Yacc: A Parser Generator†

Stephen C. Johnson
Ravi Sethi

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Since the early 1970s, *yacc* has been used to implement hundreds of languages, big and small. Its applications range from small desk calculators, to medium-sized preprocessors for typesetting, to large compiler front ends for complete programming languages.

A *yacc* specification is based on a collection of grammar rules that describe the syntax of a language; *yacc* turns the specification into a syntax analyzer. A pure syntax analyzer merely checks whether or not an input string conforms to the syntax of the language.

We can go beyond pure syntax analysis by attaching code in C or C++ to a grammar rule; such code is called an action, and is executed whenever the rule is applied during syntax analysis. Thus, a desk calculator might use actions to evaluate an expression, and a compiler front end might use actions to emit intermediate code.

Yacc allows us to build parsers from LALR(1) grammars without necessarily learning the underlying theory.

1. Introduction

Yacc is a tool for building syntax analyzers, also known as *parsers*. This section introduces the basic features of *yacc*. We review grammars, build a pure parser for real numbers, and then augment the parser to evaluate numbers during parsing. The language of real numbers is a toy; realistic examples appear in Section 2.

Uppercase letters are distinct from lowercase letters in *yacc* specifications. Thus, `digit` and `DIGIT` are distinct names. The font of a name is chosen purely for readability, so `fraction`, and *fraction* (within diagrams) refer to the same name.

1.1. Further Reading

Yacc is designed to handle a single but significant part of the total job of building a translator or interpreter for a language. The remaining parts of the job must be implemented in a host programming language, presumed to be C [9] or C++ [12].

Lex [10], a tool for building lexical analyzers, works in harmony with *yacc*. It can be easily used to produce quite complicated lexical analyzers, but there remain some languages (Fortran, for example) whose lexical analyzers must be crafted by hand.

Kernighan and Pike [8] illustrate program development using *yacc* and *lex* by gradually extending an expression evaluator into an interpreter for a language comparable to Basic. Schreiner and Friedman [11] conduct a book-length case study of how to create a compiler using *yacc* and *lex*.

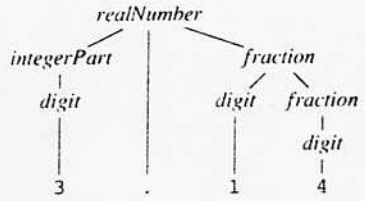
Textbooks on compilers such as [3] provide more information on the behavior and construction of parsers than will be covered here. The algorithms underlying *yacc* are also discussed in a survey of LR parsing [1]. A feature that sets *yacc* apart — its ability to build fast compact LR parsers from ambiguous grammars — is based on the theory developed in [2].

Among the earliest applications of *yacc* are *eqn* [7], a language for typesetting mathematics, and *pcc*, the Portable C Compiler [5].

† Prepared by R. Sethi from [6]. S. C. Johnson is presently with Ardent Computer, 880 West Maude Ave., Sunnyvale, California 94086.

1.2. Grammars, Reviewed

The *syntax* of a language imposes a hierarchical structure, called a *parse tree*, on strings in the language. The following is a parse tree for the string `3.14` in a language of real numbers:



The leaves at the bottom of a parse tree are labeled with *terminals* or *tokens*; tokens represent themselves. By contrast, the other nodes of a parse tree are labeled with *nonterminals*. Each node in the parse tree is based on a rule, called a *production*, that defines a nonterminal in terms of a sequence of terminals and nonterminals. The root of the parse tree for `3.14` is based on the following informally stated production:

A real number consists of an integer part, a point, and a fraction.

This production is written as follows in the notation accepted by *yacc*:

```
realNumber :  integerPart '.' fraction
           ;
```

Together, the tokens, the nonterminals, the productions, and a distinguished nonterminal, called the *start symbol*, constitute a *grammar* for a language. Both tokens and nonterminals are referred to as *grammar symbols*, or simply *symbols*.

1.3. Grammars in Yacc Specifications

The three sections of a *yacc* specification are for optional declarations, productions, and optional user-supplied routines. The productions are the heart of a specification; they comprise all but the first two and last two lines of Figure 1. The sections are separated by double percent `%%` marks — the percent symbol is generally used by *yacc* as an escape character. If the user-routines section is omitted, the second `%%` mark can be omitted as well.

```
%start lines

lines       : /* empty */
            | lines realNumber '\n'
            ;
realNumber  : integerPart '.' fraction
            ;
integerPart : digit
            | integerPart digit
            ;
fraction    : digit
            | digit fraction
            ;
digit       : '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
            ;

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

Figure 1. A complete *yacc* specification for sequences of real numbers, one per line.

Blanks, tabs, and newlines are ignored. Comments can appear wherever a name can; they are enclosed between `/*` and `*/`, as in C.

It is possible, and desirable, for the start symbol of a grammar to be declared explicitly, using the `%start` keyword, as in

```
%start lines
```

Otherwise, the start symbol is taken from the first production in the specification.

A name in the production section is presumed to represent a nonterminal unless it is explicitly declared to represent a token. There are no token declarations in Figure 1. The following declaration of token `DIGIT` is from an example later in this section.

```
%token DIGIT
```

Single-character tokens need not be declared; a *literal* is a character enclosed in single quotes. As in C, the backslash \ is an escape character within literals, and the following C escapes are recognized:

`'\n'`	newline
`'\r'`	return
`'\''`	single quote '
`'\t'`	tab
`'\b'`	backspace
`'\f'`	form feed
`'\xxx'`	*xxx* in octal

For technical reasons, the `NUL` character, `'\0'` or 0, should never be used in productions.

A production

```
realNumber :  integerPart '.' fraction
           ;
```

defines a nonterminal, called its *left side*, in terms of a sequence of grammar symbols, called its *right side*. A colon separates the two sides, and a semicolon marks the end of the production. The left side of this production is `realNumber`. Its right side consists of `integerPart`, the literal `'.'`, and `fraction`.

The right side of a production can be empty, as in the following production with no symbols between the colon and the terminating semicolon:

```
lines : ;
```

Productions with the same left side can be combined, and written with a vertical bar separating the right sides. The productions

```
integerPart :  digit
            ;
integerPart :  integerPart digit
            ;
```

can be rewritten equivalently as

```
integerPart :  digit
            |  integerPart digit
            ;
```

In words, an integer part is either a single digit or a (smaller) integer part followed by a digit. Thus, an integer part consists of a string of one or more digits.

A fraction also consists of a string of one or more digits, described by the productions

```
fraction :  digit
         |  digit fraction
         ;
```

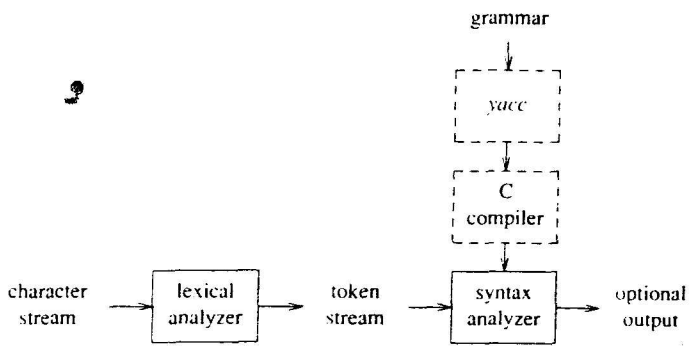


Figure 2. *Yacc* handles the syntax analysis part of an application.

The nonterminals `integerPart` and `fraction` impose different hierarchical structures on strings of digits. Note how a tree for `integerPart` grows down to the left, whereas a tree for `fraction` grows down to the right:

Semantic considerations influence the choice of hierarchical structure, and hence the choice of productions for a nonterminal, as we shall see in Section 2.

1.4. Using Yacc

When *yacc* is applied to a specification, the output is a file of C code, called `y.tab.c` (the name might differ due to local file-system conventions). Suppose that the specification in Figure 1 appears in a file `real.y`. A program for reading a sequence of real numbers can then be compiled into a file `a.out` by the following UNIX® system commands:

```
yacc real.y          generates C code into file y.tab.c
cc y.tab.c -ly       compiles executable program into file a.out
```

The flag `-ly`, which must appear after `y.tab.c`, refers to a tiny library, described below.

Figure 2 illustrates the use of *yacc*. *Yacc* and the C compiler are represented by dashed boxes since they are used once, at "compiler-construction time." The constructed compiler, consists of the lexical analyzer, the syntax analyzer, and any user routines.

Yacc confines itself to building fast parsers. All other aspects of an application, such as initialization, lexical analysis, and error reporting, must be programmed separately. Some relevant function names in the C code are as follows:

- `yyparse` is the parser generated by *yacc*. It returns 0 if the entire input is parsed successfully; otherwise it returns 1.
- `yylex` is called repeatedly by `yyparse`; it reads input characters and returns tokens. A routine that groups characters into tokens is called a *lexical analyzer*. The lexical analyzer at the bottom of Figure 1

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

simply returns each individual character as a token.

- `main` is the start-up routine. Execution of a C program begins in a function called `main`. In general, `main` might read command-line arguments and options and perform initialization before calling `yyparse`.
- `yyerror` is called by `yyparse` if an error occurs during parsing, usually with the terse message "`syntax error`." Parsing terminates when an error is detected unless "error productions" are used as described in Section 5.

The library `-ly` contains default versions of `main` and `yyerror`:

```
int main() { return yyparse(); }
#include <stdio.h>
void yyerror(s) char *s; { fprintf(stderr, "%s\n", s); }
```

This version of `main` simply returns the result obtained from `yyparse`, and this version of `yyerror` simply prints the message it is called with.

In case the `-ly` library is not available, the specification in Figure 1 can be completed by adding these three lines at the bottom.

1.5. Actions and Attributes

Actions attached to a production are executed each time the production is applied during parsing. An *action* consists of one or more C statements, enclosed in curly braces { and }. Within an action, pseudo-variables starting with $ signs refer to values associated with the symbols in the production. Such values are called *attributes*. The pseudo-variable for the left side is $$. In the local version of *yacc*, the pseudo-variable for a symbol on the right side is formed by prefixing a $ sign to either its name or its position.

The complete specification in Figure 3 includes the production and action

```
realNumber : integerPart fraction { $$ = $integerPart + $fraction; } ;
```

The action is the single statement

```
$$ = $integerPart + $fraction;
```

It defines the attribute value associated with the left side to be the sum of the attribute values associated with the two symbols on the right side.[1]

All versions of *yacc* support pseudo-variables like $1 and $2 formed from positions on the right side. Using positions, this action can be rewritten equivalently as

```
realNumber :  integerPart fraction  { $$ = $1 + $2; }  ;
```

The $-prefix notation permits one attribute per grammar symbol. By default, all attributes have the same type, `int`. This default can be changed by defining `YYSTYPE`, as on line 4 of Figure 3, where `YYSTYPE` is defined to be `double` because the specification deals with real numbers. See Section 6 for how to customize the types of attributes; that is, to allow different attributes to have different types. (As in Figure 3, any C code in the declarations section must be enclosed between %{ and %}.)

Attributes for tokens are computed by the lexical analyzer. Conceptually, a lexical analyzer returns a pair, consisting of a token and an associated attribute value. Consider, for example, the token `DIGIT` in Figure 3. When the lexical analyzer reads the character `1` it returns `DIGIT` with attribute value 1, when it reads 2 it returns `DIGIT` with attribute value 2, and so on.

Specifically, the parser `yyparse` expects the lexical analyzer `yylex` to leave the attribute value in a global variable `yylval`, which is automatically declared by *yacc* to have type `YYSTYPE`. The function `yylex` in Figure 3 assigns a value to `yylval` just before it returns the token `DIGIT`.

[1] When the same symbol x appears several times on the right side, its pseudo-variable can be written as $x#1, $x#2, $x#3.

```
%token DIGIT
%start lines
%{
#define YYSTYPE double
%}
%%
lines       : /* empty */
            | lines realNumber '\n'
                { printf("%g\n", $realNumber); }
            ;
realNumber  : integerPart '.' fraction
                { $$ = $integerPart + $fraction; }
            ;
integerPart : DIGIT
            | integerPart DIGIT
                { $$ = $integerPart*10 + $DIGIT; }
            ;
fraction    : DIGIT
                { $$ = $DIGIT*0.1; }
            | DIGIT fraction
                { $$ = ($DIGIT + $fraction)*0.1; }
            ;
%%
#include <ctype.h>
int  yylex()   {
    int c;
    c = getchar();
    if( ! isdigit(c) ) return c;
    yylval = c - '0';
    return DIGIT;
}
```

Figure 3. The actions in this *yacc* specification evaluate real numbers during parsing.

The tree in Figure 4 illustrates the evaluation of the real number `321.789`. Starting in the bottom-left corner of the figure, the lexical analyzer sets the attribute value 3 at the leftmost leaf for the token `DIGIT`. The parent of this leaf is based on the production and action

```
integerPart :  DIGIT  { $$ = $1; }  ;
```

This action is omitted from the specification in Figure 3 because, by default, the parser sets `$$`, the attribute of the left side, to `$1`, the attribute of the first symbol on the right side.

Working up the tree, the next node is based on

```
integerPart : integerPart DIGIT { $$ = $integerPart*10 + $DIGIT; } ;
```

The effect of the actions is perhaps easier to see at the nodes for `fraction`. At the only node based on

```
fraction :  DIGIT  { $$ = $DIGIT*0.1; }  ;
```

the value of `$DIGIT` is 9 and the value of `$fraction` is 0.9.

The attributes at the nodes in Figure 4 are said to be synthesized because they are defined solely in terms of the attributes at the children of the node. Since *yacc* generates bottom-up parsers, bottom-up evaluation of syn-

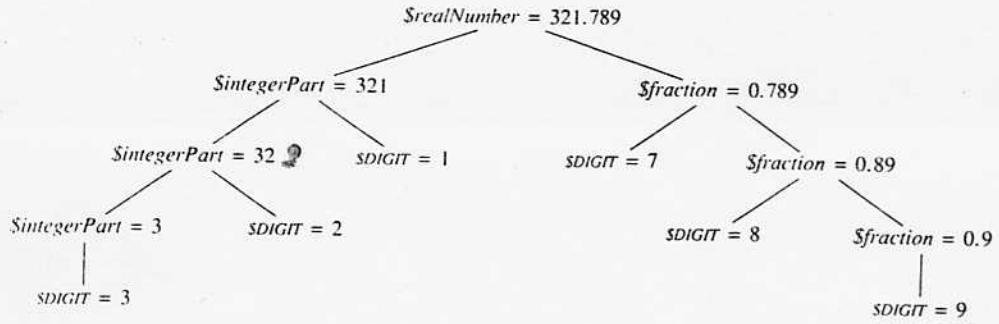


Figure 4. Attribute values during the evaluation of `321.789`.

thesized attributes fits naturally with parsing. Actions can also be used to simulate some "inherited attributes," which are context dependent.

Actions attached to productions are examined further in Section 2; their execution order becomes significant when they do input/output, assign values to variables, call functions with side effects, or otherwise affect the state of a computation.

1.6. A Style for Specifications

As in any language, choose a style that makes the code easy to read, preferably a consistent (and accepted) style that can be read by others. The main concern in a *yacc* specification is to make the productions visible through the morass of action code.

The following style hints owe much to Brian Kernighan:

- Use all capital letters for token names, all lower case letters for nonterminal names. This hint comes under the heading of "knowing who to blame when things go wrong."
- Put productions and actions on separate lines. Either can then be changed independently.
- Put all productions with the same left side together. Put the left side in only once, and let all following productions begin with a vertical bar.
- Put a semicolon only after the last production with a given left side, and put the semicolon on a separate line. New productions can then be easily added.
- Indent production bodies by one tab stop, and action bodies by two tab stops.

The examples in this paper sometimes deviate from this style to conserve space.

2. Evaluation And Translation Of Expressions

Both actions and productions must be considered when a *yacc* specification is designed. Without actions, a parser would silently analyze input strings, complaining only if it detects an error. With suitable actions, a parser can become an evaluator or a translator. Typically, the desired actions influence the choice of productions.

For example, the actions for evaluating the integer and fractional parts of a real number motivate different productions for the sequences of digits represented by `integerPart` and `fraction` in Section 1. In the integer part, the contribution of a digit depends on the number of digits to its right; the contribution of 3 in `321.789` is 300, where the number of zeros depends on the number of digits to its right. In the fractional part, however, the contribution of a digit depends on the number of digits to its left; the contribution of 9 in `321.789` is 0.009, where the number of zeros depends on the number of digits to its left.

Arithmetic expressions are a fertile source of examples for *yacc* because the specifications of expression evaluators and translators are quite short, and the ideas carry over to richer languages. This section begins with a

specification for an expression evaluator. The evaluator benefits from *yacc*'s facilities for specifying the associativity and precedence of operators within expressions. The next example, a translator from infix into postfix notation, illustrates parsing order.

2.1. Grammars for Expressions

Expressions are characterized by the operators within them; the choice of productions for expressions depends on the associativity and precedence of operators.

An operator OP is *left associative* if an expression

$$\texttt{expr}_1\ \texttt{OP}\ \texttt{expr}_2\ \texttt{OP}\ \texttt{expr}_3$$

is evaluated as if it were parenthesized as

$$(\texttt{expr}_1\ \texttt{OP}\ \texttt{expr}_2)\ \texttt{OP}\ \texttt{expr}_3$$

Similarly, the operator is *right associative* if the expression is evaluated as if it were parenthesized as

$$\texttt{expr}_1\ \texttt{OP}\ (\texttt{expr}_2\ \texttt{OP}\ \texttt{expr}_3)$$

An operator OPA has *lower precedence* than an operator OPB if the following equivalences hold (that is, the expressions to the left and right of the ≡ signs have the same values):

$$\texttt{expr}_1\ \texttt{OPA}\ \texttt{expr}_2\ \texttt{OPB}\ \texttt{expr}_3 \equiv \texttt{expr}_1\ \texttt{OPA}\ (\texttt{expr}_2\ \texttt{OPB}\ \texttt{expr}_3)$$
$$\texttt{expr}_1\ \texttt{OPB}\ \texttt{expr}_2\ \texttt{OPA}\ \texttt{expr}_3 \equiv (\texttt{expr}_1\ \texttt{OPB}\ \texttt{expr}_2)\ \texttt{OPA}\ \texttt{expr}_3$$

The traditional grammar for arithmetic expressions uses three nonterminals `expr`, `term`, and `factor`, where `expr` represents an expression and `term` and `factor` represent subexpressions. Tokens `NUMBER` and `VAR` represent numbers and variables. The grammar is

```
%token NUMBER VAR
%%
expr   :  expr '+' term  |  expr '-' term  |  term
       ;
term   :  term '*' factor  |  term '/' factor  |  factor
       ;
factor :  NUMBER  |  VAR  |  '(' expr ')'
       ;
```

In words, an expression is a sequence of terms separated by + or − signs. A term is a sequence of factors separated by * or / signs. Thus,

```
b*b - 4*a*c
```

is an expression containing two terms `b*b` and `4*a*c`. The term `4*a*c` has three factors 4, a, and c. A factor is either a number, a variable, or a parenthesized expression.

The grammar for expressions dates back to Backus's introduction of BNF [4], a notation for writing grammars.

A desk calculator can be based on this grammar by adding actions, as in

```
expr :  expr '-' term  { $$ = $expr - $term; } ;
```

This production and action respect the left associativity of the minus operator, and correctly evaluate 7−1−2 to 4 — check it by drawing a parse tree.

The traditional grammar generalizes to operators at $n \geq 1$ precedence levels; all operators at the same level have the same associativity and precedence. Set up a nonterminal $\texttt{expr}_i$ for precedence level i, with level 1 being the lowest. In the traditional grammar, the nonterminals `expr`, `term`, and `factor` correspond to $\texttt{expr}_1$, $\texttt{expr}_2$, and $\texttt{expr}_3$, respectively. If the operators at level $i < n$ are left associative, then the productions for $\texttt{expr}_i$ have the form

```
%{
#define YYSTYPE double
%}
%token NUMBER
%left '+' '-'
%left '*' '/'
%right '^'
%left UMINUS
%%
lines :  lines expr '\n'        { printf("%g\n", $expr); }
      |  lines '\n'
      |  /* empty */
      ;
expr  :  expr '+' expr          { $$ = $1 + $3; }
      |  expr '-' expr          { $$ = $1 - $3; }
      |  expr '*' expr          { $$ = $1 * $3; }
      |  expr '/' expr          { $$ = $1 / $3; }
      |  expr '^' expr          { $$ = pow($1, $3); }
      |  '-' expr %prec UMINUS  { $$ = - $expr; }
      |  '(' expr ')'           { $$ = $expr; }
      |  NUMBER
      ;
```

Figure 5. An expression evaluator based on precedence declarations for tokens.

```
expr_i :  expr_i OP expr_i+1  ;
```

Here, OP represents an operator at level i. Otherwise, if the operators at level i are right associative, then the productions have the form

```
expr_i :  expr_i+1 OP expr_i  ;
```

At each precedence level $i < n$, there is an additional production of the form

```
expr_i :  expr_i+1  ;
```

2.2. Associativity and Precedence Declarations

Yacc has special facilities for declaring the associativity and precedence of operators, which will be introduced by considering the specification in Figure 5.

The tokens of the evaluator in Figure 5 are parentheses, NUMBER, and the operators

```
'+' '-' '*' '/' '^' UMINUS
```

(Ignore UMINUS for the moment.)

The associativity and precedence of tokens are declared on lines beginning with one of the keywords %left, %right, or %nonassoc. All of the tokens on a line have the same associativity and precedence; they have lower precedence than the tokens on successive lines. These declarations will be referred to as *precedence* declarations.

(The keyword %nonassoc describes operators that do not associate with themselves; for example,

```
A .LT. B .LT. C
```

is illegal in Fortran because .LT. is nonassociative.)

The precedence declarations

```
%%
#include <stdio.h>
#include <ctype.h>
#include <math.h>
int yylex() {
    int c;
    while ( ( c = getchar() ) == ' ' );
    if ( (c == '.') || (isdigit(c)) ) {
        ungetc(c, stdin);
        scanf("%lf", &yylval);
        return NUMBER;
    }
    return c;
}
```

Figure 6. User routines for the evaluator in Figure 5.

```
%left  '+' '-'
%left  '*' '/'
%right '^'
```

specify that + and − are left associative and have lower precedence than the left-associative operators * and /, which in turn have lower precedence than the right-associative operator ^.

The operator ^ in the grammar represents exponentiation, as in 2^3, which evaluates to 8. This operator is right associative; thus, 2^2^3 is equivalent to 2^(2^3), 2^8, and 256.

The nonterminals of the grammar are `lines` and `expr`. The grammar expects a sequence of expressions, on separate lines. A typical production for `expr` has the form

```
expr :  expr '+' expr   { $$ = $1 + $3; }
```

Alternatively, we can write the action as

```
expr :  expr '+' expr   { $$ = $expr#1 + $expr#2; }
```

With these precedence declarations, the expression

```
2 ^ 2 ^ 3 * 4 - 5 * 6 - 7 * 8
```

is evaluated as if it were parenthesized as

```
( (2^(2^3))*4  -  5*6 )  -  7*8
```

The user routines for the evaluator are in Figure 6. The lexical analyzer `yylex` returns a token each time it is called. It skips blanks. If it sees a digit or a decimal point, it returns the token `NUMBER` after using the C library function `scanf` to read a number into the global variable `yylval` (as mentioned in Section 1, the attribute value, if any, associated with a token must be left in `yylval`). Otherwise, the lexical analyzer returns a single character as a token.

The parser uses precedence declarations to decide when to apply a production. Suppose that the input has the form

```
expr * expr ···
```

and the parser has to decide whether or not to apply the multiplication production

```
expr :  expr '*' expr
```

If the next symbol in the input is +, as in

```
expr * expr  + ···
```

the parser applies the multiplication production because the token + has lower precedence than *. However, if the next symbol in the input is ^, the parser defers the multiplication production because ^ has higher precedence than *.

The treatment of the unary minus operator deserves special mention. The evaluator in Figure 5 accepts the expression `10^-1` in lieu of $10^{-1} \equiv 0.1$. In other words, `10^-1` is treated like `10^(-1)`, with unary minus having higher precedence than ^. The precedence of the minus operator therefore depends on whether it is used as a binary or as a unary operator.

Yacc provides the keyword `%prec` for overriding the declared precedence of a token. A `%left` declaration in Figure 5 gives – the lowest precedence, along with +. The keyword `%prec` in

```
expr :  '-' expr  %prec UMINUS
```

overrides the declared precedence of –. When this production is applied, the high precedence of token `UMINUS` is used instead. The expression `3-10^-1` is therefore equivalent to `3-(10^(-1))`.

When several tokens appear in a production, the parser normally uses the precedence of the last token on the right side to decide whether to apply a production. The `%prec` keyword overrides this normal behavior. For example, the `%prec` in

```
expr :  '(' TYPENAME ')' expr    %prec TYPENAME
```

is necessary for the production to take the precedence of token `TYPENAME`; otherwise the production takes its precedence from the closing parenthesis.

It is recommended that precedence declarations be used in a "cookbook" fashion, until some experience is gained. How *yacc* uses precedence declarations is examined further in Section 4.

2.3. Execution Order for Actions

The execution order of actions is significant because actions can have side effects. One way to visualize the order is to imagine a traversal of a parse tree in which the children of each node are visited depth-first from left to right, starting at the root. Suppose a node has two children c and d, with c to the left of d. *Depth-first* implies that all the nodes in the subtree for c are visited before any nodes are visited in the subtree for d. The tree in Figure 7 includes actions as pseudo-symbols, attached by dashed lines. Actions are executed in the order they would be visited in a depth-first left-to-right traversal.

The parse tree in Figure 7 is based on the following specification of an infix-to-postfix translator:

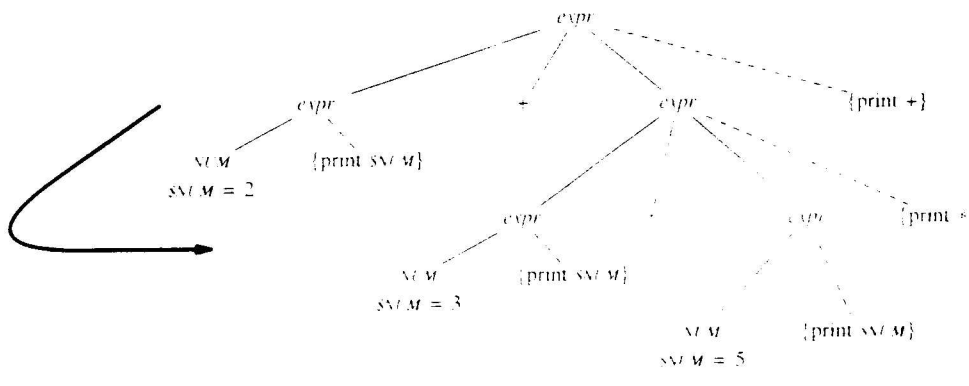


Figure 7. Actions are executed in a depth-first left-to-right order.

```
%token NUM
%left  '+'
%left  '*'
%%
expr : expr '+' expr     { printf(" +"); }
     | expr '*' expr     { printf(" *"); }
     | '(' expr ')'
     | NUM               { printf(" %d", $NUM); }
     ;
```

The translation is emitted incrementally during parsing, so the execution order of the print statements is critical. The translation of 2+3*5 is

```
2 3 5 * +
```

2.4. Actions Embedded Within Rules

Yacc permits an action to be written in the middle of a production as well as at the end; actions in the middle are called *embedded* actions.

Embedded actions are useful for keeping track of context information. For example, consider the typeset text "E_1", specified by the *eqn* input

```
E sub 1
```

The smaller point size of 1, relative to that of E, is dictated by the context; specifically, by the preceding keyword `sub`. The *eqn* grammar uses embedded actions to maintain the current point size in variable `ps`. The embedded action `ps -= del` in

```
box :  box  { ps -= del; }  SUB  box  { ps += del; }
```

reduces the point size before a subscript is processed; the other action `ps += del` restores the point size. Nonterminal `box` represents an *eqn* construct, and token `SUB` represents the input characters `sub`.

Each embedded action is implemented by manufacturing a fresh nonterminal, called a *marker* nonterminal. *Yacc* actually treats this *eqn* example as if it had been written:

```
box  :  box  _ACT  SUB  box
            { ps += del; }
     ;
$ACT :  /* empty */
            { ps -= del; }
     ;
```

The fresh marker nonterminal `_ACT` marks the position of the embedded action `ps -= del`.

Within an embedded action, `$$` refers to the attribute value of its marker nonterminal. Thus, the two occurrences of `$$` in

```
a : b  { $$ = 1; }  c  { x = $2; $$ = $c; } ;
```

refer to different nonterminals, shown explicitly in

```
a    :  b  _ACT  c   { x = $2; $$ = $c; } ;
_ACT :  /* empty */  { $$ = 1; } ;
```

In words, the effect of

```
a : b  { $$ = 1; }  c  { x = $2; $$ = $c; } ;
```

is to make 1 the attribute value of the implicit marker nonterminal in position 2, assign 1 to variable `x`, and make `$c` the attribute value of the left side `a`.

Note that `c` is at position 3, so the above production can be rewritten using `$3` instead of `$c`:

```
a : b  { $$ = 1; }  c  { x = $2; $$ = $3; }
```

3. How The Parser Works

The algorithm used to go from a grammar to a parser is complex and will not be discussed here, but the parser itself is relatively simple. Its two main actions are

- shift to the next input symbol and
- reduce by applying a production.

Some familiarity with such actions is helpful in deciphering messages about "shift/reduce" and "reduce/reduce" conflicts, which warn of potential ambiguities in the grammar that could lead the parser astray.

Yacc places a human-readable description of the generated parser into a file `y.output`, when it is invoked with the `-v` (for verbose) option. This section deals with the parsing background behind `y.output` files; parsing conflicts themselves are considered in Section 4.

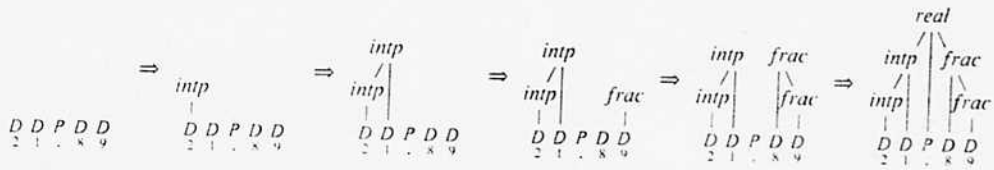
The running example in this section is the following grammar for real numbers:

```
%token D P
%%
real :  intp P frac    ;
intp :  D   |  intp D   ;
frac :  D   |  D frac   ;
```

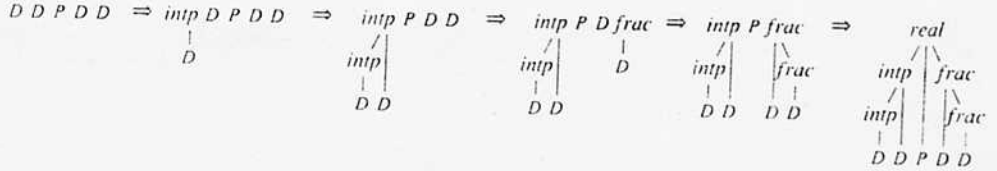
The abbreviated names `intp` for integer part, `frac` for fraction, and `D` for digit, conserve space in diagrams. The use of token `P` for a decimal point `'.'` avoids confusion with other uses of dots within `y.output` files.

3.1. Shift-Reduce Parsing

A *bottom-up* parser works from the leaves (bottom) of a parse tree towards the root. The following sequence of tree snapshots illustrates a bottom-up parse of the token stream DDPDD, corresponding to the real number `21.89`. For the moment, the digit represented by a token `D` appears below the token. The trees are



Let us redraw these trees to line up their uncovered portions: that is, the roots of the completed subtrees. The redrawn sequence is



The uncovered portions suffice, as long as the grammar is unambiguous. The real-number grammar is indeed unambiguous, so the preceding sequence of partial trees is characterized by the snapshots

D D P D D ⇒ intp D P D D ⇒ intp P D D ⇒ intp P D frac ⇒ intp P frac ⇒ real

These snapshots correspond to a sequence of reduce actions; a *reduce* action replaces the right side of a production by its left side. A *shift* action advances the parser to the next unexamined input token; such tokens are called *lookahead* symbols.

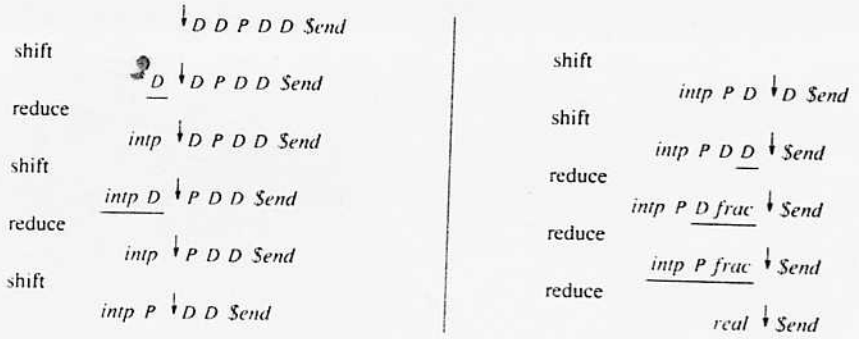


Figure 8. Shift-reduce parsing of DDPDD.

The key problem of shift-reduce parsing is that of deciding when to shift and when to reduce; *yacc* generates tables for this purpose that are explained later in this section. Meanwhile, an example of a shift-reduce parse appears in Figure 8. The input token stream is again DDPDD, and a special token `$end` marks its end. In the figure, a pointer appears before the current lookahead symbol. The first action, a shift, advances the pointer past the leftmost D:

↓D D P D D $end shift D ↓D P D D $end

The second action reduces the token D immediately to the left of the pointer (right sides to be reduced are underlined for clarity). The reduction replaces the right side D by its left side `intp`:

D ↓D P D D $end reduce intp ↓D P D D $end

After the next D is shifted, the right side `intp D` is reduced to the left side `intp`:

intp D ↓P D D $end reduce intp ↓P D D $end

Successive shift actions now advance the lookahead pointer all the way to the endmarker. Finally, a sequence of reduce actions completes the parse.

It is no accident that a right side to be reduced always appears immediately to the left of the pointer in Figure 8. This observation is the basis for a stack-implementation of shift-reduce parsing. Informally, the symbols to the left of the pointer are held on a stack, so a right side to be reduced appears at the top of the stack.

3.2. Parser States

Instead of grammar symbols, a *yacc*-generated parser works with states, which encode some parsing context together with a grammar symbol. The context summarizes prior parsing actions. For example, states tell a parser for real numbers that a digit to the left of a decimal point reduces to `intp`, but that a digit to the right reduces to `frac`.

A *state* consists of a collection of items, where an *item* is a production with a dot inserted in the right side — some versions of *yacc* use an underscore in place of the dot. One of the states of the real-number parser is

```
real :  intp.P frac
intp :  intp.D
```

The dot tells us that an `intp` has just been seen, and that the parser expects to see a P or a D.

Figure 9. States and
transitions during red

In this state, th
to obtain the item

```
real :  int
```

Since, `frac` nc
Although they are n
closure, of this state
side):

```
real :  int
frac :  .D
frac :  .D
```

Since both productio
continue adding prod

With this closur

```
frac :  D.
frac :  D.f
```

The parser states
appears in Figure 9.
due to nonterminals, a

For technical rea
derives the old startin
left of the old starting

```
$accept :  .r
```

The closure of state 0

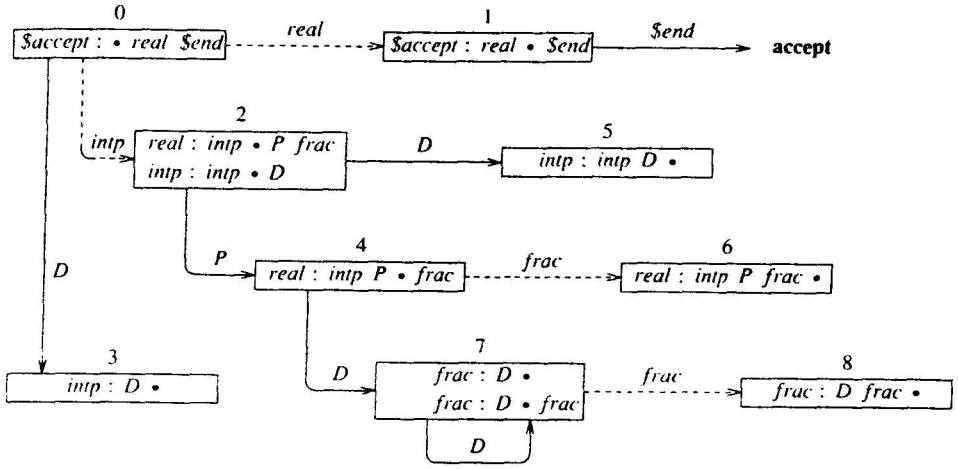


Figure 9. States and transitions for the real-number grammar. The solid arrows are for shifts, and the dashed arrows are for transitions during reductions.

In this state, the parser shifts on lookahead `P`. The shift is recorded by (conceptually) moving the dot past `P` to obtain the item

```
real :  intp P.frac
```

Since, `frac` now appears to the right of the dot, the parser expects the incoming symbols to match a `frac`. Although they are not shown, the productions for `frac` are implicitly carried with the item. The full version, or *closure*, of this state is obtained by adding the productions for `frac` (with a dot at the beginning of the right side):

```
real :  intp P.frac
frac :  .D
frac :  .D frac
```

Since both productions for `frac` begin with the token `D`, no more productions are added; otherwise, we would continue adding productions until all nonterminals to the right of the dot were considered.

With this closure, on lookahead `D`, the parser shifts to a state containing the items

```
frac :  D.
frac :  D.frac
```

The parser states and their transitions constitute an automaton. The automaton for the real-number grammar appears in Figure 9. The solid arrows are for shift transitions, due to tokens. The dashed arrows, for transitions due to nonterminals, are used during reductions.

For technical reasons, *yacc* augments a grammar by adding a new starting nonterminal `$accept`, which derives the old starting nonterminal and an endmarker `$end`. The starting state 0 has an item with a dot to the left of the old starting symbol, as in

```
$accept : .real $end
```

The closure of state 0 contains the items

```
$accept : .real $end
real    : .intp P frac
intp    : .D
intp    : .intp D
```

Since the only token to the right of a dot is D, the very first token must be a D.

Some of the states of the real-number parser in Figure 9 are (informally)

0. *The starting state*. The item `$accept: .real $end` tells us that the entire input, upto the endmarker, must match `real`.

2. *Within the integer part*. An `intp` has been seen. The lookahead token must either be a D (another digit in the integer part) or a P (the decimal point).

7. *Within the fraction part*. Shift as long as the lookahead token is a D. Otherwise, reduce the last D to `frac`.

State 2 is displayed as follows in the `y.output` file for the real-number grammar:

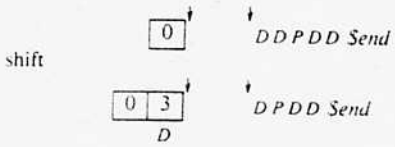
```
state 2
    real :  intp.P frac
    intp :  intp.D

    D  shift 5
    P  shift 4
    .  error
```

After the two items is a summary of the actions in this state. With lookahead D, the parser shifts to state 5, and with lookahead P it shifts to state 4. The default action (represented by ".") is to report an error.

3.3. Parsing Actions

The parser holds states on a stack, with the current state on top. The starting state of the automaton in Figure 9 is state 0. With lookahead D, the automaton shifts to state 3 (in the bottom-left corner of the figure) by pushing 3 onto the stack and removing the lookahead D from the input. For ease of comparison with Figure 8, the symbol D appears below the state in the following diagram:



The top of the state stack has its own pointer, separate from the lookahead pointer to the next input symbol.

The only item in state 3 is

```
intp :  D.
```

Whenever the dot in an item is at the end of the production, one of the possible actions is a reduction by that production. Since this state has no other actions, the parser chooses to reduce.

A reduce action has two phases: (a) pop the states corresponding to the right side, and (b) push a state corresponding to the left side. The following diagram illustrates the reduction of the leading D in DDPDD to `intp`:

pop

goto

The parser h
is as follows:

1. Based on its
If it needs on

2. Using the cur

a) A *shift* a
token.

b) A *reduce*
stack a
uncovered
to *r* (see t

c) The *accep*
when the

d) An *error*
That is, the
that would

4. Ambiguity and C

A *shift/reduce*
larly, a *reduce/reduce*

Conflicts definite
parse tree. Conflicts
yacc is capable of co
middle of a productio

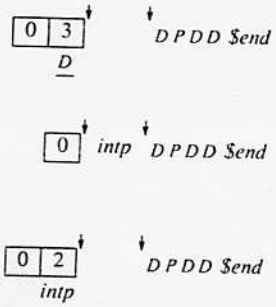
Yacc produces a
called *disambiguating*

1. In a shift/reduce

2. In a reduce/reduce

Although the effe
conflicts, experience su

The rest of this se
the intended effect, and
the `y.output` file crea



The parser has only four actions available to it, called shift, reduce, accept, and error. A move of the parser is as follows:

1. Based on its current state, the parser decides whether it needs a lookahead token to choose the next action. If it needs one, and does not already have one, it calls `yylex` to obtain the next token.

2. Using the current state p, and the lookahead token t if needed, the parser chooses an action.

   a) A *shift* action to state q is done by pushing state q onto the state stack and clearing the lookahead token.

   b) A *reduce* action by a production is done in two phases. In the first phase, the parser pops from the stack a number of states equal to the number of symbols on the right side. Let q be the state uncovered after the states are popped, and let the nonterminal on the left side of the production take q to r (see the dashed arrows in Figure 9). In the second phase, the parser pushes state r onto the stack.

   c) The *accept* action occurs after the entire input has been seen and matched. This action occurs only when the lookahead token is the endmarker `$end`.

   d) An *error* action occurs when the parser can no longer continue parsing according to the productions. That is, the input tokens seen so far and the current lookahead cannot possibly be followed by anything that would result in a legal input. See Section 5 for error recovery.

4. Ambiguity and Conflicts

A *shift/reduce conflict* occurs if the parser cannot decide between a shift action and a reduce action. Similarly, a *reduce/reduce conflict* occurs if the parser cannot decide between two legal reductions.

Conflicts definitely occur when a grammar is *ambiguous*; that is, if some input string has more than one parse tree. Conflicts can also occur when a grammar, although consistent, requires a more complex parser than *yacc* is capable of constructing. Finally, conflicts are sometimes introduced when an action is embedded into the middle of a production.

Yacc produces a parser even when conflicts occur. Rules used to choose between two competing actions are called *disambiguating rules*. The default disambiguating rules are:

1. In a shift/reduce conflict, the default is to shift.

2. In a reduce/reduce conflict, the default is to reduce by the earlier production in the specification.

Although the effect of disambiguating rules can often be achieved by rewriting the grammar to avoid conflicts, experience suggests that this rewriting is somewhat unnatural and produces slower parsers.

The rest of this section considers two situations, one in which the default disambiguating rules do not have the intended effect, and one in which they do. It is recommended that shift/reduce conflicts be investigated using the `y.output` file created by running *yacc* with the `-v` option.

4.1. To Shift or To Reduce

Yacc reports 2 shift/reduce conflicts when it is applied to

```
%token NUM
%%
expr    :  expr '\n'        { printf("\n"); }
        |  expr '-' expr    { printf(" -"); }
        |  NUM              { printf(" %g", $NUM); }
        ;
```

The hope here is to translate an infix expression, terminated by a newline, into postfix notation. Thus, we want

```
2 - 1 - 1 \n
```

to be translated into

```
2 1 - 1 - \n
```

The answer would be 0 if the expression were evaluated by subtracting 1 from 2, and then subtracting 1 from the result. Unfortunately, the output of the parser (with a suitable user-routines section) is

```
2 1 1 \n - -
```

What happened? The problem can be traced to the preference for shift in a shift/reduce conflict, which makes - right associative, and gives \n higher precedence than -.

A slightly edited version of the `y.output` file for this grammar appears in Figure 10. Unfortunately, both productions and states are numbered, leaving room for confusion. The action

```
.   reduce 2
```

refers to ***production*** 2, whereas the action

```
\n  shift 3
```

refers to ***state*** 3.

The two shift/reduce conflicts are in state 5. In this state, by default, the parser shifts the lookahead symbols \n and - instead of using the item

```
expr :  expr - expr.     (2)
```

to reduce by production (2). This reduction can occur only when the right side is on top of the stack, so, when the conflict occurs, the stack must contain states corresponding to

...	$expr_a$	-	$expr_b$

(The subscripts merely distinguish between the two occurrences of `expr`.)

The other items in state 5 tell us more about the choices faced by the parser. With lookahead \n the parser shifts, by the default disambiguating rule. Thus, it treats $expr_b$ on top of the stack as if it were in the item

```
expr :  exprb.\n
```

instead of the item

```
expr :  expra - exprb.     (2)
```

Similarly, with lookahead -, the parser treats $expr_b$ as if it were the first subexpression in

```
expr :  exprb.- expr
```

Putting these observations together, the input `2-1-1\n` results in the stack eventually containing

```
expr - expr - expr \n
```

```
state 0
    $accept : .expr $end
    NUM  shift 2
    .  error
    expr  goto 1

state 1
    $accept :  expr.$end
    expr :  expr.\n
    expr :  expr.- expr
    $end  accept
    \n  shift 3
    -  shift 4
    .  error

state 2
    expr :  NUM.     (3)
    .  reduce 3

state 3
    expr :  expr \n.     (1)
    .  reduce 1

state 4
    expr :  expr -.expr
    NUM  shift 2
    .  error
    expr  goto 5

5: conflict (shift \n, reduce 2)
5: conflict (shift -, reduce 2)
state 5
    expr :  expr.\n
    expr :  expr.- expr
    expr :  expr - expr.     (2)
    \n  shift 3
    -  shift 4
    .  reduce 2
```

Figure 10. A slightly edited `y.output` file.

A sequence of reductions now occurs; the corresponding actions print a newline and two minus signs.[2]

The addition of the precedence declarations

```
%nonassoc '\n'
%left     '-'
```

eliminates the conflicts in state 5, resulting in the new state

```
state 5
    expr :  expr.\n
    expr :  expr.- expr
    expr :  expr - expr.     (2)
    .  reduce 2
```

Here, the reduction takes precedence over the potential shifts, thereby implementing the left associativity of - and its higher precedence over \n.

4.2. Precedence Declarations

Precedence declarations give rise to disambiguating rules for resolving parsing conflicts.

As mentioned in Section 2, a precedence declaration starts with `%left`, `%right`, or `%nonassoc`. These keywords specify the associativity of the tokens in a declaration. All the tokens in a declaration have the same precedence; tokens in successive declarations have higher precedence. A token in a precedence declaration need not be, but can be, declared by `%token` as well.

[2] This explanation assumes that the newline is the last character of the input. If it is not, the parser will wait for more input after reducing the right side `expr\n`. At this point, the stack contains

```
expr - expr - expr
```

The parser needs more input before it can choose whether to shift on newline, shift on minus, or reduce on any other token.

The disambiguating rules are as follows.

1. Although declared only for tokens and literals, precedence information is attached to each production as well. The precedence and associativity of a production is that of the last token or literal on its right side. The `%prec` construction overrides this default.

2. If there is a shift/reduce conflict, and both the lookahead symbol (to be shifted) and the production (to be reduced) have precedence declarations, then the conflict is resolved in favor of the action (shift or reduce) with the higher precedence. If the precedences are the same, then the associativity is used; left associative implies reduce, right associative implies shift, and nonassociative implies error.

3. In the absence of precedence information, the default disambiguating rules given earlier in this section are used. More precisely, suppose that there is no precedence information for either the lookahead symbol or a production to be reduced by. A shift/reduce conflict is resolved by shifting. A reduce/reduce conflict is resolved by reducing by the production that appears earlier in the specification. Conflicts resolved by these default rules are reported.

Conflicts resolved by precedence are not counted in the shift/reduce and reduce/reduce conflicts reported by *yacc*. Thus, mistakes in precedence declarations can mask errors in the design of the productions.

4.3. Shift a Dangling Else

As an example of the power of the default disambiguating rules consider a fragment from a programming language involving an if-then-else construction:

```
stmt :  IF '(' expr ')' stmt
     |  IF '(' expr ')' stmt ELSE stmt
     ;
```

Here, `IF` and `ELSE` are tokens, `stmt` is a nonterminal for statements, and `expr` is a nonterminal for expressions. Call the first production the *simple-if* production and the second the *else-if* production.

These two productions are ambiguous, since

```
IF ( expr1 ) IF ( expr2 ) stmta ELSE stmtb
```

can be structured in two ways

```
IF ( expr1 ) {                  |  IF ( expr1 ) {
    IF ( expr2 ) stmta          |      IF ( expr2 ) stmta
}                               |      ELSE stmtb
ELSE stmtb                      |  }
```

The second interpretation, with an `ELSE` matching the nearest preceding unmatched `IF`, is the one taken by most programming languages. It is the interpretation obtained when shift/reduce conflicts are resolved in favor of shift actions.

The `y.output` file for a grammar containing the simple-if and if-else productions contains a shift/reduce conflict, illustrated by

```
8: shift/reduce conflict (shift 9, red'n 1) on ELSE
state 8
    stmt :  IF ( expr ) stmt.      (1)
    stmt :  IF ( expr ) stmt.ELSE stmt

    ELSE  shift 9
    .  reduce 1
```

This conflict occurs when the lookahead symbol is `ELSE` and the parser stack contains the right side of the simple-if production

```
... IF ( expr ) stmt
```


The parser chooses

```
... IF ( e
```

to be successfully r
lookahead ELSE) w

```
... stmt E
```

which cannot be par

A shift on look
contents

```
... IF ( e
```

and lookahead ELSE

```
... IF ( e
```

A reduction by the it

```
... IF ( e
```

which can be reduced

5. Error Handling

Error handling is
found, it may be nece
alter symbol-table ent

It is seldom acce
parsing continues. B
card tokens from the

Yacc provides a
reserved for error han
planned. When an er
It then behaves as if
then reset to the token
detected.

In order to prevent
until three tokens have
error state, no message

For example, a pr

```
stmt :  erro
```

would, in effect, mean
error was seen. More
statement, and start pro
it may make a false st
no error.

Actions can be use
symbol table space, etc

Productions with
are productions such as

```
stmt :  error
```

The parser chooses to shift the ELSE rather than reduce by the simple-if production. This choice allows

```
··· IF ( expr ) stmt ELSE stmt
```

to be successfully reduced by the if-else production. Note that a premature application of the simple-if rule (with lookahead ELSE) would lead to the stack contents

```
··· stmt ELSE stmt
```

which cannot be parsed further.

A shift on lookahead ELSE also matches an ELSE with the nearest preceding unmatched IF. With stack contents

```
··· IF ( expr ) IF ( expr ) stmt
```

and lookahead ELSE, the parser shifts, so the stack eventually holds

```
··· IF ( expr ) IF ( expr ) stmt ELSE stmt
```

A reduction by the if-else production now yields

```
··· IF ( expr ) stmt
```

which can be reduced by the simple-if production.

5. Error Handling

Error handling is an extremely difficult area, and many of the problems are semantic ones. When an error is found, it may be necessary to undo the effect of actions — for example, to reclaim parse-tree storage, to delete or alter symbol-table entries — and, typically, set flags to avoid generation of further output.

It is seldom acceptable to stop all processing when an error is found; further syntax errors might be found if parsing continues. But, how do we get the parser "restarted" after an error is detected. One approach is to discard tokens from the input until parsing can be continued.

Yacc provides a simple, but reasonably general, feature for discarding tokens. The token name `error` is reserved for error handling. On the right side of a production, `error` suggests a place where error recovery is planned. When an error occurs, the parser pops its stack until it enters a state where the token `error` is legal. It then behaves as if `error` were the current lookahead, and performs the action encountered. The lookahead is then reset to the token that caused the error. If no error productions are specified, parsing halts when an error is detected.

In order to prevent a cascade of error messages, the parser, after detecting an error, remains in the error state until three tokens have been successfully read and shifted. If an error is detected when the parser is already in error state, no message is given, and the input token is quietly deleted.

For example, a production

```
stmt : error
```

would, in effect, mean that on a syntax error the parser would attempt to skip over the statement in which the error was seen. More precisely, the parser will scan ahead, looking for three tokens that might legally follow a statement, and start processing at the first of these; if the beginnings of statements are not sufficiently distinctive, it may make a false start in the middle of a statement, and end up reporting a second error where there is in fact no error.

Actions can be used with these special error rules. These actions might attempt to reinitialize tables, reclaim symbol table space, etc.

Productions with just `error` on the right side are very general, but difficult to control. Somewhat easier are productions such as

```
stmt : error ';'
```

Here, upon error, the parser attempts to skip over the statement, but will do so by skipping to the next semicolon. All tokens after the error and before the next semicolon cannot be shifted, and are discarded. When the semicolon is seen, this right side will be reduced, and any "cleanup" action associated with it performed.

Another form of error production arises in interactive applications, where it may be desirable to permit a line to be reentered after an error. A possible error production might be

```
input :  error '\n'
             {  printf("Reenter last line: ");  }
         input
             { $$ = $input;  }
      ;
```

One potential difficulty with this approach is that the parser must correctly process three input tokens before it admits that it has correctly resynchronized after the error. If the reentered line contains an error in the first two tokens, the parser deletes the offending tokens, and gives no message; this is clearly unacceptable. For this reason, there is a mechanism that can be used to force the parser to believe that an error has been fully recovered from. The statement

```
yyerrok;
```

in an action resets the parser to its normal mode. The last example is better written

```
input :  error '\n'
             { yyerrok; printf("Reenter last line: "); }
         input
             { $$ = $input; }
      ;
```

As mentioned above, the token seen immediately after the `error` symbol is the input token at which the error was discovered. Sometimes, this is inappropriate; for example, an error recovery action might take upon itself the job of finding the correct place to resume input. In this case, the previous lookahead token must be cleared. The statement

```
yyclearin;
```

in an action has this effect. For example, suppose the action after error is to call some sophisticated resynchronization routine, supplied by the user, that attempts to advance the input to the beginning of the next valid statement. After this routine is called, the next token returned by `yylex` is presumably the first token in a legal statement; the old, illegal token must be discarded, and the error state reset. This can be done by a rule like

```
stmt :  error
            { resynch(); yyerrok; yyclearin; }
     ;
```

These mechanisms are admittedly crude, but do allow for a simple, fairly effective recovery of the parser from many errors; moreover, the user can get control to deal with the error actions required by other portions of the program.

6. The Yacc Environment

From a specification, *yacc* creates a file of C programs, called `y.tab.c` on most systems.

6.1. Program Organization

Consider a specification file of the form

```
%{
        ⟨user supplied code within declarations⟩
#define YYSTYPE ⟨desired type⟩
%}
        ⟨declarations section⟩
%%
        ⟨productions⟩
%%
        ⟨user-routines section⟩
```

From this specification, *yacc* creates a file `y.tab.c`, organized as follows:

```
        ⟨user supplied code within declarations⟩
#define YYSTYPE ⟨desired type⟩
        ⟨token and other declarations⟩
        ⟨user-routines section⟩
        ⟨parser tables⟩
yyparse() {   ···  }
```

Actions are incorporated into `yyparse`.

As mentioned in Section 1, `yyparse`, the code for the parser, expects the following functions to be supplied:

```
int yylex() {
    a lexical analyzer; returns a token
}
int main(···) {
    ···  yyparse();  ···
}
void yyerror(s) char *s; {
    print an error message pointed to by s
}
```

The function `main` decides when it wants to call `yyparse`, which returns 0 if the parser accepts, and 1 if an error is detected and no error recovery is possible.

A user-supplied function `yyerror` is called with a string containing an error message. Applications typically do more than simply print the message; for example, the message might be accompanied by the input line number on which the error was detected. The external integer variable `yychar` contains the lookahead token number at the time an error is detected; this may be of some interest in giving better diagnostics.

The external variable `yydebug` is normally set to 0. It it is set to a nonzero value, the parser will output a verbose description of its actions, including the tokens read and the parser actions. Depending on the operating environment, it may be possible to set this variable by using a debugging system.

6.2. Lexical Tie-Ins

The lexical analyzer `yylex` must return an integer, the token number, representing the lookahead token. An attribute value associated with the token must be placed in the global variable `yylval`. The lexical-analyzer generator *lex* [10] can be used together with *yacc*.

The parser and the lexical analyzer must agree on the token numbers in order for communication between them to take place. Token numbers may be chosen by *yacc*, or chosen by the user. In either case, the `#define` mechanism of C is used to allow the lexical analyzer to refer to these numbers symbolically. For example, the declarations section

```
token IF ELSE
```

leads to the following definitions in the file `y.tab.c`:

```
# define IF 257
# define ELSE 258
```

If `yylex` is included in the user-routines section, it is within the scope of these definitions, so `IF` and `ELSE` can be used as the names of token numbers in `yylex`.

A file `y.tab.h` containing the definition of token numbers can be created by running *yacc* with the `-d` option.

The approach of treating token names as defined constants leads to clear, easily modified lexical analyzers; the only pitfall is the need to avoid using any token names that are reserved or significant in C or the parser. For example, the use of the token names `if` and `while` will almost certainly cause severe difficulties when the lexical analyzer is compiled. The token name `error` is reserved for error handling; see Section 5.

Yacc chooses token numbers if the user does not. The default token number for a literal character is the numerical value of the character in the local character set. Other names are assigned token numbers starting at 257.

To assign a number to a token (including a literal), the first appearance of the token name or literal in the declarations section can be immediately followed by a nonnegative integer. This integer is taken to be the token number of the name or literal. Names and literals not defined by this mechanism retain their default definition. It is important that all token numbers be distinct.

For historical reasons, the endmarker must have token number 0 or negative. This token number cannot be redefined by the user; thus, all lexical analyzers must be prepared to return 0 or negative as a token number upon reaching the end of their input.

6.3. Communicating Context to the Lexical Analyzer

Some lexical decisions depend on context. For example, the lexical analyzer might want to delete blanks normally, but not within quoted strings. Or names might be entered into a symbol table in declarations, but not in expressions.

One way of handling this situation is to create a global flag that is examined by the lexical analyzer, and set by actions. For example, suppose a program consists of zero or more declarations, followed by zero or more statements. Consider:

```
%{
    int dflag;
%}
... other declarations ...
%%
prog  :  decls stmts
      ;
decls :  /* empty */          { dflag = 1; }
      |  decls  declaration
      ;
stmts :  /* empty */          { dflag = 0; }
      |  stmts  statement
      ;
... other productions ...
```

The flag `dflag` is now 1 when reading declarations and 0 when reading statements, *except for the first token in the first statement.* This token must be seen by the parser before it can tell that the declarations have ended and the productions have begun. In many cases, this single token exception does not affect the lexical scan.

This kind of "backdoor" approach can be elaborated to a noxious degree. Nevertheless, it represents a way of doing some things that are difficult, if not impossible, to do otherwise.

Some programming languages permit the user to use words like `if`, which are normally reserved, as label or variable names, provided that such use does not conflict with the legal use of these names in the programming

language. This is extremely hard to do in the framework of *yacc*; it is difficult to pass information to the lexical analyzer telling it "this instance of `if` is a keyword, and that instance is a variable". The user can make a stab at it, using flags like `dflag`, above, but it is difficult. It is better that the keywords be *reserved*; that is, be forbidden for use as variable names. There are powerful stylistic reasons for preferring this, anyway.

6.4. Support for Arbitrary Attribute Types

By default, the values returned by actions and the lexical analyzer are integers. Other types can be supported by defining `YYSTYPE` in the declarations section, as in

```
%{
#define YYSTYPE double
%}
```

`YYSTYPE` is not a normal variable, because the parser contains the following lines to define it to be `int` if it has not already been defined by the user:

```
#ifndef YYSTYPE
#define YYSTYPE int
#endif
```

Clearly, `YYSTYPE` can be defined to be any type, including a union type.

The `%union` mechanism of *yacc* attempts to make the underlying union transparent, in all but a few places where *yacc* needs help in determining which field of the union is intended.

Unions are declared in the declarations section, an example being

```
%union {
    int    ival;
    double dval;
    char * sval;
}
```

A union type with these members is created for the *yacc* value stack, and for the global variables `yylval` and `yyval`. With the `-d` option, *yacc* copies the union type into the `y.tab.h` file. The type of the union can be referred to as `YYSTYPE`.

The type of each attribute must now correspond to one of the union members. The construction

```
<name>
```

indicates a union member name. If the construction follows `%token`, `%left`, `%right`, or `%nonassoc`, then the union member name is associated with the tokens in that declaration. Another keyword `%type` is used similarly to associate union member names with nonterminals. Thus, we might use

```
%type <dval> expr term
```

There remain a couple of cases where these mechanisms are insufficient. If there is an action within a rule, the value returned by this action has no a priori type. Similarly, reference to left context values leaves *yacc* with no easy way of knowing the type. In this case, a type can be imposed on the reference by inserting a union member name, between `<` and `>`, immediately after the first `$` and immediately before the symbol name or number. An example of this usage is

```
rule :  aaa  {  $<intval>$  =  3;  }
        bbb  {  fun( $<intval>2, $<other>0 );  }
    ;
```

where the union member names `intval` and `other` are inserted within references. This syntax has little to recommend it, but the situation arises rarely.

The facilities in this subsection are not triggered until they are used: in particular, the use of `%type` will turn on these mechanisms. When they are used, there is a fairly strict level of checking. For example, use of `$`*n*

or $$ to refer to something with no defined type is diagnosed. If these facilities are not triggered, the *yacc* value stack is used to hold `int`s, as was true historically.

7. Acknowledgements

The original acknowledgements, from [6], are as follows. "*Yacc* owes much to a most stimulating collection of users, who have goaded me beyond my inclination, and frequently beyond my ability, in their endless search for 'one more feature'. Their irritating unwillingness to learn how to do things my way has usually led to my doing things their way; most of the time, they have been right. B. W. Kernighan, P. J. Plauger, S. I. Feldman, C. Imagna, M. E. Lesk, and A. Snyder will recognize some of their ideas in the current version of *yacc*. C. B. Haley contributed to the error recovery algorithm. D. M. Ritchie, B. W. Kernighan, and M. O. Harris helped translate this document into English. Al Aho also deserves special credit for bringing the mountain to Mohammed, and other favors."

This version of *yacc* has benefited from thoughtful comments by B. W. Kernighan, M. F. Fernandez, and M. Tasman.

8. References

1. Aho, A.V. and Johnson, S.C. LR parsing. *ACM Computing Surveys 6*, 2 (1974), 99-124.
2. Aho, A.V., Johnson, S.C., and Ullman, J.D. Deterministic parsing of ambiguous grammars. *CACM 18*, 8 (1975), 441-452.
3. Aho, A.V., Sethi, R., and Ullman, J.D. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass, 1986.
4. Backus, J.W. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference. In *International Conference on Information Processing, June 1959*, Unesco, Paris, 1960, pp. 125-132.
5. Johnson, S.C. A Portable Compiler: Theory and Practice. In *Proc. 5th ACM Symp. on Principles of Programming Languages*, January 1978.
6. Johnson, S.C. Yet Another Compiler Compiler. In *Unix Programmer's Manual*, Vol. 2, M.D. McIlroy and B.W. Kernighan, Eds. AT&T Bell Laboratories, Murray Hill, NJ 07974, 1979.
7. Kernighan, B.W. and Cherry, L.L. A system for typesetting mathematics. *CACM 18*, 3 (1975), 151-157.
8. Kernighan, B.W. and Pike, R. *The UNIX Programming Environment*. Prentice-Hall, 1984.
9. Kernighan, B.W. and Ritchie, D.M. *The C Programming Language*. Prentice Hall, Englewood Cliffs, NJ, 1988. 2nd Edition.
10. Lesk, M.E. and Schmidt, M. Lex — A Lexical Analyzer Generator. In *Unix Programmer's Manual, Tenth Edition*, AT&T Bell Laboratories, 1989.
11. Schreiner, A.T. and Friedman, H.G. Jr. *Introduction to Compiler Construction with UNIX*. Prentice-Hall, Englewood Cliffs, N.J, 1985.
12. Stroustrup, B. *The C++ Programming Language*. Addison-Wesley, 1986.

Appendix A. Yacc Input Syntax

This appendix has a description of the *yacc* input syntax, as a *yacc* specification. Context dependencies, etc., are not considered. Ironically, the *yacc* input specification language is most naturally specified as an LR(2) grammar; the sticky part comes when an identifier is seen in a rule, immediately following an action. If this identifier is followed by a colon, it is the start of the next rule; otherwise it is a continuation of the current rule, which just happens to have an action embedded in it. As implemented, the lexical analyzer looks ahead after seeing an identifier, and decide whether the next token (skipping blanks, newlines, comments, etc.) is a colon. If so, it returns the token `C_IDENTIFIER`. Otherwise, it returns `IDENTIFIER`. Literals (quoted strings) are also returned as `IDENTIFIER`s, but never as part of `C_IDENTIFIER`s.


```
%token
%token
%token

%token LE
%token MA
%token LC
%token RC

%start s

%%
spec    :
        ;
tail    :
        |
        ;
defs    :
        |
        ;
def     :

        ;
rword   :

        ;
tag     :
        |
        ;
nlist   :
        |
        |
        ;
nmno    :
        |
        ;
```

```
        /* grammar for the input to yacc */

        /* basic entities */
%token  IDENTIFIER  /* includes identifiers and literals */
%token  C_IDENTIFIER/* identifier (but not literal) followed by colon */
%token  NUMBER      /* [0-9]+ */

        /* reserved words: %type => TYPE, %left => LEFT, etc. */

%token LEFT  RIGHT  NONASSOC  TOKEN  PREC  TYPE  START  UNION

%token MARK   /*  the  %%  mark  */
%token LCURL  /*  the  %{  mark  */
%token RCURL  /*  the  %}  mark  */

        /*  ascii  character  literals  stand  for  themselves  */

%start  spec


%%
spec    :  defs  MARK  rules  tail
        ;

tail    :  MARK  {     In this action, eat up the rest of the file     }
        |  /*  empty:  the  second  MARK  is  optional  */
        ;

defs    :  /*  empty  */
        |  defs  def
        ;

def     :  START  IDENTIFIER
        |  UNION  {  Copy union definition to output  }
        |  LCURL  {  Copy C code to output file  }  RCURL
        |  ndefs  rword  tag  nlist
        ;

rword   :  TOKEN
        |  LEFT
        |  RIGHT
        |  NONASSOC
        |  TYPE
        ;

tag     :  /*  empty:  union  tag  is  optional  */
        |  '<'  IDENTIFIER  '>'
        ;

nlist   :  nmno
        |  nlist  nmno
        |  nlist  ','  nmno
        ;

nmno    :  IDENTIFIER              /*  NOTE:  literal  illegal  with  %type  */
        |  IDENTIFIER  NUMBER      /*  NOTE:  illegal  with  %type  */
        ;
```

```
        /*  rules  section  */
rules   :  C_IDENTIFIER  rbody  prec
        |  rules  rule
        ;

rule    :  C_IDENTIFIER  rbody  prec
        |  '|'  rbody  prec
        ;

rbody   :  /*  empty  */
        |  rbody  IDENTIFIER
        |  rbody  act
        ;

act :   '{'  {  Copy action, translate , etc.  }  '}'
        ;

prec    :  /*  empty  */
        |  PREC  IDENTIFIER
        |  PREC  IDENTIFIER  act
        |  prec  ';'
        ;
```