

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 empty 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. $S \rightarrow S a B$

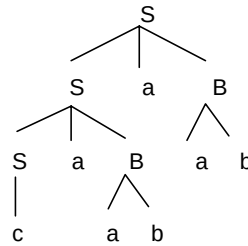


Figure 5.1 A Derivation Tree for the String **caabaab**, Using Grammar G19

2. $S \rightarrow c$
3. $B \rightarrow a b$

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:

$$\begin{aligned}
 S &\Rightarrow \underline{S a B} \Rightarrow S a \underline{a b} \Rightarrow \underline{S a B} a a b \Rightarrow S a \underline{a b} a a b \\
 &\Rightarrow \underline{c} a a b a a b
 \end{aligned}$$

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:

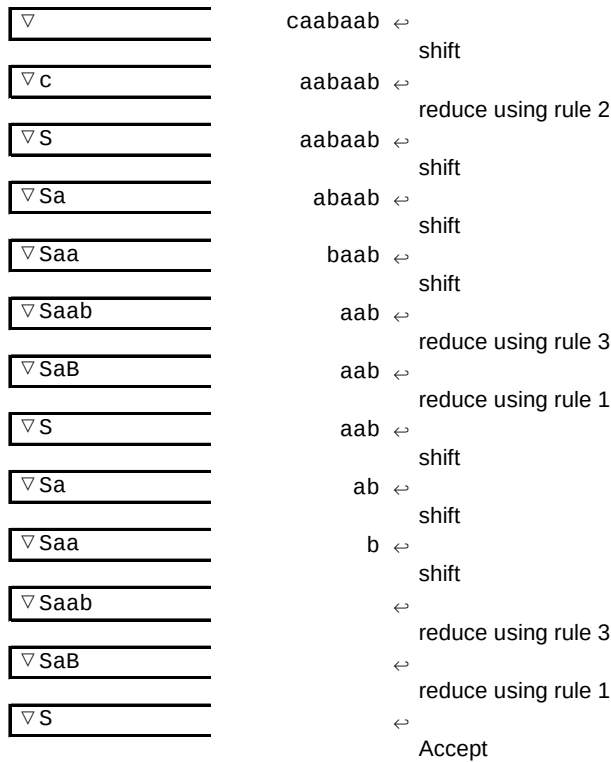


Figure 5.2 Sequence of Stack Frames Parsing caabaab Using Grammar G19

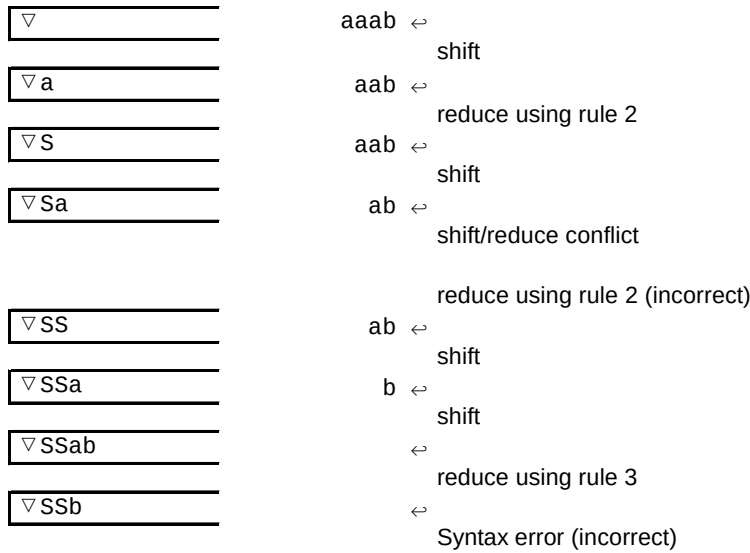


Figure 5.3 An Example of a Shift/Reduce Conflict Leading to an Incorrect Parse Using Grammar G20

G20:	G21:
1. S → S a B	1. S → S A
2. S → a	2. S → a
3. B → a b	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-

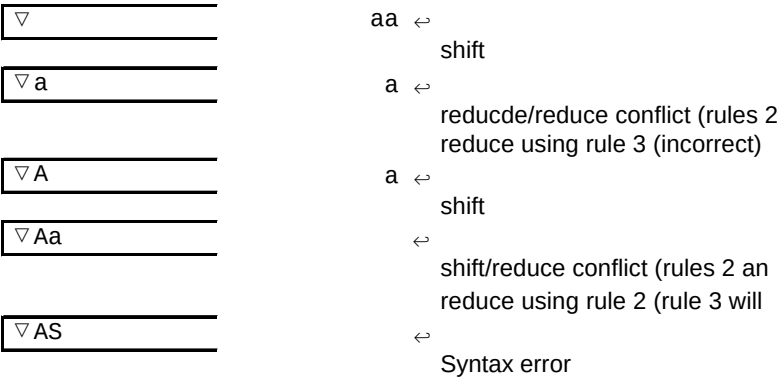


Figure 5.4 A Reduce/Reduce Conflict

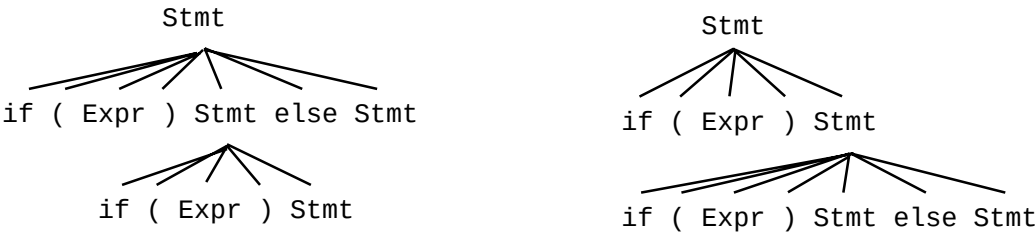


Figure 5.5 Two Derivation Trees for `if (Expr) if (Expr) Stmt else Stmt`

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` $\rightarrow$ `if (Expr) Stmt else Stmt`
- 2. `Stmt` $\rightarrow$ `if (Expr) Stmt`

The `Expr` in parentheses represents a true (non-zero) or false (zero) condition. Figure 5.5, above, shows two different derivation trees for the statement `if (Expr) if (Expr) Stmt else Stmt`. The second tree shown 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 (Expr) 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.

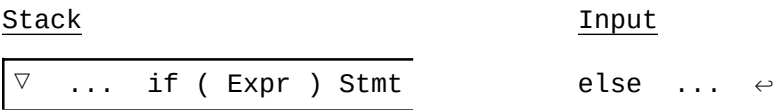


Figure 5.6 Parser Configuration Before Reading the `else` Part of an `if` Statement

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:	∇	<i>caab</i> $\leftrightarrow$
	∇c	shift
	∇S	<i>aab</i> $\leftrightarrow$
	∇Sa	reduce using rule 2
	∇Saa	<i>aab</i> $\leftrightarrow$
	$\nabla Saab$	shift
	∇SaB	<i>ab</i> $\leftrightarrow$
	∇S	shift
		<i>b</i> $\leftrightarrow$
		shift
		$\leftrightarrow$
		reduce using rule 3
		$\leftrightarrow$
		reduce using rule 1
		$\leftrightarrow$
		Accept

Exercises 5.1

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

1. $S \rightarrow S A b$
2. $S \rightarrow a c b$
3. $A \rightarrow b B c$
4. $A \rightarrow b c$
5. $B \rightarrow b a$
6. $B \rightarrow A c$

- | | | | |
|-----|---------------|-----|----------------|
| (a) | $\nabla SSAb$ | (b) | $\nabla SSbbc$ |
| (c) | $\nabla SbBc$ | (d) | $\nabla Sbbc$ |

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) acbbcbccb | |

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

1. $S \rightarrow S a b$
2. $S \rightarrow b A$
3. $A \rightarrow b b$
4. $A \rightarrow b A$
5. $A \rightarrow b b c$
6. $A \rightarrow c$

- | |
|---------------|
| (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 (Expr) if (Expr) 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. $\text{Stmt} \rightarrow \text{if (Expr) Stmt else Stmt}$
2. $\text{Stmt} \rightarrow \text{if (Expr) Stmt}$

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 method 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 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:

<u>Stack</u>	<u>Input</u>
∇ S	ab $\leftarrow$

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

<u>Stack</u>	<u>Input</u>
∇ Sa	b $\leftarrow$

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

7. $B \rightarrow Sa$

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:

<u>Stack</u>	<u>Input</u>
∇ X	b $\leftarrow$

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

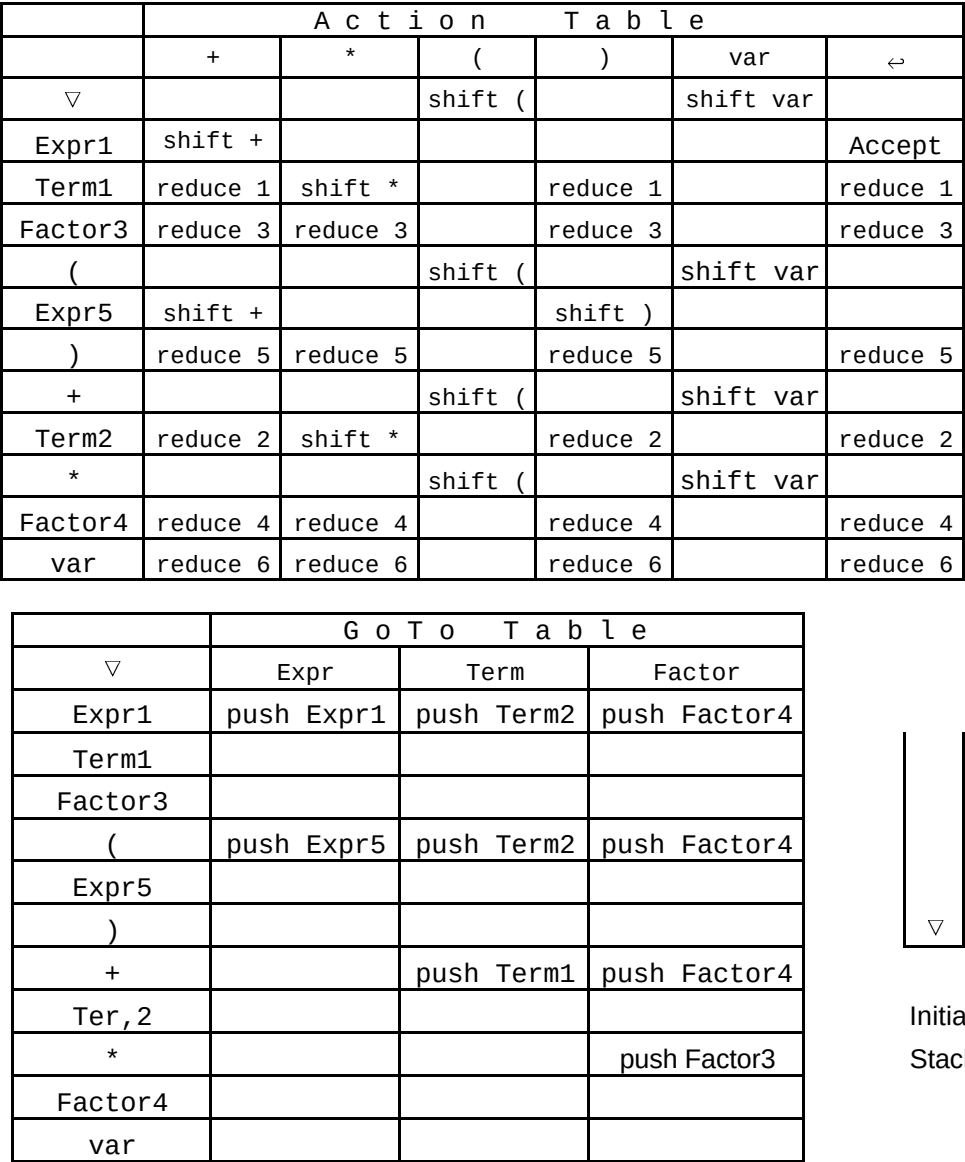


Figure 5.7 Action and Goto Tables to Parse Simple Arithmetic Expressions

being 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

Expr → Expr + Term
Expr → Term
Term → Term * Factor
Term → Factor
Factor → (Expr)
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 methods 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 method 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 methods 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 Yacc

For many grammars, the LR parsing tables can be generated automatically from the grammar. One of the most popular software systems that does this is available in the Unix programming environment; it is called **yacc** (Yet Another Compiler-Compiler). Recall from Section 1.3.4 that a **compiler-compiler** is a program which takes as input the specification of a programming language (in the form of a grammar), and produces as output a compiler for that language. By itself, yacc is really just a parser generator yielding a program which checks for syntax, but since it is possible for the user to augment it with additional features, it can be used to generate a complete compiler. An available public domain version of yacc is called **bison**. There are also personal computer versions of yacc which use Turbo Pascal as a base language instead of C.

5.3.1 Overview of Yacc

Yacc generates a C function named `yyparse()`, which is stored in a file named `y.tab.c`. This function calls a function named `yylex()` whenever it needs an input token. The `yylex()` function may be written by the user and included as part of the yacc specification, or it may be generated by the lex utility, as described in Section 2.4. A diagram showing the flow of data needed to generate and compile software is shown in Figure 5.9. It assumes that `yylex()` is being generated by lex and that there is a statement in the yacc source file to include the `yylex()` function:

```
#include "lex.yy.c"
```

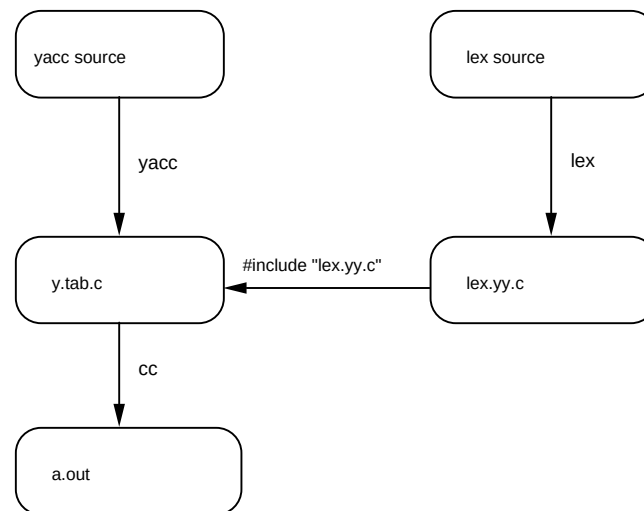


Figure 5.9 Generation and Compilation of Software Using Lex and Yacc

5.3.2 Structure of the Yacc Source File

The input to yacc is called the *yacc source file*. It consists of three parts, which are separated by the %% delimiter:

```
Declarations
%%
Rules
%%
Support Routines
```

The Declarations section contains declarations of token names, stack type, and precedence information which may be needed by yacc. It also may contain preprocessor statements (`#include` or `#define`) and declarations to be included in the output file, `y.tab.c`.

The Rules section is the grammar for the language being specified, such as Pascal. This is the most important part of the yacc source file. Each rule is of the form:

```
nonterminal:       $\alpha$       {action}
                  |  $\beta$       {action}
                  |  $\gamma$      {action}
                  .
                  .
                  .
                  ;
```

where α , β , and γ are definitions of the nonterminal. The vertical bar designates alternative definitions for a non-terminal, as in BNF. An action may be associated with each of the alternatives. This action is simply a C statement which is invoked during the parsing of an input string when the corresponding grammar rule is reduced. The rules may be written in free format, and each rule is terminated with a semicolon.

The third section of the yacc source file contains support routines, i.e., C functions which could be called from the actions in the Rules section. For example, when processing an assignment statement, it may be necessary to check that the type of the expression matches the type of the variable to which it is being assigned. This could be done with a call to a type-checking function in the third section of the yacc source file.

5.3.3 An Example Using Yacc

The purpose of this example is to translate infix expressions involving addition, subtraction, multiplication, division, and unary minus into atoms which represent the primitive operations available on most CPUs. Each atom will consist of up to four parts:

```
(Operation, Left operand, Right Operand, Result)
```

For example, if the input is $(a+b)*c$, the output would be:

```
(ADD, a, b, T1)
(MULT, T1, c, T2)
```

Whereas if the input is $-a+b*c$, the output would be:

```
(NEG, a, 0, T1)
(MUL, b, c, T2)
(ADD, T1, T2, T3)
```

Note that the unary minus operation produces a **NEG** atom which has only one operand, so the second field is ignored.

In this example, we implement the operands in an atom as character strings. In a real compiler the identifier operands would be implemented as pointers to the symbol table entry for the identifier, and the temporary results (T1, T2, etc.) would be implemented as offsets from the top of the run-time stack.

The entire yacc and lex programs for this example are shown below. The essence of the program is in the grammar, which follows the first `%`. We define the input to be a **List of Exprs**, each of which appears on a separate line and may be null. The **Expr** is then defined as we defined it in Section 3.0.3, with grammar G4, only extended to include subtraction, division, and unary operations. This grammar is clearly ambiguous, but yacc permits us to use ambiguous grammars and provides a way of resolving the ambiguity. The actions can refer to the values of the elements of the nonterminal being defined with positional notation. For example, in the following yacc definition:

```
S:    a S b C
```

the value of **a** is referred to as `$1`, **S** is `$2`, **b** is `$3`, and **C** is `$4`. The nonterminal being defined may be assigned a value; if so, it is referred to as `$$`. The value of a terminal, or token, is taken from the global variable `yylval`.

Some aspects of this yacc program need to be explained. Beginning with the first Section, the `%{` and `%}` delimiters are used to include C code in the output. They are usually used for macros (`#define`), `#include` statements, and C declarations. The macro `#define YYSTYPE string` is used to declare the type of values to be pushed onto the parse stack. In this case we wish to push strings (actually pointers to char) onto the parse stack, since we are implementing the operations and operands as simple strings. We could not have used `#define YYSTYPE char *`, however, because `YYSTYPE` is used in the resulting C program to declare several variables:

```
YYSTYPE  yyval, yyval;
```

The macro substitution would yield:

```
char * yylval, yyval;
```

which declares `yylval` to be a pointer to `char`, but `yyval` to be a `char`, which is incorrect. The use of `typedef` to define a `string` type circumvents this problem. The remainder of the inline C code, as far as the `%}`, is used by the `alloc()` function, described below. We then have some yacc declarations, the first of which, `%token`, is used to declare tokens returned by the `yylex()` function. Tokens are implemented in yacc as integers. Each character corresponds to a one byte integer token. Named tokens, such as `OPERAND` in our example, are then equated to values beginning with 256. The yacc directives `%left` and `%right` are used to declare the precedence of operations (this is how the ambiguity in the grammar is resolved). The directive `%left '+', '-'` means that these operations associate to the left (they are executed from left to right). The fact that the precedence of addition and subtraction is given before the precedence of multiplication and division, indicates that multiplication and division take precedence over addition and subtraction.

The second section of the yacc program includes the grammar for infix expressions. Each time a subexpression generates an intermediate result, we need to allocate a new temporary variable (`T1, T2, T3, . . .`). For this purpose we call the `alloc()` function, which returns the next available temporary location in the form of a string (a commercial compiler would return pointers to the locations, or offsets to a stack). Once we know where the result of the operation will be stored we can call the `atom()` function to put out an atom with the appropriate operation and operands. Each time an entire expression has been translated, we need to call the `clear()` function to return allocated storage to the system and to reset counters for the next expression.

The third section of the yacc program includes the main program, with a call to `yyparse()` to start up the translation. This section also includes functions called from our action statements in the second section. In the third section, we also include the output of lex, which is stored in the file `lex.yy.c`. This contains the `yylex()` function which scans the input and returns a lexical token each time it is called. The `yyerror()` function is called when a syntax error is encountered, and it prints the line number which caused the error. Though yacc has the capability of recovering from syntax errors (to scan for more errors), this example does not do so and merely terminates. The `alloc()` function can be called from a lex action as well as from a yacc action. It returns a pointer to an array of three characters which stores the operand in string form, whether it be an identifier (`a, b, c, . . .`) or a temporary result (`T1, T2, T3. . .`). These strings are stored in a linked list. The `clear()` function is called when an entire expression has been translated. It frees all nodes in the linked list and resets the counter for temporary locations back to 0. The `atom()` function puts out an atom to `stdout`, using the operation and operands as arguments.

The lex program is shown below:

```
%%
[a-zA-Z]      {yylval = alloc(TOKEN); return OPERAND;}
[ \t]         ;
```

```

.          return yytext[0];
\n         {lineno++; return yytext[0];}
%%

```

The yacc program is shown below:

```

%{
#include <stdio.h>
#include <alloc.h>
typedef char * string;
#define YYSTYPE string          /* type for parse stack */
#define TOKEN 1
#define TEMP 0
int temp = 0;
struct node {
    char name[3];                /* "Tn" or a,b,c,...*/
    struct node * next;          /* linked list */
};
struct node * tlist = NULL;     /* list head */
char * alloc (int type);
%}
%token OPERAND
%left '+' '-'
%left '*' '/'
%left UMINUS UPLUS
%%
List:                                /* empty list */
    | List '\n'
    | List Expr '\n' {clear ();}
    ;
Expr:
    '+' Expr %prec UPLUS  {$$ = $2;}
    | '-' Expr %prec UMINUS {$$ = alloc(TEMP);
                                atom ("NEG", $2, 0, $$); }
    | Expr '+' Expr      {$$ = alloc(TEMP);
                                atom ("ADD", $1, $3, $$); }
    | Expr '-' Expr      {$$ = alloc(TEMP);
                                atom ("SUB", $1, $3, $$); }
    | Expr '*' Expr      {$$ = alloc(TEMP);
                                atom ("MUL", $1, $3, $$); }
    | Expr '/' Expr      {$$ = alloc(TEMP);
                                atom ("DIV", $1, $3, $$); }
    | '(' Expr ')'        {$$ = $2;}
    | OPERAND             {$$ = $1;}
    ;
%%

```

```

char *programe;
int lineno = 1;
#include "lex.yy.c"

main (int argc, char *argv[])
{
    programe = argv[0];
    yyparse();
}

yyerror ( char *s)
{
    fprintf(stderr, "%s[%d]: %s\n", programe, lineno, s);
}

char * alloc (int type)
{ static struct node * last = NULL;
  struct node * ptr;
  static char t[3] = " ";

  ptr = (struct node *) malloc (sizeof (struct node));
  ptr -> next = NULL;
  if (last!=NULL) {last->next = ptr; last = ptr;}
  else tlist = last = ptr;
  if (type==TEMP)
  {
      t[0] = 'T';
      t[1] = '0' + temp++;
      strcpy (last->name, t);
  }
  else strcpy (last->name, yytext);
  return (char *) last;
}

clear ()
/* free up allocated memory, reset counters for next Expr */
{ struct node * ptr;
  while (tlist)
  { ptr = tlist->next;
    free (tlist);
    tlist = ptr;
  }
  temp = 0;
}

```

```

atom (char * operation, char * operand1,
     char * operand2, char * result)
/* put out an atom. */
{
    if (operand2) /* NEG has only one operand and result */
        printf ("\t%s %s %s %s\n", operation, operand1,
                operand2, result);
    else printf ("\t%s %s %s\n", operation, operand1,
                result);
}

```

In sample Problem 5.3 we attempt to use yacc to build simple expression trees. In this case there are two types which can be pushed on the stack — operators and pointers to nodes in the expression tree. Therefore we need to tell yacc, with a `%union` declaration, that either type may be pushed on the stack:

```

%union {
    struct node * ptr;
    char op;
}

```

It is then possible to declare a nonterminal to be one of these types, as in the declaration `%type <ptr> Expr` which shows that each `Expr` is a pointer to the subtree which defines that `Expr`. When referring to `yylval`, either in `lex` or `yacc`, we now have to specify its type, as `yylval.op`, indicating that it is a single character.

Sample Problem 5.3

Write a yacc program to build simple expression trees. Assume that the operands may be only single letters or single digits and that the operations are addition, multiplication, subtraction, and division, with the usual precedence rules. The `yylex()` function may be generated by `lex` or written directly in C. The input should be a single infix expression on each line. After the expression tree is built, dump it to `stdout` in prefix order. Example:

```

(4+a) * b
    * + 4 a b
4 + a * b
    + 4 * a b

```

Solution:

The `lex` program is:

```

%%
[a-zA-Z0-9] {yylval.op = yytext[0]; return OPERAND;}
[ \t]      ;

```

```

.                return yytext[0];
\n              {lineno++; return yytext[0];}
%%

```

The yacc program is:

```

/* this builds simple expr trees. */
%{
#include <stdio.h>
struct node {
    struct node * left;
    char op;
    struct node * right;
};
%}
%union {
    struct node * ptr;
    char op;
}
%token OPERAND
%type <ptr> Expr
%left '+' '-'
%left '*' '/'
%%
List:
    | List '\n'
    | List Expr '\n' {printf ("\t");
                      prefix ($2); /* dump expr
                                   tree */
                      printf ("\n");}
;
Expr:
    Expr '+' Expr    {$$ = alloc ($1, '+', $3);}
    | Expr '-' Expr  {$$ = alloc($1, '-', $3);}
    | Expr '*' Expr   {$$ = alloc($1, '*', $3);}
    | Expr '/' Expr   {$$ = alloc($1, '/', $3);}
    | '(' Expr ')'    {$$ = $2;}
    | OPERAND         {$$ = alloc(NULL, yylval.op,
                                   NULL);}
;
%%
char *progrname;
int lineno = 0;
#include "lex.yy.c"

main (int argc, char *argv[])

```

```

{
    progvname = argv[0];
    yyparse();                      /* call the parser */
}

yyerror (char *s)
{
    fprintf(stderr, "%s[%d]: %s\n", progvname, lineno, s);
}

struct node * alloc (struct node * left, char op,
                    struct node * right)
/* allocate a node in the expression tree. Return a pointer
   to the new node */
{ struct node * ptr;

    ptr = (struct node *) malloc (sizeof (struct node));
    ptr->left = left;
    ptr->right = right;
    ptr->op = op;

    return ptr;
}

prefix (struct node * root)
/* print out the exptree in prefix order */
{
    printf ("%c ", root->op);
    if (root->left!=NULL)
        { prefix (root->left);          /* dump left subtree */
          free (root->left); }          /* release memory */
    if (root->right!=NULL)
        { prefix (root->right);         /* dump right subtree */
          free (root->right); }         /* release memory */
    free (root);
}

```

5.3.4 Other Yacc Constructs

We have demonstrated most of the important features of yacc, and the student may find them sufficient to implement parsers for fairly interesting languages, such as the language of arithmetic expressions. However, there are some more advanced constructs which will be needed to generate the parser for MiniC.

The first such advanced feature of yacc has to do with the way yacc handles *embedded actions*. An embedded action is one which is not at the end of a grammar rule:

```
Nt:  a b {action 1} c d {action 2} ;
```

As we discussed in Section 5.3.3, `action 1` is the embedded action and is counted as one of the items in the rule (\$3 in the above example), but we will now need to understand how yacc handles embedded actions. The above rule is actually transformed by yacc into two rules, as Yacc makes up a "fake" nonterminal (which is defined as the null string) for the embedded action. The example, above, would be converted to something like the following:

```
Nt:  a b Fake c d {action 2} ;
Fake: /* null definition */ {action 1} ;
```

Normally, the user need not be aware of this transformation, and everything works fine. However, consider the following example, in which the non-embedded action makes use of a value computed in the embedded action:

```
Nt:  a b { $$ = value;} c d {printf ("%d", $$);} ;
```

In this example we wish to print the value assigned to `$$` in the embedded action. Yacc would convert this rule to two rules:

```
Nt:  a b Fake c d {printf ("%d", $$);} ;
Fake:      { $$ = value;} ;
```

This will not work, since the `$$` referred to in the definition of `Nt` is not the same as the `$$` referred to in the definition of `Fake`. However, we can fix the problem by referring to the value of the embedded action itself:

```
Nt:  a b { $$ = value;} c d {printf ("%d", $3);} ;
```

which works because yacc transforms it into the following two rules:

```
Nt:  a b Fake c d {printf ("%d", $3);} ;
Fake:      { $$ = value;} ;
```

The fake nonterminals are assigned numbers by yacc, so if you should see error messages referring to nonterminals, such as `$$23`, for example, it is probably a nonterminal that yacc made up to represent an embedded action.

If the parser stack has been declared to be a union of several types, the type of the embedded action can be specified by placing the type between the `$` and the item number. For example, if the type of the embedded action in the above action were labels, we would refer to it as `$<labels>3`.

The other advanced feature of yacc, which may be needed in constructing compilers, has to do with *inherited attributes*. Up to this point, all values that have been

passed from one rule to another (via an assignment to \$\$) have been passed in a direction corresponding to an upward direction in the derivation tree. As we saw in Section 4.6, these values are called *synthesized attributes*. We now wish to consider the problem of passing values between rules in a direction which corresponds to a downward direction in the derivation tree; i.e., we wish to use inherited attributes in our yacc grammar.

Suppose, for example, that the source language includes a simplified Switch statement, defined as follows:

```
SwitchStmt: switch ( Expr ) { CaseList }
           ;
CaseList:   case NUM : StmtList
           | CaseList case NUM : StmtList
```

We will need to put out TST atoms which compare the Expr in the SwitchStmt with each of the NUMs in the CaseList, so an attribute representing the runtime location of the Expr must be passed (down the tree) from SwitchStmt to the CaseLists, hence the need for inherited attributes. When writing actions in the definition of CaseList, we can assume that the preceding items in the SwitchStmt are on the parser stack, and they can be referred to with negative item numbers as follows:

```
SWITCH ( Expr ) { CaseList }
$-4    $-3 $-2  $-1 $0
```

Thus the action for CaseList might be:

```
CaseList: case NUM ':' {TST atom which compares $-2 with
                    $2} StmtList
          | CaseList case NUM ':' {TST atom which compares $-2
                    with $3} StmtList
```

– in which \$-2 represents the location of the result of the Expr which is to be compared with a NUM in the CaseList. These and other advanced features of yacc are explained in fine detail in Levine [1992].

Exercises 5.3

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

```
/* yacc program */
%%
line:      s '\n'
          ;
```

```

s:   'a' s 'b'
    | 'b' a 'c'
    ;
a:   'b' a 'b'
    | 'a' 'c'
    ;

%%
#include "lex.yy.c"
main ()
{ yyparse();}

/* lex program */
%%
.   return yytext[0];
\n  return yytext[0];

```

- (a) bacc (b) ab (c) abbacbc
 (d) bbacbc (e) bbacbb

2. Show the *output* produced by each of the input strings given in Problem 1, using the yacc and lex programs shown below.

```

/* yacc program */
%%
line:   s '\n'
        ;
s:   'a' s 'b' {printf ("rule 1\n");}
    | 'b' a 'c' {printf ("rule 2\n");}
    ;
a:   'b' a 'b' {printf ("rule 3\n");}
    | 'a' 'c' {printf ("rule 4\n");}
    ;

%%
#include "lex.yy.c"
main ()
{ yyparse();}

```

```

/* lex program */
%%
.    return yytext[0];
\n   return yytext[0];
%%

```

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*, 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 *yacc grammar* (not a complete yacc program, but just the part after the first `%%`) that defines a *Sexpr*. Assume that the `yyllex()` function returns either *ATOM*, `( , )`, or `.`, where *ATOM* represents any atom.
- (b) Show a *yacc grammar* (again, not a complete yacc program) that defines a *List*. Assume that the `yyllex()` function returns either *ATOM*, *NIL*, `( , )`, or `.`, where *ATOM* represents any atom other than *NIL*.
- (c) Add *actions* 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 yacc and lex 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 lex and yacc programs 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
    2.032000
2+3*5/2
    9.500000
(2+3)*5/2
    12.500000
16/(2*3 - 6*1.0)
    division by zero
```

Unfortunately, the program as shown below does not work. It contains four mistakes, some of which are syntactic lex or yacc errors; some of which are syntactic C language 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.

```
/* lex program */
%%
[0-9]+\.[0-9]*([eE][+-]?[0-9]+)? {sscanf
                                   (yytext,"%f",&yyval);
                                   return NUM; }

[      ]                               ;
```

```

"\n"                return yytext[0];
.                   return yytext[0];
%%

/* yacc program */
%token NUM
%{
#define YYSTYPE float
%}
%left "+" "-"
%left "*" "/"
list:
    | list expr '\n'    {printf ("%f\n",
                                $2);}
;
expr:      expr '+' expr    {$$ = $1 +
                                $3;}
    | expr '-' expr        {$$ = $1 -
                                $3;}
    | expr '*' expr        {$$ = $1 *
                                $3;}
    | expr '/' expr        {if ($3=0)
                                yyerror ("division by zero\n");
                                else $$ = $1 / $3;}
    | '(' expr ')'          {$$ = $1;}
    | NUM                   {$$ = $1;}
;
%%
#include "lex.yy.c"

yyerror (char *s)
{ printf ("%s: %s\n", s, yytext); }

```

6. Show lex and yacc programs which will check for proper syntax of regular expressions over the alphabet $\{0,1\}$.