yacc Parser Generator Tutorial

Table of Contents

1. Introduction

The first high-level programming language compiler took a very long time to write. Since that time, much has been learned about how to design languages and how to translate programs in high-level languages into machine instructions. With what is known today, a compiler can be written in one-tenth of the time it used to take.

Much of this improvement is due to the use of more powerful software development methods. Additionally, much more is known 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**. This document is a tutorial for **yacc**.

Although initially intended for the development of programming language compilers, **lex** and **yacc** have proven their utility in other, simpler, tasks. Together or separately they are simple enough to use that they can be easily applied to simple tasks. Examples of very simple languages are included in this tutorial.

yacc accepts a free-form description of a programming language and associated parsing actions and generates a C program to parse the language. Using a left-to-right bottom-up technique, errors in the input are detected 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. Since **yacc** generates its programs in C and uses many of C's syntactic conventions, it will be helpful if you have a working knowledge of C. Related documents include *Introduction to The COHERENT System* and *lex Lexical Generator Tutorial*.

2. Examples

This introductory section presents a few small examples that you can experiment with to get a feel of how to use **yacc**. As you type these in, change parts of the example to experiment with new ideas.

Phrases and parentheses

The first example recognizes a language made up of single characters, parentheses, and period. These characters are used to form sentences. Sentences are made up of strings of letters and groups of letters enclosed in parentheses. The groups can also include other groups.

The simplest "sentence" in the language is

```
a.
```

A sentence made up of only a group is

```
(ab).
```

A sentence with a group inside of a group is

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

Enter following **yacc** grammar into the file **paren.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 paren.y
        cc y.tab.c -ly -o paren
```

Demonstrate the parser with:

```
        paren
        a
        a.
        abc(def).
        aaa(bbb(ccc)).
        (a).
```

The result of this test will be

```
        sentence
        group
        sentence
        group
        group
        sentence
        group
        sentence
```

Simple expression processing

The next example illustrates a small language that includes two types of statements. The first resembles a procedure call, and the second is an expression. Procedure names are upper-case letters, and variables in expressions are lower-case letters.

The input shown generates a parser that identifies 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]);
```

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:
```

To test the generated program, type

```
calc
A;B;
C;
a-b-c;
a-b-c-d-e;
<ctrl-D>
```

The result will be

```
PROCEDURE is A
PROCEDURE is B
PROCEDURE is C
Subtract c from b giving E
Subtract E from a giving E
Expression
Subtract e from d giving E
Subtract E from c giving E
Subtract E from b giving E
Subtract E from a giving E
Expression
```

Summary

Two examples are given in this section that you can try out, even if you have never used **yacc** before. You only need to type them in exactly as shown to test them.

3. Background

This section discusses some of the background of **yacc**, and how parsers generated by **yacc** operate.

LR parsing

The parser generated by **yacc** is of the class of parsers commonly known as 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.

A class of parsers called LALR parsers have been derived from LR parsers. LALR(k) parsers use a Look Ahead technique whereby the next **k** elements of the input stream are used to help decide on reductions. LALR(1) parsers are small enough to be practical, and are easy to generate and fast to use.

Input specification

To use **yacc**, you will specify the grammar in BNF, or Backus-Naur Form. 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** provides the capability to include compiler actions within the grammar rules that are executed as the constructs are recognized. This greatly simplifies the entire compiler writing task. When used in combination with **lex**, **yacc** can make the process of writing a recognizer for a simple language an afternoon's task.

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.

Summary

This section discusses some of the background of **yacc** generated parsers, and outlines the operation of the parser. This information can be useful to you in using **yacc**.

4. Form of yacc programs

There are up to three sections of the specification of the language given to **yacc**. The sections are separated by the special 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.

Rules

The grammar rules describing the language are entered in a variant of BNF. The two following rules illustrate the definition of an expression:

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

Action statements enclosed in braces **{ }** specify the semantics of the language and are embedded within the rules. More information about how rules are built is given in section 5.

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 the second example in section 2, 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 only appear 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.

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.

Summary

There are three sections to a **yacc** specification. The outline of **yacc** specifications is:

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

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

```
%%
rules
```

Following sections discuss definitions and rules in detail.

5. Rules

Rules describe how programming language constructs are put together. For any given language, there can be many configurations of rules describing the language. This gives you a freedom of choice to write the rules for clarity and readability.

Rules consist 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.

In addition to describing the grammar of the language, rules can include parser actions to be performed at the time that the rule is reduced.

General form of rules

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

Rules that have the same left part may be grouped together with the left part omitted and a vertical bar signifying "or".

Part of the expression grammar from section 2

```
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>
```

The rules can also contain C programming language 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

Rules can be written completely free form for **yacc**. The rules for the expression grammar above can as well be written

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

but is certainly less readable.

There are two styles 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.

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

Finally, the terminating semicolon is lined up with the colon and vertical bar by preceding it with a single tab. 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 previously suggested style, rules with the same left part should be grouped together, and the vertical bar should be used. The left part should be immediately followed by a colon, and appear by itself on the line.

The right parts of the rule are then indented one or more tabs as necessary to make the rule and actions readable.

The vertical bar indicating an alternative right part for the rule, and the semicolon signalling the end of the rule, should be at the beginning of the line.

The outline for this style is

```
        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. You may decide to adopt a modified version of these styles, but if you use a consistent style throughout, it will make your job easier.

Summary

Specification of the rules of your **yacc** grammar is the central part of the **yacc** specification. This section gives the rules for writing rules, and suggests two styles of writing them that will improve readability.

6. Actions

In addition to generating a parser to recognize a specific language, **yacc** also provides you with the convenience of including 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 first example in section two illustrates action statements that simply print information on the terminal as productions are recognized. Consider the production for **sent** and **group** from that example.

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

The actions here are print statements that signal when a production has been recognized. 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 symbol table routines, and code generators, or intermediate form emitters.

yacc provides the capability for rules to assume a *value* to help keep track of intermediate results in 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 second example in section two illustrates the use of the value of productions. Examining the production for **expr,** we have

```
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 contains references 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**.

To illustrate, the elements of the right side are numbered, for the purposes of values, beginning at one for the leftmost element of the right side:

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

In this example, **$2** is not referenced.

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 that does arithemtic on single-digit numbers and prints the results out. Enter 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
```

The following tests **digit**:

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

digit will reply:

```
3
4
17
```

This program 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 is 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, since at different points in the language, the value can represent constant values, pointers to code generation trees, or symbol table information.

To solve this problem, **yacc** will define the **$$** and **$n** values as a *union* of several types. This is done in the definitions section with the **union** statement:

```
%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**.

So 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
```

Additionally, the rules themselves can have a type declaration, since 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.

For an illustration of the practical applications of **%union**, see the example in section 9.

Summary

This section has presented **yacc** actions and how they carry out the semantics of the grammar. Rule values and a simple calculator program are presented to illustrate the use of actions.

7. Handling ambiguities

The ideal grammar for a language is readable and unambiguous. If the grammar is easy to understand, the users of the language will benefit most from the use of it. If the language is unambiguous, the parser generator will parse the programs in the correct way.

However, many common programming language constructs are ambiguous. An example of an **if** is

```
stmnt           :     if_stmnt
                |     others
if_stmnt                :     IF cond THEN stmnt
                |     IF cond THEN stmnt ELSE stmnt
```

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_stmnt** 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 language is

```
stmnt          :       if_stmnt
               |       others
if_stmnt       :       IF cond THEN stmnt FI
               |       IF cond THEN stmnt ELSE stmnt 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

Since certain ambiguities are common, such as the ones detailed above, **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**. Similarly, 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**. Any symbol appearing in **left** or **right**.

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 grammar 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.

8. 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 immediately resume normal processing. 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.

Summary

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.

9. 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 given 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 gramatical 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**.

Additionally, you can have the parser give you a token-by-token description of its actions as it is doing them by specifying the debug option **−d**. This also generates the **y.output** file which will be 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

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

to activate it. Your parser can turn on and off the debugging at execution time by setting the variable **yydebug** (1 for on, 0 for off).

A frequent cause of grammar conflicts is the empty statement. You should use it with caution. Note that empty statements are generated by **yacc** when you specify actions in the middle of a rule rather than at the end. Consider

```
        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.

Example

This tutorial will close with a larger example that incorporates most of the features of **yacc** presented in this tutorial. You can enter it as shown, and modify it to improve its operation.

Designed for interactive input, this example will calculate the great circle path and bearing from one point on the globe to another.

Each pair of points is input on a single line. The coordinates of the origin and destination are preceeded by the keywords **FROM** and **TO** respectively, and can appear in either order. Longitude and latitude are followed by the letters **E** or **W**, and **N** and **S** respectively. Lower-case may also be used for these letters.

The numeric part of the coordinates may be entered in degrees, minutes and optional seconds, or in fractional degrees. The symbols ^, **o** and **d** specify degrees, since the raised circle customarily used for degrees is not available on most terminals. A single quote ' follows the minutes, and a double quote " follows seconds.

To illustrate the use of the program **nav**, calculate the great circle distance and initial heading from Charlestown, Indiana to Charlestown, Australia:

```
from 38d27'n 85d40'w to 151d42'e 32d58's;
```

The result will be:

```
From lat 38.450 long 85.667 To lat -32.967 long -151.700
Distance 8030.623, Init course is 258.417
```

Note that the coordinates are echoed in decimal degrees. To exit the program, type <ctrl-D>.

Enter the following **yacc** specification file into the **nav.y** :

```
%{
#include "ll.h"
#define YYTNAMES
        double fromlat, fromlon, tolat, tolon;
        extern calcpath();
%}
%union {
        double dgs;
        long dgsi;
        struct ll wh;
        }
%token NEWLINE FROM TO CIRCLE QUOTE
%token DQUOTE SEP SEMI COMMA
%token NSYM SSYM WSYM ESYM
%token <dgs> FNUM
%token <dgsi> NUM
%type <dgs> degrees long lat deg
%type <wh> where from to
%%
prob    :       single
        |       prob single
        ;
single  :       sing {
                        calcpath();
                        }
        |       error NEWLINE {
                        yyerrok; YYCLEARIN;
                        printf ("Enter line again.\n");
                        }
        ;
sing    :       from SEP to SEMI NEWLINE {
                        fromlat = $1.lat;
                        fromlon = $1.lon;
                        tolat = $3.lat;
                        tolon = $3.lon;
                        }
        |       to SEP from SEMI NEWLINE {
                        tolat = $1.lat;
                        tolon = $1.lon;
                        fromlat = $3.lat;
```

```
                                fromlon = $3.lon;
                                }
                |       to SEMI NEWLINE {
                                tolat = $1.lat;
                                tolon = $1.lon;
                                }
                ;
from            :       FROM SEP where {
                                $$ = $3;
                                }
                ;
to              :       TO SEP where {
                                $$ = $3;
                                }
                ;
where           :       lat SEP long {
                                $$.lat = $1;
                                $$.lon = $3;
                                }
                |       long SEP lat {
                                $$.lon = $1;
                                $$.lat = $3;
                                }
                ;
lat             :       degrees NSYM {
                                $$ = $1;
                                }
                |       degrees SSYM {
                                $$ = - $1;
                                }
                ;
long            :       degrees WSYM {
                                $$ = $1;
                                }
                |       degrees ESYM {
                                $$ = - $1;
                                }
                ;
degrees :       FNUM    /* e. g. 128.3 */ {
                                $$ = $1;
```

```
                                 }
                | NUM CIRCLE NUM QUOTE /* deg min */ {
                                 $$=$1 + $3/60.0;
                                 }
                | NUM CIRCLE NUM QUOTE NUM DQUOTE
                                 /* and seconds */ {
                                 $$=$1 + $3/60.0 + $5/3600.0;
                                 }
                | NUM CIRCLE NUM QUOTE FNUM DQUOTE {
                                 $$=$1 + $3/60.0 + $5/3600.0;
                                 }
                ;
%%
#include <stdio.h>
yyerror (s)
char *s;
{
    struct yytname *p;
    fprintf (stderr, "%s ", s);
    for (p = yytnames; p -> tn_name != NULL; ++)
        if (p->tn_val == yychar) {
            fprintf (stderr, "at %s", p->tn_name);
            break;
        }
    fprintf (stderr, "\n");
}
```

Both the lexical analyzer and the parser will need the following header file **ll.h**:

```
struct ll {
    double lat;
    double lon;
};
```

To turn **yacc** file **nav.y** into a program, type

```
yacc -hdr nav.tab.h -d -v nav.y
mv y.tab.c nav.y.c
```

The grammar is straightforward. The types used in the actions require a union, since integer degrees, floating point degrees, and pairs of floating point degrees are used as action values. The lexical analyzer recognizes integer and floating point numbers, and passes the value through **yylval**. The rule for **degrees** combines different style degree representations to a single double precision number.

The **N**, **S**, **E**, and **W** symbols convert a location to a signed representation. **S** and **E** result in negative degrees, **N** and **W** as positive.

The rule for **where** converts the single-numbered latitude and longitude into a double number of **<wh>** type. Note that it can process the coordinates in either order.

The rule **single** handles the destination and origin in either order. It takes the pairs of coordinates from **from** and **to** and stores them in the global variables that the calculation routine uses. The error token will halt error recovery at the end of the line, so that in case of error the user can reenter the correct line. If many great circles are being computed from the same origin, you need to enter only the destination after the first time.

And once a single set of coordinates has been recognized, the great circle is calculated by the function **calcpath**.

The error routine **yyerror** accepts an error message from the parser, and examines the table of tokens to find the name of the token where the error is detected. If it is found, it is printed. In order to get these token names in the program, the symbol **YYTNAMES** must be defined.

The following is the lexical analyzer. Enter it into the file **nav.l**.

```
%{
#include "ll.h"
#include "nav.tab.h"
%}
        int integer;
        double real;
%%
[nN]                return (NSYM);
[sS]                return (SSYM);
[eE]                return (ESYM);
[wW]                return (WSYM);
o|"^"|d             return (CIRCLE);
\"                  return (DQUOTE);
\'                  return (QUOTE);
\n                  return (NEWLINE);
from                return (FROM);
FROM                return (FROM);
to                  return (TO);
TO                  return (TO);
[0-9]+              {
                    sscanf (yytext, "%d", &integer);
                    yylval.dgsi = (long) integer;
                    return (NUM);
                    }
[0-9]+"."([0-9]+)? {
                    sscanf (yytext, "%f", &real);
                    yylval.dgs = (double) real;
                    return (FNUM);
                    }
,                   return (COMMA);
;                   return (SEMI);
[ \t]               return (SEP);
.                   {
                    printf ("Illegal character [%s]\n",
                            yytext);
                    return (yytext [0]);
                    }
```

The lexical analyzer partitions the input into tokens expected by the parser. For the symbols in the grammar, it returns the token type. It also recognizes integer and floating point numbers, and converts them to integers.

Note that the **ll.h** file is required even though there is no explicit reference to its contents. This is needed because the **%union** in **nav.y** generates the header file **nav.tab.h** referring to the **ll** structure.

Turn **lex** file **nav.l** into program by typing:

```
lex nav.l
mv lex.yy.c nav.l.c
```

Finally, the great circle calculation routine is to be entered into the file **navcalc.c**.

```
#include <stdio.h>
#include <math.h>
/*
 *   Given latitude and longitude of start and finish,
 *   calculate the great circle path.
 */
extern       double fromlon, fromlat, tolon, tolat;
calcpath ()
{
     double rad = PI / 180.0;
     double initcourse, arg, dist, d60;
     double rfromlat, rfromlon, rtolat, rtolon;

     printf ("From lat %.3f long %.3f ",
         fromlat, fromlon);
     printf ("To lat %.3f long %.3f\n",
         tolat, tolon);

     rfromlat = fromlat * rad;
     rfromlon = fromlon * rad;
     rtolat = tolat * rad;
     rtolon = tolon * rad;

     d60 = acos (
         sin (rfromlat) * sin (rtolat) +
         cos (rfromlat) * cos (rtolat) *
              cos (rfromlon - rtolon)
      );
     dist = 60 * d60 / rad;

     arg = (sin (rtolat) - cos (d60) * sin (rfromlat))
             /
             (sin (d60) * cos (rfromlat));

     initcourse = acos (arg) / rad;
     if (sin (rfromlon - rtolon) < 0)
         initcourse = 360 - initcourse;

     printf ("Distance %.3f, Init course is %.3f\n\n",
         dist, initcourse);
}
```

And now compile all three programs together.

```
	cc nav.y.c nav.l.c navcalc.c -ly -lm -ll -f -o nav
```

The standard formula is used to calculate great circle path and bearing. Note that there are several limitations that are not checked for here, such as diametrically opposite points on the globe have no unique great circle path between them. Additionally, neither of the points should be at either of the poles. These checks can be added if you wish to use the **nav** program as a general rather than a tutorial tool.

Index

User Reaction Report

To keep this manual and COHERENT free of bugs and facilitate future improvements, we would appreciate receiving your reactions. Please fill in the appropriate sections below and mail to us. Thank you.

Mark Williams Company
1430 W. Wrightwood Avenue
Chicago, IL 60614

Name: ____________________

Company: ____________________

Address: ____________________

____

Phone: ____________________ Date: ____________________

Version and hardware used: ____________________

Did you find any errors in the manual? ____________________

____

____

Can you suggest any improvements to the manual? ____________________

____

____

Did you find any bugs in the software? ____________________

____

____

Can you suggest improvements or enhancements to the software?

____

____

____

Additional comments: (Please use other side.)