Introduction to yacc

The first high-level programming language compiler took a very long time to write. Since then, much has been learned about how to design languages and how to translate programs written in high-level languages into machine instructions. With what is known today, the writing of a compiler takes a fraction of the time it used to require.

Much of this improvement is due to the use of more powerful software development methods. In addition, we know about the mathematical properties of computer programming languages. Software tools that apply this mathematical knowledge have played a large part in this improvement.

The COHERENT system provides two tools to simplify the generation of compilers. These tools are the lexical analyzer generator **lex** and the parser generator **yacc**. The following introduces **yacc**, and gives a basic course in its use.

Although initially intended for the development of compilers, **lex** and **yacc** have proven their utility in other, simpler, tasks. Examples of very simple languages are included in this tutorial.

yacc accepts a free-form description of a programming language and its associated parsing, and generates a C program that, when compiled, will parse a program written in the described language. It uses a left-to-right, bottom-up technique, to detect errors in the input as soon as theoretically possible. **yacc** generates parsers that handle certain grammatical ambiguities properly.

This manual presumes that you are familiar with computer-language parsing and formal methods of description of languages. Because **yacc** generates its programs in C and uses many of C's syntactic conventions, you should have a working knowledge of C. Related documents include *Using the COHERENT System* and *Introduction to lex*.

Examples

The following presents a few small examples that you can experiment with to get a feel of how to use **yacc**. Feel free to experiment with the examples to investigate new ideas.

Phrases and Parentheses

The first example describes a language we call **slang**, or *simple language*. **slang** consists of sentences. A sentence, in turn, consists of strings of letters or groups of letters enclosed in parentheses, terminated by a period. A group of letters can also include other groups of letters.

The simplest "sentence" in **slang** is:

```
a.
```

The following demonstrates a sentence that consists only of a group:

```
(ab).
```

As described above, a group can have another group inside it:

```
ab(cd(ef)).
```

The following gives the **yacc** grammar for **slang**. Type it into the file **slang.y.** Note that the lexical-analyzer routine **yylex** is included in the **yacc** input file. Note also that, as in C, comments are strings placed between the characters **/*** and ***/**.

```
/* Tokens (terminals) are all caps */
%token LPAREN RPAREN OTHER PERIOD
%%
run     :       sent     /* Input can be a single */
        |       run sent /* sentence or several */
        ;

sent    :       phrase PERIOD
                        {printf ("sentence\n");}
        ;
```

```
group   :       LPAREN phrase RPAREN
                        {printf ("group\n");}
        ;

phrase  :               /* empty */
        |       others
        |       group
        |       others group
        ;

others  :       OTHER   /* letters and other chars */
        |       others OTHER
        ;
%%

#include <stdio.h>
#include <ctype.h>
/* Called by the parser to get a token  */
yylex ()
{
    int c;
    c = 0;

    while (c == 0) {
       c = getchar();
       if (c == '.') return (PERIOD);
       else if (c == '(') return (LPAREN);
       else if (c == ')') return (RPAREN);
       else if (c == EOF) return (EOF);
       else if (! isalpha(c)) c = 0;
    }
    return (OTHER);
}
```

To generate and compile the parser described by this input, issue the commands

```
yacc slang.y
cc y.tab.c -ly -o slang
```

Now, invoke your new parser by typing

```
slang
```

and test it by typing the following input:

```
a
```

does not reply, since this is not a sentence. When you type:

```
a.
```

slang replies:

```
sentence
```

And if you type:

```
abc(def).
```

slang replies:

```
group
```

As you can see, **slang** recognizes groups and sentences within the input you typed, and reactes by printing an appropriate message. Try typing

```
aaa(bbb(ccc)).
(a).
```

and see what you get. To exit from **slang**, type **<ctrl-C>**.

Simple Expression Processing

The next example creates a small language that includes two types of statements. The first type of statement resembles a procedure call, and the second is an expression. Procedure names are in upper-case letters, whereas the variables in expressions are in lower-case letters. Both procedures and expressions are terminated by a semicolon.

The following code generates a parser that identifies either the procedure being called or the arithmetic expression being calculated. The lexical input routine is independently generated by **lex**.

Enter the following program into the file **calc.y**:

```
%token VARIABLE PROCEDURE
%%
prog    :       stmnt
        |       prog stmnt
        ;

stmnt   :       stat
        |       stat '\n'
        |       error '\n'
        ;

stat    :       PROCEDURE ';'
                    {printf ("PROCEDURE is %c\n", $1);}
        |       expr ';'
                    {printf ("Expression\n");}
        ;

expr    :       expr '-' expr
                    {printf
                    ("Subtract %c from %c giving E\n",
                    $3, $1);
                    $$ = 'E';
                    }
        |       VARIABLE
                    {$$ = $1;}
        ;
```

Enter the lexical-analyzer part of the program into the file **calc.lex**:

```
%{
#include "y.tab.h"
%}
%%
[A-Z]           {
                yylval = yytext [0];
                return PROCEDURE;
                }

[a-z]           {
                yylval = yytext [0];
                return VARIABLE;
                }
\n              return ('\n');
.               return (yytext [0]);
```

Now, generate the programs and compile them by typing:

```
yacc calc.y
lex calc.lex
cc y.tab.c lex.yy.c -ly -ll -o calc
```

The following messages will appear on your console:

```
1 S/R conflict
y.tab.c:
lex.yy.c:
```

For now, you can freely ignore the S/R conflict (Shift/Reduce) message from **yacc**. We shall deal with the shift and reduce notions later on. To invoke the newly generated program, type:

```
calc
```

To test it, type the following:

```
A;B;
C;
a-b-c;
a-b-c-d-e;
```

calc will reply appropriately to each line of input. To exit, type **<ctrl-C>**.

Background

Now that you have tried **yacc,** the following gives some background to it, and how the parsers that it generates operate.

LR Parsing

yacc generates a "bottom up" parser. More specifically, **yacc** generates parsers that read LALR(1) languages.

LR parsers scan the input in a left-to-right fashion. Unfortunately, LR parsers for interesting languages are unpractically large. LALR(k) parsers, which are derived from LR parsers, use a "look ahead" technique, in which the next **k** elements of the input stream are used to help determine reductions. LALR(1) parsers are small enough to be practical, are easy to generate, and are fast.

Input Specification

To generate a language with **yacc**, you must specify its grammar in Backus-Naur Form (BNF). (For a good introduction to BNF, see the section on parsing in *Applied C*.) The languages recognized by **yacc**-generated parsers are rich and compare favorably with modern programming languages. The time required to generate the parser from the input grammar is very small — less than the time required to compile the generated parsers.

In addition to generating the parser to recognize the input language, **yacc** lets you include compiler actions within the grammar rules that are executed as the constructs are recognized. This greatly simplifies the entire task of writing your compiler. When used in combination with **lex**, **yacc** can make the process of writing a recognizer for a simple language the task of an afternoon.

Parser Operation

yacc generates a compilable C program that consists of a routine named **yyparse**, and the information about the grammar encoded into tables. Routines in the **yacc** library are also used.

The basic data structure used by the parser is a *stack*, or *push down list*. At any time during the parse, the stack contains information describing the state of the parse. The state of the parse is related to parts of grammar rules already recognized in the input to the parser.

At each step of the parse, the parser can take one of four actions.

The first action is to *shift*. Information about the input symbol or nonterminal is pushed onto the stack, along with the state of the parser.

The second type of action is to *reduce*. This occurs when a grammar rule is completely recognized. Items describing the component parts of the rule are removed from the stack, and the new state is pushed onto the stack. Thus, the stack is *reduced*, and the symbols corresponding to the grammar rule are *reduced* to the left part of the rule.

Third, the parser can execute an *error* action. If the current input symbol is incorrect for the state of the stack, it is not proper for the parser either to shift or reduce. As a minimum, this state will result in an error message being issued, usually

```
Syntax error
```

yacc provides capabilities for using this error state to recover gracefully from errors in the input.

Finally, the parser can *accept* the input. This means that the *start* symbol, such as *program*, has been properly recognized and that the entire input has been accepted.

Later sections discuss how you can have the parser describe its parsing actions step-by-step.

Form of yacc Programs

A **yacc** program can have up to three sections. Each section is marked by the symbol **%%**. The first section contains declarations. The second section contains the rules of the grammar. User-written routines that are to be part of the generated program can be included in the third section. The outline of **yacc** specifications is as follows:

```
definitions
%%
rules
%%
user code
```

If there are no definitions or user code, the input can be abbreviated to

```
%%
rules
```

Definitions

The first section in a **yacc** specification is the definitions section. This section includes information about the elements used in the **yacc** specification. Additional items are user-defined C statements, such as **include** statements, that are referenced by other statements in the generated program.

Each token, such as **VARIABLE** in example program **calc**, must be predefined in a **%token** statement in the definitions section:

```
%token VARIABLE
```

Tokens are also called **terminals**. Only nonterminals appear as the left part of a rule, and terminals can appear only on the right side of a rule. This helps **yacc** distinguish terminals from nonterminals. Other types of statements that assist in ambiguity resolution appear here, and will be discussed in later sections.

Each grammar that **yacc** generates a parser for must have a *start* symbol. Once the start symbol has been recognized by the parser, its input is recognized and accepted. For a programming-language grammar, this nonterminal represents the entire program.

The start symbol should be declared in the definitions section as:

```
%start program
```

If no **%start** symbol is declared, it is taken to be the left side of the first rule in the rules section.

Rules

Your language's grammar rules must be entered in a variant of BNF. The two following rules illustrate how to define an expression:

```
exp    :     VARIABLE;
exp    :     exp '-' exp;
```

Action statements that are enclosed in braces **{ }** specify the semantics of the language, and are embedded within the rules. More information about how rules are built is given below.

User Code

Action statements may require other routines, such as common code-generating routines, or symbol table building routines. Such user code can be included in the generated parser after the rules section and a **%%** delimiter.

The following sections discuss definitions and rules in detail.

Rules

Rules describe how programming-language constructs are put together. Any given language can be described by many configurations of rules. This frees you to write the rules for clarity and readability.

A rule consists of a left part and a right part. The left part is said to *produce* the right part; or, the right part is said to *reduce to* the left part. A rule can also include the action the parser is to perform once it (the rule) is reduced.

General Form of Rules

Blanks and tabs are ignored within rules (except in the action parts). Comments can be enclosed between **/*** and ***/**. The left part of the rule is followed by a colon. Then come the elements of the right part, followed by a semicolon.

Rules that have the same left part can be grouped together with the left part omitted and a vertical bar signifying "or". For example, the grammar

```
exp     :       VARIABLE;
exp     :       exp '-' exp;
```

can be written as:

```
exp     :       VARIABLE
        |       exp '-' exp;
```

Note that these are equivalent to the BNF:

```
<exp>   ::=     VARIABLE
<exp>   ::=     <exp> - <exp>
```

A rule can also contain C statements that are the compiler actions themselves. These actions are enclosed in braces **{** and **}** and are executed by the generated parser when the grammar rule has been recognized. More will be said about actions in the following section.

Suggested Style

yacc permits you to write rules in completely free form. For example, the grammar for the above rule can be written:

```
exp:VARIABLE|exp'-'exp;
```

However, this form is much less readable.

Two styles of **yacc** grammar are in common use. The first of these is used throughout this manual.

First, start the left part at the beginning of the line; follow it with a tab; then a colon. The right part should be on the same line, also preceded by a tab.

Second, group all rules with the same left part together, and use the vertical bar aligned under the colon for all but the first rule in the group.

Third, place action items on a separate line following the associated rule, preceded by three tabs.

Finally, precede the terminating semicolon with a single tab, to align it with the colon and vertical bar.

The outline of this style is:

```
left    :       right1 right2
                        {action1}
        |       right3 right4
                        {action2}
        ;
```

This style is compact and works well for languages whose rules and actions together are simple.

For somewhat more extensive languages, or for additional flexibility in adding statements to the action part, use the following modification of the style.

```
left    :       right1 right2 {
                        action1
                        }
        |       right3 right4 {
                        action2
                        }
        ;
```

For specifications that have larger rules or more complex actions, another style is recommended.

As in the first style, group rules with the same left part, and use the vertical bar. Place the left part, with its terminating colon, on a line by itself. Then indent the right parts of the rule one or more tabs as necessary to make the rule and actions readable. Finally, the vertical bar and the semicolon should be at the beginning of the line.

The outline for this style is as follows:

```
left:
        right1 right2 {
                action1
                }
|       right3 right4 {
                action2
                }
;
```

Since the input to **yacc** can be entirely free form, there is no restriction on how to write your rules. However, if you use a consistent style throughout, it will make your job easier.

Actions

In addition to generating a parser to recognize a specific language, **yacc** also lets you include parsing action statements. With this feature, you can include C-language action statements that will be performed when specified constructs are recognized.

Basic Action Statements

The example language **slang,** described above, the action statements simply print information on the terminal as productions are recognized:

```
sent    :       phrase PERIOD
                        {printf ("sentence\n");}
        ;
group   :       LPAREN phrase RPAREN
                        {printf ("group\n");}
        ;
```

Even if your actions will be more complex, using **printf** statements in this way can help verify your grammar early in the development process.

Action Values

If the specification is for the grammar of a programming language, the actions will normally interface to routines that access symbol tables or generate code.

yacc lets rules assume a *value* to help keep track of intermediate results within rules. These values can contain symbol-table information, code-generation information, or other semantic information.

To set a value for a rule, simply use a statement of the form

```
$$ = <value>;
```

within an action statement. The symbol **$$** is the value of the production. This value can be used by other rules that use this rule as a non-terminal part.

The example program **calc,** given above, illustrates the use of the value of productions:

```
expr    :       expr '-' expr {
                    printf
                       ("Subtract %c from %c giving E\n",
                       $3, $1);
                    $$ = 'E';
                    }
        |       VARIABLE
                       {$$ = $1;}
        ;
```

The first rule's action statement sets the value of the production **expr** to **'E'**:

```
$$ = 'E';
```

The *value* of a rule is significant in that it can be used in productions including that rule as a nonterminal part.

An example is given in the first rule above. The **printf** statement refers to the items **$1** and **$3**. **yacc** interprets these symbols to mean the value of elements one and three of the right side, respectively; that is to say, **$1** refers to the value of the first **expr** in the right side of the first rule, and **$3** refers to the second **expr**, as illustrated below:

```
expr    :       expr '-' expr
                $1   $2   $3
```

calc does not reference **$2**.

The value for the tokens is provided by the lexical analyzer. The second rule for **expr** uses this to get the value of the token **VARIABLE**. The value represented by **$1** is provided by the lexical analyzer in the statement

```
yylval = yytext [0];
```

To give another example, here is a simple calculator language, called **digit,** which performs arithmetic on one-digit numbers and prints the results. Type the following grammar into the file **digit.y**:

```
%token DIGIT
%%
session :       calcn
        |       session calcn
        ;

calcn   :       expr '\n' /* print results */
                       {printf ("%d\n", $1);}
        ;

expr    :       term '+' term
                       {$$ = $1 + $3;}
        |       term '-' term
                       {$$ = $1 - $3;}
        ;

term    :       DIGIT
                       {$$ = $1;}
        ;

%%
#include <stdio.h>
yylex ()
{
    int c;
    c = 0;

    while (c == 0) { /* ignore control chars and space */
       c = getchar();
       if (c <= 0) return (c); /* could be EOF */
       if (c == '\n') return (c); /* set c to ignore */
```

```
            if ((c <= '9') && (c >= '0')) {
                yylval = c - '0';
                return (DIGIT);
            }
            if (c <= ' ') c = 0;
        }
        return (c);
    }
```

This creates the **yacc** specification file. To turn it into a program, type

```
yacc digit.y
cc y.tab.c -ly -o digit
```

To invoke the compiled progra, type:

```
digit
```

And to test it, type the following:

```
1+2
2+2
8+9
```

digit will reply, respectively:

```
3
4
17
```

To exit from **digit**, type **<ctrl-C>**.

digit is essentially an interpreter — results are calculated as numbers are typed in. When you type in

```
1+1
```

the parser recognizes the construct

```
term '+' term
```

and executes the statement that adds two numbers together. The two numbers each in turn came from the construct

```
term    :       DIGIT
```

and the value of the digit came from **yylex**. When the statement **calcn** is recognized, the value is printed as the result. Thus, the calculations are performed at the time that the constructs are recognized. If a compiler were being generated, the actions would likely build some form of intermediate code, or expression tree, as in:

```
expr    :       term '+' term
                        {$$=tree (plus, $1, $3);}
```

Structured Values

All the examples thus far have shown action values as simple **int** types. This is not sufficient for a large interpreter or compiler, because at different points in the language a value can represent a constant values, a pointer to code generation trees, or symbol table information.

To solve this problem, **yacc** allows you to define the values of **$$** and **$n** as a *union* of several types. This is done in the definitions section with the **union** statement. For example, to declare action values as an integer, tree pointer, or a symbol-table pointer, you would use the following code:

```
%union {
    int cval;
    struct tree_t tree;
    struct sytp_t sytp;
}
```

This says that action values can be a constant value **cval**, a code tree pointer **tree**, or a symbol-table pointer **sytp**.

To ensure that the correct types are used in assignments and calculations in actions in the generated C program, each token whose value will be used is declared with the appropriate type:

```
%token <tree> A B
%token <cval> CONST
```

In addition, the rules themselves can have a type declaration, as they also can pass action values. Their type is declared in the **%type** statement:

```
%type <sytp> variable
```

This declares the nonterminal **variable** to reference the **sytp** field of the value union.

The values referenced in the action statements do not need to be qualified (unless they are referencing a field of one of the union elements). **yacc** generates the necessary qualification for the references, based upon the type information provided in the **%type** and **%token** statements.

Keep in mind that productions that do not have explicit actions will default to an action of

```
$$ = $1
```

which might cause a type clash when compiling the generated parser. This is more likely to arise during debugging, when you have defined the types but have not put in the actions.

Handling Ambiguities

The ideal grammar for a language is readable and unambiguous. If the grammar is readable, its users will find it easy to use. If the language is unambiguous, the parser generator will parse the programs correctly. However, many common programming language constructs are ambiguous. Consider the following definition of an **if** statement:

```
statement      :       if_statement
               |       others
if_statement   :       IF cond THEN statement
               |       IF cond THEN statement ELSE statement
```

Consider a program that contains a statement

```
if a > b then if c < d then a = d else b = c;
```

The parser does not know by the grammar specification which **if_statement** the **else** belongs with. At the point of the **else**, the parser could correctly recognize it as part of the first **if** or the second **if**. The indentations illustrate the interpretation of the ambiguity associating the **else** with the first **if**.

```
if a > b then
    if c < d then
        a = d;
else
    b = c;
```

Associating it with the second **if**:

```
if a > b then
        if c < d then
                a = d;
        else
                b = c;
```

One solution to this ambiguity is to modify the language and rewrite the grammar. Some programming languages (including the COHERENT shell) have a closing element to the **if** statement, such as **fi**. The grammar for this approach is:

```
statement      :       if_statement
               |       others
if_statement   :       IF cond THEN statement FI
               |       IF cond THEN statement ELSE statement FI
```

Another ambiguity arises from a grammar for common binary arithmetic expressions. The following sample specifies binary subtraction:

```
exp     :       TERM
        |       exp '-' exp
        ;
```

For the program fragment

```
a - b - c
```

the parser can correctly interpret the expression as

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

or as

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

While for the **if** example, the language can be reasonably modified to remove the ambiguity, it is unreasonable in the case of expressions. The grammar can be rewritten for **exp** but it is less convenient.

How yacc Reacts

Because some ambiguities, such as the ones detailed above, are common, **yacc** automatically handles some of them.

The ambiguity exemplified by the **if then else** grammar is called a *shift-reduce* conflict. The parser generator can either choose to shift, meaning to add more elements to the parse stack, or to reduce, meaning to generate the smaller production. In the terms of **if**, the shift would match the **else** with the first **then**. Alternatively, the reduce choice will match the **else** with the latest (rightmost) unmatched **then**.

Unless otherwise specified, **yacc** resolves shift-reduce conflicts in favor of the shift. This means that the **if** ambiguity will be resolved in favor of matching the **else** with the rightmost unmatched **then**. Likewise, the expression

```
a - b - c
```

will be interpreted as

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

Additional Control

yacc provides tools to help resolve some of these ambiguities. When **yacc** detects shift-reduce conflicts, it consults the precedence and associativity of the rule and the input symbol to make a decision.

For the case of binary operators, you can define the associativity of each of the operators by use of the defining words **%left** and **%right**. These appear in the definition section with **%token**.

The usual interpretation of

```
a - b - c
```

is

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

which is called *left* associative. However, the shift/reduce conflict inherent in

```
exp '-' exp
```

is resolved in favor of the reduce, or in a right-associative manner:

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

To signal **yacc** that you want the left-associative interpretation, enter the grammer as:

```
%left '+' '-'
%token TERM
%%
expr    :       TERM
        |       expr '-' expr
        |       expr '+' expr
        ;
```

Some operators, such as assignment, require right associativity. The statement

```
a := b + c
```

is to be interpreted as

```
a := (b + c)
```

The **%right** keyword tells **yacc** that the following terminal is to right associate.

Precedence

Most arithmetic operators are left associative. For example, with the grammar

```
%right =
%left '-' '+' '*' '/'
%%
expr    :       expr '-' expr
        |       expr '*' expr
        |       expr '+' expr
        |       expr '/' expr
        |       expr '=' expr
        ;
```

The expression

```
a = b + c * d - e
```

based on associativity alone will be evaluated

```
a = (((b + c) * d) - e)
```

which is not according to custom. We normally think of * as having higher precedence than **+** or **-**, meaning that it is evaluated before other operators with the same associativity. The evaluation preferred is

```
a = (b + (c * d) - e)
```

To generate a parser with this evaluation, use several lines of **%left**, one line for each level of precedence. Each line containing **%left** describes tokens of the same precedence. The precedence increases with each line. Thus, to get the common notion of arithmetic precedence, use a grammar of

```
%right =
%left '-' '+'
%left '*' '/'
%%
expr    :       expr '-' expr
        |       expr '*' expr
        |       expr '+' expr
        |       expr '/' expr
        |       expr '=' expr
        ;
```

This method of **%left** and **%right** gives tokens a precedence and an associativity. This can eliminate ambiguities where these operators are involved. But what about the precedence of rules or nonterminals?

To specify the precedence of rules, the **%prec** keyword at the end of the rule sets the precedence of the rule to the token following the keyword. To add unary minus to the grammar above, and to give it the precedence of multiply, use **%prec** * at the end of the unary rule.

```
%right =
%left '-' '+' '*' '/'
%%
expr    :       expr '-' expr
        |       expr '*' expr
        |       expr '+' expr
        |       expr '/' expr
        |       expr '=' expr
        |       '-' expr %prec *
        ;
```

If associativity is not specified, **yacc** will report the number of shift/reduce conflicts. When associativity is specified with **%left**, **%right** or **%nonassoc**, this is considered to reduce the number of conflicts, and thus the number of conflicts reported will not include the count of these.

Error Handling

Parsers generated by **yacc** are designed to parse correct programs. If an input program contains errors, the LALR(1) parser will detect the error as soon as is theoretically possible. The error is identified, and the programmer can correct the error and recompile.

However, in most programming environments, it is unacceptable to stop compiling after the detection of a single error. **yacc** parsers attempt to go on so that the programmer may find as many errors as possible.

When an error is detected, the parser looks for a special token in the input grammar named **error**. If none is found, the parser simply exits after issuing the message

```
Syntax error
```

If the special token **error** is present in the input grammar error recovery is modified. Upon detection of an error, the parser removes items from the stack until **error** is a legal input token and processes any action associated with this rule. **error** is the lookahead token at this point.

Processing is resumed with the token causing the error as the lookahead token. However, the parser attempts to resynchronize by reading and processing three more tokens before resuming normal processing. If any of these three are in error, they are deleted and no error message is given. Three tokens must be read without error before the parser leaves the error state.

A good place to put the **error** token is at a statement level. For example, the **calc.y** example in chapter 2 defines a statement as

```
stmnt   :       stat
        |       stat '\n'
        |       error '\n'
        ;
```

Thus, any error on a line will cause the rest of the line to be ignored.

There is still a chance for trouble, however. If the next line contains an error in the first two tokens, they will be deleted with no error message and parsing will resume somewhere in the middle of the line. To give a truly fresh start at the beginning of the line, the function **yyerrok** will cause the parser to resume normal processing immediately. Thus, an improved grammar is

```
stmnt   :       stat
        |       stat '\n'
        |       error '\n'
                        {yyerrok;}
        ;
```

will cause normal processing to begin with the start of the next line.

Error recovery is a complex issue. This section covers only what the parser can do in recovering from syntax errors. Semantic error recovery, such as retracting emitted code, or correcting symbol table entries, is even more complex, and is not discussed here.

yacc reserves a special token **error** to aid in resynchronizing the parse. After an error is detected, the stack is readjusted, and processing cautiously resumes while three error-free tokens are processed. **yyerrok** will cause normal processing to resume immediately. The token causing the error is retained as the lookahead token unless **YYCLEARIN** is executed.

Summary

yacc is an efficient and easy-to-use program to help automate the input phase of programs that benefit by strict checking of complex input. Such programs include compilers and interactive command language processors.

yacc generates an LALR(1) parser, that implements the grammar specifying the structure of the input. A simple lexical analyser routine can be hand-constructed to fit in among the rules, or you can use the COHERENT command **lex** to generate a lexical analyzer that will fit with the parser.

As the structured input is analyzed and verified, you assign meaning to the input by writing semantic **actions** as part of the grammatical rules describing the structure of the input.

yacc parsers are capable of handling certain *ambiguities*, such as that inherent in typical **if then else** constructs. This simplifies the construction of many common grammars.

yacc provides a few simple tools to aid in error recovery. However, the area of error recovery is complex and must be approached with caution.

Helpful Hints

Until you have mastered **yacc**, the best way to build your program is to do it a piece at a time. For example, if you are writing a Pascal compiler, you might start with the grammar

```
%token PROG BEG END OTHER
program :      PROG tokens BEG END '.'
        ;
tokens  :      OTHER
        |      tokens OTHER
        ;
```

and with a simple lexical analyzer of:

```
PROGRAM        return (PROG);
BEGIN          return (BEG);
END            return (END);
.              return (yytext [0]);
```

With the generated program, you can easily test the grammar by feeding it simple programs. Then add items to both the lexical analyzer and **yacc** grammar. With this approach, you can see the parser working, and if it behaves differently than you expect, you can more easily pinpoint the cause.

If you have difficulty understanding what actions your parser is taking, **yacc** will produce for you a complete description of the generated parser. To use this, you should be familiar with the way LALR(1) parsers work. To get this verbose output, specify the **-v** option on the command line. The result will appear in the file **y.output**.

In addition, you can have the parser give you a token-by-token description of its actions while it does them, by specifying the debug option **-d**. This also generates the file **y.output**, which is helpful in reading the debug output. The debug code is generated when the **-d** option is used, but is not activated unless the **YYDEBUG** identifier is defined. Include some code in the definitions section to activate it:

```
%{
    define YYDEBUG
%}
```

Your parser can turn on and off the debugging at execution time by setting the variable **YYDEBUG**: one for on, zero for off.

A frequent cause of grammar conflicts is the empty statement. You should use it with caution. **yacc** generates empty statements when you specify actions in the middle of a rule rather than at the end; for example:

```
def     :      DEFINE {defstart();}
                      identifier {defid ($2);}
        ;
```

yacc generates an additional rule:

```
$def    :      /* empty */
                      {defstart();}
        ;
def     :      DEFINE $def identifier {defid ($2);}
        ;
```

The resulting empty statement can cause parser conflicts if there are similar rules and the empty statement is not sufficient to distinguish between them.

Where to Go From Here

The Lexicon article for **yacc** summarizes its command syntax and features. The tutorial for **lex**, the COHERENT lexical analyzer, describes how to combine **lex** with **yacc** to build applications simply.