CS 164 Reader #1

Course Notes
(version of 2/10/2010)

Paul N. Hilfinger
University of California, Berkeley

Copyright © 2005, 2006, 2007, 2008, 2009, 2010 by Paul N. Hilfinger. All rights reserved.

Contents

1 Introduction **7**
- 1.1 Purposes of this Course . . . 7
- 1.2 An Extremely Abbreviated History of Programming Languages . . . 7
- 1.3 Problems to be Addressed . . . 9
- 1.4 Kinds of Translators . . . 9
- 1.5 Programming Languages You May Not Know . . . 13

2 Lexical Analysis **15**
- 2.1 Introduction . . . 15
- 2.2 Tokens and Tokenizing . . . 16
- 2.3 Simple Regular Expressions . . . 17
- 2.4 Standard Extensions to Regular Expressions . . . 19
- 2.5 Using Regular Expressions . . . 21
  - 2.5.1 Using regular expressions in general-purpose languages . . . 22
  - 2.5.2 A special-purpose tool: Flex . . . 22
- 2.6 Finite-state Machines . . . 22
  - 2.6.1 Deterministic recognizers . . . 28
  - 2.6.2 Non-deterministic recognizers . . . 29
  - 2.6.3 From DFAs to programs . . . 31
  - 2.6.4 From NFAs to programs . . . 31
  - 2.6.5 From NFA to DFA . . . 34
  - 2.6.6 From regular expressions to FSAs . . . 36
- 2.7 Theoretical Limitations . . . 37
- 2.8 Flex Revisited . . . 39
  - 2.8.1 Implementation . . . 39
  - 2.8.2 Start states . . . 41
- 2.9 Implementing More General Patterns . . . 42

3 Parsing **43**
- 3.1 Introduction . . . 43
- 3.2 Production Rules . . . 44
- 3.3 Derivations . . . 46
- 3.4 Ambiguity . . . 47
- 3.5 Context-Free Grammars . . . 48
- 3.6 Syntax-Directed Translation . . . 50

3.6.1 Abstract syntax trees 51
3.6.2 Actions without values 52
3.7 Some Common Idioms 54
3.8 Top-down Implementation 56
3.8.1 From grammars to programs: recursive descent 57
3.8.2 Choosing a branch: using FIRST and FOLLOW 59
3.8.3 Computing FIRST 60
3.8.4 Computing FOLLOW 61
3.8.5 Dealing with non-LL(1) grammars. 61
3.9 General context-free parsing 62
3.9.1 An abstract algorithm 62
3.9.2 Earley's algorithm 64
3.9.3 Extracting the derivation(s) 66
3.9.4 Dealing with epsilon rules 68
3.9.5 The effects of ambiguity 70
3.10 Deterministic Bottom-up Parsing 71
3.10.1 Shift-reduce parsing 71
3.10.2 Recognizing possible handles: the LR(0) machine 74
3.10.3 Using the machine 78
3.10.4 Resolving conflicts 80
3.10.5 Using Ambiguous Grammars 83

4 Static Analysis: Scope and Types 87
4.1 Terminology 87
4.2 Environments and Static Scoping 87
4.3 Dynamic Scoping 88
4.4 Compiler Symbol Tables 90
4.5 Lifetime 93
4.6 Static and Dynamic Typing 93
4.6.1 Type Equivalence 94
4.6.2 Coercion 94
4.6.3 Overloading 95
4.7 Unification (aka Pattern-Matching) 96
4.7.1 Type inference 98
4.7.2 Recursive patterns 99
4.7.3 Overload resolution in Ada 101

Prologue

Cautionary Note

These Notes are under construction. I suggest not committing them to paper until really necessary.

Acknowledgements

My thanks to those readers who have pointed out errors in the text:

> Daniel Hui, Gary Miguel, Ryosuke Niwa, Nathan Kitchen, Michael Matloob, Yunn Feng

Chapter 1

Introduction

1.1 Purposes of this Course

Not every programmer really has to know how to build a compiler. Although this course has the structure of compilers as its ostensible topic, my real agenda is broader.

- Acquire tools for building textual interfaces or other programs that process text.
- Understand the common structure of programming languages, so as better to learn them and better design programs with language-like capabilities.
- Acquire better intuition about program performance.
- Acquire practice in reading, designing, and writing complex programs.

1.2 An Extremely Abbreviated History of Programming Languages

- Initially, programs were either built into machines, or were entered by various electro-mechanical means, such as punched cards or tape (e.g., the Analytical Engine or the Jacquard loom), or wires and switches (e.g., the Eniac).
- Around 1944 came the idea of encoding programs as numbers (machine language) stored as data in the machine, whence came the Machester Mark I and the EDSAC.
- To make machine language easier to write, read, and maintain, assembly languages were introduced in the early 1950's: symbolic names for instructions and data locations, but still machine language—far from normal notation.
- FORTRAN: mid-1950's. Stands for FORmula TRANslator. Allowed use of usual algebraic notation in expressions; control structures (jumps, etc.) still close to machine language.

- LISP: late 1950's—dynamic, symbolic data structures.
- Algol 60: Europe's answer to FORTRAN. Many syntactic features of modern languages come from Algol 60. Also, it introduced the use of BNF (Backus-Naur Form), inspired by Chomsky's work, for describing syntax.
- COBOL (late 1950s): introduces business-oriented data structures, esp. records (structs for you C-folk).
- 1960s saw proliferation of increasingly elaborate languages: e.g., APL (array manipulation), SNOBOL (string manipulation), FORMAC (formula manipulation).
- 1967–68: Simula 67, an Algol 60 derivate intended for writing discrete-event simulations, introduces concept of inheritance, making it the first "object-oriented" language in the modern sense.
- 1968: Algol 68 attempted to synthesize both the numerical, FORTRANish line, the record-oriented line (such as COBOL), with dynamic (pointer-based) data structures. It also tried to extend BNF into describing the entire language. This last effort made it incomprehensible to many, and it faded away. Nevertheless, many C/C++ features may be found in Algol 68. PL/1 was IBM's clunkier but more commercially successful attempt to meet the same goals.
- 1968: announcement of the "Software Crisis." Trend toward simple languages: Pascal, Algol W, C (later).
- 1970s: emphasis on "methodology"—well-structured, modular programs. Several experimental languages (like CLU). Smalltalk introduced, inspired in part by Simula 67.
- Early 1970s: the Prolog language—declarative logic programming language. Originally intended for natural language processing. Later, it is pushed as a general-purpose high-level programming tool.
- Mid 1970s: ML (MetaLanguage) designed to implement LCF (Logic of Computable Functions). This is a functional language with some interesting ideas—type inference and pattern-directed function definition. Has evolved since then and spawned various progeny (e.g., Haskell).
- Mid-1970s: Dept. of Defense discovers it is supporting over 500 computer languages. Starts Ada project to consolidate (and then object-oriented Ada 95).
- Increasing complexity in ideas of object-orientation until ca. 1980 with advent of C++, which has continued to complexify.
- Reprising the move to simplicity in late 1970s, introduction of Java in early 1990s.

1.3 Problems to be Addressed

- How do we describe a programming language?
  - Users must be able to understand it unambiguously.
  - Implementors must be able understand it completely.
- Given a description, how to implement it?
  - How do we know we have it right?
    * Testing
    * Automating the conversion of description to implementation.
  - How do we save work?
    * Problem: multiple languages translated to multiple targets.
    * Automation (as above)
    * Designing so we can re-use pieces
    * Interpretation
- How do we make the end result usable?
  - Reasonable handling of errors.
  - Detection of questionable constructs.
  - Compilation speed.
    * A standard approach: design language and translator to allow compiling programs in sections.
    * Or, design translators to be really fast.
  - Execution speed. Problem: can make program run faster, if we are willing to have slower compilation. How do we make this trade-off?
  - Binding time. Problem: how to handle program whose parts change constantly, or even are unavailable at translation time?

1.4 Kinds of Translators

The purpose of translation is ultimately to *execute* a program. There is a spectrum of approaches.

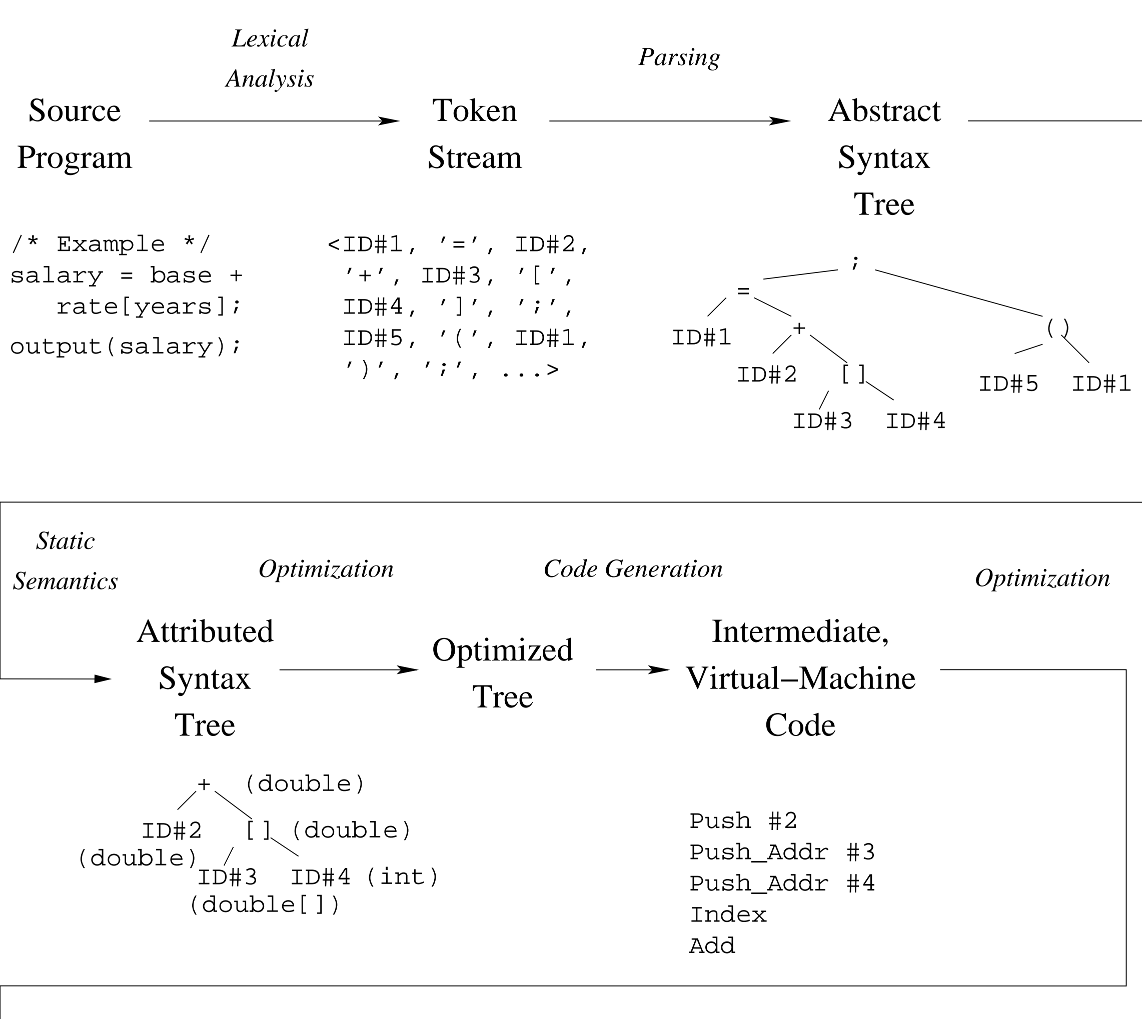
Compilation:	source $\xrightarrow{translate}$ real machine language $\xrightarrow{execute}$ actions/results
Interpretation:	source $\xrightarrow{translate}$ virtual machine language $\xrightarrow{interpret}$ actions/results
Direct Execution:	source $\xrightarrow{interpret}$ actions/results

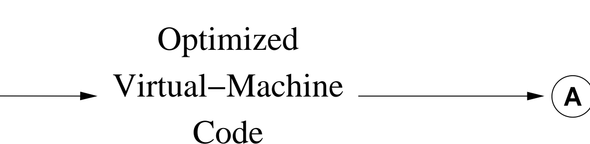
Most C/C++ systems are examples of compilation. Lisp interpretation is (in effect) an example of direct execution (the "translation" performed by the reader is trivial). Early Java systems used interpretation; now they use just-in-time compilation. These boundaries are muddy, however. Some machines, for example,

implement their instruction set by interpretation. Their "real" program is a *micro-program* that acts as an interpreter. For them, a C compiler is an example of an interpreter!

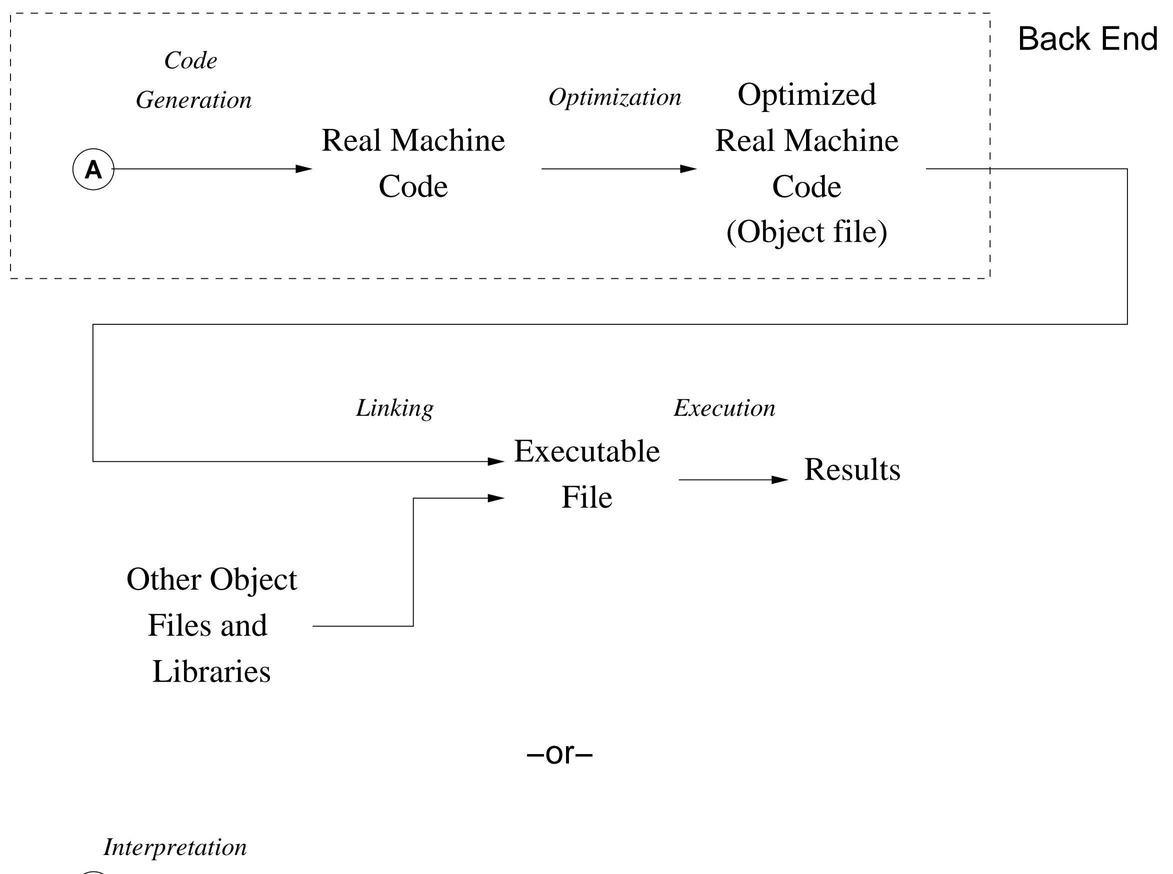
All three of these strategies are specific to an *implementation* of a language; they are *not* inherent in the language. *There are no such things as "interpreted languages" or "compiled languages."* For example, there are C interpreters, and there are Lisp compilers.

Structure of Idealized Compiler : Front End





Structure of Idealized Compiler



1.5 Programming Languages You May Not Know

```
C FORTRAN (OLD-STYLE) SORTING ROUTINE
C
      SUBROUTINE SORT (A, N)
      DIMENSION A(N)
      IF (N - 1) 40, 40, 10
10    DO 30 I = 2, N
         L = I-1
         X = A(I)
         DO 20 J = 1, L
            K = I - J
            IF (X - A(K)) 60, 50, 50
C
C FOUND INSERTION POINT: X >= A(K)
C
50             A(K+1) = X
               GO TO 30
C
C ELSE, MOVE ELEMENT UP
C
60             A(K+1) = A(K)
20       CONTINUE
         A(1) = X
30    CONTINUE
40    RETURN
      END

C ----------------------------------

C MAIN PROGRAM
      DIMENSION Q(500)
100   FORMAT(I5/(6F10.5))
200   FORMAT(6F12.5)

      READ(5, 100) N, (Q(J), J = 1, N)
      CALL SORT(Q, N)
      WRITE(6, 200) (Q(J), J = 1, N)
      STOP
      END
```

```
comment An Algol 60 sorting program;
procedure Sort (A, N)
   value N;
   integer N; real array A;
begin
   real X;
   integer i, j;
   for i := 2 until N do begin
      X := A[i];
      for j := i-1 step -1 until 1 do
         if X >= A[j] then begin
            A[j+1] := X; goto Found
         end else
             A[j+1] := A[j];
      A[1] := X;
   Found:
      end
   end
end Sort
```

```
⍝ An APL sorting program
∇ Z ← SORT A
  Z ← A[⍋A]
∇
```

```
/* A naive Prolog sort */

/* permutation(A,B) iff list B is a
   permutation of list A. */
permutation(L, [H | T]) :-
   append(V,[H|U],L),
   append(V,U,W),
   permutation(W,T).
permutation([], []).

/* ordered(A) iff A is in ascending order. */
ordered([]).
ordered([X]).
ordered([X,Y|R]) :- X <= Y, ordered([Y|R]).

/* sorted(A,B) iff B is a sort of A. */
sorted(A,B) :- permutation(A,B), ordered(B).
```

Chapter 2

Lexical Analysis

2.1 Introduction

The purpose of *syntactic analysis* is to analyze textual input so as to confirm that it is *syntactically well-formed*—that it obeys certain general structural rules dictated by the specification of the input language—and to convert it into a form that gives later parts of the compiler convenient access to this structure.

For example, we might say that in Java a conditional statement can have the form

if (*Expression*) *Statement* **else** *Statement*

In later parts of the compiler, the programmer might reasonably want a data structure that represents "an **if** statement" and that provides operations that return "the **then** clause," "the **else** clause," and "the test" from this statement. These operations would be awkward to implement if the data structure used were simply a copy of the original text of the statement (a string). Instead, a tree-like form is a better representation. This requires analyzing the original text into its constituent grammatical parts.

This task is traditionally partitioned into *lexical analysis*— which breaks the text down into the smallest useful atomic units, known as *tokens*, while throwing away (or at least, putting to one side) extraneous information, such as white space and comments—and *parsing*—which operates on tokens and groups them into useful grammatical structures. There is no sharp distinction between these two activities—I am happy to classify both under "syntactic analysis." A single monolithic subprogram could handle both simultaneously, as was done in very early compilers. To a certain extent, we divide the tasks as we do to accommodate certain techniques and certain automatic or semi-automatic tools.

We're going to start with lexical analysis. The part of a compiler that performs this task is called a *lexical analyzer, tokenizer,* or *scanner*. In brief outline,

- *Regular expressions* can describe a variety of languages (sets of strings), including the set of atomic symbols of a typical programming language.

- *Finite-state automata* (FSAs) are abstract machines that also recognize languages.

- *Deterministic finite-state automata* (DFAs) are a subset of finite-state automata that are easily converted into programs.
- There exists a translation from regular expressions into FSAs.
- There exists a translation from FSAs that happen to be nondeterministic into DFAs (and hence into programs).
- The total process of conversion from regular expression to program is automatable. In fact, we'll be using a couple of handy programs: FLEX (for producing scanners written in C or C++) and JFLEX (for producing scanners written in Java). These programs are really compilers themselves, translating succinct descriptions of programming-language syntax (a piece of it, anyway) into programs that "execute" these descriptions to extract tokens from the input.

2.2 Tokens and Tokenizing

In the context of programming-language translation we use the term *token* to refer to an abstraction for the smallest unit of program text that it is convenient when describing the syntax of a language. You don't want them to be too small. The parsing techniques we'll use in this class are designed to decide on what to do next on the basis of the next token of input. If tokens are single characters, they won't generally contain enough information to make this decision. For example, suppose a program has seen the characters '`x+y`' and the next character is a blank. This is insufficient information to determine whether '`x+y`' is to be treated as a subexpression, since if the next non-blank character is '`*`', then `y` should be grouped with whatever is after the asterisk. The lexer, on the other hand, can first eliminate whitespace, making the decision easier. Another example is '`x+y`' followed by a '`+`'. Here, the decision depends on whether the character immediately after the '`+`' is another '`+`'. If the lexer has previously grouped all '`++`'s into single tokens, the decision is easily made, with no *ad hoc* scanning ahead in special cases. Tokenizing is thus the process of bridging the gap between the input (made of characters—too small) and tokens.

As an example, a Java program (a Java *source file*) might contain the phrase

```
if(i== j)
    z = 0;  /* No work needed */
else
    z= 1;
```

which a translating program sees as a sequence of characters:

```
\tif(i== j)\n\t\tz = 0;  /* No work needed */\n\telse\n\t\tz= 1;
```

The job of the scanner is to convert this to a sequence of values such as this:

```
IF, LPAR, ID("i"), EQUALS, ID("j"), RPAR, ID("z"), ASSIGN,
INTLIT("0"), SEMI, ELSE, ID("z"), ASSIGN, INTLIT("1"), SEMI
```

Here, the upper-case symbols denote *syntactic categories* (often internally represented as integers in an actual compiler). The syntactic categories are consumed by the next stage of the compiler—the parser. When determining the structure of a program, it is not particularly important *which* identifier or integer literal appears at some point; the important point is that *some* identifier appears there. Hence, scanners typically separate the syntactic category from what I'll call the *lexical value* of the token (shown in parentheses), which gives information that the parser doesn't need, but later stages of the compiler will. In our example, the lexical value of an identifier happens to be the *lexeme* itself—the character string constituting the token.

As you can see from the example, information unnecessary to the rest of the compiler is filtered out entirely. All the blanks, tabs, newlines, and comments are gone, so that the little discrepancies in spacing around the operators are removed. This is a typical pattern in the translation process: each stage makes the job of its successors easier by removing "noise" and guaranteeing that certain errors are filtered out.

Actually, real scanners don't entirely do away with whitespace. If it is going to produce useful error messages, a compiler must keep track of where each token appears so that it can "point" at the offending part of the program in the original source file. Therefore, tokens often contain positional information—but just like the semantic value, it is separated from other parts of the token so that it can be referenced only when needed.

If we were building an actual scanner in Java, our tokens might be represented by objects with fields like this:

```
class Token {
   enum SyntacticCategory { IF, LPAR, ID, EQUALS, RPAR, ASSIGN, ... };
   SyntacticCategory syntax;
   Object value;
   Location sourcePosition;
   ...
}
```

2.3 Simple Regular Expressions

In formal language theory (as computer scientists practice it, that is), a *language* is a just set of strings—namely, all the valid *sentences, utterances,* or *words* in that language. This definition does not really correspond to the normal informal-English use of the word, but it can be useful for our purposes. The problem of creating a lexical analyzer is that of writing a program to *recognize* and extract the valid tokens making up some input text (or *source*). Now, we could just treat this as a Small Matter of Programming, and write a program from scratch that performed this task. The obvious approach might give you something like this (in Java):

```
int lastChar;
Token readToken () {
    do {
       lastChar = System.in.read ();
```

```
            if (lastChar == -1)
               return makeToken (EOF);
         } while (Character.isWhitespace (c));
         if (lastChar == '/') {
            lastChar = System.in.read ();
            if (lastChar == '*') {
                Read and discard comment
                return readToken ();
            } else if (lastChar == '/') {
                Read and discard //-style comment.
            } else
                return makeToken (SLASH);
         } else if ...
         ...
     }
```

and so on. This can go on for quite a while and become rather tedious. In particular, verifying that a program such as this in fact matches our notion of what the possible tokens are is certainly not easy. On the other hand, if you are trying to squeeze every possible bit of speed out of a lexical analyzer for some reason, you may be driven to such lengths.

It would be nice if there were a way to describe the possible tokens we're looking for directly and to have this description converted into a program automatically, so that it is a program that endures the tedium. For lexical analysis, a common description language for this purpose is the *regular expression.*

A classical regular expression, R, describes a language (set of strings) $L(R)$ (the *language of* R), according to the following recursive definition:

- The empty string, ϵ, is a regular expression, denoting the set containing just the empty string ($L(\epsilon) = \{\texttt{""}\}$).

- Any single character, c, is a regular expression, denoting the set containing just the one-character string containing c ($L(c) = \{\texttt{"}c\texttt{"}\}$).

- *Concatenation:* If R_1 and R_2 are two regular expressions defined by these rules, then R_1R_2 is a regular expression denoting the set $L(R_1R_2) = \{s_1s_2 \mid s_1 \in L(R_1) \text{ and } s_2 \in L(R_2)\}$.

- *Union:* If R_1 and R_2 are two regular expressions defined by these rules, then $R_1|R_2$ is a regular expression denoting the set $L(R_1|R_2) = L(R_1) \cup L(R_2)$.

- *Closure:* If R is a regular expression defined by these rules, then $R*$ is a regular expression[1], denoting the set $L(R*) = L(\epsilon) \cup L(R) \cup L(RR) \cup L(RRR) \cup \cdots$ (the concatenation of zero or more strings, each from $L(R)$).

For convenience, we often augment these with

[1] The '*' operator is known as the *Kleene star,* after the late Stephen C. Kleene, and pronounced (in America, anyway) like "Claynee".

- *Non-reflexive closure:* If R is a regular expression defined by these rules, then R^+ is a shorthand for $RR*$ (the concatenation of one or more strings, each from $L(R)$).

- *Optional:* If R is a regular expression defined by these rules, then $R?$ is a shorthand for $R|\epsilon$.

and the regular-expression analyzers we'll talk about all allow standard shorthands for unions of single-character tokens:

- The expression $[c_1c_2\cdots]$ is short for $c_1|c_2|\cdots$. The expression a-b in place of one of the c_i is short for the list of all one-character strings containing characters between a and b, inclusive, in whatever character collating sequence you are using (so `[f-i]` is the same as `[fghi]`).

- The expression `[^`$c_1c_2\cdots$`]` is short for $d_1|d_2|\cdots$, where the d_i are all characters that are *not* included among the c_i.

- The expression '.' generally means "the union of all single character strings not including line-terminating characters."

Normally, the closures and optional operators group more strongly than concatenation, which groups more strongly than union, and we use parentheses to clarify or disambiguate, so that "ab—cd*" is equivalent to "(ab)—(c(d*))." When writing regular expressions that programs will actually act on, there is also a tedious set of rules concerning escape sequences, so that we can describe strings that include characters such '*', '+', '|', etc.

This notation easily describes typical tokens in programming languages and many other simple strings as well. Here are a few examples:

<table>
<tr><th>Expression</th><th>Description</th></tr>
<tr><td>[-+/*=;,:()%^!@&|~\[\]]</td><td>Valid one-character tokens in Java (the square brackets are "escaped" with a backslash to indicate that they are to be interpreted as ordinary characters).</td></tr>
<tr><td>[-+/*^\%|&]=</td><td>All the complex assignment operators in Java.</td></tr>
<tr><td>[<>]=?|!=</td><td>All the comparison operators in Java.</td></tr>
<tr><td>[1-9][0-9]*</td><td>All decimal integer numerals in Java.</td></tr>
<tr><td>[a-zA-Z_][a-zA-Z0-9_]*</td><td>All identifiers in C (a letter or underscore followed by zero or more letters, digits, and underscores).</td></tr>
<tr><td>//.*</td><td>//-style comments in C++ and Java.</td></tr>
</table>

2.4 Standard Extensions to Regular Expressions

These days, most programming languages provide some kind of regular-expression matching facilities, sometimes built into the language (as in Perl), and sometimes as part of their standard library (as in Java, C, or Python). Most extend the simple expressions from §2.3 with additional ones for convenience. The following are typical. Here, R and R_i represent regular (sub)expressions. We use Python syntax.

Expression	Description
$R\{m\}$	Short for $\underbrace{R\cdots R}_{m \text{ times}}$.
$R\{m,n\}$ $R\{,n\}$	Short for $\underbrace{R\cdots R}_{m \text{ times}}\ \underbrace{R?\cdots R?}_{n-m \text{ times}}$; that is, m to n occurrences of R. By default, $m = 0$.
$R\{m,\}$	Short for $R\{m\}R*$; that is at least m occurrences of R.
`\d`	Any single digit.
`\s`	Any single whitespace character.
`\S`	Any single non-whitespace character.
`^`	Matches the empty string, but only at the beginning of a string or (if the right option is set) at the beginning of a line. This is most useful in cases where one is searching through a larger string for a substring that matches one's regular expression.
`$`	Matches the empty string, but only at the end of a string or (if the right option is set) at the end of a line.
`\`k	Where $k > 1$ is a single digit, matches the same string that was most recently matched by the kth parenthesized expression in this regular expression (the one started by the kth left parenthesis counting from the left). For example '`([a-z]+),\1`' matches "`foo,foo`" and "`bar,bar`", but not "`foo,bar`".

One useful feature of most of these regular-expression matching facilities is the ability to retrieve the string matched by some portion of the pattern—typically, by a parenthesized subexpression. Thus, in Python, one can write things like this:

```
M = re.match(r'Name:\s*(\S+),\s*(\S+)\s*(.*)', line)
# The r'...' notation is a "raw string", in which back slashes and the
# characters they preceded are left unchanged.
# M is a "match object", or null (None) if line does not match.
if M:
    print "Last name:", M.group(1)
    print "First name:", M.group(2)
    if M.group(3):  # An empty string means "false"
        print "Middle name(s):", M.group(3)
```

This ability to capture pieces of the match requires some additional description of what an expression matches. For example, so far as the basic definitions in §2.3 are concerned, the regular expression "`((\d+,)*)(\d+,?)(.*)`" simply matches the string "123,456". But this doesn't tell us what part of the string the first parenthesized expression matched (`.group(1)` in Python notation). Any of the following breakdowns might seem to be correct:

group(1)	group(3)	group(4)
"123,"	"456"	""
"123,"	"4"	"56"
""	"123,"	"456"
""	"1"	"23,456"

Similarly, the regular expression "`(ab|abcd).*`" matches the string "`abcd`", but does the parenthesized expression match "`ab`" or "`abcd`"? These questions don't arise when simply asking whether a regular expression matches as a whole, but they do arise when we ask for details of the match—of what we'll later call a *parse* or *derivation* of the target string according to the regular expression.

The usual way to deal with these questions is to adopt the "leftmost longest" rule:

- The operators '*', '+', '?', and '{···}' match *greedily*—that is, they match the maximum possible number of repetitions that are allowed by their definition, and that allow the rest of the match to succeed.

- When matching either R_1 or R_2 allows the entire pattern to match, a subexpression '$R_1|R_2$' matches R_1.

Thus, in the examples above, when "`((\d+,)*)(\d+,?)(.*)`" matches the string "123,456", groups 1, 3, and 4 match "123,", "456", and "", respectively. It chooses the first of the four alternatives for the content of the groups, and when "`(ab|abcd).*`" matches the string "`abcd`", the parenthesized group matches "ab."

However, it is sometimes convenient to alter this rule, giving rise to so-called *lazy quantifiers.* Thus, in Python

- The subexpression $R*?$ matches the *fewest* number of repetitions of R that allow the containing pattern to match.

- The subexpression $R+?$ is short for $RR*?$.

- The subexpression $R??$ matches the empty string, *unless* it must match R in order for the containing pattern to match.

So when "`((\d+,)*?)(\d+?,?)(.*)`" matches the string "123,456", groups 1, 2, and 4 match "", "1", and "23,456", respectively.

2.5 Using Regular Expressions

For the purposes of building programming-language translators, regular expressions provide a tool for describing the various tokens of interest. Let's consider a simple language (call it SL/1) containing the following tokens:

- The single-character tokens

  ```
  + - * / = ; , ( ) > <
  ```

- The multi-character operators

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

- The keywords

  ```
  if  def  else  fi  while
  ```

- Identifiers (a letter followed by 0 or more letters and digits).

- Decimal numerals (one or more digits).

Basically, we convert a string into tokens with a left-to-right scan, where at each point, we match the longest possible legal token (the *maximum munch rule*). The following characters are ignored, except as delimiters for tokens:

- All whitespace (blanks, tabs, newlines, and a few others).
- All comments, which begin with '`#`' and proceed to the next end of line.

2.5.1 Using regular expressions in general-purpose languages

The precise details depend on the particular language or library one uses to write the translator. Figure 2.1 illustrates one approach in Java, using the `java.util.Scanner` class, and for contrast, Figure 2.3 shows essentially the same analyzer presented as a Python generator.

2.5.2 A special-purpose tool: Flex

There are number of tools that are essentially *domain-specific languages* tailored to the purpose of lexical analysis or of writing simple programs that basically perform transformations of streams of tokens. Here, we describe a venerable example: FLEX, an open-source regular-expression-based lexer generator[2]. It converts description files into C or C++ programs that provide a simple interface suitable for general use, but also designed to supply input to parsing tools that we'll see later.

Figure 2.4 shows our lexer as it might be expressed in FLEX. Here, the possible lexemes are separted, and for each one, the programmer may supply an arbitary action (in C or C++) to be performed when the lexeme is encountered. The outside world sees a simple functional interface, in which calls to `yylex` provide the syntactic categories (represented as simple integers), and any additional lexical information is generally provided by means of some sort of global variable.

The emphasis in FLEX is on performance. On one platform[3], for example, on a randomly-generated file containing 166,000 lexemes on 4,700,000 lines containing 41MB of data, I saw:

Version	Time (sec)	μsec/byte
Java (Figure 2.1)	101	2.5
Python (Figure 2.3)	47	1.1
FLEX (Figure 2.4)	4	0.098

To make this possible, FLEX's regular expressions are restricted for the most part to the simple ones in §2.3. For most uses, this is not a big limitation.

2.6 Finite-state Machines

The tools described in the preceding sections suffice to write lexical analyzers, but it is also useful (and certainly traditional) to look into what goes on behind the

[2] FLEX is based on the LEX program developed by AT&T in the early days of Unix (M. E. Lesk, "Lex—a lexical analyzer generator." Computing Science Technical Report 39, AT&T Bell Laboratories, Murray Hill, N.J., 1975.) FLEX is the work of Vern Paxson, now a professor at UC Berkeley.

[3] A Sun Ultra 40 M2, running Solaris.

```
import java.io.Reader;
import java.util.Scanner;
import java.util.HashMap;
import java.util.regex.Pattern;

/** A Lexer produces a stream of tokens taken from an arbitrary Reader
 *  supplied to the constructor, in response to repeated calls to the
 *  Lexer's nextToken routine.  */
public class Lexer {

    /** The possible syntactic categories of the tokens returned by
     *  nextToken.  The arguments to the enumerals are the lexemes
     *  corresponding to the Category, when these are unique.  The
     *  Categories EOF and ERROR are artificial; they mark the end
     *  of the token stream and erroneous tokens, respectively. */
    public static enum Category {
        GTEQ(">="), LTEQ("<="), GT(">"), LT("<"), ARROW("-->"),
        PLUS("+"), MINUS("-"), STAR("*"), SLASH("/"), ASSIGN("="),
        LPAR ("("), RPAR (")"), SEMI(";"), COMMA(","),
        IF("if"), DEF("def"), ELSE("else"), FI("fi"), WHILE("while"),
        IDENT(null), NUMERAL(null), EOF(null), ERROR (null);

        final private String lexeme;
        Category (String s) {
            lexeme = s;
        }
    }

    /** The lexeme read by the last call to nextToken.  Undefined after
     *  nextToken returns EOF or before nextToken is called.  Contains
     *  the erroneous character after nextToken returns ERROR. */
    public String lastLexeme;

    /** Mapping of lexemes represented by Categories with single
     *  members to those categories. */
    private static HashMap<String, Category> tokenMap =
        new HashMap<String, Category> ();
    static {
        for (Category c : Category.values ())
            tokenMap.put (c.lexeme, c);
    }

    /** Input source. */
    private Scanner inp;
```
Continued

Figure 2.1: An SL/1 lexical analyzer that uses a Java Scanner (part 1).

Continued from Figure 2.1.

```
    /** A pattern that always matches the next token or erroneous
     *  character, except at end of file.  Group 1, if present is
     *  whitespace, group 2 is an identifier, group 3 is a numeral. */
    private static final Pattern tokenPat =
        Pattern.compile ("(\\s+|#.*)" +
                         "|>=|<=|-->|if|def|else|fi|while" +
                         "|([a-zA-Z][a-zA-Z0-9]*)|(\\d+)" +
                         "|.");

    /** A new Lexer taking input from READER. */
    public Lexer (Reader reader) {
        inp = new Scanner (reader);
    }

    /** Read the next token, storing it in lastLexeme, and returning
     *  its Category.  Returns EOF at end of file, and ERROR for
     *  erroneous input (one character). */
    public Category nextToken () {
        if (inp.findWithinHorizon (tokenPat, 0) == null)
            return Category.EOF;
        else {
            lastLexeme = inp.match ().group (0);
            if (inp.match ().start (1) != -1)
                return nextToken ();
            else if (inp.match ().start (2) != -1)
                return Category.IDENT;
            else if (inp.match ().start (3) != -1)
                return Category.NUMERAL;
            Category result = tokenMap.get (lastLexeme);
            if (result == null)
                return Category.ERROR;
            else
                return result;
        }
    }
}
```

Figure 2.2: SL/1 lexical analyzer in Java, part 2.

```
import re

r"""lexer.py: Lexical Analyzer Module.

A typical use might be this:
   for category, lexeme in lexer.readTokens (sys.stdin):
       # Process token with given category and lexeme.
"""

# Syntactic categories.  For best performance, compare against these using
# the 'is' operator, not '='

GTEQ = ">="; LTEQ = "<="; GT = ">";      LT = "<";    ARROW = "-->";
PLUS = "+";  MINUS = "-"; STAR = "*";    SLASH = "/"; ASSIGN = "="
LPAR = "(";  RPAR = ")";  SEMI = ";";    COMMA = ","
IF = "if";   DEF = "def"; ELSE = "else"; FI = "fi";   WHILE = "while";
IDENT = "IDENT"; NUMERAL = "NUMERAL"; ERROR = "ERROR"

_tokenMap = {
        GTEQ: GTEQ, LTEQ: LTEQ, ARROW: ARROW, GT: GT, LT: LT,
        PLUS: PLUS, MINUS: MINUS, STAR: STAR, SLASH: SLASH, ASSIGN: ASSIGN,
        LPAR: LPAR, RPAR: RPAR, SEMI: SEMI, COMMA: COMMA,
        IF: IF, DEF: DEF, ELSE: ELSE, FI: FI, WHILE: WHILE }

def readTokens(file):
    """A generator that returns pairs (C, L) consisting of the lexemes
    in FILE (L) and their syntactic categories (C)."""
    for token in re.finditer (r"(\s+|#.*)"
                              r"|>=|<=|-->|if|def|else|fi|while"
                              r"|([a-zA-Z][a-zA-Z0-9]*)|(\d+)"
                              r"|.",
                              file.read ()):
        L = token.group(0)
        i = token.lastindex
        if i == 1:
            pass
        elif i == 2:
            yield IDENT, L
        elif i == 3:
            yield NUMERAL, L
        else:
            yield _tokenMap.get(L, ERROR), L
```

Figure 2.3: An SL/1 lexical analyzer module in Python.

```
/*  A Flex version of our lexical analyzer (in C++). */

/* Various declarations may go before the first %% */

%{
  /* Code in %{ ... }% is inserted directly into the C/C++ program */

  /* See Figure 2.5 */
  #include "lexer.h"
%}

%option noyywrap

%%

[ \t\n\r\f]   { }

"#".*         { }

">="          { yylval = ">="; return GTEQ; }
"<="          { yylval = "<="; return LTEQ; }
"-->"         { yylval = "=="; return ARROW; }
[-+<>*/=();,] { yylval = yytext; return yytext[0]; }
"if"          { yylval = "if"; return IF; }
"def"         { yylval = "def"; return DEF; }
"else"        { yylval = "else"; return ELSE; }
"fi"          { yylval = "fi"; return FI; }
"while"       { yylval = "while"; return WHILE; }

[a-zA-Z][a-zA-Z0-9]* { yylval = yytext; return IDENT; }
[0-9]+        { yylval = yytext; return NUMERAL; }
.             { yylval = yytext; return ERROR; }

%%

/** Everything after the second %% goes into the generated program. */

void initLexer (FILE* file)
{
   yy_switch_to_buffer (yy_create_buffer (file, YY_BUF_SIZE));
}

int nextToken ()
{
   return yylex ();
}
```

Figure 2.4: A Flex lexer for SL/1, generating C++.

```
#include <string>
#include <cstdio>
using namespace std;

enum { GTEQ = 128, LTEQ, ARROW, IF, DEF, ELSE, FI, WHILE,
      IDENT, NUMERAL, ERROR };

extern int nextToken ();
extern void initLexer (FILE* file);
/* Must be supplied by the calling program. */
extern string yylval;
```

Figure 2.5: Interface to the program in Figure 2.4.

```
#include <iostream>
#include "lexer.h"

string yylval;
char* names[] =  {
      "GTEQ", "LTEQ", "ARROW", "IF", "DEF", "ELSE", "FI", "WHILE",
      "IDENT", "NUMERAL", "ERROR", "0"
};

main (int argc, char* argv[])
{
   if (argc <= 1)
      initLexer(stdin);
   else
      initLexer(fopen (argv[1], "r"));
   while (1) {
      int c = nextToken ();
      if (c == 0) break;
      if (c < 128)
          cout << "’" << yylval << "’" << endl;
      else
          cout << names[c - 128] << ": " << yylval << endl;
   }
}
```

Figure 2.6: Trivial main program that uses the Flex lexer from Figure 2.4.

scenes—at how regular expressions may be turned into programs for recognizing what they describe.

A standard approach is to organize such a program around an abstraction known as a *finite-state automaton* or *finite-state machine.* These machines have a number of applications in theoretical computer science, electrical engineering, and software design.

2.6.1 Deterministic recognizers

In its simplest form, the *deterministic finite-state automaton (DFA)* (or *... recognizer,*) a machine, M, contains the following ingredients:

- A finite set S of *states.* In the simplest case, the items in this set needn't have any properties other than being distinguishable from each other.
- A member $s_0 \in S$, called the *start state.*
- A finite *alphabet* of symbols, Σ.
- A *transition function*, $f : S \times \Sigma \rightarrow S$. For each state, σ, and symbol x, this function gives a *next state.*
- A set $F \subset S$ of *final states.*

(A theoretician usually states this as "a deterministic finite-state recognizer is a tuple $M = (S, \Sigma, s_0, f, F)$ such that....") Such a recognizer describes a language, a set of strings over the alphabet Σ—just as a regular expression does—by providing a means to *recognize* certain strings.

Consider an n-character string $\sigma = \sigma_1\sigma_2 \cdots \sigma_n$ (where all the $\sigma_i \in \Sigma$). To test this string, the machine reads the characters from left to right, maintaining an internal *state,* that can change at each character. Specifically,

1. M starts in state s_0.
2. Then, for each symbol σ_i in turn, the machine *transitions* to state s_i, using the rule $s_i = f(s_{i-1}, \sigma_i)$.
3. If at any time in this process, $f(s_{i-1}, \sigma_i)$ is undefined[4], then M *rejects* (does not recognize) σ.
4. Otherwise, M recognizes (or *accepts*) σ iff $s_n \in F$, that is, if its last state is one of the designated final states.

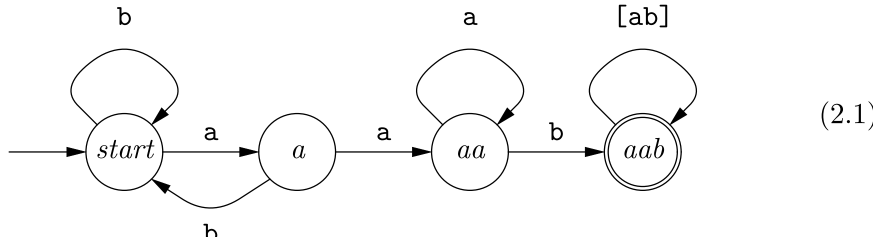
Just as for regular expressions, we'll use the notation $L(M)$ to denote the language of all strings recognized by M.

[4]Some authors avoid this problem by requiring that f be a total (everywhere-defined) function. This is always possible since you can add an *error state,* $e \notin F$, such that $f(e, x) = e$ for all $x \in \Sigma$. However, the addition of an error state leads to crowded diagrams, so consider our use of partial transition functions as a kind of abbreviation.

As an example, consider the problem of recognizing all strings of that contain the substring "aab" somewhere within them and that contain only a's and b's. For this problem, we'll take

$$\begin{aligned} S &= \{\textit{start},\ a,\ aa,\ aab\} \\ \Sigma &= \{\texttt{a, b, c}\} \\ s &= \textit{start}, \\ F &= \{aab\} \end{aligned}$$

(As you can see, Σ contains an extra character c, which will not appear in any accepted string). To describe f, the transition function, we'll use the following *state-transition diagram:*



(2.1)

The nodes (circles) in this diagram represent states. The start state is marked by an "arrow from nowhere." Final states are denoted by double circles. I've given the states suggestive labels, although none are required (e.g., state *aa* is "the state in which no *aab* has yet been seen, but we've just seen two a's in a row"). The arrows denote the values of the transition function, f. For example, the arrow labeled 'a' from state a to state aa indicates that $f(a, \texttt{'a'}) = aa$. We'll use the same bracket notation as for regular expression to indicate transitions between states that occur on multiple characters, as in the self-loop on state *aab*.

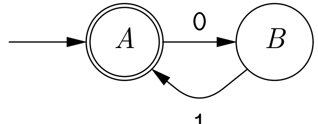
Given the string "babaabab", the machine above will go through the sequence of states

$$\textit{start} \xrightarrow{\texttt{b}} \textit{start} \xrightarrow{\texttt{a}} a \xrightarrow{\texttt{b}} \textit{start} \xrightarrow{\texttt{a}} a \xrightarrow{\texttt{a}} aa \xrightarrow{\texttt{b}} aab \xrightarrow{\texttt{a}} aab \xrightarrow{\texttt{b}} aab.$$

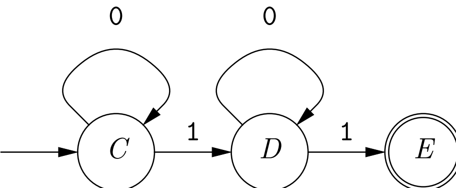
Since it ends in *aab,* a final state, M accepts "babaabab." Given the empty string, on the other hand, M ends up in *start,* which is not a final state, and so rejects the string. Given "babaa," it ends in state *aa,* which again is not a final state, and so it rejects this string. Finally, given the string "aabcaab," M will reach state *aab*, but then, encountering the character 'c' (for which there is no transition defined), will stop and reject the string (step 3 in the recognition procedure).

2.6.2 Non-deterministic recognizers

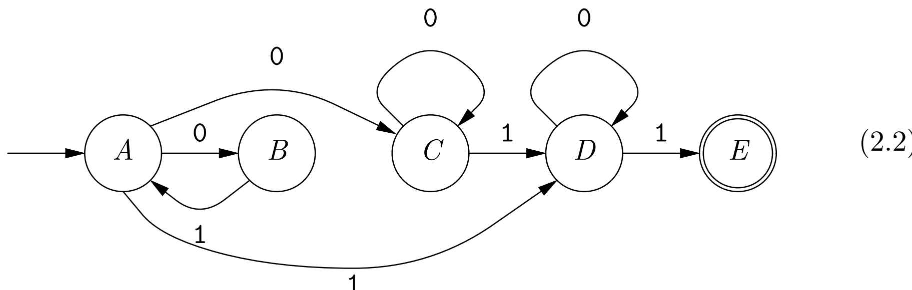
Consider the problem of building a recognizer for strings matching the regular expression '(01)*0*10*1'. It's easy to see how to match '(01)*':



and also ‘0*10*1’:



but it is less obvious how to put them together, since it is not clear when the recognizer is “in the (01)* part” and when it is “in the 0*10*1 part.” Suppose though, that we expand the definition of finite-state automata to make *both* choices, in effect, like this:



(2.2)

This is almost the same as just concatenating the two machines together with some extra edges, except that states A now has multiple edges emerging from it with the same label. The result is a *nondeterministic finite-state automaton (NFA).* We say that an NFA accepts a string iff there is *some* path—some sequence of edges—whose edge labels are the string and that ends on an accepting state. As long as at least one such path ends up in a final state, it doesn't matter how many others crash and burn. So, given the string “010101”, there is a partial path

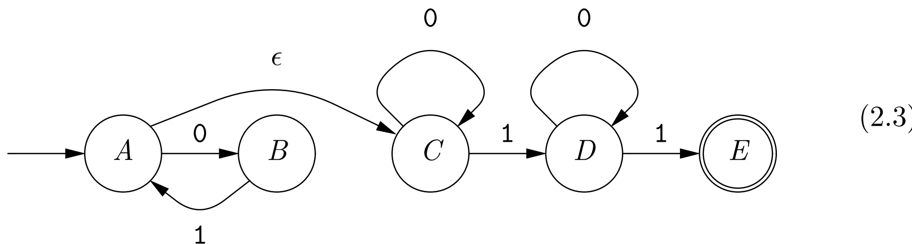
$$A \xrightarrow{0} C \xrightarrow{1} D \xrightarrow{0} D \xrightarrow{1} E$$

but there is no transition out of E for the trailing “01”, so this path fails. Nevertheless, the machine accepts “010101” because the alternative path

$$A \xrightarrow{0} B \xrightarrow{1} A \xrightarrow{0} C \xrightarrow{1} D \xrightarrow{0} D \xrightarrow{1} E$$

does consume the entire string and ends up in a final state.

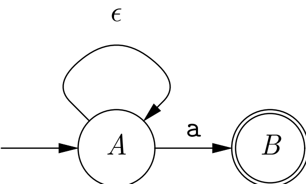
We can simplify this still more, removing one edge, by an *ϵ-transition:*



(2.3)

The meaning of this transition is that when in state A, the machine *may* transition to state C *without* consuming a character of input. The introduction of ϵ transitions

allows "infinite loops," of course, as in this rather silly machine for recognizing {"a"}:



While it is true that there is an infinite path through this machine that cycles endlessly back to state A without consuming input, the machine will still accept 'a' because there are also paths (actually infinitely many) that *do* consume the string and that end up in final state B.

Earlier, I described a DFA as a tuple $M = (S, \Sigma, s_0, f, F)$. An NFA is likewise a tuple, $M' = (S, \Sigma \cup \{\epsilon\}, s_0, f, F)$, except that now, the transition function, f maps states and characters in the alphabet (plus the special "character" ϵ) to *sets* of possible next states, or as mathematicians write,

$$f : S \ \times \ \Sigma \cup \{\epsilon\} \rightarrow 2^S.$$

(2^S is the powerset of S, often written $\mathcal{P}(S)$.)

2.6.3 From DFAs to programs

Transforming a DFA into a program is pretty straightforward. Consider machine (2.1) in §2.6.1. We can convert this into the simple C++ program shown in Figure 2.7 (Java is very similar).

The program in Figure 2.7 is sufficiently stylized that we can accomplish the same thing in a table-driven fashion, as shown in Figure 2.8. This program represents the transition function as a 2-dimensional table indexed by state and by character (after first converting the alphabet into integer 0's and 1's). In this particular example, we did not have to worry about states with no transition on one of the characters 'a' or 'b'. If there had been such transitions, we'd simply introduce an additional non-final state, ERROR, as described in the footnote in §2.6.1. The FLEX program produces essentially this kind of program, having first converted its regular expressions into DFAs (see §2.6.6.)

2.6.4 From NFAs to programs

As indicated in §2.6.2, NFAs differ from DFAs in having transition functions that return sets of states, rather than individual states, and that allow transitions on ϵ (the empty string) as well as ordinary characters. This suggests a relatively straightforward modification to the program in Figure 2.8, along the lines shown in Figure 2.9, which shows a program modeled on NFA (2.2). Here, an `int` serves to represent a set of states. Any given set of states is represented as the bitwise or of the corresponding bits, so that, for example, the set $\{C, B, D\}$—consisting of states number 1, 2, and 3—would be represented as the integer 14 ($2^1 + 2^2 + 2^3$ or `1<<B|1<<C|1<<D`). Had there been more states than the bit-size of `int`, we would have had to use a more general "bitvector" data structure.

```
bool recognize1 (string& s) {
    enum { START, A, AA, AAB } state;
    state = START;
    for (int i = 0; i < s.size (); i += 1) {
        switch (state) {
        case START:
            switch (s[i]) {
            case 'a': state = A; continue;
            case 'b': continue;
            default: return false;
            }
        case A:
            switch (s[i]) {
            case 'a': state = AA; continue;
            case 'b': state = START; continue;
            default: return false;
            }
        case AA:
            switch (s[i]) {
            case 'a': continue;
            case 'b': state = AAB; continue;
            default: return false;
            }
        case AAB:
            switch (s[i]) {
            case 'a': case 'b': continue;
            default: return false;
            }
        }
   }
   return state == AAB;
}
```

Figure 2.7: C++ program implementing DFA (2.1) in §2.6.1. For robustness, this program contains 'default' clauses for characters that are not in the alphabet.

```
enum State { START = 0, A, AA, AAB, NUM_STATES } state;
typedef int SetOfStates;
/* The set of possible final states, represented as an
 * integer in which the kth bit (1<<k) is set if state #k
 * is in the set.  */
static const SetOfStates FINAL_STATES = 1<<AAB;
static const State START_STATE = START;

static const State transition[][2] = {
/*    'a'   'b'                 */
    { A,   START },    /* START */
    { AA,  START },    /* A */
    { AA,  AAB   },    /* AA */
    { AAB, AAB   }     /* AAB */
};

bool recognize2 (string& s) {
    state = START_STATE;
    for (int i = 0; i < s.size (); i += 1) {
        int c;
        switch (s[i]) {
        case 'a': c = 0; break;
        case 'b': c = 1; break;
        default: return false;
        }
        state = transition[state][c];
    }
    return (1<<state & FINAL_STATES) != 0;
}
```

Figure 2.8: A table-driven version of Figure 2.7.

The `recognize3` procedure maintains the set of possible states, given the characters it has read so far, starting with a set containing just the start state. For each input character, it scans the set of states it might currently be in, and accumulates (into the variable `newStates`) the possible next states, as given in the table `transitions`.

The machine in Figure 2.9 does not make use of ϵ transitions, which the program accommodates by means of a separate `epsilonClosures` table. The substitute tables shown in Figure 2.10 model NFA (2.3), which does make use of them. The value of `epsilonClosures[`S`]` for any state S is the set containing S and all states that the machine can reach from S using (one or more) ϵ transitions[5].

As an example, consider the result of executing `recognize3` on the string `0101011`. Using the original set of tables in Figure 2.9, we get the following sequence of state sets:

$$\begin{aligned}\{A\} &\xrightarrow{\texttt{0}} \{B,C\} \xrightarrow{\texttt{1}} \{D,A\} \xrightarrow{\texttt{0}} \{B,C,D\} \xrightarrow{\texttt{1}} \{D,A,E\} \xrightarrow{\texttt{0}} \{B,C,D\} \\ &\xrightarrow{\texttt{1}} \{D,A,E\} \xrightarrow{\texttt{1}} \{D,E\},\end{aligned}$$

which `recognize3` represents internally as the sequence of integers:

$$1 \xrightarrow{\texttt{0}} 6 \xrightarrow{\texttt{1}} 9 \xrightarrow{\texttt{0}} 14 \xrightarrow{\texttt{1}} 25 \xrightarrow{\texttt{0}} 14 \xrightarrow{\texttt{1}} 25 \xrightarrow{\texttt{1}} 24.$$

Using the alternative tables from Figure 2.10, we get the following sequence of state sets:

$$\begin{aligned}\{A,C\} &\xrightarrow{\texttt{0}} \{B,C\} \xrightarrow{\texttt{1}} \{D,A,C\} \xrightarrow{\texttt{0}} \{B,C,D\} \xrightarrow{\texttt{1}} \{D,A,C,E\} \xrightarrow{\texttt{0}} \{B,C,D\} \\ &\xrightarrow{\texttt{1}} \{D,A,C,E\} \xrightarrow{\texttt{1}} \{D,E\},\end{aligned}$$

or internally,

$$5 \xrightarrow{\texttt{0}} 6 \xrightarrow{\texttt{1}} 13 \xrightarrow{\texttt{0}} 14 \xrightarrow{\texttt{1}} 29 \xrightarrow{\texttt{0}} 14 \xrightarrow{\texttt{1}} 29 \xrightarrow{\texttt{1}} 24.$$

2.6.5 From NFA to DFA

I don't normally juxtapose abstract processes (like automata making state transitions) with the nitty-gritty details of programs that represent such processes, so it might seem incongruous that I showed the sequence of internal variable values used in `recognize3` along with the state sets being represented. But in this case, there was a point. There are only a finite number of possible values for the `states` variable in `recognize3`, namely integers in the partially open range $[0, 2^5)$. For a given value of `states` and of `s[i]`, the program always computes the same value for `newStates`. But that means that `recognize3` is effectively doing exactly the same thing that our DFA implementation in Figure 2.8! In other words, we could use the numbers 0–31 as our states, and build a deterministic table with entries like this (for the tables from Figure 2.9):

[5] OK, a more elegant way to say this (which avoids having to say "containing S and") is that the closure is the set of all states reachable from S using *zero* or more ϵ transitions, but I find that many people have trouble with zero, for some reason.

```
enum State { A = 0, B, C, D, E, NUM_STATES } state;
typedef int SetOfStates;

static const SetOfStates epsilonClosures[NUM_STATES] = {
    1<<A, 1<<B, 1<<C, 1<<D, 1<<E
};
static const SetOfStates FINAL_STATES = 1<<E;

static const SetOfStates transitions[][2] = {
/*         '0'        '1'            */
    { 1<<B | 1<<C,  1<<D  },     /* A */
    { 0,            1<<A  },     /* B */
    { 1<<C,         1<<D  },     /* C */
    { 1<<D,         1<<E  },     /* D */
    { 0,            0     }      /* E */
};

bool recognize3 (string& s) {
    SetOfStates states;
    states = epsilonClosures[A];
    for (int i = 0; i < s.size (); i += 1) {
        int c;
        SetOfStates newStates;
        newStates = 0;
        switch (s[i]) {
        case '0': c = 0; break;
        case '1': c = 1; break;
        default: return false;
        }

        for (State p = A; p != NUM_STATES; p = (State) (p+1))
            if (states & 1<<p)
                newStates |= transitions[p][c];
        for (State p = A; p != NUM_STATES; p = (State) (p+1))
            if (newStates & 1<<p)
                newStates |= epsilonClosures[p];
        states = newStates;
    }

    return (states & FINAL_STATES) != 0;
}
```

Figure 2.9: A table-driven program for direct implementation of NFA (2.3). It represents sets of states with `int`s: bit k is 1 (where bit 0 is units bit) iff the kth state is in the set (where the 0th state is A). This version has the machinery to deal with ϵ transitions, but does not use them.

```
static const SetOfStates epsilonClosures[NUM_STATES] = {
    1<<A|1<<C, 1<<B, 1<<C, 1<<D, 1<<E
};
static const SetOfStates FINAL_STATES = 1<<E;

static const SetOfStates transitions[][2] = {
/*         '0'          '1'               */
    { 1<<B,         0     },      /* A */
    { 0,            1<<A  },      /* B */
    { 1<<C,         1<<D  },      /* C */
    { 1<<D,         1<<E  },      /* D */
    { 0,            0     }       /* E */
};
```

Figure 2.10: Alternative tables for the `recognize3` program in Figure 2.9 that include ϵ transitions, as used in NFA (2.3).

```
static const int transition[][2] = {
/*     '0'  '1'                        */
    { 0,   0   },     /* 0: { }       */
    { 6,   8   },     /* 1: { A }     */
    ...
    { 4,   9   },     /* 6: { B, C } */
    ...
    { 14,  24  },     /* 9: { D, A } */
```

This construction is completely general: *any* NFA may be converted to a DFA in this fashion. Therefore, NFAs are no more powerful than DFAs in the languages they can recognize. Because the state numbers stand for subsets of the NFA's states, we call this the *subset construction.*

Of course, in general not all subsets of states can be reached in a running program. Thus, in NFA (2.2), no string will ever cause the machine to be in the set of states $\{A, B\}$. So typically, we renumber the states when doing this transformation so as to minimize the size of the resulting state table by leaving out unreachable rows.

2.6.6 From regular expressions to FSAs

So far, we've looked at regular expressions and finite-state automata as two alternative means of recognizing languages. As it happens, however, the two are closely related. Any simple regular expression from §2.3 (and most from §2.4) is convertible into a NFA. This is a rather pretty and classical construction, so let's take a look at it.

We proceed recursively, guided by the definition of what a regular expression can be. For each of the possible forms of basic regular expression, R, we'll construct a NFA, M, with a single final state, such that $L(R) = L(M)$. Table 2.1 shows the

machines that result from each possible kind of regular expression. We use ellipses, as in:

to denote, respectively, a machine that recognizes R and the same machine, but with all its states made non-final. That is, each ellipse stands for some collection of states and arrows, and the two circles inside the ellipse denote the starting and (single) final states of the machine (which may be the same state). The table shows only basic constructs. Extensions to cover things such as R^+, `[a-z]`, or R? are reasonable straightforward (especially since those are all just shorthands anyway).

This construction allows us to convert any regular expression into a NFA, which may be further converted, as described in §2.6.5, to a DFA. The procedures outlined in §2.6.4 and §2.6.3 can then convert these machines into programs. There is, in fact, a construction that will convert any NFA into a regular expression[6], demonstrating that NFAs, DFAs, and regular expressions are all equivalent in their ability to recognize languages.

The construction is a slight variation on one due to Ken Thompson[7] and was used for searches in a text editor that applied the resulting NFA directly, without converting it to a DFA. To make this task simpler, the algorithm guarantees that each state has either a single outgoing transition on an input character from Σ or else at most two ϵ transitions. Its correctness depends on there being no outgoing states from any final state. Any number of variations on this construction are possible.

2.7 Theoretical Limitations

A DFA has a finite number of states, and therefore a finite memory. As a result, there are many languages that simply cannot be recognized by a pure DFA, and therefore not by a regular expression or NFA, either. For example, consider the set $\{a^n b a^n \mid n \geq 0\}$, the set of all strings of as followed by an equal number of as, separated by a single b. In order for a DFA to recognize this set, it would have to be in a different state after reading the string a^k for each possible value of k (since only after reading k repetitions of a would it be correct to accept the suffix ba^k). But that's impossible, since k can have any of an infinite number of values, but the DFA has only a fixed, finite number of possible states. In other words, "DFAs (and NFAs) cannot count."

Indeed, for any DFA (or NFA or regular expression) that accepts an infinite number of strings, there is some minimum length, M, such that any string longer than M that the machine accepts must have the form uxv, where u, x, and v are strings, x is not empty, the length of ux is $\leq M$, and the machine accepts *all* strings of the form ux^nv. This result is known as the *pumping lemma.*

Classically, the term "regular expression" refers only to those expressions de-

[6]It is rather tedious, however, so just take my word for it, if you don't mind.

[7]K. Thompson, "Regular expression search algorithm," *Comm. ACM* **11:6** (1968), pp. 419-422.

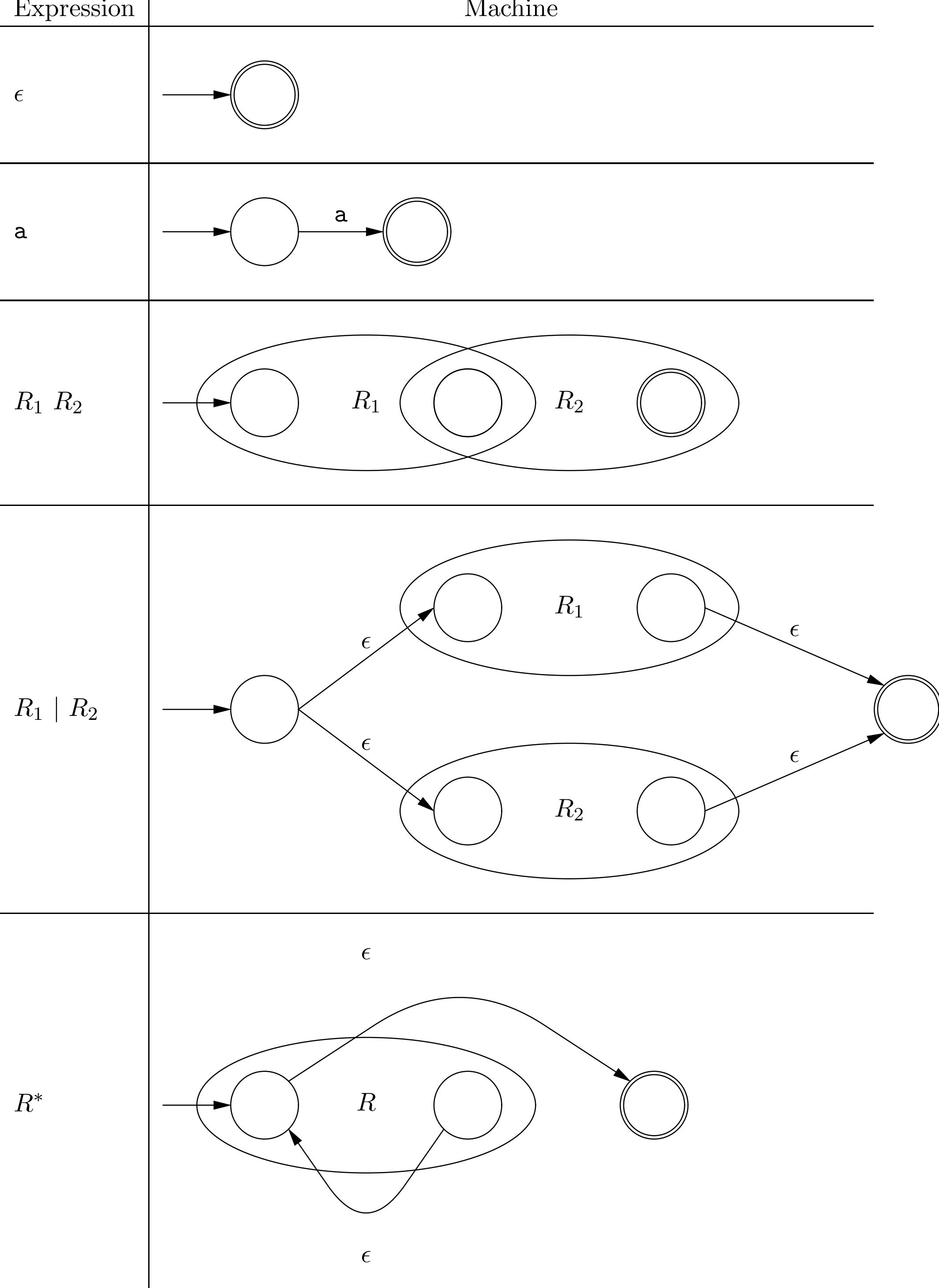


Table 2.1: Recursive construction of NFA from a regular expression.

scribed in §2.3. Unfortunately for those of us who are easily confused, it is common these days to give the label "regular expression" to constructs that go beyond these in fundamental ways, and can do things that NFAs cannot. For example, the Python regular expression `r"(a*)b\1"` recognizes $a^n b a^n$; the '`\1`' matches the same string matched by the parenthesized subexpression. This *non-linear pattern* is not a classical regular expression. I'd prefer to have it called a "pattern," but I suppose I'll to compromise on "Python regular expression" (more commonly called, alas, a "Perl regular expression").

2.8 Flex Revisited

2.8.1 Implementation

The FLEX program described in §2.5.2 converts the regular expressions it is supplied into a DFA for recognizing the language they collectively describe, using essentially the transformations described in §2.6.6 and §2.6.5, with some additional optimizations and modifications.

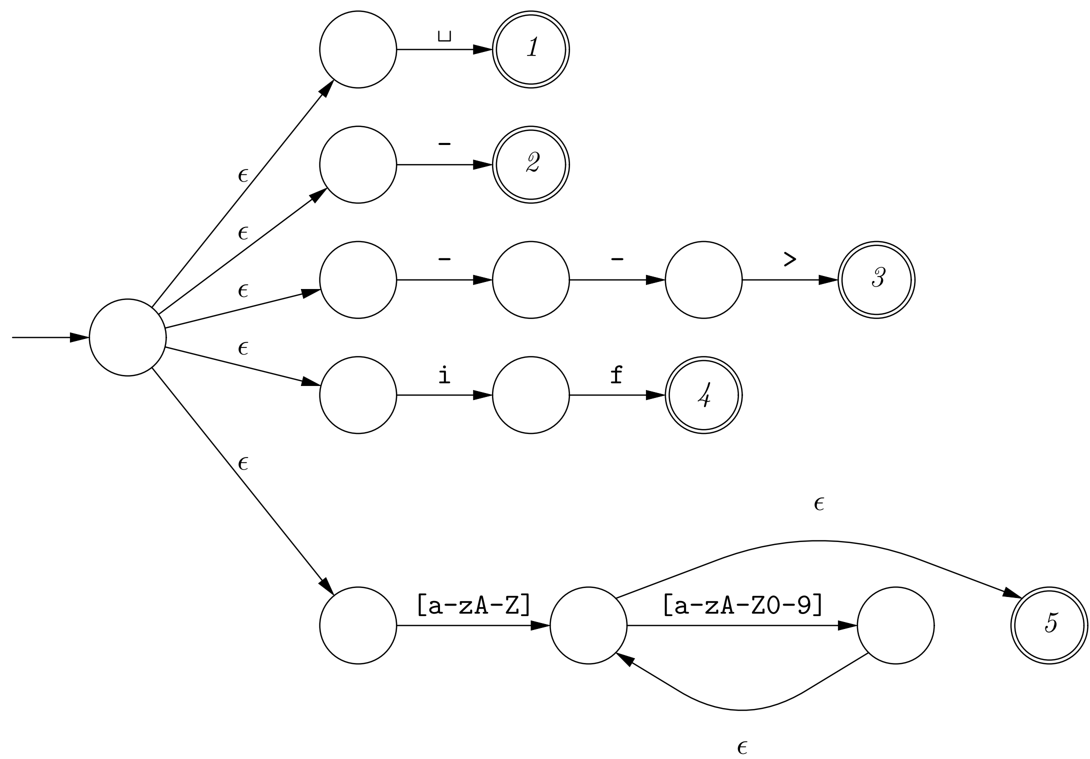
So far, we have concentrated on producing machines that simply tell whether a certain string is in the desired language—a plain yes/no determination. But consider the following stripped-down excerpt from the FLEX program in Figure 2.4:

```
" "                     { }                 /* 1 */
"-"                     { return '-'; }     /* 2 */
"-->"                   { return ARROW; }   /* 3 */
"if"                    { return IF; }      /* 4 */
[a-zA-Z][a-zA-Z0-9]* { return IDENT; }   /* 5 */
```

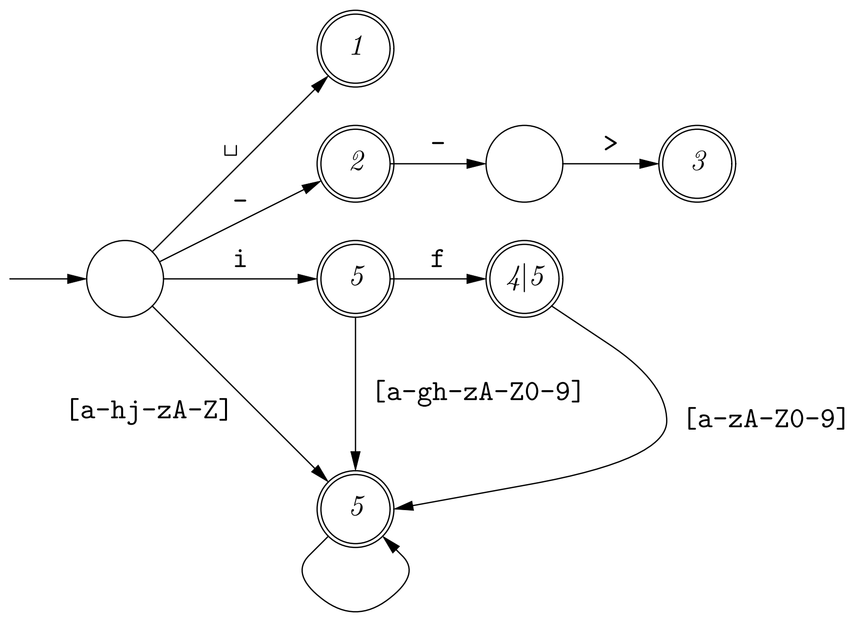
and the input "`if x --> --whilevar`". This input does not, in fact, match any of the patterns (as far as any individual pattern is concerned, it has "trailing junk"). But the first call to the generated recognizer function (called `yylex`) is supposed to give us the `IF` token. In other words, it is supposed to match a *prefix* of the input. Furthermore, it is not enough to know that some prefix of the input matched one of these patterns; the program needs to know *which* pattern in order to execute the proper **return** statement (or nothing for white space). Finally, there are ambiguities to resolve. The prefix "`if`" matches both the IF and the IDENT pattern. Which do we select? When we get to it, a prefix of the string "`-->`" matches "`-`" and another matches "`-->`". Again, which do we select? These are questions that have been unimportant up to now. FLEX happens to resolve them with two rules:

- *Maximum munch:* match the longest possible prefix of the input at each point.
- *Order determines precedence:* given two patterns that match equal-length prefixes, choose the first pattern.

Some simple modifications to our procedures so far solve both problems by imposing these two rules. Consider the following NFA for recognizing any of the tokens above. I've used multiple final states in this machine, one for each rule, and used the "`[···]`" shorthand (which technically stands for multiple edges between nodes).



Now convert this NFA into a DFA using the subset construction from §2.6.5. I'll label only the nodes that correspond to subsets containing final states with those final states.



We push input characters through this DFA as before, but with a couple of changes:

1. We continue to push characters into the machine until it rejects the string or

we reach the end of the input.

2. We keep track of the last time the machine was in a final state. When we arrive at a point where the machine rejects the next character, we back the input up so that the machine is in this state.

3. The output of the algorithm is then the label on this final state. If the state is labeled with more than one label, the result is the smallest label (corresponding to the earlier rule).

Of course, if this procedure does not result in the machine being in a final state, we report an error.

For example:

- The string "`-␣xy`" gets rejected when we reach the blank (denoted '␣'). Since the last final state traversed before this happens is labeled *2,* the program reports that we should execute rule 2 (**return** `'-'`).

- The string "`--xy`" gets rejected when we reach '`x`'. Again, the last final state traversed before this happens is labeled *2.* We back the input up to that point ("unreading" the second '`-`' and report that we should execute rule 2.

- The string "`-->y`" gets rejected when we reach '`y`'. This time, the last final state traversed is labeled *3,* indicating rule 3.

- The string "`if␣x`" gets rejected at the blank, at which point the last final state is labeled *4|5*. By provision 3 above, we report that we should execute the lowest-numbered rule, 4 (**return** `IF`).

- The string "`ifx`" ends us up at the end of the input in a final state labeled 5, so we report rule 5 (**return** `IDENT`).

2.8.2 Start states

Because FLEX produces a state machine, it is able to provide some state-machine like features. Specifically, a user may specify a set of alternative starting states. Any pattern may be specified to apply only in certain states, and any action, or other code outside the lexer, may include a statement that selects which starting state will be used for the next match.

For example, suppose you have a language in which the phrases `@decimal`, `@octal`, and `@hex` don't return tokens, but cause subsequent numerals to be interpreted a decimal, octal, or hexadecimal, respectively, with decimal being the default. You could write

```
%s  OCTAL, HEX

%%

... other tokens

"@octal"      { BEGIN OCTAL; }
```

```
"@decimal"    { BEGIN INITIAL; }
"@hex"        { BEGIN HEX; }

<INITIAL>[0-9]+  { ... }
<OCTAL>[0-7]+    { ... }
<HEX>[0-9][0-9]a-fA-F]* { ... }

... other tokens
```

2.9 Implementing More General Patterns

Since §2.7 tells us that finite automata cannot recognize languages that Python (or Perl) patterns can, it follows that matching these expressions requires a different approach from what we used for Flex. A popular approach seems to be to perform the match by interpreting the pattern directly and using a form of backtracking search. To match an expression $R_1|R_2$, for example, the matcher recursively first tries to match R_1, and if that fails, it backs up to the same point in the input and then tries matching R_2. To match a parenthesized expression, (R), it matches R and saves the bounds of the resulting match. To match a non-linear pattern such as the '`\1`' in "`(a*)b\1`" (from §2.7) after matching the '`b`', the matcher simply retrieves the last match for `a*` and compares that with the next part of the input string.

As we've already seen, the generality of this implementation scheme comes with a performance cost.

Chapter 3

Parsing

3.1 Introduction

In Chapter 2, I said that the purpose of syntactic analysis is to analyze textual input to detect and filter out errors and to convert into a form suitable for further processing. That chapter went on to describe lexical analysis, a piece of the larger task of syntactic analysis that is conveniently described and implemented using regular expressions or other pattern languages. In this chapter, we attack *parsing,* the name given to the rest of this problem.

The effect of lexical analysis is basically to change the alphabet of symbols of our language so that it consists of bigger "characters" (tokens), which we call *terminal symbols.* We do this largely for the convenience of our tools. The parsing techniques we'll use in this class are designed to decide on what to do next on the basis of the next token of input. If tokens are single characters, they won't have enough information to decide. For example, suppose a Java translator has seen the characters '`x+y`' and the next character is a blank. This is insufficient information to determine whether '`x+y`' is to be treated as a subexpression, since if the next non-blank character is '`*`', then `y` should be grouped with whatever is after the asterisk. The lexer, on the other hand, can first eliminate whitespace, making the decision easier. Another example is '`x+y`' followed by a '`+`'. Here, the decision depends on whether the character immediately after the '`+`' is another '`+`'. If the lexer has previously grouped all '`++`'s into single tokens, the decision is easily made, with no *ad hoc* scanning ahead in special cases.

Of course, the various flavors of regular expression and automata discussed in Chapter 2 already provide a way of describing and recognizing languages over an alphabet. Why not just use them once again to do the rest of syntactic analysis? First, as we saw in §2.7, regular expressions and their ilk are not well-suited to recognizing certain common features of programming languages. For example, Java allows expressions such as '`(a)`', '`((a))`', and so forth, but not '`(a))`'. However, as we saw, a pattern of the form $(^k \cdots)^k$ is not describable with plain regular expressions. Second, the emphasis so far has been on *recognizing* languages, but we also want to recover the *inner structure* of our input as well. The use of parenthesized groups suffices for simple tasks, but is clumsy for full-fledged analysis of recursively defined languages.

3.2 Production Rules

We can address these problems by moving to a different way of describing languages: using systems of rules that allow us to name constituent parts of a string. For example, here is a description of floating-point literals in a Java-like language:

Grammar 3.1. Floating-point literals, version 1.

```
digit:  '0'
digit:  '1'
digit:  '2'
    ⋮
digit:  '9'
int:  digit
int:  digit int
sign:  '+'
sign:  '-'
sign:
exponent:  'e' sign int
exponent:
literal:  sign int '.'  int exponent
```

Each line '$A : \alpha_0 \cdots \alpha_n$' can be read as "an A may be formed from an α_0 followed by an α_1,. . . , followed by an α_n." When n is 0, there are no right-hand side symbols (see the last definitions of `sign` and `expon`, for example), so that we have "A may be formed from the empty string" in such cases. The symbols that are defined to the left of some arrow are called *nonterminal symbols* (or *nonterminals* or *metavariables*) and the other symbols are called *terminal symbols* (or *terminals*). Each nonterminal symbol thus stands for a language; we typically single out one of them—called the *start symbol*—to be the principal language described by the rules (in this case, the start symbol would be `literal`). We call an entire collection of rules a *grammar*.

To save vertical space, we often use a simple shorthand for multiple rules with the same left-hand side:

```
digit:  '0' | '1' | '2' | ...  | '9'
int:  digit | digit int
sign:  '+' | '-' |
expon:  'e' sign int |
literal:  sign int '.'  int expon
```

and I will often use ϵ to make empty right-hand sides more visible, as in

```
sign:  '+' | '-' | ϵ
```

I've taken this notation (or *meta-syntax* to use the fancy terminology) from BISON, a parser-generator program. Other variations are possible. Some authors use '$\rightarrow$ ',

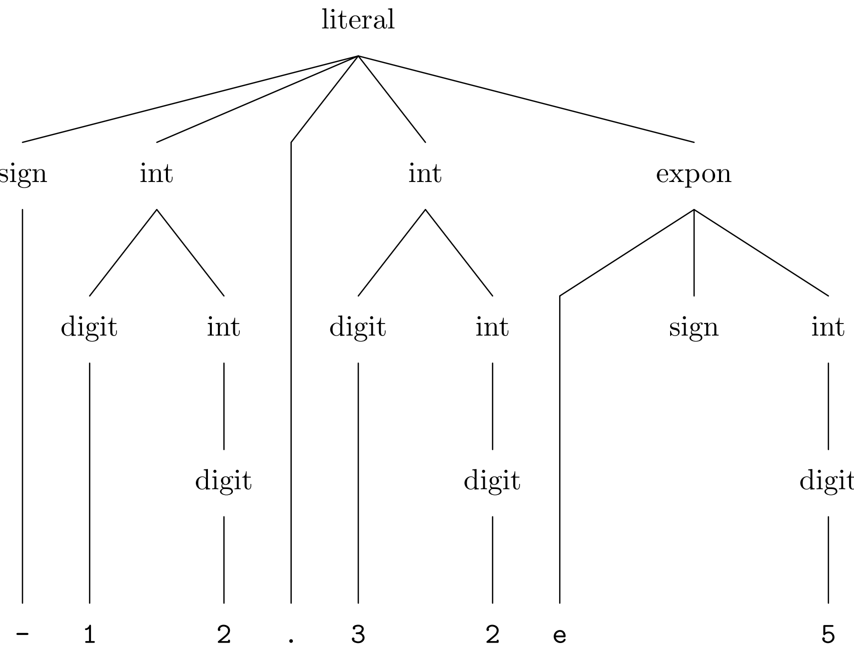


Figure 3.1: Parse tree for '-12.32e5', using Grammar 3.1

and the classic Algol 60 report used '::='. All these notations go by the name of *Backus-Naur Form* or *BNF*[1].

These rules make it convenient to talk about "the sign part" of the literal, or "the exponent part." Given a string, say '-12.32e5', we can say the string is in the language described by "literal" because we can break it down according to the rule for "literal" like this:

$$\underbrace{-}_{\text{sign}} \quad \underbrace{12}_{\text{int}} \quad \underbrace{.}_{\text{`.'}} \quad \underbrace{32}_{\text{int}} \quad \underbrace{e5}_{\text{expon}}$$

and each of these five pieces matches either a matching terminal symbol or some right-hand side of the indicated nonterminal. For example,

$$\underbrace{e}_{\text{`e'}} \quad \underbrace{\quad}_{\text{sign}} \quad \underbrace{5}_{\text{int}}$$

By repeating this process, we get the tree structure shown in Figure 3.1, which matches the entire string and shows the entire breakdown. The process of performing this matching of rules with terminal symbols is called *parsing*. The resulting tree is called a *parse tree*. The leaves of this tree are nonterminals that produce the empty string and terminal symbols.

[1] Sometimes called *Backus Normal Form,* but this name is inappropriate, I think, since the productions are "normalized" only in the rather weak sense of having a single nonterminal on the left-hand side.

3.3 Derivations

Most compilers don't really deal with parses in the form of parse trees, but are more concerned with what rules were applied. The information content is actually exactly the same. Suppose we perform a preorder traversal of the parse tree in Figure 3.1 and for each nonterminal we visit, print out the rule that was used to produce its children. We get the following list:

```
 1. literal:  sign int '.'  int expon
 2. sign:  '-'
 3. int:  digit int
 4. digit:  '1'
 5. int:  digit
 6. digit:  '2'
 7. int:  digit int
 8. digit:  '3'
 9. int:  digit
10. digit:  '2'
11. expon:  'e' sign int
12. sign:
13. int:  digit
14. digit:  '5'
```

To see how much information is present here, let's go back the other way, and re-create the tree from this list of applied rules. The top node is the start symbol, 'literal'. We now apply step (1), which tells us that the children of the root node are instances of 'sign', 'int', the terminal symbol '.', 'int', and then 'expon'. Since we did a preorder traversal to get this tree, we know that step (2) has to apply to the leftmost unprocessed nonterminal, an instance of 'sign', telling us that its child is the terminal symbol '-'. Step (1) produced two instances of 'int', but since we know that we got this list of rules from a preorder traversal, we know that step (3) has to apply to the leftmost of those instances, giving children 'digit' and 'int'. If you continue this process, you should see that we recover the parse tree in Figure 3.1.

The list of productions 1–14 defines a *leftmost derivation* of the string '-12.32e5' from the grammar. The term "leftmost" refers to the fact that each step applies to the leftmost nonterminal instance in the partially completed tree that has not yet been assigned a rule. Another way of presenting the same derivation is as a sequence of *sentential forms:*

$$
\begin{array}{llll}
\texttt{literal} & \stackrel{1}{\Longrightarrow} \texttt{sign int . int expon} & \stackrel{2}{\Longrightarrow} \texttt{- int . int expon} \\
& \stackrel{3}{\Longrightarrow} \texttt{- digit int . int expon} & \stackrel{4}{\Longrightarrow} \texttt{- 1 int . int expon} \\
& \stackrel{5}{\Longrightarrow} \texttt{- 1 digit . int expon} & \stackrel{6}{\Longrightarrow} \texttt{- 1 2 . int expon} \\
& \stackrel{7}{\Longrightarrow} \texttt{- 1 2 . digit int expon} & \stackrel{8}{\Longrightarrow} \texttt{- 1 2 . 3 int expon} \\
& \stackrel{9}{\Longrightarrow} \texttt{- 1 2 . 3 digit expon} & \stackrel{10}{\Longrightarrow} \texttt{- 1 2 . 3 2 expon} \\
& \stackrel{11}{\Longrightarrow} \texttt{- 1 2 . 3 2 e sign int} & \stackrel{12}{\Longrightarrow} \texttt{- 1 2 . 3 2 e int} \\
& \stackrel{13}{\Longrightarrow} \texttt{- 1 2 . 3 2 e digit} & \stackrel{14}{\Longrightarrow} \texttt{- 1 2 . 3 2 e 5}
\end{array}
$$

A sentential form, in other words, is just a sequence of symbols, both terminals and nonterminals. The '$\Rightarrow$' symbol may be read as "derives in one step" (in this derivation, I've notated each '$\Rightarrow$' with the number of step from the previous list of productions for this string). We keep going until there are no nonterminals left (a sentential form containing no nonterminals is also called a *sentence*). The symbol '$\stackrel{*}{\Longrightarrow}$' means "derives in 0 or more steps", and '$\stackrel{+}{\Longrightarrow}$' means "derives in 1 or more steps." Thus 'literal' $\stackrel{*}{\Longrightarrow}$ '-12.32e5' and 'literal' $\stackrel{+}{\Longrightarrow}$ '-12.32e5' and 'sign int . int expon' $\stackrel{+}{\Longrightarrow}$ '-1 digit . int expon'.

As long as we are consistent in how we apply rules, we can always effect this transfer between parse trees and derivations. For example, if we perform a preorder tree traversal of the parse tree, except that we visit children from right to left, rather than left to right, we get what is called a *rightmost derivation.* For the example above, we would apply the same rules the same number of times, but in a different order:

```
1, 11, 13, 14, 12, 7, 9, 10, 8, 3, 5, 6, 4, 2
```

If this looks a little strange, consider the sequence in reverse (appropriately called a *reverse rightmost* or *canonical* derivation, which goes backwards, "unapplying" each production in turn.) The first rule in the reversed sequence handles the '-' at the *beginning* of the string. The next rule (step 4 in the leftmost derivation) handles the '1' digit—the second character of the string. Here is the whole reverse rightmost derivation as a sequence of sentential forms, showing the corresponding step in the leftmost derivation above each (reversed) arrow:

	`- 1 2 . 3 2 e 5`	$\stackrel{2}{\Longleftarrow}$	`sign 1 2 . 3 2 e 5`
$\stackrel{4}{\Longleftarrow}$	`sign digit 2 . 3 2 e 5`	$\stackrel{6}{\Longleftarrow}$	`sign digit digit . 3 2 e 5`
$\stackrel{5}{\Longleftarrow}$	`sign digit int . 3 2 e 5`	$\stackrel{3}{\Longleftarrow}$	`sign int . 3 2 e 5`
$\stackrel{8}{\Longleftarrow}$	`sign int . digit 2 e 5`	$\stackrel{10}{\Longleftarrow}$	`sign int . digit digit e 5`
$\stackrel{9}{\Longleftarrow}$	`sign int . digit int e 5`	$\stackrel{7}{\Longleftarrow}$	`sign int . int e 5`
$\stackrel{12}{\Longleftarrow}$	`sign int . int e sign 5`	$\stackrel{14}{\Longleftarrow}$	`sign int . int e sign digit`
$\stackrel{13}{\Longleftarrow}$	`sign int . int e sign int`	$\stackrel{11}{\Longleftarrow}$	`sign int . int expon`
$\stackrel{1}{\Longleftarrow}$	`literal`		

So the reverse rightmost derivation reconstructs the parse tree from the bottom up and from left to right, whereas the leftmost (forward) derivation constructs it top down from left to right.

In principle, all kinds of other derivations are possible, but we will be chiefly interested in the leftmost and (in its reverse form) the rightmost derivation. Furthermore, although technically the term "derivation" refers to a sequence of sentential forms, we will use both these and sequences of productions (under a fixed, specified derivation order) interchangeably.

3.4 Ambiguity

It may not be obvious, but the grammar for floating-point literals that we've been using has the property that there is a unique parse tree that matches any valid string. This isn't true of all grammars. For example, suppose we tried to describe a slightly different kind of floating-point literal, in which either the integer part or the fraction part, but not both may be empty. One way to do so would be to replace the definitions of 'int' and 'literal' as follows:

```
optint:  int optint |
int:  digit optint
```

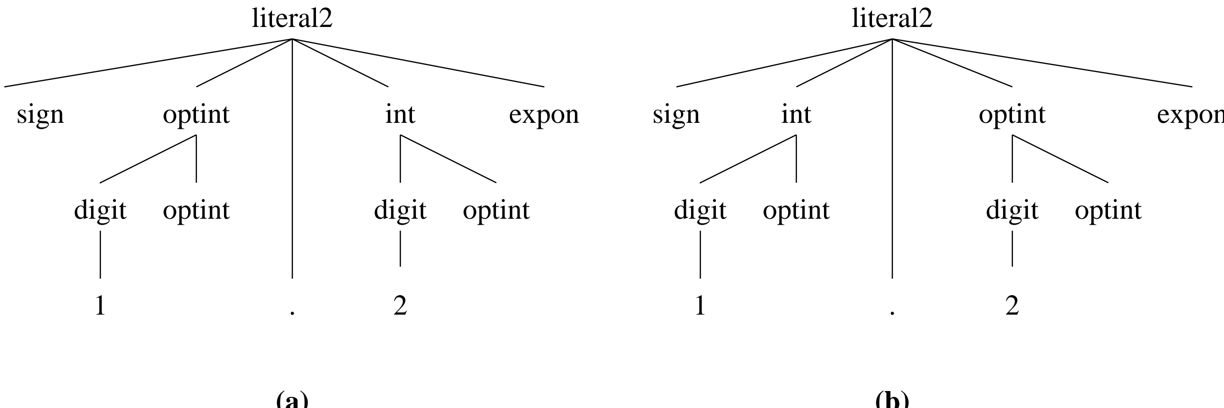


Figure 3.2: Two parses for ‘1.2’ under the ‘literal2’ grammar

```
literal2:  sign optint '.'  int expon | sign int '.'  optint expon
```

(I use ‘literal2’ to distinguish this language from the preceding). But now there are two ways to match the string ‘1.2’: either the ‘1’ or the ‘2’ can be an ‘optint’. These two choices correspond to the two different parse trees shown in Figure 3.2, with two different corresponding leftmost derivations:

```
(a) literal2 ⇒ sign optint .  int expon ⇒ optint .  int expon
             ⇒ digit optint .  int expon ⇒ 1 optint .  int expon
             ⇒ 1 .  int expon ⇒ 1 .  digit optint expon
             ⇒ 1 .  2 optint expon ⇒ 1 .  2 expon ⇒ 1 .  2
(b) literal2 ⇒ sign int .  optint expon ⇒ optint .  int expon
             ⇒ digit optint .  optint expon ⇒ 1 optint .  optint expon
             ⇒ 1 .  optint expon ⇒ 1 .  digit optint expon
             ⇒ 1 .  2 optint expon ⇒ 1 .  2 expon ⇒ 1 .  2
```

We say that the grammar is *ambiguous:* there exists at least one string for which there are two parses. For every parse tree there is still exactly one leftmost (or rightmost) derivation, but there are multiple parse trees, and hence multiple derivations.

We say that the *grammar* is ambiguous, not the *language*, because there is an unambiguous grammar for this language: simply replace the definitions for ‘literal2’ with

```
literal2:  sign int '.'  int expon
literal2:  sign '.'  int expon | sign int '.'  expon
```

(There are, in fact, languages that have only ambiguous grammars, but in practice, one doesn’t encounter them in compilers.)

3.5 Context-Free Grammars

So far, the grammars I’ve written have obeyed the following ordering restrictions:

1. Any nonterminal in the right-hand side of a rule, except possibly one that appears at the right end of the rule, is completely defined in previous rules.

2. If a nonterminal appears as the rightmost symbol on a right-hand side, it is either previously defined or it is the same as the left-hand side symbol of the rule.

That is, the following grammar would *not* qualify:

```
A: 'x' B
B: 'y' B 'z'
```

The first rule mentions a nonterminal that is not yet defined. That problem could be cured by making it the last rule. However, the second rule cannot be fixed: the nonterminal B appears in the middle of a rule with B on the left side.

Grammars that do obey these restrictions are known as *regular* or *Type 3* grammars. It's fairly easy to see that they describe the same languages as do regular expressions—that any such grammar can be converted to a regular expression and vice-versa. As such, they are subject to the same descriptive limitations as finite-state automata. Likewise, grammars obeying a slightly different constraint can easily be converted to NFAs, and are also regular. Specifically, any grammar in which only the last symbol on a right-hand side may be a nonterminl are also regular.

Unfortunately, common programming languages are clearly non-regular. For example, the languages you are probably familiar with require that parentheses be balanced. But that is just a slightly modified form of the a^nba^n example from §2.7. Now, it's true that in practice, one could limit programmers to, say, 20 levels of parenthesis nesting, and that such a language could be described by a regular expression or regular grammar. However, you would find it a rather horrendous grammar.

If we remove the regular-grammar restrictions, we get a class of grammars known as the *Type 2* or *context-free* grammars, which will be our next main object of study[2].

Such grammars can "count" arbitrarily high, at least under the right circumstances. For example, here is a language of correctly nested parentheses:

```
parens:  parens '(' parens ')' | ϵ
```

It recognizes the empty string, '()', '()()', '(())', '(()(()))', etc.

[2]In case you are interested, there are also Type 1 and Type 0 languages. Type 1 (or *context-sensitive*) languages can have more than one symbol on the left, as long as the number of symbols on the right is at least as large as the number on the left in each production (an extra rule to allow 'ϵ' in the language is also allowed). Type 0 (or *recursive*) languages, the most general, allow one or symbols on the left, and any number on the right of each production. Any language that is theoretically recognizable by a computer is of Type 0.

3.6 Syntax-Directed Translation

Finding a derivation would merely be an interesting academic exercise (and we haven't even gone into the "how" of it yet) were it not for the fact that we can use it for our Larger Purpose of translating programming languages. In some sense, a parse tree itself is a translation of a programming language into a form that makes it easy to get at the logical units of programs (the "then part" of a conditional, for example). However, we can use the parsing process to direct the formation of other kinds of translation (in fact, we usually do; parse trees are not usually produced directly). For example, suppose I wanted to translate floating literals into `doubles`. I could define how to do so by attaching *semantic actions* to the grammar rules that assign *semantic values* to the nodes of the parse tree. I'll use a slight variation of the grammar in §3.2:

Grammar 3.2. Floating-point literals, version 2.

```
digit:  '0'        { $$ = 0; }
   ⋮
digit:  '9'        { $$ = 9; }
int:  digit        { $$ = $1; }
int:  int digit    { $$ = 10*$1 + $2; }
sign:  '+'         { $$ = 1; }
sign:  '-'         { $$ = -1; }
sign:              { $$ = 1; }
exponent:  'e' sign int
                   { $$ = $2 * $3; }
exponent:          { $$ = 1; }
frac:  digit       { $$ = 0.1 * $1; }
frac:  digit frac  { $$ = 0.1 * ($1 + $2); }
literal:  sign int '.'  frac exponent
                   { $$ = $1 * ($2 + $4) * 10**$5; }
```

Each semantic rule is to be read as assigning a semantic value to each node in the parse tree[3]. Usually, the parse tree stays implicit, and we just use the values attached to it. For a rule of the form $A : \alpha_1 \cdots \alpha_n$, the notation `$$` means "the value assigned to a node produced by this rule" and `$`k means "the value that was assigned to the instance of α_k that a node produced by the rule that created it." Here, I have taken the liberty to introduce `**` as the exponentiation operator. I have also taken the liberty of re-arranging the definition of `int` according to our new freedom to write context-free grammars.

Consider parsing the string "`-12.32e5`" with this grammar. Figure 3.3 shows the resulting parse tree. The values after the colons are the semantic values assigned

[3]Here, I have used the notation of YACC and BISON for the semantic rules (the things in curly braces). Another style you'll sometimes see is to use the names of the nonterminals in the semantic rules, as in

```
frac:  digit frac { frac0 = 0.1 * (digit + frac1); }
```

Since we'll be using BISON, let's stick with its notation here to avoid confusion.

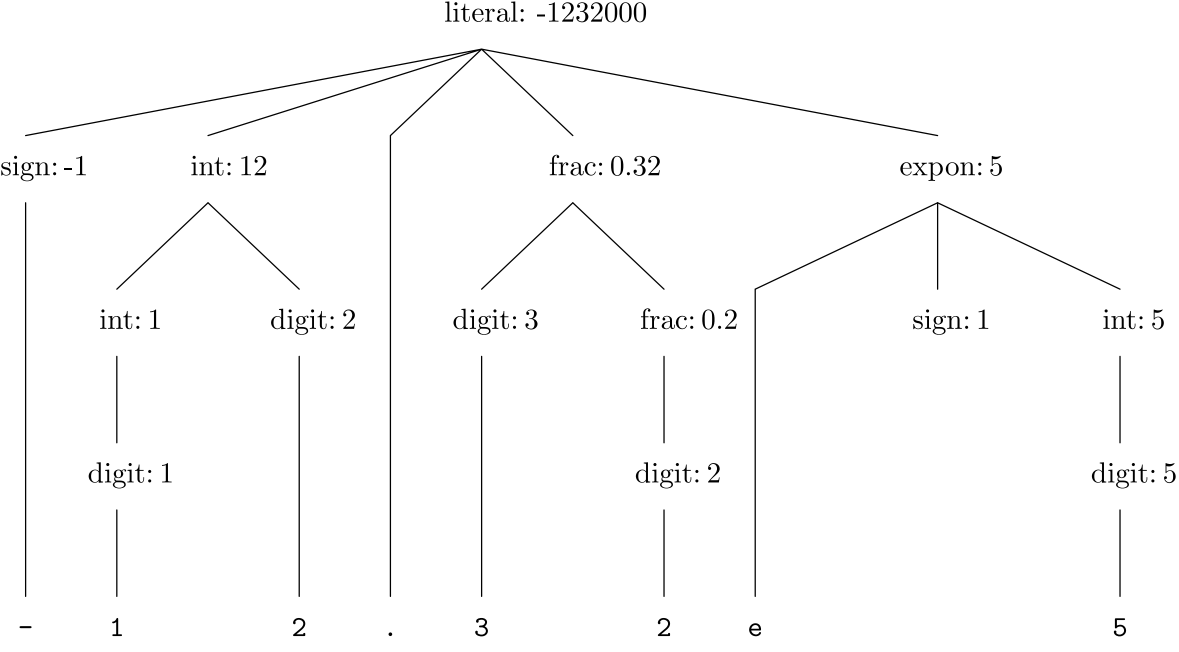


Figure 3.3: Parse tree and computed semantic values for '-12.32e5', using Grammar 3.2

by the rules of the grammar (the values assigned to $$). For example, the root node's value was computed as $-1232000 = -1 \cdot (12 + 0.32) \cdot 10^5$, according to the production for 'literal' in Grammar 3.2.

3.6.1 Abstract syntax trees

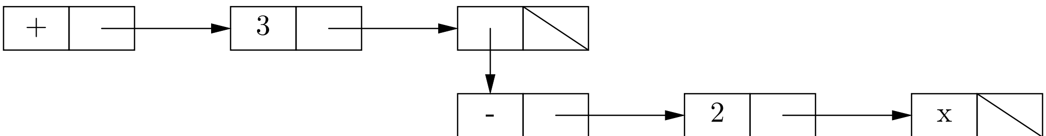
The semantic values we attach to nodes need not be restricted to simple strings or numbers. I suggested earlier that in compilers, we are often interested in converting an input program into a some kind of tree form that allows convenient access to the natural subparts of the constructs in a program. Parse trees have this property, but generally contain extraneous information (for example, they include all the original terminal symbols). They also have structures that reflect the details of the grammars we use to recognize them, rather than the purposes we have for them. It is common, therefore, to use semantic actions to produce trees corresponding to an *abstraction* of the grammar we use. Whereas we call the grammar a *concrete syntax* of our language, we call these alternative trees *abstract syntax trees* or *ASTs* for short.

For example, consider this, possibly familiar, syntax:

Grammar 3.3. Lisp subset reader.

```
expr :  SYM                  { $$ = makeAtom ($1); }
     | NUM                   { $$ = makeAtom ($1); }
     | '(' SYM list ')'      { $$ = cons ($2, $3); }
list :  expr                 { $$ = cons ($1, EMPTYLIST); }
     | expr list             { $$ = cons ($1, $2); }
```

Here, `makeAtom` simply converts a lexeme into a leaf node (or *atom* in Lisp terminology) and `cons` prepends its first argument to its second (a list). This gives the familiar Lisp representation of trees, where the node label is the head of the list and the list of children is the tail. The input string "{(+ 3 (- 2 x))}" is converted by this grammar into the Lisp structure:



3.6.2 Actions without values

In fact, it isn't necessary to compute semantic values at all. We can also use syntax-directed translation as a kind of control structure that, like some kind of **for** loop, recurses over the structure of our input and performs indicated actions. For example, consider this grammar for converting certain Lisp-style expressions (in which operators come first, as in '`(+ 3 (- 2 x))`' into unparenthesized postfix notation, in which operators come last, as in '`3 2 x - +`'.

Grammar 3.4. Lisp subset grammar for postfix printing.

```
expr :  SYM                  { printf ("%s ", $1); }
     | NUM                   { printf ("%s ", $1); }
     | '(' SYM list ')'      { printf ("%s ", $2); }
list :  expr                 { /* do nothing */ }
     | expr list             { /* do nothing */ }
```

In this grammar, we assume that `SYM` (symbol) and `NUM` (numeral) denote tokens defined by a lexical analyzer, which provides them with semantic values that are simply the lexemes themselves (so that the parser sees the input string '`123`' as a `NUM` whose value is simply the string "`123`". The input string "`(+ 3 (- 2 x))`" turns into the tokens

$$\texttt{'('}\ \ \texttt{SYM}_{+}\ \ \texttt{NUM}_{3}\ \texttt{'('}\ \ \texttt{SYM}_{-}\ \ \texttt{NUM}_{2}\ \ \texttt{SYM}_{x}\ \ \texttt{')'}\ \ \texttt{')'}$$

(the subscripts show the semantic values attached by the lexical analyzer). We get the parse tree shown in Figure 3.4

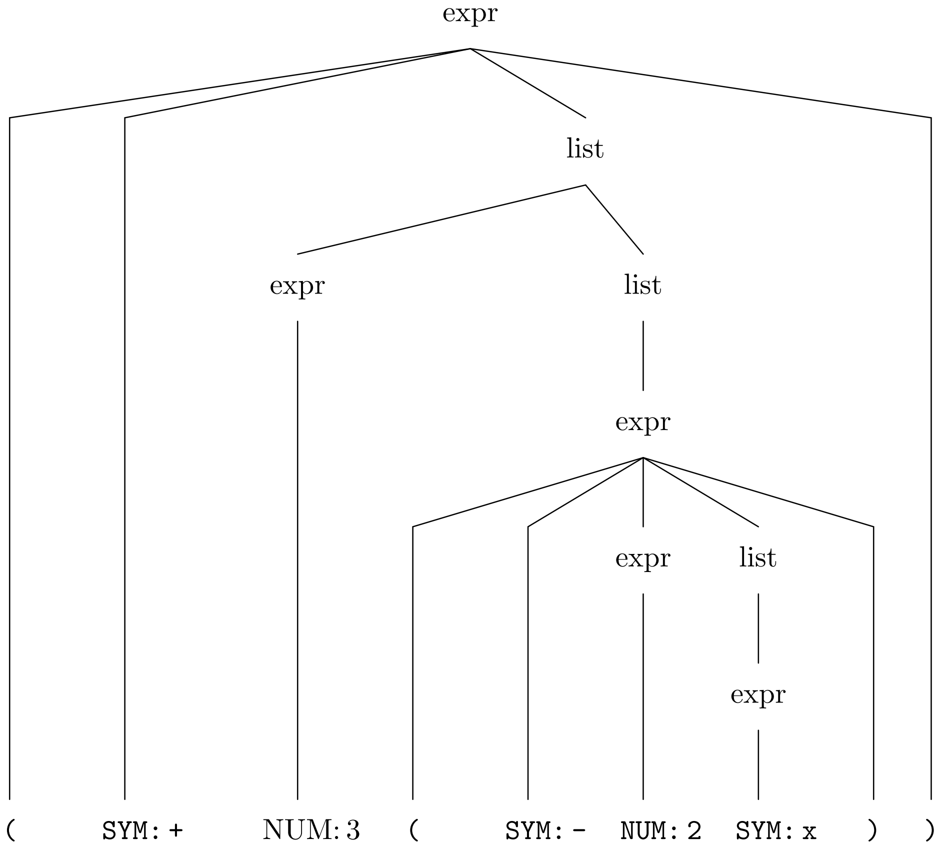


Figure 3.4: Parse tree for converting Lisp expression `(+ 3 (- 2 x))` to postfix using Grammar 3.4.

For this definition to work, we have to make an assumption about the *order* in which the semantic actions attached to the productions of the grammar are applied. Previously, we only needed to assume that the actions for all children of a node were applied before that of the node itself. Now we have to assume explicitly that we traverse the parse tree in postorder, thus executing the actions for a node's children from left to right and then immediately executing the parent node's action. If you do that for the tree in Figure 3.4, you'll see that no values will get attached to the internal nodes of the tree (there are no assignments to `$$`), but the (C-style) print statements that are executed will print "`3 2 x - +`".

I've described the application of semantic rules as tree traversals. Again, however, the actual implementation in real compilers typically performs the actions as it is forming the parse tree (and then dispenses with actually creating the parse tree!).

3.7 Some Common Idioms

Lists. Certain patterns of BNF arise over and over. This is particularly true of lists. Here are a few standard varieties of "list of items," where the symbols `item` and `sep` are either terminal symbols or are defined elsewhere. Some of these come in two versions, depending on the type of grammar you need and details of how you create lists. See §3.8 for more details. I have supplied sample semantic actions for actually accumulating a list, where `EMPTY` is the empty list, `prepend` adds an `item` to the beginning of a list, and `append` adds one to the end.

Sequence of 0 or more `items`.

```
/* Left-recursive version */
list0 : /* empty */           { $$ = EMPTY; }
      | list0 item            { $$ = append ($1, $2); }

/* Right-recursive version */
list0 : /* empty */           { $$ = empty list; }
      | item list0            { $$ = prepend ($1, $2); }
```

Sequence of 1 or more `items`.

```
/* Left-recursive version */
list1 : item                  { $$ = prepend ($1, EMPTY); }
      | list1 item            { $$ = append ($1, $2); }

/* Right-recursive version */
list1 : item                  { $$ = prepend ($1, EMPTY); }
      | item list1            { $$ = prepend ($1, $2); }
```

Sequence of 1 or more `items`, separated by `seps`.

```
/* Left-recursive version */
list1 : item                  { $$ = prepend ($1, EMPTY); }
      | list1 sep item        { $$ = append ($1, $3); }

/* Right-recursive version */
list1 : item                   { $$ = prepend ($1, EMPTY); }
      | item sep list1         { $$ = prepend ($1, $3); }
```

Sequence of 0 or more `itemss`, separated by `seps`.

```
list0 : /* empty */            { $$ = EMPTY; }
      | list1                  { $$ = $1; }
```

Infix expressions. Programming languages in the Algol family (in which I include C, C++, and Java) support some kind of infix expressions in which the operators have several different *precedences* and in which expressions involving a single type of operator *associate* or *group* in different ways. Thus, the Java expression "`x-y*z-q`" is interpreted as if written "`(x-(y*z))-q`" and not "`(x-y)*(z-q)`" or "`x-((y*z)-q)`," because '`*`' *has higher precedence* than '`-`' and '`-`' *associates to the left*.

To express these features in BNF, consider an expression such as

```
a-y/z-q/r**s**(t-1)/u
```

(the operator `**` is used in languages such as Fortran, Ada, or Python for exponentiation; it has higher precedence than `/` and associates to the right). This expression has the form of a list of one or more terms (indulge me and call them term_0s) separated by '`-`' operators. Each of these term_0s is a list of one or more of another kind of term (term_1s, let's say), separated by '`/`' operators, and each term_1 is a list of one or more of what are called *primaries,* separated by '`**`' operators. These primaries, finally, are numerals, identifiers, or entire expressions in parentheses. In other words, the grammar for these simple expressions is simply the repeated application of the "Sequence of 1 or more items, separated by..." syntax described above. We use the left- or right-recursive variations to capture grouping. Here's an example, in the form of a calculator:

Grammar 3.5. Simple expression grammar as a calculator

```
expr : term0                { $$ = $1; }
     | expr '-' term0       { $$ = $1 - $3; }
term0 : term1               { $$ = $1; }
     | term0 '/' term1      { $$ = $1 / $3; }
term1 : primary             { $$ = $1; }
     | primary "**" term1   { $$ = pow ($1, $3); }
primary : IDENTIFIER        { $$ = currentValue ($1); }
     | NUMERAL              { $$ = stringToNumber ($1); }
     | '(' expr ')'         { $$ = $2; }
```

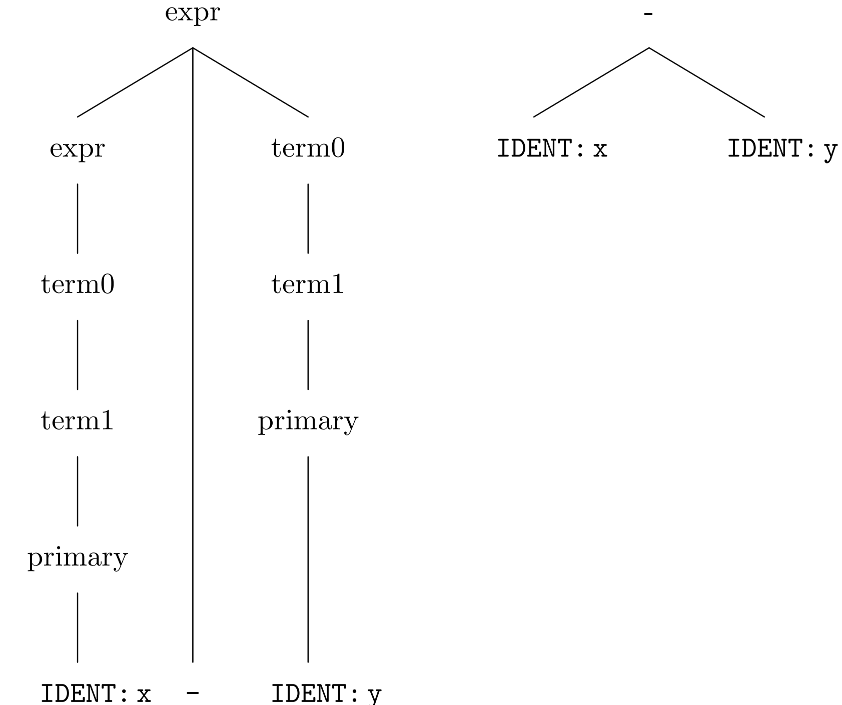
(The function names are intended to be self-explanatory.) Because the rule for '-' is left recursive (i.e., the symbol being defined appears on the left), in the expression `a-b-c`, `expr` must match `a-b` and `term0` must match `c` in the second rule for `expr`. This perfectly captures left associativity. The reverse is true for the '`**`' operator.

Grammar 3.5 calculates values, whereas a compiler writer would be more interested in producing ASTs, as in this version:

Grammar 3.6. Simple expression grammar producing ASTs

```
expr : term0                 { $$ = $1; }
     | expr '-' term0        { $$ = makeTree(MINUS, $1, $3); }
term0 : term1                { $$ = $1; }
     | term0 '/' term1       { $$ = makeTree(DIVIDE, $1, $3); }
term1 : primary              { $$ = $1; }
     | primary "**" term1    { $$ = makeTree (EXPON, $1, $3); }
primary : IDENT              { $$ = makeVar ($1); }
     | NUMERAL               { $$ = makeLiteral ($1); }
     | '(' expr ')'          { $$ = $2; }
```

(`makeTree`, `makeVar`, and `makeLiteral` create AST nodes; the details are unimportant.) The purely syntactic part of the grammar, as you can see, has not changed. By the way, this example also illustrates the difference between ASTs and parse trees. Here are the parse tree for "`x-y`" (on the left) and the AST calculated by this grammar:



3.8 Top-down Implementation

Context-free grammars resemble recursive programs: they are certainly recursively defined, and one can read a rule '$A : BCD$' as "to parse an A in the input, first parse a B, then a C, and then a D." This insight leads to a rather intuitive form of parsing known as *recursive descent*.

3.8.1 From grammars to programs: recursive descent

Consider the following grammar:

Grammar 3.7. A *really* simple expression grammar

```
p:  e '⊣'
e:  t e2          { $$ = addTerm ($1, $2); }
e2:  '+' e        { $$ = $2; }
e2:  ε            { $$ = NULL; }
t:  '(' e ')'     { $$ = $2; }
t:  i             { $$ = makeVar ($1); }
```

Here, 'i', '+', '(', ')', and '⊣' (end of file) are terminal symbols. We define

```
Tree addTerm (Tree T1, Tree T2)
{
   if (T2 == NULL)
      return T1;
   else
      return makeTree ('+', T1, T2);
}
```

Let's assume that there is a function `nextToken()` that returns the syntactic category of the next token of the input (one of `i`, `'+'`, `'('`, `')'`, and `'⊣'`), a function `scan()` that advances to the next token of input, and another version of scan that takes an argument and is defined:

```
Tree scan(x) {
   Tree t = semantic value of current token;
   if (nextToken () == x) {
      scan (); return t;
   } else
      ERROR ();
}
```

Our strategy will be to produce a set of functions, one for each nonterminal symbol. The body of each function will directly transcribe the grammar rules for the corresponding nonterminal. To start with, we'll ignore the semantic actions:

```
/* p: e '⊣' */
p () {  e (); scan ('⊣'); }

/* e: t e2 */
e () { t (); e2 (); }
```

```
/* e2: '+' e  | ϵ */
e2 () {
    if (nextToken () == '+') {
       scan ('+'); e ();
    } else if (nextToken () == '⊣' || nextToken () == ')') {
       /* ϵ */
    } else
      ERROR ();
}

/* t: '(' e ')' | i */
t () {
    if (nextToken () == '(') {
       scan ('('); e (); scan (')');
    } else if (nextToken () == i) {
       scan (i);
    } else
       ERROR ();
}
```

If you examine this closely, you should see each grammar rule transcribed into program text. Nonterminals on the right-hand sides turn into (recursive) function calls; terminals turn into calls to 'scan'. To parse a program (to start things off), you simply call 'p()'. If you trace the execution of this program for a given sentence and look at the order in which calls occur, comparing it to the parse tree for that sentence, you will see that the program essentially performs a *preorder walk* (also called "top down") of the parse tree, corresponding to a *leftmost derivation* of the tree.

Adding in semantic actions complicates things only a little. Now we make the functions return the semantic value for their tree:

```
Tree p () { Tree t1 = e (); scan ('⊣'); return t1; }

Tree e () {
   Tree t1, t2;
   t1 = t (); t2 = e2 (); return addTerm (t1,t2);
}

Tree e2 () {
    Tree t2;
    if (nextToken () == '+') {
       scan ('+'); t2 = e (); return t2;
    } else if (nextToken () == '⊣' || nextToken () == ')') {
       /* ϵ */  return NULL;
    } else
      ERROR ();
}
```

```
t () {
    Tree t1, t2;
    if (nextToken () == '(') {
       scan ('('); t2 = e (); scan (')'); return t2;
    } else if (nextToken () == i) {
       t1 = scan (i); return t1;
    } else
       ERROR ();
}
```

3.8.2 Choosing a branch: using FIRST and FOLLOW

I still haven't told you where the tests for the '**if**' statements in these functions came from. In general, you'll be faced with several rules for a given nonterminal—let's say $A : \alpha_1$, $A : \alpha_2$, etc.—where each α_i is a string of terminal and nonterminal symbols. For Grammar 3.7, for example, when A is `e2`, we have

$$\alpha_1 = \texttt{'+' e}$$
$$\alpha_2 = \epsilon$$

Recursive-descent parsers work by choosing ("predicting") which of the α_i to pursue based on the next, as yet unscanned input token. Assuming first that none of the α_i can produce the empty string, we can choose the branch of the function for A that corresponds to rule $A \rightarrow \alpha_i$ if the next symbol of input is in FIRST(α_i), which (when α_i does not produce the empty string) is defined as "the set of terminal symbols that can begin a sentence produced from α_i". As long as these sets of symbols do not overlap, we can unambiguously choose which branch to take.

Suppose one of the branches, say α_k, *can* produce the empty string, in which case we define FIRST(α_k) to contain the empty string as well as any symbols that can begin α_k. We should choose the α_k branch it if *either* the next input symbol is in FIRST(α_k) *or* the next input symbol is in FOLLOW(A), which is defined as "the set of terminal symbols that can come immediately after an A in some sentential form produced from the start symbol." Clearly, we're in trouble if more than one α_i can produce the empty string, so for this translation to recursive descent to work, we must insist that at most one branch can produce the empty string.

So in summary, if our grammar contains the following production for nonterminal A (possibly among others):

```
A : α1 ... αn
```

then in the procedure we write for A, there will be a piece that says (in effect)

```
if (nextToken () ∈ FIRST(α1 ... αn)
    || (ϵ ∈ FIRST(α1 ... αn) && nextToken () ∈ FOLLOW(A))) {
    α'1; ... α'n;
}
```

where each α_i' is the program for recognizing α_i. That is,

- If α_i is a terminal symbol, then α_i' is `scan(`α_i`)`.
- If α_i is a nonterminal symbol, then α_i' is α_i`()`.

If there is no overlap in any of the sets of terminal strings produced by the procedure above, then we say that *the grammar is LL(1)*, meaning that it can be parsed *L*eft to right to give a *L*/eftmost derivation, looking ahead at most *1* symbol of input.

3.8.3 Computing FIRST

For any sequence of symbols (terminals and nonterminals), $x = x_1 x_2 \cdots x_n$, we can define FIRST(x) recursively as follows:

$$\mathrm{FIRST}(x_1 x_2 \cdots x_n) = \left\{ \begin{array}{ll} \{\epsilon\}, & \text{if } n = 0; \\ FIRST(x_1), & \text{if } n \geq 1 \text{ and } \epsilon \notin FIRST(x_1); \\ (\mathrm{FIRST}(x_1) - \{\epsilon\}) \cup \mathrm{FIRST}(x_2 \cdots x_n), & \\ & \text{otherwise.} \end{array} \right\} \tag{3.1}$$

So once we have computed FIRST(a) for every individual symbol a, we can compute FIRST(x) for any *sequence* of symbols $x = x_1 \cdots x_n$. For example, if we know that FIRST(b) = {`'+'`, `'-'`, ϵ} and that FIRST(c) = {`'ID'`, `'('`}, then we can compute

$$\begin{aligned} FIRST(bcb) &= FIRST(b) - \{\epsilon\} \cup \mathrm{FIRST}(cb) \\ &= \{\texttt{'+'}, \texttt{'-'}\} \cup FIRST(c) \\ &= \{\texttt{'+'}, \texttt{'-'}, \texttt{'ID'}, \texttt{(}\} \end{aligned}$$

Thus, all that remains is to calculate FIRST on individual symbols.

Our procedure for doing so is an example of a *fixed-point iteration:* starting with an initial guess, we apply an updating algorithm that produces a closer guess, and keep repeating this step until our guess stops changing. In detail:

```
# Compute initial guess
for every terminal symbol τ:
    initialize FIRST(τ) = { τ }
for every nonterminal symbol A:
    initialize FIRST(A) = { }

# Compute fixed point
while True:
    for every production  A:  α1 ··· αn  in the grammar:
        Replace the value of FIRST(A) with the current guess of FIRST(α1 ··· αn
    if the values of FIRST were not changed in the iteration:
        break
```

Applied to Grammar 3.7, we get the following sequence of guesses:

FIRST	Iteration #			
	0	1	2	3
'⊣'	{'⊣' }	{'⊣' }	{'⊣' }	
'+'	{'+' }	{'+' }	{'+' }	{'+' }
'('	{'(' }	{'(' }	{'(' }	{'(' }
')'	{')' }	{')' }	{')' }	{')' }
i	{i }	{i }	{i }	{i }
p	{}	{}	{}	{'(', i }
e	{}	{}	{'(', i }	{'(', i }
e2	{}	{'+', ϵ }	{'+', ϵ }	{'+', ϵ }
t	{}	{'(', i }	{'(', i }	{'(', i }

3.8.4 Computing FOLLOW

As with FIRST, we use a fixed-point iteration to compute the value of FOLLOW for any grammar, assuming we've already computed FIRST. Again, we start with a guess at FOLLOW (namely, that FOLLOW(A) = {} for all nonterminals), and then refine it. Here's the full process:

```
for every nonterminal, A:
    initialize FOLLOW(A) = { }

while True:
    for every production of the form  A : α1 ··· αk B αk+2 ··· αn
      where B is a nonterminal:
          Replace the value of FOLLOW(B) with
              FOLLOW(B) ∪ FIRST(αk+2 ··· αn) − {ϵ}
          if ϵ ∈ FIRST(αk+2 ··· αn):
              Replace the value of FOLLOW(B) with FOLLOW(B)     ∪
FOLLOW(A)
    if the values of FOLLOW were not changed by the iteration
        break
```

3.8.5 Dealing with non-LL(1) grammars.

You will have noticed, no doubt, that Grammar 3.7 is a bit odd, compared to a normal expression grammar. For one thing, it looks rather contorted, and for another, it groups expressions to the right rather than the left (it treats 'a+b+c' as 'a+(b+c)'). However, if I try to write a more natural grammar—for example, more along the lines of Grammar 3.5—I run into trouble:

Grammar 3.8. A simple non-LL(1) grammar

```
A. p:  e '⊣'
B. e:  e '+' t     { $$ = makeTree ('+', $1, $3); }
C. e:  t           { $$ = $1; }
D. t:  '(' e ')'   { $$ = $2; }
E. t:  i           { $$ = $1; }
```

The problem is that the test to determine whether to apply the first or second rule for ‘e’ breaks down: the same symbols can start an ‘e’ as can start a ‘t’. Another problem is that the grammar is *left recursive:* from ‘e’, one can produce a sentential form that begins with ‘e’; in a program this causes an infinite recursion. Both of these cause the grammar to be non-LL(1).

Some textbooks go into a great deal of hair to get around problems like this. Frankly, I prefer to take a more practical stance. The pattern above is quite common, and is easily dealt with by means of a loop:

```
Tree e () {
   Tree t1, t3;
   t1 = t ();
   while (nextToken () == '+') {
      scan ('+'); t3 = t (); t1 = makeTree ('+', t1, t3);
   }
   return t1;
}
```

3.9 General context-free parsing

It turns out that there are completely general algorithms that work for all context-free grammars, ambiguous or not, and find all possible parses (and hence, parse trees and corresponding derivations) of any input string. To motivate the algorithm, let’s consider again the approach used for recursive descent, but with a few modifications. Consider the problem of parsing an input, $c_1 c_2 \cdots c_n$, using a particular grammar. In what follows, I’ll use a few notational conventions:

- Capital latin letters denote nonterminal grammar symbols.
- Lower case latin letters denote terminal or nonterminal grammar symbols.
- Lower case greek letters denote strings (possibly empty) of terminal and nonterminal grammar symbols.

The start symbol in our grammar will have a single rule of the form $p : \gamma \dashv$, and will appear only on the left side of this one rule. The ‘$\dashv$’ symbol will appear only at the end of any input string.

3.9.1 An abstract algorithm

Let’s first reformulate simple recursive-descent recognition as a single procedure:

```
def parse (A, S):
   """Assuming A is a nonterminal and S = c1c2...cn is a string,
      return integer j such that A can derive the string c1...cj."""
   Choose production 'A: α1α2···αm' for A (nondeterministically)
   j = 0
   for x in α1, α2, ···, αm:
       if x is a terminal:
          if x == cj+1:
              j += 1
          else:
              GIVE UP
       else:
          j += parse (x, cj+1···cn)
   return j
```

Given its comment, string S will be accepted by our grammar iff the call `parse(p,S)` successfully returns (where again, `p` is our grammar's start symbol). The idea here is the that the special "Choose" step indicates a point where, like an NFA, our program can pursue multiple options. If there is *some* sequence of choices the program can take when it encounters this step that allows a call `parse`(X, S) to return without giving up, then there is a derivation of string S from nonterminal X. Some paths through the program may give up, and others may loop endlessly (such as in the case of left recursion). But as long as one gets all the way through execution, we'll say that the parse succeeds.

We can add semantic actions to this framework without too much trouble:

```
def parse2 (A, S):
   """Assuming A is a nonterminal and S = c1c2...cn is a string,
      return a pair (j,v), where j is an integer such that A
      can derive the string c1...cj, and v is a
      semantic value computed for A from this string."""
   Choose production 'A: α1α2···αm' for A (nondeterministically),
       where the associated semantic value is f([v1,...,vm]) if
       vi is the semantic value attached to αi
   j = 0
   vals = []
   for x in α1, α2, ···, αm:
       if x is a terminal:
          if x == cj+1:
              j += 1
          else:
              GIVE UP
       else:
          jx, vx = parse2 (x, cj+1···cn)
          j += jx
          vals.add (vx)
   return j, f(vals)
```

For now, however, we will ignore semantic actions and values and concentrate on finding derivations or parse trees.

Either of these abstract procedures would solve the general parsing problem, if we could just implement them. An obvious approach would be to use a multiprocessor, and spawn a new process for each production at the "Choose" step. Unfortunately, this would generally require an unbounded number of processors (or alternatively, an unbounded amount of work for a single processor). We need something considerably more efficient. A form of memoization provides an answer.

First, let's rewrite `parse` to be purely recursive (no explicit loops), so that we can use its arguments to index our collection of memoized values. It will be convenient to fix a particular input string, $c_1 \cdots c_n$, and write the function like this:

```
def parse (X :  α • β, s, k):
    """Assuming 0 ≤ s ≤ k ≤ n X :  αβ is a production in the grammar,
    and α =*=> c_s ··· c_k, return j such that β =*=> c_{k+1} ··· c_j, and
    thus X ⇒ αβ =*=> c_s ··· c_j."""
    if β is empty:
        return k;
    Assume β has the form yδ
    if y is a terminal:
        if y == c_{k+1}:
            return parse (X :  αy • δ, s, k+1)
        else:
            return GIVE UP
    else:
        Choose a production y :  κ in the grammar
        j = parse (y :  • κ, k, k)
        return parse(X : αy • δ, s, j)
```

Now we can tell if our input string $c_1 \cdots c_n$ is in the language by seeing if `parse` $(\{p: \ \bullet\,\gamma \dashv\}$, `0, 0)` returns a value.

3.9.2 Earley's algorithm

To implement this version of our abstract parsing algorithm, we will keep track of *all* possible distinct calls (that is, calls with distinct arguments) that might arise during the application of `parse` to a given input string. The data structure used to do this is traditionally known as a *chart*, and the resulting parsers are called *chart parsers.* The particular chart parser we'll look at here (from Berkeley, as it turns out) is known as Earley's Algorithm[4].

As an example, consider the following grammar:

Grammar 3.9. Another expression grammar

```
p : e '⊣'
e : t
  | e '-' t
```

[4] J. "An efficient context-free parsing algorithm," *Comm. ACM* **13:2** (1970), pp. 94–102.

I   -   I   ⊣

0	1	2	3	4
a. `p: •e '⊣', 0` *b.* `e: •e '-' t, 0` *c.* `e: •t, 0` *d.* `t: •f, 0` *e.* `f: •I, 0`	*f.* `f: I•, 0` *g.* `t: f•, 0` *h.* `e: t•, 0` *i.* `e: e •'-' t, 0`	*j.* `e: e '-' •t, 0` *k.* `t: •f, 2` *l.* `f: •I, 2`	*m.* `f: I•, 2` *n.* `t: f•, 2` *o.* `e: e '-' t•, 0` *p.* `p: e •'⊣', 0`	*q.* `p: e '⊣'•,0`

Table 3.1: Chart from parsing "`I-I⊣`" showing just successful entries.

I   -   I   ⊣

0	1	2	3	4

a. `p: •e '⊣', 0`
b. `e: •e '-' t, 0`
c. `e: •t, 0`
d. `t: •f, 0`
e. `f: •I, 0`
`t: •t '/' f, 0`
`f: •'(' e ')', 0`
	f. `f: I•, 0`
	g. `t: f•, 0`
	h. `e: t•, 0`
	i. `e: e •'-' t, 0`
	`t: t •'/' f, 0`
	`p: e •'⊣', 0`
		j. `e: e '-' •t, 0`
		k. `t: •f, 2`
		l. `f: •I, 2`
		`t: •t '/' f, 2`
		`f: •'(' e ')', 2`
			m. `f: I•, 2`
			n. `t: f•, 2`
			o. `e: e '-' t•, 0`
			p. `p: e •'⊣', 0`
			`t: t •'/' f, 2`
				q. `p: e '⊣'•,0`

Table 3.2: Chart from parsing "`I-I⊣`" showing all entries, including those correspond to calls that eventually "give up" or loop forever.

```
t : f
  | t '/' f
f : I
  | '(' e ')'
```

where `p` is the start symbol, and '`I`' (for identifier) and quoted symbols are terminals. We'll parse the string "`I-I⊣`" by calling `parse(p :  •e ⊣, 0, 0)`.

Table 3.1 charts only the calls that succeed. We record a call parse($X: \alpha \bullet \beta, s, k$) with an entry of the form "$X: \alpha \bullet \beta, s$" in column k of the chart. Each such entry is known as an *item,* and a column of items as an *item set.* In Table 3.1, we've used letters to the left of the items to indicate the order in which the calls they record occur in the abstract program. The headings on the columns are the values of k, and, for reference, I've placed the terminal symbols c_{k+1} above the divisions between columns k and $k+1$.

Since computers don't have the prescience needed to predict exactly which calls will succeed, Earley's algorithm computes *all* calls that could be made by *some* sequence of choices in the abstract algorithm, filling in the columns of the chart and processing input symbols one at a time from left to right. Table 3.2 shows the total set of computed items. Those items that don't have enumerating letters in front represent calls that eventually fail. The algorithm avoids following infinite chains of calls because the chart represents sets of items: adding the same item twice does not change the chart.

You can think of the actual computation as another fixed-point iteration. We start with a chart that contains only the single item `p:  •e '⊣'` in column 0. Then we repeat the following (taken pretty much directly from the abstract algorithm) until the chart does not change for any of the possible choices:

```
Choose an item X : α • yδ, s from some column k
if y is a terminal and y == c_{k+1}:
    add item X : αy • δ, s to item set k+1 (if not already present)
else:
    Choose a production y :  κ in the grammar
    add item y : •κ, k to item set k (if not already present)
    if there is an item y : κ•, k in some item set j:
        add item X : αy•δ, s to item set j (if not already present)
```

Earley's algorithm does this efficiently, working on one column until the chart ceases to change and then moving on to the next.

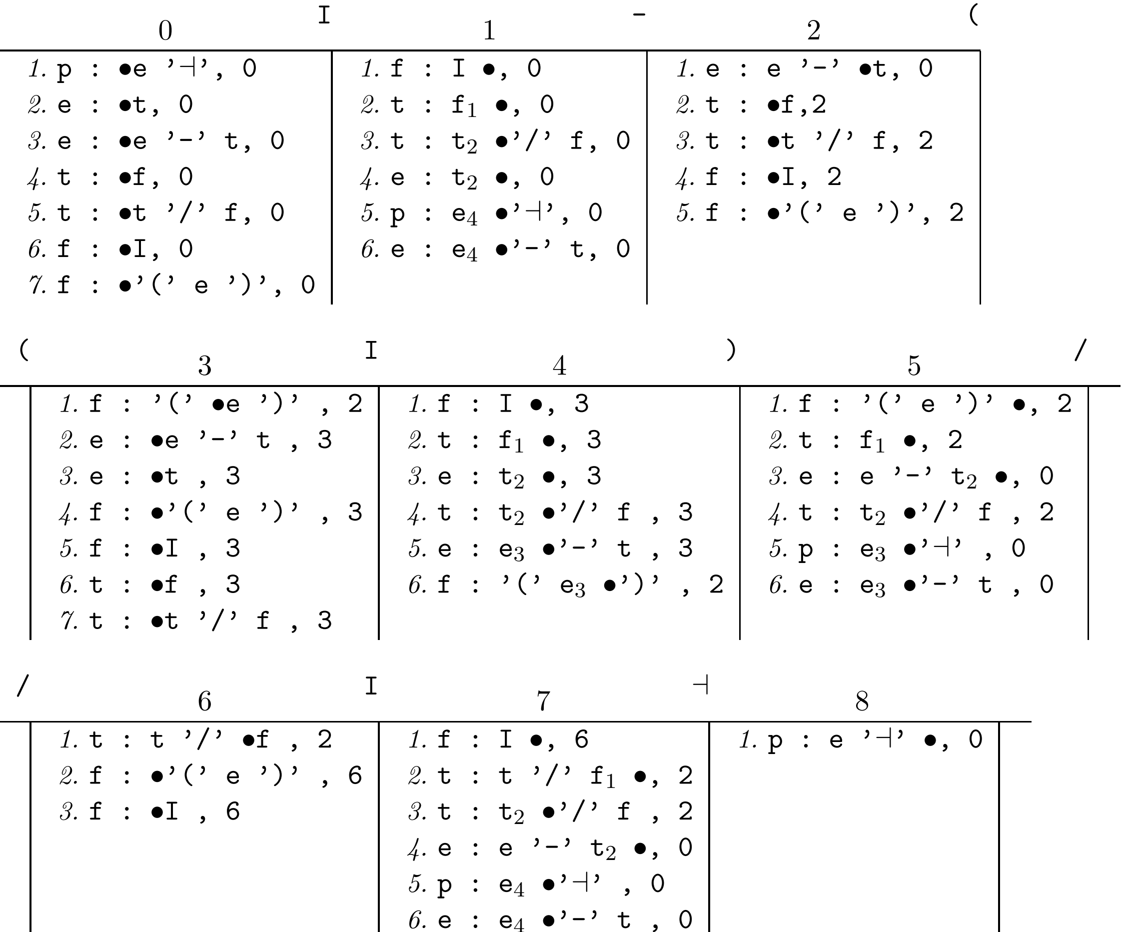
If the total number of symbols in all productions in the the grammar (including both those to the left and right of the colon) is K_G (a constant), then number of items in an entire chart is evidently $\leq K_G(n+1)(n+2)/2 \in O(n^2)$. With careful implementation, each of those items can get shifted into up to n item sets ("shifted into" means "added to after shifting the dot to the right by one"), for a total time of $O(n^3)$. For unambiguous grammars, this time bound tightens to $O(n^2)$, and for the *deterministic grammars* of most programming languages (which we'll discuss in later sections), the bound is $O(n)$, making this a rather efficient algorithm (if you are willing to ignore constant factors anyway).

Table 3.3 shows a complete run of the machine over another string, "`I-(I)/I`⊣." This table includes a new piece of notation. If you consider how the algorithm works, you'll see that whenever a dot appears in an item, I, immediately after a nonterminal symbol, A, there has to be another item of the form $A : \xi\bullet,\ j$ in that same item set. Such an item is called a *handle;* it tells us that there is a derivation of A from the input symbols $c_{j+1}\cdots c_k$, where k is the number of item set containing the handle. The semantic value for A produced by this derivation is a value that can eventually be attached to the A that precedes the dot in item I. In Table 3.3, we've numbered all the items in a given item set and subscripted each occurrence of A to the left of a dot with the handle items in that set that caused it to appear. This information is actually redundant, since we could reconstruct the information, and is unnecessary for merely recognizing valid strings, but it will be useful for extracting a derivation or parse tree out of the item sets.

3.9.3 Extracting the derivation(s)

So far, we've just used Earley's algorithm to recognize valid strings in a language. But as we've argued before, it's really the derivation that is of interest to us, since it allows us to tell what actions to trigger on what values. It's not hard to extract a derivation once we've got a complete sequence of item sets. In Figure 3.3, we added some subscripts to indicate why items got shifted. We can now use these.

Start at the last item set, which for a valid input will always contain exactly one item (or no items for an invalid input). This item is *1.* `p : e '⊣' •, 0`, which—since it appears in item set #8—tells us that the production matches all the input from position 0 (before the first charcter) up through position 8 (just after the last character). The symbol just before the dot is a terminal symbol ('⊣'), so we know to look one item set back to find out where the `e` comes from.



0	I	1	-	2	(
1. p : •e '⊣', 0		*1.* f : I •, 0		*1.* e : e '-' •t, 0	
2. e : •t, 0		*2.* t : f$_1$ •, 0		*2.* t : •f,2	
3. e : •e '-' t, 0		*3.* t : t$_2$ •'/' f, 0		*3.* t : •t '/' f, 2	
4. t : •f, 0		*4.* e : t$_2$ •, 0		*4.* f : •I, 2	
5. t : •t '/' f, 0		*5.* p : e$_4$ •'⊣', 0		*5.* f : •'(' e ')', 2	
6. f : •I, 0		*6.* e : e$_4$ •'-' t, 0			
7. f : •'(' e ')', 0					

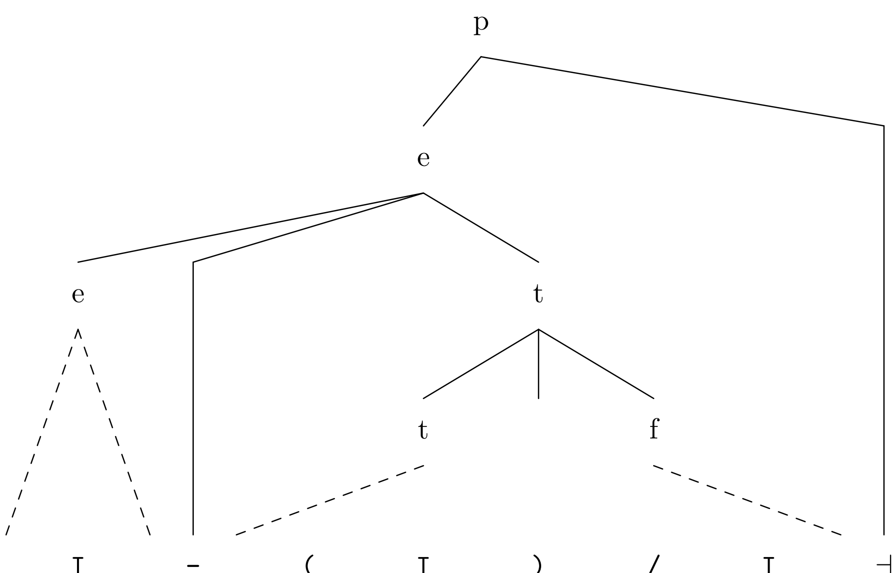
(	3	I	4	)	5	/
	1. f : '(' •e ')' , 2		*1.* f : I •, 3		*1.* f : '(' e ')' •, 2	
	2. e : •e '-' t , 3		*2.* t : f$_1$ •, 3		*2.* t : f$_1$ •, 2	
	3. e : •t , 3		*3.* e : t$_2$ •, 3		*3.* e : e '-' t$_2$ •, 0	
	4. f : •'(' e ')' , 3		*4.* t : t$_2$ •'/' f , 3		*4.* t : t$_2$ •'/' f , 2	
	5. f : •I , 3		*5.* e : e$_3$ •'-' t , 3		*5.* p : e$_3$ •'⊣' , 0	
	6. t : •f , 3		*6.* f : '(' e$_3$ •')' , 2		*6.* e : e$_3$ •'-' t , 0	
	7. t : •t '/' f , 3					

/	6	I	7	⊣	8
	1. t : t '/' •f , 2		*1.* f : I •, 6		*1.* p : e '⊣' •, 0
	2. f : •'(' e ')' , 6		*2.* t : t '/' f$_1$ •, 2		
	3. f : •I , 6		*3.* t : t$_2$ •'/' f , 2		
			4. e : e '-' t$_2$ •, 0		
			5. p : e$_4$ •'⊣' , 0		
			6. e : e$_4$ •'-' t , 0		

Table 3.3: Complete parse of 'I-(I)/I⊣' using Earley's algorithm and Grammar 3.9. The subscripts on just-shifted nonterminal symbols in an item indicate which handles (denoted by items with dots on the right) caused that item to get shifted. We use them later to reconstruct the derivation or parse tree.

We find there the item that got shifted on '⊣', namely *5.* `p : e`$_4$ `•'⊣' , 0`, which tells us to look at item 4 in the same set to find the derivation of 'e'. That item is *4.* `e : e '-' t`$_2$ `•, 0`, where says that this topmost 'e' that we are looking at came from the production `e : e '-' t`, and that these symbols match input positions after 0 up to and including 7 (the number of the item set we are now considering). We now trace this item backwards through the parse, starting with the 't' just before the dot. Its subscript tells us that we got this from item #2 in item set 7, which is *2.* `t : t '/' f`$_1$ `•, 2`. That is, this `t` matches all the text from after character position 2 up to and including character position 7.

Let's take a look at where things stand at the moment. We've built a partial tree that looks like this:



At each point, the item we are considering tells us where the text it covers starts (the position after the comma) and where it ends (the number of the item set that contains the item). We can deduce everything we need to parse the string, so as to arrive at the parse tree in Figure 3.5

3.9.4 Dealing with epsilon rules

Grammar 3.9 did not include empty (epsilon or ϵ) rules, but they don't really change things if you follow the procedure literally enough. Consider a simpler grammar:

Grammar 3.10. Still another expression grammar

```
p : e ⊣
e : s I t
t :
t : '/' e
s :
s : '-'
```

I have chosen to indicate epsilon productions without writing ϵ just to clarify that there is nothing on the right sides of those productions. For the string '`I/-I⊣`,' the

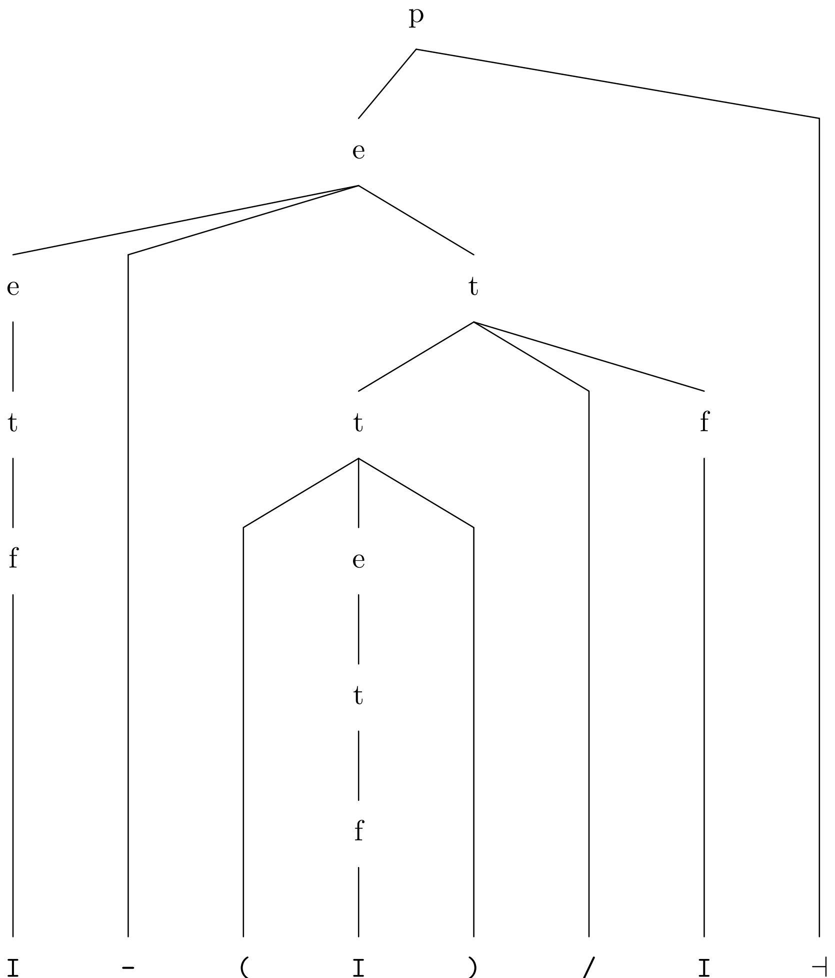


Figure 3.5: Completed parse tree for `I-(I)/I⊣`, as extracted from the item sets in Figure 3.3.

procedure from §3.9.2 gives us the following contents for column 0:

1. p : • e ⊣, 0
2. e : • s I t, 0
3. s : •, 0
4. s : • '-', 0
5. e : s_3 • I t, 0

Item #5 results from shifting item #2. The only difference from prior examples is that we just happen to take the item that we shift from the same itemset we shift it into.

Continuing with column 1:

1. e : s I • t, 0
2. t : •, 1
3. t : • '/' e, 1
4. e : s I t_2 •, 0
5. p : e_4 • ⊣, 0

and column 2:

1. t : '/' • e, 1
2. e : • s I t, 2
3. s : •, 2
4. s : • '-', 2
5. e : s_4 • I t, 2

This time, items 3 and 5 will turn out to be dead ends, since the next input character is '-'.

Here are the remaining item sets:

-	3	I	4	⊣	5
	1. s : '-' •, 2		*1.* e : s I •t, 2		IT1p : e ⊣•
	2. e : s_1 •I t, 2		*2.* t : •, 4		
			3. t : '/' e, 4		
			4. e : s I t_2 •, 2		
			5. t : '/' e_4 •, 1		
			6. e : s I t_5 •, 0		
			7. p : e_6 •⊣, 0		

3.9.5 The effects of ambiguity

Earley's algorithm can deal with ambiguous grammars without any essential changes. Let's look at a particularly simple example:

Grammar 3.11. An ambiguous expression grammar

```
p : e ⊣
e : I
e : e '-' e
```

Running the algorithm on the input '`I-I-I⊣`' gives us the results in Figure 3.6. You can see the difference from the previous run in column 5. There, item #3 gets shifted into the item set from two different places, as indicated by the two subscripts on the final 'e'. That nonterminal can either match the text matched by item #1—and the ',4' at the end of that item tells us that it matches characters after character 4 through character 5, or in other words, "`I`"—or item #2, which would be "`I-I`". This gives us two possible trees, as shown at the bottom of the figure.

3.10 Deterministic Bottom-up Parsing

Earley's algorithm is not generally used for programming languages, because, being explicitly designed, they generally avoid ambiguity and other features that would require fully general context-free parsing. These grammars are *deterministic* in the sense that during simple left-to-right processing, it is always possible to tell from the terminal symbols already processed, with possibly a little "peeking" at some fixed number of terminals ahead (usually one), the portions of the parse tree that derive the input seen so far. That is, it is unnecessary to keep other possible derivations around speculatively in case later parts of the input should prove them necessary. Deterministic langauges are unambiguous and require only space proportional to the longest sentential form encountered in the derivation.

3.10.1 Shift-reduce parsing

Let's again consider Grammar 3.8 from above, and look at a *reverse* derivation of the string 'i+(i+i)⊣':

```
 1.  i       + ( i + i ) ⊣
 2.  t       + ( i + i ) ⊣
 3.  e + ( i       + i ) ⊣
 4.  e + ( t       + i ) ⊣
 5.  e + ( e + i       ) ⊣
 6.  e + ( e + t       ) ⊣
 7.  e + ( e )           ⊣
 8.  e + t               ⊣
 9.  e ⊣
10.  p
```

Read from the bottom up, this is a straightforward rightmost derivation, but with a mysterious gap in the middle of each sentential form. The gap marks the position of the *handle* in each sentential form—the portion of the sentential form up to and including the symbols produced (reading upwards) by applying the next production

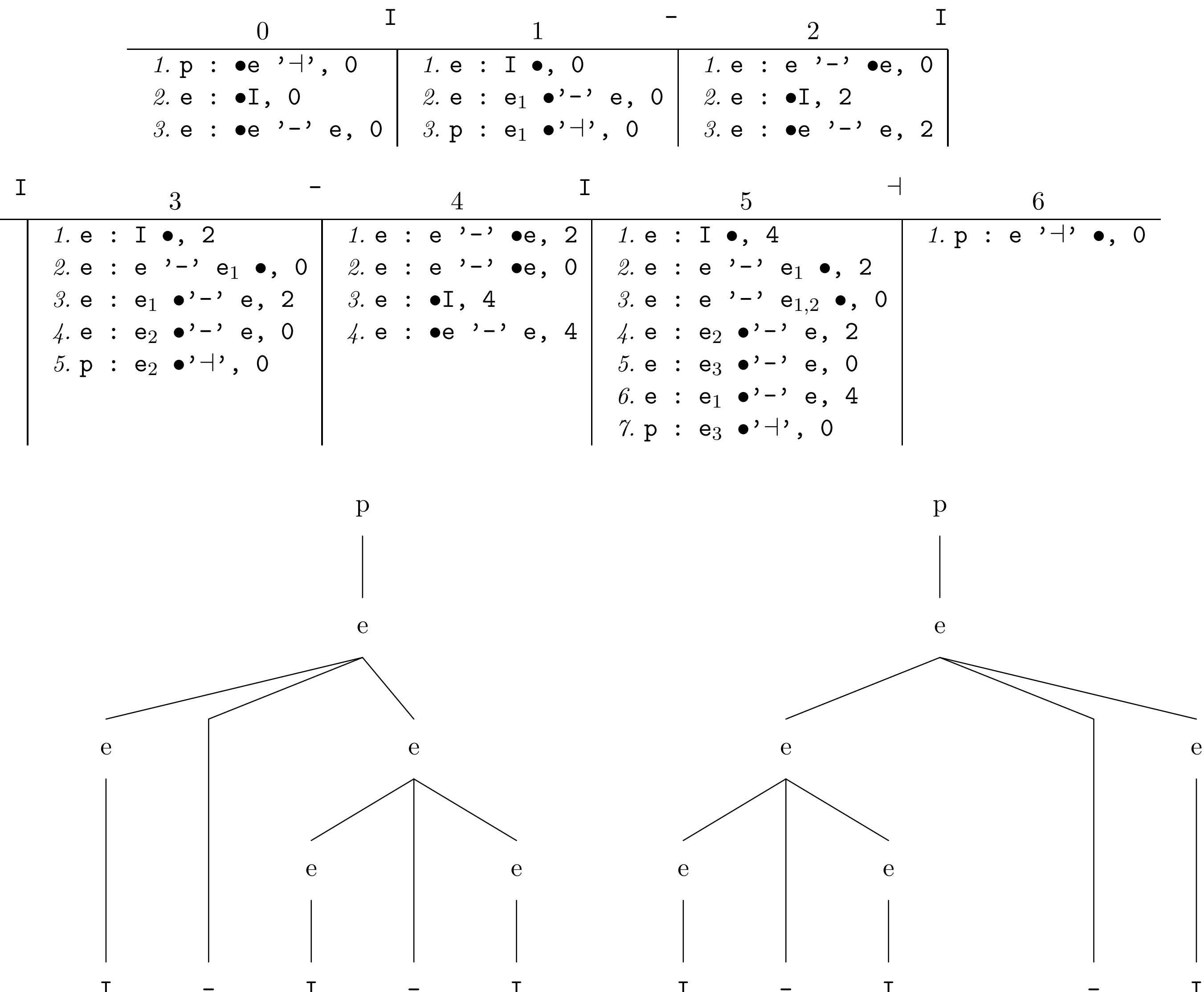


Figure 3.6: Parse of 'I-I-I⊣' by Grammar 3.11, and resulting parse trees.

or (reading downwards) the symbols about to be reduced by applying the next reverse production. Reading this downwards, you see that the gap proceeds through the input (i.e., the sentence to be parsed) from left to right. We call the symbols left of the gap "the stack" (right symbol on top) and the symbol just to the right of the gap "the lookahead symbol".

To add semantic actions, we just apply the rules attached to a given production each time we use it to reduce, attaching the resulting semantic value to the resulting nonterminal instance. For example, suppose that the semantic values attached to the three 'i's in the preceding example are leaf nodes 1, 2, and 3, respectively. Then, using $x : E$ to mean "semantic value E is attached to symbol x," we have the following parse

```
 1.  i:1       + ( i:2 + i:3 ) ⊣
 2.  t:1       + ( i:2 + i:3 ) ⊣
 3.  e:1 + ( i:2       + i:3 ) ⊣
 4.  e:1 + ( t:2       + i:3 ) ⊣
 5.  e:1 + ( e:2 + i:3       ) ⊣
 6.  e:1 + ( e:2 + t:3       ) ⊣
 7.  e:1 + ( e:(+ 2 3) )       ⊣
 8.  e:1 + t:(+ 2 3)           ⊣
 9.  e:(+ 1 (+ 2 3)) ⊣
10.  p
```

Initially, only the terminal symbols have semantic values (as supplied by the lexer). Each reduction computes a new semantic value for the nonterminal symbol produced, as directed by the grammar.

With or without semantic actions, the process illustrated above is called "shift-reduce parsing." Each step consists either of *shifting* the lookahead symbol from the remaining input (right of the gap) to the top of the stack (left of the gap), or of *reducing* some symbols (0 or more) on top of the stack to a nonterminal according to one of the grammar productions (and performing any semantic actions). Each line in the examples above represents one reduction, plus some number of shifts. For example, line 5 represents the reduction of 't' to 'e', followed by the shift of '+' and 'i'.

In all these grammars, it is convenient to have the end-of-file symbol ('⊣') and the start symbol (in the examples, 'p') occur in exactly one production. This first production then has no important semantic action attached to it. This means that as soon as we shift the end-of-file symbol, we have effectively accepted the string and can stop.

We could completely mechanize the process of shift-reduce parsing if we could determine when we have a handle on the stack, and which handle we have. The algorithm then becomes

```
while ⊣ not yet shifted:
   if handle is on top of the stack:
      reduce the handle
```

```
else:
   shift the lookahead symbol
```

Actually, this procedure applies only when we have an unambiguous grammar. With an ambiguous grammar, we have to accommodate cases where there may be multiple handles to choose from or where a certain string can also be parsed if we *don't* reduce a certain handle, but wait for one that shows up later after some more shifting or other reductions. Even for unambiguous grammars, the procedure works best if we can tell when we have a handle based only on the contents of the stack and the lookahead symbol (without looking further ahead in the input). Let's work up to this gradually.

3.10.2 Recognizing possible handles: the LR(0) machine

It turns out, interestingly enough, that although context-free languages cannot be recognized in general by finite-state machines, their rightmost handles *can* be recognized. That is, we can build a DFA that allows us to perform the "handle is on top of the stack" test by pushing the stack through the DFA from bottom to top (left to right in the diagrams above). This DFA will also tell us which production to use to reduce the handle.

To do this, I will first show how to construct a "handle grammar"—a grammar that describes all possible handles. The terminal symbols of this grammar will be all the symbols (terminal and nonterminal) of the grammar we are trying to parse. I will then show how to convert the handle grammar into an NFA, after which the usual NFA-to-DFA construction will finish the job.

The nonterminals of the handle grammar for Grammar 2 are H_p, H_e, and H_t. H_p means "a handle that occurs during a rightmost derivation of a string from 'p' ". Likewise, H_e means "a handle that occurs during a rightmost derivation of a string from 'e', and so forth. Let's start with H_p. There are two cases: either the stack consists of the handle "e ⊣" and we are ready for the final reduction, or we are still in the process of forming the 'e' and haven't gotten around to shifting the '⊣' yet—in other words, we have some handle that occurs during a derivation of some string from 'e'. Such a handle is supposed to be described by H_e. This gives us the rules:

H_p: `e` ⊣
H_p: H_e

(Again, the symbols 'e' and '⊣' are *both* terminal symbols here; H_p is the nonterminal.)

Now let's consider H_e. From the grammar, we see that one possible handle for 'e' is 't'. It is also possible that we are part way through the process of reducing to this 't', so that we have the two rules

H_e: `t`
H_e: H_t

Likewise, we also see that another possible handle is 'e + t'. It is therefore possible to have 'e +' on the stack, followed by a handle for an as-yet-incomplete 't', or finally, it is possible that the 'e' before the '+' is not yet complete. These considerations lead to the following rules for H_e:

H_e: `e '+' t`
H_e: `e '+'` H_t
H_e: H_e

(The last rule is useless, but harmless).

Continuing, the full handle grammar looks like this:

H_p: `e ⊣ |` H_e
H_e: `t |` H_t
H_e: `e + t | e +` H_t `|` H_e
H_t: `i`
H_t: `( e ) | (` H_e

This grammar has a special property: the only place that a nonterminal symbol appears on a right-hand side is at the end (the nonterminals in the handle grammar are H_p, H_e, and H_t). The grammar, in other words, is a regular grammar, as described in §3.5. This is significant because grammars with this property can be converted into NFAs very easily.

Consider, for example, the grammar

```
A: x B
B: y A
B: z
```

The NFA in Figure 3.7 recognizes this grammar. We simply translate each nonterminal into a state, and transfer to that state whenever a right-hand side calls for recognizing the corresponding nonterminal. The translation in the figure uses some epsilon transitions where they really could be avoided, because this will be convenient in the production of a machine for the handle grammar.

When we use this technique to convert the handle grammar into a NFA, we get the machine shown in Figure 3.8. I have put labels in the states that hint at why they are present. For example, the state labeled "`e: e •+ t`" is supposed to mean "the state of being part way through a handle for the production "`e: e + t`" just before the '+'." These labels (productions with a dot in them) are known as *LR(0) items*. Some of the states have the same labels; however, if you examine them, you will see that any string that reaches one of them also reaches the other, so that the identical labels are appropriate. There are no final states mentioned, because all the information we'll need resides in the labels on the states.

The final step is to convert the NFA of Figure 3.8 into a DFA (so that we can easily turn it into a program). We use the set-of-states construction that you learned previously. The labels on the resulting states are sets of LR(0) items; I leave out the labels H_p, H_e, and H_t, since they turn out to be redundant. You can verify that we

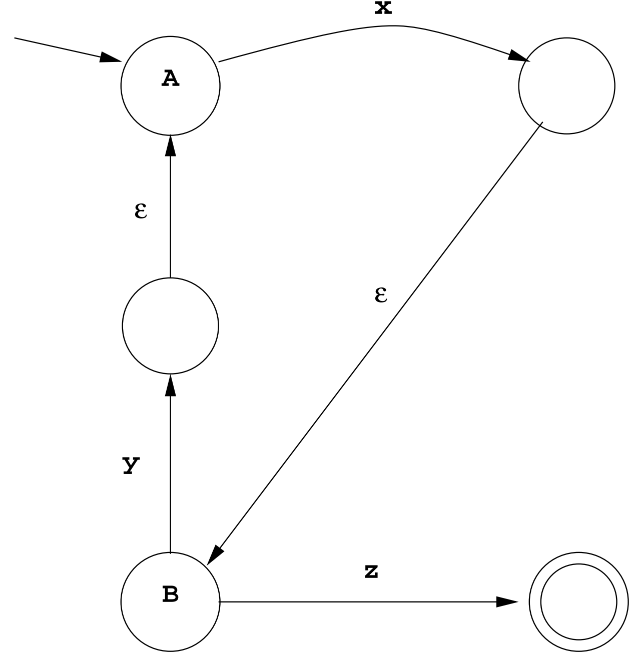


Figure 3.7: Example of converting a regular grammar into a NFA.

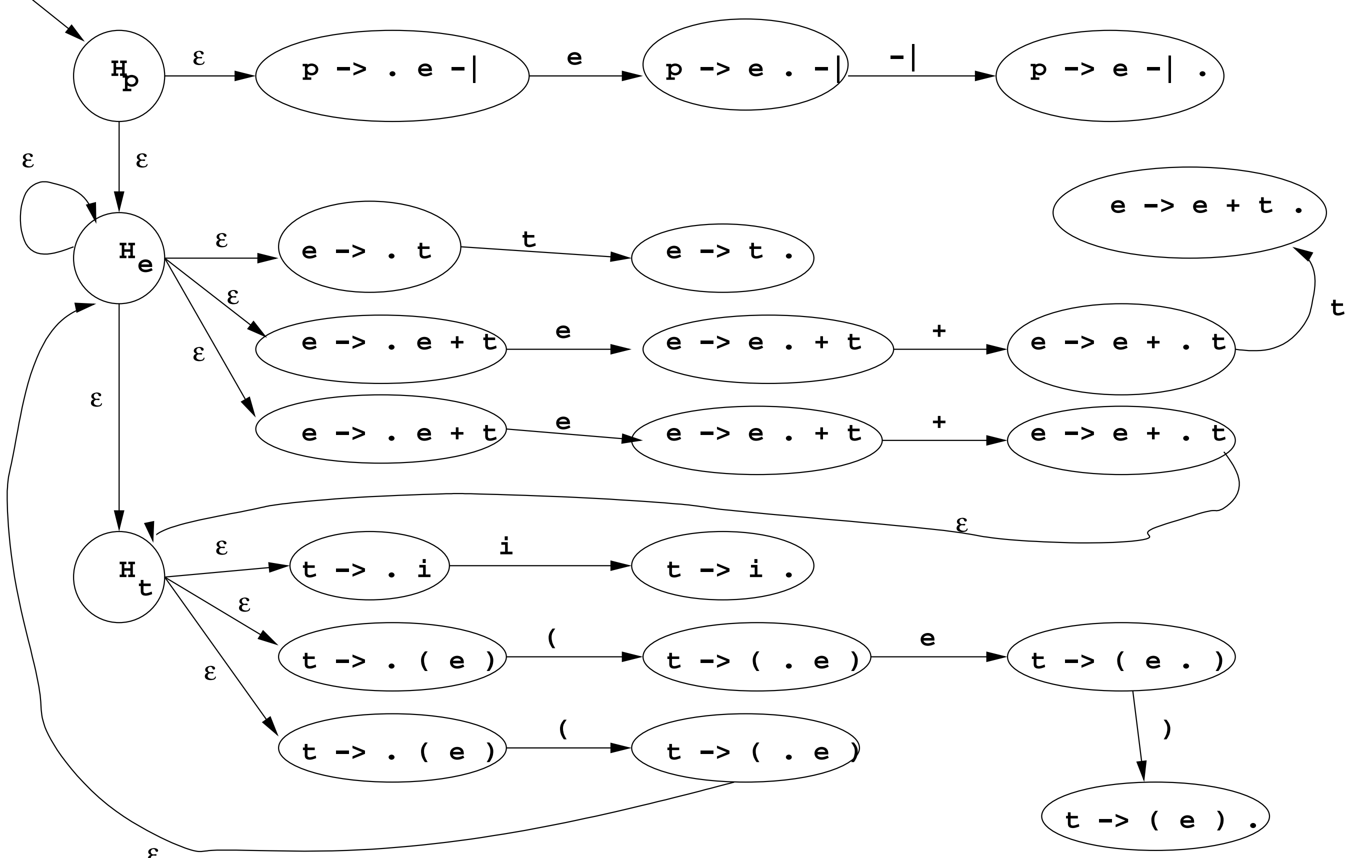


Figure 3.8: NFA from the handle grammar.

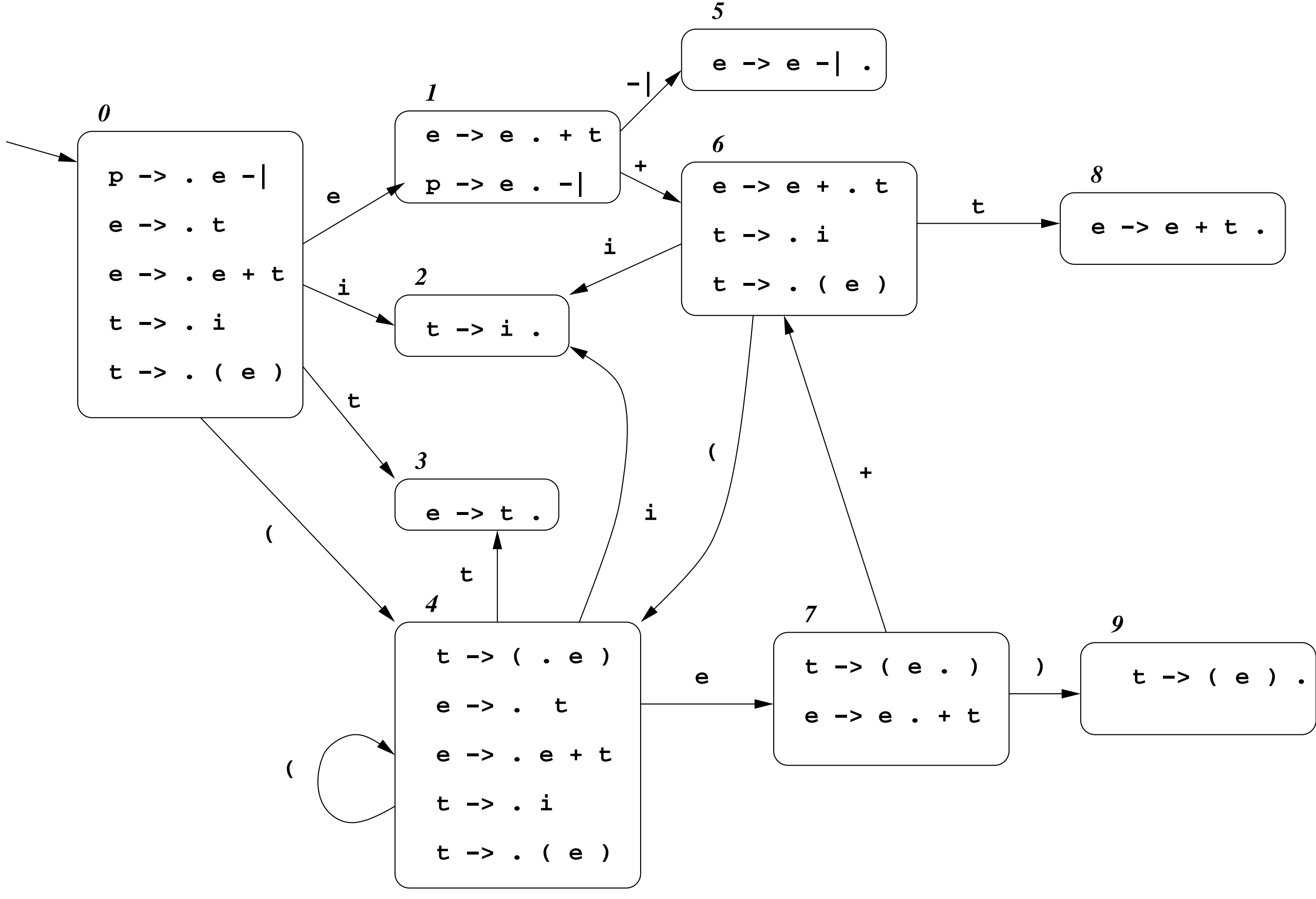


Figure 3.9: DFA constructed from Figure 3.8: The canonical LR(0) machine.

get the machine shown in Figure 3.9. It is no accident that the labels in the nodes of this DFA look like item sets from Earley's algorithm (minus the trailing input positions). They record the same information, but the trailing input positions are unnecessary for deterministic grammars.

3.10.3 Using the machine

We may represent the LR(0) machine from Figure 3.9 as a state-transition table:

	Action					*Goto*	
State	`i`	`+`	`(`	`)`	⊣	`e`	`t`
0.	s2		s4			1	3
1.		s6			s5		
2.	rE	rE	rE	rE	rE		
3.	rC	rC	rC	rC	rC		
4.	s2		s4			7	3
5.			*ACCEPT*				
6.	s2		s4				8
7.		s6		s9			
8.	rB	rB	rB	rB	rB		
9.	rD	rD	rD	rD	rD		

The numeric entries in this table (preceded by 's' (shift) in the action table and appearing plain in the goto table) come from the state transitions (arcs) in Figure 3.9. The 'r' (reduce) entries come from LR(0) items with a dot at the right (indicating a state of "being to the right of a potential handle.") The letters after 'r' refer to productions in Grammar 2.

To see how to use this table, consider again the string 'i+(i+i)⊣'. Initially, we have the situation

```
1.  0            | i + ( i + i ) ⊣
```

Here, the '`|`' separates the stack on the left from the unprocessed input on the right; the lookahead symbol is right after '`|`'. The subscript '0' indicates that the DFA state that corresponds to the left end of the stack is state 0. We use the table, starting in state 0. There is nothing on the stack, and row 0 of the table tells us that there is no reduction possible, but that we could process an 'i' token if it were on the stack. Therefore, we shift the 'i' token from the input, giving

2. $_0\mathtt{i}_2$ `| + ( i + i ) ⊣`

(The subscript 2 shows the DFA's state after scanning the 'i' on the stack). Again, we start in state 0 and scan the stack, using the transitions in the table. This leaves us in state 2. Row 2 in the table tells us that no shifts are possible, but we may reduce ('r') using production E (`t:  i`). We therefore pop the 'i' off the stack, and push a 't' back on, giving

2. $_0\mathtt{t}_3$ `| + ( i + i ) ⊣`

Running the machine over this new stack lands us in state 3, which says that no shifts are possible, but we can use reduction C (`e` → `t`), which gives

3. $_0\mathtt{e}_1$ `| + ( i + i ) ⊣`

Now the machine ends up in state 1, whose row tells us that either a '+' or an '⊣' could be next on the stack, so that we can shift either of these, leading to

3a. ${}_0\mathtt{e}_1\ \mathtt{+}_6$ | (i + i) ⊣

The state 6 entry tells us that we can shift '(', and then the state 4 entry tells us we can shift 'i', giving

3b. ${}_0\mathtt{e}_1\ \mathtt{+}_6\mathtt{(}_4\ \mathtt{i}_2$ | + i) ⊣

whereupon we see, again from the state 2 entry, that we should reduce using production E:

4. ${}_0\mathtt{e}_1\ \mathtt{+}_6\ \mathtt{(}_4\ \mathtt{t}_3$ | + i) ⊣

and the state 3 entry tells us to reduce using production C:

5. ${}_0\mathtt{e}_1\ \mathtt{+}_6\ \mathtt{(}_4\ \mathtt{e}_7$ | + i) ⊣

and so forth.

In general, then, we repeatedly perform the following steps for each shift and reduction the parser takes:

```
FINDSTATE:
  state = 0;
  for each symbol, s, on the stack,
      state = table[state][s];



FINDACTION:
  if table[state][lookahead] is sn
      push the lookahead symbol on the stack;
      advance the input;
  else if table[state][lookahead] is rk
      Let A : x1 ··· xm be production k;
      pop m symbols from the stack;
      push symbol A on the stack;
  else if table[state][lookahead] is ACCEPT
      end the parse;
```

The FINDACTION part of this fragment takes a constant amount of time for each action. However, the time required for FINDSTATE increases with the size of the stack. We can speed up the parsing process with a bit of "memoization". Rather than save the stack symbols, we instead save the *states* that scanning those symbols

results in (the subscripts in my examples above). Each parsing step then looks like this:

```
FINDACTION:
  if table[top(stack)][lookahead] is sn
      push n on the stack;
      advance the input;
  else if table[top(stack)][lookahead] is rk
      Let A : x1 ··· xm be production k;
      pop m states from the stack;
      // Reminder:  top(stack) is now changed!
      push table[top(stack)][A] on the stack;
  else if table[state][lookahead] is ACCEPT
      end the parse;
```

and our sample parse looks like this:

```
 0.  0      |   i + ( i + i ) ⊣
 1.  0 2    | + ( i + i ) ⊣
 2.  0 3    | + ( i + i ) ⊣
 3.  0 1    | + ( i + i ) ⊣
3a.  0 1 6    | ( i + i ) ⊣
3b.  0 1 6 4    | i + i ) ⊣
3c.  0 1 6 4 2    | + i ) ⊣
 4.  0 1 6 4 3    | + i ) ⊣
 5.  0 1 6 4 7    | + i ) ⊣
5a.  0 1 6 4 7 6    | i ) ⊣
5b.  0 1 6 4 7 6 2    | ) ⊣
 6.  0 1 6 4 7 6 8    | ) ⊣
 7.  0 1 6 4 7        | ) ⊣
7a.  0 1 6 4 7 9        | ⊣
 8.  0 1 6 8            | ⊣
 9.  0 1                | ⊣
9a.  0 1 5                  |
10.  ACCEPT
```

It's important to see that all we have done with this change is to speed up the parse.

3.10.4 Resolving conflicts

Grammar 2 is called an *LR(0) grammar*, meaning that its LR(0) machine has the property that each state contains either no reduction items (items with a dot at the far right) or exactly one reduction item and nothing else. In other words, an LR(0) grammar is one that can be parsed from *L*eft to right to produce a *R*ightmost derivation using a shift-reduce parser that does not consult the lookahead character (uses *0* symbols of lookahead). Few grammars are so simple. Consider, for example,

Grammar 3.

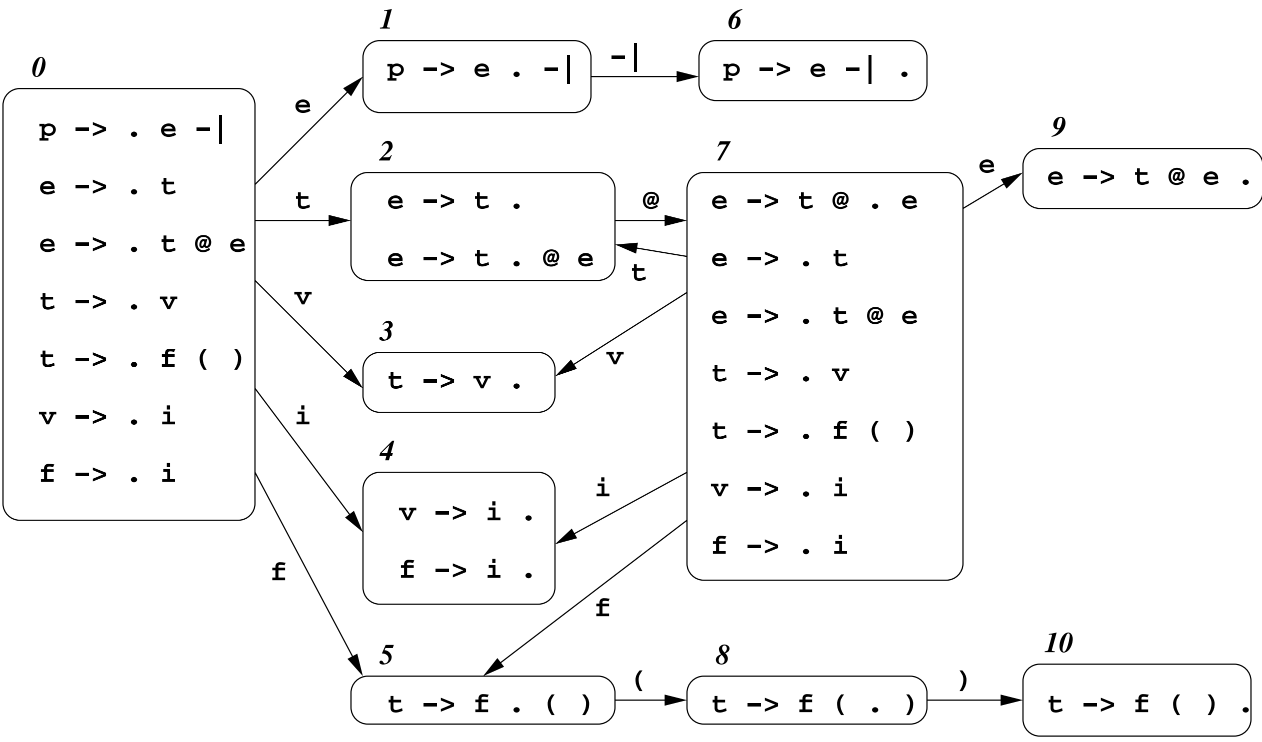


Figure 3.10: LR(0) machine for Grammar 3.

```
A. p:  e '⊣'
B. e:  t '@' e
C. e:  t
D. t:  f '(' ')'
E. t:  v
F. f:  i
G. v:  i
```

which gives us the DFA in Figure 3.10.

As you can see from the figure, there are problems in states #2 and #4. State #2 has an *LR(0) shift/reduce conflict:* it is possible *both* to reduce by reduction C *or* to shift the symbol '@'. In this particular case, it turns out that the correct thing to do is to shift when the lookahead symbol is '@' and to reduce otherwise; that is, reducing on '@' will always cause the parse to fail later on. State #4 has an *LR(0) reduce/reduce conflict:* it is possible to reduce either by reduction F or G. In this case, the correct thing to do is to reduce using F if the next input symbol is '(' and by G otherwise. We end up with the following parsing table:

	Action					*Goto*			
State	`i`	`@`	`(`	`)`	⊣	`e`	`t`	`f`	`v`
0.	s4					1	2	5	3
1.					s6				
2.	rC	s7	rC	rC	rC				
3.	rE	rE	rE	rE	rE				
4.	rG	rG	rF	rG	rG				
5.			s8						
6.			ACCEPT						
7.	s4					9	2	5	3
8.				s10					
9.	rB	rB	rB	rB	rB				
10.	rD	rD	rD	rD	rD				

Because the choice between reduction and shift, or between two reductions, depends on the lookahead symbol (in contrast to Grammar 2), we say Grammar 3 is not LR(0). However, since one symbol of lookahead suffices, we say that it is *LR(1)*—parseable from *L*eft to right producing a *R*ightmost derivation using a shift-reduce parser with *1* symbol of lookahead. In fact, Grammar 3 is what we call *LALR(1)*, the subclass of *LR(1)* for which the parsing table has the same states and columns as for the LR(0) machine, and we merely have to choose the entries properly to get the desired result. (LALR means "Lookahead LR." Since LR parsers *do* look ahead anyway, it's a terrible name, but we're stuck with it.) Yacc and Bison produce LALR(1) parsers. The class LR(1) is bigger, but few practical grammars are LR(1) without being LALR(1), and LALR(1) parsing tables are considerably smaller.

Unfortunately, it is not clear from just looking at the machine that we have filled in the problematic entries correctly. In particular, while the choice between reductions F and G in state #4 is clear in this case, the general rule is not at all obvious. As for the LR(0) shift-reduce conflict in state #2, it is obvious that if '@' is the lookahead symbol, then shifting *has* to be acceptable, but perhaps this is because the grammar is ambiguous and *either* the shift or the reduction could work, or perhaps if we looked two symbols ahead instead of just one, we would sometimes choose the reduction rather than the shift.

One systematic approach is to use the FOLLOW sets that we used in LL(1) parsing. Faced with an LR(0) reduce/reduce conflict such as '`f`→ `i`' vs. '`v`→ `i`' in state #4, we choose to reduce to `f` if the lookahead symbol is in FOLLOW(`f`), choose to reduce `v` if the lookahead symbol is in FOLLOW(`v`), and choose either one otherwise (or leave the entry blank). Likewise, we can assure that the LR(0) shift-reduce conflict in state #2 is properly resolved in favor of shifting '@' as long as '@' does not appear in FOLLOW(`e`), as in fact it doesn't. When this simple method resolves all conflicts and tells us how to fill in the LR(0) conflicts in the table, we say that the grammar is *SLR(1)* (the 'S' is for "Simple"). Grammar 3 happens to be SLR(1).

However, there are cases where the FOLLOW sets fail to resolve the conflict because they are not sensitive to the context in which the reduction takes place. Therefore, typical shift-reduce parser generators go a step further and use the full LALR(1) lookahead computation. We attach a set of *lookahead symbols* to the end of each LR(0) item, giving what is called an LR(1) item. Think of an LR(1) item

such as

```
t:  .v,   ⊣, @
```

as meaning "we could be at the left end of a handle for `t`, and after that handle, we expect to see either an '⊣' or a '@'." We add these lookaheads to the LR(0) machine in Figure 3.10 by applying the following two operations repeatedly until nothing changes, starting with empty lookahead sets:

- If we see an item of the form '$A \to \alpha.B\beta, L_1$' in a state (where L is a set of lookaheads, B is a nonterminal, and α and β are sequences of 0 or more terminal and nonterminal symbols), then for every other item in that same state of the form '$B : .\gamma, L_2$,' add the set of terminal symbols FIRST(βL_1) to the set L_2. (Here, we define FIRST(L_1) to be simply L_1. Therefore, FIRST(βL_1) is simply FIRST(β) if β does not produce the empty string, and otherwise it is FIRST(β) $\cup L_1 - \epsilon$).

- If we see an item of the form '$A \to \alpha.X\beta, L_1$' in a state, then find the transition from that state on symbol X and find item '$A : \alpha X.\beta, L_2$' in the target of that transition. Add the symbols in L_1 to L_2.

Applying these operations to the machine in Figure 3.10 gives the LALR(1) machine in Figure 3.11.

3.10.5 Using Ambiguous Grammars

The traditional method of specifying the precedence and association of operators—illustrated in Grammar 3.5, for example—involves introducing numerous nonterminals, one for each level of precedence. It would be much nicer to be able to write something like

Grammar 3.12. Ambiguous expression grammar

```
expr : IDENTIFIER
     | NUMERAL
     | '(' expr ')'
     | expr '-' expr
     | expr '/' expr
     | expr "**" expr
```

and somehow specify separately that '`**`' has highest precedence and associates to the right, '`/`' has next highest and associates left, and '`-`' has lowest and associates left. We can get exactly this effect by attempting to build the LALR(1) machine for this grammar, and then resolving the conflicts that result "by hand." The conflicts that arise show up as unresolved shift-reduce conflicts. We get states in the machine labeled with item sets containing such things as

```
expr:  expr '-' expr{•}, Lookaheads:  '-', '{⊣}', '/', '**' ')'
expr:  expr {•}'-' expr,
```

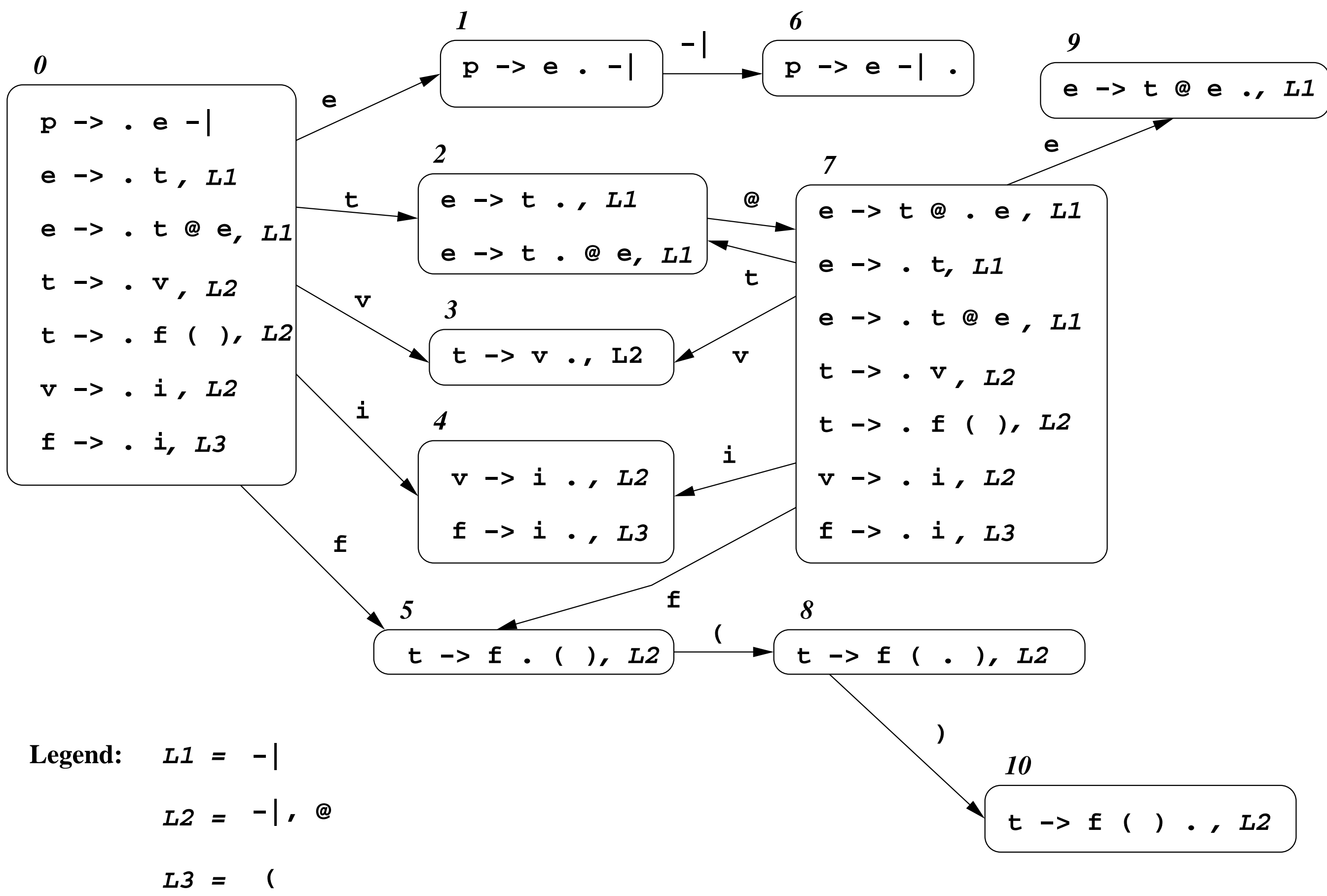


Figure 3.11: LALR(1) machine for Grammar 3.

```
expr:  expr {•}'/' expr,
...
```

If the next input symbol is '`-`', do we reduce by the first item, or do we shift by the second? The lookahead set fails to resolve this conflict. If we decide to take the reduction, the effect is to associate '-' to the left (the first of two '-' operators binds more tightly). If we shift the '`-`', the effect is to associate to the right. Likewise, by choosing to shift on seeing a '`/`', we effectively cause '`/`' to bind more tightly (have higher precedence) than '`-`'.

The Bison parser generator exploits this behavior. You can augment the ambiguous grammar with *precedence declarations:*

```
%left '-'
%left '/'
%right "**"
```

Each line represents a set of operators of increasing precedence, and the "`%left`" or "`%right`" declarations indicate how operators on the same line group. The implementation is quite simple. Whenever Bison sees a rule containing one of these terminal symbols, such as

```
expr : expr '-' expr
```

it assigns to that rule the declared precedence of the operator that appears in its right-hand side. Whenever there is a conflict between a reduction by this rule and a shift of a terminal symbol that is not resolved by the LALR(1) method, it resolves the conflict in favor of whichever has higher precedence: the rule (reduce) or the terminal symbol (shift). When considering rules and terminal symbols of the same precedence, it chooses the reduction if the symbol's precedence was declared with `%left`, and otherwise chooses the shift. The result is a pretty natural grammar specification.

The rules are general enough that the same disambiguation can be applied to things other than expressions. Normally, though, I'd avoid doing so and stick to simple uses such as the example above. In particular, it's a very bad idea to use precedence declarations to resolve arbitrary shift-reduce conflicts in your grammar. Outside of the stylized conflicts that occur in expressions, grammar conflicts usually indicate errors in your grammar that you should fix rather than try to suppress.

Chapter 4

Static Analysis: Scope and Types

4.1 Terminology

Programs, in general, are simply collections of definitions of terms, which we often call *declarations*[1]. Each declaration may *use* other declarations, referring to them by the names they *introduce.* In programming languages, the *scope of a declaration* that introduces some name is the portion of the program in which the meaning (or a possible meaning) of that name is the one given by the declaration. Many authors (from less distinguished institutions) refer loosely to the scope of a *name* as opposed to the scope of a *declaration.* The term "scope of a name," however, is clearly an inadequate notion, since the same name may be used in multiple declarations.

4.2 Environments and Static Scoping

In CS61A, you saw an abstract model of both scope rules and rules about the *extent* or *lifetime* of variables. In this model, there is, at any given time, an *environment* consisting of a linked sequence of *frames*. Each frame contains *bindings* of names to slots that can contain values. The value contained in a slot may be a function; such a value consists of a pair of values: the *body* (or code) of the function and the *environment* that gives the meaning of names used by the function when it executes.

Figure 4.1 illustrates the scope of declarations in C (or Java or C++). The sections of text controlled by the various declarations of variables, parameters, and functions are indicated by the brackets on the right. Brackets on the left indicate *declarative regions*—portions of the text of a program that bound the scopes of the declarations within. Declarations in C obey the rule that their scope runs from the declaration to the end of the innermost declarative region that contains them. The declarative regions in C are the boundaries of the source file itself, the boundaries of

[1] C and C++ distinguish *declarations,* which introduce (or re-introduce) names and some information about them from *definitions,* which provide complete information about names. A declaration of a function tells us its name, parameter types, and return type. A definition of a function tells us all this and also gives its body. In these notes, I will use the term *declaration* to refer to both of these functions.

each block ({...}), the parameter list and body of each function, and a few others.

The environment diagram in Figure 4.1 shows a snapshot of the program during its execution. To find the current meaning (binding) of any identifier, one traverses the environment structure from the current environment, following the pointers (*links*) to enclosing environments, until one finds the desired identifier. Each frame corresponds to an instance of some declarative region in the program. When a function is called, a frame is created for that call that contains the variables declared in that function with a static link that is set from the environment part of the function (the leftmost "bubbles" in the figure). Inner blocks are treated like inner functions, and have static links that point to instances of their enclosing blocks[2]. Since the language in the example is C, all named functions' environments are the *global environment,* encompassing all declarations in a given source file.

As it was presented in CS61A, these environments are dynamic entities, constructed during execution time. However, it is a property of most languages—including C, C++, Java, Pascal, Ada, Scheme, the Algol family, COBOL, PL/1, Fortran, and many others—that at any given point in a program, the chain of environment frames starting at the current environment is always the same at that point in the program, except for the actual values in the slots of environment. That is, the number of frames in the chain and the names in each of these frames is always the same, and depends only on where in the program we are. We say that these languages use use *static scoping* (also known as *lexical scoping*), meaning that their scope rules determine static regions of text that are independent of any particular execution of the program. The links between frames in this case are called *static links.*

To see why this property holds (the constancy or staticness of the environment chain), consider how the links get set in the first place. When a function value is created in a statically scoped language (i.e., as opposed to being called), its value is constructed from a *code pointer* and an *environment pointer*. The environment pointer, furthermore, is always the current frame, which is a frame containing declarations from the function whose text contains the definition of the function. The environment pointer, in other words, depends only on where in the text of the program the function is defined, and points to the same kind of frame (same names) each time. When a function is called, a new current frame is created to contain declarations of the function's parameters and local variables, and its link (in these languages) is copied from the environment pointer of the function being called. Thus, every time a frame for a given function is created, its link always points to a frame with the same structure.

4.3 Dynamic Scoping

An alternative rule (found in Logo, APL, SNOBOL, and early versions of Lisp, among other languages) defines the scope of a declaration as proceeding from the *time* at which the declaration is "executed" (or *elaborated,* to use Algol 68 terminology) to the time it terminates. Under dynamic scoping, the link of an environment

[2] WARNING: This is a *conceptual* description. The actual execution of a program involves different data structures, as we will see in later lectures

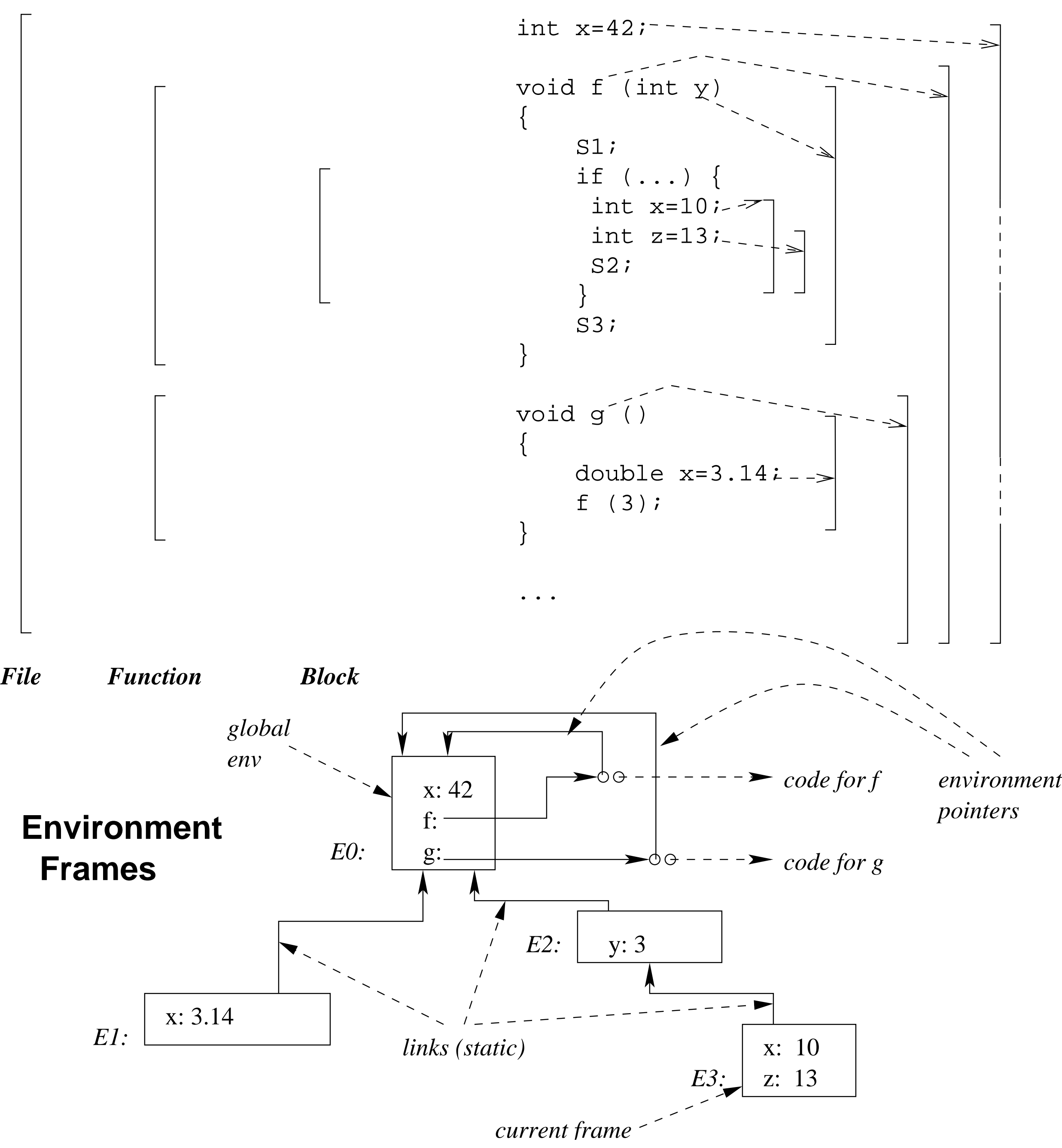


Figure 4.1: Scope of declarations in C. The brackets on the right indicate the scope of the declarations in the program. The dashed portions of the rightmost bracket (for the first declaration of `x`) indicate the portion of text in which its declaration is *hidden* by another declaration. The brackets on the left are declarative regions, which bound the scopes of items defined inside them. The environment diagram below shows the situation during execution of the inner block of `f`, when it has been called from `g`.

frame for a function is equal to the current frame at the time of the *call.* To see the distinction, consider the following program (in a C-like language):

```
int x = 3;          /* (1) */

void f(int x)       /* (2) */
{
   g ();
}

void g ()
{
   print (x);
}

void doit ()
{
   int x = 12;
   f(42);
   g();
}
```

In normal C (or C++ or Java), this program would print '3' twice. Were these languages to use dynamic scoping instead, it would print '42', and then '12'. Figure 4.2 shows the environment structure during the two calls to g under static scoping and under dynamic scoping. There isn't one declaration of `x` in the body of `g` (hence the term "dynamic").

4.4 Compiler Symbol Tables

For languages with lexical scoping, the environment model suggests properties of a data structure (or *symbol table,* as it is generally known) whereby a compiler can keep track of definitions in the program it is processing. It can use the environment structure, but rather than store *values* in the slots of the frames, it can store *declarations* (well, actually, pointers to things that represent these declarations). This data structure allows the compiler to map any identifier in the program it is processing to the declaration for that identifier (and thus to any information contained in that declaration). Figure 4.3 shows how this would work for the example in Figure 4.1.

We can make minor changes to the environment model to accommodate a range of language features:

- Java and C++ both allow overloading of functions, based on argument types. We model this by having the names include *argument signatures.* For example, the function `f` in the figures could appear in its environment frame as

  ```
  void f(int):
  ```

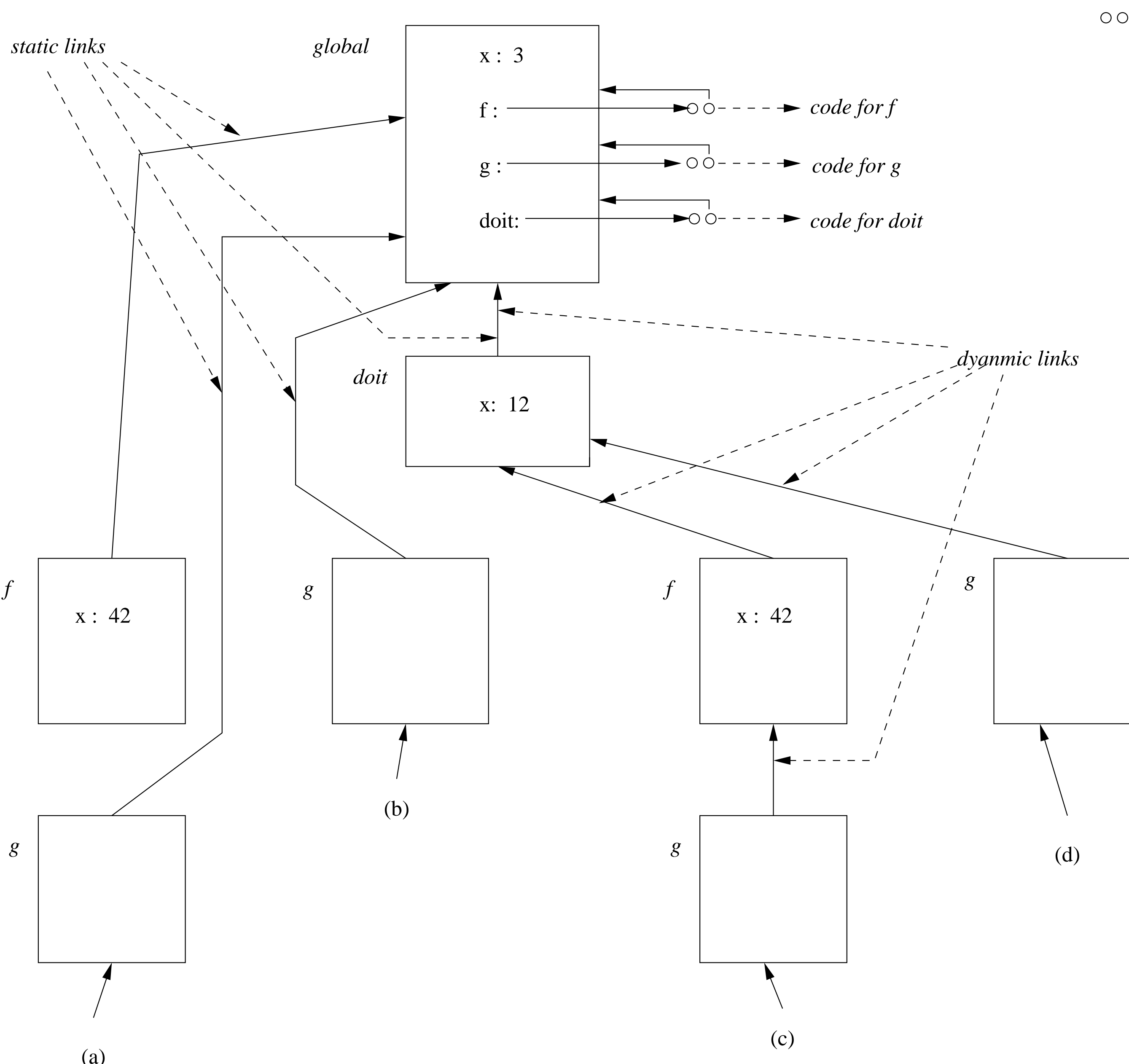


Figure 4.2: Situation during call to `g` in four situations: (a) called from `f` using static scoping, (b) called from `doit` using static scoping, (c) called from `f` using dynamic scoping, and (d) called from `doit` using dynamic scoping. Situations (a) and (b) show what happens in languages like C, C++, Scheme, and Java (both print 3). Situations (c) and (d) apply to older Lisps, APL, SNOBOL, and others (case (c) prints 42 and (d) prints 12). The links between environments are static for (a) and (b) and dynamic for (c) and (d). The static and dynamic links from `doit` happen to be identical in this case.

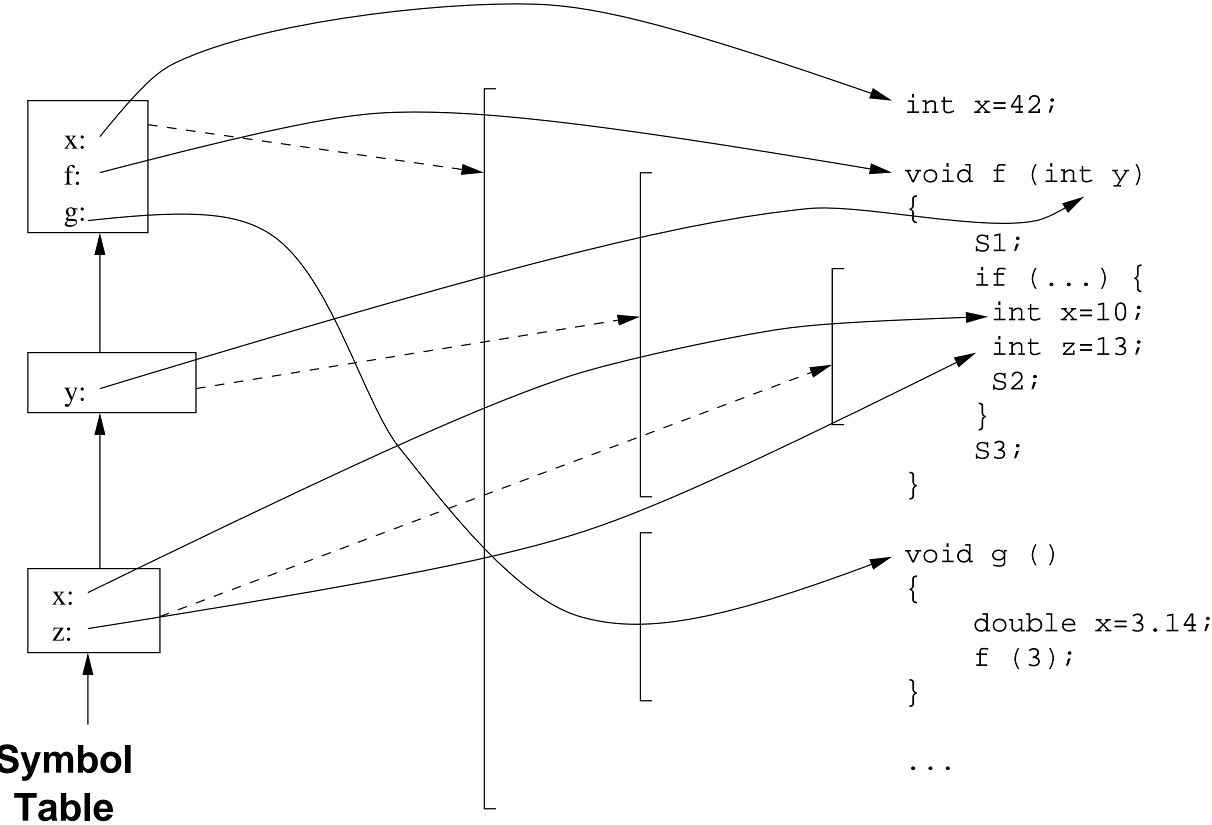


Figure 4.3: Adapting environment diagrams as symbol tables. This shows the compiler's symbol table (in the abstract) when processing the inner block of `f`. Compare this with Figure 4.1. The values in the previous diagram become declarations in the symbol table. Each frame corresponds to a declarative region.

- Many languages have structure types whose instances have *fields* (or *members*). We can represent these types (or their instances) as environment frames. In a context such as `x.y`, we look up `y` in starting from the environment that represents `x`'s type.

- When languages have structure types with inheritance, as do C++ and Java, it can be represented by having the static link of a frame representing one type point to the parent or base type of that type. Where multiple inheritance is legal (again as in C++ or Java), we can generalize the static link to be a set of pointers rather than a single one.

4.5 Lifetime

Finally, we leave with one important point. The scope of a declaration refers only to those sections of the program text where that declaration determines the meaning of its name. This is a distinct concept from that of how long the named entity actually exists. The latter is what we'll call the *lifetime* of the entity. For example, going back to Figure 4.1, the declaration of `x` inside the text of `g` is out of scope (invisible) during execution of `f`, but the variable (slot) created by that declaration does not go away. Nor does variable created by the first (global) declaration of `x` go away in the "scope holes" where it is hidden. The situation in the C++ declaration

```
void f ()
{
   Foo* x = new Foo;
   g ();
   other stuff involving x;
}
```

is even more complicated. The declaration of `x` introduces a variable called `x` and then stores into it a pointer to an *anonymous* object (created by **new**) that is not named by any declaration. The lifetime of a variable `x` ends upon return from the call to `f` that creates it. The lifetime of the anonymous thing it points to, however, continues until it is explicitly deleted. Both variable `x` and the object it points to, of course, continue to exist during the call to `g`, even though the declaration of `x` is not visible in `g`.

4.6 Static and Dynamic Typing

In programming languages, a *type* is a characteristic of an expression, parameter, variable (or other entity that is, denotes, or holds a value) that characterizes what values the entity may have (or denote or ...) and what operations may be applied to it. When, as in Scheme, the type of a quantity is not determined, in general, until a program is executed, we say the language is *dynamically typed*. When—as in C, C++, FORTRAN, COBOL, Algol, Pascal, PL/1, or Java—the type of an entity in a program is determinable solely from the text of the program, we say that the language is *statically typed*.

4.6.1 Type Equivalence

To define the typing rules of a language, one usually has to define what it means for the types of two entities to match. For example, consider the following code in C or C++:

```
struct { int x, y; } A, B;
struct { int x, y; } C;
struct T { int x, y; } D;
struct T E;
T* F;
T* G;

main()
{
    A = B;  /* OK */
    A = C;  /* Error: type mismatch */
    A = D;  /* Error: type mismatch */
    D = E;  /* OK */
    F = G;  /* OK */
}
```

The constructs '`struct {...}`' and '`...*`' are *type constructors:* given a set of *type parameters,* represented here by '`...`', they construct a new type.

As the comments show, the rule in C and C++ (also in Pascal, Ada, and many other languages) is that each distinct occurrence of `struct` creates a brand new type, differing from (not equivalent too) all other types, even those with identical declarations. `A` and `B` have the same type, since it is "generated" by the same instance of `struct`. `D` and `E` have the same type, since the definition of `D` introduces the name `T` to stand for the newly-constructed type, and the declaration of `E` then refers to that type by name. We call this kind of rule a *name equivalence* rule.

On the other hand, the types of `F` and `G` are identical, despite the fact that their types come from two distinct instances of a generator. The two instances of `T*` define types with identical *structures:* identical arguments to the type constructor. The rule in C and C++ is that structurally identical pointer types are all the same. We call this kind of rule a *structural equivalence* rule.

As you can see, Java, C, and C++ mix the two types of rule freely: array types, pointer types, reference types (C++), and function types obey structural equivalence rules, while class (struct) and union types obey name equivalence rules. The languages Pascal and Ada, on the other hand, adhere more consistently to the name equivalence model (array types, for example, are different if produced from two separate constructors). The language Algol 68 adheres consistently to structural equivalence (all the struct types in the example above are identical in Algol 68).

4.6.2 Coercion

Strict rules such as "both sides of an assignment statement must have exactly the same type" usually prove to be unduly burdensome to programmers. Most languages therefore provide some set of implicit *coercions* that automatically convert

one type of value to another. In Java, C, and C++, for example, we have the standard numeric promotions, which convert, e.g., `short` and `char` values to `int` or `long`. We also have the standard pointer conversions, which translate any pointer to an object to a `void*`, and convert `B*` to `A*` if `A` is a base class of `B`.

4.6.3 Overloading

The purpose of name resolution (scope rule checking), is to associate declarations with instances of names. Java and C++ introduces a new wrinkle—the possibility of having several declarations referred to by the same name. For example,

```
int f(int x) { ... }  /* (1) */
int f(A y) { ... }    /* (2) */
```

In the presence of these declarations, scope rule checking will tell us that `f` in

```
f(3)
```

can mean *either* declaration (1) or (2). Until we analyze the type of the expression, however, we can't tell which of the two it is.

C++ requires that the decision between declarations (1) and (2) must depend entirely on the types of the arguments to the function call. The following is illegal:

```
int f2(int x) { ... } /* (3) */
A   f2(int x) { ... } /* (4) */
int x;
...
x = f2(3)
```

On the other hand, the language Ada allows these declarations (well, in its own syntax), and is able to determine that since `f2` must return an `int` for the assignment to be legal, one declaration (3) can apply. That is, Ada uses the result types of the overloaded declarations as well as their argument types.

Both C++ and Ada provide for *default parameters:*

```
int g(int x, int y = 0) { ... } /* (5) */
```

This doesn't really introduce anything new, however; we can treat it as

```
int g(int x, int y) { ... } /* (5a) */
int g(int x) { return g(x, 0); } /* (5b) */
```

C++ does not resolve function calls based on return types, but it does allow user-defined conversions that may be chosen according to the required type of an expression. For example,

```
class A {
public:
    ...
    operator int() const { ... }
    ...
}
```

defines an operator (which is, after all, just a function with an attitude) that will convert an object of type `A` into an `int`, if one is needed in a given context. That makes it legal to write, e.g.,

```
int h(int x) { ... }
A z;
...
h(z);
```

Any system of coercions as complex as that of C++ tends to give rise to unwanted ambiguities (several different ways of coercing the arguments of to a function that match several different possible overloadings of the function). To counter this, C++ has a complex set of rules that place a preference order on implicit coercions. For example, given the definitions of `f` and `A` above, it is acceptable (unambiguous) to write

```
A z;
... f(z)...
```

despite the fact that declaration (2) matches the call and declaration (1) matches it after an application of the conversion defined from `A` to `int`. This is because the rules indicate a preference for calls that do not need user-defined conversion.

Of all the features in this section, Java uses only overloading on the argument type.

4.7 Unification (aka Pattern-Matching)

In C++, one can define *template functions,* such as

```
/* sort A[0..n-1] so that le(A[i], A[i+1]) for 0 <= i < n-1 */
template <class T>
void sort(T* A, int n, bool le(const T&, const T&))
{
    ...
}
```

which will work for any type `T`:

```
bool double_le(const double& x, const double& y)
{
    return x <= y;
}

double V[100];

... sort(V, 100, double_le) ...
```

For each distinct `T` that is used in a call to `sort`, the compiler creates a new overloading of `sort`, by filling in the '`T`' slot of the template with the appropriate type.

This feature of C++ therefore raises the questions of how one matches a call against a rule like this and how one determines what type to plug into `T`.

Given a call, such as `sort(V, 100, double_le)`, we can determine the types of all the arguments, and represent them as a list of trees, such as the following (to use Lisp notation).

```
(list
 (ArrayType (DoubleType))
 (IntType)
 (FunctionType
   ((RefToConstant (DoubleType))
    (RefToConstant (DoubleType)))
   (BoolType)))
```

Likewise, we can do the same for the formal parameter list of `sort`:

```
(list
 (ArrayType T)
 (IntType)
 (FunctionType
   ((RefToConstant T)
    (RefToConstant T))
   (BoolType)))
```

Each S-expression $(AY_1 \cdots Y_n)$ denotes a tree node with operator (label) A and n children Y_i. I've put in a dummy operator, `list`, in order to make the algorithm below easier to write.

The task of seeing whether these two types match is one of pattern matching, or to use the fancy phrase, *unification:* that is, finding a substitution for all the type variables (here, the single template parameter `T`) that makes the list of argument types and the formal parameter list match.

The algorithm finds out whether a pattern matches an argument list, and finds a *binding* of each type variable to a type expression. Initially, each type variable is unbound. As the algorithm progresses, some type variables get bound, sometimes to other type variables. As a result, there may be chains of type variables (never circular), and it's convenient to define `binding(`T`)` to mean the result of following such a chain to its end:

$$\text{binding}(T) = \begin{cases} T, & \text{if } T \text{ is unbound.} \\ binding(T'), & \text{if } T \text{ is bound to } T'. \end{cases}$$

By taking all nodes other than type variables as being unbound, we can treat `binding` as being defined on all nodes. The construct `oper(`t`)` gives the operator of tree node t, and `children(`t`)` gives its children. The operator of a node in this case is either a type constructor or a constant type (such as `int`). We start with a simple version of unification:

```
/* Return true iff the pattern P matches the pattern A, and binds */
/* type variables in A and P so as to make the two match. */
match(P, A) {
```

```
        P = binding(P);
        A = binding(A);     /* X */
        if (P == A)
            /* == means the same object pointer''.  Test is true if A
             * and P are the same type variable. */
            return true;
        if (P is an unbound type variable not found in A) {
            bind P to A;
            return true;
        }
        if (A is an unbound type variable not found in P) {  /* X */
            bind A to P;
            return true;
        }
        if (oper(P) != oper(A)
            || length(children(P)) != length(children(A)))
            return false;
        for each child cp in children(P)
                  and corresponding child ca in children(A) {
            if (! match (cp, ca))
                return false;
        }
        return true;
    }
```

For use with C++, `A` never contains type variables, so we can eliminate the two statements marked `/* X */`.

To be honest, I've idealized the process in the case of C++. It actually allows types with differing operators to match when there is a coercion between them. For example, a `char` operand can match an `int` formal, and a `char*` operand can match a `const char*` formal. Indeed, C++ doesn't really benefit all that much from this particular formalization of type matching, but it is rather more useful with some interesting (and better-designed) languages, such as those in the ML family.

4.7.1 Type inference

Languages such as ML make interesting use of unification to get static typing without having to mention the types of things explicitly. For example, consider the following function definition (the syntax is fictional):

```
sum(L) = if null(L) then 0 else head(L) + sum(tail(L)) fi;
```

Here, `if...then...else...fi` is a conditional expression, as for '`if`' in Scheme or '`?`' and '`:`' in C. The rules for this fictional language say that

- `null` is a family of functions (what is called a *polymorphic function*) whose argument types are `list of` T for all types T, and whose return type is `bool`.
- `head` is a family of functions whose argument types are `list of` T and whose return type is T for all types T.

- `tail` is a family of functions whose argument and return types are `list of` T for all types T.
- '+' takes two integers and returns an integer.
- The 'then' and 'else' clauses of an 'if' have to have the same type. That is also the type of the 'if' as a whole.
- A (one-argument) function has a type $T_0 \rightarrow T_1$ for some types T_0 and T_1. Its arguments must have type T_0 and its return value is type T_1.

To begin with, we don't know the types of `L` or `sum`. So, we initially say that their types are the type variables T_0 and T_1, respectively. Now we apply the rules using the matching procedure described above.

- `sum` is a function, so its type (T_1) must be $T_2 \rightarrow T_3$ for some types T_2 and T_3; that is `match(`T_1`, `$T_2 \rightarrow T_3$`)` must be true.
- `L` is an argument to `sum` so `match(`T_0`, `T_2`)`.
- The argument type of `null` is `list of` T_4 for some type T_4, so `match(`T_0`, list of `T_4`)`.
- If the argument type to `head` is `list of` T_4, the the return type is T_4.
- Since the operand types of '+' must be `int`, `match(`T_4`,int)`.
- The return type of `sum` is the type returned by the 'if'. That, in turn, must be the same as the type of `0` (its 'then' clause). So, `match(`T_3`,int)`.

Work this all through (executing all the '`match`' operations) and you will see that we end up with the type of `x` (T_0) being `list of int` and that of `sum` being `list of int` $\rightarrow$ `int`.

4.7.2 Recursive patterns

Suppose that we introduce a new, two-argument type constructor called `pair`, so that something of type **pair**(T_1, T_2) denotes a something that is either `null` or has a `head` of type T_1 and a tail of type T_2. In a dynamically typed language such as Scheme, such a type allows one to construct list types, which are pairs in which the tail is itself a list. This suggests a defining a list type variable, T as having the property

$$T = \texttt{pair}(T_1, T).$$

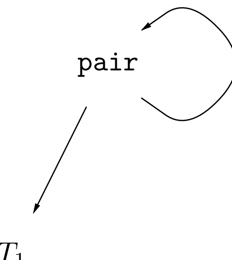
However, the algorithm given in §4.7 put a limitation on legal bindings (sometimes known as the *occurs check*):

```
if (P is an unbound type variable not found in A) { ...
```

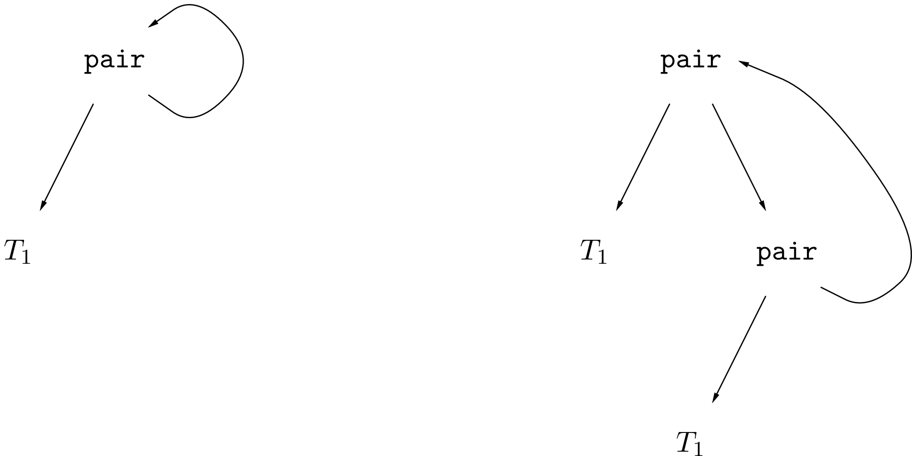
Given the algorithm, such a restriction is hardly surprising. Consider the system of equations

$$L_1 = \texttt{pair}(T_1, L_1), \quad L_2 = \texttt{pair}(T_1, L_2), \quad L_1 = L_2.$$

If you perform the indicated matches in sequence, you'll find that the algorithm goes into an infinite loop. In effect, the data structure described is infinite: a pair containing a T_1 and a pair containing a T_1 and.... More usefully, we can look at it as represented by a graph structure rather than a tree structure:



When considering structural equivalance of such types, we'll define two types as identical if any path of node labels (such as `pair`) and edges that exists in one also exists in the other, always beginning at the corresponding "start" node in each. We take edges as labeled—e.g. `pair` nodes have a left edge and a right edge—and the edge labels in any two corresponding paths must match up. Thus, the following two types are identical, even though one appears to have more nodes:



Fortunately, it's relatively simple to modify the algorithm of §4.7 to these more general types. Basically, as we traverse the structure, we link together nodes that are supposed to represent equivalent types so that if we encounter them again (due to circularities in one or the other structure), we can avoid rechecking them. After linking the nodes, we then check recursively (as in the previous algorithm) that the children are equivalent. To do all this, we extend the definition of `binding`, so that all nodes (not just type variables) may be bound to an expression (and all are initially unbound). With this understanding, the revised algorithm is nearly identical to the previous one:

```
/* Return true iff the pattern P matches the pattern A, and binds */
/* type variables in A and P so as to make the two match. */
match(P, A) {
    P = binding(P);
    A = binding(A);
```

```
        /* Both P and A are unbound. */
        if (P == A)
            return true;
        if (P is a type variable) {
            bind P to A;
            return true;
        }
        bind A to P;
        if (A is a type variable)
            return true;
        if (oper(P) != oper(A)
            || length(children(P)) != length(children(A)))
            return false;
        for each child cp in children(P)
                and corresponding child ca in children(A) {
            if (! match (cp, ca))
                return false;
        }
        return true;
    }
```

Each time we execute the body of `match`, we either return without executing any recursive calls or we reduce the number of unbound nodes by one. Therefore, the algorithm must terminate.

4.7.3 Overload resolution in Ada

As indicated in the last lecture, C++ disallows

```
int f2(int x) { ... } /* (3) */
A   f2(int x) { ... } /* (4) */
int x;
...
x = f2(3);
```

because the the call on `f2` cannot be resolved on the basis of the argument type alone. Ada, on the other hand, does allow this situation (well, with a different syntax, of course).

To make the problem uniform, first of all, we can treat all operators as functions. Thus, we rewrite

```
x = f2(3);
```

as

```
operator=(x, f2(3));
```

When only argument types matter, it is easy to resolve overloaded functions:

- Find the types of the arguments (recursively resolving overloading of nested calls).

- Look at all definitions of the function in question for one whose formal parameters match the resulting list of argument types (as for the `match` procedure above).

When the return type of a function matters, however, things get complicated.

The naive approach is to try all combinations of definitions of all functions mentioned in an expression (for the example above, all possible definitions of `operator=` with all possible definitions of `f2`). If the average number of overloadings for any function definition is k (the geometric mean, to be precise), and the number of function calls in an expression is N, then this naive procedure requires $\Omega(N^k)$ time, which is definitely bad.

A much better procedure is the following. We operate on an abstract syntax tree representing an expression.

- Perform a post-order traversal of the tree determining a *list of possible types* for the expression:
  - If the expression is a literal or variable, return its (single) possible type.
  - If the expressions is a function call,
    * recursively determine the possible types for each operand;
    * look at all overloadings of the function and find which of them match one of the possible types in each argument position.
    * return the list of return types for each of the overloadings that match.
- If this procedure results in more than one possible type for the expression as a whole, the expression is ambiguous and there is an error.
- Otherwise, perform a pre-order traversal of the tree, passing down the (single) type, R that the expression must have:
  - If the expression is a literal or variable, nothing needs to be done on this pass.
  - For a function call, check that only one of the possible overloaded definitions for the called function (determined on the post-order pass) returns the given type, R. If not, there is an error.
  - Otherwise, the single definition selected in the preceding step determines the types of the arguments. Recursively resolve the argument expressions, passing down these argument types as R.