Yacc: Yet Another Compiler-Compiler –

Computer program input generally has some structure; in fact, every computer program that does input can be thought of as defining an ``input language'' which it accepts. An input language may be as complex as a programming language, or as simple as a sequence of numbers. Unfortunately, usual input facilities are limited, difficult to use, and often are lax about checking their inputs for validity.

Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles 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 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 grammatical specification.

Yacc is written in portable C. The class of specifications accepted is a very general one: LALR(1) grammars with disambiguating rules.

In addition to compilers for C, APL, Pascal, RATFOR, etc., Yacc has also been used for less conventional languages, including a phototypesetter language, several desk calculator languages, a document retrieval system, and a Fortran debugging system.

1: Basic Specifications

Names refer to either tokens or nonterminal symbols. Yacc requires token names to be declared as such. In addition, for reasons discussed in Section 3, it is often 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 percent ``%'' 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 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. 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. 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'    newline
        '\r'    return
        '\''    single quote ``'''
        '\\'    backslash ``\''
        '\t'    tab
        '\b'    backspace
        '\f'    form feed
        '\xxx'  ``xxx'' in octal
```

For a number of technical reasons, the NUL character ('\0' or 0) 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 easier to change.

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

```
        empty :   ;
```

Names representing tokens must be declared; this is most simply done by writing

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

in the declarations section. (See Sections 3 , 5, and 6 for much more discussion). Every name not defined in the declarations section is assumed to represent a nonterminal symbol. Every nonterminal symbol must appear on the left side of at least one rule.

Of all the nonterminal symbols, one, called the start symbol, has particular importance. The parser is designed to recognize the start symbol; thus, this symbol represents the largest, most general structure described by the grammar rules. By default, the start symbol is taken to be the left hand side of the first grammar rule in the rules section. It is possible, and in fact desirable, to declare the start symbol explicitly in the declarations section using the %start keyword:

```
        %start   symbol
```

The end of the input to the parser 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 function returns to its caller after the endmarker is seen; 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. Usually the endmarker represents some reasonably obvious I/O status, such as ``end-of-file" or ``end-of-record".

2: Actions

With each grammar rule, the user may associate actions to be

performed each time the rule is recognized in the input process. These actions may return values, and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens, if desired.

An action is an arbitrary C statement, and as such can do input and output, call subprograms, and alter external vectors and variables. An action is specified 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.

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 pseudovariable ``$$'' to some value. For example, an action that 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 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, consider the rule

```
        expr    :       '('  expr  ')'   ;
```

The value returned by this rule is usually the value of the expr in parentheses. This can be indicated by

```
        expr    :        '('  expr  ')'         {   $$ = $2 ;   }
```

By default, the value of a rule is the value of the first element in it ($1). Thus, grammar rules of the form

```
        A       :       B    ;
```

frequently need not have an explicit action.

In the examples above, all the actions came at the end of their rules. Sometimes, it is desirable to get control before a rule is fully parsed. Yacc permits an action to be written in the middle of a rule as well as at the end. This rule is assumed to return a value, accessible through the usual mechanism by the actions to the right of it. In turn, it may access the values returned by the symbols to its left. Thus, in the rule

```
        A       :       B
                                {   $$ = 1;   }
                        C
                                {    x = $2;    y = $3;   }
                ;
```

the effect is to set x to 1, and y to the value returned by C.

Actions that do not terminate a rule are actually handled by Yacc by manufacturing a new nonterminal symbol name, and a new rule matching this name to the empty string. The interior action is the action triggered off by recognizing this added rule. Yacc actually treats the above example as if it had been written:

```
        $ACT    :       /* empty */
                                {  $$ = 1;  }
                ;

        A       :       B  $ACT  C
                                {   x = $2;   y = $3;  }
                ;
```

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 to build and maintain the tree structure desired. For example, suppose there is 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 the index of the newly created node. Then parse tree can be built by supplying actions such as:

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

in the specification.

The user may define other variables to be used by the actions. Declarations and definitions can appear in the declarations section, enclosed in the marks ``%{" and ``%}". These declarations and definitions have global scope, so they are known to the action statements and the lexical analyzer. For example,

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

could be placed in the declarations section, making variable accessible to all of the actions. The Yacc parser uses only names beginning in ``yy"; the user should avoid such names.

In these examples, all the values are integers: a discussion of values of other types will be found in Section 10.

<u>3: Lexical Analysis</u>

The user must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer-valued function called yylex. The function returns an integer, the token number, representing the kind of token read. If there is a value associated with that token, it should be assigned to the external variable yylval.

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

```
yylex(){
        extern int yylval;
        int c;
        . . .
        c = getchar();
        . . .
        switch( c ) {
                . . .
        case '0':
        case '1':
          . . .
        case '9':
                yylval = c-'0';
                return( DIGIT );
                . . .
                }
        . . .
```

The intent is to return a token number 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 file, the identifier DIGIT will be defined as the token number associated with the token DIGIT.

This mechanism leads to clear, easily modified lexical analyzers; the only pitfall is the need to avoid using any token names in the grammar that are reserved or significant in C or the parser; for example, the use of token names if or while 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 7).

As mentioned above, the token numbers may be chosen by Yacc or by the user. In the default situation, the numbers are chosen by Yacc. The default token number for a literal character is the numerical value of the character in the local character set. Other names are assigned token numbers starting at 257.

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

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

A very useful tool for constructing lexical analyzers is the Lex program developed by Mike Lesk.[8] These lexical analyzers are designed to work in close harmony with Yacc parsers. The specifications for these lexical analyzers use regular expressions instead of grammar rules. Lex can be easily used to produce quite complicated lexical analyzers, but there remain some languages (such as FORTRAN) which do not fit any theoretical framework, and whose lexical analyzers must be crafted by hand.

4: How the Parser Works

Yacc turns the specification file into a C program, which parses the input according to the specification given. The algorithm used to go from the specification to the parser is complex, and will not be discussed here (see the references for more information). The parser itself, however, is relatively simple, and understanding how it works, while not strictly necessary, will nevertheless make treatment of error recovery and ambiguities much more comprehensible.

The parser produced by Yacc consists of a finite state machine with a stack. The parser is also capable of reading and remembering the next input token (called the lookahead token). The current state is always the one on the top of the stack. The states of the finite state machine are given small integer labels; initially, the machine is in state 0, the stack contains only state 0, and no lookahead token has been read.

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

1. Based on its current state, the parser decides whether it needs a lookahead token to decide what action should be done; if it needs one, and does not have one, it calls yylex to obtain the next token.

2. Using the current state, and the lookahead token if needed, the parser decides on its next action, and carries it out. This may result in states being pushed onto the stack, or popped off of the stack, and in the lookahead token being processed or left alone.

The shift action is the most common action the parser takes. Whenever a shift action is taken, there is always a lookahead token. For example, in state 56 there may be an action:

```
                  IF       shift 34
```

which says, in state 56, if the lookahead token is IF, the current state (56) is pushed down on the stack, and state 34 becomes the current state (on the top of the stack). The lookahead token is cleared.

The reduce action keeps the stack from growing without bounds. Reduce actions are appropriate when the parser has seen the right hand side of a grammar rule, and is prepared to announce that it has seen an instance of the rule, replacing the right hand side by the left hand side. It may be necessary to consult the lookahead token to decide whether to reduce, but usually it is not; in fact, the default action (represented by a ``.") is often a reduce action.

Reduce actions are associated with individual grammar rules. Grammar rules are also given small integer numbers, leading to some confusion. The action

```
                    .         reduce 18
```

refers to grammar rule 18, while the action

```
                    IF        shift 34
```

refers to state 34.

Suppose the rule being reduced is

A : x y z ;

The reduce action depends on the left hand symbol (A in this case), and the number of symbols on the right hand side (three in this case). To reduce, first pop off the top three states from the stack (In general, the number of states popped equals the number of symbols on the right side of the rule). In effect, these states were the ones put on the stack while recognizing x, y, and z, and no longer serve any useful purpose. After popping these states, a state is uncovered which was the state the parser was in before beginning to process the rule. Using this uncovered state, and the symbol on the left side of the rule, perform what is in effect a shift of A. A new state is obtained, pushed onto the stack, and parsing continues. There are significant differences between the processing of the left hand symbol and an ordinary shift of a token, however, so this action is called a goto action. In particular, the lookahead token is cleared by a shift, and is not affected by a goto. In any case, the uncovered state contains an entry such as:

```
                    A         goto 20
```

causing state 20 to be pushed onto the stack, and become the current state.

In effect, the reduce action ``turns back the clock" in the parse, popping the states off the stack to go back to the state where the right hand side of the rule was first seen. The parser then behaves as if it had seen the left side at that time. If the right hand side of the rule is empty, no states are popped off of the stack: the uncovered state is in fact the current state.

The reduce action is also important in the treatment of user-supplied actions and values. When a rule is reduced, the code supplied with the rule is executed before the stack is adjusted. In addition to the stack holding the states, another stack, running in parallel with it, holds the values returned from the lexical analyzer and the actions. When a shift takes place, the external variable yylval is copied onto the value stack. After the return from the user code, the reduction is carried out. When the goto action is done, the external variable yyval is copied onto the value stack. The pseudo-variables $1, $2, etc., refer to the value stack.

The other two parser actions are conceptually much simpler. The accept action indicates that the entire input has been seen and that it matches the specification. This action appears only when the lookahead token is the endmarker, and indicates that the parser has successfully done its job. The error action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification. The input

tokens it has seen, together with the lookahead token, cannot be followed by anything that would result in a legal input. The parser reports an error, and attempts to recover the situation and resume parsing: the error recovery (as opposed to the detection of error) will be covered in Section 7.

It is time for an example! Consider the specification

```
%token  DING  DONG  DELL
%%
rhyme   :       sound  place
        ;
sound   :       DING  DONG
        ;
place   :       DELL
        ;
```

When Yacc is invoked with the -v option, a file called y.output is produced, with a human-readable description of the parser. The y.output file corresponding to the above grammar (with some statistics stripped off the end) is:

```
state 0
        $accept  :  _rhyme  $end

        DING  shift 3
        .  error

        rhyme  goto 1
        sound  goto 2

state 1
        $accept  :   rhyme_$end

        $end  accept
        .  error

state 2
        rhyme  :   sound_place

        DELL  shift 5
        .  error

        place   goto 4
state 3
        sound    :   DING_DONG

        DONG  shift 6
        .  error

state 4
        rhyme  :   sound  place_    (1)

        .   reduce  1

state 5
        place  :   DELL_    (3)

        .   reduce  3

state 6
        sound    :   DING  DONG_    (2)

        .   reduce  2
```

Notice that, in addition to the actions for each state, there is a description of the parsing rules being processed in each state. The _ character is used to indicate what has been seen, and what is yet to come, in each rule. Suppose the input is

```
        DING  DONG  DELL
```

It is instructive to follow the steps of the parser while processing this input.

Initially, the current state is state 0. The parser needs to refer to the input in order to decide between the actions available in state 0, so the first token, DING, is read, becoming the lookahead token. The action in state 0 on DING is is ``shift 3'', so state 3 is pushed onto the stack, and the lookahead token is cleared. State 3 becomes the current state. The next token, DONG, is read, becoming the lookahead token. The action in state 3 on the token DONG is ``shift 6'', so state 6 is pushed onto the stack, and the lookahead is cleared. The stack now contains 0, 3, and 6. In state 6, without even consulting the lookahead, the parser reduces by rule 2.

```
                sound   :   DING  DONG
```

This rule has two symbols on the right hand side, so two states, 6 and 3, are popped off of the stack, uncovering state 0. Consulting the description of state 0, looking for a goto on sound,

```
                sound   goto 2
```

is obtained; thus state 2 is pushed onto the stack, becoming the current state.

In state 2, the next token, DELL, must be read. The action is ``shift 5'', so state 5 is pushed onto the stack, which now has 0, 2, and 5 on it, and the lookahead token is cleared. In state 5, the only action is to reduce by rule 3. This has one symbol on the right hand side, so one state, 5, is popped off, and state 2 is uncovered. The goto in state 2 on place, the left side of rule 3, is state 4. Now, the stack contains 0, 2, and 4. In state 4, the only action is to reduce by rule 1. There are two symbols on the right, so the top two states are popped off, uncovering state 0 again. In state 0, there is a goto on rhyme causing the parser to enter state 1. In state 1, the input is read; the endmarker is obtained, indicated by ``$end'' in the y.output file. The action in state 1 when the endmarker is seen is to accept, successfully ending the parse.

The reader is urged to consider how the parser works when confronted with such incorrect strings as DING DONG DONG, DING DONG, DING DONG DELL DELL, etc. A few minutes spend with this and other simple examples will probably be repaid when problems arise in more complicated contexts.

5: Precedence

There is one common situation where the rules given above for resolving conflicts are not sufficient; this is in the parsing 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 that 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 that 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, that 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. Sometimes a unary operator and a binary operator have the same symbolic representation, but different precedences. An example is unary and binary '-'; unary minus may be given the same strength as multiplication, or even higher, while binary minus has a lower strength than multiplication. The keyword, %prec, changes 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 following token

name or literal. For example, to make unary minus have the same precedence as multiplication the rules might resemble:

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

        %%

        expr    :       expr  '+'  expr
                |       expr  '-'  expr
                |       expr  '*'  expr
                |       expr  '/'  expr
                |       '-'  expr      %prec  '*'
                |       NAME
                ;
```

A token declared by %left, %right, and %nonassoc need not be, but may be, declared by %token as well.

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 that 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. 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 has no precedence and associativity, 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.

Conflicts resolved by precedence are not counted in the number of shift/reduce and reduce/reduce conflicts reported by Yacc. This means that mistakes in the specification of precedences may 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. The y.output file is very useful in deciding whether the parser is actually doing what was intended.