Chapter 5

Bottom Up Parsing

The implementation of parsing algorithms for LL(1) grammars , as shown in Chapter 4, is relatively straightforward. However, there are many situations in which it is not easy, or possible, to use an LL(1) grammar. In these cases, the designer may have to use a bottom up algorithm.

Parsing algorithms which proceed from the bottom of the derivation tree and apply grammar rules (in reverse) are called ***bottom up parsing algorithms***. These algorithms will begin with an empy stack. One or more input symbols are moved onto the stack, which are then replaced by nonterminals according to the grammar rules. When all the input symbols have been read, the algorithm terminates with the starting nonterminal alone on the stack, if the input string is acceptable. The student may think of a bottom up parse as being similar to a derivation in reverse. Each time a grammar rule is applied to a sentential form, the rewriting rule is applied backwards. Consequently, derivation trees are constructed, or traversed, from bottom to top.

5.1 Shift Reduce Parsing

Bottom up parsing involves two fundamental operations. The process of moving an input symbol to the stack is called a ***shift*** operation, and the process of replacing symbols on the top of the stack with a nonterminal is called a ***reduce*** operation (it is a derivation step in reverse). Most bottom up parsers are called ***shift reduce*** parsers because they use these two operations. The following grammar will be used to show how a shift reduce parser works:

G19:

```
1.    S → S a B
```

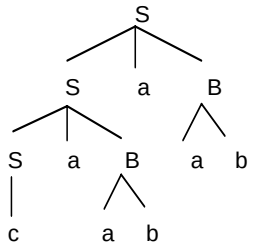

Figure 5.1 A Derivation Tree for the String caabaab, Using Grammar G19

2. S → c
3. B → a b

A derivation tree for the string caabaab is shown in Figure 5.1. The shift reduce parser will proceed as follows: each step will be either a shift (shift an input symbol to the stack) or reduce (reduce symbols on the stack to a nonterminal), in which case we indicate which rule of the grammar is being applied. The sequence of stack frames and input is shown in Figure 5.2, in which the stack frames are pictured horizontally to show, more clearly, the shifting of input characters onto the stack and the sentential forms corresponding to this parse. The algorithm accepts the input if the stack can be reduced to the starting nonterminal when all of the input string has been read.

Note in Figure 5.2 that whenever a reduce operation is performed, the symbols being reduced are always on top of the stack. The string of symbols being reduced is called a ***handle***, and it is imperative in bottom up parsing that the algorithm be able to find a handle whenever possible. The bottom up parse shown in Figure 5.2 corresponds to the derivation shown below:

S ⇒ S a B ⇒ S a a b ⇒ S a B a a b ⇒ S a a b a a b
⇒ c a a b a a b

Note that this is a right-most derivation; shift reduce parsing will always correspond to a right-most derivation. In this derivation we have underlined the handle in each sentential form. Read this derivation from right to left and compare it with Figure 5.2.

If the parser for a particular grammar can be implemented with a shift reduce algorithm, we say the grammar is ***LR*** (the L indicates we are reading input from the ***left***, and the R indicates we are finding a ***right-most*** derivation). The shift reduce parsing algorithm always performs a reduce operation when the top of the stack corresponds to the right side of a rule. However, if the grammar is not LR, there may be instances where this is not the correct operation, or there may be instances where it is not clear which reduce operation should be performed. For example, consider grammar G20:

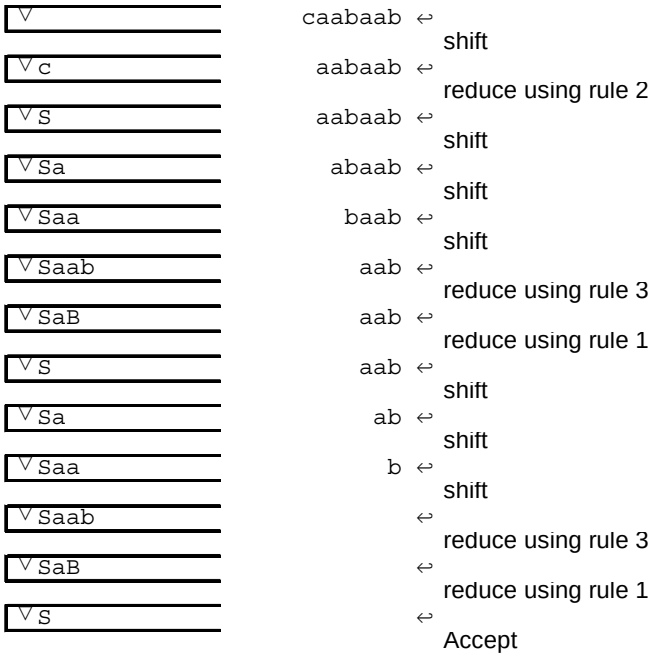

Figure 5.2 Sequence of Stack Frames Parsing `caabaab` Using Grammar G19

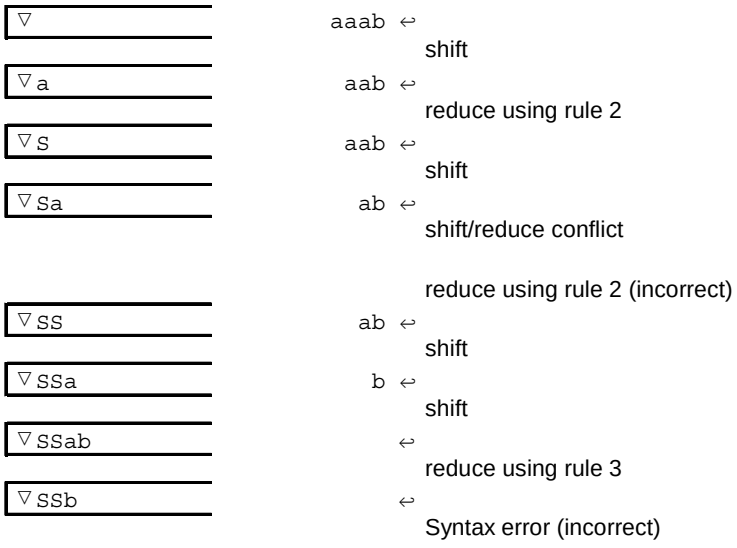

Figure 5.3 An Example of a Shift/Reduce Conflict Leading to an Incorrect Parse Using Grammar G20

G20:

```
1.    S  →  S a B
2.    S  →  a
3.    B  →  a b
```

G21:

```
1.    S  →  S A
2.    S  →  a
3.    A  →  a
```

When parsing the input string `aaab`, we reach a point where it appears that we have a handle on top of the stack (the terminal `a`), but reducing that handle, as shown in Figure 5.3, does not lead to a correct bottom up parse. This is called a ***shift/reduce conflict*** because the parser does not know whether to shift an input symbol or reduce the handle on the stack. This means that the grammar is not LR, and we must either rewrite the grammar or use a different parsing algorithm.

Another problem in shift reduce parsing occurs when it is clear that a reduce operation should be performed, but there is more than one grammar rule whose right hand side matches the top of the stack, and it is not clear which rule should be used. This is called a ***reduce/reduce conflict***. Grammar G21 is an example of a grammar with a reduce/reduce conflict.

Figure 5.4 shows an attempt to parse the input string `aa` with the shift reduce algorithm, using grammar G21. Note that we encounter a reduce/reduce conflict when the handle `a` is on the stack because we don't know whether to reduce using rule 2 or rule 3. If we reduce using rule 2, we will get a correct parse, but if we reduce using rule 3 we will get an incorrect parse.

It is often possible to resolve these conflicts simply by making an assumption. For example, all shift/reduce conflicts could be resolved by shifting rather than reducing. If this assumption always yields a correct parse, there is no need to rewrite the grammar.

In examples like the two just presented, it is possible that the conflict can be resolved by looking ahead at additional input characters. An LR algorithm that looks ahead k input symbols is called ***LR(k)***. When implementing programming languages bottom up, we generally try to define the language with an LR(1) grammar, in which case the algorithm will not need to look ahead beyond the current input symbol. An ambigu-

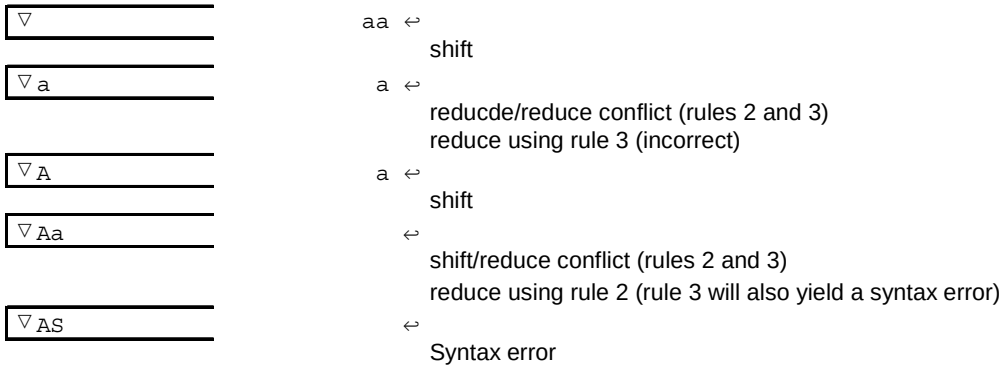

Figure 5.4 A Reduce/Reduce Conflict

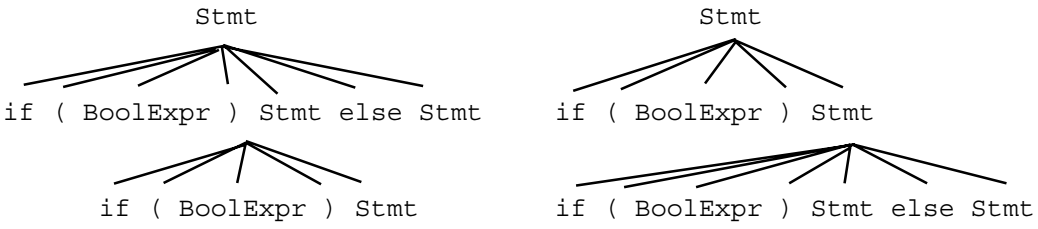

Figure 5.5 Two Derivation Trees for `if (BoolExpr) if (BoolExpr) Stmt else Stmt`

ous grammar is not LR(k) for any value of k – i.e. an ambiguous grammar will always produce conflicts when parsing bottom up with the shift reduce algorithm. For example, the following grammar for `if` statements is ambiguous:

```
1.    Stmt  →  if (BoolExpr) Stmt else Stmt
2.    Stmt  →  if (BoolExpr) Stmt
```

The BoolExpr in parentheses represents a true or false condition. Figure 5.5, above, shows two different derivation trees for the statement `if (BoolExpr) if (BoolExpr) Stmt else Stmt`. The tree on the right is the interpretation preferred by most programming languages (each `else` is matched with the closest preceding unmatched `if`). The parser will encounter a shift/reduce conflict when reading the `else`. The reason for the conflict is that the parser will be configured as shown, below, in Figure 5.6.

In this case, the parser will not know whether to treat `if (BoolExpr) Stmt` as a handle and reduce it to `Stmt` according to rule 2, or to shift the `else`, which should be followed by a `Stmt`, thus reducing according to rule 1. However, if the parser can somehow be told to resolve this conflict in favor of the shift, then it will always find the correct interpretation. Alternatively, the ambiguity may be removed by rewriting the grammar, as shown in Section 3.1.

Stack	Input
▽ ... if (BoolExpr) Stmt	else ... ↩

Figure 5.6 Parser Configuration Before Reading the `else` Part of an `if` Statement

Sample Problem 5.1

Show the sequence of stack and input configurations as the string `caab` is parsed with a shift reduce parser, using grammar G19.

Solution:

▽	caab ↩	shift
▽c	aab ↩	reduce using rule 2
▽S	aab ↩	shift
▽Sa	ab ↩	shift
▽Saa	b ↩	shift
▽Saab	↩	reduce using rule 3
▽SaB	↩	reduce using rule 1
▽S	↩	Accept

Exercises 5.1

1. For each of the following stack configurations, identify the *handle* using the grammar shown below:

```
1.   S → S A b
2.   S → a c b
3.   A → b B c
4.   A → b c
5.   B → b a
6.   B → A c
```

(a) ▽SSAb

(b) ▽SSbbc

(c) ▽SbBc

(d) ▽Sbbc

2. Using the grammar of Problem 1, show the sequence of *stack and input configurations* as each of the following strings is parsed with shift reduce parsing:

(a) `acb` (b) `acbbcb`
(c) `acbbbacb` (d) `acbbbcccb`
(e) `acbbcbbcb`

3. For each of the following input strings, indicate whether a shift/reduce parser will encounter a *shift/reduce conflict*, a *reduce/reduce conflict*, or *no conflict* when parsing, using the grammar below:

```
1.   S → S a b
2.   S → b A
3.   A → b b
4.   A → b A
5.   A → b b c
6.   A → c
```

(a) `b c`
(b) `b b c a b`
(c) `b a c b`

4. Assume that a shift/reduce parser always chooses the lower numbered rule (i.e., the one listed first in the grammar) whenever a reduce/reduce conflict occurs during parsing, and it chooses a shift whenever a shift/reduce conflict occurs. Show a *derivation tree* corresponding to the parse for the sentential form `if (BoolExpr) if (BoolExpr) Stmt else Stmt` , using the following ambiguous grammar. Since the grammar is not complete, you may have nonterminal symbols at the leaves of the derivation tree.

```
1.   Stmt → if (BoolExpr) Stmt else Stmt
2.   Stmt → if (Expr) Stmt
```

5.2 LR Parsing With Tables

One way to implement shift reduce parsing is with tables that determine whether to shift or reduce, and which grammar rule to reduce. This technique makes use of two tables to control the parser. The first table, called the ***action table***, determines whether a shift or reduce is to be invoked. If it specifies a reduce, it also indicates which grammar rule is to be reduced. The second table, called a ***goto table***, indicates which stack symbol is to be pushed on the stack after a reduction. A shift action is implemented by a push operation followed by an advance input operation. A reduce action must always specify the grammar rule to be reduced. The reduce action is implemented by a Replace operation in which stack symbols on the right side of the specified grammar rule are replaced by a stack symbol from the goto table (the input pointer is retained). The symbol pushed is not necessarily the nonterminal being reduced, as shown below. In practice, there will be one or more stack symbols corresponding to each nonterminal.

The columns of the goto table are labeled by nonterminals, and the the rows are labeled by stack symbols. A cell of the goto table is selected by choosing the column of the nonterminal being reduced and the row of the stack symbol just beneath the handle.

For example, suppose we have the following stack and input configuration:

Stack	Input
▽S	ab↩

in which the bottom of the stack is to the left. The action `shift` will result in the following configuration:

Stack	Input
▽Sa	b↩

The `a` has been shifted from the input to the stack. Suppose, then, that in the grammar, rule 7 is:

```
7.    B → Sa
```

Select the row of the goto table labeled ▽, and the column labeled `B`. If the entry in this cell is `push X`, then the action `reduce 7` would result in the following configuration:

Stack	Input
▽X	b↩

Figure 5.7 shows the LR parsing tables for grammar G5 for arithmetic expressions involving only addition and multiplication (see Section 3.1). As in previous pushdown machines, the stack symbols label the rows, and the input symbols label the columns of the action table. The columns of the goto table are labeled by the nonterminal being

	Action Table					
	+	*	(	)	var	↩
▽			shift (		shift var	
Expr1	shift +					Accept
Term1	reduce 1	shift *		reduce 1		reduce 1
Factor3	reduce 3	reduce 3		reduce 3		reduce 3
(			shift (		shift var	
Expr5	shift +			shift)		
)	reduce 5	reduce 5		reduce 5		reduce 5
+			shift (		shift var	
Term2	reduce 2	shift *		reduce 2		reduce 2
*			shift (		shift var	
Factor4	reduce 4	reduce 4		reduce 4		reduce 4
var	reduce 6	reduce 6		reduce 6		reduce 6

	Goto Table		
	Expr	Term	Factor
▽	push Expr1	push Term2	push Factor4
Expr1			
Term1			
Factor3			
(	push Expr5	push Term2	push Factor4
Expr5			
)			
+		push Term1	push Factor4
Term2			
*			push Factor3
Factor4			
var			

Initial Stack

Figure 5.7 Action and Goto Tables to Parse Simple Arithmetic Expressions

reduced. The stack is initialized with a ▽ symbol, and blank cells in the action table indicate syntax errors in the input string. Figure 5.8 shows the sequence of configurations which would result when these tables are used to parse the input string `(var+var)*var.`

Stack	Input	Action	Goto
▽	(var+var)*var ↩		
		shift (	
▽ (	var+var)*var ↩		
		shift var	
▽ (var	+var)*var ↩		
		reduce 6	push Factor4
▽ (Factor4	+var)*var ↩		
		reduce 4	push Term2
▽ (Term2	+var)*var ↩		
		reduce 2	push Expr5
▽ (Expr5	+var)*var ↩		
		shift +	
▽ (Expr5+	var)*var ↩		
		shift var	
▽ (Expr5+var	)*var ↩		
		reduce 6	push Factor4
▽ (Expr5+Factor4	)*var ↩		
		reduce 4	push Term1
▽ (Expr5+Term1	)*var ↩		
		reduce 1	push Expr5
▽ (Expr5	)*var ↩		
		shift)	
▽ (Expr5)	*var ↩		
		reduce 5	push Factor4
▽ Factor4	*var ↩		
		reduce 4	push Term2
▽ Term2	*var ↩		
		shift *	
▽ Term2*	var ↩		
		shift var	
▽ Term2*var	↩		
		reduce 6	push Factor3
▽ Term2*Factor3	↩		
		reduce 3	push Term2
▽ Term2	↩		
		reduce 2	push Expr1
▽ Expr1	↩		
		Accept	

Figure 5.8 Sequence of Configurations when Parsing `(var+var)*var`

G5

```
1.    Expr  →  Expr + Term
2.    Expr  →  Term
3.    Term  →  Term * Factor
4.    Term  →  Factor
5.    Factor  →  ( Expr )
6.    Factor  →  var
```

The operation of the LR parser can be described as follows:

1. Find the action corresponding to the current input and the top stack symbol.
2. If that action is a shift action:
   a. Push the input symbol onto the stack.
   b. Advance the input pointer.
3. If that action is a reduce action:
   a. Find the grammar rule specified by the reduce action.
   b. The symbols on the right side of the rule should also be on the top of the stack – pop them all off the stack.
   c. Use the nonterminal on the left side of the grammar rule to indicate a column of the goto table, and use the top stack symbol to indicate a row of the goto table. Push the indicated stack symbol onto the stack.
   d. Retain the input pointer.
4. If that action is blank, a syntax error has been detected.
5. If that action is Accept, terminate.
6. Repeat from step 1.

Sample Problem 5.2

Show the sequence of stack, input, action, and goto configurations for the input `var*var` using the parsing tables of Figure 5.7.

Solution

Stack	Input	Action	Goto
▽	var*var ↩		
		shift var	
▽var	*var ↩		
		reduce 6	push Factor4
▽Factor4	*var ↩		
		reduce 4	push Term2
▽Term2	*var ↩		
		shift *	
▽Term2*	var ↩		
		shift var	
▽Term2*var	↩		
		reduce 6	push Factor3
▽Term2*Factor3	↩		
		reduce 3	push Term2
▽Term2	↩		
		reduce 2	push Expr1
▽Expr1	↩		
		Accept	

There are three principle techniques for constructing the LR parsing tables. In order from simplest to most complex or general, they are called: Simple LR (SLR), Look Ahead LR (LALR), and Canonical LR (LR). SLR is the easiest technique to implement, but works for a small class of grammars. LALR is more difficult and works on a slightly larger class of grammars. LR is the most general, but still does not work for all unambiguous context free grammars. In all cases, they find a rightmost derivation when scanning from the left (hence LR). These techniques are beyond the scope of this text, but are described in Parsons[1992] and Aho [1986].

Exercises 5.2

1. Show the sequence of *stack and input configurations* and the *reduce and goto operations* for each of the following expressions, using the action and goto tables of Figure 5.7.

   (a) `var`
   (b) `(var)`
   (c) `var + var * var`
   (d) `(var*var) + var`
   (e) `(var * var`

5.3 SableCC

For many grammars, the LR parsing tables can be generated automatically from the grammar. There are several software systems designed to generate a parser automatically from specifications (as mentioned in section 2.4). In this chapter we will be using software developed at McGill University, called SableCC.

5.3.1 Overview of SableCC

SableCC is described well in the thesis of its creator, Etienne Gagnon (see `www.sablecc.org`). The user of SableCC prepares a grammar file, as described in section 2.4) as well as two java classes: Translation and Compiler. These are stored in the same directory as the parser, lexer, node, and analysis directories. Using the grammar file as input, SableCC generates java code the purpose of which is to compile source code as specified in the grammar file. SableCC generates a lexer and a parser which will produce an abstract syntax tree as output. If the user wishes to implement actions with the parser, the actions are specified in the Translation class. An overview of this software system is presented in figure 5.9.

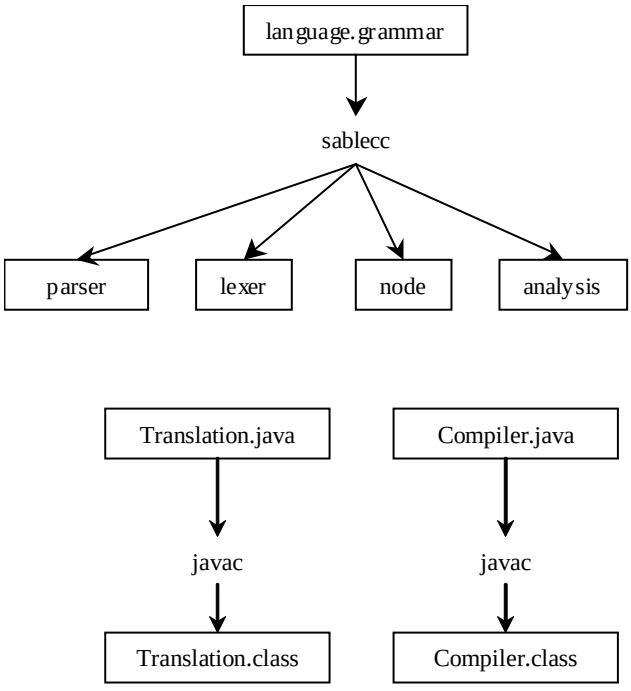

Figure 5.9 Generation and Compilation of a Compiler Using SableCC

5.3.2 Structure of the SableCC Source Files

The input to SableCC is called a grammar file. This file contains the specifications for lexical tokens, as well as syntactic structures (statements, expressions, ...) of the language for which we wish to construct a compiler. Neither actions nor attributes are included in the grammar file. There are six sections in the grammar file:

1. Package
2. Helpers
3. States
4. Tokens
5. Ignored Tokens
6. Productions

The first four sections were described in section 2.4. The Ignored Tokens section gives you an opportunity to specify tokens that should be ignored by the parser (typically white space and comments). The Productions section contains the grammar rules for the language being defined. This is where syntactic structures such as statements, expressions, etc. are defined. Each definition consists of the name of the syntactic type being defined (i.e. a nonterminal), an equal sign, an EBNF definition, and a semicolon to terminate the production. As mentioned in section 2.4, all names in this grammar file must be lower case.

An example of a production defining a while statement is shown below (`l_par` and `r_par` are left parenthesis and right parenthesis tokens, respectively):

```
stmt =  while l_par bool_expr r_par stmt ;
```

Note that the semicolon at the end is not the token for a semicolon, but a terminator for the `stmt` rule. Productions may use EBNF-like constructs. If `x` is any grammar symbol, then:

```
x?                  // An optional x (0 or 1 occurrences of x)
x*                  // 0 or more occurrences of x
x+                  // 1 or more occurrences of x
```

Alternative definitions, using |, are also permitted. However, alternatives must be labeled with names enclosed in braces. The following defines an argument list as 1 or more identifiers, separated with commas:

```
arg_list =  {single} identifier
         | {multiple} identifier ( comma identifier ) +
         ;
```

The names `single` and `multiple` enable the user to refer to one of these alternatives when applying actions in the Translation class. Labels must also be used when two identical names appear in a grammar rule. Each item label must be enclosed in brackets, and followed by a colon:

```
for_stmt =   for l_par [init]: assign_expr semi bool_expr
             semi [incr]: assign_expr r_par stmt ;
```

Since there are two occurrences of `assign_expr` in the above definition of a `for` statement, they must be labeled. The first is labeled `init`, and the second is labeled `incr`.

5.3.3 An Example Using SableCC

The purpose of this example is to translate infix expressions involving addition, subtraction, multiplication, and division into postfix expressions, in which the operations are placed after both operands. Note that parentheses are never needed in postfix expressions, as shown in the following examples:

Infix	Postfix
`2 + 3 * 4`	`2 3 4 * +`
`2 * 3 + 4`	`2 3 * 4 +`
`( 2 + 3 ) * 4`	`2 3 + 4 *`
`2 + 3 * ( 8 - 4 ) - 2`	`2 3 8 4 - * + 2 -`

There are four sections in the grammar file for this program. The first section specifies that the package name is 'postfix'. All java software for this program will be part of this package. The second section defines the tokens to be used. No Helpers are needed, since the numbers are simple whole numbers, specified as one or more digits. The third section specifies that blank (white space) tokens are to be ignored; this includes tab characters and newline characters. Thus the user may input infix expressions in free format. The fourth section, called Productions, defines the syntax of infix expressions. It is similar to the grammar given in section 3.1, but includes subtraction and division operations. Note that each alternative definition for a syntactic type must have a label in braces. The grammar file is shown below:

```
Package postfix;

Tokens
 number = ['0'..'9']+;
 plus =    '+';
 minus =   '-';
 mult =    '*';
 div =     '/';
 l_par =    '(';
 r_par =    ')';
```

```
 blank =   (' ' | 10 | 13 | 9)+ ;
 semi =    ';' ;

Ignored Tokens
 blank;

Productions
 expr =
        {term}   term |
        {plus}   expr plus term |
        {minus}  expr minus term
        ;
 term =
        {factor}  factor |
        {mult}    term mult factor |
        {div}     term div factor
        ;
 factor =
        {number}  number |
        {paren}   l_par expr r_par
        ;
```

Now we wish to include actions which will put out postfix expressions. SableCC will produce parser software which will create an abstract syntax tree for a particular infix expression, using the given grammar. SableCC will also produce a class called `DepthFirstAdapter`, which has methods capable of visiting every node in the syntax tree. In order to implement actions, all we need to do is extend `DepthFirstAdapter` (the extended class is usually called `Translation`), and override methods corresponding to rules (or tokens) in our grammar. For example, since our grammar contains an alternative, `Mult`, in the definition of `Term`, the `DepthFirstAdapter` class contains a method named `outAMultTerm`. It will have one parameter which is the node in the syntax tree corresponding to the `Term`. Its signature is

```
public void outAMultTerm (AMultTerm node)
```

This method will be invoked when this node in the syntax tree, and all its descendants, have been visited in a depth-first traversal. In other words, a `Term`, consisting of a `Term`, a `mult` (i.e. a '*'), and a `Factor` have been successfully scanned. To include an action for this rule, all we need to do is override the `outAMultTerm` method in our extended class (`Translation`). In our case we want to print out a '+' after scanning a '+' and both of its operands. This is done by overriding the `outAPlusExpr` method. When do we print out a number? This is done when a number is seen in the `{number}` alternative of the definition of `factor`. Therefore, override the method `outANumberFactor`. In this method all we need to do is print the parameter node (all

nodes have `toString()` methods, and therefore can be printed). The Translation class is shown below:

```
package postfix;
import postfix.analysis.*;  // needed for DepthFirstAdapter
import postfix.node.*;      // needed for syntax tree nodes.

class Translation extends DepthFirstAdapter
{
public void outAPlusExpr(APlusExpr node)
{// out of alternative {plus} in expr, we print the plus.
   System.out.print ( " + ");
}

public void outAMinusExpr(AMinusExpr node)
{// out of alternative {minus} in expr, we print the minus.
   System.out.print ( " - ");
}

public void outAMultTerm(AMultTerm node)
{// out of alternative {mult} in term, we print the minus.
   System.out.print ( " * ");
}

public void outADivTerm(ADivTerm node)
{// out of alternative {div} in term, we print the minus.
   System.out.print ( " / ");
}

public void outANumberFactor (ANumberFactor node)
//  out of alternative {number} in factor, we print the
number.
{   System.out.print (node + " "); }
}
```

There are other methods in the `DepthFirstAdapter` class which may also be overridden in the `Translation` class, but which were not needed for this example. They include the following:

- There is an 'in' method for each alternative, which is invoked when a node is about to be visited. In our example, this would include the method

  `public void inAMultTerm (AMultTerm node)`

- There is a 'case' method for each alternative. This is the method that visits all the descendants of a node, and it is not normally necessary to override this method. An example would be

```
public void caseAMultTerm (AMultTerm node)
```

- There is also a 'case' method for each token; the token name is prefixed with a 'T' as shown below:

```
public void caseTNumber (TNumber token)
{     // action for number tokens   }
```

An important problem to be addressed is how to invoke an action in the middle of a rule (an embedded action). Consider the `while` statement definition:

```
while_stmt =  {while} while l_par bool_expr r_par stmt ;
```

Suppose we wish to put out a `LBL` atom after the `while` keyword token is seen. There are two ways to do this. The first way is to rewrite the grammar, and include a new nonterminal for this purpose (here we call it `while_token`):

```
while_stmt =   {while} while_token l_par
               bool_expr r_par stmt ;
while_token =  while ;
```

Now the method to be overridden could be:

```
public void outAWhileToken (AWhileToken node)
{  System.out.println ("LBL") ;  }  // put out a LBL atom.
```

The other way to solve this problem would be to leave the grammar as is and override the case method for this alternative. The case methods have not been explained in full detail, but all the user needs to do is to copy the case method from `DepthFirstAdapter`, and add the action at the appropriate place. In this example it would be:

```
public void caseAWhileStmt (AWhileStmt node)
    {   inAWhileStmt(node);
        if(node.getWhile() != null)
        {  node.getWhile().apply(this)  }
      System.out.println ("LBL");   // embedded action
       if(node.getLPar() != null)
        {  node.getLPar().apply(this);  }
        if(node.getBoolExpr() != null)
        {   node.getBoolExpr().apply(this);   }
        if(node.getRPar() != null)
        {   node.getRPar().apply(this);  }
        if (node.getStmt() != null)
      {  node.getStmt().apply (this) ;  }
```

```
        outAWhileStmt (node);
    }
```

The student may have noticed that SableCC tends to alter names that were included in the grammar. This is done to prevent ambiguities. For example, `l_par` becomes `LPar`, and `bool_expr` becomes `BoolExpr`.

In addition to a Translation class, we also need a Compiler class. This is the class which contains the main method, which invokes the parser. The Compiler class is shown below:

```
package postfix;
import postfix.parser.*;
import postfix.lexer.*;
import postfix.node.*;
import java.io.*;

public class Compiler
{
public static void main(String[] arguments)
{ try
  { System.out.println("Type one expression");

    // Create a Parser instance.
    Parser p = new Parser
        ( new Lexer
          ( new PushbackReader
            ( new InputStreamReader(System.in), 1024)));

    // Parse the input.
    Start tree = p.parse();

    // Apply the translation.
    tree.apply(new Translation());

    System.out.println();
  }
  catch(Exception e)
    { System.out.println(e.getMessage()); }
}
}
```

This completes our example on translating infix expressions to postfix. The source code is available at `http://www.rowan.edu/~bergmann/books`.

In section 2.3 we discussed the use of hash tables in lexical analysis. Here again we make use of hash tables, this time using the Java class Hashtable (from java.util). This is a general storage-lookup table for any kind of objects. Use the put method to store an

object, with a key: `void put (Object key, Object value);` and use the get method to retrieve a value from the table: `Object get (Object key)`.

Sample Problem 5.3

Use SableCC to translate infix expressions involving addition, subtraction, multiplication, and division of whole numbers into atoms. Assume that each number is stored in a temporary memory location when it is encountered. For example, the following infix expression:

```
34 + 23 * 8 - 4
```

should produce the list of atoms:

```
MUL T2 T3 T4
ADD T1 T4 T5
SUB T5 T6 T7
```

Here it is assumed that 34 is stored in T1, 23 is stored in T2, 8 is stored in T3, and 4 is stored in T6.

Solution:

Since we are again dealing with infix expressions, the grammar given in this section may be reused. Simply change the package name to `exprs`.

The Compiler class may also be reused as is. All we need to do is rewrite the Translation class.

To solve this problem we will need to allocate memory locations for sub-expressions and remember where they are. For this purpose we use a java Hashtable. A Hashtable stores key-value pairs, where the key may be any object, and the value may be any object. Once a value has been stored (with a put method), it can be retrieved with its key (using the get method). In our Hashtable, the key will always be a `Node`, and the value will always be an `Integer`. The Translation class is shown below:

```
package exprs;
import exprs.analysis.*;
import exprs.node.*;
import java.util.*;           // for Hashtable
import java.io.*;

class Translation extends DepthFirstAdapter
{
// Use a Hashtable to store the memory locations for exprs
// Any node may be a key, its memory location will be the
//     value, in a (key,value) pair.

Hashtable hash = new Hashtable();

public void caseTNumber(TNumber node)
// Allocate memory loc for this node, and put it into
//  the hash table.
{ hash.put (node, alloc()); }

public void outATermExpr (ATermExpr node)
{  // Attribute of the expr same as the term
    hash.put (node, hash.get(node.getTerm()));
}

public void outAPlusExpr(APlusExpr node)
{// out of alternative {plus} in Expr, we generate an
 //   ADD atom.
   Integer i = alloc();
   hash.put (node, i);
   atom ("ADD", (Integer)hash.get(node.getExpr()),
      (Integer)hash.get(node.getTerm()), i);
}

public void outAMinusExpr(AMinusExpr node)
{// out of alternative {minus} in Expr,
 //   generate a minus atom.
    Integer i = alloc();
    hash.put (node, i);
    atom ("SUB", (Integer)hash.get(node.getExpr()),
      (Integer)hash.get(node.getTerm()), i);
}

public void outAFactorTerm (AFactorTerm node)
{ //  Attribute of the term same as the factor
        hash.put (node, hash.get(node.getFactor()));
```

```
 }

public void outAMultTerm(AMultTerm node)
{// out of alternative {mult} in Factor, generate a mult
 //  atom.
    Integer i = alloc();
    hash.put (node, i);
    atom ("MUL", (Integer)hash.get(node.getTerm()),
                 (Integer) hash.get(node.getFactor()) , i);
}

public void outADivTerm(ADivTerm node)
{// out of alternative {div} in Factor,
 // generate a div atom.
    Integer i = alloc();
    hash.put (node, i);
    atom ("DIV", (Integer) hash.get(node.getTerm()),
                 (Integer) hash.get(node.getFactor()), i);
}

public void outANumberFactor (ANumberFactor node)
{   hash.put (node, hash.get (node.getNumber())); }

public void outAParenFactor (AParenFactor node)
{   hash.put (node, hash.get (node.getExpr())); }

void atom (String atomClass, Integer left, Integer right,
Integer result)
{   System.out.println (atomClass +  " T" + left + " T"  +
            right +  " T" + result);
}

static int avail = 0;

Integer alloc()
{ return new Integer (++avail); }

}
```

Exercises 5.3

1. Which of the following input strings would cause this SableCC program to produce a *syntax error* message?

```
Tokens
 a = 'a';
 b = 'b';
 c = 'c';
 newline = [10 + 13];
Productions
  line = s newline ;
  s =       {a1} a s b
          | {a2} b w c
          ;
  w =       {a1} b w b
          | {a2} a c
          ;
```

(a) `bacc` (b) `ab` (c) `abbacbcb`
(d) `bbacbc` (e) `bbacbb`

2 . Using the SableCC program from problem 1, show the *output* produced by each of the input strings given in Problem 1, using the Translation class shown below.

```
package ex5_3;
import ex5_3.analysis.*;
import ex5_3.node.*;
import java.util.*;
import java.io.*;

class Translation extends DepthFirstAdapter
{
```

```
public void outAA1S (AA1S node)
{   System.out.println ("rule 1"); }

public void outAA2S (AA2S node)
{   System.out.println ("rule 2"); }

public void outAA1W (AA1W node)
{   System.out.println ("rule 3"); }

public void outAA2W (AA2W node)
{   System.out.println ("rule 4"); }

}
```

3. A ***Sexpr*** is an atom or a pair of Sexprs enclosed in parentheses and separated with a period. For example, if `A, B, C, ...Z` and `NIL` are all atoms, then the following are examples of Sexprs:

```
A
(A.B)
((A.B).(B.C))
(A.(B.(C.NIL)))
```

A ***List*** is a special kind of Sexpr. A List is the atom `NIL` or a List is a dotted pair of Sexprs in which the first part is an atom or a List and the second part is a List. The following are examples of lists:

```
NIL
(A.NIL)
((A.NIL).NIL)
((A.NIL).(B.NIL))
(A.(B.(C.NIL)))
```

(a) Show a *SableCC grammar* that defines a *Sexpr*.

(b) Show a *SableCC grammar* that defines a *List*.

(c) Add a Translation class to your answer to part (b) so that it will print out the total number of atoms in a List. For example:

```
((A.NIL).(B.(C.NIL)))
5 atoms
```

4. Use SableCC to implement a *syntax checker* for a typical database command language. Your syntax checker should handle at least the following kinds of commands:

```
RETRIEVE  employee_file
PRINT
DISPLAY FOR salary >= 1000000
PRINT FOR "SMITH" = lastname
```

5. The following SableCC grammar and Translation class are designed to implement a simple desk calculator with the standard four arithmetic functions (it uses floating-point arithmetic only). When compiled and run, the program will evaluate a list of arithmetic expressions, one per line, and print the results. For example:

```
2+3.2e-2
2+3*5/2
(2+3)*5/2
16/(2*3 - 6*1.0)
     2.032
     9.5
     12.5
     infinity
```

Unfortunately, the grammar and Java code shown below are incorrect. There are four mistakes, some of which are syntactic errors in the grammar; some of which are syntactic Java errors; some of which cause run-time errors; and some of which don't produce any error messages, but do produce incorrect output. Find and correct all four mistakes. If possible, use a computer to help debug these programs.

The grammar, `exprs.grammar` is shown below:

```
Package exprs;

Helpers
 digits = ['0'..'9']+ ;
 exp =    ['e' + 'E'] ['+' + '-']? digits ;
Tokens
 number = digits '.'? digits? exp? ;
 plus =    '+';
 minus =   '-';
 mult =    '*';
 div =     '/';
 l_par =    '(';
 r_par =    ')';
 newline = [10 + 13] ;
 blank =   (' ' | '\t')+;
 semi =    ';' ;

Ignored Tokens
 blank;

Productions
 exprs =    expr newline
          |  exprs embed
          ;
 embed = expr newline;
 expr =
          {term}   term |
          {plus}   expr plus term |
          {minus}  expr minus term
          ;
 term =
```

```
            {factor}  factor |
            {mult}    term mult factor |
            {div}     term div factor |
            ;
      factor =
            {number}  number |
            {paren}   l_par expr r_par
            ;
```

The Translation class is shown below:

```
package exprs;
import exprs.analysis.*;
import exprs.node.*;
import java.util.*;

class Translation extends DepthFirstAdapter
{
Hashtable hash = new Hashtable();       // store expr values

public void outAE1Exprs (AE1Exprs node)
{  System.out.println ("   " + getVal (node.getExpr())); }

public void outAEmbed (AEmbed node)
{  System.out.println ("   " + getVal (node.getExpr())); }

public void caseTNumber(TNumber node)
{ hash.put (node, new Double (node.toString())) ; }

public void outAPlusExpr(APlusExpr node)
{// out of alternative {plus} in Expr, we add the
 // expr and the term
   hash.put (node, new Double (getPrim (node.getExpr())
               + getPrim(node.getTerm())));
}

public void outAMinusExpr(AMinusExpr node)
{// out of alternative {minus} in Expr, subtract the term
 //  from the expr
   hash.put (node, new Double (getPrim(node.getExpr())
               - getPrim(node.getTerm())));
}
```

```
public void outAFactorTerm (AFactorTerm node)
{ //  Value of the term same as the factor
   hash.put (node, getVal(node.getFactor())) ;
}

public void outAMultTerm(AMultTerm node)
{// out of alternative {mult} in Factor, multiply the term
 // by the factor
   hash.put (node, new Double (getPrim(node.getTerm())
               * getPrim(node.getFactor())));
}

public void outADivTerm(ADivTerm node)
{// out of alternative {div} in Factor, divide the term by
 // the factor
   hash.put (node, new Double (getPrim(node.getTerm())
               / getPrim(node.getFactor())));
}

public void outANumberFactor (ANumberFactor node)
{   hash.put (node, getVal (node.getNumber())); }

public void outAParenFactor (AParenFactor node)
{   hash.put (node, new Double (0.0)); }

double getPrim (Node node)
{  return ((Double) hash.get (node)).doubleValue(); }

Double getVal (Node node)
{  return  hash.get (node) ; }
}
```

6. Show the *SableCC grammar* which will check for proper syntax of regular expressions over the alphabet {0,1}. Some examples are shown:

Valid	Not Valid
(0+1)*·1·1	*0
0·1·0*	(0+1)+1)
((0))	0+