-

YACC – Yet Another Compiler-Compiler

Stephen C. Johnson

Bell Laboratories,
Murray Hill, New Jersey 07974

ABSTRACT

Computer program input generally has some structure; in fact, every computer program which does input can be thought of as defining an ''input language'' which it accepts. The input languages may be as complex as a programming language, or as simple as a sequence of numbers. Unfortunately, standard input facilities are restricted, difficult to use and change, and do not completely check their inputs for validity.

Yacc provides a general tool for controlling the input to a computer program. The Yacc user describes the structures of his input, together with code which is to be invoked when each such structure is recognized. Yacc turns such a specification into a subroutine which may be invoked to handle the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user´s application handled by this subroutine.

The input subroutine produced by Yacc calls a user supplied routine to return the next basic input item. Thus, the user can specify his input in terms of individual input characters, or, if he wishes, in terms of higher level constructs such as names and numbers. The user supplied routine may also handle idiomatic features such as comment and continuation conventions, which typically defy easy specification.

Yacc is written in C[7], and runs under UNIX. The subroutine which is output may be in C or in Ratfor[4], at the user´s choice; Ratfor permits translation of the output subroutine into portable Fortran[5]. The class of specifications accepted is a very general one, called LALR(1) grammars with disambiguating rules. The theory behind Yacc has been described elsewhere[1,2,3].

Yacc was originally designed to help produce the ''front end'' of compilers; in addition to this use, it has been successfully used in many application programs, including a phototypesetter language, a document retrieval system, a Fortran debugging system, and the Ratfor compiler.

YACC – Yet Another Compiler-Compiler

Stephen C. Johnson

Bell Laboratories,
Murray Hill, New Jersey 07974

Section 0: Introduction

Yacc provides a general tool for imposing structure on the input to a computer program. The Yacc user prepares a specification of the input process; this includes rules which describe the input structure, code which is to be invoked when these structures are recognized, and a low-level routine to do the basic input. Yacc then produces a subroutine to do the input procedure; this subroutine, called a *parser,* calls the user-supplied low-level input routine (called the *lexical analyzer)* to pick up the basic items (called *tokens)* from the input stream. These tokens are organized according to the input structure rules, called *grammar rules;* when one of these rules has been recognized, then the user code supplied for this rule, called an *action,* is invoked; actions have the ability to return values and make use of the values of other actions.

The heart of the input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. For example, one grammar rule might be

```
date : month_name day ´,´ year ;
```

Here, date, month_name, day, and year represent structures of interest in the input process; presumably, month_name, day, and year are defined elsewhere. The comma “,” is quoted by single quotes; this implies that the comma is to appear literally in the input. The colon and semicolon merely serve as punctuation in the rule, and have no significance in controlling the input. Thus, with proper definitions, the input

```
July 4, 1776
```

might be matched by the above rule.

As we mentioned above, an important part of the input process is carried out by the lexical analyzer. This user routine reads the true input stream, recognizing those structures which are more conveniently or more efficiently recognized directly, and communicates these recognized tokens to the parser. For historical reasons, the name of a structure recognized by the lexical analyzer is called a *terminal symbol* name, while the name of a structure recognized by the parser is called a *nonterminal symbol* name. To avoid the obvious confusion of terminology, we shall usually refer to terminal symbol names as *token names.*

There is considerable leeway in deciding whether to recognize structures by the lexical analyzer or by a grammar rule. Thus, in the above example it would be possible to have other rules of the form

```
month_name : ´J´ ´a´ ´n´ ;
month_name : ´F´ ´e´ ´b´ ;

    . . .

month_name : ´D´ ´e´ ´c´ ;
```

Here, the lexical analyzer would only need to recognize individual letters, and month_name would be a nonterminal symbol. Rules of this sort tend to be a bit wasteful of time and space, and may even restrict the power of the input process (although they are easy to write). For a more efficient input process, the lexical analyzer itself might recognize the month names, and return an indication that a month_name was seen; in this case, month_name would be a token.

Literal characters, such as “,”, must also be passed through the lexical analyzer, and are considered tokens.

As an example of the flexibility of the grammar rule approach, we might add to the above specifications the rule

```
date : month ´/´ day ´/´ year ;
```

and thus optionally allow the form

```
7/4/1776
```

as a synonym for

```
July 4, 1776
```

In most cases, this new rule could be “slipped in” to a working system with minimal effort, and a very small chance of disrupting existing input.

Frequently, the input being read does not conform to the specifications due to errors in the input. The parsers produced by Yacc have the very desirable property that they will detect these input errors at the earliest place at which this can be done with a left-to-right scan; thus, not only is the chance of reading and computing with bad input data substantially reduced, but the bad data can usually be quickly found. Error handling facilities, entered as part of the input specifications, frequently permit the reentry of bad data, or the continuation of the input process after skipping over the bad data.

In some cases, Yacc fails to produce a parser when given a set of specifications. For example, the specifications may be self contradictory, or they may require a more powerful recognition mechanism than that available to Yacc. The former cases probably represent true design errors; the latter cases can often be corrected by making the lexical analyzer more powerful, or by rewriting some of the grammar rules. The class of specifications which Yacc can handle compares very favorably with other systems of this type; moreover, the constructions which are difficult for Yacc to handle are also frequently difficult for human beings to handle. Some users have reported that the discipline of formulating valid Yacc specifications for their input revealed errors of conception or design early in the program development.

The next several sections describe the basic process of preparing a Yacc specification; Section 1 describes the preparation of grammar rules, Section 2 the preparation of the user supplied actions associated with these rules, and Section 3 the preparation of lexical analyzers. In Section 4, we discuss the diagnostics produced when Yacc is unable to produce a parser from the given specifications. This section also describes a simple, frequently useful mechanism for handling operator precedences. Section 5 discusses error detection and recovery. Sections 6C and 6R discuss the operating environment and special features of the subroutines which Yacc produces in C and Ratfor, respectively. Section 7 gives some hints which may lead to better designed, more efficient, and clearer specifications. Finally, Section 8 has a brief summary. Appendix A has a brief example, and Appendix B tells how to run Yacc on the UNIX operating system. Appendix C has a brief description of mechanisms and syntax which are no longer actively supported, but which are provided for historical continuity with older versions of Yacc.

Section 1: Basic Specifications

As we noted above, names refer to either tokens or nonterminal symbols. Yacc requires those names which will be used as token names to be declared as such. In addition, for reasons which will be discussed in Section 3, it is usually desirable to include the lexical analyzer as part of the specification file; it may be useful to include other programs as well. Thus, every specification file consists of three sections: the *declarations, (grammar) rules,* and *programs.* The sections are separated by double percent “%%” marks. (The per-cent “%” is generally used in Yacc specifications as an escape character.)

In other words, a full specification file looks like

```
declarations
%%
rules
%%
programs
```

The declaration section may be empty. Moreover, if the programs section is omitted, the second %% mark may be omitted also; thus, the smallest legal Yacc specification is

```
%%
rules
```

Blanks, tabs, and newlines are ignored except that they may not appear in names or multi-character reserved symbols. Comments may appear wherever a name or operator is legal; they are enclosed in /* . . . */, as in C and PL/I.

The rules section is made up of one or more grammar rules. A grammar rule has the form:

```
A : BODY ;
```

A represents a nonterminal name, and BODY represents a sequence of zero or more names and literals. Notice that the colon and the semicolon are Yacc punctuation.

Names may be of arbitrary length, and may be made up of letters, dot ".", underscore "_", and non-initial digits. Notice that Yacc considers that upper and lower case letters are distinct. The names used in the body of a grammar rule may represent tokens or nonterminal symbols.

A literal consists of a character enclosed in single quotes "´". As in C, the backslash "\" is an escape character within literals, and all the C escapes are recognized. Thus

```
´\n´   represents newline
´\r´   represents return
´\´´   represents single quote "´"
´\\´   represents backslash "\"
´\t´   represents tab
´\b´   represents backspace
´\xxx´ represents "xxx" in octal
```

For a number of technical reasons, the nul character (´\0´ or 000) should never be used in grammar rules.

If there are several grammar rules with the same left hand side, the vertical bar "|" can be used to avoid rewriting the left hand side. In addition, the semicolon at the end of a rule can be dropped before a vertical bar. Thus the grammar rules

```
A : B C D ;
A : E F ;
A : G ;
```

can be given to Yacc as

```
A :  B C D |
     E F |
     G ;
```

It is not necessary that all grammar rules with the same left side appear together in the grammar rules section, although it makes the input much more readable, and easy to change.

If a nonterminal symbol matches the empty string, this can be indicated in the obvious way:

```
empty : ;
```

As we mentioned above, names which represent tokens must be declared as such. The simplest way of doing this is to write

```
%token   name1 name2 . . .
```

in the declarations section. (See Sections 3 and 4 for much more discussion). Every name not defined in the declarations section is assumed to represent a nonterminal symbol. If, by the end of the rules section, some nonterminal symbol has not appeared on the left of any rule, then an error message is produced and Yacc halts.

The left hand side of the *first* grammar rule in the grammar rules section has special importance; it is taken to be the controlling nonterminal symbol for the entire input process; in technical language it is called the *start symbol.* In effect, the parser is designed to recognize the start symbol; thus, this symbol generally represents the largest, most general structure described by the grammar rules.

The end of the input is signaled by a special token, called the *endmarker.* If the tokens up to, but not including, the endmarker form a structure which matches the start symbol, the parser subroutine returns to its caller when the endmarker is seen; we say that it *accepts* the input. If the endmarker is seen in any other context, it is an error.

It is the job of the user supplied lexical analyzer to return the endmarker when appropriate; see section 3, below. Frequently, the endmarker token represents some reasonably obvious I/O status, such as ‘‘end-of-file’’ or ‘‘end-of-record’’.

Section 2: Actions

To each grammar rule, the user may associate an action to be performed each time the rule is recognized in the input process. This action may return a value, and may obtain the values returned by previous actions in the grammar rule. In addition, the lexical analyzer can return values for tokens, if desired.

When invoking Yacc, the user specifies a programming language; currently, Ratfor and C are supported. An action is an arbitrary statement in this language, and as such can do input and output, call subprograms, and alter external vectors and variables (recall that a ‘‘statement’’ in both C and Ratfor can be compound and do many distinct tasks). An action is specified by an equal sign ‘‘=’’ at the end of a grammar rule, followed by one or more statements, enclosed in curly braces ‘‘{’’ and ‘‘}’’. For example,

```
A: ´(´ B ´)´ = { hello( 1, "abc" );  }
```

and

```
XXX: YYY ZZZ =
        {
                printf("a message\n");
                flag = 25;
        }
```

are grammar rules with actions in C. A grammar rule with an action need not end with a semicolon; in fact, it is an error to have a semicolon before the equal sign.

To facilitate easy communication between the actions and the parser, the action statements are altered slightly. The symbol ‘‘dollar sign’’ ‘‘$’’ is used as a signal to Yacc in this context.

To return a value, the action normally sets the pseudo-variable ‘‘$$’’ to some integer value. For example, an action which does nothing but return the value 1 is

```
= { $$ = 1; }
```

To obtain the values returned by previous actions and the lexical analyzer, the action may use the (integer) pseudo-variables $1, $2, . . ., which refer to the values returned by the components of the right side of a rule, reading from left to right. Thus, if the rule is

```
A: B C D ;
```

for example, then $2 has the value returned by C, and $3 the value returned by D.

As a more concrete example, we might have the rule

```
expression: ´(´ expression ´)´ ;
```

We wish the value returned by this rule to be the value of the expression in parentheses. Then we write

```
expression: ´(´ expression ´)´   = { $$ = $2 ; }
```

As a default, the value of a rule is the value of the first element in it ($1). This is true even if there is no explicit action given for the rule. Thus, grammar rules of the form

```
A: B ;
```

frequently need not have an explict action.

Notice that, although the values of actions are integers, these integers may in fact contain pointers (in C) or indices into an array (in Ratfor); in this way, actions can return and reference more complex data structures.

Sometimes, we wish to get control before a rule is fully parsed, as well as at the end of the rule. There is no explicit mechanism in Yacc to allow this; the same effect can be obtained, however, by introducing a new symbol which matches the empty string, and inserting an action for this symbol. For example, we might have a rule describing an "if" statement:

```
statement: IF ´(´ expr ´)´ THEN statement
```

Suppose that we wish to get control after seeing the right parenthesis in order to output some code. We might accomplish this by the rules:

```
statement:  IF ´(´ expr ´)´ actn THEN statement
        = { call action1 }

actn:   /* matches the empty string */
        = { call action2 }
```

Thus, the new nonterminal symbol actn matches no input, but serves only to call action2 after the right parenthesis is seen.

Frequently, it is more natural in such cases to break the rule into parts where the action is needed. Thus, the above example might also have been written

```
statement:  ifpart THEN statement
        = { call action1 }

ifpart:     IF ´(´ expr ´)´
        = { call action2 }
```

In many applications, output is not done directly by the actions; rather, a data structure, such as a parse tree, is constructed in memory, and transformations are applied to it before output is generated. Parse trees are particularly easy to construct, given routines which build and maintain the tree structure desired. For example, suppose we have a C function "node", written so that the call

```
node( L, n1, n2 )
```

creates a node with label L, and descendants n1 and n2, and returns a pointer to the newly created node. Then we can cause a parse tree to be built by supplying actions such as:

```
expr: expr ´+´ expr
        = { $$ = node( ´+´, $1, $3 ); }
```

in our specification.

The user may define other variables to be used by the actions. Declarations and definitions can appear in two places in the Yacc specification: in the declarations section, and at the head of the rules sections, before the first grammar rule. In each case, the declarations and definitions are enclosed in the marks "%{" and "%}". Declarations and definitions placed in the declarations section have global scope, and are thus known to the action statements and the lexical analyzer. Declarations and definitions placed at the head of the rules section have scope local to the action statements. Thus, in the above example, we might have included

```
%{ int variable 0; %}
```

in the declarations section, or, perhaps,

```
%{ static int variable; %}
```

at the head of the rules section. If we were writing Ratfor actions, we might want to include some COMMON statements at the beginning of the rules section, to allow for easy communication between the actions and other routines. For both C and Ratfor, Yacc has used only external names beginning in "yy"; the user should avoid such names.

Section 3: Lexical Analysis

The user must supply a lexical analyzer which reads the input stream and communicates tokens (with values, if desired) to the parser. The lexical analyzer is an integer valued function called yylex, in both C and Ratfor. The function returns an integer which represents the type of the token. The value to be associated in the parser with that token is assigned to the integer variable yylval. Thus, a lexical analyzer written in C should begin

```
yylex ( ) {
        extern int yylval;
        . . .
```

while a lexical analyzer written in Ratfor should begin

```
integer function yylex(yylval)
        integer yylval
        . . .
```

Clearly, the parser and the lexical analyzer must agree on the type numbers in order for communication between them to take place. These numbers may be chosen by Yacc, or chosen by the user. In either case, the "define" mechanisms of C and Ratfor are used to allow the lexical analyzer to return these numbers symbolically. For example, suppose that the token name DIGIT has been defined in the declarations section of the specification. The relevant portion of the lexical analyzer (in C) might look like:

```
yylex( ) {
        extern int yylval;
        int c;
        . . .
        c = getchar( );
        . . .
        if( c >= ´0´ && c <= ´9´ ) {
                yylval = c−´0´;
                return(DIGIT);
        }
        . . .
```

The relevant portion of the Ratfor lexical analyzer might look like:

```
integer function yylex(yylval)
        integer yylval, digits(10), c
        . . .
        data digits(1) / "0" /;
        data digits(2) / "1" /;
        . . .
        data digits(10) / "9" /;
        . . .
#   set c to the next input character
        . . .
        do i = 1, 10 {
                if(c .EQ. digits(i)) {
                        yylval = i−1
                        yylex = DIGIT
                        return
                }
        }
        . . .
```

In both cases, the intent is to return a token type of DIGIT, and a value equal to the numerical value of the digit. Provided that the lexical analyzer code is placed in the programs section of the specification, the identifier DIGIT will be redefined to be equal to the type number associated with the token name DIGIT.

This mechanism leads to clear and easily modified lexical analyzers; the only pitfall is that it makes it important to avoid using any names in the grammar which are reserved or significant in the chosen language; thus, in both C and Ratfor, the use of token names of ‘‘if’’ or ‘‘yylex’’ will almost certainly cause severe difficulties when the lexical analyzer is compiled. The token name ‘‘error’’ is reserved for error handling, and should not be used naively (see Section 5).

As mentioned above, the type numbers may be chosen by Yacc or by the user. In the default situation, the numbers are chosen by Yacc. The default type number for a literal character is the numerical value of the character, considered as a 1 byte integer. Other token names are assigned type numbers starting at 257. It is a difficult, machine dependent operation to determine the numerical value of an input character in Ratfor (or Fortran). Thus, the Ratfor user of Yacc will probably wish to set his own type numbers, or not use any literals in his specification.

To assign a type number to a token (including literals), 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 type number of the name or literal. Names and literals not defined by this mechanism retain their default definition. It is important that all type numbers be distinct.

There is one exception to this situation. For sticky historical reasons, the endmarker must have type number 0. Note that this is not unattractive in C, since the nul character is returned upon end of file; in Ratfor, it makes no sense. This type number cannot be redefined by the user; thus, all lexical analyzers should be prepared to return 0 as a type number upon reaching the end of their input.

Section 4: Ambiguity, Conflicts, and Precedence

A set of grammar rules is *ambiguous* if there is some input string which can be structured in two or more different ways. For example, the grammar rule

expr : expr ´–´ expr ;

is a natural way of expressing the fact that one way of forming an arithmetic expression is to put two other expressions together with a minus sign between them. Unfortunately, this grammar rule does not completely specify the way that all complex inputs should be structured. For example, if we have input of the form

expr – expr – expr

the rule would permit us to treat this input either as

(expr – expr) – expr

or as

expr – (expr – expr)

(We speak of the first as *left association* of operators, and the second as *right association).*

Yacc detects such ambiguities when it is attempting to build the parser. It is instructive to consider the problem that confronts the parser when it is given an input such as

expr – expr – expr

When the parser has read the second expr, the input which it has seen:

expr – expr

matches the right side of the grammar rule above. One valid thing for the parser to do is to *reduce* the input it has seen by applying this rule; after applying the rule, it would have reduced the input it had already seen to expr (the left side of the rule). It could then read the final part of the input:

– expr

and again reduce by the rule. We see that the effect of this is to take the left associative interpretation.

Alternatively, when the parser has seen

expr – expr

it could defer the immediate application of the rule, and continue reading (the technical term is *shifting)* the input until it had seen

expr – expr – expr

It could then apply the grammar rule to the rightmost three symbols, reducing them to expr and leaving

expr – expr

Now it can reduce by the rule again; the effect is to take the right associative interpretation. Thus, having read

expr – expr

the parser can do two legal things, a shift or a reduction, and has no way of deciding between them. We refer to this as a *shift/reduce conflict.* It may also happen that the parser has a choice of two legal reductions; this is called a *reduce/reduce conflict.*

When there are shift/reduce or reduce/reduce conflicts, Yacc still produces a parser. It does this by selecting one of the valid steps wherever it has a choice. A rule which describes which choice to make in a given situation is called a *disambiguating rule.*

Yacc has two disambiguating rules which are invoked by default, in the absence of any user directives to the contrary:

1. In a shift/reduce conflict, the default is to do the shift.
2. In a reduce/reduce conflict, the default is to reduce by the *earlier* grammar rule (in the input sequence).

Rule 1 implies that reductions are deferred whenever there is a choice, in favor of shifts. Rule 2 gives the user rather crude control over the behavior of the parser in this situation, but the proper use of reduce/reduce conflicts is still a black art, and is properly considered an advanced topic.

Conflicts may arise because of mistakes in input or logic, or because the grammar rules, while consistent, require a more complex parser than Yacc can construct. In these cases, the application of disambiguating rules is inappropriate, and leads to a parser which is in error. For this reason, Yacc always reports the number of shift/reduce and reduce/reduce conflicts which were resolved by Rule 1 and Rule 2.

In general, whenever it is possible to apply disambiguating rules to produce a correct parser, it is also possible to rewrite the grammar rules so that the same inputs are read, but there are no conflicts. For this reason, most previous systems like Yacc have considered conflicts to be fatal errors. Our experience has suggested that this rewriting is somewhat unnatural to do, and produces slower parsers; thus, Yacc will produce parsers even in the presence of conflicts.

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

```
stat :  IF ´(´ cond ´)´ stat |
        IF ´(´ cond ´)´ stat ELSE stat ;
```

Here, we consider IF and ELSE to be tokens, cond to be a nonterminal symbol describing conditional (logical) expressions, and stat to be a nonterminal symbol describing statements. In the following, we shall refer to these two rules as the *simple-if* rule and the *if-else* rule, respectively.

These two rules form an ambiguous construction, since input of the form

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

can be structured according to these rules in two ways:

```
IF ( C1 ) {
        IF ( C2 ) S1
}
ELSE S2
```

or

```
IF ( C1 ) {
        IF ( C2 ) S1
        ELSE S2
}
```

The second interpretation is the one given in most programming languages which have this construct. Each ELSE is associated with the last preceding "un-ELSE'd" IF. In this example, consider the situation where the parser has seen

```
IF ( C1 ) IF ( C2 ) S1
```

and is looking at the ELSE. It can immediately *reduce* by the simple-if rule to get

IF (C1) stat

and then read the remaining input,

ELSE S2

and reduce

IF (C1) stat ELSE S2

by the if-else rule. This leads to the first of the above groupings of the input.

On the other hand, we may *shift* the ELSE and read S2, and then reduce the right hand portion of

IF (C1) IF (C2) S1 ELSE S2

by the if-else rule to get

IF (C1) stat

which can be reduced by the simple-if rule. This leads to the second of the above groupings of the input, which is usually desired.

Once again the parser can do two valid things − we have a shift/reduce conflict. The application of disambiguating rule 1 tells the parser to shift in this case, which leads to the desired grouping.

Notice that this shift/reduce conflict arises only when there is a particular current input symbol, ELSE, and particular inputs already seen, such as

IF (C1) IF (C2) S1

In general, there may be many conflicts, and each one will be associated with an input symbol and a set of previously read inputs. The previously read inputs are characterized by the *state* of the parser, which is assigned a nonnegative integer. The number of states in the parser is typically two to five times the number of grammar rules.

When Yacc is invoked with the verbose (−v) option (see Appendix B), it produces a file of user output which includes a description of the states in the parser. For example, the output corresponding to the above example might be:

```
23: shift/reduce Conflict (Shift 45, Reduce 18) on ELSE

State 23

        stat : IF ( cond ) stat_
        stat : IF ( cond ) stat_ELSE stat

        ELSE    shift 45
        .       reduce 18
```

The first line describes the conflict, giving the state and the input symbol. The state title follows, and a brief description of the grammar rules which are active in this state. The underline "_" describes the portions of the grammar rules which have been seen. Thus in the example, in state 23 we have seen input which corresponds to

IF (cond) stat

and the two grammar rules shown are active at this time. The actions possible are, if the input symbol is ELSE, we may shift into state 45. In this state, we should find as part of the description a line of the form

stat : IF (cond) stat ELSE_stat

because in this state we will have read and shifted the ELSE. Back in state 23, the alternative action, described by ".", is to be done if the input symbol is not mentioned explicitly in the above actions; thus, in this case, if the input symbol is not ELSE, we should reduce by grammar rule 18, which is presumably

    stat : IF ´(´ cond ´)´ stat

Notice that the numbers following "shift" commands refer to other states, while the numbers following "reduce" commands refer to grammar rule numbers. In most states, there will be only one reduce action possible in the state, and this will always be the default command. The user who encounters unexpected shift/reduce conflicts will probably want to look at the verbose output to decide whether the default actions are appropriate. In really tough cases, the user might need to know more about the behavior and construction of the parser than can be covered here; in this case, a reference such as [1] might be consulted; the services of a local guru might also be appropriate.

There is one common situation where the rules given above for resolving conflicts are not sufficient; this is in the area of arithmetic expressions. Most of the commonly used constructions for arithmetic expressions can be naturally described by the notion of *precedence* levels for operators, together with information about left or right associativity. It turns out that ambiguous grammars with appropriate disambiguating rules can be used to create parsers which are faster and easier to write than parsers constructed from unambiguous grammars. The basic notion is to write grammar rules of the form

    expr : expr OP expr

and

    expr : UNARY expr

for all binary and unary operators desired. This creates a very ambiguous grammar, with many parsing conflicts. As disambiguating rules, the user specifies the precedence, or binding strength, of all the operators, and the associativity of the binary operators. This information is sufficient to allow Yacc to resolve the parsing conflicts in accordance with these rules, and construct a parser which realizes the desired precedences and associativities.

The precedences and associativities are attached to tokens in the declarations section. This is done by a series of lines beginning with a Yacc keyword: %left, %right, or %nonassoc, followed by a list of tokens. All of the tokens on the same line are assumed to have the same precedence level and associativity; the lines are listed in order of increasing precedence or binding strength. Thus,

```
%left ´+´ ´−´
%left ´*´ ´/´
```

describes the precedence and associativity of the four arithmetic operators. Plus and minus are left associative, and have lower precedence than star and slash, which are also left associative. The keyword %right is used to describe right associative operators, and the keyword %nonassoc is used to describe operators, like the operator .LT. in Fortran, which may not associate with themselves; thus,

    A .LT. B .LT. C

is illegal in Fortran, and such an operator would be described with the keyword %nonassoc in Yacc. As an example of the behavior of these declarations, the description

```
%right ´=´
%left ´+´ ´−´
%left ´*´ ´/´

%%

expr :
        expr ´=´ expr |
        expr ´+´ expr |
        expr ´−´ expr |
        expr ´*´ expr |
        expr ´/´ expr |
        NAME ;
```

might be used to structure the input

```
a = b = c*d − e − f*g
```

as follows:

```
a = ( b = ( ((c*d)−e) − (f*g) ) )
```

When this mechanism is used, unary operators must, in general, be given a precedence. An interesting situation arises when we have a unary operator and a binary operator which have the same symbolic representation, but different precedences. An example is unary and binary ´−´; frequently, unary minus is given the same strength as multiplication, or even higher, while binary minus has a lower strength than multiplication. We can indicate this situation by use of another keyword, %prec, to change the precedence level associated with a particular grammar rule. %prec appears immediately after the body of the grammar rule, before the action or closing semicolon, and is followed by a token name or literal; it causes the precedence of the grammar rule to become that of the token name or literal. Thus, to make unary minus have the same precedence as multiplication, we might write:

```
%left ´+´ ´−´
%left ´*´ ´/´

%%

expr :
        expr ´+´ expr |
        expr ´−´ expr |
        expr ´*´ expr |
        expr ´/´ expr |
        ´−´ expr %prec ´*´ |
        NAME ;
```

Notice that the precedences which are described by %left, %right, and %nonassoc are independent of the declarations of token names by %token. A symbol can be declared by %token, and, later in the declarations section, be given a precedence and associativity by one of the above methods. It is true, however, that names which are given a precedence or associativity are also declared to be token names, and so in general do not need to be declared by %token, although it does not hurt to do so.

As we mentioned above, the precedences and associativities are used by Yacc to resolve parsing conflicts; they give rise to disambiguating rules. Formally, the rules work as follows:

1. The precedences and associativities are recorded for those tokens and literals which have them.
2. A precedence and associativity is associated with each grammar rule; it is the precedence and associativity of the last token or literal in the body of the rule. If the %prec construction is used, it overrides this default. Notice that some grammar rules may have no precedence and associativity associated with them.
3. When there is a reduce/reduce conflict, or there is a shift/reduce conflict and either the input symbol or the grammar rule, or both, has no precedence and associativity associated with it, then the two disambiguating rules given at the beginning of the section are used, and the conflicts are reported.
4. If there is a shift/reduce conflict, and both the grammar rule and the input character have precedence and associativity associated with them, then the conflict is resolved in favor of the action (shift or reduce) associated with the higher precedence. If the precedences are the same, then the associativity is used; left associative implies reduce, right associative implies shift, and nonassociating implies error.

There are a number of points worth making about this use of disambiguation. There is no reporting of conflicts which are resolved by this mechanism, and these conflicts are not counted in the number of shift/reduce and reduce/reduce conflicts found in the grammar. This means that occasionally mistakes in the specification of precedences disguise errors in the input grammar; it is a good idea to be sparing with precedences, and use them in an essentially ‘‘cookbook’’ fashion, until some experience has been gained. Frequently, not enough operators or precedences have been specified; this leads to a number of messages about shift/reduce or reduce/reduce conflicts. The cure is usually to specify more precedences, or use the %prec mechanism, or both. It is generally good to examine the verbose output file to ensure that the conflicts which are being reported can be validly resolved by precedence.

Section 5: Error Handling

Error handling is an extremely difficult area, and many of the problems are semantic ones. When an error is found, for example, it may be necessary to reclaim parse tree storage, delete or alter symbol table entries, and, typically, set switches to avoid putting out any further output.

It is generally not acceptable to stop all processing when an error is found; we wish to continue scanning the input to find any further syntax errors. This leads to the problem of getting the parser ‘‘restarted’’ after an error. The general class of algorithms to do this involves reading ahead and discarding a number of tokens from the input string, and attempting to adjust the parser so that input can continue.

To allow the user some control over this process, Yacc provides a simple, but reasonably general, feature. The token name ‘‘error’’ is reserved for error handling. This name can be used in grammar rules; in effect, it suggests places where errors are expected, and recovery might take place. The parser attempts to find the last time in the input when the special token ‘‘error’’ is permitted. The parser then behaves as though it saw the token name ‘‘error’’ as an input token, and attempts to parse according to the rule encountered. The token at which the error was detected remains the next input token after this error token is processed. If no special error rules have been specified, the processing effectively halts when an error is detected.

In order to prevent a cascade of error messages, the parser assumes that, after detecting an error, it remains in 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 error message is given, and the input token is quietly deleted.

As a common example, the user might include a rule of the form

```
statement :  error ;
```

in his specification. This would, in effect, mean that on a syntax error the parser would attempt to skip over the statement in which the error was seen. (Notice, however, that it may be difficult or impossible to tell the end of a statement, depending on the other grammar rules). 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.

The user may supply actions after these special grammar rules, just as after the other grammar rules. These actions might attempt to reinitialize tables, reclaim symbol table space, etc.

The above form of grammar rule is very general, but somewhat difficult to control. Somewhat easier to deal with are rules of the form

```
statement :  error ´;´  ;
```

Here, when there is an error, the parser will again attempt to skip over the statement, but in this case will do so by skipping to the next ";". All tokens after the error and before the next ";" give syntax errors, and are discarded. When the ";" is seen, this rule will be reduced, and any "cleanup" action associated with it will be performed.

Still another form of error rule arises in interactive applications, where we may wish to prompt the user who has incorrectly input a line, and allow him to reenter the line. In C we might write:

```
inputline:    error ´\n´ prompt inputline
        = { $$ = $4; };

prompt:       /* matches no input */
        = {  printf( "Reenter last line: " ); };
```

There is one difficulty with this approach; the parser must correctly process three input tokens before it is prepared to admit that it has correctly resynchronized after the error. Thus, if the reentered line contains errors in the first two tokens, the parser will simply delete the offending tokens, and give no message; this is clearly unacceptable. For this reason, there is a mechanism in both C and Ratfor which can be used to force the parser to believe that resynchronization has taken place. One need only include a statement of the form

```
yyerrok ;
```

in his action after such a grammar rule, and the desired effect will take place; this name will be expanded, using the "# define" mechanism of C or the "define" mechanism of Ratfor, into an appropriate code sequence. For example, in the situation discussed above where we want to prompt the user to produce input, we probably want to consider that the original error has been recovered when we have thrown away the previous line, including the newline. In this case, we can reset the error state before putting out the prompt message. The grammar rule for the nonterminal symbol prompt becomes:

```
prompt:       /* matches no input */
        = {
              yyerrok;
              printf( "Reenter last line: " );
        } ;
```

There is another special feature which the user may wish to use in error recovery. As mentioned above, the token seen immediately after the "error" symbol is the input token at which the error was discovered. Sometimes, this is seen to be 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 user wishes a way of clearing the previous input token held in the parser. One need only include a statement of the form

```
yyclearin ;
```

in his action; again, this expands, in both C and Ratfor, to the appropriate code sequence. For example, suppose the action after error were to call some sophisticated resynchronization routine, supplied by the user, which attempted to advance the input to the beginning of the next valid statement. After this routine was called, the next token returned by yylex would presumably be the first token in a legal statement; we wish to throw away the old, illegal token, and reset the error state. We might do this by the sequence:

```
statement :  error
        = {
                resynch( );
                yyerrok ;
                yyclearin ;
        } ;
```

These mechanisms are admittedly crude, but do allow for a simple, fairly effective recovery of the parser from many errors, and have the virtue that the user can get ‘‘handles’’ by which he can deal with the error actions required by the lexical and output portions of the system.

Section 6C: The C Language Yacc Environment

The default mode of operation in Yacc is to write actions and the lexical analyzer in C. This has a number of advantages; primarily, it is easier to write character handling routines, such as the lexical analyzer, in a language which supports character-by-character I/O, and has shifting and masking operators.

When the user inputs a specification to Yacc, the output is a file of C programs, called ‘‘y.tab.c’’. These are then compiled, and loaded with a library; the library has default versions of a number of useful routines. This section discusses these routines, and how the user can write his own routines if desired. The name of the Yacc library is system dependent; see Appendix B.

The subroutine produced by Yacc is called ‘‘yyparse’’; it is an integer valued function. When it is called, it in turn repeatedly calls ‘‘yylex’’, the lexical analyzer supplied by the user (see Section 3), to obtain input tokens. Eventually, either an error is detected, in which case (if no error recovery is possible) yyparse returns the value 1, or the lexical analyzer returns the endmarker token (type number 0), and the parser accepts. In this case, yyparse returns the value 0.

Three of the routines on the Yacc library are concerned with the ‘‘external’’ environment of yyparse. There is a default ‘‘main’’ program, a default ‘‘initialization’’ routine, and a default ‘‘accept’’ routine, respectively. They are so simple that they will be given here in their entirety:

```
main( argc, argv )
int argc;
char *argv[ ]
{
        yyinit( argc, argv );
        if( yyparse( ) )
                return;
        yyaccpt( );
}

yyinit( ) { }

yyaccpt( ) { }
```

By supplying his own versions of yyinit and/or yyaccpt, the user can get control either before the parser is called (to set options, open input files, etc.) or after the accept action has been done (to close files, call the next pass of the compiler, etc.). Note that yyinit is called with the two ‘‘command line’’ arguments which have been passed into the main program. If neither of these routines is redefined, the default situation simply looks like a call to the parser, followed by the termination of the program. Of course, in many cases the user will wish to supply his own main program; for example, this is necessary if the parser is to be called more than once.

The other major routine on the library is called ‘‘yyerror’’; its main purpose is to write out a message when a syntax error is detected. It has a number of hooks and handles which attempt to make this error message general and easy to understand. This routine is somewhat more complex, but still approachable:

```
extern int yyline;  /* input line number */

yyerror(s)
char *s;
{
        extern int yychar;
        extern char *yysterm[ ];

        printf("\n%s", s );
        if( yyline )
                printf(", line %d,", yyline );
        printf(" on input: ");
        if( yychar >= 0400 )
                printf("%s\n", yysterm[yychar−0400] );
        else switch ( yychar ) {
        case ´\t´: printf( "\\t\n" ); return;
        case ´\n´: printf( "\\n\n" ); return;
        case ´\0´: printf( "$end\n" ); return;
        default: printf( "%c\n" , yychar ); return;
        }
}
```

The argument to yyerror is a string containing an error message; most usually, it is ‘‘syntax error’’. yyerror also uses the external variables yyline, yychar, and yysterm. yyline is a line number which, if set by the user to a nonzero number, will be printed out as part of the error message. yychar is a variable which contains the type number of the current token. yysterm has the names, supplied by the user, for all the tokens which have names. Thus, the routine spends most of its time trying to print out a reasonable name for the input token. The biggest problem with the routine as given is that, on Unix, the error message does not go out on the error file (file 2). This is hard to arrange in such a way that it works with both the portable I/O library and the system I/O library; if a way can be worked out, the routine will be changed to do this. *Beware:* This routine will not work if any token names have been given redefined type numbers. In this case, the user must supply his own yyerror routine. Hopefully, this ‘‘feature’’ will disappear soon.

Finally, there is another feature which the C user of Yacc might wish to use. The integer variable yydebug is normally set to 0. If it is set to 1, the parser will output a verbose description of its actions, including a discussion of which input symbols have been read, and what the parser actions are. Depending on the operating environment, it may be possible to set this variable by using a debugging system.

Section 6R: The Ratfor Language Yacc Environment

For reasons of portability or compatibility with existing software, it may be desired to use Yacc to generate parsers in Ratfor, or, by extension, in portable Fortran. The user is likely to work considerably harder doing this than he might if he were to use C.

When the user inputs a specification to Yacc, and specifies the Ratfor option (see Appendix B), the output is a file of Ratfor programs called ‘‘y.tab.r’’. These programs are then compiled, and provide the desired subroutine.

The subroutine produced by Yacc which does the input process is an integer function called ‘‘yypars’’. When it is called, it in turn repeatedly calls ‘‘yylex’’, the lexical analyzer supplied by the user (see Section 3). Eventually, either an error is detected, in which case (if no error recovery is possible) yypars returns the value 1, or the lexical analyzer returns the endmarker (type number 0), and the parser accepts. In this case, yypars returns 0.

Unlike the C program situation (see Section 6C) there is no library of Ratfor routines which must be used in the loading process. As a side effect of this, *the user must supply a main program which calls yy-*

pars. A suggested Ratfor main program is

```
integer yypars
n = yypars(0)
if( n .EQ. 0 ) {
        . . . here if the program accepted
} else {
        . . . here if there were unrecoverable errors
}
end
```

Notice that there is no easy way for the user to get control when an error is detected, since the Fortran language provides only a very crude character string capability.

There is another feature which the Ratfor user might wish to use. The argument to yypars is normally 0. If it is set to 1, the parser will output a verbose description of its actions, including a discussion of which input symbols have been read, and what the parser actions are. During the input process, the value of this debug flag is kept in a common variable yydebu, which is available to the actions and may be set and reset at will.

Statement labels 1 through 1000 are reserved for the parser, and may not appear in actions; note that, because Ratfor has a more modern control structure than Fortran, it is rarely necessary to use statement labels at all; the most frequent use of labels in Ratfor is in formatted I/O.

Because Fortran has no standard character set and not even a standard character width, it is difficult to produce a lexical analyzer in portable Fortran The usual solution is to provide a routine which does a table search to get the internal type number for each input character, with the understanding that such a routine can be recoded to run far faster for any particular machine.

Finally, we must warn the user that the Ratfor feature of Yacc has been operational for a much shorter time than the other portions of the system. If past experience is any guide, the Ratfor support will develop and become more powerful and better human engineered in response to user complaints and requirements. Thus, the potential Ratfor user might do well to contact the author to discuss his own particular needs.

Section 7. Hints for Preparing Specifications

This section contains miscellaneous hints on preparing efficient, easy to change, and clear specifications. The individual subsections are, more or less, independent; the reader seeing Yacc for the first time may well find that this entire section could be omitted.

Input Style

It is difficult to input rules with substantial actions and still have a readable specification file. The following style hints owe much to Brian Kernighan, and are officially endorsed by the author.

a. Use all capital letters for token names, all lower case letters for nonterminal names. This rule comes under the heading of ‘‘knowing who to blame when things go wrong.‘‘

b. Put grammar rules and actions on separate lines. This allows either to be changed without an automatic need to change the other.

c. Put all rules with the same left hand side together. Put the left hand side in only once, and let all following rules begin with a vertical bar.

d. Indent rule bodies by one tab stop, and action bodies by two tab stops.

The example in Appendix A is written following this style, as are the examples in the text of this paper (where space permits). The user must make up his own mind about these stylistic questions; the central problem, however, is to make the rules visible through the morass of action code.

Common Actions

When several grammar rules have the same action, the user might well wish to provide only one code sequence. A simple, general mechanism is, of course, to use subroutine calls. It is also possible to put a label on the first statement of an action, and let other actions be simply a goto to this label. Thus, if the user had a routine which built trees, he might wish to have only one call to it, as follows:

```
expr :
        expr ´+´ expr =
        {  binary:
                $$ = btree( $1, $2, $3 );
        } |
        expr ´−´ expr =
        {
                goto binary;
        } |
        expr ´*´ expr =
        {
                goto binary;
        } ;
```

Left Recursion

The algorithm used by the Yacc parser encourages so called ‘‘left recursive’’ grammar rules: rules of the form

```
name : name rest_of_rule ;
```

These rules frequently arise when writing specifications of sequences and lists:

```
list :
        item |
        list ´,´ item ;
```

and

```
sequence :
        item |
        sequence item ;
```

Notice that, in each of these cases, the first rule will be reduced for the first item only, and the second rule will be reduced for the second and all succeeding items.

If the user were to write these rules right recursively, such as

```
sequence :
        item |
        item sequence ;
```

the parser would be a bit bigger, and the items would be seen, and reduced, from right to left. More seriously, an internal stack in the parser would be in danger of overflowing if a very long sequence were read. Thus, the user should use left recursion wherever reasonable.

The user should also consider whether a sequence with zero elements has any meaning, and if so, consider writing the sequence specification with an empty rule:

```
sequence :
        | /* empty */
        sequence item ;
```

Once again, the first rule would always be reduced exactly once, before the first item was read, and then the second rule would be reduced once for each item read. Experience suggests that permitting empty sequences leads to increased generality, which frequently is not evident at the time the rule is first written. There are cases, however, when the Yacc algorithm can fail when such a change is made. In effect, conflicts might arise when Yacc is asked to decide which empty sequence it has seen, when it hasn´t seen enough to know! Nevertheless, this principle is still worth following wherever possible.

Lexical Tie-ins

Frequently, there are lexical decisions which depend on the presence of various constructions in the specification. 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 these situations is to create a global flag which is examined by the lexical analyzer, and set by actions. For example, consider a situation where we have a program which consists of 0 or more declarations, followed by 0 or more statements. We declare a flag called ‘‘dflag’’, which is 1 during declarations, and 0 during statements. We may do this as follows:

```
%{
        int dflag ;
%}
%%
program :
        decls  stats ;

decls :
        = /* empty */
        {
                dflag = 1;
        } |
        decls declaration ;

stats :
        = /* empty */
        {
                dflag = 0;
        } |
        stats statement ;

        . . .  other rules . . .
```

The flag dflag is now set to zero when reading statements, and 1 when reading declarations, *except for the first token in the first statement.* This token must be seen by the parser before it can tell that the declaration section has ended and the statements have begun. Frequently, however, this single token exception does not affect the lexical scan required.

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

Bundling

Bundling is a technique for collecting together various character strings so that they can be output at some later time. It is derived from a feature of the same name in the compiler/compiler TMG [6].

Bundling has two components – a nice user interface, and a clever implementation trick. They will be discussed in that order.

The user interface consists of two routines, “bundle” and “bprint”.

```
bundle( a1, a2, . . ., an )
```

accepts a variable number of arguments which are either character strings or bundles, and returns a bundle, whose value will be the concatenation of the values of a1, . . ., an.

```
bprint( b )
```

accepts a bundle as argument and outputs its value.

For example, suppose that we wish to read arithmetic expressions, and output function calls to routines called “add”, “sub”, “mul”, “div”, and “assign”. Thus, we wish to translate

```
a = b – c*d
```

into

```
assign(a,sub(b,mul(c,d)))
```

A Yacc specification file which does this is given in Appendix D; this includes an implementation of the bundle and bprint routines. A rule and action of the form

```
expr:
        expr ´+´ expr =
        {
                $$ = bundle( "add(", $1, ",", $3, ")" );
        }
```

causes the returned value of expr to be come a bundle, whose value is the character string containing the desired function call. Each NAME token has a value which is a pointer to the actual name which has been read. Finally, when the entire input line has been read and the value has been bundled, the value is written out and the bundles and names are cleared, in preparation for the next input line.

Bundles are implemented as arrays of pointers, terminated by a zero pointer. Each pointer either points to a bundle or to a character string. There is an array, called *bundle space,* which contains all the bundles.

The implementation trick is to check the values of the pointers in bundles – if the pointer points into bundle space, it is assumed to point to a bundle; otherwise it is assumed to point to a character string.

The treatment of functions with a variable number of arguments, like bundle, is likely to differ from one implementation of C to another.

In general, one may wish to have a simple storage allocator which allocates and frees bundles, in order to handle situations where it is not appropriate to completely clear all of bundle space at one time.

Reserved Words

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, since it is difficult to pass the required information to the lexical analyzer which tells it “this instance of if is a keyword, and that instance is a variable”. The user can make a stab at it, using the mechanism described in the last subsection, but it is difficult.

A number of ways of making this easier are under advisement, and one will probably be supported eventually. Until this day comes, I suggest that the keywords be *reserved;* that is, be forbidden for use as variable names. There are powerful stylistic reasons for preferring this, anyway (he said weakly . . .).

Non-integer Values

Frequently, the user wishes to have values which are bigger than integers; again, this is an area where Yacc does not make the job as easy as it might, and some additional support is likely. Nevertheless, at the cost of writing a storage manager, the user can return pointers or indices to blocks of storage big enough to contain the full values desired.

Previous Work

There have been many previous applications of Yacc. The user who is contemplating a big application might well find that others have developed relevant techniques, or even portions of grammars. Yacc specifications appear to be easier to change than the equivalent computer programs, so that the ‘‘prior art’’ is more relevant here as well.

Section 8: User Experience, Summary, and Acknowledgements

Yacc has been used in the construction of a C compiler for the Honeywell 6000, a system for typesetting mathematical equations, a low level implementation language for the PDP 11, APL and Basic compilers to run under the UNIX system, and a number of other applications.

To summarize, Yacc can be used to construct parsers; these parsers can interact in a fairly flexible way with the lexical analysis and output phases of a larger system. The system also provides an indication of ambiguities in the specification, and allows disambiguating rules to be supplied to resolve these ambiguities.

Because the output of Yacc is largely tables, the system is relatively language independent. In the presence of reasonable applications, Yacc could be modified or adapted to produce subroutines for other machines and languages. In addition, we continue to seek better algorithms to improve the lexical analysis and code generation phases of compilers produced using Yacc.

This document would be incomplete if I did not give credit 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. Al Aho also deserves recognition for bringing the mountain to Mohammed, and other favors.

References

1 Aho, A.V. and Johnson, S.C., ‘‘LR Parsing’’, Computing Surveys, Vol 6, No 2, June 1974, pp. 99-124.

2 Aho, A.V., Johnson, S.C., and Ullman, J.D., ‘‘Deterministic Parsing of Ambiguous Grammars’’, Proceedings of the A.C.M. Symposium on Principles of Programming Languages, October 1973, pp. 1-21; to appear in CACM.

3 Aho, A.V. and Ullman, J.D., Theory of Parsing, Translation, and Compiling. Volume 1 (1972) and Volume 2 (1973), Prentice-Hall, Englewood Cliffs, N.J.

4 Kernighan, B. W., Ratfor, a Rational Fortran

5 Ryder, B. B., ‘‘The PFORT Verifier,’’ Software−Practice and Experience, Vol 4 (1974), pp 359-377.

6 McIlroy, M. D., A Manual for the TMG Compiler-writing Language

7 Ritchie, D. M., C Reference Manual

Appendix A: A Simple Example

This example gives the complete Yacc specification for a small desk calculator; the desk calculator has 26 registers, labeled a through z, and accepts arithmetic expressions made up of the operators +, −, *, /, % (mod operator), & (bitwise and), | (bitwise or), and assignment. If an expression is an assignment at the top level, the value is not printed; otherwise it is. As in C, an integer which begins with 0 (zero) is assumed to be octal; otherwise, it is assumed to be decimal.

As an example of a Yacc specification, the desk calculator does a reasonable job of showing the way that precedences and ambiguities are used, as well as showing how simple error recovery operates. The major oversimplifications are that the lexical analysis phase is much simpler than for most applications, and the output is produced immediately, line by line. Note the way that decimal and octal integers are read in by the grammar rules; frequently, this job is better done by the lexical analyzer.

```
%token DIGIT LETTER/* these are token names */
%left ´|´                     /* declarations of operator precedences */
%left ´&´
%left ´+´ ´−´
%left ´*´ ´/´ ´%´
%left UMINUS                  /* supplies precedence for unary minus */
%{                            /* declarations used by the actions */
        int base;
        int regs[26];
%}

%%   /* beginning of rules section */

list :          /* list is the start symbol */
        |       /* empty */
        list stat ´\n´ |
        list error ´\n´ =
        {
                yyerrok ;
        } ;

stat :
        expr =
        {
                printf("%d\n", $1) ;
        } |
        LETTER ´=´ expr =
        {
                regs[$1] = $3 ;
        } ;

expr :
        ´(´ expr ´)´ =
        {
                $$ = $2 ;
        } |
        expr ´+´ expr =
        {
```

```
                $$ = $1 + $3 ;
        } |
        expr ´−´ expr =
        {
                $$ = $1 − $3 ;
        } |
        expr ´*´ expr =
        {
                $$ = $1 * $3 ;
        } |
        expr ´/´ expr =
        {
                $$ = $1 / $3 ;
        } |
        expr ´%´ expr =
        {
                $$ = $1 % $3 ;
        } |
        expr ´&´ expr
        {
                $$ = $1 & $3 ;
        } |
        expr ´|´ expr
        {
                $$ = $1 | $3 ;
        } |
        ´−´ expr  %prec UMINUS
        {
                $$ = − $2 ;
        } |
        LETTER
        {
                $$ = regs[$1] ;
        } |
        number ;

number :
        DIGIT =
        {
                $$ = $1 ;
                base = 10 ;
                if( $1 == 0 )
                        base = 8 ;
        } |
        number DIGIT =
        {
                $$ = base * $1 + $2 ;
        } ;

%%              /* start of programs */

yylex( ) /* lexaical analysis routine */
```

```
{
        /* returns LETTER for a lower case letter, yylval = 0 through 25 */
        /* return DIGIT for a digit, yylval = 0 through 9 */
        /* all other characters are returned immediately */

        int c ;

        while( (c=getchar( )) == ´ ´ )
                ;
        if( c >= ´a´ && c <= ´z´ ) {
                yylval = c − ´a´ ;
                return( LETTER ) ;
        }
        if( c >= ´0´ && c <= ´9´ ) {
                yylval = c − ´0´ ;
                return( DIGIT ) ;
        }
        return( c ) ;
}
```

Appendix B: Use of Yacc on Unix

Suppose that the Yacc specification is on a file called yfile. If the actions are in C, Yacc is invoked by

yacc yfile

The output appears on file y.tab.c To compile the parser and load it with the Yacc library, use the command

cc y.tab.c −ly

If Yacc is invoked with the option −v:

yacc −v yfile

a verbose description of the parser is produced on file y.output. The C user should consult section 6C for more information about the run time environment.

If the actions are in Ratfor, the user should invoke Yacc with the option −r:

yacc −r yfile

The Ratfor output appears on file y.tab.r It may be compiled by

rc −2 y.tab.r

Note that when Yacc is used to produce Ratfor programs, there is no need to load these programs with any library.

If the −v action is also invoked:

yacc −rv yfile

a verbose description of the parser is produced on file y.output. The Ratfor user should consult section 6R for more information about the run time environment.

Appendix C: Old Features Supported but not Encouraged

This appendix mentions synonyms and features which are supported for historical continuity, but, for various reasons, are not encouraged.

1. Literals may be delimited by double quotes “"” as well as single quotes “´”.
2. Literals may be more than one character long. If all the characters are alphabetic, numeric, or _, the type number of the literal is defined, just as if the literal did not have the quotes around it. Otherwise, it is difficult to find the value for such literals.

   The use of multi-character literals is likely to mislead those unfamiliar with Yacc, since it suggests that Yacc is doing a job which must be actually done by the lexical analyzer.
3. Most places where % is legal, backslash “\” may be used. In particular, \\ is the same as %%, \left the same as %left, etc.
4. There are a number of other synonyms:

   %< is the same as %left
   %> is the same as %right
   %binary and %2 are the same as %nonassoc
   %0 and %term are the same as %token
   %= is the same as %prec
5. The curly braces “{” and “}” around an action are optional if the action consists of a single C statement. (They are always required in Ratfor).

Appendix D: An Example of Bundling

The following program is an example of the technique of bundling; this example is discussed in Section 7.

```
/* warnings:
1.	This works on Unix; the handling of functions with a variable number of arguments is different on
	different systems.
2.	A number of checks for array bounds have been left out to avoid obscuring the basic ideas, but
	should be there in a practical program.
	 */


%token NAME

%right ´=´
%left ´+´ ´−´
%left ´*´ ´/´

%%

lines :
	= /* empty */
	{
		bclear( ) ;
	} |
	lines expr ´\n´ =
	{
		bprint( $2 ) ;
		printf( "\n" ) ;
		bclear( ) ;
	} |
	lines error ´\n´ =
	{
		bclear( ) ;
		yyerrok;
	} ;

expr :
	expr ´+´ expr =
	{
		$$ = bundle( "add(", $1, ",", $3, ")" );
	} |
	expr ´−´ expr =
	{
		$$ = bundle( "sub(", $1, ",", $3, ")" );
	} |
	expr ´*´ expr =
	{
		$$ = bundle( "mul(", $1, ",", $3, ")" );
```

```
        } |
        expr ´/´ expr =
        {
                $$ = bundle( "div(", $1, ",", $3, ")" );
        } |
        ´(´ expr ´)´ =
        {
                $$ = $2;
        } |
        NAME ´=´ expr =
                $$ = bundle( "assign(", $1, ",", $3, ")" );
        } |
        NAME ;

%%

#define nsize 200
char    names[nsize], *nptr { names };

#define bsize 500
int     bspace[bsize], *bptr { bspace };

yylex( )
{
        int c;

        c = getchar( );
        while( c == ´ ´ )
                c = getchar( );
        if( c>=´a´ && c<=´z´ ) {
                yylval = nptr;
                for( ; c>=´a´ && c<=´z´; c=getchar( ) )
                        *nptr++ = c;
                ungetc( c );
                *nptr++ = ´\0´;
                return( NAME );
        }
        return( c );
}

bclear( )
{


        nptr = names;
        bptr = bspace;
}

bundle( a1,a2,a3,a4,a5 )
{
        int i, j, *p, *obp;
```

```
	p = &a1;
	i = nargs( );
	obp = bptr;

	for( j=0; j<i; ++j )
		*bptr++ = *p++;
	*bptr++ = 0;
	return( obp );
}

bprint( p )
int *p;
{

	if( p>=bspace && p< &bspace[bsize] ) /* bundle */
		while( *p != 0 )
			bprint( *p++ );
	else printf( "%s",  p );
}
```