# 3. Parsing

## *3.1 Introduction*

Once we have identified the tokens in our program, we then want to determine its syntactic structure. That is, we want to put the tokens together to make the larger syntactic entities: expressions, statements, methods and class definitions. This process of determining the syntactic structure of a program is called *parsing*.

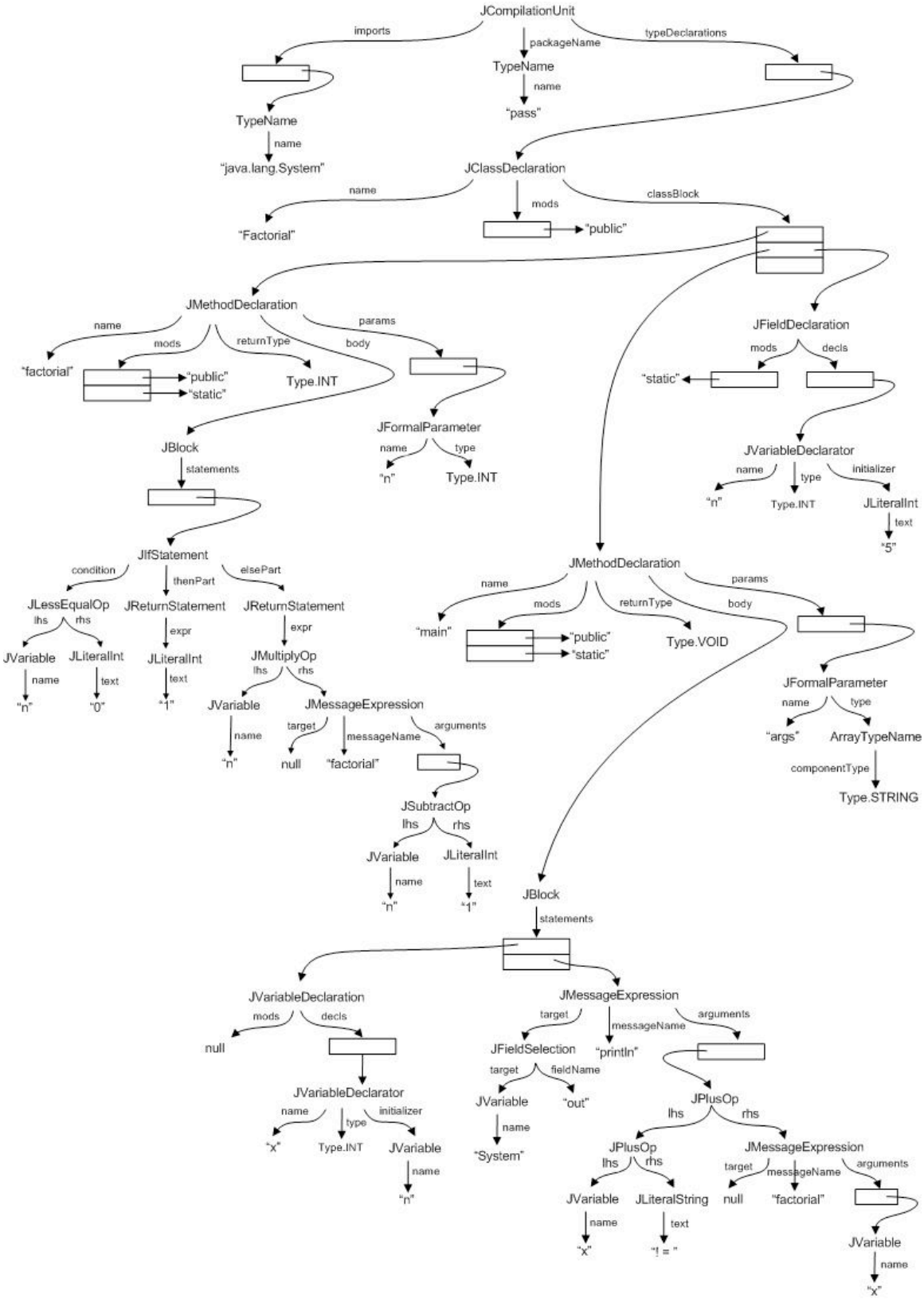
Firstly, we wish to make sure the program is syntactically valid -- that it conforms to the grammar that describes its syntax. As the parser parses the program it should identify syntax errors and report them and the line numbers they appear on. Moreover, when the parser does find a syntax error, it should not just stop, but it should report the error, and gracefully recover so that it may go on looking for additional errors.

Secondly, the parser should produce some representation of the parsed program that is suitable for semantic analysis. In the *j--* compiler, we produce an abstract syntax tree (AST).

For example, given the *j--* program we saw in Chapter 2,

```
package pass;

import java.lang.System;

public class Factorial
{
    // Two methods and a field

    public static int factorial(int n)
    {
        if (n <= 0)
            return 1;
        else
            return n * factorial(n - 1);
   }

    public static void main(String[] args)
    {
        int x = n;
        System.out.println(x + "! = " + factorial(x));
   }

    static int n = 5;
}
```

we would like to produce an abstract syntax tree such as that in Figure 3.1.




3-3

**Figure 3.1 An AST for the Factorial Program**

Notice that the nodes in the AST represent our syntactic structures. These structures are usually described by what is called a BNF (Backus-Naur Form) grammar. A BNF grammar describes the syntax of programs in a programming language. For example, Appendix B at the end of this book uses BNF for describing the syntax of *j--* and Appendix C describes the syntax of Java. BNF is described in the next section.

Why are we interested in a tree representation for our program? Because it is easier to analyze and decorate (with type information) a tree than it is to do the same with text. The abstract syntax tree makes that syntax, which is implicit in the program text, explicit. This is the purpose of parsing.

Before discussing how we might go about parsing *j--* programs, we must first talk about context-free grammars and the languages they describe.

## *3.2 Context-free Grammars and Languages*

### 3.2.1 Backus-Naur Form (BNF) and its Extensions

The grammars that we use to describe programming languages are inherently recursive and so are best described by what we call context-free grammars (cfgs). For example, the context-free rule

*(3.1)* S ::= **`if (`**E) S

says that, if E is an expression and S is a statement,

**`if (`**E) S

is also a statement. Or, we can read the rule as, a statement S can be written as an **`if`**, followed by a parenthesized expression E and finally a statement S. That is, we can read the "::=" as "can be written as a". There are abbreviations possible in the notation. For example, we can write

*(3.2)* S ::= **`if (`**E) S
| **`if (`**E) S **`else`** S

as shorthand for

(3.3) S ::= **`if (`**E) S
S ::= **`if (`**E) S **`else`** S

That is, we can read the "|" as "or". So, a statement S can be written either as an **`if`** followed by a parenthesized expression E, and a statement S; or as an **`if`**, followed by a parenthesized expression E, a statement S, an **`else`**, and another statement S.

This notation for describing programming language is called Backus-Naur Form (BNF) for John Backus and Peter Naur who used it to describe the syntax for Algol 60 in the Algol 60 Report (Naur, 1963).

The rules have all sorts of names: *BNF-rules*, *rewriting rules*, *production rules*, *productions*, or just *rules* (when the meaning is clear from the context). We shall use the terms *production rule* or simply *rule*.

There exist many extensions to BNF, made principally for reasons of expression and clarity. Our notation for describing *j--* syntax follows that used for describing Java (Gosling et al, 2005).

In our grammar, the square brackets indicate that a phrase is optional, e.g., (3.2) and (3.3) could have been written as

*(3.4)* S ::= **`if (`**E) S [**`else`** S]

says an S may be written as **if (**E) S, optionally followed by **else** S. Curly braces denote the Kleene closure, indicating that the phrase may appear zero or more times. For example,

*(3.5)* E ::= T {**+** T}

says that an expression E may be written as a term T, followed by zero or more occurrences of **+** followed by a term T:

T
T **+** T
T **+** T **+** T
…

Finally, one may use the alternation sign | inside right-hand-sides, using parentheses for grouping, e.g.

*(3.6)* E ::= T {(**+** | **-**) T}

meaning that the additive operator may be either **+** or **-**, allowing for

T **+** T **-** T **+** T

This extended BNF allows us to describe such syntax as that for a *j--* compilation unit,

```
compilationUnit ::= [package qualifiedIdentifier ;]
                    {import qualifiedIdentifier ;}
                    {typeDeclaration} EOF
```
*(3.7)*

which says that a *j--* compilation unit may optionally have a package clause at the top, followed by zero or more imports, followed by zero or more type (e.g. class) declarations[1]. The **EOF** represents the end-of-file.

These extensions do not increase the expressive power of BNF since they can be expressed in classic BNF. For example, if we allow the empty string ε on the right-hand-side of a production rule, then the optional

[X Y Z]

can be expressed as T, where T is defined by the rules

T ::= X Y Z
T ::= ε

where T occurs in no other rule. Likewise,

{X Y Z}

can be expressed as U, where U is defined by the rules

U ::= X Y Z U
U ::= ε

where U appears nowhere else in the grammar. Finally,

(X | Y | Z)

can be expressed as V, where V is defined by the rules

V ::= X
| Y

[1] Yes, syntactically this means that the input file can be empty. We often impose additional rules, enforced later, saying that there must be at most one public class declared.

| Z

where V appears nowhere else in the grammar.

Even though these abbreviations express the same sorts of languages as classic BNF, we use them for convenience. They make for more compact and more easily understood grammars.

### 3.2.2 A Grammar and the Language it describes

We define a context-free grammar (cfg) as a tuple, G = (N,T,S,P) where

- N is a set of non-terminal symbols, sometimes called *non-terminals*,
- T is a set of terminal symbols, sometimes called *terminals*,
- S ∈ N is a designated non-terminal, called the start symbol, and
- P is a set of production rules, sometimes called *productions* or *rules*.

For example, a context-free grammar that describes (very) simple arithmetic expressions is

G = (N,T,S,P)
where N = {E, T, F} is the set of non-terminals,
T = {**+**, *****, **(**,), **id**} is the set of terminals,
S = E is the start symbol, and
P = { E ::= E **+** T,
*(3.8)* E ::= T,
T ::= T ***** F,
T ::= F,
F ::= **(**E),
F ::= **id**}

A little less formally, we can denote the same grammar simply as a sequence of production rules. For example, (3.9) denotes the same grammar as does (3.8).

E ::= E **+** T
E ::= T
*(3.9)* T ::= T ***** F
T ::= F
F ::= **(**E)
F ::= **id**

We may surmise that the symbols (here E, T and F) that are defined by at least one production rule are non-terminals. Conventionally, the first defined non-terminal (i.e., E

here) is our start symbol. Those symbols that are not defined (here **+**, *****, **(**,), and **id**) are terminals.

The start symbol is important to us because it is from this symbol, using the production rules, that can generate strings in a language. For example, since we designate E to be the start symbol in the grammar above, we can record a sequence of applications of the production rules, starting from E to the sentence **id + id * id** as follows.

E ⇒ E **+** T ⇒ T **+** T ⇒ F **+** T ⇒ id **+** T ⇒ id **+** T ***** F ⇒ **id +** F ***** F ⇒
**id + id *** F ⇒ **id + id * id**

We call this a *derivation*. When a string of symbols derives another string of symbols in one step, that is, by a single application of a production rule, we say that first string *directly derives* the second string. For example,

E directly derives E **+** T
E **+** T directly derives T **+** T
T **+** T directly derives F **+** T
and so on …

When one string can be re-written as another string, using zero or more production rules from the grammar, we say the first string *derives* the second string. For this *derives* relation, we usually use the symbol ⇒*. For example,

E ⇒* E (in zero steps)
E ⇒* **id +** F ***** F
T **+** T ⇒* **id + id * id**

We say the *language* L(G) that is described by a grammar G consists of all the strings (sentences) comprised of only terminal symbols, that can be derived from the start symbol. A little more formally, we can express the language L(G) for a grammar G with start symbol S and terminals T as,

*(3.10)* L(G) = {w | S ⇒* w and w ∈ T*}

For example, in our grammar above,

E ⇒* **id + id * id**
E ⇒* **id**
E ⇒* **(id + id) * id**

so, L(G) includes each of

**id + id * id**
**id**
**(id + id) * id**

and infinitely more finite sentences.

We are interested in languages, those strings of terminals that can be derived from a grammar's start symbol. There are two kinds of derivation that will be important to us when we go about parsing these languages: left-most derivations and right-most derivations.

A *left-most derivation* is a derivation in which at each step, the next string is derived by applying a production rule for rewriting the *left-most* non-terminal. For example, we have already seen a left-most derivation of **id + id * id**:

E ⇒ E **+** T ⇒ T **+** T ⇒ F **+** T ⇒ **id +** T ⇒ **id +** T ***** F ⇒ **id +** F ***** F ⇒
**id + id *** F ⇒ **id + id * id**

Here we have under-lined the left-most non-terminal in each string of symbols in the derivation, to show that it is indeed a left-most derivation.

A *right-most derivation* is a derivation in which at each step, the next string is derived by applying a production rule for rewriting the *right-most* non-terminal. For example, the right-most derivation of **id + id * id** would go as follows.

E ⇒ E **+** T ⇒ E **+** T ***** F ⇒ E **+** T *** id** ⇒ E **+** F *** id** ⇒ E **+ id * id** ⇒
T **+ id * id** ⇒ F **+ id * id** ⇒ **id + id * id**

We use the term, *sentential form* to refer to any string of (terminal and non-terminal) symbols that can be derived from the start symbol. So, for example in the previous derivation,

E
E **+** T
E **+** T ***** F
E **+** T ***** id
E **+** F *** id**
E **+ id * id**
T **+ id * id**
F **+ id * id**
**id + id * id**

Clearly any sentential form consisting solely of terminal symbols is a sentence in the language.

An alternative representation of a derivation is the parse tree, a tree that illustrates the derivation of an input string (at the leaves) from a start symbol (at the root). For example, Figure 3.2 shows the parse tree for **`id + id * id`**.

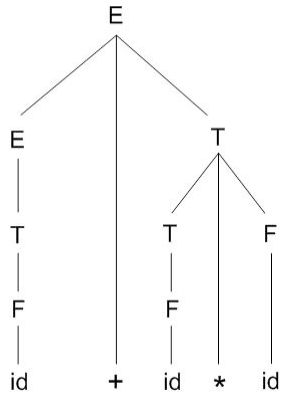



**Figure 3.2 A Parse Tree for `id + id * id`**

Consider the *j--* grammar in Appendix B. In this grammar, compilationUnit is the start symbol; from it one may derive syntactically legal *j--* programs. Not every syntactically legal *j--* program is in fact a bona fide *j--* program. For example, the grammar permits such fragments as

```
5 * true;
```

The *j--* grammar in fact describes a superset of the programs in the *j--* language. To restrict this superset to the set of legal *j--* programs, we need additional rules, such as type rules requiring, for example, that the operands to **`*`** must be integers. Type rules are enforced during semantic analysis.

### 3.2.3 Ambiguous Grammars and Unambiguous Grammars

Given a grammar G, if there exists a sentence s in L(G) for which there are more than one leftmost derivations in G (or equivalently, either more than one rightmost derivations, or more than one parse tree for s in G), we say that the sentence s is *ambiguous*. Moreover, if a grammar G describes at least one ambiguous sentence, the grammar G is *ambiguous*. If there is no such sentence, that is if *every* sentence is derivable by a unique leftmost (or rightmost) derivation (or has a unique parse tree), we say the grammar is *unambiguous*.

***Example***

For example, consider the grammar,

*(3.11)* E ::= E **+** E | E ***** E | **(**E) | **id**

And, consider the sentence **id+id*id**. One leftmost derivation for this sentence is

E ⇒ E **+** E ⇒ **id +** E ⇒ **id +** E ***** E ⇒ **id + id *** E ⇒ **id + id * id**

Another leftmost derivation of the same sentence is

E ⇒ E ***** E ⇒ E **+** E ***** E ⇒ **id +** E ***** E ⇒ **id + id *** E ⇒ **id + id * id**

Therefore, the grammar is ambiguous. It is also the case that the sentence has two *rightmost* derivations in the grammar:

E ⇒ E **+** E ⇒ E **+** E ***** E ⇒ E **+** E *** id** ⇒ E **+ id * id** ⇒ **id + id * id**

and

E ⇒ E ***** E ⇒ E *** id** ⇒ E **+** E *** id** ⇒ E **+ id * id** ⇒ **id + id * id**

These two rightmost derivations for the same sentence also show the grammar is ambiguous. Finally, the two parse trees, illustrated in Figure 3.3, for the same sentence also demonstrate ambiguity.

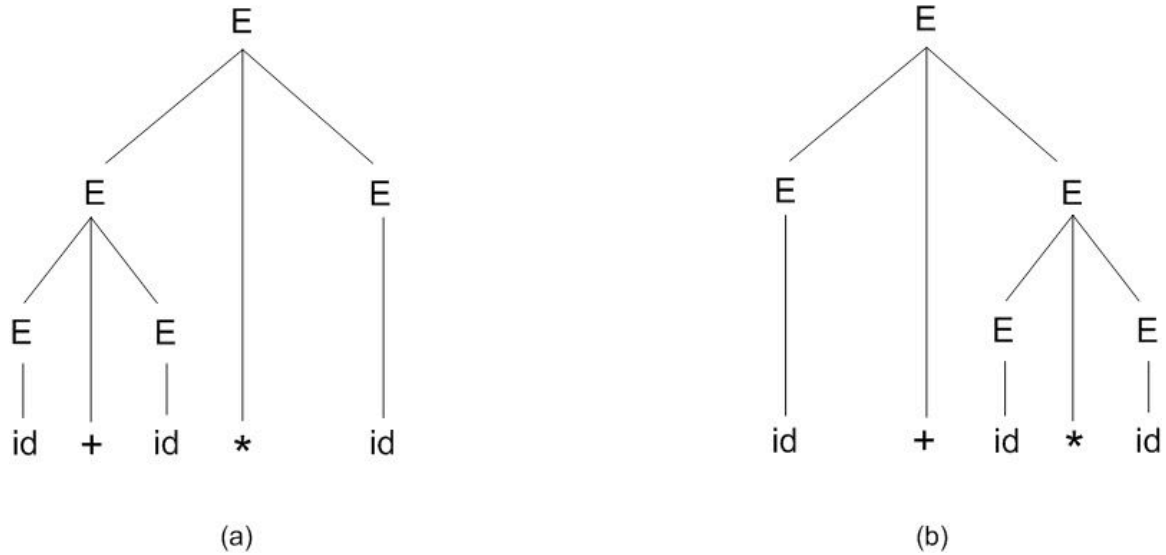



**Figure 3.3 Two Parse Trees for id + id * id**

Clearly, we would rather have unambiguous grammars describe our programming language, because ambiguity in the parsing can lead to ambiguity in assigning semantics (meaning) to programs. For example, in **`id + id * id`**, which operation is applied first: addition or multiplication? From our math days, we would like the multiplication operator **`*`** to be more binding than the addition operator **`+`**. Notice that the grammar in (3.11) does not capture this precedence of operators in the way that the grammar in (3.9) does.

***Example***

As another example, consider the grammar describing conditional statements

*(3.12)*

S ::= **`if (`**E) S
  | **`if (`**E) S **`else`** S
  | **`s`**
E ::= **`e`**

Here, the token **`e`** represents an arbitrary expression and the **`s`** represents a (non-conditional) statement.

Consider the sentence,

*(3.13)* **`if (e) if (e) s else s`**.

If we look at this sentence carefully, we see that it nests one conditional statement within another. One might ask: to which **`if`** does the **`else`** belong? We cannot know; the sentence is ambiguous. More formally, there exist two leftmost derivations for this sentence:

$\underline{S} \Rightarrow$ **`if (`** $\underline{E}$ **`)`** S **`else`** S $\Rightarrow$ **`if (e)`** $\underline{S}$ **`else`** S $\Rightarrow$ **`if (e) if (`**$\underline{E}$**`)`** S **`else`** S $\Rightarrow$

**`if (e) if (e)`** $\underline{S}$ **`else`** S $\Rightarrow$ **`if (e) if (e) s else`** $\underline{S}$ $\Rightarrow$

**`if (e) if (e) s else s`**

and

$\underline{S} \Rightarrow$ **`if (`**$\underline{E}$**`)`** S $\Rightarrow$ **`if (e)`** $\underline{S}$ $\Rightarrow$ **`if (e) if (`**$\underline{E}$**`)`** S **`else`** S $\Rightarrow$

**`if (e) if (e)`** $\underline{S}$ **`else`** S $\Rightarrow$ **`if (e) if (e) s else`** $\underline{S}$ $\Rightarrow$

**`if (e) if (e) s else s`**

The two differing parse trees in Figure 3.4 for the same sentence (3.13), also demonstrate ambiguity.

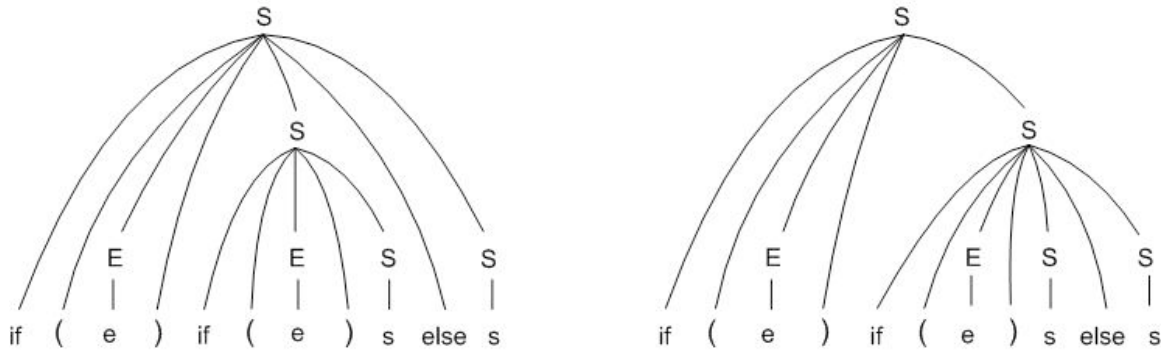



**Figure 3.4 Two Parse Trees for** if (e) if (e) s else s

If unambiguous grammars are so important, why do we see so many languages having this ambiguous conditional statement? All of *j--*, Java, C and C++ include precisely this same ambiguous conditional construct. One could easily modify the syntax of conditionals to remove the ambiguity. For example, the programming languages S-Algol and Algol-S defines conditionals so:

```
            S ::= if E do S
(3.14)          | if E then S else S
                | s
            E ::= e
```

Here the **do** indicates the simpler conditional (having no **else** clause) while the **then** indicates the presence of an **else** clause. A simple solution but it does change the language.

But programmers have become both accustomed to (and fond of) the ambiguous conditional. So both programmers and compiler writers have learned to live with it. Programming language reference manuals include extra explanations, such as

*In the conditional statement, the* **else** *clause goes with the nearest preceding* **if**.

And compiler writers handle the rule as a special case in the compiler's parser, making sure that an **else** is grouped along with the closest preceding **if**.

The *j--* grammar (and the Java grammar) have another ambiguity, which is even more difficult. Consider the problem of parsing the expression

```
x.y.z.w
```

Clearly **w** is a field; if it were a method expression, then that would be evident in the syntax:

```
x.y.z.w()
```

But, what about **x.y.z** ? There are several possibilities depending on the types of the names **x**, **y** and **z**.

- If **x** is the name of a local variable, it might refer to an object with a field **y**, referring to another object with a field **z**, referring to yet another object having our field **w**. In this case, the expression would be parsed as a cascade of field selection expressions.

- Alternatively, **x.y** might be the name of a package, in which the class **z** is defined. In this case, we would parse the expression by first parsing **x.y.z** as a fully qualified class and then parse **w** as a static field selection of that class.

- Other possibilities, parsing various permutations of (possibly qualified) class names and field selection operations, also exist.

The parser cannot determine just how the expression **x.y.z** is parsed because types are not decided until after it has parsed the program and constructed its abstract syntax tree (AST). So the parser represents **x.y.z** in the AST by an **AmbiguousName** node. Later on, after declarations have been processed, the compiler re-writes this node as the appropriate sub-tree.

These two examples of ambiguity in *j--* (and in Java) make the point that in compiling everyday programming languages, one must deal with messiness. One relies on formal techniques where practical, because formal techniques lead to more correct compilers. But once in awhile, formality fails and one must deal with special cases.

In general, there is no algorithm that can tell us whether or not an arbitrary grammar is ambiguous or not. That is, ambiguity is not decidable. But there are large classes of grammars (and so programming languages) that both can be shown to be decidable and for which efficient parsers can be automatically constructed. These classes of grammars are large enough to describe the many programming languages in use today. We look at these classes of grammars in the next sections.

Because of their recursive nature, parsing languages described by context-free grammars requires the use of a pushdown stack. In general, we can parse any language that is described by a context-free grammar but more often than not, the algorithms that do so involve backtracking.

But there are classes of grammars whose languages may be parsed *without* backtracking; the parsing is said to be *deterministic*. At each step of the parsing algorithm, the next step may be determined simply by looking at both the state of the pushdown stack and the incoming symbol (as the input sentence is scanned from left to right).

These deterministic parsers may be roughly divided into two categories:

- Top-down parsing: where the parser begins at the start symbol and, step-by-step derives the input sentence and producing either a parse tree or (more likely) an abstract syntax tree (AST) from the root down to its leaves.
- Bottom-up parsing: where the parser begins at the input sentence and, scanning it from left to right, applies the production rules, replacing right-hand sides with left-hand-sides, in a step-by-step fashion for reducing the sentence to obtain the start symbol. Here the parse tree or abstract syntax tree is built from the bottom leaves up to the top root.

## *3.3 Top-down Deterministic Parsing*

There are two popular top-down deterministic parsing strategies: parsing by recursive descent and LL(1) parsing. Both parsers scan the input from left to right, looking at and scanning just one symbol at a time.

In both cases, the parser starts with the grammar's start symbol as an initial goal in the parsing process; that is, the goal is to scan the input sentence and parse the syntactic entity represented by the start symbol. The start symbol is then rewritten, using a BNF rule replacing the symbol with the right-hand-side sequence of symbols.

For example, in parsing a *j--* program, our initial goal is to parse the start symbol, compilationUnit. The compilationUnit is defined by a single extended-BNF rule (3.15).

```
        compilationUnit ::= [package qualifiedIdentifier ;]
(3.15)                      {import qualifiedIdentifier ;}
                            {typeDeclaration} EOF
```

So, the goal of parsing a compilationUnit in the input can be rewritten as a number of sub-goals:

1. If there is a **`package`** statement in the input sentence, then we parse that.
2. If there are **`import`** statements in the input, then we parse them.
3. If there are any type declarations, then we parse them
4. Finally we parse the terminating **`EOF`** token.

Parsing a token, like **`package`**, is simple enough. The token should be the first token in the as yet un-scanned input sentence. If it is, we simply scan it; otherwise we raise an error.

Parsing a non-terminal is treated as another parsing (sub-) goal. For example, in the **`package`** statement, once we have scanned the **`package`** token, we are left with parsing a qualifiedIdentifier. This is defined by yet another BNF rule (3.16).

*(3.16)* qualifiedIdentifier ::= **`IDENTIFIER`** {**`.`** **`IDENTIFIER`**}

Here we scan an **`IDENTIFER`** (treated by the lexical scanner as a token). And, so long as we see another period in the input, we scan that period and scan another **`IDENTIFIER`**.

That we start at the start symbol, and continually rewrite non-terminals using BNF rules until we eventually reach leaves (the tokens are the leaves) makes this a *top-down* parsing technique. Because we, at each step in parsing a non-terminal, replace a parsing goal with a sequence of sub-goals, we often call this a *goal-oriented* parsing technique.

How do we decide what next step to take? For example, how do we decide whether or not there are more **`import`** statements to parse? We decide by looking at the next un-scanned input token. If it is an **`import`**, we have another **`import`** statement to parse; otherwise we go on to parsing type declarations. This kind of decision is more explicitly illustrated by the definition for the statement non-terminal (3.17).

*(3.17)*
```
statement ::= block
            | if parExpression statement [else statement]
            | while parExpression statement
            | return [expression] ;
            | ;
            | statementExpression ;
```

When faced with the goal of parsing a statement, we have six alternatives to choose from, depending on the next un-scanned input token:

1. if the next token is a **`{`**, we parse a block;
2. if the next token is an **`if`**, we parse an **`if`** statement;
3. if the next token is a **`while`**, we parse a **`while`** statement;
4. if the next token is a **`return`**, we parse a **`return`** statement;
5. if the next token is a semicolon, we parse an empty statement;
6. otherwise (or based on the next token being one of a set of tokens, any one of which may begin an expression) we parse a statementExpression.

In this instance, the decision may be made looking at the next single un-scanned input token; when this is the case, we say that our grammar is LL(1). In some cases, one must

look ahead several tokens in the input to decide which alternative to take. In all cases, because we can predict which of several alternative right-hand-sides of a BNF rule to apply, based on the next input token(s), we say this is a predictive parsing technique.

There are two principal top-down (goal-oriented, or predictive) parsing techniques available to us:

1. parsing by recursive descent; and
2. LL(1) or LL(k) parsing.

### 3.3.1 Top-down Parsing by Recursive Descent

Parsing by recursive descent involves writing a method (or procedure) for parsing each non-terminal according to the production rules that define that non-terminal. Depending on the next un-scanned input symbol, the method decides which rule to apply and then scans any terminals (tokens) in the right hand side by calling upon the **`Scanner`**, and parses any non-terminals by recursively invoking the methods that parse them.

This is a strategy we use in parsing *j--* programs. We already saw the method **`compilationUnit()`** that parses a *j--* compilationUnit in section (1.4.3).

As another example, consider the rule defining qualifiedIdentifier.

*(3.18)* qualifiedIdentifier ::= **`IDENTIFIER`** {**`. IDENTIFIER`**}

As we saw above, parsing a qualifiedIdentifier such as **`java.lang.Class`** is straightforward:

1. One looks at the next incoming token and if it is an identifier, scans it. If it is not an identifier, then one raises an error.
2. Repeatedly, so long as the next incoming token is a period:
    a. One scans the period.
    b. One looks at the next incoming token and if it is an identifier, scans it. Otherwise, one raises an error.

The method for implementing this not only parses the input, but also constructs an AST node for recording the fully qualified name as a string. The method is quite simple but introduces two helper methods: **`have()`** and **`mustBe()`**, which we use frequently throughout the parser.

```
private TypeName qualifiedIdentifier()
{
    int line = scanner.token().line();
    mustBe(IDENTIFIER);
    String qualifiedIdentifier = scanner.previousToken().image();
    while (have(DOT)) {
```

```
            mustBe(IDENTIFIER);
            qualifiedIdentifier +=
                "." + scanner.previousToken().image();
        }
        return new TypeName(line, qualifiedIdentifier);
    }
```

The method, **`have()`** is a predicate. It looks at the next incoming token (supplied by the **`Scanner`**), and if that token matches its argument, then it scans the token in the input and returns true. Otherwise, it scans nothing and returns false.

The method, **`mustBe()`** requires that the next incoming token match its argument. It looks at the next token and if it matches its argument, it scans that token in the input. Otherwise, it raises an error.[2]

As another example, consider our syntax for statements.

*(3.19)*

```
statement ::= block
            | if parExpression statement [else statement]
            | while parExpression statement
            | return [expression] ;
            | ;
            | statementExpression ;
```

As we saw above, the problem parsing a sentence against (3.19) is deciding which rule to apply when looking at the next un-scanned token.

```
private JStatement statement()
{
    int line = scanner.token().line();
    if (see(LCURLY)) {
        return block();
    }
    else if (have(IF)) {
        JExpression test = parExpression();
        JStatement consequent = statement();
        JStatement alternate = have(ELSE) ? statement() : null;
        return new JIfStatement(line, test, consequent,
            alternate);
    }
    else if (have(WHILE)) {
        JExpression test = parExpression();
        JStatement statement = statement();
        return new JWhileStatement(line, test, statement);
    }
    else if (have(RETURN)) {
```

[2] As we shall see below, **`mustBe()`** provides a certain amount of error recovery.

```
            if (have(SEMI)) {
                return new JReturnStatement(line, null);
            }
            else {
                JExpression expr = expression();
                mustBe(SEMI);
                return new JReturnStatement(line, expr);
            }
        }
        else if (have(SEMI)) {
            return new JEmptyStatement(line);
        }
        else {// Must be a statementExpression
            JStatement statement = statementExpression();
            mustBe(SEMI);
            return statement;
        }
    }
```

Notice the use of **`see()`** in looking to see if the next token is a left curly bracket **`{`**, the start of a block. Method **`see()`** is a predicate which simply looks at the incoming token to see if it is the token that we are looking for; in no case is anything scanned. The method, **`block()`** scans the **`{`**. On the other hand, **`have()`** is used to look for (and scan, if it finds it) either an **`if`**, a **`while`**, a **`return`**, or a **`;`**. If none of these particular tokens are found, then we have neither a block, an if-statement, a while-statement, a return-statement, nor an empty statement; we assume the parser is looking at a statement expression. Given that so many tokens may begin a statementExpression, we treat that as a default; if the next token is not one that may start a statementExpression then **`statementExpression()`** (or one of the methods to which it delegates the parse) will eventually detect the error. The important thing is that the error is detected before any additional tokens are scanned so its location is accurately located.

## Look-ahead

The parsing of statements works because we can determine which rule to follow in parsing the statement based only on the next un-scanned symbol in the input source. Unfortunately, this is not always the case. Sometimes we must consider the next few symbols in deciding what to do. That is, we must *look ahead* in the input stream of tokens to decide which rule to apply. For example, consider the syntax for simple unary expression.

```
        simpleUnaryExpression ::= ! unaryExpression
(3.20)                          | (basicType) unaryExpression  / / cast
                                | (referenceType) simpleUnaryExpression // cast
                                | postfixExpression
```

For this, we need special machinery. Not only must we differentiate between the two kinds (basic type and reference type) of casts; we must also distinguish a cast from a postfix expression that is a parenthesized expression, for example **`(X)`**.

Consider the Parser code for parsing a simple unary expression.

```
private JExpression simpleUnaryExpression()
{
    int line = scanner.token().line();
    if (have(LNOT)) {
        return new JLogicalNotOp(line, unaryExpression());
   }
    else if (seeCast()) {
        mustBe(LPAREN);
        boolean isBasicType = seeBasicType();
        Type type = type();
        mustBe(RPAREN);
        JExpression expr = isBasicType ? unaryExpression()
            : simpleUnaryExpression();
        return new JCastOp(line, type, expr);
   }
    else {
        return postfixExpression();
   }
}
```

Here we use the predicate **`seeCast()`** to distinguish casts from parenthesized expressions, and **`seeBasicType()`** to distinguish between casts to basic types from casts to reference types. Now, consider the two predicates.

First, the simpler **`seeBasicType()`**.

```
private boolean seeBasicType()
{
    if (see(BOOLEAN) || see(CHAR) || see(INT)) {
        return true;
   }
    else {
        return false;
   }
}
```

The predicate simply looks at the next token to see whether or not it denotes a basic type. The method **`simpleUnaryExpression()`** can use this because it has factored out the opening parenthesis, which is common to both kinds of casts.

Now consider the more difficult **`seeCast()`**.

```
private boolean seeCast()
```

```
{
    scanner.recordPosition();
    if (!have(LPAREN)) {
        scanner.returnToPosition();
        return false;
   }
    if (seeBasicType()) {
        scanner.returnToPosition();
        return true;
   }
    if (!see(IDENTIFIER)) {
        scanner.returnToPosition();
        return false;
   }
    else {
        scanner.next(); // Scan the IDENTIFIER
        // A qualified identifier is ok
        while (have(DOT)) {
            if (!have(IDENTIFIER)) {
                scanner.returnToPosition();
                return false;
            }
        }
   }
    while (have(LBRACK)) {
        if (!have(RBRACK)) {
            scanner.returnToPosition();
            return false;
        }
   }
    if (!have(RPAREN)) {
        scanner.returnToPosition();
        return false;
   }
    scanner.returnToPosition();
    return true;
}
```

Here, **`seeCast()`** must look ahead in the token stream to consider a sequence of tokens in making its decision. But our lexical analyzer, **`Scanner`** keeps track of only the single incoming symbol. For this reason, we define a second lexical analyzer, **`LookaheadScanner`**, which encapsulates our **`Scanner`** but provides extra machinery that allows one to look ahead in the token stream. It includes a means of remembering tokens (and their images) that have been scanned, a method **`recordPosition()`** for marking a position in the token stream, and **`returnToPosition()`** for returning the lexical analyzer to that recorded position (i.e. for backtracking). Of course, calls to the two methods may be nested, so that one predicate (e.g. **`seeCast()`**) may make use of another (e.g. **`seeBasicType()`**). Therefore, all of this information must be recorded on a pushdown stack.

## Error Recovery

What happens when the parser detects an error? This will happen when **`mustBe()`** comes across a token that it is not expecting. The parser could simply report the error and quit. But we would rather have the parser report the error, and then continue parsing so that it might detect any additional syntax errors. This facility for continuing after an error is detected is called *error recovery*.

Error recovery can be difficult. The parser must not only detect syntax errors but it must sufficiently recover its state so as to continue parsing without introducing additional spurious error messages. Many parser generators[3] provide elaborate machinery for programming effective error recovery.

For the *j--* Parser, we provide limited error recovery in the **`mustBe()`** method, which was proposed by (Turner, 1977). First, consider the definitions for **`see()`** and **`have()`**.

```
private boolean see(TokenKind sought)
{
    return (sought == scanner.token().kind());
}

private boolean have(TokenKind sought)
{
    if (see(sought)) {
        scanner.next();
        return true;
   }
    else {
        return false;
   }
}
```

These are defined as one would expect. Method **`mustBe()`**, defined as follows, makes use of a boolean flag, **`isRecovered`**, which is true if either no error has been detected or if the parser has recovered from a previous syntax error. It takes on the value false when it is in a state in which it has not yet recovered from a syntax error.

```
boolean isRecovered = true;

private void mustBe(TokenKind sought)
{
    if (scanner.token().kind() == sought) {
        scanner.next();
        isRecovered = true;
```

[3] A *parser generator* is a program that will take a context-free grammar (of a specified class) as input and produce a parser as output. We shall discuss the generation of parsers in subsequent sections, and we shall discuss a particular parser generator, JavaCC, in section 3.5.

```
    }
     else if (isRecovered) {
         isRecovered = false;
         reportParserError("%s found where %s sought",
             scanner.token().image(), sought.image());
    }
     else {
              // Do not report the (possibly spurious) error,
             // but rather attempt to recover by forcing a match.
         while (!see(sought) && !see(EOF)) {
             scanner.next();
        }
         if (see(sought)) {
             scanner.next();
             isRecovered = true;
        }
    }
}
```

When **`mustBe()`** first comes across an input token that it is not looking for (it is in the recovered state) it reports an error and goes into an un-recovered state. If, in a subsequent use of **`mustBe()`**, it finds another syntax error, it does not report the error, but rather it attempts to get back into a recovered state by repeatedly scanning tokens until it comes across the one it is seeking. If it succeeds in finding that token, it goes back into a recovered state. It may not succeed but instead scan to the end of the file; in this case, parsing stops. Admittedly, this is a very naïve error recovery scheme, but it works amazingly well for its simplicity[4].

### 3.3.2 LL(1) Parsing

The recursive invocation of methods in the recursive descent parsing technique depends on the underlying program stack for keeping track of the recursive calls. The LL(1) parsing technique makes this stack explicit. The first L in its name indicates a left-to-right scan of the input token stream; the second L signifies that it produces a *left parse*, which is a leftmost derivation. The (1) indicates we just look ahead at the next *one* symbol in the input to make a decision.

Like recursive descent, the LL(1) technique is top-down, goal-oriented and predictive.

#### 3.3.2.1 The LL(1) Parsing Algorithm

[4] It reminds us of the story of the dancing dog: one doesn't ask how well the dog dances but is amazed that he dances at all.

An LL(1) parser works in typical top-down fashion. At the start, the start symbol is pushed onto the stack as the initial goal. Depending on the first token in the input stream of tokens to be parsed, the start symbol is replaced on the stack by a sequence of symbols from the right-hand-side of a rule defining the start symbol. Parsing continues by parsing each symbol as it is removed from the top of the stack:

- If the top symbol was a terminal, it scans that terminal from the input. If the next incoming token does not match the terminal on the stack then an error is raised.
- If the top symbol was a non-terminal, the parser looks at the next incoming token in the input stream to decide what production rule to apply to expand the non-terminal taken from the top of the stack.

Every LL(1) parser shares the same basic parsing algorithm, which is table-driven. A unique parsing table is produced for each grammar. This table has a row for each non-terminal that can appear on the stack, and a column for each terminal token, including special terminator **#** to mark the end of the string. The parser consults this table, given the non-terminal on top of the stack and the next incoming token to determine which BNF rule to use in rewriting the non-terminal. It is important that the grammar be such that one may always unambiguously decide what BNF rule to apply; equivalently, no table entry may contain more than one rule.

For example, consider the following grammar.

*(3.21)*

1. E ::= T E'
2. E' ::= **+** T E'
3. E' ::= ε
4. T ::= F T'
5. T' ::= ***** F T'
6. T' ::= ε
7. F ::= **(**E)
8. F ::= **id**

This grammar describes the same language as that described by the grammar (3.8) in section 3.2.2. Another grammar that, using the extensions described in section 3.2.1, describes the same language is

*(3.22)*

E ::= T {**+** T}
T ::= F {***** F}
F ::= **(**E) | **id**

Such a grammar lends itself to parsing by recursive descent. But, the advantage of the grammar in (3.21) is that we can define an LL(1) parsing table for it. The parsing table for the grammar in (3.21) is given in Figure (3.5).

| | + | * | ( | ) | id | # |
|---|---|---|---|---|---|---|
| E | | | 1 | | 1 | |
| E′ | 2 | | | 3 | | 3 |
| T | | | 4 | | 4 | |
| T′ | 6 | 5 | | 6 | | 6 |
| F | | | 7 | | 8 | |

**Figure 3.5 LL(1) Parsing Table for the Grammar in (3.21)**

The numbers in the table's entries refer to the numbers assigned to BNF rules in the example grammar (3.21). An empty entry indicates an error: when the parser has the given non-terminal on top of the stack and the given token as the next input symbol, the parser is in an erroneous state. The LL(1) parsing algorithm is parameterized by this table. As we mentioned earlier, it makes use of an explicit pushdown stack.

***Algorithm 3.1: LL(1) Parsing Algorithm***

Input: a sentence w, where w is a string of terminals followed by a terminator **#**.
Output: a *left-parse*, which is a left-most derivation for w.

```
Initially,
        Stack stk initially contains the terminator # and the start symbol S, with S on top;
        Symbol sym points to the first symbol in the sentence w.
for (;;) {
        Symbol top = stk.pop();
        if (top == sym == #)  halt successfully;
        else if (top is a terminal symbol) {
                if (top == sym) {
                        advance sym to point to the next symbol in w;
                } else {
                        halt with error: a sym found where a top was expected;
                }
        } else if (top is a non-terminal, Y) {
                int index = table[Y,sym];
                if (index != err} {
                        rule = rules[index];
```

```
                    Say rule is  Y ::= X1X2…Xn
                    Push Xn,..., X2, X1 onto the stack stk, with X1 on top
                }
                else {
                   halt with an error.
                }
        }
}
```

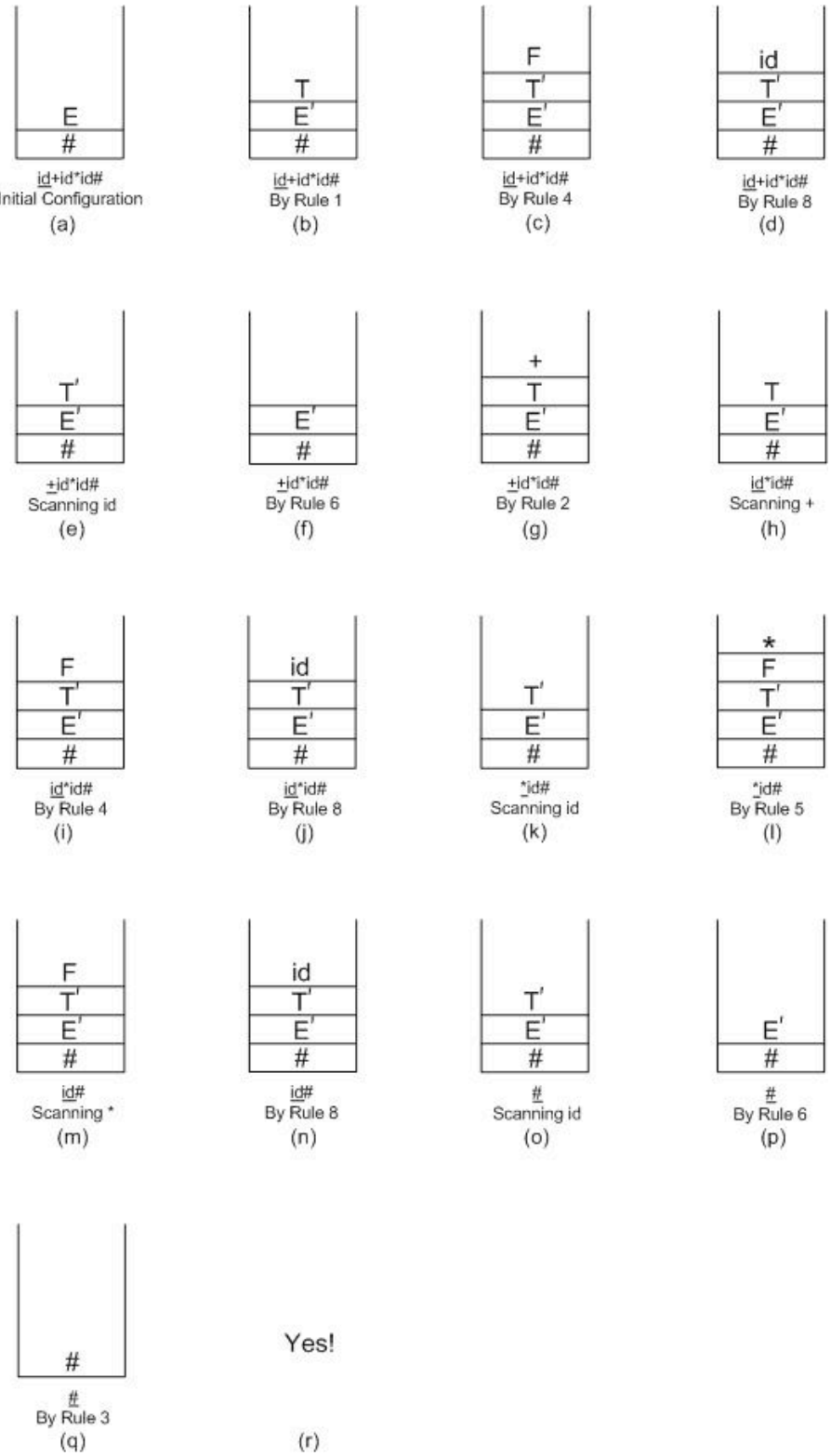



**Figure 3.6 The Steps in Parsing `id+id*id` Against the LL(1) Parsing Table in Figure 3.5**

For example, consider parsing the input sentence, **id + id * id #**. The parser would go through the states illustrated in Figure 3.6.

a. Initially, the parser begins with a **#** and E on top of its stack. E represents the goal -- the entity that we wish to parse.
b. Seeing an **id** as the incoming token, table[E,id] = 1 tells us to apply rule 1, E ::= T E', and replace the E on the stack with the right hand side, T and E', with T on top. Basically, we plan to accomplish the goal of parsing an E by accomplishing the two sub goals of parsing a T and then a E'.
c. Seeing an **id** for the incoming token, we apply rule 4, T ::= F T', and replace the T on the stack with the right hand side F and T', with F on top.
d. We apply rule 8, F ::= **id**, replacing the top goal of F with an **id**.
e. The goal of parsing the **id** on top of the stack is trivially satisfied: we pop the **id** from the stack and scan the **id** in the input; the next incoming symbol is now a **+**.
f. Now we have the goal of parsing a T' when looking at the input, **+**. We apply rule 6, T' ::= ε, replacing the T' on the stack with the empty string (that is, nothing at all). Just the goal E' remains on the stack.
g. Now the goal E', when looking at the input token **+**, is replaced using rule 2 by **+**, T and E', with the **+** on top.
h. The **+** is trivially parsed; it is popped from the stack and scanned from the input. The next incoming token is **id**.
i. Applying rule 4, T ::= F T', we replace the T on the stack by F and T', with F on top.
j. Applying rule 8, we replace the F on the stack with **id**.
k. The goal of parsing the **id** on top of the stack is trivially satisfied: we pop the **id** from the stack and scan the **id** in the input; the goal on the stack is now a T' and the next incoming symbol is now a *****.
l. We apply rule 5 to replace the T' on the stack with a *****, F and another T'. The ***** is on top.
m. The ***** on top is easily parsed: it is popped from the stack and scanned from the input.
n. The goal F, atop the stack, when looking at an incoming **id**, is replaced by an **id** using rule 8.
o. The **id** is popped from the stack and scanned from the input, leaving a T' on the stack and the terminator **#** as the incoming token.
p. We apply rule 6, replacing T' by the empty string.
q. We apply rule 3, replacing E' by the empty string.
r. There's a **#** on top of the stack and a **#** as input. We're done!

An alternative to Figure 3.6 for illustrating the states that the parser goes through is as follows:

| **Stack** | **Input** | **Output** |
|---|---|---|
| **#**E | **id+id*id#** | |
| **#**E'T | **id+id*id#** | 1 |

| | | |
|---|---|---|
| #E'T'F | **id+id*id#** | 4 |
| #E'T'**id** | **id+id*id#** | 8 |
| #E'T' | **+id*id#** | |
| #E' | **+id*id#** | 6 |
| #E'T**+** | **+id*id#** | 2 |
| #E'T | **id*id#** | |
| #E'T'F | **id*id#** | 4 |
| #E'T'**id** | **id*id#** | 8 |
| #E'T' | ***id#** | |
| #E'T'F***** | ***id#** | 5 |
| #E'T'F | **id#** | |
| #E'T'**id** | **id#** | 8 |
| #E'T' | **#** | |
| #E' | **#** | 6 |
| # | **#** | 3 |

We are left with the question: How do we construct the parsing table in Figure 3.5? Consider what the entries in the table tell us:

table[Y, a] = i, where i is the number of the rule Y ::= $X_1X_2\ldots X_n$

says that if there is the goal Y on the stack and the next un-scanned symbol is a, then we can rewrite Y on the stack as the sequence of sub goals $X_1X_2\ldots X_n$. So, the question becomes, when do we replace Y with $X_1X_2\ldots X_n$ as opposed to something else? If we consider the last two rules of our grammar (3.21),

(7) F ::= **(**E)
(8) F ::= **id**

and we have the non-terminal F on top of our parsing stack, the choice is simple. If the next un-scanned symbol is an open parenthesis **(**, we apply rule (7); and if it is an **id**, we apply rule (8). That is,

table[F, (] = 7
table[F, **id**] = 8

The problem becomes slightly more complicated when the right-hand-side of the rule either starts with a non-terminal or is simply ε.

In general, assuming both α and β are (possibly empty) strings of terminals and non-terminals,

table[Y, a] = i, where i is the number of the rule $Y ::= X_1X_2\ldots X_n$

if either

1. $X_1X_2\ldots X_n \Rightarrow^* a\alpha$, or
2. $X_1X_2\ldots X_n \Rightarrow^* \varepsilon$, and there is a derivation $S\# \Rightarrow^* \alpha Ya\beta$, that is, a can follow Y in a derivation.

For this we need two helper functions, first and follow.

### 3.3.2.2 First and Follow

We define,

$$first(X_1X_2\ldots X_n) = \{a \mid X_1X_2\ldots X_n \Rightarrow^* a\alpha, a \in T\}$$

That is, the set of all terminals that can start strings derivable from $X_1X_2\ldots X_n$. Also, if

$$X_1X_2\ldots X_n \Rightarrow^* \varepsilon$$

Then we say that $first(X_1X_2\ldots X_n)$ includes ε.

We define two algorithms for computing first: one for a single symbol and the other for a sequence of symbols. The two algorithms are both mutually recursive and make use of *memoization* – that is, what was previously computed for each set is remembered upon each iteration.

***Algorithm 3.2: Compute first(X) for all symbols X in a Grammar G.***

1. For each terminal x, first(x) = {x}.
2. For each non-terminal X, first(X) = {}.
3. If there is a rule X ::= ε in P, add ε to first(X).
4. Repeat until no new symbols are added to any set:
   For each rule $Y ::= X_1X_2\ldots X_n$ in P,
   add all symbols from $first(X_1X_2\ldots X_n)$ to first(Y).

***Algorithm 3.3: Compute first($X_1X_2\ldots X_n$) for a Grammar G.***

1. Set S = $first(X_1)$.
2. i = 2
3. While ε ∈ S and i <= n {
   S = S - ε
   Add $first(X_i)$ to S

i = i + 1

4. Return S

Algorithm 3.3 says that to compute first for a sequence of symbols $X_1$ $X_2$…Xn, we start with first($X_1$). If $X_1 \Rightarrow^* \varepsilon$, then we must also include first($X_2$). If $X_2 \Rightarrow^* \varepsilon$, then we must also include first($X_3$). And, so on.

***Example***

For example, consider our example grammar in (3.21).

The computation of first for the terminals, by step 1 of Algorithm 3.2 is trivial:

first(**+**) = {**+**}
first(*****) = {*****}
first( **(**) = { **(**}
first(**)** ) = {)}
first(**id**) = {**id**}

Step 2 of Algorithm 3.2 gives

first(E) = {}  first(T') = {}
first(E') = {}  first(F) = {}
first(T) = {}

Step 3 of Algorithm 3.2 yields

first(E') = {ε}  first(T') = {ε}

Step 4 of Algorithm 3.2 is repeatedly executed until no further symbols are added. The first round yields

first(E) = {}
first(E') = {**+**, ε}
first(T) = {}
first(T') = {*****, ε}
first(F) = { **(**, **id**}

The second round of step 4 yields

first(E) = {}
first(E') = {**+**, ε}

first(T) = { **(**, **id**}
first(T') = {*****, ε}
first(F) = { **(**, **id**}

The third round of step 4 yields

first(E) = { **(**, **id**}
first(E') = {**+**, ε}
*(3.23)* first(T) = { **(**, **id**}
first(T') = {*****, ε}
first(F) = { **(**, **id**}

The forth round of step 4 adds no symbols to any set, leaving us with (3.23).

We are left with the question as to when is a rule,

$$X ::= \varepsilon$$

applicable? For this we need the notion of *follow*.

We define,

$$follow(X) = \{a \mid S \Rightarrow^* wX\alpha \text{ and } \alpha \Rightarrow^* a\ldots\},$$

that is, all terminal symbols that start terminal strings derivable from what can follow X in a derivation. Another definition is as follows.

1. follow(S) contains **#**, i.e. the terminator follows the start symbol..
2. If there is a rule $Y ::= \alpha X \beta$ in P, follow(X) contains first(β) - {ε}.
3. If there is a rule $Y ::= \alpha X \beta$ in P and either $\beta = \varepsilon$ or first(β) contains ε, follow(X) contains follow(Y).

This definition suggests a straightforward algorithm.

***Algorithm 3.4: Compute follow(X) for all non-terminals X in a Grammar G.***

1. follow(S) = {**#**}.
2. follow(X) = {} for all non-terminals $X \neq S$.
3. Repeat until no new symbols are added to any set:
   for each rule $Y ::= X_1X_2\ldots X_n$ in P,
   for each non-terminal $X_i$,
   add first($X_{i+1}X_{i+2}\ldots X_n$) - {ε}. to follow($X_i$), and
   if $X_i$ is the rightmost symbol or first($X_{i+1}X_{i+2}\ldots X_n$) contains ε,

add follow(Y) to follow($X_i$).

***Example***

Again, consider our example grammar (3.21).

By steps 1 and 2 of algorithm 3.4,

follow(E) = {**#**}
follow(E') = {}
follow(T) = {}
follow(T') = {}
follow(F) = {}

Round 1 of step 3 yields,

- From rule 1, E ::= T E', follow(T) contains first(E') - {ε} = {**+**}, and because first(E') contains ε, it also contains follow(E). Also, follow(E') contains follow(E). So, we have

  follow(E') = {**#**}
  follow(T) = {**+**, **#**}

- We get nothing additional from rule 2, E' ::= **+** T E'. follow(T) contains first(E') - {ε}, but we saw that in rule 1. Also, follow(E') contains follow(E'), but that says nothing.

- We get nothing additional from rule 3, E' ::= ε.

- From rule 4, T ::= F T', follow(F) contains first(T') - {ε} = {*}, and because first(T') contains ε, it also contains follow(T). Also, follow(T') contains follow(T). So, we have

  follow(T') = {**+**, **#**}
  follow(F) = {**+**, ***, **#**}

- Rules 5 and 6 give us nothing new (for the same reasons rules 2 and 3 did not).

- Rule 7, F ::= **(**E), adds) to follow(E), so

  follow(E) = {), **#**}

- Rule 8 gives us nothing.

Summarizing round 1 of step 3, gives

follow(E) = {), **#**}
follow(E') = {**#**}
follow(T) = {**+**, **#**}
follow(T') = {**+**, **#**}
follow(F) = {**+**, ***, **#**}

Now, in round 2 of step 3, the) that was added to follow(E) trickles down into the other follow sets:

- From rule 1, E ::= T E', because first(E') contains ε, follow(T) contains follow(E). Also, follow(E') contains follow(E). So, we have

  follow(E') = {), **#**}
  follow(T) = {**+**,), **#**}

- From rule 4, T ::= F T', because first(T') contains ε, follow(F) contains follow(T). Also, follow(T') contains follow(T). So, we have

  follow(T') = {**+**,), **#**}
  follow(F) = {**+**, ***,), **#**}

So round 2 produces

follow(E) = {), **#**}
follow(E') = {), **#**}
*(3.24)* follow(T) = {**+**,), **#**}
follow(T') = {**+**,), **#**}
follow(F) = {**+**, ***,), **#**}

Round 3 of step 3 adds no new symbols to any set, so we are left with (3.24).

### 3.3.2.3 Constructing the LL(1) Parse Table

Recall, assuming both α and β are (possibly empty) strings of terminals and non-terminals,

table[Y, a] = i, where i is the number of the rule Y ::= $X_1X_2\ldots X_n$

if either

1. $X_1X_2\ldots X_n \Rightarrow^* a\alpha$, or

2. $X_1X_2\ldots X_n \Rightarrow^* \varepsilon$, and there is a derivation $S\# \Rightarrow^* \alpha Y a \beta$, that is, a can follow Y in a derivation.

This definition, together with the functions, first and follow, suggest Algorithm 3.5.

***Algorithm 3.5: Construct an LL(1) Parse Table for a Grammar G=(N,T,S,P)***

For each non-terminal Y in G,
  For each rule $Y ::= X_1X_2\ldots X_n$ in P with number i,
    1. For each terminal a in $first(X_1X_2\ldots X_n) - \{\varepsilon\}$, add i to table[Y, a].
    2. If $first(X_1X_2\ldots X_n)$ contains ε, for each terminal (or #) in follow(Y), add i to table[Y, a].

***Example***

Let as construct the parse table for (3.21)

- For the non-terminal E, we consider rule 1: E ::= T E'. first(T E') = { **(**, **id**}. So,
  table[E, **(**] = 1
  table[E, **id**] = 1
  Because first(T E') doesn't contain ε, we need not consider follow(E).

- For the non-terminal E', we first consider rule 2: E' ::= **+** T E'; first(**+** T E') = {**+**} so

  table[E', **+**] = 2

  Rule 3: E' ::= ε is applicable for symbols in follow(E') = {), **#**}, so

  table[E',)] = 3
  table[E', **#**] = 3

- For the non-terminal T, we consider rule 4: T ::= F T'. first(F T') = { **(**, **id**}, so

  table[T, **(**] = 4
  table[T, **id**] = 4

  Because first(F T') does not contain ε, we need not consider follow(T).

- For non-terminal T', we first consider rule: 5: T' ::= ***** F T'; first(***** F T'), so

  table[T', *****] = 5

Rule 6: E' ::= ε is applicable for symbols in follow(T') = {**+**,), **#**}, so

table[T', **+**] = 6
table[T',)] = 6
table[T', **#**] = 6

- For the non-terminal F, we have two rules. Firstly, given rule 7: F ::= **(**E), and that first( **(**E)) = { **(**},

  table[F, **(**] = 7

Secondly, given rule 8: F ::= **id**, and since (obviously) first(**id**) = {**id**},

table[F, **id**] = 8

### 3.3.2.4 The LL(1) and LL(k) Grammars

We say a grammar is LL(1) if the parsing table produced by Algorithm 3.5 produces no conflicts, that is no entries having more than one rule. If there were more than one rule, then the parser would no longer be deterministic.

Furthermore, if a grammar is shown to be LL(1) then it is unambiguous. This is easy to show. An ambiguous grammar would lead to two leftmost derivations that differ in at least one step, meaning at least two rules are applicable at that step. So an ambiguous grammar cannot be LL(1).

It is possible for a grammar not to be LL(1) but LL(k) for some k > 1. That is, the parser should be able to determine its moves looking k symbols ahead. In principle, this would mean a table having columns for each combination of k symbols. But this would lead to very large tables; indeed the table size grows exponentially with k and would be unwieldy even for k = 2. On the other hand, an LL(1) parser generator based on the table construction Algorithm 3.5, might allow one to specify a k-symbol look ahead for specific non-terminals or rules. These special cases can be handled specially by the parser and so need not lead to overly large (and, most likely sparse) tables. The JavaCC parser generator, which we discuss in section 3.5, makes use of this focused k-symbol look ahead strategy.

### 3.3.2.5 Removing Left Recursion and Left-Factoring Grammars

Not all context-free grammars are LL(1). But for many that are not, one may define equivalent grammars (that is, grammars describing the same language) that are LL(1).

### *Left Recursion*

One class of grammar that is not LL(1) is a grammar having a rule with left recursion, for example *direct left recursion*,

*(3.25)*
$$Y ::= Y\alpha$$
$$Y ::= \beta$$

Clearly, a grammar having these two rules is not LL(1), because, by definition, first($Y\alpha$) must include first($\beta$) making it impossible to discern which rule to apply for expanding Y. But introducing an extra non-terminal, an extra rule, and replacing the left recursion with right recursion easily removes the direct left recursion:

*(3.26)*
$$Y ::= \beta\ Y'$$
$$Y' ::= \alpha\ Y'$$
$$Y' ::= \varepsilon$$

### *Example*

Such grammars are not unusual. For example, the first context-free grammar we saw (3.8) describes the same language as does the (LL(1)) grammar (3.21). We repeat this grammar as (3.27).

*(3.27)*
$$E ::= E + T$$
$$E ::= T$$
$$T ::= T\ *\ F$$
$$T ::= F$$
$$F ::= (E)$$
$$F ::= \texttt{id}$$

The left recursion nicely captures the left-associative nature of the operators **+** and *****. But because the grammar has left-recursive rules, it is not LL(1). We may apply the left-recursion removal rule (3.26) to this grammar.

Firstly, applying the rule to E to produce

$$E ::= TE'$$
$$E' ::= \texttt{+}\ T\ E'$$
$$E' ::= \varepsilon$$

Applying the rule to T yields

$$T ::= F\ T'$$
$$T' ::= \texttt{*}\ F\ T'$$

T' ::= ε

Giving us the LL(1) grammar,

E ::= T E'
E' ::= **+** T E'
E' ::= ε
*(3.28)* T ::= F T'
T' ::= ***** F T'
T' ::= ε
F ::= **(**E)
F ::= **id**

Where have we seen this grammar before?

Much less common, particularly in grammars describing programming languages, is *indirect left recursion*. Algorithm 3.6 deals with these rare cases.

***Algorithm 3.6: Left Recursion Removal for a Grammar G=(N,T,S,P)***

1. Arbitrarily enumerate the non-terminals of G: $X_1, X_2, \ldots, X_n$ .
2. For i := 1 to n do
   For j:= 1 to i-1 do
   Replace each rule in P having the form $X_i ::= X_j\alpha$ by the rules
   $X_i ::= \beta_1\alpha \mid \beta_2\alpha \mid \ldots \mid \beta_k\alpha$
   Where $X_j ::= \beta_1 \mid \beta_2 \mid \ldots \beta_k$ are the current rules defining $X_i$
   Eliminate any immediate left recursion using (3.25).

***Example***

Consider the following grammar.

S ::= Aa | b
A ::= Sc | d

In step 1 we can enumerate the non-terminals using subscripts to record the numbering: $S_1$ and $A_2$. This gives us a new set of rules:

$S_1$ ::= $A_2$a | b
$A_2$ ::= $S_1$c | d

In the first iteration of step 2 (i=1), no rules apply. In the second iteration (i=1, j=2), the rule

$A_2$ ::= $S_1$c

applies. We replace it with two rules, expanding $S_1$, to yield

$S_1$ ::= $A_2$a | b
$A_2$ ::= $A_2$ac | bc | d

We then use the transformation (3.26) to produce the grammar

$S_1$ ::= $A_2$a | b
$A_2$ ::= bc$A_2$' | d$A_2$'
$A_2$' ::= ac$A_2$'
$A_2$' ::= ε

Or, removing the subscripts,

S ::= Aa | b
A ::= bcA' | dA'
A' ::= acA'
A' ::= ε

### *Left Factoring*

Another common property of grammars that violates the LL(1) property is when two or more rules defining a non-terminal share a common prefix

Y ::= αβ
Y ::= αγ

The common α violates the LL(1) property. But, so long as first(β) and first(γ) are disjoint, this is easily solved by introducing a new non-terminal:

Y ::= αY'
*(3.29)* Y' ::= β
Y' ::= γ

### *Example*

For example, reconsider (3.14)

S ::= **`if`** E **`do`** S

| **if** E **then** S **else** S
| **s**
E ::= **e**

following the re-writing rule (3.29), we can reformulate the grammar as

S ::= **if** E S'
| **s**
S' ::= **do** S
| **then** S **else** S
E ::= **e**

## 3.4 Bottom-up Deterministic Parsing

In bottom-up parsing, one begins with the input sentence and scanning it from left-to-right, recognizes sub-trees at the leaves and builds a complete parse tree from the leaves up to the start symbol at the root.

### 3.3.1 The Shift-Reduce Parsing Algorithm

For example, consider our old friend, the grammar (3.8) repeated here as (3.30).

*(3.30)*
1. E ::= E **+** T
2. E ::= T
3. T ::= T ***** F
4. T ::= F
5. F ::= **(**E)
6. F ::= id

And say we want to parse the input string, **id+id*id**. We would start off with the initial configuration.

| Stack | Input | Action |
|---|---|---|
| **#** | **id+id*id#** | |

At the start, the terminator is on the stack and the input consists of the entire input sentence followed by the terminator. The first action is to *shift* the first un-scanned input symbol (it is underlined) onto the stack.

| Stack | Input | Action |
|---|---|---|
| **#** | **id+id*id#** | shift |
| **#id** | **+id*id#** | |

From this configuration, the next action is to *reduce* the id on top of the stack to an F using rule 6.

| Stack | Input | Action |
|---|---|---|
| **#** | **id+id*id#** | shift |
| **#id** | **+id*id#** | reduce (6) |
| #F | **+id*id#** | |

From this configuration, the next two actions involve reducing the F to a T (by rule 4), and then to an E (by rule 2).

| Stack | Input | Action |
|---|---|---|
| **#** | **id+id*id#** | shift |
| **#id** | **+id*id#** | reduce (6) |
| #F | **+id*id#** | reduce (4) |
| #T | **+id*id#** | reduce (2) |
| #E | **+id*id#** | |

The parser continues in this fashion, by a sequence of shifts and reductions, until we reach a configuration where #E is on the stack (E on top) and the sole un-scanned symbol in the input is the terminator, **#**. At this point, we have reduced the entire input string to the grammar's start symbol E, so we can say the input is *accepted*.

| Stack | Input | Action |
|---|---|---|
| **#** | **id+id*id#** | shift |
| **#id** | **+id*id#** | reduce (6) |
| #F | **+id*id#** | reduce (4) |
| #T | **+id*id#** | reduce (2) |
| #E | **+id*id#** | shift |
| #E**+** | **id*id#** | shift |
| #E**+id** | ***id#** | reduce (6) |
| #E+F | ***id#** | reduce (4) |
| #E+T | ***id#** | shift |
| #E+T***** | **id#** | shift |
| #E+T***id** | **#** | reduce (6) |
| #E+T*F | **#** | reduce (3) |
| #E+T | **#** | reduce (1) |
| #E | **#** | accept |

Notice that the sequence of reductions 6, 4, 2, 6, 4, 6, 3, 1 represents the rightmost derivation of the input string but in *reverse*:

$\underline{E} \Rightarrow E + \underline{T} \Rightarrow E + T * \underline{F} \Rightarrow E + \underline{T} * id \Rightarrow E + \underline{F} * id \Rightarrow \underline{E} + id * id \Rightarrow$
$\underline{T} + id * id \Rightarrow \underline{F} + id * id \Rightarrow id + id * id$

That it is in reverse makes sense because this is a bottom-up parse. The question arises: how does the parser know when to *shift* and when to *reduce*? When reducing, how many symbols on top of the stack play a role in the reduction? And, when reducing, by *which rule* does it make its reduction?

For example, in the derivation above, when the stack contains **#**E**+**T and the next incoming token is a *****, how do we know that we are to *shift* (the * onto the stack) rather than *reduce* either the E**+**T to an E or the T to an E?

Notice two things:

1. Ignoring the terminator **#**, the stack configuration combined with the un-scanned input stream represents a sentential form in a rightmost derivation of the input.
2. The part of the sentential form that is reduced to a non-terminal is always on top of the stack. So all actions take place at the *top* of the stack. We either shift a token onto the stack, or we reduce what is already there.

We call the sequence of terminals on top of the stack that are reduced to a single non-terminal at each reduction step the *handle*. More formally, in a rightmost derivation,

$S \Rightarrow^* \alpha Y w \Rightarrow \alpha\beta w \Rightarrow^* uw$, where uw is the sentence,

the *handle* is the rule $Y ::= \beta$ and a position in the right sentential form $\alpha\beta w$ where $\beta$ may be replaced by Y to produce the previous right sentential form $\alpha Y w$ in a rightmost derivation from the start symbol S. Fortunately, there are a finite number of possible handles that may appear on top of the stack.

So, when a handle appears on top of the stack,

| Stack | Input |
|---|---|
| **#**$\alpha\beta$ | w |

we reduce that handle ($\beta$ to Y in this case).

Now if $\beta$ is the sequence $X_1, X_2, \ldots, X_n$, then we call any subsequence, $X_1, X_2, \ldots, X_i$, for $i \leq n$ a *viable prefix*. Only viable prefixes may appear on the top of the parse stack. If there is not a handle on top of the stack and shifting the first un-scanned input token from the input to the stack results in a viable prefix, a shift is called for.

## 3.4.2 LR(1) Parsing

One way to drive the shift/reduce parser is by a kind of DFA that recognizes viable prefixes and handles. The tables that drive our LR(1) parser are derived from this DFA.

### 3.4.2.1 The LR(1) Parsing Algorithm

Before showing how the tables are constructed, let us see how they are used to parse a sentence. The LR(1) parser algorithm is common to all LR(1) grammars and is driven by two tables, constructed for particular grammars: an Action table and a Goto table.

***Algorithm 3.6: The LR(1) Parsing Algorithm***

The algorithm is a state machine with a pushdown stack, driven by two tables: Action and Goto. A configuration of the parser is a pair, consisting of the state of the stack and the state of the input:

| Stack | Input |
|---|---|
| $s_0\ X_1\ s_1\ X_2\ s_2\ \ldots\ X_m\ s_m$ | $a_k\ a_{k+1}\ \ldots\ a_n$**#** |

where the $s_i$ are states, the $X_i$ are (terminal or non-terminal) symbols, and $a_k\ a_{k+1}\ \ldots\ a_n$ are the un-scanned input symbols. This configuration represents a right sentential form in a rightmost derivation of the input sentence,

$X_1X_2\ldots\ X_m\ a_k\ a_{k+1}\ \ldots\ a_n$

Input: w, the input sentence to be parsed, followed by the terminator **#**.
Output: A rightmost derivation in reverse.

Initially, the parser has the configuration,

| Stack | Input |
|---|---|
| $s_0$ | $a_1\ a_2\ \ldots\ a_n$ **#** |

where $a_1\ a_2\ \ldots\ a_n$ is the input sentence.

Repeat until either the sentence is parsed or an error is raised
If Action[$s_m$, $a_k$] = s $s_i$, the parser executes a *shift* (the s stands for "shift") and goes into state $s_i$, going into the configuration

| Stack | Input |
|---|---|
| $s_0\ X_1\ s_1\ X_2\ s_2\ \ldots\ X_m\ s_m\ a_k\ s_i$ | $a_{k+1}\ \ldots\ a_n$**#** |

Otherwise, if Action[$s_m$, $a_k$] = r i (the r stands for "reduce"), where i is the number of the production rule Y ::= $X_j$ $X_{j+1}$… $X_m$ , then replace the symbols and states $X_j$ $s_j$ $X_{j+1}$ $s_{j+1}$ … $X_m$ $s_m$ by Ys, where s = Goto[$s_{j-1}$, Y]. The parser then outputs production number, i. The parser goes into the configuration,

| Stack | Input |
|---|---|
| $s_0$ $X_1$ $s_1$ $X_2$ $s_2$ … $X_{j-1}$ $s_{j-1}$ Ys | $a_{k+1}$ … $a_n$**#** |

Otherwise, if Action[$s_m$, $a_k$] = accept then the parser halts and the input has been successfully parsed.

Otherwise, if Action[$s_m$, $a_k$] = error then the parser raises an error. The input is not in the language.

For example, consider (again) our grammar for simple expressions, now in (3.31).

*(3.31)*

1. E ::= E **+** T
2. E ::= T
3. T ::= T ***** F
4. T ::= F
5. F ::= (E)
6. F ::= **id**

The Action and Goto tables are given in Figure 3.7.

| | Action | | | | | | Goto | | |
|---|---|---|---|---|---|---|---|---|---|
| | + | * | ( | ) | id | # | E | T | F |
| 0 | | | s4 | | s5 | | 1 | 2 | 3 |
| 1 | s6 | | | | | accept | | | |
| 2 | r2 | s7 | | | | r2 | | | |
| 3 | r4 | r4 | | | | r4 | | | |
| 4 | | | s11 | | s12 | | 8 | 9 | 10 |
| 5 | r6 | r6 | | | | r6 | | | |
| 6 | | | s4 | | s5 | | | 13 | 3 |
| 7 | | | s4 | | s5 | | | | 14 |
| 8 | s16 | | s15 | | | | | | |
| 9 | | s17 | | r2 | | r2 | | | |
| 10 | r4 | r4 | | r4 | | | | | |
| 11 | | | s11 | | s12 | | 18 | 9 | 10 |
| 12 | r6 | r6 | | r6 | | | | | |
| 13 | r1 | s7 | | | | r1 | | | |
| 14 | r3 | r3 | | | | r3 | | | |
| 15 | r5 | r5 | | | | r5 | | | |
| 16 | | | s11 | | s12 | | | 19 | 10 |
| 17 | | | s11 | | s12 | | | | 20 |
| 18 | s16 | | | s21 | | | | | |
| 19 | r1 | s19 | | r1 | | | | | |
| 20 | r3 | r3 | | r3 | | | | | |
| 21 | r5 | r5 | | r5 | | | | | |

**Figure 3.7 The Action and Goto Tables for the Grammar (3.31)**
**(blank entry = error)**

Consider the steps for parsing **id+id*id**. Initially, the parser is in state 0, so a 0 is pushed onto the stack.

| Stack | Input | Action |
|---|---|---|
| 0 | **id+id*id#** | |

The next incoming symbol is an **id**, so we consult the Action table, Action[0,**id**] to determine what to do in state 0 with an incoming token, **id**. The entry is s5, so we shift the **id** onto the stack and go into state 5 (pushing the new state onto the stack above the **id**).

| Stack | Input | Action |
|---|---|---|
| 0 | **id+id*id#** | shift 5 |
| 0**id**5 | **+id*id#** | |

Now, the 5 on top of the stack indicates we are in state 5 and the incoming token is **+**, so we consult Action[5,**+**]; the r6 indicates a reduction using rule 6: F ::= **id**. To make the reduction, we pop 2*k items off the stack where k is the number of symbols in the rule's right hand side; in our example k=1 so we pop both the 5 and the **id**.

| Stack | Input | Action |
|---|---|---|
| 0 | **id+id*id#** | shift 5 |
| 0**id**5 | **+id*id#** | reduce 6, output a 6 |
| 0 | **+id*id#** | |

Because we are reducing the right hand side to an F in this example, we push the F onto the stack.

| Stack | Input | Action |
|---|---|---|
| 0 | **id+id*id#** | shift 5 |
| 0**id**5 | **+id*id#** | reduce 6, output a 6 |
| 0F | **+id*id#** | |

And finally, we consult Goto[0,F] to determine which state the parser, initially in state 0, should go into after parsing an F. Since Goto[0,F] = 3, this is state 3. We push the 3 onto the stack to indicate the parser's new state.

| Stack | Input | Action |
|---|---|---|
| 0 | **id+id*id#** | shift 5 |
| 0**id**5 | **+id*id#** | reduce 6, output a 6 |
| 0F3 | **+id*id#** | |

From state 3 and looking at the incoming token **+**, Action[3,**+**] tells us to reduce using rule 4: T ::= F.

| Stack | Input | Action |
|---|---|---|
| 0 | **id+id*id#** | shift 5 |
| 0**id**5 | **+id*id#** | reduce 6, output a 6 |
| 0F3 | **+id*id#** | reduce 4, output a 4 |
| 0T2 | **+id*id#** | |

From state 2 and looking at the incoming token **+**, Action[2,**+**] tells us to reduce using rule 2: E ::= T.

| Stack | Input | Action |
|---|---|---|
| 0 | **id+id*id#** | shift 5 |
| 0**id**5 | **+id*id#** | reduce 6, output a 6 |
| 0F3 | **+id*id#** | reduce 4, output a 4 |
| 0T2 | **+id*id#** | reduce 2, output a 2 |
| 0E1 | **+id*id#** | |

From state 1 and looking the incoming token **+**, Action[3,**+**] = s6 tells us to shift (the **+** onto the stack and go into state 6.

| Stack | Input | Action |
|---|---|---|
| 0 | **id+id*id#** | shift 5 |
| 0**id**5 | **+id*id#** | reduce 6, output a 6 |
| 0F3 | **+id*id#** | reduce 4, output a 4 |
| 0T2 | **+id*id#** | reduce 2, output a 2 |
| 0E1 | **+id*id#** | shift 6 |
| 0E1**+**6 | **id*id#** | |

Continuing in this fashion, the parser goes through the following sequence of configurations and actions:

| Stack | Input | Action |
|---|---|---|
| 0 | **id+id*id#** | shift 5 |
| 0**id**5 | **+id*id#** | reduce 6, output a 6 |
| 0F3 | **+id*id#** | reduce 4, output a 4 |
| 0T2 | **+id*id#** | reduce 2, output a 2 |
| 0E1 | **+id*id#** | shift 6 |
| 0E1**+**6 | **id*id#** | shift 5 |
| 0E1**+**6**id**5 | ***id#** | reduce 6, output 6 |
| 0E1**+**6F3 | ***id#** | reduce 4, output 4 |

| | | |
|---|---|---|
| 0E1**+**6T13 | ***id#** | shift 7 |
| 0E1**+**6T13*****7 | **id#** | shift 5 |
| 0E1**+**6T13*****7**id**5 | **#** | reduce 6, output 6 |
| 0E1**+**6T13*****7F14 | **#** | reduce 3, output 3 |
| 0E1**+**6T13 | **#** | reduce 1, output 1 |
| 0E1 | **#** | accept |

In the last step, the parser is in state 1 and the incoming token is the terminator **#**; Action[1,**#**] says we *accept* the input sentence; the sentence has been successfully parsed.

Moreover, the parser has output 6, 4, 2, 6, 4, 6, 3, 1, which is a rightmost derivation of the input string in reverse: 1, 3, 6, 4, 6, 2, 4, 6, that is

$$\underline{E} \Rightarrow E \texttt{ + } \underline{T} \Rightarrow E \texttt{ + } T \texttt{ * } \underline{F} \Rightarrow E \texttt{ + } \underline{T} \texttt{ * id} \Rightarrow E \texttt{ + } \underline{F} \texttt{ * id} \Rightarrow \underline{E} \texttt{ + id * id} \Rightarrow$$
$$\underline{T} \texttt{ + id * id} \Rightarrow \underline{F} \texttt{ + id * id} \Rightarrow \texttt{id + id * id}$$

A careful reading of the preceding steps suggests that explicitly pushing the symbols onto the stack is unnecessary because the symbols are implied by the states themselves. An industrial strength LR(1) parser will simply maintain a stack of states. We include the symbols only for illustrative purposes.

For all of this to work, we must go about constructing the tables, Action and Goto. To do this, we must first construct the grammar's LR(1) canonical collection.

### 3.4.2.2 The LR(1) Canonical Collection

The LR(1) parsing tables, Action and Goto, for a grammar G are derived from a DFA for recognizing the possible handles for a parse in G. This DFA is constructed from what is called an *LR(1) canonical collection*, in turn a collection of sets of *items* of the form,

*(3.32)* [Y ::= α•β, **a**]

where Y ::= αβ is a production rule in the set of productions P, α and β are (possibly empty) strings of symbols, and **a** is a *look-ahead*. The item represents a potential handle. The • is a position marker that marks the top of the stack, indicating that we have parsed the α and still have the β ahead of us in satisfying the Y. The look-ahead symbol, **a**, is a token that can follow Y (and so, αβ) in a legal rightmost derivation of some sentence.

- If the position marker comes at the start of the right hand side in an item,

  [Y ::= •αβ, **a**]

the item is called a *possibility*. One way of parsing the Y is to first parse the α and then parse the β, after which point the next incoming token will be an **a**. The parse might be in the following configuration

| Stack | Input |
|---|---|
| #γ | u**a**… |

where αβ =>* u, where u is a string of terminals

- If the position marker comes after a string of symbols α but before a string of symbols β in the right hand side in an item,

  [Y ::= α•β, **a**]

  the item indicates that α has been parsed (and so is on the stack) but that there is still β to parse from the input:

| Stack | Input |
|---|---|
| #γα | v**a**… |

  where β =>* v, where v is a string of terminals.

- If the position marker comes at the end of the right hand side in an item,

  [Y ::= αβ•, **a**]

  the item indicates that the parser has successfully parsed αβ in a context where Y**a** would be valid, the αβ can be reduced to a Y, and so αβ is a handle. That is, the parse is in the configuration

| Stack | Input |
|---|---|
| #γαβ | **a**… |

  and the reduction of αβ would cause the parser to go into the configuration,

| Stack | Input |
|---|---|
| #γY | **a**… |

A non-deterministic finite-state automaton (NFA) that recognizes viable prefixes and handles can be constructed from items like that in 3.32. The items record the progress in parsing various language fragments. We also know that, given this NFA, we can

construct an equivalent DFA using the subset construction that we saw in section 2.7. The states of the DFA for recognizing viable prefixes are derived from *sets* of these items, which record the current state of an LR(1) parser. Here, we shall construct these sets, and so construct the DFA, directly (instead of first constructing a NFA).

So, the states in our DFA will be constructed from sets of items like that in 3.32. We call the set of states the *canonical collection*.

To construct the canonical collection of states we first must augment our grammar G, with an additional start symbol S' and an additional rule,

S' ::= S

so as to yield a grammar G', which describes the same language as does G, but which does not have its start symbol on the right hand side of any rule. For example, augmenting our grammar (3.31) for simple expressions gives us the augmented grammar in (3.33).

*(3.33)*

0. E' ::= E
1. E ::= E **+** T
2. E ::= T
3. T ::= T ***** F
4. T ::= F
5. F ::= **(**E)
6. F ::= **id**

We then start constructing item sets from this augmented grammar. The first set, representing the initial state in our DFA, will contain the LR(1) item,

*(3.34)* {[E' ::= • E**, #**]}

which says that parsing an E' means parsing an E from the input, after which point the next (and last) remaining un-scanned token should be the terminator **#**. But at this point, we have not yet parsed the E; the • in front of it indicates that it is still ahead of us.

Now, that we must parse an E at this point, means that we might be parsing either an E + T (by rule 1 of 3.31) or a T (by rule 2). So the initial set would also contain,

*(3.35)*

[E ::= • E **+** T, **#**]
[E ::= • T, **#**]

In fact, the initial set will contain additional items implied by (3.34). We call the initial set (3.34) the *kernel*. From the kernel, we can then compute the *closure*, that is all items implied by the kernel. Algorithm 3.7 computes the closure for any set of items.

***Algorithm 3.7: Computing the closure of a set of items.***
Input: a set of items, s.
Output: closure(s).

1. Add s to closure(s).
2. Repeat until no new items may be added:
   If closure(s) contains an item of the form,
   [Y ::= α•Xβ, **a**]
   add the item
   [X ::= • γ, **b**]
   for every rule X ::= γ in P and for every token **b** in first(β**a**).

***Example***

To compute the closure of our kernel (3.34), that is closure({[E' ::= • E, #]}), by step 1 is initially

*(3.36)* {[E' ::= • E, #]}

We then invoke step 2. Because the • comes before the E, and because we have the rule E ::= E + T and E ::= T, we add [E ::= • E + T, #] and [E ::= • T, #] to get

*(3.37)*
{[E' ::= • E, #]
[E ::= • E + T, #]
[E ::= • T, #]}

The item [E ::= • E + T, #] implies

[E ::= • E + T, +]
[E ::= • T, +]

because first (+ T #) = {+}. Now, given that these items differ from previous items only in the lookaheads, we can use the more compact notation

[E ::= • E + T, #/+]

for representing the two items

[E ::= • E + T, #] and
[E ::= • E + T, +].

So we get

(3.38)
{[E' ::= • E, #],
 [E ::= • E + T, #/+],
 [E ::= • T, #/+]}

The items [E ::= • T, #/+] imply additional items (by similar logic), leading to

(3.39)
{[E' ::= • E, #],
 [E ::= • E + T, #/+],
 [E ::= • T, #/+],
 [T ::= • T * F, #/+/*],
 [T ::= • F, #/+/*]}

And finally the items [T ::= • F, #/+/*] imply additional items (by similar logic), leading to

(3.40)
$s_0$ = {[E' ::= • E, #],
 [E ::= • E + T, #/+],
 [E ::= • T, #/+],
 [T ::= • T * F, #/+/*],
 [T ::= • F, #/+/*],
 [F ::= • (E), #/+/*],
 [F ::= • id, #/+/*]}

The item set (3.40) represents the initial state $s_0$ in our canonical LR(1) collection.

As an aside, notice that the closure of {[E' ::= • E, #]} represents all of the states in an NFA, which could be reached from the initial item by ε-moves alone, that is without scanning anything from the input. That portion of the NFA that is equivalent to the initial state $s_0$ in our DFA is illustrated in Figure 3.8.

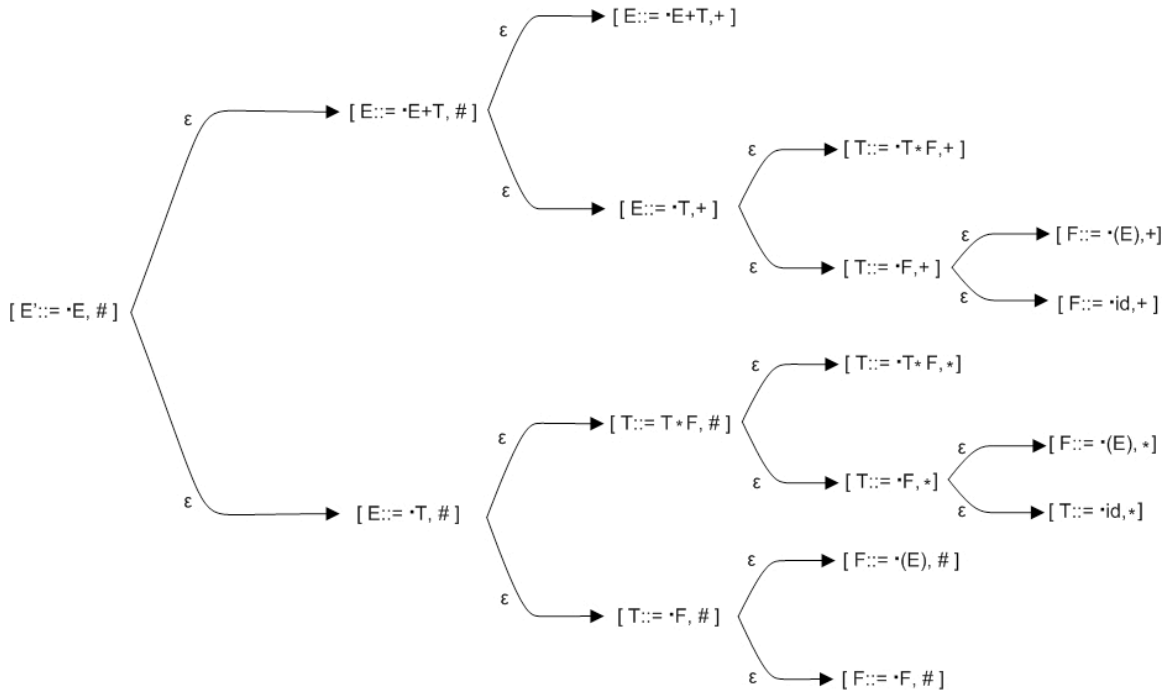



**Figure 3.8 The NFA Corresponding to $s_0$**

We now need to compute all of the states and transitions of the DFA that recognizes viable prefixes and handles. For any item set s, and any symbol $X \in (T \cup N)$, goto(s,X) = closure(r), where r = {[Y ::= $\alpha$X•$\beta$, a] | [Y ::= $\alpha$•X$\beta$, a]}[5]. That is, to compute goto(s,X) , we take all items from s with a • before the X and move it after the X; we then take the closure of that. Algorithm 3.8 does this.

***Algorithm 3.8: Computing goto.***
Input: a state s, and a symbol $X \in (T \cup N)$
Output: The state, goto(s,X).

1. r = {}
2. For each item [Y ::= $\alpha$•X$\beta$, a] in s, add [Y ::= $\alpha$X•$\beta$, a] to r.
3. Return closure(r).

***Example***

For example, consider the computation of goto($s_0$, E), where $s_0$ is in (3.38). The two relevant items in $s_0$ are [E' ::= • E , **#**] and [E ::= • E **+** T, **#**/**+**]. Moving the • to the

[5] The | operator is being used as the set notation "for all"; *not* the BNF notation for alternation.

right of the E in the two items gives us {[E' ::= E • , #], [E ::= E • + T, #/+]}. The closure of this set is the set itself; let's call this state $s_1$.

goto($s_0$, E) = $s_1$ = {[E' ::= E • , #],
[E ::= E • + T, #/+]}

In a similar manner, we can compute,

goto($s_0$, T) = $s_2$ = {[E ::= T •, #/+],
[T ::= T • * F, #/+/*]}

goto($s_0$, F) = $s_3$ = {[T ::= F •, #/+/*]}

goto($s_0$, **(**) involves a closure since moving the • across the **(** puts it in front of the E.

goto($s_0$, **(**) = $s_4$ = {[F ::= **(** • E), #/+/*],
[E ::= • E + T,)/+],
[E ::= • T,)/+],
[T ::= • T * F,)/+/*],
[T ::= • F,)/+/*],
[F ::= • **(**E),)/+/*],
[F ::= • **id**,)/+/*]}

goto($s_0$, **id**) = $s_5$ = {[F ::= **id** •, #/+/*]}

We continue in this manner, computing goto for the states we have, and then for any new states repeatedly until we have defined no more new states. This gives us the canonical LR(1) collection.

***Algorithm 3.9: Computing the LR(1) Collection.***
Input: a context-free grammar G.
Output: The canonical LR(1) collection of states c = {$s_0$, $s_1$,…, $s_n$}

1. Define an augmented grammar G' which is G with the added non-terminal S' and added production rule S' ::= S, where S is G's start symbol. The following steps apply to G'. Enumerate the production rules beginning at 0 for the newly added production.
2. c = {$s_0$}, where $s_0$ = closure({[S' ::= • S, #]}).
3. Repeat until no new states are added to c,
   For each s in c, and for each symbol X ∈ T ∪ N,
   If goto(s,X) ≠ ∅ and goto(s,X) ∉ c, add goto(s,X) to c.

***Example***

3-54

We can now resume computing the LR(1) canonical collection for the simple expression grammar, beginning from state $s_1$.

goto($s_1$,**+**) = $s_6$ = {[E ::= E **+** • T, **#**/**+**],
[T ::= • T ***** F, **#**/**+**/*****],
[T ::= • F, **#**/**+**/*****],
[F ::= • **(**E), **#**/**+**/*****],
[F ::= • **id**, **#**/**+**/*****]}

There are no more moves from $s_1$. Similarly, from $s_2$,

goto($s_2$,*****) = $s_7$ = {[T ::= T ***** • F, **#**/**+**/*****]
[F ::= • **(**E), **#**/**+**/*****],
[F ::= • **id**, **#**/**+**/*****]}

Notice that the closure of {[T ::= T ***** • F, **#**/**+**/*****]} carries along the same lookaheads since no symbol follows the F in the right hand side. There are no gotos from $s_3$, but several from $s_4$.

goto($s_4$,E) = $s_8$ = {[F ::= **(**E •**)**, **#**/**+**/*****],
[E ::= E • **+** T,)/**+**]}

goto($s_4$,T) = $s_9$ = {[E ::= T •,)/**+**],
[T ::= T • ***** F,)/**+**/*****]}

goto($s_4$,F) = $s_{10}$ = {[T ::= F •,)/**+**/*****]}

goto($s_4$, **(**) = $s_{11}$ = {[F ::= **(**• E),)/**+**/*****],
[E ::= • E **+** T,)/**+**],
[E ::= • T,)/**+**],
[T ::= • T ***** F,)/**+**/*****],
[T ::= • F,)/**+**/*****],
[F ::= • **(**E),)/**+**/*****],
[F ::= • **id**,)/**+**/*****]}

Notice that $s_{11}$ differs from $s_4$ in only the lookaheads for the first item.

goto($s_4$,**id**) = $s_{12}$ = {[F ::= **id** •,)/**+**/*****]}

There are no moves from $s_5$, so consider $s_6$.

goto($s_6$,T) = $s_{13}$ = {[E ::= E **+**T •, **#**/**+**],
[T ::= T • ***** F, **#**/**+**/*****]}

Now, goto($s_6$,F) = {[T ::= F •, **#**/**+**/*****]} but that is $s_3$. goto($s_6$, **(**) is closure({[F ::= **(**• E), **#**/**+**/*****]}), but that is $s_4$. And goto($s_6$,**id**) is $s_5$.

goto($s_6$,F) = $s_3$
goto($s_6$, **(**) = $s_4$
goto($s_6$,**id**) = $s_5$

Consider $s_7$, $s_8$, and $s_9$.

goto($s_7$,F) = $s_{14}$ = {[T ::= T ***** F •, **#**/**+**/*****]}
goto($s_7$, **(**) = $s_4$
goto($s_7$,**id**) = $s_5$
goto($s_8$,**)**) = $s_{15}$ = {[F ::= **(**E) •, **#**/**+**/*****]}

goto($s_8$,**+**) = $s_{16}$ = {[E ::= E **+** • T,)/**+**],
[T ::= • T ***** F,)/**+**/*****],
[T ::= • F,)/**+**/*****],
[F ::= • **(**E), )/**+**/*****],
[F ::= • **id**,)/**+**/*****]}

goto($s_9$,*****) = $s_{17}$ = {[T ::= T ***** • F,)/**+**/*****]
[F ::= • **(**E), )/**+**/*****],
[F ::= • **id**,)/**+**/*****]}

There are no moves from $s_{10}$, but several from $s_{11}$.

goto($s_{11}$,E) = $s_{18}$ = {[F ::= **(**E •**)** , )/**+**/*****],
[E ::= E • **+** T,)/**+**]}

goto($s_{11}$,T) = $s_9$
goto($s_{11}$,F) = $s_{10}$
goto($s_{11}$, **(**) = $s_{11}$
goto($s_{11}$,**id**) = $s_{12}$

There are no moves from $s_{12}$, but there is a move from $s_{13}$.

goto($s_{13}$,*****) = $s_7$

There are no moves from $s_{14}$ or $s_{15}$, but there are moves from $s_{16}$, $s_{17}$, $s_{18}$, and $s_{19}$.

goto($s_{16}$,T) = $s_{19}$ = {[E ::= E **+**T •,)/**+**],
[T ::= T • ***** F,)/**+**/*****]}
goto($s_{16}$,F) = $s_{10}$
goto($s_{16}$, **(**) = $s_{11}$
goto($s_{16}$, **id**) = $s_{12}$
goto($s_{17}$, F) = $s_{20}$ = {[T ::= T ***** F •,)/**+**/*****]}
goto($s_{17}$, **(**) = $s_{11}$
goto($s_{17}$, **id**) = $s_{12}$
goto($s_{18}$,**)** ) = $s_{21}$ = {[F ::= **(**E) •, )/**+**/*****]}
goto($s_{18}$,**+**) = $s_{16}$
goto($s_{19}$,*****) = $s_{17}$

There are no moves from $s_{20}$ or $s_{21}$, so we are done. The LR(1) canonical collection consists of twenty-two states $s_0$… $s_{21}$. The entire collection is summarized in Figure 3.9.

$s_0$ = {[E' ::= • E**, #**], goto($s_0$, E) = $s_1$
[E ::= • E **+** T, **#**/**+**],
[E ::= • T, **#**/**+**], goto($s_0$, T) = $s_2$
[T ::= • T ***** F, **#**/**+**/*****],
[T ::= • F, **#**/**+**/*****], goto($s_0$, F) = $s_3$
[F ::= • **(**E), **#**/**+**/*****], goto($s_0$, **(**) = $s_4$
[F ::= • **id**, **#**/**+**/*****]} goto($s_0$, **id**) = $s_5$

$s_1$ = {[E' ::= E •**, #**],
[E ::= E **•** **+** T, **#**/**+**]} goto($s_1$,**+**) = $s_6$

$s_2$ = {[E ::= T •, **#**/**+**],
[T ::= T • ***** F, **#**/**+**/*****]} goto($s_2$,*****) = $s_7$

$s_3$ = {[T ::= F •, **#**/**+**/*****]}

$s_4$ = {[F ::= **(**• E), **#**/**+**/*****], goto($s_4$,E) = $s_8$
[E ::= • E **+** T,)/**+**],
[E ::= • T,)/**+**], goto($s_4$,T) = $s_9$
[T ::= • T ***** F,)/**+**/*****],
[T ::= • F,)/**+**/*****], goto($s_4$,F) = $s_{10}$
[F ::= • **(**E),)/**+**/*****], goto($s_4$, **(**) = $s_{11}$
[F ::= • **id**,)/**+**/*****]} goto($s_4$,**id**) = $s_{12}$

$s_5$ = {[F ::= **id** •, **#**/**+**/*****]}

$s_6$ = {[E ::= E **+** • T, **#**/**+**], goto($s_6$,T) = $s_{13}$
[T ::= • T ***** F, **#**/**+**/*****],
[T ::= • F, **#**/**+**/*****], goto($s_6$,F) = $s_3$
[F ::= • **(**E), **#**/**+**/*****], goto(($s_6$, **(**) = $s_4$
[F ::= • **id**, **#**/**+**/*****]} goto(($s_6$,**id**) = $s_5$

$s_7$ = {[T ::= T ***** • F, **#**/**+**/*****] goto($s_7$,F) = $s_{14}$
[F ::= • **(**E), **#**/**+**/*****], goto($s_7$, **(**) = $s_4$
[F ::= • **id**, **#**/**+**/*****]} goto($s_7$,**id**) = $s_5$

$s_8$ = {[F ::= **(**E •), **#**/**+**/*****], goto($s_8$,**)** ) = $s_{15}$
[E ::= E • **+** T,)/**+**]} goto($s_8$,**+**) = $s_{16}$
$s_9$ = {[E ::= T •,)/**+**],
[T ::= T • ***** F,)/**+**/*****]} goto($s_9$,*****) = $s_{17}$

$s_{10}$ = {[T ::= F •,)/**+**/*****]}

$s_{11}$ = {[F ::= **(**• E),)/**+**/*****], goto($s_{11}$,E) = $s_{18}$
[E ::= • E **+** T,)/**+**],
[E ::= • T,)/**+**],goto($s_{11}$,T) = $s_9$
[T ::= • T ***** F,)/**+**/*****],
[T ::= • F,)/**+**/*****], goto($s_{11}$,F) = $s_{10}$
[F ::= • **(**E),)/**+**/*****], goto($s_{11}$, **(**) = $s_{11}$
[F ::= • **id**,)/**+**/*****]} goto($s_{11}$,**id**) = $s_{12}$

$s_{12}$ = {[F ::= **id** •,)/**+**/*****]}

$s_{13}$ = {[E ::= E **+**T •, **#**/**+**],
[T ::= T • ***** F, **#**/**+**/*****]} goto($s_{13}$,*****) = $s_7$

$s_{14}$ = {[T ::= T ***** F •, **#**/**+**/*****]}

$s_{15}$ = {[F ::= **(**E) •, **#**/**+**/*****]}

$s_{16}$ = {[E ::= E **+** • T,)/**+**], goto($s_{16}$,T) = $s_{19}$
[T ::= • T ***** F,)/**+**/*****],
[T ::= • F,)/**+**/*****], goto($s_{16}$,F) = $s_{10}$
[F ::= • **(**E), )/**+**/*****], goto($s_{16}$, **(**) = $s_{11}$
[F ::= • **id**,)/**+**/*****]} goto($s_{16}$,**id**) = $s_{12}$

$s_{17}$ = {[T ::= T ***** • F,)/**+**/*****] goto($s_{17}$,F) = $s_{20}$

F ::= • **(**E), )/**+**/*****], goto($s_{17}$, **(**) = $s_{11}$
[F ::= • **id**,)/**+**/*****]} goto($s_{17}$,**id**) = $s_{12}$

$s_{18}$ = {[F ::= **(**E •**)** , )/**+**/*****], goto($s_{18}$,**)** ) = $s_{21}$
[E ::= E • **+** T,)/**+**]} goto($s_{18}$,**+**) = $s_{16}$

$s_{19}$ = {[E ::= E **+**T •,)/**+**],
[T ::= T • ***** F,)/**+**/*****]} goto($s_{19}$,*****) = $s_{17}$

$s_{20}$ = {[T ::= T ***** F •,)/**+**/*****]}

$s_{21}$ = {[F ::= **(**E) •, )/**+**/*****]}

**Figure 3.9 The LR(1) Canonical Collection for the Grammar (3.31)**

We may now go about constructing the tables Action and Goto.

## 3.4.2.3 Constructing the LR(1) Parsing Tables

The LR(1) parsing tables Action and Goto are constructed from the LR(1) canonical collection, as prescribed in Algorithm 3.10.

***Algorithm 3.10: Constructing the LR(1) Parsing Tables for a Context-Free Grammar.***
Input: a context-free grammar G = (N,T,S,P).
Output: The LR(1) tables Action and Goto.

1. Compute the LR(1) canonical collection c = {$s_0, s_1, \ldots, s_n$}. State i of the parser corresponds to the item set $s_i$. State 0, corresponding to the item set $s_0$, which contains the item [S' ::= • S, **#**], is the parser's initial state.
2. The Action table is constructed as follows.
   a. For each transition, goto($s_i$, **a**) = $s_j$, where **a** is a terminal, set Action[i,**a**] = s j. The s stands for "shift".
   b. If the item set $s_k$ contains the item [S' ::= S •, **#**], set Action[k,**#**] = accept.
   c. For all item sets $s_i$, if $s_i$ contains an item of the form [Y ::= $\alpha$ •, **a**], set Action[i,**a**] = r p, where p is the number corresponding to the rule Y ::= $\alpha$. The r stands for "reduce".
   d. All undefined entries in Action are set to *error*.
3. The Goto table is constructed as follows.
   a. For each transition, goto($s_i$, Y) = $s_j$, where Y is a non-terminal, set Goto[i,Y] = j.
   b. All undefined entries in Goto are set to *error*.

If all entries in the Action table are unique then the grammar G is said to be LR(1).

***Example***

Let us say we are computing the Action and Goto tables for the arithmetic expression grammar in (3.31). We apply Algorithm 3.9 for computing the LR(1) canonical collection. This produces the twenty-two item sets shown in Figure 3.9. Adding the

extra production rule and enumerating the production rules gives us the augmented grammar in (3.41).

*(3.41)*

0. E' ::= E
1. E ::= E **+** T
2. E ::= T
3. T ::= T ***** F
4. T ::= F
5. F ::= **(**E)
6. F ::= **id**

We must now apply steps 2 and 3 of Algorithm 3.10 for constructing the tables Action and Goto.

Both tables will each have twenty-two rows for the twenty-two states, derived from the twenty-two item sets in the LR(1) canonical collection: 0 to 21. The Action table will have six columns, one for each terminal symbol: **+**, *****, **(**,), **id**, and the terminator **#**. The Goto table will have three columns, one for each of the original non-terminal symbols: E, T and F. The newly added non-terminal E' does not play a role in the parsing process. The tables are illustrated in Figure 3.7. To see how these tables are constructed, let us derive the entries for several states.

Firstly, let us consider the first four states of the Action table.

- The row of entries for state 0 is derived from item set $s_0$.
  - By step 2a of Algorithm 3.10, the transition goto($s_0$, **(**) = $s_4$ implies Action[0,(] = s4, and goto($s_0$, **id**) = $s_5$ implies Action[0,id] = s5. The s4 means "shift the next input symbol **(**onto the stack and go into state 4"; the s5 means "shift the next input symbol **id** onto the stack and go into state 5."
- The row of entries for state 1 is derived from item set $s_1$.
  - By step 2a, the transition goto($s_1$,**+**) = $s_6$ implies Action[1,**+**] = s6. Remember, the s6 means "shift the next input symbol **+** onto the stack and go into state 6".
  - By step 2b, because item set $s_1$ contains [E' ::= E **•, #**], Action[1,**#**]=accept. This says, that if the parser is in state 1 and the next input symbol is the terminator **#**, the parser *accepts* the input string as being in the language.
- The row of entries for state 2 is derived from item set $s_2$.
  - By step 2a, the transition goto($s_2$,*****) = $s_7$ implies Action[2,*] = s7.
  - By step 2c, the items[6] [E ::= T **•**, **#**/**+**] imply two entries: Action[2, **#**] = r2 and Action[2, **+**] = r2. These entries say, that if the parser is in state 2 and the

[6] Recall that the [E ::= T **•**, **#**/**+**] denotes *two* items: [E ::= T **•**, **#**] and [E ::= T **•**, **+**].

next incoming symbol is either a **#** or a **+**, reduce the T on the stack to a E using production rule 2: E ::= T.

- The row of entries for state 3 is derived from item set $s_3$.
    - By step 2c, the items [T ::= F •, **#**/**+**/*****]} imply three entries: Action[3, **#**]= r4, Action[3, **+**] = r4 and Action[3, *****] = r4. These entries say, that if the parser is in state 3 and the next incoming symbol is either a **#**, a **+** or a *****, reduce the F on the stack to a T using production rule 4: T ::= F.

All other entries in rows 0, 1 2 and 3 are left blank to indicate an error. If for example, the parser is in state 0 and the next incoming symbol is a **+**, the parser raises an error. The derivations of the entries in rows 4 to 21 in the Action table (see Figure 3.7) are left as an exercise.

Now let us consider the first four states of the Goto table.

- The row of entries for state 0 is derived from item set $s_0$.
    - By step 3a of Algorithm 3.10, the transition goto($s_0$, E) = 1 implies Goto[0,E]=1, goto($s_0$, T) = 2 implies Goto[0,E]=4, and goto($s_0$, F) = 3 implies Goto[0,E]=3. The entry Goto[0,E]=1 says that in state 0, once the parser scans and parses an E, the parser goes into state 1.
- The row of entries for state 1 is derived from item set $s_1$. Because there are no transitions on a non-terminal from item set $s_1$, no entries are indicated for state 1 in the Goto table.
- The row of entries for state 2 is derived from item set $s_2$. Because there are no transitions on a non-terminal from item set $s_2$, no entries are indicated for state 2 in the Goto table.
- The row of entries for state 3 is derived from item set $s_3$. Because there are no transitions on a non-terminal from item set $s_3$, no entries are indicated for state 3 in the Goto table.

All other entries in rows 0, 1 2 and 3 are left blank to indicate an error. The derivations of the entries in rows 4 to 21 in the Goto table (see Figure 3.7) are left as an exercise.

### 3.4.2.4 Conflicts in the Action Table

There are two different kinds of conflicts possible for an entry in the Action table:

1. The first is the shift-reduce conflict, which can occur when there are items of the forms,

   [Y ::= $\alpha$•, **a**] and
   [Y ::= $\alpha$•**a** $\beta$, **b**]

The first item suggests a reduce if the next un-scanned token is an **a**; the second suggests a shift of the **a** onto the stack.

Although such conflicts may occur for unambiguous grammars, a common cause are ambiguous constructs such as,

S ::= **if (**E) S
S ::= **if (**E) S **else** S

As we saw in section 3.2.3, language designers will not give up such ambiguous constructs for the sake of parser writers. Most parser generators that are based on LR grammars permit one to supply an extra disambiguating rule. For example, the rule in this case would be to favor a shift of the **else** over a reduce of the "**if (**E) S" to an S.

2. The second kind of conflict that we can have is the reduce-reduce conflict. This can happen when we have a state containing two items of the form,

   [X ::= α•, **a**] and
   [Y ::= β•, **a**]

   Here, the parser cannot distinguish which production rule to apply in the reduction.

Of course, we will never have a shift-shift conflict, because of the definition of goto for terminals. Usually, a certain amount of tinkering with the grammar is sufficient for removing bona fide conflicts in the Action table for most programming languages.

### 3.4.3 LALR(1) Parsing

### 3.4.3.1 Merging LR(1) States

A LR(1) parsing table for a typical programming language such as Java can have thousands of states, and so thousands of rows. One could argue that, given the inexpensive memory nowadays, this is not a problem. On the other hand, smaller programs and data make for faster running programs so it would be advantageous if we might be able to reduce the number of states. LALR(1) is a parsing method that does just this.

If you look at the LR(1) canonical collection of states in Figure 3.8, which we computed for our example grammar (3.31), you will find that many states are virtually identical – they differ only in their look-ahead tokens. Their cores – the *core* of an item is just the rule and position marker portion – are identical. For example, consider states $s_2$ and $s_9$.

$s_2$ = {[E ::= T •, **#**/**+**], [T ::= T • * F, **#**/**+**/*]}

$s_9$ = {[E ::= T •,)/**+**], [T ::= T • * F,)/**+**/*]}

They differ only in the lookaheads **#** and). Their cores are the same:

$s_2$ = {[E ::= T •], [T ::= T • * F]}

$s_9$ = {[E ::= T •], [T ::= T • * F]}

What happens if we merge them, taking a union of the items, into a single state, $s_{2.9}$ ? Because the cores are identical, taking a union of the items merges the lookaheads.

$s_{2\,9}$ = {[E ::= T •, **#**/**)**/**+**], [T ::= T • * F, **#**/**)** /**+**/*]}

Will this cause the parser to carry out actions that it is not supposed to? Notice that the lookaheads play a role only in reductions and never in shifts. It is true that the new state may call for a reduction that was not called for by the original LR(1) states; yet an error will be detected before any progress is made in scanning the input.

Similarly, looking at the states in Figure 3.8, one can merge states $s_3$ and $s_{10}$, $s_2$ and $s_9$, $s_4$ and $s_{11}$, $s_5$ and $s_{12}$, $s_6$ and $s_{16}$, $s_2$ and $s_{17}$, $s_8$ and $s_{18}$, $s_{13}$ and $s_{19}$, $s_{14}$ and $s_{20}$, and states $s_{15}$ and $s_{21}$. This allows us to reduce the number of states by ten. In general, for bona fide programming languages, one can reduce the number of states by an order of magnitude.

## 3.4.3.2 LALR(1) Table Construction

There are two approaches to computing the LALR(1) states, and so the LALR(1) parsing tables.

### *LALR(1) table construction from the LR(1) states*

In the first approach, we first compute the full LR(1) canonical collection of states, and then perform the state merging operation illustrated above for producing what we will call the LALR(1) canonical collection of states.

***Algorithm 3.11: Constructing the LALR(1) Parsing Tables for a Context-Free Grammar.***
Input: a context-free grammar G = (N,T,S,P).
Output: The LALR(1) tables Action and Goto.

1. Compute the LR(1) canonical collection c = $\{s_0, s_1, \ldots, s_n\}$.

2. Merge those states whose item cores are identical. The items in the merged state are a union of the items from the states being merged. This produces a LALR(1) canonical collection of states.
3. The goto function for each new merged state is the union of the goto for the individual merged states.
4. The entries in the Action and Goto tables are constructed from the LALR(1) states in the same way as for the LR(1) parser in Algorithm 3.10.

If all entries in the Action table are unique then the grammar G is said to be LALR(1). For example, reconsider our grammar for simple expressions from (3.31), and (again) repeated here as (3.42).

*(3.42)*

0. E' ::= E
1. E ::= E **+** T
2. E ::= T
3. T ::= T ***** F
4. T ::= F
5. F ::= **(**E)
6. F ::= **id**

Step 1 of Algorithm 3.11 has us compute the LR(1) canonical collection that was shown in Figure 3.9, and is repeated here in Figure 3.10.

$s_0$ = {[E' ::= • E, #], goto($s_0$, E) = $s_1$
[E ::= • E + T, #/+],
[E ::= • T, #/+], goto($s_0$, T) = $s_2$
[T ::= • T * F, #/+/*],
[T ::= • F, #/+/*], goto($s_0$, F) = $s_3$
[F ::= • (E), #/+/*], goto($s_0$, ( ) = $s_4$
[F ::= • id, #/+/*]} goto($s_0$, id) = $s_5$

$s_1$ = {[E' ::= E •, #],
[E ::= E • + T, #/+]} goto($s_1$,+) = $s_6$

$s_2$ = {[E ::= T •, #/+],
[T ::= T • * F, #/+/*]} goto($s_2$,*) = $s_7$

$s_3$ = {[T ::= F •, #/+/*]}

$s_4$ = {[F ::= ( • E), #/+/*], goto($s_4$,E) = $s_8$
[E ::= • E + T,)/+],
[E ::= • T,)/+], goto($s_4$,T) = $s_9$
[T ::= • T * F,)/+/*],
[T ::= • F,)/+/*], goto($s_4$,F) = $s_{10}$
[F ::= • (E),)/+/*], goto($s_4$, ( ) = $s_{11}$
[F ::= • id,)/+/*]} goto($s_4$,id) = $s_{12}$

$s_5$ = {[F ::= id •, #/+/*]}

$s_6$ = {[E ::= E + • T, #/+], goto($s_6$,T) = $s_{13}$
[T ::= • T * F, #/+/*],
[T ::= • F, #/+/*], goto($s_6$,F) = $s_3$
[F ::= • (E), #/+/*], goto(($s_6$, ( ) = $s_4$
[F ::= • id, #/+/*]} goto(($s_6$,id) = $s_5$

$s_7$ = {[T ::= T * • F, #/+/*] goto($s_7$,F) = $s_{14}$
[F ::= • (E), #/+/*], goto($s_7$, ( ) = $s_4$
[F ::= • id, #/+/*]} goto($s_7$,id) = $s_5$

$s_8$ = {[F ::= ( E •), #/+/*], goto($s_8$,) ) = $s_{15}$
[E ::= E • + T,)/+]} goto($s_8$,+) = $s_{16}$

$s_9$ = {[E ::= T •,)/+],
[T ::= T • * F,)/+/*]} goto($s_9$,*) = $s_{17}$

$s_{10}$ = {[T ::= F •,)/+/*]}

$s_{11}$ = {[F ::= ( • E),)/+/*], goto($s_{11}$,E) = $s_{18}$
[E ::= • E + T,)/+],
[E ::= • T,)/+], goto($s_{11}$,T) = $s_9$
[T ::= • T * F,)/+/*],
[T ::= • F,)/+/*], goto($s_{11}$,F) = $s_{10}$
[F ::= • (E),)/+/*], goto($s_{11}$, ( ) = $s_{11}$
[F ::= • id,)/+/*]} goto($s_{11}$,id) = $s_{12}$

$s_{12}$ = {[F ::= id •,)/+/*]}

$s_{13}$ = {[E ::= E +T •, #/+],

[T ::= T • * F, #/+/*]} goto($s_{13}$,*) = $s_7$

$s_{14}$ = {[T ::= T * F •, #/+/*]}

$s_{15}$ = {[F ::= (E) •, #/+/*]}

$s_{16}$ = {[E ::= E + • T,)/+], goto($s_{16}$,T) = $s_{19}$
[T ::= • T * F,)/+/*],
[T ::= • F,)/+/*], goto($s_{16}$,F) = $s_{10}$
[F ::= • (E), )/+/*], goto($s_{16}$, () = $s_{11}$
[F ::= • id,)/+/*]} goto($s_{16}$,id) = $s_{12}$

$s_{17}$ = {[T ::= T * • F,)/+/*] goto($s_{17}$,F) = $s_{20}$

F ::= • (E), )/+/*], goto($s_{17}$, () = $s_{11}$
[F ::= • id,)/+/*]} goto($s_{17}$,id) = $s_{12}$

$s_{18}$ = {[F ::= (E •) , )/+/*], goto($s_{18}$,) ) = $s_{21}$
[E ::= E • + T,)/+]} goto($s_{18}$,+) = $s_{16}$

$s_{19}$ = {[E ::= E +T •,)/+],
[T ::= T • * F,)/+/*]} goto($s_{19}$,*) = $s_{17}$

$s_{20}$ = {[T ::= T * F •,)/+/*]}

$s_{21}$ = {[F ::= (E) •, )/+/*]}

**Figure 3.10 The LR(1) Canonical Collection for the Grammar (3.42)**

Merging the states and re-computing the goto's gives us the LALR(1) canonical collection illustrated in Figure 3.11

$s_0$ = {[E' ::= • E, #], goto($s_0$, E) = $s_1$
[E ::= • E + T, #/+],
[E ::= • T, #/+], goto($s_0$, T) = $s_{29}$
[T ::= • T * F, #/+/*],
[T ::= • F, #/+/*], goto($s_0$, F) = $s_{3.10}$
[F ::= • (E), #/+/*], goto($s_0$, () = $s_{4.11}$
[F ::= • id, #/+/*]} goto($s_0$, id) = $s_{5.12}$

$s_1$ = {[E' ::= E •, #],
[E ::= E • + T, #/+]} goto($s_1$,+) = $s_{6.16}$

$s_{2.9}$ = {[E ::= T •, #/) /+],
[T ::= T • * F, #/) /+/*]} goto($s_{2.9}$,*) = $s_{7.17}$

$s_{3.10}$ = {[T ::= F •, #/) /+/*]}

$s_{4.11}$ = {[F ::= (• E), #/) /+/*], goto($s_{4.11}$,E) = $s_{8.18}$
[E ::= • E + T,)/+],
[E ::= • T,)/+], goto($s_{4.11}$,T) = $s_{2.9}$
[T ::= • T * F,)/+/*],
[T ::= • F,)/+/*], goto($s_{4.11}$,F) = $s_{3.10}$
[F ::= • (E),)/+/*], goto($s_{4.11}$, () = $s_{4.11}$
[F ::= • id,)/+/*]} goto($s_{4.11}$,id) = $s_{5.12}$

$s_{5.12}$ = {[F ::= id •, #/) /+/*]}

$s_{6.16}$ = {[E ::= E + • T, #/) /+], goto($s_{6.16}$,T) = $s_{13.19}$
[T ::= • T * F, #/) /+/*],
[T ::= • F, #/) /+/*], goto($s_{6.16}$,F) = $s_{3.10}$
[F ::= • (E), #/) /+/*], goto(($s_{6.16}$, () = $s_{4.11}$
[F ::= • id, #/) /+/*]} goto(($s_{6.16}$,id) = $s_{5.12}$

$s_{7.17}$ = {[T ::= T * • F, #/) /+/*] goto($s_{7.17}$,F) = $s_{14.20}$
[F ::= • (E), #/) /+/*], goto($s_{7.17}$, () = $s_{4.11}$
[F ::= • id, #/) /+/*]} goto($s_{7.17}$,id) = $s_{5.12}$

$s_{8.18}$ = {[F ::= (E •), #/) /+/*], goto($s_{8.18}$,) ) = $s_{15.21}$
[E ::= E • + T,)/+]} goto($s_{8.18}$,+) = $s_{6.16}$

$s_{13.19}$ = {[E ::= E +T •, #/) /+],
[T ::= T • * F, #/) /+/*]} goto($s_{13.19}$,*) = $s_{7.17}$

$s_{14.20}$ = {[T ::= T * F •, #/) /+/*]}

$s_{15.21}$ = {[F ::= (E) •, #/) /+/*]}

**Figure 3.11 The LALR(1) Canonical Collection for the Grammar (3.38)**

The LALR(1) parsing tables are given in figure 3.12.

| | Action | | | | | | Goto | | |
|---|---|---|---|---|---|---|---|---|---|
| | + | * | ( | ) | id | # | E | T | F |
| 0 | | | s4 | | s5 | | 1 | 2 | 3 |
| 1 | s6 | | | | | accept | | | |
| 2.9 | r2 | s7.17 | | r2 | | r2 | | | |
| 3.10 | r4 | r4 | | r4 | | r4 | | | |
| 4.11 | | | s11 | | s12 | | 8.18 | 2.9 | 3.10 |
| 5.12 | r6 | r6 | | r6 | | r6 | | | |
| 6.16 | | | s4 | | s5 | | | 13.19 | 3.10 |
| 7.17 | | | s4 | | s5 | | | | 14.20 |
| 8.18 | s16 | | | s15 | | | | | |
| 13.19 | r1 | s7 | | | | r1 | | | |
| 14.20 | r3 | r3 | | | | r3 | | | |
| 15.21 | r5 | r5 | | | | r5 | | | |

**Figure 3.12 The LALR(1) Parsing Tables for the Grammar (3.42)**

Of course, this approach of first generating the LR(1) canonical collection of states and then merging states to produce the LALR(1) collection consumes a great deal of space. But once the tables are constructed, they are small and workable. An alternative approach, which does not consume so much space, is to do the merging as the LR(1) states are produced.

### *Merging the states as they are constructed*

Our algorithm for computing the LALR(1) canonical collection is a slight variation on Algorithm 3.9; it is Algorithm 3.12.

***Algorithm 3.12: Computing the LALR(1) Collection of States.***
Input: a context-free grammar G.
Output: The canonical LALR(1) collection of states c = $\{s_0, s_1, \ldots, s_n\}$

1. Define an augmented grammar G' which is G with the added non-terminal S' and added production rule S' ::= S, where S is G's start symbol. The following steps apply to G'. Enumerate the production rules beginning at 0 for the newly added production.
2. c = {$s_0$}, where $s_0$ = closure({[S' ::= • S, **#**]}).
3. Repeat until no new states are added to c,
   For each s in c, and for each symbol X ∈ T ∪ N,
   If goto(s,X) ≠ ∅ and goto(s,X) ∉ c,
   Let s = goto(s,X) to c.
   Check to see if the cores of an existing state in c are equivalent to the cores of s.
   If so, merge s with that state.
   Otherwise, add s to the collection c.

There are other enhancements we can make to Algorithm 3.12 to conserve even more space. For example, as the states are being constructed, it is enough to store their kernels. The closures may be computed when necessary, and even these may be cached for each non-terminal symbol.

### 3.4.3.2 LALR(1) Conflicts

There is the possibility that the LALR(1) table for a grammar may have conflicts where the LR(1) table does not. Therefore, while it should be obvious that every LALR(1) grammar is a LR(1) grammar, not every LR(1) grammar is a LALR(1) grammar.

How can these conflicts arise? A shift-reduce conflict cannot be introduced by merging two states, because we merge two states only if they have the same core items. If the merged state has an item that suggests a shift on a terminal **a** and another item that suggests a reduce on the look-ahead **a**, then at least one of the two original states must have contained both items, and so caused a conflict.

On the other hand, merging states can introduce reduce-reduce conflicts. An example arises in grammar given in exercise 3.20.

Even though LALR(1) grammars are not so powerful as LR(1) grammars, they are sufficiently powerful to describe most programming languages. This, together with their small (relative to LR) table size makes LALR(1) family of grammars an excellent candidate for the automatic generation of parsers. Stephen C. Johnson's YACC, for "yet another compiler-compiler" (Johnson, 1975), based on LALR(1) techniques, was probably the first practical bottom-up parser generator. GNU has developed an open-source version called Bison (Bison, 2008).

### 3.4.4 LL or LR?

Figure 3.13 illustrates the relationships among the various categories of grammars we have been discussing.

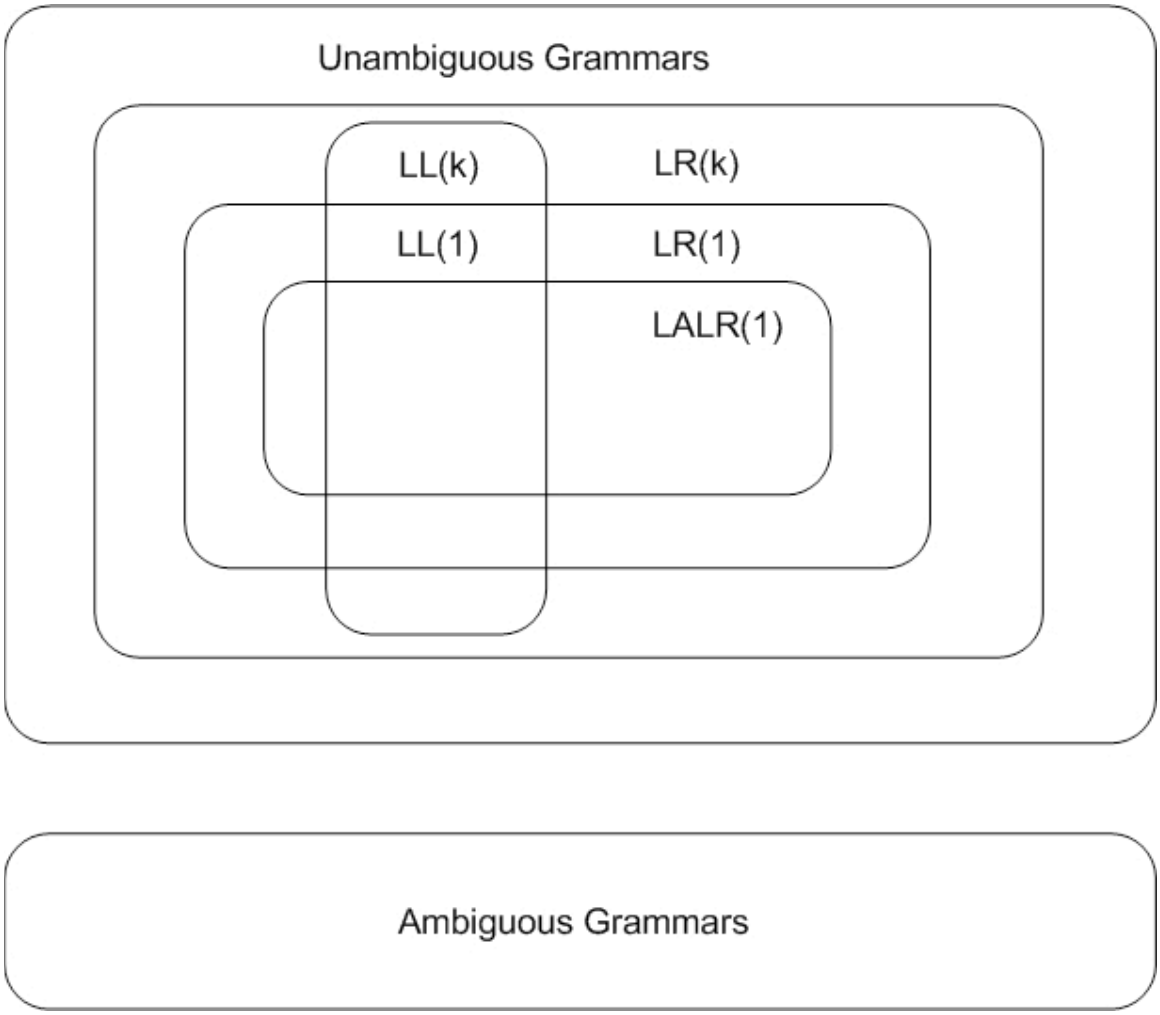



**Figure 3.13 Categories of Context-free Grammars and their Relationship**

Theoretically, LR(1) grammars are the largest category of grammars that can be parsed deterministically while looking ahead just one token. Of course, LR(k) grammars for k>1 are even more powerful, but one must look ahead k tokens more importantly, the parsing tables must (in principle) keep track of all possible token strings of length k. So, in principle, the tables can grow exponentially with k.

LALR(1) grammars make for parsers that are almost as powerful as LR(1) grammars but result in much more space-efficient parsing tables. This goes some way in explaining the popularity of parser generators like yacc and Bison.

LL(1) grammars are the least powerful category of grammars that we have looked at. Every LL(1) grammar is an LR(1) grammar and every LL(k) grammar is an LR(k) grammar.

Also, every LR(k) grammar is an unambiguous grammar. Indeed, the LR(k) category is the largest category of grammars for which we have a means for testing membership. There is no general algorithm for telling us whether an arbitrary context-free grammar is unambiguous. But we can test for LR(1) or LR(k) for some k, and if it is LR(1) or LR(k) then it is unambiguous.

In principle, recursive descent parsers work only when based on LL(1) grammars. But as we have seen, one may program the recursive descent parser to look ahead a few symbols, in those places in the grammar where the LL(1) condition does not hold.

LL(1), LR(1), LALR(1) and recursive descent parsers have all been used to parse one programming language or another. LL(1) and recursive descent parsers have been applied to most of Nicklaus Wirth's languages, e.g. Algol-W, Pascal and Modula. Recursive descent was used to produce the parser for the first implementations of C but then using YACC; that and the fact that YACC was distributed with Unix popularized it. YACC was the first LALR(1) parser generator with a reasonable execution time.

Interestingly, LL(1) and recursive descent parsers are enjoying greater popularity, for example for the parsing of Java. Perhaps it is the simplicity of the predictive top-down approach. It is now possible to come up with (mostly LL) predictive grammars for most programming languages. True, none of these grammars are strictly LL(1); indeed they are not even unambiguous – look at the if-else statement in Java and almost every other programming language. But these special cases may be handled specially, for example by selective looking ahead k symbols in rules where it is necessary, and favoring the scanning of an **`else`** when it is part of an **`if`**-statement. There are parser generators that allow the parser developer to assert these special conditions in the grammatical specifications. One of these is JavaCC, which we discuss in the next section.

## *3.5 Parser Generation Using JavaCC*

In chapter 2 we saw how JavaCC can be used to generate a lexical analyzer for *j--* from an input file (`j--.jj`) specifying the lexical structure of the language as regular expressions. In this section, we'll see how JavaCC can be used to generate a LL(k) recursive descent parser for *j--* from a file specifying its syntactic structure as EBNF (extended BNF) rules.

Besides containing the regular expressions for the lexical structure for *j--,* the **`j--.jj`** file also contains the syntactic rules for the language. The Java code between the **`PARSER_BEGIN(JavaCCParser)`** and **`PARSER_END(JavaCCParser)`** block in **`j--.jj`** file is copied verbatim to the generated **`JavaCCParser.java`** file in the **`jminusminus`** package. This code defines helper functions, which are available for use within the generated parser. Some of the helpers include: **`reportParserError()`** for reporting errors, and **`recoverFromError()`** for recovering from errors. Following this block is the specification for the scanner for *j--*, and following that is the specification for the parser for *j--*.

We now describe the JavaCC syntactic specification. The general layout is this: we define a start symbol, which is a high level non-terminal (**`compilationUnit`** in case of *j--*) that references lower level non-terminals. These lower level non-terminals in turn reference the tokens defined in the lexical specification.

When building a syntactic specification, we are not limited to literals and simple token references. We can use the following EBNF syntax.

- **`[a]`** for "zero or one", or an "optional" occurrence of **`a`**
- **`(a)*`** for "zero or more" occurrences of **`a`**
- **`a | b`** for alternation, i.e., either **`a`** or **`b`**
- **`()`** for grouping

The syntax for a non-terminal declaration (or, production rule) in the input file almost resembles that of a java method declaration; it has a return type (could be **`void`**), a name, can accept arguments, and has a body, which specifies the extended BNF rules along with any actions that we want performed as the production rule is parsed. Besides, it also has a block preceding the body; this block declares any local variables used within the body. Syntactic actions, such as creating an AST node, are java code embedded within blocks. JavaCC turns the specification for each non-terminal into a java method within the generated parser.

As an example, let's look at how we specify the rule (3.18) repeated here,

qualifiedIdentifier ::= **`IDENTIFIER`** {**`. IDENTIFIER`**}

for parsing a qualified identifier using JavaCC.

```
private TypeName qualifiedIdentifier():
{
    int line = 0;
    String qualifiedIdentifier = "";
}
{
```

```
    try {
        <IDENTIFIER>
        {
            line = token.beginLine;
            qualifiedIdentifier = token.image;
        }
        (
            <DOT> <IDENTIFIER>
            {qualifiedIdentifier += "." + token.image;}
        )*
    }
    catch (ParseException e) {
        recoverFromError(new int[] {SEMI, EOF}, e);
    }
    {return new TypeName(line, qualifiedIdentifier);}
}
```

Let us walk through the above method in order to make sense out of it.

- The qualifiedIdentifier non-terminal, as in the case of the hand-written parser, is **`private`** method and returns an instance of **`TypeName`**. The method does not take any arguments.

- The local variable block defines two variables, **`line`** and **`qualifiedIdentifier`**; the former is for tracking line number in the source file, and the latter is for accumulating the individual identifiers (**`x`**, **`y`**, and **`z`** in **`x.y.z`** for example) into a qualified identifier (**`x.y.z`** for example). The variables **`line`** and **`qualifiedIdentifier`** are used within the body that actually parses a qualified identifier.

- In the body, all parsing is done within the **`try-catch`** block. When an identifier token[7] **`<IDENTIFIER>`** is encountered, its line number in the source file and its image are recorded in the respective variables; this is an action and hence is within a block. We then look for zero or more occurrences of the tokens **`<DOT> <IDENTIFIER>`** within the **`()*`** EBNF construct. For each such occurrence, we append the image of the identifier to the **`qualifiedIdentifier`** variable; this is also an action and hence is java code within a block. Once a qualified identifer has been parsed, we return (again an action and hence is java code within a block) an instance of **`TypeName`**.

[7] The token variable stores the current token information; **`token.beginLine`** stores the line number in which the token occurs in the source file and **`token.image`** stores the token's image (for example, the identifier name in case of the **`<IDENTIFIER>`** token).

- JavaCC raises a **`ParseException`** when encountering a parsing error. The instance of **`ParseException`** stores information about the token that was found and the token that was sought. When such an exception occurs, we invoke our own error recovery method **`recoverFromError()`** and try to recover to the nearest semi colon (**`SEMI`**) or to the end of file (**`EOF`**). We pass to this method the instance of **`ParseException`** so that the method can report a meaningful error message.

As another example, let's see how we specify the non-terminal **`statement`** (3.19) repeated here,

statement ::= block
| **`if`** parExpression statement [**`else`** statement]
| **`while`** parExpression statement
| **`return`** [expression] **`;`**
| **`;`**
| statementExpression **`;`**

for parsing statements in *j--*.

```
private JStatement statement():
{
    int line = 0;
    JStatement statement = null;
    JExpression test = null;
    JStatement consequent = null;
    JStatement alternate = null;
    JStatement body = null;
    JExpression expr = null;
}
{
    try {
        statement = block() |
        <IF> {line = token.beginLine;}
        test = parExpression()
        consequent = statement()
        [
            <ELSE> alternate = statement()
       ]
        {statement =
            new JIfStatement(line, test, consequent,
                             alternate);} |
        <WHILE> {line = token.beginLine;}
        test = parExpression()
        body = statement()
        {statement = new JWhileStatement(line, test,
                                         body);} |
        <RETURN> {line = token.beginLine;}
        [
            expr = expression()
```

```
        ]
         <SEMI>
         {statement = new JReturnStatement(line, expr);} |
         <SEMI>
         {statement = new JEmptyStatement(line);} |
         // Must be a statementExpression
         statement = statementExpression()
         <SEMI>
    }
     catch (ParseException e) {
         recoverFromError(new int[] {SEMI, EOF}, e);
    }
     {return statement;}
}
```

We'll jump right into the **`try`** block to see what's going on.

- If the current token is **`<LCURLY>`**, which marks the beginning of a block, the lower level non-terminal **`block`** is invoked to parse block-statement. The value returned by **`block`** is assigned to the local variable **`statement`**.

- If the token is **`<IF>`**, we get its line number and parse an **`if`** statement; we delegate to the lower level non-terminals **`parExpression`** and **`statement`** to parse the test expression, the consequent and the (optional) alternate statements. Note the use of the **`|`** and **`[]`** JavaCC constructs for alternation and option. Once we have successfully parsed an **`if`** statement, we create an instance of the AST node for an **`if`** statement and assign it to the local variable **`statement`**.

- If the token is **`<WHILE>`**, **`<RETURN>`**, or **`<SEMI>`**, we parse a **`while`**, **`return`**, or an empty statement.

- Otherwise, it must be a statement expression, which we parse by simply delegating to the lower level non-terminal **`statementExpression`**. In each case we set the local variable **`statement`** to the appropriate AST node instance.

Finally, we return the local variable **`statement`** and this completes the parsing of a *j--* statement.

## Look-ahead

As in the case of a recursive descent parser, we cannot always decide which production rule to use in parsing a non-terminal just by looking at the current token; we have to *look*

*ahead* at the next few symbols to decide. JavaCC offers a function called **LOOKAHEAD** that we can use for this purpose. Here is an example in which we parse a simple unary expression in *j--*, expressed by the BNF rule (3.20), and repeated here,

simpleUnaryExpression ::= **!** unaryExpression
| **(**basicType) unaryExpression  / / cast
| **(**referenceType) simpleUnaryExpression // cast
| postfixExpression

```
private JExpression simpleUnaryExpression():
{
    int line = 0;
    Type type = null;
    JExpression expr = null, unaryExpr = null,
        simpleUnaryExpr = null;
}
{
    try {
        <LNOT> {line = token.beginLine;}
        unaryExpr = unaryExpression()
        {expr = new JLogicalNotOp(line, unaryExpr);} |
        LOOKAHEAD(<LPAREN> basicType() <RPAREN>)
        <LPAREN> {line = token.beginLine;}
        type = basicType()
        <RPAREN>
        unaryExpr = unaryExpression()
        {expr = new JCastOp(line, type, unaryExpr);} |
        LOOKAHEAD(<LPAREN> referenceType() <RPAREN>)
        <LPAREN> {line = token.beginLine;}
        type = referenceType()
        <RPAREN>
        simpleUnaryExpr = simpleUnaryExpression()
        {expr = new JCastOp(line, type, simpleUnaryExpr);} |
        expr = postfixExpression()
   }
    catch (ParseException e) {
        recoverFromError(new int[] {SEMI, EOF}, e);
   }
    {return expr ;}
}
```

We use the **LOOKAHEAD** function to decide between a cast expression involving a basic type, a cast expression involving a reference type, and a postfix expression, which could also begin with an **LPAREN** (**(x)** for example). Notice how we are spared the chore of writing our own non-terminal-specific look-ahead functions. Instead, we simply invoke JavaCC **LOOKAHEAD** function by passing in tokens we want to look ahead. Thus we don't have to worry about backtracking either; **LOOKAHEAD** does it for us behind the scenes.

Also notice how, as in **`LOOKAHEAD(<LPAREN> basicType() <RPAREN>)`**, we can pass both terminals and non-terminals to **`LOOKAHEAD`**.

## Error Recovery

JavaCC offers two error recovery mechanisms, namely *shallow* and *deep* error recovery. We employ the latter in our implementation of a JavaCC parser for *j--*. This involves catching within the body of a non-terminal, the **`ParseException`** that is raised in the event of a parsing error. The exception instance **`e`** along with **`skipTo`** tokens is passed to our **`recoverFromError()`** error recovery function. The exception instance has information about the erroneous token that was found and the token that was expected, and **`skipTo`** is an array of tokens that we would like to skip to in order to recover from the error. Here's the function.

```
private void recoverFromError(int[] skipTo, ParseException e)
{
    // Get the possible expected tokens
    StringBuffer expected = new StringBuffer();
    for (int i = 0; i < e.expectedTokenSequences.length; i++) {
        for (int j = 0; j <
              e.expectedTokenSequences[i].length; j++) {
            expected.append("\n");
            expected.append("    ");
            expected.append(tokenImage[
                e.expectedTokenSequences[i][j]]);
            expected.append("...");
        }
   }

    // Print error message
    if (e.expectedTokenSequences.length == 1) {
        reportParserError("\"%s\" found where %s sought",
            getToken(1), expected);
   }
    else {
        reportParserError("\"%s\" found where one of %s sought",
            getToken(1), expected);
   }

    // Recover
    boolean loop = true;
    do {
        token = getNextToken();
        for (int i = 0; i < skipTo.length; i++) {
            if (token.kind == skipTo[i]) {
                loop = false;
                break;
            }
        }
   } while(loop);
```

```
}
```

Firstly, the function, from the token that was found and the token that was sought, constructs and displays an appropriate error message. Secondly, it recovers by skipping to the nearest token in the **`skipTo`** list of tokens.

In the current implementation of the parser for *j--*, all non-terminals specify **`SEMI`** and **`EOF`** as **`skipTo`** tokens. This error recovery scheme could be made more sophisticated by specifying the follow of the non-terminal as **`skipTo`** tokens.

Note that when **`ParseException`** is raised, control is transferred to the calling non-terminal. Thus when an error occurs within higher non-terminals, the lower non-terminals go unparsed.

### Generating a Parser Versus Hand-Writing a Parser

If you compare the JavaCC specification for the parser for *j--* with the hand-written parser, you'll notice that they are very much alike. This would make you wonder whether we are gaining anything by using JavaCC. The answer is, yes we are. Here are some of the benefits.

- Lexical structure is much more easily specified using regular expressions.
- EBNF constructs are allowed.
- Lookahead is easier; it is given as a function and takes care of backtracking.
- Choice conflicts are reported when lookahead is insufficient
- Sophisticated error recovery mechanisms are available.

Other parser generators, including ANTLR[8] for Java, also offer the above advantages over hand-written parsers.

## *3.6 Tuples – an Alternative Intermediate Representation*

An abstract syntax tree (AST) is an excellent intermediate representation (IR) for programs for several reasons:

1. An AST roughly reflects the syntax of the original program.
2. An AST is easily traversed for performing semantic analysis.
3. It is easy to add fields to the various nodes for decorating the tree with additional information, such as types.

[8] http://www.antlr.org/

4. Modifications to programs represented as an AST are easily made using tree surgery. Tree surgery may also be used to apply optimizations, replacing slow code with better code.
5. An AST is easily traversed for generating a linear sequence of code, for example an assembly language program or the code attribute for a .class file.

For these reasons, we use an AST in our compiler for *j--*. But it's not the only IR available to us. An alternative representation is that using tuples.

A *tuple* is a statement of the form

```
x = y op z
```

where **x**, **y**, and **z** are names, constants, or compiler-generated temporaries, and **op** is an operator, such as arithmetic or a logical operator. Note that **x** and **y** cannot be compound expressions; only one operator is allowed on the right hand side of the statement. For example, an expression like **a + b * c** in the source language might be translated into a sequence

```
t1 = b * c
t2 = a + t1
```

where **t1** and **t2** are compiler-generated temporary names. Complex expressions and nested flow-of-control statements can be expressed using tuples as a linear list of statements. Tuples are very useful for target code generation and optimization because they are so close to machine code.

Each tuple stores three addresses, two for the operands and one for the result. Hence tuples are also called "three-address code."

Tuples provide a linearized representation of a syntax tree. The names in the tuples correspond to the interior nodes of the syntax tree. Variable names can appear directly in the tuples, so there are no statements corresponding to the leaf nodes in the syntax tree. For example, consider the *j--* assignment statement

```
x = -a + b * c
```

The syntax tree for the above statement is shown below.

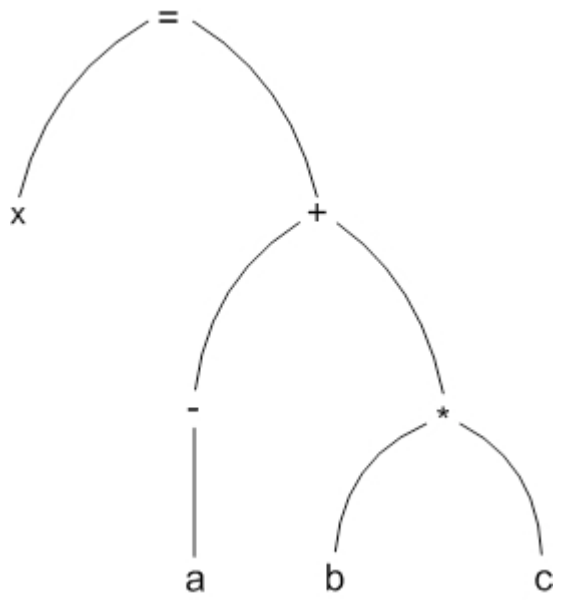



**Figure 3.14 Syntax tree for `x = -a + b * c`**

We can express the same assignment statement using tuples as follows:

```
t1 = - a
t2 = b * c
t3 = t1 + t2
x  = t3
```

**Tuple Statement Types**

Tuples are similar to assembly code. There are flow-of-control statements and statements can have named labels associated to them. The label represents the index of a tuple statement in the array holding the intermediate representation. The labels can be resolved into actual indices in a separate pass. Common tuple types are:

1. Assignment statements of the form **`x = y op z`**, where **`op`** is a binary arithmetic or logical operator.
2. Assignment statements of the form **`x = op y`**, where **`op`** is a unary operator.
3. Copy statements of the form **`x = y`**, where the value of **`y`** is assigned to **`x`**.
4. The unconditional jump **`goto L`**, which implies that the next statement to be executed is the tuple statement labeled **`L`**.
5. Conditional jumps such as **`if x relop y goto L`**. This statement applies the relational operator (**`<, <=, =, !=, >, >=`**) to **`x`** and **`y`**, and executes the tuple labeled **`L`** if **`x`** is related to **`y`**; if **`x`** is not related to **`y`**, the statement following **`if x relop y goto L`** is executed.

6. A method call **`m(x1, x2, ..., xn)`** can be expressed as a sequence of tuples

```
arg x1
arg x2
...
arg xn
call m, n
```

   Since calls can be nested, we need to indicate the number of actual arguments in **`call m, n.`**

   The tuple **`return y`** is meant to return from a method, where the optional argument **`y`** is value returned.

7. Indexed assignments of the form **`x[i] = y`** and **`x = y[i]`**. The former sets the contents of memory at location **`i`** units after **`x`** to the value of **`y`**, while the latter sets the value of x to the contents of memory at location **`i`** units after location **`y`**.

In designing an intermediate form, one must wisely choose the set of allowable operators, such that it is rich enough to implement the operations in the source language. It is easier to implement a small operator set on a new target machine, but such a reduced instruction set would force the front end to produce long sequences of tuples for some of the source language operations, consequently burdening the optimizer and code generator if good target code is desirable.

Tuples can be represented as records with fields for the operator and the operands. Quadruples and triples are two such representations.

A quadruple is a record structure with four fields: **`op`**, **`arg1`**, **`arg2`**, and **`result`**. The **`op`** field contains the internal code for the operator, **`arg1`** and **`arg2`** are the operands, and **`result`** is the result of the operation. Statements with unary operators do not use **`arg2`**. Operators like **`arg`** use just one address. Conditional and unconditional jumps put the target label in **`result`**. The quadruples for the *j*-- assignment statement **`x = -a + b * c`** is shown in Figure 3.15.

| | op | arg1 | arg2 | result |
|---|---|---|---|---|
| 0 | - | a | | t1 |
| 1 | * | b | c | t2 |
| 2 | + | t1 | t2 | t3 |
| 3 | = | t3 | | x |

**Figure 3.15 Quadruples for `x = -a + b * c`**

The contents of **arg1**, **arg2**, and **result** are usually pointers to the symbol-table entries for the names represented by these fields, wherein temporary names must be entered into the symbol table as they are created.

Triples avoid entering temporary names into the symbol table by referring to a temporary value by the position of the tuple that computes it. Thus, triples can be represented by records with only three fields: **op**, **arg1**, and **arg2**. The fields **arg1** and **arg2** are pointers to the symbol table (for user-defined names or constants) or pointers to the triple structure (for temporary names). The triples for the *j*-- assignment statement **x = -a + b * c** is shown in Figure 3.16.

| | op | arg1 | arg2 |
|---|---|---|---|
| 0 | - | a | |
| 1 | * | b | c |
| 2 | + | 0 | 1 |
| 3 | = | x | 2 |

**Figure 3.16 Triples for `x = -a + b * c`**

We'll make use of tuples in Chapter 7, where we translate JVM code to native MIPS code.

## *Further Reading*

(Aho and Ullman, 1972) gives a thorough and classic overview of context-free parsing.

The context-free syntax for Java may be found in (Gosling et al, 2005); see chapters 2 and 18. This book is also published online on the Sun Developer Network at http://java.sun.com/docs/books/jls.

LL(1) parsing was introduced in (Lewis and Stearns, 1968) and (Knuth, 1971). Recursive descent was introduced in (Lewis, Rosenkrantz and Stearns, 1976). The simple error-recovery scheme used in our parser comes from (Turner, 1977).

See chapters 5 in (Copeland, 2007) for more on how to generate parsers using JavaCC. See chapter 7 for more information on error recovery. See chapter 8 for a case study -- parser for JavaCC grammar. JavaCC itself is open-source software, which may be obtained from https://javacc.dev.java.net/. Also, see (van der Spek, Plat and Pronk, 2005) for a discussion of error recovery in JavaCC.

See (Johnson, 1975) for an introduction to YACC. The canonical open-source implementation of the LALR(1) approach to parser generation is (Bison, 2008).

Other shift-reduce parsing strategies include both simple-precedence and operator-precedence parsing. These are nicely discussed in (Gries, 1971).

## *Exercises*

3.1 Consult Chapter 18 of *The Java Language Specification, Third Edition* (see Further Reading, above). There you will find a complete specification of Java's context-free syntax.

a. Make a list of all the expressions that are in Java but not in *j--*.
b. Make a list of all statements that are in Java but not in *j--*.
c. Make a list of all type declarations that are in Java but not in *j--*.
d. What other linguistic constructs are in Java but not *j--*?

3.2 Consider the following grammar.

S ::= **(**L) | **a**
L ::= L S | ε

a. What language does this grammar describe?
b. Show the parse tree for the string **`(a () (a (a)))`**.
c. Derive an equivalent LL(1) grammar.

3.3 Show that the following grammar is ambiguous.

S ::= **a**S**b**S | **b**S**a**S | ε

3.4 Show that the following grammar is ambiguous. Come up with an equivalent grammar that is not ambiguous.

E ::= E **`and`** E | E **`or`** E | **`true`** | **`false`**

3.5 Write a grammar that describes the language of Roman numerals.

3.6 Write a grammar that describes Lisp s-expressions.

3.7 Write a grammar that describes a number of (zero or more) **a**'s followed by an equal number of **b**'s.

3.8 Show that the following grammar is not LL(1).

S ::= **`a b`** | A **b**

A ::= **a a** | c **d**

3.9 Consider the following context-free grammar.

S ::= B **a** | **a**
B ::= **c** | **b** C B
C ::= **c** C | ε

a. Compute first and follow for S, B and C.
b. Construct the LL(1) parsing table for this grammar.
c. Is this grammar LL(1)? Why or why not?

3.10 Consider the following context-free grammar.

S ::= A **a** | **a**
A ::= **c** | **b** B
B ::= **c** B | ε

a. Compute first and follow for S, A and B.
b. Construct the LL(1) parsing table for the grammar.
c. Is this grammar LL(1)? Why or why not?

3.11 Consider the following context-free grammar.

S ::= A **a**
A ::= **b d** B | **e** B
B ::= **c** A | **d** B | ε

a. Compute first and follow for S, A and B.
b. Construct an LL(1) parsing table for this grammar.
c. Show the steps in parsing **eada**.

3.12 Consider the following context-free grammar.

S ::= AS | **b**
A ::= SA | **a**

a. Compute first and follow for S and A.
b. Construct the LL(1) parsing table for the grammar.
c. Is this grammar LL(1)? Why or why not?

3.13 Show that the following grammar is LL(1)

S ::= A **a** A **b**
S ::= B **b** B **a**
A ::= ε
B ::= ε

3.14 Consider the following grammar.

E ::= E **or** T | T
T ::= T **and** F | F
F ::= **not** F | **(**E) | **i**

a. Is this grammar LL(1)? If not, derive an equivalent grammar that is LL(1).
b. Construct the LL(1) parsing table for the LL(1) grammar.
c. Show the steps in parsing **not i and i or i**.

3.15 Consider the following grammar.

S ::= L **=** R
S ::= R
L ::= ***** R
L ::= **i**
R ::= L

a. Construct the canonical LR(1) collection.
b. Construct the Action and Goto tables.
c. Show the steps of the parse for ***i=i** .

3.16 Consider the following grammar.

S ::= **(**L) | **a**
L ::= L **,** S | S

a. Compute the canonical collection of LR(1) items for this grammar.
b. Construct the LR(1) parsing table for this grammar.
c. Show the steps in parsing the input string **((a,a),a)**.
d. Is this a LALR(1) grammar?

3.17 Consider the following grammar.

S ::= A **a** | **b** A **c** | **d c** | **b d a**
A ::= **d**

a. What is the language described by this grammar?

b. Compute first and follow for all non-terminals.
c. Construct the LL(1) parsing table for this grammar. Is it LL(1)? Why?
d. Construct the LR(1) canonical collection, and the Action and Goto tables for this grammar. Is it LR(1)? Why or why not?

3.18 Is the following grammar LR(1)? LALR(1)?

S ::= C C
C ::= **a** C | **b**

3.19 Consider the following context-free grammar.

S ::= A **a** | **b** A **c** | **d c** | **b d a**
A ::= **d**

a. Compute the canonical LR(1) collection for this grammar.
b. Compute the Action and Goto tables for this grammar.
c. Show the steps in parsing **bdc**.
d. Is this a LALR(1) grammar?

3.20 Show that the following grammar is LR(1) but not LALR(1).

S ::= **a** B **c** | **b** C **d** | **a** C **d b** B **d**
B ::= **e**
C ::= **e**

3.21 Modify the **Parser** to parse and return nodes for the **double** literal and the **float** literal.

3.22 Modify the **Parser** to parse and return nodes for the **long** literal.

3.23 Modify the **Parser** to parse and return nodes for all the additional operators that are defined in Java but not yet in j--.

3.24 Modify the **Parser** to parse and return nodes for conditional expressions, e.g. `(a>b) ? a : b.`

3.25 Modify the **Parser** to parse and return nodes for the for-statement, including both the basic for-statement and the enhanced for-statement.

3.26 Modify the **Parser** to parse and return nodes for the switch statement.

3.27 Modify the **Parser** to parse and return nodes for the try-catch-finally statement.

3.28 Modify the **`Parser`** to parse and return nodes for the throw statement.

3.29 Modify the **`Parser`** to deal with a throws clause in method declarations.

3.30 Modify the **`Parser`** to deal with methods and constructors having variable arity, that is, variable number of arguments.

3.31 Modify the **`Parser`** to deal with both static blocks and instance blocks in type declarations.

3.32 Modify the **`j--.jj`** file in the compiler's code tree for adding the above (3.21 through 3.31) syntactic constructs to *j--*.

3.33 Say we wish to add a do-until statement to *j--*. For example,

```
do
      x = x * x;
until (x > 1000);
```

a. Write a grammar rule for defining the context-free syntax for a new do-until statement.
b. Modify the **`Scanner`** to deal with any necessary new tokens.
c. Modify the **`Parser`** to parse and return nodes for the do-until statement.