The Dissertation Committee for Donovan Michael Kolbly

Certifies that this is the approved version of the following dissertation:

Extensible Language Implementation

Committee:

Gordon Novak, Supervisor

Don Batory

Don Fussell

Calvin Lin

Robert Strandh

Extensible Language Implementation

by

Donovan Michael Kolbly, B.S.; M.S.

Dissertation

Presented to the Faculty of the Graduate School of

the University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 2002

Extensible Language Implementation

Publication No. ______________

Donovan Michael Kolbly, Ph.D.
The University of Texas at Austin, 2002

Supervisor: Gordon Novak

This work presents several new approaches to the construction of extensible languages, and is the first system to combine local, dynamically extensible context-free syntax with the expressive power of meta-level procedures. The unifying theme of our system is that meaning should be computed relative to local context.

We show how this theme is manifest in an implementation of a Scheme macro system which achieves hygienic macro expansion without rewriting. Additionally, our Scheme macro system makes available compile-time meta-objects for additional power in writing macros; macros that pattern match on compile-time types for optimization at macro-processing time are one example. This approach is currently in use in our RScheme implementation of Scheme.

We also show the how this approach is applied to languages with conventional syntax, using Java as an example. We present a dynamically extensible parser based on the Earley parsing algorithm. This approach is practical as well as flexible; a straightforward implementation in C parses a 600-line (2777 token) file in about 44ms on an 866MHz Pentium III.

We also describe a language extension framework that makes possible an extensible variant of Java, in which new syntax can be supplied by the casual programmer with only limited knowledge of the underlying compiler implementation or approach. This finally makes available to Java programmers the easy access to structured macro facilities that Lisp programmers find so powerful. Finally, we demonstrate this framework by constructing a deterministic finite automaton language extension to Java.

Contents

Abstract **iv**

Chapter 1 Introduction **1**

1.1 Motivation 1

1.1.1 Our Contribution 2

1.2 Modularity, Reusability, and Extensibility 3

1.3 Syntax-Directed Translation 7

1.4 Summary of Basic Approach 8

1.5 Scope of Work 8

Chapter 2 Macro Systems **10**

2.1 Introduction 10

2.2 Background 11

2.2.1 C Macros 12

2.2.2 Lisp Macros 20

2.2.3 Scheme Macros 25

2.2.4 Systems Related to Macros 27

2.3 A Taxonomy of Scheme Macros 29

2.3.1 Call-by-name Inline Procedures 29

2.3.2 Advertent Capture 30
2.3.3 Explicit Intentional Capture 31
2.4 Our Contribution 32
2.4.1 RScheme Macros 32
2.4.2 Type Reflective Macros 33
2.5 Our Implementation 33
2.5.1 Operation 34
2.5.2 Reflection Operators 46

Chapter 3 Extensible Parsing 50

3.1 Introduction 50
3.2 Background 53
3.2.1 What We Would Like to Do 53
3.2.2 Granularity of Grammar Changes 54
3.2.3 Applications of Our Approach 54
3.2.4 Limitations 55
3.3 Contour Sensitivity 56
3.3.1 Major Styles of Environment Passing 56
3.3.2 General Mechanism 57
3.4 Implementing Contour Sensitivity 58
3.4.1 Frequent Grammar Changes 58
3.4.2 Simple Interpretation 60
3.5 Implementation of Continuation Passing Parser 61
3.5.1 General Description 61
3.5.2 Representation of Rules 62
3.5.3 Dynamic Rule Compilation 65

3.5.4 Major Styles in Terms of Mechanism 65

3.5.5 Examples 67

3.5.6 Performance 71

Chapter 4 Extensible Earley Parsing 72

4.1 Introduction 72

4.2 Description of Earley Parsing 72

4.2.1 States 74

4.2.2 State Sets 74

4.2.3 Initial Conditions 75

4.2.4 Processing 75

4.2.5 Prediction 77

4.2.6 Scanning 78

4.2.7 Completion 79

4.2.8 Relationship to Tomita Parsing 80

4.3 Advantages to the Earley Approach 81

4.3.1 Flexibility 81

4.3.2 Extensibility 82

4.3.3 Understandability 84

4.3.4 Complexity 85

4.4 Drawbacks to the Earley Approach 85

4.4.1 Expressiveness 85

4.4.2 Performance 86

4.5 Extensibility 87

4.5.1 Scope Issues 87

4.6 Our Implementation 89

4.6.1 Details 89

4.6.2 Meta-syntax 90

4.6.3 Meaning Computation 92

4.6.4 Performance 95

4.7 Literal Equivalence 97

4.8 Improvements to Basic Earley 101

4.8.1 Conflict resolution 101

4.8.2 Pruning states using FIRST 102

4.8.3 Approximating FIRST 102

Chapter 5 Compiler Extension Framework **104**

5.1 Capabilities of Extension Framework 104

5.1.1 Declarative, Pattern-Based Transformation 105

5.1.2 Pattern Matching Synthesized Attributes 105

5.1.3 Procedural Code-Production Mechanisms 106

5.2 Elements of an Extension Framework 106

5.3 Implementation 107

5.3.1 Meta-language 107

5.3.2 Syntax Evaluation 109

5.3.3 Recursive Compilation 110

5.3.4 Pattern Variables 111

5.3.5 Local Grammar Changes 114

5.3.6 In-line Computation 116

5.4 Declarative Transformations 116

5.5 Synthesized Attributes 117

5.6 Procedural Code Production 119

5.6.1 Token Sequences 119
5.6.2 Compilation 119
5.6.3 Environments and Syntax 119
5.6.4 Reflection 120
5.7 Modular Syntax 120
5.8 Full Meta-syntax 121
5.8.1 Syntax Declarations 123
5.8.2 Syntax Rules 124
5.8.3 Syntax Pattern Elements 124
5.8.4 Actions 127
5.8.5 Local Variables 132
5.8.6 Inline Actions 132
5.8.7 Example 133
5.9 Issues and Future Work 135
5.9.1 Substitution Conformance 135
5.9.2 Translation Recursion 136
5.9.3 Meta-syntax Scope 137
5.9.4 Syntax Module Templates 137

Chapter 6 An Application of an Extensible Language 138

6.1 Introduction 138
6.2 Implementation Approach 140
6.2.1 Declaring the Extension 141
6.2.2 Building the Final Meaning 142
6.2.3 Declaring the State-Keeping Variable 145
6.2.4 Declaring the Java Class's Entry Method 146

6.2.5 Declaring the Java Class's Accessor Methods 147
6.2.6 States Within an Automaton 147
6.2.7 Building the State Switcher 151
6.2.8 Declaring transitions 152
6.2.9 Symbolic State Names 153
6.3 Example Use of the DFA Extension 155
6.3.1 Sample Extended-Java File 155
6.3.2 Generated class definition 155
6.3.3 Generated process() method 157

Chapter 7 Final Words 159

7.1 Related Work 159
7.1.1 Syntactic Exposures 159
7.1.2 Term Rewriting 159
7.1.3 Hygienic Macro Expansion 160
7.1.4 Reusable Generative Programming 161
7.1.5 Adaptable Grammars 161
7.1.6 Open Compilers and MOPs 162
7.2 Limitations and Future Work 165
7.2.1 Meta-Object Protocol 165
7.2.2 Error Reporting 165
7.2.3 Synthesized Attributes 166
7.3 Conclusions 169

Bibliography 171

Vita 177

Chapter 1

Introduction

The advantages of extensible languages have long been realized by the Lisp community. The ability to easily adjust the language to fit the application, rather than to always adjust the application to fit the language, is at the heart of what Lisp programmers consider the deep power of Lisp [17, 33]. In this work, we show how that power can be made more accessible and powerful even in Scheme, as well as available to programmers in languages with conventional syntax such as Java. The tree-structured transformations of Lisp macros integrated with an extensible parser allow the concepts to be unified. In addition, a well-structured compiler meta-object protocol exposes relevant aspects of the compilation process and provides powerful programmable hooks for extending the language to fit the application.

1.1 Motivation

This work is motivated by the fact that much existing work in extensible languages is either insufficiently expressive in the kinds of extensions that are permitted (*i.e.*, function libraries) or expressed at the wrong level or in the wrong ways (*e.g.*, purely procedural transformations operating over text strings).

That is, we are primarily motivated by:

- Ease of language implementation
- Ease of language extension
- Ease of re-engineering language implementation (*e.g.*, to change performance tradeoffs to deal with new technologies or new usage patterns)

We recognize the following themes:

- Extensible languages should have extensible compilers.
- Meaning should be expressed naturally through context, especially through *contour sensitive* contexts, which preserve lexical scoping throughout transformation.
- An extensible compiler should have a friendly interface and be integrated with languages using conventional grammars.
- Objects used during the front-end processing of a program should be reified, and be the domain objects of meta-programming.

1.1.1 Our Contribution

With these themes and motivations in mind, in this work we describe an approach that provides:

- Context-local and dynamically modifiable concrete syntax,

- Full context-free syntactic power, and
- A means for defining procedural meta-level code for arbitrary computation at compile-time.

Furthermore, we illustrate this approach in a system with Java as a base language.

1.2 Modularity, Reusability, and Extensibility

The primary means of creating large and complex software systems has been by building relatively simple *program modules*, and composing those modules into larger, more complex software systems. The process for developing the large software system can then be decomposed into the development of smaller program modules. Smaller program modules are easier to understand, develop, and test, and well-constructed modules can be reused to build other software applications.

A refined use of modularization is for *program layering.* Program layering arranges modules into layers whose role is to transform the program concepts at a higher level of abstraction to those at a lower level of abstraction. A layer is then *extending* the language below the layer, creating a new (possibly superset) language.

Figure 1.1 shows how we draw the relationship between the language L0 below the layer, the module layer, and the language created by the layer, L1.

Layering is a powerful structuring tool, and has been used in systems from the Basic Linear Algebra Subroutines to the 7-layered OSI protocol stack

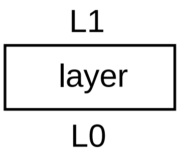


Figure 1.1: A simple layered program

for network communication. The modularity provided by well-architected layering leads to highly effective reuse of modules.

If Figure 1.1 is drawn on its side and the layer module is thought of as a process, an approach to language extension known as *preprocessing* is revealed:

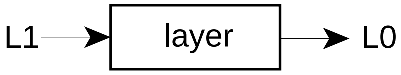


Figure 1.2: A preprocessing approach

In this approach, language L1 is consumed by the layer process, generating an appropriate program in L0 as output. The language L1 is said to be preprocessed into L0 because the eventual execution of the original program is done in an L0 execution environment.

Be it layering or preprocessing, the manner in which this modularity is expressed can be divided into two broad categories. The first category is extension within the language, in which the language itself contains the means to create or modify layers. The second category is extension from outside the language, which requires the separate implementation of a language translator, and is restricted to expressing modules using the preprocessing approach.

Extension from within the language is often accomplished via *program* *libraries.* This technique is so well known – since the days of Fortran, at least – that it is not given a second thought. A language without the ability to be extended via function libraries is immediately recognized as unsuitable for serious programming.

Large and complex software systems have been built using the program library technique. Some libraries are so large and complex that sophisticated systems are required just in order to compose their constituents (*e.g.*, see the work on the composition of astronomical library components by Stickel and Waldinger [37]). Other libraries are really procedural interfaces to a specialized language environment. For example, RenderMan [41] and OpenGL [44] both maintain a complex virtual machine which is managed by a library of API calls that encode the constructs and primitives of the language, including nesting and conditionals. Still other libraries vastly extend the power and usefulness of the base language (*e.g.*, the Java core API and the C++ Standard Template Library).

These kinds of libraries are mostly collections of procedural abstractions. The C++ STL is a notable exception in this respect; it represents a library of programming templates which are instantiated at compile-time to create the application.

Procedural abstraction libraries are limited in their expressiveness in most languages. For example, the evaluation of arguments is usually defined to take place before a function is called. Call-by-value semantics for argument passing make it impossible to implement a conditional construct such as `if` using a function library style extension.

Another problem with procedural abstractions is that the syntax of function calls is usually constrained by the language. For example, most

Fortran-derived languages adopt the notation from mathematics of the function name followed by, in parentheses, a comma-separated list of arguments. Programmers of such languages would immediately recognize the example:

```
atan2( y, x )
```

as a call to a function named `atan2` with two arguments, `y` and `x`.

Sometimes, language extensions are in the form of *macros*. These have the added expressiveness of being able to define special forms, but macro semantics are usually that of textual substitution. Many macro implementations give little or no thought to preserving the relevant expression context (*i.e.*, the context at the point of macro definition for the purpose of evaluating the macro body, and the context at the point of macro use for the purpose of evaluating the macro arguments.)

In more severe cases, language extensions have taken the form of *preprocessors*. C++ was originally implemented this way, as a preprocessor for transforming C++ into C[1]. Sometimes the preprocessor is itself extensible.

The C preprocessor and C++ both are extensible preprocessors in the sense that the language they provide can be extended by the input program. In the language of C macros, this comes in the form of the `#define` directive. In C++, a class definition adds to the preprocessor's internal structures representing the program being transformed and thereby constitutes an extension. In both cases, the preprocessor defines, as part of the language it provides, an extension framework. The eventual output of the C preprocessor has all macros removed, and corresponds to the plain C code. The C preprocessor is such a general internally-extensible filter, tied only loosely to the C language,

[1] Since the original implementation, however, the C++ language has become complicated enough to warrant its own compiler.

that it has been used as an extensibility mechanism for other languages, such as IDL [29].

Like the C preprocessor, the C++ translator generates plain C code as its output. The translation required to provide the C++ language is quite a bit more sophisticated, and involves building complex internal data structures, applying internal transformations on parse trees, and unparsing the result into textual C code for consumption by the plain C compiler. C++ supports relatively sophisticated language extensions in the form of *templates*, which essentially support abstractions over types[2].

Preprocessing can also be regarded as *language implementation by transformation.* If the input language and output language are sufficiently different, it is often regarded this way (the line becomes somewhat blurry when the input language is a strict superset of the output language).

1.3 Syntax-Directed Translation

Another major force in much of the development of language systems has been the notion of *syntax-directed translation.* In this metaphor, the syntactic structures of the language correspond to semantic structures. As a result, the translation of a program, or its *assignment of meaning*, can be embedded in the parsing process.

Lisp has never been saddled with the notion of syntax-directed translation, instead basing translation on the top-down processing of a simple parse tree. This gives Lisp programmers a more powerful means of expressing trans-

[2] As opposed to functions, which abstract over values. That is, a function computes a new value by being instantiated with a set of values as its arguments. A template creates a new construct by being instantiated with types as arguments.

formations, because they can operate on trees instead of streams of tokens.

1.4 Summary of Basic Approach

Our basic approach is to compile expressions in environments. A compiler normally makes use of some kind of compile-time environment to keep track of the meaning of identifiers. This can be in the form of a global symbol table or, more typically, a structured representation that reflects the nested structure of the program.

Our approach, then, is to extend the use of this sort of symbol table information to permit the effective interpretation of macros in their appropriate contexts. As we show, this approach allows the macros themselves to reflect (examine and operate upon) these contexts, which makes possible new kinds of expressive power and convenience [39].

This approach is extended to traditional language syntax using an interpretive parsing approach. In the parsing interpreter, the grammar becomes a part of the compile-time context, to be manipulated to implement the desired context-dependent aspect of the extensible syntax.

1.5 Scope of Work

There are three distinct but related aspects of this new approach to front-end compilation:

- Avoiding common problems with traditional macros (*e.g.*, providing hygienic macro processing).

- Extending the power of Scheme-style macros, exposing the operation of the front-end and allowing the programmer to intervene in a structured, timely manner.

- Supporting conventional grammars.

These aspects provide the structure for the rest of this discussion, and are the main points of this work.

The issue of providing syntactic extensions in a Lisp-like language (*i.e.*, Scheme) involves an approach where macro processing is fully integrated with normal compiler processing. Integrating macro processing with the compiler gives macros both efficiency and power. This approach is efficient because no rewriting of the source program need be done, in contrast to traditional macro processing which does rewrite the source program into a "macro expanded" program. Additional power for macros is derived from the use of compile-time protocols to allow macros access to the information that is normally available at compile-time. No additional work is required to make such information available – the compiler typically needs the information anyway – but because macro processing is integrated with compilation, the macro system can access that information in a convenient manner.

Languages with conventional surface grammars such as Java and C also need mechanisms for syntactic extensibility. Chapter 3 shows how an *extensible parser* can be constructed that applies the general principle of context-sensitive processing to the parsing problem for a very simple parser. Chapter 4 applies the extensible parsing approach to a more sophisticated parser based on the Earley parsing algorithm.

Chapter 2

Macro Systems

2.1 Introduction

In this chapter, we describe a kind of language extension mechanism that is capable of expressing new constructs with much more flexibility than conventional library mechanisms support. The kind of language extension mechanism we address is that of *macro systems*. A macro system is a language facility that allows the user to write localized transformations (macros) that can be applied to the source program before later phases of the program processing system attempt to assign a meaning to the source program.

We first discuss the very different macro systems available in the C language and the Lisp family of languages, including Common Lisp and Scheme. We then provide a tentative taxonomy of Scheme macros – a catalog of common usage patterns adapted from the work of Carl [10].

Then we elaborate upon our approach, which integrates macro processing with compilation. That is, macros are elaborated in a computation interleaved with the normal work of compiling, instead of operating as a prepass. We show how that can lead to some additional expressive capabilities for macro writers because some compile-time information is available at macro processing time.

2.2 Background

As we indicated in the previous chapter, the procedural abstraction is a primary means of building modular program subsystems. However, many languages constrain the form and semantics of procedure calls. For example, a procedure call form typically evaluates all of its arguments exactly once before invoking the called procedure. In some cases, the common semantics of a procedure call are insufficient to express the desired construct.

For example, if the application makes repeated use of an operation like *swap*, it makes sense to encapsulate the functionality within an abstraction so it can be reused effectively. However, the swap operation does not lend itself to a procedural abstraction because both of its arguments are to be interpreted as both *lvalues* and *rvalues* – that is, as locations suitable for being stored into (*lvalues*) as well as locations to be read (*rvalues*)[1] [38]. Hence, a fixed function calling protocol like call-by-value breaks down for operations like *swap*.

Typically, because the function calling protocol is fixed in a language, procedural abstractions cannot implement *special forms*. Special forms are program constructs which have special rules for evaluating their constituent

[1] The use of an argument as both an lvalue and an rvalue can be dealt with in languages with explicit pointers by simply passing a pointer to the locations by value. However, some languages with call-by-value semantics, such as Scheme and Tcl, do not have explicit pointers.

parts. `if` and assignment (`=`) are typical examples of special forms. `if` is a special form because its consequent part is only evaluated or executed if the conditional expression yields a true result. The assignment operation is a special form because its left-hand side (the target of the assignment) is not evaluated; rather, its meaning is a location into which the value of the right-hand side is to be stored.

In some cases, a special form does not constitute an expression in the sense of an execution that takes place at runtime to produce a value. Instead, a special form may represent a definition or other operation which affects the state of the compiler. Pragmas, definitions, declarations, and imports are all common examples of this kind of special form.

A language extension technique commonly used when the function calling protocol breaks down is that of *macros*. A macro is a user-defined transformation of the source program that takes place at compile time.

Because macros do their transforming at compile time, their invocation protocol is more flexible than the usual call-by-value function calling convention. For example, they may arrange to have some arguments evaluated or executed only in certain cases or more than once.

2.2.1 C Macros

The C language defines a standard preprocessor for C programs [23]. The preprocessor implements an extensible macro language, allowing some kinds of macros to be defined in C programs. However, C macros are almost completely distinct from the rest of the C language. Practically the only thing C macros share with the rest of the language is the lexical (token) structure and the syntax of function calls. The lack of cooperation between the C macro processor (typically implemented as a separate textual filter program) and the rest of the C language results in problems with both syntax and context.

Definitions in a program intended for the C preprocessor are lexically quite distinct from definitions intended for the plain C compiler. For example, a typical function definition in C looks like:

```
int sqr( int a )
{
  return a * a;
}
```

whereas an analogous macro definition looks like:

```
#define sqr(a) ((a)*(a))
```

Pre-ANSI C preprocessors considered a macro definition to expand to a sequence of characters. The lack of lexical structure in the expansion created opportunities to violate the lexical integrity of the macro arguments. For example, a macro such as the following does not preserve the lexical integrity of its arguments:

```
#define quote(a) "a"
```

A use of the macro `quote` such as “`quote(1 + 2)`” does not preserve the apparent lexical structure of the arguments to the call – the argument appears to be three tokens, whereas in fact the argument is interpreted as a string of 5 characters, with spaces being significant.

When the ANSI committee refined the C language, the recognition of the problems with violating lexical integrity prompted the specification to change so that the C preprocessor considers the body of a macro definition to be a sequence of tokens instead of a sequence of characters.

In some cases, it is accepted practice (and in fact a common idiom) to want to violate the lexical integrity of the arguments in a C macro. To support those cases, the ANSI committee defined special operators, “#” and “##”, to bypass the default behavior.

One place this feature is used is in the generation of families of language objects; that is, sets of related objects, typically sharing a portion of their name. For example, an application that uses C structures to record information about normal language procedures might make use of a C macro to build the appropriate structures automatically. The following macro, for example, illustrates such a use:

```
#define PROC(name,arg) \
  int name( int ); \
  struct info name ## _info = \
    { name, \
      # name }; \
  int name( int arg )
```

This macro can be used to declare a procedure with a common interface (*i.e.*, takes a single integer argument and returns an integer result) like so:

```
PROC(sqr,x)
{
  return x * x;
}
```

The occurrence of the PROC macro creates the appropriate structure declaration and fills it in. The use of the special ## operator in the macro definition creates a new token (identifier) composed of the characters of its arguments. Hence, the above example use creates a variable sqr_info of type struct info. Furthermore, the use of the special # operator creates a string whose contents are the characters of the macro argument, in this case "sqr". Using these special operators, the sqr_info structure can be initialized with the name of the procedure declared using the PROC macro.

The # operator is especially useful as a simple form of introspection, allowing a macro to transform the text of an argument into a program data object. The standard C assert macro makes use of this operator to provide an informative error message when a program assertion fails. Lisp programmers are familiar with this general technique as the ability for macros to quote an argument.

Despite these special operations, the default behavior in the C preprocessor is to preserve the lexical structure. Nevertheless, the lack of *grammatical* structure in the expansion creates opportunities to violate the grammatical integrity of the macro arguments.

Consider, for example, the following naïve implementation of the sqr macro:

```
#define sqr(a)  a*a
```

This macro definition appears to capture the intended meaning, but in fact is an error waiting to happen. Because the C preprocessor does not honor the grammatical structure of the macro arguments, a grammatical misinterpretation can arise. For example, consider what happens if the programmer uses `sqr(x+1)`. The rule of substitution of strings of tokens gives rise to the token sequence `x+1*x+1`, which means the same as `x+(1*x)+1`. However, the intended meaning was probably `(x+1)*(x+1)`, a very different expression.

Competent C programmers are familiar with the consequences of the C preprocessor not preserving the grammatical structure of the macro arguments. As a matter of habit, they defend against this kind of misinterpretation by inserting parentheses around the uses of macro arguments, as in the following definition:

```
#define sqr(a) ((a)*(a))
```

The invocation protocol for macros is one of textual substitution, so side effects in an argument to the `sqr` macro occur twice. Repeated side-effects can lead to peculiar situations. For example, the probability that `sqr(random())` returns a square number is almost nil, whereas it would appear to always do so.

Despite its limitations, the textual substitution protocol makes C preprocessor macros sufficient for defining special forms. The lack of any enforcement of evaluation order means that the macro definition can typically arrange to evaluate its arguments exactly as many times as desired.

However, the textual substitution protocol interacts poorly with the inability in the C language to define local variables within an expression. Lisp programmers who write macros are familiar with the problems of repeated side-effects and, like C macro writers inserting parentheses to prevent grammatical misinterpretation, routinely insert binding constructs to ensure that certain arguments are evaluated exactly once and in the right order. However, the C language enforces a dichotomy between statements and expressions. Statements may contain expressions, but not vice-versa. Since all the binding constructs in C are statements, an expression cannot bind a temporary variable to hold onto the value of an expression that should be evaluated once, such as the argument to `sqr`.

An extension to C supported by the `gcc` compiler is the ability to write a statement as an expression by enclosing it in braces within parentheses. This construct creates a *statement expression*. Hence, it is possible to write:

```
#define sqr(a) ({ int temp = (a); \
                  temp * temp; })
```

However, in ANSI C this is not possible, so a safe inlined version of `sqr` is impossible to write.

Another advantage of the textual substitution protocol for macro invocation in C, compared to the functional calling protocol, is that of polymorphism. That is, the polymorphism available with the standard mathematical operators in C is available to the caller of a macro using those operators.

For example, some versions of the `sqr` macro given above, such as:

```
#define sqr(a) ((a)*(a))
```

can operate on both integral and floating-point arguments, returning a value of the same type. This works because the macro substitution process has no knowledge of types, and textual substitution produces expressions to which the normal C compiler can apply the usual polymorphic arithmetic operators. Polymorphism of this nature is of relatively little use in C because of the limited availability of polymorphic operators, but C++ programmers can make considerable use of this capability.

When the programmer uses the statement expression extension of gcc, preserving this polymorphism in the safe version requires the use of an additional extension operator. The meaning of the typeof operator applied to some expression e is the type which is the result of e. Therefore, the typeof operator can be used to declare a variable of a type which is not known to the author of the macro definition. Extending our running sqr macro to preserve numeric polymorphism this way, we have:

```
#define sqr(a) ({ typeof(a) temp = (a); \
                  temp * temp; })
```

Even with the use of C extensions, C macros still have problems dealing with temporary names. The following example illustrates the problem:

```
#define dist(x,y) ({ typeof(x) x_temp = (x); \
                     typeof(y) y_temp = (y); \
                     sqrt( x_temp * x_temp + \
                           y_temp * y_temp ); })
```

If the expression given y involves a variable named x_temp, its occurrence in the macro expansion refers to the temporary variable defined by the macro.

The problem with internal names conflicting with program source names is somewhat ameliorated by the choice of obscure names for internal identifiers. However, the problem cannot be completely avoided if a macro might be used recursively or occur in different expansion paths.

For example, in C, macros can't expand recursively because there's no macro-time control construct to avoid indefinite recursion. However, two macros can both be defined in terms of some other macro, which results in the same kind of sharing.

```
#define UTIL_START      { int secretflag; secretflag = 0;
#define UTIL_END(proc)  if (secretflag) proc(secretflag); }
#define UTIL_TAG        secretflag = __line__

#define TRACK_ERRORS_START  UTIL_START
#define TRACK_ERROR_END     UTIL_END(got_error)
#define ERROR_HERE          UTIL_TAG

#define TRACK_WARN_START    UTIL_START
#define TRACK_WARN_END      UTIL_END(got_warning)
#define WARNING_HERE        UTIL_TAG
```

If a warning-tracking block is used inside of an error-tracking block, their `secretflag` variables get confused. In C, the way around this is to either duplicate the code (*i.e.*, not have a separate `UTIL` abstraction), or to pass around uniquifying identifiers (which might have to come from the source program, if the same kind of blocks of this sort can be nested, which may in turn happen via macro expansion!)

Another problem with C macros is their limited surface syntax – the use of a macro always looks like a function call, even when its semantics are more similar to constructs in the language with different syntax. For example, a macro to implement a new kind of looping construct would wind up looking like a function call rather than a while statement.

2.2.2 Lisp Macros

In a Lisp compiler (or interpreter), the basic processing of input programs is somewhat different from that of traditional FORTRAN-derived languages. The input program is subjected to lexical analysis and parsing, as usual, but the result of parsing is essentially a parse tree represented as a Lisp data structure, instead of an abstract syntax tree. The part of the Lisp compiler that assigns meaning to the input program (*i.e.*, builds an abstract syntax tree) operates on this parse tree data structure. In the traditional language approach, that part of the compiler functions inside the parser, assigning meaning during the parse itself.

Lisp, like C, also appreciates the utility of macros. The same comments above that made macros desirable in C apply to Lisp as well. Because Lisp, like C, has a fixed and call-by-value function calling protocol, the need to control the evaluation of arguments gives rise to the need for macros. The ability to portably inline code is another classic reason to use macros in Lisp, as well as in C.

Macros are implemented somewhat differently in Lisp than they are in C. Instead of a separate, preprocessing-based language layer, macros are executed at compile time, interleaved with the normal compilation [22][2]. When the compiler encounters a parse tree whose head names a macro, the compiler executes the body of the macro, with the other branches of the parse tree bound as the arguments to the macro. The result of executing the macro

[2] Actually, Common Lisp does not specify exactly when execution of macro forms happens, which gives rise to a whole new class of macro errors.

definition's body is a new parse tree, which the compiler then proceeds to compile as usual.

One thing to notice is that the body of the macro is *arbitrary Lisp code*. The complete power of the programming language is available to the programmer at compile time for the purpose of expanding the use of a macro. This is very different from most macro systems, and C's in particular, which have only very limited expressiveness available to the macro system[3].

Mostly because macros are so common in Lisp, a special operator was introduced to make it easy to write the body of a macro. This is the `quasiquote` or backquote operator, and it works essentially by letting the programmer write a parse tree as data with holes in it. For example,

```
`(* ,x 2)
```

represents a parse tree for a 3-element list with `*` at its head, a `2` in the last position, and a hole in the middle position. The execution of such a form effectively fills in the holes by evaluating the expressions within `unquote`s or commas. Hence, if the value of `x` is the parse tree `(+ 1 2)`, then the result of executing the above quasiquoted form would be:

```
(* (+ 1 2) 2)
```

The use of quasiquote makes it quite easy to write many Lisp macros. Extending the running example above for C, one could write a naïve macro in Lisp for computing the square of an argument:

```
(defmacro sqr (x)
  `(* ,x ,x))
```

[3] For example, in C it is not possible to write a macro `repeat(n,x)` which expands to n copies of the expression x.

In the context of C, we saw how the sqr macro would inadvertently evaluate its argument twice. This naïve macro in Lisp would do likewise.

Unlike C, Lisp does not distinguish statements from expressions. In effect, everything is an expression, including binding constructs. This uniformity makes fixing the multiple-evaluation problem straightforward:

```
(defmacro sqr (x)
  `(let ((temp ,x))
     (* temp temp)))
```

Furthermore, Lisp is dynamically typed, whereas C is statically typed. Hence, the lauded polymorphism available to C macros is trivially available in Lisp macros as well, even with a binding construct to hold temporary values[4]. In a dynamic type system, the polymorphism is realized at the leaves of the computation tree so intermediate compiler passes and intermediate variables need not replicate knowledge of the data type. (On the down side, this makes it much harder for compilers to check types and do type-based optimizations.)

The above macro definition still has the problem of name clashes of temporary names. The sqr macro is too simple to illustrate this problem, so instead consider a macro for the binary form of the Lisp or special form:

```
(defmacro or (a b)
  `(let ((temp ,a))
     (if temp
         temp
         ,b)))
```

[4] This benefit comes at the cost of either runtime performance or compiler complexity.

Recall that the intention is that `or` returns the value of its first argument if it is not `nil` (and in which case it does not evaluate its second argument). Otherwise `or` returns the value of its second argument.

The use of a temporary variable prevents repeated evaluation of the first argument when it turns out to be true, but the name given to that temporary variable can clash with names in the second argument. Consider the following use of the `or` macro:

```
(defun uncomfortablep (temp)
  (or (> temp 80)
      (< temp 65)))
```

The expansion of this macro results in:

```
(defun uncomfortablep (temp)
  (let ((temp (> temp 80)))
    (if temp
        temp
        (< temp 65))))
```

The identifier `temp` in `(< temp 65)` is meant to refer to the argument of the function `uncomfortablep`, and instead winds up referring to the local variable by the same name.

This problem can be somewhat mitigated by the choice of even more obscure names. However, when macros can be used recursively, no assignment of obscure names can protect against an inadvertent conflict. Fortunately, because Lisp macros have the complete expressiveness of the language[5] for gen-

[5] The language in which the macros are written is properly called the *meta*-language, but in Lisp the meta-language is the same as the underlying (target) language.

erating expansion parse trees, a macro can construct a new identifier on each invocation. This is the so-called *gensym* approach for dealing with variable capture problems in Lisp macros. The following modified or macro illustrates this approach:

```
(defmacro or (a b)
  (let ((temp (gensym)))
    `(let ((,temp ,a))
       (if ,temp
           ,temp
           ,b))))
```

Now, each time the or macro is called, a new, fresh identifier is created for use as the name of the temporary variable. Since the identifier is new, it cannot conflict with any other identifier in the program, whether generated by gensym or not.

By appropriate use of temporary variables to avoid multiple evaluation and the gensym approach to avoid name clashes, Lisp macros can be written that correctly provide their intended semantics. However, due to these pitfalls, doing so may involve considerable work and obscure bugs may lurk undetected.

Lisp macros are restricted to defining new expressions[6]. That is, the compiler does not recognize the use of a macro in places that are not semantically expressions, *e.g.*, in the formal arguments specification of a procedure.

[6] Common Lisp has a related but separate mechanism for defining macros for assignment locations. setf macros allow the definition of new kinds of arguments to the assignment special forms.

2.2.3 Scheme Macros

The Scheme language takes a slightly different approach to achieving the goal of macro support. Scheme macros are in some ways intermediate between Lisp macros and C macros. Like Lisp macros, they operate on structured parse trees (not abstract syntax trees). Like C macros, they are declarative and do not provide all of the expressive power of the language to the macro programmer.

The main distinguishing characteristic of Scheme macros is that they automatically provide *hygienic macro expansion* [14]. Hygienic macro expansion refers to the avoidance of name clashes such as we saw in the `or` macro example in the previous section.

Scheme macros are defined using the `define-syntax` form. A set of patterns are given with `syntax-rules`, and in each pattern the special symbol "_" acts as a place-holder for the name of the macro being defined. The special symbol "..." is used to denote a repeating element of the argument pattern. For example, a simple Scheme macro for `or` is:

```
(define-syntax or
  (syntax-rules ()
    ((_ a b)
     (let ((temp a))
       (if temp
           temp
           b)))))
```

This defines a macro which matches a two-argument invocation of `or`, for example, `(or (cat) (dog))`.

The macro expansion process as defined in R^{5}RS automatically ensures that names introduced in different contexts (*i.e.*, inside the macro definition *vs.* at the use site) do not inadvertently refer to each other. In this case, the `temp` variable that is used to prevent multiple evaluation of the first argument, `a`, does not conflict with any identifier `temp` in the argument `b`.

Note that Scheme macros have to deal with suppressing multiple evaluation of arguments. This is a property of all macro systems that can express special forms, because the whole purpose of a special form is to permit the evaluation of an argument either zero times or more than once.

Scheme achieves hygienic macro expansion by making the macro expansion process aware of the binding constructs such as `let` so that it can automatically rename variables when necessary. If the above `or` macro were used in something like:

```
(define (uncomfortable? temp)
  (or (> temp 80)
      (< temp 65)))
```

then the effect is as though the `gensym` approach in Lisp were used, but automatically. The result is something like:

```
(define (uncomfortable? temp)
  (let ((temp.1 (> temp 80)))
    (if temp.1
        temp.1
        (< temp 65))))
```

Because the macro expander is aware of the variable binding role of `let`, it can create a new name for its variables and keep track of the mapping from source names to renamed variables.

Most Scheme implementations appear to implement the hygienic semantics of Scheme macros using a renaming technique. That is, hygiene is achieved by automatically computing the necessary generated symbols.

The renaming transformation is a process which is interleaved with compilation, but separate from that compilation. As a result, the macro expander has to be aware of all the constructs in the language which affect scope and to handle any renaming appropriately. Fortunately, this is usually easy because the system designer typically implements only a few basic forms in the compiler proper and uses macro expansion to handle the rest. For example, the core compiler might implement only `lambda`, `letrec`, `set!` and a few others. Most binding constructs, like `let*` and `do`, would be implemented as macros. The macro expander only needs to be aware of the constructs directly understood by the core compiler because the expander already knows how to handle general hygienic expansion of macros.

2.2.4 Systems Related to Macros

Macro systems are not the only means for allowing the user to define special forms. More flexible argument passing techniques can generalize the traditional function calling protocol sufficiently that special forms are accessible.

One such flexible argument passing technique is that of *call-by-name.* In call-by-name argument passing, the evaluation of the arguments to a procedure is under the control of the called procedure. This achieves an effect somewhat like macros, and special forms can be written using this parameter passing technique.

For example, consider the following conditional construct (in a pseudo-Algol language):

```
define until( WHAT, TO_EXIT )
  begin
    while true do
      begin
        WHAT;
        if TO_EXIT then return;
      end;
  end.
```

Call-by-name has problems, though. In particular, it is known that a call-by-name calling convention cannot implement *swap* safely. Furthermore, it seems that the use of such a subtle calling convention as call-by-name can be quite dangerous to use as frequently as function calling is used, as well as being rather inefficient to use as the default calling convention.

Another argument passing technique that is flexible enough to implement some special forms is that of *call-by-need* or *lazy evaluation*. In this technique, arguments are evaluated zero or one times, and only when needed. Conditional constructs can be implemented using call-by-need. However, new binding constructs cannot be implemented in Haskell because, although typical implementations use a transformational process to convert the exposed language to a lower-level core language (*desugaring*), these sugar-coating facilities are not exposed to the programmer.

2.3 A Taxonomy of Scheme Macros

2.3.1 Call-by-name Inline Procedures

Perhaps the simplest use of Scheme macros is as *call-by-name inline procedures.* In this case, macro definitions are regarded as declarations of inline procedures whose arguments are passed by name rather than by value.

Call-by-name inline procedures are also the easiest to implement correctly. The technique of *syntactic closures* was introduced by Bawden and Rees [6] to solve the hygiene problem for call-by-name inline procedures. The syntactic closure approach is sufficient for implementing macros used as call-by-name inline procedures, and our approach is based on extending these ideas.

As an example, the `or` macro illustrates the common use of macros as call-by-name inline procedures.

```
(define-syntax or
  (syntax-rules ()
   ((_ term)
    term)
   ((_)
    #f)
   ((_ term terms ...)
    (let ((temp term))
      (if temp
          temp
          (or terms ...))))))
```

2.3.2 Advertent Capture

Inline call-by-name procedures, and a straightforward implementation using syntactic closures, cannot express binding constructs. The ability to create bindings in a macro body whose variables are visible to expressions that are arguments to the macro requires *advertent capture.* The sense of advertent is that the capture of identifiers by the body of the macro is done on purpose, with due consideration of the intended semantics.

The Scheme form `let*` is easily expressed using a macro with advertent capture:

```
(define-syntax let*
  (syntax-rules ()
    ((_ () body ...)
     (begin body ...))
    ((_ ((var init) bdg ...) body ...)
     (let ((var init))
       (let* (bdg ...) body ...)))))
```

In this macro, the bindings represented by `var` are inserted by the macro, and capture references within `body`. For example, a use of the `let*` macro such as:

```
(let* ((begin 'start)
       (end 'stop))
  (list begin end))
```

would expand into:

```
(let ((begin.1 'start))
  (let ((end.1 'stop))
    (begin
      (list begin.1 end.1))))
```

where `begin.1` and `end.1` are identifiers constructed by the macro expansion process. They are chosen to be unique, and hence do not clash with any reference in the body of the macro, in particular the reference to `begin`.

2.3.3 Explicit Intentional Capture

A third general use of macros is to insert a binding into an environment where the inserted name does *not* occur as an argument to the macro. We call this *explicit intentional capture.*

Implementing something like C `while`, which permits the use of `break` within its body, requires explicit intentional capture. Consider the following procedure which makes use of a hypothetical `while` special form:

```
(define (with-each-datum port proc)
  (while #t
    (let ((datum (read port)))
      (if (eof-object? datum)
          (break)
          (proc datum)))))
```

Here, the special form `while` introduces a new binding for `break` in the scope of its body. The name `break` is not explicitly referenced by the invoker of the macro, so the advertent capture rules cannot apply. Explicit intentional capture makes it possible to write `while`, and at the same time makes the macro author explicitly aware that they are bypassing the normal scope rules.

2.4 Our Contribution

Our approach is novel in two respects. First, it does not explicitly rewrite the source program; there is no notion of a "transformed" output that is subsequently fed into a macro-deficient compiler. Second, it allows the meaning of special forms (including syntax) to be propagated upward in the compilation process. The latter corresponds to a controlled form of eager macro processing.

2.4.1 RScheme Macros

The RScheme implementation of Scheme supports Scheme macros, but the system achieves hygienic macro expansion using a technique which is not based on preprocessing and renaming of variables, which most Scheme implementations use. The RScheme implementation integrates macro expansion with compilation; there is no macro expansion *per se*.

Because macro processing is fully integrated with compilation, RScheme's macro facility does not need to be explicitly aware of the binding constructs in the underlying language. Instead, the macro facility directly manipulates the compile-time objects representing variables and bindings. Furthermore, the order of macro processing is well-defined, since the RScheme system does not have a separate interpreter.

2.4.2 Type Reflective Macros

The integration of the macro system with the compiler makes possible additional taxa of Scheme macros. An RScheme macro can use compile-time information to do new kinds of pattern matching. Being able to pattern match on compile-time information allows certain kinds of optimizations to be expressed using the macro system, which makes this ability a powerful language development tool as well as valuable to the end user.

For example, a runtime system might have two primitive procedures for adding numbers, one to be used when both arguments are known at compile-time to be small integers. A macro can define a pattern that only matches when that condition is satisfied and generates the appropriate, fast instruction. The default rule could invoke the slower primitive procedure.

This general approach has been used in the RScheme system to structure the interface between high-level code and low-level primitives with differing performance tradeoffs. Examples are elaborated in Section 2.5.2.

2.5 Our Implementation

Our implementation is based on a simple recursive compiler operating over a surface parse of the input program. The surface parse is the result of the Scheme `read` procedure, and hence is a tree-structured representation of the source text. The tree structure is laid out explicitly by the user; there is no understanding of the language grammar or semantics that are applied at this time. The recursive structure of compilation is over this tree, so the program subtext at any given point is the input to the compilation procedure.

The return value from the compilation procedure is the *meaning* associated with that program subtext in the context of that point in the system. In particular, it is either intermediate code or a meta-object denoting a variable.

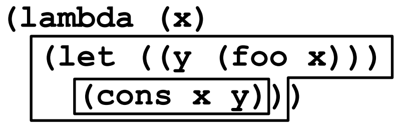
The structure of intermediate code as used in our implementation is not important for this discussion – it may be any appropriate representation which has variable bindings completely resolved. In our implementation, it is a simple tree-code representation which can be immediately fed into the back end of the RScheme compiler for code generation. Furthermore, textual references to variables are encoded as object references to the corresponding *variable* meta-object. No binding ambiguity can arise, because the variable reference points to the actual variable meta-object. Meta-objects denoting variables represent occurrences of variable definitions in the input program.

2.5.1 Operation

To compile an expression in an environment, we maintain a data structure representing the "location" of the expression. By location here we mean a complete indication of the scope of the expression, which is sufficient to completely resolve the meaning of any identifier that may occur in that expression.

An expression's location is a 2-tuple of its *place* and its *environment*. The place denotes the lexical position within the source text. The environment denotes the mapping from identifier-place tuples to actual variables. By actual variables, we mean completely resolved compile-time meta-objects that represent either a collection of run-time bindings or a distinct compile-time binding.

To illustrate this terminology, consider the following program fragment:



```
(lambda (x)
  (let ((y (foo x)))
    (cons x y)))
```

The rectilinear contours are a visualization of the *places* in this program fragment. The `lambda` introduces a new contour for its argument. Likewise, the `let` introduces a contour for its variable. This fragment, with places *P1* and *P2*, exercise the the following bindings:

$$\begin{array}{rcl} \langle \texttt{lambda}, \mathit{top} \rangle & \rightarrow & \mathit{lambda\text{-}sf} \\ \langle \texttt{x}, \mathit{P1} \rangle & \rightarrow & \mathit{x\text{-}var} \\ \langle \texttt{y}, \mathit{P2} \rangle & \rightarrow & \mathit{y\text{-}var} \end{array}$$

Example 1

Let us walk through an initial, simple example. This example does not illustrate the more subtle effects when macros are involved; it just shows how the system works in the simple case. This should make it clear that the correct result is produced at least for code that doesn't use macros.

This simple procedure `cons`'s the head of the first list onto the second list, for example, turning `(a b)` and `(3 2 1)` into `(a 3 2 1)`:

```
(lambda (x y)
  (let ((z (car x)))
    (cons z y)))
```

Initially, the *place* is *top* and the *envt* consists of (along with many more like this):

$$\begin{aligned}
\langle \texttt{lambda}, \mathit{top} \rangle &\rightarrow \textit{lambda-sf} \\
\langle \texttt{let}, \mathit{top} \rangle &\rightarrow \textit{let-sf} \\
\langle \texttt{cons}, \mathit{top} \rangle &\rightarrow \textit{cons-tlv} \\
\langle \texttt{car}, \mathit{top} \rangle &\rightarrow \textit{car-tlv}
\end{aligned}$$

The *-sf* suffix is a mnemonic to indicate meta-objects denoting special forms. Likewise, the *-tlv* suffix is used to name top-level variables. In the actual implementation, these names correspond to meta-objects which are subclasses of `<variable>`. Each special form meta-object has a compile-time procedure, its *handler*, associated with it, which is responsible for implementing the semantics of the form. The handler is invoked to process an occurrence of the form, and is provided with the complete compile-time environment.

The meaning of the form `(lambda ...)` is determined by the meta-object that is the meaning of the head part. That is, to compute the meaning of a list structure, the meaning of the head position is computed, and then computing the meaning for the entire form is delegated to the head's meaning.

Computing the meaning of a symbol involves a double loop. The outer loop is over the nesting of the *place*. That is, we start at the current *place*, and if we can't find a binding in the environment for that place, we try its outer contour until we run out of places to look. If we run out of places, the symbol is unbound in this place, which is an error[7].

Since the current *place* is *top*, we search the *envt* (bottom to top) for a tuple $\langle \texttt{lambda}, \mathit{top} \rangle$. In this case, we find it – it is bound to the variable

[7] In the actual implementation, giving up means that the symbol presumably refers to an as-yet undefined variable; *i.e.*, it is a forward reference. We do not address those engineering issues in this discussion, assuming that all variables are defined.

lambda-sf – so we return that as the meaning of the symbol `lambda` in the place *top*.

Now we return to the problem of determining the meaning of `(lambda ...)`, whose head means `lambda-sf`. The behavior of special forms for computing compositions is to invoke the special form's handler, the compile-time procedure associated with the special form for just this purpose. This procedure is one of the primary gateways from the main recursive compilation driver to special-case code.

The handler for *lambda-sf* parses the "lambda list" – the procedure arguments[8]. Having parsed the arguments, the *lambda-sf* handler constructs a new contour, since it knows that its body is in a new scope, and the arguments are bound in the environment with respect to the new place. Call the new place P. Then, in this case, the handler adds:

$$\begin{aligned} \langle \mathtt{x}, P \rangle &\rightarrow \textit{x-var} \\ \langle \mathtt{y}, P \rangle &\rightarrow \textit{y-var} \end{aligned}$$

to the end of the environment chain.

The meta-objects *x-var* and *y-var* are compile-time objects that represent a collection of storage locations at runtime.

Having established the bindings, the meaning of the body, `(let ...)`, is computed in the new place, P, and environment. As before, the structure is recognized as a list, and the head looked up. In this case, the first iteration of the outer lookup – an attempt to locate $\langle \mathtt{let}, P \rangle$ in the environment – fails, so the outer place, *top*, is checked and $\langle \mathtt{let}, \textit{top} \rangle$ is found to be bound to *let-sf*.

[8] How the lambda list is parsed is to use the internal pattern-matching mechanism from the inside of the compiler, since the pattern matching already knows how to expand pattern variables in pursuit of a match.

Example 2

Let us take as a further example a more difficult case. In this case, we define a macro and call it. However, we still do nothing very complicated. In fact, this example could be handled well with a syntactic closures approach [6].

```
(lambda (temp x y)
  (let-syntax ((or (syntax-rules ()
                    ((_ a b)
                     (let ((temp a))
                       (if temp
                           temp
                           b))))))
    (or (temp x)
        (temp y))))
```

In this example, the first argument to the procedure is intended to be another procedure which obtains the temperature value of an object, or returns `#f` if the temperature is not known.

As before, initially *place* is *top* and the environment consists of:

$$\begin{array}{rcl} \langle \texttt{lambda}, \mathit{top} \rangle & \rightarrow & \textit{lambda-sf} \\ \langle \texttt{let}, \mathit{top} \rangle & \rightarrow & \textit{let-sf} \\ \langle \texttt{let-syntax}, \mathit{top} \rangle & \rightarrow & \textit{let-syntax-sf} \\ \langle \texttt{if}, \mathit{top} \rangle & \rightarrow & \textit{if-sf} \end{array}$$

Again, the processing of `lambda` creates a new place – call it Q – and extends the environment with:

$$\begin{aligned}\langle \texttt{temp}, Q\rangle &\rightarrow \textit{temp-var0}\\ \langle \texttt{x}, Q\rangle &\rightarrow \textit{x-var}\\ \langle \texttt{y}, Q\rangle &\rightarrow \textit{y-var}\end{aligned}$$

Now, when the `let-syntax` form is processed, the handler[9] creates a place inside Q – call it R – and extends the environment with:

$$\langle \texttt{or}, R\rangle \rightarrow \textit{or-rules}$$

Here, *or-rules* is a kind of special form that captures the syntax rules, the place Q, and the environment up through the binding for `y`. (Note that if this had been a `letrec-syntax` form, the place R would be captured and hence its environment would include the `or` binding.)

The determination of meaning for the body of the `let-syntax` is the usual. In this case, when `or` is found in contour R, its meaning is a syntactic abstraction. The meaning of a list whose first element means a syntactic abstraction is determined by finding an appropriate expansion using pattern matching in the syntax rules.

Here we have only one pattern, and it matches. The result of finding a pattern match is that the current place reverts to the place of definition, and then a new contour (place) is created to represent the scope of the pattern variables. In this case, this means we bind the identifiers `a` and `b` to pattern variable objects that capture the source text and place. The environment itself is not reverted.

[9] Located via the binding $\langle \texttt{let-syntax}, \textit{top}\rangle \rightarrow \textit{let-syntax-sf}$

Let us call the new place S, in which case we bind:

$$\begin{aligned} \langle \mathtt{a}, S \rangle &\rightarrow \textit{a-pv} \\ \langle \mathtt{b}, S \rangle &\rightarrow \textit{b-pv} \end{aligned}$$

where *a-pv* denotes the text `(temp x)` in place R, and *b-pv* denotes the text `(temp y)` in place R.

We now compute the meaning of the body of the matched syntax rule in the so extended environment and in place R. The `let` form is recognized as usual.

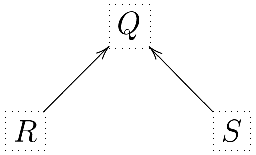
Notice at this point that we are computing meanings as usual – there is no explicit recognition that we are inside the body of a syntactic template.

It is during the computation of the meaning of the initial value expression for the `let` that we first encounter a pattern variable. That is, the lookup of $\langle \mathtt{a}, S \rangle$ finds *a-pv*. To compute the meaning of a pattern variable, we compute the meaning of its text in its place of origin, in this case `(temp x)` in R. The environment chain is unchanged – it monotonically increases with the depth of recursive compilation and implicitly shrinks when a recursive compilation exits.

At this point, the environment contains:

$$\begin{array}{rcl}
\langle \texttt{lambda}, \mathit{top} \rangle & \rightarrow & \mathit{lambda\text{-}sf} \\
\langle \texttt{let}, \mathit{top} \rangle & \rightarrow & \mathit{let\text{-}sf} \\
\langle \texttt{let-syntax}, \mathit{top} \rangle & \rightarrow & \mathit{let\text{-}syntax\text{-}sf} \\
\langle \texttt{if}, \mathit{top} \rangle & \rightarrow & \mathit{if\text{-}sf} \\
\langle \texttt{temp}, Q \rangle & \rightarrow & \mathit{temp\text{-}var0} \\
\langle \texttt{x}, Q \rangle & \rightarrow & \mathit{x\text{-}var} \\
\langle \texttt{y}, Q \rangle & \rightarrow & \mathit{y\text{-}var} \\
\langle \texttt{or}, R \rangle & \rightarrow & \mathit{or\text{-}rules} \\
\langle \texttt{a}, S \rangle & \rightarrow & \mathit{a\text{-}pv} \\
\langle \texttt{b}, S \rangle & \rightarrow & \mathit{b\text{-}pv}
\end{array}$$

And the environment chain looks like:

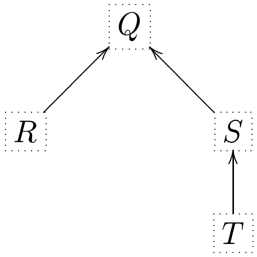


Since `(temp x)` is a list, we compute the meaning of the first element as usual. In this case, there is no $\langle \texttt{temp}, R \rangle$, but we find $\langle \texttt{temp}, Q \rangle$ instead, which is a regular variable that represents the first formal argument to the procedure we're compiling. Since the meaning of the first element of the list is a regular program variable, the entire list must be a procedure call. Thus, the meaning of the remaining elements of the list are determined, and the meaning of the entire `(temp x)` is a combination.

This works for all occurrences of this general style. In general, any binding that is added to the environment from within the syntax rule has a *place* attribute which is different from that of any other binding. Specifically, bindings created inside the syntax rule are in place S (or a descendant), and thus do not match a binding meant for place R.

Having computed the meaning of the initial value expression for the `let` inside the syntax template, the *let-sf* handler creates a new contour, T, and extends the environment with `temp` in T:

$$\langle \texttt{temp}, T\rangle \quad\rightarrow\quad \textit{temp-var1}$$



With this in place, the identifier `temp` inside the `let` body (*i.e.*, in T) matches $\langle \texttt{temp}, T\rangle$ instead of $\langle \texttt{temp}, R\rangle$. The rest of the compilation in this example proceeds similarly, obtaining the desired result.

Example 3

Here we illustrate how the system operates in the presence of advertent capture, and how it detects the implicit capture rule.

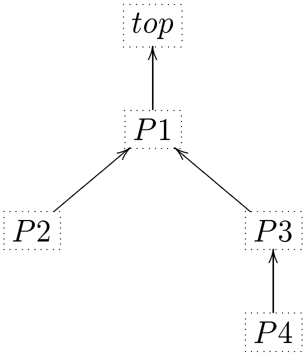
```
top: (lambda (n)
 P1:   (let-syntax
          ((for (syntax-rules ()
                  ((_ (var init limit) body ...)
 P3:               (let loop ((var init))
P4:                   (if (< init limit)
                          (begin
                            body ...
                            (loop (+ var 1)))))))))

P2:     (for (i 0 n)
          (print i))))
```

To solve this problem, we revisit the observation made previously – that the special forms use the internal pattern matching mechanism to match their own arguments. In this case, `let` uses the pattern matcher to match its arguments, and by doing so, obtains the expansion of `var` along with its place. When `let` goes to bind `var`, the `let` actually binds `i` in the place of call, so the environment during the compilation of `body` looks like:

$$
\begin{array}{rcl}
\langle \texttt{lambda}, \textit{top} \rangle & \rightarrow & \textit{lambda-sf} \\
\ldots & & \\
\langle \texttt{n}, \textit{P1} \rangle & \rightarrow & \textit{n-var} \\
\langle \texttt{for}, \textit{P2} \rangle & \rightarrow & \textit{for-rules} \\
\langle \texttt{var}, \textit{P3} \rangle & \rightarrow & \textit{var-pv} = \langle \texttt{i}, \textit{P2} \rangle \\
\langle \texttt{init}, \textit{P3} \rangle & \rightarrow & \textit{init-pv} = \langle \texttt{0}, \textit{P2} \rangle \\
\langle \texttt{limit}, \textit{P3} \rangle & \rightarrow & \textit{limit-pv} = \langle \texttt{n}, \textit{P2} \rangle \\
\langle \texttt{body}, \textit{P3} \rangle & \rightarrow & \textit{body-pv} = \langle \texttt{(print i)}, \textit{P2} \rangle \\
\langle \texttt{i}, \textit{P2} \rangle & \rightarrow & \textit{i-var}
\end{array}
$$

Where the places are arranged so:



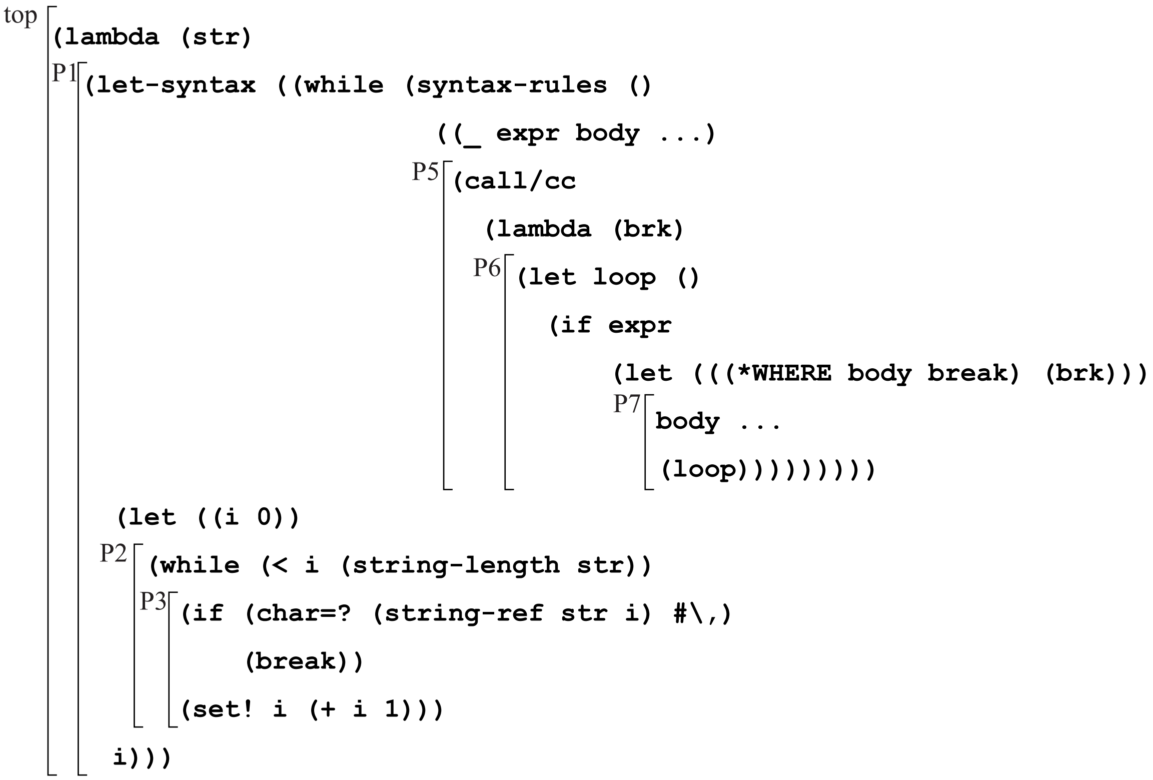
Note that there is no need to bind $\langle$`var`$, P4\rangle$, because any use of `var` in $P4$ expands into `i` in $P2$ anyway by virtue of its expansion.

Example 4

This example illustrates the implementation of explicit intentional capture. Consider the following procedure definition that uses a local macro implementing `while`, with the lexical places *top*, $P1$, ..., $P7$.

This example uses the `call/cc`[10] primitive of Scheme to implement non-local transfer of control. The `call/cc` procedure calls its argument (here, the `lambda` with body $P6$) with one value, which is a procedure (here, bound to `brk`). A call to that procedure (`brk`) does not return, and instead causes `call/cc` to return. This kind of non-local control transfer is familiar to C programmers as `setjmp/longjmp`.

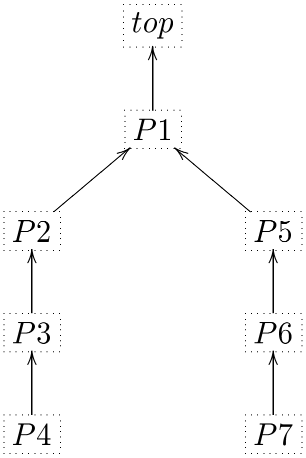
[10] Formally, `call/cc` is `call-with-current-continuation`. The name is abbreviated for obvious reasons.



```
(lambda (str)
  (let-syntax ((while (syntax-rules ()
                        ((_ expr body ...)
                         (call/cc
                           (lambda (brk)
                             (let loop ()
                               (if expr
                                   (let (((*WHERE body break) (brk)))
                                      body ...
                                      (loop))))))))))
    (let ((i 0))
      (while (< i (string-length str))
        (if (char=? (string-ref str i) #\,)
            (break))
        (set! i (+ i 1)))
    i)))
```

The compilation of this form proceeds just as before, until the compiler gets to creating the binding for `break`. At this point, we introduce the `*WHERE` operator. Its purpose is to signal the compiler explicitly that a symbol is being intentionally captured. In this case, the form `(*WHERE body break)` tells the compiler's binding facility to interpret the symbol `break` as if it had come from the same place as the value of the pattern variable `body` (*i.e.*, from the call site).

As a result, during the processing of the body of the inner `let`, we might have:



And:

$$
\begin{array}{rcl}
\langle \texttt{lambda}, top \rangle & \rightarrow & \textit{lambda-sf} \\
\ldots & & \\
\langle \texttt{let}, top \rangle & \rightarrow & \textit{let-sf} \\
\langle \texttt{str}, \textit{P1} \rangle & \rightarrow & \textit{str-var} \\
\langle \texttt{while}, \textit{P2} \rangle & \rightarrow & \textit{while-rules} \\
\langle \texttt{i}, \textit{P3} \rangle & \rightarrow & \textit{i-var} \\
\langle \texttt{expr}, \textit{P5} \rangle & \rightarrow & \textit{expr-pv} \\
\langle \texttt{body}, \textit{P5} \rangle & \rightarrow & \textit{body-pv} \\
\langle \texttt{brk}, \textit{P6} \rangle & \rightarrow & \textit{brk-var} \\
\langle \texttt{break}, \textit{P3} \rangle & \rightarrow & \textit{break-var}
\end{array}
$$

2.5.2 Reflection Operators

At this point, we are ready to introduce the remaining reflective operators that are implemented and give examples of how to use them and when they are appropriate. We have already seen the use of the `*WHERE` operator (abbreviated "`::`" in RScheme, in analogy to C++'s scoping operator, so that `body::break` is equivalent to `(*WHERE body break)`).

Aliasing

For purposes of aliasing (*i.e.*, ensuring that two symbols in different places refer to the same object), we introduce the `let-alias` form. For example, if the `while` that we had above required that `break` be available both in its body and in its expression argument (since those are different expressions) we would use `let-alias` to ensure it the necessary dual availability:

```
(define-syntax while
  (syntax-rules ()
    ((_ expr body ...)
     (call/cc
       (lambda (brk)
         (let loop ()
           (let ((body::break (brk)))
             (let-alias ((expr::break body::break))
               (if expr
                   (begin
                     body ...
                     (loop))))))))))))
```

Note that `let-alias` is simply a binding construct that does not create any new meta-objects; it only re-links existing meta-objects under new names or scopes.

Type-based Pattern Matching

A powerful application of the interleaving of compilation with macro expansion is type-based pattern matching. This application gives us the ability to define *type-polymorphic inline procedures*, as is done in the mapping of primitive operations in *RScheme* to user accessible procedures.

For example,

```
(define-syntax binary+
  (syntax-rules ()
    ((_ (*IS x <fixnum> :constant) (*IS y <fixnum> :constant))
     (*EVAL (+ x y)))
    ((_ (*IS x <fixnum>) (*IS y <fixnum>))
     (fixnum+ x y))
    ((_ x (*IS y <fixnum> constant))
     (let ((temp x))
       (if (fixnum? temp)
           (fixnum+ x y)
           (generic+ x y))))
    ((_ x y)
     (generic+ x y))))
```

The *IS operator is used to reflect on compile-time type information and other attributes. In order to determine if a parameter matches, the compiler is obliged to compile the expression and determine its type. In general, this is risky if the meta-system can have side-effects on the compile-time state. In the current implementation, we leave it to the macro developer to be aware of any such issues. The *EVAL operator is used to evaluate code at compile time, which here is used to actually do the work of the optimization.

Another approach that was tried was to tentatively compile the argument, and reuse the resulting meaning as the pattern variable expansion. This approach has the disadvantage of not allowing the argument to appear in a different environment (although if the argument appears in a different environment in use, then the author needs to be aware of the possibility that the expression may exhibit different attributes during use than during pattern matching!).

Consider the following example of using synthesized attributes:

```
(define-syntax for
  (syntax-rules ()
    ((_ (var
          (*IS init :side-effect-free)
          (*IS limit :side-effect-free))
        body ...)
     ⟨ implementation exploiting lack of side-effects in init and limit ⟩)
    ((_ (var init limit) body ...)
     ⟨ fallback implementation ⟩)))
```

This ability can be considered a simple kind of *fact-based pattern matching*, where the available facts are encoded by the type system and other accessible properties of the compile-time context. This could be generalized to manipulating arbitrary (and potentially domain-specific) synthesized and inherited attributes, as is done in McMicMac by Krishnamurthi *et al.* [27].

Chapter 3

Extensible Parsing

3.1 Introduction

Implementors of conventional computer languages have long been concerned with the problem of *parsing.* Parsing is the process of turning a linear string of characters representing a program into a structured representation that is closer to the meaning of the program. *Conventional languages*, as we use the term here, refers to languages whose syntax is derived from Algol. This family includes Pascal, C, and, more recently, Java and C#. From a syntactic point of view, these languages are in significant contrast to the Lisp family, in which the parse structure is coded explicitly by the programmer, making the job of syntax analysis trivial for a Lisp compiler.

In this chapter, we introduce the notion of *extensible parsing.* Extensible parsing is a generalization of the traditional idea that the grammar, or syntax, of the language is fixed at language design time. Instead, the grammar can evolve as the processing of a program takes place. In many ways, the ability to modify or extend the grammar during program processing is analogous to the definition of macros in a Lisp-like language. However, in a conventional language, there is no manifest tree-like data structure on which the macros might operate.

In this chapter, we give a simple implementation of an extensible parser that is based on the recursive-descent parsing strategy. We are not concerned with the performance of the extensible parser but instead use it to make concrete the ideas we present as making up the extensible parsing framework.

The main idea of extensible parsing is that the grammar is a data object to be manipulated at the runtime of the compiler, much as environment and scope were manipulated in Chapter 2. However, the grammar is not manipulated arbitrarily. Instead, we illustrate some common kinds of grammar changes that behave in fairly regular ways. We call these regular patterns *grammar change styles.* The two major styles are top-level forms and nested, block-like constructs.

Top-level forms tend to make grammar changes that live beyond the occurrence of the form itself. For example, in C a top-level form that defines a new type (*i.e.*, a `typedef`) has a scope that extends to the end of the program unit.

Block-like constructs, on the other hand, tend to support nesting and make grammar changes whose extent is contained within the form itself. For example, a C `while` statement might introduce a new grammar rule that makes `break` a valid construct within its body.

Although performance is not the main focus of this chapter, it is worth remarking upon the focus of much of the research into efficient parsing techniques over the past 30 years or so. This research primarily leverages off the fact that language grammar was fixed and known at compiler design time. Since the compiler would be invoked many times for the one time that the compiler itself was compiled, a large amount of processing was warranted in producing an efficient compiler. In the case of the language grammar, this meant that an almost arbitrary amount of work was justified in constructing a parser that would be efficient at runtime (*i.e.*, when the compiler ran, which is to say, when it was compiling some other program).

The common computer science technique of moving computation across the barrier between compile time and run time was applied with excellent results in the field of parsing techniques. Typically, at build time, an abstraction of the workings of the parser would be constructed and formed into a finite-state automaton. The automaton would be encoded into *parse tables* that would be built into the compiler as static data structures to be interpreted at runtime.

However, since computers are roughly 1000 times faster than they were when much of this research took place, and more and more time is being spent in optimization phases instead of front-end processing like parsing, the benefits of this body of language research are less clear today. In particular, if a considerable amount of expressive power or programmer flexibility is available at the expense of some parsing time, it seems a tradeoff well worth making. The usefulness and consequent construction of a *flexible* parser, at the cost of preprocessing for efficient runtime execution, is the subject of this chapter. In Chapter 4, we return to the subject of efficiency and give an implementation that maintains the flexibility of extensible parsing but with improved efficiency.

3.2 Background

Extensible parsing is not a new idea. Cardelli [9] uses lambda calculus as a base language and shows how it can be extended to support the embedding of other languages such as SQL. However, for Cardelli grammar changes are global in scope, making the use of grammar changes suitable only in certain circumstances.

A theoretical framework for extensible parsing is available in the recent work on adaptable grammars, such as that of Schutt [35]. However, as far as we can tell, there is no implementation underlying this work.

3.2.1 What We Would Like to Do

Our approach seeks to permit grammar changes both with high frequency and with local scope. An example of grammar extensions using this approach would be the `while` statement that introduces the `break` statement. In this approach, the `while` would create a label that is the target of a `goto` generated by the `break`. Furthermore, we would like for each nested `while` statement of this kind to use unique labels. It therefore becomes necessary for grammar changes to manipulate lexical scopes.

In a longer perspective, we imagine introducing rules for entire object system extensions. This might involve first-class representations of whole sub-grammar changes that can be stored with the meta-objects for classes. For example, this might be a means to allow the implementation of C++ as a collection of syntactic extensions on top of C.

In essence, we are trying to obtain the benefits of Lisp and Scheme macros for more traditional programming languages. Recall that Lisp and

Scheme macros manipulate a tree representation of the program rather than, for example, the token strings that are manipulated by C (`cpp`) macros. Lisp languages make use of a two-level grammar, where the first level parses token streams into a tree representation. The second level takes those trees as input and generates abstract syntax trees. Lisp and Scheme macros operate at the second level, and the first level is fixed and trivial in structure. These two levels are intermixed in ordinary programming languages, making it necessary to modify the grammar to achieve the effect of Lisp macros. With this approach we imagine being able to start with a small subset of Java and building up the entire language using syntactic extensions.

3.2.2 Granularity of Grammar Changes

For our approach to obtain maximum usefulness, we must allow grammar changes on a very fine granularity. In particular, we must be able to parse the input corresponding to one part of a grammar rule using a different grammar from that used to parse the rest of the input. The grammar is represented by the parse environment, which is passed around and modified in the parsing process to achieve the parsing of different parts in different environments.

3.2.3 Applications of Our Approach

With this approach, we can express non-context-free constructs in a structured way. An example of such a non-context-free construct is C's `while` and `break`, as mentioned above.

Another typical example is the `typedef` in C. In the traditional approach, the tokenizer is patched to reflect a new lexical category for an identifier, which widens the interface between the parser and scanner components. In our approach, we can handle this exclusively in the parser – we would introduce a new grammar rule for *type-name* that implicitly re-categorizes the new type name. Because the grammar understands the scope contours of the language, such a grammatical re-categorization follows the normal scoping rules of the language (which would be more difficult to implement in the scanner, which has no concept of language scope).

3.2.4 Limitations

Lisp macros, by virtue of working on internal data structures, are not limited to any particular order of inspection of their parts. A grammatical approach, however, is. Tokens only become available in a fixed, left-to-right, order. Hence, we would have difficulty expressing grammatical constructs such as Haskell's `where` clause because the variable declarations come after the body. Likewise, Java's `try`/`catch` is problematic to process syntactically because the `catch` modifier comes after the body that it modifies.

Note that we can parse these constructs, but we cannot make use of fine-grained grammar changes to do so. Some kinds of grammar changes are still possible, but any changes that involve the meaning values from the suffix of the construct cannot determine the grammatical structure of the prefix. In any case, a compiler using our approach could still perform traditional manipulations of the meaning structures to implement the intended semantics.

3.3 Contour Sensitivity

While this approach has a much greater expressive power than pure context-free grammar processing, it is not as unstructured as a general-purpose programming language for parsing. We *extend* the power of a traditional grammatical framework without going to the extreme of a full programming language, which gives arbitrary expressiveness with no grammatical structure. Since we know of no other name for this kind of power, we choose the name "contour sensitivity" because the constructs are sensitive to the grammatical contours.

3.3.1 Major Styles of Environment Passing

Even the power of contour sensitivity might be too much in some cases. In fact, we have only been able to distinguish two major usage patterns of contour sensitivity in grammars.

The first usage pattern is characterized by file-level global or forward-scope constructs such as Scheme's `define` and C's `typedef`. In this style, which we call *sequence style*, language entities following the construct are all in the scope of the construct.

The second usage pattern is characterized by local scope, such as Scheme's `let` or C's block-local variables. In this style, which we call *block style*, only entities *contained* in the construct are in its scope. (Notice, though, that local variables within a block in C follow the first usage pattern, as in Scheme's `let*`.)

3.3.2 General Mechanism

In order to explain the general mechanism, we need some additional terminology. We assume the reader is familiar with basic parsing terminology.

A *rule* maps a non-terminal to a sequence of terminals and non-terminals.

A *parser* is a procedure – derived from a rule – which implements the parsing of the input, recursively calling other parsers to recognize the non-terminals in the corresponding rule.

The *meaning* is the result of compiling an input string. Meanings are usually abstract syntax trees but can have other representations such as code, or can even be values in the case of a syntax-directed interpreter. Rules are thought of as producing a meaning as a composition of the meanings of its components. Terminals have appropriate elementary meanings corresponding to concrete lexemes.

A *parse environment* maps non-terminals to sequences of parsers. Each parser in the sequence corresponds to a rule for the non-terminal.

A *rule* is used to parse input using an environment called the *inherited* environment. The result of such a parse (if successful) is the meaning as well as another environment called the *synthesized* environment.

The top-level parse procedure has as arguments a non-terminal to be parsed and an environment in which to interpret it. The parse procedure looks up the non-terminal in the given environment, obtaining the associated sequence of parsers. The top-level procedure calls the parsers in order with the same environment until one succeeds.

A parser, in turn, takes a parse environment and a sequence of tokens to be parsed. The parser attempts to recognize an instance of the corresponding rule at the beginning of the given sequence of tokens. If it succeeds, the parser passes the remaining input sequence and a possibly modified environment to its continuation.

When a rule contains a terminal, the corresponding parser simply checks that the input sequence contains that terminal. However, when a rule contains a non-terminal, the corresponding parser contains a call to the top-level parse procedure. The parser at this point is free to pass an environment of its choice to the top-level parse procedure. For example, in implementing `while`/`break` as mentioned above, the parser associated with `while`, if invoked with environment E, would pass an environment E' to parse its body. E' would be E augmented with a parser that recognizes `break` as a statement.

Similarly, a parser is free to use or ignore the parse environment returned by the recursive call to the top-level parse procedure. For example, in implementing `typedef`, the parser that recognizes sequences of top-level forms passes the synthesized parse environment from one form as the inherited parse environment to the next.

3.4 Implementing Contour Sensitivity

3.4.1 Frequent Grammar Changes

Traditional parsing methods use heavy preprocessing of the grammar in order to speed up runtime performance. Such preprocessing techniques are appropriate when the grammar is fixed and when parsing would otherwise be too slow. Preprocessing usually means constructing some kind of automaton to recognize sequences of input tokens and/or nonterminals.

As we have already discussed, we are targeting applications that require frequent grammar changes. Such applications naturally include embedded languages such as SQL statements in a C program. Often, however, even though a single source language is involved, our approach can still be very useful. Traditional languages are usually described by context-free grammars, even though they are not actually context free.

We have already mentioned `typedef`s in C where the interpretation of a sequence of tokens depends on whether an identifier is a variable or a type. For instance, the token sequence `x * y` can be a variable declaration of `y` as a pointer if `x` is a type or an arithmetic multiplication expression if `x` is a variable. The traditional solution to this problem is to patch the lexical analyzer so that when a `typedef` has been seen, the corresponding identifier is subsequently considered to be a type name and not a variable identifier.

We also mentioned the `break` statement in C which is valid only inside loops. The usual solution to this problem is to always consider `break` a statement and then to make a second pass over the abstract syntax tree and reject its use in other contexts.

With our approach, such simple context sensitivity is naturally expressed within the framework of the grammar. Other examples include type verification of operands to operators, checking whether certain expressions are compile-time computable, and more.

All of these examples require that frequent grammar changes be handled efficiently. That requirement excludes heavy preprocessing of the grammar. Fortunately, parsing is now such a small fraction of language processing that the total time remains small even with a substantial increase in the time to parse the source.

Preprocessing of the grammar is not completely excluded. The method of Heering, Klint, and Rekers preprocesses the grammar incrementally [18]. Such an approach can be very efficient and can adaptively adjust to the frequency of grammar changes. That is, if grammar changes are very frequent, then only the part of the automaton that is needed is preprocessed. Otherwise, if grammar changes are infrequent, parsing becomes faster over time as more and more of the automaton is computed.

This research is primarily concerned with the mechanisms and style for extensible parsing. The tradeoff between (possibly incremental) preprocessing time (and software complexity) and parsing time for typical applications is left for future work. Research into the tradeoffs as applied to typical applications is complicated by the fact that, since few incremental grammars exist, there is not a corpus of typical applications to examine.

For this chapter, we avoid discussing these tradeoffs and concentrate on a purely interpretive approach, which unfortunately has poor worst-case behavior. Even so, it performs well in practice.

3.4.2 Simple Interpretation

The best way to minimize preprocessing is to avoid it altogether. We therefore represent the grammar as a collection of independent rules that are interpreted by the parser.

The parser itself uses backtracking whenever it fails to recognize a sequence of tokens. All possible rules for a nonterminal are tried in order until one succeeds[1].

[1] Our continuation-passing parser cannot handle left recursion (direct or indirect) and is not fully backtracking. However, it has no lookahead constraint, so it is more restricted

Notice that for very deeply nested definitions, our parser can require exponential time to recognize a sequence of tokens. A simple memoization trick could be used to avoid such behavior, but since the purpose of this implementation is only to demonstrate the feasibility of our framework and not to have an extremely fast parser, we have not implemented such optimizations. Despite this extremely bad worst-case behavior, the parser is actually sufficiently fast for most purposes.

Furthermore, the parser described here has the same limitation as LL parsers in that it cannot handle left recursion. We are currently working on parsing techniques that allow both fast parsing and the flexibility required for frequent grammar changes. In Chapter 4 we discuss the adaptation of the more powerful Earley [16] parsing algorithm to our environment passing framework.

3.5 Implementation of Continuation Passing Parser

3.5.1 General Description

In section 3.3.2 we gave a general description of the mechanism used to implement the parser. However, we deliberately did not expand on the actual mechanism for passing control among the parsers. In this section, we fill in the remaining aspects of the implementation.

In general, our implementation uses explicit continuation passing, in which each parser receives two continuations, one for success and one for failure. Thus, our implementation backtracks over failures and tries new possi-

than $LL(\infty)$ but differently limited than $LL(k)$ for any k.

bilities until it either fails at the top level or succeeds.

3.5.2 Representation of Rules

Since grammar changes are expected to be frequent, our representation of the grammar is one for which change operations are cheap. We also want a relatively high degree of interpretation relative to compilation.

We accomplish the goal of having inexpensive grammar changes by representing alternative rules for a non-terminal as a simple list structure. The list contains the names of non-terminals that are the constituent rules. This indirection through the name of the non-terminal is an essential part of our mechanism.

Notice that some trivial grammar transformations must be applied in order to fit this representation. For example, consider the following simple statement grammar:

$$\begin{array}{rcl}
\mathit{stmt} & \rightarrow & \text{``}\texttt{let}\text{''}\quad \mathit{var}\quad \text{``}\texttt{=}\text{''}\quad \mathit{expr}\quad \text{``}\texttt{;}\text{''} \\
\mathit{stmt} & \rightarrow & \text{``}\texttt{\{}\text{''}\quad \mathit{stmtlist}\quad \text{``}\texttt{\}}\text{''} \\
\mathit{stmt} & \rightarrow & \text{``}\texttt{print}\text{''}\quad \mathit{expr}\quad \text{``}\texttt{;}\text{''} \\
\mathit{stmtlist} & \rightarrow & \epsilon \\
\mathit{stmtlist} & \rightarrow & \mathit{stmt}\quad \mathit{stmtlist}
\end{array}$$

Each alternative for the non-terminals *stmt* and *stmtlist* is broken out into its own rule and given an arbitrary name. The non-terminal itself is made into an alternative sequence consisting of those constituents. Hence, the grammar is transformed into something like the following, where every entry is either alternatives among non-terminals or a sequence of items.

$$
\begin{array}{rcl}
\mathit{stmt} & \rightarrow & S_1 \quad | \quad S_2 \quad | \quad S_3 \\
S_1 & \rightarrow & \text{``}\mathtt{let}\text{''} \quad \mathit{var} \quad \text{``}\mathtt{=}\text{''} \quad \mathit{expr} \quad \text{``}\mathtt{;}\text{''} \\
S_2 & \rightarrow & \text{``}\mathtt{\{}\text{''} \quad \mathit{stmtlist} \quad \text{``}\mathtt{\}}\text{''} \\
S_3 & \rightarrow & \text{``}\mathtt{print}\text{''} \quad \mathit{expr} \quad \text{``}\mathtt{;}\text{''} \\
\mathit{stmtlist} & \rightarrow & L_1 \quad | \quad L_2 \\
L_1 & \rightarrow & \mathit{stmt} \quad \mathit{stmtlist} \\
L_2 & \rightarrow & \epsilon
\end{array}
$$

Right-hand sides that are simply or-separated sequences of non-terminal names are represented as lists of those non-terminal names. These lists are then interpreted by a special alternative-parsing procedure.

With this additional information, we can now describe the complete mechanism. The top-level parse procedure takes an input stream, a non-terminal to be parsed, an environment in which to interpret the non-terminal, a success continuation, and a failure continuation. All success continuations follow the same protocol. They take a meaning value, the remaining input stream, and a possibly modified environment.

Similarly, all failure continuations follow a protocol. They take a single argument, the input stream at the point of failure. The current implementation does not take advantage of this information, but it could be useful for error reporting.

The top-level parser procedure operates by looking up the non-terminal name in the given environment. This lookup could yield either an elementary parser or a list of alternative non-terminal names.

In the case of a list of alternative non-terminal names, the top-level parse procedure calls a special alternative-parsing procedure. The arguments to the specialized procedure are the input stream, the list of alternative non-terminal names, the environment for interpreting them, the success continuation and the failure continuation. The specialized alternative-parsing procedure calls the top-level parse procedure for each alternative in the list, passing a failure continuation whose effect is to continue with the alternatives. Since the success continuation is the same, when any alternative succeeds, the parse of the whole alternative sequence succeeds.

If the lookup does not yield a list of alternative non-terminal names, it yields an elementary parser. In this case, the elementary parser is called with the same arguments as the top-level parse procedure except that no non-terminal is needed.

A parser corresponding to a sequence of terminals and non-terminals uses a protocol dual to that of the alternative-parsing procedure. For an element in the sequence that is a terminal, the parser simply checks that the input stream contains that terminal and calls the failure continuation if not. If the input stream does contain the terminal, it simply continues with the next element in the sequence and the rest of the input stream. For elements in the sequence which are non-terminals, the parser calls the top-level parse procedure, passing a success continuation whose effect is to continue with the elements in the sequence. The failure continuation is the same as given for the whole parser, so when the parse of any element fails, the parse of the whole sequence fails. Similarly, when a non-terminal parse succeeds, the input stream from the success continuation is used to continue parsing the sequence.

Since each success continuation receives a parse environment from the recursive call, at any point in the sequence multiple parse environments are available. Hence, the elementary parser can implement the appropriate grammar contour by selecting the environment that implements the particular contour. It can even compute an entirely new environment if necessary.

3.5.3 Dynamic Rule Compilation

There are two ways of generating elementary parsers. A hand-coded procedure that follows the parser protocol may be inserted into the grammar by hand, thereby bootstrapping the system. Alternatively, they may be generated from grammar rules consisting of sequences of terminals and non-terminals.

3.5.4 Major Styles in Terms of Mechanism

Let us now return to our two major contour styles to explain how these styles can be implemented in terms of the mechanism described in this section. Recall that the sequence style requires that modifications to the environment be visible to succeeding language constructs. This is easily implemented by an elementary parser that uses the environment of the success continuation to parse the next element in the sequence. Thus, changes to the environment that occur during the parse of one element are visible during the parse of the next element.

Consider, for example, the following grammar fragment:

$$L \rightarrow S\ L$$

which might be part of a grammar for recognizing sequences L of top-level statements S. The recursive call to parse L would be handed the parse environment received by the success continuation that was passed to the recursive call to parse S. The parser for S would of course have to call its success continuation with an appropriately modified environment.

For block style, modifications to the environment are contained entirely within the scope of the construct. We implement this style by passing a modified environment to the call to the top-level parser for the language construct representing the body and then not using that environment any more.

Consider, for example, the following grammar fragment for a `while` construct:

$$W \rightarrow \texttt{while}\ E\ S$$

Here, E and S denote expression and statement constituents, respectively. The recursive call to parse S would be given a parse environment extended with the `break` construct. Neither this modified environment nor the one passed to the success continuation of this recursive call would be used again.

Since the `while` statement is designed to have no non-local effect on the environment, it calls its success continuation with the same parse environment with which it was called. Hence, it interacts with a containing parser for statements using the sequence style of contours by making no change. Similarly, if the S part of the `while` contains a sequence, non-local modifications within that sequence would not be passed on to the successor of the `while`. These two styles allow for a combination of sequences and blocks, where each block creates a new scope for a sequence within which to operate.

3.5.5 Examples

We now present an example to illustrate the above points. In this example, we walk through the operation of the continuation-passing extensible parser, using a simple statement grammar similar to that described earlier, including a `while` statement that implements loop termination by extending the set of valid statements to include `break`.

Several things are worth noting, as they limit the utility of this implementation in practice. First, this parser does not support full backtracking. That is, a parse decision in a local area of the grammar may consume input which is later necessary to make some other parse work. Second, the parsing activity is eager – it matches longer prefixes first.

The latter shows up in the sample grammar in the rule for *stmtlist*. The production L_1 must occur before L_2, since L_2 matches anything and hence *stmtlist* is always empty.

```
{
  let x = 5;
  while (p(x)) {
    if (x <= 0)
      break;
    x = f(x);
  }
}
```

In this parse tree, for brevity we elide the non-terminals that denote alternatives, *e.g.*, *stmt* and *expr*. Instead, the alternative that succeeds is shown.

Here is the complete grammar we'll use for this example:[2]

$$
\begin{array}{rcl}
\mathit{stmt} & \rightarrow & S_1 \;\mid\; S_2 \;\mid\; S_3 \;\mid\; S_4 \;\mid\; S_5 \;\mid\; S_6 \\
S_1 & \rightarrow & \text{“}\texttt{let}\text{”} \quad \mathit{var} \quad \text{“}\texttt{=}\text{”} \quad \mathit{expr} \quad \text{“}\texttt{;}\text{”} \\
S_2 & \rightarrow & \text{“}\texttt{\{}\text{”} \quad \mathit{stmtlist} \quad \text{“}\texttt{\}}\text{”} \\
S_3 & \rightarrow & \text{“}\texttt{print}\text{”} \quad \mathit{expr} \quad \text{“}\texttt{;}\text{”} \\
S_4 & \rightarrow & \text{“}\texttt{if}\text{”} \quad \text{“}\texttt{(}\text{”} \quad \mathit{expr} \quad \text{“}\texttt{)}\text{”} \quad \mathit{stmt} \\
S_5 & \rightarrow & \text{“}\texttt{while}\text{”} \quad \text{“}\texttt{(}\text{”} \quad \mathit{expr} \quad \text{“}\texttt{)}\text{”} \quad \Delta^1 \quad \mathit{stmt} \\
S_6 & \rightarrow & \mathit{var} \quad \text{“}\texttt{=}\text{”} \quad \mathit{expr}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{expr} & \rightarrow & \mathit{testexpr} \;\mid\; \mathit{funcallexpr} \;\mid\; \mathit{varexpr} \;\mid\; \mathit{literalexpr}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{stmtlist} & \rightarrow & L_1 \;\mid\; L_2 \\
L_1 & \rightarrow & \mathit{stmt} \quad \mathit{stmtlist} \\
L_2 & \rightarrow & \epsilon
\end{array}
$$

When the parse starts, the main parse procedure is invoked with *stmt* as the goal non-terminal, the token string as input, the default environment which contains the grammar and any global top-level variables, a success procedure for accepting the input, and a failure procedure to reject the input.

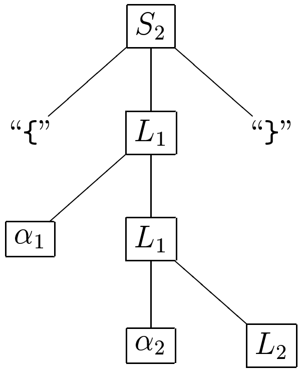
In the initial grammar, *stmt* is bound to a sequence of alternatives, S_1, ..., S_6. For *stmt*, the main parse procedure dispatches to the alternative-list

[2] Since this is a recursive descent parser, the actual representation of the *expr* non-terminal has had its left recursion eliminated, a mechanical process described by Aho *et al.* [1] which produces a lengthy grammar. For brevity, since we are not concerned here with expression parsing, we show *expr* in its unfactored form.

parsing procedure, which in turn attempts to parse S_1, ..., S_6 with the same input it was given, the same success procedure, and a failure procedure that goes on to the next alternative.

In this case, S_1 immediately fails when it tries to match "`let`". The failure procedure returns to the alternative-list process to try the next one, *i.e.*, S_2.

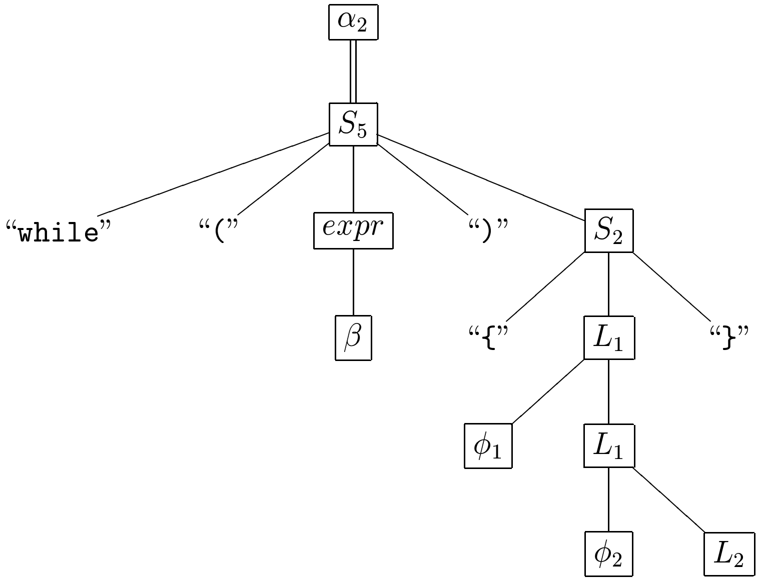
S_2 starts by matching the open brace, and then trying to parse a *stmtlist*, which succeeds. In the process of succeeding, the *stmtlist* breaks down into a sequence of L_1 nodes (one for each statement parsed) with an L_2 at the end, as shown here:



where α_1 is the `let` statement, and α_2 is the `while` statement. As mentioned earlier, the behavior of the L_1 elementary parser is to use the environment passed to the success procedure as the environment for the remainder *stmtlist*.

That is, α_1 produces an environment that is its input environment augmented with a binding for the variable `x`, and the L_1 parse that contains α_1 passes that augmented environment along as context for the parse of α_2. This is an example of the sequence grammar change style as applied to the program environment (here it's not being used to change the grammar *per se*, but the environment and grammar are kept together).

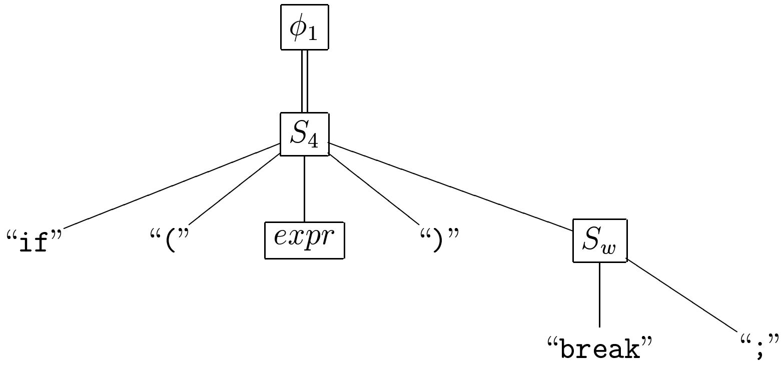
When it comes to parsing the while statement itself, which is α_2, we illustrate how block-style grammar changes take place. In this case, the parse tree looks like this:



The elementary parser for S_5 parses the first 4 items as usual but then passes an extended grammar to the *stmt* parse which matches S_2 in this case. The extended grammar includes a new statement non-terminal, S_w, and a redefinition of *stmt* which includes S_w. That is, in the extended environment, the grammar includes:

$$\begin{array}{rcl} \mathit{stmt} & \rightarrow & S_w \mid S_1 \mid S_2 \mid S_3 \mid S_4 \mid S_5 \mid S_6 \\ S_w & \rightarrow & \text{“}\texttt{break}\text{”} \quad \text{“};\text{”} \end{array}$$

Now when parsing the statements in the body of the `while`, *e.g.*, ϕ_1, the `break` statement is recognized and parsed, producing:



Once the parse of ϕ_1 is complete, the extended environment is discarded – it is not passed to the success procedure of S_5 itself, since it is not the intention of the `while` construct to manipulate the environment or grammar of a sequence in which it occurs.

3.5.6 Performance

The parser described in this chapter is intended to be an illustration of the capabilities of an environment-passing parser, and not an exemplar of performance. In fact, the parser described here is exponentially slow in some cases. The implementation is only intended to illustrate an application of the framework and the general ideas of interpreted extensible parsing. Despite its potential slowness, however, the implementation is simple enough that it performs tolerably well in practice. Some care in constructing the grammar for efficient recursive-descent parsing leads to quite reasonable performance.

Chapter 4

Extensible Earley Parsing

4.1 Introduction

This chapter describes the adaptation of a relatively efficient parsing algorithm, that of Earley [16, 15], for the purpose of extensible parsing. The Earley algorithm is essentially an interpreted version of a table-driven parser. However, the Earley algorithm can handle all context-free grammars, including ambiguous ones, and it is relatively efficient. In fact, even though the Earley algorithm is interpretive in nature, it achieves computational bounds commensurate with that of other parsers that do a lot of pre-compilation.

4.2 Description of Earley Parsing

Like an automaton built by a parser generator, the Earley parsing technique works by keeping track of a set of rules that are in the process of being recognized. However, instead of interpreting an abstraction (*i.e.*, an automaton) of the possible rules that might be recognized, the Earley technique tracks the actual, concrete rules that describe the input seen so far.

For example, the states of an LR parser represent the same set of partially recognized rules as an Earley parser does. However, an LR parser generator pre-computes the possible sets and the possible transitions between sets for all valid inputs, while the Earley technique computes the set at parse time. Computing the set at parse time is important for our purpose, since we are changing the grammar during the parse process.

A language is characterized by its vocabulary, which is a set of symbols used to describe valid sentences in the language. Terminal symbols are those that appear in a sentence, and non-terminals are those that are used in the grammar to describe the abstract forms of the language.

The grammar is a set of *rules.* Each rule consists of a non-terminal and a sequence of vocabulary symbols. For example, a rule which consists of the non-terminal N and the sequence of vocabulary symbols a, B, c, D is written:

$$N \rightarrow a\, B\, c\, D$$

We use the convention that lower case letters denote terminals and upper case letters denote non-terminals. We also use ϵ to denote an empty right-hand side.

When describing the state of a parse in progress, a rule is written with a dot in it to denote the current position of the parse activity. This is called an *item.* For example,

$$N \rightarrow a\, B\, \cdot\, c\, D$$

indicates a rule undergoing a match, wherein an a and a B have been recognized, and a c is about to be recognized.

4.2.1 States

Each state of an Earley parse is an item together with a *back pointer* which refers back to the position in the input which gave rise to the rule. In the discussion about pure Earley parsing, we write states as illustrated by this example:

$$\langle N \rightarrow a\, B\ \cdot\ c\, D, 3\rangle \tag{4.1}$$

This denotes the state with rule $N \rightarrow a\ B\ c\ D$, with the dot at position 2 (*i.e.*, with 2 elements to the left of the dot). In particular, this state is saying that an a and a B have been recognized so far, and that a c and a D must be recognized next in order for an N to be recognized. State 4.1 also has a back pointer to position 3 in the input, which means that the recognition of this occurrence of an N began at position 3 in the input.

4.2.2 State Sets

Each position in the input (N+1 of them for an input of length N) is associated with a set of states. The set of states encodes both recursion and parallelism. The recursions correspond to attempts to expand non-terminals in the right-hand side of a rule. The parallelism corresponds to the different possible expansions for a particular non-terminal.

For example, in the following grammar:

$$\begin{array}{rcl} \mathit{pgm} & \rightarrow & \mathit{stmt}\ \mathit{pgm} \\ \mathit{pgm} & \rightarrow & \epsilon \\ \mathit{stmt} & \rightarrow & \mathit{ifstmt} \\ \mathit{stmt} & \rightarrow & \mathit{callstmt} \end{array}$$

when a *stmt* is being recognized, there are two possibilities (*ifstmt* and *callstmt*), which are tracked in parallel in different subsets of the state set.

4.2.3 Initial Conditions

Initially, the state $\langle \phi \rightarrow \cdot S, 0 \rangle$ is the content of the state associated with position 0. This state represents the circumstance that, initially, nothing has yet been recognized, and the entire program (represented by the start symbol S) is about to be recognized.

4.2.4 Processing

Processing proceeds by, for each successive position in the input, iterating the operations of completion and prediction on a state set until a fixed point is reached. Then the scan operation is used to construct the initial contents of the state set for the next position in the input[1].

In pseudo-code, processing works like:

```
$ssa[0] = initial();$
$i = 0;$
**while**$(i < length(input))$
  **begin**
    $iterate(ssa[i], i);$
    $ssa[i+1] = scan(ssa[i], input[i]);$
    $i = i + 1;$
  **end**;
```

[1] This description corresponds to our implementation but differs slightly from Earley's explanation in that the next state is built while processing the current state.

where *ssa* is the array of state sets and *input* is the array of input tokens.

In this implementation, *ss* acts like an array for the purposes of iteration (that is, new entries are added to the end) and like a set for purposes of duplicate elimination. With this in mind, one iteration step looks like:

```
procedure $iterate(ss, i)$
  $k = 0;$
  while$(k < length(ss))$
    begin
      $ss = ss \cup derived(ss[k], i);$
      $k = k + 1;$
    end
```

When a valid input sentence of length n has been processed, the contents of $ssa[n]$ contain a state $\langle \phi \to S\ \cdot, 0 \rangle$. This indicates that an entire S has been recognized, starting at the beginning of the string. If a state of this form does not appear in $ssa[n]$, then the input string was not a sentence in the language.

Computing the derivatives of a state (the *derived* function) depends on what is to the right of the dot. There are three cases to consider: (1) there is a non-terminal to the right of the dot, (2) there is a terminal to the right of the dot, and (3) there is nothing to the right of the dot. These are described in detail in the following sections. Note that case (2) is not handled by *derived*; it is handled by the call to *scan* in the main processing loop.

4.2.5 Prediction

Prediction is an attempt to match a non-terminal, and occurs when a state has a non-terminal to the right of its dot. For example, consider the state:

$$\langle N \rightarrow a \ \cdot \ B\, c\, D, 3\rangle$$

This state represents a parse in progress in which an a has just been seen, and a B is expected. The processing of this state within a state set (*i.e.*, the behavior of the *iterate* function on this state) introduces new states corresponding to the possible expansions for B.

In general, if A is the non-terminal to the right of the dot in a state s and i is the current position in the input, then for each rule r of the form $A \rightarrow \beta$, the result of *iterate* contains $\langle A \rightarrow \cdot\beta, i\rangle$. Note that β could be empty, in which case this new rule is completable within this state set. Furthermore, if β is empty, then we have to be careful to make sure that the completion does happen for the state s being processed. Otherwise, it might occur that $A \rightarrow \epsilon$ has already been introduced and completed within this state set on behalf of some other non-terminal B. Duplicate elimination within ss would keep s from processing the completion of B. We handle this as a special case of a prediction that results in an ϵ-rule state that is already in the state set.

```
procedure $predict(s)$
  $A = s.next;$
  $result = \emptyset;$
  for $r$ in $G(A)$
    $result = result \cup \langle r, k\rangle;$
  return $result;$
```

```
end
```

where k is the input position corresponding to the current state, and $G(A)$ denotes the lookup of the non-terminal symbol A in the grammar G, which returns a list of rules. $s.next$ is the symbol after the dot in s.

4.2.6 Scanning

Scanning is the process of matching a token in the input string, and occurs when a state has a terminal to the right of its dot. Scanning is the means for initializing the contents of the state set corresponding to the next position in the input stream. Successful scanning advances the dot past a terminal, and corresponds to consuming an input token.

To scan a state s,

$$s = \langle N \rightarrow \beta_1 \ \cdot a\, \beta_2, j\rangle$$

provided that a appears next in the input, then the state set for the next input position includes:

$$s' = \langle N \rightarrow \beta_1\, a \ \cdot \beta_2, j\rangle$$

```
procedure $scan(ss, token)$
  begin
    $result = \emptyset$;
    for $s$ in $ss$
      if $s.next \in T$
         and $match(s.next, token)$ then
          $result = result \cup s'$;
    **return** *result*;
  **end**
```

where T is the set of terminal symbols, and s' is the same as s but with the position of the dot advanced by one.

4.2.7 Completion

Completion is analogous to reduction in a traditional parser. A state can be completed when an item has its dot to the right; that is, there is nothing after the dot.

Unlike reduction in a traditional parser, in Earley parsing meanings (*e.g.*, abstract syntax trees) are not computed during completion. Instead, the parser maintains a record of the completions that took place during parsing and when parsing is complete, that record is analyzed to construct the corresponding meaning. The reason for this delayed approach to computing meaning is to maintain polynomial complexity bounds in the face of ambiguous grammars. That is, since Earley can parse sentences using an ambiguous grammar, computing the meaning during the parse can lead to a state explosion which would degrade Earley's asymptotic complexity results to exponential instead of $O(n^3)$.

To compute the derivative states of a completable state s, where:

$$s = \langle N \to \beta \cdot, j \rangle$$

$ssa[j]$ is examined to determine which states led to state s. In particular, the states leading to s are those states in $ssa[j]$ with an N to the right of the dot because those created the prediction that we are now completing. For each state in $ssa[j]$ of the form:

$$\langle M \rightarrow \beta_1 \, \cdot \, N \, \beta_2, m \rangle$$

the derived set of s includes $\langle M \rightarrow \beta_1 N \cdot \beta_2, m \rangle$. Note that β_2 could be empty, in which case another completion takes place, for M this time.

Thus, completion of a state s is the process whereby the dot is advanced past a non-terminal in some other state m. The state m is in the state set corresponding to the input position referred to by the back pointer of state s.

4.2.8 Relationship to Tomita Parsing

The Tomita [40] approach to parsing is worth mentioning because it shares the ability of the Earley parser to parse sentences using arbitrary context-free grammars and uses a similar parallel approach. However, a Tomita style parser creates explicitly parallel parsers when an uncertainty is encountered. However, to avoid exponential blowup, a Tomita parser must then go through extra work to join together different forked parsers that reach the same point in parsing through different paths.

The Earley approach is generally equivalent to the Tomita approach if the explicitly parallel parsers are regarded as being implemented using user-level threads with light-weight state and a specialized thread scheduler which advances each thread in lock-step with the input stream. The back pointers in the state link together the “stack” of activations of the thread.

4.3 Advantages to the Earley Approach

4.3.1 Flexibility

One of the main advantages of the Earley approach to parsing is that it gracefully handles arbitrary context-free grammars. Traditional wisdom is that large classes of grammars can be converted into a convenient canonical form such as LR or LL. However, this canonicalizing conversion often distorts the resulting concrete tree, which makes it difficult to compose meanings.

For example, consider the following grammar G for array references:

$$\begin{array}{lll} E & \rightarrow & E \quad \text{“}\texttt{[}\text{”} \quad E \quad \text{“}\texttt{]}\text{”} \\ E & \rightarrow & id \end{array}$$

This can be converted into LL form using the techniques described in Aho *et al.* [1]. Doing so results in a grammar G':

$$\begin{array}{lll} E & \rightarrow & id \quad E\text{'} \\ E\text{'} & \rightarrow & \text{“}\texttt{[}\text{”} \quad E \quad \text{“}\texttt{]}\text{”} \quad E\text{'} \\ E\text{'} & \rightarrow & \epsilon \end{array}$$

However, when presented with the sentence `x[y]`, grammars G and G' behave very differently. G' generates the parse tree shown in Figure 4.1.

Since E' is not an expression – in fact, it has no correspondence to the constructs in the language – it is difficult to associate a meaning with it that corresponds to some semantics of the language. In order to make use of grammar G', an artificial meaning must be assigned to E'. This artificial meaning must then be taken apart in order to build the correct meaning for the expression E.

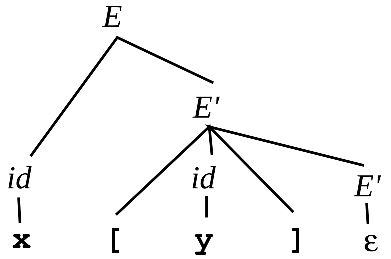


Figure 4.1: Parse tree for `x[y]` with grammar G'.

As a consequence, the meaning computations in G' must break apart the meanings of their constituent parts. On the other hand, if G could have been parsed directly, its meaning computations would be simple compositions of their parts, each with a direct correspondence to the semantics of the language.

Furthermore, many techniques for producing efficient parsers for static grammars rely on being able to compute global properties of the grammar. For example, token lookahead sets are one of the first tricks for improving parser efficiency, and the computation involves the entire grammar. This is difficult to do when the grammar keeps changing, as in an extensible language.

4.3.2 Extensibility

As mentioned previously (c.f. 4.2), an Earley parser explicitly processes the rules of a context-free grammar in its original form. This makes the parser easy to extend, because it is clear what needs to happen when a grammar change takes place – the grammar representation is simply updated. There is no additional processing necessary, since the parser operates on a direct representation of the grammar.

An Earley parser can also be thought of as processing multiple candidate parses in parallel. We can achieve scoped, local grammar changes by augmenting the states of these parses with a *parse environment* that contains the grammar. Hence, the parser can be parsing *multiple grammars at once.* Consider, for example, the following grammar:

$$\begin{aligned} A &\rightarrow \texttt{a}\, B\, \texttt{b} \\ A &\rightarrow \texttt{a}\, \Delta_1\ B\, \texttt{c} \end{aligned}$$

In this example, the interpretation of B depends on what follows it[2]. That is, a B followed by a c is to be interpreted under the influence of grammar change Δ_1.

The Earley parser would normally introduce two states to correspond to the two active possibilities:

$$\begin{aligned} \langle A &\rightarrow \cdot\, \texttt{a}\, B\, \texttt{b}, k\rangle \\ \langle A &\rightarrow \cdot\, \texttt{a}\, B\, \texttt{c}, k\rangle \end{aligned}$$

If the grammar augments Earley parsing state, then it's clear how this can work:

$$\begin{aligned} \langle A &\rightarrow \cdot\, \texttt{a}\, B\, \texttt{b}, k, G\rangle \\ \langle A &\rightarrow \cdot\, \texttt{a}\, B\, \texttt{c}, k, \Delta_1(G)\rangle \end{aligned}$$

where $\Delta_1(G)$ denotes the influence of the grammar change on the original grammar G.

[2] Doing this a lot in a language can make it very difficult to read, because you can't understand a program in a straightforward left-to-right manner. However, several languages have some flavor of this when it comes to exceptional cases, specifically because they want to keep the thinking for exceptional cases out of the way of the normal case. In any case, it's interesting that an extensible Earley parser can handle this!

Cardelli's [9] mechanism for syntactic extension at the surface grammar level operates at a global level. Our approach makes possible local and scoped changes to the grammar.

So, our approach adds a *parse environment* to the state which includes the grammar. In addition, since we do not want to run back over the parse states to build a parse tree, and we aren't worried about parsing ambiguous grammars, we collect the meanings in the parse states as well. Hence, our states look like:

$$\langle N \rightarrow a\, B\ \cdot\ c\, D, 3, \langle \mu_0, \mu_1 \rangle, \alpha \rangle$$

which denote the same states as in (4.1) but with accumulated meanings $\langle \mu_0, \mu_1 \rangle$ and in environment α.

The meaning sequence $\langle \mu_0, \mu_1 \rangle$ is exactly as long as the position of the dot, and its elements are in correspondence with the vocabulary symbols to the left of the dot. In this example, μ_0 is the meaning resulting from the parse of a and μ_1 is the meaning resulting from the parse of B.

4.3.3 Understandability

The Earley parsing approach is a straightforward implementation of a parser for general context-free grammars. A straightforward implementation makes system development, debugging, and maintenance more cost effective. Furthermore, despite the direct implementation, an Earley parser is not inefficient in the common case. Before Earley, generic parsers for context-free grammars were somewhat less efficient even when operating on unambiguous grammars.

4.3.4 Complexity

The theoretical (asymptotic) complexity of the Earley approach is not bad, and it is dependent on the class of grammar on which it is operating. According to Earley [16], the algorithm is $O(n^3)$ in the worst case. For unambiguous grammars, an $O(n^2)$ bound is obtained.

Indeed, for many practical language grammars, it appears that $O(n)$ performance is expected. The latter class of grammars are those for which the size of the state set does not grow with the length of the input string. Earley calls these grammars *bounded state* grammars. Furthermore, bounded state grammars include most LR(k) grammars as well.

4.4 Drawbacks to the Earley Approach

4.4.1 Expressiveness

If we weren't concerned about the issue of programmer extensibility, then one drawback to the Earley approach is that it may be *too* general. That is, most of its flexibility is wasted because it seems that many kinds of grammar changes – indeed, the most structured and hence most understandable ones – have no greater expressive power than static context-free grammars. This is true because the dynamic grammar can be converted into an equivalent (although somewhat larger) static grammar by appropriate sub-grammar expansions and substitutions[3].

However, we take the position that a grammar expressed as a dynamic

[3] Although we did not work out the theoretical details of this process, a few casual translations suggested that the process is similar in spirit to how non-deterministic finite automata are converted into equivalent deterministic automata.

grammar is more clear and natural. For example, taking the example of `while` and `break`, it is somewhat clearer to define statements in general and express the exception case that `break` is a valid kind of statement inside a `while` body. The alternative is to explain that there are two sets of statements – those inside `while` and those not – and that the two sets are the same except that `break` is in one set and not the other. This isn't a difficult factoring problem if the new kind of thing (`break` in this case) were only allowed directly inside the `while`. If that were the case, the programmer could simply write:

$$
\begin{aligned}
\mathit{stmt} &\rightarrow \texttt{while}\quad \text{``}\texttt{(}\text{''}\quad \langle c\!:\mathit{expr}\rangle\quad \text{``}\texttt{)}\text{''}\\
&\qquad\quad \text{``}\texttt{\{}\text{''}\quad \langle \mathit{conseq}\!:\mathit{while\_stmt\_list}\rangle\quad \text{``}\texttt{\}}\text{''}\\
\mathit{while\_stmt\_list} &\rightarrow \mathit{in\_while\_stmt}\quad \mathit{while\_stmt\_list}\\
\mathit{while\_stmt\_list} &\rightarrow \epsilon\\
\mathit{in\_while\_stmt} &\rightarrow \mathit{stmt}\\
\mathit{in\_while\_stmt} &\rightarrow \text{``}\texttt{break}\text{''}\quad \text{``}\texttt{;}\text{''}
\end{aligned}
$$

However, a `break` statement is allowed inside another statement in the body of the `while`, for example, an `if` statement. With some statements using substatements as constituents, the entire *stmt* sub-grammar has to be replicated and translated to account for the addition of the `break` statement. This explosion, which multiplies with each statement with this property, is why static semantics as a compiler construction technique is so widely accepted.

4.4.2 Performance

Although the theoretical performance (asymptotic complexity) of the Earley approach is good, especially considering its flexibility, the practical performance is less impressive. In practice, the performance suffers from the difficulties of all interpreters; systems that make heavy use of compile-time information (and compile-time static knowledge) to build efficient parsers can achieve better performance than those that don't. For example, the `yacc` parser generator spends a relatively large amount of time preprocessing the grammar in order to build a structure that is very fast to interpret at runtime. An Earley parser does not utilize precomputation of this sort, and hence it must recompute the same kinds of information at parse time, incurring that time cost for each program compiled.

Despite somewhat less efficiency than a parser like `yacc`, an Earley parser can achieve sufficient performance for practical purposes. Our implementation written in C parses a 600-line Java program in about 44ms on an 866MHz Pentium III. Considering the flexibility that this approach enjoys, this seems plenty fast for production use.

4.5 Extensibility

4.5.1 Scope Issues

An extensible parser can be thought of as dealing with multiple static grammars. Each grammar change creates a new grammar for parsing a portion of the input program. The scope issue to be resolved is determining in which grammar a particular non-terminal being recognized is to be interpreted.

The *scope* of a grammar change is the set of non-terminal symbols over which the change applies[4].

[4] Since we use a conventional separate lexical scanner, we don't allow changes to the lexical analyzer and hence can take the meaning of terminal symbols as fixed and global.

What does scope look like in an Earley parser?

The same scope issues that arise in hygienic macro expansion come up in extensible parsing, and we show how those issues are addressed.

However, when the grammar is no longer a static entity, an additional scope issue arises, which is: What is the scope of a change to the grammar? An example of the scope issue is in the parsing of something like Java's `while` statement where `break` is a valid statement inside `while` but not elsewhere. The scope of the grammar change that makes `break` a valid statement should only be the body of the `while`, not its expression part and not statements that follow the `while`.

The usual approach for handling this sort of construct is to make `break` be a valid statement in all contexts and then to check the *static semantics* of the program after parsing is complete or using parse-time attributes (in fact, there is a whole discipline for defining static semantics [42].)

Our approach uses the ability to extend the grammar during parsing. When a `while` statement is encountered, the rule for the statement can make a change to the grammar for the duration of the parsing of its body. This change is the extension of the set of valid statements to include `break`.

Hygiene in an Earley parser

Grammar changes in our approach have dynamic extent and indefinite scope. In this respect they are like special variables in Lisp. A grammar rule anywhere may make use of a grammar change as long as the change is still active.

Future work should look at relaxing this restriction – there is some suggestion that with some optimizations for the purpose, scanning could take place in the grammar itself!

4.6 Our Implementation

We have two implementations of an extensible Earley parser. One is written in Scheme and is used for testing new ideas and validating concepts. The other is written in C and is intended primarily for benchmarking, so that the interpretive overhead of the Scheme implementation doesn't overshadow the actual performance of the parser.

The Scheme implementation is designed for flexibility and not for speed. As a test bed for concepts, it is more convenient to work out how a feature can be implemented in a highly dynamic environment such as Scheme than in a pedantic, low-level environment like C. This implementation does not engage in any of the performance optimizations described later[5]. The C implementation, by contrast, is tuned for performance. It uses standard systems programming optimizations to implement the algorithm efficiently.

4.6.1 Details

Literal Equivalence

In our current implementation, we do nothing special to handle literal equivalence. Literal equivalence for identifier tokens is determined by equivalence of their characters. As a result, we get a kind of context-sensitive keyword determination. An identifier plays the role of a syntactic keyword if it occurs at a place where one is expected.

[5] The Scheme implementation does employ the optimization described in section 4.8.2, pruning states using FIRST. This optimization is only because we wanted to exercise the FIRST-pruning optimization in the test bed before implementing it in the low-level implementation. We also use the Scheme implementation to precompute the FIRST sets when grammars are being precompiled.

Note that this behavior, combined with poor programmer practice, can lead to programs that are difficult to read. PL/I, for example, exhibits this kind of context-sensitive keyword identification [20]. Consider:

if(x) = then + then; (1)

if(x) = then + then then; (2)

In this example, (1) is an assignment of the sum "then+then" to the array "if", and (2) is a test of the value x against the sum "then+then". To mitigate this problem, we implement keywords as distinguished entities, but they are scoped to the grammar change module which introduces them.

Disambiguation

Our implementation computes meaning as the parse progresses, instead of in a post-pass as traditional Earley parsing does. To prevent an explosion of states, we use a disambiguating rule to collapse ambiguous cases. This is discussed more fully in section 4.8.1.

4.6.2 Meta-syntax

The notation we use is essentially standard BNF (*e.g.*, see Aho *et al.* [1]). Occurrences of vocabulary elements on the right-hand side of a rule are given variable names to represent the meaning for the corresponding parsed input element during the computation of the meaning for the whole rule[6]. For a more complete explanation of our meta-syntax, see Section 5.8.

For a simple example, a rule for an **if** statement might be:

[6] These variable names play the same role as the `$`k variables in the action part of `yacc` grammar rules.

$$\mathit{ifstmt} \rightarrow \texttt{if}\ \texttt{(}\ \langle c\!:\!\mathit{expr}\rangle\ \texttt{)}\ \langle \mathit{conseq}\!:\!\mathit{stmt}\rangle\ \texttt{else}\ \langle \mathit{alt}\!:\!\mathit{stmt}\rangle$$

The typographical distinctions herein which signal different roles for identifiers, such as between `if` and *ifstmt*, are written in the meta-syntax with quotes. Other special symbols are entered using existing operator symbols such as <, > and :. The above rule might be written in our meta-syntax as:

```
ifstmt = "if" "(" <c:expr> ")"
         <conseq:stmt>
         "else" <alt:stmt>
```

This defines a rule called `ifstmt`, which begins with a syntactic literal (`if`), followed by an expression `c` in parentheses, followed by a statement (`conseq`), followed by another syntactic literal (`else`), with another statement `alt` at the end.

It is necessary in this implementation to use existing operator symbols because the tokenizer is not integrated with the parser. This implies that tokenization is not an extensible facility in this implementation. Hence, all meta-syntactic notation must either use the existing tokens of the language being implemented or add new tokens to the language globally. In general, we do not want to change the lexical structure of the base language, so existing token forms are used to express meta-syntactic forms.

Future work includes the incorporation of an extensible scanner. The ability to extend the lexical structure in a local and scoped way makes certain kinds of language extensions much more natural and eliminate the necessity to use existing tokens in meta-syntactic forms. It should also allow user extensions to the language to specify new lexical categories.

4.6.3 Meaning Computation

Because we are compiling and not just recognizing, we have to associate a meaning with grammar rules. We implement two ways of computing a new meaning; a primitive, procedural meaning function, and a recursive compilation to produce new meaning.

The following definition for the traditional `if` statement illustrates the notation used in this discussion:

$$
\begin{aligned}
\textit{ifstmt} \rightarrow\ & \texttt{if (} \langle c : \textit{expr} \rangle \texttt{) } \langle \textit{conseq} : \textit{stmt} \rangle \\
& \texttt{else } \langle \textit{alt} : \textit{stmt} \rangle \\
& \Rightarrow \mathsf{makeif}(c, \textit{conseq}, \textit{alt})
\end{aligned}
$$

In this example, the non-terminal is *ifstmt*, the pattern is:

$$\texttt{if (} \langle c : \textit{expr} \rangle \texttt{) } \langle \textit{conseq} : \textit{stmt} \rangle \texttt{ else } \langle \textit{alt} : \textit{stmt} \rangle$$

where c, *conseq*, and *alt* are pattern variables. The expression for computing the meaning is:

$$\mathsf{makeif}(c, \textit{conseq}, \textit{alt})$$

This example also illustrates using an extra-grammatical function, makeif, to compute the overall meaning of the complete *ifstmt*. Taking as arguments the appropriate pattern variables, makeif builds, using the facilities of the underlying metalanguage, a complete meaning value.

The simple functional form of this meaning computation is what we mean by *composition of meaning*. That is, the new meaning is composed from the old (lower-level) meanings.

Since the parser is itself a mechanism for computing meanings, the second approach simply reuses that mechanism. In this case, the replacement text and a grammatical type is handed back to the parser/compiler in a recursive sub-parse:

$$\begin{aligned} \mathit{ifstmt} \rightarrow{} & \texttt{if}\ \texttt{(}\ \langle c\,{:}\,\mathit{expr}\rangle\ \texttt{)}\ \langle \mathit{conseq}\,{:}\,\mathit{stmt}\rangle \\ & \overset{\mathit{stmt}}{\Longrightarrow} [\![\ \texttt{if}\ \texttt{(}\ c\ \texttt{)}\ \mathit{conseq}\ \texttt{else}\ \{\ \}\ ]\!] \end{aligned}$$

Here, the meaning for a single-branch `if` (*i.e.*, an `if` statement without an alternative part) is determined by computing the meaning for the program fragment given in double-brackets. The portion inside the double-brackets is referred to as the *template*. The template, much like a quasiquote form in Lisp, acts like a token string with holes. In this case, the token string denotes a full two-branch `if`.

The holes are filled in with parts taken from the original form. In this example, *c* and *conseq* are pattern variables and denote the meanings resulting from the parse of an *expr* and a *stmt*. When these tokens are encountered during the parse, the meaning obtained during the parse of the left-hand side is immediately substituted, and it is as if the production were complete.

Note that this is essentially the same approach used in compiling with Scheme macros, except that the meanings of the constituent parts are determined by the pattern rather than by their use in the template or body part. Recall that in Scheme macros, the occurrence of a pattern variable in a template represents the parse tree for the corresponding part of the input, coupled with a compile-time environment to protect hygiene. In our parsing approach, the pattern variable denotes a complete meaning, such as an abstract syntax tree.

The assignment of meaning during pattern recognition is necessary, or at least convenient, because there are no other syntactic cues to guide a surface parse of an occurrence of the macro. In Scheme, by contrast, the parenthetical list representation is a simple syntax which requires no knowledge of macros or meanings to build trees out of source text.

One consequence of this approach is that grammar changes that only become apparent during the parse of the right-hand side (the replacement text) are not known during the parse of the left-hand side (the occurrence text).

One environmental interaction that comes up in extensible parsing but doesn't manifest in Scheme macro processing is the declaration of the surface syntactic types of the arguments. For example, in a pattern declaration:

$$\mathit{forstmt} \rightarrow \texttt{for}\ \langle v : \mathit{var} \rangle\ \texttt{=}\ \langle e : \mathit{expr} \rangle\ \ldots$$

var is a kind of variable reference in the *pattern* that does not manifest in Scheme macro processing, *i.e.*, it is a reference to a non-terminal name. In Scheme macros, there are no names given to syntactic roles in a pattern. That is, there are no pattern abstractions in Scheme macros.

Furthermore, the meaning of the identifier *var* should be determined in the environment of use, even though it occurs in the environment of definition. Thus, for example, the user can change the syntax of *expr*. That change is visible and used by any rule which uses *expr* in its pattern, even if the rule has no knowledge of the possibility that *expr* has changed.

However, the *template* is interpreted in the environment of definition, preventing local changes to the grammar from affecting the syntactic interpretation of the template[7]. For example, let us suppose that in addition to the `if` statement mentioned previously there is a construct, `with`, which adds a

[7] This is like the second part of the hygiene condition; references (implicit in the syntactic construction) in the template should not be captured by bindings other than those present at the point of macro definition.

new kind of expression, it, to the grammar:

$$\mathit{withstmt} \rightarrow \texttt{with (} \langle e : \mathit{expr} \rangle \texttt{) } \Delta_1[\ \langle \mathit{body} : \mathit{stmt} \rangle\ ]$$

where Δ_1 represents a grammar change whose scope is the bracket-delimited portion of the rule and whose effect is to add the symbol it to the alternatives for *expr*.

Now consider an input program fragment such as:

```
with (foo())
  if (it)
    print(it);
```

The with establishes a new kind of *expr* which consists of the symbol it. Even though the *ifstmt* rule was written with no knowledge of the future *withstmt* rule the occurrence of *expr* in *ifstmt* should consider it to be a valid expression. This is true even though the occurrence of *expr* in the pattern *ifstmt* is well outside the static scope of the change made by Δ_1.

Hence, although we cannot implicitly carry a grammar change from the right-hand side (template) into the left-hand side (pattern) of a translation rule, we can isolate productions that make use of a non-terminal N from the local effects of redefining N.

4.6.4 Performance

Practical measurements have been made of Earley parsing in the context of improving its performance using precomputation approaches [30]. Unfortunately, since that work did not give the implementation of unmodified Earley, it is difficult to draw conclusions about the general performance of the unmodified Earley approach.

Our implementation is designed from the start to be fast, since one of our points is that the Earley approach is not prohibitively expensive. Our C implementation uses traditional performance-improving techniques such as inlining, as well as carefully chosen data structure representations. Furthermore, since modern compilers spend much more time in optimization and other back-end processing than in front-end processing like parsing, if the parser is a bit slower because it is completely interpretive (as in Earley), the overall system cost is still not prohibitive.

We don't have a large body of language implementations designed to be extensible. Therefore, in order to understand the performance implications of our extensible grammar approach, we made a pessimistic estimation of the frequency of grammar changes. In particular, we assumed that a grammar change could take place at every input token.

This analysis leads us to the natural advantage of the Earley parsing technique. Since Earley parsing is essentially an interpretive process, there is no processing of any grammar changes required in order to start recognizing against a modified grammar. Thus, even if grammar changes are extraordinarily frequent, performance is essentially unchanged.

Even in the absence of *any* grammar changes, performance is tolerable in practice. As mentioned previously, without any grammar changes (*i.e.*, with a completely static grammar), our C parser implementation parses a 600-line Java program (2777 tokens) in about 44ms on an 866MHz Pentium III.

4.7 Literal Equivalence

Scheme macros have the ability to specify pattern elements that must match literally. Since this is most often useful for identifiers (for example, `else`), the question arises as to how to distinguish identifiers that denote pattern variables from identifiers that are keywords. Scheme answers this by having the developer provide an explicit list of identifiers along with the syntax definition. The identifiers in the list are then considered literals in the patterns rather than pattern variables.

Furthermore, in Scheme, these syntactic literals are *scoped*, meaning that the determination of literal equivalence takes into account the normal scoping of the language. The syntactic literals are not *reserved* words, and the identifiers may be rebound, leading to new meanings (and hence a failure to match in the pattern).

In Scheme, the `cond` special form can be implemented using a macro. To do so requires the use of a scoped syntactic literal for recognizing the `else` clause correctly. The following definition of `cond` illustrates the idea:

```
(define-syntax cond
  (syntax-rules (else)
    ((_ (test body ...) clause ...)
     (if test
         (begin body ...)
         (cond clause ...)))
    ((_)
     #f)
    ((_ (else body ...))
      (begin body ...))))
```

Here, `else` is a syntactic keyword and is therefore declared explicitly in the `syntax-rules` clause.

The scoped nature of the `else` keyword is illustrated by the following example:

```
(let ((else #f)
      (never #f))
  (cond
   (never 0)
   (else 1)))
```

If the occurrence of `else` were interpreted as referring to the syntactic keyword, this program fragment would evaluate to `1`. Instead, the local binding of `else` shadows its meaning as a syntactic keyword so that its occurrence denotes the usual variable reference instead, and hence this program fragment evaluates to `#f` instead.

In an extensible parsing context, it is also necessary to specify syntactic literals. In a Scheme system, there are only a few special cases where syntactic literals are required. The `else` clause in a `cond` is one of just a few examples. However, in parsing traditional languages, syntactic literals are used to recognize essentially all constructs. Syntactic literals are the keywords of the language, such as `if` and `while`, and introduce most statements[8].

[8] It appears that statements are usually *introduced* by such distinguished keywords partly for computation and partly for comprehension reasons. Computationally, any parser has an easier time if it can recognize the kind of construct it is parsing as soon as possible. Recursive descent parsers, a mainstay of hand-constructed parser techniques, rely on this early recognition of constructs. It also seems that the same principle applies in the cognitive

How does the system determine whether or not a syntactic literal matches a particular token in the input? There are two basic approaches to making that determination. The first, more traditional approach, marks certain identifiers as special *reserved words*. The second approach allows literal equivalence to be a local property of an identifier.

In the traditional reserved-word approach, any occurrence of something that looks like an identifier but is made up of a particular character sequence is regarded as a different class of token – an instance of a reserved word. For example, `while` may look somewhat like an identifier, but its actual content causes it to be classified by the scanner as a reserved word, or, more generally, a syntactic literal. In this way, since the lexeme is never regarded as an identifier, there is no confusion between identifiers and reserved words. Keywords are not even considered identifiers by the lexical analyzer; their special roles are determined and assigned during lexical analysis and are fixed and global in the language. This approach has the advantage of clarity – there is no ambiguity about the syntactic role of `while`.

However, this approach fails when the language is to be extended dynamically, because the author of a particular module cannot know what syntactic literals are going to be used by some other extension to the language – the global nature of reserved words breaks the modularity of the language system. Hence, an extensible language system must support a means to scope syntactic literals to their textual regions of relevance and avoid influencing other textual regions of the program.

This second approach makes literal equivalence a local property of an

process; it is simply easier for humans to read a program when a left-to-right scan reveals the structure of the program in a top-down manner. Standard mathematical expression grammars are an interesting counterexample.

identifier, determined by it context of occurrence. We accomplish this by making the lexical analyzer extensible in exactly one dimension: the set of identifiers that are interpreted as syntactic keywords is associated with the current grammar.

Once scope has been introduced to manage the modularity of syntactic literals, another question arises: How is a syntactic literal in a pattern determined to match a syntactic literal in the input? This is the same question as applies to program variables – how is a reference to a program variable known to refer to one particular declared variable or another. To recall the example for program variables, consider the simple case:

```
int foo( int x )
{
  print( x );
  if (x == 0) {
    int x = 9;
    print( x );
  }
}
```

The second occurrence of a reference to the program variable `x` clearly refers to the second declaration – the mechanisms of lexical scope ensure this by properly managing the compile-time context.

The problem is the same for syntactic literals but recalls the approach of Scheme where syntactic keywords are lexically interpreted as identifiers. During compilation, these keywords are recognized as being "bound" to syntactic markers like `else` and including special forms like `if` and `let`.

4.8 Improvements to Basic Earley

In addition to using Earley as the core algorithm for creating an extensible parser, we have made some simple changes that simplify parsing and improve performance without significantly impairing the incrementally extensible benefits.

4.8.1 Conflict resolution

In the presence of an ambiguous grammar, the general Earley algorithm can return all parse trees for a given input string. This can be done with no extra space cost in a *recognizer*. However, a *parser* that builds meaning during the parse can require exponential space to encode the meanings of all the different parse trees. In practical language design, it is useful to have a simple rule for eliminating ambiguities locally, that is, as soon as an ambiguous parse is detected in the input.

We have developed an approach for resolving conflicts between alternative possible rule reductions that is simple to understand and trivial to implement. In our approach, we attempt to resolve conflicts eagerly. Conflicts arise when parsing an ambiguous phrase, so essentially we resolve the ambiguity as soon as the conflict is detected. The disambiguation rule we adopt is to preserve the earlier rule in the grammar and discard the later rule. This has the advantage of being easy to understand and fully deterministic.

This is actually quite easy to do by implementing the parser to be *rule-order preserving*. That is, by evolving the Earley states in an ordered fashion, we know that in an Earley state containing two *conflicting* completions, the completions are processed in exactly the same order in which they were added as a result of the prediction which introduced them. Conflicting completions are those that are for the same non-terminal and are covering the same substring of input, that is, their predicted-from pointers are to the same state. Then, we simply ensure that the predictions are introduced in the same order as the occurrence of their rules in the grammar. We keep only the first conflicting completion, and the user has an easy-to-understand model for the resolution of conflicts in parsing.

4.8.2 Pruning states using FIRST

Observation of the operation of the the Earley-based parser indicates that many tuples are introduced that are dropped in the transition to the next state (*i.e.*, reading the next input token). For our 2777-token Java program, without FIRST pruning, there are 1225560 tuples created, compared to 618620 if pruning is done. Commensurate with this gain, overall parser running time is almost cut in half (86ms versus 44ms).

Even better, these tuples can not be introduced at all by initially verifying that the succeeding input token is not in the FIRST set[9] of a dotted rule to be predicted. This way, no prediction succeeds that does not lead to a rule that matches the next token of input.

4.8.3 Approximating FIRST

Unfortunately, it is relatively expensive to compute the FIRST set because it is a global property of a grammar and would in principle need to be recomputed whenever the grammar changes. Part of the difficulty lies in empty

[9] See Aho *et al.* [1] for a description of the FIRST set.

(ϵ) rules; you can't simply recursively expand the left-hand non-terminal – if a non-terminal can expand into nothing, the FIRST computation has to check the next grammar element in the rule, too.

However, it turns out that an easy-to-compute approximation of FIRST gets most of the benefit. In fact, the numbers cited above are based on using this approximation. The approximation is to punt on epsilon productions and assume they can match anything. In the usual computation of FIRST, an epsilon production causes the invoking rule context to check the next grammar element for its FIRST. In our approximation, the FIRST of an epsilon production is defined to be the universal set. This is a conservative estimate of the real FIRST but makes the scope of the computation much more local and hence easier to recompute.

Chapter 5

Compiler Extension Framework

In the last chapter, we introduced a compiler built using the Earley algorithm as the core engine. Prior to that, in Chapter 2, we introduced the theme of extensible programming as a discipline with supporting technology from the language framework, specifically in the context of Lisp systems. In this chapter, we bring these concepts together and show how a few additional capabilities in an extensible compiler can bring the full power of the extensible programming discipline to bear in a conventional syntax.

5.1 Capabilities of Extension Framework

The additional capabilities we wish to add are:

- Declarative, pattern-based transformations
- Transformations based on synthesized attributes
- Arbitrary procedural mechanisms to produce code

5.1.1 Declarative, Pattern-Based Transformation

A declarative, pattern-based transformation is a pair consisting of a target language pattern and a template for the translation expressed in a reduced form of the target language. The restriction on the form of the template is to prevent indefinite recursion of transformation; as a matter of practice, the pattern is constructed to match a "high level" language construct, and the template makes use of only lower level constructs.

Using pattern-based transformations, a language system developer can easily define a translation from one set of language features into another. This capability is important because many language features are easily understood in terms of simpler, more primitive features. Indeed, new language features are often defined for the programmer in terms of existing features. For example, the "+=" operator is often defined in terms of the existing "+" and "=" operators.

5.1.2 Pattern Matching Synthesized Attributes

Since the result of parsing is to produce meaning values, and parsing takes place while trying to match a pattern in a pattern-based transformation, there is additional opportunity to include attributes of the meaning in the pattern matching process. Thus, the transformation can match synthesized attributes in the elements of the pattern part.

Transformations based on synthesized attributes extend the capabilities of declarative, pattern-based transformations by letting patterns match on computed properties of the constituent parts. For example, a formatting procedure like `printf` can be expanded at compile-time if the format string is a compile-time constant. The property of an expression being a compile-time constant is a synthesized attribute of an expression.

5.1.3 Procedural Code-Production Mechanisms

Arbitrary procedural mechanisms to produce code are the ultimate fall-back for the language system developer. This is the escape hatch when the declarative, pattern-based transformations are too weak, and the problem cannot be expressed locally in terms of synthesized attributes. Procedural mechanisms are also how a language system is bootstrapped, since in order to begin, the language system must be expressed in terms of some other available language.

5.2 Elements of an Extension Framework

Supporting the *target language* is the eventual purpose of a language system. It is in the target language that end users express the solutions to their problems. In the lexicon of layering, the target language is the interface at the top of the layer.

An extensible parser requires two languages in order to be useful. First, there must be a language which can describe the syntax of the new feature. Because this constitutes a syntax for describing syntax, it is called *meta-syntax*. Second, there must be a language for describing how to compute the meaning (*e.g.*, intermediate code) for the new feature.

The *meta-language* provides the means to implement the target language. In terms of layering, it is the means by which the layer itself can be implemented. In the context of this work, we say that the meta-language is the language used to express the computation of meaning.

An extensible parser requires some notation to define the syntax of the target language and to associate parsing activity with actions expressed in the meta-language. This notation is the meta-syntax, and it contains as a sub-language the meta-language.

Parsing is the process of applying the rules of the current grammar to identify target language elements (as defined using meta-syntax) in the input sequence of tokens. *Compiling* is parsing plus the invocation of the actions written in the meta-language to produce a meaning for the target language element (usually some form of intermediate code).

5.3 Implementation

To achieve the desired capabilities, our approach involves extending the Earley parser described in the previous chapter with several features. The following features are added:

- Meta-language: a notation for expressing the composition of intermediate code, possibly employing recursive compilation, pattern variables, and the results of in-line computation
- Local grammar changes: the ability to parse some parts of a rule using a different or modified grammar
- In-line computation: the ability to execute meta-language code during the parse before the completion of a rule

5.3.1 Meta-language

The meta-language includes the following features:

- Syntax evaluation: the ability to translate syntax from the intermediate code compiled from meta-syntax to the internal representation used by the parser

- Recursive compilation: the invocation of the compiler as part of an action computation (*i.e.*, from the meta-language) to compile new strings of tokens into intermediate code

- Pattern variables: the ability to reference the non-terminals of a production's right-hand side from within a recursive compilation

For ease in developing the meta-language itself, we use a bootstrapping process to make the power of the extensible compiler available for implementing the meta-language. Since we claim that this approach is valuable in the development of language systems, what better (or first) system to which to apply the approach than the system's meta-language!

The bootstrapping proceeds in three phases:

(1) The initial grammar is not expressed in meta-syntax notation at all. Instead, the initial grammar is expressed using the intermediate code to which the meta-syntax normally compiles and which the parser consumes. This is necessary because initially there is no grammar with which to parse meta-syntax. The grammar defined in this phase is for meta-syntax and is limited to just enough to express what is needed in the next phase. In our implementation, the initial grammar compiles most of the meta-syntax part but very little of the meta-language. This initial grammar lacks in-line computation in the meta-syntax and has only variable references, literals, and function calls in the meta-language.

(2) With basic meta-syntax in place and a sufficient meta-language, we can now extend the initial grammar using meta-syntax instead of hard-coded syntactic intermediate code. In this phase, we extend the meta-syntax grammar to include in-line computation and extend the meta-language to include assignment statements, basic conditionals (`if`), block constructs, and the primitives used for doing on-the-fly grammar changes (`syntax`).

(3) Having most functionality in place, we flesh out the grammar to make it fully featured. Here, we write `while` in terms of `if` and `goto`, add one-branch `if`, and provide other convenience statements and expressions.

5.3.2 Syntax Evaluation

Syntax evaluation is the process of turning meta-syntactic intermediate code into the data structures that drive the actual parser. The meta-syntactic intermediate code is the meaning computed by parsing the meta-syntax.

The interpreted nature of Earley makes this process very straightforward for us, compared to what would be necessary for a table-driven LALR parser. This fact is due to the inherent similarity between the data structures of an Earley parser and the representation of the meta-syntax.

We also use this evaluation process to support some convenience notations in the meta-syntax. We support repetition, optional parts, and alternatives, as illustrated in Figure 5.1, where A and B denote pattern subparts.

In support of extending the basic syntax with convenience notations, the syntax evaluation process turns the extensions into the flattened representation expected by Earley. For example:

repetition	$A*$
optional part	$[A]$
alternative forms	$A \mid B$

Figure 5.1: Extensions to basic meta-syntax.

$$S \rightarrow \text{“(”} \quad A* \quad \text{“)”} \qquad\qquad \texttt{= f(\$2)}$$

is transformed into:

$$\begin{array}{ll} S \rightarrow \text{“(”} \quad T_1 \quad \text{“)”} & \texttt{= f(\$2)} \\ T_1 \rightarrow A \quad T_1 & \texttt{= cons(\$1,\$2)} \\ T_1 \rightarrow \epsilon & \texttt{= '()} \end{array}$$

where T_1 is a newly generated non-terminal[1]. The actions associated with the T_1 rules have the effect of building the meaning of $A*$ as a list of A meanings.

Similarly, the optional construct $[A]$ produces a meaning which is either the false value in the meta-language or the meaning of A. The alternative construct $A|B$ produces a meaning which is the meaning of the subpart that matched.

5.3.3 Recursive Compilation

The ability to recursively invoke the parser is presented as a procedure in the meta-language, `compile`, which accepts a non-terminal name, a token string, and an optional set of pattern variable bindings[2].

[1] The name T_1 is actually formed from the name of A by appending a `__star`, which is necessary to allow the extension author access to the entire repetition construct. An example of this is in the next chapter.

[2] The very name of this procedure, `compile`, shows our perspective – this operation is one to produce intermediate code, and is not merely determining syntactical structure.

Recursive compilation is mostly achieved simply by following the engineering practice of avoiding global variables. With that, the parser initialization procedure is refined to allow a parse to start at a non-terminal other than the grammar's start symbol. As Earley parsing was originally defined, the grammar contains a distinguished entry point, ϕ (see Section 4.2.3). Supporting recursive compilation then reduces to building an appropriate ϕ rule on demand.

Now when trying to parse a token string L with respect to a non-terminal A, the framework builds an initial state $\langle \phi_k \rightarrow \cdot\, A, 0\rangle$ where k is newly generated[3]. The parser then runs with L as input. In the final state, if L is a valid occurrence of A, then there is a tuple $\langle \phi_k \rightarrow A\, \cdot, 0\rangle$, the meaning for which is the value of the recursive compilation.

5.3.4 Pattern Variables

The meta-language needs a way to refer to the meanings that have been built up by compiling the elements of a pattern. Some systems use numerical variables (*e.g.*, see YACC [21]), others use symbolic variables (*e.g.*, see the work of Cardelli *et al.* [9].) Since the audience of our meta-syntax is fairly broad, we believe symbolic names are the better choice. These variables are used in two ways – they are the variables used in the meta-language for composing new meaning, and they are referenced in declarative translations.

We do not extend the parser *per se* to implement symbolic pattern variables. Instead, in the process of compiling the meta-language into an

[3] In fact, we don't even need to generate a new symbol – we simply build a new anonymous `<production>` object, which achieves the effect of having defined a new non-terminal. Since the ϕ name never appears on the right-hand side of any rule, the production does not need a name.

executable representation suitable for the runtime environment of the compiler, the compiler maps the positional syntactic parameters to the symbolic names. For example, in Figure 5.2, the action part is compiled into a representation like:

```
(lambda (ignore s)
  (compile 'stmt '...))
```

which the runtime environment calls with arguments that are the meaning of the `twice` identifier token and the `stmt`, respectively. The meaning of the identifier is discarded because it is not assigned a name in the production.

The parser is extended to recognize when a pattern variable is being used in a token string. The parser substitutes the associated meaning value when it encounters an identifier which is bound to a meaning in the syntactic environment. To implement this, the environment structure in the parser includes a pattern variable symbol table. The value of a symbol is a meaning value and a non-terminal name. When the parser is predicting $\langle N \rightarrow \alpha \cdot A \beta \rangle$, it checks to see if the next input token is an identifier whose name occurs in the pattern variable symbol table and which was generated by parsing an A. If this happens, the meaning is appended to the tuple, and $\langle N \rightarrow \alpha\, A\ \cdot\ \beta \rangle$ is put into the next parse state, just as if an actual A had been parsed out of the input at that point.

Taking Figure 5.2 as an example, `s` is a pattern variable that gets bound to the meaning resulting from parsing the *stmt* that follows the "`twice`" keyword. In the recursive compilation, the body of the block is expecting a *stmt*, so the occurrence of `s` matches and the *stmt* production is completed. This is repeated again for the second occurrence of `s`.

```
twice_stmt
  = "twice" <s:stmt>
    == compile( :stmt, [[ { s s } ]] );
```

Figure 5.2: The `twice` statement, illustrating pattern variables and substitution conformance.

In our language for describing language extensions, a rule is comprised of a three parts. The first part is a non-terminal name (*e.g.*, `twice_stmt`) followed by an equals sign (“`=`”). The second part is a sequence of elements which are to make up an occurrence of the form (*e.g.*, the keyword `twice` and a *stmt*) followed by a double-equals sign (“`==`”). The third part is the expression which is to compute the meaning of the form from its constituents (*e.g.*, an invocation of the `compile` operator). Inside the third part (the meaning expression), a bare colon (“`:`”) introduces a symbol in the meta-language (*e.g.*, `:stmt`), and double-brackets (“`[[`” ... “`]]`”) enclose a string of tokens which are to be recursively compiled. This notation is described more fully in Section 5.8.

Note that there is no “`;`” after the `s`, which looks a little strange to the casual observer. However, this follows from the expansion of the block form and the fact that `s` is a statement. Figure 5.3 shows how the *block* expansion works out to the level of the `s`, which matches the *stmt* non-terminal.

In some cases, non-terminals are organized into some kind of meaning hierarchy. In fact, to express precedence in this kind of grammar, it is common to see very deep hierarchies[4]. For example, a *primary* is a valid *expr*. In this

[4] Our Java grammar has 21 levels of nesting from *expr* to *identifier*. In a case like this, pre-processing the grammar, as is done in most compiler generators, can greatly improve

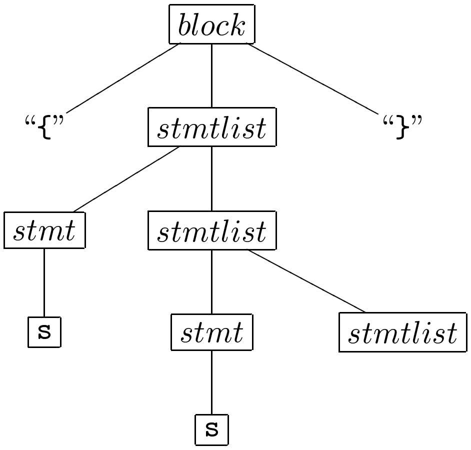


Figure 5.3: Parse tree for translation of the `twice` statement.

case, something parsed as an *primary* could be used where a *expr* is expected. See Figure 5.4 for an example. In our implementation, this inheritance happens automatically because the Earley parser keeps expanding *expr* in the current parse state until it gets to *primary*, at which point it matches the pattern variable `e`.

5.3.5 Local Grammar Changes

Normally, the parser uses the environment of a tuple for resolving grammar lookups (recall that a parse tuple has an associated grammar environment). A local grammar change is an operator which tells the parser to use a different environment for parsing a particular non-terminal in a pattern. The meta-syntax for a local grammar change pattern part looks like:

performance. Pre-processing wins because (1) *expr* is a very common grammatical element, and (2) a single identifier is a very common expr. Our interpreted approach frequently expands 21 non-terminals just to discover that `x` is an expression!

```
expr = "getter" <e:primary>
  == compile( :expr, [[ lambda () e ]] )
```

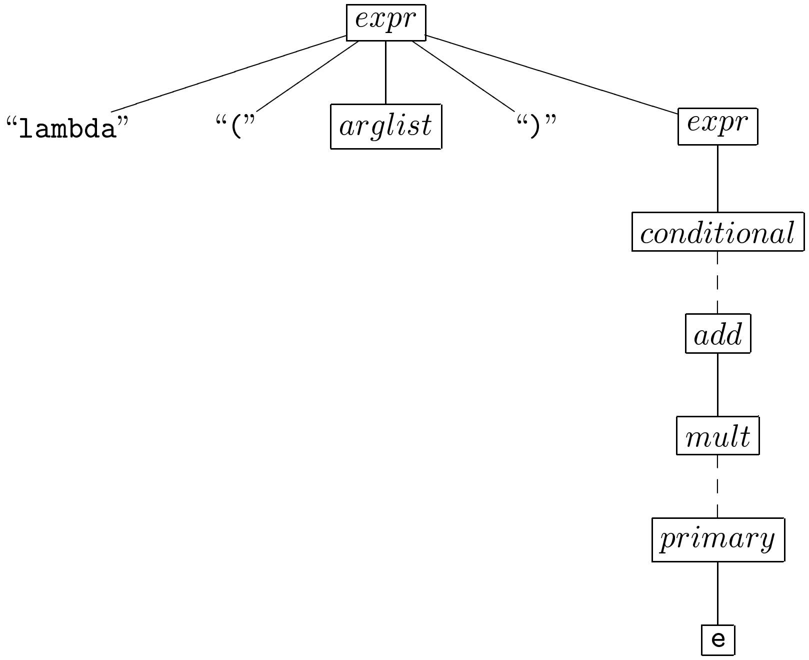


Figure 5.4: Parse tree illustrating how a *primary* gets used as an *expr*.

$$< pv \; : \; nt \; : \; ev >$$

where *pv* is the name of the pattern variable to bind the resulting meaning, *nt* is the non-terminal to parse, and *ev* is the name of the variable which contains the environment in which parse *nt* (and the grammar in which to find *nt* itself).

The *ev* is either a rule local variable (Section 5.3.6) or the global name of a syntax module (Section 5.7). For example, `<q:query:sql_syntax>` in a pattern means to parse the non-terminal *query* as found in the `sql_syntax` module and to use the name `q` to refer to the resulting meaning within this rule.

5.3.6 In-line Computation

To support complex computations to build environments for use in grammar changes, we introduce the ability to do computation in-line with the parsing of the right-hand-side[5]. This allows a single environment to be used for several distinct non-terminals.

In a generalization of pattern variables, the meta-language supports variables that are local to the production. These variables are used to store environments built by in-line computations for use in grammar change operations. As an example of using this feature to extend the syntax environment before processing a subsequent phrase, take:

```
stmt
  = "foo" { enew = extend( envt, ... ); }
    <s:stmt:enew>
    == s;
```

The function `extend` is exposed to the meta-language for the purpose of building new environments (see Section 5.6.3.)

5.4 Declarative Transformations

The general strategy for supporting declarative transformations was presented in the previous chapter. To apply this strategy, our meta-language includes a literal constant notation for a sequence of tokens. Such a literal constant is typically used to supply the argument to a recursive compilation, as in:

[5] This is not a new concept. YACC [21] supports the same thing, although in that system this feature is not used to compute new grammars on the fly!

```
xc_stmt
    = "if" "(" <e:expr> ")" <t:xc_stmt>
      == compile( :xc_stmt, [[ if (e) t else {} ]] );
```

The `compile` meta-language operator takes two arguments – a non-terminal of the grammar, and a token sequence. It returns the meaning of the token sequence when compiled as the given non-terminal. `compile` is not a plain procedure, because it reflects on the scope of the meta-language expression. That is, it arranges for the token sequence to be compiled in an environment that includes the pattern variables present in the right-hand side of the syntactic definition. For example, in the above definition for `xc_stmt`, the use of `compile` includes an environment that binds `e` to the result of compiling the *expr* and `t` to the result of compiling the *xc_stmt*. This notation is similar to that of Cardelli [9], apart from some details of meta-syntax.

With declarative transformation and the other meta-programming tools in hand, we can realize the full implementation of `while` in terms of `if` and `goto`, with `break` defined locally for the body. Figure 5.5 shows how `while` is expressed for the meta-language as it appears in the third phase of bootstrapping.

5.5 Synthesized Attributes

Because we have an expressive meta-language, the action associated with a given production can do arbitrary analysis on the meaning of the pattern variables. However, there is no way to *select* one of several otherwise ambiguous matches depending on that analysis. We introduce the concept of *syntax*

```
xc_while
  = "while" "(" <e:expr> ")"
    { loop = gensym();
      e2 = extend( envt,
                   syntax( xc_stmt = break_stmt; ),
                   syntax( break_stmt
                           = "break" ";"
                           == compile( :xc_stmt,
                                       [[return loop;]] ); ));
    }
    <s:xc_stmt:e2>
      == compile( :xc_stmt,
                  [[ loop: if (e) { s goto loop; } ]] );
```

Figure 5.5: Definition of `while` for the meta-language using extensible syntax features.

guards to enable this ability. The meta-syntax for this feature is a meta-language expression preceded by "`/;`" and located before the action part. For example, consider the following simple optimization implemented in the grammar using a guard to check if an operand is zero:

```
sum
  = <e1:expr> "+" <e2:expr>
    /; zeroq(e2)  /* check for always-0 right operand */
    == e1;
```

Our system implements this by arranging for a special meaning value to be returned as the meaning when the guard expression fails. When completion processing encounters the special value, it discards the entire tuple.

5.6 Procedural Code Production

The standard library of the meta-language provides several facilities that are used to procedurally produce code. Principally, these facilities provide access to the machinery of the compiler and the objects of intermediate code. For our purposes, we use the RScheme runtime system as the execution environment for the meta-language, although the meta-language syntax is closer to that of Java.

5.6.1 Token Sequences

In this meta-language, we use Scheme data structures to represent compile-time objects. A token is a pair, and a sequence of tokens is a list. Since the underlying meta-language is Scheme, all of the list management functions are available for building and destructuring token sequences.

5.6.2 Compilation

The function `compile_in` acts like the `compile` operator but takes an explicit environment and a set of pattern variables and their meanings as an argument. Apart from being the underlying implementation of `compile`, this is used when finer control over the environment is required.

5.6.3 Environments and Syntax

The `extend` function takes a syntax environment and a set of syntax rules and returns a new, extended environment. The syntax rules are produced using the `syntax` special form, which wraps a meta-syntax form. For example,

```
    syntax( stmt = new_stmt; )
```

is an expression which produces syntax intermediate code, suitable for use in extending an environment. This can then be used to extend an environment like so:

```
    extend( envt, syntax( stmt = new_stmt; ) )
```

5.6.4 Reflection

The `literal` function takes an object of the meta-language and returns the intermediate code for an expression whose effect is to produce a corresponding object in the target language. How objects in the meta language translate to target objects is determined by the adaption to the target language within the extensible compiler framework, which is covered in Chapter 6 for Java as a target language.

5.7 Modular Syntax

Our approach supports modularizing syntax, much as in the vocabularies described by Krishnamurthi [26]. In our system, we use syntax modules to separate out the meta-syntax (and its associated meta-language syntax) from the target language syntax. A syntax module is realized as a named syntactic environment, so a module can make use of another module by using the grammar change operator to reference the target module and a non-terminal in it. For example, the gateway between a target language in our current system and the meta-syntax might be expressed like so:

```
decl
  = "syntax" <u:unit:meta_syntax>
    == apply( extend, envt, u );
```

One consequence of this modularization is to distinguish keywords from different sub-languages. For example, the module `meta_syntax` might use as a keyword the identifier "`to`", which is not a keyword in the target language. This is desirable because some sub-languages define a lot of keywords that would pollute the parent language. For example, SQL uses keywords heavily, and reserving those words in a C target language would be prohibitive. Our modularity approach enables the designer to support an SQL extension facility without dragging all of the SQL keywords into the target language:

```
expr
  = "sql" <q:query:sql_syntax>
    == list( :sql_gateway, q );
```

5.8 Full Meta-syntax

In this section we give a more complete description of the syntax and semantics of our meta-syntax, including the embedded meta-language. To describe the meta-syntax of our system as a Java extension framework, we use fairly standard extended BNF notation. In this description, a production is written as a non-terminal, followed by an arrow, followed by a sequence of elements. Each element is either:

- a non-terminal name, such as *foo*,

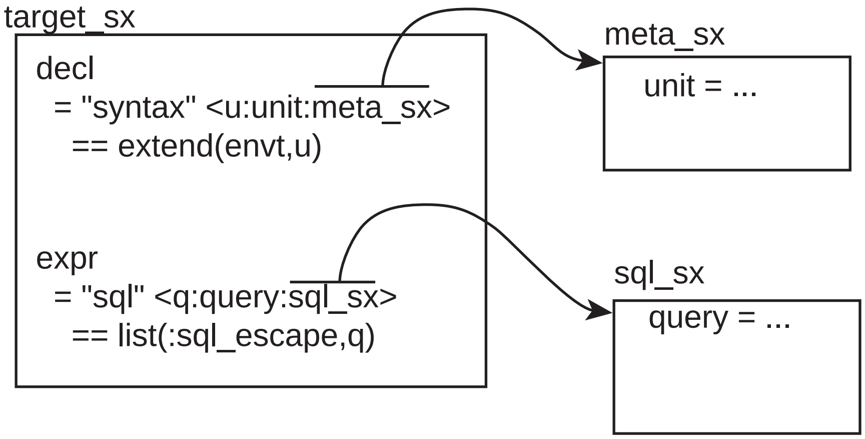


Figure 5.6: Syntax modules being used to contain sub-languages.

- a reference to a category of tokens, such as `id`,
- a literal token, written in quotes, such as "`foo`" or "`=`", or
- an element followed by *, such as *foo**, to denote zero or more occurrences of the element *foo*.

For example,

$$expr \quad \rightarrow \quad expr \quad \text{“}\texttt{.}\text{”} \quad \texttt{id}$$

is a claim that one possible form for the grammatical element *expr* is an *expr* followed by a literal "`.`" token, followed by an identifier. This is a *left-recursive* rule because the non-terminal being defined occurs as the first element in the right-hand side of the production. For describing our system's meta-syntax, we only need to define two token categories: `id` for identifier tokens and `string` for string tokens. The description of the meta-language requires the additional token categories `int` for integer literals and `num` for other numeric literals.

5.8.1 Syntax Declarations

In this section we describe the top-level structure of a syntactic declaration in our extended Java implementation. The basic form is that of a syntactic extension local to a file, which is introduced using the keyword `syntax`.

$$\mathit{java\_tl\_decl} \;\rightarrow\; \text{“}\texttt{syntax}\text{”} \quad \mathit{syntax\_decls} \quad \mathit{java\_tl\_decl}$$

A syntactic extension plays the role of a top-level declaration, and conveys to the following declaration an environment extended with the newly defined syntax. The same idea applies if the syntax is being imported from an separate file:

$$\mathit{java\_tl\_decl} \;\rightarrow\; \text{“}\texttt{import}\text{”} \quad \text{“}\texttt{syntax}\text{”} \quad \mathit{name} \quad \text{“}\texttt{;}\text{”} \quad \mathit{java\_tl\_decl}$$

This form causes a named syntactic declaration in the file indicated by *name* to be imported and supplied to the remaining *java_tl_decl* forms. The file is located according to the usual rules of Java package naming.

Syntax declarations themselves are enclosed in braces:

$$\mathit{syntax\_decls} \;\rightarrow\; \text{“}\texttt{\{}\text{”} \quad \mathit{syntax\_decl}^{*} \quad \text{“}\texttt{\}}\text{”}$$

Each declaration consists of a non-terminal name (`id`) and one or more “`|`”-separated syntax-rules that denote alternative productions for the non-terminal:

$$\begin{aligned}
\mathit{syntax\_decl} &\;\rightarrow\; \texttt{id} \quad \text{“}\texttt{=}\text{”} \quad \mathit{syntax\_rules} \quad \text{“}\texttt{;}\text{”} \\
\mathit{syntax\_rules} &\;\rightarrow\; \mathit{syntax\_rule} \\
\mathit{syntax\_rules} &\;\rightarrow\; \mathit{syntax\_rule} \quad \text{“}\texttt{|}\text{”} \quad \mathit{syntax\_rules}
\end{aligned}$$

5.8.2 Syntax Rules

As seen above, each non-terminal is associated with a set of syntax rules that define the valid expansions of that non-terminal. In the general case, a single syntax rule is a sequence of syntax pattern elements followed by a rule action:

$$\textit{syntax\_rule} \quad \rightarrow \quad \textit{pat\_elem}^{*} \quad \textit{rule\_action}$$

A shorthand notation is provided for the common case that one non-terminal is valid in the place of another:

$$\textit{syntax\_rule} \quad \rightarrow \quad \texttt{id}$$

This would be used, for example, to write that a *block* is a kind of *stmt*:

```
  stmt = block;
```

This declaration states that anywhere a `stmt` is expected, a `block` can be supplied. Of course, the kind of meaning (*e.g.*, intermediate code) built by `block` would have to be compatible with the kind of meaning expected by users of `stmt`.

5.8.3 Syntax Pattern Elements

Syntax pattern elements define the right-hand side of a grammar rule. The pattern elements specify what constitutes a valid occurrence of the non-terminal being defined.

Primitive Pattern Elements

The two most basic kinds of pattern elements are tokens (terminal symbols) and non-terminal identifiers:

$$\begin{aligned} \textit{pat\_elem} &\rightarrow \textit{pat\_token} \\ \textit{pat\_elem} &\rightarrow \textit{pat\_nt} \end{aligned}$$

A token pattern is represented as a literal string, which matches the same token on input. It is an error if the string does not scan as exactly one token.

$$\textit{pat\_token} \rightarrow \texttt{string}$$

For example,

```
  break_stmt = "break" ";"
```

is part of a syntax declaration for *break_stmt*, with a syntax rule pattern list that contains two pattern elements. Each of the two pattern elements matches a literal token. The first token must be the identifier `break` and the second token must be the semicolon delimiter "`;`".

Composite Pattern Elements

The other basic kind of pattern element is a binding construct, which is used to bind a syntax variable to the meaning from a constituent pattern element:

$$\begin{aligned} \textit{pat\_elem} &\rightarrow \textit{pat\_bind} \\ \textit{pat\_bind} &\rightarrow \text{“<”} \quad \texttt{id} \quad \text{“:”} \quad \textit{pat\_elem} \quad \text{“>”} \end{aligned}$$

(See also Section 5.8.6 for another *pat_bind* form)

We also allow the syntax author some convenience notations in defining the syntax pattern. The following *pat_op* element is used to define repeating and optional syntactic patterns:

$$\begin{array}{rcl} \textit{pat\_elem} & \rightarrow & \textit{pat\_op} \\ \textit{pat\_op} & \rightarrow & \text{“(”} \quad \textit{pat\_elem}^{*} \quad \text{“)”} \quad \textit{pat\_opcode} \\ \textit{pat\_opcode} & \rightarrow & \text{“}\texttt{*}\text{”} \\ \textit{pat\_opcode} & \rightarrow & \text{“?”} \end{array}$$

The opcode `*` denotes zero or more occurrences of the *pat_elem* sequence. The meaning that is constructed at parse time for a `*` construct is a list of the meanings of each occurrence. For example, consider the following fragment:

```
block_stmt = "{" <body:(stmt)*> "}"
```

The meaning bound to the syntax variable `body` is structured as a list of lists, each item of the outer list representing one occurrence of the pattern sequence `(stmt)`, and each inner list being of length 1, the single item being the meaning of the `stmt` occurrence. For example, when presented with the input `{ a=1; b=2; }`, the variable `body` is the structure $\langle\langle\mathcal{M}(\texttt{a=1})\rangle, \langle\mathcal{M}(\texttt{b=2})\rangle\rangle$ where $\mathcal{M}(x)$ denotes the meaning resulting from compiling x.

The opcode “?” causes the pattern sequence to match zero or one occurrence. The resulting meaning is either `#f` (the unique **false** value in the meta-language's underlying interpreter) if no occurrence was matched or a list of the meanings of the pattern elements. Consider, for example:

```
if_stmt = "if" <e:expr> "then" <t:stmt> <f:[ "else" stmt ]>
```

Here, `f` takes on either a 2-item list value $\langle\mathcal{M}(\texttt{else}), \mathcal{M}(\textit{stmt})\rangle$ or `#f`, depending upon whether the `else` clause was matched in the input or not.

In addition to the postfix “?” operator, an optional element sequence can also be written using the standard `[ ··· ]` notation:

$$\begin{aligned} \mathit{pat\_elem} &\rightarrow \mathit{pat\_opt} \\ \mathit{pat\_opt} &\rightarrow \text{“}\texttt{[}\text{”} \quad \mathit{pat\_elem}^{*} \quad \text{“}\texttt{]}\text{”} \end{aligned}$$

5.8.4 Actions

The meta-syntax contains as a sub-language the meta-language, which defines the actions and expressions used to compute the meaning resulting from a parse match. The simplest arrangement is that the action is an expression in the meta-language, which is to be evaluated when the production is completed in the Earley parser:

$$\mathit{rule\_action} \rightarrow \text{“}\texttt{==}\text{”} \quad \mathit{xc\_expr}$$

The *xc_expr* is evaluated with respect to any variables bound by *pat_bind* pattern elements.

Since the meta-language uses Scheme as its underlying interpreter, a special expression is defined to make Scheme symbol values denotable using Java’s lexical rules:

$$\mathit{xc\_expr} \rightarrow \text{“}\texttt{:}\text{”} \quad \texttt{id}$$

The “`:`” is used to introduce a meta-language literal symbol value. For example, `:stmt` is a meta-language expression which evaluates to the symbol `stmt` in the underlying interpreter. Similarly, some global variables are defined to hold other well-known Scheme values.

Meta-language Variable	Equivalent Scheme Expression
`nil`	`'()`
`true`	`#t`
`false`	`#f`

Token Sequences

In order to perform a recursive compilation, a meta-language procedure needs to have a token string to compile. Token strings are entered as literal objects delimited with double square-brackets. Conceptually, a token sequence is a token of the meta-language, although it is actually implemented in the current system using a grammar rule for matching a string of bracket-balanced tokens.

Procedures

The meta-language supports procedure call expressions in the usual notation:

$$
\begin{aligned}
\mathit{xc\_expr} &\rightarrow \texttt{id} \quad \text{“(”} \quad \mathit{call\_args} \quad \text{“)”} \\
\mathit{call\_args} &\rightarrow \epsilon \\
\mathit{call\_args} &\rightarrow \mathit{call\_args\_ne} \\
\mathit{call\_args\_ne} &\rightarrow \mathit{xc\_expr} \quad \text{“,”} \quad \mathit{call\_args\_ne}
\end{aligned}
$$

The meta-language also provides access to all the normal procedures of the underlying interpreter. Primarily, these are used for manipulating the data structures which make up the intermediate code of an extension (for example, see Section 6.2.2.) Since the underlying interpreter is RScheme, most Scheme procedures are available [14].

For manipulating lists, common procedures are the list constructors `list` and `cons`, and the list accessors `car` (get first item), `cdr` (get rest of items), `cadr` (get second item), etc. The list iteration procedures `map` and `for-each` are also available.

The Compile Operator

The `compile` operator is the primary means for continuing the compilation recursively from within meta-language code. It is invoked syntactically like a function with two required arguments and one optional argument. The first argument is a symbol denoting the non-terminal that is to drive the compilation. The second argument is the token sequence to be compiled. The optional third argument is a set of syntax bindings to be used in the compile.

The `compile` operator is special (*i.e.*, it is not a normal procedure) because it knows about the syntax variables that have been defined using *pat_bind*. By default, `compile` interprets an identifier in the supplied token string that matches the name and non-terminal type in a *pat_bind* construct as meaning the previously parsed value. The optional third argument extends this set of bindings with an explicit list of three-item lists. The three items in the list are the identifier in the token string to match, the non-terminal that represents to type of meaning, and the meaning value.

Embedded Syntax

Since a meta-language procedure may need to manipulate the syntactic environment (see the `extend` operator, below), it is necessary to have a way of "quoting" syntax rules so they can be managed as first-class objects. The `syntax` special form does just that:

$$\textit{xc\_expr} \;\rightarrow\; \text{“}\texttt{syntax}\text{”} \quad \text{“}\texttt{(}\text{”} \quad \textit{syntax\_decl} \quad \text{“}\texttt{)}\text{”}$$

The value of a syntax form is a *syntax rule set*, the internal representation of a set of syntax declarations, suitable for installing into the syntactic environment.

The Extend Operator

The basic operation on syntactic environments is to extend them with new rules. The extend operator does that, taking as arguments an environment and a sequence of syntax rule sets and returning a new environment which incorporates the indicated rule sets.

The following example extends the syntax environment with a trace statement in the context of an enclosing tracing. The example does this by computing a new environment in an inline action using the extend operator, and then using that environment to compile the block of control. No additional computation of the meaning is required in this example (*i.e.*, the meaning can simply be body), because any uses of trace inside the body have already been properly compiled.

```
stmt = "tracing"
       { e = extend( envt,
                     syntax( stmt = "trace" ";"
                                  == ... ) ); }
       <body:block:e>
       == body;
```

Anonymous Procedures

Anonymous procedure abstractions using `lambda` are also available. Syntax variables defined using *pat_bind* elements are available to the procedure like other variables with lexical scope.

$$\begin{array}{rcl}
\mathit{xc\_expr} & \rightarrow & \text{“}\texttt{lambda}\text{”} \quad \text{“}\texttt{(}\text{”} \quad \mathit{xc\_lambda\_name\_list} \quad \text{“}\texttt{)}\text{”} \quad \mathit{xc\_block} \\
\mathit{xc\_lambda\_name\_list} & \rightarrow & \epsilon \\
\mathit{xc\_lambda\_name\_list} & \rightarrow & \mathit{xc\_lambda\_name} \\
\mathit{xc\_lambda\_name\_list} & \rightarrow & \mathit{xc\_lambda\_name} \quad \text{“}\texttt{,}\text{”} \quad \mathit{xc\_lambda\_name\_list}
\end{array}$$

In addition to defining normal meta-language arguments, syntax variables can be declared as procedure arguments, which makes the identifier available to the `compile` operator.

$$\begin{array}{rcl}
\mathit{xc\_lambda\_name} & \rightarrow & \texttt{id} \\
\mathit{xc\_lambda\_name} & \rightarrow & \texttt{id} \quad \text{“}\texttt{:}\text{”} \quad \texttt{id}
\end{array}$$

The second form is used to declare an argument whose value is a meaning of the given type. This is useful, for example, in situations like the following:

```
stmt = "traceall" "{" <stmtlist:(stmt)*> "}"
       == map( lambda( s:stmt )
                 { compile( :stmt, [[ { trace(); s } ]] ); },
               map( car, stmtlist ) );
```

Here, an anonymous procedure is being used to iterate a compilation over a collection of statements. Without the convenience of specifying the syntactic type in the `lambda` argument list, implementing that would be somewhat more verbose:

```
stmt = "traceall" "{" <stmtlist:(stmt)*> "}"
       == map( lambda( s )
                 { compile( :stmt,
                            [[ { trace(); s } ]],
                            list( list( :s,
                                        :stmt,
                                        car(s) ) ) );
                 },
               stmtlist );
```

5.8.5 Local Variables

Local variables are implicitly declared by assigning to their name. They can be used to remember values between inline actions (see below) and the action expression.

$$\textit{xc\_expr} \;\rightarrow\; \texttt{id} \;\; \text{``=''} \;\; \textit{xc\_expr}$$

5.8.6 Inline Actions

We also permit the evaluation of meta-language code during the recognition part of the parse.

$$\textit{pat\_elem} \;\rightarrow\; \textit{xc\_block}$$

In general, this is functionally equivalent to defining a unique non-terminal T_k with an empty pattern and the *xc_block* as the action, and using T_k where the *xc_block* occurs as a pattern element. That is:

```
foo = <a:A> { F(); } <b:B>  == G(a,b)
```

is functionally similar to:

```
foo = <a:A> Fk <b:B>         == G(a,b);
Fk = /* empty */             == F();
```

However, this kind of inline action is supported specially in order to let any local variables created by the *xc_block* be visible to the remaining pattern elements and *rule_action* of the current rule. Local variables created by an inline action let the programmer parse a non-terminal in a different environment using a variant of the *pat_bind* form. This form is used to parse an occurrence of a non-terminal with respect to a different parse environment:

pat_bind → “<” `id` “:” `id` “:” `id` “>”

The three `id`’s are, respectively, the name to which to bind the resulting meaning, the non-terminal to be parsed, and the name of the pattern-local variable containing the environment to be used.

So, for example, we can say:

```
while_stmt = "while" <e:expr> { brkenv = ··· } <body:stmt:brkenv>
```

where ··· denotes some additional meta-language code to compute a syntax environment. This allows meta-language code to easily define the environment of compilation for subsequent non-terminals.

5.8.7 Example

For a complete example of some meta-syntax, consider a syntax declaration for a `twice` statement (the body of which appears in Figure 5.2):

```
syntax {
  twice_stmt
    = "twice" <s:stmt>
      == compile( :stmt, [[ { s s } ]] );
}
```

The entire construct is interpreted as a kind of top-level Java declaration (a *java_tl_decl*), comprised of a single *syntax_decl* for the non-terminal `twice_stmt`[6].

The `twice_stmt` non-terminal is associated with one *syntax_rule* that consists of two pattern elements, the literal identifier "`twice`" followed by some kind of `stmt`, with the result bound to the syntax variable `s`.

Furthermore, this declaration states that when a `twice_stmt` is recognized (the completion step in the Earley parser), the way to compute the meaning is to invoke the `compile` operator with two arguments, the symbol `stmt` and the 4-token sequence `{ s s }`. Because `s` is a syntax variable in the pattern list, the `compile` operator recognizes occurrences of the token `s` in the token string as references to the meaning computed by parsing the non-terminal `stmt`.

[6] Note that the three names referred to in this sentence are non-terminal names: *java_tl_decl* and *syntax_decl* are non-terminals of the meta-language, and `twice_stmt` is a non-terminal of the target language. We render them with a different typographical style to emphasize the distinction, but in fact both are present in the same grammar. This fact makes it possible to extend the extension framework from within the language, a hallmark of a reflective system [33].

5.9 Issues and Future Work

5.9.1 Substitution Conformance

One issue that arises when doing pattern variable substitutions at the grammatical level is determining conformance of meaning. For example, if the original production compiles something as a *stmt*, then the meaning is intermediate code appropriate for a *stmt*. Such intermediate code should probably not be used where, for example, a *decl* is needed.

One of the problems with our method of determining conformance occurs when there are anonymous pattern structures. For example, in our metagrammar, we permit constructs like `<slist:(stmt)*>` (see Section 5.3.2).

However, there is no way to determine conformance for the resulting meaning. Hence, we prohibit the use of such a definition in a declarative transformation. From an extensible language design perspective, this implies that the grammar must use named non-terminals for any element which is to be reused in a macro pattern. Future work should ascertain how to ameliorate this limitation.

Some systems rely on static type checking to ensure that pattern variables are used in places compatible with the obtained meaning [9, 34]. This approach is generally too restrictive for our purposes, since it requires too much whole-grammar analysis for correctness[7].

[7] Although it would be useful to implement static checking where possible for grammar modules that are developed as units. This would help address the testing problem common to most purely interpreted systems.

5.9.2 Translation Recursion

The way our approach eagerly computes meanings as soon as constructs are recognized (*i.e.*, at completion), combined with our approach to transformational compilation, can lead to unbounded recursion. This is similar to the problem of left recursion in a recursive-descent parser [1]. The problem arises when one form is translated into another that contains a completable instance of itself. Eager completion means that a "completable instance" may only be a prefix of the translation string. This comes up, for example, in the translation of one-branch `if` to two-branch `if`. The natural rule to try to write is:

```
xc_onebranchif
  = "if" "(" <e:expr> ")" <t:xc_stmt>
    == compile( :xc_stmt, [[ if (e) t else {} ]] );
```

but, because this rule itself is completable after the `t` in the expansion, the system loops trying to determine the meaning of the first 5 tokens of the translation.

For our meta-language extensions, we worked around this by expressing the translation directly in terms of a `xc_twobranchif` instead of in terms of an `xc_stmt`. This works for some cases but is not a general solution in two ways: (1) it requires knowledge of the non-terminal that bypasses the recursion, and (2) it assumes the recursion is at the top level. With respect to (2), note that the solution does not apply when a construct nested within the token string passed to `compile` matches the rule being parsed, for example:

```
xc_onebranchif
  = "if" "(" <e:expr> ")" <t:xc_stmt>
    == compile( :xc_stmt, [[ { if (e) t else {} } ]] );
```

triggers the same indefinite recursion.

5.9.3 Meta-syntax Scope

To scale a language system built using our approach, a way of managing the scope of syntactic identifiers needs to be developed. In our current implementation, syntactic identifiers are realized as symbols, and their scope is correspondingly global, at least within a given syntax module. This is useful in some ways – for example, a syntax module can extend the definition of *expr* – but being able to manage these identifiers in a more controlled way, with only well-defined export points, would be more scalable.

5.9.4 Syntax Module Templates

The logical next step for syntax modules is to support syntax module templates, which are parameterized modules. This would enable more sophisticated reuse of syntax modules, enabling a syntactic concept (*e.g.*, expressions) to be applied uniformly in different areas of a language system. For example, in our system itself, the concept of expressions shows up in both the meta-language and the target language, yet we can't reuse the syntax module because it generates different intermediate code. One way of parameterizing modules would be to supply the set of intermediate code constructor bindings, thereby tailoring how intermediate code is produced.

Chapter 6

An Application of an Extensible Language

6.1 Introduction

In this chapter, we apply our approach to the problem of defining an extension to the Java language. The application we develop enables the simple definition of a finite-state machine with actions on state transitions (*i.e.*, a Mealy machine).

The language extension we consider is as follows:

$$\begin{array}{rcl}
\textit{automatondecl} & \rightarrow & \text{``}\texttt{automaton}\text{''} \quad \textit{name} \quad \text{``}\texttt{\{}\text{''} \quad \textit{vardecl}^{*} \quad \textit{statedecl}^{+} \quad \text{``}\texttt{\}}\text{''} \\
\textit{vardecl} & \rightarrow & \textit{visibility} \quad \textit{type} \quad \textit{name} \quad \text{``}\texttt{=}\text{''} \quad \textit{initvalue} \quad \text{``}\texttt{;}\text{''} \\
\textit{visibility} & \rightarrow & \text{``}\texttt{public}\text{''} \quad | \quad \text{``}\texttt{private}\text{''} \\
\textit{statedecl} & \rightarrow & \textit{name} \quad [\text{``}\texttt{accept}\text{''}] \quad \text{``}\texttt{\{}\text{''} \quad \textit{transition}^{*} \quad \text{``}\texttt{\}}\text{''} \\
\textit{transition} & \rightarrow & \textit{int} \quad \text{``}\texttt{->}\text{''} \quad \textit{name} \quad (\textit{block} \quad | \quad \text{``}\texttt{;}\text{''})
\end{array}$$

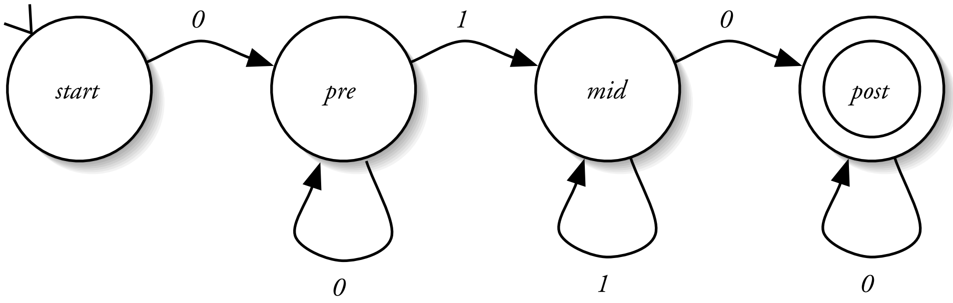


Figure 6.1: A simple string recognizer in graphical state machine representation

For simplicity, we consider only automata over an integer vocabulary.

In the following example, the automaton defined using this extension is constructed to recognize strings of 1's surrounded by non-empty strings of 0's and to count the number of 1's. For example, the string "00001111000" is recognized by this automaton, and the number of 1's is four. Figure 6.1 shows this state machine in diagrammatic form.

```
automaton InteriorString {
  public int count = 0;
  start { 0 -> pre; }
  pre   { 1 -> mid { count++; }
          0 -> pre; }
  mid   { 1 -> mid { count++; }
          0 -> post; }
  post accept { 0 -> post; }
}
```

The objective of this extension is to transform an automaton declaration, such as the one above, into a Java class of the same name that implements the automaton's behavior. The Java class provides a `process` method that takes a segment of input supplied as an integer array. The public variables declared in the automaton are available as public instance variables of the Java class.

The generated class implements the `Automaton` interface, which consists of the following methods:

```
public void process( int input[] );
public String getState();
public boolean isAccepting();
```

The `process` method is responsible for executing the automaton with some input symbols. This method attempts to consume the input (sub)string and either leaves the automaton in the appropriate state or throws a `RuntimeException` if a particular transition is not possible. The `getState` method returns the current state of the automaton as a string. The `isAccepting` method answers whether or not the current state of the automaton is an accepting state.

6.2 Implementation Approach

This application illustrates the occasional need to deconstruct meaning values (*e.g.*, intermediate code). This arises because the lower-level constructs (in this case, the state declarations) contribute to different top-level constructs (in this case, the methods of the automaton class).

To support automata declarations as an extension to Java, we define a syntactic extension module called `dfa` that defines a new kind of type declaration. In the base Java language, a type declaration is a class or interface declaration. Since an automaton acts like a kind of class, it is natural to extend the syntactic construct for classes.

6.2.1 Declaring the Extension

The outermost portion of the extension itself is shown in Figure 6.2. Here, the name of the syntax module is being declared as `dfa`, and the main entry point at `type_decl` is being installed[1].

```
syntax dfa {
  type_decl
   = "automaton" <aname:java_id>
      "{" <vars:(field_decl)*>
          <states:(statedecl)*>
      "}"
   == ⟨typebody⟩
  ⟨statedecl⟩
  ⟨transition⟩
}
```

Figure 6.2: Top-level structure of the DFA extension to Java. The `field_decl` non-terminal is part of the underlying Java grammar.

The grammar symbol name `type_decl`, and its role in the grammar of the underlying language, must be known to the author of the extension. In general, our approach requires knowing something about the specific grammar that is used to realize the underlying language. Although this requirement

[1] Note that we extend the `type_decl` non-terminal, which is from the underlying Java grammar. Likewise, we reuse `field_decl`, the grammatical element for class field declarations. There is no need for us to redefine `field_decl`, since the underlying language provides us what we need. In fact, we thereby get to reuse all the machinery for parsing field declarations and their initializers.

follows from the need to install new grammatical productions in the grammar, it does expose some details of the implementation. In practice, we expect that language systems built to be extensible would incorporate the grammar as part of the language specification, with all the important non-terminal names standardized.

6.2.2 Building the Final Meaning

Approaching this application top-down, the next thing to define is the meta-level code that compiles the entire automaton construct. In Figure 6.3, the basic structure of the automaton compiler is laid out. The strategy is to build up the automaton's class definition from pieces defined in subsequent sections. These pieces are then assembled by the call to `compile` into a single class definition.

As a matter of bookkeeping, the first statement deconstructs the meanings that are bound to `states` that matched the `(statedecl)*` repetition pattern. The meaning of a repetition is a list of the meanings of the elements. However, each element is a sequence, the meaning of which is a list of the meaning of its elements. Hence, the meaning structure for `states` is a list (one entry per `field_decl`) of lists (one entry in the list is the `field_decl` meaning itself). Since we are only concerned with the list of `field_decl` meanings, the first thing we do is pull those meanings out.

This code fragment also shows how meta-language code can compute the value of a set of syntactic variables that are interpreted during parse time. The special operator `compile` normally takes two arguments: a symbol denoting the non-terminal to parse, and a string of tokens that is the fragment to

```
typebody ⇒
  sx = map( car, states );
  ⟨internal-state⟩
  ⟨process-method⟩
  ⟨getstate-method⟩
  ⟨accepting-method⟩
  ⟨statename-statics⟩
  compile( :type_decl,
           [[ class aname implements Automaton { cbody } ]],
           list( list( :cbody,
                       :class_body_decl__star,
                       append( list( cb1 ),
                               vars,
                               statenames,
                               list( cb2, cb3, cb4 ) ) ) ) ) ); };
```

Figure 6.3: Meta-language code for building the meaning of a complete automaton construct.

be parsed. The `compile` operator also implicitly understands any syntactic variables that are in scope from the pattern. For example, in Figure 6.3, the identifier `aname` in the token string passed to `compile` refers to the meaning that was parsed by the `java_id` pattern in Figure 6.2. `compile` can take an optional third argument which is a *computed* set of syntactic variables.

In this case, we are procedurally constructing the elements that make up the class body. In order to insert the meaning of the parts into the final meaning, we define a local syntactic variable `cbody` to take on the meaning of the class body. Furthermore, we compute `cbody` by appending several meaning fragments to form the complete body (the code referred to by *internal-state*, *process-method*, *etc.*, is responsible for building these fragments and is described in detail below.)

This example also illustrates what happens when a repetition construct is transformed by the compiler extension framework. In our Java grammar, a class body is defined as follows:

```
class_body
  = "{" <b:(class_body_decl)*> "}"
    == ...;
```

When the `class_body_decl` is expanded into the direct grammar representation required by the Earley parser, it is transformed into something like:

```
class_body
  = "{" <b:class_body_decl__star> "}"
    == ...;
class_body_decl__star
  = /* empty */ == nil
  | class_body_decl__1 class_body_decl__star == cons( $1, $2 );
class_body_decl__1
  = class_body_decl == list( $1 );
```

(The `list` operator on the last line is why the first statement in Figure 6.3 appears.)

Since the construction of the automaton requires supplying a sequence of class body declarations whose length depends on procedural meta-language code, we need to build up the entire sequence and supply it as the meaning associated with a `class_body_decl__star`.

One of the benefits of our approach starts to become clear here. The automaton extension is written entirely in terms of a well-defined meta-language and the underlying language; a class definition can be constructed without any knowledge of the form of the intermediate code. Only the syntactic elements of the underlying grammar need be identified (*i.e.*, as we mentioned above, that a `type_decl` is the appropriate kind of syntactic object to supply a class definition.)[2]

6.2.3 Declaring the State-Keeping Variable

Our Java class requires an instance variable in which to store the current state of the automaton. The code to declare this instance variable is shown in Figure 6.4. This intermediate code fragment is incorporated into the main class object when the class body is assembled in Figure 6.3.

internal-state ⇒

```
  cb1 = compile( :class_body_decl,
                 [[ private int state = start; ]] );
```

Figure 6.4: Meta-language code to build the internal state variable for the resulting Java class implementation.

The identifier `state` is the literal name of the internal variable that is used to maintain the current state of the automaton. The token `start` is the name of the start state, which is bound to an appropriate integer representation in the resulting class by the code generated in *statename-statics* (Figure 6.12).

[2] In some cases, as we'll see, it is necessary to deconstruct the meaning values produced by the underlying grammar. In this application, it only becomes necessary for literal values and identifiers, for which it is easy to provide appropriate meta-language operators.

6.2.4 Declaring the Java Class's Entry Method

The main entry point to the Java class that this extension produces is the `process` method, which is responsible for advancing the state of the automaton according to a sequence of input symbols represented by an array of integers. Figure 6.5 has the meta-language code fragment that generates the `process` method.

Input symbols are represented as integers in this implementation, so `process` takes an array of integers and executes the state machine for each symbol. If a transition cannot be made from a given state with a certain symbol as input, then a `RuntimeException` is thrown (see Section 6.2.8).

process-method ⇒

```
  cb2 = compile( :class_body_decl,
                 [[
                 // main entry point
                 public void process( int input[] )
                 {
                     int i;
                     for (i=0; i<input.length; i++)
                         {
                             int symbol = input[i];
                             switch (state) { state_clauses }
                         }
                 }
                 ]],
                 list( list( :state_clauses,
                             :switch_clause__star,
                             map( cadr, sx ) ) ) );
```

Figure 6.5: Meta-language code to build the `process` method in the Java class implementation.

6.2.5 Declaring the Java Class's Accessor Methods

In this application, each generated Java class supports two accessor methods, `isAccepting` and `getState`. See Figures 6.6 and 6.7, respectively. The approach to implementing both is similar and involves a switch on the current state of the automaton. The switch clause for each state returns the appropriate value (*i.e.*, the state name as a string literal in the case of the `getState` value, and a boolean value in the case of `isAccepting`.) Note that the compilation of the actual case clause is done when the state is being parsed, in Figure 6.8.

getstate-method ⇒

```
  cb3 = compile( :class_body_decl,
                 [[
                 // current state inspector
                 public String getState()
                 {
                   switch (state) { return_state_clauses }
                   return "?unknown";
                 }
                 ]],
                 list( list( :return_state_clauses,
                             :switch_clause__star,
                             map( caddr, sx ) ) ) );
```

Figure 6.6: Meta-language code to build the `getState` method.

6.2.6 States Within an Automaton

At this point, we are ready to describe the part of the grammar that is responsible for parsing the states within an automaton. Figure 6.8 shows the *accepting-method* ⇒

```
  cb4 = compile( :class_body_decl,
                 [[
                 // currently in an accept state?
                 public boolean isAccepting()
                 {
                   switch (state) { return_accept_clauses }
                   return false;   // unknown state
                 }
                 ]],
                 list( list( :return_accept_clauses,
                             :switch_clause__star,
                             map( cadddr, sx ) ) ) );
```

Figure 6.7: Meta-language code to build the `isAccepting` method.

statedecl production. Each state comprises:

- a state name (matching *java_id* and bound to the syntax variable `n`),
- a flag indicating whether it is an accept state (matching *accept_flag*, which is defined at the bottom of Figure 6.8, and bound to `a`),
- a set of explicit transitions (bound to `t`), and
- a default transition which is really the error handler (bound to `d`).

In this application, we use list-oriented data structures to communicate intermediate code between the non-terminal *statedecl* and *type_decl*. Figure 6.8 gives the *statedecl* fragment of code, which builds the main list structure we use in this application. The data types we build here using lists could be formalized, and a larger application may benefit from static type checking to constrain the grammar itself (for example, see Cardelli *et al.* [9]). However, our approach is dynamically typed, which we find more suitable for experimental work. The list, which is the meaning associated with *statedecl*, has four items:

```
statedecl ⇒
  statedecl = <n:java_id> <a:accept_flag>
              "{" <t:(transition)*>
                  <d:default_transition>
              "}"
      == list( n,
               ⟨stateswitcher⟩,
               compile( :switch_clause, [[ case n: return strform; ]],
                        list( list( :strform,
                                    :expr,
                                    literal(tostring(n)) ) ) ),
               compile( :switch_clause, [[ case n: return aa; ]],
                        list( list( :aa, :expr, a ) ) ) );

  accept_flag = "accept"    == compile( :expr, [[ true ]] )
              | /* empty */ == compile( :expr, [[ false ]] );
```

Figure 6.8: Sub-language for declaring states within an automaton.

- the name of the state,
- the clause that goes into the "big switch" in the `process()` method (see the definition of *stateswitcher* in Section 6.2.7),
- the clause that goes into the switch statement in the `getState` method, and
- the clause that goes into the switch statement in the `isAccepting` method.

These clauses are assembled into the appropriate switch statements in the top-level action (see Figure 6.3).

In the implementation of exposing the accept flag, in which we compile a switch clause[3] to be used in Figure 6.7, we essentially build the literal answer to the question “is this an accept state?” for each different state. We expect that the underlying language compiler applies some common subexpression elimination to optimize the code that appears in the resulting statement:

```
switch (state) {
  case 0: return false;
  case 1: return false;
  case 2: return false;
  case 3: return true;
}
```

If the compiler did not do so, and this bloated code were a problem, then a little additional work at a higher level (*i.e.*, in Figure 6.7) could do some application-specific optimization.

This fragment illustrates the use of the extension framework operator `literal`, which is responsible for producing the target language meaning that creates a value equivalent to the meta-language argument that is its argument. For example, `literal(3)` produces intermediate code which, when executed, evaluates to the integer value “3” in the target language. In this case, we are using `literal` to convert the state name identifier (which is a symbol object in the metalanguage) to a string literal in the target language to be returned as the value of the `getState` method.

[3] A switch clause is a constituent of a `switch` statement, which is Java’s multi-way branch statement. A switch clause is also known as a `case` statement.

6.2.7 Building the State Switcher

Figure 6.9 shows the meta-language code that constructs the switch clause for the current state in the `process` method. For each state of the automaton, the `process` method switches on the next input symbol. The identifier `symbol` in the argument to the `compile` operator refers to the local variable declared in the compiled code in Figure 6.5.

```
stateswitcher ⇒
  compile( :switch_clause, [[ case n:
                                switch (symbol) {
                                  symclauses
                                }
                                break; ]],
           list( list( :symclauses,
                       :switch_clause__star,
                       append( t, list(d) ) ) ) )
```

Figure 6.9: Meta-language code for building the switch clause for a single state that makes up part of the "big switch" in the `process` method.

The clauses which make up the body of the switch on the input symbol are supplied to the `compile` operator in the alias `symclauses`. The `symclauses` alias plays the role of a sequence of switch clauses (formally, a *switch_clause__star*), which we construct by appending the cases for each individually defined transition with the default (or error) transition. Section 6.2.8 shows how the switch clause intermediate code is built up for each transition, including the default.

6.2.8 Declaring transitions

The transitions among the states of the automaton are declared using the grammar elements defined in Figure 6.10. There are two forms for a transition declaration. The first form is used when the automaton programmer is not executing any code on the state transition. The second form is used to supply an action to perform when the automaton makes that transition.

transition ⇒

```
  transition = <k:integer_literal_expr> "->" <n:name> ";"
      == compile( :switch_clause,
                  [[ case k: state = n; break; ]] );

  transition = <k:integer_literal_expr> "->" <n:name> <b:block>
      == compile( :switch_clause,
                  [[ case k: b state = n; break; ]] );
```

Figure 6.10: Sub-language extension for the declaration of a single transition within a state.

Both forms are structurally similar. In each, there is the identification of the input symbol that is used to trigger the transition. The input symbol is grammatically an *integer_literal_expr* and is bound to `k`. `k` becomes the `case` *expr* in the resulting switch clause. The destination, or target, state is also identified by name and bound to the syntax variable `n`.

The difference between the forms is that the second form expects a Java `block`, which is bound to the syntax variable `b`. In turn, the user's action block, `b`, is employed just before the assignment to update the `state` variable in the resulting switch clause. In the sample automaton, the transition action block form is used to count the number of “1” symbols in the middle of the string. The meaning that we generate for a *transition* non-terminal is simply the switch clause that is used when `process` switches on the input symbol.

We also need to handle the case where the input symbol is not valid in the current state. The specification requires that we throw a `RuntimeException` in that case. We achieve this by defining a *default_transition*, which plays the role similar to that of a *transition* but is not specific to any particular input symbol. Figure 6.11 shows how the *default_transition* is handled. The default transition is realized as the `default` clause on the switch that dispatches on the input symbol in the current state. We factored this out into a separate non-terminal partly by analogy with the other transitions, and partly as a hook for extending the application to allow the user to specify a different default transition behavior.

```
default-transition ⇒
  default_transition
    = /* empty... no override of error behavior for now */
    == compile( :switch_clause,
                [[ default: throw new RuntimeException(); ]] );
```

Figure 6.11: Hook for declaring default transition behavior.

6.2.9 Symbolic State Names

One feature of this application is that states are numbered automatically; the user does not have to deal with state identifiers. This is analogous to the way, for example, the `yacc` compiler generator builds internal dispatch tables with short integer names, but the programmer only has to deal with the symbol state names. This is the kind of detail that practical language users demand.

This feature is implemented by maintaining a mapping from symbolic state names to numbers assigned by the extension. A straightforward mapping is sufficient in this case. The symbols are assigned identifiers 0, 1, ... in the order in which the declarations appear in the automaton.

If we also define Java-level symbols to map the symbols to the internal id's, then this simplifies the extension author's job. We do so in Figure 6.12. For each state, a Java class variable is defined that takes on the value of the internally assigned identifier. With these definitions supplied in the generated Java class, the author may now do local compilations using Java to map the symbolic state names to the internal integer number. For example, see Figure 6.8, where the compilation of the various `switch` clauses can proceed without having to map to state numbers in meta-language code.

statename-statics ⇒

```
  statenames = map( lambda(k,n)
      {
          compile( :class_body_decl,
                   [[ private final static int name = k; ]],
                   list( list( :name, :java_id, n ),
                         list( :k, :expr, literal(k) ) ) );
      },
                    range( length( sx ) ),
                    map( car, sx ) );
```

Figure 6.12: Building the Java definitions of symbol state names, mapping state names to internal identifiers.

Usually, this is done for convenience in implementing the language extension and sometimes incurs a cost because the underlying language is not likely to have optimizations to deal with the generated code structures. In this case, we expect the underlying language compiler to inline the values of these symbols (they are declared “final” so the compiler knows they cannot change at runtime), resulting in no runtime performance penalty.

6.3 Example Use of the DFA Extension

In this section, we return to the sample finite state machine described in the introduction of this chapter and show the Java code equivalent to what the DFA language extension generates to implement it.

6.3.1 Sample Extended-Java File

This is the input file, which imports the DFA syntax and defines an automaton called `InteriorString`.

```
import syntax DFA.dfa;

automaton InteriorString {
  public int count = 0;

  start { 0 -> pre; }
  pre { 1 -> mid { count++; }
        0 -> pre; }
  mid { 1 -> mid { count++; }
        0 -> post; }
  post accept { 0 -> post; }
}
```

6.3.2 Generated class definition

Here is the Java code generated as a result of processing the above automaton declaration. The `process` method of the class definition is elaborated in the next section.

```
class InteriorString implements Automaton {

  private int state = start;
  public int count = 0;
  private final static int start = 0;
  private final static int pre = 1;
  private final static int mid = 2;
  private final static int post = 3;

  public String getState(  ) {
    switch ( state )
      {
        case start:
          return "start";
        case pre:
          return "pre";
        case mid:
          return "mid";
        case post:
          return "post";
      }
    return "?unknown";
  }

  public void process( int input[] ) { ... }

  public boolean isAccepting(  ) {
    switch ( state )
      {
        case start:
          return false;
        case pre:
          return false;
        case mid:
          return false;
        case post:
          return true;
      }
    return false;
  }
}
```

6.3.3 Generated process() method

Here is the `process` method. Note the characteristic nested-`switch` structure generated by this Java extension and the use of symbolic names to avoid extra work in the extension itself.

```
public void process( int input[] ) {
  int i;
    for ( i = 0; i < input.length; i++ ) {
      int symbol = input[i];
      switch ( state )
        {
          case start:
            switch ( symbol )
              {
                case 0:
                  state = pre;
                  break;
                  default:throw new RuntimeException(  );
              } break;
          case pre:
            switch ( symbol )
              {
                case 1:
                  {
                    count++;
                  }

                  state = mid;
                  break;
                case 0:
                  state = pre;
                  break;
                default:
                  throw new RuntimeException(  );
              }
            break;
          case mid:
            switch ( symbol )
              {
                case 1:
                  {
                    count++;
                  }

                  state = mid;
                  break;
                case 0:
                  state = post;
                  break;
                default:
                  throw new RuntimeException(  );
              }
            break;
          case post:
            switch ( symbol )
              {
                case 0:
                  state = post;
                  break;
                default:
                  throw new RuntimeException(  );
              }
            break;
        }
    }
}
```

Chapter 7

Final Words

7.1 Related Work

7.1.1 Syntactic Exposures

Earlier work in generalizing syntactic closures, syntactic exposures [10], is along the same lines as our approach, with the interleaving of compilation and expansion for the purpose of improved pattern matching abilities. Our implementation handles some cases of advertent capture better, especially when macros are used to define macros and pattern variables are used in macro pattern rules.

7.1.2 Term Rewriting

There is some similarity between our general approach and the way some term rewriting systems compute normalized head forms eagerly. In defining an evaluation strategy for term rewriting, the system described by Nakamura *et al.* [31, 32] computes a function ψ of the operator, which is used as meta-data to drive the rewriting of terms in the expression. For example, when presented with an expression like $x + y$, the rewriting of sub-terms x and y is controlled by the value of the strategy function $\psi(+)$. The eager evaluation of the head form (the head form here is the operator +) is being done for the same reason we resolve the head form first – in order to determine the appropriate action to apply to the expression controlled by the operator. In essence, both their approach and ours compute the control information first and then delegate the interpretation of the entire form based on the result of that control information. However, our approach achieves the effects without the space cost of general rewriting.

7.1.3 Hygienic Macro Expansion

As pointed out earlier, our approach achieves the general goals of hygienic macro expansion as described by Clinger and Rees [13]. The interleaved expansion and macro scanning process described there is similar to our lazy processing, although we interleave with actual compilation. Interleaving compilation with macro processing opens the door to macro dispatch (*i.e.*, pattern matching) based on the intermediate results of compilation (synthesized attributes), which we exploit in our type-reflective macros.

Our general approach is similar to that of syntactic closures [25], in that we capture the syntactic environment at the point of macro definition. However, our implementation separates the lexical location ("place") from the dynamic location. Among other things, this allows us to identify distinct variables in one interleaved compilation pass instead of re-processing the input with relabeled identifiers.

7.1.4 Reusable Generative Programming

The work by Krishnamurthi *et al.*on McMicMac [27] views extensible languages over the Lisp family as a generative programming problem. The main thrust in that work is to develop languages using reusable modules of linguistic features called *vocabularies*. Their vocabulary development takes place in a separate space, so it is not reflective in the same sense. Their framework addresses the issues of our internal compiler meta-object protocol, an aspect that, while important for its engineering considerations, is not the main thrust of our work.

Their vocabulary abstraction is especially interesting, as it could address the proliferation of compilation types in our system. That is, we have separate sub-compilers for list forms, atoms, *etc.*, which we implement using generic function dispatch. Their dispatching approach could unify the different levels of dispatch in our implementation. And, whereas we use the existing language's module framework to organize our syntactic extensions, their concept of vocabularies is a useful way to abstract extensions.

Although we believe their framework could be used to implement our approach, their default assumption about how to handle macros in such a language is still based on source→source transformation, and hence is less efficient and unable to leverage compile-time knowledge to drive transformation.

7.1.5 Adaptable Grammars

In the literature of extensible parsing, the subclass of adaptable grammars known as *Recursive Adaptable Grammars*, or RAGs, contains a notion similar to that of contour sensitivity, and the goals in expressiveness are similar to our own [35]. RAGs are the result of an effort to preserve the modeling and analytic benefits of context free grammars while permitting local variations in the grammar.

It is interesting to note that some would push the adaptable grammar to the point of subsuming all attribute calculations within the syntax. For example, Christiansen [12] is concerned with the use of adaptable grammars to unify static semantics with the grammar, similar to our example of making `break` a valid statement only in the scope of a breakable construct. The general approach involves defining a grammar rule to store information that is traditionally stored in a symbol table or computed from the meaning *a la* attribute grammars. Our own opinion is that this goes too far and that the language implementation benefits from having the power of both a declarative description of mostly-context-free patterns to match surface syntax and the full procedural expressiveness of an underlying meta-language.

Furthermore, the specific implementation described by Christiansen [12] seems to utilize a global grammar table, and thus has difficulty cleaning up when exiting block scopes. Although the thrust of our work is not specifically in the translation of static semantic constraints into the grammar, the ease with which our implementation manages local grammar changes would make such a strategy somewhat more straightforward.

7.1.6 Open Compilers and MOPs

In developing a language system that makes use of an extensible grammar, the language designer typically defines a particular protocol that can be followed by the language extender to define a new language extension. This protocol involves things like the well-defined non-terminals through which most of the extensions are expected to take place. For example, we expect that most systems defining a conventional programming language will have hooks for statements, expressions, and variable declarations, at the least. The use of *vocabularies* in McMicMac [27] as a language extension modularity mechanism is a more formalized approach to what would be well-known hook non-terminals in the implementation described here.

The Intentional Programming project [2, 36] is concerned with the definition of language abstractions as transformations on meaning structures. However, that work operates directly against the meaning structures, with textual syntax limited to an input mechanism. Furthermore, transformations are implemented procedurally against the compiler's meta-objects, which unnecessarily separates the intention developer from the developer in the target language.

Other work in transformational programming, such as that of Visser [43], incorporates the ability to define transformations in terms of the concrete syntax of the program, along with the ability to incorporate new syntax. However, their syntax is statically defined (although partitioned into composable modules) and global for a module, whereas our system supports local syntax extensions.

The Jakarta tool suite (JTS) is a facility that addresses the issue of language extensibility as a set of generator components [4]. This has the benefit of supporting modularity of the language extension features; in fact, JTS leverages the component composition of the JTS framework itself. One of the main focuses of the JTS work is support for composable domain-specific languages. Our approach achieves syntactic modularity but has no special support for modularizing its meta-language procedures. Incorporating advanced modularization features into our approach, as is done in JTS, is a rich area for future research.

The <bigwig> project [7] is targeted at defining interactive Web services and can handle syntactic extensions together with the transformations to more primitive language structures. However, in <bigwig>, macros must start with an identifier, which shares the limitations of C's macro system; for example, new infix operators cannot be introduced. Also, there is no procedural meta-language for expressing transformations more complex than pattern-based transformations.

The Java Syntactic Extender [3] work uses an approach similar to that of C for defining macros. The set of syntactic forms which are considered extensible are statically limited, although JSE improves upon C by allowing statements as well as procedure call forms to be extended. JSE is not dynamically extensible in the sense that our system allows users to define new syntactic forms of any sort.

Other approaches to language extensibility have been taken. The meta-object protocol approach [24, 11] enables the extension author to define how certain language constructs are processed. In these systems, the underlying object system provides the structural framework to which the extensions are attached. The extensions defined using our approach are associated with the syntactic representation of the program

A meta-program in our approach executes in the context of the compiler, but is not explicitly part of the compiler. The domain objects of the meta-programmer in our approach are meanings, parse environments, and token strings. For example, using just our approach, the meta-programmer cannot alter the internal processing of a built-in language construct such as the layout of a data structure without redefining the syntactic entities that introduce them. This is in contrast to the open compiler work of Lamping *et.al* [28], which is specifically intended to incrementally redefine internal compiler processing using a compile-time meta-object protocol.

7.2 Limitations and Future Work

7.2.1 Meta-Object Protocol

Considerably more formalism can be developed around the syntax and semantics of our meta-language itself. A more systematic development of the meta-language would allow the definition of a meta-object protocol (MOP) as the foundation of the extensible language framework [28]. With a well-defined MOP in place, the meta-syntax can be constructed in terms of that protocol, enabling user-defined extensions to our meta-language[1].

7.2.2 Error Reporting

More work is needed to determine how best to report syntactic errors in the context of a dynamic grammar. As in most language systems, the emphasis in our work is on core functionality when presented with correct input and, to a lesser extent, on detecting incorrect input. But for a language system to be practical, it must report errors in a way that is useful to the programmer.

The interpretive nature of the Earley parser is useful for error reporting

[1] User-defined extensions are possible in our current implementation. In fact, our system is bootstrapped in multiple stages using our own meta-syntax to do so. However, the syntactic hooks necessary to do so are not well formalized.

in some ways, because it is easy to compute the set of constructs which are being parsed at any given point, even in the presence of syntactic extensions[2]. However, more work is needed to refine these capabilities in the current implementation to report even more useful error conditions and ideally to implement error recovery so that parsing can continue on a best-effort basis.

7.2.3 Synthesized Attributes

More work is needed to understand the relationship between the eager evaluation of intermediate code and the lazier evaluation of meaning on the basis of the template. This is especially a problem as the template may try to introduce binding constructs which, depending upon context, could change how the pattern element should be compiled.

Static Syntax Type Checking

Our implementation uses dynamic type checking in the meta-language. Other systems [9, 8, 5] apply static type rules to ensure that any meaning constructed by an extension is a valid composition of primitive meaning operators and values. Furthermore, they ensure that the occurrence of the extension construct in a program always produces the right kind of meaning at the right place in the target program. Additional work is needed to analyze applications of our approach to ascertain whether static checking is helpful or a hindrance.

[2] In fact, we found it so easy that just for the purpose of debugging the examples in Chapter 6, we significantly improved the error reporting capabilities

Convenience Notations

One of the practical limitations of our current extensible parsing system revolves around the problem of interpolating single syntactic constructs into a sequence. For example, in Figure 6.3 we had to explicitly `append` several pieces together in order to form the body of a `class` definition. We would rather have been able to write something like:

```
compile( :type_decl,
         [[ class aname implements Automaton
            {
              private int state = start;
              vars
              statenames
              process_method
              getstate_method
              isaccepting_method
            }
         ]] );
```

and let the parsing engine do the interpolation of `vars` and `statenames`. There are several ways this could be implemented within the current framework. The basic idea is to extend the expansion of the repetition construct. Recall that the repetition construct is responsible for transforming a Kleene star pattern (*e.g.*, "*decl**") into grammatical primitives suitable for interpretation by the Earley parser.

One approach would be, in the expansion of "$A*$," to define a rule such as:

$$A\_\_star \rightarrow A\_\_star \quad A\_\_star \qquad \texttt{= append(\$1,\$2)}$$

This has the disadvantage of being extremely ambiguous. While this could be parsed, it tends to degenerate into $O(n^2)$ complexity.

Another approach would be to define a non-terminal name which could not be parsed directly (call it `A__seq`), and could only be satisfied by a syntactic substitution. Then, an additional rule in the repetition expansion could be defined:

$$A\_\_star \rightarrow A\_\_seq \quad A\_\_star \qquad \texttt{= append(\$1,\$2)}$$

This has the disadvantage of introducing a new and irregular kind of object into the parsing engine.

Another approach would be to require the programmer to signal the interpolation. Then, the repetition expansion could include:

$$A\_\_star \rightarrow \text{``}\texttt{@}\text{''} \quad \text{ident} \quad A\_\_star \qquad \texttt{= append(expand(\$2),\$3)}$$

This runs the risk of conflicting with the grammar of the target language, although for any particular target language a suitable indicator could be identified. This also ambiguates the use of the alias identifier in the token stream as standing for itself in an expansion and standing for what it denotes as an alias.

In general, more applications should be built using this approach in order to identify the practical limitations of the system and point the way to additional convenience notations. To make this work more widely available, and to test its end-to-end performance, one area of future work is to integrate the current extensible Java into a full compiler. The Jikes implementation [19] looks especially promising for this purpose, as it is fast and well structured for replacing the parser front end.

7.3 Conclusions

The expressive power of Lisp macros can be made available to programmers of languages with traditional syntax, which allows the easy development of new language features and the modular construction of domain-specific language extensions. Furthermore, integrating syntactic extensibility with true compilation enables the reflection of synthesized attributes into syntactic processing, which in turn increases the expressive range of declarative, syntax-based language extensions.

This work is the first to efficiently apply the concepts of declarative and procedural macro processing to the domain of languages with a traditional syntax, such as Java and C. We have shown how macro processing can be interleaved with compilation, and how an efficient, locally extensible parser can be used to execute macro expansions at parse time.

With an extensible parsing framework that includes the capabilities of macro transformation, a language system that adds extensibility to Java can be constructed. An extensible Java implementation can be used to rapidly develop new language features and to define domain-specific languages as illustrated in Chapter 6.

This is the first system to combine support for local, dynamically extensible context-free syntax with control by meta-level procedures. It is clear to us that this capability leads to a more robust programming paradigm, in which the roles of programmer and language author blur, and application-specific language extensions become a standard mechanism for modular abstraction.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] W. Aitken, B. Dickens, P. Kwiatkowski, O. de Moor, and D.Richter. Transformation in intentional programming. In *Proceedings Fifth International Conference on Software Reuse*, Victoria, B.C., Canada, June 1998. IEEE.

[3] Jonathan R. Bachrach and Keith Playford. The Java syntactic extender (JSE). In *Proceedings of the 2001 ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2001).* ACM, 2001.

[4] Don Batory, Bernie Lofaso, and Yannis Smaragdakis. JTS: tools for implementing domain-specific languages. In *Proceedings Fifth International Conference on Software Reuse*, pages 143–53, Victoria, B.C., Canada, June 1998. IEEE.

[5] Alan Bawden. First-class macros have types. In *Conf. Rec. POPL '00: 27th ACM Symp. Princ. of Prog. Langs.*, pages 133–141, 2000.

[6] Alan Bawden and Jonathan Rees. Syntactic closures. In *Conference on LISP and Functional Programming*, pages 86–95, 1988.

[7] C. Brabrand and M. Schwartzbach. Growing languages with metamorphic syntax macros, 2000. Submitted for publication; http://www.brics.dk/bigwig/.

[8] Luca Cardelli, Florian Matthes, and Martín Abadi. Extensible grammars for language specialization. In *Proceedings of the Fourth International Workshop on Database Programming Languages, August 1993, Manhattan, New York*. Springer Verlag, 1994.

[9] Luca Cardelli, Florian Matthes, and Martín Abadi. Extensible syntax with lexical scoping. SRC Research Report 121, Digital Equipment Corporation, 1994. ftp://gatekeeper.dec.com/pub/DEC/SRC/research-reports/.

[10] Stephen Paul Carl. Syntactic exposures – a lexically-scoped macro facility for extensible languages. Master's thesis, University of Texas at Austin, 1996. ftp://ftp.cs.utexas.edu/pub/garbage/carl-msthesis.ps.

[11] Shigeru Chiba. A metaobject protocol for C++. In *Proceedings of the 1995 ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 1995)*, 1995.

[12] H. Christiansen. A survey of adaptable grammars. *SIGPLAN Notices*, 25(11):35–44, 1990.

[13] Will Clinger and Jonathan Rees. Macros that work. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 155–162, January 1991.

[14] R. Kent Dybvig. *The Scheme Programming Language.* Prentice Hall, second edition, 1996.

[15] Jay Earley. *An Efficient Context-free Parsing Algorithm.* PhD thesis, Carnegie-Mellon University, 1968.

[16] Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, February 1970.

[17] Paul Graham. *On Lisp.* Prentice Hall, 1994.

[18] J. Heering, P. Klint, and J. Rekers. Incremental generation of parsers. *IEEE Transactions on Software Engineering*, 16(12):1344–1351, 1990. Also in SIGPLAN notices 24(7):179-191, 1989.

[19] IBM. IBM developerWorks - Open Source Software - Jikes' Home. http://oss.software.ibm.com/developerworks/opensource/jikes/.

[20] ISO. PL/I. Technical Report 6160, ISO, 1979.

[21] Steven C. Johnson. Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979.

[22] Guy L. Steele Jr. *Common Lisp: The Language.* Digital Press, 1984.

[23] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, second edition, 1988.

[24] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[25] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *1986 ACM Conference on Lisp and Functional Programming*, pages 151–159, 1986.

[26] Shriram Krishnamurthi. *Linguistic Reuse.* PhD thesis, Rice University, 2001.

[27] Shriram Krishnamurthi, Matthias Felleisen, and Bruce F. Duba. From macros to reusable generative programming. Technical Report TR 00-364, Rice University, 2000. http://www.ccs.neu.edu/scheme/pubs/tr00-364.ps.gz.

[28] John Lamping, Gregor Kiczales, Luis Rodriguez, and Erik Ruf. An architecture for an open compiler. In *Proceedings of the International Workshop on Reflection and Meta-Level Architecture*, 1992.

[29] X/Open Company Ltd. DCE: Remote procedure call. Technical Report P312, X/Open Company Ltd., 1995. http://www.linuxworld.com/linuxworld/lw-1999-09/lw-09-corba_1-2.html.

[30] Philippe McLean and R. Nigel Horspool. A faster Earley parser. In *Proceedings of the International Conference on Compiler Construction*, pages 281–293, 1996.

[31] M. Nakamura and K. Ogata. The evaluation strategy for head normal form with and without on-demand flags. In *In Proc. of 3rd International Workshop on Rewriting Logic and its Applications, WRLA'00*, volume 36 of *Electronic Notes in Theoretical Computer Science.* Elsevier Sciences, 2001.

[32] Masaki Nakamura. *Evaluation Strategies for Term Rewriting Systems.* PhD thesis, Japan Advanced Institute of Science and Technology, 2002.

[33] Christian Queinnec. *Lisp in Small Pieces.* University of Cambridge, 1994.

[34] Tim Sheard and Neal Nelson. Type safe abstractions using program generators. Technical Report 95-013, Oregon Graduate Institute of Science and Technology, 1995.

[35] John N. Shutt. Recursive adaptable grammars. Master's thesis, Worcester Polytechnic Institute, 1993.

[36] C. Simonyi. The death of computer languages, the birth of intentional programming. Microsoft Technical Report MSR-TR-95-52, Microsoft Research, 1995.

[37] Mark Stickel and Richard Waldinger. Deductive composition of astronomical software from subroutine libraries. In *Twelfth International Conference on Automated Deduction*, pages 341–355, June 1994.

[38] Christopher Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1/2):11–49, 2000.

[39] Gregory T. Sullivan. Aspect-oriented programming using reflection and metaobject protocols. *Communications of the ACM*, 44(10):95–97, 2001.

[40] M. Tomita. *Efficient Parsing for Natural Language.* Kluwer, 1985.

[41] Steve Upstill. *The Renderman Companion: A Programmer's Guide to Realistic Computer Graphics.* Addison-Wesley, 1990.

[42] Arie van Deursen. The static semantics of pascal. In Arie van Deursen, Jan Heering, and Paul Klint, editors, *Language Prototyping: An Algebraic Specification Approach*, number 5 in AMAST Series in Computing, pages 31–52. World Scientific, 1996.

[43] Eelco Visser. Meta-programming with concrete object syntax. Technical Report UU-CS-2002-028, Institute of Information and Computing Sciences, Utrecht University, 2002. To appear in LNCS, October, 2002.

[44] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide*. Addison-Wesley, third edition, 1999.

Vita

Donovan Michael Kolbly was born in Apple Valley, California on October 30, 1967, the son of Phyllis Stevenson Kolbly and Richard Bauer Kolbly. After completing his High School education at Barstow High School in Barstow, California, he entered the Physics program at California State Polytechnic University, Pomona. He received a Bachelor of Science degree from that institution in March 1990. During the summer of 1990, he attended an extension program of the University of New Mexico in Los Alamos, New Mexico. In the Fall of that year, he entered the Graduate School at the University of Texas at Austin in the Physics department. In 1991, he transfered to the Computer Sciences department, and obtained a Master of Science degree in December 1994.

Permanent Address: 8710 Mosquero Circle
Austin, Texas 78748

This dissertation was typeset with LaTeX 2_ε by the author.