set to a function which returns the environment rather than to the environment itself. As in Navel in Navel above, the recursive device of defining directly a final environment in terms of itself will not work because Navel is applicative ordered. Thus, for the direct translation:

```
let ee = getenv tree initstore ee lam c.c
in ...
```

the right hand side call to `getenv` is evaluated and the argument `ee` has no value.

Hence, a layer of abstraction is introduced. As will be seen below, the environment is found by applying this final environment function to a dummy argument. While this overcomes applicative order it results in the classic normal order problem of repeated construction of the final environment whenever a jump is encountered.

The formation of the environment depends on the structure of the program. At each stage, an environment so far and the final environment are required:

```
getenv : <statement> -> E -> E -> C -> E
```

For a labelled single statement, the environment so far is updated with a continuation formed from that statement and the final environment, so that the statement may be a jump to a forward or backward label:

```
getenv [i s] env finalenv cont = new env [i] (mstate [s] finalenv cont)
```

For a statement sequence, the environment from the rest of the statements is found using the environment from the first. Note that the continuation for the first statement is formed from the rest of the statements:

```
getenv [s1 s2] env finalenv cont =
 getenv [s2] (getenv [s1] env finalenv
              (mstate [s2] finalenv cont)) finalenv cont
```

For an unlabelled statement, the environment is returned:

```
getenv [s] env finalenv cont = env
```

The Navel for environment construction is:

```
def getenv tree e ee c =
  rule tree of
  {number command} -> new e tree^number (mstate tree^command ee c),
  {statement statements} -> getenv tree^statements
                             (getenv tree^statement e ee
                              (mstate tree^statements ee c)) ee c,
  e;
```

Next the meaning of statements is presented:

```
mstate : <statement> -> E -> C -> S -> S
```

For a sequence of statements, the first is carried out using the second as continuation:

```
mstate [s1 s2] e c s = mstate [s1] e (mstate [s2] e c) s
```

For a labelled command, the command is carried out:

```
mstate [i s] e c s = mstate [s] e c s
```

For an assignment, the continuation is carried out using the store from the assignment:

```
mstate [LET n = e] e c s = c (new s [n] (mexp [e] s))
```

For a jump, the continuation is found in the environment and carried out with the current store:

```
mstate [GOTO i] e c s = (e [i]) s
```

For a conditional jump, if the condition is true then the continuation is found in the environment. Otherwise, the continuation is the initial continuation:

```
mstate [IF l THEN i] e c s = if mlog [l] s
                             then e [i] s
                             else c s
```

The Navel for statements is:

```
def mstate tree e c s =
 rule tree of
  {statement statements} ->
   mstate tree^statement e (mstate tree^statements e c) s,
  {number command} -> mstate tree^command e c s,
  {"LET" identifier "=" exp} ->
   c (new s tree^identifier (mexp tree^exp s)),
  {"GOTO" number} -> e() tree^number s,
  {"IF" logexp "THEN" number} ->
   if mexp tree^logexp s
   then e() tree^number s
   else c s,
  exit (tree:"not a statement");
```

Note the application of the environment to a dummy argument in the cases for `IF` and `GOTO.`

The semantic equations for logical and arithmetic expressions are not shown here as they are similar to and simpler than those for the language in the previous section.

The Navel is then sewn together as:

```
def run text =
 let tree:rest = statements text
 in
  if tree=() | rest<>()
```

```
    then "syntax error":rest
    else mprog tree;
```

Consider running the above example:

```
def fac2 = "  LET F = 1
              LET N = 3
            1 IF N=0 THEN 2
              LET F = F*N
              LET N = N-1
              GOTO 1
            2 LET FAC = F";
run fac2
```

The parse tree, labelled at salient points, is:

```
TREE == [statement "LET":[identifier "n"]:"=":[exp [number "3"]]]:
         [statements
          [statement "LET":[identifier "f"]:"=":[exp [number "1"]]]:
          [statements
           [statement [number "1"]:
                      [command
CONT1 ==               "IF":[logexp [exp [identifier "n"]]:"=":
                                    [exp [number "0"]]]:
                       "THEN":[number "2"]]]:
           [statements
BODY ==     [statement "LET":[identifier "f"]:"=":
                        [exp [base [identifier "f"]]:"*":
                              [base [identifier "n"]]]]:
            [statements
             [statement "LET":[identifier "n"]:"=":
                         [exp [term [identifier "n"]]:"-":
                               [term [number "1"]]]]:
             [statements
              [statement "GOTO":[number "1"]]:
               [statements [number "2"]:
                           [command
CONT2 ==                    "LET":[identifier "fac"]:"="
                              [exp [identifier "f"]]]]]]]]]

TREE == whole tree
CONT1 == tree for: IF n=0 THEN 2
BODY == tree for code after IF
CONT2 == tree for: LET fac = f
```

The final environment, `FINAL,` results from:

```
FINAL == lam l.(getenv TREE initenv FINAL lam c.c)
```

The body of this function is equivalent to:

```
new
 (new
   initenv
   1
   (mstate CONT1 FINAL (mstate BODY FINAL lam c.c)))
 2
 (mstate CONT2 FINAL lam c.c)
```

This environment results from two nested applications of `new.` Label `2` has as continuation an application of `mstate` to `LET fac = f` in the final environment, with `lam c.c` as continuation. Hence, jumps to label `2` will invoke evaluation of `LET fac = f` in the current state and terminate.

Similarly, label `1` has as continuation an application of `mstate` to `IF N=0 THEN 2` in the final environment. The continuation is formed by applying `mstate` to the statements following this command, in the final environment, with `lam c.c` as continuation. Thus, jumps to label `1` will invoke the statement sequence starting with the `IF.`

The bottom level environment entry is the initial environment:

```
lam num.(exit num:"not a label")
```

This returns an error in the event of a jump to an undefined label.

Chapter 8

Rule generalisation and context sensitive parsing in Navel

8.1. Introduction

Where context free syntax addresses the presence or absence of constructs, context sensitive syntax enables the identification of more specific construct properties. Although context sensitivity is deemed irrelevant for DS, it is essential for practical language implementations so that appropriate checks may be carried out once at compile time rather than repetitively at run time. A number of notations have been proposed for defining context sensitive syntax but their addition to language implementation systems adds extra stages and decreases system flexibility. In Navel, grammar rules are full values. Context sensitive aspects of syntax may be expressed through abstraction over grammar rules without additional notation.

This chapter considers how abstraction over Navel rules enables the specification and processing of context sensitive syntax. First of all, context free generalisations of Navel rules are discussed. Next, the 2-level grammar, attribute grammar, and dynamic syntax and grammar form approaches to context sensitivity are reviewed briefly. Finally, there is discussion, with simple examples, of the use of rule abstraction in Navel for context sensitive checking in a style analogous to dynamic syntax.

8.2. Generalising context free grammars

In chapter 4 it was noted that Navel rules are full values and may be abstracted over. This enables the construction of higher order rules to generalise context free constructs.

For example, consider:

```
def optname = {name | {}};
def optword = {word | {}};
```

which both recognise zero or one occurrences of a construct. These might be generalised by:

```
def zeroorone base = {base | {}};
```

However, the definition:

```
def optword = zeroorone word;
```

will build a parse tree with `base` as the tag for a single occurrence. Navel is call by value: in

```
zeroorone word
```

`word` is replaced by the associated rule value. Furthermore, `base` is within a rule and so it is used as a tag:

```
optword "fish" ==>
[base "fish"]]:
```

In:

```
def zeroorone base = {(base) | {}};
```

however, `base` is replaced by its value and so is not used as a tag. If `base` is associated subsequently with a rule for a non-terminal then that non-terminal will be used as the tag:

```
def optword = zeroorone {word};
optword "fish" ==>
[word "fish"]
```

Consider:

```
def words = {word {words | {}}};
def numbers = {number {numbers | {}}};
```

Both rules recognise one or more occurrences of a base construct and have the common structure:

```
def oneormore base recurse = {(base) (zeroorone recurse)};
```

For example:

```
def words = oneormore {word} {words};
{words} "ape bat cat" ==>
[words [word "ape"]:[words [word "bat"]:[words [word "cat"]]]]
```

A common form of rule is to recognise recursive sequences of constructs separated by infix operators, as for example in the factored rules for `term` and `exp` in:

```
def base = {number | word | "(" exp ")"};
def term = {base {{"*" | "-"} term | {}}};
def exp = {term {{"*" | "/"} exp | {}}};
```

Such rules may be generalised as:

```
def infix base op recurse = {(base) (zeroorone {(op) (recurse)})};
```

Thus, the previous example may be written as:

```
def base = {number | word | "(" exp ")"};
def term = infix {base} {"*" | "/"} {term};
def exp = infix {term} {"+" | "-"} {exp};
```

However, such rule abstraction does not alter the context free nature of Navel rules.

8.3. Approaches to context sensitivity

In chapter 2, it was noted that while context sensitivity is strictly of no concern for DS, some aspects are implicit in DS. Furthermore, it was noted that practical systems based on DS benefit from some provision of static context sensitive checks.

Context free grammars correspond to Chomsky Type 2 rules:

```
A -> α - α ∈ {terminals + non-terminals}*
```

When an A has been recognised, it is only known that the sentential form α has been found but no information about its composition beyond its top level structure is available. Indeed, if there are several cases for A then even the top level structure may not be known.

Context sensitive grammars correspond to Chomsky Type 1 rules:

```
α A ξ -> α β ξ - A ∈ non-terminals
               - α, β, ξ ∈ {terminals + non-terminals}*
```

Here, when an A has been recognised it is known that a β has been found in the context α `...` ξ. Thus, information about what has already been found beyond the top level structure may be available. However, the amount of information is limited. Context sensitive grammars are recognised by linear bounded automata, which, as their name suggests, can only hold finite amounts of contextual information. An equivalent definition of context sensitive rules is:

```
α -> β  - α, β ∈ {terminals + non-terminals}*
where |α| <= |β|
```

The size constraint indicates that the amount of information available after recognition can be no more than that which has been recognised. In effect, this restricts context information to a finite lexicon.

Chomsky Type 1 grammars are extremely unwieldy for practical use. A number of notations have been proposed for defining context sensitive aspects of syntax. Below are brief presentations of 2-level Grammars, attribute grammars, and grammar forms and dynamic syntax. These are followed by discussion of context sensitive parsing in Navel, which is closely related to grammar forms and dynamic syntax.

8.3.1. 2-level grammars

This brief account is after Cleaveland and Uzgalis [32].

2-level grammars are a powerful generalisation of context free grammars, based on two levels of grammar definition. They are also known as W-grammars after their inventor van Wijngaarden.

2-level grammars have their own terminology. Protonotions are atomic values, denoted by sequences of lower case letters, which act as components of context free terminal and non-terminal symbols. Metanotions are variables, denoted by sequences of upper case letters. Hypernotions are sequences of protonotions and metanotions. These generalise entire non-terminal and terminal symbols because metanotions are associated with hypernotions. Hypernotions ending with the protonotion "symbol" act as terminals.

A metarule associates a metanotion with hypernotion options. Meta rules are often recursive enabling the production of an infinity of arbitrary length hypernotions, termed terminal metaproductions.

A hyperrule associates a hypernotion with hyperalternative options, where a hyperalternative is a sequence of hypernotions. The presence of metanotions means that left hand side hypernotions generalises non-terminal names and right hand side hyperalternatives generalise terminal and non-terminal sequences.

In a hyperrule, all occurrences of the same metanotion on both the left and right hand sides may be replaced with the same terminal metaproduction, generated from a metarule starting with that metanotion as sentence symbol. This is termed consistent substitution. This may lead to new non-terminals as well as to new sentential forms. Repeated consistent substitution may generate ultimately context free rules when no metanotions remain.

Consistent substitution enables context sensitivity as the presence of metanotions enables information to be transmitted through hyperrules. This leads to hyperrule modification reflecting specific context conditions. However, 2-level grammars are actually equivalent to Chomsky Type 0 languages as there is no restriction on the size of hyperrules after substitution. Thus they have the same expressive power as Turing machines, the λ calculus, recursive function theory and most programming languages, and may also be used to define semantics.

Substantial 2-level grammars are somewhat complex. Following their use in the definition of ALGOL 68 [159] they have fallen into disuse for the formal definition of languages. However, Edupuganty and Bryant [42] suggest that 2-level grammars may be viewed as a functional programming language. They compare 2-level grammar, LISP and Prolog solutions for typical functional programming examples, and argue that there is a close correspondence between the 2-level grammars and the LISP, and hence to a functional style. They also claim, less convincingly, that 2-level grammars are easier to read because of their natural language style.

Navel rule generalisation enables directly a very weak form of consistent substitution on the right hand side of rules. All occurrences of a bound variable in a rule which is the body of a function are replaced effectively with the same argument value when that function is called. However, Navel does not allow identifier abstraction and so there is no equivalent of non-terminal generalisation. This may be achieved tortuously by constructing explicitly strings representing local rule definitions which are then `eval`ed.

8.3.2. Attribute grammars

This brief account is after Watt and Madsen [155].

Attribute grammars were first proposed by Knuth and are based on extensions to context free grammars through the association of attributes with salient terminal and non-terminal symbols. Actions then specify how attributes are processed when the salient symbols are reached during parsing. Inherited attributes, which pass from the left hand side to the right hand side of rules, are distinguished from synthesised attributes, which pass from the right hand side to the left hand side. Thus, synthesised attributes are used to pass information back from sentential form recognition to the associated non-terminal and inherited attributes are used to pass information from a non-terminal to the corresponding sentential form. Together, inherited and synthesised attributes enable the transmission of information from one part of a sentential form to another.

In an attribute grammar, salient symbols are followed by attribute variables, which are preceded by operators to indicate whether an attribute is inherited or synthesised. Additionally, each rule may have associated attribute actions which are written in some notation to express the manipulation of attribute values. For example, McGettrick [95] uses actions reminiscent of Prolog goals, with functors for independently defined rules applied to source and destination attribute variables, and also English descriptions. Aho and Ullman [3] use assignment statements and imperative boolean and arithmetic expressions.

Attribute grammars are used widely in the definition of languages, for example for Pascal [157]. They are also used widely in language definition based transducer systems, for example GAG [80], HLP [119], the Synthesiser Generator [124] and Coco [121]. Attribute grammar use in the PSG and Paulson's systems for DS was noted in chapter 2.

Johnsson [73] has proposed the use of attribute grammars as a functional programming technique. He suggests additional notation for Lazy ML and discusses how it may be transformed into pure LML. Frost [49] also suggests the use of functional programming to support attribute grammars as a programming technique. He introduces four new combinators into a lazy functional language notation and gives Miranda equivalents.

Attribute techniques may be employed in Navel by separating out the components of rules and using local definitions to sequence rule component application and attribute actions. A variant of this is discussed below. Lin [89] proposes extending Navel rules to attribute grammars.

8.3.3. Dynamic syntax and grammar forms

Dynamic syntax was proposed by Hanford and C. Jones [62] in 1971. Their particular concern was with language constructs which imply context constraints that cannot be captured by context free rules. For example languages with declarations have requirements to do with the contexts in which declared entities can subsequently appear. They proposed that the context free parse of the start of a symbol sequence should guide the generation of context free rules for parsing the rest of a symbol sequence, containing constraints determined by that initial parse. For example, parsing declarations would result in the construction of new rules to constrain assignments and expressions to refer to identifiers of appropriate types.

Their approach was to recast grammars as functions, which they termed dynamic production systems, using a variant of Landin's applicative expression notation used in defining the SECD machine. Typically, functions are parameterised constructed objects containing sentential forms and various sets of rules. A formal parameter may and usually will appear in the sentential forms and the rules. When the function is applied to a rule, the formal parameter is associated with that rule. When the sentential form is matched and the rule succeeds, all occurrences of the formal parameter associated with the argument rule are replaced by the recognised text. Thus, sentential form use instantiates the constructed rules. Subsequently, those rules are selected explicitly and combined with others to build new constructed objects.

They give the following example of a simple declaration:

```
real-type-declaration -> <text:"real"ˆx,
                              rule:real-simple-variable -> x>
where x -> identifier
```

The intention is that `x` is associated with `identifier.` When:

```
"real"ˆx
```

is matched against say:

```
real size
```

`real` matches `real`, `identifier` matches `size` and so `x` is bound to `size` giving:

```
rule: real-simple-variable -> "size"
```

Thus, `real-simple-variable` will recognise only `size` as valid in subsequent constructs which employ it.

The authors also discuss abstraction over the application of the sentential form to delay its use. This enables the definition of what are effectively two pass rules for handling cases where constructs may be used before they are defined, for example for forward jumps.

This work was carried out during an investigation of constrained generation of test cases from language grammars: the authors do not indicate if it was used for parsing.

Ginsburg and Rounds [54] build on the concepts of dynamic syntax to develop grammar forms. These are partially defined context free rules with associated functions for what are effectively constrained rules. They describe a two stage algorithm for grammar forms. First of all, the source text is scanned using language dependent control rules to build lists of constructs, typically declared identifiers and labels, which determine contexts. The second stage then removes duplications and instantiates the function for the constrained rules from the lists. The function is used to augment effectively the original partial grammar with the new constrained rules during the full parse. They also discuss techniques for handling local constraints, for example for local declarations within blocks.

Implementing grammar forms depends heavily on the use of operational control rules in extracting context specific rules. It also involves two passes through the source text.

Dynamic syntax is closely related to the approach to context sensitive processing in Navel described, but not theorised, in the next section.

8.4. Context sensitivity in Navel

The approach illustrated below is most akin to dynamic syntax but has resonances with 2-level and attribute grammars. The behaviour of a Navel rule may be summarised as:

```
apply rule to string
if match succeeds then
 return tree and rest of string
else
 return failure indicator and whole string
```

Using the results of partial parsing to guide rule construction for subsequent parsing involves making these stages explicit. In general, the approach is to break a rule down into its components and then:

```
apply initial rule components to string
if match succeeds then
 construct new rule(s) for subsequent rule components
 apply new rule(s) to rest of string
 construct overall tree
 return tree and remains of string
else
 return failure indicator and original string
```

Like dynamic syntax and 2-level grammars, rules applied subsequently are determined by the effects of the rules applied so far. Like dynamic syntax and attribute grammars, rule application and subsequent analysis of effects are separated out.

This approach is now illustrated through three examples. First of all the problem of recognising $a^n b^n c^n$ for arbitrary `n` is considered. Next, checking assignment of variables before use in expressions for a small imperative language is discussed. Finally, declaration and type checks for a slightly larger imperative language are presented.

8.5. Example - $a^n b^n c^n$

Consider the classic context sensitive problem of recognising:

$a^n b^n c^n$

n is unknown until all the as have been recognised. Thus, rules to recognise n bs and n cs might be constructed step by step during the as recognition. Every time an a is recognised a new layer to recognise additional bs and cs is added to the corresponding rules:

```
def an a b c atree bn cn text =
 let t1:s1 = {a} text
 in
  if t1<>()
  then an a b c [an t1:atree] {b bn} {c cn} s1
  else
   let t2:s2 = {bn cn} s1
   in (atree:t2):s2;
```

an is to recognise some number of as followed by the same number of bs and cs. atree is the tree so far for the a sequence. bn and cn are the rules for recognising as many bs and cs as there are as in that tree.

An attempt is made to match one a. If it succeeds then a layer of tree is built, the rules for bs and cs are extended, and the search for as continues. Otherwise, the requisite number of bs and cs are checked and the final tree is built. Initially:

```
def anbncn a b c = an a b c [an ()] {} {};
```

Here the rules for bs and cs are empty, and the tree for as reflects that none have been found, as if the empty rule had succeeded.

Consider:

```
def abc = anbncn {"a"} {"b"} {"c"};
abc "aabbcc" ==>

an {"a"} {"b"} {"c"} [an ()] {} {} "aabbcc"
```

First of all, one letter a is recognised by:

```
{"a"}
```

so the recursive call is:

```
an {"a"} {"b"} {"c"} [an [a "a"]:[an ()]]
                     {b bn} {c cn} "abbcc"
where b == {"b"}
      bn == {}
      c == {"c"}
      cn == {}
```

Once again, a letter a is recognised so the recursive call is:

```
an {"a"} {"b"} {"c"} [an [a "a"]:[an [a "a"]:[an ()]]]
                     {b bn} {c cn} "bbcc"
where b == {"b"}
      bn == {b bn}
      where b == {"b"}
            bn == {}
      c == {"c"}
      cn == {c cn}
      where c == {"c"}
```

```
            cn == {}
```

There are no more `a`s so:

```
{bn cn}
where bn = {b bn}
      where b == {"b"}
            bn == {b bn}
            where b == {"b"}
                  bn == {}
      cn == {c cn}
      where c == {"c"}
            cn == {c cn}
            where c == {"c"}
                  cn == {}
```

is applied to:

```
"bbcc"
```

`bn` recognises 2 `b`s and `cn` recognises 2 `c`s returning the tree:

```
[bn [b "b"]:[bn [b "b"]:[bn ()]]]:[cn [c "c"]:[cn [c "c"]:[cn ()]]]
```

which, joined to the tree for the `a`s gives:

```
[an [a "a"]:[an [a "a"]:[an ()]]]:
 [bn [b "b"]:[bn [b "b"]:[bn ()]]]:
  [cn [c "c"]:[cn [c "c"]:[cn ()]]]
```

Griswold and Griswold [59] present a related solution in Icon to this specific problem.

8.6. Example - assignment before use

Consider the problem of checking variable assignment before use in a simple imperative language with assignment and output commands, and a function which returns the next value from the input:

```
def comm = {word ":=" exp | "output" exp};
def exp = {base "+" exp | base};
def base = {"0" | "1" | "read"};
```

Initially, there are no assigned variables:

```
def assigned = fail;
```

Every time an assignment is encountered, the rule for assigned variables is extended to recognise the assigned variable if it has not already been assigned. The function:

```
def new id old = {(id) | (old)};
```

is used to extend rules. When passed an identifier, `id,` and the old rule for assigned variables, `old,` it returns a rule which tries to recognise that identifier before using the old assigned variable rule.

When an assignment has been recognised, the left hand side identifier is checked to see if it is recognised by the rule for assigned identifiers. If it is then it has been assigned already and the current rules for assigned identifiers and commands are returned. Otherwise, the identifier is added to the rule for assigned identifiers which is then used to build a new rule for recognising commands:

```
 def check id:assigned:comm =
  let t:r = assigned id
  in
```

```
   if t<>()
   then assigned:comm
   else
    let newassigned = new id assigned
    in newassigned:(makecomm newassigned);

def makecomm assigned =
 let base = {assigned | "0" | "1" | "read"}
 and exp = {base "+" exp | base}
 and comm = {word ":=" exp | "output" exp}
 in {comm};
```

Note that the rule for a `base` now includes a rule for assigned variables.

Command sequences are recognised by a function. First of all it parses the string to identify up the first command. If it is the only command in the sequence then the tree for it is returned. Otherwise, if it is an assignment then new command and assigned identifier rules are constructed. The commands rule is called recursively to recognise the rest of the sequence and a final tree is constructed and returned:

```
def comms assigned:comm text =
 let t1:r1 = comm text
 in
  if t1=()
  then exit ("bad command":text)
  else
   if r1=()
   then [comms t1]:
   else
    let t2:r2 =
     rule t1^comm of
     {word ":=" exp} -> comms (check (t1^comm^word):assigned:comm) r1,
     comms assigned:comm r1
    in
     if t2=()
     then exit ("bad commands":t1)
     else [comms t1:t2]:r2;
```

The rule for a program is constructed from the rule for commands with the initial assigned identifier and command rules:

```
def program = comms assigned:{comm};
```

Consider:

```
program "a:=0 b:=a c:=a+b"
```

First of all, `a:=0` is recognised by the initial rule for `comm` and the tree:

```
[word "a"]:":=":[exp [base "0"]]
```

is constructed. The rest of the string has still to be parsed. `check` is called to see whether or not `a` is a known assigned variable. It is not so a new rule:

```
{"a" | fail}
```

is built for assigned variables and a new rule for `comm` is constructed within which only the variable `a` may appear in expressions on the right of assignments.

This new rule is then used to parse `b := a c:=a+b`. The tree:

```
[word "b"]:":=":[exp [base [assigned "a"]]]
```

is constructed as a is valid on the right of the assignment. The rest of the string has still to be parsed. check is called to see whether or not b is a known assigned variable. It is not so a new rule:

```
{"b" | {"a" | fail}}
```

is built for assigned variables and a new rule for comm is constructed within which both b and a are valid names on the left of assignments.

The string c:=a+b is then parsed with the new rule. The tree:

```
[word "c"]":=":[exp [base [assigned "a"]]:"+":[base [assigned "b"]]]
```

is returned as both a and b have been assigned. The string is now empty and so the final tree:

```
[comms
 [comm [word "a"]:":=":[exp [base "0"]]]:
 [comms
  [comm [word "b"]:":=":[exp [base [assigned "a"]]]]:
  [comms
   [comm [word "c"]":=":[exp [base [assigned "a"]]:"+":
                              [base [assigned "b"]]]]]]]
```

is constructed.

8.7. Example - declaration and type checks

Consider the following rules which describe the syntax of a language fragment with declarations, assignment and integer arithmetic:

```
def program = {declarations statements};
def declarations = {declaration {declarations | {}}};
def declaration = {{"long" | "short"} word};
def statements = {statement {statements | {}}};
def statement = {word ":=" expression};
def expression = {base {"+" base | {}}};
def base = {word | number | "(" expression ")"};
```

Suppose that it is required that no identifier may appear in more than one declaration and that an identifier must appear in a declaration before it appears in a statement. It is also intended that the language should enable long and short precision arithmetic, that variables be typed according to the precision of the values they may hold and that long values may not be coerced to short. Thus, if the identifier on the left of a statement appears in a declaration preceded by short then all identifiers in the expression on the right of the statement must also appear in declarations preceded by short.

As with the example in chapter 6, environment use in DS may encompass name checking. If a name in a command has not appeared in a preceding declaration then the environment will not have an address for it. Similarly, if a name in a declaration has appeared in a preceding declaration then the environment will already have an associated address for it. The environment function may also be extended to enable the recording and checking of type information for names.

Name checking may be transferred to the syntax with the construction of appropriate rules from declarations. For the above language, separate rules are used to check whether or not identifiers have been declared as short or long. Initially, nothing has been declared and so these rules should not match anything:

```
def init_short = fail;
def init_long = fail;
```

As before, rules are extended with the function:

```
def new old name = {(name) | (old)};
```

When a declaration is found, the rules are used to check the identifier for re-declaration and the appropriate rule is then extended with the identifier:

```
def decls string:long:short =
 let tree:rest = {{"short" | "long"} word} string
 in
  if tree=()
  then long:short:string
  else
   if hd(long tree^word)<>() | hd(short tree^word)<>()
   then exit (tree^word:"declared already")
   else
    rule tree of
    {"long" word} -> decls rest:(new long tree^word):short,
    {"short" word} -> decls rest:long:(new short tree^word),
    ();
```

Separate rules are required to recognise assignment to variables declared as long and short. They are built from a schema which defines local rules for recognising expressions:

```
def makeassign nametype basetype =
 let base = {(basetype) | number | "(" expression ")"}
 and expression = {base {"+" base | {}}}
 in {(nametype) ":=" expression};
```

Programs are parsed by a function which uses `decls` to parse the declarations and construct the rules for longs and shorts. They are then used with the assignment schema to construct the assignment rules which are incorporated into local rules for recognising statements:

```
def program string =
 let long:short:rest = decls string:init_long:init_short
 in
  let lassign = makeassign {long} {long | short}
  and sassign = makeassign {short} {short}
  and statement = {lassign | sassign}
  and statements = {statement {statements | {}}}
  in statements rest;
```

The rule associated with `lassign` will recognise as valid assignments to `long` identifiers with `long` or `short` identifiers in the right hand side expressions. The rule associated with `sassign` will recognise as valid assignments to `short` identifiers with only `short` identifiers in the right hand side expressions.

Consider:

```
program "short a long b short c c:=a+a b:=c+b"
```

`decls` recognises `short a`, checks that `a` has not been declared already and adds it to the rule for recognising short variables:

```
{"a" | fail}
```

Next, `decls` recognises `long b`, checks that `b` has not been declared already and adds it to the rule for recognising long variables:

```
{"b" | fail}
```

`decls` now recognises `short c`, checks that `c` has not been declared already and adds it to the rule for recognising short variables:

```
{"c" | {"a"  fail}}
```

There are no more declarations so a new set of rules is built for commands. For long assignment, only `b` may appear on the left but either `b`, `a` or `c` may appear on the right. For short assignment, either `a` or `c` may appear on the left or right. Thus, the final tree for:

```
c:=a+a b:=c+b
```

is:

```
[statement
 [sassign
  [short "c"]:
  ":=":
  [exp [base [short "a"]]:"+":[base [short "a"]]]]]:
 [statements
  [statement
   [lassign
    [long "b"]:
    ":=":
    [exp [base [short "c"]]:"+":[base [long "b"]]]]]]
```

Note that no tree for declarations has been constructed, to simplify presentation.

8.8. Discussion

In the DS in previous chapters, an update function:

```
new old lv rv l = if l=lv
                  then rv
                  else old l;
```

was used to extend store and environment functions. Here:

```
 def new id old = {(id) | (old)};
```

is used in the same way to extend rules. `lv` and `old` in the update function correspond to `id` and `old` in the rule extender. The return of `rv` for `lv` in the update function corresponds to the return of a tree for the successful recognition of `id.` There is no equivalent for `l` because rules have implicit formal parameters. Here it may be made explicit by:

```
def new id old l = {(id) | (old)} l;
```

The use of the update and rule extender functions are further analogous. In the first example language above, assignments both introduce and update name/value associations. In a DS for this language, for a name in an expression, the store function would be used to find the corresponding value. If the required name had not been introduced by a previous assignment then the store function would return an error. Thus, the use of a store function encompasses implicitly the checking of this requirement.

This approach has not yet been applied to more substantial languages. In principle it should extend to a variety of constructs. For example, after array declarations, the new rules would contain checks for consistency between the number of declared and applied subscripts, for appropriate subscript expressions and for the use of array elements in type specific contexts. After record declarations, the new rules would check for type consistency for record element access and use. After procedure or function declarations, the new rules would check for consistency in number and type between formal and actual parameters. For block structure with local declarations, the scope of new rules after declarations would be restricted to the parsing of the block body.

Forward references are somewhat harder to encompass. Perhaps a list could be kept of unknown forward references. When a reference was finally satisfied the item referred to could be added to a rule for known forward references. After parsing, if any of the unknown references could not be parsed by the known reference rule then they were not satisfied. It would be nice to do this using only rules, for example by maintaining a rule for unknown references. However, meta-syntactic operators would be required to intersect rules to determine if the unknown reference rule contained additional terminals to those in the known reference rule.

It is not clear how well this approach scales up. It becomes hard to maintain the illusion that parsing is being carried out by rules with a little additional Navel augmentation when the Navel augmentation is making explicit the bulk of the parse actions. However, this criticism applies to dynamic syntax in general rather than solely to Navel.

Christiansen [30] pursued a related approach to dynamic syntax contemporaneously with this work. His generative grammars are based on generalised attribute grammars where derivation relations depend on special inherited attributes. He has then applied this approach to Prolog [29] to enable context sensitive checks through the generation of new rules during parsing.

An inverse approach can be used to implement Jones and Hanford's original intention of generating constrained texts. For example, Michaelson [98] discusses the generation of limericks from rules where the appropriate lines rhyme and consistent gender pronouns are used.

Chapter 9

Conclusions

9.1. Introduction

In this conclusion, the dissertation is reviewed chapter by chapter. Avenues for future research into formal language definition based systems are then proposed.

9.2. Review

9.2.1. Chapter 1 - Introduction

Chapter 1 provided an overview of this research. The need for formality in Computing and reasons for its slow adoption were considered. In particular, the absence of appropriate tools was identified as a major hindrance to the wider use of formal definitions for language development. The overall objective of this work was then summarised as the investigation of language implementation through the interpretation of formal definitions, as programs in a unitary notation, based on a functional language with integral extensions for syntactic processing.

9.2.2. Chapter 2 - Denotational semantics and its implementation

Chapter 2 introduced DS and contrasted it with the operational and axiomatic approaches. Objections by DS proponents to the direct implementation of formal definitions because of the primacy of mathematics as an abstract practice were presented. Pragmatic rejoinders based on the close correspondence between DS and OS, and between DS and programming, were then considered, which suggest readings of DS definitions as interpreters.

These sections are of necessity somewhat rhetorical. While it would be interesting to investigate the philosophical roots of the dispute between "purists" and "pragmatists", this would involve addressing fundamental questions as to the nature of mathematics; well beyond the scope of this thesis. The reader may have gleaned that the author’s sympathies lie with the pragmatists. A fuller reply to the purists might be based on the following sketch:

> Mathematics is a material, social practice. Mathematical systems are based on rule governed symbol manipulation with physical representations, for example as marks on paper, charges in computer memories or electro-chemical states in neurones. Such systems have no necessary utility. However, the development of mathematics is spurred by profound success in its use in modeling, predicting and changing material reality. In particular, there are a variety of formal models of computation, for example Turing machines, λ calculus and recursive function theory, which have all been shown to be mutually equivalent and to have the same expressive powers. As systems in themselves, all they offer is rules for syntactic manipulation of symbol sequences: they have no semantics other than in relation to other symbol systems which may act as their models. No one system has absolute primacy. Different systems have different theoretical and practical properties which make them more or less appropriate for different purposes. The significance of theories of computability lies in their use as models for physical computing machines and abstractions from such machines like programming languages. This enables the application of mathematical rigour to the computational animation of applied models of material reality.
>
> The equivalence of different mathematical systems amplifies the illusion that they must all have something more primary in common. However, mathematics does not depend upon the existence of abstract entities to which physical symbol systems approximate. Indeed, as some DS proponents have acknowledged, mathematics may be pursued without such philosophical assumptions. However, Platonism has important practical implications as its acolytes oppose operational readings of formal theories and hence seek to restrict the application of such theories through the animation of definitions as programs.

> There is a long history of such priestly obscurantism in mathematics. For example, in Europe, at least until the 14th century, practical arithmetic was carried out using tally sticks and counting boards whereas written numbers were the domain of a small elite who used them for ritual or state purposes. This divorce held back social and scientific advances as number systems were never tested and refined in the light of practice, thereby delaying the development of simple notations for zero and for positional indication of places in numbers [96].
>
> Historically, restrictions on the application of theory help to reproduce the social status of theoreticians as a privileged minority by delaying challenges to their philosophies through replicable empirical experimentation. Today, while Platonism is a minority taste in Mathematics, DS proponents have a justifiably high standing in the Computing community. It would be unfortunate if their assertions that particular mathematical systems transcend mundane computation were to hold back such systems' practical application.

Following discussion of the theoretical and practical status of DS, a number of systems for language implementation based on DS were considered. It was noted that such systems' adherence to full DS and their extensions for concrete syntax restricts their flexibility for experimentation with formal language definitions. Interpretive DS implementation using extant programming languages was then considered on the grounds that working with a unitary notation may be better suited to incremental definition animation. It was noted that this approach usually involves the simplification of multi-staged DS definitions. The particular suitability of functional language, due to their close correspondences to the DS meta language, was identified. However, programming languages generally lack constructs for syntactic processing.

The addition of concrete syntax constructs to a functional language was identified as a basis for DS animation, aided by the conflation and omission of DS stages. Explicit domain definitions may be subsumed by the use of types provided by the language, and with a weakly typed language, semantic equation domain signatures may be omitted. The direct association of concrete syntax and semantic equations obviates the need for explicit abstract syntax and focuses attention on final representations of programs in defined languages. It was suggested that these simplifications do not compromise significantly the overall rigour of DS but ease greatly the direct animation of definitions as interpreters.

9.2.3. Chapter 3 - The Navel core language

Chapter 3 started by considering the relationship between DS notations and functional languages. A brief but comprehensive survey of functional languages was then presented. This situates SASL, Navel's primary influence, as what might be termed a second generation functional language between the first generation LISP and POP2, and contemporary polymorphic typed languages. SASL was chosen as a basis for Navel on utilitarian and historical grounds, the most significant lack for DS animation being that of case structured function definitions with pattern matching. Core Navel was then discussed in some detail through informal examples and its DS. The DS provides a rigorous basis for Navel and, in particular, for the implementation of Navel in Navel discussed in chapter 7.

9.2.4. Chapter 4 - From syntax to semantics in Navel

Chapter 4 presented extensions to core Navel for syntactic processing and associating syntactic constructs with semantic equations. Syntax handling constructs in other languages were reviewed. Navel rules were then introduced, based on a generalisation of BNF where non-terminal left hand sides and sentential right hand sides of rules are uncoupled. This separation enables the close integration of rules as special forms of function values. Within rules, terminals are Navel strings and non-terminals are Navel identifiers. The field type consisting of paired tags and values was introduced. Rule application to strings returns trees formed from lists of strings for matched terminals and fields for matched non-terminals. A new construct for identifying the rule that constructed a tree was introduced, to overcome the absence of case structured definitions. A variety of small examples demonstrate that full Navel is simpler than but still close to the DS notation. The DS for syntactic processing completes the formal definition of full Navel.

9.2.5. Chapter 5 - Navel implementation, environment and extensions

Chapter 5 discussed the Navel implementation. The current system is based on a parse tree interpreter and is, effectively, an SECD machine: the stack and dump are implicit in the nested interpreter states, the environment is the Navel stack holding bound variable bindings and distributed amongst closures holding free variable bindings, and the control is the current parse tree node. During compilation, the locations of values associated with identifiers are predicted, thus avoiding any run-time environment searching. The implementation is not particularly novel: similar techniques are used in other functional language implementations. However, benchmarks suggest that performance is comparable to that of implementations of other languages of the same vintage.

The Navel environment was then considered. This provides basic interactive facilities for modular program development using a command line interface. Once again, this is not particularly novel for a pre-windows system but suffices for sustained use.

The weakest aspect of the environment is the all or nothing trace facility which provides too much and too detailed information about all stages of function evaluation. Some means of focussed tracing, for example by identifying explicitly particular constructs, would be preferable.

Finally, Navel extensions which ease DS implementation were presented. In particular the `exit` construct for non-standard termination of evaluation provides a default for failed rule/tree association which avoids the need for explicit error passing in DS definitions.

9.2.6. Chapter 6 - Language implementation from DS in Navel

Chapter 6 looked at the implementation of a substantial imperative language fragment from its DS. Once again, the close correspondence between a DS and the equivalent Navel is demonstrated. Benchmarks suggest that significant test programs in the defined language may be run in acceptable times. Comparison with other systems shows that while Navel is slower than en-masse systems for executing examples in a defined language, it is faster for definition and example processing. Thus, Navel may be more satisfactory as a basis for interactive language experimentation.

9.2.7. Chapter 7 - Navel in Navel and continuation passing semantics in Navel

Chapter 7 provided an overview of the implementation of Navel in Navel based on the DS in chapters 3 and 4. Techniques to overcome parse ambiguities due to the lack of explicit lexical definitions and to enable normal order evaluation of mutual recursive references through abstraction and lazy lists were discussed. Comparisons in terms of lines of code suggest that Navel in Navel is considerably smaller than Navel in C: predictably, tests show that it is also considerably slower. Examples demonstrate that Navel programs, including Navel based language definitions, may be run on Navel in Navel, confirming that Navel may be used with the DS for substantial languages.

Continuation passing semantics were then considered briefly. The DS of a simple imperative language with forward and backward jumps implemented in Navel shows that Navel may be used with Standard DS.

9.2.8. Chapter 8 - Rule generalisation and context sensitive parsing in Navel

Chapter 8 investigated the use of abstraction over rules to generalise parsing. Simple rule abstraction generalises context free parsing through the construction of parameterised rules analogous to higher-order functions. The 2-level grammar, attribute grammar and dynamic syntax approaches to context sensitivity were surveyed. The use of parameterised rules with explicit information passing between stages of sentential forms was proposed as a realisation of dynamic syntax in a functional context. The ability to carry out context sensitive parsing in Navel was demonstrated through rules to recognise the classic $a^n b^n c^n$ construct. The approach was then applied to simple examples drawn from imperative language static semantics. These show that a simple unitary notation integrating syntax within a functional language suffices for the expression of context sensitive as well as context free aspects of concrete syntax. Full formalisation of dynamic syntax in Navel would be an interesting area for further work.

9.3. Future directions

Navel satisfies the original objectives of this research, enabling and demonstrating the practicality of experimentation with DS through programming in a language which integrates syntax and semantic functions whilst retaining a close correspondence to DS. However, Navel reflects in age: there have been significant advances in language design and computer technology since its inception. Future research might take advantage of these developments.

It is useful to distinguish between the language underlying a system for DS and the system environment. Languages are general purpose tools. Many problems involve syntactic processing for data recognition and validation, in particular for the development of text based interfaces. Thus, languages which provide constructs for handling syntax have relevance beyond language implementation. Similarly, an environment for language development could correspond exactly to DS practice, using contemporary interface techniques. The underlying system might be based on a unitary language but details could be hidden from the user unless required explicitly.

9.3.1. Extensions to contemporary functional languages

There are several benefits to extending extant languages rather than inventing new ones. First of all, augmenting a language is simpler than implementing a successor, especially if the language contains constructs which correspond closely to or may be used to implement indirectly the extensions, for example through translation. Secondly, advantage may be taken of mature implementation technologies. In particular, contemporary functional language implementations are now comparable in response and efficiency to those for imperative languages. Thirdly, new concepts are more likely to gain currency if they augment ones which are already familiar.

Contemporary functional languages like SML, Miranda and Haskell are based on case structured pattern matching and provide discriminated union datatypes. For example, a binary tree of integers might be defined in SML by:

```
datatype itree = inil | inode of int * itree * itree
```

This specifies that an `itree` is either empty, `inil`, or a node, indicated by constructor `inode`, consisting of an integer value and branches to other `itree`s. Thus, the tree:

```
          7
   +------+------+
   4             10
+--+--+       +--+--+
1     6       9     12
```

has representation:

```
inode(7,
      inode(4,
            inode(1,inil,inil),
            inode(6,inil,inil)),
      inode(10,
            inode(9,inil,inil),
            inode(12,inil,inil)))
```

A function to add the leaf values is:

```
fun addleaves inil = 0 |
    addleaves (inode(v,l,r)) = v+(addleaves l)+(addleaves r)
```

Pattern matching and discriminated union types are well suited to the construction, identification and manipulation of parse trees. The form of discriminated unions might be extended to enable the representation of trees. One discriminated union would be defined for the tree for each non-terminal, with a number of options corresponding to the right hand side sentential forms.

Discriminated union types only allow general type expressions rather than specific values in definitions. For terminal symbols, this might be extended to allow the presence of specific strings. A constructor would identify a non-terminal. For example, a binary number:

```
<digit> ::= 1 | 0
<binary> ::= <digit> <binary> | <digit>
```

might have parse tree:

```
datatype d = zero of "0" | one of "1"
datatype b = digit of d | binary of d * b
```

Thus for:

```
101
```

the tree:

```
          <binary>
     +-------+-------+
  <digit>         <binary>
     |       +-------+-------+
     1    <digit>         <binary>
             |               |
             0            <digit>
                             |
                             1
```

would be:

```
binary(one "1",
       binary(zero "0",
              binary(digit(one "1"))))
```

In SML, the type name `real` is used to coerce an integer to a real. Similarly, parsing might be achieved by coercing a string with a datatype to return a tuple of the tree and rest of string:

```
b "11+10" ==>
(binary(one "1",binary(digit(one "1")))),"+10")
```

For each datatype whose type is used in a coercion, and for all the datatypes it refers to, a parse function could be generated automatically at compile time and appropriate calls planted for parsing at run time. In a definition, the right hand side structure identifies the sequence of parse functions to be applied and the constructor decorates the resulting tree.

Pattern matching would then be used to inspect trees. For example, to evaluate a binary number:

```
fun md (zero v) = 0 |
    md (one v) = 1

and mb (digit d) v = 2*v+(md d) |
    mb (binary(d,b)) v = mb b (2*v+(md d))

and run s =
 let val (tree,rest) = b s
 in mb tree 0
```

However, while these natural extensions are simpler than Navel's additional constructs, they are also more restrictive. Datatypes are somewhat different in form to grammars. In particular the datatype name identifies a type rather than a constructor corresponding to a non-terminal. Constructors in trees would identify which option was chosen. Thus, grammars would need to be converted explicitly to datatypes with constructors chosen to reflect a common non-terminal for the various options.

Furthermore, though apparently desirable, the use of single string constants or single datatype names as options in datatype definitions is inappropriate. At present, in discriminated union definitions, an option consisting of a type expression must be decorated with a constructor, even if that expression is a single type name. This is to distinguish the presence of a type from that of a zero arity constructor representing a new constant value. Thus, for consistency, single string constants should also be decorated with constructors as they are instances of the `string` type. However, this adds a layer of constructor to datatype denotations directly analogous to redundant nodes for singleton productions in concrete syntax trees. Finally, datatype definitions are not values. Hence, the equivalent status of rules and functions, and full rule generalisation are lost. Datatype parameterisation allows some generalisation but not the dynamic creation of rules at run time.

An alternative would be to augment an extant language with parser constructs, like Navel rules, which behave like functions. The system could deduce an appropriate datatype to represent the resultant parse trees and inform the user of its form. Here, as in Navel, abstraction over rules would allow full generalisation and dynamic creation of rules. However, this involves more substantial extensions to a language than those above, which extend existing orthogonality with few syntactic or semantic implications. Michaelson [97] contains further discussion of these issues.

Such an extended modern language would be useful in its own right for programming in general. It might also form the basis of the DS based environment proposed in the next section.

9.3.2. An environment for DS development

Recent advances in computer performance and functionality have made windows based interface technology available widely. A major criticism of DS based systems was that though multi-staged they are often monolithic for use. A windows based system would enable a flexible environment for DS development and experimentation, while retaining multiple stages for design.

Initially, the concrete syntax for a language, in BNF or another suitable representation, might be developed in a window. The system could deduce an appropriate representation for concrete parse trees and display them for test strings as two dimensional graphical structures.

Particular concrete constructs might be nominated as abstract syntax domains. The system could then deduce the form of the abstract syntax by folding concrete constructs together and asking the user to identify significant terminal symbols at each level. Such system deduction would retain the relationship between each concrete rule and the corresponding abstract construct to guide tree construction during parsing. The abstract syntax could be displayed en-masse alongside the concrete syntax, or in a separate window for each syntax domain.

The system could also deduce an appropriate data type to represent abstract syntax trees. Now, application of the sentence concrete rule would construct an abstract syntax tree which again could be displayed in a two dimensional graphical form. The form of the datatype need not be displayed to the user: semantic functions could be associated directly with abstract syntax constructs.

From the abstract syntax, the system could generate top level skeletons for semantic functions for each abstract syntax domain, with patterns for each domain form. The user could then identify semantic domains and the functionality of each semantic function. Then, the skeletons could be filled in by the user, with the system carrying out consistency checks on domains. Such skeletons might be displayed in separate windows, with links to the corresponding concrete and abstract syntax windows, to focus on semantically significant constructs.

At each stage, appropriate DS typefaces and symbol conventions could be used to distinguish different components of a definition: modern windows interfaces simplify greatly the use of multi-font text. Indeed, the system could infer the intended types of components from the selected typefaces or symbols, and carry out structural and consistency checks accordingly.

The system could also deduce the global implications of changes to any stage of a definition. Other stages could then be modified automatically or the user requested to guide such changes.

Such a system could be used transparently without reference to underlying representations. However, in the background the system could build and manipulate a program which implements the defined language, in an extended functional language of the form discussed above. That program could then be compiled independently to provide a stand alone implementation of the defined language. In particular, common domains could be predefined with efficient implementations in the underlying language hidden from the user.

This speculative account might form the outline of a fruitful longer term research project. The result should be a system which facilitates greater acceptance and use of formal language definitions in language development.

Appendix A
Navel in Navel

A.1 Syntax

```
def program = {{defns ";" | {}} expressions};
def defns = {defn {";" defns | {}}};
def defn = {"def" (name) {nameseq | {}} "=" expression};
def nameseq = {namelist {nameseq  | {}}};
def namelist = {namebase {":" {namelist | {}} | {}}};
def namebase = {(name) | "(" namelist ")"};
def name = {{"if" | "then" | "else" | "def" | "let" | "in" | "and" |
             "not" | "lam" | "rule" | "case" | "of" | "number" | "word" |
             "identifier" | "character" | "isnumb" | "ischar" |
             "isbool" | "isfield" | "islist" | "isstring" | "isfunc" |
             "isrule" | "true" | "false" | "fail" | "hd" | "tl" | "char"}
            {" " | "\n" | "\;" | "}" | ")" | "]" | "," | "->" | ":" |
             "(" | "[" | "{" | "   "}
            fail | identifier};
def expressions = {expression {";" expressions | {}}};
def expression =
 {ifexp | caseselect  | ruleselect | application  | letexp};
def ifexp = {"if" expression "then" expression "else" expression};
def caseselect = {"case" expression "of" cases};
def cases = {expression {"->" expression "," cases | {}}};
def application = {logexp {application | {}}};
def letexp = {"let" definitions "in" expression};
def definitions = {{(name) nameseq | namelist} "=" expression
                   {"and" definitions | {}}};
def logexp = {logterm {"|" logexp | {}} | lambda};
def lambda = {"lam" nameseq "." logexp};
def logterm = {logfactor {"&" logterm | {}}};
def logfactor = {{"not" | {}} logbase};
def logbase = {list {{"<=" | "=" | ">=" | "<>" | "<" | ">"} list | {}}  |
              {"isnumb" | "ischar" | "isbool" | "isfield" |
               "islist" | "isstring" | "isfunc" | "isrule"} aexp};
def list = {aexp {":" {list | {}} | {}}};
def aexp = {term {{"+" | "-"} aexp | {}} | "true" | "false" | string  |
            prodrule |  field};
def term = {factor{{"*" | "/" | "%"}term | {}}};
def factor= {{"-" | {}}base {indexes | {}}};
def base = {(name) | number | "(" expression ")" | "()" |
            {"hd" | "tl"} base  | "’"character"’" | "char" expression};
def string = {"\"" (stringend)};
def stringend = {" " (stringend) | "\n" (stringend) | "\"" |
                 (character) (stringend)};
def prodrule = {"{" rulebody "}" | "{}" | sysselector | "fail"};
def rulebody = {ruleexp {"|" rulebody | {}}};
def ruleexp = {ruleterm {ruleexp | {}}};
def ruleterm = {(name) | string | "(" expression ")" | prodrule};
def sysselector = {"number" | "word" | "identifier" | "character"};
def field = {"["{(name) | sysselector} expression "]"};
def indexes = {index {indexes | {}}};
def index = {"ˆ" {(name) | sysselector} {select | {}} | select};
def select = {"@" {number | (name) | "(" expression ")"}};
def ruleselect = {"rule" expression "of" rcases};
def rcases = {"{" rcase "}" "->" expression "," rcases | expression};
def rcase = {{(name) | string | sysselector} {rcase | {}}};
```

A.2 Intermediate pass

```
def abstract t =
rule t of
{defns ";" expressions} ->
 [d (abstract t^defns)]:[e (abstract t^expressions)],
{defn ";"  defns} -> [d (abstract t^defn)]:[d (abstract t^defns)],
{defn} -> [d (abstract t^defn)],
{expression ";" expressions} ->
 [e (abstract t^expression)]:";":[e (abstract t^expressions)],
{expressions} -> abstract t^expressions,
{expression} -> [e (abstract t^expression)],
{expression ";" program } ->
 [e (abstract t^expression)]:";":[e (abstract t^program)],
{"def" identifier "=" expression} ->
 "def":[id t^identifier]:"=":[e (abstract t^expression)],
{"def" identifier nameseq "=" expression} ->
 "def":[id t^identifier]:"=":
  (anameseq t^nameseq [e (abstract t^expression)]),
{ifexp} -> aifexp t^ifexp,
{caseselect} -> acaseselect t^caseselect,
{ruleselect} -> aruleselect t^ruleselect,
{application} -> abstract t^application,
{letexp} -> aletexp t^letexp,
{logexp} -> abstract t^logexp,
{logexp application} -> r_to_l_application t,
{logterm} -> abstract t^logterm,
{logterm "|" logexp} -> r_to_l_logexp t,
{lambda} -> alambda t^lambda,
{logfactor} -> abstract t^logfactor,
{logfactor "&" logterm} -> r_to_l_logterm t,
{"not" logbase} -> "not":[e (abstract t^logbase)],
{logbase} -> abstract t^logbase,
{list} -> abstract t^list,
{list "<=" list} -> [e (abstract t^list)]:"<=":[e (abstract t^list@2)],
{list "=" list} -> [e (abstract t^list)]:"=":[e (abstract t^list@2)],
{list ">=" list} -> [e (abstract t^list)]:">=":[e (abstract t^list@2)],
{list "<>" list} -> [e (abstract t^list)]:"<>":[e (abstract t^list@2)],
{list "<" list} -> [e (abstract t^list)]:"<":[e (abstract t^list@2)],
{list ">" list} -> [e (abstract t^list)]:">":[e (abstract t^list@2)],
{"isnumb" aexp} -> "isnumb":[e (abstract t^aexp)],
{"ischar" aexp} -> "ischar":[e (abstract t^aexp)],
{"isbool" aexp} -> "isbool":[e (abstract t^aexp)],
{"isfield" aexp} -> "isfield":[e (abstract t^aexp)],
{"islist" aexp} -> "islist":[e (abstract t^aexp)],
{"isstring" aexp} -> "isstring":[e (abstract t^aexp)],
{"isfunc" aexp} -> "isfunc":[e (abstract t^aexp)],
{"isrule" aexp} -> "isrule":[e (abstract t^aexp)],
{aexp} -> abstract t^aexp,
{aexp ":"} -> [e (abstract t^aexp)]:":":[e [nil ()]],
{aexp ":" list} -> [e (abstract t^aexp)]:":":[e (abstract t^list)],
{term} -> abstract t^term,
{term "+" aexp} -> r_to_l_exp t,
{term "-" aexp} -> r_to_l_exp t,
{"true"} -> [b true],
{"false"} -> [b false],
{string} -> astring (tl (t^string)),
{sysselector} -> t^sysselector,
{prodrule} -> aprodrule t^prodrule,
{"{}"} -> t,
{field} -> abstract t^field,
{"fail"} -> t,
```

```
{factor} -> abstract t^factor,
{factor "*" term} -> r_to_l_term t,
{factor "/" term} -> r_to_l_term t,
{factor "%" term} -> r_to_l_term t,
{base} -> abstract t^base,
{"-" base} -> "-":[e (abstract t^base)],
{base indexes} -> [e (abstract t^base)]:[inds (abstract t^indexes)],
{"-" base indexes} ->
 "-":[e (abstract t^base)]:[inds (abstract t^indexes)],
{identifier} -> [id (t^identifier)],
{number} -> [n (val (t^number))],
{"(" expression ")"} -> [e (abstract t^expression)],
{"()"} -> [nil ()],
{"hd" base} -> "hd":[e (abstract t^base)],
{"tl" base} -> "tl":[e (abstract t^base)],
{"'"character"'"} -> [chr (hd (t^character))],
{"char" expression} -> "char":[e (abstract t^expression)],
{"[" identifier expression "]"} ->
 [field (t^identifier):(abstract t^expression)],
{"[" sysselector expression "]"} ->
 [field (t^sysselector):(abstract t^expression)],
{index} -> [ind (abstract t^index)],
{index indexes} -> [ind (abstract t^index)]:[inds (abstract t^indexes)],
{"^" identifier} -> "^":[id t^identifier],
{"^" sysselector} -> "^":[id t^sysselector],
{"^" identifier select} ->
 "^":[id t^identifier]:[sel (abstract t^select)],
{"^" sysselector select} ->
 "^":[id t^sysselector]:[sel (abstract t^select)],
{select} -> [sel (abstract t^select)],
{"@" number} -> "@":[n (val (t^number))],
{"@" identifier}  -> "@":[id (t^identifier)],
{"@" "(" expression ")"} -> "@":[e (abstract t^expression)],
{namelist "=" expression} ->
 [ns (abstract t^namelist)]:"=":[e (abstract t^expression)],
{namelist "=" expression "and" definitions}  ->
 [ns (abstract t^namelist)]:"=":
  [e (abstract t^expression)]:"and":[d (abstract t^definitions)],
{identifier nameseq "=" expression} ->
 [id t^identifier]:"=":(anameseq t^nameseq [e (abstract t^expression)]),
{identifier nameseq "=" expression "and" definitions} ->
 [id t^identifier]:"=":(anameseq t^nameseq [e (abstract t^expression)]):
  "and":[d (abstract t^definitions)],
{namebase} -> abstract t^namebase,
{namebase ":"} -> [ns (abstract t^namebase)]:":",
{namebase ":" namelist} ->
 [ns (abstract t^namebase)]:":":[ns (abstract t^namelist)],
{identifier} -> [id (t^identifier)],
{"(" namelist ")"} -> (abstract t^namelist),
exit (t:"abstract syntax error");

def aprodrule t =
rule t of
{"{" rulebody "}"} -> [r (aprodrule t^rulebody)],
{"{}"} -> t,
{ruleexp} -> aprodrule t^ruleexp,
{ruleexp "|" rulebody} ->
 [r (aprodrule t^ruleexp)]:"|":[r (aprodrule t^rulebody)],
{ruleterm} -> aprodrule t^ruleterm,
{ruleterm ruleexp} ->
 [r (aprodrule t^ruleterm)]:[r (aprodrule t^ruleexp)],
{identifier} -> [id (t^identifier)],
```

```
{string} -> astring (tl (t^string)),
{"(" expression ")"} -> "(":[e (abstract t^expression)]:")",
{sysselector} -> t^sysselector,
{"fail"} -> t,
{prodrule} -> aprodrule t^prodrule,
exit (t:"error in prodrule");

def r_to_l_application l:r = swap_application (abstract l^logexp):r;

def aifexp t = "if":[e (abstract t^expression)]:
               "then":[e (abstract t^expression@2)]:
               "else":[e (abstract t^expression@3)];

def acaseselect t =
 "case":[e (abstract t^expression)]:"of":[c (acases t^cases)];

def acases t =
rule t of
{expression} -> [e (abstract t^expression)],
{expression "->" expression "," cases} ->
 [e (abstract t^expression)]:"->":[e (abstract t^expression@2)]:",":
  [c (acases t^cases)],
exit (t:"error - acases");

def aruleselect t = "rule":[e (abstract t^expression)]:
                    "of":[c (arcases t^rcases)];

def arcases t =
 rule t of
 {"{" rcase "}" "->" expression "," rcases} ->
  [r (arcase t^rcase)]:"->":[e (abstract t^expression)]:",":
   [c (arcases t^rcases)],
 {expression} -> [e (abstract t^expression)],
 exit (t:"error - rcases");

def arcase t =
 rule t of
 {identifier} -> [id t^identifier],
 {string} -> astring (tl (t^string)),
 {sysselector} -> t^sysselector,
 {identifier rcase} -> [r [id t^identifier]]:[r (arcase t^rcase)],
 {string rcase} -> [r (astring (tl (t^string)))]:[r (arcase t^rcase)],
 {sysselector rcase} -> [r t^sysselector]:[r (arcase t^rcase)],
 exit (t:"error rcase");


def swap_application l:r =
rule r^application of
{logexp} -> [e l]:[e (abstract r^application^logexp)],
{logexp application} ->
 swap_application
  ([e l]:[e (abstract r^application^logexp)]):r^application,
exit (l:r:"error-swap_application");

def aletexp t =
 "let":[d (abstract t^definitions)]:"in":[e (abstract t^expression)];

def anameseq n e =
rule n of
{namelist} -> [e "lam":[ns (abstract n^namelist)]:".":e],
{namelist nameseq} ->
 [e "lam":[ns (abstract n^namelist)]:".":(anameseq n^nameseq e)],
```

```
exit (n:e:"error - anameseq");

def alambda t = anameseq t^nameseq [e (abstract t^logexp)];

def r_to_l_logexp l:o:r = swap_logexp (abstract l^logterm):r;

def swap_logexp l:r =
rule r^logexp of
{logterm}-> [e l]:"&":[e (abstract r^logexp^logterm)],
{logterm "|" logexp} ->
 swap_logexp ([e l]:"|":[e (abstract r^logexp^logterm)]:):l^logexp,
exit (l:r:"error-swap_logexp");

def r_to_l_logterm l:o:r = swap_logterm (abstract l^logfactor):r;

def swap_logterm l:r =
rule r^logterm of
{logfactor} -> [e l]:"&":[e (abstract r^logterm^logfactor)],
{logfactor "&" logterm} ->
 swap_logterm ([e l]:"&":[e (abstract r^logterm^logfactor)]:):l^logterm,
exit (l:r:"error-swap_logterm");

def r_to_l_exp l:o:r = swap_exp (abstract l^term):o:r;

def swap_exp l:o:r =
 rule r^aexp of
 {term} -> [e l]:o:[e (abstract r^aexp^term)],
 {term "+" aexp} ->
  swap_exp ([e l]:o:[e (abstract r^aexp^term)]):"+":r^aexp,
 {term "-" aexp} ->
  swap_exp ([e l]:o:[e (abstract r^aexp^term)]):"-":r^aexp,
 exit (l:o:r:"error-swap_exp");

def r_to_l_term l:o:r = swap_term (abstract l^factor):o:r;

def swap_term l:o:r =
 rule r^term of
 {factor} -> [e l]:o:[e (abstract r^term^factor)],
 {factor "*" term} ->
  swap_term ([e l]:o:[e (abstract r^term^factor)]):"*":r^term,
 {factor "/" term} ->
  swap_term ([e l]:o:[e (abstract r^term^factor)]):"/":r^term,
 {factor "%" term} ->
  swap_term ([e l]:o:[e (abstract r^term^factor)]):"%":r^term,
 exit (l:o:r:"error-swap_term");

def val n = nval n 0;

def nval n v =
 if n=()
 then v
 else nval (tl n) 10*v+(hd n)-'0';

def astring s =
 if (tl s)=()
 then [nil ()]
 else stringjoin (hd s) (astring (tl s));

def stringjoin s1 s2 =
 if s1=()
 then s2
 else let t = stringjoin (tl s1) s2
```

```
      in [cons [chr (hd s1)]:t];

def abs text =
 let tree:rest = program text
 in
  if tree=()
  then "syntax error"
  else abstract tree;
```

A.3 Core semantics

```
def mprog p s =
 rule p of
 {d e} ->
  let news x = mdecls pˆd s news
  in mexp pˆe (news()),
 mexp pˆe s;

def mdecls d s news =
 rule d of
 {d d} -> mdecls dˆd@2 (mdecls dˆd s news) news,
 {d} -> mdecls dˆd s news,
 {"def" id "=" e} ->
  let e1 = mexp dˆe s
  in
   if not (isfield e1)
   then new s dˆid e1
   else
    rule e1 of
    {fc} -> new s dˆid [fc (news()):(tl (e1ˆfc))],
    {rc} -> new s dˆid [rc (news()):(tl (e1ˆrc))],
    new s dˆid e1,
 exit (d:"not a decl");

def new old lv rv l =
 if l=lv
 then rv
 else old l;

def mexp exp s =
 rule exp of
 {e ";" e} ->
  let e1 = writeln (mexp expˆe s)
  in mexp expˆe@2 s,
 {b} -> exp,
 {chr} -> exp,
 {n} -> exp,
 {id} -> s expˆid,
 {nil} -> exp,
 {e ":" e} ->
  let h = mlazy  expˆe s
  and t = mlazy expˆe@2 s
  in [cons h:t],
 {cons} -> exp,
 {"lam" ns "." e} ->
  [fc s:(lam a.lam s.(mexp expˆe (bind expˆns a s)))],
 {fc} -> exp,
 {e e} -> apply (mexp expˆe s) (mexp expˆe@2 s),
 {"if" e "then" e "else" e} ->
  let b = mexp expˆe s
  in
```

```
  rule b of
  {b} ->
   if b^b
   then mexp exp^e@2 s
   else mexp exp^e@3 s,
  exit (b:"not a boolean"),
{"case" e "of" c} -> mcase (mexp exp^e s) exp^c s,
{"let" d "in" e} ->
  let news x = mletdefs exp^d s news
  in mexp exp^e (news()),
{"-" e} -> [n -((mexp exp^e s)^n)],
{"not" e} ->
 let b = mexp exp^e s
 in
  rule b of
  {b} -> [b not b^b],
  exit (b:"not boolean"),
{"isnumb" e} ->
 rule mexp exp^e s of
 {n} -> [b true],[b false],
{"ischar" e} ->
 rule mexp exp^e s of
 {chr} -> [b true],[b false],
{"isbool" e} ->
  rule mexp exp^e s of
  {b} -> [b true],[b false],
{"islist" e} ->
  rule mexp exp^e s of
  {cons} -> [b true],[b false],
{"isstring" e} ->
 if checkstring (mexp exp^e s)
 then [b true]
 else [b false],
{"isfunc" e} ->
 rule mexp exp^e s of
 {fc} -> [b true],[b false],
{"isfield" e} ->
 rule mexp exp^e s of
 {field} -> [b true],[b false],
{"isrule" e} ->
 rule mexp exp^e s of
 {rc} -> [b true],[b false],
{"char" e} -> [chr (char (mexp e^exp s))],
{"hd" e} ->
 let l = mexp exp^e s
 in
  rule l of
  {cons} -> msusp (hd (l^cons)),
  exit (exp^e:"not a list but":l),
{"tl" e} ->
 let l = mexp exp^e s
 in
  rule l of
  {cons} -> msusp (tl (l^cons)),
  exit (exp^e:"not a list but":l),
{e} -> mexp exp^e s,
{e "+" e} -> marith lam x.lam y.x+y (mexp exp^e s) (mexp exp^e@2 s),
{e "-" e} -> marith lam x.lam y.x-y (mexp exp^e s) (mexp exp^e@2 s),
{e "*" e} -> marith lam x.lam y.x*y (mexp exp^e s) (mexp exp^e@2 s),
{e "/" e} -> marith lam x.lam y.x/y (mexp exp^e s) (mexp exp^e@2 s),
{e "%" e} -> marith lam x.lam y.x%y (mexp exp^e s) (mexp exp^e@2 s),
{e "&" e} ->
```

```
  let e1 = mexp exp^e s
  and e2 = mexp exp^e@2 s
  in
   rule e1 of
   {b} ->
    rule e2 of
    {b} -> [b (e1^b)&(e2^b)],
    exit (e2:"not boolean"),
   exit (e1:"not boolean"),
 {e "|" e} ->
  let e1 = mexp exp^e s
  and e2 = mexp exp^e@2 s
  in
   rule e1 of
   {b} ->
    rule e2 of
    {b} -> [b (e1^b)|(e2^b)],
    exit (e2:"not boolean"),
   exit (e1:"not boolean"),
 {e "=" e} -> [b (mexp exp^e s)=(mexp exp^e@2 s)],
 {e "<>" e} -> [b (mexp exp^e s)<>(mexp exp^e@2 s)],
 {e "<=" e} -> mcomp lam x y.x<=y (mexp exp^e s) (mexp exp^e@2 s),
 {e "<" e} -> mcomp lam x y.x<y (mexp exp^e s) (mexp exp^e@2 s),
 {e ">=" e} -> mcomp lam x y.x>=y (mexp exp^e s) (mexp exp^e@2 s),
 {e ">" e} -> mcomp  lam x y.x>y (mexp exp^e s) (mexp exp^e@2 s),
 {e inds} -> mindex (mexp exp^e s) exp^inds s,
 {r} -> [rc s:(parse exp)],
 {rc} -> exp,
 {"character"} -> [rc ():(lam str.lam s.(charmatch str))],
 {"word"} -> [rc ():(lam str.lam s.(wordmatch str))],
 {"identifier"} -> [rc ():(lam str.lam s.(idmatch str))],
 {"number"} -> [rc ():(lam str.lam s.(numbmatch str))],
 {"{}"} -> [rc ():(lam str.lam s.([nil ()]:str))],
 {"fail"} -> [rc ():(lam str.lam s.("rfail":str))],
 {field} -> exp,
 {"rule" e "of" c} -> mrule (mexp exp^e s) exp^c s,
 exit (exp:"not yet implemented");

def mlazy exp s =
 rule exp of
 {e} -> mlazy exp^e s,
 {e e} -> [e (mlazy exp^e s)]:[e (mlazy exp^e@2 s)],
 mexp exp s;

def apply f a =
 rule f of
 {fc} ->
  let s:e = f^fc
  in e a s,
 {rc} ->
  if checkstring a
  then
   let s:r = f^rc
   in
    let tree:rest = r a s
    in
     if tree="ofail" | tree="rfail"
     then [cons [nil ()]:a]
     else [cons tree:rest]
  else exit (a:"not a string for parsing with":fn),
 exit (f:"not a function or rule");
```

```
def bind n a s =
 rule n of
 {id} -> new s n^id a,
 rule a of
 {cons} ->
  rule n of
  {ns ":"} ->
   if (msusp (tl (a^cons)))<>[nil ()]
   then exit ("list":n:"doesn't match":a)
   else bind n^ns (msusp (hd (a^cons))) s,
  {ns ":" ns} -> bind n^ns@2 (msusp (tl (a^cons)))
                             (bind n^ns (msusp (hd (a^cons))) s),
  exit (ns:"strange bound variable structure"),
 exit (a:"not an argument list");

def msusp exp =
 rule exp of
 {e e} -> apply (mexp exp^e ()) (mexp exp^e@2 ()),
 exp;

def mcase exp c s =
 rule c of
 {e "->" e "," c} ->
  if exp=(mexp c^e s)
  then mexp c^e@2 s
  else mcase exp c^c s,
 {e} -> mexp c^e s,
 exit (c:"not a case");

def mletdefs dd s news =
 rule dd of
 {id "=" e} -> letbind [id dd^id] (mexp dd^e s) s news,
 {id "=" e "and" d} ->
  mletdefs dd^d (letbind [id dd^id] (mexp dd^e s) s news) news,
 {ns "=" e} -> letbind dd^ns (mexp dd^e s) s news,
 {ns "=" e "and" d} ->
  mletdefs dd^d (letbind dd^ns (mexp dd^e s) s news) news,
 exit (dd:"not a declaration");

def letbind n a s news =
 rule n of
 {id} ->
  if not (isfield a)
  then new s n^id a
  else
   rule a of
   {fc} -> new s n^id [fc (news()):(tl (a^fc))],
   {rc} -> new s n^id [rc (news()):(tl (a^rc))],
  new s n^id a,
 rule a of
 {cons} ->
  rule n of
  {ns ":"} ->
   if (msusp (tl (a^cons)))<>[nil ()]
   then exit (n:"doesn't match":a)
   else letbind n^ns (musup (hd (a^cons))) s news,
  {ns ":" ns} -> letbind n^ns@2 (msusp (tl (a^cons)))
                    (letbind n^ns (msusp(hd (a^cons))) s news) news,
  exit (n:"weird bound variables"),
 exit (n:"doesn't match":a);

def checkstring s =
```

```
 if s = [nil ()]
 then true
 else
  rule s of
  {cons} ->
   rule msusp (hd (s^cons)) of
   {chr} -> checkstring (msusp (tl (s^cons))),
   false,
  false;

def marith f e1 e2 =
 rule e1 of
 {n} ->
  rule e2 of
  {n} -> [n (f e1^n e2^n)],
  {chr} -> [n (f e1^n e2^chr)],
  exit (e2:"not number or character"),
 {chr} ->
  rule e2 of
  {n} -> [n (f e1^chr e2^n)],
  {chr} -> [n (f e1^chr e2^chr)],
  exit (e2:"not number or character"),
 exit (e1:"not number or character");

def mcomp c x y =
 rule x of
 {n} ->
  rule y of
  {n} -> [b c x^n y^n],
  {chr} -> [b c x^n y^chr],
  exit (x:y:"not compatible types for comparison"),
 {chr} ->
  rule y of
  {n} -> [b c x^chr y^n],
  {chr} -> [b c x^chr y^chr],
  exit (x:y:"not compatible types for comparison"),
 exit (x:y:"not compatible types for comparison");

def mindex exp index s =
 rule index of
 {ind} -> mindex exp index^ind s,
 {ind inds} -> mindex (mindex exp index^ind s) index^inds s,
 {"^" id} -> getfield exp index^id 1,
 {"^" id sel} -> getfield exp index^id (indnum index^sel s),
 {sel} ->
  let i = indnum index^sel s
  in
   if i<1
   then exit (i:"invalid index")
   else getindex exp i,
 exit ("index type":index:"not yet implemented");

def indnum i s =
 rule i of
 {"@" n} -> i^n,
 {"@" id} -> s i^id,
 {"@" e} -> mexp i^e s,
 exit ("indnum error":i);

def getfield exp i n =
 rule exp of
 {field} ->
```

```
   if n<>1
   then exit (exp:"doesn’t have a ^":i:"@":n)
   else
    if i=hd (exp^field)
    then msusp (tl (exp^field))
    else exit (exp:"doesn’t have a ^":i:"@":n),
  {cons} ->
   rule hd (exp^cons) of
   {field} ->
    if i=hd ((hd (exp^cons))^field)
    then
     if n=1
     then msusp (tl ((hd (exp^cons))^field))
     else getfield (msusp (tl (exp^cons))) i n-1
    else getfield (msusp (tl (exp^cons))) i n,
   getfield (msusp (tl (exp^cons))) i n,
  exit (exp:"doesn’t have a ^":i:"@":n);

def getindex exp i =
 rule exp of
 {cons} ->
  if i=1
  then msusp (hd (exp^cons))
  else getindex (msusp (tl (exp^cons))) i-1,
 exit (exp:"not a list");

def initstate id = (exit (id:"not declared"));

def run text =
 let tree:rest = program text
 in
  if tree=()
  then "syntax error"
  else mprog (abstract tree) initstate;
```

A.4 Rules semantics

```
def parse r str s =
 rule r of
 {r "|" r} -> optmatch r str s,
 {r r} -> seqmatch r str s,
 {id} -> nontermmatch r^id (s r^id) str,
 {"{}"} -> [nil ()]:str,
 {"fail"} -> "rfail":str,
 {"character"} -> sysmatch charmatch "character" str,
 {"number"} -> sysmatch numbmatch "number" str,
 {"word"} -> sysmatch wordmatch "word" str,
 {"identifier"} -> sysmatch idmatch "identifier" str,
 {"(" e ")"} -> expmatch (mexp r^e s) str,
 {r} ->
  let tree:rest = parse r^r str s
  in
   if tree="rfail"
   then "ofail":str
   else tree:rest,
 {cons} ->
  if checkstring r
  then strmatch r str
  else exit (r:"not a string for rule use"),
 exit (r:"not implemented for parsing");
```

```
def optmatch r str s =
  let t:rest = parse r^r str s
  in
   if t="ofail"
   then parse r^r@2 str s
   else t:rest;

def seqmatch r str s =
 let t1:rest1 = parse r^r str s
 in
  if t1="ofail" | t1="rfail"
  then t1:str
  else
   let t2:rest2 = parse r^r@2 rest1 s
   in
    if t2="ofail" | t2="rfail"
    then t2:str
    else (seqjoin t1 t2):rest2;

def seqjoin t1 t2 =
 if t1=[nil ()]
 then t2
 else
  if not (islist t1)
  then
   if t2=[nil ()]
   then t1
   else [cons t1:t2]
  else
   let t = seqjoin (tl t1) t2
   in
    if (hd t1)=[nil ()]
    then t
    else
     if t=[nil ()]
     then (hd t1)
     else [cons (hd t1):t];

def nontermmatch i r str =
 let t:rest = expmatch r str
 in
  if t="ofail" | t ="rfail"
  then t:str
  else [field i:t]:rest;

def expmatch r str =
 rule r of
 {rc} ->
  let s:rr = r^rc
  in rr str s,
 parse r str ();

def match first next str =
 if str = [nil ()]
 then "ofail":str
 else
  let h:t = (msusp (hd (str^cons))):(msusp (tl (str^cons)))
  in
   if first h^chr
   then
    let tt:rr = rest next t
    in
```

```
      if tt="ofail"
      then [cons h:[nil ()]]:t
      else [cons h:tt]:rr
    else
     if h^chr=' ' | h^chr='0
     then match first next t
     else "ofail":[cons h:t];

def rest next str =
 if str = [nil ()]
 then "ofail":str
 else
  let h:t = (msusp (hd (str^cons))):(msusp (tl (str^cons)))
  in
   if next h^chr
   then
    let tt:rr = rest next t
    in
     if tt="ofail"
     then [cons h:[nil ()]]:t
     else [cons h:tt]:rr
   else "ofail":(h:t);

def sysmatch match tag str =
 let t:r = match str
 in
  if t="ofail"
  then "ofail":r
  else [field tag:t]:r;

def charmatch str =
 if str=[nil ()]
 then "ofail":str
 else [cons (msusp (hd (str^cons))):(msusp (tl (str^cons)))];

def numbchar c = c>='0' & c<='9';

def numbmatch  = match numbchar numbchar;

def wordchar c = c>='A' & c<='Z'| c>='a' & c<='z';

def wordmatch = match wordchar wordchar;

def idchar c = (wordchar c) | (numbchar c);

def idmatch = match wordchar idchar;

def strmatch s str =
 if s=[nil ()] | str=[nil ()]
 then "ofail":str
 else
  let h:t = (msusp (hd (str^cons))):(msusp (tl (str^cons)))
  in
   if s^cons^chr=h^chr
   then
    let tt:rr = reststrmatch (tl (s^cons)) t
    in
     if tt="ofail"
     then "ofail":(h:t)
     else [cons h:tt]:rr
   else
    if h^chr=' ' | h^chr='0
```

```
    then strmatch s t
    else "ofail":(h:t);

def reststrmatch s str =
 if s=[nil ()]
 then [nil ()]:str
 else
  if str=[nil ()]
  then "ofail":str
  else
   let h:t = (msusp (hd (str^cons))):(msusp (tl (str^cons)))
   in
    if h^chr=s^cons^chr
    then
     let tt:rr = reststrmatch (tl (s^cons)) t
     in
      if tt="ofail"
      then "ofail":(h:t)
      else [cons h:tt]:rr
    else "ofail":(h:t);

def mrule exp c s =
 rule c of
 {r "->" e "," c} ->
  if rulematch exp c^r
  then mexp c^e s
  else mrule exp c^c s,
 {e} -> mexp c^e s,
 exit (c:"not a rule case");

def rulematch t r =
 if (checkstring t) & (checkstring r)
 then t=r
 else
 rule t of
 {cons} ->
  rule r of
  {r r} ->
   if rulematch (hd (t^cons)) r^r
   then rulematch (tl (t^cons)) r^r@2
   else false,
  false,
 {field} ->
  rule r of
  {id} -> (r^id)=(hd (t^field)),
  {"character"} -> (hd (t^field))="character",
  {"identifier"} -> (hd (t^field))="identifier",
  {"number"} -> (hd (t^field))="number",
  {"word"} -> (hd (t^field))="word",
  false,
 exit (t:"not a parse tree for rule matching");
```

A.5 Examples

Nested mutual recursion

```
def add x y =
 if y=0
 then x
 else add x+1 y-1;
```

```
add 3 2 ==> [n 5]

def mult x y =
 if y=0
 then 0
 else add x (mult x y-1);

mult 2 3 ==> [n 6]

def pow x n =
 if n=0
 then 1
 else mult x (pow x n-1);

pow 2 3 ==> [n 8]
```

List processing

```
def length l =
 if l=()
 then 0
 else 1+(length (tl l));

length 1:2:3: ==> [n 3]

def append l1 l2 =
 if l1=()
 then l2
 else
  let h = hd l1
  in
   let t = append (tl l1) l2
   in h:t;

append (1:2:3:) (4:5:6:) ==>
 [cons [n 1]:[cons [n 2]:[cons [n 3]:[cons [n 4]:
  [cons [n 5]:[cons [n 6]:[nil ()]]]]]]]

append "hello" "there" ==>
 [cons [chr ’h’]:[cons [chr ’e’]:[cons [chr ’l’]:[cons [chr ’l’]:
  [cons [chr ’o’]:[cons [chr ’t’]:[cons [chr ’h’]:[cons [chr ’e’]:
   [cons [chr ’r’]:[cons [chr ’e’]:[nil ()]]]]]]]]]]]

def find v l =
 if l=()
 then 1
 else
  if (hd l)=v
  then 1
  else 1+(find v (tl l));

find 3 (1:2:3:4:) ==> [n 3]

find 6 (1:2:3:4:) ==> [n 5]

def rev1 l1 l2 =
 if l1=()
 then l2
 else
  let h = hd l1
  in rev1 (tl l1) (h:l2);
```

```
def rev l = rev1 l ();

rev 1:2:3:4:5:6: ==>
 [cons [n 6]:[cons [n 5]:[cons [n 4]:[cons [n 3]:[cons [n 2]
  [cons [n 1]:[nil ()]]]]]]]

rev "hello" ==>
 [cons [chr ’o’]:[cons [chr ’l’]:[cons [chr ’l’]:[cons [chr ’e’]:
  [cons [chr ’h’]:[nil ()]]]]]]

def split l =
 if l=()
 then ():
 else
  let h:t = split (tl l)
  in ((hd hd l):h):((tl hd l):t);

split (1:1):(2:4):(3:9): ==>
 [cons [cons [n 1]:[cons [n 2]:[cons [n 3]:[nil ()]]]]
       [cons [n 1]:[cons [n 4]:[cons [n 9]:[nil ()]]]]]
```

Indexing infinite list

```
def sq x = x*x;

def sqs n = (sq n):(sqs n+1);

let s = sqs 0
in s@20 ==> [n 361]
```

Parsing lazy list

```
def strn n =
 if n=0
 then ""
 else ’a’:(strn n-1);

def as = {"a" as | "a"};

{as} (strn 4) ==>
 [cons]
  [field "as":
         [cons
          [cons [chr ’a’]:[nil ()]]:
          [field "as":
                 [cons
                  [cons [chr ’a’]:[nil ()]]:
                  [field "as"
                   [cons
                    [cons [chr ’a’]:[nil ()]]:
                    [field "as":
                           [cons [chr ’a’]:[nil ()]]]]]]]]]:
  [nil ()]]
```

Appendix B - Test language for Navel in Navel

B.1 Definition

```
def statements = {statement ";" statements | (statement)};
def statement = {(input) | (output) | (assign) | (test) |
                 (iterate) | (block)};
def input = {"read" identifier};
def output = {"write" exp};
def assign = {identifier ":=" exp};
def test = {"if" condition "then" statement {"else" statement | {}}};
def iterate = {"while" condition "do" statement};
def block = {"begin" statements "end"};
def condition = {exp {"<=" | "<" | "=" | ">=" | ">" | "<>"} exp};
def exp = {term {"+" | "-"} exp | (term)};
def term = {base {"*" | "/"} term | (base)};
def base = {identifier | number | "(" exp ")"};

def new old lv rv l =
 if l=lv
 then rv
 else old l;

def add v l =
 if l=()
 then v:
 else
  let t = add v (tl l)
  in (hd l):t;

def mstate t s:i:o =
 rule t of
 {"read" identifier} -> (new s tˆidentifier (hd i)):(tl i):o,
 {"write" exp} ->
  let ov = mexp tˆexp s
  in s:i:(add ov o),
 {identifier ":=" exp} -> (new s tˆidentifier (mexp tˆexp s)):i:o,
 {statement ";" statements} ->
  mstate tˆstatements (mstate tˆstatement s:i:o),
 {"begin" statements "end"} -> mstate tˆstatements s:i:o,
 {"while" condition "do" statement} ->
  miterate tˆcondition tˆstatement s:i:o,
 {"if" condition "then" statement "else" statement} ->
  if mexp tˆcondition s
  then mstate tˆstatement s:i:o
  else mstate tˆstatement@2 s:i:o,
 {"if" condition "then" statement} ->
  if mexp tˆcondition s
  then mstate tˆstatement s:i:o
  else s:i:o,
 ("not a statement":t);

def miterate c st s:i:o =
 if mexp c s
 then miterate c st (mstate st s:i:o)
 else s:i:o;

def mexp e s =
 rule e of
 {exp "<" exp} -> (mexp eˆexp s)<(mexp eˆexp@2 s),
```

```
 {exp "<=" exp} -> (mexp eˆexp s)<=(mexp eˆexp@2 s),
 {exp "=" exp} -> (mexp eˆexp s)=(mexp eˆexp@2 s),
 {exp ">=" exp} -> (mexp eˆexp s)>=(mexp eˆexp@2 s),
 {exp ">" exp} -> (mexp eˆexp s)>(mexp eˆexp@2 s),
 {exp "<>" exp} -> (mexp eˆexp s)<>(mexp eˆexp@2 s),
 {term "+" exp} -> (mexp eˆterm s)+(mexp eˆexp s),
 {term "-" exp} -> (mexp eˆterm s)-(mexp eˆexp s),
 {base "*" term} -> (mexp eˆbase s)*(mexp eˆterm s),
 {base "/" term} -> (mexp eˆbase s)/(mexp eˆterm s),
 {"(" exp ")"} -> mexp eˆexp s,
 {number} -> val eˆnumber,
 {identifier} -> s eˆidentifier,
 ();

def val n = nval n 0;

def nval n v =
 if n=()
 then v
 else nval (tl n) ((hd n)-'0'+10*v);

def run s i =
 let t:r = statements s
 in
  if t=fail | r<>()
  then "syntax error":r
  else
   let finals:finali:finalo =
        mstate t (lam id.(id:"not declared")):i:()
   in finalo;
```

B.2 Examples

```
run "write 2" ();
== [cons [n 2]:[nil ()]]

run "write 2;write 2*2" ();
== [cons [n 2]:[cons [n 4]:[nil ()]]]

run "a:=7;write a" ();
== [cons [n 7]:[nil ()]]

run "read a;write 2*a" 21:;
== [cons [n 42]:[nil ()]]

run "read a;read b;c:=a*b;d:=2*c;write d" 6:7:;
== [cons [n 84]:[nil ()]]

run "read n;f:=1;i:=1;
     while i<=n do
     begin
          f:=f*i;write i;write f;i:=i+1
     end" 6:;
== [cons [n 1]:[cons [n 1]:[cons [n 2]:[cons [n 2]:
    [cons [n 3]:[cons [n 6]:[cons [n 4]:[cons [n 24]:
     [cons [n 5]:[cons [n 120]:[cons [n 6]:[cons [n 720]:
      [nil ()]]]]]]]]]]]]]

run "read x;read y;prod:=0;
     while x>0 do
     begin
```

```
          c:=y;
          while c>0 do
          begin
               prod:=prod+1;write prod;c:=c-1
          end;
          x:=x-1
     end" 3:2:
== [cons [n 1]:[cons [n 2]:[cons [n 3]:
    [cons [n 4]:[cons [n 5]:[cons [n 6]:[nil ()]]]]]]]
```

References

1. A. Aasa, K. Petersson, and D. Synek, “Concrete syntax for data objects in functional languages,” in *Proceedings of the 1988 ACM Conference on LISP and Functional Programming,* Snowbird, Utah, (July 1988).

2. H. Abelson, G. J. Sussman, and J. Sussman, *Structure and Interpretation of Computer Programs,* MIT, (1985).

3. A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques and tools,* Addison-Wesley, (1986).

4. S. Allan and et al, in *SISAL: streams and iteration in a single assignment language. Language Reference Manual Version 1.2*, Dept of Computer Science, Colorado State University, (January 1985).

5. L. Allison, *A practical introduction to denotational semantics,* CUP, (1986).

6. E. B. Anderson, F. C. Belz, and E. K. Blum, “Issues in the formal specification of programming languages,” in *Formal description of programming concepts*, ed. E. J. Neuhold, pp. 1-30, North-Holland, (1978).

7. D. Andrews and W. Henhapl, “Pascal,” in *Formal specification and software development*, ed. D. Bjorner & C. B. Jones, pp. 175-251, Prentice-Hall, (1982).

8. L. Augustsson and T. Johnsson, *Lazy ML User’s Manual,* Programming Methodology Group, Dept of Computer Science, Chalmer’s Institute of Technology, Goteborg, Sweden, (1987).

9. J. Backus, “Can programming be liberated from the von Neumann style? A functional style and its algebra of programs,” *Communications of the ACM,* **Vol. 21**, (8), pp. 613-641, (August 1978).

10. R. Bahlke and G. Snelting, “The PSG - Programming System Generator,” *ACM SIGPLAN Notices,* **Vol. 20**, (7), pp. 28-33, (July 1985).

11. R. Bahlke and G. Snelting, “The PSG system: from formal language definitions to interactive programming environments,” *ACM Transactions on Programming Languages and Systems,* **Vol. 8**, (4), pp. 547-576, (October, 1986).

12. R. Barret, A. Ramsay, and A. Sloman, *POP-11: a practical language for artificial intelligence,* Ellis Horwood, (1985).

13. D. W. Barron, J. N. Buxton, D. F. Hartley, E. Nixon, and C. Strachey, “The main features of CPL,” *Computer Journal*, **Vol. 6**, pp. 134-43, (1963).

14. H. Bekic, D. Bjorner, W. Henhapl, C. B. Jones, and P. Lucas, “A formal definition of a PL/1 subset,” Vienna TR25.139, IBM, Vienna Laboratory, (1974).

15. D. Berry, “Generating program animators from programming language semantics,” ECS-LFCS-91-163, LFCS, University of Edinburgh, (1991).

16. D. Bjorner, C. B. Jones, and (Eds.), *The Vienna Development Method: the metalanguage,* Springer Verlag LNCS 61, (1978).

17. D. Bjorner, “Rigorous development of interpreters and compilers,” in *Formal specification and software development*, ed. D. Bjorner & C. B. Jones, pp. 271-320, Prentice-Hall, (1982).

18. J. Bodwin, L. Bradley, K. Kanda, D. Litle, and U. Pleban, “Experience with an experimental compiler generator based on denotational semantics,” *SIGPLAN Notices: Proceedings of SIGPLAN’82 Symposium on Compiler Construction*, **Vol. 17**, (6), pp. 216-229, (June, 1982).

19. C. Bohm, “The CUCH as a formal and description language,” in *Formal language description languages*, ed. T. B. Steel, pp. 179-197, North Holland, (1966).

20. J. M. Brady, *The theory of Computer Science: a programming approach*, Chapman and Hall, (1977).

21. R. A. Brooker and D. Morris, "A general translation program for phrase structure languages," *Journal of the ACM*, **Vol. 9**, (1), pp. 1-10, (1962).

22. W. H. Burge, *Recursive programming techniques,* Addison-Wesley, (1975).

23. R. M. Burstall, J. S. Collins, and R. J. Popplestone, *Programming in POP-2,* Edinburgh University Press, (1977).

24. R. M. Burstall, D. B. MacQueen, and D. T. Sanella, "HOPE: an experimental applicative language," CSR-62-80, Dept of Computer Science, University of Edinburgh, (1980).

25. L. Cardelli, *ML under UNIX*, Bell Laboratories, Murray Hill, NJ 07974, USA, (1983).

26. L. Cardelli, "The functional abstract machine," *Polymorphism*, **Vol. 1**, (1), (1983).

27. R. Carrick, J. Cole, and R. Morrison, "An introduction to PS-algol programming: 2nd edition," Persistent Programming Research Report 31, Dept of Computational Science, University of St Andrews/Dept of Computing Science, University of Glasgow, (1987).

28. S-J. Chao and B. R. Bryant, "Denotational semantics for program analysis," *SIGPLAN Notices*, **Vol. 23**, (1), pp. 83-91, (1988).

29. H. Christiansen, "Context-sensitive parsing in full Prolog," 5/1986, Roskilde University Centre, Computer Science, Denmark, (1986).

30. H. Christiansen, "Syntax, semantics and implementation strategies for programming languages with powerful abstraction mechanisms," in *Proceedings of 18th Hawaii Conference on System Sciences*, (1985).

31. A. Church, *The Calculi of Lambda Conversion,* Princeton University Press, (1941).

32. J. Cleaveland and R. Uzgalis, *Grammars for Programming Languages,* 4, Elsevier Computer Science Library (Programming Languages Series), Amsterdam, Holland, (1977).

33. W. F. Clocksin and C. S. Mellish, *Programming in Prolog,* Springer-Verlag, (1981).

34. I. Cottam, "discussed in 'Workshop on Software Tools for Formal Methods'," *FACS FACTS*, **Vol. 8**, (2), (February 1986).

35. G. Cousineau and G. Huet, *The CAML Primer, Version 2.6.1,* Projet Formal, INRIA-ENS, France, (1990).

36. A. J. T. Davie and R. Morrison, *Recursive descent compiling,* Ellis-Horwood, (1981).

37. A. J. T. Davie, *An introduction to functional programming systems using Haskell,* CUP, (1992).

38. T. Despeyroux, "Executable specification of static semantics," in *Semantics of Data Types,* pp. 215-233, Springer-Verlag LNCS 173, (1984).

39. E. Dijkstra, *A discipline of programming,* Prentice-Hall, (1976).

40. J. E. Donahue, *Complementary definitions of programming language semantics,* Springer-Verlag LNCS 42, (1976).

41. G. Dromey, *Program derivation: the development of programs from specifications,* Addison-Wesley, (1989).

42. B. Edupuganty and B. R. Bryant, "Two-level grammars as a functional programming language," *Computer Journal*, **Vol. 32**, (1), pp. 36-44, (February 1989).

43. *The EuLisp Definition Version 0.6,* University of Bath, (July, 1989).

44. A. Evans, "PAL: a language designed for teaching programming linguistics," in *Proceedings of the ACM 23rd National Conference*, pp. 395-403, (August 1968).

45. A. J. Field and P. G. Harrison, *Functional programming*, Addison-Wesley, (1988).

46. R. W. Floyd, “Assigning meaning to programs,” in *Mathematical Aspects of Computer Science*, ed. J. T. Schwartz, pp. 19-32, XIX American Mathematical Society, (1967).

47. J. Foderaro and K. Sklowar, *The FRANZ LISP Manual*, University of California at Berkley, U.S.A., (1983).

48. D. P. Friedman and D. S. Wise, “CONS should not evaluate its arguments,” in *Automata, languages and programming: 3rd International Colloquium*, ed. S. Michaelson & R. Milner, pp. 257-284, Edinburgh University Press, (1976).

49. R. A. Frost, “Constructing programs as executable attribute grammars,” *Computer Journal*, **Vol. 35**, (4), pp. 376-389, (August 1992).

50. D. Furber, “A survey of teaching of programming to Computing undergraduates in U.K. Universities and Polytechnics,” *Computer Journal*, **Vol. 35**, (5), pp. 530-533, (October 1992).

51. K. Futatsugi, J. A. Goguen, J-P Jouannaud, and J. Meseguer, “Principles of OBJ2,” in *Proceedings of 12th ACM Symposium on Principles of Programming Languages*, pp. 52-65, (1985).

52. R. M. Gallimore, D. Coleman, F. H. Ali, and V. Stavridou, “UMIST OBJ: a language for executable program specification,” *Computer Journal*, **Vol. 32**, (5), pp. 413-421, (October 1989).

53. H. Ganzinger, “Some principles for the development of compiler descriptions from denotational language definitions,” TUM-I8006, Technische Universitat Munchen, (May, 1980).

54. S. Ginsburg and E. Rounds, “Dynamic Syntax Specification Using Grammar Forms,” *IEEE Transactions on Software Engineering*, **Vol. SE-4**, (1), pp. 44-55, (1978).

55. J. A. Goguen, “Parameterised programming,” *IEEE Transactions on Software Engineering*, **Vol. 10**, (5), pp. 528-543, (September 1984).

56. M. J. C. Gordon, “Operational reasoning and denotational semantics,” in *Proceedings of International Symposium on Proving and Improving Programs*, pp. 83-98, Arc-et-Senans, France, (July, 1975).

57. M. J. C. Gordon, *The denotational description of programming languages: an introduction*, Springer-Verlag, (1979).

58. D. Gries, *The Science of programming*, Springer-Verlag, (1981).

59. R. E. Griswold and M. T. Griswold, *The Icon programming language*, Prentice-Hall, (1983).

60. R. E. Griswold, J. F. Poage, and I. P. Polonsky, *The SNOBOL4 Programming Language*, Prentice-Hall, (1971).

61. D. Guideman, “A continuation based semantics for Icon expressions,” TR 86-15, Department of Computer Science, University of Arizona, (April 1986).

62. K. Hanford and C. Jones, “Dynamic syntax: A concept for the definition of the syntax of programming languages,” TR 12.090, IBM Tech. Rep., (1971).

63. I. J. Hayes and C. B. Jones, “Specifications are not (necessarily) executable,” *Software Engineering Journal*, **Vol. 4**, (6), pp. 330-338, (1989).

64. P. Henderson and J. H. Morris, “A lazy evaluator,” in *Proceedings of 3rd ACM Symposium on Principles of Programming Languages*, pp. 95-103, (1976).

65. D. F. Hendry and B. Mohan, “BCL1 Manual,” ICSP 110, Institute of Computer Science, University of London, (June 1969).

66. W. Henhapl and C. B. Jones, “ALGOL 60,” in *Formal specification and software development*, ed. D. Bjorner & C. B. Jones, pp. 141-173, Prentice-Hall, (1982).

67. C. A. R. Hoare and F. E. Lauer, “Consistent and complementary formal theories of the semantics of programming languages,” *Acta Informatica*, **Vol. 3**, pp. 135-153, (1974).

68. C. A. R. Hoare, “An axiomatic approach to computer programming,” *Communications of the ACM*, **Vol. 12**, (10), pp. 322-329, (1969).

69. R. J. W. Housden, “The definition & implementation of LSIX in BCL,” *The Computer Journal*, **Vol. 12**, (1), pp. 15-23, (February 1969).

70. J. Hughes, “Why functional programming matters,” Report 16, Programming Methodology Group, University of Goteborg/Chalmers University of Technology, (November 1984).

71. K. Iverson, *A programming language*, Wiley, (1962).

72. S. Johnson, “YACC: Yet Another Compiler Compiler,” No. 32, Computing Science Technical Report, Bell Laboratories, Murray Hill, NJ 07974, USA, (1975).

73. T. Johnsson, “Attribute grammars as a functional programming paradigm,” Report 42, Programming Methodology Group, University of Goteborg and Chalmers University of Technology, Sweden, (September 1987).

74. C. B. Jones, “Compiler design,” in *Formal specification and software development*, ed. D. Bjorner & C. B. Jones, pp. 253-269, Prentice-Hall, (1982).

75. C. B. Jones, *Systematic software development using VDM*, Prentice-Hall, (1986).

76. N. D. Jones and D. A. Schmidt, “Compiler generation from denotational semantics,” in *Semantics directed compiler generation*, ed. N. D. Jones, pp. 70-93, Springer-Verlag LNCS 94, (1980).

77. S. L. Peyton Jones, *The implementation of functional programming languages*, Prentice-Hall, (1987).

78. P. Jouvelot, “Designing new languages or new language manipulation systems using ML,” *SIGPLAN Notices*, **Vol. 21**, (8), pp. 40-52, (August, 1986).

79. A. Kaldewaij, *Programming: the derivation of algorithms*, Prentice-Hall, (1990).

80. U. Kastens, “The GAG-System - a tool for compiler construction,” in *Methods and tools for compiler construction*, ed. B. Lohro, pp. 166-181, CUP, (1984).

81. H. A. Klaeren and H. Petzsch, “The development of an interpreter by means of abstract algebraic software specifications,” in *International Colloquium on Formalization of Programming Concepts*, pp. 335-346, Springer-Verlag LNCS 107, (1981).

82. S. C. Kleene, *Introduction to metamathematics*, pp. North-Holland, (1952).

83. P. J. Landin, “A correspondence between ALGOL 60 and Church’s lambda calculus, Part I,” *Communications of the ACM*, **Vol. 8**, (2), pp. 89-101, (February 1965).

84. P. J. Landin, “A correspondence between ALGOL 60 and Church’s lambda calculus, Part II,” *Communications of the ACM*, **Vol. 8**, (3), pp. 158-165, (March 1965).

85. P. J. Landin, “The mechanical evaluation of expressions,” *Computer Journal*, **Vol. 6**, pp. 308-320, (1964).

86. P. J. Landin, “The next 700 programming languages,” *Communications of the ACM*, **Vol. 9**, (3), (March 1966).

87. P. Lee and U. Pleban, “A realistic compiler generator based on high-level semantics,” in *Proceedings of 14th ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pp. 284-295, Munich, West Germany, (January 1987).

88. M. Lesk and E. Schmidt, “Lex - A Lexical Analyzer Generator,” No. 39, Computing Science Technical Report, Bell Laboratories, Murray Hill, NJ 07974, USA, (1975).

89. X. Lin, “Adding attributes to Navel grammars,” Technical Report 88/4, Dept of Computer Science, Heriot Watt University, (February 1988).

90. V. Linnemann, “Context-free Grammars and Derivation Trees as Programming Tools,” CSRG-117, Computer Systems Research Group, University of Toronto, (August 1980).

91. P. Lucas and K. Walk, “On the formal description of PL/1,” *Annual Review in Automatic Programming*, **Vol. 6**, (3), pp. 105-182, (1969).

92. H. Maurer and W. Stucky, “Ein Vorschlag fuer die Verwendung syntax-orientierter Methoden in hoeheren Programmiersprachen,” *Angewandte Informatik*, **Vol. 5**, pp. 189-195, (1976).

93. J. McCarthy, “Recursive functions of symbolic expressions and their computation by machine, Part 1,” *Communications of the ACM*, **Vol. 3**, pp. 184-195, (1960).

94. J. McCarthy, “A basis for a mathematical theory of computation,” in *Computer Programming and Formal Systems*, ed. P. Braffort & D. Hirschberg, pp. 33-70, North-Holland, (1965).

95. A. D. McGettrick, *The definition of programming languages,* CUP, (1980).

96. J. McLeish, *Number: from ancient civilisation to the computer,* Flamingo, (1992).

97. G. Michaelson, “Grammars and implementation independent structure representation,” in *Proceedings of 3rd International Workshop on Persistent Object Systems*, ed. J. Rosenberg & D. Koch, pp. 19-28, Springer-Verlag, Newcastle, Australia, (August 1990).

98. G. Michaelson, “Text generation from grammars,” *Information and Software Technology*, **Vol. 32**, (8), pp. 566-568, (October 1990).

99. G. Michaelson, *An introduction to functional programming through lambda calculus,* Addison-Wesley, (1989).

100. R. E. Milne and C. Strachey, *A theory of programming language semantics,* Chapman and Hall, (1976).

101. R. Milner, *Communication and concurrency,* Prentice-Hall, (1989).

102. R. Milner, M. Tofte, and R. Harper, *The definition of Standard ML,* MIT, (1990).

103. R. Milner, “A theory of type polymorphism in programming,” *Computer and Systems Science*, **Vol. 17**, (3), (1978).

104. R. Morrison, F. Brown, R. Connor, and A. Dearle, “The Napier88 reference manual,” Persistent Programming Research Report 77, Dept of Computational Science, University of St Andrews/Dept of Computing Science, University of Glasgow, (July 1989).

105. P. Mosses, “SIS - Semantics Implementation System: Reference Manual and User Guide,” DAIMI MD-30, Computer Science Dept., Aarhus University, Denmark, (August 1979).

106. P. D. Mosses, “The semantics of semantic equations,” in *Proceedings of 3rd Symposium on Mathematical Foundations of Computer Science*, pp. 409-422, Springer-Verlag, LNCS vol.28, Warszawa, Poland, (1974.).

107. T. Nicholson and N. Foo, “A denotational semantics for Prolog,” *ACM Transactions on Programming Languages and Systems*, **Vol. 11**, (4), pp. 650-665, (October, 1989).

108. H. R. Nielson and F. Nielson, *Semantics with applications: a formal introduction,* Wiley, (1992).

109. R. S. Nikhil, K. Pingali, and Arvind, “Id Nouveau,” Computation Structures Group Memo 265, Laboratory for Computer Science, MIT, (July 1986).

110. G. O’Neill, “Automatic translation of VDM specifications into Standard ML programs,” DITC 196/92, National Physical Laboratory, (February, 1992).

111. F. G. Pagan, “ALGOL 68 as a metalanguage for denotational semantics,” *Computer Journal*, **Vol. 22**, (1), pp. 63-66, (February 1979).

112. L. Paulson, “Compiler generation from denotational semantics,” in *Methods and tools for compiler construction,* ed. B. Lohro, pp. 219-250, CUP, (1984).

113. L. Paulson, *A compiler generator for semantic grammars*, Department of Computer Science, Stanford University, (December, 1981). PhD Thesis .

114. F. Pereira, *C Prolog User’s Manual Version 1.1*, EdCAAD, Dept. of Architecture, University of Edinburgh, Edinburgh, Scotland, (1982).

115. Plato, *Timaeus,* Penguin, (1965).

116. U. F. Pleban, “Compiler prototyping using formal semantics,” *SIGPLAN Notices: Proceedings of the ACM SIGPLAN’84 Symposium on Compiler Construction*, **Vol. 19**, (6), pp. 94-105, (June, 1984).

117. U. F. Pleban and P. Lee, “An automatically generated realistic compiler for an imperative language,” in *Proceedings of SIGPLAN’88 Conference on Programming Language Design and Implementation*, pp. 222-232, Atlanta, USA, (June 1988).

118. G. D. Plotkin, “A structural approach to operational semantics,” DAIMI FN-19, Arrhus University, Denmark, (September 1981).

119. K. J. Raiha, “Attribute grammar design using the compiler writing system HLP,” in *Methods and tools for compiler construction*, ed. B. Lohro, pp. 183-206, CUP, (1984).

120. M. Rakovsky and P. Collier, “From standard to implementation denotational semantics,” in *Semantics directed compiler generation*, ed. N. D. Jones, pp. 94-139, Springer-Verlag LNCS 94, (1980).

121. P. Rechenberg and H. Mossenbock, *A compiler generator for microcomputers,* Prentice Hall, (1989).

122. “Report on the functional programming language Haskell,” CSC/89/R5, Department of Computing Science, University of Glasgow, (March 1989).

123. J. H. Reppy, *Concurrent programming with events,* Dept of Computer Science, Cornell University, (November 1990).

124. T. W. Reps, *Generating language-based environments,* MIT, (1984).

125. J. C. Reynolds, “Definitional interpreters for higher-order programming languages,” in *Proceedings of 25th ACM National Conference*, pp. 717-740, (1972).

126. J. C. Reynolds, “GEDANKEN - a simple typeless language based on the principle of completeness and the reference concept,” *Communications of the ACM*, **Vol. 13**, (5), pp. 308-319, (May 1970).

127. M. Richards and C. Whitby-Strevens, *BCLP - the language and its compiler,* CUP, (1979).

128. V. Royer, “Transformations of denotational semantics in semantics directed compiler generation,” *Proceedings of SIGPLAN’86 Symposium on Compiler Construction, in SIGPLAN Notices,* **Vol. 21**, (7), pp. 68-73, (June, 1986).

129. D. A. Schmidt, *Denotational Semantics: a Methodology for Language Development,* Allyn & Bacon, Inc, (1986).

130. D. A. Schmidt, “Denotational semantics as a programming language,” CSR-100-82, Department of Computer Science, University of Edinburgh, (January, 1982).

131. D. A. Schmidt, “Detecting global variables in denotational specifications,” CSR-143-83, Department of Computer Science, University of Edinburgh, (September, 1983).

132. D. Scott, “Lattice theory, data types and semantics,” in *Courant Computer Science Symposium 2: Formal semantics of programming languages*, ed. R. Rustin, pp. 65-106, (1972).

133. D. S. Scott, “Lectures on a mathematical theory of computation,” PRG-19, Programming Research Group, University of Oxford, (1980).

134. D. S. Scott and C. Strachey, “Towards a mathematical semantics for computer languages,” PRG-6, Oxford University Computer Laboratory Programming Research Group, (August 1971).

135. R. Sethi, “Control flow aspects of semantics-directed compiling,” *ACM Transactions on Programming Languages and Systems*, **Vol. 5**, (4), pp. 554-595, (October, 1983).

136. K. Slonneger, *Denotational Semantics in Prolog*, University of Iowa, (1990).

137. G. L. Steele, *Common LISP: the language*, Digital, (1984).

138. S. Stepney, D. Whitley, D. Cooper, and C. Grant, “A demonstrably correct compiler,” *Formal Aspects of Computing*, **Vol. 3**, (1), pp. 58-101, (Jan-March 1991).

139. J. E. Stoy, *Denotational Semantics: the Scott-Strachey approach to programming language theory*, MIT, (1977).

140. J. E. Stoy, “Some mathematical aspects of functional programming,” in *Functional programming and its applications*, ed. J. Darlington, P. Henderson & D. A. Turner, pp. 217-252, CUP, (1982).

141. C. Strachey, *Fundamental concepts in programming languages*, Programming Research Group, University of Oxford, (1967).

142. C. Strachey, “Towards a formal semantics,” in *Formal language description languages*, ed. T. B. Steel, pp. 198-220, North-Holland, (1966).

143. M. Takeichi, *Inserting inject operations into denotational semantics*, 4, pp. 365-381, New Generation Computing, (1986).

144. M. Tofte, *Compiler generators*, Springer-Verlag, (1990).

145. K. R. Traub, *Implementation of non-strict functional programming languages*, Pitman, (1991).

146. A. M. Turing, “On computable numbers, with an application to the Eintscheidungsproblem,” *Proceedings of the London Mathematical Society, Ser. 2-42*, pp. 230-265, (1936).

147. D. A Turner, “The semantic elegance of applicative languages,” in *Proceedings of Conference on Programming Languages and Computer Architecture*, pp. 85-92, Portsmouth, New Hampshire, (October 1981).

148. D. A. Turner, “Miranda: a non-strict functional language with polymorphic types,” in *Proceedings of Conference on Functional Programming Languages and Computer Architecture*, Springer Verlag LNCS 201, (1985).

149. D. A. Turner, *SASL language manual*, Dept of Computational Science, University of St Andrews, (1973).

150. D. A. Turner, “SASL language manual,” CS/75/1, Dept of Computational Science, University of St Andrews, (September 1975).

151. D. A. Turner, “SASL language manual (Revised 1979 W. R. Cambell),” CS/79/3, Dept of Computational Science, University of St Andrews, (1976).

152. *UCSD P-system and UCSD Pascal Users’ Manual (2nd Edition)*, Softech Microsystems, San Diego, USA, (1981).

153. M. Wand, “A semantic prototyping system,” *SIGPLAN Notices: Proceedings of the SIGPLAN ’84 Symposium on Compiler Construction*, **Vol. 19**, (6), pp. 213-221, (June, 1984).

154. D. H. D. Warren, “An abstract Prolog instruction set,” Technical note 309, SRI International, Menlo Park, USA, (1983).

155. D. Watt and O. Madsen, “Extended Attribute Grammars,” *The Computer Journal*, **Vol. 26**, (2), pp. 142-153, (1983).

156. D. A. Watt, *Programming language syntax and semantics*, Prentice-Hall, (1991).

157. D. A. Watt, “An extended attribute grammar for Pascal,” *SIGPLAN Notices*, **Vol. 14**, (2), pp. 60-74, (1979).

158. P. Wegner, “The Vienna Definition Language,” *ACM Computing Surveys*, **Vol. 4**, (1), pp. 5-63, (1972).

159. A. van-Wijngaarden and et al, “Revised Report on the Algorithmic Language Algol 68,” *Acta Informatica*, **Vol. 5**, (1), pp. 1-236, (1975).

Contents