INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Practical Earley Parsing and the SPARK Toolkit

by

John Daniel Aycock
B.Sc., University of Calgary, 1993
M.Sc., University of Victoria, 1998

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

We accept this dissertation as conforming
to the required standard

Dr. R. N. Horspool, Supervisor (Department of Computer Science)

Dr. J. H. Jahnke, Departmental Member (Department of Computer Science)

Dr. M.-A. D. Storey, Departmental Member (Department of Computer Science)

Dr. K. F. Li, Outside Member
(Department of Electrical and Computer Engineering)

Dr. T. A. Proebsting, External Examiner (Microsoft Research)

Supervisor: Dr. R. N. Horspool

Abstract

Domain-specific, "little" languages are commonplace in computing. So too is the need to implement such languages; to meet this need, we have created SPARK (Scanning, Parsing, And Rewriting Kit), a toolkit for little language implementation in Python, an object-oriented scripting language.

SPARK greatly simplifies the task of little language implementation. It requires little code to be written, and accommodates a wide range of users — even those without a background in compiler theory. Our toolkit is seeing increasing use on a variety of diverse projects.

SPARK was designed to be easy-to-use with few limitations, and relies heavily on Earley's general parsing algorithm internally, which helps in meeting these design goals. Earley's algorithm, in its standard form, can be hard to use; indeed, experience with SPARK has highlighted several problems with the practical use of Earley's algorithm. Our research addresses and provides solutions for these problems, making some significant improvements to the implementation and use of Earley's algorithm.

First, Earley's algorithm suffers from the *performance problem.* Even under optimum conditions, a standard Earley parser is burdened with overhead. We extend directly-executable parsing techniques for use in Earley parsers, the results of which run in time comparable to the much-more-specialized LALR(1) parsing algorithm.

Second is what we call the *delayed action problem.* General parsers like Earley

must, in the worst case, read the entire input before executing any semantic actions associated with the grammar rules. We attack this problem in two ways. We have identified conditions under which it is safe to execute semantic actions on the fly during recognition; as a side effect, this has yielded space savings of over 90% for some grammars. The other approach to the delayed action problem deals with the difficulty of handling context-dependent tokens. Such tokens are easy to handle using what we call "Schrödinger's tokens," a superposition of token types.

Finally, Earley parsers are complicated by the need to process grammar rules with empty right-hand sides. We present a simple, efficient way to handle these empty rules, and prove that our new method is correct. We also show how our method may be used to create a new type of LR(0) automaton which is ideally suited for use in Earley parsers.

Our work has made Earley parsing faster and more space-efficient, turning it into an excellent candidate for practical use in many applications.

Examiners:

Dr. R. N. Horspool, Supervisor (Department of Computer Science)

Dr. J. H. Jahnke, Departmental Member (Department of Computer Science)

Dr. M.-A. D. Storey, Departmental Member (Department of Computer Science)

Dr. K. F. Li, Outside Member
(Department of Electrical and Computer Engineering)

Dr. T. A. Proebsting, External Examiner (Microsoft Research)

Table of Contents

List of Tables

List of Figures

Acknowledgments

Where to begin?

Nigel Horspool has been steadily trying to teach me since 1996 what this whole research thing is all about. Hopefully some of the lessons have sunk in.

As my supervisor, Nigel has been involved in discussions regarding the material in this thesis since the beginning, and some ideas presented here are attributable to him. Specifically, he wisely insisted on DEEP executing parent sets, and skipping terminal code after it had been executed once. The grammar in Figure 7.1 was derived from an example he supplied which broke my first attempt to fix empty rules. In addition, he pointed out why the Perl grammar was causing problems, requested mean item retention data, and asked what happened when empty rules met useless nonterminals.

A variety of papers comprise this thesis. I have received innumerable helpful comments on them from Nigel Horspool, Shannon Jaeger, and anonymous referees from *Software — Practice and Experience*, *Information Processing Letters*, *CC 2001*, and *IPC7*. Shannon Jaeger and Jim Uhl quickly tackled the proofreading task after the thesis was complete, for which I am extremely grateful.

Many have kindly contributed bits of information. Jon Bentley and Mary Shaw

told me the etymology of "little languages"; Chris Verhoef described the uses of general parsers in software reengineering; Friwi Schröer gave me some help with ACCENT; Thilo Ernst supplied an update on TRAP's status.

Users of SPARK have always been forthcoming with comments, commendations, and complaints. Through their feedback, I have been shown clever ways to apply SPARK that I never would have discovered otherwise. I would especially like to thank Rick White for discovering the bug with empty rules.

I will miss my morning coffee runs with Mike Zastre, in which we have shared joys, sorrows, and devised strategies for handling unruly supervisors. Mike helped me find a coherent design for Figure 2.3, and he and his wife Susanne Reul verified the translation of a key part of Schrödinger's paper.

I would also like to thank my officemate Kelvin Yeow, who has politely refrained from complaining while I wrote my thesis. This, despite the fact that the "i" key on my keyboard keeps jamming and making a loud squeaking noise every single time I press it.

Most importantly, I would like to thank my family. Shannon, Melissa, and Amanda all made sacrifices during the last few years, and it is to them that I dedicate this work.

Chapter 1

Introduction

The development of the first high-level programming language was heralded not with a bang, but a whimper. This language, Zuse's Plankalkül, was devised in 1945 but unpublished and unknown until the 1970s [14, 41, 60]. By that time, high-level language development was in full swing. Languages such as ALGOL, APL, Fortran, LISP, and Simula gave an increasing amount of variety as well as establishing entirely new programming paradigms.

In contrast, there are relatively few big, general-purpose languages developed now. Those that do appear, such as Java[1] and C#, tend to be little more than variations on earlier themes, despite the fanfare that typically accompanies their arrival. The major thrust of compiler research is now efficient implementation of programming languages rather than language development *per se* [45].

What is underappreciated is the ubiquity of language in computing to describe

[1] Java is a trademark or registered trademark of Sun Microsystems, Inc. in the United States and other countries.

smaller, more specific areas. Configuration files, HTML documents, shell scripts, network protocols — all are little structured languages, yet may lack the generality and features of full-blown programming languages.

About 1980, Mary Shaw coined the term "little language" to describe this phenomenon [18]; the term was later popularized by Jon Bentley [17]. Although the preferred label now is "domain-specific language," the idea is the same. Shaw used the term to draw attention to the fact that, at the time, only a small amount of effort was spent in little language design compared to larger languages, yet the effects of a bad little language design could be disproportionately high [85].

Could we not just design a single, "perfect" little language, and re-use it for all application domains? Some think so — Shivers [86] presents an alternative to little languages and a Scheme-based implementation framework. Tcl was also developed to address the proliferation of little languages [75]. However, the reality is that no convergence is likely in the foreseeable future; new little language designs still debut frequently.

Of course, both design *and* implementation techniques can be used across the spectrum of languages. Whether writing an interpreter for a little language, or compiling a little language into another language, compiler techniques can be used.

In many cases, an extremely fast compiler is not needed, especially if the input programs tend to be small. Instead, issues can predominate such as compiler development time, maintainability of the compiler, and the ability to easily add new language features. Such prototyping is the strong suit of Python [15], an object-oriented scripting language. When this work began in 1998, what Python did not have was a tool

to support implementation of little languages from start to finish, which led to our development of SPARK, the Scanning, Parsing, And Rewriting Kit.[2]

SPARK is an object-oriented framework supporting compilation of little languages. SPARK is easy to use, even by nonspecialists, and is being applied to an increasing number of areas. Roughly half of the objects that SPARK supplies rely internally on Earley's parsing algorithm [30, 31]. It is a general algorithm, capable of using any context-free grammar — most parsing algorithms in practical use today only handle various subsets of unambiguous grammars.

Experience with SPARK has demonstrated a number of practical problems with the use of Earley's algorithm. Our research addresses these problems: token interpretation, parsing speed, on-the-fly execution of semantic actions, and handling of grammar rules with empty right-hand sides. These results are generally applicable outside of the context of SPARK; some of this work has already appeared, and more is to appear, as separate papers.

The remainder of this thesis is organized in the following manner. We begin by presenting SPARK, both in terms of its usage and its internal workings. We then give some formal definitions, and a precise specification of Earley's algorithm in which we characterize some of the algorithm's problems. From there, our research work:

- Directly-executable Earley parsing. This improves performance of Earley's algorithm to the point where it is comparable to the less general, faster LALR(1)

[2] Another Python-based tool for compilation of domain-specific languages, TRAP [32, 33], was announced in 1999. It required a user-initiated compiler build phase, used a less powerful parsing method, and had considerably more complicated semantics. However, TRAP is no longer being actively developed [34].

parsing algorithm.

- Schrödinger's tokens, a technique for easily handling context-dependent tokens in conjunction with general parsers like Earley. Applications include little languages as well as programming languages which have been traditionally hard to parse, such as PL/I.

- Early actions, in which we establish conditions under which semantic actions can be executed in an Earley parser during recognition. This is shown to dramatically reduce the parser's run time space consumption.

- An improved way to process empty rules in Earley parsers, allowing simplification of the algorithm. This leads to the construction of parsing automata which are tailored for efficient use in an Earley parser.

Finally, we conclude this thesis with some directions for future work.

Chapter 2

SPARK: Scanning, Parsing, And Rewriting Kit[3]

SPARK is a framework for compilation of little languages that plays many rôles in the research we will later present. SPARK has incited us to look at the problems we will describe; it has acted as a testbed for some of our solutions; it will be the beneficiary of some of our results.

First unveiled in 1998, SPARK is now on its sixth release. In that time, SPARK has received a favorable mention in print [28] and has been used in a wide variety of projects, a selection of which are listed below. Our projects are denoted by a circle; those done by others are bulleted. Other people's projects without citations were communicated to us via email.

- Compiling Guide [69], a web programming language

[3] An earlier version of this work appeared in [7].

- Compiling a subset of Java
- Experimental type inferencing for Python [8]
- Bytecode decompilation (now maintained by others)

- VHDL parsing
- Extraction of embedded program documentation
- GUI building
- Linux[4] kernel configuration system [80]
- Interfacing with IRAF (astronomical software) [98]
- Fortran interface description [29]
- Producing syntax charts from a grammar
- Domain-specific extensions to Python

In this chapter we introduce SPARK and show how its design motivated our research work.

2.1 Model of a Compiler

Like most nontrivial pieces of software, compilers are generally broken down into more manageable modules, or phases. The design issues involved and the details of each

[4]Linux is a trademark of Linus Torvalds.

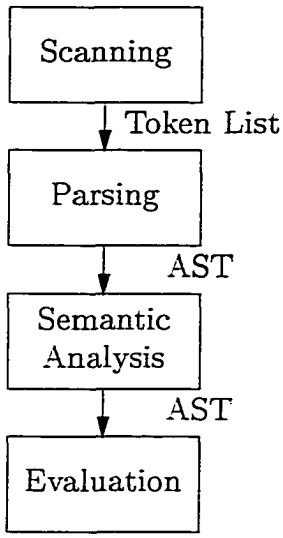


Figure 2.1: Compiler model.

phase are too numerous to discuss here in depth; there are many excellent books on the subject, such as [3] and [6].

We begin with a simple model of a compiler having only four phases, as shown in Figure 2.1:

1. Scanning, or lexical analysis. Breaks the input stream into a list of tokens. For example, the expression "2 + 3 * 5" can be broken up into five tokens: number plus number times number. The values 2, 3, and 5 are attributes associated with the corresponding number token.

2. Parsing, or syntax analysis. Ensures that a list of tokens has valid syntax according to a grammar — a set of rules that describes the syntax of the language. For the above example, a typical expression grammar would be:

   ```
   expr ::= expr + term
   ```

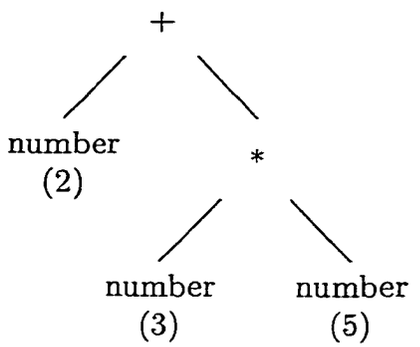


Figure 2.2: Abstract syntax tree (AST).

```
expr ::= term
term ::= term * factor
term ::= factor
factor ::= number
```

In English, this grammar's rules say that an expression can be an expression plus a term, an expression may be a term by itself, and so on. Intuitively, the symbol on the left-hand side of "::=" may be thought of as a variable for which the symbols on the right-hand side may be substituted [97]. Symbols that don't appear on any left-hand side — like +, *, and number — correspond to the tokens from the scanner.

The result of parsing is an abstract syntax tree (AST), which represents the input program. For "2 + 3 * 5," the AST would look like the one in Figure 2.2.

3. Semantic analysis. Traverses the AST one or more times, collecting information and checking that the input program had no semantic errors. In a typical programming language, this phase would detect things like type conflicts, redefined

identifiers, mismatched function parameters, and numerous other errors. The information gathered may be stored in a global symbol table, or attached as attributes to the nodes of the AST itself.

4. Evaluation. This phase may directly interpret the program, or output code in C or assembly which would implement the input program. Evaluation may be implemented by another traversal of the AST, or by matching patterns in the AST. The value of expressions as simple as those in the example grammar could be computed on the fly in this phase.

Each phase performs a well-defined task, and passes a data structure on to the next phase; Grune et al. [45] refer to this as a "broad compiler." Note that information only flows one way, and that each phase runs to completion before the next one starts.[5] This is in contrast to oft-used techniques which have a symbiosis between scanning and parsing, where not only may several phases be working concurrently, but a later phase may send some feedback to modify the operation of an earlier phase.

Certainly all little language compilers won't fit this model, but it is extremely clean and elegant for those that do. The main function of the compiler, for instance, distills into three lines of Python code which reflect the compiler's structure:

```
f = open(filename)
evaluate(semantic(parse(scan(f))))
f.close()
```

Unlike Gaul, the rest of this chapter is only in two parts.[6] First, we will examine each of the four phases, showing how our framework can be used to implement the

[5] Wortman suggests some anecdotal evidence indicating that parts of production compilers may be moving towards a similar model [102].

[6] With apologies to Cæsar [24].

Phase	*Class*
Lexical analysis	GenericScanner
Syntax analysis	GenericParser
Syntax analysis	GenericASTBuilder
Semantic analysis	GenericASTTraversal
Evaluation	GenericASTTraversal
Evaluation	GenericASTMatcher

Table 2.1: SPARK classes, by functionality. Some classes are potentially useful for more than one phase.

little expression language above. Following this will be a discussion of some of the inner workings of the framework's classes, where the reliance of SPARK on Earley parsing will become evident.

2.2 The Framework

A common theme throughout this framework is that the user should have to do as little work as possible. For each phase, our framework supplies a class which performs most of the work; these are summarized in Table 2.1. The user's job is simply to create subclasses which customize the framework.

As the code implementing our running example is distributed throughout this chapter, it can be difficult to gauge factors such as code size and the consistency of SPARK's interface. Appendix A gathers the code together for one implementation, without comment.

2.2.1 Lexical Analysis

Lexical analyzers, or scanners, are typically implemented in one of two ways. The first is to write the scanner by hand; this may still be the method of choice for very small languages, or where use of a tool to generate scanners automatically is not possible. The second method is to use a scanner generator tool, like Lex [67, 68], which takes a high-level description of the permitted tokens, and produces a finite state machine which implements the scanner.

Finite state machines are equivalent to regular expressions; in fact, one typically uses regular expressions to specify tokens to scanner generators! Since Python has regular expression support, it is natural to use them to specify tokens. (As a case in point, the Python module "tokenize" has regular expressions to tokenize Python programs.)

So GenericScanner, our generic scanner class, requires a user to create a subclass of it in which they specify the regular expressions that the scanner should look for. Furthermore, an "action" consisting of arbitrary Python code can be associated with each regular expression — this is typical of scanner generators, and allows work to be performed based on the type of token found.

Below is a simple scanner to tokenize expressions. The parameter to the action routines is a string containing the part of the input that was matched by the regular expression.

```
class SimpleScanner(GenericScanner):
    def __init__(self):
        GenericScanner.__init__(self)
```

```
def tokenize(self, input):
    self.rv = []
    GenericScanner.tokenize(self, input)
    return self.rv

def t_whitespace(self, s):
    r' \s+ '

def t_op(self, s):
    r' \+ | \* '
    self.rv.append(Token(type=s))

def t_number(self, s):
    r' \d+ '
    t = Token(type='number', attr=s)
    self.rv.append(t)
```

A few words about the syntax and semantics of Python are in order. This code defines the class `SimpleScanner`, a subclass of `GenericScanner`. All methods have an explicit `self` parameter; `__init__` is the class' constructor, and it is responsible for invoking its superclass' constructor if necessary. Methods may optionally begin with a documentation string ("docstring" in Python parlance) which is ignored by Python's interpreter but, unlike a regular comment, is retained and accessible at run time. A method which is empty (save for an optional documentation string) has no effect when executed.

Object instantiation uses the same syntax as function calls; in the above code, `Token` objects are being created. Both object instantiation and function invocation can make use of "keyword arguments," which permit actual and formal parameters to be associated by name rather by their position in the argument list. Some final minutiæ: `[]` is the empty list, and an "r" prefixing a string denotes a "raw" string

Input	*Method*	*Token Added*
2	t_number	number (attribute 2)
space	t_whitespace	
+	t_op	+
space	t_whitespace	
3	t_number	number (attribute 3)
space	t_whitespace	
*	t_op	*
space	t_whitespace	
5	t_number	number (attribute 5)

Table 2.2: Trace of SimpleScanner.

in which backslash characters are not treated as escape sequences.

Returning to the scanner itself, each method whose name begins with "t_" is an action; the regular expression for the action is placed in the method's documentation string. (The reason for this unusual design, using reflection, is explained in Section 2.3.1.)

When the tokenize method is called, a list of Token instances is returned, one for each operator and number found. The code for the Token class is omitted; it is a simple container class with a type and an optional attribute. White space is skipped by SimpleScanner, since its action code does nothing. Any unrecognized characters in the input are matched by a default pattern, declared in the action GenericScanner.t_default. This default method can of course be overridden in a subclass. A trace of SimpleScanner on the input "2 + 3 * 5" is shown in Table 2.2.

Scanners made with GenericScanner are extensible, meaning that new tokens may be recognized simply by subclassing. To extend SimpleScanner to recognize floating-point number tokens is easy:

```
class FloatScanner(SimpleScanner):
    def __init__(self):
        SimpleScanner.__init__(self)

    def t_float(self, s):
        r' \d+ \. \d+ '
        t = Token(type='float', attr=s)
        self.rv.append(t)
```

How are these classes used? Typically, all that is needed is to read in the input program, and pass it to an instance of the scanner:

```
def scan(f):
    input = f.read()
    scanner = FloatScanner()
    return scanner.tokenize(input)
```

Here, the entire input is read at once with the `read` method. Once the scanner is done, its result is sent to the parser for syntax analysis.

2.2.2 Syntax Analysis

The outward appearance of GenericParser, our generic parser class, is similar to that of GenericScanner.

A user starts by creating a subclass of GenericParser, containing special methods which are named with the prefix "p_". These special methods encode grammar rules in their documentation strings; the code in the methods are actions which get executed when one of the associated grammar rules are recognized by GenericParser.

The expression parser subclass is shown below. Here, the actions are building the AST for the input program. AST is also a simple container class; each instance of

AST corresponds to a node in the tree, with a node type and possibly child nodes. The grammar's start symbol is passed to the constructor. In the code, ExprParser's constructor assigns a default value to its start symbol argument so that it may be changed later by a subclass.

```
class ExprParser(GenericParser):
    def __init__(self, start='expr'):
        GenericParser.__init__(self, start)

    def p_expr_1(self, args):
        ' expr ::= expr + term '
        return AST(type=args[1], left=args[0], right=args[2])

    def p_expr_2(self, args):
        ' expr ::= term '
        return args[0]

    def p_term_1(self, args):
        ' term ::= term * factor '
        return AST(type=args[1], left=args[0], right=args[2])

    def p_term_2(self, args):
        ' term ::= factor '
        return args[0]

    def p_factor_1(self, args):
        ' factor ::= number '
        return AST(type=args[0])

    def p_factor_2(self, args):
        ' factor ::= float '
        return AST(type=args[0])
```

ExprParser builds the AST from the bottom up. Figure 2.3 shows the AST in Figure 2.2 being built, and the sequence in which ExprParser's methods are invoked.

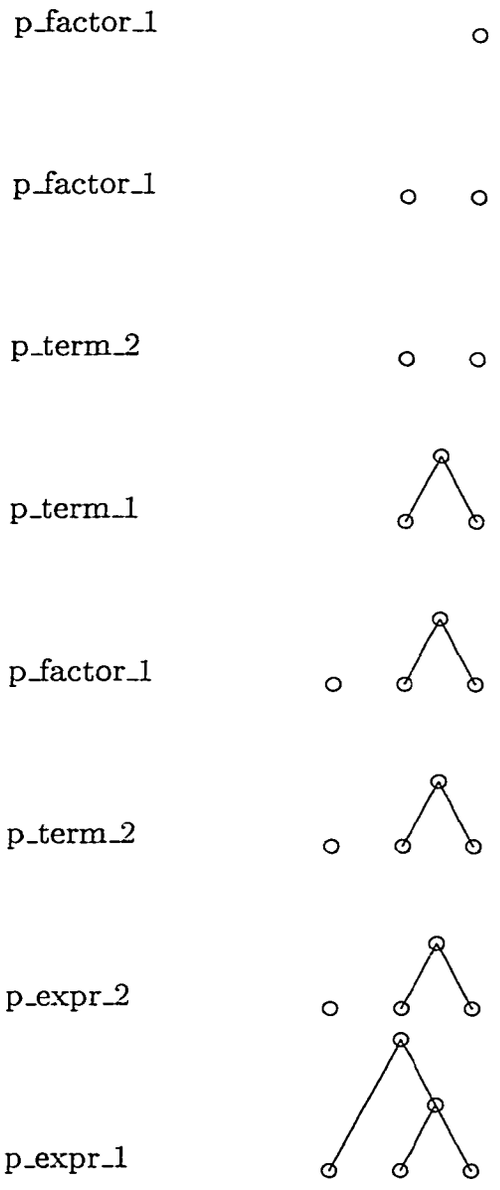


Figure 2.3: AST construction.

The "args" passed in to the actions are based on a similar idea used by Yacc [53, 68], a prevalent parser generator tool. Each symbol on a rule's right-hand side has an attribute associated with it. For token symbols like +, this attribute is the token itself. All other symbols' attributes come from the return values of actions which, in the above code, means that they are subtrees of the AST. The index into args comes from the position of the symbol in the rule's right-hand side. In the running example, the call to p_expr_1 has `len(args) == 3`: args[0] is expr's attribute, the left subtree of + in the AST; args[1] is +'s attribute, the token +; args[2] is term's attribute, the right subtree of + in the AST.

The routine to use this subclass is straightforward:

```
def parse(tokens):
    parser = ExprParser()
    return parser.parse(tokens)
```

Although omitted for brevity, ExprParser can be subclassed to add grammar rules and actions, the same way the scanner was subclassed.

Writing actions to build ASTs for large languages can be tedious. An alternative is to use the GenericASTBuilder class instead of GenericParser, which automatically constructs the tree:

```
class AnotherExprParser(GenericASTBuilder):
    def __init__(self, AST, start='expr'):
        GenericASTBuilder.__init__(self, AST, start)

    def p_expr_1(self, args):
        ' expr ::= expr + term '
    def p_expr_2(self, args):
        ' expr ::= term '
```

```
def p_term_1(self, args):
    ' term ::= term * factor '
def p_term_2(self, args):
    ' term ::= factor '

def p_factor_1(self, args):
    ' factor ::= number '
def p_factor_2(self, args):
    ' factor ::= float '
```

(A more abbreviated way to express this may be found in Section 2.3.3.) The constructor is passed the AST class, so GenericASTBuilder knows how to instantiate AST nodes.

By default, GenericASTBuilder constructs a *concrete* syntax tree which, as Figure 2.4 shows, faithfully reflects the structure of the grammar. Depending on the node type being built, one of two methods is invoked to construct the node: GenericASTBuilder.terminal or GenericASTBuilder.nonterminal. The user may override these with methods which shape an AST rather than a concrete syntax tree.

After syntax analysis is complete, the parser has produced an AST, and verified that the input program adheres to the grammar rules. Next, the input's meaning must be checked by the semantic analyzer.

2.2.3 Semantic Analysis

Semantic analysis is performed by traversing the AST. Rather than spread code to traverse an AST all over the compiler, we have a single base class, GenericASTTraversal, which knows how to walk the tree. Subclasses of GenericASTTraversal supply methods which get called depending on what type of node is encountered.

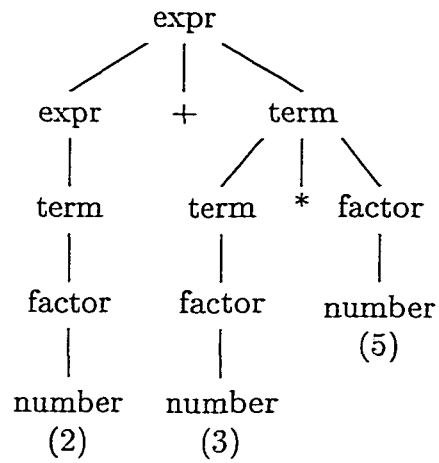


Figure 2.4: Concrete syntax tree.

To determine which method to invoke, GenericASTTraversal will first look for a method with the same name as the node type (augmented by the prefix "n_"), then will fall back on an optional default method if no more specific method is found.

Of course, GenericASTTraversal can supply many different traversal algorithms. We have found three useful: preorder, postorder, and a pre/postorder combination. (The latter allows methods to be called both on entry to, and exit from, a node.)

For example, say that we want to forbid the mixing of floating-point and integer numbers in our expressions, raising an exception if such mixing occurs:

```
class TypeCheck(GenericASTTraversal):
    def __init__(self, ast):
        GenericASTTraversal.__init__(self, ast)
        self.postorder()

    def n_number(self, node):
        node.exprType = 'number'
    def n_float(self, node):
        node.exprType = 'float'
```

```
    def default(self, node):
        # this handles + and * nodes
        leftType = node.left.exprType
        rightType = node.right.exprType
        if leftType != rightType:
            raise 'Type error.'
        node.exprType = leftType
```

We have found semantic checking code easier to write and understand by taking the (admittedly less efficient) approach of making multiple traversals of the AST — each pass performs a single task.

TypeCheck is invoked from a small glue routine:

```
def semantic(ast):
    TypeCheck(ast)
    #
    #  Any other GenericASTTraversal classes
    #  for semantic checking would be
    #  instantiated here...
    #
    return ast
```

After this phase, we have an AST for an input program that is lexically, syntactically, and semantically correct — but that does nothing. The final phase, evaluation, remedies this.

2.2.4 Evaluation

As already mentioned, the evaluation phase can traverse the AST and implement the input program, either directly through interpretation, or indirectly by emitting some code.

Our expressions, for instance, can be easily interpreted. Below, `int` and `float` are built-in functions which convert strings to integers and floating-point numbers, respectively.

```
class Interpreter(GenericASTTraversal):
    def __init__(self, ast):
        GenericASTTraversal.__init__(self, ast)
        self.postorder()
        print ast.value

    def n_number(self, node):
        node.value = int(node.attr)
    def n_float(self, node):
        node.value = float(node.attr)

    def default(self, node):
        left = node.left.value
        right = node.right.value

        if node.type == '+':
            node.value = left + right
        else:
            node.value = left * right
```

An alternative is to use the GenericASTMatcher class. Here, patterns to look for in the AST are specified in a linearized tree notation, which looks remarkably like grammar rules. GenericASTMatcher determines a way to cover the AST with these patterns, then executes actions associated with the chosen patterns.

For example, the code below also interprets our expressions. The AST covering is shown in Figure 2.5.

```
class AnotherInterpreter(GenericASTMatcher):
    def __init__(self, ast):
        GenericASTMatcher.__init__(self, 'V', ast)
```

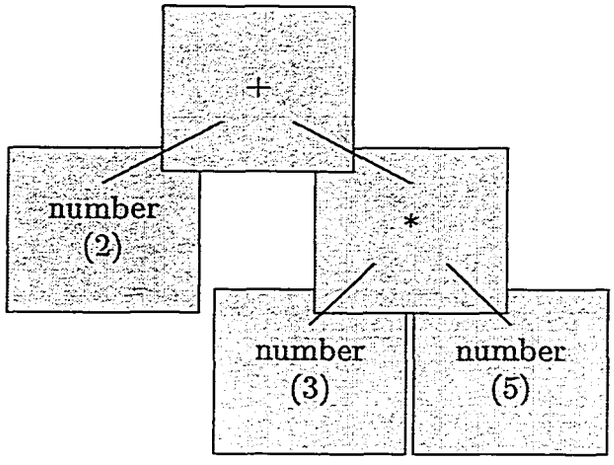


Figure 2.5: Pattern covering of AST.

```
        self.match()
        print ast.value

    def p_number(self, node):
        ' V ::= number '
        node.value = int(node.attr)
    def p_float(self, node):
        ' V ::= float '
        node.value = float(node.attr)

    def p_add(self, node):
        ' V ::= + ( V V ) '
        node.value = node.left.value + node.right.value
    def p_multiply(self, node):
        ' V ::= * ( V V ) '
        node.value = node.left.value * node.right.value
```

The patterns specified may be arbitrarily complex, so long as all the nodes specified in the pattern are adjacent in the AST. To match both + and * nodes, for instance, this method could be added:

```
    def p_addmul(self, node):
```

```
' V ::= + ( V * ( V V ) ) '
node.value = node.left.value + \
             node.right.left.value * \
             node.right.right.value
```

2.3 Inner Workings

2.3.1 Reflection

Extensibility presents some interesting design challenges. The generic classes in the framework, without any modifications made to them, must be able to divine all the information and actions contained in their subclasses, subclasses that didn't exist when the generic classes were created.

Fortunately, an elegant mechanism exists in Python to do just this: reflection. Reflection refers to the ability of a Python program to query and modify itself at run time (this feature is also present in other languages, like Java and Smalltalk).

Consider, for example, our generic scanner class. GenericScanner searches itself and its subclasses at run time for methods that begin with the prefix "t_." These methods are the scanner's actions. The regular expression associated with the actions is specified using a well-known method attribute that can be queried at run time — the method's documentation string.

This wanton abuse of documentation strings can be rationalized. Documentation strings are a method of associating meta-information — comments — with a section of code. Our framework is an extension of that idea. Instead of comments intended

for humans, however, we have meta-information intended for use by our framework. As the number of reflective Python applications grows, it may be worthwhile to add more formal mechanisms to Python to support this task. Coincidentally, this has just happened with the most recent release of Python.

2.3.2 GenericScanner

Internally, GenericScanner works by constructing a single regular expression which is composed of all the smaller regular expressions it has found in the action methods' documentation strings. Each component regular expression is mapped to its action using Python's symbolic group facility.

Unfortunately, there is a small snag. Python follows the Perl semantics [96] for regular expressions rather than the POSIX semantics [51], which means it follows the "first then longest" rule — the leftmost part of a regular expression that matches is always taken, rather than using the longest match. In the above example, if GenericScanner were to order the regular expression so that "`\d+`" appeared before "`\d+\.\d+`", then the input 123.45 would match as the number 123, rather than the floating-point number 123.45. To work around this, GenericScanner makes two guarantees:

1. A subclass' patterns will be matched before any in its parent classes.

2. The default pattern for a subclass, if any, will be matched only after all other patterns in the subclass have been tried.

One obvious change to GenericScanner is to automate the building of the list of tokens — each "t_" method could return a list of tokens which would be appended to the scanner's list of tokens. The reason this is not done is because it would limit potential applications of GenericScanner. For example, in one compiler we used a subclass of GenericScanner as a preprocessor which returned a string; another scanner class then broke that string into a list of tokens.

2.3.3 GenericParser

GenericParser is actually more powerful than was alluded to in Section 2.2.2. At the cost of greater coupling between methods, actions for similar rules may be combined together rather than having to duplicate code — our original version of ExprParser is shown below. For clarity, we use Python's triple-quoted strings which allow a string to span multiple lines.

```
class ExprParser(GenericParser):
    def __init__(self, start='expr'):
        GenericParser.__init__(self, start)

    def p_expr_term(self, args):
        '''
            expr ::= expr + term
            term ::= term * factor
        '''
        return AST(type=args[1], left=args[0], right=args[2])

    def p_expr_term_2(self, args):
        '''
            expr ::= term
            term ::= factor
        '''
```

```
        return args[0]

    def p_factor(self, args):
        '''
            factor ::= number
            factor ::= float
        '''
        return AST(type=args[0])
```

Taking this to extremes, if a user is *only* interested in parsing and doesn't require an AST, ExprParser could be written:

```
class ExprParser(GenericParser):
    def __init__(self, start='expr'):
        GenericParser.__init__(self, start)

    def p_rules(self, args):
        '''
            expr ::= expr + term
            expr ::= term
            term ::= term * factor
            term ::= factor
            factor ::= number
            factor ::= float
        '''
```

In theory, GenericParser could use any parsing algorithm for its engine. However, we chose the Earley parsing algorithm [30, 31] which has several nice properties for this application [46]:

1. It is one of the most general algorithms known; it can parse all context-free grammars whereas the more popular LL and LR techniques cannot. This is important for easy extensibility; a user should ideally be able to subclass a parser without worrying about properties of the resulting grammar.

2. It generates all its information at run time, rather than having to precompute sets and tables. Since the grammar rules aren't known until run time, this is just as well!

Unlike most other parsing algorithms, Earley's method parses ambiguous grammars. Ambiguity can present a problem since it is not clear which actions should be invoked. When this occurs, GenericParser calls GenericParser.resolve to choose between the possible input derivations. Users may override this method to implement their own behaviour.

To accommodate a variety of possible parsing algorithms (including the one we used), GenericParser only makes one guarantee with respect to when the rules' actions are executed. A rule's action is executed only after all the attributes on the rule's right-hand side are fully computed. This condition is sufficient to allow the correct construction of ASTs.

There are other general parsing algorithms besides Earley's algorithm. In particular, generalized LR (GLR) parsing [89] would be another candidate for use in SPARK. However, we used Earley parsing in preference to GLR parsing for the following technical and non-technical reasons:

1. Earley parsing has better worst-case performance, in terms of computational complexity.

2. GLR parsing will not work with all context-free grammars without modification [89], whereas Earley parsing does.

3. GLR parsers typically require a "compiler build" phase, which we wanted to avoid in order to enhance SPARK's ease of use. (Although this can be done lazily at parse time [47].)

4. Having implemented and worked with both types of parser, we find Earley parsers simpler to implement and reason about than GLR parsers.

2.3.4 GenericASTBuilder

GenericASTBuilder works by hijacking GenericParser's operation. The action associated with each "p_" method is re-routed to an internal GenericASTBuilder method which performs tree construction.

Experience with SPARK has shown that GenericASTBuilder is an excellent labor-saving device. For example, we used it in our project that decompiled bytecode. However, it can sometimes be difficult to specify exactly how to transform a concrete syntax tree into an AST. In practice, one often ends up using an AST design which is tolerable but not ideal, simply because it is easier to construct with GenericAST-Builder. A means of improving on this situation is the topic of future work.

2.3.5 GenericASTTraversal

GenericASTTraversal is the least unusual of the generic classes. It could be argued that its use of reflection is superfluous, and the same functionality could be achieved by having its subclasses provide a method for every type of AST node; these methods could call a default method themselves if necessary.

The problems with this non-reflective approach are threefold. First, it introduces a maintenance issue: any additional node types added to the AST require all GenericASTTraversal's subclasses to be changed. Second, it forces the user to do more work, as methods for all node types must be supplied; our experience, especially for semantic checking, is that only a small set of node types will be of interest for a given subclass. Third, some node types may not map nicely into Python method names — we prefer to use node types that reflect the little language's syntax, like +, and it isn't possible to have methods named "n_+."[7] This latter point is where it is useful to have GenericASTTraversal reflectively probe a subclass and automatically invoke the default method.

2.3.6 GenericASTMatcher

GenericASTMatcher currently operates using a Graham/Glanville code generator [42]. The input AST is linearized using a preorder tree traversal, retaining structural information by insertion of balanced parentheses. For example, the AST in Figure 2.2 would be represented as

```
+ ( number * ( number number ) )
```

GenericParser is then used to parse the linearized AST using the grammar (i.e., the patterns specified in the "p_" methods) supplied by the user.

We note that Earley parsing has been applied to Graham/Glanville code generation before. Christopher et al. [26] concluded that Earley's algorithm solved all of

[7]Not directly, anyway...

the extant problems with the unadulterated Graham/Glanville technique, but with an enormous execution time compared to more naïve code generation algorithms.

One future possibility would be to exchange the Graham/Glanville engine for a more sophisticated one, using (for example) bottom-up rewrite systems [39]. Another interesting idea would be to allow more general patterns, akin to those in the XML Path Language [101].

2.3.7 Design Patterns

Although developed independently, the use of reflection in our framework is arguably a specialization of the Reflection pattern [23]. We speculate that there are many other design patterns where reflection can be exploited. To illustrate, GenericASTTraversal wound up somewhere between the Default Visitor [74] and Reflection patterns, although it was originally inspired by the Visitor pattern [40].

Two other design patterns can be applied to our framework too. First, the entire framework could be organized explicitly as a Pipes and Filters pattern [23]. Second, the generic classes could support interchangeable algorithms via the Strategy pattern [40]; parsing algorithms, in particular, vary widely in their characteristics, so allowing different algorithms could be a boon to an advanced user.

2.3.8 Class Structure

Figure 2.6 shows the class structure of the user-visible classes in SPARK, along with their key methods. Due to the generality and flexibility of Earley's algorithm, many

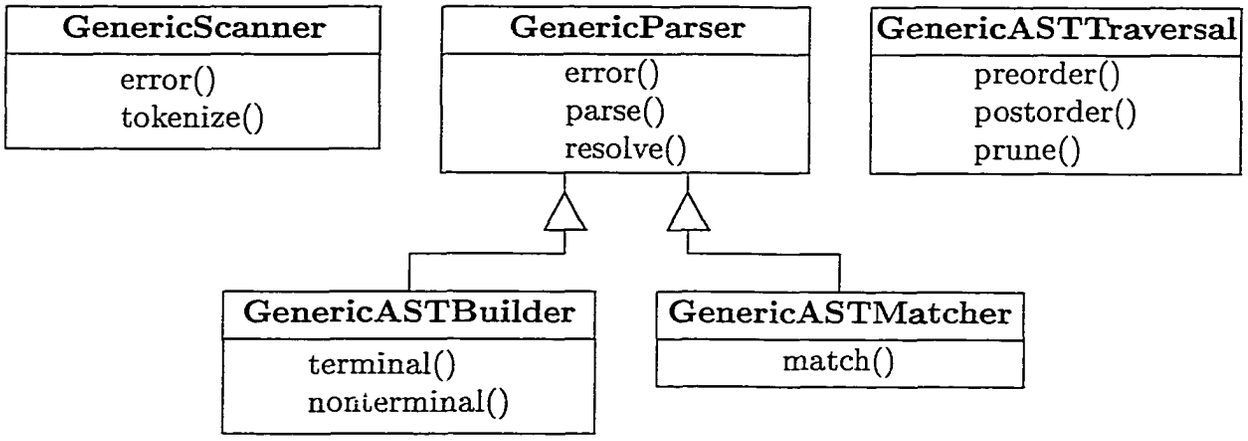


Figure 2.6: SPARK's class structure.

classes have grown to depend on GenericParser. Our experience was that this dependence tended to amplify the problems with Earley's algorithm, which we discuss in the next chapter.

2.4 Summary

SPARK is a framework we have developed to build compilers in Python. It uses reflection and design patterns to produce compilers which can be easily extended using traditional object-oriented methods. Many of the objects supplied by SPARK rely internally on Earley's parsing algorithm.

Chapter 3

Languages, Grammars, and Earley Parsing

In this chapter we formalize some ideas about languages and grammars, and present Earley's parsing algorithm. Over half of the classes supplied by SPARK rely on Earley's algorithm.

3.1 Languages and Grammars

A language is a set of strings over a finite alphabet [70]; in the context of parsing, we may intuitively think of this alphabet as the set of tokens that a scanner may return. If we denote the alphabet as Σ, then various finite and infinite languages may be formed by taking subsets of Σ^*, the Kleene closure of Σ. For example, if $\Sigma = \{a, b\}$, then some languages over Σ would be $\{b\}$, $\{a, b\}$, $\{b, ab, aab, aaab\}$, and

$\{a, aa, aaa, \ldots\}$. The empty string is denoted by ϵ, and the length of a string α is written $|\alpha|$.

Different categories of languages exist, and one prevalent classification scheme is the Chomsky hierarchy [25, 48]. In particular, we are interested in Type 2 languages: context-free languages, or CFLs. To define what a CFL is, though, we must first define what a grammar is.

Any language may be described by an infinite number of grammars. By way of analogy, one may think of a language as being like the game of chess. The rules of chess may be described an infinite number of ways — rules in English, rules in French, rules in Russian, and so on — but they all describe the same game. And indeed, a grammar primarily consists of a set of rules. Given a grammar G, the language generated by that grammar is denoted $L(G)$. A CFL is simply any language generated by a context-free grammar (CFG).

Formally, a CFG G is a quadruple (N, Σ, R, S), where

N is a set of nonterminal symbols,

Σ is a set of terminal symbols, $\Sigma \cap N = \emptyset$,

R is a set of grammar rules, $R \subseteq N \times (N \cup \Sigma)^*$, and

$S \in N$ is a start symbol.

N, Σ, and R are all finite sets [48, 62]. A grammar rule (A, α) is typically written $A \rightarrow \alpha$ (although grammar rules in SPARK use "::=" rather than "$\rightarrow$" for pragmatic reasons).

CFGs are usually written informally as a set of grammar rules. Unless a start symbol is explicitly given, the first rule is conventionally the "start rule," meaning

that the nonterminal on its left-hand side is the grammar's start symbol. Also, when several grammar rules have a common left-hand side, such as $A \rightarrow \alpha$ and $A \rightarrow \beta$, they may be written using the shorthand form $A \rightarrow \alpha \mid \beta$.

We will be assuming the use of augmented grammars in the remainder of our work. An augmented grammar is a simple extension of a CFG which adds a new start symbol S', $S' \notin N$, and a new rule $S' \rightarrow S$.

Standard notation [3] is used when discussing grammars in the abstract. Briefly, lowercase letters represent terminal symbols, uppercase letters early in the alphabet are nonterminal symbols, and uppercase letters late in the alphabet can be either terminals or nonterminals. Greek letters denote strings of zero or more terminal and nonterminal symbols. More concrete instances of grammars extend these conventions: punctuation characters (e.g., parentheses) are taken to be terminal symbols, and meaningful words are used for both terminal and nonterminal symbols. We have enclosed terminal symbols in quotes where the type of symbol is not immediately evident from the context.

The application of grammar rules is captured in the notion of derivation. Given a grammar rule $A \rightarrow \alpha$, we may replace the occurrence of A with α in the string $\beta A \gamma$. When this happens, $\beta A \gamma$ is said to derive $\beta \alpha \gamma$ in one step, written $\beta A \gamma \Rightarrow \beta \alpha \gamma$. If the leftmost nonterminal is replaced, then this is a leftmost derivation, as in $AAA \Rightarrow_L \alpha AA$; one may have a rightmost derivation as well. A sequence of derivation steps can be applied to a string of symbols, which is summarized by the notation $\Rightarrow^*$ (derives in zero or more steps) and $\Rightarrow^+$ (derives in one or more steps). In a more general sense, the derivation of an input refers to the entire sequence of derivation steps taken to

$a, b, \ldots$	terminal symbols
$A, B, \ldots$	nonterminal symbols
$\ldots, X, Y, Z$	terminal or nonterminal symbol
$\alpha, \beta, \ldots$	zero or more terminal and nonterminal symbols
ϵ	empty string
$\|\alpha\|$	length
$A \Rightarrow \alpha$	derives (in one step)
$A \Rightarrow^* \alpha$	derives in zero or more steps
$A \Rightarrow^+ \alpha$	derives in one or more steps
$A \Rightarrow_L \alpha$	leftmost derivation
$A \Rightarrow_R \alpha$	rightmost derivation

Table 3.1: Notation summary.

derive an input w in $L(G)$ from the start symbol S.

Finally, a CFG G is ambiguous if there are two or more distinct leftmost derivations for some string w in $L(G)$ [70].[8] For example, the grammar $S \rightarrow SS \mid x$ is ambiguous because the leftmost derivations

$$S \Rightarrow_L SS \Rightarrow_L SSS \Rightarrow_L xSS \Rightarrow_L xxS \Rightarrow_L xxx \text{ and}$$

$$S \Rightarrow_L SS \Rightarrow_L xS \Rightarrow_L xSS \Rightarrow_L xxS \Rightarrow_L xxx$$

exist for the string xxx.

Much of the notation presented here is summarized in Table 3.1.

3.2 Earley Parsing

Parsing algorithms work backwards in a way. Given a CFG G and an input $w \in \Sigma^*$, a parser's job is to answer two questions:

[8] Or equivalently, two or more distinct rightmost derivations [3].

1. Is $w \in L(G)$? This decision task is referred to as recognition.

2. If $w \in L(G)$, then what sequences of derivation steps were used to derive w?

Earley's parsing algorithm is one of a family of general parsing algorithms which can parse using any context-free grammar, including ambiguous grammars.[9]

In the past, general parsers have not been widely used in compilers due to efficiency concerns: all other things being equal, the more general parsing algorithms tend to be slower due to extra overhead [46]. However, this is becoming less of a concern with increases in processor speed and memory capacity. There are now a number of parser generators and other tools using these general algorithms [7, 84, 92, 95], as well as approaches to making the algorithms faster [9, 11, 44, 71].

General parsing algorithms have some advantages. No "massaging" of a context-free grammar is required to make it acceptable for use in a general parser, as is required by more efficient algorithms like the LALR(1) algorithm used in Yacc [53, 68]. Using a general parser thus reduces programmer development time, eliminates a source of potential bugs, and lets the grammar reflect the input language rather than the limitations of a compiler tool.

General algorithms also work for ambiguous grammars, unlike their more efficient counterparts. Some programming language grammars, such as those for Pascal, C, and C++, contain areas of ambiguity. For some tasks ambiguous grammars may be deliberately constructed, such as a grammar which describes multiple dialects of a language for use in software reengineering [91], or a grammar for Graham/Glanville

[9] In the remainder of this thesis, context-free grammars should be assumed unless otherwise stated.

code generation [42].

The primary objection to general parsing algorithms, then, is not one of functionality. The problems with general parsing algorithms are twofold:

1. Speed. General parsing algorithms are not as efficient as more specialized algorithms.

2. Lack of programmer control. General parsing algorithms must, in general, read and verify their entire input first [46]; this behaviour is contrary to that of specialized algorithms. With a general parsing algorithm, semantic actions associated with grammar rules may not be executed until after the input is recognized, eliminating a favorite compiler implementation trick: altering the operation of the scanner and parser on the fly. We refer to this as the "delayed action problem."

Our research has addressed these problems with respect to Earley's algorithm. Before we can elaborate on our solutions to these problems, however, a description of Earley's algorithm is necessary.

Earley's algorithm works by building a sequence of sets, sometimes called Earley sets. Given an input $a_1 a_2 \ldots a_n$, the parser builds $n + 1$ sets: one initial Earley set S_0, and one Earley set S_i for each input symbol a_i. An Earley set contains Earley items, which consist of three parts: a grammar rule; a position in the grammar rule's right-hand side indicating how much of that rule has been seen, denoted by a dot ($\bullet$); a pointer back to some previous "parent" Earley set. For instance, the Earley item $[A \rightarrow a \bullet Bb, 12]$ indicates that the parser has seen the first symbol of the grammar

rule $A \rightarrow aBb$, and points back to Earley set S_{12}. We use the term "core Earley item" to refer to an Earley item less its parent pointer: $A \rightarrow a \bullet Bb$ in the above example.

The three steps below are applied to Earley items in S_i until no more can be added; this constructs S_i and primes S_{i+1}.

SCANNER. If $[A \rightarrow \cdots \bullet b \cdots, j]$ is in S_i and $a_{i+1} = b$, add $[A \rightarrow \cdots b \bullet \cdots, j]$ to S_{i+1}.

PREDICTOR. If $[A \rightarrow \cdots \bullet B \cdots, j]$ is in S_i, add $[B \rightarrow \bullet\alpha, i]$ to S_i for all rules $B \rightarrow \alpha$ in G.

COMPLETER. If a "final" Earley item $[A \rightarrow \cdots \bullet, j]$ is in S_i, add $[B \rightarrow \cdots A \bullet \cdots, k]$ to S_i for all Earley items $[B \rightarrow \cdots \bullet A \cdots, k]$ in S_j.

An Earley item is added to a Earley set only if it is not already present in the Earley set. The initial set S_0 holds the single Earley item $[S' \rightarrow \bullet S, 0]$ prior to Earley set construction, and the final Earley set must contain $[S' \rightarrow S \bullet, 0]$ upon completion in order for the input string to be accepted. Figure 3.1 shows an example of Earley parser operation.

The Earley algorithm may employ lookahead to reduce the number of Earley items in each Earley set, but we have found the version of the algorithm without lookahead suitable for our purposes. We also restrict our attention to input recognition rather than parsing proper. Construction of parse trees in Earley's algorithm is done after recognition is complete, based on information retained by the recognizer, so this division may be done without loss of generality.

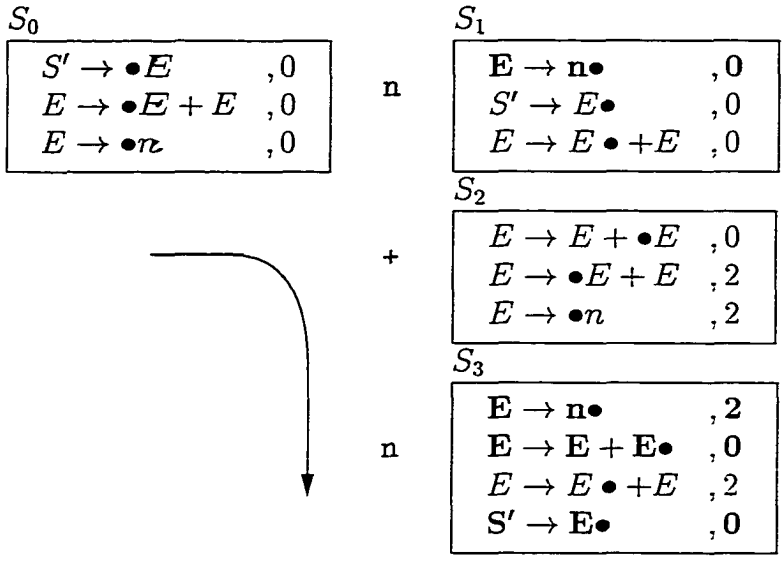


Figure 3.1: Earley sets for the ambiguous grammar $E \rightarrow E + E \mid n$ and the input n + n. Emboldened items are ones which correspond to the input's derivation.

3.3 Summary

Reading formal definitions is only slightly more exciting than watching paint peel. That notwithstanding, general parsing algorithms, and Earley's algorithm in particular, suffer from two problems: the speed problem and the delayed action problem.

Chapter 4

Directly-Executable Earley Parsing[10]

SPARK is somewhat unusual in that it uses Earley's algorithm, and can operate using arbitrary context-free grammars. Most parsers in use today are only capable of handling subsets of context-free grammars: LL, LR, LALR. And with good reason – efficient linear-time algorithms for parsing these subsets are known.

For LR parsers, dramatic speed improvements have been obtained by producing hard-coded, or directly-executable parsers [13, 20, 50, 76, 77]. These directly-executable LR parsers implement the parser for a given grammar as a specialized program, rather than using a typical table-driven approach. Unfortunately, these techniques do not directly apply to general parsing algorithms. LR parsers, being specialized algorithms, can rely on the grammar being unambiguous and the exis-

[10] An earlier version of this work appeared in [10].

tence of a single stack within the parser, for instance. These same helpful simplifying assumptions cannot be made in a general parser.

In this chapter we extend directly-executable techniques for use in Earley's general parsing algorithm, to produce and evaluate what we believe is the first directly-executable Earley parser. The speed of our parsers is shown to be comparable to deterministic parsers produced by Bison, an implementation of Yacc.

4.1 DEEP: a Directly-Executable Earley Parser

4.1.1 Observations

There are a number of observations about Earley's algorithm which can be made. By themselves, they seem obvious, yet taken together they shape the construction of our directly-executable parser.

Observation 1. Additions are only ever made to the current and next Earley sets,[11] S_i and S_{i+1}.

Observation 2. The COMPLETER does not recursively look back through Earley sets; it only considers a single parent Earley set, S_j.

Observation 3. The SCANNER looks at each Earley item exactly once, and this is the only place where the dot may be moved due to a terminal symbol.

[11]To avoid confusion later, we use the unfortunately awkward terms "Earley set" and "Earley item" throughout this chapter.

Observation 4. Movement of the dot due to a nonterminal symbol only takes place when the COMPLETER looks back at a parent Earley set.

Observation 5. Movement of the dot over a symbol, be it terminal or nonterminal, is a common, unifying operation on Earley items.

Observation 6. Earley items added by PREDICTOR all have the same parent, i.

Observation 5, especially, sparked the notion that it was possible to have a directly-executable Earley parser. But at what level of granularity?

4.1.2 Basic Organization

The contents of an Earley set depend on the input and are not known until run time; we cannot realistically precompute one piece of directly-executable code for every possible Earley set. We can assume, however, that the grammar is known prior to run time, so we begin by considering how to generate one piece of directly-executable code per Earley item.

Even within an Earley item, not everything can be precomputed. In particular, the value of the parent pointer will not be known ahead of time. Given two otherwise identical Earley items $[A \rightarrow \alpha \bullet \beta, j]$ and $[A \rightarrow \alpha \bullet \beta, k]$, the invariant part is the core Earley item. The code for a directly-executable Earley item, then, is actually code for the core Earley item; the parent pointer is maintained as data. A directly-executable Earley item may be represented as the tuple

$$(\textit{code for } A \rightarrow \alpha \bullet \beta, \textit{ parent})$$

the code for which is structured as shown in Figure 4.1. Each terminal and nonterminal symbol is represented by a distinct number; the variable sym can thus contain either type of symbol. Movement over a terminal symbol is a straightforward implementation of the SCANNER step, but movement over a nonterminal is complicated by the fact that there are two actions that may take place upon reaching an Earley item $[A \rightarrow \cdots \bullet B \cdots, j]$, depending on the context:

1. If encountered when processing the current Earley set, the PREDICTOR step should be run.

2. If encountered in a parent Earley set (i.e., the COMPLETER step is running) then movement over the nonterminal may occur according to Observation 4. In this case, sym cannot be a terminal symbol, so the predicate `ISTERMINAL()` is used to distinguish these two cases.

The code for final Earley items calls the code implementing the parent Earley set, after replacing sym with the nonterminal symbol to move the dot over. By Observation 2, no stack is required as the call depth is limited to one. Again, this should only be executed if the current set is being processed, necessitating the `ISTERMINAL()`.

4.1.3 Earley Set Representation

An Earley set in the directly-executable representation is conceptually an ordered sequence of *(code, parent)* tuples followed by one of the special tuples:

(*end of current Earley set code*, -1)
(*end of parent Earley set code*, -1)

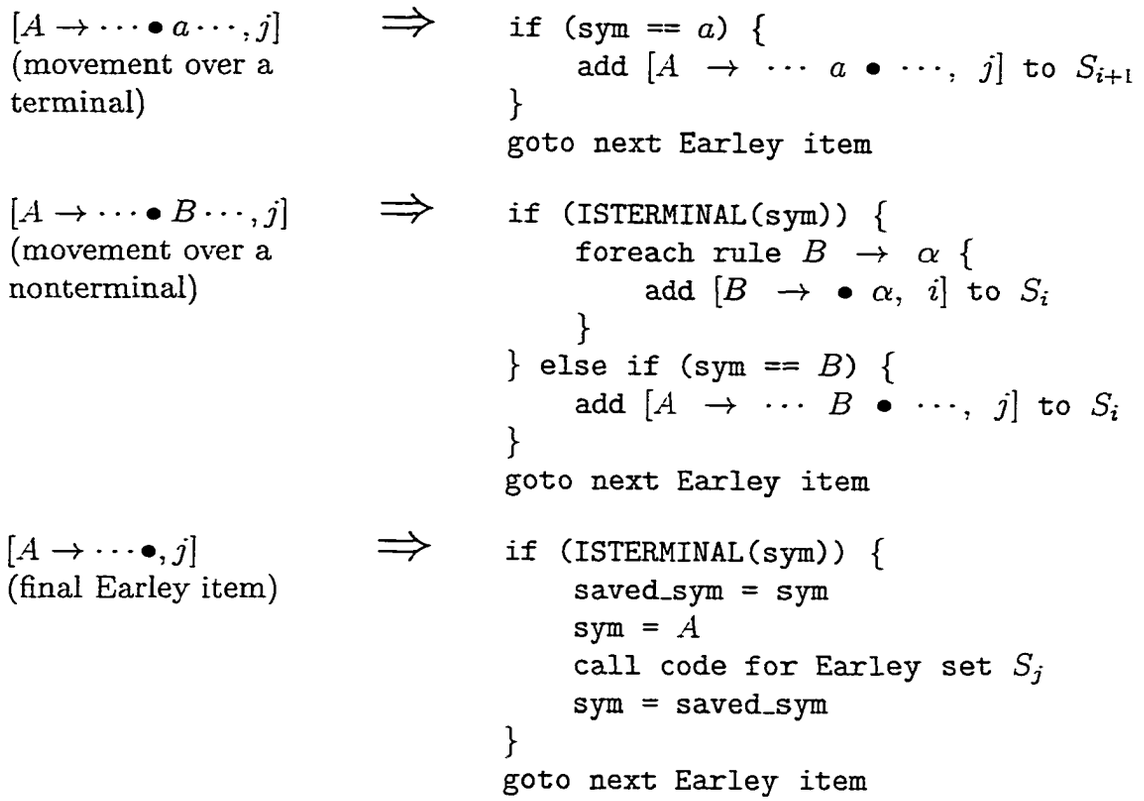


Figure 4.1: Pseudocode for directly-executable Earley items.

The code at the end of a parent Earley set simply contains a `return` to match the `call` made by a final Earley item. Reaching the end of the current Earley set is complicated by bookkeeping operations to prepare for processing the next Earley set. The parent pointer is irrelevant for either of these.

In practice, our DEEP implementation splits the tuples. DEEP's Earley sets are in two parts: a list of addresses of code, and a corresponding list of parent pointers. The two parts are separated in memory by a constant amount, so that knowing the memory location of one half of a tuple makes it a trivial calculation to find the other half.

Having a list of code addresses for an Earley set makes it possible to implement the action "`goto next Earley item`" with direct threaded code [16]. With threaded code, each directly-executable Earley item jumps directly to the beginning of the code for the next Earley item, rather than first returning to a dispatching loop or incurring function call overhead. Not only does threaded code proffer speed advantages by avoiding the call/return mechanism [55], but it can work better with branch prediction hardware on modern CPUs [35]. We implement this in a reasonably portable fashion using the first-class labels in GNU C.[12]

How is an Earley item added to an Earley set in this representation? First, recall that an Earley item is only placed in an Earley set if it does not already appear there. We will use the term "appending" to denote an Earley item being placed into an Earley set; "adding" is the process of determining if an Earley item should be appended to an Earley set. (We discuss adding more in Section 4.1.4.) Using

[12]This is an extension to ANSI C.

this terminology, appending an Earley item to an Earley set is done by dynamically generating threaded code. We also dynamically modify the threaded code to exchange one piece of code for another in two instances:

1. When the current Earley set is fully processed, "end of current Earley set" must be exchanged for "end of parent Earley set."

2. Observation 3 implies that once the SCANNER has looked at an Earley item, any code looking for terminal symbols is superfluous. By modifying the threaded code, DEEP skips the superfluous code on subsequent invocations.

Appending leaves DEEP with a thorny problem of memory management. Observation 1 says that Earley items — threaded code plus separate parent pointers — can be appended to one of two Earley sets. We also maintain an array whose i^{th} entry is a pointer to the code for Earley set S_i, for implementation of `call`, giving us a total of five distinct, dynamically-growing memory areas.

Instead of complex, high-overhead memory management, we have the operating system assist us by memory-mapping oversized areas of virtual memory. This is an efficient operation because the operating system will not allocate the virtual memory pages until they are used. We can also protect key memory pages so that an exception is caused if DEEP should exceed its allocated memory, absolving us from performing bounds checking when appending. This arrangement is shown in Figure 4.2, which also demonstrates how the current and next Earley sets alternate between memory areas.

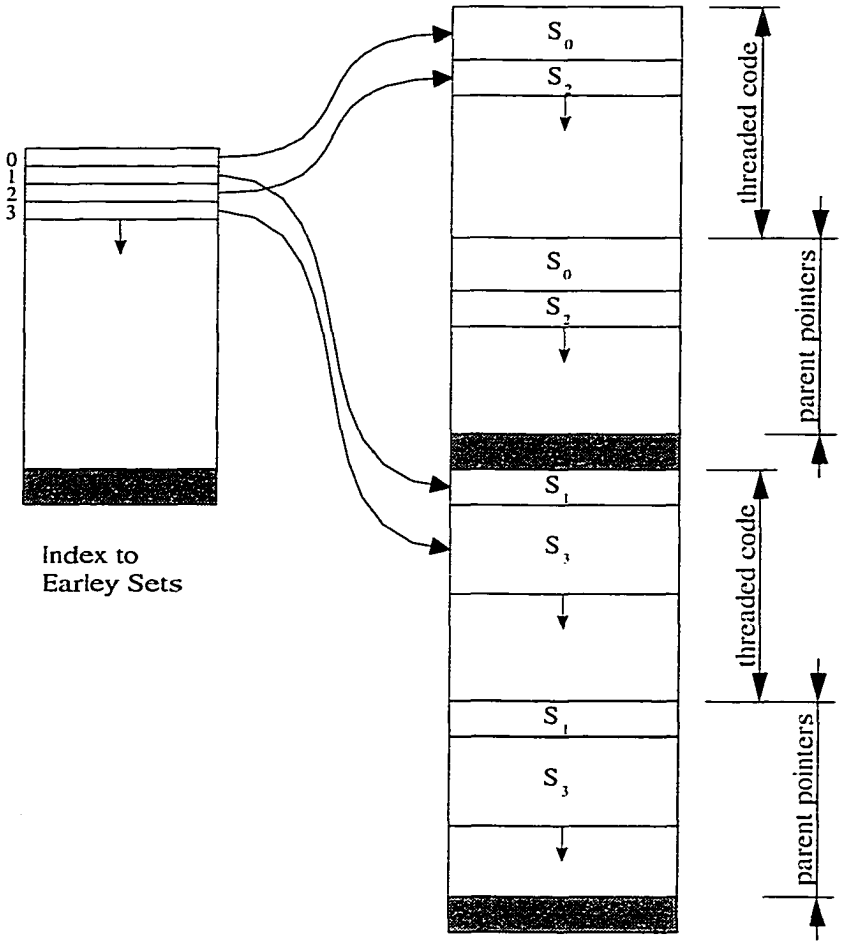


Figure 4.2: Memory layout for DEEP. S_2 and S_3 are the current and next Earley sets, respectively; the shaded areas are protected memory pages.

4.1.4 Adding Earley Items

As mentioned, adding an Earley item to an Earley set entails checking to ensure that it is not already present. Earley suggested using an array indexed by the parent pointer, each entry of which would be a list of Earley items to search [31]. Instead, we note that core Earley items may be enumerated, yielding finite, relatively small numbers.[13] A core Earley item's number may be used to index into a bitmap to quickly check the presence or absence of *any* Earley item with that core.

When two or more Earley items exist with the same core, but different parent pointers, we construct a radix tree [59] for that core Earley item — a binary tree whose branches are either zero or one — which keeps track of which parent pointers have been seen. Radix trees have two nice properties:

1. Insertion and lookup, the only operations required, are simple.

2. The time complexity of radix tree operations during execution is $\log i$, where i is the number of tokens read, thus growing slowly even with large inputs.

To avoid building a radix tree for a core Earley item until absolutely necessary, we cache the first parent pointer until we encounter a second Earley item with the same core. An example of adding Earley items is given in Figure 4.3.

4.1.5 Sets Containing Items which are Sets Containing Items

Earley parsing has a deep relationship with its contemporary, LR parsing [30]. Here we look at LR(0) parsers — LR parsers with no lookahead. As with all LR parsers,

[13] To be precise, the number of core Earley items is $\sum_{A \to \alpha \in G} (|\alpha| + 1)$.

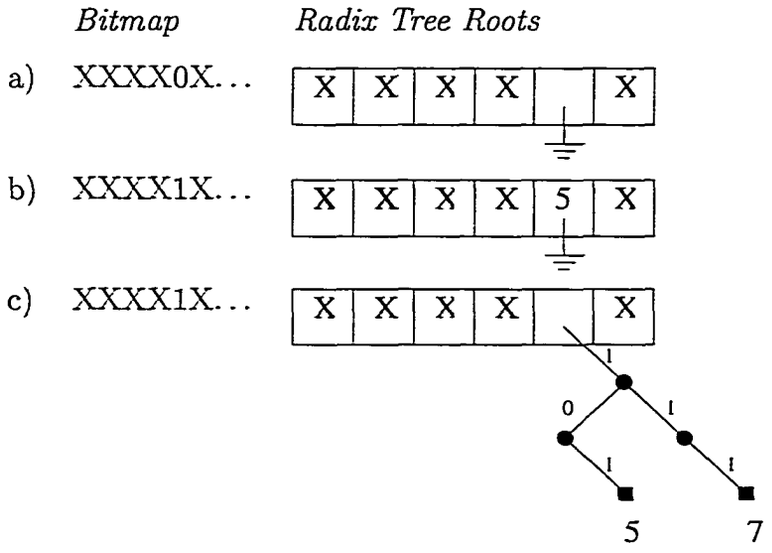


Figure 4.3: Adding Earley items: (a) initial state; (b) after appending Earley item #4, parent S_5; (c) after appending Earley item #4, parent S_7. In the radix tree, a circle (•) indicates an absent entry, and a square (■) indicates an existing entry. "X" denotes a "don't care" entry.

an LR(0) parser's recognition is driven by a deterministic finite automaton (DFA) which is used to decide when the right-hand side of a grammar rule has been seen. A DFA state corresponds to a set of LR(0) items, and an LR(0) item is exactly the same as a core Earley item.

How is an LR(0) DFA constructed? Consider a *non*deterministic finite automaton (NFA) for LR(0) parsing, where each NFA state contains exactly one LR(0) item. A transition is made from $[A \to \cdots \bullet X \cdots]$ to $[A \to \cdots X \bullet \cdots]$ on the symbol X, and from $[A \to \cdots \bullet B \cdots]$ to $[B \to \bullet\alpha]$ on ϵ; the start state is $[S' \to \bullet S]$. This NFA may then be converted into the LR(0) DFA using standard methods [3].[14]

The conversion from NFA to DFA yields, as mentioned, DFA states which are sets of LR(0) items. Within each LR(0) set, the items may be logically divided into kernel items (the initial item and items where the dot is not at the beginning) and nonkernel items (all other items) [3]. We explicitly represent this logical division by splitting each LR(0) DFA state into two states (at most), leaving us with an almost-deterministic automaton, the LR(0) $\overline{\text{DFA}}$.

Consider the expression grammar G_E:

$$\begin{array}{lll} S' \to E & T \to T * F & F \to n \\ E \to E + T & T \to T/F & F \to -F \\ E \to E - T & T \to F & F \to +F \\ E \to T & & F \to (E) \end{array}$$

Figure 4.4 shows a partial LR(0) $\overline{\text{DFA}}$ for G_E. In the original LR(0) DFA, states 0 and 1, 2 and 10, 18 and 19, 24 and 25 were merged together.

Returning to Earley parsing, the core Earley items in an Earley set may be represented using one or more states in an LR(0) DFA [71]. The problem with doing so is

[14] Of course, there are much more efficient ways of constructing the LR(0) DFA...

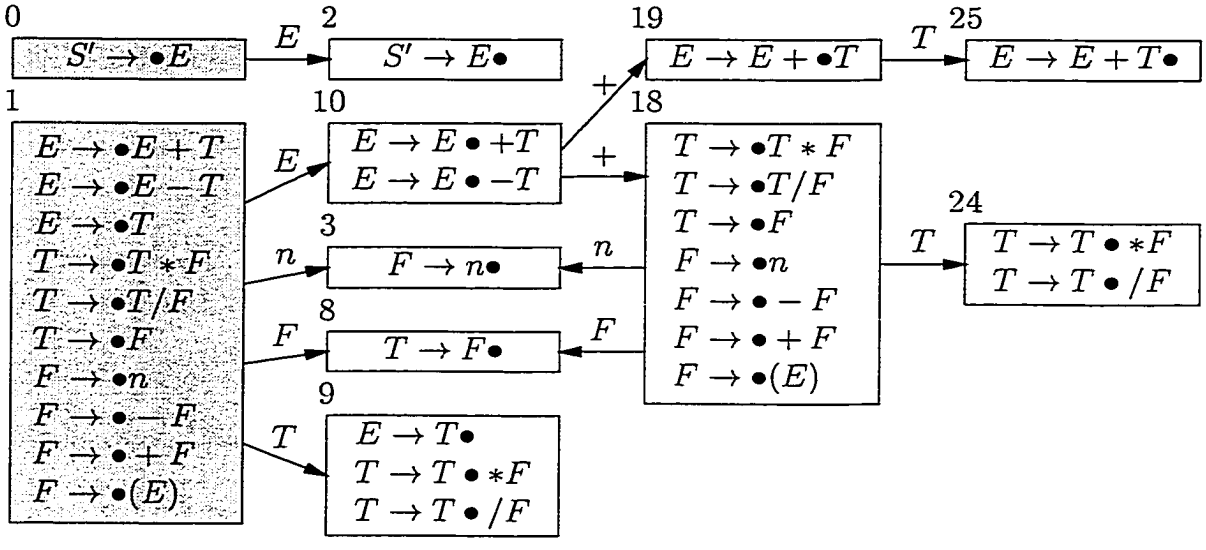


Figure 4.4: Partial LR(0) $\overline{\text{DFA}}$ for G_E. Shading denotes start states.

that keeping track of which parent pointers and LR(0) items belong together results in a complex, inelegant implementation: tuples containing lists inside lists. However, we realized as a result of Observation 6 that the PREDICTOR really just corresponds to making a transition to a "nonkernel" state in the LR(0) $\overline{\text{DFA}}$. Pursuing this idea, we represent Earley items in DEEP as the tuples

(*code for LR(0)* $\overline{\text{DFA}}$ *state, parent*)

Figure 4.5 shows some Earley sets for G_E; Figure 4.6 shows the same Earley sets recoded using the LR(0) $\overline{\text{DFA}}$ states.

Through this new representation, we gain most of the efficiency of using an LR(0) DFA as the basis of an Earley parser, but with the benefit of a particularly simple representation and implementation. The prior discussion in this section regarding DEEP still holds, except the directly-executable code makes transitions from one

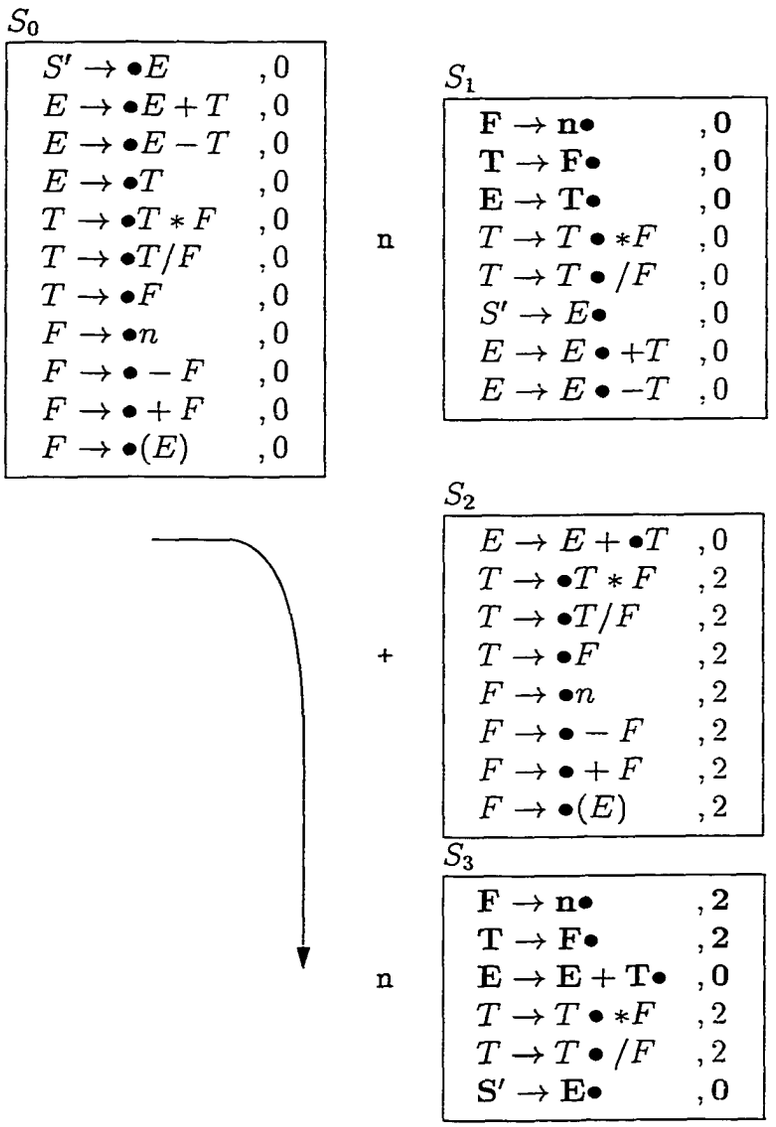


Figure 4.5: Earley sets for the expression grammar G_E, parsing the input `n + n`.

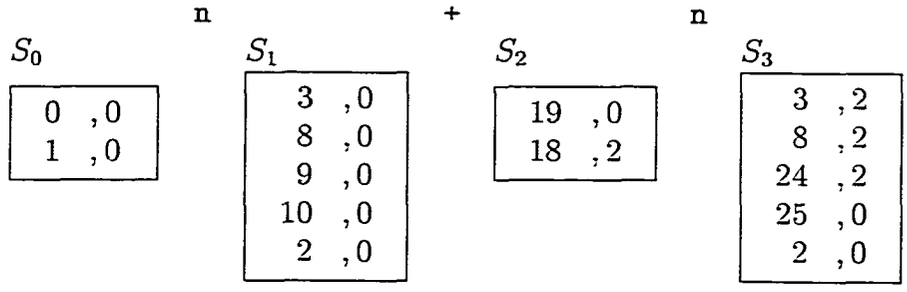


Figure 4.6: Earley sets for the expression grammar G_E, parsing the input n + n, encoded using LR(0) $\overline{\text{DFA}}$ states.

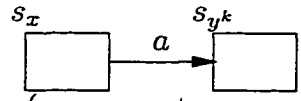
LR(0) $\overline{\text{DFA}}$ state to another instead of from one Earley item to another.

Figure 4.7 restates the pseudocode from Figure 4.1, using $\overline{\text{DFA}}$ states. There are three major changes to note. First, a state may have either one or two transitions for a given symbol as a result of splitting DFA states. Second, the explicit PREDICTOR code is gone; Earley items that would have been added by the PREDICTOR are now clustered together in the nonkernel $\overline{\text{DFA}}$ states. Third, the "goto" is no longer shown in the pseudocode, but is still present in the final code. A $\overline{\text{DFA}}$ state may have a combination of the pseudocode shown — movement over a terminal as well as final Earley items, for instance — and the "goto" must always fall at the end of a state's code. We give an example to show how the pseudocode fits together in the next section.

4.1.6 Implementation Rumination

DEEP parsers are generated by a 650-line Python script. This script takes a grammar as input, produces the LR(0) $\overline{\text{DFA}}$ states, and emits the states' corresponding C code.

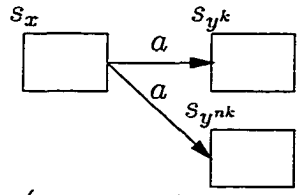
For G_E, the LR(0) $\overline{\text{DFA}}$ consists of twenty-five states. What follows is the anno-



(movement over a terminal)

$\Rightarrow$

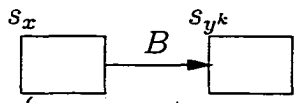
```
if (sym == a) {
    add (s_{y^k}, j) to S_{i+1}
}
```



(movement over a terminal)

$\Rightarrow$

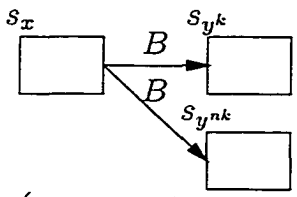
```
if (sym == a) {
    add (s_{y^k}, j) to S_{i+1}
    add (s_{y^nk}, i+1) to S_{i+1}
}
```



(movement over a nonterminal)

$\Rightarrow$

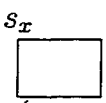
```
if (sym == B) {
    add (s_{y^k}, j) to S_i
}
```



(movement over a nonterminal)

$\Rightarrow$

```
if (sym == B) {
    add (s_{y^k}, j) to S_i
    add (s_{y^nk}, i) to S_i
}
```



(s_x contains final Earley items $[A_1 \rightarrow \cdots \bullet, j_1]$, $[A_2 \rightarrow \cdots \bullet, j_2]$, ...)

$\Rightarrow$

```
if (ISTERMINAL(sym)) {
    saved_sym = sym
    sym = A_1
    call code for Earley set S_{j_1}
    sym = A_2
    call code for Earley set S_{j_2}
    ...
    sym = saved_sym
}
```

Figure 4.7: Pseudocode for directly-executable $\overline{\text{DFA}}$ states. The parent pointer of s_x is j; kernel and nonkernel states are denoted k and nk respectively.

tated C code for five of these states.

```
state0:
    if (ISTERMINAL(sym)) {
        switch (sym) {
        }
        REPLACEME(&&state0nt);
        DISPATCH();
    }
state0nt:
    switch (sym) {
      case NT_E:
        GEN_CUR(&&state2, 2, MYPARENT());
        break;
    }
    DISPATCH();
```

Each state has two main labels associated with it. The first is the entry point initially used when processing the current Earley set; the second is used when the symbol in sym is a nonterminal. We will call this the terminal code and nonterminal code, respectively. The REPLACEME macro dynamically modifies the threaded code to skip the terminal code upon subsequent processing of the state. By "skip the terminal code" we mean that the terminal code, whose entry point here is state0, is only used until such time as it sees a terminal symbol. Thereafter, the nonterminal code is directly entered at state0nt. The argument to REPLACEME is the address of a label, which is supplied by the && operator.

At first glance, it appears that the ISTERMINAL guarding the terminal code is superfluous, because the terminal code cannot be invoked in the first place if the symbol in sym is a nonterminal. This is *almost* always the case. However, Figure 4.8 helps

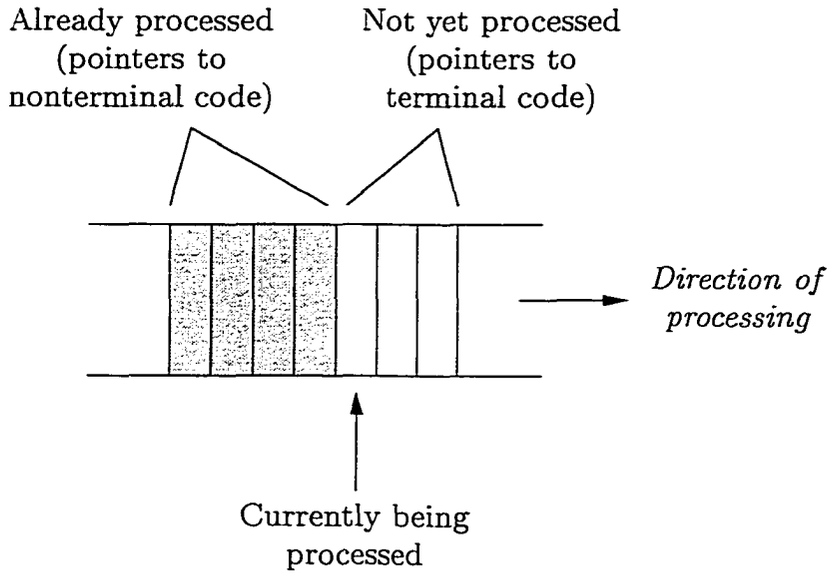


Figure 4.8: Threaded code in the current Earley set, S_i, during processing.

illustrate the problem. Midway through processing S_i, the parser could encounter a state containing a final Earley item $[A \rightarrow \bullet, i]$ which would force it to call Earley set S_i as the parent set, meaning that `sym` would hold the nonterminal symbol A. This is not an issue for states in S_i that have already been processed, but it necessitates the guard for states in S_i that have not yet been processed.

Back to the code, `DISPATCH` causes execution to go to the next piece of threaded code. `NT_` is the prefix used to encode nonterminal symbols, and `MYPARENT` simply returns the parent pointer. `GEN_CUR` is a macro which adds a state to the current Earley set, S_i; it is supplied with the label and integral number of the state to add, as well as the desired parent pointer.

As is obvious, the code generated is somewhat naïve. We have assumed that a C compiler with a reasonable optimizer is being used, which can clean up some of

the detritus like the empty switch statement. This is in contrast to earlier work on directly-executable LR parsers [50, 76] which expended considerable effort on low-level optimizations.

```
state1:
    if (ISTERMINAL(sym)) {
        switch (sym) {
          case 'n':
            GEN_NEXT(&&state3, 3, MYPARENT());
            break;
          case '-':
            GEN_NEXT(&&state4, 4, MYPARENT());
            GEN_NEXT(&&state5, 5, curstate + 1);
            break;
          (Cases for + and ( omitted for brevity.)
        }
        REPLACEME(&&state1nt);
        DISPATCH();
    }
state1nt:
    switch (sym) {
      case NT_F:
        GEN_CUR(&&state8, 8, MYPARENT());
        break;
      case NT_T:
        GEN_CUR(&&state9, 9, MYPARENT());
        break;
      case NT_E:
        GEN_CUR(&&state10, 10, MYPARENT());
        break;
    }
    DISPATCH();
```

State 1, State 0's nonkernel companion, has nontrivial terminal code. GEN_NEXT is an analogue to GEN_CUR, adding a state to the next Earley set, S_{i+1}. The terminal

code contains instances where both one *and* two transitions are made on a single symbol.

```
state2:
    if (ISTERMINAL(sym)) {
        switch (sym) {
          case T_EOF:
            GEN_NEXT(&&state11, 11, MYPARENT());
            break;
        }
        REPLACEME(&&state2nt);
        DISPATCH();
    }
state2nt:
    switch (sym) {
    }
    DISPATCH();

state11:
    if (ISTERMINAL(sym)) {
        switch (sym) {
        }
        ACCEPT();
        retaddr = NULL;
        REPLACEME(&&state11nt);
        DISPATCH();
    }
state11nt:
    switch (sym) {
    }
    DISPATCH();
```

These two states deal with the end of file. State 2's terminal code adds State 11 to S_{i+1}, which will execute the ACCEPT macro upon processing and thus exit the parser. The unreachable code following the ACCEPT is left for the compiler to optimize away.

```
state3:
    if (ISTERMINAL(sym)) {
        switch (sym) {
        }
        CALL(MYPARENT(), NT_F, &&state3L1);
state3L1:
        retaddr = NULL;
        REPLACEME(&&state3nt);
        DISPATCH();
    }
state3nt:
    switch (sym) {
    }
    DISPATCH();
```

Finally, State 3 contains a call back to a parent Earley set. The CALL macro manages the bookkeeping information shown in Figure 4.7's pseudocode. One extra label must be generated to follow each CALL in order to have an address to return to. The variable retaddr stores the return address, as is probably apparent, but also acts as a flag indicating whether or not a CALL is in progress. This flag information is used when the end of the current state is reached, to determine if a call return must be executed.

4.1.7 A Deeper Look at Implementation

While the implementation description in the last section should be sufficient to re-create DEEP, some detail has been hidden behind macros. For completeness if not clarity, we discuss the definition of those macros in this section. To make the presentation as comprehensible as possible, we have abstracted slightly away from C syntax.

Internally, there are two frequently-used pointers. One is a pointer into the array of threaded code for the current Earley set, and is called cp. The other, ap, points to the corresponding parent pointer.[15] The two pointers' values are separated by a constant value, ARGOFFSET.

That said, the mechanisms for dynamically modifying code and fetching the parent pointer are quite simple:

```
#define REPLACEME(x)     *cp = x
#define MYPARENT()       *ap
```

The dispatching and call/return definitions are also relatively straightforward. Note that state is an index to the Earley sets: the i^{th} entry of state points to Earley set S_i.

```
#define DISPATCH()
        ap = ap + 1
        cp = cp + 1
        goto **cp

#define CALL(i, arg, ret)
        retaddr = ret
        save_sym = sym
        save_cp = cp
        sym = arg
        cp = state[i]
        ap = cp + ARGOFFSET
        goto **cp
```

[15] ap is an abbreviation of "argument pointer."

```
#define RETURN()
        sym = save_sym
        cp = save_cp
        ap = cp + ARGOFFSET
        goto *retaddr
```

By far the ugliest definition is that of GEN_CUR. First, some background. The variable genptr points to the location where the piece of next threaded code should be placed. For adding Earley items, a bitmap and a set of radix trees are used, as described in Section 4.1.4. Two of each are required — one for adding to the current state, one for adding to the next — and so the bitmaps (bm) and radix tree roots (roots) are actually two-dimensional arrays. The value of the variable bank is used to alternate between the two entries of the arrays. Finally, the present function performs a radix tree search, creating the appropriate radix tree entry if it does not already exist.

```
#define GEN_CUR(label, stateno, parent)
        if (!BIT_ISSET(&bm[bank], stateno)) {
                BIT_SET(&bm[bank], stateno)
                *genptr = label
                *(genptr + ARGOFFSET) = parent
                genptr = genptr + 1
                *genptr = &&endofstate
                roots[bank][stateno].tree = NULL
                roots[bank][stateno].lazy = parent
        } else if (!present(stateno, parent, bank)) {
                *genptr = label
                *(genptr + ARGOFFSET) = parent
                genptr = genptr + 1
                *genptr = &&endofstate
        }
```

The definition of `GEN_NEXT` is analogous and has been omitted.

4.2 Evaluation

We compared DEEP with three different parsers:

1. ACCENT, an Earley parser generator [84].
2. A standard implementation of an Earley parser [49].
3. Bison, the GNU incarnation of Yacc. (Bison is table-driven.)

All parsers were implemented in C, used the same (`flex`-generated) scanner, and were compiled with `gcc` version 2.7.2.3 using the `-O` flag. Timings were conducted on a 200 MHz Pentium with 64 M of RAM running Debian GNU/Linux version 2.1. The times shown are the sum of user and system times reported by the `time` command.

Figure 4.9 shows the performance of all four on G_E. As expected, Bison is the fastest, but DEEP is a close second, markedly faster than the other Earley parsers.

In Figure 4.10 the parsers (less Bison) operate on an extremely ambiguous grammar. Again, DEEP is far faster than the other Earley parsers. The performance curves themselves are typical of the Earley algorithm, whose time complexity is $O(n)$ for most LR(k) grammars, $O(n^2)$ for unambiguous grammars, and $O(n^3)$ in the worst case.[31]

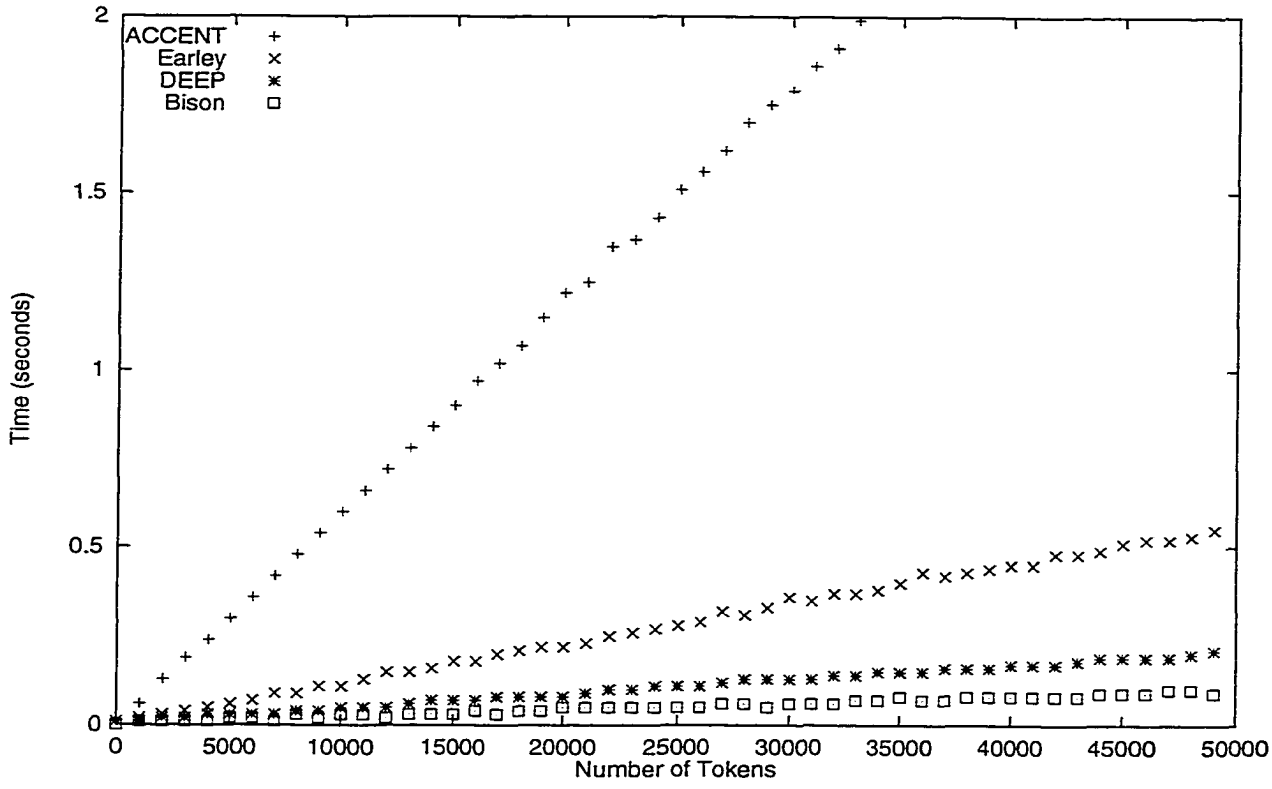


Figure 4.9: Timings for the expression grammar, G_E.

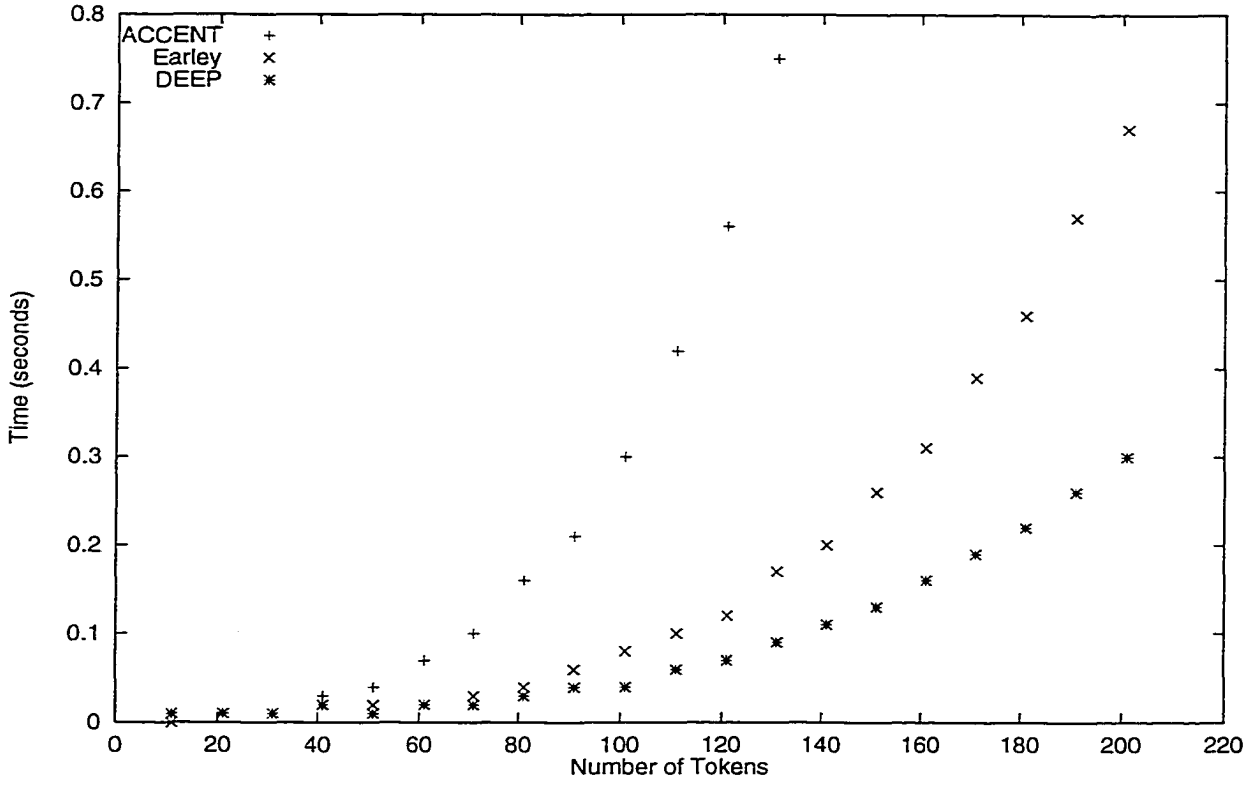


Figure 4.10: Timings for the ambiguous grammar $S \rightarrow SSx|x$.

4.3 Improvements

Next, we tried DEEP on the Java 1.1 grammar [43] which consists of 350 grammar rules.[16] Suffice it to say that only the extremely patient would be content to wait while gcc compiled and optimized this monster. The generated parser was over 19,000 lines of C code, 98% of which were in a single function. To make DEEP practical, its code size had to be reduced.

Applying Observation 3, code looking for terminal symbols may only be executed once during the lifetime of a given Earley item. In contrast, an Earley item's nonterminal code may be invoked many times. To decrease the code size, we excised the directly-executable code for terminal symbols, replacing it with a single table-driven interpreter which interprets the threaded code. Nonterminal code is still directly-executed, when COMPLETER calls back to a parent Earley set. This change reduced compile time by over 90%.

Interpretation allowed us to trivially make another improvement, which we call "Earley set compression." Often, states in the LR(0) $\overline{\text{DFA}}$ have no transitions on nonterminal symbols; the corresponding directly-executable code is a no-op which can consume both time and space. The interpreter looks for such cases and removes those Earley items, since they cannot contribute to the parse. We think of Earley set compression as a space optimization, because only a negligible performance improvement resulted from its implementation.

The version of DEEP using partial interpretation and Earley set compression is

[16] This number refers to grammar rules in Backus-Naur form, obtained after transforming the grammar from [43] in the manner they prescribe.

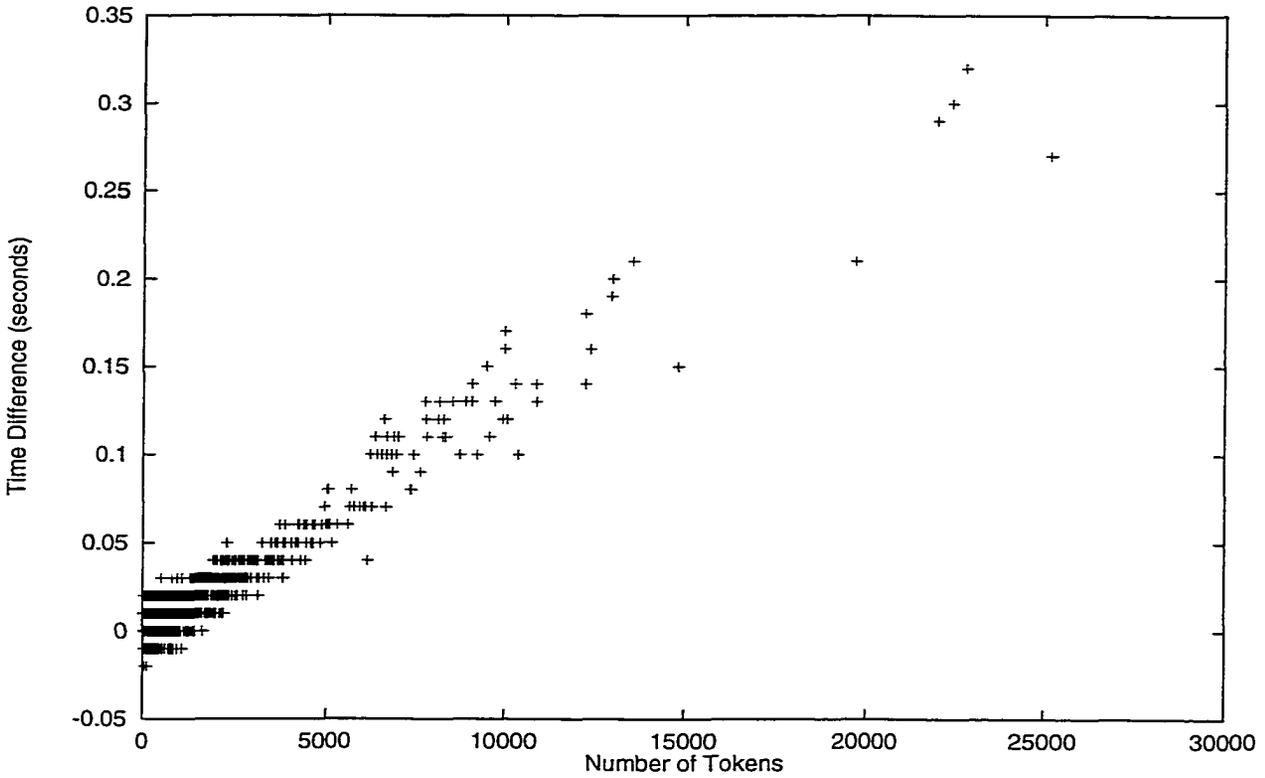


Figure 4.11: Difference between SHALLOW and Bison timings for Java 1.1 grammar, parsing 3350 Java source files from JDK 1.2.2 and Java Cup v10j.

called SHALLOW. Figure 4.11 compares the performance of SHALLOW and Bison on realistic Java inputs. SHALLOW can be two to five times slower, although the difference amounts to only fractions of a second — a difference unlikely to be noticed by end users.

One typical improvement to Earley parsers is the use of lookahead. Earley suggested that the COMPLETER should employ lookahead, but this was later shown to be a poor choice [22]. Instead, it was demonstrated that the use of one-token lookahead by the PREDICTOR yielded the best results [22]. This "prediction lookahead"

avoids placing Earley items into an Earley set that are obvious dead ends, given the subsequent input. However, the LR(0) $\overline{\text{DFA}}$ naturally clusters together PREDICTOR's output. Where prediction lookahead would avoid generating many Earley items in a standard Earley parser, it would only avoid one Earley item in SHALLOW if all the predicted dead ends fell within a single LR(0) $\overline{\text{DFA}}$ state.

We instrumented SHALLOW to track Earley items that were unused, in the sense that they never caused more Earley items to be added to any Earley set, and were not final Earley items. Averaging over the Java corpus, 16% of the Earley items were unused. Of those, prediction lookahead could remove *at most* 19%; Earley set compression removed 76% of unused Earley items in addition to pruning away other Earley items. We conjecture that prediction lookahead is of limited usefulness in an Earley parser using any type of LR automaton.

4.4 Related Work

Appropriately, the first attempt at direct execution of an Earley parser was made by Earley himself [30]. For a subset of the CFGs which his algorithm recognized in linear time, he proposed an algorithm to produce a hardcoded parser. Assuming the algorithm worked and scaled to practically-sized grammars — Earley never implemented it — it would only work for a subset of CFGs, and it possessed unresolved issues with termination.

The only other reference to a directly-executable "Earley" parser we have found is Leermakers' recursive ascent Earley parser [64, 65, 66]. He provides a heavily-

recursive functional formulation which, like Earley's proposal, appears not to have been implemented. Leermakers argues that the directly-executable LR parsers which influenced our work are really just manifestations of recursive ascent parsers [65], but he also notes that he uses "recursive ascent Earley parser" to denote parsers which are not strictly Earley ones [66, page 147]. Indeed, his algorithm suffers from a problem handling cyclic grammar rules, a problem not present in Earley's algorithm (and consequently not present in our Earley parsers).

Using deterministic parsers as an efficient basis for general parsing algorithms was suggested by Lang in 1974 [63], and has been applied in Earley parsers [71] and Earley-like parsers [5, 21, 93]. However, none have explored the benefits of using an almost-deterministic automaton and exploiting Earley's ability to simulate nondeterminism.

4.5 Future Work

By going from DEEP to SHALLOW, we arrived at a parser suited for practical use. This came at a cost, however: as shown in Figure 4.12, the partially-interpreted SHALLOW is noticeably slower than the fully-executable DEEP. Even with the slowdown, SHALLOW's timings are still comparable to Bison's. One area of future work is determining how to retain DEEP's speed while maintaining the practicality of SHALLOW.

On the other side of the time/space coin, we have yet to investigate ways to reduce the size of generated parsers. Comparing the sizes of stripped executables,

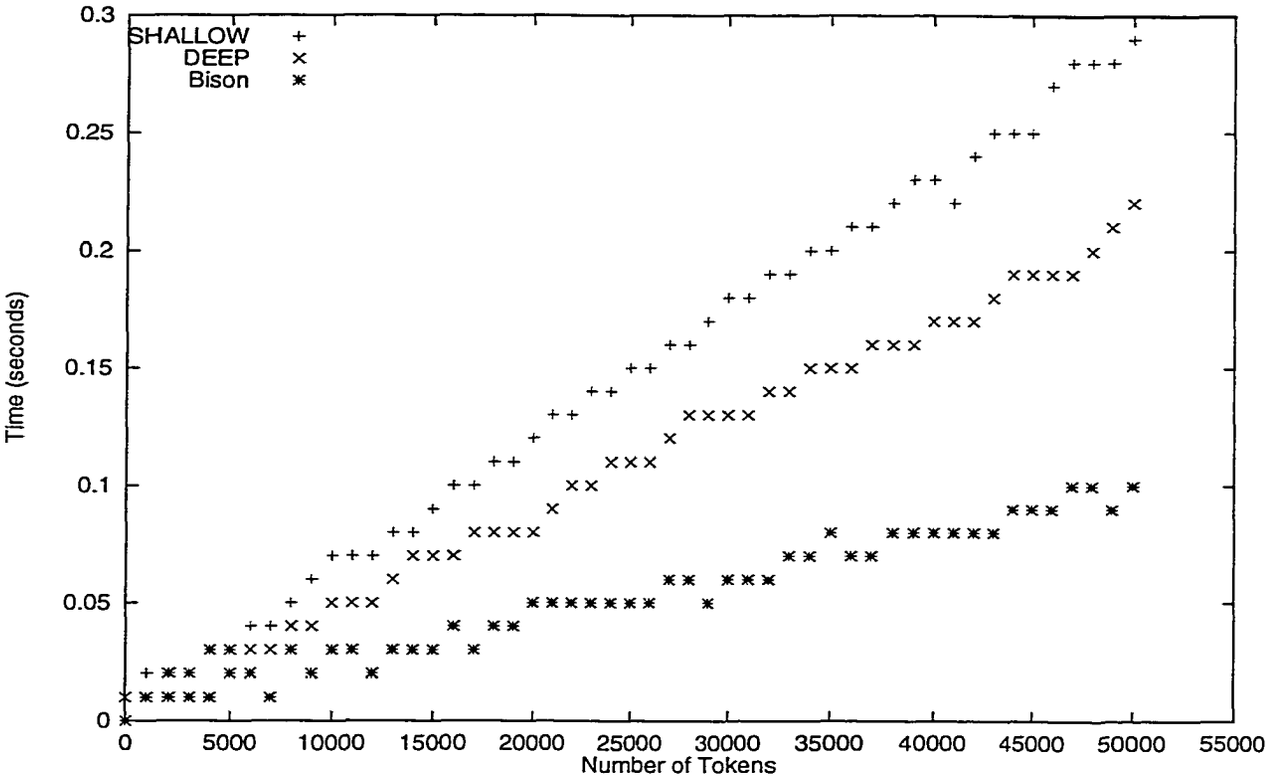


Figure 4.12: Performance impact of partial interpretation of G_E.

SHALLOW parsers for G_E and Java were 1.5 and 9.0 times larger, respectively, than the corresponding Bison parsers. Since the *worst* space increase reported for directly-executable LR parsers[17] was a factor of four [76], clearly we have some work to do.

Additionally, we have not yet explored the possibility of using optimizations based on grammar structure. One such example is elimination of unit rules,[18] grammar rules such as $A \rightarrow B$ with only a single nonterminal on the right-hand side [46]. Techniques like this have been employed with success in other directly-executable parsers [50, 77].

[17] References [20, 50, 76, 77] were taken into account.
[18] Also called chain rule elimination.

Will DEEP ever make its way into SPARK? The answer is a definite "not yet." The work described in this chapter gives an idea of how fast Earley's algorithm can operate, and is interesting in that respect. However, we have not currently reconciled the amount of precomputation required to generate a DEEP parser with the requirements of SPARK, namely that the parser perform all of its work at run time. This is another area of future study.

4.6 Summary

Directly-executable LR parsing techniques can be extended for use in general parsing algorithms such as Earley's algorithm. The result is a directly-executable Earley parser which is substantially faster than standard Earley parsers, to the point where it is comparable with LALR(1) parsers produced by Bison.

Chapter 5

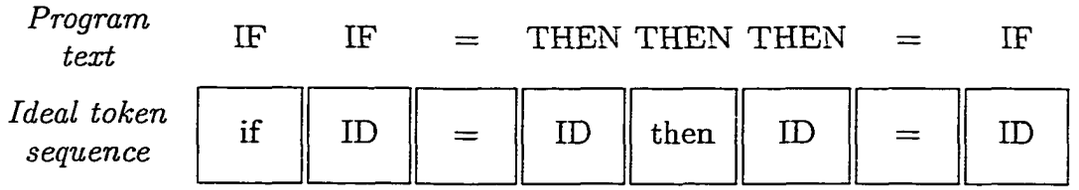
Schrödinger's Token[19]

From performance, we now shift our focus to the delayed action problem. Recall that the delayed action problem arises because Earley parsers must, in general, read the entire input before executing semantic actions. As a result, the parser cannot feed back information to the scanner during recognition, which leaves the scanner with no idea how to handle context-dependent tokens. As we discuss in this chapter, the delayed action problem can be addressed not by fixing the parsing algorithm, but by exploiting it.

Usually, the type of a token that the scanner returns — the class of words that the token describes — is clear-cut. Problems arise, however, when a token's type depends upon context information that the scanner is not privy to. An example of this problem comes from PL/I, where statements like the following are legal [3, 37]:

IF IF = THEN THEN THEN = IF

[19] An earlier version of this work appeared in [12].



stmt → ifstmt | asgnstmt
ifstmt → "if" expr "then" stmt
asgnstmt → "ID" "=" expr
expr → "ID" "=" "ID"

Figure 5.1: Ideal token sequence and (simplified) grammar.

Ideally, the scanner should divine which uses of "if" and "then" correspond to keywords, and which correspond to identifiers; this would allow the parser to use grammar rules which directly reflect the structure of the language (PL/I, in this case). This ideal token sequence and grammar are shown in Figure 5.1. Unfortunately, the necessary context information to produce this token sequence is not directly available to the scanner.

In PL/I, this problem arises because keywords like "if" are not reserved, and can be used in other contexts; most modern languages have avoided this particular difficulty. And while compilation of PL/I is no longer a hot topic, this same problem still arises when compiling little languages.

More generally, we do not restrict ourselves to keywords, and consider *any* situation where a token's type is context-dependent. One language may be embedded within another, as SQL can be embedded within C, C++, or COBOL; the interpretation of tokens may vary greatly between an SQL construct and the host language! There may also be many dialects of a single language which software re-engineering

tools are obliged to recognize [91].

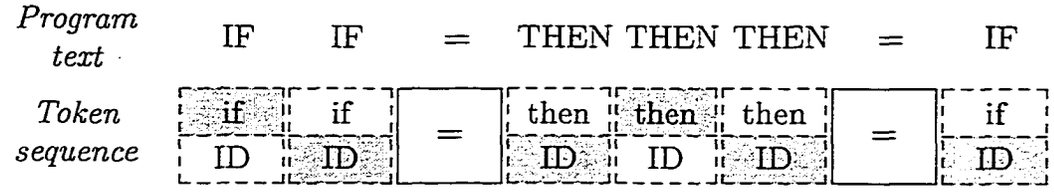
In this chapter we discuss a simple technique for handling context-dependent tokens, its implementation, and some applications. We have used this technique in SPARK with excellent results. For comparison purposes, alternative techniques are also presented.

5.1 Schrödinger's Token

In 1935, Erwin Schrödinger published a paper on quantum mechanics in which he posed the exaggerated case of the now-famous Schrödinger's Cat [83]. A cat and a flask of lethal acid share accommodation in a steel chamber, along with some radioactive material. If an atom in the radioactive material decays — there is a 50% chance of this happening — the flask is broken, and the global cat population is decreased by one. However, the decay of the atom, and the fate of the cat, are unknown until the chamber is opened; effectively, the cat exists in a superposition of "alive" and "dead" states until that time.

This idea of superposition is a faithful model of the token interpretation problem. In lieu of context information, a token does *not* have a unique type but instead has a superposition of types: it is all of its possible types simultaneously, until the parser peeks inside the chamber and resolves the token's type with context information. We use the term "Schrödinger's token" to refer to a token which has a superposition of types.

Returning to our PL/I example, Figure 5.2 shows the new token sequence from



stmt → ifstmt | asgnstmt
ifstmt → "if" expr "then" stmt
asgnstmt → "ID" "=" expr
expr → "ID" "=" "ID"

Figure 5.2: Token sequence and grammar, using Schrödinger's tokens. The shaded token types indicate which interpretation is eventually used by the parser.

the scanner. "If" and "then" are now Schrödinger's tokens, since they can be either a keyword or an identifier; the "=" token still has a unique type.[20] Notice that the grammar is unchanged from the "ideal" grammar in Figure 5.1 — the use of Schrödinger's tokens is thus programmer-friendly, in that no modifications to the grammar are required, and therein lies a problem.

Physical superposition embodies[21] nondeterminism, something which current computers are not particularly adept at. Parsing algorithms commonly used for compilers, such as the LALR(1) algorithm in Yacc [53, 68], are deterministic and cannot generally cope with the fact that input involving a Schrödinger's token may (temporarily) not have a unique parse. Instead, we use more general parsing techniques such as generalized LR parsing [89] or Earley's algorithm [30, 31], which effectively simulate nondeterminism if necessary to handle ambiguity in the grammar.

We note that the Schrödinger's token technique has been used in the past by

[20] One could argue that "=" also has a dual interpretation, as a comparison operator or an assignment operator.

[21] Or, in the cat's case, possibly disembodies...

computational linguists [99, 72], who are accustomed to using more general parsing techniques. Specifically, the technique was used in natural language parsing to model the case where a single word can belong to many different parts of speech. To the best of our knowledge, computational linguists have no name for this technique, nor has it been applied outside the area of natural language.

5.2 Alternative Techniques

There are a number of other techniques[22] for dealing with context-dependent tokens. We present them here in a parser-independent manner but, in practice, many of them involve more programmer effort than is initially apparent. This is because techniques requiring grammar modification often trigger an orgy of grammar rewriting in order to make a modified grammar palatable to a particular parsing engine.

5.2.1 Lexical Feedback

The scanner can be made aware of context if the parser and scanner share some state. The parser uses this shared state to communicate context information to the scanner; this is referred to as lexical feedback [53, 68]. For example, if we were to use lexical feedback to have the scanner return a "then" token rather than an "ID" token, we would add actions to the "ifstmt" rule in Figure 5.1:

[22]Some of these techniques are folklore. Consequently, not all the citations in this section refer to original sources.

ifstmt → "if" expr
{ *expectKeyword := true* } "then"
{ *expectKeyword := false* } stmt

The scanner would return a "then" or an "ID" token based on the value of the "expectKeyword" flag.

Lexical feedback has a number of problems in practice. It couples the scanner and parser tightly: not only do they share state, but the parser and scanner must operate in lock-step. The scanner cannot, for example, tokenize the entire input in a tight loop, or operate in a separate thread of execution, because at any moment the parser may direct it to change how it interprets the input. Additionally, the programmer must fully understand how and when the parser handles tokens, otherwise subtle bugs may be introduced.

5.2.2 Enumeration of Cases

If the scanner insists upon returning the wrong token type in the wrong context, the grammar can be modified such that the "wrong" token type is accepted as well as the right token type: the programmer enumerates all the valid cases in the grammar [38, 27]. In our example, the problem is that the scanner can return an "if" or "then" token, when we really want an "ID" token. We replace all occurrences of "ID" with a new nonterminal that accepts all valid token types that we might see from the scanner:

```
    stmt → ifstmt | asgnstmt
  ifstmt → "if" expr "then" stmt
asgnstmt → id "=" expr
    expr → id "=" id
      id → "ID" | "if" | "then"
```

Besides these grammar modifications, the correct addition of a new keyword requires changes to logically unrelated grammar rules.

5.2.3 Language Superset

An alternative approach to handling context-dependent tokens is for the scanner to *never* attempt to distinguish between them, and to always return the most generic token type. The grammar can then be rewritten to accept a superset of the original language. For our running example, this would look like:

stmt	→	ifstmt | asgnstmt
ifstmt	→	"ID" expr "ID" stmt
asgnstmt	→	"ID" "=" expr
expr	→	"ID" "=" "ID"

The problem, of course, is that eventually an "ifstmt" must have its generic "ID" tokens examined, to ensure that the first one is "if" and the second is "then." This can be done in actions associated with the grammar rules, or deferred to later semantic processing. In either case, the programmer must perform work that the parser was supposed to do in the first place! Furthermore, the intent of the grammar rules is now hidden, making maintenance difficult.

5.2.4 Manual Token Feed

The "manual token feed" method is analogous to manually feeding paper to a printer one sheet at a time. The parser is modified so that when it would normally flag an error, it instead calls a programmer-supplied subroutine with the offending token. The subroutine can then feed the parser an alternate token, and resume parsing. The

process of feeding in alternate tokens can continue until the parse succeeds, basically creating a backtracking parser. However, as Kanze points out in his paper describing the technique [54], there are a number of situations where it doesn't work, due to the parser occasionally being in an inopportune state when an error is noted.

5.2.5 Synchronization Symbols

Tarhio [88] proposed a technique using what he called "synchronization symbols." As with the language superset approach, the scanner always returns a generic type for context-dependent tokens. The difference is that the parser injects new tokens — synchronization symbols — into the token stream on the fly, that supply context information. This achieves the same result as lexical feedback, but without tying the parser to the scanner. Using this technique, and the synchronization symbols "if keyword," "then keyword," and "identifier," our grammar would be:

stmt	→	ifstmt | asgnstmt
ifstmt	→	if expr then stmt
asgnstmt	→	id "=" expr
expr	→	id "=" id
if	→	sync "if keyword"
then	→	sync "then keyword"
id	→	sync "identifier"
sync	→	"ID" { *insert appropriate synchronization symbol, based on context* }

A number of assumptions must be made for synchronization symbols to work: the token stream must be mutable; actions associated with grammar rules must be executed by the parser at the earliest convenience, to insert the synchronization symbols in time; the parser must not have looked ahead at later tokens already. And, as with

most of these methods, the grammar must be modified.

5.2.6 Oracles

Abrahams [1] describes a compiler in which PL/I was parsed using a relatively weak method — LL(1) parsing. Here, neither the scanner *nor* the parser were powerful enough to collect all context information by themselves. When the parser required additional context information in order to decide how to interpret tokens, it invoked an oracle. The oracle called the scanner repeatedly to "peek ahead" into the token stream, looking for patterns that would provide context information. Upon finding such a pattern, the oracle rewound the scanner to its previous location; the parser could then continue with context information from the oracle, letting the scanner re-scan the tokens.

In this example, the oracle can be seen as compensating for an insufficiently–powerful parsing method, rather than a means to handle context-dependent tokens. However, one could just as easily devise an arrangement where an oracle is invoked by the scanner, giving the scanner enough context information to return the appropriate token type.

In practice, oracles may not always be feasible due to the performance impact of reprocessing input and the complexity of constructing a correct oracle.

5.2.7 Scannerless Parsing

Clearly, if the central problem behind context-dependent tokens is a communication barrier between the scanner and parser, then merge them together to remove the barrier! This is the promise of scannerless parsing [82, 94]. The programmer supplies a single grammar which describes both lexical and syntactic rules, making it easy to express the context of tokens.

There are tradeoffs, however. The programmer must supply a grammar which specifies the legal placement of whitespace and comments. Besides being error-prone, this does not decrease the grammar size nor increase its readability. The other concern is efficiency, because scannerless parsing uses more powerful pushdown automata to handle individual characters rather than finite-state automata. This latter point is hard to judge, as scannerless parsing tools are not available, and no timing results have been published.

5.2.8 Discussion of Alternatives

Use of Schrödinger's tokens is closest in spirit to enumeration of cases. The two methods accomplish the same goal, albeit with different tradeoffs: enumeration of cases leaves the scanner unchanged and modifies the grammar, the opposite of Schrödinger's tokens. More broadly, Schrödinger's tokens suffer from none of the problems of alternative techniques, such as extensive grammar modification and limiting assumptions placed on the scanner and parser's operation.

5.3 Implementation

Implementation of Schrödinger's tokens can be divided into two logical parts: support required in parsing tools, and support by the user of those parsing tools, the programmer.

5.3.1 Programmer Support

Given a parser generator which supports Schrödinger's tokens, the programmer need only supply a scanner which produces said tokens. This can be done in a hand-crafted scanner as well as using scanner generator tools.

Figure 5.3 gives a partial Lex [68, 67] specification for our running example. A distinct token type, "SCHRODINGER," is returned to the parser to signal the appearance of a Schrödinger's token. The set of possible types is stored in a global array for use by the parser; for pedagogical reasons, a maximum of two superpositioned types is imposed. The usual disambiguation rules for preferring the longest match are applied.

How is the overlap between token definitions determined? In our experience, this can usually be done by inspection without much difficulty in the typical case where a single lexeme corresponds to multiple token types. However, it is possible to construct a scanner generator tool which would look for such cases automatically.

```
%%

=               return '=';
if              return schrodinger(IF, ID);
then            return schrodinger(THEN, ID);
[a-zA-Z]+       return ID;

%%

extern int types[2];

int schrodinger(int type1, int type2) {
    types[0] = type1;
    types[1] = type2;
    return SCHRODINGER;
}
```

Figure 5.3: Partial Lex specification using Schrödinger's tokens.

5.3.2 Parser Tool Support

Parser generators must have appropriate support for Schrödinger's tokens. We emphasize that this is a one-time tool modification, invisible to the end user/programmer.

For expository purposes, we begin with Yacc [53, 68]. Yacc uses an LALR(1) parsing algorithm, a flavor of shift/reduce parsing. In shift/reduce parsing, the parser shifts values onto a stack until a valid right-hand side of some grammar rule is seen atop the stack, at which time the parser reduces, popping values from the stack. The decision as to what parsing action to take is controlled by a deterministic automaton, whose transitions Yacc encodes in a table. At its core, Yacc's parsing algorithm is a loop, indexing into a table based on the topmost stack value and the current input symbol, and performing the action specified in the table entry. Figure 5.4a shows this

```
a = input()
while (true) {
    s = top_of_stack()
    switch (action[s, a]) {
        case shift s':
            ...
        case reduce A → α:
            ...
        case accept:
            ...
        case error:
            ...
    }
}
```

(a)

```
a = input()
while (true) {
    s = top_of_stack()
    foreach t ∈ a.types {
        switch (action[s, t]) {
            case shift s':
                ...
            case reduce A → α:
                ...
            case accept:
                ...
            case error:
                ...
        }
    }
}
```

(b)

Figure 5.4: Pseudocode for (a) the LALR(1) parsing algorithm, and (b) conceptual modifications for Schrödinger's token support.

algorithm in pseudocode form.

As shown in Figure 5.4b, supporting Schrödinger's tokens is a matter of performing an action for each of the superpositioned token types. Unfortunately, this is where LALR(1) parsing and Schrödinger's tokens part ways. Even a simple example such as the one in Figure 5.2 would require the parser to be in two distinct automaton states simultaneously, which is not possible in a deterministic automaton.

Where Yacc's deterministic LALR(1) algorithm would fail, a general parsing algorithm like Earley's algorithm or generalized LR (GLR) parsing [89] simulates nondeterminism, effectively running multiple parsers in parallel. A GLR parser, for instance, employs the algorithm in Figure 5.4a. However, instead of the single parsing

stack that LALR(1) parsers have, a GLR parser has a set of parsing stacks: conceptually, one stack for each derivation currently being considered. A GLR parser must thus use the LALR(1) algorithm for each of its stack tops.

Regardless of the general parsing algorithm used, the required change is the same as outlined above. Any operation that depends on a token's type must be repeated for all the superpositioned types in a Schrödinger's token. In the Earley and GLR parsers we have examined, very few lines of code need modification.

5.3.3 Schrödinger's Tokens and SPARK

Depending on the parser's design, no changes at all may be required to support Schrödinger's tokens. In an object-based parser such as the Earley parser we use in SPARK, tokens are black boxes. The parser invokes a comparison method within each token to discern information about its type, shown here as pseudocode:

```
class Token:
    def compare(self, type):
        if type == self.type:
            return true
        return false
```

SPARK's GenericParser uses this comparison method exclusively to probe the type of a token. All the programmer must do is to modify a token's comparison method to respond "yes" when the parser compares the token to any of the superpositioned token types. This requires no parser modification, only a change to the token's comparison method:

```
class Token:
```

```
def compare(self, type):
    if type ∈ self.types:
        return true
    return false
```

In this parsing model, a one-line change to the token's comparison method is sufficient for Schrödinger's tokens to work.

5.4 Applications

Schrödinger's tokens have many uses. We present three application areas: domain-specific languages, fuzzy parsing, and whitespace-optional languages.

5.4.1 Domain-specific Languages

Domain-specific languages are often designed and implemented in an *ad hoc* fashion, making the use of traditional compiler techniques difficult, in part due to context-dependent tokens.

Configuration files are one example. Although they vary greatly in complexity, a simple format involves only key-value pairs, as in the `/etc/resolv.conf` file from our workstation:

```
search      csc.uvic.ca
nameserver  142.104.96.1
nameserver  142.104.6.1
```

On every line, the first word is the key, and the remaining words on the line are the value. If we were to process this using compiler tools, it would be reasonable to consider making each key a reserved word. This way, the grammar would reflect the

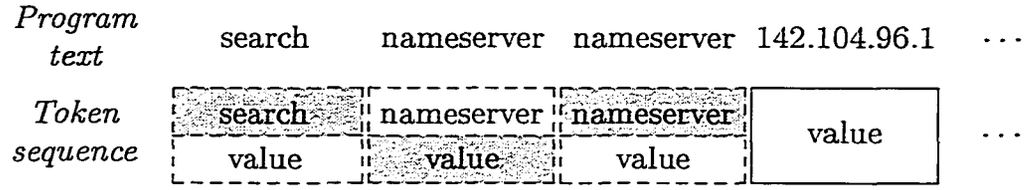


Figure 5.5: Schrödinger's tokens for parsing key-value pairs.

structure of the language, and appropriate actions could easily be associated with each different key:

```
searchstmt      →  "search" "value"       { action for search }
nameserverstmt  →  "nameserver" "value"   { action for nameserver }
...
```

However, nothing except good taste prevents us from re-using a key's name as a value, perhaps changing the first input line to "search nameserver." Figure 5.5 shows how the scanner can create Schrödinger's tokens to easily handle this case.

Some other domain-specific languages, and examples of how Schrödinger's tokens apply to their implementation:

- Command-line arguments. On UNIX[23] systems, commands like `find` and `expr` allow complicated expressions as command-line arguments. Keywords are not reserved and may appear as arguments.

- Text-based network protocols. A number of network protocols, such as FTP [78], HTTP [19], and SMTP [79], use little languages for client-server communication. Again, keywords are not reserved.

- Programming languages. Some modern domain-specific programming languages have context-dependent tokens. We experienced this when re-implementing

[23]UNIX is a registered trademark of The Open Group in the United States and other countries.

Guide [69] using compiler tools; the initial implementation was an *ad hoc* Perl script. The data description language ASN.1 [52] also makes provision for non-reserved keywords that are distinguished by their context.

5.4.2 Fuzzy Parsing

Koppler [61] defined "fuzzy parsing" to be parsing which only recognizes part of an input. Fuzzy parsers are useful for software re-engineering and tools which only look for certain features in their input. A fuzzy parser could extract all the data structure definitions from a program, for instance, or find all public class members in a Java program.

A fuzzy parser operates by skipping tokens until it sees a specified "anchor symbol," a sentinel token that indicates the start of the input sequence that the fuzzy parser recognizes. After parsing this input sequence, the fuzzy parser reverts to skipping tokens again.

From an engineering perspective, a fuzzy parser is looking for a signal amidst noise. Figure 5.6 shows how this idea can be applied to create a fuzzy parser that locates class definitions in C++ programs, for the purposes of discovering the class hierarchy (an example from Koppler's paper). The scanner makes *every* token a Schrödinger's token with a superposition of two types: the token's actual type that would be returned normally, and "noise." The grammar can then skip uninteresting input tokens by interpreting them as noise.

The scanner, once configured to return Schrödinger's tokens in this manner, can be re-used without modification for any fuzzy parsing of C++. Fuzzy parsers constructed

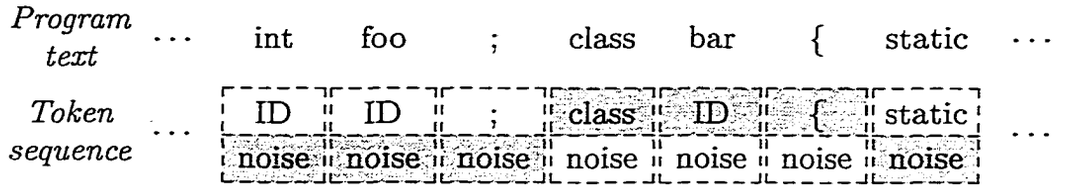


Figure 5.6: Fuzzy parsing of C++ using Schrödinger's tokens.

with Schrödinger's tokens and a general parser are more powerful than those described by Koppler, because the "signal" they look for need not begin with an anchor symbol.

5.4.3 Whitespace-optional Languages

Schrödinger's tokens were envisaged as a means of representing one piece of text that may have multiple token types. However, there are some languages where even locating token boundaries is a Herculean task. The classic example is Fortran, where whitespace is optional, and scanning is tricky [81, 36]. For example, the partial input "DO57I=" may correspond to two or four tokens, depending on the context. (Two abutted identifiers or integers are assumed to be invalid.)

Figure 5.7 suggests how to address this using Schrödinger's tokens. There are two distinct token sequences; the shorter of the two sequences is padded out with a special "null" type. These null tokens must be ignored by the parser, which can be accomplished by grammar modifications that permit an "ID" to be followed by zero or more null tokens.

This idea may be used to handle lexical ambiguity between subranges and real number representations in some languages, when it is unclear if "1..2" denotes a range of values or two adjacent real numbers. As shown in Figure 5.8, this idea also makes

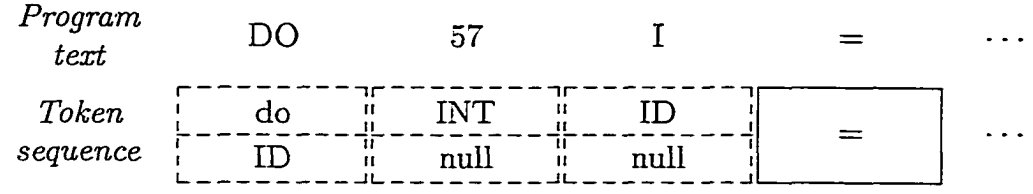


Figure 5.7: Schrödinger's tokens for parsing Fortran. The token interpretation is not shown due to lack of sufficient context.

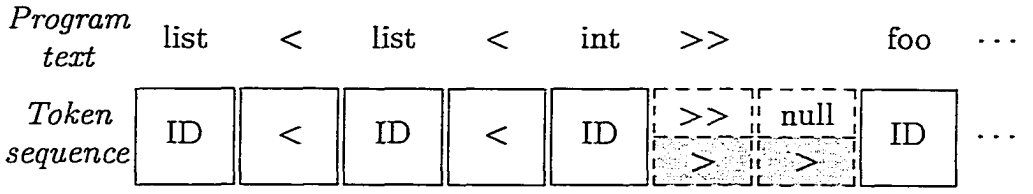


Figure 5.8: Schrödinger's tokens for parsing C++ template syntax.

it straightforward to resolve the ambiguity in C++ between the right shift operator ">>" and nested template parameters (C++ template parameter lists are terminated by the ">" symbol) [87].

5.5 Summary

A superposition of token types — a Schrödinger's token — represents situations where a token's interpretation is dependent on context. Using general parsing algorithms, Schrödinger's tokens have five major advantages compared to other methods for addressing the same problem:

1. No grammar modification is required; the grammar can accurately reflect the language being parsed, enhancing readability and maintenance.

2. Robustness: use of Schrödinger's tokens does not require the programmer to

understand the parser's internal mechanisms. As well, no assumptions are made as to when the parser performs any actions that are associated with grammar rules.

3. Accurate modelling of context-dependent tokens, capturing the scanner's uncertainty with respect to token types.

4. No tight coupling between scanner and parser. This again simplifies maintenance, and allows scanners and parsers to be seen as interchangeable software components.

5. Support for Schrödinger's tokens may require as little as a one-line code change.

Chapter 6

Early Action in an Earley Parser

In the worst case, an Earley parser has to read its entire input prior to constructing any parse trees — this is the delayed action problem. As semantic actions associated with grammar rules are not executed until parse trees are built (explicitly or implicitly), execution of semantic actions does not happen on the fly. This prevents any immediate action from being taken during input recognition, such as file inclusion or macro expansion.

However, it has been recognized for some time that this worst case, having to read the entire input, does not typically occur in programming language grammars; rather, they tend to only exhibit local ambiguities [63]. Exploiting this idea, we present conditions under which an Earley parser *can* build parse trees during recognition, giving a solution to the delayed action problem as well as yielding space savings.

6.1 Safe Earley Sets

Earley's algorithm pursues all possible derivations of its input at once. From the parser's point of view, it is alternating between two states. There are unambiguous parts of the input's derivation, where the parser is keeping track of only a single derivation sequence. There are also ambiguous parts, where the parser is tracking multiple derivation sequences. At the extremes are inputs whose derivation is completely unambiguous, and inputs that are totally ambiguous.

What characteristics does an Earley set have when recognizing an unambiguous part of the derivation? To help answer this question, we define an Earley set S_i as *safe* if it has the following properties:

Property 1. S_i must contain at least one final Earley item.

Property 2. For every final Earley item $[A \rightarrow X_1 X_2 \ldots X_n \bullet, p] \in S_i$, there exists no other Earley item $[B \rightarrow \alpha X_n \bullet \beta, q] \in S_i$.

Property 3. No final Earley item $[A \rightarrow \bullet, p]$ exists in S_i. This is a special case of Property 2, because there is effectively a symbol — albeit the empty string — before the $\bullet$.

Property 4. A total ordering $\prec$ exists on the final Earley items in S_i. Given two final Earley items $I_1 = [A \rightarrow \alpha \bullet, p]$ and $I_2 = [B \rightarrow X_1 X_2 \ldots X_n \bullet, q]$, we say $I_1 \prec I_2$ if and only if $X_n \Rightarrow^* \alpha$. Combined with Property 2, this precludes Earley items produced by cycles of the form $A \Rightarrow^+ A$ from being part of a safe set.

Without loss of generality, we assume that all Earley items have been generated for S_i before its candidacy as a safe set is considered. (In other words, SCANNER, PREDICTOR, and COMPLETER have finished running.)

A final Earley item is not guaranteed to be part of a derivation of the input; it could be an eventual dead-end path. However, this is *not* the case in a safe set. We prove this property of safe sets, then show that this implies that a safe set corresponds to an unambiguous point in the parse.

Theorem 1 *Final Earley items in a safe Earley set S_i are part of some derivation of the input.*

Proof. Given a final Earley item $I = [A \rightarrow X_1 X_2 \ldots X_n \bullet, p] \in S_i$, there are two cases to consider. (The third possibility, where the right-hand side of the rule is empty, is precluded by Property 3.)

Case 1. X_n is a terminal symbol. By Property 2, there can be no other way for the input symbol a_i to be consumed. I must therefore be part of a derivation of the input.

Case 2. X_n is a nonterminal symbol. I must have been added to S_i by COMPLETER, which means that there must also be a final Earley item $I' = [X_n \rightarrow Y_1 Y_2 \ldots Y_n \bullet, q]$ in S_i. What can Y_n be? Again, there are two cases:

Case 2a. Y_n is a terminal. This is Case 1; I' must be part of some derivation. As I' is directly responsible for I being in S_i, and there can be no other

way for X_n to be consumed (by Property 2), I must also be part of some derivation.

Case 2b. Y_n is a nonterminal. We have arrived back at Case 2! Property 4's total ordering of final Earley items ensures that there is a ordered sequence of final Earley items in S_i. This sequence must terminate with a final Earley item that has a terminal symbol before the dot, otherwise COMPLETER could not have been invoked for any item in S_i (recall that empty rules are not present in safe sets). Employing the same reasoning as in Case 2a, I' and I must be part of some derivation.

I is therefore always part of some input derivation. □

Corollary 1 *If Earley set S_i is safe, then it corresponds to an unambiguous part of the input's derivation.*

Proof. For an ambiguity to exist, there must be two or more derivations of a sentence. In all the above cases, there is a unique way to derive each terminal and nonterminal, so no ambiguity can coexist with a safe Earley set. □

We observe that Property 4 follows from the other three properties. This is captured in the following lemma.

Lemma 1 *If Properties 1–3 hold for an Earley set S_i, then Property 4 holds.*

Proof. We look at the ways that a final item may be added to Earley set S_i:

1. Added by SCANNER (running on S_{i-1}). If Property 2 holds for S_i, there can only be one such final item.

2. Added by COMPLETER. If Property 3 holds for S_i, then the only way to cause COMPLETER to run initially on S_i is by virtue of SCANNER having added a final item to S_i. Thereafter, only other final items added to S_i by COMPLETER can cause COMPLETER to run on S_i again.

   What happens when COMPLETER adds final items to S_i? The answer depends on how many final items it adds. If COMPLETER adds one new final item each time, then Property 4 is preserved, because there is a distinct ordered sequence. As mentioned, cycles are prevented if Property 2 holds.

   On the other hand, if COMPLETER adds several final items, then the dot must have been moved over the same nonterminal symbol to create those final items. This would result in a violation of Property 2.

□

Lemma 1 implies that it is sufficient to verify Properties 1–3 to ascertain whether a given Earley set is a safe set.

6.2 Practical Implications

A safe Earley set is straightforward to detect during recognition of the input, and finding one has two important ramifications.

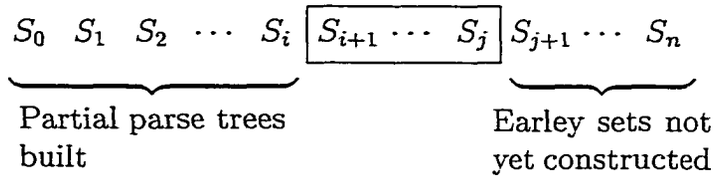


Figure 6.1: Window on Earley sets. S_i is the last safe set; S_j is being considered for candidacy as a safe set.

6.2.1 Construction of Partial Parse Trees

By Theorem 1, recognizing a safe set means that the set corresponds to a single derivation path. It is therefore possible to enumerate all derivation paths prior to that point, since we know we cannot be in the middle of a local ambiguity. In a compiler-compiler, we would typically want to execute actions associated with a single derivation of the input. Upon reaching a safe set, we can choose a derivation path and resolve any local ambiguities prior to the safe set by various means: heuristics, calling user-defined routines, disambiguating rules [2], or more elaborate means [56].

In practice, a window effectively exists on the Earley sets, as seen in Figure 6.1. The trailing edge of the window is at the last safe set; the leading edge is at the set currently being considered for safety. All possible partial parse trees have been built prior to the window. Earley sets in the window contain information about partial parse trees we look forward to building upon discovering another safe set, since this is the only condition under which the trailing edge of the window moves.

Intuitively, one can think of the window expanding in two circumstances. First, the window may expand when all of the symbols in a rule have not yet been seen. In $A \rightarrow abc$, for instance, we would have to see all of a, b, and c before we would have a

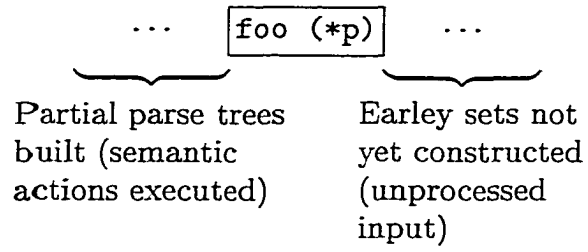


Figure 6.2: Local ambiguity in C++. Is this a declaration or a function invocation?

```
constructSet(S_i)
if issafe(S_i):
    constructTrees(lastsafe, i)
    lastsafe = i
```

Figure 6.3: Pseudocode for construction of partial parse trees.

final item to consider. Second, ambiguity causes window expansion, since ambiguity necessarily entails multiple input derivations. In ambiguous C++ declarations, the window would expand to cover the Earley sets involved in the locally-ambiguous declaration; only after the end of the declaration could the trailing window edge move forward. This is sketched in Figure 6.2.

Figure 6.3 gives some pseudocode which shows how Earley's algorithm would be modified to incorporate construction of partial parse trees. The variable `lastsafe` would have an initial value of zero.

6.2.2 Space Savings

Recognizing a safe set admits a simple way to save space during parsing, whose formal justification comes from the following theorem [4].

Theorem 2 (Aho and Ullman) $[A \rightarrow \alpha \bullet \beta, i] \in S_j$ *if and only if* $\alpha \Rightarrow^* a_{i+1} \cdots a_j$

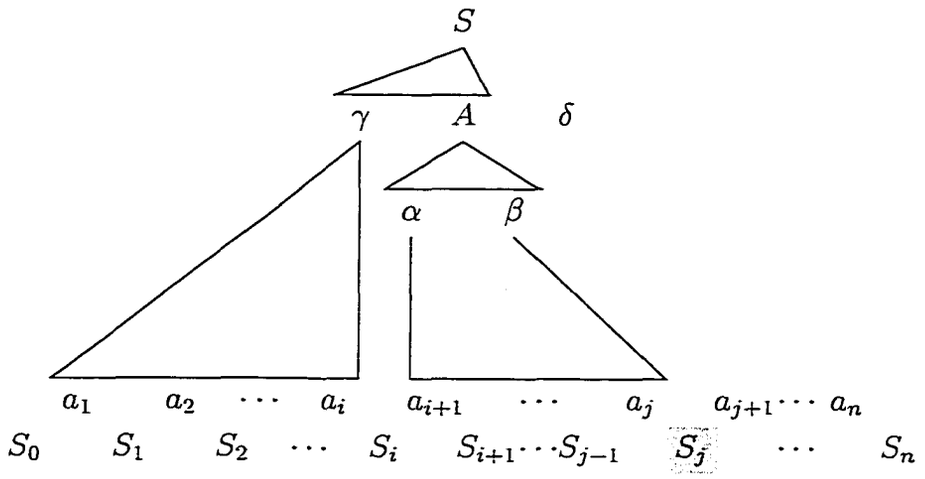


Figure 6.4: Saving space. The shaded set is a safe set.

and there are strings γ *and* δ *such that* $S \Rightarrow^* \gamma A \delta$ *and* $\gamma \Rightarrow^* a_1 \cdots a_i$.

In the case of a safe set, we are dealing with final Earley items: $[A \rightarrow \alpha\beta\bullet, i] \in S_j$. Then $\alpha\beta \Rightarrow^* a_{i+1} \cdots a_j$ and, by Corollary 1, we know that this can be the only way of doing so. As a result, the Earley sets $S_{i+1} \ldots S_{j-1}$ can never be referenced again and may be deleted. This is shown graphically in Figure 6.4.

6.3 Empirical Results

We applied our technique to grammars for five widely-used programming languages: Java, Modula-2, Oberon-2, Perl, and Python. These grammars have extremely different characteristics. The Java grammar is unambiguous, not even sporting a dangling-else problem. At the other end of the spectrum, the Perl grammar makes heavy use of Yacc's conflict resolution mechanisms to avoid almost *five hundred* shift/reduce

	Java	*Modula-2*	*Modula-2 (ISO)*
Grammar	Java 1.1	m2c[a] Derivative	ISO/IEC IS 10514
BNF Rules	350	249	388
LALR(1) Conflicts[b]	0	0	2 s/r, 21 r/r
Corpus Size (Files)	3350	607	607
Corpus Sources	JDK 1.2.2	Coco/R 1.5	Coco/R 1.5
	Java Cup v10j	PMOS 2.2	PMOS 2.2
		Ulm Modula-2 3.0b8	Ulm Modula-2 3.0b8
	Oberon-2	*Perl*	*Python*
Grammar	[73]	Perl 5.6.0	Python 1.5.2
BNF Rules	197	189	271
LALR(1) Conflicts	3 s/r	485 s/r	10 s/r
Corpus Size (Files)	580	645	1039
Corpus Sources	OOC 001009	BioPerl 0.6.2	Python 1.5.2
	VisualOberon 001115	Catalog 1.02	Grail 0.6
		FreeWRL 0.14	
		Perl 5.6.0	

[a]By C. Boldyreff, in the `comp.compilers` archive.
[b]"s/r" stands for a shift/reduce conflict, "r/r" for a reduce/reduce conflict.

Table 6.1: Grammar and corpora characteristics.

conflicts (we retained all these conflicts in the Perl grammar when we ran our experiments). The grammars and their corpora are summarized in Table 6.1.

As shown in Table 6.2, 69% of the Earley sets for the corpora of each language contained final items, on average. This represents the absolute best case: even using an oracle, actions could only be executed at these points. The rather high percentage for Modula-2 reflects a large number of ϵ-rules in key places: statically, the Modula-2 grammar contains 34 ϵ-rules compared to only seven for the ISO Modula-2 grammar.

Of those Earley sets with final items, how many were safe sets? Table 6.3 answers this question. Unfortunately, it raises others. We had expected Perl's ambiguous

	Total Sets	*Sets with Final Items*	*%*
Java	2390614	1415818	59
Modula-2	506789	462955	91
Modula-2 (ISO)	506789	363262	72
Oberon-2	722371	482049	67
Perl	3590153	2180437	61
Python	1074560	675859	63

Table 6.2: Earley sets containing final items.

	Sets with Final Items	*Safe Sets*	*%*
Java	1415818	919965	65
Modula-2	462955	62303	13
Modula-2 (ISO)	363262	54674	15
Oberon-2	482049	102370	21
Perl	2180437	682709	31
Python	675859	147979	22

Table 6.3: Safe sets.

grammar to present the biggest challenge to safe set detection, yet we found the second-highest percentage of safe sets for Perl! We conjecture that this may be due to the grammar structure of the various languages, but we have no conclusive proof of this as yet.

The mean window size is a measure which indicates how often, on average, we would be able to execute semantic actions using our technique. As Table 6.4 shows, this number is quite low for Java, Oberon-2, and Perl; the result is tolerable for the other languages. The consistency of the results for like grammars is interesting to note for the two Modula-2 grammars. We noticed the same phenomenon for the Python grammar and a transformed version of it (which we will describe shortly) that had a mean window size of 30.0.

	Mean Window Size
Java	4.1
Modula-2	39.2
Modula-2 (ISO)	39.4
Oberon-2	11.9
Perl	7.5
Python	26.7

Table 6.4: Mean window size.

	Mean Set Retention (%)	*Mean Item Retention (%)*
Java	12	13
Modula-2	26	26
Modula-2 (ISO)	24	23
Oberon-2	18	18
Perl	59	70
Python	9	9

Table 6.5: Mean set and item retention.

Finally, the mean set retention measures the ratio between Earley sets that cannot be discarded versus the total number of sets, at each point in the parse. Table 6.5 gives the results.

The mean set retention appears to be a good indicator of memory savings by the Earley parser. The memory savings by deleting Earley sets during parsing is realized, ultimately, by the fact that there are fewer Earley items in memory. Therefore, we have also gathered data on mean *item* retention, the number of Earley items kept at each point in the parse — it correlates well with mean set retention. We caution, however, that the mean item retention may vary depending on the parser. For example, techniques such as prediction lookahead, Earley set compression, and implementing Earley sets using LR states can all affect the number of items in an

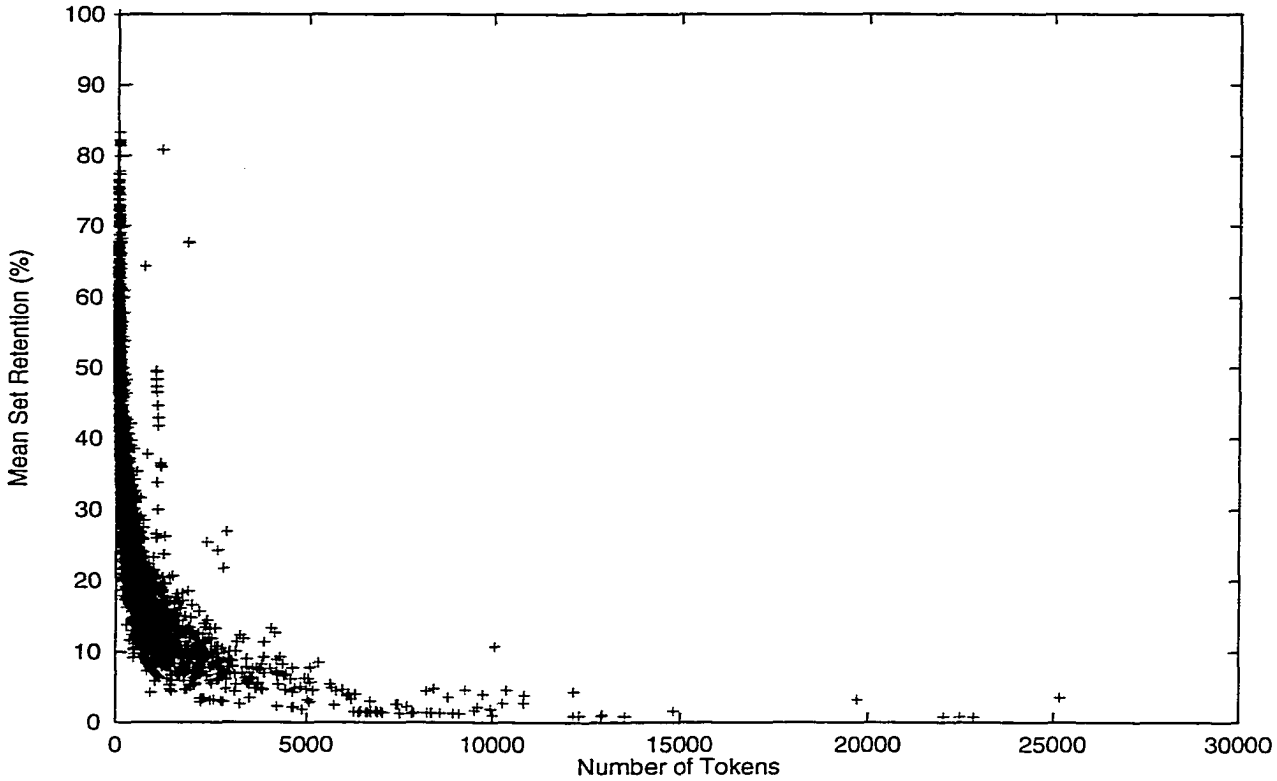
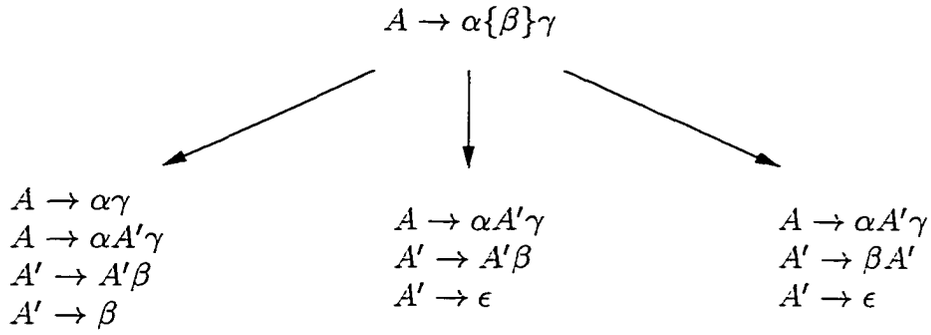


Figure 6.5: Mean set retention and input size.

Earley set. (We used an unadulterated Earley parser when gathering mean item retention data.) We observe that employing one of the above techniques and pruning out some Earley items may cause more safe sets to emerge.

For Java, the mean set retention means that, on average, we can delete 88% of Earley sets during parsing. Comparing this to the number of tokens in the Java input, Figure 6.5 shows that the space savings increase rapidly with the size of the input.

One nagging question: why is the mean set retention so high for Perl? We hypothesize that this is due to the use of right-recursive rules in the Perl grammar, that



a) Left recursive, no ϵ-rules b) Left recursive c) Right recursive

Figure 6.6: Converting EBNF iteration into BNF.

is, rules of the form $A \rightarrow \alpha A$. Such rules are discouraged in grammars for LR-family parsers because they force the parser to keep far more information on its stack than the equivalent left-recursive rules.

To demonstrate the ill effects caused by right recursive rules, we performed an experiment. The Python grammar, as distributed, is written using extended BNF rules (EBNF) [100] which we convert automatically to BNF via a Python script. The EBNF iteration construct, denoted by braces, is used throughout the Python grammar as a notational shorthand: $A \rightarrow \alpha\{\beta\}\gamma$ means that zero or more occurrences of β are allowed. This construct may be translated into BNF in at least three ways, shown in Figure 6.6; we used the form in Figure 6.6a to produce the above results.

We generated two additional grammars for Python by applying the different conversions for iteration. Table 6.6 gives the characteristics of these grammars, repeating the previous Python grammar specifics for comparison purposes. Using these grammars to parse the Python corpus clearly shows the dramatic negative effect caused

	Python	*Python (lrec)*	*Python (rrec)*
Grammar	Python 1.5.2	Python 1.5.2	Python 1.5.2
BNF Rules	271	215	215
LALR(1) Conflicts	10 s/r	6 s/r	44 s/r
Iteration Conversion	a	b	c

Table 6.6: Flavors of Python grammar.

	Python	*Python (lrec)*	*Python (rrec)*
Mean Window Size	26.7	30.0	84.7
Mean Set Retention (%)	9	12	92
Mean Item Retention (%)	9	12	95

Table 6.7: Python grammar results.

by right recursion (Table 6.7), confirming our hypothesis.

6.4 Previous Work

The only prior work in this area seems to have been by Earley himself [30]. He described a means of determining if a final item belongs to the input's derivation, using k symbols of lookahead in the best case, an elaborate bookkeeping scheme in the worst case. Reclamation of space was treated as a separate issue: Earley gave criteria for determining when a given item is no longer necessary, and proposed that a garbage collector could be built on this basis (he never did so, however). In contrast to Earley's proposals, our method is simple and unifies both areas.

6.5 Future Work

The problem remaining with construction of partial parse trees is one of usability. If we incorporated this scheme into SPARK's parser, the user would have no concrete idea as to when their semantic actions would be executed. This can be problematic — for example, say that variable information needs to be entered into a symbol table immediately after a declaration is parsed. How can the user verify that this will happen, and will continue happening even if the grammar is changed? One possible approach would be to devise some way that the user can mark such critical spots in the grammar. Another way would be to separate semantic concerns from parsing entirely, and move to a more powerful formalism.

One such formalism is attribute grammars [57]. In an attribute grammar, semantic actions are associated with grammar rules. These semantic actions are not arbitrary code, though — they specify the computation of attributes in such a way that they can be understood by an "attribute evaluator." This evaluator computes dependencies amongst attributes, and orders their computation appropriately. Using attribute grammars in SPARK would effectively remove the need for semantic analysis using GenericASTTraversal, which can be seen as a crude, manual way of accomplishing the same task.

Returning to the future, one straightforward extension to the work in this chapter would be to employ Follow set information when looking for a safe Earley set. The Follow set of a nonterminal A, $Follow(A)$, is the set of all terminals a such that $S \Rightarrow^* \alpha A a \beta$ [3]. We speculate that this might give the same improvement as SLR(1)

parsing (which uses Follow sets) does over LR(0) parsing (which doesn't).

6.6 Summary

A safe Earley set exhibits four properties which are easy to check for during the parser's operation. Discovering a safe set allows partial parse trees to be constructed during recognition of the input, and also identifies Earley sets that can be deleted. Our results on programming language grammars show that this last point can result in an enormous space savings.

Chapter 7

Running Earley on Empty

It is embarrassing to admit that a bug deserves credit for the work in this chapter.

Like most parsing algorithms, Earley parsers suffer from additional complications when handling grammars containing ϵ-rules, i.e., rules of the form $A \rightarrow \epsilon$ which crop up frequently in grammars of practical interest. Earley's descriptions of his algorithm carefully describe the problem, which we just as carefully ignored when implementing SPARK's Earley parser. Failing to handle ϵ-rules properly causes an Earley parser to erroneously reject certain inputs — surely enough to raise the flag of suspicion. Yet this bug in SPARK sat unnoticed over a year, through three public releases!

Subsequent testing showed that it was remarkably difficult to trigger this bug under normal circumstances. This led us to study the nature of the interaction between Earley parsing and ϵ-rules more closely, and arrive at a straightforward remedy to the problem.

7.1 The Problem of ϵ

In terms of implementation, Earley sets are built in increasing order as the input is read. Also, each set is typically represented as a list of items, as suggested by Earley [30, 31]. This set representation is particularly convenient, because the list of items acts as a "work queue" when building the set: items are examined in order, applying SCANNER, PREDICTOR, and COMPLETER as necessary; items added to the set are appended onto the end of the list.

At any given point i in the parse, then, we have two partially-constructed sets. SCANNER may add items to S_{i+1}, and S_i may have items added to it by PREDICTOR and COMPLETER. It is this latter possibility, adding items to S_i, which causes grief with ϵ-rules.

When COMPLETER processes an item $[A \rightarrow \bullet, j]$ which corresponds to the ϵ-rule $A \rightarrow \epsilon$, it must look through S_j for items with the dot before an A. Unfortunately, for ϵ-rule items, j is always equal to i — COMPLETER is thus looking through the partially-constructed set S_i.[24] Since implementations process items in S_i in order, if an item $[B \rightarrow \cdots \bullet A \cdots, k]$ is added to S_i after COMPLETER has processed $[A \rightarrow \bullet, j]$, COMPLETER will never add $[B \rightarrow \cdots A \bullet \cdots, k]$ to S_i. In turn, items resulting directly and indirectly from $[B \rightarrow \cdots A \bullet \cdots, k]$ will be omitted too. This effectively prunes potential derivation paths, which can cause correct input to be rejected. Figure 7.1 gives an example of this happening.

Two methods of handling this problem have been proposed. Grune and Jacobs

[24] $j = i$ for ϵ-rule items because they can only be added to an Earley set by PREDICTOR, which always bestows added items with the parent pointer i.

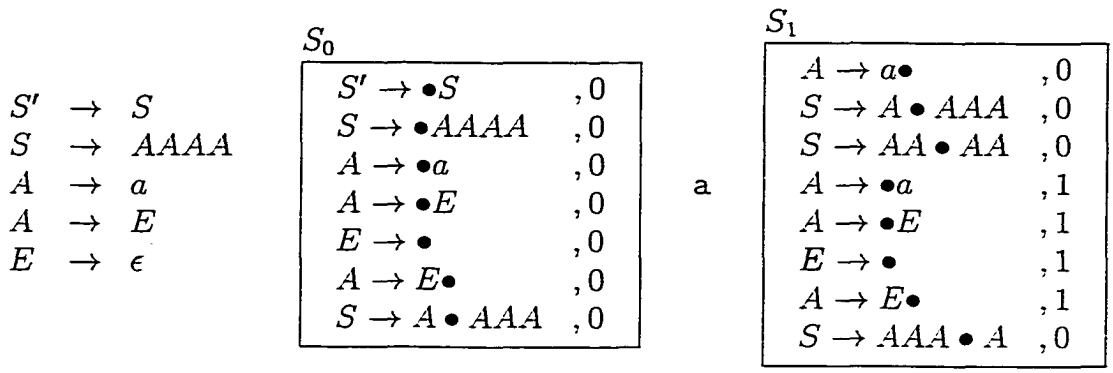


Figure 7.1: An unadulterated Earley parser rejects the valid input a. Missing items in S_0 sound the death knell for this parse.

aptly summarize one approach:

> 'The easiest way to handle this mare's nest is to stay calm and keep running the Predictor and Completer in turn until neither has anything more to add.' [46, page 159]

Aho and Ullman [4] specify this method in their presentation of Earley parsing, and it is used by ACCENT [84], a compiler-compiler which generates Earley parsers.

The other approach was suggested by Earley [30, 31]. He proposed having COMPLETER note that the dot needed to be moved over A, then looking for this whenever future items were added to S_i. For efficiency's sake, the collection of nonterminals to watch for should be stored in a data structure which allows fast access. We used this method initially to fix the bug in SPARK.

In our opinion, neither approach is very satisfactory. Repeatedly processing S_i, or parts thereof, involves a lot of activity for little gain. It also complicates our directly-executable Earley parser, which expects to process the contents of S_i in one

pass. On the other hand, Earley's solution requires an extra, dynamically-updated data structure and the unnatural mating of COMPLETER with the adding of items. Ideally, we want a solution which retains the elegance of Earley's algorithm, only processes items in S_i once, and has no run time overhead from updating a data structure.

7.2 An "Ideal" Solution

Our solution involves a simple modification to PREDICTOR, based on the idea of nullability. A nonterminal A is said to be nullable if $A \Rightarrow^* \epsilon$; terminal symbols, of course, can never be nullable. The nullability of nonterminals in a grammar may be easily precomputed using well-known techniques [6, 37]. Using this notion, our PREDICTOR can be stated as follows (our modification is in boldface):

> If $[A \rightarrow \cdots \bullet B \cdots, j]$ is in S_i, add $[B \rightarrow \bullet\alpha, i]$ to S_i for all rules $B \rightarrow \alpha$.
> **If B is nullable, also add $[A \rightarrow \cdots B \bullet \cdots, j]$ to S_i.**

In other words, we eagerly move the dot over a nonterminal if that nonterminal can derive ϵ and effectively "disappear." Using the grammar from the ill-fated Figure 7.1, Figure 7.2 demonstrates how our modified PREDICTOR fixes the problem.

7.3 Proof of Correctness

Our solution is correct in the sense that it produces exactly the same items in an Earley set S_i as would Earley's method of handling ϵ-rules. In the following proof,

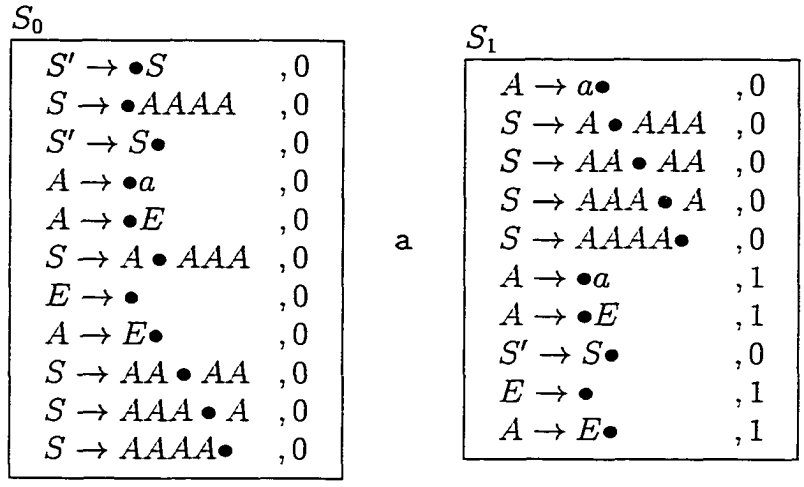


Figure 7.2: An Earley parser accepts the input a, using our modification to PREDICTOR.

we write $E(S_i)$ and $E'(S_i)$ to denote the contents of Earley set S_i as computed by Earley's method and our method, respectively. Earley's SCANNER, PREDICTOR, and COMPLETER steps are denoted by E_S, E_P, and E_C; ours are E'_S, E'_P, and E'_C.

We begin with some results which will be used later:

Lemma 2 *Let I be the Earley item $[A \rightarrow \alpha\bullet, i]$. If $I \in S_i$, then $\alpha \Rightarrow^* \epsilon$.*

Proof. There are two cases, depending on the length of α:

Case 1. $|\alpha| = 0$. The lemma is trivially true, because α must be ϵ.

Case 2. $|\alpha| > 0$. The key observation here is that the parent pointer of an Earley item indicates the Earley set where the item first appeared. In other words, the item $[A \rightarrow \bullet\alpha, i]$ must also be present in S_i, and because both it and I are in S_i, it means that α must have made its debut *and* been recognized without consuming any terminal symbols. Therefore $\alpha \Rightarrow^* \epsilon$.

□

Lemma 3 *If* $[B \rightarrow \bullet\alpha, i] \in E'(S_i)$ *and* $\alpha \Rightarrow^* \epsilon$, *then* $[B \rightarrow \alpha\bullet, i]$ *will be added to* $E'(S_i)$ *during processing.*

Proof. We again look at the length of α:

Case 1. $|\alpha| = 0$. True, because $\alpha = \epsilon$.

Case 2. $|\alpha| > 0$. Let $m = |\alpha|$. Then $\alpha = X_1 X_2 \cdots X_m$. Because $\alpha \Rightarrow^* \epsilon$, none of the constituent symbols of α can be terminals. Furthermore, $X_l \Rightarrow^* \epsilon, 1 \leq l \leq m$. When processing $[B \rightarrow \bullet X_1 X_2 \cdots X_m, i]$, E'_P would add $[B \rightarrow X_1 \bullet X_2 \cdots X_m, i]$, whose later processing by E'_P would add $[B \rightarrow X_1 X_2 \bullet \cdots X_m, i]$, and so on until $[B \rightarrow X_1 X_2 \cdots X_m \bullet, i]$ — also known as $[B \rightarrow \alpha\bullet, i]$ — had been added to $E'(S_i)$.

□

Next, we establish containment properties of $E(S_i)$ and $E'(S_i)$. Our approach works backwards in a way. We pick some arbitrary Earley item I produced by one algorithm, and determine the preconditions which must have existed for I to be present. Assuming those same preconditions, we show that the other algorithm must also produce I. We assume that Earley sets $S_0, S_1, \ldots, S_{i-1}$ are identical for both algorithms, and that set S_i is the first set in which $E(S_i)$ and $E'(S_i)$ differ.

Lemma 4 $E(S_i) \subseteq E'(S_i)$.

Proof. Assume that there is some Earley item $I \in E(S_i)$ which is not contained in $E'(S_i)$. We can classify I according to the position of the dot:

Case 1. The dot is at the beginning. I must be of the form $[A \rightarrow \bullet\alpha, i]$ (this includes ϵ-rules where $\alpha = \epsilon$). If I is the initial item $[S' \rightarrow \bullet S, 0]$, then I is definitely present in $E(S_i)$ and $E'(S_i)$. Otherwise, I can only have been added to $E(S_i)$ by E_P, which must have processed some earlier item $I' \in E(S_i)$, where $I' = [B \rightarrow \cdots \bullet A \cdots, j]$. However, both E_P and E'_P add I when processing I', so if $I' \in E'(S_i)$, then I must be in both $E(S_i)$ and $E'(S_i)$.

Case 2. The dot is not at the beginning, and there is a terminal symbol to the left of the dot. Here, I must look like $[A \rightarrow \cdots a \bullet \cdots, j]$, and must have been added as a results of E_S processing Earley set S_{i-1}. But because E_S is the same as E'_S, I must again be in both $E(S_i)$ and $E'(S_i)$.

Case 3. The dot is not at the beginning, and there is a nonterminal symbol to the left of the dot. This is the most interesting case. I must be $[A \rightarrow \cdots B \bullet \cdots, k]$, and must have arisen by E_C processing an earlier item in $E(S_i)$, $I' = [B \rightarrow \alpha\bullet, j]$. If $j \neq i$, then I would have to be in $E(S_i)$ and $E'(S_i)$; E_C and E'_C operate the same way on S_j.

If $j = i$ instead, then $\alpha \Rightarrow^* \epsilon$ by Lemma 2. Consequently, the item $I'' = [A \rightarrow \cdots \bullet B \cdots, k]$ is also in $E(S_i)$. This is mirrored in our algorithm: if $I'' \in E'(S_i)$, then E'_P will add the item $[B \rightarrow \bullet\alpha, i]$, which will eventually cause I' to be added to $E'(S_i)$ by Lemma 3.

Now consider the order in which I'' and I' appear:

Case 3a. I'' is present when I' is processed. In this case, E_C and E'_C work the same way, and I is added to both $E(S_i)$ and $E'(S_i)$.

Case 3b. I'' is not present when I' is processed. In E, the fact that the dot needs to be moved over B would have been dynamically recorded; I would be added when I'' is processed. The dot would be moved in E' when processing I'' also, by E'_P, because B is nullable.

$E(S_i) \subseteq E'(S_i)$ by contradiction, since no I can be chosen which is in $E(S_i)$ but not in $E'(S_i)$. □

Lemma 5 $E(S_i) \supseteq E'(S_i)$.

Proof. We take the same tack as before: posit the existence of an Earley item $I \in E'(S_i)$ which is not in $E(S_i)$. We have the same three cases when examining the position of I's dot — the first two cases are identical, save for exchanging "E" and "E'" in the text, and they have been elided.

Case 3. The dot is not at the beginning, and there is a nonterminal symbol to the left of the dot. I must be $[A \rightarrow \cdots B \bullet \cdots, k]$, and was added to $E'(S_i)$ one of two ways. Looking at the nullability of B:

Case 3a. B is not nullable. I is thus added by E'_C processing $I' = [B \rightarrow \alpha\bullet, j] \in E'(S_i)$. As B is not nullable, $\alpha \neq \epsilon$ and therefore at least one terminal symbol is consumed: $j \neq i$. If $I' \in E(S_i)$, then I must be in $E(S_i)$ because E_C and E'_C work identically on S_j.

Case 3b. B is nullable. One possibility is that I may be added by E'_C processing I'. If $j \neq i$, then this degenerates to Case 3a above. If $j = i$ instead, then $I'' = [A \rightarrow \cdots \bullet B \cdots, k] \in E'(S_i)$; this would also be the situation if I

had been added the alternate way, by E'_P. If I'' were in $E(S_i)$, E_P would add $[B \rightarrow \bullet\alpha, i]$ to $E(S_i)$, among others. I' must inevitably be added to $E(S_i)$ too (if not, then Earley's algorithm would have failed by omitting the derivation $B \Rightarrow^* \epsilon$), processing of which would cause I to be added to $E(S_i)$.

As before, no I can be chosen which is in $E'(S_i)$ but not in $E(S_i)$, proving $E(S_i) \supseteq E'(S_i)$ by contradiction. □

Theorem 3 $E(S_i) = E'(S_i)$.

Proof. This is a direct result of Lemmas 4 and 5. Our solution thus produces the same result as Earley's. □

We note that this proof holds even if the grammar contains "useless" nonterminal symbols [46], that cannot derive a string of terminal symbols. Useless nonterminals should be considered not nullable, as they can never derive ϵ, and are therefore not a party to any issues Earley's parsing algorithm has with ϵ-rules.

7.4 Precomputation and Representation

The Earley items added by our modified PREDICTOR can be precomputed. Moreover, this precomputation can take place in such a way as to produce a new variant of LR(0) DFA which is very well suited to Earley parsing.

Figure 7.3 shows the LR(0) automaton for the grammar in Figure 7.1. Recall that in Section 4.1.5, we gave a way to split the LR(0) states to create our LR(0) $\overline{\text{DFA}}$, a convenient structuring of the state machine for use in an Earley parser.

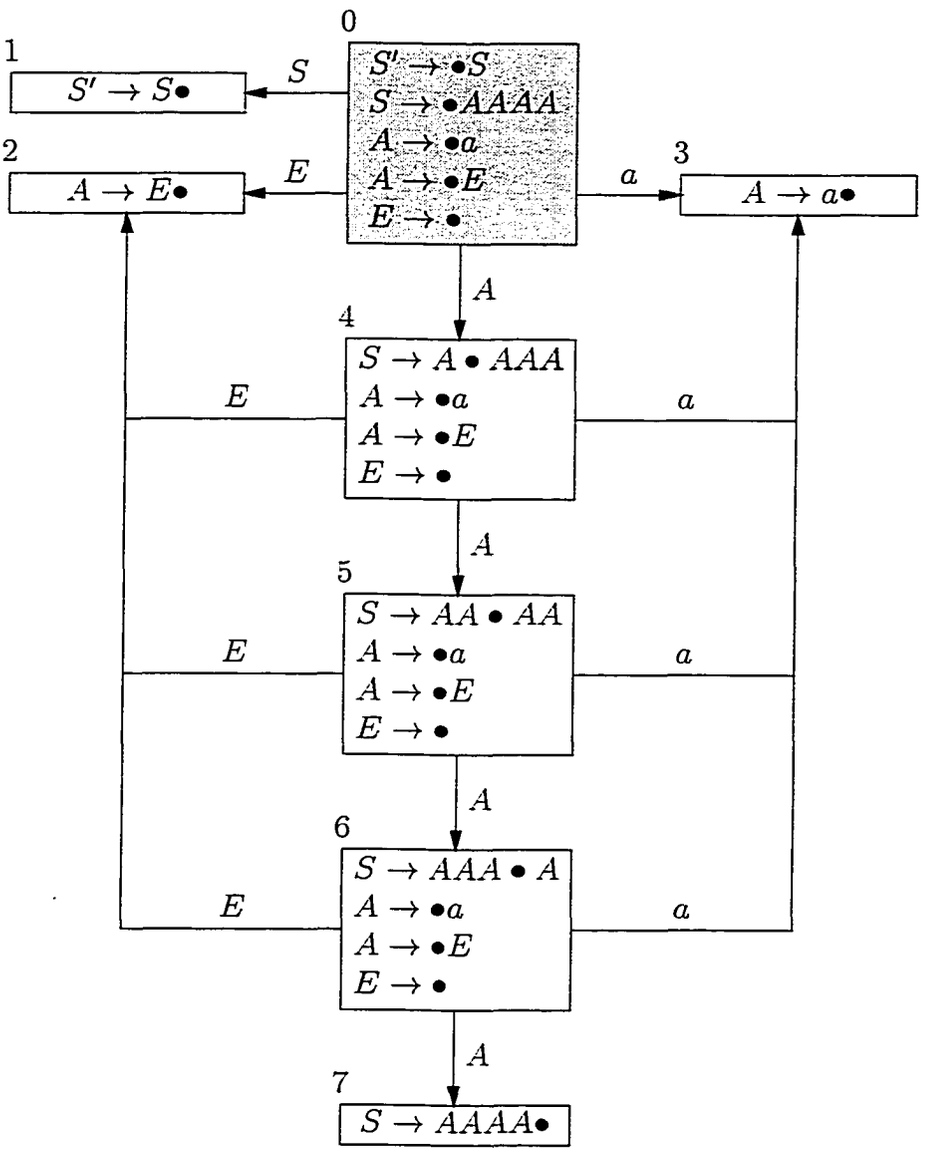


Figure 7.3: LR(0) automaton for the grammar in Figure 7.1.

We observe that some $\overline{\text{DFA}}$ states must always appear together in an Earley set. This idea is captured in the theorem below; we present the theorem in terms of the LR(0) DFA rather than our $\overline{\text{DFA}}$ for the time being. The function $GOTO(L, A)$ returns the LR(0) state reached if a transition on A is made from state L.

Theorem 4 *If an LR(0) item $l = [A \rightarrow \bullet\alpha]$ is contained in LR(0) state $L \in S_i$ and $\alpha \Rightarrow^* \epsilon$, then $GOTO(L, A)$ must also be in S_i.*

Proof. As part of an Earley item, l must have the parent pointer i because the dot is at the beginning of the item. By Lemma 3, the Earley item $I = [A \rightarrow \alpha\bullet, i]$ will be added to S_i. As a result of COMPLETER processing of I (whose parent pointer is i, making COMPLETER look "back" at the current set S_i), transitions will be attempted on A for every LR(0) state in S_i. Thus $GOTO(L, A)$ must be added to S_i. □

We can treat Theorem 4 as the basis of a closure algorithm, combining LR(0) states that always appear together, iterating until no more state mergers are possible. For the LR(0) automaton in Figure 7.3, the resulting "LR(0) ϵ-DFA" states would be

$\{0, 1, 2, 4, 5, 6, 7\}$	$\{1\}$
$\{4, 2, 5, 6, 7\}$	$\{2\}$
$\{5, 2, 6, 7\}$	$\{3\}$
$\{6, 2, 7\}$	$\{7\}$

The LR(0) ϵ-DFA is drawn in Figure 7.4. Of course, the ϵ-DFA states can be split into kernel and nonkernel items as in Section 4.1.5, to form an LR(0) ϵ-$\overline{\text{DFA}}$.

Pseudocode for an Earley parser which would use the LR(0) ϵ-$\overline{\text{DFA}}$ is given in Figure 7.5. This code assumes that Earley items in S_i and S_{i+1} are implemented as worklists, and that the items themselves are (state, parent) pairs. The `goto` function returns the state transitions made in the split ϵ-DFA, given a state and grammar

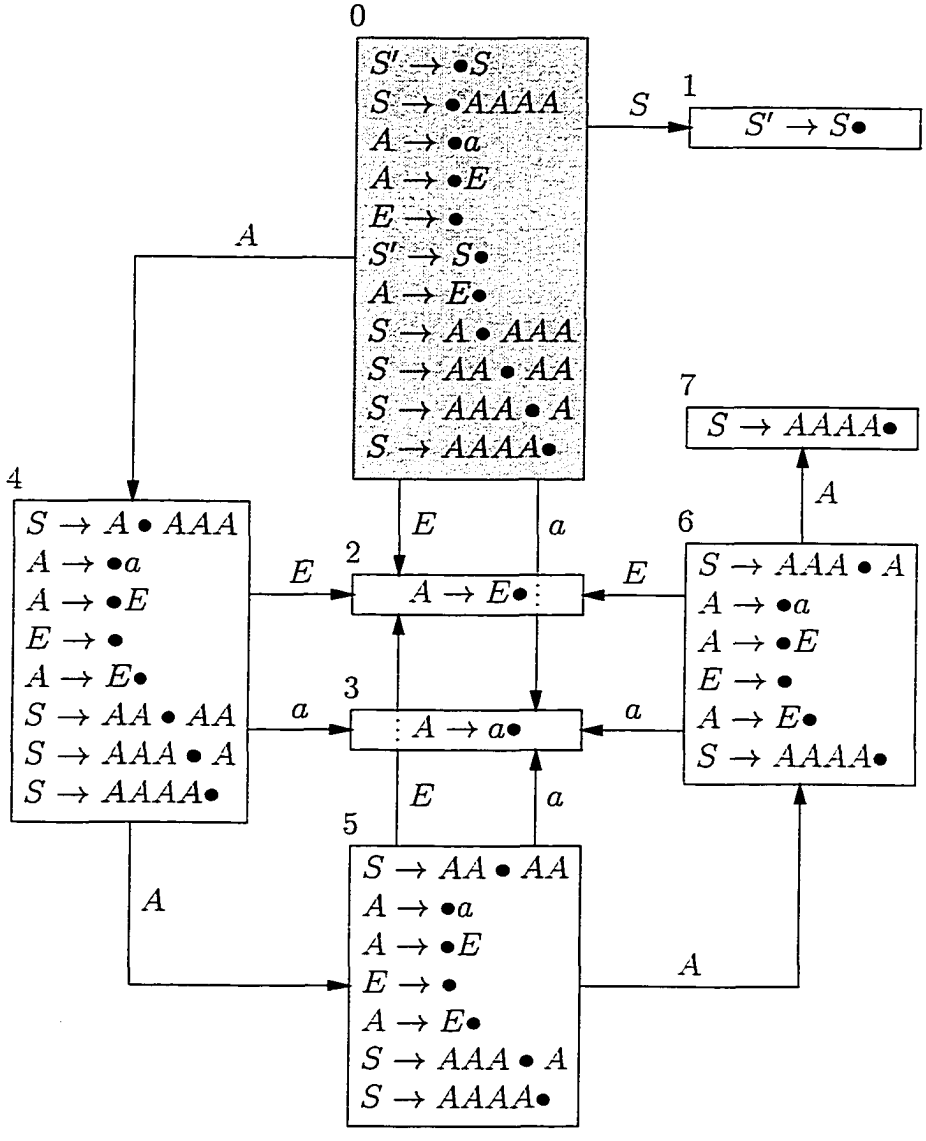


Figure 7.4: LR(0) ϵ-DFA.

```
foreach (state, parent) in S_i:
    (k, nk) ← goto(state, x_{i+1})
    if k ≠ Λ:
        add (k, parent) to S_{i+1}
        if nk ≠ Λ:
            add (nk, i+1) to S_{i+1}

    if parent = i:
        continue

    foreach A → α in completed(state):
        foreach (pstate, pparent) in S_parent:
            (k, nk) ← goto(pstate, A)
            if k ≠ Λ:
                add (k, pparent) to S_i
                if nk ≠ Λ:
                    add (nk, i) to S_i
```

Figure 7.5: Pseudocode for processing Earley set S_i using an LR(0) ϵ-$\overline{\text{DFA}}$.

symbol. There can be at most two of these transitions — one to a kernel state k, one to a nonkernel state nk — and the absence of a transition is denoted by the value Λ. The completed function supplies a list of grammar rules completed within a given state. Recall that an Earley item is only added to an Earley set if it is not already present.

Can the original problem with empty rules recur when using the ϵ-DFA in an Earley parser? Happily, it can't. Recall that the problem with which we began this chapter was the addition of an Earley item $[B \rightarrow \cdots \bullet A \cdots, k]$ to S_i *after* COMPLETER processed $[A \rightarrow \bullet, i]$: the dot never got moved over the A even though $A \Rightarrow^* \epsilon$. With the ϵ-DFA, say that state l contains the troublesome item $[B \rightarrow \cdots \bullet A \cdots]$. All items $[A \rightarrow \bullet\alpha]$ must be in l too. If A is nullable, then there has to be some value of α such

that $A \Rightarrow \alpha \Rightarrow^* \epsilon$, and the dot must be moved over the A in state l by Theorem 4.

7.5 Summary

Implementations of Earley's parsing algorithm can easily handle ϵ-rules using the simple modification to PREDICTOR outlined here. Precomputation yields a new variant of LR(0) state machine suitable for use in DEEP and other Earley parsers based on finite automata.

Chapter 8

Conclusion

The lines of inquiry we have pursued over the last few years first presented themselves in a serendipitous manner. SPARK was created to fill a need unrelated to our research; Earley's algorithm best matched our design constraints for SPARK; our experience using SPARK highlighted the problems with Earley's algorithm and brought a number of research problems to bear.

The contributions of our work are sevenfold:

1. We demonstrated how to construct a directly-executable Earley parser, the first of its kind. The performance of the resulting parsers works in time comparable to that of the much-less general LALR(1) algorithm, even on non-toy grammars. This is particularly important as it suggests that direct execution may be viable for other types of general parsers.

2. For Earley parsers that use LR automata, we gathered data which indicates that prediction lookahead is not as valuable a strategy as our Earley set compression.

This prunes unneeded Earley items so that the parser uses less space at run time.

3. A technique for handling context-dependent tokens — Schrödinger's tokens — was presented, along with applications of the technique. Some awkward real languages, like PL/I, can be handled with this technique. Schrödinger's tokens are not limited to Earley parsing, but can be used with any general parsing algorithm. Our experience using Schrödinger's tokens in SPARK has been very positive.

4. We specified conditions under which semantic actions can be executed in an Earley parser prior to the completion of recognition. While the practical value of this to the end user has not yet been ascertained, we showed that the related space savings could be quite remarkable — over 90% in some cases.

5. We discovered a simple, efficient way to handle the problem with empty rules in Earley parsers, and proved it correct. Our solution to the problem lets Earley sets be processed in one pass, with no dynamic modification of data structures, giving a faster and more robust implementation.

6. A novel, easy-to-use representation for Earley sets was given, for use with Earley parsers that employ LR(0) automata. This permits simpler implementation of automata-based Earley parsers.

7. We gave a way of transforming a standard LR(0) automaton to make it more useful in an Earley parser, which can be combined with the above-mentioned representation. Again, this produces a simpler, more efficient implementation

of Earley's algorithm.

We think that much of this work can be expanded to other types of general parsing algorithms. Of particular interest is the possibility of constructing other types of directly-executable general parsers. Also, we would like to find other new automata that are tailored for general parsers, whose use simplifies the parsing algorithm.

When Earley wrote his Ph.D. over thirty years ago, computing conditions were very different than they are now. Unfortunately, Earley's algorithm seems to be synonymous with "slow" insofar as the programming language community is concerned. We hope that our work helps to dispel this notion, and that it stimulates some renewed interest in general parsing techniques.

In retrospect, what we find interesting from a research point of view was that we were able to take a very general algorithm, address its shortcomings, and make it suitable for practical use. In the presence of our abundance of current computing power, the time may be ripe to re-examine old, powerful, general algorithms that have been left by the wayside.

References

[1] P. W. Abrahams. The CIMS PL/I compiler. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 107–116, August 1979.

[2] A. V. Aho, S. C. Johnson, and J. D. Ullman. Deterministic parsing of ambiguous grammars. *Communications of the ACM*, 18(2):441–452, August 1975.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[4] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation, and Compiling, Volume 1: Parsing.* Prentice-Hall, 1972.

[5] M. A. Alonso, D. Cabrero, and M. Vilares. Construction of efficient generalized LR parsers. In *Proceedings of the Second International Workshop on Implementing Automata*, pages 131–140, 1997.

[6] A. W. Appel. *Modern Compiler Implementation in Java.* Cambridge University Press, 1998.

[7] J. Aycock. Compiling little languages in Python. In *Proceedings of the 7th International Python Conference*, pages 69–77, 1998.

[8] J. Aycock. Aggressive type inference. In *Proceedings of the 8th International Python Conference*, pages 11–20, 2000.

[9] J. Aycock and N. Horspool. Faster generalized LR parsing. In *Proceedings of the 8th International Conference on Compiler Construction*, pages 32–46, 1999.

[10] J. Aycock and N. Horspool. Directly-executable Earley parsing. In *Proceedings of the 10th International Conference on Compiler Construction*, pages 229–243, 2001.

[11] J. Aycock, N. Horspool, J. Janoušek, and B. Melichar. Even faster generalized LR parsing. *Acta Informatica*, 2001. To appear.

[12] J. Aycock and R. N. Horspool. Schrödinger's token. *Software — Practice and Experience*, 31(8):803–814, 2001.

[13] D. T. Barnard and J. R. Cordy. SL parses the LR languages. *Computer Languages*, 13(2):65–74, 1988.

[14] F. L. Bauer and H. Wössner. The "Plankalkül" of Konrad Zuse: A forerunner of today's programming languages. *Communications of the ACM*, 15(7):678–685, July 1972.

[15] D. Beazley. *Python Essential Reference*. New Riders, 1999.

[16] J. R. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, June 1973.

[17] J. Bentley. Little languages. In *More Programming Pearls*, pages 83–100. Addison-Wesley, 1988.

[18] J. Bentley, September 2000. Personal communication.

[19] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext transfer protocol — HTTP/1.0, 1996. RFC 1945.

[20] A. Bhamidipaty and T. A. Proebsting. Very fast YACC-compatible parsers (for very little effort). *Software — Practice and Experience*, 28(2):181–190, February 1998.

[21] S. Billot and B. Lang. The structure of shared forests in ambiguous parsing. In *Proceedings of the 27th Annual Meeting of the Association for Computational Linguistics*, pages 143–151, 1989.

[22] M. Bouckaert, A. Pirotte, and M. Snelling. Efficient parsing algorithms for general context-free parsers. *Information Sciences*, 8:1–26, 1975.

[23] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, 1996.

[24] G. Julius Cæsar. *Cæsar's Commentaries on the Gallic and Civil Wars*. Harper & Brothers, 1855. Translation.

[25] N. Chomsky. On certain formal properties of grammars. *Information and Control*, 2:137–167, 1959.

[26] T. W. Christopher, P. J. Hatcher, and R. C. Kukuk. Using dynamic programming to generate optimized code in a Graham-Glanville style code generator. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pages 25–36, 1984.

[27] C. Clark. Keywords: special identifier idioms. *ACM SIGPLAN Notices*, 34(12):18–23, 1999.

[28] P. F. Dubois. Cafe Dubois. *Computing in Science & Engineering*, 1(6):71, November/December 1999. *Scientific Programming* column.

[29] P. F. Dubois. *Pyfort Reference Manual.* Lawrence Livermore National Laboratory, sixth edition, 2000.

[30] J. Earley. *An Efficient Context-Free Parsing Algorithm.* PhD thesis, Carnegie-Mellon University, August 1968.

[31] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, February 1970.

[32] T. Ernst. The TRAP generic compiler prototyping system. Unpublished paper, 1999.

[33] T. Ernst. TRAPping Modelica with Python. In *Proceedings of the 8th International Conference on Compiler Construction*, pages 288–291, 1999. Tool demo.

[34] T. Ernst, January 2001. Personal communication.

[35] M. A. Ertl and D. Gregg. Hardware support for efficient interpreters: Fast indirect branches (draft), 2000.

[36] S. I. Feldman. Implementation of a portable Fortran 77 compiler using modern tools. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 98–106, 1979.

[37] C. N. Fischer and R. J. LeBlanc, Jr. *Crafting a Compiler.* Benjamin/Cummings, 1988.

[38] G. A. Fisher, Jr. and M. Weber. LALR(1) parsing for languages without reserved words. *ACM SIGPLAN Notices*, 14(11):26–30, 1979.

[39] C. W. Fraser, D. R. Hanson, and T. A. Proebsting. Engineering a simple, efficient code generator generator. In *ACM Letters on Programming Languages and Systems*, volume 1(3), pages 213–226, 1992.

[40] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison-Wesley, 1995.

[41] W. K. Giloi. Konrad Zuse's Plankalkül: The first high-level, "non von Neumann" programming language. *IEEE Annals of the History of Computing*, 19(2):17–24, 1997.

[42] R. S. Glanville and S. L. Graham. A new method for compiler code generation. In *5th Annual ACM Symposium on Principles of Programming Languages*, pages 231–240, 1978.

[43] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[44] S. L. Graham, A. M. Harrison, and W. L. Ruzzo. An improved context-free recognizer. *ACM Transactions on Programming Languages and Systems*, 2(3):415–462, 1980.

[45] D. Grune, H. E. Bal, C. J. H. Jacobs, and K. G. Langendoen. *Modern Compiler Design.* Wiley, 2000.

[46] D. Grune and C. J. H. Jacobs. *Parsing Techniques: A Practical Guide.* Ellis Horwood, 1990.

[47] J. Heering, P. Klint, and J. Rekers. Incremental generation of parsers. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 179–191, 1989.

[48] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[49] R. N. Horspool, 2000. Personal communication.

[50] R. N. Horspool and M. Whitney. Even faster LR parsing. *Software — Practice and Experience*, 20(6):515–535, 1990.

[51] IEEE. *IEEE Standard for Information Technology — Portable Operating System Interface (POSIX) — Part 2: Shell and Utilities*, 1993. Volumes 1 and 2.

[52] International Telecommunication Union. *Specification of Abstract Syntax Notation One (ASN.1)*, 1993.

[53] S. C. Johnson. YACC — yet another compiler compiler. *UNIX Programmer's Manual, 7th Edition*, 2B, 1978.

[54] J. Kanze. Handling ambiguous tokens in LR-parsers. *ACM SIGPLAN Notices*, 24(6):49–54, 1989.

[55] P. Klint. Interpretation techniques. *Software — Practice and Experience*, 11:963–973, 1981.

[56] P. Klint and E. Visser. Using filters for the disambiguation of context-free grammars. In *Proceedings of the ASMICS Workshop on Parsing Theory*, pages 1–20, 1994.

[57] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968. Errata in [58].

[58] D. E. Knuth. Semantics of context-free languages: Correction. *Mathematical Systems Theory*, 5(1):95–96, 1971.

[59] D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching.* Addison-Wesley, second edition, 1998.

[60] D. E. Knuth and L. T. Pardo. The early development of programming languages. In N. Metropolis, J. Howlett, and G.-C. Rota, editors, *A History of Computing in the Twentieth Century*, pages 197–273. Academic Press, 1980.

[61] R. Koppler. A systematic approach to fuzzy parsing. *Software — Practice and Experience*, 27(6):637–649, 1997.

[62] D. C. Kozen. *Automata and Computability.* Springer-Verlag, 1997.

[63] B. Lang. Deterministic techniques for efficient non-deterministic parsers. In J. Loeckx, editor, *Automata, Languages, and Programming (LNCS #14)*, pages 255–269. Springer-Verlag, 1974.

[64] R. Leermakers. A recursive ascent Earley parser. *Information Processing Letters*, 41:87–91, 1992.

[65] R. Leermakers. Recursive ascent parsing: from Earley to Marcus. *Theoretical Computer Science*, 104:299–312, 1992.

[66] R. Leermakers. *The Functional Treatment of Parsing.* Kluwer Academic, 1993.

[67] M. E. Lesk and E. Schmidt. Lex — a lexical analyzer generator. *UNIX Programmer's Manual, 7th Edition*, 2B, 1975.

[68] J. R. Levine, T. Mason, and D. Brown. *Lex & Yacc.* O'Reilly & Associates, second edition, 1992.

[69] M. R. Levy. Web programming in Guide. *Software — Practice and Experience*, 28(11):1581–1603, 1998.

[70] H. R. Lewis and C. H. Papadimitriou. *Elements of the Theory of Computation.* Prentice-Hall, 1981.

[71] P. McLean and R. N. Horspool. A faster Earley parser. In *Proceedings of the International Conference on Compiler Construction (CC '96)*, pages 281–293, 1996.

[72] R. Milne. Lexical ambiguity resolution in a deterministic parser. In S. I. Small, G. W. Cottrell, and M. K. Tanenhaus, editors, *Lexical Ambiguity Resolution*, pages 45–71. Morgan Kaufmann, 1988.

[73] H. Mössenböck. *Object-Oriented Programming in Oberon-2.* Springer-Verlag, 1993.

[74] M. E. Nordberg III. Variations on the visitor pattern. In *Collected Papers from the PLoP '96 and EuroPLoP '96 Conferences.* Washington University, 1997. Technical Report WUCS-97-07.

[75] J. K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[76] T. J. Pennello. Very fast LR parsing. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, volume 21(7) of *ACM SIGPLAN Notices*, pages 145–151, 1986.

[77] P. Pfahler. Optimizing directly executable LR parsers. In *Compiler Compilers, Third International Workshop, CC '90*, pages 179–192. Springer-Verlag, 1990.

[78] J. Postel and J. Reynolds. File transfer protocol (FTP), 1985. RFC 959.

[79] J. B. Postel. Simple mail transfer protocol, 1982. RFC 821.

[80] E. S. Raymond. The CML2 language: Python implementation of a constraint-based interactive configurator. In *Proceedings of the 9th International Python Conference*, 2001. To appear.

[81] A. H. J. Sale. The classification of FORTRAN statements. *Computer Journal*, 14(1):10–12, 1971.

[82] D. J. Salomon and G. V. Cormack. Scannerless NSLR(1) parsing of programming languages. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 170–178, 1989.

[83] E. Schrödinger. Die gegenwärtige situation in der quantenmechanik. *Die Naturwissenschaften*, 23:807–812,823–828,844–849, 1935. Translation in Trimmer [90].

[84] F. W. Schröer. The ACCENT compiler compiler, introduction and reference. Technical Report 101, German National Research Center for Information Technology, June 2000.

[85] M. Shaw, January 2001. Personal communication.

[86] O. Shivers. A universal scripting framework. In *Concurrency and Parallelism, Programming, Networking, and Security*, pages 254–265. Springer-Verlag, 1996.

[87] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, third edition, 1997.

[88] J. Tarhio. LR parsing of some ambiguous grammars. *Information Processing Letters*, 14(3):101–103, 1982.

[89] M. Tomita. *Efficient Parsing for Natural Languages.* Kluwer Academic, 1986.

[90] J. D. Trimmer. The present situation in quantum mechanics: a translation of Schrödinger's "cat paradox" paper. *Proceedings of the American Philosophical Society*, 124(5):323–338, 1980.

[91] M. van den Brand, A. Sellink, and C. Verhoef. Current parsing technologies in software renovation considered harmful. In *International Workshop on Program Comprehension*, pages 108–117, 1998.

[92] A. van Deursen. Introducing ASF+SDF using the λ-calculus as example. In *Executable Language Definitions*. PhD thesis, University of Amsterdam, 1994.

[93] M. Vilares Ferro and B. A. Dion. Efficient incremental parsing for context-free languages. In *Proceedings of the 5th IEEE International Conference on Computer Languages*, pages 241–252, 1994.

[94] E. Visser. Scannerless generalized-LR parsing. Technical Report P9707, University of Amsterdam Programming Research Group, 1997.

[95] T. A. Wagner and S. L. Graham. Incremental analysis of real programming languages. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 31–43, June 1997.

[96] L. Wall, T. Chistiansen, and R. L. Schwartz. *Programming Perl*. O'Reilly & Associates, second edition, 1996.

[97] F. W. Weingarten. *Translation of Computer Languages*. Holden-Day, 1973.

[98] R. L. White and P. Greenfield. Using Python to modernize astronomical software. In *Proceedings of the 8th International Python Conference*, pages 103–109, 2000.

[99] T. Winograd. *Understanding Natural Language*. Academic Press, 1972.

[100] N. Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? *Communications of the ACM*, 20(11):822–823, 1977.

[101] World Wide Web Consortium. XML Path Language (XPath), Version 1.0. W3C Recommendation (16 November 1999), J. Clark and S. DeRose, editors.

[102] D. B. Wortman. Compiling at 1000 MHz and beyond. In *Systems Implementation 2000*, pages 194–206. Chapman & Hall, 1998.

Appendix A

Sample SPARK Specification

In Chapter 2 we used a little arithmetic expression language as a running example. This appendix collects the code together for one implementation of that language.

```
import sys
filename = sys.argv[1]
f = open(filename)
evaluate(semantic(parse(scan(f))))
f.close()
```

```
class SimpleScanner(GenericScanner):
    def __init__(self):
        GenericScanner.__init__(self)

    def tokenize(self, input):
        self.rv = []
        GenericScanner.tokenize(self, input)
        return self.rv

    def t_whitespace(self, s):
        r' \s+ '

    def t_op(self, s):
        r' \+ | \* '
        self.rv.append(Token(type=s))

    def t_number(self, s) :
        r' \d+ '
        t = Token(type='number', attr=s)
        self.rv.append(t)


class FloatScanner(SimpleScanner):
    def __init__(self):
        SimpleScanner.__init__(self)

    def t_float(self, s):
        r' \d+ \. \d+ '
        t = Token(type='float', attr=s)
        self.rv.append(t)


def scan(f):
    input = f.read()
    scanner = FloatScanner()
    return scanner.tokenize(input)
```

```
class ExprParser(GenericParser):
    def __init__(self, start='expr'):
        GenericParser.__init__(self, start)

    def p_expr_term(self, args):
        '''
            expr ::= expr + term
            term ::= term * factor
        '''
        return AST(type=args[1], left=args[0], right=args[2])

    def p_expr_term_2(self, args):
        '''
            expr ::= term
            term ::= factor
        '''
        return args[0]

    def p_factor(self, args):
        '''
            factor ::= number
            factor ::= float
        '''
        return AST(type=args[0])


def parse(tokens):
    parser = ExprParser()
    return parser.parse(tokens)
```

```
class TypeCheck(GenericASTTraversal):
    def __init__(self, ast):
        GenericASTTraversal.__init__(self, ast)
        self.postorder()

    def n_number(self, node):
        node.exprType = 'number'

    def n_float(self, node):
        node.exprType = 'float'

    def default(self, node):
        # this handles + and * nodes
        leftType = node.left.exprType
        rightType = node.right.exprType
        if leftType != rightType:
            raise 'Type error.'
        node.exprType = leftType


def semantic(ast):
    TypeCheck(ast)
    #
    #  Any other GenericASTTraversal classes
    #  for semantic checking would be
    #  instantiated here...
    #
    return ast
```

```
class Interpreter(GenericASTMatcher):
    def __init__(self, ast):
        GenericASTMatcher.__init__(self, 'V', ast)
        self.match()
        print ast.value

    def p_number(self, node):
        ' V ::= number '
        node.value = int(node.attr)

    def p_float(self, node):
        ' V ::= float '
        node.value = float(node.attr)

    def p_add(self, node):
        ' V ::= + ( V V ) '
        node.value = node.left.value + node.right.value

    def p_multiply(self, node):
        ' V ::= * ( V V ) '
        node.value = node.left.value * node.right.value


def evaluate(ast):
    Interpreter(ast)
```

Selected Publications

J. Aycock, N. Horspool, J. Janoušek, and B. Melichar. Even Faster Generalized LR Parsing. To appear in *Acta Informatica* (accepted February 2001).

J. Aycock and N. Horspool. Schrödinger's Token. *Software — Practice and Experience 31*, 8 (2001), pp. 803–814.

J. Aycock and N. Horspool. Directly-Executable Earley Parsing. *CC 2001 — 10th International Conference on Compiler Construction (LNCS 2027)*, Springer-Verlag, 2001, pp. 229–243.

J. Aycock and N. Horspool. Simple Generation of Static Single-Assignment Form. *CC 2000 — 9th International Conference on Compiler Construction (LNCS 1781)*, Springer-Verlag, 2000, pp. 110–124.

J. Aycock. Aggressive Type Inference. *Proceedings of the 8th International Python Conference*, 2000, pp. 11–20.

J. Aycock and M. Levy. An Architecture for Easy Web Page Updating. *ACM Crossroads* 6.2, 1999, pp. 15–18.

J. Aycock and N. Horspool. Faster Generalized LR Parsing. *CC '99 — 8th International Conference on Compiler Construction (LNCS 1575)*, Springer-Verlag, 1999, pp. 32–46.

J. Aycock. Compiling Little Languages in Python. *Proceedings of the 7th International Python Conference*, 1998, pp. 69–77.

J. Aycock. Converting Python Virtual Machine Code to C. *Proceedings of the 7th International Python Conference*, 1998, pp. 43–50.