UNIVERSITY OF SOUTHAMPTON

Modular Grammars for Programming Language Prototyping

by

Stephen Robert Adams

A thesis submitted for the degree of
Doctor of Philosophy

in the
Faculty of Engineering
Department of Electronics and Computer Science

21 March 1991

Contents

University of Southampton

ABSTRACT

Faculty of Engineering

Electronics and Computer Science

Doctor of Philosophy

Modular Grammars for Programming Language Prototyping

by Stephen Robert Adams

Programming languages and formal notations are central to computer science. A significant part of computer science research is conducted by constructing programming language prototypes and studying their properties. I argue that the conventional modular structure of language processors as a sequence of phases is ill-suited to language implementation and propose an alternative to *phase oriented modularity*: to structure the language processor according to the facets of the implemented language. A language should be viewed as a collection of *language fragments*, each fragment defining an independent and reusable modular component. Given this structure new languages can be built cheaply by using different combinations of language fragments, reusing many existing fragments. Cheap language implementation also makes it feasible to perform *language oriented programming* where *application oriented languages* are constructed as abstractions of problem domains enabling concise solutions to be rapidly constructed for particular problems. This report describes how to build language processors from language fragments.

Language oriented modular structure is investigated in three arenas: interpreters, parsers and attribute grammars. In the first case each language fragment is embodied in an interpreter. The structure of the interpreters is changed to allow them to be combined. A modular parsing system is developed. Both the interpreters and modular parsers suffer from *structure clashes* which make some problems difficult to express. This is overcome with modular attribute grammars. A modular attribute evaluator is developed using a lazy functional programming language as a vehicle for experimentation. The resulting system has interesting properties: it allows language fragments to be based on the semantic structure of the language as well as the syntactic structure and includes support for programming by explicit refinement.

Finally, I argue that the implementation of languages as collections of component fragments may alter our perception of programming languages themselves.

Acknowledgements

There are many people who have helped make this work possible. In particular I would like to thank my supervisor, Peter Henderson, for applying an appropriate combination of the stick and the carrot. At times I have been an exasperating student and I would like to thank him for his patience in these times.

I would like to thank my colleague and friend David De Roure for many things. As a Ph.D. candidate one year ahead of me he demonstrated that there was a light at the end of the tunnel, and he has been a constant source of support and wisdom. David's support was not only moral—we had many interesting discussions on my work and he proof-read the entire draft thesis.

Chris Johnson and Andy Gravell have my appreciation for always being prepared to discuss concepts that were bothering me, even when the discussion ranged far from their areas of specialty.

I thank Olivier Danvy and his colleagues at DIKU for their hospitality during my visit in March 1989.

I would like to thank all those who read my final draft. The final work has benefitted greatly from their comments. In particular I thank Mark Dobie, Andy King and Ian Heath who are not otherwise acknowledged here.

Finally I would like to thank David Barron, Peter Henderson and Joan Lake for the effort that they put in to sorting out my Science and Engineering Research Council grant. My case was not as simple as it could have been!

1 Language oriented programming

This thesis describes a new method of structuring programming language implementations. An implementation is structured as a collection of *language fragments* which describe, largely independently, the different concepts which make up the language. The importance of an appropriate structure for language implementation follows from the importance of languages to computing, and a particular style of computing which I call *language oriented programming* which promises to reduce software engineering costs.

What is language oriented programming? Formal notations are fundamental to software engineering. Programming languages are an obvious example. Less obvious examples are the languages used to describe data formats, user interface appearance, the configuration of complex systems and in turn the languages used to describe all of these.

There is a trend towards more powerful languages—languages that let the user achieve more with less work or less training. Consider a database package on a personal computer. The package comes with a language for manipulating the database, a '4GL'. A relatively novice user can create a quite sophisticated application in an afternoon. It would take an expert programmer several days to reproduce the application using, say, a C compiler. The power of the *application oriented language* (e.g. the 4GL tool) gives a large increase in productivity. Given a new application there is a cost below which it is worthwhile developing and using an application oriented language rather than creating the application without one. The style of development where an application oriented language is created as a tool to assist in the engineering of an application is called *language oriented programming.*

Language oriented programming has some clear benefits. As the application oriented language is more powerful than a general purpose language we can expect the application to be more concise. This makes the application more manageable as there is less investment in application specific code. The language is designed for expressing problems in a particular problem domain so the problem description mirrors the problem itself. As a result of this closeness, changes in the problem require corresponding and obvious changes in the description. Similar applications can be built cheaply as they reuse the application oriented language. The design of an application oriented language also serves to clarify which issues are general issues and which are incidental to an individual application.

Currently the cost of building tools is too high for language oriented programming to be a common software engineering practice. Where the cost can be shared between many applications the practice is commonplace as witnessed by the 4GL example cited above. If the tool building cost can be reduced to the level where language oriented programming is cost effective for single or small numbers of applications then we will see a change in the way that software engineering is done.

To build an application oriented language implementation economically we need an mature language oriented programming technology. Language based tools are expensive to build because of the way that their implementations are structured. It is my thesis that a different implementation structure can substantially reduce language implementation cost. This in turn will lead to language oriented programming becoming a common engineering methodology.

A significant part of computer science research is performed by constructing programming language prototypes and studying their properties. The organization of a language implementation determines how easy or hard it is to change it to support a new construct or to reuse a part of it in a new language processor. In this report I argue that the traditional modular decomposition of a language processor is guided by implementation issues and as a result it is not conducive to good software engineering practices like reuse. An alternative organization is to decompose the implementation according to the language that is being implemented, i.e. according to the syntactic and semantic properties of the language. This poses the question "How does one write a language processor so that its modular structure reflects the structure of the language as a set of component features?" To answer this question requires a re-examination of what goes into a module, how modules are defined, how they are used and the tools used to support these language definition activities. A focus of this work is the rapid prototyping of programming languages and dialects, as it is here that module reuse pays its biggest dividends.

I have assumed that the reader has some knowledge of programming language implementation; the level provided by an introductory course is sufficient. I have also assumed that the reader has some knowledge of functional programming. The languages that I use are Common Lisp and Miranda[1]; a prior knowledge of either would help but it is not essential.

1.1 Language prototypes

A researcher might build a language prototype for any of a number of reasons.

Experimentation with new language features. New language constructs or new meanings of existing language constructs are often investigated by adding to or altering some existing implementation. This is true of both small scale experiments like adding a new control flow construct to an imperative language, and of larger experiments, for example the current vogue for adding object oriented programming to a language. The prototype implementation will be used to express and run test programs that demonstrate the new feature. The programs will be used to assess whether or not the new feature offers an advantage by being sufficiently expressive or having some expected performance.

Confidence. The implementor may want to try out some part of a language to check that the proposed method of implementation is correct or feasible. This is the traditional role of prototyping.

A similar reason for prototyping is to provide an executable version of some theoretical work. For example, one might like to try out a particular semantics although the mathematical properties of the semantics are really the subject of study. A working prototype gives confidence in the theory.

[1] Miranda is a trademark of Research Software Ltd.

Measurement. The implementor may want to make quantitative measurements of some aspect of a language, implementation or some programs written in that language. The prototype would execute the programs under instrumentation, for example, monitoring the amount and distribution of storage that is allocated.

Implementation technique. The novel feature may be an implementation technique, either as an algorithm in the language processor (interpreter or translator) or an end product of a translation. An example of the first is a new symbol table management strategy. An example of the second is a new stack frame allocation mechanism like a heap based stack to support multiple threads or continuations. The goal of implementation prototyping is to generate a more effective language processor.

Rapid generation of application oriented languages. As computers are used in more applications there is a need to invent suitable languages to describe the problems in the application area. The 'application' may even be a configurable tool to help with some other prototyping or software engineering activity; this is what Abelson & Sussman call metalinguistic abstraction [Abelson & Sussman 1985]. A prototype of such an application oriented language can be used to check that the proposed language is suitable for describing the problems and that its implementation is feasible.

These reasons for prototyping can be summarized as linguistic, algorithmic and implementation driven. My main interest is in the first and last items in the above list, that is, linguistic prototyping, as the conventional structure of language processors does not provide adequate support for this kind of prototyping.

1.2 How language prototypes are usually built

A language processor prototype is an ordinary language processor in which a minimum of effort has been spent in implementing the features that are not of interest to the prototype builder. A prototype is often constructed around an existing implementation in order to reuse parts of the implementation. Sometimes no suitable implementation exists and so the entire prototype must be constructed.

The traditional structure of a language processor is a sequence of phases like lexical analysis, parsing, type-checking, intermediate code generation, optimization and target code generation (figure 1.1). This is a phase oriented decomposition of the language processor. In addition there are some activities that are not restricted to a single phase, like symbol table management and error handling. They are usually implemented as service modules that are used by the other phases. The size of a phase module is related to the size of the language. A language with a large syntax has a large parser and a language with many features has a large semantic analysis phase. Often a large language leads to modules that are too large to be understood.

Some prototyping activities are served well by the implementation oriented structure of several phases. These activities are typically those concerned with some aspect of implementation. For example, if you were interested in code generation you would plug in your own code generator. The same goes for any other module in the language processor. For measurement a new runtime system or intermediate code generator can be used to count the required information. A prototype of a new algorithm for

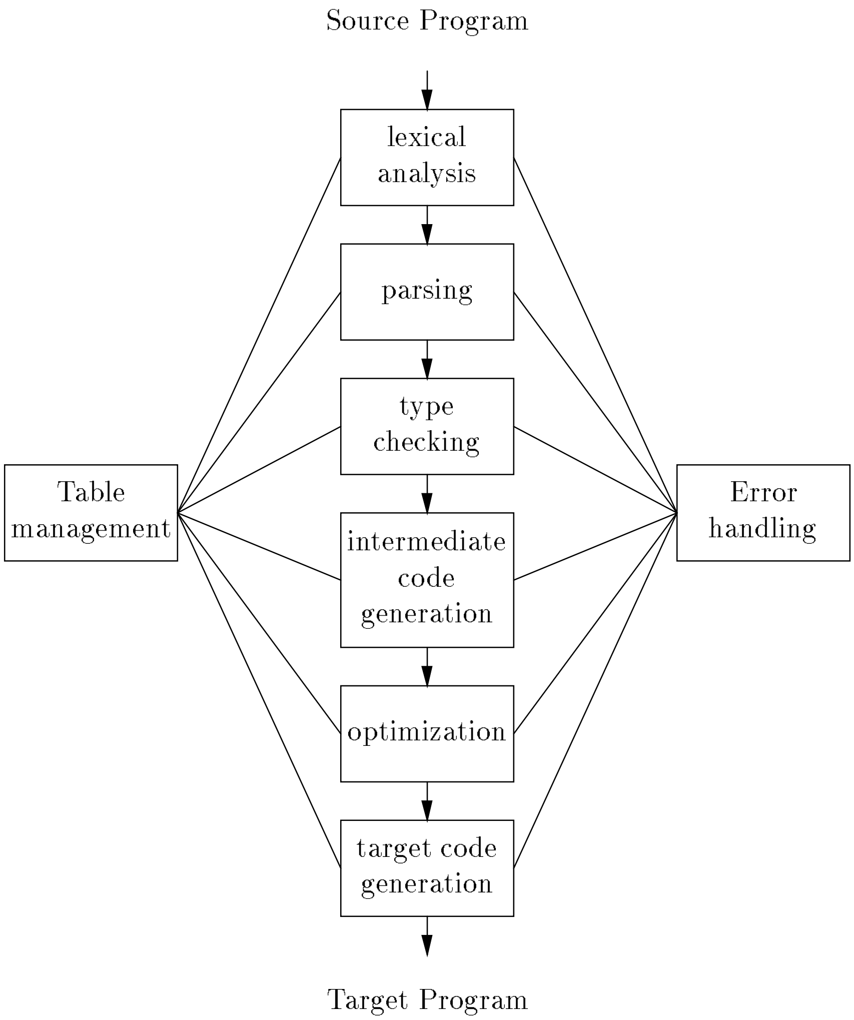


Figure 1.1: Traditional organization

part of a module is accommodated by changing or replacing that module. In each of these cases it is possible to do effective prototyping by taking an existing traditionally structured processor and replacing a single module. Occasionally the interfaces between the modules will need slight adjustment but the bulk of the effort is restricted to one module.

Other changes are not so well served by the traditional structure. The addition of a new language feature affects many of the modules. The new form has new syntax and possibly uses new symbols or keywords, and so requires changes in the lexical analyser and parser. The type checker will probably have to be altered, at best it will have to check the types of the components of the new feature, at worst the new feature introduces new types that require substantial changes in the algorithms. Similar comments apply to intermediate code generation. In short, the modular structure of a traditional language processor is inadequate for prototyping new dialects.

A common practice is to target the translation to some high level language. C

[Kernighan & Ritchie 1978] is a common choice[2]. The benefit of doing this is that the C compiler performs all of the code generation tasks; cost is a slower system because the C code is generated only to be parsed and typechecked by the C compiler. This is reusing large parts of the C compiler module through the 'interface' of the source language. Another benefit is portability—C compilers run on most platforms. I have used Lisp as a target language in a similar way in chapter 6.

1.3 Language fragments: An alternative structure

We have seen that the traditional language processor structure is not well suited to the prototyping of new languages and dialects. Why is this? Koskimies [1988] suggests

> "The reason for this is probably the fact that programming languages are regarded as complex, indivisible objects. Hence, from the software engineering point of view the language to be implemented is a black box: the implementor tends to think in terms of the services needed by the translation process, and does not try to divide the language itself into logical pieces that would be represented by separate modules."

Instead of a single indivisible entity I propose that a language should be viewed as a collection of *language fragments*. A language fragment is a cohesive set of language features that form a natural unit. A language definition would then resemble the *structure of the language* rather than the traditional phase structure. In this I include the semantic structure (e.g. data types and variable scopes) as well as the more obvious syntactic structure. The language definition should resemble the body of the reference manual rather than the appendix containing the collected syntax.

Three examples of potential language fragments are: expressions, imperative statements and record types. Expressions are an especially good language fragment—almost all languages have a substantial expression sub-language that is nearly the same as the next language. Re-implementing this fragment for each language is an unnecessary burden, especially for a prototype builder who is experimenting with some other aspect of the language and does not care too much about the details of the expression, only requiring that the expressions look and behave as a casual reader would expect.

A 'record types' fragment is a different example as it more obviously affects other concerns like storage allocation and type checking. It would be concerned with all aspects that were unique to aggregate structures, like component selection, storage requirements and the new aggregate type rules; but not with general issues of, say, the expressions in which the selected components are used or variable storage allocation or the other type rules of the language.

Whilst this is a convincing argument for language fragments, we need to understand how languages should be modularized, i.e. what decompositions of a language are appropriate for reuse. It is clear in conventional modular programming that a poorly considered assignment of functions and responsibilities between modules does more harm than good. We have learned how to construct effective modular decompositions of conventional programs, for example Parnas advocates abstract data types [Parnas 1972]. Similarly we need to learn what constitutes a good language fragment and a poor one.

[2]There are many examples. Kyoto Common Lisp translates the Lisp into C and uses the C compiler to produce an object code file which is linked into the running Lisp program. The original AT&T C++ implementation was a preprocessor that produced an equivalent C program.

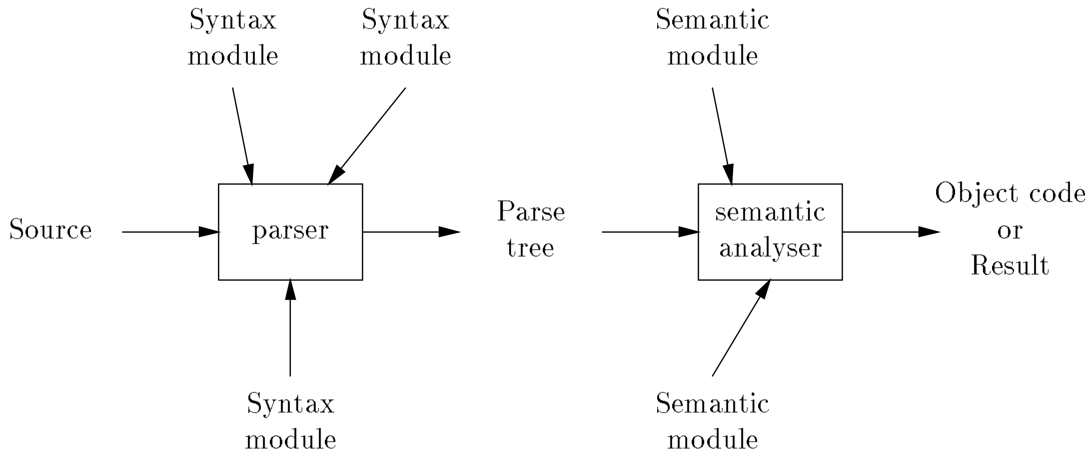


Figure 1.2: Syntactic and semantic modules

Tools are required to help us construct and manage language fragments. Consider how Smalltalk has been used as user interface prototyping tool. Object oriented programming, and in particular, Smalltalk, has revolutionized how we think about graphics programming and user interfaces. The effect is clear in the modern generation of computers which interact with users via windows and mice rather than command line instructions. Appropriate tools and a system of language fragments promise to revolutionize programming language implementation in a similar fashion.

Unlike some other attempts to impose a language oriented structure on language implementations, I have made a distinction between syntactic and semantic modularity This is illustrated in the new language processor organization in figure 1.2. The distinction is of great importance as it allows semantic modules to be decoupled from the concrete syntax and so apply to more languages. A syntax module specifies part of the appearance of the language. The semantic modules are based on attribute grammars [Knuth 1968], but they are rather unconventional as they allow definitions which summarize the behaviour of the module over uninteresting syntax. In chapter 3 I demonstrate the need for a module to summarize its behaviour over the syntactic objects which are irrelevant to its behaviour.

1.4 Contribution

These are the main contributions of this work:

- I have shown how to write language processors that are naturally structured around the language features rather than having the traditional implementation oriented structure. This structure considers a language to be the sum of several *language fragments* rather than a single indivisible entity.

- I have demonstrated programming by explicit refinement in an attribute grammar based language processor, a method similar to deriving subclasses in an object oriented language. This method allows new processors to be specified by *how they differ* from a previous processor.

There are several lesser contributions:

- Metalinguistic constructs. I have shown how to parse programs containing quoted fragments of other programs in a top-down parsing scheme and discussed how these ideas transfer to bottom-up parsing.
- The correspondence between attribute grammars and functional programming [Chirica & Martin 1979], [Johnsson 1987] has been manipulated to factor out the attribute style dependencies, allowing both standard and novel structuring of the functional equivalent to the attribute grammar.
- The concrete grammar and the grammar used in the attribute grammar modules do not have to be strongly coupled. Indeed, this can be a disadvantage.

1.5 Overview of related work

Chapter 2 surveys related work. Each work is related in one or more ways: by modularity, mechanism or notation. Some are concerned with modularity in the description of a language processor or expressing the modularity inherent in the language being described. SDF (Syntax Definition Formalism) [Heering *et al.* 1989a] provides an alternative for the term syntax for algebraic specifications. An algebraic term representing, say, an if-statement can then be written as the if-statement rather than the term form: if(condition,yes,no). An algebraic specification is built up from modules and the definitions define both the concrete syntax and the abstract syntax (internal term structure) so the definition of the syntax is completely modular. The concrete syntax is used both to represent fragments of a program that is being processed and for a nice notation for expressing the processor. This is possible because algebraic specifications blur the distinction between program and data—everything is one term which is reduced according to a set of rewrite rules. The importance of this work is that it introduces complete syntactic modularity.

Dueck and Cormack [1990] have developed a modular attribute grammar system in response to the difficulties of writing large attribute grammar specifications. Their system has many similarities to mine and I compare them in detail in section 5.15.

Koskimies has contributed to the modular structure of language processors. The work has concentrated on writing the processor as a collection of modules in a conventional modular language like Modula-2 or Oberon. In [Koskimies 1988] each nonterminal in the context free grammar is assigned to a module. That module imports the modules for the symbols which appear on the right side of the grammar productions. Using this organization the addition, deletion or alteration of one production will only affect one module. In [Koskimies 1990] a new technique is developed that allows recursive descent parsing to be performed in this one-nonterminal-to-one-module regime. The importance of this work is that it allows single-pass recursive descent parsers which are traditionally implemented as one monolithic module to be implemented as a quite large number of manageable modules.

The use of a functional programming language to implement an attribute grammar is fundamental to my development of a modular system. I have built upon two works: [Chirica & Martin 1979] and [Johnsson 1987]. What these works have in common is that they describe how to translate a monolithic attribute grammar into a functional program that computes the value specified by the attribute grammar. Frost is concerned with building a program out of smaller units according to a calculus of interpreters

[Frost 1990]. The result of this is a better model for constructing and reasoning about programs. What is of interest in this work is the precedent for using a functional language (Frost uses Miranda, as I do) to implement a prototype of the calculus which, in essence, is a language processor.

There is an important difference in the approach of these works and my own. Chirica and Martin are interested in constructing formal models and understanding attribute grammars. Frost and Johnsson are interested in programming paradigms that are based on attribute computation. My use is in a different direction. I using functional programming to enhance attribute grammars. Attribute grammars, the thing that is fixed in the other works, is the thing that changes here.

1.6 Overview

The structure and reusability of programming language processors and their components can be improved by use of modular syntax and modular semantics. The two modular aspects need not be tightly coupled. Each aspect offers some benefits but the benefits are greater when both are used together. The benefits of language oriented decomposition are accented with language prototyping because the conventional implementation oriented modular decomposition into phases tends to be shallower and thus less useful.

Chapter 2 is a survey of related work. The rest of this report describes three investigations of modularity in language implementations. The first is a mainly computational approach, working with the executable definitions of interpreters for the languages. The second is syntactic, introducing and describing the implementation of a system for using modular syntax in a variety of applications. The third investigation is concerned with a modular decomposition of attribute grammars.

I investigate how language fragments may be supported by conventional programming paradigms in chapter 3. In particular the functional, object oriented and algebraic specification paradigms are stretched to provide a fragment oriented modular description of a simple programming language.

Chapter 4 is concerned with syntactic modularity. A context free grammar describing the syntax of a language is composed of several sub-grammars. This allows the syntax of a language to be constructed from a set of grammar modules. This chapter discusses the efficient parsing of a grammar that has a modular structure and how program fragments can be parsed and represented in a meta-programming environment. It describes the "parser" box of figure 1.2.

In chapter 5 attribute grammars are described and related to functional programming. The relationship with functional programming is exploited to derive a system of modular attribute grammars that are only weakly coupled to the concrete grammar. This is used as a mechanism for describing semantic fragments, i.e. individual semantic aspects of a language. The programming methodology described in this chapter implements the "semantic analysis" box of figure 1.2. The modular attribute grammars also support a new method of programming—explicit refinement. A module may redefine what is specified by another module, either improving or correcting the behaviour of that module, or overriding that module to define an alternative semantics. This chapter is the most exciting.

Chapter 6 is an extended example of the concepts introduced in the previous two chapters. It contains a complete example of two prototype languages with examples of their use and a comprehensive description of their implementation.

Finally, chapter 7 concludes by summarizing the achievements and discusses how language oriented modularity might affect the development of programming languages.

2 Survey

This chapter surveys the current trends, techniques and tools used in the description of programming languages. There are three major categories into which these items can be placed:

- The description of language objects and program objects as entities to be manipulated by other programming systems.
- The parsing problem, that is converting the input text into the structured form defined by a grammar.
- Attribute grammars and developments in the use and expressiveness of this formalism.

I also discuss how some existing language system features can be subverted to language prototyping.

2.1 Programs as data

2.1.1 GRAMPS

GRAMPS (GRAmmar-based MetaProgramming Scheme) [Cameron & Ito 1984] is a method of deriving an abstract data type (ADT) to describe a program from the language's grammar. The grammar is used as a basis for deriving a data type and operations to represent and manipulate a program. The direct correspondence between the abstract data type and the original grammar facilitates its use in metaprogramming (i.e. the writing of applications that manipulate programs). The ADT provides type recognition (i.e. syntactic type), component selection, construction, context determination and editing. GRAMPS is a useful and uniform methodology for using a conventional programming language to do metaprogramming. The paper is useful because it lists the operations that are required for metaprogramming.

2.1.2 TXL

TXL [Cordy *et al.* 1988] is a rapid prototyping system for extending an existing programming language. The system consists of:

- A base language on which extensions are built.
- A dialect description language for specifying the syntax and semantics of the extension.
- A processor that reduces a program written in the extended language into a program in the base language.

The transformer is context sensitive, which makes it more powerful than traditional context free preprocessors and extensible languages.

A dialect is described in two parts. Syntactic forms are described using a BNF-like notation. The dialect forms are integrated into the base language grammar by replacing existing base language nonterminals with the new nonterminals. Nonterminals which have the same name are redefined, those with new names are simply added to the grammar. The semantics of the dialect are described by a set of rules which transform the new syntactic forms into base language structures.

TXL is capable of arbitrary general pattern matching, recursive transformations, arbitrary code motion, generation of unique identifiers and reference to auxiliary support routines. Dialects are easier to describe in a base language like Turing [Holt *et al.* 1988] where there are few restrictions on the order of declarations and statements, than, say, a base language like Pascal.

The range of dialects that can be described is restricted by the power of the base language. It is difficult to add features for which there is no support for an underlying model (for example, data parallelism). The unit of reuse is the base definition on which the dialect is built—another dialect will be built on this same base.

2.1.3 Concrete Syntax for Data Objects

Inductive data types can be seen as a grammar for a language which has the values of the data type as the sentences of the language. Many modern functional programming languages can define inductive data types but, as observed by Aasa *et al.*, this correspondence is not taken to its conclusion in providing linguistic and input/output support for the data types as syntactic objects [Aasa *et al.* 1988].

Aasa *et al.* extend ML with concrete data types. Concrete data types are defined using a new type definition, *conctype*. Elements in the language are enclosed in *quotation brackets* "`[|`" and "`|]`". A simple example of a type definition is one for binary numbers:

```
conctype BinNumber = [|0|]
                   | [|1|]
                   | [|<BinNumber>0|]
                   | [|<BinNumber>1|]
```

A new form of expression, called a *quotation expression* is used to write down concrete data type values, e.g. `[|101|]`. Quotation patterns are allowed in pattern matching. An antiquotation symbol "`^`" can be used to put an ordinary ML expression or pattern in a quotation expression or pattern. These ideas are illustrated by the successor function:

```
fun succ [|0|]     = [|1|]
|   succ [|1|]     = [|10|]
|   succ [|^b 0|]  = [|^b 1|]
|   succ [|^b 1|]  = [|^(succ b)0|]
```

A set of concrete types is compiled into a set of inductive types that describe the parse tree.

It is sometimes necessary to specify the type of an antiquotation variable in a pattern. Take for example the following concrete type for an expression that can be a variable, an integer or some other things not specified here:

```
conctype Exp = [|<Var>|]  |  [|<Integer>|]  |  ...
```

Assume that Var and Integer are defined elsewhere. In the pattern `[|^n|]` it is not possible to decide the type of `n` as it could be either Var or Integer. A solution is to specify the type in the pattern like this: `[|^(n:Integer)|]`.

The concrete types integrate neatly into the ML type system, for example, a conc-type may be polymorphic. Unfortunately, parsing a program with (multiple) concrete data types is complicated by type issues. The expected type or the type of any antiquotations can determine the correct parse of a quotation expression (e.g. `n:Integer` above). Parsing a quotation pattern can determine the type of antiquotations. Both expressions and patterns can determine the assignment of polymorphic type variables. Thus parsing of quotations has to be integrated with the type inference system. The solution adopted is based on Earley's algorithm [Earley 1970]. The algorithm is generalized to operate under type variable substitutions and to make such substitutions where appropriate.

Extensions to the basic system include special concrete types for lists. In effect this replaces the automatically generated constructors with normal list constructors. The benefit of this is that standard list processing functions like `map` and `foldleft` can be used to traverse the list-like parts of the syntax tree.

2.1.4 CAML

CAML [Cousineau & Huet 1989] is a functional language in the ML family. It is a statically typed language with polymorphic typing. The implementation is based on the Categorical Abstract Machine [Cousineau *et al.* 1985], from which the name CAML is derived: Categorical Abstract Machine Language. CAML is of interest as a language tool because it is integrated with notations for parsing and printing object languages [Mauny 1989]. It bears similarities to 'concrete types' of Aasa *et al.* but the approach is quite different.

Parsers are modeled as *stream pattern* matching functions which yield a computed value, typically an abstract syntax tree. Printers, on the other hand, are functions from some data type to a stream of formatting commands based on the arrangement of boxes of text. There is a special notation for specifying each of parsers and printers. The typing of this system is well integrated with the typing system of the underlying ML dialect, but it is an extension of the typing system rather than an extension of the type inference mechanism.

CAML differs from 'concrete types' in that the latter is an extension of notation for inductive types, whereas CAML supports a parsing method that may produce values of any type desired by the programmer. The type of the parse tree has to be declared in the program in addition to specifying the parser. The parser is a kind of function definition, not a type declaration. The 'antiquotation' escape feature of concrete types can be added to a CAML grammar by explicitly calling the system parser to parse the ML part of the escape. This is illustrated in figure 2.1 which shows an example grammar for arithmetic expressions. Note how the grammar builds the parse tree with constructors (`'Constant` etc.), and that the abstract syntax does not have any constructor for a parenthesized expression. The syntax `<<`...`>>` is used to write values and patterns that are parsed to produce the abstract syntax. An example showing the use of these features is the following function to evaluate an expression:

```
let calc = function
   'Constant(n)    ->  n
|  <<^e1 + ^e2>>  -> calc(e1)+calc(e2)
|  <<^e1 * ^e2>>  -> calc(e1)*calc(e2)
```

```
type aexpr =
  Constant of num
| Addition of (aexpr * aexpr)
| Multiplication of (aexpr * aexpr)

grammar for programs Expr_meta =
precedences
  left "+";
  left "*";
rule entry exp =
   parse NUM n                        -> 'Constant(n)
       | exp e1; "+"; exp e2          -> 'Addition(e1,e2)
       | exp e1; "*"; exp e2          -> 'Multiplication(e1,e2)
       | "("; exp e; ")"              -> e
       | "^"; {parse_caml_expr0()} e -> e
```

Figure 2.1: A CAML example: arithmetic expressions

The antiquotation facility has to be programmed whenever it will be needed. However, having to do this leaves the syntax of the escape in the hands of the programmer. Usually the abstract syntax has fewer nonterminals than the concrete syntax and most nonterminals in the abstract syntax will have an escape programmed in.

The parser stream pattern language has two pattern forms for specifying iteration in the stream (repeated patterns), one pattern for 0-lists and another for 1-lists. The result returned by these patterns are CAML lists of the repeated items, making it easy to parse textual lists into the list data structure.

A complete grammar is converted into a Yacc specification (Yacc is described in section 2.2.1). This imposes some limitations on the form of the grammar. It must be possible to reduce all higher order parsers to a first order parser and the grammar generated from this parser must be LALR(1).

One of the more interesting features of 'parsers as stream destructors' in Mauny's paper is the use of higher order parsers to write multi-level grammars. A multi-level grammar uses one grammar to derive another grammar which is used for parsing the input. Unfortunately, the use of Yacc as a parser generator and the ensuing limitations mean that it is not possible to use higher order parsers in a general fashion, nor is it possible to use them to build modular grammars. It is possible, however, to use explicit calls to other parsers to mix languages, and an example of this is using the system parser for parsing escapes. The same 'trick' can be used to mix other languages.

2.1.5 Elder I

Elder I is a system for experimenting with languages. The weaknesses of this system motivated the work leading to the modular grammars presented in this thesis. Elder I is based on Lisp. Language processors are built as either interpreters or collections of macros, or both. Programs in the languages are represented as Lisp s-expressions. The program is either interpreted or transformed into Lisp by macros.

An important component of the system is a 'rules' macro that allows functions and macros to be written as rewrite rules rather than using the standard lisp defining forms. The rewrite rules are a little like the pattern matching in Miranda or ML. ELDER I has been used to build implementations of *me too*, CSP, Petri Nets, Unity, generic declarations and systems dynamics models.

ELDER I has several weaknesses. It is a library of Lisp macros and functions. The implemented languages must have a Lisp s-expression syntax as functions and macros that manipulate program expect this. It is possible to bolt a parser in front of the Lisp part of the language processor, and ELDER I contains a parsing language to support this, but reasoning about the program is restricted to its Lisp syntax. For example, the *me too* language has a set comprehension operator which allows expressions like `{x*x | x:{-5..5}}`[1]. This expression has the Lisp syntax

```
(setof (* x x) (range -5 5))
```

and all the parts of the implementation that handle set comprehensions use this form.

An interesting observation made while using the system was that some languages worked together and others did not. Those that worked together were implemented by macros and were largely independent. Making the language fragments work together was rather *ad hoc*. Some language fragments, like generic declarations, are sufficiently general to work well with most other fragments but unfortunately this is not true of many other fragments. This observation lead to the development of modular attribute grammars which have a better compositional model and work better together.

2.2 Parsing

2.2.1 Yacc

Yacc [Johnson 1975] is a well known standard parser generator. It generates an LALR(1) parse table and can employ operator precedence and associativity information to resolve shift-reduce conflicts. When a production in the grammar is recognized a reduction occurs and the action associated with the production is executed. This action can update global variables and use the values associated with the subparts of the production to construct a new value for the whole production. Yacc only provides facilities for constructing values for a nonterminal from the values of the subparts, so it only allows synthesized attributes, i.e. it is S-attributed. S-attribution is useful for constructing a parse tree but too constraining for more sophisticated computations.

There is no help for creating modular descriptions. I include Yacc here because it (and related parser generators like Bison) are used by the compiler-writing community as a standard.

2.2.2 Parse-Tree Annotations

NewYacc [Purtilo & Callahan 1989] is an extension to Yacc (see above) that associates expressions in a string based language with the grammar rules. Like Yacc, a NewYacc program is 'compiled' into a C parser that calls a custom lexical analyser. Unlike Yacc, there is a string-valued language that is used to perform the attribute calculations, and this, too, is compiled into C. While not as powerful as some other compiler generating systems, it allows the definition of multiple synthesized attributes and limited, but

[1] The set of squares of the numbers −5 to 5, i.e. {0,1,4,9,16,25}.

useful, use of inherited attributes. The original Yacc is only really well suited to the synthesis of a single attribute. NewYacc is intended to be used as a 'glue' tool that makes it easy to extract information from a program or to make minor modifications to a program, e.g. add extra instrumentation code to track certain events. The support for multiple synthesized attributes extends to the top level. The user of a NewYacc program can elect to compute any of the top level attributes at run time. A consequence of this is that the grammar definition tends to be reused: for a related set of programs a single source is maintained which contains one copy of the grammar and the several programs that calculate useful things about the program.

2.2.3 Grammar Inheritance

Aksit *et al.* have implemented a parser generator system based on *grammar inheritance* [Aksit *et al.* 1990]. The idea of grammar inheritance is analogous to inheritance in object oriented languages—a grammar may inherit rules (grammar productions and their actions) from a *super-grammar* just like an object class inherits methods from a super-class. This organization allows a grammar to be reused and extended by deriving new grammars which inherit the bulk of an existing grammar and just add their own bits. The management system ensures that if a grammar is changed then all the grammars which inherit from that grammar are updated to reflect this fact. Grammar inheritance is independent of class inheritance and may be implemented in any system, but the authors feel that it is best combined with class inheritance.

Aksit *et al.* do *not* mention multiple grammar inheritance which I would have thought to have been an interesting extension. With multiple inheritance it would be possible to define a new grammar by inheriting from two grammars and in so doing combine the two languages. It appears that each grammar in the system forms a complete language without any unresolved references or undefined parts rather than a language fragment.

2.2.4 SDF

The syntax definition formalism SDF [Heering *et al.* 1989a] was developed to overcome the restrictions of the term-oriented syntax of the algebraic specification language ASF [Bergstra *et al.* 1989]. SDF is not restricted to use with ASF but the literature does not seem to contain any references to the use of SDF with another formalism. Central to the integration of SDF with ASF is the correspondence between the abstract syntax and algebraic signatures. A signature consists of declarations of sorts (types) and constants and functions over these sorts.

SDF uses one definition to define both the abstract syntax and the concrete syntax. The lexical and context-free syntax are handled in a uniform way. The translation from parse trees to abstract syntax is implicit. The abstract syntax is represented by a signature which is derived from the concrete syntax production. Arbitrary context free grammars are allowed. Ambiguity may be resolved by priority and associativity rules. Together, these eliminate much of the need for 'structuring' nonterminals. This means that the concrete grammar is quite close to the abstract syntax, making the automatically derived abstract syntax easy to use and uncluttered with irrelevant structuring information.

In addition to the normal sorts (like INTEGER, STATEMENT and EXPRESSION), there are derived sorts which are convenient for describing lists of items:

```
module Layout
  exports
    lexical syntax
      [ \t\n]          -> LAYOUT
      "%%" ~[\n]* "\n" -> LAYOUT

module Identifiers
  imports Layout
  exports
    sorts ID .
    lexical syntax
      [a-z] [a-z0-9]* -> ID

module Expressions
  imports Identifiers
  exports
    sorts EXP
    context-free syntax
      ID                ->  EXP
      EXP "+" EXP       ->  EXP  left
      EXP "*" EXP       ->  EXP  left
      "(" EXP ")"       ->  EXP  bracket
  priorities
    EXP "+" EXP -> EXP  <  EXP "*" EXP -> EXP
```

Figure 2.2: Simple SDF example

s^* is the sort of lists of 0 or more items of sort s.

s^+ is the sort of lists of 1 or more items of sort s.

$\{s\ t\}^*$ is the sort of lists of 0 or more items of sort s, and the items are separated by the symbol t.

$\{s\ t\}^+$ is analogous to $\{s\ t\}^*$.

The lexical syntax is given in a different section to the context free syntax, but it works in a similar way. The syntax defines a concrete form for the underlying signature. Figure 2.2 shows a small example of an SDF description of arithmetic expressions with variables. It demonstrates the use of both lexical and context-free syntax. The `Layout` module defines the lexical syntax of things that should be ignored: whitespace and comments from "`%%`" to the end of the line. The `Identifiers` module imports the `Layout` module and defines identifiers. More features of SDF are illustrated by the expressions module, which imports the identifiers (and hence the layout) and defines expressions. The imported module is used to build new syntax. The operators "`+`" and "`*`" are defined as left associative and given relative priorities, which makes the expressions unambiguous. Parenthesized expressions are declared with the "bracket" attribute which means that the syntax has no semantic significance and thus there is no corresponding abstract syntax (algebra term) for this concrete syntax.

The parser used by SDF has to accept *any* context free grammar. A variant of Tomita's parser [Tomita 1980] is used. The algorithm works from any shift-reduce parsing table, e.g. LR(0), LALR(1). Whenever the parser is in a configuration where there is a shift-reduce or reduce-reduce conflict a parser with its own stack is spawned for each of the conflicting actions. All of the spawned parsers work in lock-step, accepting one symbol at a time. A more accurate parser table generating algorithm will produce less conflicts and result in faster parsing. The result of parsing like this is a collection of parse trees. SDF chooses one of the parse trees on the ambiguity resolving criteria, e.g. priority.

The context free grammar is converted into a signature. Roughly, each rule in the grammar is converted into a function or constructor. Bracket rules and lists are handled specially.

2.3 Attribute grammars

Attribute grammars (AG) are a declarative formalism for describing the syntax and semantics of a programming language together [Knuth 1968]. An attribute grammar is based on a context free grammar that describes the syntax of a language. Each production in the a context free grammar is augmented with some rules that specify the value of some attributes that are associated with that production. Typically one of the attributes will be the translation T of the program into another language $T : L_1 \rightarrow L_2$. AGs have the advantage that they have a simple formal model and there are known techniques for converting them into efficient compilers. With a lazy functional language they can even be executed directly. I use this in chapter 5.

A problem with an attribute grammar description is that it grows too large to be manageable. Apart from the context free grammar there is nothing to help the writer organize the definition, and there is no way to break a large definition into modules or informal smaller units.

2.3.1 LINGUIST

LINGUIST-86 [Farrow 1982] is a commercially developed translator writing system based on attribute grammars. The attribute grammar is transformed into an alternating-pass attribute evaluator. The Attributed Parse Tree (APT) is kept on secondary storage. Alternate passes traverse the APT from left-to-right or right-to-left. The evaluator only writes out the values attributes from one scan if they are needed in a subsequent scan (i.e. temporary attributes are discarded). The importance of LINGUIST is that the evaluators are efficient enough to run on microcomputers hence demonstrating that attribute grammars are a realistic tool.

An optimization, *static subsumption*, eliminates many copy rules in the generated evaluator. Generated compilers process input at speeds which compare favourably with commercial compilers. Measurements show that the evaluators are I/O bound.

2.3.2 Chameleon

Chameleon [Mamrak *et al.* 1985] is a comprehensive data translation system with the following characteristics: it is derived from a *formal model* of the translation task; it supports the *building* of translation tools; it supports the *use* of translation tools; and 4) it is accessible to its targeted end-users. A software architecture to achieve the translation capability is fully implemented. Translators have been generated using the architecture.

The generation of translators is directed by defining a *standard form*. An item in a particular *variant form* may be *translated up* into the standard form, and standard form objects may be *translated down* into any variant form. The generation of translators for these translations is assisted by the system. Thus given N variant forms and a standard form it is possible to translate a text in one variant form into any other variant form by translating to the standard form and back down to the other variant. N definitions (of variant forms) gives N^2 translators.

Chameleon has been applied to translating documents between mark-up languages.

2.3.3 Reversible attribute grammars

Reversible attribute grammars [Yellin & Mueckstein 1986] are attribute grammars for which it is possible to automatically derive an inverse mapping. Given an attribute grammar defining a translation $T : L_1 \rightarrow L_2$ from language L_1 to language L_2, it is shown how to automatically synthesize the inverse attribute grammar specifying the inverse translation $T^{-1} : L_2 \rightarrow L_1$. It is necessary to restrict the attribute grammar to a *restricted inverse form* (RIF) attribute grammar. Yellin & Mueckstein give an algorithm for the inversion. Extensions to RIF grammars and methods of converting non-RIF grammars to RIF grammars are considered.

The unit of reuse is the grammar—it defines *two* translations. This work is used in Chameleon (section 2.3.2).

2.3.4 MAGGIE

Specifications written as attribute grammars tend to become large and complex. This is inevitable because the AG formalism does not provide for any modularity or abstraction. Dueck and Cormack review these problems and propose modular attribute grammars (MAGs) as a solution to these engineering problems [Dueck & Cormack 1990]. Their system, called MAGGIE, is a processor that combines MAGs to create the equivalent monolithic attribute grammar.

A MAG defines one or more output attributes from zero or more input attributes. The complete AG is the collective effect of several MAGs. One MAG might handle declarations and the environment, another might be concerned with the allocation of storage etc. An essential feature is that modular attribute grammars specify the attribute calculations for sets of productions rather than individual productions. This means that the behaviour of the attributes can be specified in detail for the important parts of the syntax for the module, and in a general fashion for the other productions in a grammar.

2.3.5 Higher order attribute grammars

Higher order attribute grammars [Vogt *et al.* 1989] make attribute grammar processing more powerful by allowing parts of the derivation tree to be defined by attribute expressions. These parts of the tree are parsed as empty strings and later calculated like ordinary attributes. Then the attributes synthesized by the new piece of tree are available for use like ordinary attributes. This feature has two main uses:

- It allows for an easy implementation of macro features in the language.
- It supports phase oriented modularity as follows. Each phase synthesizes a translation attribute. This value is then 'grafted' into the derivation tree. This new tree synthesizes the translation attribute for the second phase and so on.

The technique is more powerful than these examples suggest. The number of higher order 'expansions' is a run time property of the translator depending on the program being translated. This is richer than a fixed number of phases.

2.3.6 Attribute Coupled Grammars

Attribute coupled grammars (ACGs) are one solution to the modularity problem of attribute grammars. Consider an attribute grammar α which defines a translation from one language into another language. The translation is a mapping from derivation trees of the one language ($G1$) to derivation trees in the other language ($G2$). This sort of attribute grammar is called and *attribute coupling* in [Ganzinger & Giegerich 1984] and is denoted $\alpha : G1 \rightarrow G2$. The attribute coupling is a description (i.e. program text), its meaning is a translation from the term algebra T_{G1} of the derivation trees of $G1$ to term algebra T_{G2} of $G2$, denoted $\mathbf{T}\alpha : T_{G1} \rightarrow T_{G2}$. The key point of ACGs is that both the domain and range of the translation are programs represented by term algebras, so the translations may be composed. For $\alpha : G1 \rightarrow G2$ and $\beta : G2 \rightarrow G3$ the composition is $\mathbf{T}\alpha \circ \mathbf{T}\beta : T_{G1} \rightarrow T_{G3}$. A complete compiler may be expressed as a composition of several phases.

Attribute couplings can be composed at a descriptional level. Given attribute couplings $\alpha : G1 \rightarrow G2$ and $\beta : G2 \rightarrow G3$ there exists an attribute coupling $\gamma : G1 \rightarrow G3$ such that $\mathbf{T}\alpha \circ \mathbf{T}\beta = \mathbf{T}\gamma$. This γ can be constructed from α and β by an operation called descriptional composition, written $\alpha©\beta$. Descriptional composition is an operation on descriptions, i.e. the result is a single attribute grammar which describes the composite translation.

Descriptional composition offers a solution to the problem of the inefficiency of using a large number of intermediate forms. For example, the 5 phase description using 4 intermediate forms may be implemented as two translations with one computed intermediate form.

$$\mathbf{T}\alpha_1 \circ \mathbf{T}\alpha_2 \circ \mathbf{T}\alpha_3 \circ \mathbf{T}\alpha_4 \circ \mathbf{T}\alpha_5 = \mathbf{T}(\alpha_1©\alpha_2©\alpha_3) \circ \mathbf{T}(\alpha_4©\alpha_5)$$

Giegerich [Giegerich 1988] investigates the properties of attribute grammars under descriptional composition. For example, the above translation may be chosen because the two translations are efficient classes of attribute grammar.

In summary, attribute coupled grammars describe phase-oriented decompositions. Descriptional decomposition allows the implementation to be modularized independently from the description.

2.4 Non-specific tools

Some languages and language systems have features that can be used for prototyping new languages even though they were not designed for this purpose. An example of a system that has been widely used like this is the C language pre-processor. The preprocessor is a macro language that is usually used for conditional compilation and inter-module communication of data types and constants by textually 'including' header files that contain the shared definitions. A simple example of this feature's subversion to linguistic experimentation is the following set of macros. Each `#define` defines a name (identifier) and some text that is substituted for the name wherever it appears as a complete identifier in the program text.

```
#define IF     if(
#define THEN   ) {
#define ELSE   } else {
#define ENDIF  }
```

Using these macros, the input text

```
IF x<0 THEN
  x=5;  y=z;
ELSE
  x=x+1;  z*z;
ENDIF
```

is expanded to the legal C text

```
if( x<0 ) {
  x=5;  y=z;
} else {
  x=x+1;  z*z;
}
```

The problems with this approach are:

- Only certain forms may be redefined. The C preprocessor only allows text of the form "`identifier`" or "`identifier(expression,expression,...)`" to be expanded. Other syntactic constructs cannot be handled.

- The macro language is limited by the macro expansion process—for example, it impossible to devise macros that will automatically define a new variable in the closest enclosing scope because macros define replacement text.

- This mechanism is insecure. There is nothing to prevent the programmer from using the native syntax or from using the new macros in ridiculous places.

These comments also apply to the use of operator definitions like those allowed by Ada and C++ and by Smalltalk methods. For example, in C++ the programmer may *overload* most of the operators in the language. Interestingly, this includes the function call operator "α`(`β`)`" and the array indexing operator "α`[`β`]`". Even though these operators may be overloaded to handle new types, their versatility is limited as it is impossible to make the "... `[`...`]`" operator support, say, Miranda's list comprehension: "`[`*expr*`|`*pattern*`<-`*expr*`]`". Stroustroup, in his C++ book [Stroustrup 1986], gives the advice that operators should only be overloaded in algebraically conservative ways. For example, "`*`" and "`+`" should always be overloaded with operations which have the relative properties of multiplication to addition.

2.5 Summary

This chapter has surveyed programs and systems designed to assist with three aspects of language implementation—treating programs a data objects, parsing the textual form of notations and the use of attribute grammars to describe syntax directed computation. In many cases there has been little or no attempt to control the complexity of the description by allowing it to be composed from modules. Those systems that do provide assistance usually have the weakness that the language is considered to be a single entity. Each module is concerned with an entire language, for example each of Chameleon's variant forms, or each attribute coupled grammar phase. It is evident that the modularity properties of language descriptions can be improved.

3 Language fragments

The purpose of the chapter is to investigate what happens when a language definition is broken into pieces with a view to re-assembling the pieces later, or putting them together in other ways to create new languages that share some part of their heritage and flavour with the original language. This chapter describes some experiments that investigate combining forms for programming language fragments. By combining form I mean the thing that takes two (or more) language fragments and yields a single new fragment that somehow is the 'sum' of the parts. The objective of the experiments is to develop a style or notation that can express individual language fragments and combine those language fragments to make new fragments or even complete languages. The experimentation proceeded by writing language fragments and fragment combining mechanisms using a variety of approaches or computational styles. The same fragments were written in each style to allow comparison of the strengths and weaknesses between the approaches. A common task was set: to express two languages, say $L1$ and $L2$, and to combine a language fragment, say C, with each of $L1$ and $L2$ to produce new languages $L1 \oplus C$ and $L2 \oplus C$.

The success of each approach can be judged by considering some questions about how $L1$, $L2$, C and $\oplus$ are expressed: is the expression concise? is it understandable? does it allow the addition of more fragments by the same means (i.e. is it uniformly extensible)? how do the components interact, and is this evident in the structure of their specification?

Three approaches are compared:

- Functional. Each language is represented by an evaluation function. Combination of fragments involves the composition of higher order functions.
- Object-Oriented. The abstract syntax is represented by objects. A program is evaluated by sending a message to the object that represents the program.
- 'Algebraic'. A language definition consists of a type-like description of the abstract syntax and a collection of rules for reducing the syntax to a result. Combining languages extends the syntax description and combines the rule sets by adding rules to complete the set.

The rest of this chapter comprises a description of the three languages chosen to correspond to $L1$, $L2$ and C, followed by a description of each approach to building combining forms. Then some performance measurements are given. This is followed by a discussion on the merits and issues raised by the three approaches. First, however, I describe what I mean by abstract syntax, the internal representation of a program.

3.1 Abstract syntax

In the rest of this chapter I will discuss various programs that manipulate other 'subject' programs. The subject programs need to be represented as a data structure to be amenable to manipulation. This section describes how a program may be represented as an abstract syntax tree and how that tree has an unambiguous linear form when written in a term-language.

The abstract syntax identifies the essence of a piece of program, namely the type or sort of the piece and what its components are. A program fragment is a member of one of a set of *sorts*. Each sort represents some class of program constructs, like expressions, statements or definitions. Within each sort there are usually several variants. Each variant has a unique *operation symbol*. The operation symbol has an *arity* which describes the sort to which the symbol belongs and the sorts of the components. For example, an assignment statement is a variant of a statement and it has a variable and an expression as components, and we will give it the operation symbol 'Assign'. Thus we say Assign has the arity $variable \times expression \rightarrow statement$. As we prefer to write the abstract syntax in a generative style, saying 'a statement can be an assignment statement', the abstract syntax for the assignment statement is written with the operation symbol in front of the components which are listed in parentheses, like this:

statement → Assign(variable,expression)

As a very small but complete example, this is an abstract syntax for a language of expressions for adding and multiplying integers:

comb → Add(comb,comb)
comb → Mul(comb,comb)
comb → Num(integer)

There are two sorts in this definition. The integer sort is built in. Sort comb is defined with three operation symbols. Comb may be thought of as a discriminated union of product types where each operation symbol tags a different member of the union. A program may be written in a tree or linear form by (a) selecting the appropriate operation symbol for the fragment of program and then (b) replacing all the sort names for this symbol with the tree or linear form of the corresponding subparts of the program fragment. The concrete syntax (the rules describing what the program actually looks like) might have infix operators, parenthesized expressions and associativity and precedence rules for associating the correct operands with the operators. In the abstract syntax all these details have been removed and any potential ambiguities in the concrete syntax have been resolved—the abstract syntax cannot be ambiguous because of its restricted form. The linear and tree forms of the abstract syntax for the sentence "`2*3+4*5`" is shown in figure 3.1.

3.2 Subject languages

Three subject languages were chosen for the experiment to find combining forms: 'Combinations', 'CCS' and 'Parameters'. Combinations is an expression language similar to the arithmetic expressions available in most programming languages. 'CCS' is a subset of Milner's Calculus of Communicating Systems [Milner 1980]. Parameters is a language fragment that introduces named items in using a *let*-like syntax. Combinations and CCS were chosen because they require different evaluation mechanisms; this is explained below. This choice sets the challenge of inventing a language combination method that is quite general. The rest of this section describes these languages.

Abstract syntax:

comb → Add(comb,comb)
comb → Mul(comb,comb)
comb → Num(integer)

Sentence: "2*3+4*5"

Linear form:

Add(Mul(Num(2),Num(3)),
    Mul(Num(4),Num(5)))

Abstract syntax tree:

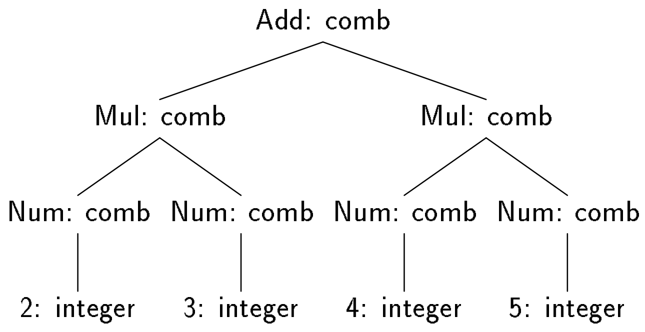


Figure 3.1: Abstract syntax

3.2.1 Combinations

The combination language is a simple expression language. Terms in the language are either values or combinations of values that are combined with an operator, for example addition. An abstract syntax for combinations over numbers has already been given:

comb → Add(comb,comb)
comb → Mul(comb,comb)
comb → Num(integer)

This is intended to capture the flavour of expressions like $(2+3) \times 5$ which has abstract syntax

Mul(Add(Num(2),Num(3)), Num(5))

An expression in this language can be evaluated by applicative order evaluation: to evaluate a comb expression, the subexpressions are evaluated to produce *values* which are then converted into new values by application of the operation. The type of such an evaluator is $eval : comb \to value$.[1] An evaluator for the comb language is a simple function that does a case-analysis on the input. Each case selects the appropriate action for the input form. The function is called recursively to evaluate the subparts.

[1] Throughout I consider a type to be a set of values. Some types, e.g. integers, are usually treated as primitive. Other types are constructed from simpler types:

$t_1 \cup t_2$	Union. The type comprises values of t_1 and t_2.
$t_1 \times t_2$	Cartesian product. Pairs of values, the first of type t_1, the second of type t_2.
$t_1 \to t_2$	Map. The function taking values of type t_1 and yielding values of type t_2.
$name(t_1, \ldots, t_n)$	Algebraic type, constructed from $t_1, \ldots t_n$.
$\text{list}(t)$	The type of lists of items of type t (an algebraic type)
$[t]$	Same as $\text{list}(t)$
$name : t$	The entity *name* has type t

```
eval_comb(form) =
  case form of
    Add(e1,e2)  → eval_comb(e1) + eval_comb(e2)
    Mul(e1,e2)  → eval_comb(e1) × eval_comb(e2)
    Num(n)      → n
  end
```

The case statement matches form against pieces of abstract syntax, in the order written, and returns the value specified at the sharp end of the matching arrow.[2]

3.2.2 CCS subset

The Calculus of Communicating Systems is a mathematical notation, invented by Robin Milner, for describing the behaviour of concurrent systems. I will give a brief overview of CCS; it is not the purpose of my work to investigate CCS or concurrency, I use a subset of CCS as an example of a language that requires a different concept of 'running' to the combinations language. Interested readers should refer to [Milner 1980] or [Milner 1989].

A system is described as an *agent*. An agent is a system whose behaviour consists of discrete *actions*. An agent may be atomic or decomposed into sub-agents acting concurrently and interacting.

Actions are taken from a set of labels $\mathcal{L}$. Labels are either names, or complimentary names (co-names). Co-names are written with an overbar and the co-name of a co-name is the name, i.e. $\overline{\overline{foo}} = foo$. Additionally there is the *silent action* τ. An action accomplishes the transition of a system from one state to another state. The transition from state P to state Q by action l is written

$$P \xrightarrow{l} Q$$

The CCS language has seven combinators. The agents that these combinators denote are defined by the transitional semantics, which is a collection of inference rules for inferring the possible transitions of the compound agent. One of the classic examples from CCS (and also CSP [Hoare 1985]) is that of the vending machine. Consider a chocolate vending machine. The machine has two slots, one each for 1p and 2p coins, two selector buttons to select a small or large chocolate, and a collection tray where the chosen chocolate is delivered. A small chocolate is selected by inserting a 1p, pressing the 'small' button and taking the chocolate from the tray. The vending machine, VM, is

[2] A value matches the pattern if it has the same operation symbols in the same places. Other names in the pattern are bound to the subparts of the form corresponding to the same positions, e.g. if form is Mul(Num(2),Num(99)) the second pattern would match and e1 would be bound to Num(2) and e2 to Num(99). It is an error if no case matches which would happen if form was of the wrong type or there was another comb operation symbol. The pattern which consists of just a name will always match. A pattern can contain more than one operation symbol. For example the evaluator might be 'optimized' by adding a new pattern to avoid addition to zero as the first pattern:

```
    Add(Num(0),e2)  → eval_comb(e2)
```

So many functions have a case form as their body that it is convenient to write the pattern matching into the function definition, like this:

```
eval_comb(Add(e1,e2))  = eval_comb(e1) + eval_comb(e2)
eval_comb(Mul(e1,e2))  = eval_comb(e1) × eval_comb(e2)
eval_comb(Num(n))      = n
```

defined as an agent which interacts with its environment at its five ports `1p`, `2p`, `small`, `large` and `collect` in the way described.

$$VM \stackrel{\text{def}}{=} \texttt{2p.large.collect.}VM + \texttt{1p.small.collect.}VM$$

The vending machine accepts either a 1p or a 2p. This commits it to allowing only one of the buttons to be pressed, after which it issues the chocolate. Another agent can be defined to model a user. The user *SLIM* takes a small chocolate bar and then does nothing:

$$SLIM \stackrel{\text{def}}{=} \overline{\texttt{1p}}.\overline{\texttt{small}}.\overline{\texttt{collect}}.Nil$$

The user and vending machine are made to interact by composing them:

$$SYSTEM = VM \mid SLIM$$

The actions of the system can be found by rewriting the system according to the transitional semantics. The rule that is used most in this example is the rule that two agents in composition may interact showing only the silent action to the observer.

$$\begin{array}{ll} & VM \mid SLIM \\ = & (\texttt{2p.large.collect.}VM + \texttt{1p.small.collect.}VM) \mid (\overline{\texttt{1p}}.\overline{\texttt{small}}.\overline{\texttt{collect}}.Nil) \\ \xrightarrow{\tau} & (\texttt{small.collect.}VM) \mid (\overline{\texttt{small}}.\overline{\texttt{collect}}.Nil) \\ \xrightarrow{\tau} & (\texttt{collect.}VM) \mid (\overline{\texttt{collect}}.Nil) \\ \xrightarrow{\tau} & (VM) \mid (Nil) \\ = & VM \end{array}$$

We can see that after *SLIM* has left the vending machine it can continue as usual. This interaction may be reduced to

$$VM \mid SLIM \xrightarrow{\tau} VM$$

The mathematical concrete syntax used by Milner is inconvenient for computer use. The corresponding abstract syntax used in the following discourse is given in figure 3.2

A CCS program is executed by symbolically reducing it to a simpler program. The expansion law shows how an agent may be rewritten in *standard concurrent form*. The program is executed by reducing it to standard concurrent form (by the expansion law) and choosing one of the prefixes. This process is repeated. This is execution by reduction rather than execution by evaluation, so intermediate results in the execution are valid expressions in the language. The type of the reduction step is $reduce : agent \rightarrow agent$, and the type of the choice step is $choose : agent \times choice \rightarrow agent$. These can be composed into one step of type $step : agent \times choice \rightarrow agent$. This differs from the type of the evaluator for combinations in that the result type is part of the domain of the evaluator function, allowing the reduction step to be re-applied.

3.2.3 Parameters

Both of the previous languages have meanings in their own right. They are complete definitions of a language. The parameters language is not complete in this sense. It extends a language by providing constructs for naming and reusing items in the language. The Parameters language fragment introduces two constructs into another language. The two constructs are the introduction of a name for an expression, and the use of that name:

		Abstract syntax	Milner's notation
agent	$\rightarrow$	Nil	Nil
	$\mid$	Prefix(label,agent)	$label.Agent$
	$\mid$	Sum(agent$_1$,agent$_2$)	$Agent_1 + Agent_2$
	$\mid$	Compose(agent$_1$, agent$_2$)	$Agent_1 \mid Agent_2$
	$\mid$	Restrict(agent, labels)	$Agent\backslash labels$
	$\mid$	Relabel(agent,function)	$Agent[function]$
	$\mid$	If(expression,agent$_1$,agent$_2$)	**if** $expression$ **then** $Agent_1$ **else** $Agent_2$
label	$\rightarrow$	Out(labelname)	$\overline{labelname}$
	$\mid$	In(labelname)	$labelname$
	$\mid$	Tau	τ

Figure 3.2: CCS abstract syntax and concrete syntax

expression $\rightarrow$ Let(name, expression$_1$, expression$_2$)
expression $\rightarrow$ Param(name)

The meaning that we will associate with this construct is that within expression$_2$ the form 'Param(name)' is equivalent to expression$_1$.

3.3 Functional formulation

The functional approach to describing languages is by writing an interpreter for the language in a (largely) functional style. The language is embodied in the interpreter. This approach is widely used in operational semantics (e.g. the Lisp 1.5 metacircular interpreter [McCarthy *et al.* 1965]) and in denotational semantics (e.g. the Scheme language semantics in [Rees & Clinger 1986]). My approach more closely resembles the former in that data structures may be used to represent some semantic domains. This might be compared with completion semantics [Henson & Turner 1982].

After writing an interpreter as a definition of a language I develop some language level operations, like $\oplus$ introduced earlier. Since the languages are embodied in the functions that are the language (fragment) interpreters this must be done by means available to functions: composition, abstraction and application. Consequently the functional approach is mainly concerned with writing the language embodiments (the interpreters) in a style which allows the apparatus of functional programming to provide the desired operations for language fragment combination.

Because a language is represented by its interpreter function then a language fragment that is built from other fragments is a higher order function. The *language* $\longleftrightarrow$ *function* duality extends to higher order functions like this:[3]

[3]In some ways this misses the boat. I don't actually make an operator like $\oplus$ that creates a new language $L1 \oplus C$, rather we create the operator $(\oplus C)$ which can be applied to a language $L1$ in postfix form: $L1(\oplus C)$. So I have not actually extracted $\oplus$. This would be difficult to do because we don't have enough information—after all, the $\oplus$ 'map override' operator which may be defined as

$$f \oplus g = \{x \mapsto y \mid x \in dom\, f \cup dom\, g;\ y = \text{if } x \in dom\, g \text{ then } g(x) \text{ else } f(x)\}$$

needs the domains of the argument functions visible to *construct* $f \oplus g$. Working with intensional functions with infinite domains rather than finite extensional functions precludes this possibility.

language fragment	$\longleftrightarrow$	function (interpreter)
fragment operation	$\longleftrightarrow$	higher order function
fragment that extends another fragment	$\longleftrightarrow$	abstraction—a higher order function parameterized by the interpreter function for the subject language

To develop these ideas I take a small language fragment (called Prim2) that allows a primitive function of two arguments to be called with the values of two subexpressions. The abstract syntax is

comb → Prim2(primitive, comb_1, comb_2)
primitive → Divide
primitive → *some other primitives*

This fragment is intended as an extension to a base language that defines other types of expressions, like the combinations language.

3.3.1 Extending interpreters

The prim2 fragment has to be written as a function. Logically, as an extension mechanism, prim2 is a mapping from a subject language to a new language that is the subject language augmented by the new operations that it provides. Because these languages are embodied in their interpreters the function for prim2 is a higher order function. The augmented language interpreter either interprets forms that it knows about or passes on the job to the original interpreter. The original interpreter is a function that takes a piece of abstract syntax and returns the value denoted by the abstract syntax. It inspects the argument provided and decides how to calculate the result. If the evaluation requires the value denoted by a subpart of the program the function calls itself recursively. The interpreter int for the Comb language is a typical example. It looks like this:

```
int(form) =
   case form of
      Add(e1,e2)  → int(e1) + int(e2)
      Mul(e1,e2)  → int(e1) × int(e2)
      Num(n)      → n
   end
```

This function takes a program and returns a value. Its type is *program* → *value*.

Now we attempt to write a function that extends this (or another) interpreter with the Prim2 construct. The function is called add_prim because it extends a base language and it is a higher order function. It takes a function and returns a function of the same type, so its type is (*program* → *value*) → (*program* → *value*). This is the first try:

```
add_prim2(base) =
   new where
         new(Prim2(fun,e1,e2)) = apply_prim(fun, new(e1), new(e2))
         new(other_form)       = base(other_form)

apply_prim(Divide, n, m) = n/m
...

new_int = add_prim(int)
```

The function new_int is the interpreter for the extended language. It is formed by calling add_prim with another interpreter (in this case int) as a parameter. Add_prim returns the interpreter for the extended language, the function new. New works in a similar manner to int, it performs a pattern matching case analysis on the input, with one of two possible outcomes. Either the input form is the new language feature, which is evaluated by applying the primitive function to the values for the operands, or the form is not recognised. In this case the original function, base (which in this case is int), is called to evaluate the form.

This solution works well except for the case where a new piece of abstract syntax is inside an old piece. In this case the prim2 function calls base which calls *itself* to handle the subform. We really want the base function to call the new, augmented function, but this is not possible without modifying the base function. It is not satisfactory to alter the base function, i.e. int to call new_int instead of itself because when int was written it was not known that it would be extended by prim2 and in any case doing so would preclude using int by itself and with any further extension. The base function might be the subject of extension by several fragments like prim2, so any single replacement for the recursive call site(s) will be inadequate. The problem is a matter of binding times—if the function for the recursive call is bound (decided) too early then the decision gets in the way of further use of the fragment.

A general solution is to postpone the decision and communicate the decision using a parameter which I call self because this reminds the reader that the call is recursive. Each interpreter fragment takes the additional parameter and each call to another interpreter, including self, passes the additional parameter. The effect of this is to delay the binding to the latest possible moment—the call. Rewriting int and the local function new (in add_prim2) to use the self mechanism gives these definitions:

```
int′(self, form) =
   case form of
      Add(e1,e2)  → self(self, e1) + self(self, e2)
      Mul(e1,e2)  → self(self, e1) × self(self, e2)
      Num(n)      → n
   end

add_prim2′(base) =
   new where
         new(self, Prim2(fun,e1,e2)) = apply_prim(fun, self(self,e1), self(self,e2))
         new(self, other_form)       = base(self, other_form)
```

An interpreter for a language is created by instantiating self, 'tying the knot' in a sequence of calls to make the calls recursive. An interpreter for the base language is constructed by calling int′ with self set to int′. This interpreter behaves like the original interpreter because wherever the original interpreter called itself explicitly the open version calls self:

```
interp1(program) = int′(int′,program)
```

The augmented language int′⊕prim2′ is constructed like this:

```
i2f = add_prim2′(int′)
interp2(program) = i2f(i2f,program)
```

3.3.2 Types and recursion

The types of these functions are a no permitted in some type systems. The trouble is that the type of a function like int′ is recursive. The first parameter, **self** has the same type as the whole function! If α is the type of int then

$$\alpha = \alpha \times program \rightarrow value$$

This is called the non-shallow type problem and is a well known problem of the Hindley-Milner type system which is the basis of the type systems of Miranda and ML [Field & Harrison 1988]. The situation can be improved slightly by using the Y combinator. The Y combinator, defined as Y(f)=f(Y(f)), is a mechanism for introducing recursion into the pure lambda calculus. A recursive function is written with the recursive call as a parameter, like int′ but curried[4].

```
int''(self) =
   dispatch
   where
      dispatch(form) =
         case form of
            Add(e1,e2)  → self(e1) + self(e2)
            Mul(e1,e2)  → self(e1) × self(e2)
            Num(n)      → n
         end

interp1' = Y(int'')
```

The other difference is that the recursive call does not explicitly pass on the recursive function. This is done by the recursive definition of Y. The type problems are more easily solved because the only source of non-shallow types is Y which can be specially built into the language and type checker. Using Y makes the definition of add_prim2 more cumbersome because it must be curried and deal with the curried version of int:

```
add_prim2''(base) =
   new
   where
      new(self) =
         dispatch
         where
            dispatch(Prim2(fun,e1,e2)) = apply_prim(fun, self(e1), self(e2))
            dispatch(other_form)       = base_self(other_form)
            base_self = base(self)

i2f = add_prim2''(int'')
interp2' = Y(i2f)
```

[4] A curried function is a function of several parameters that has been rewritten as a function of the first few of those parameters that returns a function of the rest of the parameters. In this example int′, a function of the two arguments (self,form) is curried, turning it into a function of self which returns a function of form.

3.3.3 Dynamic semantics

This far the combining forms have only extended the syntax of the language. The individual fragments have been mainly concerned with organising and sharing the case analysis correctly between the modules. Recourse to the recursive Y operator precludes any language construct in one fragment from altering the interpretation or meaning of parts of another construct.

The explicit handling of the recursive function self allows contextual interpretation. This can be achieved by passing some function other than self in the recursive call to self. The parameters language fragment would be difficult to write without using this mechanism. Recall the parameters fragment:

$$\begin{array}{l}\mathsf{expression} \longrightarrow \mathsf{Let(name,\ expression_1,\ expression_2)} \\ \mathsf{expression} \longrightarrow \mathsf{Param(name)}\end{array}$$

The meaning that is to be given to this construct is that in $\mathsf{expression_2}$ the form 'Param(name)' is equivalent to $\mathsf{expression_1}$. Thus the interpretation of a Param item depends on the surrounding Lets, or in other words, a Let determines the behaviour of a Param that it contains. The correct effect can be achieved by evaluating the body of the Let with a function that checks to see if it is evaluating the new Param name. If this is the case then the defining value of the name is evaluated. In other cases the form is not the new name so the normal interpreter is called.

Like 'self(self, *expr*)' is a metaphor for evaluating *expr*, 'newscope(newscope, *expr*)' is a metaphor for evaluating *expr* with the extended, or contextually parameterized, interpreter. The extended interpreter is contextually parameterized because it has access to the parts of the Let definition. Note that replacing the call 'self(self, v)' with 'newscope(newscope, v)' changes the semantics from a let to a letrec because v is interpreted in a context where n is defined.

```
add_param (base) =
  new where
     new(self, Param(n)) = error("Param not inside Let")
     new(self, Let(n,v,b)) =
        newscope(newscope, b)
        where newscope(self', Param(n')) = self(self, v),           if n = n'
                                         = self(self', Param(n')),  if n ≠ n'
              newscope(self', other)     = self(self', other)
     new(self, other) = base(self, other)
```

In summary, the interpreter is dynamically changed to implement the dynamic semantics of the language fragment.

3.3.4 Evaluation by rewriting: An interpreter for CCS

The add_param function developed in the previous section is suitable for the combinations language because this language evaluates the program to a form that has no residual context. The initial program may have had forms like Let which alter the interpretation of the program fragment which it contains but the computed result does not have this context-sensitive structure. On the other hand, executing a language like CCS which is evaluated by rewriting the program to produce another program does produce a result that has a context-sensitive structure.

A CCS system specification is executed by rewriting it in standard concurrent form (SCF) and then choosing one of the actions that is presented by the standard concurrent form. A system is in SCF form if it is a summation of prefixes. The vending machine is an example of a system in SCF. It is in the form $a.P + b.Q + \cdots + c.R$. A prefix agent is also in SCF because it is the 'sum' of one agent. The evaluator for CCS that is presented now represents a program in SCF as a set of (label, agent) pairs, each representing a Prefix term. The CCS agent is reduced to a set of Prefix terms and the user chooses one of these terms. If the chosen term is a prefix by the silent action (i.e. of the form $\tau.something$) this represents some internal action of the system, otherwise the term is a labeled prefix and the term represents an interaction between the system that the CCS agent is modelling and the user.

Consider the the vending machine customer *SLIM*,

$$SLIM \stackrel{\text{def}}{=} \overline{\texttt{1p}}.\overline{\texttt{small}}.\overline{\texttt{collect}}.Nil$$

This is represented by the abstract syntax Prefix(Out("1p"),*rest*), where *rest* is the abstract syntax for the agent $\overline{\texttt{small}}.\overline{\texttt{collect}}.Nil$, i.e.

Prefix(Out("small"), Prefix(Out("collect"),Nil)).

The prefix agent $a.P$ gives the SCF set $\{(a, P)\}$. This can be written into a skeleton interpreter:

```
scf(self, agent) =
   case agent of
      Prefix(label,agnt) → {(label,agnt)}
      ...
   end
```

Now consider the following agent which is part of the language CCS⊕Parameters:

let $P = b.Nil$ **in** $a.P$

This agent is supposed to be equivalent to the agent $a.b.Nil$. The abstract syntax for this agent is

Let("P", Prefix(In("b"), Param("Q")), Prefix(In("a"), Param("P")))

If we naively run the interpreter on this agent we get the wrong answer:

{(In("a"), Param("P"))}

Pushing the context into the dynamically created interpreter has caused the surrounding context to become invisible. What we want is to get the result

{(In("a"), Let("P", Prefix(In("b"), Param("Q")), Param("P")))}

or something equivalent.

A solution to this is to pass another function to correctly reconstruct the term. This function, which is called ctx (short for *context*), is modified hand-in-hand with self to maintain the invariant

self(ctx, self, form) = interp(no_ctx, interp, ctx(self))

where interp is the top-level interpreter and no_ctx is the empty context, i.e. the identity function.

```
trans(ctx, self, agent) =
   case agent of
      Nil                        → trans_nil()
      Prefix(label,agent)        → trans_prefix(label, ctx(agent))
      Sum(agent1,agent2)         → trans_sum(self(ctx, self, agent1),
                                             self(ctx, self, agent2))
      Compose(agent1,agent2) → trans_compose(self(ctx,self,agent1),
                                                 self(ctx, self, agent2))
      Restrict(agent,labels)     → trans_restrict(self(ctx, self, agent), labels)
      Relabel(agent,function)    → trans_relabel(self(ctx, self, agent), function)
      If(cond,agent1,agent2)     → if eval(cond) then self(ctx, self, agent1)
                                                else self(ctx, self, agent2)
   end
```

Figure 3.3: CCS interpreter dispatch

```
add_param(base) =
   eval where
         eval(ctx, self, exp) =
            case exp of
               Let(name, val, body) →
                  newint(newctx, newint, body)
                  where
                     newint(ctx′, self′, Param(n)) = newint(newctx, newint, val),
                                                      if n = name
                     newint(ctx′, self′, other)       = self(ctx′, self′, other)
                     newctx(form) = ctx(Let(name, val, form))
               other → base(ctx, self, exp)
            end

no_ctx(form) = form

ccs2 = add_param(scf)
ccs_interp(agent) = ccs2(no_ctx, ccs2, agent)
```

Figure 3.4: The add_param higher order function

3.4 Object Oriented

One of the characteristics of object-oriented (OO) programming is that the definition of an operation on an object is associated with the definition of the type of the object. If the same operation, for example printing, is defined for several different types of object then the operation is defined for each type with the definition of that type. The operation usually has a different effect for each type of object, for example a character string can presumably be printed by copying its characters to the output but printing an integer requires that the digits are calculated from the integer's numeric value. The second characteristic of OO programming is that the types are arranged in a hierarchy and the definition of an operation may be *inherited* from types higher in the hierarchy. This allows economy of expression as behaviour shared by several types may be defined once for the several types to inherit. Types in the sense used here are usually called *classes* in the OO paradigm and the operations are called *methods*. An object of a certain class is said to be an *instance* of that class, and the components of the object which are like record fields are stored in *instance variables*. The classes above a class in the class hierarchy, from which the class may inherit methods, are called *superclasses*. The act of invoking an operation for an object is called *sending* a *message* (the name and parameters of the operation) to the object. This terminology is appropriate because the object 'knows' what the appropriate behaviour is by virtue of being an instance of a particular class: the system can determine which definition of the operation to use.

This is in contrast to operation oriented programming where the operation is defined once and selects the appropriate variation of the operation depending on the type of the parameters. When a new type is added to an operation oriented system the system must be modified at all the operations which apply to the new object. In an OO system the new class is simply added to the system piecemeal as it contains all the definitions of the operations which apply to the type, and other operations may be inherited from the superclass(es).

3.4.1 Object oriented program representation

The OO approach was investigated because of its locality of definition. In programming language definition terms this means that if a programming language construct is represented as an object that is an instance of a class representing that particular kind of construct then the semantics of that construct can be defined with its abstract syntax. The hope is that the 'factored out' semantics will avoid the compositional complexities encountered in the functional approach.

Abstract syntax, grammars and OO system structuring are strongly related:

Abstract syntax	**grammar**	**OO**
syntactic category (sort)	nonterminal	independent class
operation symbol & arity	production	subclass
term	string	tree of objects

An object oriented interpreter can be written in the following manner:

- Declare a class for each sort in the abstract syntax.
- Declare a subclass for each variant of the sort. The subclass has an instance variable for each component of the variant and the type of that instance variable is the class corresponding to the sort of the component.
- Declare a method called eval for each subclass.

Taking the combinations language defined as below and applying these rules we obtain the object oriented program in figure 3.5.

comb → Num(integer)
comb → Comb(op,comb,comb)
op → Add
op → Mul

```
class comb

class comb_number: subclass of comb
   instance variables  value:integer
   method eval()
      return value

class comb_comb: subclass of comb
   instance variables  op:operation; e1,e2:comb
   method eval()
      return op.operate(e1.eval(),e2.eval())

class operation

class add: subclass of operation
   method operate(e1,e2:integer)
      return e1+e2

class mul: subclass of operation
   method operate(e1,e2:integer)
      return e1×e2
```

Figure 3.5: Object oriented version of the combinations language

Note that the evaluation method for the operation class has been called operate. It too could have been called eval, but the result of evaluating the operation by itself is a function on integers so it was thought better to have the method perform the appropriate operation on the data rather than synthesize a new object that is a function. Sending an eval message to a comb_comb object causes that object to send an eval message to the two subexpressions and then to send an operate message with the results to the combination's operation.

3.4.2 Object oriented language extension

As mentioned earlier, an object oriented program can be extended to handle items of a different type simply by adding the definition of the new class. This is used to extend the interpreter described in the previous section with the parameters language.

Let us assume that there is a symbol-table class called environment available for our use. It understands the following methods. The details of its implementation are unimportant.

```
class environment
   method extend(name:identifier, value:object)   —add a binding
   method retract()                               —'pop' most recent binding
   method lookup(name:identifier):object          —find the corresponding value
end
```

```
generic add_param (base)

   env : environment

   class let: subclass of base
      instance variables  name:identifier; defn:base; body:base
      method eval()
         env.extend(name,defn)            —note 1
         let result = body.eval() in
            env.retract()
            return result

   class param: subclass of base
      instance variables  name:identifier
      method eval()
         (env.lookup(name)).eval()        —note 1

end add_param

add_param(comb)                           —make the combined language
```

Figure 3.6: Object oriented version of the parameters language

The combinations language can be extended with the parameters language by adding two new subclasses:

```
class let: subclass of comb
class param: subclass of comb
```

However, it would be nice to be able to reuse the parameters language fragment with another language, so the dependence on the comb class needs to be abstracted out. A simple way of doing this is to make the parameters language generic with respect to comb. The parameters language can then be instantiated with the language that is to be extended. The generic facility is a 'compile time' feature and is just a dressed-up form of macro. A simple pre-processor (e.g. the C language pre-processor) could be used to implement the feature in a language that does not already possess the facility.

The generic form add_param is given in figure 3.6. The new forms, let and param work by maintaining an environment that is referenced by the global variable env. Let extends the environment with a new binding for the evaluation of its body. The binding is removed from the environment afterwards. In effect env is a dynamic variable. The param form simply looks up the name in the environment and evaluates the form that it is bound to.

This implementation gives the param language dynamic procedural binding because the caller's environment is used to evaluate the code that was stored in the environment at use time. Changing the two lines marked '*—note 1*' to 'env.extend(name,defn.eval())' and 'env.lookup(name)' respectively would change the implementation to one where values are evaluated in their defining environment at definition time. Another way of doing this is to use closures and this will be described shortly.

```
class closure: subclass of base
  instance variables  body:base; closed_env:environment
  method eval()
    let saved_env = env in
      env := closed_env
      let result = body.eval() in
        env := saved_env
        return result

method base.freeze()
  return new closure(self, env)
```

Figure 3.7: Additions to add_param for re-write evaluation. The entire contents of this figure should appear inside the generic add_param in figure 3.6.

As was the case with the functional approach, the first version of the parameters language fragment was unsuitable for use with an evaluator that works by rewriting the source because certain bits of information are moved from the program source into the evaluator's state. This can be fixed by adding a method to capture the evaluator's state and make it an explicit object. A new class called closure is introduced to represent the program fragment in this context in figure 3.7. The entire contents of this figure should appear inside the generic add_param. Further, the base class has a method called freeze defined for capturing the context. This requires that the original language fragment also defines a freeze method so that the method is not undefined in the original language. Note that freeze is defined once and inherited by all of the subclasses. The freeze method corresponds roughly to the extra ctx parameter in the functional formulation.

3.5 Algebraic approach

The previous approaches to writing combinable modules concentrated on using existing control flow mechanisms to bring together various parts of the language. Both approaches yielded some quite complex results and issues. In this section I take it for granted that the syntactic case analysis can be resolved by merely stating that the modules can act together. This relieves us of the burden of the complexities of the functional approach while giving us the benefits of the 'distributed dispatch' of the object oriented approach. Algebraic specification is an established formalism for combining modules in this way.

An algebraic module comprises some sorts (types), some operation symbols used to form terms over the sorts, and some rewrite rules that define which which combinations of terms are equivalent. The sorts and operation symbols form the *signature* of the algebra. The sorts and operation symbols defined for abstract syntax in section 3.1 form a signature. In previous sections I have used a production notation to describe the abstract syntax. Algebraic specifications usually define the operation symbols differently:

```
algebra combinations
  import
    integers
  sorts
    comb, op
  operations
    comb → Comb(op, comb, comb)
    comb → Num(integer)
    op → Add
    op → Mul
    ev : comb → integer
  equations
    ev(Comb(Add,c1,c2)) = ev(c1) + ev(c2)
    ev(Comb(Mul,c1,c2)) = ev(c1) × ev(c2)
    ev(Num(n))          = n
end combinations
```

Figure 3.8: The combinations language as an algebraic module

abstract syntax	comb → Add(comb, comb)
algebraic specification	Add(comb, comb) → comb *—or—* Add : comb × comb → comb

I will use both of these notations, and always distinguish sorts and variables from operation symbols by reserving names with initial upper-case letters for the operations. This ensures that an operation symbol that is a constant is not ambiguous. Having two equivalent notations is useful—I will use the abstract syntax style for definitions of operation symbols that are actually part of the abstract syntax and the algebraic style for operations that are not. Algebraic specification does not distinguish between functions and data but it is useful to maintain a distinction.

A signature defines the type of a module. This signature declares two sorts and an operation. It captures the essence of a 1-place function.

```
signature mapping
  sorts
    domain, range;
  operations
    value_at : domain → range
```

Many modules can conform to a signature. The combinations module in figure 3.8 defines a language like the combinations language introduced earlier. The language is slightly different—the operations (addition or multiplication) have been given a sort of their own. The combinations module conforms to the mapping signature under the substitution of comb for domain, integer for result and ev for value_at. It imports the module integers which provides the sort integer and its operations, like addition and multiplication. New modules and signatures can be defined by substitution. For example, the signature evaluator looks just like a mapping from expressions to results:

signature evaluator = mapping[expr/domain, result/range, eval/value_at]

Modules may be parameterized. The parameter is typed with a signature. The parameterized module may be instantiated with any module that conforms to the signature. A module corresponding to the parameters language fragment is sketched:

```
algebra add_params (lang : evaluator)
  import
    identifiers      —provides the sort 'name'
  operations
    expr → Let(name, expr, expr)
    expr → Param(name)
  equations
  ... more later...
end add_params
```

The meaning of the new constructs will be described as additional equations for the ev function. The Let-form

Let(*name*, e_1, e_2)

creates a new scope around e_2 in which *name* is visible and bound to e_1. Thus within e_2, name is an abbreviation for e_1. Wherever Param(*name*) appears in e_2 the associated value of *name* is substituted instead. The implementor of this fragment must make some decisions which determine what kind of 'let' the fragment implements. The choices are

- Evaluation time—should e_1 be evaluated when it is associated with the name or when it is substituted later?
- Visibility inside definition—should name be visible only in e_2, or also in the definition e_1, which would yield a recursive or **letrec** kind of definition.
- Visibility inside other Let forms—how does a nested Let with the same name, or a different name, affect the visibility of name?

The actual choice is arbitrary, the key point being that these decisions are confined to the language fragment. As we shall see, there are other issues which cannot be confined to the fragment and depend on its context.

A first approximation to the language semantics is the re-write rules

Let(n, v, Param(n))	→ v	
Param(n)	→ error	—*a name by itself is undefined*
Let(n, v_1, Let(n, v_2, body))	→ Let(n, v_2, body)	—*shadowing n*
Let(n_1, v_1, Let(n_2, v_2, body))	→ Let(n_2, Let(n_1, v_1, v_2), Let(n_1, v_1, body))	
		—*circular!*

This approach is inadequate on two accounts.

- It is difficult to specify the meaning of nested Lets with different names without writing a circular equation.

```
algebra add_params (lang : evaluator)
  import
    environments[name/domain, expr/range]      —provides sort 'env'
    identifiers                                —provides sort 'name'
  operations
    expr → Let(name, expr, expr)
    expr → Param(name)
    expr → Closure(expr, env)
  equations
    eval(Closure(Let(n,d,b),e)        → eval(Closure(b,extend(e,n,Closure(d,e))))
    eval(Closure(Param(n),e))         → eval(lookup(n,e))
    eval(Closure(Closure(b,e),e′))    → eval(Closure(b,e))

    eval(Param(n))   → error("Undefined parameter",n)
    eval(Let(n,d,b))  → eval(Closure(b,extend(Empty_env,n,d)))
end add_params
```

Figure 3.9: The parameters language as an algebraic module.

- It is not possible to say what happens when the body of a Let is an unknown language construct like Comb.

A solution to the first problem is to collect the effects of successive Lets in a *closure*. A closure is a representation of a piece of program with bindings of its free variables, and as such it has two parts: the piece of program and a mapping from the free variables to the things to which they are bound, called an *environment*. A closure can be thought of as a lazy substitution—instead of substituting for the names immediately, the substitutions are kept in a separate place until needed when the names are evaluated. A closure can be introduced as a piece of new abstract syntax which carries an environment and may be used anywhere a normal piece of syntax may be used. As it is not part of the language per se, the new syntax construct is called pseudo-syntax. Pseudo-syntax allows us to represent derived pieces of abstract syntax that are internal to the evaluation mechanism but visible to any transformations that may later be performed on the derived grammar.

The second problem is tackled by automatically generating rules for cases not specified in the fragment. Some rules *cannot* be specified for a parameterized fragment because the parameter syntax is unknown. All that the formal parameter tells the module add_param is that it has two sorts and a mapping between them called ev. In the combined language of parameterization and combinations we would like the rule

$$\mathsf{Let}(n,\ v,\ \mathsf{Comb}(op,\ e_1,\ e_2)) \ \rightarrow \mathsf{Comb}(op,\ \mathsf{Let}(n,\ v,\ e_1),\ \mathsf{Let}(n,\ v,\ e_2))$$

to be generated automatically. In this case the sub-expressions inherit the context of being in a Let. Informally, they do so because the sub-expressions might contain a Param form referring to the name.

The module for the parameters language is given in figure 3.9. The first equations determine how the new language features behave in the context of a closure. A Let adds a new binding to the environment. Note that the value of the bound variable is closed in

```
signature identifiers_sig
   sorts name
end

signature environments_sig
   sorts domain, range, env
   operations
      Empty_env → env
      lookup : domain × env → range
      extend : env × domain × range → env
end
```

Figure 3.10: Signatures of the **identifiers** and **environments** modules. The environments module provides an extensible mapping and the identifiers module provides names.

its defining environment. A Param form is reduced by extracting the bound value from the environment. A closure within a closure has the same value as the inner closure because the closure, by definition, is complete and contains no variables that are not defined in the inner closure's environment. The last two equations state that a Param outside a Let is an error, and that the outermost Let starts with an empty environment.

How should additional rules be automatically generated? Consider instantiating the parameters module with the combinations module. Inspection reveals that the patterns for the Closure operation symbol are incomplete. Extra equations are needed to determine what happens for the cases

```
Closure(Comb(op,e1,e2),env)  → ?
Closure(Num(n),env)          → ?
```

The two simplest options are to ignore (discard) the context or to raise an error. Neither option is adequate.

The only other assumption that might be made is that the substructures of a term should inherit the context. This is not appropriate for all of the sorts in the abstract syntax. The sorts that should inherit the context can be found by looking at the *containment relation* of the abstract syntax, which is a relation between the sorts of the abstract syntax. Two sorts σ_1 and σ_2 are related by the containment relation, written $\sigma_1 \rightarrow_c \sigma_2$, if the abstract syntax contains an operator op such that $\sigma_1 \rightarrow op(..., \sigma_2, ...)$. The transitive closure of the containment relation, written $\sigma_1 \rightarrow_c^* \sigma_2$, tells us which components of an operation symbol should inherit the context. When completing the equations for a context of sort σ_c a subterm of sort σ inherits the context only if $\sigma \rightarrow_c^* \sigma_c$. In the combinations language comb$\rightarrow_c^*$comb, comb$\rightarrow_c^*$op and comb$\rightarrow_c^*$integer (all trivially). thus only the comb sort should inherit the context. This gives the equations

```
eval(Closure(Comb(op,e1,e2),env))  → eval(Comb(op,Closure(e1,env),
                                                  Closure(e2,env)))
eval(Closure(Num(n),env))          → eval(Num(n))
```

3.5.1 Performance and partial evaluation

A tool was written that completed rules on a Lisp representation of the algebraic specifications. The performance of the code was poor because so much time was being spent reconstructing the intermediate terms. Consider the derived equation

```
eval(Closure(Num(n),env))  → eval(Num(n))
```

This constructs the term Num(n) only to pull it apart again in the case analysis implied by the pattern matching selection in eval. Every time the same equation will be selected:

```
eval(Num(n)) → n
```

This needless work can be avoided by transforming the program to a more efficient one by partial evaluation. The eval function can be *specialized* since we know something about its argument. A specialized version of a function is generated by symbolically evaluating the function with the argument, in this case Num(n), replacing all the calls to eval with the calls to the respective specialized versions. For the Num case this yields

```
eval(Closure(Num(n),env))  → eval_Num(n)
eval_num(n) → n
```

which may be *unfolded* to produce

```
eval(Closure(Num(n),env))  → n
```

A more interesting example shows another transformation technique in action: *variable splitting*. If a function is being specialized with respect to a value that is partially known and there are several unknown parts of the value then each unknown part can be passed as a separate value. Symbolic evaluation is done on the known part of the value. The automatically generated equation

```
eval(Closure(Comb(op,e1,e2),env))  →
      eval(Comb(op,Closure(e1,env),Closure(e2,env)))
```

generates several specialized equations. Often a specialized equation is called from several places but only has to be generated once. This saves on the time required to transform the program and on its size. The specialized equations for the Comb case are:

```
eval(Closure(Comb(op,c1,c2),env))  → eval_Comb(op, c1, env, c2)

eval_Comb(Add, c1, env, c2)  → eval_Closure(c1, env) + eval_Closure(c2, env)
eval_Comb(Mul, c1, env, c2)  → eval_Closure(c1, env) × eval_Closure(c2, env)

eval_Closure(comb, env)...
    all the cases for Closure(...) including:
eval_Closure(Comb(op,c1,c2), env) → eval_Comb(op, c1, env, c2)
```

Note how this program looks remarkably like an interpreter written to pass around an environment. These improvements were done by hand on the Lisp version of the function eval, strictly applying the methodology described by Mogensen [Mogensen 1989b]. Another possibility would be to generate the specialized functions at the same time as the pattern matching completions. The combinations$\oplus$parameters language generates eleven specialized functions, many of the simple like eval_Num. The speed increase is a factor of 5.

3.6 Discussion

This chapter has been a voyage through several different ways of writing programming language fragments so that those fragments can be combined to produce larger fragments and complete languages. The interpreters that have been developed are restricted to a two-phase compositional view of the program, i.e. the result of each program fragment is defined in terms of its parts and some contextual information. The contextual information is passed down the structure, either by means of dynamically chosen functions or by dynamically scoped variables, and the result is passed back up. It is not clear that this is a suitable scheme for all forms of program.

The efficiency of the methods discussed also leaves much to be desired. Although partial evaluation can be used to improve efficiency it is not a panacea. Partial evaluation is very difficult to apply to programs with dynamic control, which is the case in both the functional and object oriented approaches.

What the experimentation has succeeded in doing, however, is revealing some of the issues involved in working with language fragments. The most important observations that can be made are

- A fragment needs a way of controlling parts of the language about which it has no specific knowledge. Although I have shown ways of doing this by inheritance or the creation of default equations it is a major point and needs to be understood better.

- Introducing new semantic domains, like an environment or a store is a major source of difficulty, especially when the fragments were not written with this kind of extension in mind.

- The examples given in this chapter have had the flow of information roughly parallel to the flow of control. This allowed, for example, the use of a dynamic variable to store the environment in the object-oriented problem. If the flow of information is not parallel with the flow of control in the interpreter then none of the techniques used here can work.

It is with the advantage of these insights, most notably the first, that the modular grammars of the next few chapters were developed.

4 Modular Syntax

In this chapter I develop a modular grammar-based system for writing programs that define and manipulate other programs, in other words a modular metaprogramming system. The system is called ELDERII. In the rest of this chapter I will discuss context free grammars (CFGs, the grammatical basis for the system) and the choice of a CFG parser. Then I will introduce modularity and metaprogramming. Later I discuss the implementation issues.

4.1 Grammars

Context free grammars, and their concrete representation, BNF (Backus Naur Form), are an established formalism for describing the syntax of programming languages. A language is a set of strings over a given alphabet or set of symbols. It is usually infinite. CFGs describe the class of languages called context free languages. There are useful classes of CFGs for which it is possible to construct an efficient recognizer that determines if an input belongs to the language described by the grammar.

Grammars are a concise and declarative description of the syntax of a notation. They are also of use in the prototyping of programming languages and notations because they establish a framework upon which other parts of the prototype can be based.

4.1.1 Terminology

A context free grammar is a set of rules which describe a language. For example, a rule might say that we can parenthesize expressions:

> "if E is an expression then `(`E`)` is an expression."

A formal definition pins down all the components of this statement. A context free grammar G is a tuple (N, T, S, P) consisting of four components: a set of nonterminal symbols N, a set of terminal symbols T, a start symbol or *axiom* S and a set of rules or *productions* P.

- Terminal symbols (or terminals) are the basic constituents which make up the input. The input is a sequence or string of terminals. Typical terminals are keywords (like "`if`"), identifiers or names ("`i`", "`x`", "`My_Func`"), numbers ("`56`", "`-3.4`"), operator symbols ("`+`", "`>=`") and punctuation ("`,`", "`..`").

  Two classes of terminals are distinguished, depending on whether or not they carry any information in addition to that implied by their presence in the input.

  - *Constant terminals* carry no extra information. When specifying a grammar we usually write the string that comprises the terminal in quotes.
  - *Variable terminals* are terminals that have variant forms; there might be an infinite number of variants. A *number* might be a terminal. Both of

the inputs "5" and "42" are *numbers* but they denote different integers. A variable terminal has a *value* which can be constructed from the textual form of the terminal. For example, the value of the identifier terminal "`My_Func`" might be a string of the characters that make up the name (`"My_Func"`) or something derived from that string (e.g. the Lisp symbol `MY-FUNC`). When specifying a grammar we write the name of the terminal which stands for any of the variants.

Terminals are also called *tokens*.

- Nonterminals denote sets of strings. A nonterminal usually 'stands for' a class of language constructs like an *expression* or a *definition*. The vocabulary, V, is the set of all symbols, both terminals and nonterminals. $V = N \cup T$. In talking about grammars we use the greek letters $\alpha, \beta, \ldots$ to denote arbitrary strings in V^*. The notation V^* means the set of strings constructed from symbols in V, including the empty string, ϵ. V^+ means the same but excluding ϵ.

- One nonterminal is distinguished as the start symbol or axiom. This nonterminal is just like any other nonterminal except that the set of strings denoted by this symbol is the language defined by the grammar.

- The rules define how terminals and nonterminals may be combined to form string. A rule is a nonterminal followed by an arrow ($\rightarrow$) followed by a sequence of 0 or more terminals or nonterminals. The rule means that the set of strings denoted by the left side nonterminal includes the set of strings denoted by the right side sequence.

Now I introduce some useful standard terminology for discussing how a CFG describes a language. The terms introduced are: derivation, sentence and sentential form. If $\alpha \mathrm{B} \beta$ is a string in V^+, where $\alpha, \beta \in V^*$ and $\mathrm{B} \in N$, and there is a production $\mathrm{B} \rightarrow \gamma$ in P, then $\alpha\gamma\beta$ is said to be *directly derivable* from $\alpha \mathrm{B} \beta$ because B can be replaced by γ. This fact may be written

$$\alpha \mathrm{B} \beta \ \Rightarrow \ \alpha\gamma\beta$$

We say that a string δ is *derivable* from $\alpha \mathrm{B} \beta$ if either $\delta = \alpha \mathrm{B} \beta$, or there is a sequence of strings $\delta_1, \ldots, \delta_k$ for some $k \geq 0$ such that:

$$\alpha \mathrm{B} \beta \Rightarrow \delta_1 \Rightarrow \cdots \Rightarrow \delta_k \Rightarrow \delta$$

If δ is derivable from $\alpha \mathrm{B} \beta$ we write this as:

$$\alpha \mathrm{B} \beta \Rightarrow^* \delta$$

The subset of V^* that we can derive from the start symbol S is called the set of *sentential forms*. Often restrictions are placed on derivation, for example in LL parsing only the leftmost nonterminal may be expanded which restricts the set of sentential forms. If a sentential form contains only terminal symbols then it is a *sentence* in the language. Sentential forms are half-built sentences. The sequence $\alpha \mathrm{B} \beta, \delta_1, \ldots, \delta_k, \delta$ is called a *derivation* of δ from $\alpha \mathrm{B} \beta$.

A grammar is *ambiguous* if there is a sentence in its language which has more than one derivation.

4.1.2 Classes of grammars

The *recognition problem* is the problem of taking a grammar and a string and answering the question 'is this string in the language described by the grammar?' Rather than a 'yes or no' answer we usually want to know in detail how the string satisfies the grammar. This is the *parsing problem*. In the general case parsing for an arbitrary CFG is quite difficult. The performance of a parsing algorithm is usually measured as its time complexity as a function of the length of the input, n. All of the algorithms for solving the general problem have poor worst-case performance. There are various classes of CFG for which algorithms have been invented that have better characteristics. There are two main categories of parsing algorithm—top-down and bottom-up.

4.1.3 Top-down parsing

Top-down parsing algorithms work from the axiom of the grammar. The derivation tree starts out as the axiom of the grammar and grows down during parsing. A nonterminal (usually the leftmost in the derivation tree) is parsed by deciding which production of the nonterminal matches the input. The derivation tree is extended by the r.h.s. of that production. It is in how this decision is made that the top-down algorithms differ.

A full backtracking search tries each production in turn, and if a decision turns out to be incorrect (the input cannot be parsed) then the effect of the choice is undone and the next choice is tried. In the worst case this algorithm is exponential, and it takes this exponential time if the input is in error. This algorithm is clearly unsuited to a realistic application.

LL(k) parsing works by deciding on one of the productions and never reversing this decision, so it is linear in time. The choice is made by inspecting the next k input tokens. The common case is LL(1), where a single token is used as the look-ahead. For LL parsing to be used the grammar must have the property that the r.h.s. of no two of the productions for a single nonterminal can possibly expand to strings which start with the same token, otherwise the parser cannot pick a single production with certainty.

An intermediary between these two is the class of top-down parsing languages (TDPLs). These parse by a backtracking search, but the backtracking is limited to within the current nonterminal's set of productions. A nonterminal is never re-parsed to see if the input can match the same production at a higher level in a different way. This greatly cuts down parsing time.

4.1.4 Bottom-up parsing

Bottom up parsing works from the input. If part of the input matches the r.h.s. of a production then that part is replaced with the l.h.s. of the production. This process is repeated until the input is 'reduced' to the axiom of the grammar. The bottom up algorithms differ in how they make the decision that a string in the (partially reduced) input matches the r.h.s. of a production and whether or not to replace it with the l.h.s.

4.1.5 Choosing a parsing method

Most programming languages can be conveniently specified by an LALR(1) grammar. Two different grammars may describe the same language , i.e. derive the same set of strings. Sometimes one grammar may be amenable to a particular parsing algorithm but the other not. For example, the LR languages do not form a hierarchy—they all describe the same set of languages. The LALR(i) languages are the same as the LALR(j) languages for all i and j. A particular grammar might not be LALR(1) but if the language is then it can be parsed by another LALR(1) grammar.

This can be illustrated by a simple grammar for program options. The idea is that program may be given a set of options, perhaps from the command line or a configuration file. Each option is of the form "*name* = *value*" where *name* is a word and *value* is a possibly empty sequence of words and numbers. Options are listed one after the other. A program to print out files that uses command line options might be invoked with parameters like this:

```
print file = part_a lisp  printer = lp  heading = Theorem Prover
```

We wish to describe the syntax of the options using a grammar. A suitable grammar is:

$$\begin{aligned}
&\mathsf{options} \rightarrow \mathsf{option\ options}\\
&\mathsf{options} \rightarrow \mathsf{option}\\
&\mathsf{option} \rightarrow \mathsf{word}\ \text{“=”}\ \mathsf{words}\\
&\mathsf{words} \rightarrow\\
&\mathsf{words} \rightarrow \mathsf{word}\\
&\mathsf{words} \rightarrow \mathsf{number}
\end{aligned}$$

The symbols *word* and *number* are value terminals which are provided by the basic input routine. This grammar is not LALR(1). A parser based on this grammar looking ahead at a *word* token cannot tell if the word is the start of a new option or part of the current list of words. The parser needs to look at the next two input symbols to see if they are *word* and "=" respectively. The language belongs to a simpler class of languages, the regular languages, as all the recursive structure is of an iterative nature. Rewriting the grammar as a regular grammar[1] (and hence also an LALR grammar) ruins the meaningful structure of the grammar:

$$\begin{aligned}
&\mathsf{options} \rightarrow\\
&\mathsf{options} \rightarrow \mathsf{word}\ \text{“=”}\ \mathsf{options}'\\
&\mathsf{options}' \rightarrow\\
&\mathsf{options}' \rightarrow \mathsf{number\ options}'\\
&\mathsf{options}' \rightarrow \mathsf{word\ options}''\\
&\mathsf{options}'' \rightarrow\\
&\mathsf{options}'' \rightarrow \mathsf{number\ options}'\\
&\mathsf{options}'' \rightarrow \text{“=”}\ \mathsf{options}'\\
&\mathsf{options}'' \rightarrow \mathsf{word\ options}''
\end{aligned}$$

It would be difficult to reason about the source program using this grammar because the structure of the rules has been destroyed. It is not clear which words in the above grammar denote the names of the options and which ones are part of their values: word in the last production could be either. In this case it is possible to rewrite the grammar as an LALR(1) grammar in a way that maintains this information, but the result is not entirely satisfactory:

[1] The rewritten grammar is regular because all productions have at most one nonterminal on the r.h.s. and it is always the last (rightmost) r.h.s. symbol. All regular languages can be expressed in this form and all grammars in this form describe regular languages [Aho *et al.* 1986].

options $\rightarrow$
options $\rightarrow$ word "=" words
words $\rightarrow$
words $\rightarrow$ word "=" words
words $\rightarrow$ word words
words $\rightarrow$ number words

The problems with this grammar are that option name appears in two places and the start of subsequent options is buried in the productions for *words*[2]. This example shows that a sufficiently powerful parser construction technique should be chosen so that artificial and difficult rewriting of grammars is not required.

For ElderII I chose a TDPL over a shift-reduce parser for two reasons: it is easier to understand how combined languages will operate; and TDPLs allow an uncomplicated and modular implementation. Reasoning about combined languages is easier because the reasoning is in terms of production ordering and what productions may shadow other productions. Reasoning about LR parsers involves understanding shift-reduce and reduce-reduce conflicts and the LR(0) automaton which is a global property of the grammar.

A TDPL allows a modular decomposition in the implementation because it is implemented as a set of largely independent parsing procedures, one for each nonterminal. Thus a fundamental unit of the language maps onto a fundamental unit of the implementation. This eases implementation of a modular parser which would otherwise be burdened with a complex mapping between the grammar and its parser.

4.1.6 Limitations of backtracking top-down parsers

A backtracking parser by its nature may re-examine parts of the input. It is easy to construct grammars which demonstrate this:

stmt $\rightarrow$ lvalue ":=" expr	—*assignment*
stmt $\rightarrow$ ident "(" exprlist ")"	—*procedure call*
⋮	
lvalue $\rightarrow$ ident "(" indexlist ")"	—*array reference*
index $\rightarrow$ ident	
expr $\rightarrow$ ident	

Consider the input

```
foo(x,y,z);
```

The parser first parses "`foo(x,y,z)`" as an lvalue. Because this is not followed by "`:=`" it backtracks and re-parses the input as a procedure call. Sometimes even limited backtracking can lead to exponential behaviour. If, say, expr $\Rightarrow^* \alpha$ stmt β then this

[2]The 'best' LALR(1) rewrite is this but it does not add to the discussion:

options $\rightarrow$
options $\rightarrow$ word "=" words
words $\rightarrow$ options
words $\rightarrow$ word words
words $\rightarrow$ number words

might happen. On an input like "`foo(bar(x));`" on each reading of the top level the inner form "`bar(x)`" is scanned twice. Backtracking costs more if it occurs at a higher level in the grammar. Fortunately most languages are lists of things at the higher levels so it is unlikely that the pathological case will happen. Importantly, many languages are not LL or LR but are *nearly* LL.[3]

This happens when many constructs are introduced by a keyword. TDPLs can parse many not-quite LL or LR languages. The above grammar is not LR(k) for any k. There is a reduce-reduce conflict between the reductions $\mathsf{expr} \rightarrow \mathsf{ident}$ and $\mathsf{index} \rightarrow \mathsf{ident}$.

The shared part of the two **stmt** productions can be revealed by substituting the productions **lvalue** into the first **stmt** production. This transformation has the disadvantage of any source level grammar transformations in that it alters the shape of the parse tree.

TDPLs also compromise the declarative nature of grammars. The ordering of productions is significant. Consider the two productions

$$\begin{array}{ll} \mathsf{A} \rightarrow \mathsf{B} & \mathsf{B} \Rightarrow^* \delta \\ \mathsf{A} \rightarrow \mathsf{C} & \mathsf{C} \Rightarrow^* \delta\beta \end{array}$$

The production $\mathsf{A} \rightarrow \mathsf{C}$ will never be chosen on input $\delta\beta$ because the parser will accept δ for the production $\mathsf{A} \rightarrow \mathsf{B}$. To rectify this problem the productions must be transposed.

TDPLs share with LL parsing methods the weakness that they cannot parse grammars with left-recursive rules like

$$\mathsf{expr} \rightarrow \mathsf{expr}\ \text{“+”}\ \mathsf{term}$$

There is a simple extension to TDPLs that allows direct left recursion to be recognized and treated specially. This makes TDPLs more useful as it allows the programmer to specify both left and right recursive structures.

4.2 Modular Grammars

There are two main types of metaprogramming application where grammar fragments can be used

- extension of an existing definition with new features
- creation of a new language built from components of fragments

In this section I describe modular grammars and how they may be used in both applications.

4.2.1 Basic grammar notation

Grammars are described in ElderII using a BNF-like language called "*grammar*". An example of its use is the grammar for arithmetic expressions shown in figure 4.1. The choice operator (e.g. `mulop -> "*" | "/"`) is entirely equivalent to the two productions in the same order as the choices (as for `addop`).

Like most systems built around language notations, we use the grammar notation to describe itself. This description, called the meta-grammar, is shown as an additional example of the notation in figure 4.2.

[3] By 'nearly LL' I mean that there are a small number of productions that prevent a grammar for the language from being LL. If these productions are removed from the grammar or new symbols added to these productions then the language would be LL.

```
grammar expressions .

terminal ident number .

expr -> expr addop term | term .
term -> term mulop factor | factor .
factor -> "(" expr ")" | number | ident .
addop -> "+" .
addop -> "-" .
mulop -> "*" | "/" .
```

Figure 4.1: A grammar for expressions

A grammar consists of a header, which names the grammar, followed by some declarations, followed by a list of productions. The grammar notation for productions is similar to the BNF notation that we have been using to discuss grammars. The 'keyboard form' of the production arrow ($\rightarrow$) is "`->`", and the right side is terminated with a period[4]. Productions may also be given as alternates separated by vertical bars.

The "`terminal`" directive specifies which names denote value terminals. "`axiom`" specifies which nonterminal is the start symbol, which defaults to the l.h.s. symbol of the first production. In the expressions grammar it is not specified so it defaults to "`expr`". Comments are handled by a separate lexical analyser to the main text. The "`comments`" directive specifies which comment style is used. Other directives are described later. Both the `terminal` and `comments` directives are tie-ins with lexical analysis. The named value terminals must be provided by the lexical analyser.

4.2.2 Grammar Modules

Each grammar definition is a module. One module can be used in another by importation. At the simplest level modules can be used to divide up a grammar into several parts. One grammar then "`import`"s all of the other parts. An example of this use is in extending a language with a few additional bits of syntax. The original language forms one module. A new module can import that 'base' module and define the extensions to the syntax. While modules are a useful structuring tool, parameterized modules are more powerful.

4.2.3 Parametric Grammars

This section describes parametric grammars and how they are used. Parameterized grammars have a list of symbols following their name. These are the formal parameters. The grammar `hierlevel` (figure 4.3) has three parameters, `level`, `nextlevel` and `operator`. This grammar represents a single level in a precedence hierarchy of expressions constructed from left-associative binary operators. Actual symbols are provided by the grammar which imports the parametric grammar. They are substituted into the

[4]This syntax was chosen for reasons similar to those discussed in section 4.1.2—without the period the grammar cannot be parsed with a single look-ahead without nonlocal backtracking.

```
grammar grammar .

terminal ident string .
comments shellCommentLexer .

grammar -> header directives rules .

header -> ident ident parameters "." .       #ident1 = ``grammar''

parameters -> "(" idents ")" .
parameters -> .

directives -> directive directives .
directives -> .

directive -> "import" ident idents "." .     #use another grammar
directive -> "export" idents "." .           #export names for use
directive -> "terminal" idents "." .         #value terminals
directive -> "external" ident string "." .   #special nonterminal
directive -> "axiom" ident "." .             #grammar axiom
directive -> "comments" ident "." .          #comment lexer
directive -> "lexer" ident "." .             #name non-default lexer

idents -> ident idents .
idents -> .

rules -> rule rules .
rules -> rule .

rule -> ident "->" choice "." .

choice -> sequence "|" choice .
choice -> sequence .

sequence -> collection sequence .
sequence -> .

collection -> ident .                #nonterminal and value terminal
collection -> string .               #constant terminal
```

Figure 4.2: The meta-grammar. `shellCommentLexer` is a lexical analyser that removes comments starting with a hash symbol and continuing to the end of the line, like the comments above. The grammar requires that the lexical analyser provides two types of value terminal: `ident` and `string`. In effect the grammar imports these from the lexical analyser.

```
grammar hierlevel (level nextlevel operator) .

level -> level operator nextlevel .
level -> nextlevel .
```

Figure 4.3: Grammar module for a precedence level

```
grammar hierdemo .

import hierlevel expr term addop .
import hierlevel term factor mulop .
import hierlevel factor primary applyop .

terminal ident number .

primary -> ident .
primary -> number .
primary -> "(" expr ")" .

addop -> "+" .   addop -> "-" .

mulop -> "*" .   mulop -> "/" .

applyop -> "." .
applyop -> "->" .
```

Figure 4.4: Using the precedence level module.

---

body of the parametric grammar and the resulting rules are merged with the rules of the importing module.

Parameterized grammars can be imported several times, each *instance* being parameterized with different symbols. The grammar `hierdemo` (figure 4.4) imports `hierlevel` three times. The pattern of sharing of symbols between the importations yields a three level precedence hierarchy, with *addop*s having the lowest precedence and and *applyop*s the highest:

$$\begin{aligned}&\mathsf{expr} \longrightarrow \mathsf{expr\ addop\ term}\\&\mathsf{expr} \longrightarrow \mathsf{term}\\&\mathsf{term} \longrightarrow \mathsf{term\ mulop\ factor}\\&\mathsf{term} \longrightarrow \mathsf{factor}\\&\mathsf{factor} \longrightarrow \mathsf{factor\ applyop\ primary}\\&\mathsf{factor} \longrightarrow \mathsf{primary}\end{aligned}$$

Arbitrarily complicated parameterized grammars can be specified and used. Two main 'flavours' of use are apparent:

- Small scale constructs that abstract common patterns, like the precedence hierarchy above and lists.

- Large scale constructs—grammars for entire language fragments like 'expressions', 'statements', 'parameterization'.

4.3 Metaprogramming

We have seen how ELDERII supports the definition of modular grammars. Now we see how to use these grammars as a basis for writing programs that manipulate other programs, i.e. metaprograms. The key to metaprogramming is to have a sufficiently rich set of operations that apply to the program data type.

4.3.1 The power of the system

Cameron and Ito identify six kinds of operation that should be supported by a metaprogramming system in [Cameron & Ito 1984].

1. Type recognition. "Is X an if-statement?", including what Cameron calls 'syntagmatic types', which is membership of a higher level construct, e.g. a number is recognised as an expression.
2. Component selection. "Pick the predicate of the conditional."
3. Construction. "Make an assignment-statement with variable X and expression Y."
4. Context determination. "Find the nearest enclosing subprogram definition of node X."
5. Editing. "Delete the third statement of the block." Delete, insert, replace and splice.
6. Lexeme coercion. "Return the value represented by the numeric lexeme Z."

Cameron identified these operations with an imperative metaprogramming language in mind. In a functional paradigm many of the editing operations would be rewritten as combinations of selection and construction.

4.3.2 Partial parses, patterns and constructors

The essential operations of a metaprogramming system are differentiation between different syntactic objects and the construction of new syntactic objects. The ELDERII approach to both aspects is by using textual program fragments. Program fragments are little pieces of program, possibly containing *escapes*. For example,

```
while <expression> do <statement>
```

is a Pascal statement with two escapes. Linguistically, a program fragment is a sentential form in the language of some nonterminal in the grammar. Escapes are unexpanded nonterminals in that sentential form. An escape may be annotated with a name. This is useful when there are several escapes in a fragment which share the same nonterminal. A list of two expressions might be written

```
<expression e1>, <expression e2>
```

A program fragment has a different meaning in different contexts. It can be used as a pattern, i.e. a predicate, or it can be used as a constructor. As a pattern or predicate, a program fragment determines whether or not a syntactic object can be derived from the sentential form that the pattern represents. This is equivalent to matching the parse tree of the syntactic object against a data structure pattern for the partial parse tree of the sentential from. This is true because a parse tree is a direct record of the derivation. Escapes in the program fragment are variables in the pattern. In the appropriate scope for the pattern, the variable is bound to the subtree of the syntactic object which is the derivation of that nonterminal in the sentential form. The pattern variable is named after the nonterminal, unless the nonterminal is annotated, in which case it is named after the annotation. If a pattern variable is repeated then all subtrees denoted by that variable must be equal, i.e. they must by syntactically identical[5]. For example,

```
<expression>, <expression>
```

will only match a list of two identical expressions, binding the expression to the variable `expression`, whereas

```
<expression e1>, <expression e2>
```

will match a list of any two expression, binding `e1` to the first and `e2` to the second.

When used as a constructor, a program fragment specifies a complete parse tree. An escape in the fragment identifies a subtree, the structure of the subtree rooted at the escape is determined by the value of a variable in the current scope. Again, the variable is named after the nonterminal unless the escape is annotated in the fragment in which case the variable has the name of the annotation.

The constructor

```
<expression>, <expression>
```

builds a parse tree for a list of two identical (in fact shared) expressions determined by the value of the variable `expression`.

```
<expression e1>, <expression e2>
```

builds a parse tree for a list of two expressions determined by the value of variables `e1` and `e2`, which may or may not be equal.

4.3.3 Metaprogramming in Lisp

Metaprogramming operations are provided in Lisp for two reasons:

- these operations were necessary for bootstrapping the system.
- programmers might wish to write their application in Lisp.

The two basic operations are `match-syntax` and `build-syntax`. Both are operations on parse trees and they are implemented as macros which expand into the Lisp code to perform the matching or construction. `Match-syntax` is a case-like form which compares an argument with several parse tree specifications. The syntax of `match-syntax` is

[5]more correctly: they must have identical derivations

```
(match-syntax  expr  grammar  entry
  (parse-tree-1    action1  action2 ...)
  (parse-tree-2    action1 ...)
  ⋮
  (parse-tree-n    action1 ...))
```

Each *parse-tree-i* is a string constant containing a partial parse or sentential form in the language of the nonterminal *entry* in the grammar *grammar*. *Expr* is evaluated. The resulting tree is matched against each *parse-tree-i* in turn. The option with the first matching tree is selected. Subtrees of *expr* which correspond to escapes in the pattern are bound to variables with the appropriate names. Then *action*1, *action*2, etc. are evaluated in order, the value of the last one is the value of the whole `match-syntax` construct. *Parse-tree-n* may be the constant `T` which always matches the *expr* and serves as a default case. Note that *expr* is the only form which is always evaluated, and *grammar*, *entry* and the patterns are never evaluated. The patterns are expanded to Lisp matching code at macro-expansion time. The final Lisp program contains only a sequence of instructions that inspect the value of *expr*. The strings for the parse-tree-*i* and the parse tree structures that they represent are compiled into this more efficient form.

An arbitrary tree can be tested for membership in a nonterminal's language by using a pattern that is an escaped *entry*. This Lisp expression tests if *expr* is a doobrie:

```
(match-syntax expr grammar doobrie
  ("<doobrie>"  T)
  (T            NIL))
```

`Build-syntax` has a simple syntax. The form

```
(build-syntax grammar entry text)
```

returns a parse tree in the language of the nonterminal *entry* of the grammar *grammar*. The parse tree is specified by *text* which is a string constant containing a partial parse. Any escape in the parse tree takes the value of the variable of the appropriate name.

A third metaprogramming macro is `macro-scheme`. Its purpose is to concisely specify a set of syntax translations.

```
(macro-scheme  expr  grammar
  (entry
    (tree    actions...)
    (tree    actions...)
    ⋮
    (tree    actions...))
  ⋮
  (entry
    (tree    actions...)
    (tree    actions...)
    ⋮
    (tree    actions...)))
```

Expr is evaluated. `macro-scheme` returns a tree of the same syntactic type as *expr*. An *entry* is chosen depending on the syntactic type of *expr*. If there is no such entry

lingua
level2
base

Figure 4.5: Hierarchy of languages

then *expr* is returned unaltered. Each of the *tree* patterns in the chosen entry are matched against *expr* in turn. The first one that matches leads to the evaluation of the corresponding *actions*, just like with `match-syntax`. If no *tree* matches then *expr* is returned unaltered. The actions may include the special action

```
(replacement text)
```

which is equivalent to

```
(build-syntax grammar entry text)
```

where *grammar* and *entry* are the immediately enclosing *grammar* and *entry* items in the macro-scheme. This ensures that the returned value is of the correct syntactic type. The following macro scheme replaces expressions of the form "$x + x$" by "$2 \times x$". The two subexpressions have to be syntactically identical and this is assured be using the same variable to name both subexpressions:

```
(macro-scheme an-expression expressions
  (expr
    ("<term t1> + <term t1>"  (replacement "2*(<term t1>)"))))
```

4.3.4 Metaprogramming in lingua

Lingua is a the third language in a hierarchy of languages that was written to experiment with defining new dialects of a language. The hierarchy is shown in figure 4.5. Each language in the hierarchy is a superset of the previous language. The *base* language provides function definition and expressions with arithmetic operators and function calls. The *level2* language adds list constructors and pattern matching to this, including pattern matching at the function definition level. Finally, *lingua* adds syntax constructors and patterns.

4.3.4.1 base

Base provides a simple applicative functional programming language. It allows function definition, conventional 'arithmetic' infix expressions and comparisons, function calls, an if-then-else conditional and assignment to global variables.

4.3.4.2 level2

Level2 extends base with a constructor syntax for lists and pattern matching. There are two forms of pattern matching: a pattern matched function definition and an explicit pattern match. The former has the following syntax:

```
foo pattern ... -> expr
  | pattern ... -> expr
  ⋮
end.
```

This defines a function `foo`. The function chooses the first set of patterns that match and returns the value of the corresponding *expr*. If no patterns match then an error is signalled. Variables introduced in the pattern are bound to the corresponding parts of the arguments for the evaluation of *expr*. Each case must have the same number of patterns as the function has a fixed number of arguments.

The explicit pattern match has a similar syntax:

```
match expr with
    pattern -> expr
  | pattern -> expr
  ⋮
end
```

The `match` form is an expression which may appear in the place of any other expression, unlike the function definition which must appear in the place of a definition. The first expression is evaluated once and matched against each pattern in turn, like the pattern matched function. Level2 has no *let* construct. Instead of the basic let expression `let` *var* `=` *expr* `in` *body*, the following match 'metaphor' can be used, as the variable pattern will match anything and always bind to the expression.

```
match expr with var -> body end
```

List patterns and constructors use the following notation. Lists items are written between square brackets. A double colon is the infix 'cons' function.

`[]`	the empty list
`[1,2,3]`	a list of three integers
`head::tail`	the infix list constructor
`1::2::3::[]`	a list of three integers

4.3.4.3 lingua

Lingua adds syntax constructors and patterns to level2. Syntax constructors and patterns have the following syntax. There are three forms.

```
grammar.entry:[[ text ]]
entry:[[ text ]]
[[ text ]]
```

The double square brackets are syntax brackets. This notation is used because it is similar to the syntax brackets used in the programming language semantics literature.[6] *Grammar* is an identifier naming the grammar which defines the syntax of *text*. *Entry* is another identifier which names the nonterminal to which *text* belongs. If *grammar* is omitted then the default grammar is used. Note that the identifiers *grammar* and *entry* are not variables but directly name the grammar. If they were variables then it would not be possible to compile the program fragment into a pattern or constructor expression! Thus the paper syntax

$[\![$ **while** *expr* **do** *statement* $]\!]$

has the keyboard syntax

```
pascal.statement:[[ while <expr> do <statement> ]]
```

[6]CAML has a similar system, but it uses the symbols "<<" and ">>" because they look like French quotation symbols

4.4 Comparison with Cameron's criteria

Type recognition. The system supports type recognition by pattern matching. The following kinds of type recognition are supported:

- Subclassing. The pattern "expr:[[<term>]]" tests if the subject tree is an expression which is a term.
- General class membership. The pattern "expr:[[<expr>]]" tests if the subject is any tree of the type expr.

It is not possible to do tests which require induction over the grammar. For example, a tree that is a number will fail either of the above tests. The syntactic sorts are disjoint and unit productions do not imply subsorting.

Component selection. Components are selected by pattern matching. For example, the predicate of an if-statement is selected like this:

```
pred expr:[[if <expr p> then <expr e1> else <expr e2>]] -> p
```

Construction. Construction is supported through program fragment expressions.

Context determination. Context determination (for example, find the enclosing block) is done by tracking the required information as the program tree is traversed. This must be done because any tree can be shared between several programs or program fragments have an arbitrary number of parents. Which structure is *the* parent of stmts in the result from this transformation of a Pascal repeat statement to a while statement?

```
stmt:[[ repeat <stmts> until <expr> ]] ->
  stmt:[[ begin <stmts>;
                 while not <expr> do begin <stmts> end;
          end ]]
```

Editing. A program fragment is 'edited' by constructing a new program fragment with the required changes. This is more declarative than Cameron's editing operations as it does not cause side-effects. When incorporated in a pass over the entire program it is quite efficient.

Lexeme coercion. The value of a lexeme can be obtained by matching the lexeme against its pattern.

4.5 Discussion on metaprogramming facilities

ElderII provides facilities for defining grammars using both plain and parameterized grammar modules. This allows a user to quickly construct a grammar by using existing subgrammars. The system was used for constructing some programming language prototypes (for an example, see chapter 7) and the findings are discussed here. The main problem with ElderII is that, while grammars can be constructed in a modular style, there is no support for writing programs that use the modular grammars.

- It is not possible to write a program that applies to one module and will still work correctly in a language that uses that module with extensions.

- There is no facility to write programs that are based on abstractions provided by the parameterized modules. For example, you cannot write a single function that operates on one level of a precedence hierarchy and use it for all levels; or a function that works for all lists provided by instantiating a module.

The chapter on modular attribute grammars shows how to make up for these shortcomings.

4.6 Lexical analysis

Lexical analysis is the conversion of a stream of characters into a stream of terminals which are a higher level representation of the input. Performing lexical analysis as a separate phase to parsing has several benefits:

1. Ambiguities can be resolved at this level, for example between identifiers and keywords.
2. The size of the input can be reduced as tokens are often built from several characters. Layout and comments are completely removed from the input. This reduction in size reduces the time and space requirements of subsequent processing.
3. The grammar is simplified because layout and comments are removed from the input and no longer appear in the grammar and variant forms of the same symbol may be reduced to a single form.

4.6.1 Algebra of lexical analysers

Lexical analysers are used frequently in a metaprogramming environment, so it makes sense to establish what operations are allowed on them. To this end I describe a lexical analyser as a type and define two useful operations on that type.

A lexical analyser (lexer) is a function that performs lexical analysis on a stream of characters and returns a stream of tokens. I chose a single-step formulation where the tokens are yielded one at a time. The type of a lexer under this formulation is

$$lexer : [char] \rightarrow token \times [char]$$

A lexer takes a list of characters and returns a pair

- the token indicated by a non-null prefix of the list
- the rest of the list after removing the prefix that constitutes the token.

There are two special tokens: FAIL and EOF. The token FAIL is returned if the lexer cannot return a valid token. EOF is returned at the end of the input stream. Every lexer should return EOF when presented with an empty input list.

4.6.1.1 sequencing

A sequence of two lexers behaves like the first lexer, except when the first lexer would return a FAIL token. Then it behaves like the second lexer.

$$\mathsf{L} = \mathsf{L}_1 \ ; \ \mathsf{L}_2$$

$$
\begin{array}{l}
(\mathsf{L}_1;\ \mathsf{L}_2)\ \mathsf{stream} = \\
\quad \textbf{let}\ (\mathsf{token},\mathsf{stream}') = \mathsf{L}_1(\mathsf{stream})\ \textbf{in} \\
\quad\quad \textbf{if}\ \mathsf{token} = \mathsf{FAIL}\ \textbf{then}\ \mathsf{L}_2(\mathsf{stream}) \\
\quad\quad\quad\quad\quad\quad\quad\quad \textbf{else}\ \ (\mathsf{token},\mathsf{stream}')
\end{array}
$$

For example, the following lexer would test for identifiers, numbers and then strings, in that order:

$$\mathsf{lexer} = \mathsf{idents}\ ;\ \mathsf{numbers}\ ;\ \mathsf{strings}$$

4.6.1.2 hiding

A lexer can be used to hide tokens from another lexer. If the first lexer recognises a token then that part of the input is discarded and the first lexer is called again. This provides a comment facility which discards part of the input.

$$
\begin{array}{l}
\mathsf{hiding}_{\mathsf{L}_1,\mathsf{L}_2}(\mathsf{stream}) = \\
\quad \textbf{let}\ (\mathsf{token},\mathsf{stream}') = \mathsf{L}_1(\mathsf{stream})\ \textbf{in} \\
\quad\quad \textbf{if}\ \mathsf{token} \in \{\mathsf{FAIL},\mathsf{EOF}\}\ \textbf{then}\ \mathsf{L}_2(\mathsf{stream}) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \textbf{else}\ \ \mathsf{hiding}_{\mathsf{L}_1,\mathsf{L}_2}(\mathsf{stream}')
\end{array}
$$

4.6.1.3 Implementation

The input stream is modeled as a lazy list of characters. Laziness allows the input to be interactive. The lazy list is built using the technique where a lazy value is represented as a 0-place function closure called a suspension [Henderson 1980]. The suspension initially contains a function that computes the value (a *thunk*) and a flag which is set to indicate that the function must be called. The first time the suspension is activated it inspects the flag and calls the thunk and then stores the value. The flag is set to indicate that the function has been called. On subsequent calls the suspension knows from the flag the the function has been called and simply returns the stored value. Instead of using a flag the type of the list tail is used. (This is fine in Lisp because Lisp is dynamically typed.) If the type of the tail is a function then it is a thunk which is to be called to yield the tail of the list. If it is not a function then it is either the end of the list (`NIL`) or a cons cell with the next character in the car position. The input list is stored in a variable of dynamic scope, and the lexer function is written as a side-effecting zero-argument function returning a token. This interface facilitates the production of hand-written lexers. Many descriptions of how to build lexical analysers assume an imperative style of programming in which the input is read in a character at a time and the token is returned without reference to the input stream or its change of state.

The combining forms described above are implemented as higher order functions. Since the input stream is made global and is not single threaded[7] the combining forms must remember the state of the input stream and restore it on failure. As there are few combining forms and many lexers this is a convenient compromise.

The approach taken in the above has been pragmatic; the aim being to generate a simple framework for combining lexical analysers in meaningful and useful ways, rather

[7] A variable (function parameter) is single threaded in a program if its old value is never used after a new value has been assigned to it (by another function call). A single threaded parameter in a functional program may be replaced by a global variable without altering the meaning of the program. Stream is not single threaded in the lexer combining forms; stream might be used after calling the first parameter lexer L_1.

than generating a complete theory. There is scope for extending this 'algebra' of lexers to include all the standard combining operations of regular expressions. Higher order functions would be hopelessly inefficient if used at a too fine grained level, for example, combining one lexer for each token. For efficiency the lexers and their combining forms would have to be reduced to a form which did not re-scan the input. If conventional lexer-generation techniques such as those used by *lex* [Lesk 1975] are used then the incremental approach of [Heering *et al.* 1987] could make lexer generation more efficient.

4.6.2 Standard lexer

The system provides a standard lexer that is adequate for many experimental languages. The standard lexer recognizes the following classes of terminal:

> *Numbers*, including signed and unsigned integer, decimal and 'scientific' exponent representations. The terminal name `number` is used in grammars to refer to numbers.
>
> *Strings*, which are restricted to C-style strings, delimited by double quotation characters ("). The terminal name `string` is used to refer to string tokens.
>
> *Identifiers*, consisting of a letter followed by alphanumeric and underscore characters. The terminal name `ident` is used to refer to identifiers.
>
> *Keywords*, a subset of identifiers, behave like 'reserved' words. These terminals are written in double quotes in a grammar specification, e.g. `"while"`.
>
> *Symbols*, which are sequences of non-alphanumeric characters, are also written in double quotes, e.g. `"<="`.

Numbers, strings and identifiers are value terminals and keywords and symbols are constant terminals. Layout characters are ignored except that they may separate terminals.

The standard lexer is implemented as a higher order function that is parameterized with respect to the keywords and symbols. It is called with a set of keywords and a set of symbols and returns a specialized lexer. If no lexer is specified for the grammar then the standard lexer is used as a default, and is automatically parameterized with the keywords and symbols of the grammar.

Acceptable lexer speed is achieved by dispatching on the first input character to a function for handling each particular class of terminal. Constructing a specialized lexer is done by initializing three structures used by the lexer. The returned lexer is a closure containing the initialised structures. One can say that the standard lexer is table driven. The structures are:

- The fast dispatch table
- A hash table of keywords that is checked when an identifier is detected
- A set trie structures, one for each symbol dispatch character, for recognizing all of the symbols beginning with that dispatch character. Typically, the tries are small because the 'bushiest' first node is replaced by the fast dispatch table.

If performance is found to be inadequate then standard techniques could be used to convert the lexer specification into a deterministic finite state automaton [Aho *et al.* 1986], perhaps lazily [Heering *et al.* 1987].

4.6.3 The Metalexer

A language which describes partial parse trees requires a lexical notation or *escape* to indicate the omitted part of the parse tree. The metalexer provides a standard syntax for the metalevel escapes. To summarise, this notation allows two forms of escape, unnamed and named:

- An opening angle-bracket followed directly by the name of a nonterminal or value terminal in the grammar, followed directly by a closing angle-bracket, for example "`<factor>`".
- An opening angle-bracket, followed directly by the name of a nonterminal or value terminal in the grammar, followed by whitespace, an identifier and closing angle-bracket, for example "`<factor f1>`".

Any other input causes the metalexer to return the FAIL token. The metalexer is a higher order function parameterized by the names of the nonterminals and value terminals in a grammar. It returns a lexer that is specialized to recognising escapes for that grammar. The returned lexer is to be combined with the 'natural' lexer for the grammar by using the 'sequence' lexer combining form. Implementation of the metalexer is similar to the standard lexer and the comments that apply to the implementation of the standard lexer also apply to the metalexer.

A *level shift* terminal is returned when the metalexer recognises an escape. The level shift terminal is a value terminal with three components to the value:

- The grammar being parsed
- The nonterminal or value terminal that is escaped
- The identifier used in the escape. This defaults to the name of the nonterminal or value terminal.

As a level shift is a value terminal all of this information is available in the parse tree.

4.7 Parsing in a top-down framework

In this section a top-down parser is developed. The parser uses the grammar as a specification of a Top Down Parsing Language (TDPL) of [Aho & Ullman 1973], in a similar manner to the implementation of Meta-Lisp [Lajos 1990]. It has a limited backtracking ability. This, together with a special approach for direct left recursive rules, provides a powerful parser which is usually efficient.

A TDPL gives an operational reading to a grammar. Each nonterminal in the grammar defines a parsing function. The parsing function attempts to match some prefix of the input stream. If the match succeeds then the prefix is removed from the input stream and a parse tree representing the recognised input is returned.

The parsing function takes a sequence of tokens and returns either a failure or a pair: the recognised parse tree and the remainder of the input.

$$parser : [token] \rightarrow (parsetree, [token]) \cup \{FAIL\}$$

Using one parsing function per nonterminal has the advantage that special parsing functions can be defined for handling exceptional circumstances, for example, changes in the language or lexical conventions being used.

4.7.1 Generating parser functions

This section describes how a Lisp parsing function is generated from a set of rules. All the rules have the same left side nonterminal.

The input stream is implemented as a lazy list of tokens. Only the tail of the list is lazy. The list is lazy because metalevel parsing might require an alternative lexical analyser to be used. The alternative lexical analyser might interpret the input as a different sequence of tokens.

The Lisp function returns `NIL` to indicate failure, or a pair represented as a cons cell containing the parse tree in the car and the remainder of the stream in the cdr. Thus the parser function can be viewed as a function which replaces a prefix of the stream with its parse tree, or returns `NIL`. Because Lisp regards any value other than `NIL` to indicate a 'true' value, the Lisp parser function can also be viewed as a predicate which indicates truth if it can match some prefix of the input stream.

Single symbols are parsed differently depending on what type of symbol they are.

Nonterminals are parsed by calling the parser function for that nonterminal.

Constant terminals are parsed by testing the token in the input stream.

Variable terminals are parsed by testing the tag of the token in the input stream.

BNF provides two basic constructions: alternatives and sequences.

Alternatives. The essence of parsing the rules

$$\begin{array}{l} X \rightarrow \alpha_1 \\ \quad\vdots \\ X \rightarrow \alpha_n \end{array}$$

is to try to match each α_i with the input in turn, returning the first match.

Sequences. The essence of parsing the sequence of symbols $X_1\ X_2\ \ldots\ X_n$ is to parse the symbols in turn, each matching against the input not consumed by the previous symbol.

4.7.2 The basic translation scheme

Now we consider how to compile a list of rules into a parser function. Take the following set of rules for a nonterminal X. The $X_{i,j}$ are either terminals or nonterminals:

$$\begin{array}{ll} X \rightarrow X_{1,1}\ X_{1,2}\ \ldots\ X_{1,k_1} & (r_1) \\ X \rightarrow X_{2,1}\ X_{2,2}\ \ldots\ X_{1,k_2} & (r_2) \\ \quad\vdots & \\ X \rightarrow X_{n,1}\ X_{n,2}\ \ldots\ X_{n,k_n} & (r_n) \end{array}$$

The operation of the parse function is as follows. First $X_{1,1}$ attempts to parse the input. If it succeeds then $X_{1,2}$ attempts to parse the remaining input. If all the $X_{1,j}$ succeed then a parse tree is constructed for the production r_1 containing all the subtrees recognised by the $X_{1,j}$. This parse tree and the remaining input not matched by X_{1,k_1} are returned.

If any the $X_{1,j}$ fails then alternative r_1 fails. The parsing function for X then backtracks and tries to parse r_2. If all the alternatives fail then the whole parsing function fails and returns `NIL`. Backtracking is limited to being local to the parsing function. Once the parsing function has returned a value then it will not be retried as, for example, a Prolog parser might. This makes it possible to control the parse time and removes the need for a trail.

The set of rules is compiled into a Lisp or-and tree. The name `ij.sj` denotes the *(parse-tree,stream)* pair returned by the parsing function for $X_{i,j}$ and `sj` is the input stream component of this pair:

```
(defun X (s0)
  (or (let ((i1.s1 (X_{1,1} s0)))
         (and i1.s1
              (let ((s1 (cdr i1.s1)))
                (let ((i2.s2 (X_{1,2} s1)))
                   (and i2.s2
                        (let ((s2 (cdr i2.s2)))
                          ...
                          (let ((ik_1.sk_1 (X_{1,k_1} s2)))
                               (and ik_1.sk_1
                                    (let ((sk_1 (cdr ik_1.sk_1)))
                                      ...
                                      (cons
                                        (create parse tree node,
                                           using s1 ... sk_1)
                                        sk_1))))))))))
      ⋮
      (let ((i1.s1 (X_{n,1} s0)))
         (and i1.s1
              ⋱
                    ))))
```

The structure of the and-part is partly obscured by the operations that maintain the correct position within the input stream. Ignoring these issues and considering the $X_{i,j}$ as predicates the structure of the parser function is:

```
(or (and X_{1,1} X_{1,2} ... X_{1,k_1})
    (and X_{2,1} X_{2,2} ... X_{2,k_2})
      ⋮
    (and X_{n,1} X_{n,2} ... X_{n,k_n}))
```

The special case of the empty rule

$X \rightarrow$

is handled uniformly because `(and)` reduces to the identity element for conjunction, *true*; a nonterminal with just the null production compiles to a function that does no testing.

As described so far this form of the parser function is inadequate for a metaprogramming system because:

- It is inefficient if two adjacent productions start with the same symbol $X_{i,1}$ because if the first alternative fails then the $X_{i,1}$ function is called again.
- It loops endlessly if given a left-recursive grammar.
- It cannot parse language shifts.

The next few sections show how these problems are overcome.

4.7.3 Removing local backtracking

The efficiency of the basic translation scheme can be improved when several adjacent rules have the same prefix. The basic parsing function can be transformed so that shared prefixes are only parsed once. This is a variation on the technique for converting grammars to LL grammars called *left factoring*. Consider the rules

$$
\begin{aligned}
&X \rightarrow A\ B\\
&X \rightarrow C\ D\ G\\
&X \rightarrow C\ E\\
&X \rightarrow F\ B
\end{aligned}
$$

The or-and tree for these rules can be rewritten by factoring out the shared prefixes:

```
(or (and A B)
    (and C (or (and D G)
               (and E)))
    (and F B))
```

Another issue is that some rules may shadow other rules. If a set of productions has two productions and the r.h.s. of one production is prefix of the other then the longest rule should be tested first otherwise it will never match. This is easily detected and corrected when factoring shared prefixes. The factored tree for

$$
\begin{aligned}
&X \rightarrow C\ D\\
&X \rightarrow C\ D\ G\\
&X \rightarrow F\ B
\end{aligned}
$$

is

```
(or (and C D (or (and)
                 (and G)))
    (and F B))
```

The term `(or ... (and) ...)` is simply rewritten as `(or ... ... (and))`.

4.7.4 Left recursion

The inability of many top-down parsing techniques to handle any kind of left recursion is a major impediment to their use. Direct left recursion is handled as a special case as in [Lajos 1990]. While this technique cannot handle general left recursion, direct left recursion is essential for constructing correctly structured parse trees from expressions containing 'left associative' operators like "+". The directly left recursive rules

$$
\begin{aligned}
X &\rightarrow \alpha_1 \\
&\vdots \\
X &\rightarrow \alpha_m \\
X &\rightarrow X\ \beta_{m+1} \\
&\vdots \\
X &\rightarrow X\ \beta_n
\end{aligned}
$$

can be compiled into a parser function by recognising the base case first and building up the parse tree for the recursive productions with a loop. First the α_i are tried in order. If they all fail then X has no parse and the parse function fails. If an α_i succeeds then it is taken as the base case of the recursion, and an X has been recognised. Next an attempt is made to extend the parse by seeing if any β_i follows the recognised X. If a β_i is recognised then one of the recursive cases has been recognised an the corresponding extended parse tree is built and the β is removed from the input. This process is repeated until the prefix of the remaining input does not match any β_i. The or-and tree version of the loop is:

```
X′ := (or α₁ ... αₘ)
while X′≠ FAIL
  X := X′
  X′ := (and X′ (or  βₘ₊₁ ... βₙ))
the final parse is X
```

Factoring of shared prefixes is done for both the α_i and the β_i.

4.7.5 Parsing escapes

Now I describe how the TDPL can parse sentential forms containing escapes. The first point to make is that if the metalexer is not used then the parser will never see the escapes. Parsing an 'ordinary' program is a special case of parsing a program fragment containing escapes. The escape mechanism is enabled by parsing with the metalexer and disabled by parsing with the standard lexer. Escapes can be parsed with a simple modification to the parsing function. The input contains escapes as value terminals, so each nonterminal X in the grammar can be augmented with the production

$$X \rightarrow X\_escape$$

where X_escape is an escape terminal for the nonterminal X. The parsing function is extended to check for an escape terminal with the correct nonterminal as though the productions had been augmented with this additional production. The left recursion case is handled correctly since the new terminal is one of the α_i. Inputs like "`<expr>+1`" will parse as a single `expr` as intended.

4.7.6 Parsing language shifts

The lingua language allows program fragments to appear as expressions and patterns. This section describes the special actions that were required to make the parser for

```
#
#       lingua.g - metalinguistic support
#

grammar lingua .

import level2 .

external languageShift0 "(lambda (s) (lingua-shift s nil nil))" .
external languageShift1 "(lambda (s) (lingua-shift s nil I1.S1))" .
external languageShift2 "(lambda (s) (lingua-shift s I1.S1 I3.S3))" .

primary -> languageShift .
pat0 -> languageShift .


languageShift -> "[" "[" languageShift0 "]" "]" .
languageShift -> ident ":" "[" "[" languageShift1 "]" "]" .
languageShift -> ident "." ident ":" "[" "[" languageShift2 "]" "]" .
```

Figure 4.6: Adding language fragments as patterns and constructors

lingua parse the fragments in other languages. Special action is required because parsing fragments in an arbitrary *named* language is a context sensitive computation. Recall that in a TDPL each nonterminal may be compiled into a parsing routine. The routine has a standard interface: it takes a stream of tokens and returns a *(parse-tree, remaining-stream)* pair, or it returns *failure*. In principle any function providing the same interface can be used in place of a nonterminal. The *grammar* language makes provision for specifying arbitrary parsing functions via the `external` directive. This directive has the form

`external` *`name string`* `.`

What this means is that nonterminal *name* is not an ordinary nonterminal with an ordinary parsing function. The rule compiler handles external nonterminals differently. Instead of a call to a parsing function for *name* the Lisp code written in the string is called instead.

The nature of the replacement mechanism means that the replacement code must have the same type as a parsing function: a function of one argument returning a parse tree and a residual token stream. To achieve left-context dependence the left context must be made available to the replacement function. If the replacement function is just a named function, e.g. "`myparser`", then this is not possible. However, if the replacement form is an open-coded function, i.e. a lambda-form, then the intermediate return values from the parse functions for the left-context are in scope. The parse for the kth left-context symbol is bound to the variable `I`k`.S`k. Right-context dependence is not possible.

The external mechanism is obviously not context free as the replacement function can make arbitrary computational decisions as to the acceptability of the input. A simple demonstration of these ideas is the grammar fragment below:

```
external same "(lambda (s)
                 (if (equal (car I1.S1) (car I2.S2))
                     (cons 'same s)
                     NIL))" .
foo -> bar bar same .
bar -> ...
```

This fragment accepts two `bar`s only if they are identical. The external `same` compares the parse trees of the first and second symbols in the same production. The parse tree of the first `bar` is written as "`(car I1.S1)`" because the parse function returns the *pair* containing the parse tree and the remaining input. If `bar` accepts the language $\{\mathtt{a}^n\mathtt{b}^n \mid n \in \mathbb{N}\}$ then `foo` accepts the language $\{\mathtt{a}^n\mathtt{b}^n\mathtt{a}^n\mathtt{b}^n \mid n \in \mathbb{N}\}$, which is clearly not context free.

The external parser mechanism is used to construct a language form that parses fragments in any language known to the system. Figure 4.6 details a grammar module that extends the language *level2* with patterns and expressions for program fragments. Three forms of program fragment are allowed:

> *grammar*.*entry*:`[[` *text* `]]`
> *entry*:`[[` *text* `]]`
> `[[` *text* `]]`

Each form uses a different external that calls a general handler, `lingua-shift`.

The syntax brackets, "`[[`" and "`]]`", are specified as pairs of single symbols instead of single symbols comprised of two characters. This is to avoid problems with the lexical analysis of the combined language. If the latter course were taken then inputs like "`a[b[i]]`" would be analysed incorrectly, returning the token "`]]`" instead of two "`]`"s.

The function `lingua-shift` takes the stream and the grammar and entry name and returns either the parse or failure. At first this may seem a simple issue because the parser function can be found from the grammar and entry names by looking it up in the system descriptions of the known grammars. Complications arise because the new language may have completely different lexical conventions. The implementation of `lingua-shift` is quite involved because it has to work with the lexical analysers to ensure the correct synchronisation on the input. The following happens when `lingua-shift` is called:

1. The current lexer is suspended. The last token read is placed back on the input.
2. The metalexer for the specified grammar is activated on the input.
3. The appropriate parse function for the specified grammar and entry is found. It is called, using the restarted token stream. The metalexer is used rather than the lexer to make it possible to use escapes.
4. If the parse function fails then this is reported.
5. The new metalexer is suspended, and the last token read is placed back on the input.

n	10 levels	20 levels	40 levels	50 levels
100	0.52	0.80	1.35	1.60
200	1.01	1.60	2.66	3.23
300	1.51	2.40	4.05	4.85
400	2.08	3.22	5.35	6.40
500	2.53	4.05	6.70	8.06

Table 4.1: Parsing timings

6. The original lexer is restarted.

Tokens have to be placed back on the input because there is a one-token look-ahead required by the TDPL. This process also motivates my decision for a lexical analyser to return the token FAIL instead of raising an error when it can't understand the input. If the first bit of the program fragment after the opening syntax bracket just happens not to be a token in the lingua language then the fail token is returned. An error would not allow the correct metalexer the 'second chance'. The same reasoning applies to the closing syntax bracket "`]]`", which might not be a (prefix of a) legal token in the program fragment language.

4.7.7 Time behaviour

This section looks at the time behaviour of the modular parsers. It is important that implementation of the modular parser does not have any substantial parse-time costs and also that the parser can be constructed efficiently.

4.7.7.1 Parsing time

The performance of a parser constructed from a modular grammar is checked to see if it is reasonable. It is expected that the the run time is linear with respect to the length of the input and linear with respect to the size of the parse tree that must be constructed.

The test that was devised to show this used the hierlevel module to construct expression grammars with different depths of precedence hierarchy. Each grammar was tested on the same set of inputs. The inputs had lengths of n=100,200,300,400 and 500 tokens. The inputs were all sequences of numbers separated by operators of the lowest precedence. This produces the largest parse tree because each number is at the bottom of a long chain of unit productions. Grammars with hierarchies of depth d=10,20,40 and 50 levels were used. The results are as expected, as can be verified from table 4.1. The parse time is strictly linear in both the size of the input and the depth of the hierarchy. The size of the parse tree for input size n and hierarchy depth d is $4n + 4nd - 3$ cons cells. This is independently linear in both n and d.

4.7.7.2 Reducing parser construction time—Parse function abstraction

Grammars which are derived from subgrammars and imported grammars often share substantial sections. This section looks at methods which use this characteristic of related grammars to reduce the time and space requirements of generating a parser for the derived grammar.

If a nonterminal appears in two languages with identical sets of productions then the analysis and parser generation for the nonterminal may be shared. This observation can be carried further—nonterminals may have sets of productions which are sufficiently similar that the parser code may be shared. For example, the rule sets

$$\mathsf{expr} \rightarrow \mathsf{expr\ addop\ term}$$
$$\mathsf{expr} \rightarrow \mathsf{term}$$

and

$$\mathsf{term} \rightarrow \mathsf{term\ mulop\ factor}$$
$$\mathsf{term} \rightarrow \mathsf{factor}$$

have the same structure, namely

$$\mathsf{X}_1 \rightarrow \mathsf{X}_1\ \mathsf{X}_2\ \mathsf{X}_3$$
$$\mathsf{X}_1 \rightarrow \mathsf{X}_3$$

The structure is revealed by uniformly replacing nonterminals in the productions with standard names X_1,X_2,... Instead of generating a parsing function for each rule set in the system, a single parsing function is generated for each different structure. This parsing function is parameterized by the actual nonterminals to yield the desired parsing function. The parsing function also needs to be parameterized by the l.h.s. symbol even if it does not occur on the r.h.s. because of the check for escapes (section 4.7.5).

Since the order of the rules is significant in a TDPL it is sufficient to compare the structure of a rule set with the structure of rule sets that are already in the system. This can be done efficiently by looking up the structure in a hash table.

If the order of the rules is unimportant then comparing a structure with another is more difficult. In general this problem is combinatoric, but the search space may be reduced by discriminating between rule set structures on the following measures. Each measure partitions the space of rule set structures.

- The distribution of lengths of the right-hand-sides of the rules. A rule set with two rules of length 2 and one rule of length 1 cannot unify with a rule set with two rules of length 1 and one rule of length 2. This subsumes the measure of the number of rules in a rule set.

- The distribution of the frequencies of the X_i. A structure with four X_2s and an X_1 would not unify with a structure with two X_1s and an X_2.

- The distribution of the frequencies of X_i within a rule partitions the rule set into smaller sets.

Any (or all) of these measures could be used to construct a cheap hash function that did most of the work.

Parse function abstraction clearly saves space and some time. If the new language is a minor extension of the old language with then the number of new parser functions needed is a function of the number of nonterminals on the l.h.s. of the additional productions. Typically this is far fewer than the number of nonterminals in the original language. It is also worth noting that the parser functions for the production sets of a parameterized grammar are usually only generated once.

4.7.8 Reducing parse tree space requirements

A naive parse tree representation is one that contains a node for each production that is reduced in the parse of the input. Such parse trees consume quite large amounts of storage. More space-efficient alternatives are

- automatically generated abstract syntax trees [Noonan 1985].
- 'production trees' [Waddle 1990].

Both alternatives are algorithms that produce a representation for use in later phases of a compiler. The result of applying these algorithms usually removes the distinction between related nonterminals in the grammar. For example, the abstract syntax for a *factor* is usually identical to that for an *expression*.

Since our system uses syntax directly for the manipulation of programs it must maintain the ability make this distinction. The space saving measures described below are not as comprehensive as those of [Noonan 1985] and [Waddle 1990] but they are responsible for much of the success of these approaches. I have concentrated on those approaches which yield the most saving with the least disruption to the correspondence between the concrete syntax and the internal form. The intention is to present the user/programmer with the illusion that the entire parse tree is stored.

4.7.8.1 Constant terminal elision

Constant terminals in the right side do not provide any information over that already provided by the production tag. Both of

expr → "(" *expr* ")"

P_k	"("	*expr*	")"

expr → "(" *expr* ")"

P_k	*expr*

contain the same information because the position of the constant terminals can be recovered from the P_k tag. Space can be saved by omitting the constant terminals from the right side. The same reasoning does not apply to variable terminals like identifiers and numbers because each instance of a terminal may denote a different object. The value of a value terminal must be stored.

4.7.8.2 Unit production elision

A unit production is a production where the right side consists of a single nonterminal or a single variable terminal, for example

term → factor
variable → ident

Between 20% and 30% of the rules in a typical grammar are of this form[8]. The actual savings may be greater than this suggests. In a parse tree even more of the rules are likely to be unit productions.

Many of these rules come from two metaphors

[8]For example, lingua has 81 productions, 20 are nonterminal unit productions and five are variable terminal unit productions

Program	Naive tree cells	Constant terminals cells	factor	Unit productions cells	factor
P1	52464	47690	0.91	12968	0.27
P2	12884	11593	0.95	3647	0.31

Table 4.2: Effectiveness of space-saving measures

- the hierarchy of nonterminals used to express precedence levels
- 'end case' productions of list-like nonterminals

Unfortunately while unit production elision has the potential for considerable space saving, it does not fit well with ELDERII because the unit productions carry *type* information. Programs (for example, `macro-scheme`) need to know the root nonterminal of the parse tree.

4.7.8.3 Effectiveness

The effect of both of the above measures was calculated for two example programs. The results of these calculations appear in table 4.2. Removing unit productions reduces the size of the parse trees to less than a third of the original size. Removing constant terminals saves between 5% and 10% of the storage. These figures are roughly in line with those reported by Waddle. The effects of each measure are additive because they affect independent sets of productions. Given a saving of say 70% from unit production elision the importance of the constant terminals is magnified. It is unfortunate that the unit production elision is awkward to integrate into the current system.

4.7.9 Integrating alternative languages with the host Lisp system

When using several notations with different syntactic and lexical conventions it is necessary to be able to determine which conventions are to be used to input a particular piece of text. Files containing text are tagged. Each file starts with the line

`#language` *grammar* *entry* *compiler*

The text `#l` is a Common Lisp read macro. When the Lisp reader sees this it invokes a function which inspects the rest of the line and reads the rest of the file accordingly. *Grammar* and *entry* specify what grammar is used to parse the rest of the file. *Compiler* is the name of a Lisp function that translates the parse tree into a Lisp object (perhaps a sequence of Lisp definitions). An important compiler function is `grammar->lisp` which converts a grammar description into a parser. If no translation is required then the function `identity` can be used. Using the Common Lisp read-macro to call the appropriate parser neatly integrates all the new notations into the Lisp environment, allowing the file to be read, loaded and even compiled.

4.7.10 Bootstrapping ElderII

An earlier section (4.3.4) has described how a metaprogramming language was built on top of an ordinary language with pattern matching. I describe now how the grammar system was originally built. The grammar rule compiler is written in Lisp. Initially a very simple *boot grammar* was hand coded in the internal representation of grammar

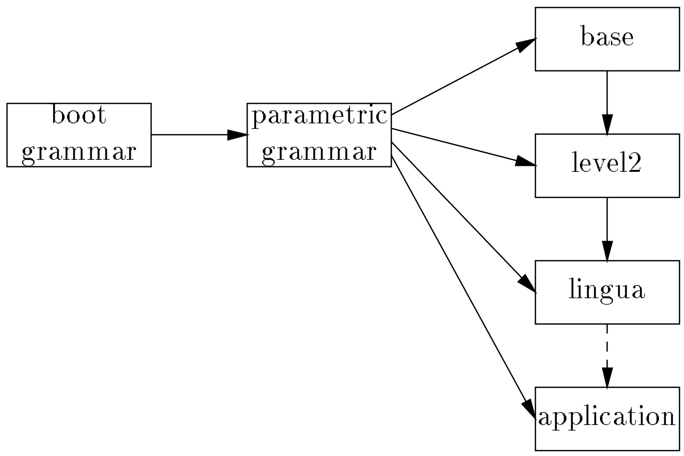


Figure 4.7: Bootstrapping.

rules. This grammar could only describe simple productions and consists of six rules. The grammar rule compiler was used to convert it into a parser. The boot grammar was then used to define the *grammar* grammar as displayed in figure 4.2. Next, the Lisp metaprogramming tools were used write a program to convert specifications written in the full grammar notation into the internal form understood by the rule compiler. Subsequent stages of translation are also written with the Lisp tools as the lingua language is defined quite late in the bootstrapping process.

4.8 Other modular parsing techniques

I chose a top-down parsing language as a basis for the modular parsing system of ElderII. There is a straightforward relation between the context free grammar and the parsing procedures which makes it simple to implement a modular parser. Each parsing function uses only information local to the nonterminal which is accepts. Other parsing techniques use information which is global to the grammar. This complicates the modular implementation. In this section I describe two alternative approaches to the problem of parsing modular grammars. Then I describe how I would solve the problems of parsing program fragments with a shift-reduce parser.

4.8.1 Lazy LL(1) parsing

Koskimies describes a technique based on a recursive descent implementation of an LL(1) parser [Koskimies 1990]. The program is structured as a set of modules, one for each nonterminal. The modules are conventional modules in the sense of those of Modula2 or Ada.

The key to LL parsing is to determine the selection set for each production. The selection set is the set of input symbols which may start the r.h.s. of production (and those symbols which may follow the r.h.s. if it is nullable). If the grammar is LL(1) then the selection sets of the productions of any nonterminal are disjoint. The parser chooses the production which has the next input symbol in its selection set.

The selection sets are derived from the FIRST and FOLLOW sets. Unfortunately the FIRST and FOLLOW sets are global properties of a grammar. This poses the problem: how can we write a recursive descent style parsing routine for a nonterminal knowing only the productions for that nonterminal and not knowing the productions for other nonterminals or even the terminals used in them. Essentially, the whole language is only known at link or run time.

Koskimies solves this problem by *lazy parser construction*. Conventional parsing tables are constructed at run time as they are needed. All the information to do this is available when the modules are linked. Compare this with limited backtracking as used by ELDERII. The correct parsing actions are found by trying to parse and backtracking when this fails. In effect the parsing information is regenerated each time it is needed. Koskimies rejects the backtracking approach as semantic actions are integrated with the parsing actions and this makes the behaviour of the compiler nondeterministic. This is not a problem in ELDERII because parsing is done prior to any semantic evaluation.

The lazy parser construction could be done from an encoded form of the grammars. The result would be an interpreted parser. This would loose the benefits of compiling the parser to a set of recursive descent procedures. A new data structure, the *start tree*, encodes the relationship between symbols and their start sets. The start trees are constructed lazily and are used to predict the appropriate production. The method is as efficient as conventional LL(1) recursive descent parsing. Interestingly, the start trees contain information about the nonterminals that contribute to them. This would allow the technique to be extended to parse nonterminal escapes (see sections 4.3.2 & 4.7.5).

4.8.2 Modular LR parsing

An LR parser is based around the LR(0) automaton. Each state in this automaton corresponds to a set of partially parsed productions that might correspond to the input. The LR(0) automaton is constructed directly from the grammar. The modular LR parsing problem is: given two LR(0) automata constructed from two grammars G_1 and G_2, how do we construct the compound automaton corresponding to the grammar $G_1 \cup G_2$?

The construction problem is a hard as constructing the compound LR(0) automaton from scratch. The problem is that the compound automaton usually bears little resemblance to the original automata because the grammars interfere with each other. Rekers [Rekers 1989] suggests that instead of trying to construct the compound automaton from the subparts, one huge automaton is created containing the union of all the grammar modules. The automaton for any subset of the modules can then be found by restricting the compound automaton to those actions related to productions in the subset. This system (MPG) has been implemented by Rekers and is an extension of an earlier incremental (rules are added one at a time) and lazy parser generator, IPG [Heering *et al.* 1989b] that are part of SDF. MPG is incremental—adding one production to a module causes an adjustment to the compound automaton which is reflected through the restricted view (if the edited module is one the subset defining the language being parsed).

The MPG solution does not address all of the issues of modular parsing. For example, the issues of parameterized modules is not discussed. Another failing of IPG (and hence MPG and SDF) is that some parse tree disambiguation is done after the parsing is complete. The parser that SDF uses is a variant of Tomita's parser [Tomita 1980]. When there is a parsing conflict (shift-reduce or reduce-reduce) a parser is spawned for each of the conflicting actions. If the input is ambiguous then several parse trees are the result. If one of the parsers fails then that thread is abandoned. Only when all the threads fail is the input unparsable and in error. Left and right associative operators are handled by the parsing table but operator precedence is handled by a separate process of choosing one of the parse trees from the multiple results. This is inefficient. An expression like "`a1*b1+a2*b2+a3*b3`" has 20 different parses if operator associativity is done by the parse table (as in SDF). It is clear that a program with several expressions like this will have a huge number of parses.

4.8.3 Other LR issues

The following sections discuss the solution of some of the parsing problems that would be faced implementing ELDERII using a shift-reduce parsing scheme. It is assumed that the reader is familiar with the principles of shift-reduce parsing, which are explained in [Aho *et al.* 1986].

4.8.3.1 Parsing multiple axioms

Usually an LR parser is constructed to parse a single axiom. The grammar is augmented with the production "$\mathsf{start}' \rightarrow \mathsf{start}\ \$$" where $\$$ is the end of input symbol. When the end of input symbol is about to be shifted the input is accepted. If a grammar has multiple axioms (i.e. is to be used to parse for the language of more than one nonterminal) then something similar must be done for each nonterminal. It is incorrect to add the set of productions

$$\mathsf{start}' \rightarrow \mathsf{X_i}\ \$$$

as this will most likely cause ambiguity and parse conflicts. The solution adopted by Horspool in the ILALR parser generator [Horspool 1990] is to add the set of productions

$$\mathsf{start}' \rightarrow \$_{\mathsf{X_i}}\ \mathsf{X_i}\ \$_{\mathsf{X_i}}$$

Each nonterminal is bracketed by its own private end of input symbol. To parse for $\mathsf{X_i}$ the symbol $\$_{\mathsf{X_i}}$ is pushed in front of the input. This selects the correct nonterminal. The production also ends with the nonterminal's private end of input symbol, and not $\$$. This ensures that $\$$ does not appear in any follow sets other than those that it would normally have, which might cause shift-reduce conflicts.

4.8.3.2 Input containing nonterminals

A conventional LR parsing table is divided into two parts: an ACTION table and a GOTO table. The ACTION table is consulted with the next input terminal to decide whether to shift, reduce, accept the input or raise an error. When a reduction takes place the GOTO table is consulted with the newly recognized nonterminal. This distinction hides the fact that both tables encode part of the LR automaton and it is really a representation optimization that yields a small efficiency factor in accessing and encoding the parsing tables. The distinction is possible because nonterminals never appear in the input. The GOTO table encodes the shift actions of the nonterminals.

An alternative more uniform vision of a shift-reduce parser's action is gained by discarding the GOTO table and encoding everything in the ACTION table. In this scheme a reduction is performed by removing the r.h.s. of the production from the stack and placing the l.h.s. nonterminal back on the input. The next primitive cycle will see the nonterminal and select the shift action that would normally be part of the

GOTO table. The parser will always be in the state revealed by removing the r.h.s. from the stack which is a state expecting that nonterminal in the input. A nonterminal is only ever the first item in the input and is always shifted. (Using a GOTO table reduces the size of the parse table and the number of primitive operations, giving a small improvement in parsing speed.)

The nuisance of reduced performance caused by removing the GOTO table is outweighed by the benefit of making it possible to modify the parser to accept input containing arbitrary nonterminals (not just the ones put there by reduction). The modifications are merely to include the nonterminals in all the sets used to decide on the parsing actions. For example, take this grammar for begin-end blocks of procedure calls:

$$\begin{aligned}&\mathsf{S} \rightarrow \text{“}\texttt{begin}\text{”}\ \mathsf{SL}\ \text{“}\texttt{end}\text{”}\\ &\mathsf{S} \rightarrow \text{“}\texttt{proc}\text{”}\\ &\mathsf{SL} \rightarrow \mathsf{SL}\ \mathsf{S}\\ &\mathsf{SL} \rightarrow \epsilon\end{aligned}$$

The state containing the reduction $\mathsf{SL} \rightarrow \epsilon$ has the items

$$\{\ \mathsf{S}\rightarrow\text{“}\texttt{begin}\text{”}\bullet\mathsf{SL}\ \text{“}\texttt{end}\text{”},\quad \mathsf{SL}\rightarrow\bullet\mathsf{SL}\ \mathsf{S},\quad \mathsf{SL}\rightarrow\bullet\ \}$$

The reduction is chosen when the input is in FOLLOW(SL), which contains the three tokens {“begin”,“end”,“proc”}. Ordinarily there is no table entry for S for this state. If the follow set is extended to include contributing nonterminals (in this case just S) then the reduction can occur on the input being the nonterminal S, allowing sentences like

`begin` ⟨S⟩ `proc` ⟨S⟩ `proc end`

to be parsed. The first S requires the new reduction, the second requires the shift that was already part of the GOTO table.

4.8.3.3 Language shifts

The problems of parsing a language shift, for example “`expr:[[<term>+<term>]]`”, in a shift-reduce framework are similar to the problems encountered in building ELDERII. The activities of substituting an alternative lexical analyser are the same. The main difference is in handling errors caused by the illegal prefix “`]]`”. A top down parser handles the error gracefully. It fails on any production that requires the illegal input to be shifted and backtracks to return as much as possible. In contrast, a shift reduce parser detects the error at the earliest possible moment. The solution is to modify the parser's error behaviour when parsing language shifts, and is similar to the approach taken by CAML[9]. Recall that the parser will parse an expr by the production $\mathsf{start'} \rightarrow \$_{\mathsf{expr}}\ \mathsf{expr}\ \$_{\mathsf{expr}}$. The symbol $\$_{\mathsf{expr}}$ is placed on the input after “`[[`”. A second $\$_{\mathsf{expr}}$ is required to end the language shift, but the correct position is not known in advance. The parser is modified to insert the correct $\$_{\mathsf{X_i}}$ at the first parsing error. The error can occur for two reasons: either the terminal starting at “`]]`” is not a terminal in the language in which case the lexical analyser returns the FAIL token, or it is a valid token but it is inappropriate. Parsing is resumed, hopefully shifting the $\$_{\mathsf{expr}}$ and reducing to expr. If there are further errors then the input is really in error.

[9] Personal communication

Note that both the top-down and shift-reduce parser may inadvertently accept part of the "]]" if it could be part of the language shift. Not many formal languages use unbalanced square brackets so this is unlikely to happen. In this regard the syntax brackets "[[" and "]]" used in ElderII are an improvement over the 'french quotation' signs "<<" and ">>" used in CAML. These signs cause problems in parsing text of languages that have relational operators and C-style shift operators which are not balanced like brackets. Such languages are common enough to warrant a different notation.

4.9 Summary

This chapter has described the design, use and implementation of modular context free grammars. Modular CFGs are useful for describing syntaxes that consist of several large mostly independent modules as well as for describing dialects which differ only modestly from some base language.

The grammars have been used for applications that manipulate both programs and more conventional data as syntactic objects. To make these manipulations convenient the concrete syntax of the subject language was used. I have described the special considerations in the design and implementation of the parsers that allow program fragments to be used in efficient pattern matching and data structure construction.

The exercise of using the modular grammars, and in particular the parameterized modular grammars, has revealed that more support is needed for *using* a modular grammar. It is difficult to write a program that still works when the grammar is extended or used as part of another notation. These problem are similar to those discovered in chapter 3 and are addressed in the next chapter.

Finally, the techniques that I developed and used are based on a particular type of parsing technique: limited backtracking top down parsing. I show how to achieve similar results with a shift-reduce parser.

5 Modular Attribute Grammars

Modular attribute grammars are a solution to the problem of dividing up a translation task according to the features of the language. The modular system was developed by a process of taking a conventional attribute grammar and generalizing the statements and definitions which that attribute grammar could state. The system was developed hand-in-hand with an executable version, written in a lazy functional language.

The development was in three stages. First, a correspondence between attribute grammars and (lazy) functional programs was established. The correspondence was described by Johnsson [1987]. I give an algorithm for writing an attribute grammar as a functional program written in Miranda, quite similar to Johnsson's algorithm, but based on the parse tree rather than the actione of a shift-reduce parser. Second, the form of the functional program was changed to separate the concerns of calculating the attributes and the nature of their dependencies. This yielded a concise and general attribute grammar processor which is parameterized by a mapping from grammar productions to the attribute equations. The third and most interesting stage was the replacement of the monolithic mapping from productions to attribute equations by a structured composition of several functions, each responsible for some different aspect of the attribute grammar. Each function, being the subject of an assignment of responsibility, is a module.

This chapter describes these three stages. The result is a modular attribute grammar processor based not on the conventional graph-theoretic analysis of dependencies, but on an execution model. The work was inspired by the relationship between lazy functional languages and attribute grammars [Johnsson 1987] and the result is similar in several respects to the modular attribute grammars of [Dueck & Cormack 1990], from which the terminology and formal description of an attribute grammar is taken.

5.1 Classical Attribute Grammars

Attribute grammars are a declarative formalism for describing the syntax, semantics and translation of programming languages and other formalisms. The syntax is provided by a context free grammar (CFG). The semantics and translation are described by a set of rules for calculating attribute values which are associated with nodes in the parse tree.

An attribute grammar (AG) is a tuple (N, T, S, P, A, R):

N is the set of nonterminal symbols.

T is the set of terminal symbols. V, the vocabulary, is the set of all symbols. $V = N \cup T$.

S is the start symbol. $S \in N$.

P is a set of productions. Each production $p \in P$ written $X_0 \rightarrow X_1 X_2 \ldots X_n$ where $X_0 \in N$ and $X_i \in V$ $(1 \leq i \leq n)$.

A is a set of attribute symbols used to name attribute values.

R is a set of attribution rules or attribute computations, each of the form (p, D, U, f), where $p \in P$, D is an attribute identification of the form (i, a) $(0 \leq i \leq |p|, a \in A)$ which uniquely identifies an attribute in the production p by its position and name. U is a sequence of identifications of the same form as D, and f is a function which defines the attribute identified by D in terms of those identified by U. An attribute reference (i, a) is written $X_i n{\uparrow}a$ or $X_i n{\downarrow}a$ where n selects nth symbol X_i from the left of the production p, and may be omitted if X_i occurs only once in p, or if $n = 1$ and it is the l.h.s. symbol. An up arrow indicates a synthesized attribute and a down arrow indicates an inherited attribute; this explained below.

A *parse tree* is a derivation of a string from the context free grammar $G = (N, T, S, P)$. Each leaf is labeled by a terminal. Each expansion of some production $p \in P = X_0 \longrightarrow X_1 X_2 \ldots X_n$ where $X_0 \in N$ in the derivation is represented by an internal tree node labeled with X_0 and with children labeled $X_1, X_2, \ldots, X_n$ in order.

Each node in the parse tree has associated with it a set of attributes and attribute values. The values of the attributes are defined by the attribution rules. Two kinds of attribute are distinguished depending on where the values are defined: *synthesized* and *inherited* attributes. The attribute rule (p, D, U, f) where D is of the form $(0, a)$ specifies the computation of synthesized attribute a for each node n in the parse tree representing the expansion of production p in the derivation. If D is of the form (i, a) for $i > 0$ then the attribute rule specifies the computation of the inherited attribute a for each node which is the ith child of a node n representing the expansion of p. The attribute value is the value obtained by applying f to the values identified by U. Each $(i, b) \in U$ denotes the value of attribute b of either n (if $i = 0$) or the ith child of n. Synthesized attributes may be thought of traveling up the parse tree, and inherited attributes as traveling down the tree.

Attribute rules are usually written as equations beneath the production to which they apply. The order of the equations is of no importance. The following simple attribute grammar illustrates most of the above points. It calculates the value of a summation of numbers and "x"s, for example, the value of "3+x+6+x":

```
expr → expr "+" expr
    expr1↑value = expr2↑value + expr3↑value
    expr2↓param = expr↓param
    expr3↓param = expr↓param
expr → number
    expr↑value = number↑value
expr → "x"
    expr↑value = expr↓param
```

Taking the middle rule first, the rule synthesizes a value attribute for expr, and the value is that of the number. The last synthesizes a value attribute for expr, and its value is determined from the inherited attribute called param. The first rule sums the value attributes of the subexpressions. It is also passed down the inherited param attribute to the subexpressions. Notice how the different exprs are identified by number and that expr1 may be written as expr.

This example is typical in that it computes a value that depends on the contextual information held in the inherited attributes. However, it is simplistic: a real application would compute some structured value instead of a number; a traditional translator would compute an intermediate or assembly code program. The dependencies between the attribute would also be more complex: in a real application the inherited attributes would themselves be computed depending on other parts of the input, for example a declaration of "`x`" as scalar or vector might alter the operation(s) chosen to perform the addition.

An attribute grammar specifies a static semantics because an attribute of a production or nonterminal may attain only one value. A dynamic semantics must therefore be defined by a static collection of dynamic parts, e.g. functions which may be invoked multiple times to run the program.

5.1.1 Terminal attributes

I extend the attribute grammar framework to include terminals as well as productions. Value terminals like numbers and identifiers denote particular members of a class of terminals, and have some property to identify the individual, e.g. the text of the input token. This is usually made visible to the attribute grammar as a single synthesized attribute that is computed by the lexical analyser. Allowing a terminal to be used in place of a production in the attribute grammar framework gives the programmer more control. A single synthesized attribute is still provided by the lexical analyser but additional attributes may be computed, either making up for some inadequacy of the lexical analyser or responding to some contextual constraint. An example of the former is when a lexical analyser returns an identifier in mixed case when the language is not supposed to be case sensitive. The programmer could define a new attribute for the identifier in which all the letters have been changed to upper-case. An example of the latter is interpreting a number in the context of a different radix or choosing an operator depending on an expected type.

5.1.2 Connection with lazy functional languages

Attribute computations are declarative, that is in the AG there is a statement declaring what the value of a particular attribute is. This value is declared exactly once and is described as a function of the values of other attributes of the symbols in the rule in which the attribute value is declared. The fact that some attributes are defined as functions of other attributes establishes a dependency relationship between the attributes. Much work has been done to use this relationship to calculate an efficient evaluation order for the attributes [Farrow 1982, Kastens *et al.* 1982]. The resulting evaluators usually place restrictions on the form of the dependency relationship.

Using a lazy functional language relieves the programmer or processor from the burden of determining an evaluation order for the attributes. The normal order reduction of the lazy language ensures that an attribute is only evaluated when it is needed. No sophisticated analysis is needed to determine an attribute evaluation order. With careful naming of expressions and provided that there are no circular dependencies each attribute will only be evaluated once.[1] In effect the dependency analysis is done

[1] Some circularities not normally permitted might be feasible just because we are using a lazy functional language. The key observation is that laziness of the functional language also applies to the expressions in the attribute equations. The value of `r` is well defined (as the tenth element of the infinite list p) in the following cyclic definition:

```
p = 1:[a*5|a<-q]; q = [a*3|a<-p]; r = p!10
```

This might be used to express iterative computations as a sequence of iterated results. It would

P1:	goal $\longrightarrow$ expr	
		goal$\uparrow$val = expr$\uparrow$val
P2:	expr $\longrightarrow$ term	
		expr$\uparrow$val = term$\uparrow$val
P3:	expr $\longrightarrow$ expr addop term	
		expr$\uparrow$val = callop(addop$\uparrow$operator, expr2$\uparrow$val, term$\uparrow$val)
P4:	term $\longrightarrow$ factor	
		term$\uparrow$val = factor$\uparrow$val
P5:	term $\longrightarrow$ term mulop factor	
		term$\uparrow$val = callop(mulop$\uparrow$operator, term2$\uparrow$val, factor$\uparrow$val)
P6:	factor $\longrightarrow$ int	
		int$\downarrow$scale = 0
		factor$\uparrow$val = int$\uparrow$val
P7:	factor $\longrightarrow$ "(" expr ")"	
		factor$\uparrow$val = expr$\uparrow$val
P8:	int $\longrightarrow$ digit	
		digit$\downarrow$scale = int$\downarrow$scale
		int$\uparrow$val = digit$\uparrow$val
P9:	int $\longrightarrow$ int digit	
		int2$\downarrow$scale = int$\downarrow$scale+1
		digit$\downarrow$scale = int$\downarrow$scale
		int$\uparrow$val = int2$\uparrow$val + digit$\uparrow$val
P10:	digit $\longrightarrow$ "0"	
		digit$\uparrow$val = 0
P11:	digit $\longrightarrow$ "1"	
		digit$\uparrow$val = $2^{\text{digit}\downarrow\text{scale}}$
P12:	addop $\longrightarrow$ "+"	
		addop$\uparrow$operator = add
P13:	mulop $\longrightarrow$ "*"	
		mulop$\uparrow$operator = mul

Figure 5.1: Binary arithmetic AG

dynamically as part of the evaluation process.

An evaluator can be constructed by writing one function per nonterminal in the grammar. The function has a case for each production of the nonterminal, in total there is one function case per production. Each function case specifies the synthesized attributes of the head nonterminal and the inherited attributes of the right side symbols. As a running example I use parts of the binary arithmetic expressions example from [Knuth 1968], which is reproduced in figure 5.1. This example is also used by Dueck & Cormack. Using the same example will allow direct comparison.

An attribute evaluator in Miranda can be constructed using the following methodology. The result is a single program produced from a single complete attribute grammar.

probably be a profligate consumer of storage (keeping all iterations) but the possibility of expressing iterative algorithms in an attribute grammar framework seems worthy of further investigation.

Symbols	inherited attributes	synthesized attributes
goal, expr, term, factor		val
int	scale	val
digit	scale	val
addop, mulop		operator

Figure 5.2: Binary arithmetic AG attributes

In anticipation of a modular formulation I call the program produced by applying this methodology the *monolithic translation.*

1. Create abstract syntax data types for the parse tree, labeled by the productions:

```
goal   ::= P1 expr
expr   ::= P2 term | P3 expr addop term
term   ::= P4 factor | P5 term mulop factor
factor ::= P6 int | P7 expr
int    ::= P8 digit | P9 int digit
digit  ::= P10 | P11
addop  ::= P12
mulop  ::= P13
```

   Any data structure could be used; the important thing is to be able to identify the production. This choice is particularly efficient since a dispatch on an algebraic type can be compiled into a computed jump, a constant time operation.

2. Determine the inherited and the synthesized attributes for each nonterminal. These attributes will be grouped together as two tuples of attributes: a tuple of inherited attributes and a tuple of synthesized attributes.

3. For each nonterminal *nt* (e.g. *expr*) create a function *nt*S. This function has a case for each production P_j

   $nt \rightarrow \mathsf{X}_1\ \mathsf{X}_2\ \ldots\ \mathsf{X}_n$

   The general form of the function case is

   *nt*S *production inherited_attributes* =
   synthesized_attributes
   where
   synthesized_attributes = $(e_1, \ldots, e_k)$

   Production is a pattern matching the production P_j, the e_i are the expressions for calculating the synthesized attribute values.

4. Extend the function to communicate attribute values to and from each symbol of the right side of the production. This is easily done by extending the where-clause above with a line for each right side symbol X_i defining its synthesized attributes as the appropriate function of its inherited attributes:

   X_i*synthesized* = X_iS *production*.X_i X_i*inherited_expressions*

The synthesized and inherited attributes are again tuples. $production.X_i$ is the field of *production* corresponding to the parse tree for X_i. The *letrec*-like scoping rules of Miranda's 'where' clause ensure that all of the attributes are visible in the expressions denoting the attribute values.

Applying these rules to the binary arithmetic grammar we obtain the code in figure 5.3. Each nonterminal has an attribution function which is a function of two parameters: the derivation tree and the inherited attributes. Each function is a case analysis for the productions for that nonterminal. For example, `intS` dispatches on productions P8 and P9, and there is a single inherited attribute, `int_scale`. Each function case calculates the synthesized attributes as a function of the inherited attributes and the attributes of the r.h.s. symbols.

The construction is essentially the same as that given by Johnsson, but differs as follows. First, the target language is Miranda instead of lazy ML. Second, the parse tree is partitioned into separate types which yields a separate function for each nonterminal. This could be relaxed at the expense of permitting illegal productions (which is not a major concern if the transformation is done automatically). Third, Johnsson's scheme was developed to investigate using attribute grammars as a functional programming paradigm. As a result the methodology was not applied to a large or diverse grammars and, as presented in [Johnsson 1987], would present unreasonable type constraints: all nonterminals must have the same number of identically typed attributes. This can be avoided by tagging.

5.2 Formulation of a general attribute grammar processor

I have described how an attribute grammar evaluator may be translated directly into a program in a lazy functional language. The synthesized attributes and the inherited attributes are both collected together as a single synthesized and a single inherited attribute. For each grammar production one function case is written to calculate the attributes.

By a process of abstraction we can arrive at a formulation of the attribute grammar which is completely independent of the number or form of the productions or the number and type of the attributes. With the attribute calculations abstracted out it is possible to break up the abstracted parts and recombine them in different ways. This provides a method of factoring the attribute rules into separate independent or loosely coupled sets or *modules*.

The following generalizations are taken to enable one general function to be written in place of all the attribution functions: the parse tree types are merged, attribute collections are reduced to a uniform type and the variety of individual attribute types are replaced by their discriminated union.

The parse tree type is redefined so that components of a production may be selected without specific knowledge of which production it is. A parse tree in its most general form consists of nonterminal derivations (internal nodes) and terminals (leaf nodes). The right side of the production is a list, so there is no constraint on its length.

```
production  ::= P nonterminal [production] | T terminal
```

For the example grammar the terminals and nonterminals are

```
goalS (P1 expr) () = exprS expr ()

exprS (P2 term) () = termS term ()
exprS (P3 expr2 addop term) () =
      (expr_val)
      where
      expr_val = callop addop_operator expr2_val term_val
      (addop_operator) = addopS addop ()
      (expr2_val)      = exprS expr2 ()
      (term_val)       = termS term ()

termS (P4 factor) () = factorS factor ()
termS (P5 term2 mulop factor) () =
      (term_val)
      where
      term_val = callop mulop_operator term2_val factor_val
      (mulop_operator) = mulopS mulop ()
      (term2_val)      = termS term2 ()
      (factor_val)     = factorS factor ()

factorS (P6 int) () =
        (factor_val)
        where
        factor_val = int_val
        (int_val) = intS int (0)
factorS (P7 expr) () = exprS expr ()

intS (P8 digit) (int_scale) =
     (int_val)
     where
     int_val = digit_val
     (digit_val) = digitS digit (int_scale)
intS (P9 int2 digit) (int_scale) =
     (int_val)
     where
     int2_scale  = int_scale+1
     digit_scale = int_scale
     int_val     = int2_val + digit_val
     (int2_val)  = intS int2 (int2_scale)
     (digit_val) = digitS digit (digit_scale)

digitS P10 (digit_scale) = 0
digitS P11 (digit_scale) = pow 2 digit_scale

addopS P12 () = ADD
mulopS P13 () = MUL

op ::= ADD | MUL
callop ADD n m = n+m
callop MUL n m = n*m

pow x n = entier(exp(n*(log x)) + 0.1)

k5 = P9 (P9 (P8 P11) P10) P11                        ||parse tree for '101'
demo = (P1 (P3 (P2 (P4 (P6 k5))) P12 (P4 (P6 k5))))  ||parse for '101+101'
```


Figure 5.3: A complete evaluator for the binary arithmetic language

Parse tree:

```
  int
  / \
int digit
 |    |
digit |
 |    |
 1    0
```

Sentence: "10"

Miranda data structure:

```
(P Int [P Int [P Digit [T ONE]],
        P Digit [T ZERO]])
```

Figure 5.4: Representation of a parse tree

```
nonterminal ::= Goal | Expr | Term | Factor | Int | Digit |
                Addop | Mulop
terminal    ::= LPAR | RPAR | ONE | ZERO | PLUS | TIMES
```

These type definitions use a special case of Miranda's algebraic types and are analogous to the enumerated scalar types of Pascal, C and Ada. Figure 5.4 relates an input sentence, its parse tree and its Miranda data structure representation.

A set of (synthesized/inherited) attributes for a production is generalized from a tuple, in which the attribute is identified by its position, to an explicit mapping from attribute names to attribute values. This implies that all attributes must be of the same type, so the new attribute type is the tagged union of all the original attribute types. In the example there are three attributes and two attribute types. The mapping is implemented as a list of name-value pairs:

```
attribute      ::= Val | Operator | Scale
attribute_type ::= N  num |
                   OP (attribute_type->attribute_type->attribute_type)
attributes     == [(attribute,attribute_type)]
```

These definitions say that there are three attribute names, val, operator and scale. There are two types which attribute values can attain, numbers and operators. Numbers are tagged with `N`. Operators are functions taking two attribute types values and returning a third, and are tagged with `OP`. The mapping from attribute names to values is declared as the type abbreviation `attributes`.

It is convenient to collect the inherited and synthesized attributes of a node and its production:

```
node  ::= NODE production attributes attributes
```

The attribute calculations of each production fall into a similar general pattern. The attribute calculations for a single production specify the calculation of the attributes of several nodes. They specify the calculation of the synthesized attributes of the l.h.s. symbol and the inherited attributes of each r.h.s. symbol. The calculations might require the synthesized or inherited attributes of any of the symbols in a production. The process of deciding which attributes to calculate depends on the production. Once the production is identified the attribution can be thought of as a set of (partial) mappings from attribute names to attribute values, one mapping per symbol in the production. All the attribute calculations are captured in one function:

```
attrfuns :: production -> node -> [node] -> (attributes,[attributes])
```

The function calculates the attribute values for any production in the grammar, hence the production argument to determine which production. It also takes all the attributes visible in the rule as two arguments—the attributes of the symbol on the left collected in a single `node` and the attributes of the symbols on the right in a list of `node`s. The value returned is a pair—a set of attributes that are the synthesized attribute values of the left side symbol and a list of sets of attributes that are the inherited attribute values for the right side symbols.

The `attrfuns` function is then used in a completely generic evaluator function:

```
synthesize prodn inh =
  synth prodn
  where
  synth (P root rhs) =
     syn
     where
     rsyns = map2 synthesize rhs rinhs
     children = map3 NODE rhs rinhs rsyns
     (syn,rinhs) = attrfuns prodn (NODE prodn inh syn) children
  synth (T any) =
     syn where (syn,rinhs) = attrfuns prodn (NODE prodn inh syn) []
```

Notice how the arguments passed to `attrfuns` depend on values returned by the function. This expresses the complex dependencies possible between the attributes. It is only possible to state this directly because Miranda is a lazy language. If the AG is undefined because it has circularities the evaluation will fail to terminate or the system will report the infamous "Black hole" message when an attempt is made to reduce the cyclic redex. The generic part of the attribute grammar is the data type `production` and the function `synthesize`. The specific attribute grammar is determined by the definitions of `terminals`, `nonterminals`, `attribute`, `attribute_type`, and `attrfuns`.

The attribute calculations can be written as one function with one case per production and one case per terminal. The general form of this function is shown if figure 5.5. The naming conventions used are:

$X_i tree$	a pattern that matches a tree for symbol X_i
$X_i?\_attribute_j$	the name of an attribute, a value of type `attribute` (? is either *synthesized* or *inherited*).
$e_{i,j}$	an expression calculating the value of the attribute named by $X_i?\_attribute_j$.
$X_i node$	a Miranda variable bound to the `node` holding the inherited and synthesized attributes for symbol X_i.

Within the expressions $e_{i,j}$ a synthesized attribute reference $X_k \uparrow a$ and an inherited attribute reference $X_k \downarrow a$ are written respectively as

$X_k node$ `$u` a and $X_k node$ `$d` a

The functions `u` and `d` are used as infix operators to recover the value of an attribute from the node. Note that a terminal naturally has no symbols on the right side, so the second list in the pair is alway empty. Finally, since the data type representing a parse tree has been 'opened out' and allows illegal trees the last clause of the function raises an error.

```
attrfuns (P X_0 [X_1 tree, X_2 tree, ..., X_n tree]) head body =
   ([(X_0 synthesized_attribute_1, e_{0,1}), (X_0 synthesized_attribute_2, e_{0,2}),...],
    [ [(X_1 inherited_attribute_1, e_{1,1}),...],
       ⋮
      [(X_i inherited_attribute_1, e_{i,1}), (X_i inherited_attribute_2, e_{i,2}),...],
       ⋮
      [(X_n inherited_attribute_1, e_{n,1}),...]])
   where
   [X_1 node, X_2 node, ..., X_n node] = body

attrfuns production_2 head body =
   ...
⋮
attrfuns (T terminal_i) head body =
   ([synthesized attributes of terminal_i],
    [])
⋮
attrfuns other head body = error "Illegal parse tree"
```

Figure 5.5: The general form of an attribute calculation function

The above description is very abstract and will probably make more sense when an example has been considered. The Miranda code for each production in the AG corresponds directly to the conventional AG definition. It is easier to write than the abstract description suggests. One of the productions in the binary arithmetic expressions AG is

$$\mathsf{int} \rightarrow \mathsf{int\ digit}$$
$$\mathsf{int2}{\downarrow}\mathsf{scale} = \mathsf{int}{\downarrow}\mathsf{scale}+1$$
$$\mathsf{digit}{\downarrow}\mathsf{scale} = \mathsf{int}{\downarrow}\mathsf{scale}$$
$$\mathsf{int}{\uparrow}\mathsf{val} = \mathsf{int2}{\uparrow}\mathsf{val} + \mathsf{digit}{\uparrow}\mathsf{val}$$

The corresponding `attrfuns` function case for this production is

```
attrfuns (P Int [P Int any, P Digit any']) head body =
    ([(Val, add (int2A $u Val) (digitA $d Val))]
     [
       [(Scale, add (head $d Scale) (N 1))],
       [(Scale, (head $d Scale))]
    ])
    where [int2A,digitA] = body
```

This is a direct translation of the original production and could be performed by a quite simple preprocessor.

The pattern matches the production $\mathsf{int} \rightarrow \mathsf{int\ digit}$. It looks slightly more complicated than this only because the match must inspect subtrees of the left hand side to determine their type. add is the same function as + except that it takes and returns values of the type of the tagged union of attribute types, it is defined as `add (N a) (N b) = N(a+b)`.

A critical feature of how the function case is written is that `body` is decomposed into the 2-list *after* the case has been chosen. This is essential. The temptation to move the pattern matching against `body` to the top line and dispose of the name `body` altogether must be resisted because it causes the program to fail. The pattern match against `body` cannot be evaluated until the length of `body` is known. The length is determined by the result of the calls to `map2` and `map3` in `synthesize` which in turn use the result calculated in *this* function case. Once `attrfuns` has committed to a choice, the structure of the result can be calculated (a pair of lists) and used to build sufficient of `body` to allow the pattern match. The demand driven nature of lazy evaluation allows us to write such a blatantly recursively-defined program, but great care must be taken to prevent a value being needed before it can possibly have been calculated. Strict adherence to the general scheme prevents this particular pitfall.

5.3 Modular decomposition

The attribute calculations have been removed from the mechanism of traversing the parse tree, and placed in a function called `attrfuns`. This function can be replaced by another to give a different attribute grammar based on the same context free grammar, or the same attribute grammar, calculating the same values, but expressed in a different way. In particular, the calculations may be expressed as the combined result of several functions. Each function is responsible for some aspect of the calculation and plays the rôle of a module. It is possible to organize the several functions so as to decompose the problem according to different parts of the syntax or according to different sets of attributes. I call these two axes of decomposition *syntactic modularity* and *semantic modularity* respectively.

It is convenient to name the type of a module. Because an attribution function plays the rôle of a module we declare the type of this function as the type abbreviation `module`.

```
module == production -> node -> [node] -> (attributes,[attributes])
attrfuns :: module
```

5.3.1 Syntactic modularity

A limited amount of syntactic modularity can be achieved with the original translation scheme because it required a separate function to be written for each nonterminal. The actions for a single nonterminal can be changed by replacing its function with another. These functions are not independent because they call the attribution functions for the symbols on the r.h.s. of the productions. It is more likely that an AG is written with a smaller number of modules in mind. This is because different syntactic things often belong together, either as subparts of a larger single concept or things which belong harmoniously together, e.g. the definition of objects and their use. The example grammar in figure 5.1 may have been conceived as two grammars—an expression grammar (productions P1…P7, P12 and P13) and a number grammar (P8…P11). The attribution function can modified to allow several similar functions to reflect this natural decomposition of the grammar.

Each module is formed by omitting the cases for the syntax belonging to the other module(s). In isolation, the modules are undefined for the syntax that has been removed. Instead of signalling an error when the syntax tree is not recognised, another module can be invoked to handle the unrecognized cases. This is done by writing the name of the default module as the last production in the attribute grammar module. If this default module is made a parameter of the module we are writing then no commitment to the structure of the whole language is made until the modules are composed:

module expressions(default)
  —productions P1,...,P7,P12,P13 appear here
  default

module numbers(default)
  —productions P8,...,P10 appear here
  default

module main = expressions(numbers(error))

The translation of this extension to Miranda is straightforward. Each module is a curried function, taking the default module as the first parameter. The module composition is function application:

```
expressions default =
    attrfuns where  attrfuns P1 head body = ...
                    attrfuns P2 head body = ...
                    ⋮
                    attrfuns = default
numbers default =
    attrfuns where  attrfuns P8 head body = ...
                    ⋮
                    attrfuns P11 head body = ...
                    attrfuns = default

attrfuns_error any head body = error "..."

attrfuns = expressions (numbers attrfuns_error)
```

The cases in the *where*-clause for the local definitions of `attrfuns` are identical to the corresponding cases of the original function. The last case has been written with no parameters; as Miranda is a curried language this is a correct and concise way of passing all three parameters on to the default.

5.3.2 Semantic modularity

In the preceding section I showed how an AG can be decomposed into syntactic modules. The basic approach was to devolve responsibility for the different parts of the grammar to different modules (functions). This is simple to do because the structure of the attribution function is a dispatch on the syntax.

Semantic decomposition devolves the calculation of different (sets of) attributes to different modules.

If the attributes are temporally ordered, i.e. there is a sequence of sets of attribute names such that all attributes of one name set may be evaluated before any of the next name set, then the semantic modularity expresses a phase oriented decomposition. Complications arise because all the attributes must be available to the functions for calculating other attributes. If the decomposition was guaranteed to be phase oriented then there is a simple implementation. Each set of attributes is calculated in order as a function of the previous set. Often, and because of the conveniences that we shall discover later, this is not possible. In practical terms it is necessary to collect and combine the the attributes calculated by each module function.

Each module computes some subset of the attributes. This is easily accomplished—the function simply omits the calculation of those attribute-value pairs that are not its responsibility. Often an attribute will be local to some subset of the productions. In the example AG (figure 5.1) the attribute scale only appears in some factor, int and digit productions. In these cases it is not necessary to specify each case for the other productions. The cases that contribute nothing are simply omitted when writing a module. Each module computes only those attributes that are important to it and for only those productions that define the attribute. A module that defines the scale attribute is defined by leaving out everything else:

module scale
  factor $\longrightarrow$ int
    int$\downarrow$scale = 0
  int $\longrightarrow$ digit
    digit$\downarrow$scale = int$\downarrow$scale
  int $\longrightarrow$ int digit
    int2$\downarrow$scale = int$\downarrow$scale+1
    digit$\downarrow$scale = int$\downarrow$scale

Modules communicate via the attributes that they define and the attributes that they use. For example, a typechecking module might assign a value to a type attribute for every expression and subexpression. This attribute could then be used by a code generation module to select the appropriate instructions. Thus the type attribute is an export of the type checking module and an import to the code generation module.[2]

The attributes calculated by the modules need to be combined. Each module computes for each node a partial set of attributes, i.e. a partial mapping from attribute names to attribute values. All the module contributions are collected to get a complete set of attributes for a node. At first it might seem that taking the union is a suitable way of combining the partial mappings. The function override operator is more suitable because it is more flexible. The function override operator $\oplus$ is defined by

$$f \oplus g = \{x \mapsto y \mid x \in dom\, f \cup dom\, g;\ y = \textbf{if}\ x \in dom\, g\ \textbf{then}\ g(x)\ \textbf{else}\ f(x)\}$$

When the function domains are disjoint both union and function override produce the same result. When the domains overlap function override defines a definite value, the

[2]It may indeed be possible to enforce this in the Miranda translation by declaring the module type to be polymorphic in the attribute names calculated, i.e. something like

```
module * == production -> node -> [node] ->
([(*,attribute_type)],[[(*,attribute_type)]])
```

Then the modules could not contain 'free' attribute names but only those provided by some form of parameterization.

value defined by the latter function g. This can be used to good effect in both modifying the behaviour of another module and in defining general patterns of attribute calculation that can be refined by other modules.

Each module might calculate some synthesized attributes for the left side symbol of the node and some inherited attributes for each of the right side symbols. To combine the results of two modules each partial map produced by one module is combined with the corresponding partial map produced by the other. The `comb` operator does this. It takes two attribution functions and returns a new attribution function that computes the combined results. The identity value for `comb` is the attribution function `nointerest` that specifies no attribute values. It is both a left- and right-identity. It is the default case for each module, and returns an empty partial map for the left hand side symbol and each right hand side symbol. These two functions complete the general part of the modular AG evaluator.

```
comb :: module -> module -> module
comb f1 f2 prodn head body =
     (s1 $oplus s2, map2 oplus i1 i2)
     where
     (s1,i1) = f1 prodn head body
     (s2,i2) = f2 prodn head body

nointerest :: module
nointerest (T any)      head body = ([], [])
nointerest (P any rhs)  head body = ([], map (const []) rhs)
```

5.3.3 Modular programming

The combining form for modules allows good software engineering practices. If the modules compute different attributes then they do not interfere. The coupling between modules is restricted to two forms of interaction. A module may use values computed by other modules. As described above, the attribute names form a narrow mutual interface. A less constraining form of coupling occurs when two modules calculate the same attributes. If the syntax defined for the modules is disjoint then there is no interaction.

If, on the other hand, two modules define the same attribute for some of the same syntax then the latest module (rightmost of the combined modules) takes precedence. The latest module redefines some of the actions of the previous modules. This can be viewed as a process of explicit *refinement* where each module defines a new level of refinement, or iterative refinement where each iteration is formed by adding new (re-)definitions rather then rewriting the original program, so keeping a history of the refinement process.

Another view of the module precedence is that it is a form of object oriented programming. Early modules correspond to less specialized classes. Later modules refine some of the previous classes. The next section shows how to make general statements like 'all statements do such and such'. A later module might refine this by stating 'assignment statements do this differently'. The module combination method shares with object oriented programming the essential property that common behavior shared between different objects can be defined once. These points taken further in section 5.14.

5.4 Generalized modules

A classical attribute grammar production applies only to one grammar production and calculates a fixed set of attribute values. I have introduced modules that allow both syntax and attributes to be factored out but the programmer still needs a complete knowledge of the syntax to construct a complete attribute grammar. I investigate the benefits obtained by replacing parts of a typical attribute grammar production with a general term. An expected benefit is that of *reuse*. The general form is shorter than its equivalent AG, and often more understandable. If something has to be stated many times there is always the possibility that one instance may differ from the others. This might be intentional, in which case it is difficult to spot the deviation from the general case. If the difference is unintentional then the bug is hard to find amongst all of the similar looking cases.

There is a lot of scope for generalization of an typical attribute grammar definition. Take for example the following definition for a comma-separated expression list. The environment (symbol table) is passed down as an inherited attribute. An expression list synthesizes a code attribute which is a list of the code attributes synthesized by the individual expressions[3]. Thus the expression list collects code from the expressions.

module expression_list
  exprlist → expr "," exprlist
    expr↓env = exprlist1↓env
    exprlist2↓env = exprlist1↓env
    exprlist1↑code = [expr↑code] ++ exprlist2↑code

  exprlist → expr
    expr↓env = exprlist↓env
    exprlist↑code = [expr↑code]

Generalization is done by concentrating on some part of the definition and making that part less restrictive. Any name in this definition, like exprlist or env, can be replaced by a new abstract name. Abstract names are written starting with upper-case letters, or in italics, to distinguish them from the existing concrete names which are written entirely in lower-case. How the new name is treated determines the type of generalization. I distinguish three types of generalization: widening, abstraction and quantification. The definition can be left with the new name *free*. This is called *widening*. Anything may be substituted for the new name, so widening is used to generalize pattern matching.

The definition may be formalized by declaring the new names to be formal parameters to the new definition. This is called *abstraction* or parameterization. For example expr and exprlist can be replaced throughout by X and Xlist respectively. The notation for abstraction is the placing of the formal parameters in parentheses following the module name. A concrete module is obtained by substituting back concrete names for the parameters (e.g. expr and exprlist) and eliminating the formals. This is called instantiating the abstraction.

An abstraction can be used to from a *quantification* by instantiating it with all suitable names.[4]

[3] I use the Miranda syntax for list expressions: lists of items are written commas between square brackets and separated by commas; [] is the empty list and ++ (or ++) is the list append operator.

[4] The difference between widening and quantification is that with quantification the elements are

In the following sections I abstract various parts of the grammar definition to determine what notational conveniences can be obtained by doing so. The abstractions that are investigated are ones which are feasible in a programming language. This is partly the result of developing these ideas in concert with writing the functional attribute grammar processor which I describe later.

5.5 Generalization by production patterns

A classical attribute grammar specifies a set of attribute calculations for each production in the underlying context free grammar. The production itself may be widened by replacing it with a pattern that matches several productions in the CFG. Taking the previous example productions and their attribution rules and uniformly replacing 'expr' by 'X' and 'exprlist' by 'Xlist' yields

module any_list_code
  Xlist → X "," Xlist
    X↓env = Xlist1↓env
    Xlist2↓env = Xlist1↓env
    Xlist1↑code = [X↑code] ++ Xlist2↑code
  Xlist → X
    X↓env = Xlist↓env
    Xlist↑code = [X↑code]

The production pattern will match any production with a consistent substitution of grammar symbol names for X and Xlist. Typically this will include other lists, for example, formal parameter lists or variable lists in a Pascal variable declaration.

Production patterns can be formed by means other than replacing a terminal or nonterminal with a free name. Dueck & Cormack use an ellipsis notation ("...") to indicate zero or more symbols on the r.h.s. of a production. The pattern "A → ... B" is an example of this notation and identifies B as the rightmost r.h.s. symbol of any production with at least one r.h.s. symbol.

I also use a notation that treats the r.h.s. as a sequence of symbols. The symbols are indexed by subscripts, and may appear indexed in the attribution equations. For example, it might be that some aspect of code generation depends on a compiler option. The compiler options need to be transported to the parts of the parse tree that use them. This can be simply stated in one general rule:

$\mathsf{A} \to \mathsf{B}_1\ \mathsf{B}_2\ \ldots\ \mathsf{B}_n$
  $\mathsf{B}_i{\downarrow}\mathsf{options} = \mathsf{A}{\downarrow}\mathsf{options},\ \forall\, i \in \{1, 2, \ldots, n\}$

Each r.h.s. symbol inherits the options attribute. The attribute calculation is replicated for each i. The special case of the empty production "A →" is handled as expected—there are no r.h.s. symbols so no attribute calculations are generated.

Production patterns are particularly important, and are a key contribution of Dueck & Cormack. They allow one production pattern, or a small fixed number, to stand for an arbitrarily large set of other productions. The options example is a single production pattern that applies to any language. The mechanism of module reuse is that the module is 'duplicated' to apply to every matching rule in the grammar.

blindly substituted. Widening requires that the substituted items are legal and this can only be used for testing whereas quantification is generative.

5.6 Generalization by abstraction

Any part of a definition may be abstracted. The abstraction describes a general property which is made to apply to a specific instance by instantiation. Instantiation is requested by the programmer, so reuse of abstract modules is controlled by the programmer. Attributes may be abstracted. This allows us to describe patterns of attribute calculation without having to fix the names of the attributes involved. Production patterns may be abstracted, allowing common definitions to be selectively applied to similar pieces of syntax. Finally, the functions used in the attribute calculations may be abstracted to provide higher order calculations analogous to mapping.

As a demonstration of abstraction I apply abstraction to the code list module by naming the abstracted parts. The non-abstract module has the disadvantage that the unit production case is too applicable. It would apply to unwanted productions like expr → term as well as the desired productions like exprlist → expr. By abstraction we obtain more control over the module although we have to instantiate it, for example for exprs and defns. The two instantiated modules are combined with the module override operator '⊕'[5].

```
module code_list(X,Xlist)
   Xlist → X "," Xlist
      X↓env = Xlist1↓env
      Xlist2↓env = Xlist1↓env
      Xlist1↑code = [X↑code] ++ Xlist2↑code
   Xlist → X
      X↓env = Xlist↓env
      Xlist↑code = [X↑code]

module desired_lists = code_list(expr,exprlist) ⊕ code_list(defn,defnlist)
```

5.6.1 Pervasive inheritance

We have already seen an example of pervasive inheritance in the compiler options example. The attribute name can be generalized to provide pervasive inheritance for any particular attribute. The module *pervasive_inheritance* is parameterized by an attribute name. The instantiation "pervasive_inheritance(options)" yields the compiler options example:

module pervasive_inheritance(in)

  $\mathsf{A} \rightarrow \mathsf{B}_1\ \mathsf{B}_2\ \ldots\ \mathsf{B}_n$

    $\mathsf{B}_i{\downarrow}\mathsf{in} = \mathsf{A}{\downarrow}\mathsf{in},\ \forall\, i \in \{1, 2, \ldots, n\}$

Note that the equation is replicated for each symbol on the r.h.s.

5.6.2 Bucket brigade

Some languages require the derivation tree to be processed in a left-to-right traversal. This is the case in languages like Pascal in which named things may only be used after their definition in the program text. A left-to-right traversal of the derivation tree is equivalent to a one-pass traversal of the source. An attribute grammar with this

[5] I use ⊕ as both a function override operator and a module override operator. The correct operator is apparent by context.

property is called a regular right part attribute grammar. The bucket brigade module describes this attribute dependency.

Two attributes are used: one (in) to carry the information in a downwards direction and a second one (out) to carry it up the tree:

$$
\begin{array}{l}
\textbf{module}\ \mathsf{bucket\_brigade(in,out)} \\
\quad \mathsf{A} \rightarrow \epsilon \\
\qquad \mathsf{A}\!\uparrow\!\mathsf{out} = \mathsf{A}\!\downarrow\!\mathsf{in} \\
\quad \mathsf{A} \rightarrow \mathsf{B_1}\ \mathsf{B_2}\ \ldots\ \mathsf{B_n},\ n > 0 \\
\qquad \mathsf{B_1}\!\downarrow\!\mathsf{in} = \mathsf{A}\!\downarrow\!\mathsf{in} \\
\qquad \mathsf{B_i}\!\downarrow\!\mathsf{in} = \mathsf{B_{i\text{-}1}}\!\uparrow\!\mathsf{out},\ \forall\, i \in \{2, \ldots, n\} \\
\qquad \mathsf{A}\!\uparrow\!\mathsf{out} = \mathsf{B_n}\!\uparrow\!\mathsf{out}
\end{array}
$$

At each level in the derivation tree the value passed in from the left and is chained along the right part of the production, passing through each subtree in turn. Then this is passed out to the right. In the case of null productions (and of terminals) the value is passed straight out again because there is no right side.

To use this module some additional work is needed. By itself the bucket brigade just threads the value all through the tree without computing anything useful. First, the module should be instantiated with the desired attribute names. Second, the inherited attribute *in* must be defined for the axiom of the grammar. Finally, some productions will do useful computation and alter the value of the attribute. As a simple example the module get_vars computes a set of all the variable names used in a program.

$$
\textbf{module}\ \mathsf{get\_vars} = \mathsf{bucket\_brigade(set\_in,\ set\_out)} \oplus \mathsf{get\_vars\_detail}
$$

$$
\begin{array}{l}
\textbf{module}\ \mathsf{get\_vars\_detail} \\
\quad \mathsf{axiom} \rightarrow \mathsf{Prog} \\
\qquad \mathsf{Prog}\!\downarrow\!\mathsf{set\_in} = \emptyset \\
\quad \mathsf{variable} \rightarrow \mathsf{ident} \\
\qquad \mathsf{variable}\!\uparrow\!\mathsf{set\_out} = \mathsf{variable}\!\downarrow\!\mathsf{set\_in} \cup \{\mathsf{ident}\!\uparrow\!\mathsf{name}\}
\end{array}
$$

At the top level the set is initialized to the empty set. The set is passed all through the tree by the bucket brigade, and everywhere a variable appears it is added to the set. Retrieving set_out at the top level gives the set of all the variables used in the program.

5.6.3 Harvest & Sow

The bucket brigade processes things in left to right order. This is ideal for languages like Pascal in which items are defined before their use. An alternative supported by many languages is that all the items in a scope are mutually visible. This is easily specified by the harvest_and_sow module. Definitions are 'harvested' from a scope and then incorporated in the environment.

$$
\begin{array}{l}
\textbf{module}\ \mathsf{harvest(out)} \\
\quad \mathsf{A} \rightarrow \mathsf{B_1}\ \mathsf{B_2}\ \ldots\ \mathsf{B_n} \\
\qquad \mathsf{A}\!\uparrow\!\mathsf{out} = \bigcup_{\mathsf{i} \in \{1 \ldots \mathsf{n}\}} \mathsf{B_i}\!\uparrow\!\mathsf{out} \\
\quad \mathsf{A} \\
\qquad \mathsf{A}\!\uparrow\!\mathsf{out} = \emptyset
\end{array}
$$

$$
\textbf{module}\ \mathsf{harvest\_and\_sow(in,\ out)} = \mathsf{harvest(out)} \oplus \mathsf{pervasive\_inheritance(in)}
$$

As with the bucket brigade, we need to interface this behaviour to the defining and scoping constructs of the language. Assume there are two syntactic sorts definition and scope.

module environment = harvest_and_sow(env, defs) $\oplus$ scopes $\oplus$ definitions

module definitions
defn $\rightarrow$ "define" ident "=" thing
defn$\uparrow$defs = {ident$\mapsto$thing}

module scopes
scope $\rightarrow$ "begin" things "end"
things$\downarrow$env = scope$\downarrow$env $\oplus$ things$\uparrow$defs
scope$\uparrow$defs = scope$\downarrow$env

A definition simply makes a tiny environment containing itself. Multiple definitions are collected by the harvest module. At a scope boundary two things happen: the definitions inside the scope are added to the definitions from outside the scope and the definitions inside the scope are made invisible to those outside. Because they are used in conjunction with the harvest-and-sow module all that the definition and scope modules have to do is deal with the specific constructs that affect these constructs. The definition is resilient to changes in other parts of the language, for example, expression syntax.

The harvest-and-sow module is similar to the bucket brigade because they both provide for passing a variable through a tree. The advantage of the harvest-and-sow approach is that the individual definitions only affect the out attribute. The bucket brigade is more fundamental as it is possible to code the harvest-and-sow using the bucket brigade but not the other way round. In the example the bucket brigade would be used to collect definitions from each scope, using the in and out attributes to do the job of the defs attributes. In this sense the bucket brigade is an incremental version of the harvest-and-sow paradigm, and is potentially more efficient as subsets of the definitions are always added one-by-one.

5.7 Generalization by quantification

Consider a grammar with a hierarchy of expression nonterminals that are there purely to enforce the syntactic operator precedence, something like the precedence hierarchy defined in figure 4.4. An expression "5" might have the derivation tree

expr $\rightarrow$ term $\rightarrow$ factor $\rightarrow$ number $\rightarrow$ 5

Most of the unit productions carry no semantic information. One of the rules for a term is written:

term $\rightarrow$ factor
factor$\downarrow$env = term$\downarrow$env
factor$\downarrow$type = term$\downarrow$type
term$\uparrow$code = factor$\uparrow$code
⋮

There is a *copy rule* for each attribute. Traditionally, these copy rules are automatically inserted by the attribute grammar processor [Farrow 1982].

The collection of inherited attributes of a symbol is called the *context* of the symbol. The context is a mapping from the attribute names to their values. I use the notation $\mathsf{X}\!\downarrow$ to stand for the context of X, and the membership predicate '$\mathit{a} \in \mathsf{X}\!\downarrow$' to test if the context is defined for the attribute name a. This is shorthand for $\mathit{a} \in \mathsf{domain}(\mathsf{X}\!\downarrow)$. $\mathit{a} \notin \mathsf{X}\!\downarrow$ implies that $\mathsf{X}\!\downarrow\!\mathit{a}$ is an error. Analogously, the collection of synthesized attributes of a symbol is called the *syntext* of the symbol. The term "syntext" is used with this meaning by Nord & Pfenning [1988].An alternative to writing out all the copy rules is to use quantification:

$$\mathsf{factor}\!\downarrow\!\mathit{attr} = \mathsf{term}\!\downarrow\!\mathit{attr},\ \forall\, \mathit{attr} \in \mathsf{term}\!\downarrow$$
$$\mathsf{term}\!\uparrow\!\mathit{attr} = \mathsf{factor}\!\uparrow\!\mathit{attr},\ \forall\, \mathit{attr} \in \mathsf{term}\!\downarrow$$

In cases like this where a symbol's entire context or syntext is defined to be the same as other symbol's context or syntext it is convenient to omit the quantification, and directly say that one symbol's context (or syntext) is the same as another's:

$$\begin{array}{l} \mathsf{term} \rightarrow \mathsf{factor} \\ \quad \mathsf{factor}\!\downarrow = \mathsf{term}\!\downarrow \\ \quad \mathsf{term}\!\uparrow = \mathsf{factor}\!\uparrow \end{array}$$

5.7.1 Unit production copy rule

Production patterns allow explicit copy rules like the above example to be generalized further. A useful case is the unit production copy rule. This rule applies to all unit productions. Most unit productions have no semantic meaning, which makes this module a useful reusable component.

$$\begin{array}{l} \textbf{module}\ \mathsf{unit\_production\_copy\_rule} \\ \quad \mathsf{A} \rightarrow \mathsf{B} \\ \qquad \mathsf{B}\!\downarrow = \mathsf{A}\!\downarrow \\ \qquad \mathsf{A}\!\uparrow = \mathsf{B}\!\uparrow \end{array}$$

A similar situation occurs with bracketing constructs. Bracketing constructs occur in almost all languages, and are usually there to allow the programmer to group things together or to make the program clearer by emphasizing existing groupings. Examples are parentheses in expressions and begin-end blocks in some imperative languages. Bracketing may alter how the input is parsed but it has little impact on the semantics of the language. A general definition for bracketed illustrates abstraction, production patterns and quantification. The definition may be instantiated with the terminals "(" and ")", which will make parentheses 'invisible' to the semantic analysis throughout the language.

$$\begin{array}{l} \textbf{module}\ \mathsf{ingore\_brackets(Left,\ Right)} \\ \quad \mathsf{A} \rightarrow \mathsf{Left\ B\ Right} \\ \qquad \mathsf{B}\!\downarrow = \mathsf{A}\!\downarrow \\ \qquad \mathsf{A}\!\uparrow = \mathsf{B}\!\uparrow \end{array}$$

5.7.2 List abstractions

Many notations contain lists and sequences of things. To handle these in a concise and efficient manner it is necessary to abstract out the complexity of the individual list forming productions. Special list facilities are available in many metaprogramming systems (e.g. the Ergo attribute system, CAML and SDF). This is not necessary in modular attribute grammars, although it is still useful for compact syntax description. Syntactic lists are isomorphic to the list data structure provided by or easily constructed in many programming languages. The following sections show how modular attribute grammars can be used to exploit this by creating an attribute view of the list derivation tree that is a list data structure. The conventional programming operations of mapping and filtering can then be used on the list data structures. List abstractions insulate the programmer from the particular concrete list syntax of the subject language.

5.7.2.1 1-lists

The following module defines the attribute calculations for a comma-separated list of one or more items. It is another generalization of the expression list defined at the beginning of this chapter. The module generalizes both the syntax and the attributes.

module one_list(List,Item)
  List $\rightarrow$ Item
    List$\uparrow$*attr* = [Item$\uparrow$*attr*], $\forall$ *attr* $\in$ Item$\uparrow$
  List $\rightarrow$ Item "," List
    List1$\uparrow$*attr* = [Item$\uparrow$*attr*] ++ List2$\uparrow$*attr*, $\forall$ *attr* $\in$ Item$\uparrow$

Like the first generalization, this one produces a list of values for the attributes of the list items. Unlike the first example, a list is produced for *every* synthesized attribute of the items. This is specified by quantifying the attributes. The two list productions are called the base case and the inductive case. Each attribute of the base case has as a value the singleton list containing the value of the corresponding attribute of the item. The inductive case prepends the new item attribute value on the front of the list.

5.7.2.2 0-lists

A list of zero or more items may be empty. Usually the empty list case has little semantic import. As an example take a list where there is no separator between the list items. To the programmer the attributes look the same as the previous type of list.

module zero_list(List,Item)
  List $\rightarrow$
    List$\uparrow$ = [], $\forall$ *attr* $\in$ *synthesized attributes*
  List $\rightarrow$ Item List
    List1$\uparrow$*attr* = [Item$\uparrow$*attr*] ++ List2$\uparrow$*attr*, $\forall$ *attr* $\in$ Item$\uparrow$

The base case synthesizes every possible attribute.

5.7.2.3 Left-associative lists

The above lists are right associative in that the derivation tree collects items into a list from the right. The derivation tree is skewed to the right. Left-associative lists produce derivation trees that are skewed to the left. A left-associative list has the list and item parts the other way round in the inductive production. The base production is the same:

$$\begin{array}{l} \mathsf{List} \rightarrow \mathsf{Item} \\ \qquad \mathsf{List}{\uparrow}\mathit{attr} = [\mathsf{Item}{\uparrow}\mathit{attr}],\ \forall\, \mathit{attr} \in \mathsf{Item}{\uparrow} \\ \mathsf{List} \rightarrow \mathsf{List}\ \text{“,”}\ \mathsf{Item} \\ \qquad \mathsf{List1}{\uparrow}\mathit{attr} = \mathsf{List2}{\uparrow}\mathit{attr} \mathbin{+\!\!+} [\mathsf{Item}{\uparrow}\mathit{attr}],\ \forall\, \mathit{attr} \in \mathsf{Item}{\uparrow} \end{array}$$

The only difference in the attribution rules is that the arguments to the list append operator "$+\!\!+$" are reversed.

5.7.2.4 Discussion

The list examples raises some interesting issues, not all of which are convenient.

It is not possible to associate a single type with an attribute name. If an attribute of Item has the type α then the same attribute of List has type $list(\alpha)$. If the attribute grammar without a list module allows a type to be assigned to each (symbol,attribute) pair then the list module preserves this property with the stated relationship.

I have presented 1-lists and 0-lists which both synthesize attributes which are lists in the underlying attribution language. Not all systems take this approach. Notably CAML [Cousineau & Huet 1989] distinguishes 0-lists and 1-lists. 0-lists are given a list type but 1-lists are given a type $t \times [t]$ where t is the type of the individual elements. This is sensible from a type checking point of view as the constructed value is always a member of the type and all members of the type could have been constructed from a parsed input. This compares with my approach where the empty list has the correct type for a 1-list. However, I believe that the product type for 1-lists is inconvenient to use as the programmer will nearly always want to treat all the items in the list the same way.

Empty lists have the problem that they synthesize a large number of attributes that have the value of the empty list data structure. A non-empty list might synthesize fewer attributes than the empty list. This leaves us with two different kinds of 'undefinedness' for an attribute—one for empty lists which synthesize an empty list data structure and another for non-empty lists which do not synthesize that attribute at all. The predicate "$\mathsf{List}{\uparrow}\mathsf{attr} = []$" is different from "$\mathsf{attr} \notin \mathsf{List}{\uparrow}$".

A partial solution to this problem is to parameterize the list module by an attribute or a set of attributes that should be synthesized. Notations for manipulating syntactically bound structures as convenient data structures might benefit from the work of Wadler. Manipulation of data types via an isomorphism is discussed in the work on views [Wadler 1987]. Monad comprehensions [Wadler 1990], which are like Miranda's list comprehensions but apply to any data type with a 'cons'-like and 'nil'-like constructors, might be a viable alternative notation for programming list-like attributes without recourse to actual list data structures.

To summarize, universal quantification of attributes is useful for describing default copy rules and indicating which language constructs have no semantic importance. It is also useful for imposing a more convenient data structure view on parts of the derivation tree. Traditionally, attribute grammar processors inserted copy rules at some of the points mentioned above. The ability to state such rules concisely means that automatic copy rule insertion is not really needed.

5.8 Higher order operations

Conventional attribute grammars have two non-interchangeable concepts: The derivation tree and the attributes. It is not possible to define an attribute by part of the structure tree or vice-versa. Vogt *et al.* define higher order attribute grammars (HAGs) by extending attribute grammars with new operations that permit these operations [Vogt *et al.* 1989]. The derivation tree has a type and may be manipulated like any other data type in the attribution language. A convenient linear form somewhat like Prolog terms is used for writing tree expressions. The two new capabilities are

- attribute := derivation tree
- nonterminal := tree valued expression

The first is the simplest. The derivation tree is available for direct inspection. For example, new trees may be constructed using parts of the current tree. The second allows the parse tree to be modified. A nonterminal in the CFG may be specially marked as a nonterminal attribute (NTA) with an overbar. An NTA may be assigned to. During parsing the NTA is treated as if it has a single null production, i.e. it does not affect the language accepted. In attribute evaluation the nonterminal is defined as the value of some expression. The NTA has attributes like any other nonterminal in the derivation tree. Attributes of the NTA may be used as normal in the attribute evaluations for the production. There are constraints on the dependencies, for example, a synthesized attribute of a NTA cannot be used to decide what tree to assign to that nonterminal. Vogt *et al.* use this mechanism to write multi-phase compilers. Each phase calculates a new (i.e. transformed) version of the program which is then processed within with the attribute grammar framework.

The explicit manipulation of contexts and syntexts provide a similarly powerful notation. The context and syntext of a derivation tree node can be diverted to pass through another, computed, tree. In previous chapters I have shown how syntax constructors and syntax patterns offer a convenient way of specifying parse trees, and I use this notation in the following description. A production pattern can be seen as a shorthand for the special case of a flat (i.e. unnested) syntax pattern. The following are equivalent:

term → factor	term:⟦⟨factor⟩⟧
List → List "," Item	List:⟦⟨List⟩ , ⟨Item⟩⟧
no flat equivalent	term:⟦⟨term⟩+⟨factor⟩-⟨factor⟩⟧

The second case has nonterminals in the syntax pattern which are variables. In general this would require parsing the syntax pattern text at run time for every match, and might cause syntax errors at attribute evaluation time. Restricting cases with variable nonterminals to flat patterns alleviates this problem. This is a reasonable restriction as patterns with variable nonterminals are meta-statements about the grammar and should have a meaning independent of the particular grammar.

In addition to syntax constructors and syntax patterns, this next example illustrates *local attributes*. Local attributes are attributes that are local to a set of attribute calculations but not attached to any part of the derivation tree. They are not part of any symbol's context or syntext so they are not visible as the inherited or synthesized attribute of a symbol in some other production, and not visible in any other module.

They are used to name sub-computations, for notational convenience and to prevent repeated computation. The simple transformation rule for the *me too* empty set notation may be written

$$
\begin{aligned}
&\mathsf{factor}{:}[\![\texttt{\{\}}]\!] \\
&\quad \mathsf{trans} = \mathsf{factor}{:}[\![\texttt{emptyset}]\!] \\
&\quad \mathsf{trans}{\downarrow} = \mathsf{factor}{\downarrow} \\
&\quad \mathsf{factor}{\uparrow} = \mathsf{trans}{\uparrow}
\end{aligned}
$$

The local attribute *trans* is calculated as a new derivation tree which is to replace the original from. This tree is effectively spliced into the original tree by diverting the context of the l.h.s. symbol into the new tree and the syntext of the new tree back as the syntext of the original production. In simple cases like this where the context and syntext are copied directly the whole transformation can be abbreviated to the almost notationally optimal form of overwriting the l.h.s. with the transformed tree:

$$
\begin{aligned}
&\mathsf{factor}{:}[\![\texttt{\{\}}]\!] \\
&\quad \mathsf{factor} = [\![\texttt{emptyset}]\!]
\end{aligned}
$$

The l.h.s. is not actually overwritten. This form is merely a convenient abbreviation of the previous version.

Sometimes it is not sufficient to copy the context or syntext unaltered. The attributes might have to be 'conditioned' for the new tree. One situation where this occurs is where a counter is used to generate new names. Each time a new name is required it is generated from the value of the counter and the counter is incremented. (The counter is called gensym after the Lisp function of the same name which yields a new unique symbol each time that it is called.) Take the problem of replacing the wildcard pattern of ML, Prolog and Haskell with a unique name. Each time the wildcard pattern ("_") is encountered it is replaced by a freshly generated name. The value of the gensym attribute is incremented for the processing of the transformed pattern. This may be written in two ways. The first uses quantification to copy all the attributes except gensym which is handled specially.

$$
\begin{aligned}
&\mathsf{pattern}{:}[\![\_]\!] \\
&\quad \mathsf{new} = \mathsf{pattern}{:}[\![\langle\mathsf{ident\ name}\rangle]\!] \\
&\quad \mathsf{name} = \text{“\_G”} +\!\!\!+ \mathsf{number\_to\_string}(\mathsf{pattern}{\downarrow}\mathsf{gensym}) \\
&\quad \mathsf{new}{\downarrow}\mathit{attr} = \mathsf{pattern}{\downarrow}\mathit{attr},\ \forall\, \mathit{attr} \in \mathsf{pattern}{\downarrow} - \{\mathsf{gensym}\} \\
&\quad \mathsf{new}{\downarrow}\mathsf{gensym} = \mathsf{pattern}{\downarrow}\mathsf{gensym}{+}1 \\
&\quad \mathsf{pattern}{\uparrow} = \mathsf{new}{\uparrow}
\end{aligned}
$$

Alternatively, the context can be seen as a function and overridden with the exceptional case. This is a more convenient notation.

$$\mathsf{new}{\downarrow} = \mathsf{pattern}{\downarrow} \oplus \{\mathsf{gensym} \mapsto \mathsf{pattern}{\downarrow}\mathsf{gensym}{+}1\}$$

Here the context was pre-processed for the translation of the transformed tree. Naturally, the syntext may be post-processed.

Higher order attribute grammars have more scope for circular dependencies. To avoid circularities the replacement tree must be calculated without reference to attributes which depend on the syntext of the replacement.

If the rewritten tree is a local attribute then the transformations must be specified as the last module, otherwise the transformations will be invisible to other modules. A safer alternative is to make the transformed, attributed tree an attribute of the l.h.s. Then it would not be local a single production in one module. In this way the programmer is forced to make a decision about which tree the attribute is taken from, rather than leaving this up to the module composition. Where Vogt et al write

$$\begin{array}{l} \mathsf{X} \rightarrow \mathsf{Tree}\ \overline{\mathsf{NewTree}} \\ \quad \mathsf{X}\!\uparrow\!\mathsf{a} = \mathsf{NewTree}\!\uparrow\!\mathsf{b} \\ \quad \mathsf{NewTree} = \mathsf{f}(\ldots) \end{array}$$

we write

$$\begin{array}{l} \mathsf{X} \rightarrow \mathsf{Tree} \\ \quad \mathsf{X}\!\uparrow\!\mathsf{a} = \mathsf{X}\!\uparrow\!\mathsf{NewTree}\!\uparrow\!\mathsf{b} \\ \quad \mathsf{X}\!\uparrow\!\mathsf{NewTree} = \mathsf{f}(\ldots) \end{array}$$

5.9 A puzzle

The use of transformations with separate modules can give some surprising results. One particularly surprising thing is that subsequent modules override the action of the transformation, and see the untransformed tree. For this reason I suggest transformation should be the purpose of an attribute grammar rather than just one of several steps. It is always possible to reprocess the transformed program later in a multi-phase application. The rest of this section describes this phenomenon in detail by posing a puzzle and explaining the answer. It works through the module composition operations in detail.

The grammar used is one for 'extended' peano numbers. Peano numbers are enumerated by repeatedly taking the successor of zero. The numbers are `0`, `s0`, `ss0`, ... This algebra is extended with a 'double successor' operator which is equivalent to two successor operators, i.e. `d`=`ss`. The grammar:

$$\begin{array}{l} \mathsf{number} \rightarrow \mathsf{num} \\ \mathsf{num} \rightarrow \text{"}\mathtt{s}\text{"}\ \mathsf{num} \\ \mathsf{num} \rightarrow \text{"}\mathtt{d}\text{"}\ \mathsf{num} \\ \mathsf{num} \rightarrow \text{"}\mathtt{0}\text{"} \end{array}$$

The equivalence of the "`s`" and "`d`" forms is defined by a transformation module which replaces the d-form:

$$\begin{array}{l} \textbf{module}\ \mathsf{transform} \\ \quad \mathsf{num}{:}[\![\mathsf{d}\langle\mathsf{num}\rangle]\!] \\ \qquad \mathsf{num} = [\![\mathsf{ss}\langle\mathsf{num}\rangle]\!] \end{array} \qquad \text{or equivalently} \qquad \begin{array}{l} \textbf{module}\ \mathsf{transform} \\ \quad \mathsf{num}{:}[\![\mathsf{d}\langle\mathsf{num}\rangle]\!] \\ \qquad \mathsf{new} = \mathsf{num}{:}[\![\mathsf{ss}\langle\mathsf{num}\rangle]\!] \\ \qquad \mathsf{new}\!\downarrow\ = \mathsf{num1}\!\downarrow \\ \qquad \mathsf{num1}\!\uparrow\ = \mathsf{new}\!\uparrow \end{array}$$

To observe the transformation we calculate a value that depends directly on the shape of the tree. A simple function of the tree is the number of nodes in the tree, which is calculated by summing the sizes of the subtrees of a node and adding one. Terminals count as one node. The count module is parameterized because we will use it twice:

$$\begin{array}{l} \textbf{module}\ \mathsf{count(sum)} \\ \quad \mathsf{A} \rightarrow \mathsf{B_1}\ \mathsf{B_2}\ \ldots\ \mathsf{B_n} \\ \qquad \mathsf{A}\!\uparrow\!\mathsf{sum} = 1 + \mathsf{B_1} + \mathsf{B_2} + \cdots + \mathsf{B_n} \\ \quad \mathsf{A} \\ \qquad \mathsf{A}\!\uparrow\!\mathsf{sum} = 1 \end{array}$$

```
number
  |
 num₁              new:  num₃
 / \                     / \
d  num₂                s₁  num₄
    |                      / \
    0                    s₂  num₅
                              |
                              0
```

Figure 5.6: Parse trees

The main module is:

$$\textbf{module}\ \mathsf{main} = \mathsf{count(pre)} \oplus \mathsf{transformation} \oplus \mathsf{count(post)}$$

Now that the grammar has been completely described, we ask the question

> What does *main* calculate for the input "`d0`"?

The answer is that it calculates pre=7 and post=5, because count(pre) counts the number of nodes in the transformed tree and count(post) counts the number of nodes in the original tree. It might be thought of as surprising that the earlier module sees the larger transformed tree and the latter module doesn't.

The parse tree for the input "`d0`" and the transformed part "`ss0`" are shown in figure 5.6. Each module overrides the calculations of the previous module. The equations of the syntext of each node by each module are given in the following table (The contexts have been omitted because they are all empty.)

node	*node*↑pre by *count*(*pre*)	*node*↑ *tranf.*	*node*↑post by *count*(*post*)
number	$1+\text{num}_1\uparrow\text{pre}$	{}	$1+\text{num}_1\uparrow\text{post}$
num_1	$1+$"`d`"$\uparrow\text{pre}+\text{num}_2\uparrow\text{pre}$	new↑	$1+$"`d`"$\uparrow\text{post}+\text{num}_2\uparrow\text{post}$
num_2	$1+$"`0`"$\uparrow\text{pre}$	{}	$1+$"`0`"$\uparrow\text{post}$
$\text{num}_3(\mathit{new})$	$1+$"$\mathtt{s}_1$"$\uparrow\text{pre}+\text{num}_4\uparrow\text{pre}$	{}	$1+$"$\mathtt{s}_1$"$\uparrow\text{pre}+\text{num}_4\uparrow\text{post}$
num_4	$1+$"$\mathtt{s}_2$"$\uparrow\text{pre}+\text{num}_5\uparrow\text{pre}$	{}	$1+$"$\mathtt{s}_2$"$\uparrow\text{post}+\text{num}_5\uparrow\text{post}$
num_5	$1+$"`0`"$\uparrow\text{pre}$	{}	$1+$"`0`"$\uparrow\text{post}$

The syntext for node num_1 is

$$\{\text{pre} \mapsto 1+\text{“}\mathtt{d}\text{”}\uparrow\text{pre}+\text{num}_2\uparrow\text{pre}\} \oplus \text{new}\uparrow \oplus \{\text{post} \mapsto 1+\text{“}\mathtt{d}\text{”}\uparrow\text{post}+\text{num}_2\uparrow\text{post}\}$$

$$\begin{aligned} = {} & \{\text{pre} \mapsto 1+\text{“}\mathtt{d}\text{”}\uparrow\text{pre}+\text{num}_2\uparrow\text{pre}\} \oplus \\ & (\{\text{pre} \mapsto 1+\text{“}\mathtt{s}_1\text{”}\uparrow\text{pre}+\text{num}_4\uparrow\text{pre}\} \oplus \{\} \oplus \{\text{post} \mapsto 1+\text{“}\mathtt{s}_1\text{”}\uparrow\text{pre}+\text{num}_4\uparrow\text{post}\}) \oplus \\ & \{\text{post} \mapsto 1+\text{“}\mathtt{d}\text{”}\uparrow\text{post}+\text{num}_2\uparrow\text{post}\} \end{aligned}$$

$$\begin{aligned} = {} & \{\text{pre} \mapsto 1+\text{“}\mathtt{d}\text{”}\uparrow\text{pre}+\text{num}_2\uparrow\text{pre}\} \oplus \\ & \{\text{pre} \mapsto 1+\text{“}\mathtt{s}_1\text{”}\uparrow\text{pre}+\text{num}_4\uparrow\text{pre}, \quad \text{post} \mapsto 1+\text{“}\mathtt{s}_1\text{”}\uparrow\text{pre}+\text{num}_4\uparrow\text{post}\} \oplus \end{aligned}$$

$$\{\text{post} \mapsto 1+\text{"d"}\uparrow\text{post}+\text{num}_2\uparrow\text{post}\}$$

$$= \{\text{pre} \mapsto 1+\text{"}\mathtt{s}_1\text{"}\uparrow\text{pre}+\text{num}_4\uparrow\text{pre}, \quad \text{post} \mapsto 1+\text{"d"}\uparrow\text{post}+\text{num}_2\uparrow\text{post}\}$$

From this is can be seen that pre takes its value from the transformed tree (num_4) and post takes its value from the original tree (num_2).

5.10 Implementation of generalizations

The previous sections have described how useful and widely applicable modules can be written by using generalizations of the traditional attribute grammar scheme. In most cases the generalizations can be implemented by using analogous features of Miranda in defining the module functions.

Abstraction is accomplished by abstraction. An example of this has already been seen in the section on syntactic modularity (section 5.3.1). If a module has a parameters $p_1, \ldots, p_n$ then they are written as the parameters of a function returning a module, like this:

```
my_module p1 p2 ... pn =
  ag where ag pattern1 head body = ...
              ⋮
```

Each of the parameters is in scope for the entire module body (the local definition ag), which is the desired effect.

Production patterns can usually be implemented by pattern matching against the parse tree. Recall the the tree node for the production *lhs* $\rightarrow$ *rhs* is represented as (`P` *lhs′ rhs′*) where *lhs′* is the internal program name for the l.h.s. symbol and *rhs′* is a list of trees representing the derivation trees of the r.h.s. . An ordinary production like

factor $\rightarrow$ unop factor

has an implementation pattern

$$\texttt{(P Factor [}\underbrace{\texttt{P Unop any}}_{1}\texttt{, }\underbrace{\texttt{P Factor any'}}_{2}\texttt{])}$$

The right side is constrained to be two trees, one tagged with the symbol `Unop` (1), the other with `Factor` (2). There is no constraint on the derivation tree for these nodes. If, say, factor2 is used as a tree valued expression then it stands for the entire construction "`(P Factor any')`". A production with pattern variables relaxes the constraint on the symbol values, although some symbol values may be constrained to be the same, like E here:

E $\rightarrow$ Op E   `(P e [op,P e any])`

In the general case the production pattern can be replaced by a predicate:

```
attrfuns prodn had body = ..., if matches prodn
```

This is not the preferred choice, however, since many compilers produce better code for pattern matching, especially when matching several similar patterns. A predicate can be used in conjunction with a pattern. Sometimes it is necessary to use a predicate, an example is where the pattern contains names which are module parameters, like the list modules. As the function definition is written in a curried form, the parameter names in the pattern part are new names and fail to constrain the production pattern correctly. The correct way to handle this is to use a predicate to explicitly test for equality. The module one_list would be coded like this

```
one_list list item =
  ag where ag (P list' [P item' any]) head body =
                 ..., if list'=list & item'=item
          ⋮
```

The pattern '$\mathsf{A}\rightarrow\mathsf{B}_1\ \mathsf{B}_2\ \ldots\ \mathsf{B}_n$' is translated to the most general production pattern: `(P lhs rhs)`. The tree of symbol B_i may be obtained by list indexing: `rhs!(i-1)`, and the number n is the length of the right side, i.e. `#rhs`. Production patterns which are written as syntax patterns (i.e. using the $[\![\ldots]\!]$ notation) are implemented as ordinary patterns which are compiled by the techniques described in chapter 4.

Quantification is an expression of a repeated or iterated construct and is naturally implemented as a loop. The Miranda metaphor for a loop is a list comprehension. A loop with i quantified over a range is written `[`*expr*`|`*i*`<-`*range*`]`. Synthesized attributes have to be generated in separate lists for each r.h.s. symbol in the production. Whereas access to $\mathsf{B}_i\!\downarrow\! a$ is written as `(rhs!(i-1)) $d a`, definition of the same attribute is accomplished by adding the pair $(a, value)$ to the ith list of inherited attributes. The pervasive inheritance module is written:

```
pervasive_inheritance attr =
  ag where ag (P lhs rhs) h r =
            ([], [ [(attr, h $d attr)] | i<-[1..#rhs] ])
         ag = nointerest
```

The more complex case of several attributes defined for overlapping sets of r.h.s. symbols is constructed by looping over all the symbols and building up the attribute set for each symbol if its index is in the appropriate set. The rule

$\mathsf{A} \rightarrow \mathsf{B}_1\ \mathsf{B}_2\ \ldots\ \mathsf{B}_n$
  $\mathsf{B}_i\!\downarrow\! a = \mathsf{f(i)},\ \forall\ \mathsf{i} \in \mathsf{S}_1$
  $\mathsf{B}_j\!\downarrow\! b = \mathsf{g(j)},\ \forall\ \mathsf{j} \in \mathsf{S}_2$

is compiled into this code:

```
module_name (P lhs rhs) head body =
    ([], [attrs k | k <- [1..#rhs]])
    where  attrs i = (f' i) ⊕ (g' i)
           f' i = [(a,f i)], if i ∈ s1
                = [], otherwise
           g' i = [(b,g i)], if i ∈ s2
                = [], otherwise
```

5.10.1 Semantic patterns

Sometimes syntax alone is insufficient to match a production. For example, the module rule

$\mathsf{E} \rightarrow \mathsf{L\ Op\ R}$
  $\mathsf{E}\!\uparrow\!\mathsf{val} = \mathsf{apply\_op(Op}\!\uparrow\!\mathsf{operator,\ L}\!\uparrow\!\mathsf{val,\ R}\!\uparrow\!\mathsf{val)}$

is used to detect binary expressions. This would be translated to the Miranda definition:

```
ag (P e [l,op,r]) head body =
   ([(Val,apply_op (opA $u Operator) (lA $u Val) (rA $u Val))],
    [[],[],[]])
   where [lA,opA,rA] = body
```

This rule will match any production with three symbols on the right side, for example stmts $\rightarrow$ stmts "," stmt. This example would lead to an error at attribute evaluation time if stmts was asked for a val attribute and "," did not synthesize an operator attribute. The r.h.s. symbols do not fulfill their promise to synthesize val and operator attributes. The module rule must be extended to match only if the symbols synthesize the required attributes. Here is what the extended code looks like:

```
ag (P e [l,op,r]) head body =
   ([pair], [[],[],[]])
   where pair = (Val,apply_op (opA $u Operator)
                               (lA $u Val) (rA $u Val)),
                  if opA $synthesizes Operator &
                     lA  $synthesizes Val &
                     rA  $synthesizes Val &
               = (Dummy, Void), otherwise
         [lA,opA,rA] = body
```

The code synthesizes a val attribute if the required attributes are defined. If they are not defined then the dummy attribute is synthesized in its place. Recall that in section 5.2 where the basic form of the attribution function was developed it was noted that the parameters head and body must not be decomposed (i.e. has parts accessed) until the pattern matching had committed. This was necessary to avoid a cyclic redex, and is the reason why the dummy attribute is synthesized if the right side symbols cannot synthesize the required attributes.

5.11 A debugging module

The debugging module synthesizes a debug attribute which is a textual 'picture' of the attributed parse tree. Each production node in the attributed tree is printed with a list of attributes and their values. The debug attribute itself is omitted from the list of attributes because that would cause an infinite regress as the debug attribute would have to contain a printed description of itself. The r.h.s. symbols are indented in order under the l.h.s. symbol, so the printout is an inorder traversal of the attributed tree. This module is useful in debugging attribute grammars which are run on small inputs. For large inputs the volume of output is overwhelming and it would be a good idea to pass down a predicate to filter out unwanted parts of the tree. The plain version of the module is sketched below[6]:

[6]The notation $\{a\} \lhd f$, taken from the Z notation [Spivey 1989], is called domain subtraction and denotes the function which behaves like f except that it is undefined at a.

$$
\begin{array}{l}
\mathsf{A} \rightarrow \mathsf{B}_1\ \mathsf{B}_2 \ldots\ \mathsf{B}_n \\
\quad \mathsf{A}{\uparrow}\mathsf{debug} = \quad \mathsf{show}(\mathsf{A}) \\
\qquad\qquad\qquad {+\!\!+}\text{``}\ \mathtt{syn:}\text{''}{+\!\!+}\mathsf{map\_image}(\{\mathsf{Debug}\}{\lhd}\mathsf{A}{\uparrow}) \\
\qquad\qquad\qquad {+\!\!+}\text{``}\ \mathtt{inh:}\text{''}{+\!\!+}\mathsf{map\_image}(\mathsf{A}{\downarrow}) \\
\qquad\qquad\qquad {+\!\!+}\mathsf{newline} \\
\qquad\qquad\qquad {+\!\!+}\mathsf{indent}(\mathsf{B_i}{\uparrow}\mathsf{debug})\ \forall\, \mathsf{i} \in \{1..n\} \\
\quad \mathsf{map\_image}(\mathsf{m}) = \mathsf{flatten}(\{\mathsf{show}(\mathsf{a}){+\!\!+}\mathsf{show}(\mathsf{v}) \mid (\mathsf{a}{\mapsto}\mathsf{v}) \in \mathsf{m}\}) \\
\mathsf{A} \\
\quad \mathsf{A}{\uparrow}\mathsf{debug} = \mathsf{show}(\mathsf{A}){+\!\!+}\mathsf{newline}
\end{array}
$$

The module assumes a greater reflective capacity in the attribution language that previously used. The show function is assumed to take any object and return its printable representation. It also assumes that the syntext can be modified and then passed as a value to another function. The rules for translating modules to Miranda do not cover all of these points, and this module might raise special issues when generating efficient code as discussed in the following section.

5.12 Sources of inefficiency

There are several sources of inefficiency in the general attribute grammar processor. Each source can be traced to a stage in the generalization of the attribute grammar processor. On small grammars the modular Miranda program is several hundred times slower than the equivalent unstructured program written according to the rules given in section 5.1.2. I now pinpoint and discuss the individual sources of inefficiency. The next section describes how a more efficient evaluator may be generated from the same description.

Compare the behaviour of a program that evaluates a modular attribute grammar with a classical attribute grammar processor:

- The parse tree is pattern-matched by each module. A classical processor can identify the production in constant time by its production number. This is not the case in the modular case as the pattern determines the shape of the parse tree node rather than its identity.
- The partial attribute sets computed by each module are combined. A classical processor does not do anything analogous.
- Attributes are accessed by searching the context or syntext. A classical processor accesses the attribute from a record field (or global variable in some optimized processors).
- Values in our system have been tagged to that they are all of the same type. Tagging incurs a penalty that a less constraining type system or a classical AG would not incur.

The factors affecting the run time are

N The size of the parse tree. N is the number of nodes in the parse tree.

M Number of modules. Each module inspects every parse tree node. Therefore there is a pattern match against the parse tree for every module. The attribute sets computed by each module are combined, there are $M-1$ combinations for each parse tree node

P The average complexity of the pattern match in a module. This might be proportional to the size of the modules, but in many cases it will be nearly constant because pattern matches on algebraic data types can often be compiled into small trees of case statements.

A The average number attributes associated with a parse tree node. This affects the combination of the attribute sets. The combination is $O(A^2)$ in the current implementation. It affects the access time of an attribute. A linear search for the attribute takes time $O(A)$.

These factors interact with each other. A rough analysis is given. It is easiest to combine these factors by thinking of the grammar evaluation taking place in two steps: setting up the attribute equations and reducing the attribute equations. In reality these are thoroughly interleaved by the process of lazy evaluation. The setup time comprises:

$O(N \times M \times P)$	time spent pattern-matching against the tree
$O(N \times (M-1) \times A^2)$	time to combine the partial attribute maps

The equation evaluation stage calculates the value of each attribute at each node. I assume that the equations have, on average, a small constant number of references to other attributes—a reasonable assumption because many of the rules are simple (like copy rules) and the attribute space is divided into small sets of related attributes. The time is then $O(NA \times A)$, the second A incurred from looking up attribute values. This gives an overall time of

$$O(M \times N \times P + M \times N \times A^2 + NA \times A)$$

The number of attributes probably grows faster than the size of the modules so $M.N.A^2$ is the dominant term. Compare this with NA for a classical attribute grammar (NA because all the attributes are calculated once in some order).

5.13 Removing inefficiency

In the previous section I identified the sources of inefficiency in the Miranda program that models a modular attribute grammar. If modular attribute grammars are to be a realistic tool these inefficiencies must be removed. This section discusses how this might be done. The basic idea is to transform the modular program to produce an unstructured program that does not have any of the overheads due to the module operators and the concessions made to a uniform representation to allow the operators to be used. The ideas in this section have yet to be tested by producing a program which does this transformation. Many of these ideas point to areas open to further research.

The modular program is inefficient because (a) it does lots of pattern matching; (b) the attributes are stored in discrete maps which are combined and are used to look up the attributes values; and (c) all of the values are tagged. Very little can be done about this if the modular attribute grammar is presented by itself. When the underlying context free grammar is known this information can be used to guide the derivation of a specialized version of the modular grammar. The specialized version should look like the

$$\text{expr} \rightarrow \text{expr addop term}$$
$$\text{expr} \rightarrow \text{term}$$

```
synthesize_expr prodn inh =
  synth prodn
  where
  synth (P Expr [P Expr any, P Addop any', P Term any'']) =
        syn
        where
        rsyns0 = synthesize_expr  (P Expr any)   rinhs0
        rsyns1 = synthesize_addop (P Addop any') rinhs1
        rsyns2 = synthesize_term  (P Term any'') rinhs2
        children = [NODE (P Expr any)   rinhs0 rsyns0,
                    NODE (P Addop any') rinhs1 rsyns1,
                    NODE (P Term any'') rinhs2 rsyns2]
        (syn,rinhs) = attrfuns prodn (NODE prodn inh syn) children
        rinhs0 = rinhs!0
        rinhs1 = rinhs!1
        rinhs2 = rinhs!2
  synth (P Expr [P Term any]) =
        syn
        where
        rsyns0 = synthesize_expr (P Term any) rinhs0
        children = [NODE (P Term any) rinhs0 rsyns0]
        (syn,rinhs) = attrfuns prodn (NODE prodn inh syn) children
        rinhs0 = rinhs!0
```

Figure 5.7: A grammar fragment and the corresponding specialized version of `synthesize`. Note that as intermediate results like `rinhs1` become visible they have to be named.

program in figure 5.3. In the following sections I will describe how this can be achieved and the restrictions which might be placed on the attribute grammar modules to help this process.

5.13.1 Syntax

Once the syntax is known the modular grammar can be specialized to produce a set of attribute grammar processors, one for each nonterminal. This set is built by specializing the `synthesize` function (page 92) with the axiom's production to produce the function `synthesize_`*axiom*. To specialized `synthesize` with a nonterminal it is written out with one case for every production of the nonterminal. Because for each case the production is known, the call `map2 synthesize ...` in can be unfolded and the calls to `synthesize` replaced by calls to the appropriate specialized versions. An example of the result of this transformation is given in figure 5.7.

After this transformation the program has the same top-level structure as the evaluator derived in section 5.1.2, i.e. each nonterminal has a function and that function has a case for each production of the nonterminal. The next task is to specialize the attribution function. Again this is done by symbolically executing the function with the production, doing as much work as possible on the limited information of the production template. What can be achieved is (a) a single or small number of productions from each module is selected, often this will be the nointerest case; and (b) the calls to `comb` can be completely unfolded.

5.13.2 Attribute sets

Keeping the attributes in a mapping from names to values causes inefficiency in both the construction of the map and in accessing the attributes later. The current implementation of the $\oplus$ operator assumes nothing of the attribute names except that they can be compared for equality. The map is represented as an unordered list of (name,value) pairs. For two maps f and g of lengths $|\ \mathsf{f}\ |$ and $|\ \mathsf{g}\ |$ the operation $\mathsf{f}\oplus\mathsf{g}$ takes time $O(|\ \mathsf{f}\ | \times |\ \mathsf{g}\ |)$. If an ordering is placed on the attribute names (any ordering will do) then a more efficient representation can be used. Example 1: the map is stored as an ordered list. The operation $\oplus$ is implemented as an unbalanced merge (preferring g over f), complexity $O(|\ \mathsf{f}\ | + |\ \mathsf{g}\ |)$. Lookup is still $O(A)$. Example 2: the map is implemented as a balanced tree. The $\oplus$ operation is implemented as successive updates giving time $O(|\ \mathsf{g}\ | \log |\ \mathsf{f}\ |)$ and access in time $O(\log A)$.

A better approach than any change in representation would be to infer the exact structure of the attribute map and then replace the map with a tuple that could be constructed in one n-tupling operation (presumably $O(A)$) and accessed by projection functions[7] ($O(1)$). It is possible to infer the structure of partially known data types using the latest techniques from partial evaluation; this is discussed under feasibility below.

To aid the inference of a tuple structure it is necessary to choose a data structure that has the property that it has the same final shape regardless of the order of the operations in its construction. This allows the lookup operation to be replaced by a precomputed direct access function because an attribute will always be stored in the same place. The problem of the shape of the data structure arises because the attributes of a nonterminal can be specified in a different order for each production. Most balanced trees do not have the desired behaviour and unordered lists certainly don't. Ordered lists do have this property.

Once the attribute map operations have been reduced to a single constructor function and a set of projection functions the map is isomorphic to an n-tuple and operations can be replaced by tuple operations which, being primitive operations, are probably more efficient.

5.13.3 Tagging

Type inference can detect if a type tag is always the same. If the attribute map had been replaced by a tuple then the constraint that all of the attributes must have the same type is lifted and the type tags can be removed. This is really an extension of the operation of converting the map into a tuple. In the tagged case an attribute map (represented as an ordered list) could have the structure

[7] A projection function is a function that returns part of (or one of) its argument(s). The projection function $\pi_1(x, y) = x$ projects the point (x, y) onto the x-axis.

`[(attr1,`*`value1`*`), (attr2,`*`value2`*`), (attr3,`*`value3`*`)]`

This would be converted into the tuple

`(`*`value1,value2,value3`*`)`

Assume that it is inferred with the map structure that attr1 always has the tag "`N`" and attr2 always has the tag "`OP`". Then the map has the structure

`[(attr1,N `*`value1'`*`), (attr2,`*`value2`*`), (attr3,OP `*`value3'`*`)]`

which can be converted into a tuple with different types for each slot:

`(`*`value1',value2,value3'`*`)`

5.13.4 Feasibility

I believe that the specialization approach to converting a modular attribute grammar into a monolithic AG is feasible using recently developed technology. The major techniques used are *specialization*, *driving*, *abstract interpretation* and *type refinement*.

Specialization is the symbolic evaluation of a function with a (partially determined) value. Expressions in the function are marked as either known (static) or unknown (dynamic) at specialization time. Operations on static values are performed by the specializer and residual code is generated to performed the operations on the dynamic data.

Driving is a technique of seeding a partial evaluator with sufficient information for it to generate interesting specializations but not enough that the partial evaluator tries to generate an infinite program. An example of this is where the attribute evaluator is specialized with respect to the productions for each nonterminal. If the partial evaluator was this eager and given the whole description of the grammar then it would try to generate a specialized function for every possible parse tree. This is prevented by guiding the partial evaluator by giving it one nonterminal at a time.

Partial evaluation requires a degree of abstract interpretation to decide if a value is known or unknown at specialization time. More information can be used in the abstract interpretation. The Fuse partial evaluator [Weise & Ruf 1988] for Scheme uses an interesting technique: concrete values are used in the abstract interpretation as far as possible. Only when a concrete value is combined with a symbolic value is information lost. Fuse is able to deduce and use information about data structures; the authors give an example where an environment is updated with an unknown list of assignments. The partial evaluator can deduce that bindings in the environment stay in the same place even though the assignments are unknown. This power can be used to determine that the position in the attribute maps is the same. A test example has been tried with the $\oplus$ operator to verify this.

Type refinement is the replacement of one data type and its operations with another. Arity raising [Mogensen 1989a] is a kind of type refinement. Arity raising is the splitting of a compound object that is passed as an argument to a function into several parts and passing them all separately. The transformed function has a parameter for each part and thus has more arguments and hence a raised arity. Arity raising is used where the data structure is partly known; e.g. a parameter might be a known to be list of two unknown items. It is more efficient to pass the two items separately rather than construct the list only to pull it apart again inside the function. It is clear that the refinement of an attribute map to a tuple is a similar transformation to arity raising.

The parse tree data type may also be refined. In the original monolithic translation I gave an example parse tree representation that is more efficient than the general representation that was used subsequently: each production has a unique tag which means that only one part of the parse tree node needs to be inspected to identify the production. When the modular program has been specialized with respect to a context free grammar there is no need for the general representation as the decision whether a particular production and module rule match has been precomputed by the specialization process. The remaining pattern matches serve two purposes. The residual match of a pattern that patches a flat (unnested) production identifies the subtrees of the production. A non-flat pattern, such as specified by the higher order attribute grammar pattern

$$\mathsf{expr}{:}[\![\texttt{<expr}_1\texttt{>+<term}_1\texttt{>+<term}_2\texttt{>}]\!]$$

will have some remaining tests. In both cases the original parse tree data type,

```
production ::= P nonterminal [production] | T terminal
```

can be replaced by the type that is a tagged union with one tag per production and one tag per terminal. Constant terminals and empty productions have no substructures, other productions have tree substructures and value terminals have some data (like an identifier's name):

$$\begin{array}{ll}
\texttt{tree ::=} & \texttt{P}_1\ \texttt{tree}_1\ \ldots\ \texttt{tree}_{|\mathsf{rhs}_1|}\ \ | \\
 & \texttt{P}_{\mathsf{k}}\ \texttt{tree}_1\ \ldots\ \texttt{tree}_{|\mathsf{rhs}_{\mathsf{k}}|}\ \ | \\
 & \texttt{CT}_1\ |\ \texttt{CT}_2\ |\ \ldots\ |\ \texttt{CT}_{\#\mathsf{ct}}\ \ | \\
 & \texttt{VT}_1\ \texttt{datum}_1\ |\ \ldots\ |\ \texttt{VT}_{\#\mathsf{vt}}\ \texttt{datum}_{\#\mathsf{vt}}
\end{array}$$

The substitution of this type throughout is straightforward, but care should be taken to convert all instances which includes the construction of replacement parse trees in modules which have higher order attribute grammar rules. The benefit of this type refinement is that the dispatch in specialized functions like synthesize_expr can be compiled into a constant time operation as the tags in the topmost tree data structure all differ.

There are three features of the modular attribute grammar notation and the modular Miranda program that warrant further discussion. These are: that the partial evaluation must handle higher order functions; the quantified rules; and the use of semantic matching.

Higher order functions are not a problem because it is only the functions map, comb and oplus that need to be unfolded to achieve the required specialization. The rest of the specialization is first-order. The latest partial evaluators can handle higher order functions in any case [Bondorf 1990].

Quantified rules, like in the pervasive inheritance module, look hard to handle symbolically but remember that the right side of the production is known at specialization time and the loop(s) can be unrolled symbolically.

The most awkward thing to handle is semantic matching where a pattern is matched only if the symbols synthesize or inherit the appropriate attributes (section 5.10.1, page 111). The context free grammar and the attribute map domains are static and can be removed by partial evaluation. Generating a dummy attribute when the semantic match fails changes the shape of the attribute map, making it dynamic (determined at run time) and thus not possible to reduce at specialization time. The only way to avoid this is to prove that, for the single production in question, the semantic match will always be true (or false). This task is essentially the same as that performed by Dueck & Cormack's definabilty analysis and is a global property of the attributed grammar. It is clearly out of the league of current partial evaluators to (a) perceive the need for such a proof and (b) do anything about it. There are two approaches to this problem: either the partial evaluator would have special knowledge built into it to do the job, or the modular grammars could be written more carefully so that they do not require semantic matching. The former is obviously preferable as it removes an unnecessary burden for the programmer. I give an example of how grammars could be written more carefully. Consider the binary operator pattern used in section 5.10.1. The pattern was

E → L Op R

This pattern was too general. It might be replaced by the patterns

E → L addop R
E → L mulop R
etc

A convenient notation (e.g. replacing "Op" with "Op:{addop,mulop}" and maintaining a single pattern) would make this easier, but it does rob some of the power of the modular grammars and so is not a very satisfactory solution.

5.14 Designing modular attribute grammars

Now that modular attribute grammars have been described it is apt to discuss some of the issues involved in the design of modules. Attribute grammar modules alone do not guarantee a clean, reusable design; like any other tool they can be misused.

5.14.1 Coverage

I use the term *coverage* to mean the set of entities that a module in some way affects. The syntactic coverage of a module is the set of productions for which the module defines some attributes. The semantic coverage of a module is the set of attributes that are calculated by that module. Sometimes it is convenient to group some related productions or attributes together rather than consider their coverage separately. Parameterized modules may, of course, have variable coverage, depending on the particular instantiation of their parameters.

The coverage of a module can be depicted by filling in a grid like the one below. Each square indicates an interaction between a syntactic and a semantic feature. The square is marked with the module that specifies the interaction.

	syntax1	syntax2	syntax3	...	syntax$n$
semantics1	X	Y			
semantics2	X	Y	Y		Y
semantics3	X	Z	X		
⋮					
semantics$m$					X

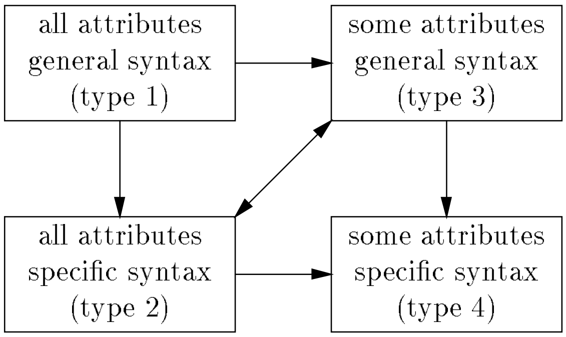


Figure 5.8: Module types

The ability for modules to define the behaviours of attributes in a general manner, without recourse to specific productions, allows the coverage grid to be filled. Without this ability it would only be possible to fill in the 'diagonal' of the grid. This was one of the shortcomings of the investigations and solutions in chapter 3. The attribute grammar modules offer a neat solution to the coverage problem.

5.14.2 Issues

Different semantic modules contribute to the language description at different levels of generality. Some modules define general rules for coverage, others define specific details of the implementation of small parts of the language. Modules can be roughly categorized by the precision with which they pinpoint the language features which they cover and the shape of the coverage. The categorization has implications for the software engineer who specifies how the modules should be combined. In the following it pays to remember that modules are combined by one module refining the definitions provided by the previous module. Consider the two axes of attribute coverage and syntax coverage, with values indicating general or specific applicability (figure 5.8). It is clear that type 4 modules should refine the more general statements made by other types of module, and that type 1 modules should be refined by all other types of module. What is not clear is whether type 2 modules should be refined by type 3 modules or the other way round. If one takes a semantics (i.e. attribute) oriented stance then type 2 modules should be refined by type 3 modules. This stance is acceptable because semantics is more important than syntax.

As a concrete example compare the coverage of the following modules which have been given earlier. The type of each module is listed withe the module. One module from each type has been chosen to illustrate how they interact.

unit_production_copy_rule		type 1	(p. 103)
get_vars:	bucket_brigade	type 3	(p. 101)
	get_vars_detail	type 4	(p. 101)
one_list	(instantiated with variablelist and variable)	type 2	(p. 104)

In all of the experiments undertaken these (and/or similar) modules have been combined in the order 1-2-3-4. Without a discipline to order them these modules give conflicting definitions.

The unit production copy rule module matches many productions and many attributes and is clearly a 'basic general behaviour' or type 1 module. The one_list module is a type 2 module. It is specific to a part of the syntax (variablelists) but still general in the attributes. It is a 'general syntactic behaviour' module. The bucket brigade module is restricted to a small number of attribute but applies to a wide general syntax. The get_vars_detail module is a type 4 module. It is specific to a few concrete productions and a few concrete attributes.

Type 4 modules are usually the instrument used to change a language into a new similar language.

Modules are frequently composed of submodules, for example get_vars on page 101. What is the type of a module composed of submodules of different types?

This is a difficult question to answer because the mixed-type compound module behaves in part like all of the submodule types—the type 4 parts are harmless to generalizations made by previous modules combined with the compound module, but lower types may override the refining effects of previous modules. This topic has not been thoroughly investigated and deserves more attention. One approach would be to make the combination mechanism sensitive to the module types and combine the modules according to the ordering of there types. This raises the question of what to do with modules that are inherently mixed-type, for example a module that is written with a special very specific case preceding a more general case, and how the module type can be recognised automatically.

5.14.3 Refinement and revision

Module composition by overriding offers the opportunity to program by refinement. A new module can modify the behaviours specified by earlier modules. This same mechanism allows three different programming practices:

- Refinement. A new module provides more specific behaviour for a previous more general module. An example of this is the get_vars module on page 101. The get_vars_detail submodule provides refinement for the more general bucket_brigade instance.

- Incremental extension. A language can be extended by (1) extending the concrete syntax and then (2) adding one or more modules which provide the refinements required for the new features.

- Revision. Sometimes it is interesting to a language prototype implementor to alter the meaning of part of a language. A simple example is to generate additional code to gather statistics, say every time a variable is referenced. The prototype builder might want alter the behaviour for new special cases—for example it might be more convenient to implement the 'self' pseudo-variable in an object oriented language by interposing a module that handles this case.

If two modules perform the same function but in different ways then a complete language with one module can be converted into the language with the other module by composing the attribute grammar with the alternative module. For example, consider two modules for stack-frame management. The first module implements stack frames as linked lists, the second as offsets into a stack array. Implementation Imp1 is built using the first module. The implementation built by 'Imp1 $\oplus$ module2' reads as 'language Imp1 but with a linear stack'. For example, consider an implementation of CSP [Hoare 1985]. There are two different semantics for the alternative command—angelic (permitting

```
module env                              module def
  (1) 'goal → A ...                       (4) A → ...  B ...
      A.env = 0;                              B.def = B.env;
  (2) A → B ...                           (5) A → ...  B
      B.env = A.env;                          A.def = B.def;
  (3) A → ...  B C ...                    (6) A →
      C.env = B.def;                          A.def = A.env;
```

Figure 5.9: Dueck & Cormack's bucket brigade. Quick explanation: (2) The left symbol of the r.h.s. inherits env from the left part and (3) other symbols in the r.h.s. inherit env from the def attribute synthesized by their left neighbour. (6) An empty production sends its env back as a synthesizes def attribute, otherwise (5) def is synthesized by the rightmost symbol of the r.h.s. In (4) the r.h.s. symbol is a terminal which cannot synthesize def (by (5) or (6)), so env is passed along as def

more user errors) and demonic (committing to a choice immediately). An angelic implementation of CSP could be constructed by composing a demonic implementation with a angelic_alternative module.

5.15 Comparison with Dueck & Cormack's MAGs

There are similarities between the modular attribute grammars presented here and those of Dueck & Cormack and it is interesting is it to make a detailed comparison. Dueck & Cormack use several modular attribute grammars (MAGs) to define a language implementation. A single MAG is a set of patterns and associated templates. The patterns are applied to a context free grammar. For each match between a production and pattern a set of attribute computations is generated. Both matching and computation generation are constrained—pattern matching uses *definability* and the generated computations are selected by *need*. Dueck & Cormack define these new concepts.

There are two angles from which I compare my modular attribute grammars with those of Dueck & Cormack. The first is a comparison of the expressiveness of the two notations. The second compares implementation issues.

5.15.1 Model and derivation

As two attempts to describe attribute computations in a modular framework my approach and Dueck & Cormack's solution bear a number of similarities. The origins of the two approaches differ. My approach was based on the generalization of an attribute grammar evaluator. The evaluator was generalized and then stretched to provide modular facilities which were then projected back into the attribute grammar notation. This has influenced the result: the modules are composed in an order; production patterns are applied in order; the versatility of the production patterns reflects the ability of the underlying evaluator. In contrast Dueck & Cormack's MAGs were developed as a preprocessor for a standard monolithic attribute grammar evaluator.

5.15.2 Module coupling and dependence

Modules communicate by the attributes that they use (import) and the attributes that they define (export). The analogy with module importation and exportation is weak because any module may use or compute any attribute. Neither system provides a notation for defining the imports and exports or any other restraint. Several modules may define a particular attribute. This raises the question how to resolve any conflicts between the attribute values calculated by different modules[8]. Dueck & Cormack avoid the issue and tend to stick to the safe methodology of defining one attribute per module. My approach is to define an order in which the modules are applied. Later modules can override the decisions made by previous modules. This is expanded into a programming methodology based on refinement.

Both systems allow the definition of mutually dependent modules. An example mutual dependence of this is appears in the bucket brigade in figure 5.9. Mutual dependence is a gain over phase oriented decomposition in which each module depends on the previous module as it allows different but related concerns to be described independently without placing an execution order on the parts of the computation.

5.15.3 Syntax abstraction

The pattern matching nature of the modules provides a tool for abstracting away from the concrete syntax. Traditionally an implementor might construct an abstract syntax tree during parsing and write an attribute grammar for the abstract syntax. Perhaps the abstract syntax is constructed in one phase of an attribute grammar and then processed by another phase. Often the translation from the concrete syntax to the abstract syntax is simple. Dueck & Cormack describe how patterns can be used to work directly with the concrete syntax as though it was an abstract syntax. They identify two common mappings from concrete to abstract syntax: *grouping* and *elision*. Grouping is when the concrete syntax has several different nonterminals with similar meanings which can be represented in the abstract syntax by one nonterminal. The expression hierarchy is rich in potential groupings. The nonterminals expr, term and factor can be grouped as a single expression sort, as can the operators from each level of the hierarchy. The productions expr $\rightarrow$ expr addop term and term $\rightarrow$ term mulop factor are grouped as a single binary expression tree node expr $\rightarrow$ Binop(expr,operator,expr).

Elision is removal of irrelevant detail. Concrete syntax is often contains irrelevant detail, for example, the parenthesized expression and unit productions. The "..." pattern of MAGs is better at eliding irrelevant symbols (e.g. parentheses) but the wholesale copying of the context or syntext that my scheme permits is more convenient for specifying copy rules.

Both systems share the ability to impose more than one abstract view on the parse tree. While the expressions may be abstracted as above, a complete program might also use the pervasive_inheritance module to transfer information down the tree. This imposes another abstract view: the tree is simply a tree with uniform type and nodes of varying arity, with abstract syntax tree $\rightarrow$ $\mathsf{Node}_\mathsf{k}(\mathsf{tree}_1,\mathsf{tree}_2,\ldots,\mathsf{tree}_\mathsf{k})$.

5.15.4 Terminals

In my model terminals may have inherited and synthesized attributes just like nonterminals. This is slightly unconventional but it makes the treatment of all vocabulary symbols uniform and provides the flexibility that additional unit rules like

[8] This is a different issue to conflicts generated internally in a module, which is discussed later

X → ... eqsym ...
eqsym → "="

give without needing the device. To an attribute grammar pattern a terminal looks a bit like a null production (having no constituents save an automatically synthesized attribute), allowing the pattern matching to extend uniformly to terminals.

In the example of a Dueck & Cormack module (figure 5.9) the B in template (4) is a terminal. It is a terminal because otherwise a def attribute would have been synthesized by rules (5) or (6), since all productions are caught by (5) (non-nullable) and (6) (nullable). This is quite complicated reasoning for a pattern match—consider a module which has a greater number of more complex patterns. I feel that the intention that the bucket brigade passes through the terminal would be better indicated by either a notation for matching terminals explicitly or by treating terminals and nonterminals uniformly.

5.15.5 Copy rules

Neither system implements automatically generated copy rules. Dueck & Cormack state

> "We hesitate to incorporate automatic generation of copy rules into the MAG translator because this is a particular solution for a particular problem that may be addressed by a more general solution."

I agree with their sentiment and I have shown in previous sections how modules like pervasive_inheritance and unit_production_copy_rule provide a framework in which copy rules are defunct.

5.15.6 Rule precedence

Rule matching in the two systems differs significantly. In my system the rules in a single module are matched against the parse tree in order. The first matching pattern is taken and the attributes for the module are calculated according to the attribute equations for the matching pattern. A pattern can match a parse tree in only one way. This is a control-flow based matching and can be thought of as operating at attribute evaluation time. The only way that several sets of attribute equations can come to bear is if they are in separate modules.

Dueck & Cormack's MAGs differ in all of these respects. The MAGs are processed to produce a monolithic attribute grammar. Each MAG provides a set of patterns. All the patterns are matched against a production in all possible ways. Attribute calculations may be produced for all of the matches. This process yields many attribute calculation rules. Some can be eliminated because they are impossible: not all of the required attributes are available: a symbol A may be requested to produce an attribute a but there is no rule to calculate A.a. Any calculation relying on A.a is eliminated. This is formalized as a set of *definable* attributes. Similarly, not all of the calculations are needed. There may be ones which calculate values which can never be used—e.g. a calculation may be specified for an attribute B.b which does not appear in the r.h.s. of any calculation. This is formalized as a set of *needed* attributes. Only after the definable and needed attributes have been deduced are the attribute equations generated. There may still be ambiguous rules—two equations specifying the same attribute. This is solved by placing a partial order on the rules and selecting the 'earliest' rule. The partial order gives precedence to earlier rules in the module, so the rules within a module should be given in order of importance.

My modular grammars allow a module to match only one pattern against a parse tree node. This can be circumvented by making the module out of submodules. Each basic submodule has the one pattern limit but the composite module has no no such limit. Thus is is possible to mimic the MAG with several patterns which all match one production. Submodule composition with the $\oplus$ operator will give the later submodules priority over the earlier submodules, precisely the reverse of a Dueck & Cormack MAG. It is not possible to mimic the MAG pattern which matches a single production in several ways.

The semantic matching is equivalent to Dueck & Cormack's definability set. I have no equivalent to the need set except that the attribute evaluator is lazy and will automatically never evaluate an attribute that it does not need.

Dueck & Cormack define no rule ordering *between* modules. I presume this is because they never have two modules which define the same attribute, even for completely unrelated syntax. In this situation there can be no ambiguity as the modules compute disjoint sets of attributes. Extending the rule ordering to include patterns from different modules would give Dueck & Cormack's MAGs the power to define refinement modules. The ordering can be extended like like this: Patterns within a single MAG are ordered $pt < pt'$ if pt occurs before pt' in the MAG. Dueck & Cormack use this ordering. The MAGs may be ordered according to the order of their composition $M_1 \oplus M_2 \oplus \cdots \oplus M_k$. Two modules M and M' are ordered $M < M'$ if M occurs earlier than M' in the composition. Now the pattern ordering can be extended across all modules by labeling the patterns with their modules. The single item pt is replace by the pair (M, pt). Pairs are ordered

$$
\begin{array}{lll}
(M', pt') < (M, pt) & \text{if } M < M' & \text{later modules are preferred} \\
(M, pt) > (M, pt') & \text{if } pt < pt' & \text{rules in order within a module}
\end{array}
$$

5.15.7 Generics

I have used many generic modules. For example, the bucket_brigade is a generic module that may be instantiated to produce left-to-right data flow for several attributes. Dueck & Cormack observe

> "... We find many modules, not necessarily concerned solely with attribute propagation by copy rule, that share a similar overall appearance. A possible solution is to provide generic modules, parameterized by attribute and production symbols, that can be used to produce module instances. This would make it possible to incorporate an instance of a generic general-purpose bucket-brigade module as a sub-module of any module, or, indeed, to compose modules entirely of instances of generic modules."

Generic (parameterized) module have been implemented and used. To the list of parameterizable parts of a module—attribute and production symbols—I have added modules themselves (syntactic modularity, section 5.3.1). Generic modules have proven so useful that it is difficult to conceive of general purpose modules without a generic facility. Without parameterization it is not possible to reuse the same module for different purposes or to use pre-existing modules that by accident use attributes with the same name.

5.16 Extensions

The modular attribute grammars would be an improved software engineering tool if they supported some of the following extensions.

5.16.1 Local attribute names

It would be safer if a module or group of modules could hide the attributes that are used to communicate internally. For example, it has been noted that the harvest_and_sow module can be implemented by using the bucket_brigade module and some additional attributes. Local attribute names could hide the new name(s) from a user of the module, and prevent accidental use.

5.16.2 Renaming

An alternative mechanism to parameterization for module re-use is renaming. A renamed module is structurally the same as the original module but some names (for example, attribute names or grammar symbols) are replaced with other names. Renaming has the advantages that the kind of reuse need not be anticipated and the original module can be any module, in particular an unparameterized module that is functional in its own right. Parameterization will still be required for parameterization with respect to other modules.

5.16.3 Declaration of imports and exports

A module's interface is the set of attributes that it defines and the set of attributes that it uses. This could be formalized with declarations, for example

> **module** code_list(X, Xlist)
>   **uses** code, env
>   **provides** code
>   Xlist → X "," Xlist   *—productions as on page 100*

The advantages of a formal declaration of the interface are

- The module can be checked against the interface.
- Modules which refine other modules can be detected by inspecting the interface rather than the implementation. If two modules provide the same attribute then one will refine the other.
- Modules that are parameters would have to conform to a description of an interface.

I have not worked out the details of a system with any of these extensions but a starting point would be algebraic specification languages like OBJ [Goguen & Winker 1988] and ASF [Bergstra *et al.* 1989] which have versatile and well developed module combination facilities.

5.17 Summary

I have presented a system of modular attribute grammars. A formulation for writing an attribute grammar as a program in a lazy functional language was derived from the work of Johnsson [1987].The formulation was generalized to factor out the attribute equations from the parse tree traversal, which yielded a single function which characterizes attribute grammar evaluation. The factored out equations were then amenable to being determined by different functions. This allowed a degree of syntactic modularity to be introduced to the language specification by apportioning different productions to different functions. A combining form was developed that allowed different functions to calculate different attributes. This allows semantic modularity where a module is concerned with a single semantic issue represented by a small number of attributes. The productions and attributes were generalized to allow a function with a few cases to apply to a large number of productions from the context free grammar. This gives the modules reusablility as they no longer tied to a specific grammar. In short, a monolithic attribute grammar evaluator was transformed into an evaluator that allowed syntactic modularity, semantic modularity and module reuse.

The translation of grammar modules to a Miranda program is described. This translation is analysed and suggestions are given for improving its efficiency. The suggestions are based on directly attacking the inefficiency introduced by the process of generalization. It is argued that this can be done automatically using the latest program transformation techniques from the field of partial evaluation.

Software engineering with the modular grammars is described. It is discussed how modular attribute grammars support an interesting paradigm of explicit refinement and two other operations that are particularly useful to language prototyping: incremental extension and revision of semantics.

The modular attribute grammars are compared in detail with a similar system developed by Dueck & Cormack [1990].

6 Extended examples

This chapter contains extended examples of the ElderII system and the modular attribute grammars. The first section describes the *me too* language and how ElderII is used to translate the *me too* set expressions into the lingua language. This is not a complete implementation of *me too* as only the set expressions are considered, and the set expression syntax is built on top of the grammar for lingua rather than the proper grammar for *me too*. The second section shows a modular attribute grammar for the description of complete *me too* and a simulation language.

6.1 *me too*—a description

me too is a mathematical language for the design and prototyping of software components. It is based on functional programming and constructive set theory. The theory is used to guide the implementation of maps, relations and sequences which are all represented as sets.

A syntax for the *me too* set notation is given in figure 6.1. Braces are used to denote a set. The empty set is written "`{}`". Set elements can be listed:

```
{1,2,3,4,5}
```

Set elements in a listed set are expressions and if two or more of the expressions evaluate to the same value then naturally only one copy will be in the set:

```
{2+0,1+1,2*1}  =  {2}
```

Set elements may also be specified by an integer range. This is an extension to the original *me too* syntax, equivalent to the `range` function.

```
{1..5}  =  range(1,5)  =  {1,2,3,4,5}
{5..1}  =  range(5,1)  =  {}
```

Sets can be specified by *set comprehensions*. A set comprehension gives a prototype element expression and a list of generators and predicates. Generators state what values variables in the prototype expression may assume and predicates restrict or filter these values to those which satisfy the predicate.

```
range(f) = {y | (x,y):f}
range({(1,3),(2,3),(3,1)})  =  {1,3}

{x*x | x:{-5..5}}         =  {0,1,4,9,16,25}
{x | x:{1..5}; even(x)}   =  {2,4}
```

The prototype expression may even be a set expression. This is used in the `triangle` function, a function which generates a set of sets, and has been used to benchmark ElderII. The number of elements in the set is the nth triangular number $\frac{n(n+1)}{2}$. Notice that this works because, for example, the sets $\{1,3\}$ and $\{3,1\}$ are the same set so that i,j pairs below the diagonal can be thought of as being folded to points above the diagonal, giving a triangular shape.

```
triangle(n) = {{i,j} | i:{1..n}; j:{1..n}}
triangle(4) = {{1},{1,2},{1,3},{1,4},{2},{2,3},{2,4},{3},{3,4},{4}}

{ card(triangle(n)) | n:{1..10}}  =  {1,3,6,10,15,21,28,36,45,55}
```

Additional operators are defined

```
{1,2,3} union {3,4,5}     =  {1,2,3,4,5}
'bar in {'foo,'bar,'baz}  =  true
```

In addition to sets, *me too* supports tuples. Elements of tuples are written as a list in parentheses. The pair "`(5,'blue)`" is a tuple with two parts, a number and symbol. A symbol is like a string and is imported as part of the lingua grammar. A relation is a set of pairs. Two items x and y are related if the pair (x, y) is in the set. A finite mapping (function) is a relation in which the first element of the pair is unique.

6.2 Translating *me too* set expressions with ElderII

me too is implemented by translating the expressions into lingua expressions which perform the same operations. The *me too* sets are implemented by transforming the set expressions into a functional form. A set expression is replaced by an equivalent expression, containing no set syntax, that calls library functions to perform operations on the sets.

The set data type is implemented as an unordered list of elements, with a type tag `SET` at the head of the list. The list is unordered because there is no obvious ordering of the elements and it is not the purpose of this work to consider efficiency of the implementation of the underlying data types. A production implementation of *me too* would probably implement sets with an asymptotically more efficient data structure, perhaps using hash tables, AVL trees or 2–3 trees. All of these would require placing an ordering on the sets and their elements, and an arbitrary ordering would have to be contrived for this purpose. As the original *me too* implementation used unordered lists I do likewise to allow accurate comparisons to be made between the two implementations.

The transformations from the set syntax to a purely functional syntax as written in *lingua* are given in figure 6.2. A set comprehension is compiled into an expression comprising mainly of three functions, *mapset*, *filterset* and *unionstar* which implement the following set operations.

$$mapset(f, S) \;=\; \{f(x) \mid x \in S\} \;=\; \bigcup_{x \in S} \{f(x)\}$$

$$filterset(f, S) \;=\; \{x \mid x \in S \wedge f(x)\} \;=\; \bigcup_{x \in S} \textit{if } f(x) \textit{ then } \{x\} \textit{ else } \emptyset$$

$$unionstar(S) \;=\; \bigcup_{S' \in S} S'$$

All the set operations are ultimately reduced to one of three primitive operations:

- the empty set $\emptyset$.
- making a singleton set $\{x\}$.
- finding the union of two sets $S \cup T$.

```
grammar metoo .

import lingua .

primary -> set .

set -> "{" "}" .
set -> "{" expr "|" generator moregenerators "}" .
set -> "{" elementlist "}" .

generator -> pattern ":" primary .

moregenerators -> ";" generator moregenerators .
moregenerators -> ";" filter moregenerators .
moregenerators -> .

filter -> primary .

elementlist -> elementlist "," elementspec .
elementlist -> elementspec .

elementspec -> expr ".." expr .
elementspec -> expr .
```

Figure 6.1: *me too* set expressions

Multiple generators and filters are reduced to the simple case of a single generator. Consecutive filters are combined, which is more efficient that filtering twice[1]:

$$\{e(x) \mid x \in G \wedge p_1(x) \wedge p_2(x)\} \;=\; \{e(x) \mid x \in G \wedge p'(x)\}$$

where $p'(x) = p_1(x) \wedge p_2(x)$. A filter may be combined with a generator by filtering the source of the generator:

$$\{e(x) \mid x \in G \wedge p(x)\} \;=\; \{e(x) \mid x \in \{x' \mid x' \in G \wedge p(x')\}\}$$

The case of multiple generators is more interesting. Each value x_i from the first generator parameterizes the set of values produced by the second generator:

$$\begin{aligned} & \{e(x, y) \mid x \in G_1 \wedge y \in G_2\} \\ = \;& \{e_{x_1}(y) \mid y \in G_2\} \cup \{e_{x_2}(y) \mid y \in G_2\} \cup \ldots \\ = \;& \textstyle\bigcup_{x \in G_1} \{e(x, y) \mid y \in G_2\} \end{aligned}$$

Enumerated sets are reduced by taking the union over the parts:

$$\{e_1, e_2..e_3, e_4\} \;=\; \{e_1\} \cup range(e_2, e_3) \cup \{e_4\}$$

$$range(a, b) \;=\; \begin{cases} \emptyset, & a > b \\ \{a\} \cup range(a + 1, b), & a \leq b \end{cases}$$

[1]This is also the case in the original *me too* implementation

```
   def metooexpandonce
1        metoo.primary:[[ { } ]] ->
         metoo.primary:[[emptyset]]

2      | metoo.primary:[[ { <expr> | <pattern> : <primary e2> } ]] ->
         metoo.primary:[[ mapset(fun <pattern> -> <expr> end, <primary e2>) ]]

3      | metoo.primary:[[ { <expr> | <generator g1>; <generator g2>
                              <moregenerators> } ]] ->
         metoo.primary:[[ unionstar({{ <expr> | <generator g2> <moregenerators>} |
                              <generator g1> } ) ]]

4      | metoo.primary:[[ { <expr> | <generator>; <primary e1> ; <primary e2>
                              <moregenerators> } ]] ->
         metoo.primary:[[ { <expr> | <generator>; (<primary e1> and <primary e2>)
                              <moregenerators> } ]]

5      | metoo.primary:[[ { <expr> | <pattern> : <primary e2> ;
                             <primary f>  <moregenerators> } ]] ->
         metoo.primary:[[ { <expr> | <pattern> :
                             filterset(fun <pattern> -> <primary f> end,
                             <primary e2>)
                        <moregenerators> } ]]

6      | metoo.primary:[[ { <elementlist>, <elementspec> } ]] ->
         metoo.primary:[[ setunion({<elementlist>}, {<elementspec>}) ]]

7      | metoo.primary:[[ {<expr>} ]] ->
         metoo.primary:[[ singletonset(<expr>) ]]

8      | metoo.primary:[[ { <expr from> .. <expr to> } ]] ->
         metoo.primary:[[ range(<expr from>, <expr to>) ]]

9      | metoo.addexpr:[[ <addexpr> union <term> ]] ->
         metoo.addexpr:[[ setunion(<addexpr>,<term>) ]]

10     | metoo.comparexpr:[[ <comparexpr> in <addexpr> ]] ->
         metoo.comparexpr:[[ metoomember(<comparexpr>,<addexpr>) ]]

11     | other -> other
    end.
```

Figure 6.2: Translation of *me too* set expressions. Italic numbers label the function cases and are not part of the program

6.2.1 Implementation

The `metooexpandonce` function is called repeatedly on a form until it yields its input. Then it is called repeatedly on its subtrees. The function specifies the set of transformations described above. There are one or more patterns containing each new construct. Inspection shows that each construct is eventually re-written out of the parse tree.

The first rule is the simplest. It simply replaces the set expression "`{}`" with the replacement identifier.

The structure of the rules for the list of generators has important implications for the re-write rules. A poor choice will increase the number of cases that have to be specified in the transformation rules. The generator list has been specified as a right-associative list, i.e. it is of the form `(a ; (b ; (c ; d)))`. This aids traversing the list from left to right, as the first item in the list is always easily extracted as it is in the same position. Compare this with the less favourable left-associative list, where the whole tree must be traversed to extract the leftmost item: `(((a ; b) ; c) ; d)`. Also the generator list has been specified to have a null ending. Compare this with the alternative productions:

```
set -> "{" expr "|" generators "}"
generators -> generator ";" generators .
generators -> generator .
generators -> filter ";" generators .
generators -> filter .
```

These productions are right-associative, but the final generator matches a different production to all of the others. To extract the first generator from the syntax as given in figure 6.2 this pattern is sufficient:

```
metoo.set:[[ { <expr> | <generator> <moregenerators> } ]]
```

The alternative productions require two cases—one for the case where the list contains only one element and another for the case where the list contains more:

```
metoo.set:[[ { <expr> | <generator> } ]]
metoo.set:[[ { <expr> | <generator> ; <generators> } ]]
```

Some rules rewrite a part of the parse tree in context. An example of this is the fourth case in `metooexpandonce`, where the two primaries are joined to form one primary, but only if the two primaries are the second and third elements of a generator list. This transformation has two other significant points. The first is that the replacement expression is in parentheses. This makes the `and` expression a primary. Without the parentheses the constructor fragment would not parse. Secondly, this rule precedes the fifth case which absorbs a filter into the generator. If the fourth and fifth cases were reversed then the fourth case would be shadowed by the fifth case and never be matched.

The sixth case splits an enumerated set into subsets which are combined by union. Because union is associative and commutative we don't care that the element specification list is left-associative. This rule could also have been written as below to use the `union` operator. The advantage of this is that if the union operator function (`setunion`) is changed it now occurs in only one place in the transformations. Note too that parentheses are required by the operator precedence.

```
| metoo.primary:[[ { <elementlist>, <elementspec> } ]] ->
  metoo.primary:[[ ({<elementlist>} union {<elementspec>}) ]]
```

The union operator is simply converted into a function call to a library function by case 9.

Cases 7 and 8 convert the 'simple' sets consisting of a singleton set or a single range into the appropriate library function call. The "`..`" rule is only rewritten in the context of it being the only thing in the element list. Together cases 6, 7 and 8 compile an enumerated set into nested function calls. For example the expression "`{a,3..5,b}`" is transformed by the following steps:

```
{a,3..5,b}
setunion({a,3..5},{b})
setunion(setunion({a},{3..5}), singletonset(b))
setunion(setunion(singletonset(a), range(3,5)), singletonset(b))
```

All of the transformations make the assumption that the library items like `mapset`, `emptyset`, `range`, etc. will be in scope in the final program.

6.3 Experiments with Attribute Grammars

I have advocated the use of attribute grammar modules for structuring and reusing programming language fragments. This section gives an extended example of these ideas in use. As complete examples the modules demonstrate the issues on a larger scale than the description in previous chapters. In particular this section demonstrates

- Module reuse. Syntax and attribute grammar modules are reused in the construction of two translators.
- Scopes. One of the translators uses type information and has scopes whereas the other does not. This does not prevent module reuse.
- Higher order attribute grammars. There is a larger example of using derivation tree transformations.

Translators for two languages are built using both syntax and attribute grammar modules. The two languages are *me too*, the executable specification language based on constructive set theory that was described earlier in this chapter, and *SysDyn*, a language for specifying systems dynamics models. An example application is described for each language so that the translators can be fully appreciated. All of the attribute grammar modules presented in this section have been translated into working Miranda programs by the methodology described in the previous chapter.

The translator for *me too* has two phases. The first is the transformation of the set expressions sublanguage of *me too* into expressions into the functional subset. The second is the compilation of the functional subset into Common Lisp. Both tasks illustrate processes that are difficult to achieve with purely syntax based transformations. As there are two phases this translator exhibits phase oriented decomposition in addition to language oriented decomposition.

The *SysDyn* translator is a single phase translator. It has many components in common with the *me too* translator. The *SysDyn* language reuses the concrete syntax

module combinations
  A → Left Op Right
    A↑code = "(" ++Op↑operator++" " ++Left↑code++" " ++Right↑code++")"
  A → Op Arg
    A↑code = "(" ++Op↑operator++" " ++Arg↑code++")"
  A → Func "(" Args ")"
    A↑code = "(" ++Func↑code++" " ++catlist(Args↑code)++")"

module operator
  relop → "="
    relop↑operator = "EQUAL"
  addop → "+"
    addop↑operator = "+"
  addop → "union"
    addop↑operator = "SET-UNION"
  etc.

Figure 6.3: The combinations and operator modules

of expressions. The grammar modules used to define the syntax of *me too* and *SysDyn* appear in appendix B.

Both of the translators produce Lisp code as their output. For simplicity of presentation this code is generated as a string, the textual form of the Lisp program. It would be more efficient to generate the internal form of the Lisp program (cons cell, symbols, atoms etc.) but the mechanics of doing this might obscure the points that I wish to demonstrate.

6.3.1 Translation of expressions

Expressions form an important part of both *me too* and *SysDyn*. Basic expressions are translated by two modules: combinations and operator (figure 6.3). The combinations module is concerned with the basic form of expressions and is general. The operators module is concerned with the meanings of the individual operators and is likely to be specific to a language.

The combinations module recognises three basic forms of expression: infix operators, prefix operators and function application. The pattern matching relies on both syntactic cues and semantic cues. The function call case is distinguished by the parentheses and the operator cases are distinguished by the operator's ability to synthesize an operator attribute. The combinations module exports the code attribute and imports the operator attribute. The first rule of the combinations module translates an infix expression like "1+2" into the Lisp prefix form, e.g. "(PLUS 1 2)". The only part of the attribute equations that requires further explanation is that in the function case it is assumed that the variable pattern Args matches a nonterminal that synthesizes a code attribute that is a list of the code attributes of each of the arguments. This is easily arranged by using a list abstraction (section 5.7.2).

The operator module synthesizes an operator attribute for each operator production. It is uninteresting and might be quite long as it must define each syntactic operator in

module stat

```
stats → stat "."
    stats↑code = stat↑code
stats → stat "." stats
    stats↑code = stat↑code ++ newline ++ stats↑code
stat → var "=" expr
    stat↑code = "(SETQ "++var↑code++" "++expr↑code++")"
stat → "type" var "=" expr
    stat↑code = "()"
stat → var ":" expr
    stat↑code = "()"
stat → var "(" varlist ")" "=" expr
    stat↑code = "(DEFUN "++var↑code++" ("++catlist(varlist↑code)++") "
                          ++expr↑code++")"
```

Figure 6.4: Translating statements

the language.

It is possible to decompose this module into submodules according to some criteria, e.g. a module for each of relop, addop etc. A better decomposition would be into 'common' operators like "+" that appear in (nearly) all languages and specific operators that are unlikely to appear in another language. The common operators module would be reusable.

6.3.2 Statements and definitions

me too statements appear only at the top level, and define global entities. Each statement is terminated by a full-stop. There are four types of statement:

- variable definition (`v = e`)
- type definition (`type t = e`)
- type conformance (`v : e`)
- function definition (`f(vars) = e`)

Global variables are mapped onto Lisp global variables (symbol values) and global functions are mapped onto Lisp global functions (symbol functions). This distinction between global functions and global variables is a feature of Common Lisp (and other versions of Lisp known collectively as 2-lisps). This distinction is explained later. Type information does not generate any code[2].

Global variable definition is translated into a Lisp SETQ operation. The *me too* definition *var* = *expr* is translated into the Lisp operation

```
(SETQ symbol expression)
```

[2]Declared type information is not used at all. This is in keeping with the original implementation of *me too* [Henderson 1986]

```
module let
   factor → "let" defnlist expr
      factor↑code = "(LET (" ++namepairs++ ") " ++expr↑code++)
      namepairs = map2(embrace,defnlist↑bindname, defnlist↑code)
      embrace(x,y) = "(" ++x++ " " ++y++ ")"
   defn → var "(" varlist ")" "=" expr
      defn↑code = "#'(LAMBDA (" ++catlist(varlist↑code)++ ") " ++expr↑code++ ")"
      defn↑bindname = var↑code
   defn → var "=" expr
      defn↑code = expr↑code
      defn↑bindname = var↑code
```

Figure 6.5: Local definitions

where *symbol* is the Lisp symbol assigned to hold the variable and *expression* is the translation of ***expr***. There is no need to generate code to create or allocate the symbol because the Lisp system creates it when it is first mentioned to the system. In a similar manner function definitions are translated to the Lisp `defun` form. The *me too* function definition add(x,y)=x+y would be translated into

```
(DEFUN ADD (X Y) (+ X Y))
```

6.3.3 Let expressions

The *me too* language contains a let-form for making local definitions. The semantics is that all the expressions in the let form are evaluated in parallel and then the body expression is evaluated in the scope where the names are bound to the resulting values. The names are not visible to each other. This is like the Lisp `LET` form so it is natural to translate the *me too* let form into the Lisp `LET` form. The `LET` form also allows local functions[3], e.g.

```
let  twice(f,x) = f(f(x))
     a = 3*5
   twice(sqrt,a)
```

although the functions cannot be mutually recursive because they cannot 'see' each other. The let module translates a let expression (figure 6.5). The definitions are used to construct a list of names (bindname) and a parallel list of expression code to compute the value for that name. The lists are combined pairwise to construct the Lisp definitions. The local function is interesting in that it is translated into an anonymous λ-expression. The parameter list is constructed in the same way as for a global function.

6.3.4 Function call types

This section describes how type information may be used to process a program. As an example I use the Common Lisp property that functions and variables have different

[3]This is an extension of the original *me too* language which has been added to introduce scoping issues

name-spaces. Generating Lisp code requires that this is taken into account. While this example might seem a little unusual as a 'type' problem, it shares with other type problems the feature that names in the input language are treated differently according to how they are defined.

The modules combinations and let between them translate most *me too* expressions into correct Lisp code. However, there is a problem with functions because the target language, Common Lisp, is a 2-lisp. Different versions of Lisp are classified as either a 1-lisp or a 2-lisp depending on how functions and values are bound to names. A 2-lisp allows a name to designate both a variable and a function. A 1-lisp only allows a name to designate a variable and a function is just one of the possible values that the variable can assume. Scheme [Rees & Clinger 1986] is a 1-lisp. Common Lisp is a 2-lisp. In this Common Lisp dialogue `F` is both a variable and a function (user input is italic and "`>`" is the Lisp system prompt)

```
> (defun f (x) (* x x))    ;define the function
F
> (setq f 100)             ;assign to the variable
100
> (f 5)                    ;use the function
25
> (+ f 1)                  ;use the variable
101
```

A consequence of having a name denote a separate variable and function is that either the function or variable interpretation must be chosen for each occurrence of a name in a program. The default interpretation is to use the function if the name occurs at the head of a list, otherwise use the variable. Consider the function twice which applies a function f to an argument and then again to the result of the first application. In *me too* we define it like this:

```
twice(f,x) = f(f(x))
```

If this is naively translated into a 2-lisp the result is

```
(defun twice (f x) (f (f x)))
```

This will call the global function `F` (defined above) instead of the parameter which is variable. To use a function stored in a variable the `funcall` function is used to apply the function to its arguments. The correct translation of twice is

```
(defun twice (f x) (funcall f (funcall f x)))
```

So `funcall` is used to 'convert a value into a function'. The second problem is the converse: converting a function into a value so that it can be passed as a parameter. The Lisp special form `function` achieves this. To call `twice` with the `sqrt` function and `5` this special form is used like this:

```
(twice (function sqrt) 5)
```

The code "`(function sqrt)`" may be abbreviated to `#'sqrt`. 2-lisps may be implemented by associating two storage cells with each symbol: a value cell and a function cell. When the symbol is in an argument position the value stored in the value cell is

```
module ftypes =
      pervasive_inheritance(env)
   ⊕ contours
   ⊕ twolisp_funcall

module contours
   program → stats
      stats↓env = stats↑decls

   stats → stat “.” stats
      stats_1↑decls = stat↑decls ∪ stats_2↑decls
      stats → stat “.”
      stats_1↑decls = stat↑decls

   stat → var “=” expr
      stat↑decls = {var↑name ↦ OBJ}
   stat → var “(” varlist “)” “=” expr
      stat↑decls = {var↑name ↦ FUNC}
      expr↓env = stat↓env ⊕ reduce(⊕,varlist↑decls)

   var → ident
      var↑decls = {ident↑name ↦ OBJ}

   factor → “let” defnlist expr
      inner_env = factor↓env ⊕ reduce(⊕,defnlist↑decls)
      defnlist↓env = inner_env
      expr↓env = inner_env
   defn → var “=” expr
      defn↑decls = {var↑name ↦ OBJ}
   defn → var “(” varlist “)” “=” expr
      defn↑decls = {var↑name ↦ OBJ}
      expr↓env = defn↓env ⊕ reduce(⊕,varlist↑decls)


module twolisp_funcall
   A → var “(” Args “)”
      A↑code = “(”++calltype++var↑code++“ ”++catlist(Args↑code)++“)”
      calltype = “”, if var↑name ∉ A↓env
               = “”, if A↓env(var↑name) = FUNC
               = “FUNCALL ”, if A↓env(var↑name) = OBJ
   factor → var
      A↑code = var↑name,          if var↑name ∉ factor↓env
             = “#'”++var↑name,    if A↓env(var↑name) = FUNC
             = var↑name,          if A↓env(var↑name) = OBJ
   ident
      ident↑code = ident↑name
```

Figure 6.6: Modules for function and value types

used. When the symbol is in a function position the function stored as the value of the function cell is used. The `function` special form means 'the value stored in the function cell'.

The *me too* translator needs to be improved so that it generates `funcall` and `function` forms at the appropriate places. This is essentially a type coercion problem. Identifiers are allocated to symbol value cells or symbol function cells according to their definition. Thus there is a two point type domain {OBJ,FUNC} of names denoting respectively the value cell and the function cell of a symbol. A use of a symbol may be inappropriate to its type and has to be coerced to the correct type by the use of either `funcall` or `function`.

While this problem is specific to the choice of Common Lisp as a target language, the problem is an example of an important problem in language definition—finding out what a name or construct means so that it can be handled appropriately. The elements of this problem are *definition*, *visibility* and *use*. Each element can be made the responsibility of its own module.

The ftypes (figure 6.6) module is a composite module that corrects the *me too* implementation for a 2-lisp. This module illustrates the following features:

- Declarations and environments. All definitions add some information about the defined name to an environment which assigns a type (OBJ or FUNC) to each name visible at that point.
- Scopes. Some names (e.g. function parameters) have restricted visibility.
- Reuse of general modules. The pervasive_inheritance module is used to abstract away from irrelevant syntax.
- The refinement methodology of programming. ftypes redefines how a function call and variable use are translated. This module overrides the naive implementation specified by the combinations module. The rest of the translation is unaffected.

The module is a composition of three submodules. The first submodule ensures that the environment is available at all points in the derivation tree. The contours module collects definitions and alters the pervasive inheritance of the environment to make the definitions visible in their scopes. It is called the contours module because it implements the contour model of nested (local) scopes. Finally, twolisp_funcall redefines how a function call and variable reference are translated to Lisp to take the type into account.

Definitions are collected by the decls attribute. Both declarations and environments are modeled as functions from names to types (OBJ—defined as a variable, FUNC—defined as a function). The functions can be extended or combined with the function override operator ($\oplus$).

A sequence of statements collects all the definitions which eventually end up being turned round to form the environment (the first production of the module). Thus the top-level environment will contain type information for exactly those names which are defined in the program. Each of the defining forms synthesizes a decls attribute that specifies the implementation type of the form. A name is defined in: a top level definition, a local definition (let) and as a parameter to a function. Note that top level functions are of the FUNC type but local functions are of the OBJ type. This is because local functions are translated into λ-expressions which are values.

The environment of a function is extended with the local declarations of the parameters of the function. Names are added by overriding the environment which has the effect of hiding more global definitions of the same name. New declarations are obtained by reducing the decls attribute synthesized by the varlist or defnlist. This is done because the list abstractions used elsewhere in the *me too* translator yield a list of declarations which have to be combined.

The environment is used by the twolisp_funcall module to decide when to use `funcall` or `function`. The variable name is looked up in the environment[4] and if it is of the 'wrong' type then the appropriate form is used. If the name does not appear in the environment then it has not been defined in the program. In this case it is assumed that the default action is adequate, allowing Lisp functions and variables to be used from *me too* without defining them. A more secure alternative is signal an error. Those Lisp functions and variables which are permitted to be used from *me too* can be added to the environment in the first production of the contours module.

6.3.5 Translation of *me too* sets to functional subset

This section describes how *me too* expressions are translated into the underlying functional subset of *me too*. The translations are similar to those described in section 6.2.1, but done using modular attribute grammars and higher order attribute operations.

Some specific translations require a unique symbol generator. In section 5.8 I described how a gensym attribute can be used to do this. In particular I described how the gensym attribute should be incremented, but I did not describe the default behaviour. The default behaviour is to pass the gensym counter in a traversal of the tree. It is convenient to use the bucket_brigade module for this purpose rather than inventing some other traversal. The attributes g_in and g_out are used to copy the gensym number through the tree and it is initialized to zero at the root of the tree:

module gensym_bucket_brigade
  program → stats
    stats↓g_in = 0
  bucket_brigade(g_in, g_out)

The transset module (figure 6.7) specifies a similar set of transformations to those specified by the `metooexpandonce` function described earlier. Many of the transformations are essentially the same, for example, replacing "`{}`" with the identifier `emptyset`. The transformations are superficially different in that they are for different symbols in the grammar—the ELDERII example added the *me too* set operations to the lingua language where the 'smallest' expression type is a primary; the complete *me too* language has a shallower expression hierarchy and the set expressions are rooted at factor.

A profound difference is that lingua has anonymous functions and *me too* does not. The implication of this is that translating the set mapping and filtering operations is a simple syntactic transformation for lingua but not for 'set-less' *me too*. The nameless functions that are needed for the set operations have to be defined as named functions in the enclosing scope. For example, the fifth rule might translate the set comprehension `{x*x|x:S}` into the expression

[4] Since the environment is a function mapping names to types, a name is looked up by applying the environment to the name, hence the form A↓env(var↑name).

```
module transset
   factor:⟦{}⟧
      factor = ⟦emptyset⟧

   factor:⟦{⟨expr⟩}⟧
      factor = ⟦singletonset(⟨expr⟩)⟧

   factor:⟦{⟨expr e1⟩..⟨expr e2⟩}⟧
      factor = ⟦range(⟨expr e1⟩,⟨expr e2⟩)⟧

   factor:⟦{⟨elementspec⟩,⟨elementlist⟩}⟧
      factor = ⟦({⟨elementspec⟩} union {⟨elementlist⟩})⟧

   factor:⟦{⟨expr⟩ | ⟨var⟩:⟨expr source⟩}⟧
      newform = factor:⟦let ⟨ident newname⟩(⟨var⟩) = ⟨expr⟩
                              mapset1(⟨ident newname⟩, ⟨expr source⟩)⟧
      newname = make_name(factor↓g_in)
      factor↑ = newform↑
      newform↓ = factor↓ ⊕ {g_in ↦ factor↓g_in+1}

   factor:⟦{⟨expr⟩ | (⟨varlist⟩):⟨expr source⟩}⟧
      newform = factor:⟦let ⟨ident newname⟩(⟨varlist⟩) = ⟨expr⟩
                              mapsetn(⟨ident newname⟩, ⟨expr source⟩)⟧
      newname = make_name(factor↓g_in)
      factor↑ = newform↑
      newform↓ = factor↓ ⊕ {g_in ↦ factor↓g_in+1}

   etc...
```

Figure 6.7: The transset module.

```
let _NewFunc_42(x) = x*x
  mapset1(_NewFunc_42, S)
```

The new name is generated from the inherited g_in attribute and the new expression replaces the set expression. The gensym number is incremented before being passed on, as described in section 5.8. The case where the generator generates a tuple of values is translated into a call to `mapsetn` with a function that takes each element of the tuple as a separate parameter. `Mapset1` and `mapsetn` are implemented slightly differently—`mapset1` applies the function to each element of a set and builds a set of the results whereas `mapsetn` applies the function to each tuple in the set, spreading the tuple elements between the function arguments.

6.3.6 Sundries

Two minor modules are described here for completeness. If_then_else translates a conditional into Lisp in an obvious manner. The single virtue of this module is that it extends either *me too* or *SysDyn* with a conditional expression.

module if_then_else
  A → "if" Cond "then" ThenPt "else" ElsePt
    A↑code = "(IF "++Cond↑code++" "++ThenPt↑code++
                    " "++ElsePt↑code++")"

Finally, constants generates the Lisp code for constant forms:

module constants
  constant → number
    constant↑code = number_to_string(number↑value)
  constant → string
    constant↑code = """ ++ string↑value ++ """

6.3.7 Top level

The complete *me too* translator is assembled by combining these modules:

module metoo_to_lisp = translation ⊕ transset
    unitproductioncopyrules
  ⊕ ignorebrackets("(",")")
  ⊕ splaylist(exprlist,expr)
  ⊕ splaylist(varlist,var)
  ⊕ combinations ⊕ constants ⊕ operator ⊕ if_then_else
  ⊕ splayzlist(defnlist,defn) ⊕ stat ⊕ let ⊕ ftypes

Note that the transset module is placed last so that the transformations on the parse tree are visible to the whole attribute grammar (see section 5.9 for a full explanation of why this is important).

6.4 *SysDyn*

Systems dynamics is a systems modelling style that concentrates on aggregate levels and rates of flow [Kreutzer 1986]. A system is described in terms of a set of states or *levels* which represent the amount of some 'substance' like money, population or goods, and the changes in the levels, or *rates*. Levels and rates are represented by variables and expressions which are bound in a set of difference equations that describe how the levels and rates affect one another.

The *SysDyn* language defines a model as a set of levels and rates and auxiliary definitions. Each level and rate has an initial value which is a constant expression and an expression defining how the value depends on the other levels and rates. The concrete syntax is

model → "model" ident parts
parts → part parts
parts →
part → "level" var "initial" expr "then" expr
part → "rate" var "initial" expr "then" expr
part → var "=" expr

A model is run by setting the levels and rates to their initial values and then stepping through time. At each time step new values are calculated for the levels and then the rates. This process is repeated until the simulation time exceeds some preset bound. A systems dynamics model is really a set of first-order coupled differential equations.

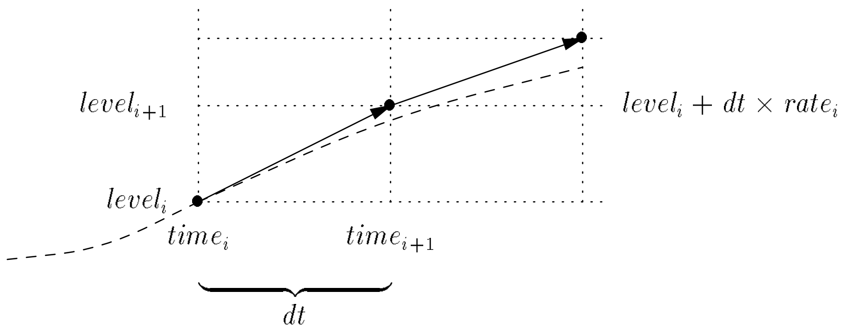


Figure 6.8: Euler's method

```
model Gonts_Lovecycles

level Romantic initial 1000 then Romantic+dt*(OutSane-InLove)
level LoveSick initial 1    then LoveSick+dt*(InLove-OutLove)
level Sane     initial 0    then Sane+dt*(OutLove-OutSane)

rate InLove  initial 0 then InfectionProbability*Romantic*LoveSick
rate OutLove initial 0 then LoveSick/LoveSicknessDuration
rate OutSane initial 0 then Sane/ImmunityDuration

InfectionProbability = 0.001
LoveSicknessDuration = 8  #days
ImmunityDuration = 700    #days

dt = 1
time_limit = 40
```

Figure 6.9: Gont's lovecycles—*SysDyn* model

6.4.1 *SysDyn* example—Gont's lovecycles

Systems dynamics models have been applied to a wide range of problem domains including economics, epidemiology, engineering and nature reserve management. The example that I present here is a fun epidemiology model: DreamWorld's lovesickness epidemics (borrowed from [Kreutzer 1986]). There are three kinds of people in the model: lovesick people who are madly in love; romantics, who are susceptible to lovesickness; and sane people who are not susceptible. Lovesickness is a contagious disease. A romantic is infected by a lovesick person with a daily probability of 0.1% Lovesickness lasts on average 8 days, after which the subject becomes sane and immune to lovesickness. Sanity lasts on average 700 days, after which the subject becomes romantic again[5]. The lovesickness model is written in the *SysDyn* language in figure 6.9. The difference equations solve the differential equations by Euler's method as described in figure 6.8. There are three levels which model the three populations. Each population

[5]The lovesickness model is not unrealistic as an epidemiology model. Like many normally nonfatal diseases, lovesickness is contagious, lasts about a week and confers a long term immunity which lasts a few years.

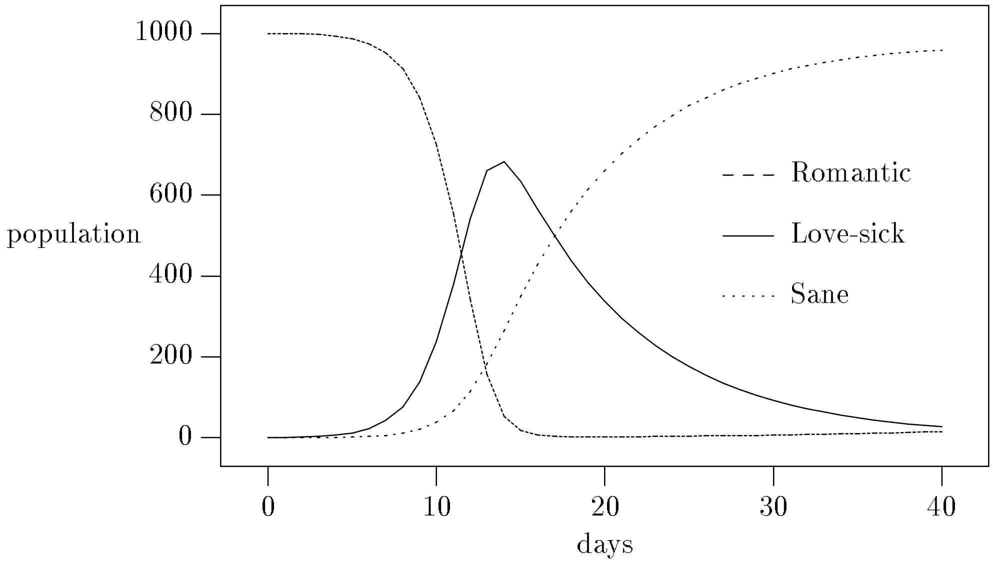


Figure 6.10: Gont's lovecycles—graph of results

has one input and one output which are the positive and negative rates in the equations.

6.4.2 *SysDyn*—implementation

A *SysDyn* model is a set of level, rate and parameter definitions. The translator translates these into a Lisp program that computes the values of the levels and rates through time. The model can be thought of as having two parts—a basic framework that handles the iterative nature of the level and rate computations that is the same for all models, and a 'formula' part that is determined by the particular model.

The framework organises these activities:

- initialization
- reporting results
- (re)computing levels and rates
- termination of the model

The framework is essentially a loop with these activities plugged into the right places and is specified by the `scheme` module (figure 6.11). An invariant of the loop is that the levels and rates are correct for the time indicated by the variable `time`. Both the levels and the rates are recomputed in a two-step process. First the new values are computed and assigned to new variables. Then the new values are copied over the old values. This 'parallel assignment' is done because levels may be defined in a mutually recursive fashion, where the rate of one level is a function of another level, i.e. a rate can be a formula containing levels as well as rate variables. The levels are incremented and 'shifted' before rates to maintain the invariant that the levels and rates are appropriate to the value of `time`. This scheme produces correct results because rate variables and expressions are a function of the levels and the new levels are a function of the rates.

Each of the operations (printing, shifting etc.) is determined by the `part`s specified in the model. Each of these operations is handled by a separate module. The

```
module sysdyn = unitproductioncopyrules ⊕ ignorebrackets("(", ")") ⊕
                exprs ⊕ model

module exprs = splaylist(exprlist,expr) ⊕ splaylist(varlist,var) ⊕
               combinations ⊕ constants ⊕ operator ⊕ ite ⊕

module model = splayzlist(parts,part) ⊕ scheme ⊕
               initial ⊕ increment ⊕ shift ⊕ printinfo

module scheme
  model → "module" name parts
    model↑code =
      "(progn
        (setq time 0 dt 1)
        (progn init)
        (loop
          print
          (setq time (+ time dt))
          (if (> time time_limit)(return))
          inclev
          shiftlev
          incrate
          shiftrate
      ))"
      init = reduce(++, parts↑initial)
      print = model↑printCode
      inclev = reduce(++, parts↑incrementLevel)
      shiftlev = reduce(++, parts↑shiftLevel)
      incrate = reduce(++, parts↑incrementRate)
      shiftrate = reduce(++, parts↑shiftRate)
  ident
    ident↑code = ident↑name
    ident↑next = ident↑name ++ "_new"

module shift
  part → "level" V "initial" I "then" E
    part↑shiftLevel = "(setq " ++ V↑code ++ " " ++ V↑code ++"_new)"
    part↑shiftRate = ""
  part → "rate" V "initial" I "then" E
    part↑shiftLevel = ""
    part↑shiftRate = "(setq " ++ V↑code ++ " " ++ V↑code ++"_new)"
  part → V "=" E
    part↑shiftLevel = ""
    part↑shiftRate = ""
```

Figure 6.11: The basic *SysDyn* scheme

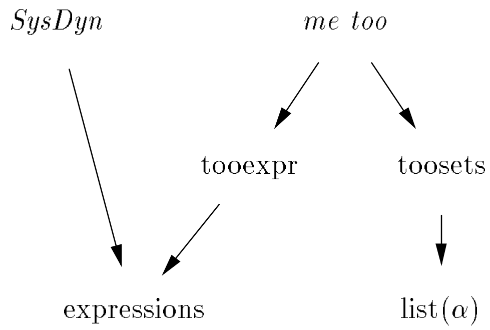


Figure 6.12: Summary of grammar module usage

operations for all the individual parts of the model are collected by a list abstraction (splayzlist(parts, part)). Since the levels and rates are incremented at different times the shift and increment modules synthesize two attributes. This has the effect of dividing the levels and rates without explicitly filtering either the syntax tree or a list-attribute.

module	**attribute**	**purpose**
initial	initial	code to initialize level, rate and parameter
increment	incrementLevel	code to increment level(s)
	incrementRate	code to increment rate
shift	shiftLevel	code to copy new value of level over old
	shiftRate	code to copy new value of rate over old
printinfo	printFormat	the Common Lisp format to print out the value
	printCode	the code to access the item to be printed (part) or code to do the printing (model)

Expressions in the language are translated by the same modules as those used for *me too*.

6.5 Summary

I have presented some larger examples of the ideas and techniques in action. These included a reasonable sized application of ElderII and the application of modular attribute grammars to the construction of two different language translators.

It was found that syntactic transformations can be difficult to use. In particular, the lack of a list primitive in the grammar makes programming more difficult and repetitive because common operations like mapping and filtering have to be written from scratch. These problems are alleviated by the list abstractions that are defined as modular attribute grammars.

The attribute grammar examples demonstrated module reuse. A diagrammatic summary of the module relationships is given in figure 6.13. All of the attribute grammar modules were translated into Miranda and executed to verify their correctness.

Conventional issues like handling declarations and environments were demonstrated with the *me too* declaration types. This example is interesting because it shows how a later module can refine a previous module without the modules being incompatible—the simplistic operation of the combinations module which was adequate for *SysDyn* was

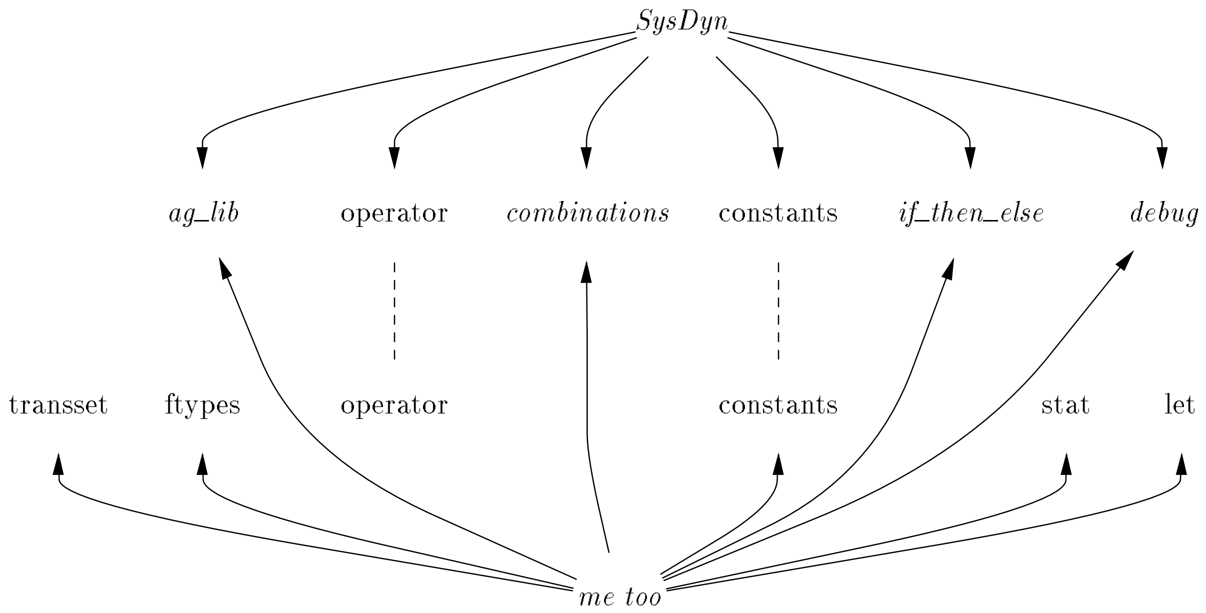


Figure 6.13: Summary of attribute grammar module usage. Truly shared modules are italicized. An arrow means that the module at the tail of the arrow is built using the module at the head of the arrow. ag_lib contains several useful 'infrastructure' modules, for example pervasive_inheritance and splayzlist. Dashed lines mean that the modules should be the same or have a major shared component but it just did not happen that way. These modules should probably be reverse-engineered to increase the module reuse.

refined to generate code depending on the semantic context. Further, the environment was added to the translator without unduly affecting other modules of the translator.

Finally, the *me too* set expressions sublanguage demonstrates program transformation by higher order attribute grammar operations.

7 Conclusions

7.1 Summary

In the previous chapters I have tried to answer the question “How can one write a language processor so that it is structured according to its component language fragments?” This question is interesting because the answer tells us how to construct and use a library of existing fragments and perhaps a little ‘glue’ to rapidly construct a new language processor. The ‘fragment library’ approach is especially suited to building language prototypes as the library is likely to contain fragments that are adequate even if they are not exactly what the implementor had in mind. An implementor of a standardized language is more constrained.

Within some quite stringent constraints, modularity can be achieved using conventional programming paradigms. This is the conclusion of chapter 3, where modular interpreters were written in functional, object oriented and algebraic styles in order to answer the question “Are conventional structuring mechanisms adequate for expressing language fragments?” The restrictions are quite severe and stem from the single control flow thread of the interpreter. Control flow dominates in two ways. First, it is difficult to reconcile modules which scan the program in different ways. One module has to be called to do its information gathering by the other, so the master module needs to be written with anticipation of the subordinate module’s requirements. Secondly, modules have to be able to make decisions on how to handle syntax which is unknown to the module. This is in order to allow the language to be extended by composing the modules.

Modularity in the description of the syntax is less troublesome. In chapter 4 I described a modular parsing system based on top-down parsing. The parser is modular in its description as each nonterminal’s set of productions is translated into an independent parsing function, and the grammar description is modular, the final grammar being constructed by forming a composite grammar from all of the components in a similar way to algebraic specification module composition[1]. Chapter 4 also describes is how to parse program fragments. This is discussed in terms of its implementation in the top-down parser and how it might be integrated with a shift-reduce parser. A small functional programming language called lingua was written. The distinguishing feature of this language is that program fragments are integrated with the pattern matching and data constructor syntax.

The previous two chapters had shown that syntactic modularity was easier to achieve than semantic modularity. It was also evident that static modularity, like the grammar descriptions, was easier to understand than dynamic modularity, like the composition of interpreter fragments.

With these lessons in mind I took a static description of semantics, i.e. attribute grammars, and made them modular. The primary tool in this process was a lazy

[1]As is done in the implementation of OBJ [Goguen & Winker 1988] and ASF [Bergstra *et al.* 1989].

functional programming language which allowed experimentation with the evaluation mechanism. By a process of increasing abstraction in the evaluator I was able to transform a monolithic attribute grammar processor into a form which allowed separate abstract attribute grammar modules.

I call the modules *abstract* here because one production in an attribute module may specify the attribute computations for many attributes and/or productions in the concrete syntax. For example, the bucket_brigade module uses just two rules to specify the attribute relationship for any context free grammar. Abstraction makes semantic attribute grammar modules resilient to changes in the rest of the language and allows them to be reused with different concrete syntaxes. This makes a strong case for defining the syntactic and semantic modules independently.

As well as increased generality, the functional basis for the attribute grammar evaluator allows increased specificity. By this I mean that more restrictive patterns can be specified as well as less restrictive ones. I use this in the higher order attribute operations where the unit being matched by a rule is not a single production in the concrete grammar but a combination of productions forming an incomplete parse tree, i.e. a program fragment. This allows the program to be manipulated by recognizing special cases and by specifying equivalent forms. A program's meaning can be described either by attribute equations or by rewriting the program to another form, or by both.

7.2 Major results

The main result of this research is a modular attribute grammar system that demonstrates a way of writing modules for each facet of a language and then combining the modules to make a whole language. Each such module embodies a language fragment. The modular attribute grammar system is one solution to the problem of writing a language processor as language oriented decomposition into a set of language fragments. Other language oriented decompositions were investigated, namely the decomposition of interpreters and modular syntax. Both of these shared the shortcoming that it was difficult and sometimes awkward to create modules that worked together other than at a simple syntactic level—consider the CCS example and how it needed to be changed to allow interworking with language fragments that managed an environment. The modular attribute grammars have the clear advantage that the modules can be written without awkward constraints, and that a module can use production patterns to specify behaviour over parts of the context free grammar that are unknown or of no semantic interest.

An interesting observation is that the syntax modules and the attribute grammar modules work well together. This is because the AG modules can be abstract with respect to the concrete syntax. As Dueck & Cormack observe, this abstraction serves to remove the need for an intermediate abstract syntax. Since the attribute grammars are matched on syntactic and semantic cues they can apply to a large range of grammars. There is no need for the syntactic modular structure of the concrete syntax to impose on the semantic modular structure. Dueck & Cormack do not mention concrete syntax in [Dueck & Cormack 1990]. I think that concrete syntax modules are a benefit to the language prototype builder. Although the syntax and AG modules are only weakly coupled, using syntax modules with AG modules that were developed at the same time gives some confidence that the attribute modules will work with the syntax modules. An implementor builder can then import the syntax modules into the syntax and the

AG modules into semantics.

The third main result is the application of programming by explicit refinement, or programming by 'difference', to language processors. Programming by refinement allows existing modules to be tailored to handle new cases, increasing the reuse of the existing modules. It also allows general statements about the language structure to be coded (e.g. "sub-parts of a form are evaluated in the same environment", "things must be defined before they are used"). Other language features can then impose control on the over-general but valid statements. The relationship between explicit refinement and object oriented programming is discussed in the following section.

7.3 Engineering with language fragments

The modular systems described in this report were developed in response to the inadequate software engineering properties of the conventional structure of language processors. In this section I discuss the engineering qualities of the new system.

Modules can be written so that each one has a different *assignment of responsibility*. This shows that the modules in the system can obey the basic principle of modular decomposition. However, there is more to consider: how modules communicate and how they prevent inadvertent communication. The interface between modules is crude: a module imports those attributes that it assumes are defined. It exports any attributes for which it defines the values. Several modules may export the same attribute, which export 'wins' depends on the syntactic overlap between the modules and their order of composition. Communication is uncontrolled and undeclared; and at the end of chapter 5 I have suggested improvements aimed at correcting this weakness. The issue of the function of a module as a general 'infrastructure' module of a specific 'feature' module is also addressed in this chapter. The fundamental point of this is that there is a clash between a module's autonomy and the desire to be able to override its behaviour.

To explain this point further I compare the facility of explicit refinement by attribute redefinition with object oriented programming. In object oriented (OO) programming a data type is represented by a *class*. A class has several values and several operations or *methods* associated with it. *Subclasses* are defined by redefining the methods and defining new methods over and above those defined by one class (or more in the case of multiple inheritance). There is usually control over which parts of an object are visible from outside. For example, C++ allows the programmer to define an object component or method to be public, private or protected.

Compare this with AG modules. An AG module defines some attribute values for some class of syntactic and semantic structures (i.e. the attributes are defined by syntactic match and attribute availability). Another AG module independently defines a similar set of attributes. Which module corresponds to the superclass and which corresponds to the subclass depends on the order in which the modules are composed. A better way to look at this is to think of the final language processor as a subclass inheriting from all the modules (classes). Conflicts are resolved by the programmer defined order of the module composition. This poses problems for those wishing to implement a similar system in an object oriented programming language—the multiple inheritance model is bound to differ from this. Smalltalk only allows single inheritance, C++ only allows apparent conflicts if they can be resolved to the same actual method, and CLOS [Bobrow *et al.* 1988] has a complicated scheme which depends more on the depth of the inheritance relationship than the order of 'importation'.

Although modular attribute grammars and object oriented programming both have the ability to define shared (object/production) behaviour and to refine that behaviour by deriving a new subclass/grammar, I have not said that the system is object oriented because the parallel does not extend far enough to find convincing grammar analogues of the objects themselves, the classes and the pseudovariables **self** and **super**.

From a software engineering point of view the reliance of one module on another to perform some function raises the question: should the first module be either part of or declared as required by the second? This is easily catered for in the case of one module consuming an attribute generated by another module: the modules should declare their imported and exported attributes. In the case of modules sharing the responsibility of generating an attribute it is not yet clear what is required in the declaration of each module. I suggest that the declarations at least state that the module may be incomplete. A tool to determine which modules redefine entities defined by other modules would be useful, especially if it detects the case where a module is entirely shadowed by one or more refinements.

Another engineering question relating to refinement is: at what point does refinement become unmanageable? If each module composition is read as 'except' then it is clear that a long cascade of refinements will become unintelligible. This effect has been noted by Standish [1975]. At what level a modular attribute grammar language definition becomes unreadable is not yet known. This effect suggests that at some level it is more profitable to derive a new language from a set of fragments than to add 'hacking' modules to an existing language, or even to define the new language features as a library language fragment that can be reused in many languages.

In summary, there are many engineering aspects of language oriented decomposition that warrant further investigation.

7.4 Implications for language design

The methods and processes used to achieve some goal often affect our perception of how that goal may be achieved. If language processors can be conveniently structured as a collection of fragments reflecting the conceptual structure of the language, what changes can we expect in programming languages as a result of being released from the conventional structure?

One might expect that the freedom to pick and choose modules would result in a proliferation of essentially similar but incompatible languages. Where one language uses a particular syntax to express something, another language uses something slightly different. This is not too different from the current state of affairs. The expression "2×3+4×5" may evaluate to 26 (Pascal), 46 (APL), 50 (left-to-right) or 70 (+ first), depending on order of evaluation, and some languages would consider that input as in error (e.g. PostScript and FORTH).

The availability of convenient modules might encourage their reuse. This could have the effect of making languages more alike than different. Programming would become easier as rules like '*"×" is done left-to-right and before "+"*' are set in the common language. Consider the modern 'integrated package' consisting of a spreadsheet, database and word processor. Each is programmed in its own language. If there are similarities between these languages they are usually by painstaking design. How much easier the implementation would be if the languages were based on the same modules, with refinement modules to define the different parts! It would also make things easier for the programmer as the basics of each language would be the same, making it easier to learn the next language as the only things that differ from the last one are those that *must* be different.

The facility to reuse language fragments would encourage those implementing application languages to take the 'easy' option of buying in language fragments to ensure compatibility with a standard common subset. This would also benefit language researchers as, by common heritage, their language would be more accessible to the 'public'.

7.5 New avenues

To achieve the state of the art aspired to by the previous section much needs to be done. I suggest four areas for further endeavour: efficiency, module engineering, incremental algorithms and object oriented approaches.

Efficiency. The mechanism for appropriate modularity in language implementation needs to be made more efficient. I have outlined how partial evaluation could be used to significantly reduce the execution time of the language processor. However, this raises the issue of the efficiency of this procedure—will generating a processor by partial evaluation be fast enough? An alternative would be to take the lessons learned and incorporate these in a more conventional scheme like that of Dueck & Cormack.

Software engineering. While we have written language fragment modules that accomplish both syntactic and semantic modularity, we need to know more about what makes a good module. Parnas considers two modular structures to a simple problem of reading, processing and writing out lines of words [Parnas 1972]. The first is structured around the control, one module handles reading, the next handles processing etc. The modules are called in order to accomplish the task. The second structure is based on abstract data types. One module handles words and how to access and modify them. The other modules use this module to manipulate the data. This is a superior design because the components can work in isolation and any changes in the requirements affect fewer modules. This lesson carries forward directly to language implementations, but it also carries forward in another way. The control structured solution is similar to a syntax linked modular structure. Indeed, attribute grammars are devoid of control except for the selection of the attribute rules depending on the particular syntax. What is the analogue of Parnas's data type structured solution? I suggest that it is the module which accepts a liberal variety of syntactic constructs, selecting on the attributes provided rather than the syntactic forms. In this way the module is insulated from changes in the syntax.

Incremental algorithms. Research into incremental algorithms for performing language related tasks is popular. The usual motivation is that interactive environments require rapid feedback to small changes. Incremental algorithms work by reusing results of a computation. There are two obvious candidates for incremental algorithms. Consider the language processor L defined by syntax modules G_i and attribute grammar modules M_i:

$$L = (G_1 \oplus G_2 \oplus \cdots \oplus G_n) \; + \; (M_1 \oplus M_2 \oplus \cdots \oplus M_m)$$

The two opportunities are:

- a change in a G_i or the addition of G_{n+1}
- a change in an M_i or the addition of M_{m+1}

Incremental algorithms to perform these changes would make an interactive language definition workbench possible, and allow small changes to be made to large existing languages very efficiently. Incremental parser generators are already available, e.g. [Heering *et al.* 1989b] and [Horspool 1990].

Object oriented programming. The relationship between attribute grammar modules and object oriented programming deserves more attention. It might be possible to integrate the two. Koskimies [Koskimies 1988] has made progress in this area but the solutions are rigidly based on the concrete syntax. It appears that the introduction of semantic modules into this framework would be difficult, but the result would be worth-while.

7.6 On what has been achieved

Language oriented programming is a software engineering paradigm in which special purpose application oriented languages are constructed as a tool for giving succinct and powerful descriptions of the application. The application oriented language is an abstraction of the problem domain which exposes the salient details to the application and hides other details. For language oriented programming to be practicable it must be possible to construct language processors cheaply. A recognised key to reducing software costs is re-use.

Conventionally structured language processors have poor reuse properties, especially when one considers the kind of reuse that is most useful to language oriented programming: in importation of 'ready made' language fragments like arithmetic or imperative control flow.

Modular attribute grammars allow the implementation to be structured according to the language features. A module or collection of modules can capture the essence of a language fragment. This allows modules to be created which have the property that they can be reused in new implementations and new languages. Modular grammars overcome the restrictions of the conventional language processor structure.

New languages can be constructed by assembling existing language fragments. This greatly reduces the cost of building new languages as only a small proportion of the implementation has to be built from scratch.

Variant languages can be defined by adding new modules that specify an alternative implementation for parts of the language. The variant reuses much of the original implementation. Object oriented programming is a success because classes can be reused by creating a subclass that uses inheritance to get most of the functionality and using new definitions to tailor the class to the particular requirements. Similarly, the ability of attribute grammar modules to redefine almost any part of the language is important for reuse as it is difficult to guess in advance how a module may be reused.

The methodology used in this investigation is sound. Lazy functional programming was used as a tool to expose a method for calculating attributes defined by a composition of attribute grammar modules. There is a correspondence between attribute grammars and functional programs which allows an attribute grammar to be translated into a functional program. This correspondence was manipulated to put it in a form which abstracted the attribute grammar. By program transformation an executable modular attribute grammar processor was derived which served as a prototype attribute grammar processor allowing all of the attribute grammar programs in this report to be executed. The use of such a prototype gives weight to the argument that language fragments are a suitable structure for language processors.

In summary, the main contributions are

- A method of modelling languages as collections of modular language fragments.
- A prototype tool for implementing language processors from modules.
- A description of the language oriented programming software engineering methodology based on creating application oriented languages as abstractions of the problem domain.

Language fragments, as embodied in the modular attribute grammars, is superior to the conventional phase oriented structure for implementing language processors. The structure is superior because it has better software engineering properties which will reduce the cost of building language implementations. This in turn will reduce the cost of language oriented programming.

A CCS inference rules

As has been described in the main part of this report, a CCS agent (program) is executed by converting it to standard concurrent form (SCF) and then choosing one of the terms of the SCF expansion. This note describes an algorithm based on transitional semantics for deriving the SCF of a CCS agent.

A.1 Inference rules

The equivalence of CCS agents is defined by a set of inference rules. These are given by Milner [1980]and are reproduced here:

$$\textbf{Act}\ \frac{}{\alpha.P \xrightarrow{\alpha} P}$$

$$\textbf{Sum}_1\ \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \qquad \textbf{Sum}_2\ \frac{Q \xrightarrow{\alpha} Q'}{P + Q \xrightarrow{\alpha} Q'}$$

$$\textbf{Com}_1\ \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \qquad \textbf{Com}_2\ \frac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P \mid Q'}$$

$$\textbf{Com}_3\ \frac{P \xrightarrow{\alpha} P' \qquad Q \xrightarrow{\overline{\alpha}} A'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

$$\textbf{Rel}\ \frac{P \xrightarrow{\alpha} P'}{P[f] \xrightarrow{f(\alpha)} P'[f]} \qquad \textbf{Res}\ \frac{P \xrightarrow{\alpha} P'}{P \backslash L \xrightarrow{\alpha} P' \backslash L}\ (\alpha, \overline{\alpha} \notin L)$$

$$\textbf{Con}\ \frac{P \xrightarrow{\alpha} P'}{A \xrightarrow{\alpha} P'}\ (A \stackrel{\text{def}}{=} P)$$

Note that there is no rule for *Nil* because no transitions can be infered from the null agent.

A.2 Conversion to SCF

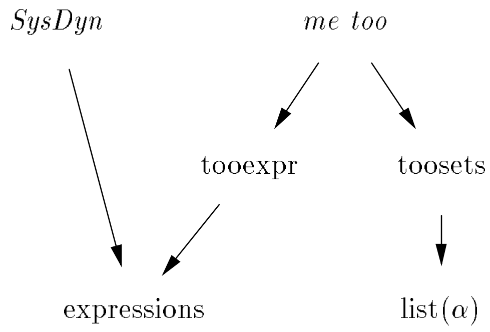
The SCF of an agent is constructed by the (expensive but obviously correct) method of enumerating all the possible inferences. This is done in a bottom-up manner. All of the possible inferences of the sub-agents of an agent are combined according to the inference rules. Since we are asking "what transitions are possible from agent P?" we write the answers not as the transitions $\{P \xrightarrow{\alpha_1} Q_1, P \xrightarrow{\alpha_2} Q_2, \ldots, P \xrightarrow{\alpha_n} Q_n\}$ but as the equivalent SCF agent $\alpha_1.Q_1 + \alpha_2.Q_2 + \cdots + \alpha_n.Q_n$ which is represented as the set of pairs $\{(\alpha_1, Q_1), (\alpha_2, Q_2), \ldots, (\alpha_n, Q_n)\}$. The set of all the valid inferences can be deduced inductively.

$$
\begin{array}{lcl}
scf(Nil) & = & \{\} \\
scf(\alpha.P) & = & \{(\alpha, P)\} \\
scf(P+Q) & = & scf(P)\cup \\
 & & scf(Q) \\
scf(P \mid Q) & = & \{(\alpha, P' \mid Q) \mid (\alpha, P') \in scf(P)\}\cup \\
 & & \{(\alpha, P \mid Q') \mid (\alpha, Q') \in scf(Q)\}\cup \\
 & & \{(\tau, P' \mid Q') \mid (\alpha, P') \in scf(P) \wedge (\overline{\alpha}, Q) \in scf(Q)\} \\
scf(P\backslash L) & = & \{(\alpha, P'\backslash L) \mid (\alpha, P') \in scf(P) \wedge \alpha \notin L\} \\
scf(P[f]) & = & \{(f(\alpha), P'[f]) \mid (\alpha, P') \in scf(P)\}
\end{array}
$$

B ***me too*** and ***SysDyn*** syntax modules

B.1 Summary

This diagram summarizes the use relationship between the syntax modules. An arrow points from the importing module to the imported module.



B.2 expr

```
#language grammar grammar grammar->lisp

grammar expressions .

import list exprlist expr .

expr -> expr relop simplexpr .
expr -> simplexpr .

simplexpr -> simplexpr addop term .
simplexpr -> term .

term -> term mulop factor .
term -> factor .

factor -> "if" expr "then" expr "else" expr .
factor -> "(" expr ")" .
factor -> unop factor . #
factor -> var "(" exprlist ")" .
factor -> var .
factor -> constant .
```

```
relop -> "=" .
relop -> "<" .
relop -> "<=" .
relop -> "<>" .
relop -> ">" .
relop -> ">=" .

addop -> "+" .
addop -> "-" .
addop -> "or" .
addop -> "^" . #sequence concatenation

mulop -> "*" .
mulop -> "/" .
mulop -> "mod" .
mulop -> "and" .

unop -> "-" .
unop -> "not" .
```

B.3 list

```
#language grammar grammar  grammar->lisp

grammar list (thinglist thing) .

thinglist -> thing "," thinglist .
thinglist -> thing .
thinglist -> .
```

B.4 tooexpr

```
#language grammar grammar grammar->lisp

grammar tooexpr .

import expressions .

factor -> "(" expr ")" .
factor -> "(" exprlist ")" .

factor -> "let" defnlist expr .

defnlist -> defn defnlist .
```

```
defnlist -> .

defn -> var "=" expr .
defn -> var "(" varlist ")" "=" expr . #local function

relop -> "->" .
```

B.5 toosets

```
#l grammar grammar grammar->lisp

grammar toosets .

import list varlist var .
import list elementlist elementspec .

factor -> setexpr .

setexpr -> "{" expr "|" generator moregenerators "}" .
setexpr -> "{" elementlist "}" .
setexpr -> "exists" pattern ":" expr expr .
setexpr -> "all" pattern ":" expr expr .

generator -> pattern ":" expr .

moregenerators -> ";" generator moregenerators .
moregenerators -> ";" filter moregenerators .
moregenerators -> .

filter -> expr .

elementspec -> expr ".." expr .
elementspec -> expr .

pattern -> "(" varlist ")" .
pattern -> var .

relop -> "in" .
addop -> "union" .
mulop -> "intersect" .
```

B.6 too

```
#l grammar grammar grammar->lisp

grammar too .
```

```
axiom defns .

terminal ident string number .
comments shellCommentLexer .

import tooexpr .
import toosets .

program -> stats .
stats -> stat "." stats .
stats -> stat "." .

stat -> var "=" expr .
stat -> "type" var "=" expr .
stat -> var ":" expr .
stat -> var "(" varlist ")" "=" expr .
stat -> expr .

constant -> number .
constant -> string .
constant -> "'" ident .

var -> ident .
```

B.7 sysdyn

```
#language grammar grammar grammar->lisp

grammar sysdyn .

import expressions .

terminal ident number .
comments shellCommentLexer .

model -> "model" ident parts .

parts -> part parts .
parts -> .

part -> "level" var "initial" expr "then" expr .
part -> "rate" var "initial" expr "then" expr .
part -> var "=" expr .

var -> ident .
constant -> number .
```

Bibliography

[Aasa *et al.* 1988] Annika Aasa, Kent Petersson & Dan Synek. "Concrete Syntax for Data Objects in Functional Languages", *in LISP and Functional Programming*, pp. 96–105.

[Abelson & Sussman 1985] Harold Abelson & Gerald Jay Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, Massachusetts.

[Aho & Ullman 1973] A. V. Aho & J. D. Ullman. *The Theory of Parsing, Translation, and Compiling.* Prentice-Hall.

[Aho *et al.* 1986] A. V. Aho, R. Sethi & J. D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley.

[Aksit *et al.* 1990] Mehmet Aksit, René Mostert & Boudewijn Haverkort. "Compiler Generation Based on Grammar Inheritance", Technical Report 90–07, University of Twente.

[Bergstra *et al.* 1989] J. A. Bergstra, J. Heering & P. Klint, (eds.). *Algebraic Specification.* Addison-Wesley.

[Bobrow *et al.* 1988] D. G. Bobrow, L. G. De Michiel, R. P. Gabriel, S. E. Keene, G. Kiczales & D. A. Moon. "Common Lisp Object System Specification", *SIGPLAN Notices*, 23 (9).

[Bondorf 1990] Anders Bondorf. "Automatic Autoprojection of Higher Order Recursive Equations", *in* N. D. Jones, (ed.), *3rd European Symposium on Programming*, Lecture Notes in Computer Science 432, pp. 70–87. IEEE, Springer-Verlag.

[Cameron & Ito 1984] Robert D. Cameron & M. Robert Ito. "Grammar-Based Definition of Metaprogramming Systems", *ACM Transactions on Programming Languages and Systems*, 6 (1): 20–54.

[Chirica & Martin 1979] Laurian M. Chirica & David F. Martin. "An Order-Algebraic Definition of Knuthian Semantics", *Mathematical Systems Theory*, 13 (1): 1–27.

[Cordy *et al.* 1988] James R. Cordy, Charles D. Halpern & Eric Promislow. "TXL: A Rapid Prototyping System for Programming Language Dialects", *in 1988 International Conference on Computer Languages*, pp. 280–285. IEEE.

[Cousineau & Huet 1989] Guy Cousineau & Gérard Huet. "The CAML Primer", Technical report, INRIA-Ens.

[Cousineau *et al.* 1985] G. Cousineau, P. L. Curien & M. Mauny. "The Categorical Abstract Machine", *in* J. P. Jouannaud, (ed.), *IFIP Intl. Conf. on Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science 201, pp. 50–64. Springer-Verlag.

[Dueck & Cormack 1990] G. D. P. Dueck & G. V. Cormack. "Modular Attribute Grammars", *The Computer Journal*, 33 (2): 164–172.

[Earley 1970] J. Earley. "An Efficient Context-free Parsing Algorithm", *Communications of the ACM*, 13 (2): 94–102.

[Farrow 1982] Rodney Farrow. "LINGUIST-86: Yet Another Translator Writing System Based On Attribute Grammars", *in SIGPLAN Notices, Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pp. 160–171. ACM.

[Field & Harrison 1988] Anthony J. Field & Peter G. Harrison. *Functional Programming.* Addison-Wesley.

[FPCA89] *Functional Programming Languages and Computer Architecture.* ACM.

[Frost 1990] R. A. Frost. "Towards a Calculus of Lazy Interpreters", *in ISLIP 90.* ACM.

[Ganzinger & Giegerich 1984] Harald Ganzinger & Robert Giegerich. "Attribute Coupled Grammars", *in Proceedings of the SIGPLAN '84 Symposium on Compiler Construction, SIGPLAN Notices.* ACM.

[Giegerich 1988] Robert Giegerich. "Composition and Evaluation of Attribute Coupled Grammars", *Acta Informatica*, 25: 355–423.

[Goguen & Winker 1988] Joseph A. Goguen & Timothy Winker. "Introducing OBJ3", Technical Report SRI-CSL-88-9, SRI International.

[Heering *et al.* 1987] J. Heering, P. Klint & J. Rekers. "Incremental Generation of Lexical Scanners", Technical Report CS-R8761, Centrum voor Wiskunde en Informatica (CWI), Amsterdam.

[Heering *et al.* 1989a] J. Heering, P. R. H. Hendriks, P. Klint & J. Rekers. "The Syntax Definition Formalism SDF—Reference Manual", Technical Report CS-R8926, Centrum voor Wiskunde en Informatica (CWI), Amsterdam.

[Heering *et al.* 1989b] J. Heering, P. Klint & J. Rekers. "Incremental Generation of Parsers", *in Proc. SIGPLAN '89 Conference on Programming Language Design and Implementation*, pp. 179–191.

[Henderson 1980] Peter Henderson. *Functional Programming, Application and Implementation.* Prentice Hall.

[Henderson 1986] Peter Henderson. "Functional Programming, Formal Specification and Rapid Prototyping", *IEEE Transactions on Software Engineering*, SE-12 (2).

[Henson & Turner 1982] Martin C. Henson & Raymond Turner. "Completion Semantics and Interpreter Generation", *in Ninth Annual ACM Symposium on Principles of Programming Languages*, pp. 242–254.

[Hoare 1985] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall.

[Holt *et al.* 1988] Richard C. Holt *et al.* *The Turing programming language: design and definition.* Prentice Hall.

[Horspool 1990] R. Nigel Horspool. "Incremental Generation of LR Parsers", *Computer Languages*, 15 (4): 205–223.

[Johnson 1975] S. C. Johnson. "Yacc—Yet Another Compiler-Compiler", Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, N. J.

[Johnsson 1987] Thomas Johnsson. "Attribute Grammars as a Functional Programming Paradigm". *in Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science 274. Springer-Verlag.

[Kastens *et al.* 1982] U. Kastens, E. Zimmermann & B. Hut. "GAG—a practical Compiler Generator". Lecture Notes in Computer Science 141. Springer-Verlag.

[Kernighan & Ritchie 1978] Brian W. Kernighan & Dennis M. Ritchie. *The C programming language.* Prentice Hall.

[Knuth 1968] D. E. Knuth. "Semantics of context-free languages", *Math. Syst. Theory*, 2 (2): 127–145. Correction in Math. Syst. Theory 5(1), 95-96 (1971).

[Koskimies 1988] Kai Koskimies. "Software Engineering Aspects in Language Implementation", *in* D. Hammer, (ed.), *Compiler Compilers and High Speed Compilation*, Lecture Notes in Computer Science 371, pp. 39–51. Springer-Verlag.

[Koskimies 1990] Kai Koskimies. "Lazy Recursive Descent Parsing for Modular Language Implementation", *Software—Practice and Experience*, 20 (8): 749–772.

[Kreutzer 1986] Wolfgang Kreutzer. *System Simulation Programming Styles and Languages.* Addison-Wesley.

[Lajos 1990] Gyorgy Lajos. "Language Directed Programming in Meta-Lisp", *in 1st European Conference on the Practical Applications of LISP*, pp. 223–233.

[Lesk 1975] M. E. Lesk. "Lex—a lexical analyzer generator", Computing Science Technical Report 39, AT&T Bell Laboratories, Murray Hill, N. J.

[Mamrak *et al.* 1985] Sandra A. Mamrak, Michael J. Kaelbling, Charles K. Nicholas & Michael Share. "Chameleon: A System for Solving the Data-Translation Problem", *IEEE Transactions on Software Engineering*, 15 (9): 1090–1108.

[Mauny 1989] Michel Mauny. "Parsers and Printers as Stream Destructors and Constructors Embedded in Functional Languages", *in Functional Programming Languages and Computer Architecture*, pp. 360–370. ACM.

[McCarthy *et al.* 1965] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart & Michael I. Levin. *LISP 1.5 Programmers's Manual, (second ed.).* MIT Press.

[Milner 1980] Robin Milner. *A Calculus of Communicating Systems.* Lecture Notes in Computer Science, Vol. 92, Springer-Verlag.

[Milner 1989] Robin Milner. *Communication and Concurrency.* Prentice Hall.

[Mogensen 1989a] Torben Mogensen. *Binding Time Aspects of Partial Evaluation.* PhD thesis, DIKU, University of Copenhagen, Denmark.

[Mogensen 1989b] Torben Mogensen. "Separating Binding Times in Language Specifications", *in Functional Programming Languages and Computer Architecture*, pp. 12–25. ACM.

[Noonan 1985] Robert E. Noonan. "An Algorithm for Generating Abstract Syntax Trees", *Computer Languages*, 10 (3/4): 225–236.

[Parnas 1972] David L. Parnas. "On the Criteria To Be Used in Decomposing Systems into Modules", *Communications of the ACM*, 15 (12): 1053–1058.

[Purtilo & Callahan 1989] James J. Purtilo & John R. Callahan. "Parse-Tree Annotations", *Communications of the ACM*, 32 (12): 1467–1477.

[Rees & Clinger 1986] Jonathan Rees & William Clinger. "Revised^3 Report on the Algorithmic Language Scheme", *SIGPLAN Notices*, 21 (12): 37–79.

[Rekers 1989] J. G. Rekers. "Modular Parser Generation", Technical Report CS-R8933, Centrum voor Wiskunde en Informatica (CWI), Amsterdam.

[Spivey 1989] J. M. Spivey. *The Z Notation.* Prentice Hall.

[Stroustrup 1986] Bjarne Stroustrup. *The C++ programming language.* Addison-Wesley.

[Tomita 1980] Masaru Tomita. *Efficient Parsing for Natural Language.* Kluwer Academic Publishers.

[Vogt *et al.* 1989] H. H. Vogt, S. D. Swierstra & M. F. Kuiper. "Higher Order Attribute Grammars", *in Proc. SIGPLAN '89 Conference on Programming Language Design and Implementation*, pp. 131–145.

[Waddle 1990] Vance E. Waddle. "Production Trees: A Compact Representation of Parsed Programs", *ACM Transactions on Programming Languages and Systems*, 12 (1): 61–83.

[Wadler 1987] Philip Wadler. "Views: A way for pattern matching to cohabit with data abstraction", *in 14th Annual ACM Symposium on Principles of Programming Languages*, pp. 307–313.

[Wadler 1990] Philip Wadler. "Comprehending Monads", *in Proceedings, Lisp and Functional Programming.* ACM.

[Weise & Ruf 1988] Daniel Weise & Erik Ruf. "Computing Types During Program Specialization", Technical Report TR-441, draft, Center for Integrated Systems, Stanford.

[Yellin & Mueckstein 1986] Daniel M. Yellin & Eva-Maria M. Mueckstein. "The Automatic Inversion of Attribute Grammars", *IEEE Transactions on Software Engineering*, SE-12 (5): 590–599.