CS39003 Compilers Laboratory
Autumn 2025

A short YACC tutorial

Syntax Analysis

- A high-level programming language is specified by a context-free grammar.
- The terminal symbols are the tokens.
- Non-terminal symbols are associated with productions (or rules).
- A lexical analyzer identifies the tokens from an input program.
- A **syntax analyzer** (or a **parser**) tries to fit the grammar rules with the sequence of tokens.
- Lex (or flex) is the lexical analyzer.
- Yacc (or bison) is the syntax analyzer.
- Yacc uses LALR(1) parsing by default.
- Shift-reduce and reduce-reduce conflicts can be identified by yacc.
- Some conflicts and ambiguities can be removed by special yacc constructs.

Using lex and yacc Together

- You need to write both a lex file (extension `.l`) and a yacc file (extension `.y`).
- The parsing function is

```
yyparse();
```

- This function will keep on calling `yylex()` to get the tokens one after another.
- For every token (not to be ignored), lex should return a token type.
- A token type may be a character (0–255) or a user-defined name (257–).
- There may be multiple terminal and non-terminal symbols on the right side of a production.
- Yacc names these as `$1`, `$2`, `$3`, and so on.
- The left side of the production (a non-terminal, has nothing to do with lex) is denoted by `$$`.
- The lex variable `yytext` cannot simultaneously match all the tokens on the right side.
- Before returning a token type, each lex action should set a global yacc variable `yylval` (using the current `yytext`).
- Yacc action against a production combines `$1`, `$2`, `$3`, . . . to `$$`. This `$$` (if not the root of the parse tree) will be some `$i` for the reduction at the parent node of the parse tree.

Using lex and yacc Together

- Compile the yacc file with the `-d` flag. This creates two files `y.tab.h` and `y.tab.c`.

```
yacc -d example.y
```

- `y.tab.h` contains the necessary token definitions, and `y.tab.c` contains the code for the LALR(1) parser.
- In order that lex can return correct token numbers (and also the types of `yylval`), your lex file must contain the following inclusion.

```
#include "y.tab.h"
```

- Now, run lex to generate `lex.yy.c`.

```
lex example.l
```

- In your C/C++ file containing `main()`, make these inclusions.

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

- You may combine as follows. This uses a default `main()` function to call `yyparse()` and exit.

```
gcc -o example y.tab.c lex.yy.c -ly -ll
```

Structure of a Yacc Program

%{
C header to go verbatim to `y.tab.c`
%}

Definitions for the parser:

- *— Define the tokens*
- *— Set up the types of terminal and non-terminal symbols*
- *— Set operator precedence and associativity*

%%

Productions for the non-terminal symbols
Actions for each reduction step

%%

User code section (*may contain the* `main()` *function*)

Example: Grammar for Expressions

We work with the following unambiguous grammar for arithmetic expressions. Only integer constants (signed) are used as lowest-level operands.

$$
\begin{aligned}
E &\rightarrow E + T \mid E - T \mid T \\
T &\rightarrow T * F \mid T / F \mid T \,\%\, F \mid F \\
F &\rightarrow \text{NUM} \mid (E) \mid -(E)
\end{aligned}
$$

Our input file contains a sequence of expressions separated by new lines. We add the following productions for lines L and the program P.

$$
\begin{aligned}
L &\rightarrow E \,\backslash n \\
P &\rightarrow P\, L \mid L
\end{aligned}
$$

The start symbol is P.

The Lex File expr.l

```
%{
#include "y.tab.h"
%}

ws      [ \t]+
ows     [ \t]*
dgt     [0-9]
num     [+-]?{dgt}+
op      [+\-*/%]
punc    [()]

%%

{ws}            { }
{num}           { yylval = atoi(yytext); return NUM; }
{op}            { return yytext[0]; }
{punc}          { return yytext[0]; }
\\{ows}\n       { }
\n              { return '\n'; }
.               { fprintf(stderr, "    *** Invalid character '%c'\n", yytext[0]); }

%%

int yywrap ( ) { return 1; }
```

The Lex File `expr.l`

- Note the inclusion of `y.tab.h` in the header.
- White spaces are ignored.
- A number is a signed integer, so `yytext` (a string) is converted to an `int` using `atoi`.
  - This converted integer sets the `yylval` to be used by the parser.
  - The token type `NUM` is defined in `y.tab.h` by yacc.
  - Yacc will take a parsing decision based on the token type `NUM`. The value of the token will be accessed by yacc as `$i`, where `i` is the position (one-based) of `NUM` on the right side of the production.
- Operators and punctuation symbols are single characters. The corresponding ASCII value is returned. We do not need to set the `yylval` here (there is no harm in doing so anyway), because the exact symbol can be identified by the type of the token.
- The new-line character acts as the end-of-expression marker. If a line ends with a backslash, then the next line will be treated as the continuation of the current line. Lex will ignore the new-line character in this case.

Example 1: Yacc File with Default Behavior

- The only non-character token to be used by lex is `NUM`. This is defined by the `%token` construct in the Definitions Section of the yacc file.
- For every non-character terminal symbol (token), this construct is to be used.
- There is no need to declare all the non-terminal symbols.
- The default data type for each grammar symbol (terminal or non-terminal) is `int`.
- If we use other data types, then the type of each terminal or non-terminal symbol needs to be specified (see Example 3).
- We write no `main()` function in this example.
- We compile `y.tab.c` and `yy.lex.c` with the `-ly` and `-ll` flags.
- A default `main()` function will be introduced in the final `a.out`.
- The head of the first production in the Rules Section is used by default as the start symbol.
- If you want to change the default behavior, use the `%start` directive.

Example 1: Definition Section of `expr.y`

```
%{
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int exprno = 1;
extern int yylex();
void yyerror(char *);
%}

%token NUM
%start P

%%
```

Example 1: Rules Section of `expr.y` (Actions inside Braces)

```
E       : E '+' T           { $$ = $1 + $3; }
        | E '-' T           { $$ = $1 - $3; }
        | T                 { $$ = $1; }
        ;

T       : T '*' F           { $$ = $1 * $3; }
        | T '/' F           { $$ = $1 / $3; }
        | T '%' F           { $$ = $1 % $3; }
        | F                 { $$ = $1; }

F       : NUM               { $$ = $1; }
        | '(' E ')'         { $$ = $2; }
        | '-' '(' E ')'     { $$ = -$3; }
        ;

P       : P L               { }
        | L                 { }
        ;

L       : E '\n'            { printf("Expression %d evaluates to %d\n", exprno, $1); ++exprno; }
        | '\n'              { }
        ;

%%
```

$E \to E + T \mid E - T \mid T$

$T \to T * F \mid T \,/\, F \mid T \,\%\, F \mid F$

$F \to \text{NUM} \mid (E) \mid -(E)$

$P \to P\,L \mid L$

$L \to E\ \backslash n$ (also ignore empty lines)

Example 1: Remaining Things

User-Code Section

```
void yyerror ( char *msg )
{
   fprintf(stderr, "*** Error: %s\n", msg);
}
```

Makefile

```
all: expr.y expr.l
        yacc -d expr.y
        lex expr.l
        gcc y.tab.c lex.yy.c -ly -ll

run: all
        ./a.out < input.txt

clean:
        -rm -f a.out lex.yy.c y.tab.c y.tab.h
```

Example 1: Sample Input and Output

Input file

```
1234567890

1 - 2 * 3 + 4 % 5 * 6 - 7

-3 - 4 * -5

((8 * 7) -- (6 + 5)) / -(4 - -3)

11 - 5 - 3 + ((2 + 2) * 3)

4 \
  + ( (14 * (6 + 12 + 2 * 16)) + 9 ) \
       * (11 * (3 + 21) + 15 * 8 * 13) \
       * 19 \
  + 5 * 18 \
  + 1 \
  + 10 * (7 + 17) \
  + 20
```

Sample run

```
$ make run
yacc -d expr.y
lex expr.l
gcc y.tab.c lex.yy.c -ly -ll
./a.out < input.txt
Expression 1 evaluates to 1234567890
Expression 2 evaluates to 12
Expression 3 evaluates to 17
Expression 4 evaluates to -9
Expression 5 evaluates to 15
Expression 6 evaluates to 24571459
$
```

Example 2: Write Your Own C File with `main()`

File `evalexpr.c`

```
#include <stdio.h>
#include <stdlib.h>
#include "lex.yy.c"
#include "y.tab.c"

int main ( int argc, char *argv[] )
{
   if (argc > 1) yyin = (FILE *)fopen(argv[1],"r");
   yyparse();
   fclose(yyin);
   exit(0);
}
```

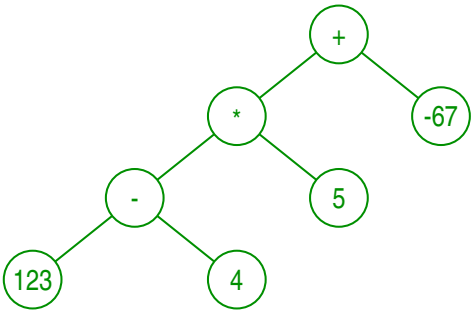
Makefile

```
all: expr.y expr.l evalexpr.c
        yacc -d expr.y
        lex expr.l
        gcc evalexpr.c

run: all
        ./a.out input.txt

clean:
        -rm -f a.out lex.yy.c y.tab.c y.tab.h
```

Example 3: Building a Syntax Tree

- We will continue with our expressions grammar.
- Instead of computing the values of expressions, we will generate a syntax tree (expression tree) for each input expression.
- These trees will be stored in a list of trees.
- After the parsing finishes, we will evaluate the trees one by one.
- The value of `NUM` will be of type `int`.
- The values of the operators will not be needed.
- The values of the non-terminal symbols are pointers to nodes in the trees.
- We can no longer use the default `int` type for all grammar symbols.

Expression tree for (123 - 4) * 5 + -67



Example 3: Definitions Section of `expr.y`: C Header

- This part (enclosed by `%{` and `%}`) will consist of the following.
  - The standard `#include`'s and the lex and yacc function prototypes.
  - Custom-made numbering of the operators (their ASCII values can be used instead).
  - A data type for tree nodes (binary tree in our case).
  - Global variables.
  - Prototypes of functions on these trees (we will use these functions in the actions of yacc).
- This part will go verbatim to `y.tab.c`.

```
%{
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void yyerror(char *);
```

Example 3: Definitions Section of `expr.y`: C Header

```
#define ARG 0
#define ADD 1
#define SUB 2
#define MUL 3
#define DIV 4
#define REM 5
#define NEG 6

typedef struct _node {
   int type;                    /* One of the above types */
   int value;                   /* The numeric value (to be evaluated later) */
   struct _node *left, *right;  /* Two child pointers */
} node;
typedef node *nodeptr;

nodeptr TREE[1024];
int nexpr = 0;

nodeptr createleaf ( int ) ;
nodeptr createbinary ( nodeptr, nodeptr, int ) ;
nodeptr createunary ( nodeptr, int ) ;
void evaltree ( nodeptr ) ;
%}
```

Example 3: Definitions Section of `expr.y`: Yacc Directives

- We need to define the type of each grammar symbol as a union of two types.
  - An integer value for the terminal token `NUM`.
  - A node pointer for each non-terminal symbol.
- Each member of the union can be a basic C data type or a pointer to a basic or user-defined data type.
- `typedef`-ed names cannot be used.
- We associate to each token and to each non-terminal, its type.

```
%union {
   int value;
   struct _node *nodeptr;
}

%token <value> NUM
%type <nodeptr> P L E T F
%start P

%%
```

Example 3: Definitions Section of `expr.y`: Rules Section

```
E       : E '+' T       { $$ = createbinary($1, $3, ADD); }
        | E '-' T       { $$ = createbinary($1, $3, SUB); }
        | T             { $$ = $1; }
        ;

T       : T '*' F       { $$ = createbinary($1, $3, MUL); }
        | T '/' F       { $$ = createbinary($1, $3, DIV); }
        | T '%' F       { $$ = createbinary($1, $3, REM); }
        | F             { $$ = $1; }

F       : NUM           { $$ = createleaf($1); }
        | '(' E ')'     { $$ = $2; }
        | '-' '(' E ')' { $$ = createunary($3, NEG); }
        ;

P       : P L           { }
        | L             { }
        ;

L       : E '\n'        { TREE[nexpr] = $1; ++nexpr; }
        | '\n'          { }
        ;

%%
```

Example 3: Implement the Tree Functions and `main()`

We write these codes in a separate file `exprtree.c`.

```
#include <stdio.h>
#include <stdlib.h>
#include "lex.yy.c"
#include "y.tab.c"

nodeptr createleaf ( int arg )
{
   nodeptr p;

   p = (nodeptr)malloc(sizeof(node));
   p -> type = ARG;
   p -> value = arg;
   p -> left = p -> right = NULL;
   return p;
}

nodeptr createbinary ( nodeptr L, nodeptr R, int OP )
{
   nodeptr p;

   p = (nodeptr)malloc(sizeof(node));
   p -> type = OP;
   p -> left = L;
   p -> right = R;
   return p;
}
```

Example 3: exprtree.c (Continued)

```
nodeptr createunary ( nodeptr L, int OP )
{
   nodeptr p;

   p = (nodeptr)malloc(sizeof(node));
   p -> type = OP;
   p -> left = L;
   p -> right = NULL;
   return p;
}

void evaltree ( nodeptr T )
{
   if (T == NULL) return;
   if (T -> left) evaltree(T -> left);
   if (T -> right) evaltree(T -> right);
   switch (T -> type) {
      case ARG: break;
      case ADD: T -> value = (T -> left -> value) + (T -> right -> value); break;
      case SUB: T -> value = (T -> left -> value) - (T -> right -> value); break;
      case MUL: T -> value = (T -> left -> value) * (T -> right -> value); break;
      case DIV: T -> value = (T -> left -> value) / (T -> right -> value); break;
      case REM: T -> value = (T -> left -> value) % (T -> right -> value); break;
      case NEG: T -> value = -(T -> left -> value); break;
      default: fprintf(stderr, "*** Unknown node type %d in expression tree\n", T -> type); break;
   }
}
```

Example 3: exprtree.c (Coninued)

```
int main ( int argc, char *argv[] )
{
   int i;

   if (argc > 1) yyin = (FILE *)fopen(argv[1], "r");

   yyparse();
   printf("\n+++ Parsing done\n+++ Going to evaluate the expression trees\n\n");

   fclose(yyin);

   for (i=0; i<nexpr; ++i) {
      evaltree(TREE[i]);
      printf("    Expression %d evaluates to %d\n", i+1, TREE[i] -> value);
   }

   exit(0);
}
```

Example 3: Is that All?

- In the first two examples, we have used the following action of lex on a numeric token.

  ```
  {num}            { yylval = atoi(yytext); return NUM; }
  ```

  This worked because the value of `NUM`-type tokens was `int`.
- Now, our token value is a union, so we need to refer to the correct component.

  ```
  {num}            { yylval.value = atoi(yytext); return NUM; }
  ```
- Other lex rules (these do not set `yylval`) do not need any change.

Makefile

```
all: expr.y expr.l exprtree.c
        yacc -d expr.y
        lex expr.l
        gcc exprtree.c

run: all
        ./a.out input.txt

clean:
        -rm -f a.out lex.yy.c y.tab.c y.tab.h
```

Example 3: Output on Our Sample Input File

```
$ make run
yacc -d expr.y
lex expr.l
gcc exprtree.c
./a.out input.txt

+++ Parsing done
+++ Going to evaluate the expression trees

    Expression 1 evaluates to 1234567890
    Expression 2 evaluates to 12
    Expression 3 evaluates to 17
    Expression 4 evaluates to -9
    Expression 5 evaluates to 15
    Expression 6 evaluates to 24571459
$
```

Default Action of Yacc

- You may omit the specifications of union types of terminals and non-terminals if their values are not needed.
- Examples
  - We have not defined the union types of the operators and punctuation symbols.
  - The types of `P` and `L` are not needed, so you can omit these non-terminals in the `%type` directive.
- However, if you specify the type of a non-terminal A, then every production with A as the head should set a value for `$$`.
- If you do not specify an action against such a production, yacc uses the default formula

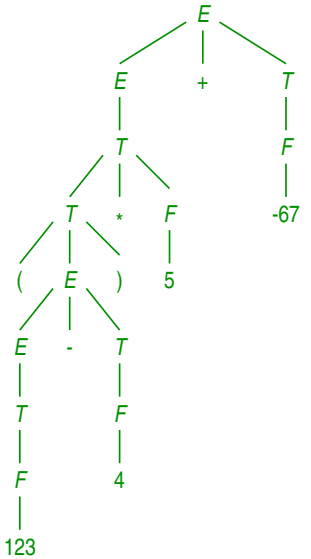
```
$$ = $1;
```

- This default action will fail in one of the following cases.
  - The union type of `$1` is undefined.
  - The union type of `$1` is different from the union type of `$$`.
  - There is no `$1` at all. This is applicable to productions of the form $A \rightarrow \epsilon$.

Exercise

- Write a yacc program to generate the parse tree for an input expression under our expression grammar.
- A parse tree shows the derivation process.
- The leaves contain the terminal symbol.
- Each internal node is marked by a non-terminal that is expanded at that step of derivation.
- Each node of the tree should store a (token, value) pair (along with at most three child pointers).

Parse tree for (123 - 4) * 5 + -67



Example 4: Intermediate-Code Generation

- Compilers usually do not generate explicit parse/syntax trees.
- At compile time itself, a sequence of three-address instructions of the following form is generated.

```
output = input1 op input2
```

- We store the values of the final expressions (r-values) as `%1`, `%2`, `%3`, and so on.
- The intermediate results are stored in temporaries with the names `t1`, `t2`, `t3`, and so on.
- The value of each non-terminal will be a (category, attribute) pair.
  - The category is ARG for a numeric value, and the attribute is that integer.
  - For a temporary, the category is TMP, and the attribute is the number of the temporary.
- The `%union` directive will consist of the following.
  - For a non-terminal, a pointer to the (category, attribute) pair.
  - For a NUM terminal, an integer.
  - No need to have a field for operators and parentheses.
- The following program only prints the three-address instructions. In reality, the three-address instructions are stored in a list for future processing (optimization and target-code generation).

Example 4: Intermediate-Code Generation (Definitions Section)

```
%{
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int yylex();
void yyerror(char *);

int exprno = 0;
int tempno = 0;

#define ARG 0
#define TMP 1

struct pair {
   int cat;
   int attr;
};
```

Example 4: Intermediate-Code Generation (Definitions Section)

```
struct pair *createarg ( int ) ;
struct pair *createbinary ( struct pair *, char, struct pair * ) ;
struct pair *createunary ( char, struct pair * ) ;
void setexpr ( struct pair * );

%}

%union {
   int value;
   struct pair *ptr;
}

%token <value> NUM
%type <ptr> P L E T F
%start P

%%
```

Example 4: Intermediate-Code Generation (Rules Section)

```
E       : E '+' T       { $$ = createbinary($1, '+', $3); }
        | E '-' T       { $$ = createbinary($1, '-', $3); }
        | T             { $$ = $1; }
        ;

T       : T '*' F       { $$ = createbinary($1, '*', $3); }
        | T '/' F       { $$ = createbinary($1, '/', $3); }
        | T '%' F       { $$ = createbinary($1, '%', $3); }
        | F             { $$ = $1; }

F       : NUM           { $$ = createarg($1); }
        | '(' E ')'     { $$ = $2; }
        | '-' '(' E ')' { $$ = createunary('-', $3); }
        ;

P       : P L           { }
        | L             { }
        ;

L       : E '\n'        { setexpr($1); }
        | '\n'          { }
        ;

%%
```

Example 4: Implementation of the functions (codegen.c)

```
#include <stdio.h>
#include <stdlib.h>
#include "lex.yy.c"
#include "y.tab.c"

struct pair *createarg ( int arg )
{
   struct pair *p;

   p = (struct pair *)malloc(sizeof(struct pair));
   p -> cat = ARG;
   p -> attr = arg;
   return p;
}

void printarg ( struct pair *arg )
{
   if (arg -> cat == ARG) printf("%d", arg -> attr);
   else printf("t%d", arg -> attr);
}

void setexpr ( struct pair *arg )
{
   ++exprno;
   printf("%%%d = ", exprno);
   printarg(arg);
   printf("\n\n");
}
```

Example 4: Implementation of the functions (codegen.c)

```
struct pair *createbinary ( struct pair *arg1, char op, struct pair *arg2 )
{
   struct pair *p;

   p = (struct pair *)malloc(sizeof(struct pair));
   p -> cat = TMP;
   p -> attr = ++tempno;
   printf("t%d = ", tempno);
   printarg(arg1);
   printf(" %c ", op);
   printarg(arg2);
   printf("\n");
   return p;
}

struct pair *createunary ( char op, struct pair *arg )
{
   struct pair *p;

   p = (struct pair *)malloc(sizeof(struct pair));
   p -> cat = TMP;
   p -> attr = ++tempno;
   printf("t%d = %c", tempno, op);
   printarg(arg);
   printf("\n");
   return p;
}
```

Example 4: Implementation of the functions (codegen.c)

```
int main ( int argc, char *argv[] )
{
   if (argc > 1) yyin = (FILE *)fopen(argv[1],"r");
   yyparse();
   fclose(yyin);
   exit(0);
}
```

Makefile

```
all: expr.y expr.l codegen.c
        yacc -d expr.y
        lex expr.l
        gcc codegen.c

run: all
        ./a.out input.txt

clean:
        -rm -f a.out lex.yy.c y.tab.c y.tab.h
```

Example 4: Sample Run on Our `input.txt`

```
%1 = 1234567890

t1 = 2 * 3
t2 = 1 - t1
t3 = 4 % 5
t4 = t3 * 6
t5 = t2 + t4
t6 = t5 - 7
%2 = t6

t7 = 4 * -5
t8 = -3 - t7
%3 = t8

t9 = 8 * 7
t10 = 6 + 5
t11 = -t10
t12 = t9 - t11
t13 = 4 - -3
t14 = -t13
t15 = t12 / t14
%4 = t15

t16 = 11 - 5
t17 = t16 - 3
t18 = 2 + 2
t19 = t18 * 3
t20 = t17 + t19
%5 = t20

t21 = 6 + 12
t22 = 2 * 16
t23 = t21 + t22
t24 = 14 * t23
t25 = t24 + 9
t26 = 3 + 21
t27 = 11 * t26
t28 = 15 * 8
t29 = t28 * 13
t30 = t27 + t29
t31 = t25 * t30
t32 = t31 * 19
t33 = 4 + t32
t34 = 5 * 18
t35 = t33 + t34
t36 = t35 + 1
t37 = 7 + 17
t38 = 10 * t37
t39 = t36 + t38
t40 = t39 + 20
%6 = t40
```

Using a Symbol Table

- So far, we have used integer operands only. Let us now introduce variables.
- Each line can be a standalone expression, or an assignment.
- We also allow variables and their negations as lowest-level operands in expressions.
- Here is the modified grammar. The start symbol is again P.

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid E - T \mid T \\
T & \rightarrow & T * F \mid T / F \mid T \,\%\, F \mid F \\
F & \rightarrow & \mathrm{NUM} \mid \mathrm{ID} \mid -\mathrm{ID} \mid (E) \mid -(E) \\
L & \rightarrow & E \,\backslash n \mid \mathrm{ID} = E \,\backslash n \\
P & \rightarrow & P\, L \mid L
\end{array}
$$

Using a Symbol Table

- We maintain a symbol table ST of defined (name, value) pairs. For simplicity, ST is implemented as an (unsorted) array of pairs.
- We use three functions.
  - `STindex(name)`: Given a `name`, find the index in ST where `name` resides, or -1 if `name` is not stored in ST.
  - `STload(name)`: Return the value stored in ST, corresponding to `name`, or 0 if `name` is not stored in ST.
  - `STstore(name,value)`: Store the given (name, value) pair in ST. If `name` is already present in ST, then its current value is overwritten. Otherwise a new entry is created in ST.
- The productions $F \rightarrow$ ID and $F \rightarrow -$ ID will use lookup in ST.
- The production $L \rightarrow E \ \backslash n$ will print the values of the standalone expressions as `%1`, `%2`, ...
- The production $L \rightarrow$ ID $=$ $E \ \backslash n$ will print what value is assigned to what variable. This will also store the (name, value) pair in ST.

Example 5: Using a Symbol Table (Definitions Section)

```
%{
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int exprno = 0;
int yylex();
void yyerror(char *);

typedef struct {
   char *name;
   int value;
} stpair;
stpair ST[4096];
int nstentry = 0;

int STindex ( char * ) ;
int STload ( char * ) ;
void STstore ( char *, int ) ;
%}

%union {
   int value;
   char *name;
}

%token <value> NUM
%token <name> ID
%start P
%type <value> P L E T F

%%
```

Example 5: Using a Symbol Table (Rules Section)

```
E       : E '+' T           { $$ = $1 + $3; }
        | E '-' T           { $$ = $1 - $3; }
        | T                 { $$ = $1; }
        ;

T       : T '*' F           { $$ = $1 * $3; }
        | T '/' F           { $$ = $1 / $3; }
        | T '%' F           { $$ = $1 % $3; }
        | F                 { $$ = $1; }

F       : NUM               { $$ = $1; }
        | ID                { $$ = STload($1); }
        | '-' ID            { $$ = -STload($2); }
        | '(' E ')'         { $$ = $2; }
        | '-' '(' E ')'     { $$ = -$3; }
        ;

P       : P L               { }
        | L                 { }
        ;

L       : E '\n'            { ++exprno; printf("  -------  %%%d = %d\n", exprno, $1); }
        | ID '=' E '\n'     { STstore($1,$3); printf("  ---  Assigning %s = %d\n", $1, $3); }
        | '\n'              { }
        ;

%%
```

Example 5: Using a Symbol Table (evalandset.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "lex.yy.c"
#include "y.tab.c"

int STindex ( char *name )
{
   int i;

   for (i=0; i<nstentry; ++i)
      if (!strcmp(name,ST[i].name)) return i;
   return -1;
}

int STload ( char *name )
{
   int i;
   char errmsg[1024];

   i = STindex(name);
   if (i == -1) {
      sprintf(errmsg, "Undefined variable %s (returning 0)", name);
      yyerror(errmsg);
      return 0;
   }
   return ST[i].value;
}
```

Example 5: Using a Symbol Table (evalandset.c)

```
void STstore ( char *name, int value )
{
   int i;

   i = STindex(name);
   if (i == -1) {
      i = nstentry;
      ST[i].name = strdup(name);
      ++nstentry;
   }
   ST[i].value = value;
}

int main ( int argc, char *argv[] )
{
   if (argc > 1) yyin = (FILE *)fopen(argv[1],"r");
   yyparse();
   fclose(yyin);
   exit(0);
}
```

Makefile

```
all: expr.y expr.l evalandset.c
        yacc -d expr.y
        lex expr.l
        gcc evalandset.c

run: all
        ./a.out input.txt

clean:
        -rm -f a.out lex.yy.c y.tab.c y.tab.h
```

Example 5: Using a Symbol Table (Sample Run)

Input

```
a = 123
b = 1 - 2 * 3 + 4 % 5 * 6 - 7
c = -3 - b * -5

- ((a * b) -- (a + b)) /- (c--c)

x1 = a / b - 1
x2 = a % b - 1
y = (x1 * x1 + x2 * x2) / (x1 * x2 + -1)

_b = -(-b)
b = \
    4 \
      + ( (14 * (6 + b + 2 * 16)) + 9 ) \
          * (11 * (b + b) + 15 * 8 * (b + 1)) \
          * 19 \
      + 5 * 18 \
      + 1 \
      + 10 * (7 + 17) \
      + c \
      - 37
-b

z = x + y
```

Output

```
$
make run
yacc -d expr.y
lex expr.l
gcc evalandset.c
./a.out input.txt
  ---  Assigning a = 123
  ---  Assigning b = 12
  ---  Assigning c = 57
  -------  %1 = 14
  ---  Assigning x1 = 9
  ---  Assigning x2 = 2
  ---  Assigning y = 5
  ---  Assigning _b = 12
  ---  Assigning b = 24571459
  -------  %2 = -24571459
*** Error: Undefined variable x (returning 0)
  ---  Assigning z = 5
$
```

Ambiguous Grammars: Operator Precedence

- Consider the following grammar for expressions (numeric operands only).

$$E \rightarrow E + E \mid E - E \mid E * E \mid E \,/\, E \mid E \,\%\, E \mid -E \mid (\, E \,) \mid \text{NUM}$$

- This grammar is much simpler to code than our earlier grammar for expressions.

```
E       : E '+' E           { $$ = $1 + $3; }
        | E '-' E           { $$ = $1 - $3; }
        | E '*' E           { $$ = $1 * $3; }
        | E '/' E           { $$ = $1 / $3; }
        | E '%' E           { $$ = $1 % $3; }
        | '(' E ')'         { $$ = $2; }
        | '-' E             { $$ = -$2; }
        | NUM               { $$ = $1; }
        ;
```

Ambiguous Grammars: Operator Precedence

- This grammar is ambiguous. On our first sample input, it gives the following output.

```
Expression 1 evaluates to 1234567890
Expression 2 evaluates to -13
Expression 3 evaluates to -23
Expression 4 evaluates to -9
Expression 5 evaluates to 21
Expression 6 evaluates to -1478503644
```

- We can specify the precedence and associativity of tokens in the Definitions Section. `%left`, `%right`, `%nonassoc` mean left-associative, right-associative, and non-associative, respectively. Operators defined in later lines have higher precedence.

```
%left '+' '-'
%left '*' '/' '%'
%nonassoc NEG
```

- We also write the production $E \rightarrow -E$ as follows.

```
...
          | '-' E %prec NEG     { $$ = -$2; }
...
```

`NEG` is a symbolic token of higher precedence than the binary operators. The reduction $E \rightarrow -E$ is carried out at the precedence of `NEG`.

Ambiguous Grammars: Dangling-else Problem

- Consider the following two productions.

$$S \rightarrow \text{IF } (C)\ S \mid \text{IF } (C)\ S\ \text{ELSE}\ S$$

- Here, IF and ELSE are tokens for the keywords `if` and `else`. S and C are non-terminals for statements and conditions.
- In the sentential form `IF (C) IF (C) S ELSE S`, we want `ELSE` to be attached to the second `IF`. That is, if `ELSE` appears at the input, shifting it should have higher preference than the reduction by the first rule.
- Add the following lines to the Definitions section, so the token `ELSE` has higher precedence (because it appears later) than the symbolic precedence name `REDUCTION_PRECEDENCE`.

```
%nonassoc REDUCTION_PRECEDENCE
%nonassoc ELSE
```

- In the Rules Section, use the following.

```
S : IF '(' C ')' S %prec REDUCTION_PRECEDENCE    { . . . }
  | IF '(' C ')' S ELSE S                        { . . . }
```

Actions inside the Production Body: Marker Non-Terminals

- So far, all yacc actions are shown at the end of the productions.
- These actions are carried out during the respective reductions.
- If you want to do some work in the middle of a production body, one possibility is to split the production body before every such action.
- Another option is to use **marker** non-terminals.
- A marker non-terminal derives only the empty string ϵ, and so does not interfere with the language of the grammar.
- Yacc may support embedded actions written as `{...}` anywhere inside a production body.
- Yacc automatically generates marker non-terminals to handle the embedded actions.
- Here, we use explicit marker non-terminals for better understanding, and for having potentially enhanced controls.

Example 6: Using Marker Non-Terminals

- Suppose that our grammar accepts only the strings a^+b^+ and c^+d^+.
- We want to compute counts of (*a*'s and *b*'s) or (*c*'s and *d*'s) in a valid input.
- We write a yacc file `marker.y` for this.
- Two marker non-terminals *M* and *N* are used in the program.
- For this simple pattern matching, we do not involve lex, but write our own function `yylex()`.

```
%{
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>

int na, nb, nc, nd;

int yylex ( )
{
   char nextchar;

   scanf("%c", &nextchar);
   if ((nextchar >= 'a') && (nextchar <= 'd')) return nextchar;
   return 0;
}

void yyerror ( char * ) ;
%}

%%
```

Example 6: Using Marker Non-Terminals

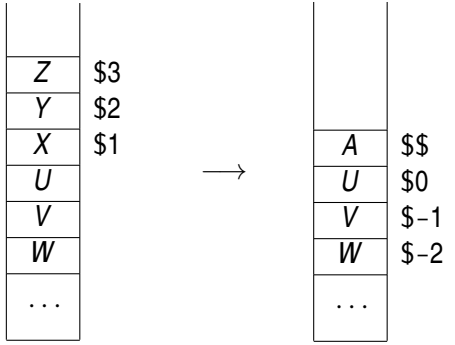
```
S       : A M B { printf("+++ %d b's are read\n", nb); }
        | C N D { printf("+++ %d d's are read\n", nd); }
        ;

A       : 'a' { na = 1; } | A 'a' { ++na; } ;
B       : 'b' { nb = 1; } | B 'b' { ++nb; } ;
C       : 'c' { nc = 1; } | C 'c' { ++nc; } ;
D       : 'd' { nd = 1; } | D 'd' { ++nd; } ;

M       : { printf("+++ %d a's are read\n    Going to read b's now\n", na); } ;
N       : { printf("+++ %d c's are read\n    Going to read d's now\n", nc); } ;

%%

void yyerror ( char *msg ) { fprintf(stderr, "%s\n", msg); }
```

Sample Runs

```
$ yacc marker.y
$ gcc y.tab.c -ly
$ ./a.out
aaaaabbbbbbbbbbb
+++ 5 a's are read
    Going to read b's now
+++ 12 b's are read
$ ./a.out
ccccccccccccccccddddddddd
+++ 16 c's are read
    Going to read d's now
+++ 8 d's are read
$
```

Accessing the Parse Stack

- The end-of-body actions demonstrated in our examples so far generate `$$` as a function of the grammar symbols `$1`, `$2`, `$3`, `...` appearing in the body of the productions.
- On some occasions, the elements waiting in the parse stack for future reductions are needed.
- Think of a production $A \rightarrow XYZ$. A reduction by this production pops X, Y, and Z (named as `$1`, `$2`, and `$3`), and pushes A (`$$`) to the top of the stack.
- You can access the stack symbols beneath this top (storing A) as `$0`, `$-1`, `$-2`, and so on (in the sequence from top to bottom).
- If `$-`i leads you to beyond the bottom of the stack, the result is undefined.
- We can set some attributes of A in terms of the items beneath it in the stack.
- A typical use of this is in the implementation of L-attributed grammars.
- This can very well be used for productions $M \rightarrow \epsilon$ involving marker non-terminals M.



Example 7: Accessing the Parse Stack

- The following code `showstack.y` reads a line from the input, and prints the stack contents near the top after every reduction.
- In order that we never access beneath the bottom of the parse stack, we start by pushing three symbols (+) at the beginning.
- We use a customized `yylex()` that inserts these three symbols at the beginning of the input.
- The following grammar is used. Here *c* is a character-type token that matches anything other than the new-line character.

$$\begin{aligned} S &\rightarrow +++T \\ T &\rightarrow c \mid c\,T \end{aligned}$$

- We use the default `int` data type for stack elements (so no `%union` is needed).
- Characters are stored as integers, and printed as characters.
- We use `yylval` and a token, so invoking yacc with the `-d` option, and including `y.tab.h` are necessary.

Example 7: Printing the Parse Stack

```
%{
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include "y.tab.h"

int atbegin = 0;

int yylex ( )
{
   char nextchar;

   if (atbegin < 3) { ++atbegin; yylval = '+'; return '+'; }

   scanf("%c", &nextchar);
   if (nextchar != '\n') {
      yylval = nextchar;
      return CHR;
   }
   return 0;
}

void yyerror ( char * ) ;
%}
```

Example 7: Printing the Parse Stack

```
%token CHR

%%

S       : '+' '+' '+' T { $$ = 'S'; printf("    Final reduction done\n"); } ;

T       : CHR    {
                 $$ = 'T';
                 printf("    $-2 = %c, $-1 = %c, $0 = %c, $$ = %c, $1 = %c\n",
                        $-2, $-1, $0, $$, $1);
               }
        | CHR T  {
                 $$ = 'T';
                 printf("    $-2 = %c, $-1 = %c, $0 = %c, $$ = %c, $1 = %c, $2 = %c\n",
                        $-2, $-1, $0, $$, $1, $2);
               }
        ;

%%

void yyerror ( char *msg ) { fprintf(stderr, "%s\n", msg); }
```

Example 7: Printing the Parse Stack (Sample Run)

Yacc shifts all the symbols from the user to the stack. At the end of the input, reduction by $T \rightarrow c$ is used once, and subsequently reductions by $T \rightarrow cT$ are used to consume all the input characters pushed to the stack. The final reduction uses $S \rightarrow +++T$.

```
$ yacc -d showstack.y
$ gcc y.tab.c -ly
$ ./a.out
abcdefghijkl12345678
    $-2 = 5, $-1 = 6, $0 = 7, $$ = T, $1 = 8
    $-2 = 4, $-1 = 5, $0 = 6, $$ = T, $1 = 7, $2 = T
    $-2 = 3, $-1 = 4, $0 = 5, $$ = T, $1 = 6, $2 = T
    $-2 = 2, $-1 = 3, $0 = 4, $$ = T, $1 = 5, $2 = T
    $-2 = 1, $-1 = 2, $0 = 3, $$ = T, $1 = 4, $2 = T
    $-2 = l, $-1 = 1, $0 = 2, $$ = T, $1 = 3, $2 = T
    $-2 = k, $-1 = l, $0 = 1, $$ = T, $1 = 2, $2 = T
    $-2 = j, $-1 = k, $0 = l, $$ = T, $1 = 1, $2 = T
    $-2 = i, $-1 = j, $0 = k, $$ = T, $1 = l, $2 = T
    $-2 = h, $-1 = i, $0 = j, $$ = T, $1 = k, $2 = T
    $-2 = g, $-1 = h, $0 = i, $$ = T, $1 = j, $2 = T
    $-2 = f, $-1 = g, $0 = h, $$ = T, $1 = i, $2 = T
    $-2 = e, $-1 = f, $0 = g, $$ = T, $1 = h, $2 = T
    $-2 = d, $-1 = e, $0 = f, $$ = T, $1 = g, $2 = T
    $-2 = c, $-1 = d, $0 = e, $$ = T, $1 = f, $2 = T
    $-2 = b, $-1 = c, $0 = d, $$ = T, $1 = e, $2 = T
    $-2 = a, $-1 = b, $0 = c, $$ = T, $1 = d, $2 = T
    $-2 = +, $-1 = a, $0 = b, $$ = T, $1 = c, $2 = T
    $-2 = +, $-1 = +, $0 = a, $$ = T, $1 = b, $2 = T
    $-2 = +, $-1 = +, $0 = +, $$ = T, $1 = a, $2 = T
    Final reduction done
$
```

Example 8: Type declarations

- As a more useful application of marker non-terminals and stack queries, consider a grammar for generating type declarations of the following form.

  ```
  type var1, var2, ..., varn;
  ```

- For simplicity, we restrict such declarations as follows.
  - `type` can be one of `char`, `int`, `float`, and `double`.
  - Each `var` is a valid name, or a single pointer (like `*name`), or a 1-d array (like `name[size]`).
  - A type declaration does not span across multiple lines. Empty lines would be ignored.
  - All pointers are of the same size (assumed to be 8 bytes).
- Once `type` is read from the first token (a keyword) in a line, that base type applies throughout the entire line. A marker non-terminal *N* remembers the base type (and some related information) in global variables, immediately after it is read. (Inherited attributes can be used instead.)
- The yacc program will print all individual variable declarations, and will also compute the total memory requirement for these variables. The program will not check whether there are multiple declarations for the same name. In reality, the variable names and types should be stored in a symbol table which will prohibit duplicate definitions under the same names.

Example 8: Type declarations (Lex File `typedecl.l`)

```
%{
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "y.tab.h"
%}

ws      [ \t]+
id      [a-zA-Z_][a-zA-Z0-9_]*
num     [0-9]+
typ     (char|int|float|double)

%%

{ws}    { }
{typ}   { yylval.str = strdup(yytext); return TYPE; }
{id}    { yylval.str = strdup(yytext); return ID; }
\[      { return yytext[0]; }
\]      { return yytext[0]; }
{num}   { yylval.val = atoi(yytext); return NUM; }
,       { return yytext[0]; }
\*      { return yytext[0]; }
;       { return yytext[0]; }
\n      { return yytext[0]; }
.       { fprintf(stderr, "*** Lex error: Bad character '%c'\n", yytext[0]); }

%%

int yywrap ( ) { return 1; }
```

Example 8: Type declarations (Yacc File `typedecl.y`)

- We use three tokens: types and names are strings, whereas array sizes are integers.
- We use a `%union` as the yacc stack-element type.
- The `%token` and `%type` directives will associate types with the grammar symbols.
- When we refer to stack elements as `$0`, `$-1`, `$-2`, and so on, yacc does not consult the `%token` and `%type` directives.
- We have to explicitly specify the type within angular brackets between `$` and the following position indicator (`0,-1,-2,...`).
- Examples: `$<str>0`, `$<nodeptr>-1`, `$<lexval>-2`, and so on.
- During reduction, it is not necessary to write the types in `$$`, `$1`, `$2`, `$3`, and so on.
- We use two marker non-terminals.
  - *M* will be used to print the line number.
  - *N* will be used to store the base type and size in global variables.

Example 8: Type declarations (Yacc File `typedecl.y`)

```
%{
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* Custom-made short-cuts for the base types */
#define CHR 1
#define INT 2
#define FLT 3
#define DBL 4

int lineno = 1;     /* Line number in the input file */
int basetype;       /* One of the above custom-made values */
int basesize;       /* We use the sizeof() operator to determine the size of the base type */
int ptrsize = 8;    /* All pointers are assumed to be of size 8 bytes */
int memreq = 0;     /* Total memory requirement of all the variables */

extern int yylex();
extern FILE *yyin;
extern void yyerror(char *);

void printtype ( int ) ;   /* Printing the integer short-cuts as strings */
%}
```

Example 8: Type declarations (Yacc File `typedecl.y`)

```
union {
   int val;
   char *str;
}

%token <val> NUM
%token <str> TYPE ID

%type <val> PROG DECL DECLIST VARLIST VAR M N

%start PROG

%%

PROG    : DECLIST  { printf("\n--- Total memory requirement = %d\n", memreq); }
        ;

DECLIST : DECL DECLIST  { }
        | DECL  { }
        ;

DECL    : M TYPE N VARLIST ';' '\n'  { ++lineno; }
        | '\n'  { ++lineno; }
        ;
```

Example 8: Type declarations (Yacc File `typedecl.y`)

```
M       : { printf("+++ Line %d\n", lineno); } ;

N       : {
            if (!strcmp($<str>0,"char")) { basetype = CHR; basesize = sizeof(char); }
            else if (!strcmp($<str>0,"int")) { basetype = INT; basesize = sizeof(int); }
            else if (!strcmp($<str>0,"float")) { basetype = FLT; basesize = sizeof(float); }
            else if (!strcmp($<str>0,"double")) { basetype = DBL; basesize = sizeof(double); }
          } ;

VARLIST : VAR
        | VAR ',' VARLIST
        ;

VAR     : ID  {
                printf("\t"); printtype(basetype); printf(" variable: %s\n", $1);
                memreq += basesize;
              }
        | '*' ID  {
                    printf("\t"); printtype(basetype); printf(" pointer: %s\n", $2);
                    memreq += ptrsize;
                  }
        | ID '[' NUM ']'  {
                            printf("\t"); printtype(basetype);
                            printf(" array of size %d: %s\n", $3, $1);
                            memreq += basesize * $3;
                          }
        ;
```

Example 8: Type declarations (Yacc File `typedecl.y`)

```
%%

void yyerror ( char *msg ) { fprintf(stderr, "*** Error: %s\n", msg); }

void printtype ( int T )
{
   switch (T) {
      case CHR: printf("char"); break;
      case INT: printf("int"); break;
      case FLT: printf("float"); break;
      case DBL: printf("double"); break;
      default: printf("unknown"); break;
   }
}

int main ( int argc, char *argv[] )
{
   if (argc > 1) yyin = (FILE *)fopen(argv[1],"r");
   yyparse();
   fclose(yyin);
   exit(0);
}
```

Example 8: Type declarations (Sample Run)

Input file

```
char a, b, *cptr, CARR[10];
int i, j, *iptr, IARR[100];

float r, s, t, FARR[50], *fptr;
double x1, x2, x3, DARR[200], *dptr;
```

Output

```
$ make run
yacc -d typedecl.y
lex typedecl.l
gcc y.tab.c lex.yy.c
./a.out input.txt
+++ Line 1
        char variable: a
        char variable: b
        char pointer: cptr
        char array of size 10: CARR
+++ Line 2
        int variable: i
        int variable: j
        int pointer: iptr
        int array of size 100: IARR
+++ Line 4
        float variable: r
        float variable: s
        float variable: t
        float array of size 50: FARR
        float pointer: fptr
+++ Line 5
        double variable: x1
        double variable: x2
        double variable: x3
        double array of size 200: DARR
        double pointer: dptr

--- Total memory requirement = 2288
```

Example 9: Expression Grammar with Partial Sums and Products

- Let us consider the following grammar with integer variables.

$$
\begin{aligned}
E &\rightarrow T\ E' \\
E' &\rightarrow +\ T\ E' \mid -\ T\ E' \mid \epsilon \\
T &\rightarrow F\ T' \\
T' &\rightarrow *\ F\ T' \mid /\ F\ T' \mid \%\ F\ T' \mid \epsilon \\
F &\rightarrow \mathrm{NUM} \mid (\ E\ ) \mid -\ (\ E\ )
\end{aligned}
$$

  This grammar is free from left recursion, and is particularly suited to LL(1) parsers. Yacc uses LALR(1) parsing (by default), but can still handle this grammar.

- We also process one expression in each line, so we use some additional productions.

$$
\begin{aligned}
P &\rightarrow P\ L \mid L \\
L &\rightarrow E\ \backslash n \mid \backslash n
\end{aligned}
$$

Example 9: Expression Grammar with Partial Sums and Products

- The problem with the expression grammar is that partial sums and partial products need to pass down the parse tree as inherited attributes. (The binary operators are left-associative.)
- As soon as a term is evaluated, its value should update the partial sum.
- As soon as a factor is evaluated, its value should update the partial product.
- The end-of-production actions effectively handle only synthesized attributes.
- In the grammar, T and F appear not at the end of the productions.
- We should use marker non-terminals immediately after T and F to update and propagate the partial sums and products.
- Marker non-terminals derive ϵ only, and have no bearing on the input language.
- However, they are valid grammar symbols, and so (in the parse stack) can store values and/or pointers, like any other grammar symbol.
- Since all operands and results are integers in our example, we use the default stack-element type (`int`) of yacc. Writing the types in `$0`, `$-1`, `$-2`, ... is not necessary.

Example 9: The Lex File, and the Yacc Definitions

- We use the same lex file as in Example 1.
- The initial definition part of the yacc file is also straightforward.

```
%{
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

extern FILE *yyin ;
extern int yylex() ;
void yyerror ( char * ) ;

int nexpr = 0;
%}

%token NUM
%start P

%%
```

Example 9: Handling Partial Sums

- We use the marker non-terminals M, M_1, M_2, M_3.
- M initializes the partial sum by the value of the first term.
- M_1 and M_2 update the previous partial sum by incorporating the value of the next term.
- M_3 converts the last partial sum to the final sum.
- Subsequently, the end-of-production actions for E and E' (E' is written as `E1`) pass the final sum verbatim to the top of the tree.

```
E       : T M E1          { $$ = $3; } ;

M       : { $$ = $0; } ;

E1      : '+' T M1 E1     { $$ = $4; }
        | '-' T M2 E1     { $$ = $4; }
        | M3              { $$ = $1; }
        ;

M1      : { $$ = $-2 + $0; } ;
M2      : { $$ = $-2 - $0; } ;
M3      : { $$ = $0; } ;
```

Example 9: Handling Partial Products

- Now, five marker non-terminals N, N_1, N_2, N_3, N_4 are used.
- N is for initializing the partial product by the value of the first factor.
- N_1, N_2, N_3 move the updated partial product down the tree.
- N_4 converts the last partial product to the final product for moving up the tree.

```
T       : F N T1          { $$ = $3; } ;

N       : { $$ = $0; } ;

T1      : '*' F N1 T1     { $$ = $4; }
        | '/' F N2 T1     { $$ = $4; }
        | '%' F N3 T1     { $$ = $4; }
        | N4              { $$ = $1; }
        ;

N1      : { $$ = $-2 * $0; } ;
N2      : { $$ = $-2 / $0; } ;
N3      : { $$ = $-2 % $0; } ;
N4      : { $$ = $0; } ;
```

Example 9: The Remaining Part of the Yacc File

```
F       : NUM           { $$ = $1; }
        | '(' E ')'     { $$ = $2; }
        | '-' '(' E ')' { $$ = -$3; }
        ;

L       : E '\n'        { ++nexpr; printf("%%%d = %d\n", nexpr, $1); }
        | '\n'          { }
        ;

P       : P L           { }
        | L             { }
        ;

%%

void yyerror ( char *msg ) { fprintf(stderr, "    +++ Error: %s\n", msg); }
```

On the input file of Example 1, we get the expected output.

```
%1 = 1234567890
%2 = 12
%3 = 17
%4 = -9
%5 = 15
%6 = 24571459
```