Write text parsers with yacc and lex

Martin Brown May 31, 2006

Examine the processes behind building a parser using the lex/flex and yacc/bison tools, first to build a simple calculator and then delve into how you can adopt the same principles for text parsing. Parsing text -- that is, understanding and extracting the key parts of the text -- is an important part of many applications. Within UNIX®, many elements of the operating system rely on parsing text, from the shell you use to interact with the system, through to common tools and commands like awk or Perl, right through to the C compiler you use to build software and applications. You can use parsers in your UNIX applications (and others), either to build simple configuration parsers or even to build the ultimate: your own programming language.

Before you start

UNIX® programmers often find that they need to understand text and other structures with a flexible, but standardized format. By using the lex and yacc tools, you can build a parsing engine that processes text according to specific rules. You can then incorporate it into your applications for everything from configuration parsing right up to building your own programming language. By the end of this tutorial, you'll understand how to define lexical elements, write yacc rules, and use the rule mechanism to build and define a range of different parsing engines and applications.

About this tutorial

There are many ways to understand and extract text in UNIX. You can use grep, awk, Perl, and other solutions. But sometimes you want to understand and extract data in a structured, but unrestricted format. This is where the UNIX lex and yacc tools are useful. The previous tools, awk, Perl, along with the shell and many other programming languages, use lex and yacc to generate parsing applications to parse and understand text and translate it into the information, or data structures, that you need.

Lex is a lexical analysis tool that can be used to identify specific text strings in a structured way from source text. Yacc is a grammar parser; it reads text and can be used to turn a sequence of words into a structured format for processing.

In this tutorial, you'll examine how to use lex and yacc, first to build a calculator. Using the calculator as an example, you'll further examine the output and information generated by the lex and yacc system and study how to use it to parse other types of information.

Prerequisites

To use the examples in this tutorial, you will need to have access to the following tools:

- Lex: This tool is a standard component on most UNIX operating systems. The GNU flex tool provides the same functionality.
- Yacc: This tool is standard on most UNIX operating systems. The GNU bison tool provides the same functionality.
- C compiler: Any standard C compiler, including Gnu CC, will be fine.
- Make tool: This tool is required to use the sample Makefile to simplify building.

GNU tools can be downloaded from the GNU Web site, or your local GNU mirror.

Lexical analysis with lex

The first stage to writing a text parser is to be able to identify what it is you are reading. There are many different methods for this, but the easiest is to use lex, a tool that converts input information into a series of tokens.

What is lexical analysis?

When you write a program in any language or type a command at the command line, have you thought about what goes on behind the scenes to turn what you type into a set of instructions?

The process is very simple, and yet quite complex. It is complex, because there are a seemingly endless array of possible combinations and sequences of information that can be typed. For example, to iterate through a hash within the Perl language, you might use a sequence like Listing 1.

Listing 1. Iterating through a hash within Perl

```
foreach $key (keys %hash)
{
...
}
```

Each of those items has significance, in different ways, and this is where the simplicity of the process is apparent. There is a structure to the expression shown in Listing 1, there are specific rules in programming languages just as there are with human languages. Therefore, if you break down the input into the combination of what you are seeing and the structure of that information, actually parsing the content is quite simple.

In order to understand the information supplied to a text parsing application, there are two phases. The first is simply to identify what has been typed or provided to an application. You must be able to identify the key words, phrases, or character sequences from the input source so that you can determine what to do with them. The second process is to understand the structure of that information -- the grammar -- so that the input can be both validated and operated on. An excellent example of grammar is the use of parentheses in most programming languages. It is fairly obvious that the following is wrong:

```
{ function)( {
```

The braces are not matched, and the parentheses are in the wrong order. For a parser to understand and recognize it, it must know the correct sequences and what to do when it matches the sequence.

Lexical analysis starts with the process of identify the input data, and that can be handled by the lex tool.

The lex tool

The lex tool (or the GNU tool, flex) uses a configuration file to generate C source core, which you can then use either to make a standalone application, or you can use it within your own application. The configuration file defines the character sequences you expect to find in the file that you want to parse, and what should happen when this sequence is discovered. The format of the file is straightforward, you specify the input sequence and the result, separated by a space (or tab). For example:

```
sequence do-something
```

Listing 2 shows a very simple definition that accepts words and prints out a string based on the supplied word.

Listing 2. Simple lex definition

```
%{
#include <stdio.h>
%}

%%
begin  printf("Started\n");
hello  printf("Hello yourself!\n");
thanks printf("Your welcome\n");
end    printf("Stopped\n");
%%
```

The first block, defined by the `%{...%}`, defines the text that will be inserted into the generated C source. In this case, because the examples later use the `printf()` function, you ensure that the `stdio.h` header is included.

The second block, identified by the `%%` sequence, contains the definitions of the identified string input and the result. In these cases, for a simple word, an appropriate response is printed.

Generating the C source

To generate the C source that will actually parse some input text, run lex (or flex) on the file shown in Listing 1. Lex/flex files have the dot-suffix of 'l', so the above file could be called exampleA.l. To generate the C source:

```
$ flex exampleA.l
```

Regardless of the tool you are using, the output will be called lex.yy.c. Examination of this file is not for the faint-hearted; the process behind the parser that it identifies is actually quite complicated

and is based around a complex table-based parsing system that matches the input text against the definitions from your original lex definition. Because of this association, the code is comparatively memory-hungry, especially on much larger and complex files.

The benefits of flex over lex are the addition of a number of options designed to improve performance (either for memory or speed), debugging options, and improved control of the scanner behavior (for example, to ignore case). You can generate C source that is close to that generated by the original lex tool using the -l command line option when generating the C source code.

Now that you have the C source, you can compile this into an application to test the process:

```
$ gcc -o exampleA lex.yy.c -lfl
```

The flex library (included with `-lfl`, use `-ll` for lex) incorporates a simple `main()` function that executes the analysis code. When running the generated application, it waits for input. Listing 3 shows the input (and output) of the application.

Listing 3. Simple lex application input/output

```
$ exampleA
begin
Started

hello
Hello yourself!

end
Stopped

thanks
Your welcome

hello thanks
Hello yourself!
 Your welcome

hello Robert
Hello yourself!
 Robert
```

For single line input ('begin'), the application responds with the command you supplied, in this case, to print out the word 'Started'. For multi-word lines where the word is identified, the application runs both commands separated by a space. For tokens not recognized (and that includes whitespace), these are just echoed back.

The example shows the basic operation of the system, but you have only used standard words. To use other combinations, such as characters and elements, there are a range of different solutions available.

Identifying elements

Identified elements do not have to be the fixed strings shown earlier; the identifier supports regular expressions and special (for example, punctuation) characters, as shown here in Listing 4.

Listing 4. Regular expressions and special characters

```
%{
#include <stdio.h>
%}

%%
[a-z]    printf("Lowercase word\n");
[A-Z]    printf("Uppercase word\n");
[a-ZA-Z] printf("Word\n");
[0-9]    printf("Integer\n");
[0-9.]   printf("Float\n");
";"      printf("Semicolon\n");
"("      printf("Open parentheses\n");
")"      printf("Close parentheses\n");
%%
```

The examples in Listing 4 should be self explanatory, and the same principles can be used for any regular expression or special character that you want to be able to parse.

True tokenization

Earlier examples have built C source code that is essentially standalone. While there is no problem with this approach, it isn't all that useful for parsing text or other items where there are multiple words, phrases, or sequences in a given instruction.

Parsing sequences in this way requires a grammar parser, something that will define the sequence of tokens. But the grammar parser must know what tokens are expected. To return an identifying token, the operation when a token is identified is changed for echoing a string within the lex definition. For example, you could rewrite the original example to that shown in Listing 5.

Listing 5. Returning tokens

```
%{
#include <stdio.h>
#include <y.tab.h>
%}

%%
begin  return BEGIN;
hello  return HELLO;
thanks return THANKS;
end    return END;
%%
```

Now, rather than printing out a string when the 'hello' token is identified, you return the token name. This name will be used within yacc to build up the grammar.

The token names have not been explicitly defined in this example. They are actually specified in the y.tab.h file, which is automatically generated by yacc when parsing the yacc grammar file.

Extracting variable data

If you want to actually extract a value (for example, you want to be able to read a number or string), then you must specify how the input is converted into the type you want. This is not normally very useful within the confines of lex, but it is vital when working with the yacc tool to build a full parser.

Two variables are generally used to exchange information; the `yytext` variable holds the raw data read by lex during parsing, while `yylval` is used to exchange the actual value between the two systems. You'll look at how this integration works in more detail later in this tutorial but, for the purposes of identifying information, you might use a lex definition like this:

```
[0-9]   yylval=atoi(yytext); return NUMBER;
```

In the above line, you set the value of `yylval` by converting the input string fragment that matches the regular expression (and which is stored in `yytext`) into an integer using the standard `atoi()` function. Note that although you have converted the value, you still have to return the value type so that yacc knows what the value type is and can used the token within it's own definitions.

Let's see how this works by looking at how yacc defines the grammar structure.

Grammar analysis with yacc

Grammar definitions are used to define a sequence of tokens and what the result should be. In general this is used in combination with lex, which actually reads the text, and the two make up the parser.

Basic grammar with yacc

The basic structure of the yacc grammar definition is to define the sequence of expected tokens. For example, you might define the expression A + B, where A and B are numbers using the following definition:

```
NUMBER PLUSTOKEN NUMBER { printf("%f\n",($1+$3)); }
```

The `NUMBER` is the identifying token for a `NUMBER`, and the `PLUSTOKEN` is the identifying token for the plus sign.

The code within the braces defines what to do when this sequence is identified, in this case, the C code to add two numbers together is given. Note that you use the nomenclature of `$1` to specify the first token in the grammar expression and `$3` for the third item.

For the grammar sequence to be identified, you must give the sequence a name, as shown in Listing 6.

Listing 6. Naming the sequence

```
addexpr: NUMBER PLUSTOKEN NUMBER
         {
             printf("%f\n",($1+$3));
         }
         ;
```

The name of the grammar group is `addexpr` and the definition for the group is terminated by a semicolon.

A grammar definition can contain multiple sequences (and related operations) that are part of a particular group. For example, in a calculator, the addition and subtraction processes are similar operations so you might group them together. The individual definitions within a group are separated by the pipe (|) symbol (see Listing 7).

Listing 7. Individual definitions within a group are separated by the pipe symbol

```
addexpr: NUMBER PLUSTOKEN NUMBER
         {
             printf("%f\n",($1+$3));
         }
         | NUMBER MINUSTOKEN NUMBER
         {
             printf("%f\n",($1-$3));
         }
         ;
```

Now that you understand the basic grammar specification, you need to understand how you combine multiple sequences and their priority.

Grammar sequence and precedence

In most languages, whether textual, mathematical, or programming, there is usually some sort of precedence or significance to different phrases that give them priority over similar, but different, combinations.

For example, in most programming languages within a mathematical expression, multiplication has higher precedence than addition or subtraction. For example, the expression: `4+5*6` evaluates to: `4+30`.

This ultimately evaluates to 34. The reason is that the multiplication operation happens first (5 times 6 equals 30) and this is then used in the final calculation, which adds the result to 4 to give 34.

Within yacc, you define the precedence by defining multiple grammar sequence groups and linking them; the order of the individual expressions within the group helps to define the precedence. In Listing 8 you can see the code that would be used to define the above behavior.

Listing 8. Linking grammar and providing precedence

```
add_expr: mul_expr
        | add_expr PLUS mul_expr  { $$ = $1 + $3; }
        | add_expr MINUS mul_expr { $$ = $1 - $3; }
        ;
mul_expr: primary
        | mul_expr MUL primary { $$ = $1 * $3; }
        | mul_expr DIV primary { $$ = $1 / $3; }
        ;
primary: NUMBER { $$ = $1; }
        ;
```

All expressions evaluated by yacc are processed from the left to the right, so in a compound expression like your original example (for example, `4+5*6`) yacc will look for the best match, according to the rules, that matches part of your expression.

Matches are processed in the order defined by the rules, top to bottom. So, the process starts by first trying to match against the grammar rules in `add_expr`. However, because multiplication has higher precedence, the rules state that yacc should first try `mul_expr`. This forces yacc to actually try to match against a multiplication statement first (as defined in `mul_expr`). If this fails, it then tries the next rule in the `add_expr` block, and so on, until either the expression has been resolved, or no match could be found and an error is generated.

When evaluating the equation `4+5*6` with the above ruleset, yacc first tries `add_expr`, gets redirected to `mul_expr` (as the first rule to match), and in turn gets redirected to primary. The primary rule sets the value of the numbers in the expression. The `$$` effectively sets the return value for that portion of the expression.

Once the numbers have been matched, yacc identifies a multiplication expression from the definition in this line: `| mul_expr MUL primary { $$ = $1 * $3; }`, which sets the return value for the `5*6` portion of the original expression. Yacc generates the code that performs the calculation (extracting the values that had previous been set by the primary rule), and replacing that original portion of the input expression with the value of the calculation. This means that yacc now has to match the following expression: `4+30`.

It can match this expression using the rule: `| add_expr PLUS mul_expr { $$ = $1 + $3; }`, which ultimately calculates the final result of 34.

Topping and tailing the grammar definition

The grammar definition sets the rules for parsing the input tokens and their format and layout into something that can be used. Remember that at all times, the rulesets defined here tell yacc how to identify the input and execute a fragment of C code. The yacc tool generates the C code required to parse this information; yacc doesn't do the parsing.

The code in the braces is quasi-C source code. Yacc handles the translation from the original into the C source code based on your definitions and the rest of the code that is required to actually parse the content.

In addition to the rulesets that define how to parse the input, there are additional functions and options that help to define the rest of the C source code. The basic format for the entire yacc definition file is similar to that used by lex/flex, as shown in Listing 9 below.

Listing 9. Yacc file format

```
%{
/* Global and header definitions required */
%}

/* Declarations (Optional token definitions) */

%%
/* Parsing ruleset definitions
%%

/* Additional C source code */
```

The global/header definitions are the same as those used in lex/flex files and are required if you are using special functions when parsing and processing the output.

Token definitions relate not only to the tokens expected, but also the precedence within the parsing process. This adjusts how yacc processes rules by forcing it to examine tokens in a specific order, and also affects how it deals with the expressions when compared to the rules.

For example, you would normally define the `+` token to have left precedence, that is, any expression to the left of this token is matched first, so that the expression `4+5+6` is evaluated first as `4+5` and then `9+6`.

For some tokens, however, you would want to specify right precedence; for example, to evaluate the logical `NOT`, (for example, `! (expr)`) you would want `expr` evaluated before its value was inversed by the `!` token.

By specifying the precedence of the tokens here, in addition to controlling the precedence of the relationship between rules, you can be very specific about how the input is evaluated.

There are three main token types, `%token` (no precedence), `%left` (precedence is given to the input on the left of this token) and `%right` (precedence is given to the input to the right of this token). For example, Listing 10 defines a number of tokens according to their appropriate precedence:

Listing 10. Defining a number of tokens according to their precedence

```
%token EQUALS POWER
%left PLUS MINUS
%left MULTIPLY DIVIDE
%right NOT
```

Note that the precedence is specified in increasing precedence from top to bottom, and tokens on the same line are given the same precedence. In the example in Listing 10, `MULTIPLY` and `DIVIDE` have the same precedence, which is higher than the precedence given to `PLUS` and `MINUS`.

Note that any tokens not defined here, but incorporated in the rules, will raise an error.

Customizing initialization and errors

Two key functions that you will want defined are the `main()` function, into which you can place your own custom initialization and other processes, and the `yyerror()`, which prints out an error when a

parsing rule for the input cannot be found. The actual parsing function that is generated by yacc is called `yyparse()`. Typical definitions for the custom functions are shown in Listing 11.

Listing 11. Custom functions for a parser

```
#include <stdio.h>
#include <ctype.h>
char *progname;
double yylval;

main( argc, argv )
char *argv[];
{
  progname = argv[0];
  yyparse();
}

yyerror( s )
char *s;
{
    fprintf( stderr ,"%s: %s\n" , progname , s );
}
```

The `yyerror()` function accepts a string, and this is output in combination with the program name when an error is raised.

Compiling grammar into C

Yacc (originally short for yet another compiler compiler) and the GNU bison tool both take a grammar definition file and generate the necessary C source to create a parser that processes the appropriate input.

Grammar definition files generally have the extension of .y.

When you run yacc, it creates a file called yy.tab.c by default. If you are using yacc in combination with lex, then you will also want to generate a C header file, which contains the macro definitions for the tokens you are identifying in both systems. To generate a header file in addition to the C source, use the `-d` command line option:

```
$ yacc -d calcparser.y
```

Although bison performs the same basic operation, by default, it generates files that are identified by the prefix of the source filename. So, in the above example, using bison would actually generate the files calcparser.tab.c and calcparser.tab.h.

The distinction is important, not only because you need to know which files to compile, but also because you will need to import the correct header file in your lex.l definition so that it reads the correct token definitions.

When compiling a lex/yacc application, the general process is:

1. Run yacc on your parser definition.
2. Run lex on your lexical definition.

3. Compile the generated yacc source.
4. Compile the generated lex source.
5. Compile any other modules.
6. Link lex, yacc, and your other sources into an executable.

I tend to use a Makefile like the one shown here in Listing 12.

Listing 12. Simple lex/yacc Makefile

```
YFLAGS        = -d

PROGRAM       = calc

OBJS          = calcparse.tab.o lex.yy.o fmath.o const.o

SRCS          = calcparse.tab.c lex.yy.c fmath.c const.c

CC            = gcc

all:            $(PROGRAM)

.c.o:           $(SRCS)
                $(CC) -c $*.c -o $@ -O

calcparse.tab.c:        calcparse.y
                bison $(YFLAGS) calcparse.y

lex.yy.c:       lex.l
                flex lex.l

calc:           $(OBJS)
                $(CC) $(OBJS)  -o $@ -lfl -lm

clean:;         rm -f $(OBJS) core *~ \#* *.o $(PROGRAM) \
                y.* lex.yy.* calcparse.tab.*
```

Exchanging data with lex

The primary method of data exchange between lex and yacc is the token definitions that are generated when you create the C source from yacc and generate the header file that contains the C macros for each token.

However, if you want to exchange more than a token, you must set up a value that can hold the information you want to exchange.

There are two stages to this process. First, you should define the `YYSTYPE` according to your needs. This is the default value used to exchange data between the two systems. For a basic calculator, you might want to define this as an int (actually the default), float, or double. If you want to exchange text, set this as a character pointer or array. If you want to exchange both, then you should create a union structure that can hold both values.

You should then also declare the `yylval` variable (and indeed any other variable you want to share) as this type in both your yacc and lex definitions.

For example, within the header block in both files you would specify the type (see Listing 13).

Listing 13. Specifying the type

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

Within the lex definition, you would also define that the `yylval` variable was an external variable of this type:

```
extern YYSTYPE yylval;
```

Finally, in the custom C initialization block of the yacc file, you would define the actual variable:

```
YYSTYPE yylval;
```

Now you can exchange data between the C source generated by both systems.

Building a calculator

Using the techniques demonstrated in the previous section, you can build a natural expression calculator.

Calculator basics

You can build a program that is capable of processing expressions, as shown in Listing 14.

Listing 14. Program capable of processing expressions

```
4+5
(4+5)*6
2^3/6
sin(1)+cos(PI)
```

By defining additional tokens and grammar rules, you can extend that functionality even further to include a range of functions and equations, not only built into the standard C language and math library, but also those that you specifically define.

To be able to parse all of these different elements, the first stage is to define the tokens that will be identified by the lexical analysis component. You can see the full lex definition in Listing 15.

Listing 15. Lex file for an advanced calculator

```
%{
#define YYSTYPE double
#include "calcparse.tab.h"
#include <math.h>
extern double yylval;
%}
D       [0-9.]
%%
[ \t]   { ; }
log      return LOG;
pi      return PIVAL;
sin     return SIN;
cos     return COS;
```

```
tan     return TAN;
and     return AND;
not     return NOT;
xor     return XOR;
or      return OR;
reg     return REGA;
ans     return ANS;
fix     return FIX;
sci     return SCI;
eng     return ENG;
const   return CONST;
bintodec return BINTODEC;
dectobin return DECTOBIN;
{D}+    { sscanf( yytext, "%lf", &yylval ); return NUMBER ; }
[a-zA-Z_]+ return IDENT;
"["     return OPENREG;
"]"     return CLOSEREG;
"<<"    return LEFTSHIFT;
">>"    return RIGHTSHIFT;
"++"    return INC;
"--"    return DEC;
"+"     return PLUS;
"-"     return MINUS;
"~"     return UNARYMINUS;
"/"     return DIV;
"*"     return MUL;
"^"     return POW;
"!"     return FACT;
"("     return OPENBRACKET;
")"     return CLOSEBRACKET;
"%"     return MOD;
"^^"     return XOR;
"!!"     return NOT;
"="     return ASSIGN;
"&&"    return LAND;
"||"    return OR;
"|"     return IOR;
"&"     return AND;
"~~"     return COMPLEMENT;
"\n"    return EOLN;
```

There are a range of tokens here, many of which will be self explanatory. The tokens include basic math operations, a number of functions (sin, cos, and so forth) and logical operators. Also note that you are using a double for the values, using `sscanf()` to parse the string of numbers and a decimal point into a suitable double value.

Calculator grammar

Based on the tokens in the previous section, a number of grammar rules exist that are used to parse these. The full code for the grammar parser is shown in Listing 16. Let's take a closer look at some of the highlights and how the system works.

Listing 16. The calculator grammar file

```
%{
#include <alloca.h>
#include <math.h>
#include <stdlib.h>
#include <stddef.h>
#include <ctype.h>
#define YYSTYPE double
double calcfact();
double reg[99];
```

```
double ans;
char format[20];
%}

%token NUMBER SPACE MOD RIGHTSHIFT LEFTSHIFT SEMICOLON SIN EOLN PIVAL
%token PLUS MINUS DIV MUL POW OPENBRACKET CLOSEBRACKET UNARYMINUS
%token COS TAN ASIN ACOS ATAN FACT INC DEC LAND OR COMPLEMENT
%token NOT XOR ASSIGN IOR AND OPENREG CLOSEREG REGA ANS FIX SCI ENG
%token CONST
%left PLUS MINUS
%left MUL DIV
%left UNARYMINUS
%left LAND OR XOR NOT AND IOR
%left LOG
%%
list:   /* nothing */
        | list EOLN
        | list expr EOLN
          { printf( format , (double) $2 ); ans=$2; }
        ;
expr:   conditional_expr
        ;
conditional_expr: logical_or_expr
        ;
logical_or_expr: logical_and_expr
        | logical_or_expr OR logical_and_expr
          { $$ = (int) $1 || (int) $3; }
        ;
logical_and_expr: inclusive_or_expr
        | logical_and_expr LAND inclusive_or_expr
          { $$ = (int) $1 && (int) $3; }
        ;
inclusive_or_expr: exclusive_or_expr
        | inclusive_or_expr IOR exclusive_or_expr
          { $$ = (int) $1 | (int) $3; }
        ;
exclusive_or_expr: and_expr
        | exclusive_or_expr XOR and_expr
          { $$ = (int) $1 ^ (int) $3; }
        ;
and_expr: shift_expr
        | and_expr AND shift_expr
          { $$ = (int) $1 & (int) $3; }
        ;
shift_expr: pow_expr
        | shift_expr LEFTSHIFT pow_expr
          { $$ = (int) $1 << (int) $3; }
        | shift_expr RIGHTSHIFT pow_expr
          { $$ = (int) $1 >>(int) $3; }
        ;
pow_expr: add_expr
        | pow_expr POW add_expr { $$ = pow($1,$3); }
        ;
add_expr: mul_expr
        | add_expr PLUS mul_expr  { $$ = $1 + $3; }
        | add_expr MINUS mul_expr { $$ = $1 - $3; }
        ;
mul_expr: unary_expr
        | mul_expr MUL unary_expr { $$ = $1 * $3; }
        | mul_expr DIV unary_expr { $$ = $1 / $3; }
        | mul_expr MOD unary_expr { $$ = fmod($1,$3); }
        ;
unary_expr: assign_expr
        | MINUS primary %prec UNARYMINUS { $$ = -$2; }
        | INC unary_expr { $$ = $2+1; }
        | DEC unary_expr { $$ = $2-1; }
        | NOT unary_expr { $$ = !$2; }
```

```
        | LOG unary_expr { $$ = log($2); }
        ;
assign_expr: postfix_expr
        | REGA OPENREG expr CLOSEREG ASSIGN postfix_expr
          { reg[(int)$3]=$6; $$=$6; }
        | REGA OPENREG expr CLOSEREG
          { $$=reg[(int)$3]; }
        | REGA
          { int i;
            for(i=0;i<100;i++)
              if (reg[i]!=0)
                 printf("%02d = %.2f\n",i,reg[i]);
            $$=0;
          }
        ;
postfix_expr: primary
        | postfix_expr INC { $$ = $1+1; }
        | postfix_expr DEC { $$ = $1-1; }
        | postfix_expr FACT
          { $$ = calcfact((unsigned long int)$1); }
        ;
 primary: NUMBER { $$ = $1; }
        | PIVAL { $$ = M_PI; }
        | OPENBRACKET expr CLOSEBRACKET { $$ = $2; }
        | ANS { $$ = ans; }
        | CONST OPENBRACKET expr CLOSEBRACKET { $$ = constval($3); }
        | set_format
        ;
set_format: function_call
        | FIX OPENBRACKET expr CLOSEBRACKET
             { sprintf(format,"%%.%df\n",(int)$3); $$=0; }
        | FIX { sprintf(format,"%%f\n"); $$=0; }
        | SCI OPENBRACKET expr CLOSEBRACKET
             { sprintf(format,"%%.%dg\n",(int)$3); $$=0; }
        | SCI { sprintf(format,"%%g\n"); $$=0; }
        | ENG OPENBRACKET expr CLOSEBRACKET
             { sprintf(format,"%%.%de\n",(int)$3); $$=0; }
        | ENG { sprintf(format,"%%e\n"); $$=0; }
        ;
function_call: SIN OPENBRACKET expr CLOSEBRACKET
              { $$ = (cos($3)*tan($3)); }
        | COS OPENBRACKET expr CLOSEBRACKET
              { $$ = cos($3); }
        | TAN OPENBRACKET expr CLOSEBRACKET
              { $$ = tan($3); }
        | ASIN OPENBRACKET expr CLOSEBRACKET
              { $$ = asin($3); }
        | ACOS OPENBRACKET expr CLOSEBRACKET
              { $$ = acos($3); }
        | ATAN OPENBRACKET expr CLOSEBRACKET
              { $$ = atan($3); }
        ;
%%

#include <stdio.h>
#include <ctype.h>
char *progname;
double yylval;

main( argc, argv )
char *argv[];
{
  progname = argv[0];
  strcpy(format,"%g\n");
  yyparse();
}
```

```
yyerror( s )
char *s;
{
  warning( s , ( char * )0 );
  yyparse();
}

warning( s , t )
char *s , *t;
{
  fprintf( stderr ,"%s: %s\n" , progname , s );
  if ( t )
    fprintf( stderr , " %s\n" , t );
}
```

There are three global structures available for the rest of the application:

- The `reg` array is used as a general memory register, where you can place values and results from calculations.
- The `ans` variable contains the value of the last calculation.
- The `format` is used to hold the output format to be used when printing results.

The results of a calculation are only printed out when the input contains an end-of-line character (identified by the `EOLN` token). This enables a long calculation to be entered on a single line, and for the contents to be parsed and processed before the value is printed out. That operation uses the format specified by the global format variable, and the result is also stored in `ans`.

The bulk of the parser is a combination of identifying a fragment and calculating the result, until individual components of a larger calculation are finally resolved to the final value.

Setting the output format

The format, which is just a suitable format string for `printf()`, is updated by using a function-style call to set the precision and format of the output. For example, to set fixed decimal point output to limit the numbers to the right of the decimal point, you can use `fix(3)`, or you can reset to the default format: `fix()`.

You can see the effect in the sequence in Listing 17.

Listing 17. Output format

```
$ calc
1/3
0.333333
fix(3)
0.000
1/3
0.333
```

Because the format string is global and results are only ever printed out by one rule, it means that you can easily use the format string to control the output.

Calculator registers

The register is basically an array of floating point values that can be accessed by using a numerical reference. The parsing of this section is handled by the fragment shown in Listing 18.

Listing 18. Registers in a calculator

```
assign_expr: postfix_expr
        | REGA OPENREG expr CLOSEREG ASSIGN postfix_expr
          { reg[(int)$3]=$6; $$=$6; }
        | REGA OPENREG expr CLOSEREG
          { $$=reg[(int)$3]; }
        | REGA
          { int i;
            for(i=0;i<100;i++)
              if (reg[i]!=0)
                printf("%02d = %.2f\n",i,reg[i]);
            $$=0;
          }
        ;
```

The first rule allows for the assignment of a value and the parser enables you to use an expression for the register reference, but only to use a `postfix_expr` for the value. This limits the value you can assign to a register, as demonstrated by the sequence in Listing 19.

Listing 19. Limiting the value you can assign to a register

```
reg[0]=26
26
reg[0]
26
reg[1]=(4+5)*sin(1)
7.57324
reg[1]
9
reg[4+5]=29
29
reg[9]
29
```

Note that the middle expression `(4+5)*sin(1)` returns the correct result, but only assigns the value of the first part of the expression to the register. This is because the rules only allow for the assignment of expressions that match the `postfix_assign` rule, and it is actually the `primary` rule that ultimately describes the rule for a parenthesized statement. Because you do not allow the entire input line to be part of the assignment, the parser matches the first fragment (the `(4+5)`) and assigns that value to the register. Because the assignment to the register also returns the register value, the remainder of the calculation continues and prints out the correct result.

There is one simply solution to this. As a simple input rule, you could require that assignment to a register must be enclosed in parentheses:

```
reg[1]=((4+5)*sin(1))
```

This wouldn't require any changes to the rules, since the above method works with your existing parser, but might require changes to the documentation.

The current system also has a benefit in that you can next register assignments. This is made possible because assignments to the register also return the register result. Hence, the following sequence works just fine (see Listing 20).

Listing 20. Register results

```
reg[1]=(45+reg[2]=(3+4))
52
reg[1]
52
reg[2]
7
```

Extending the principles of lex and yacc

The calculator shows the basic process, but there is much more to the lex and yacc tools and how they can be used.

Playing with your calculator parser

Because the rules that parse the input information and what you do with that information can be set and controlled individually, it is actually possible to independently alter both the way the information is extracted and how it is treated.

When I was first introduced to lex and yacc, my original aim was to build a Reverse Polish Notation (RPN) calculator. With RPN, you supply the numbers first, and the operator afterwards, for example, to add two numbers together, you would use: `4 5 +`.

For certain people, this sequence makes more sense, especially if you consider how you would write the same calculation when learning addition at school:

```
4
5 +
----
9
----
```

For more complex calculations, you can load the stack with numbers and then use the arguments -- results are automatically placed back on the stack. `4+5*6` can be written as: `4 5 6 * +`.

For a computer, it is also more straightforward. You can implement an RPN calculator with a stack. When the parser identifies a number, you push it onto a stack, and when you see an operator, you pop the values off the stack and perform the calculation. It simplifies a lot of the processing and complex parsing that you had to go through above to perform normal expression parsing.

However, with a little more work, you can actually create a parser that converts your natural expression input into RPN. Even better, you can convert your RPN into standard expression format.

For example, converting expressions into RPN:

```
$ equtorpn
4+5*6
4 5 6 * +
```

What's interesting here is that the precedence order that was defined in the parsing rules (the yacc parser is a simpler version of the main calculator shown here) leads to an RPN that matches the manual example given above.

It's so good that you can feed the output of the equtorpn tool into the input of the RPN calculator and get the same result:

```
$ equtorpn|rpn
4+5*6
34
```

For the full examples and code for the rpn, equtorpn, and rpntoequ applications and further discussion of how they work, please visit the MCslp Coalface Web site (see Related topics).

Moving to proper text parsing

The examples have shown how you would build an expression parser that turns a mathematical expression in order that the expression can be calculated. This demonstrates the importance of the sequence, parsing rules, and issues like precedence. The same rules can be applied to text processing systems, but processing text can be more complicated unless you institute the same strict rules as have been enforced for the calculator example.

Listing 21 shows the lex file to create a simple set/get state parser, while Listing 22 shows the yacc rules to actually parse the output.

Listing 21. Text based tokenizing with lex

```
%{
#include <stdio.h>
#include "text.tab.h"
%}

%%
set       return SET;
state     return STATE;
mode      return MODE;
get       return GET;
[a-zA-Z]+  { yylval=strdup(yytext); return STRING; }
%%
```

Listing 22. Yacc rules for parsing text

```
%{
#include <stdlib.h>
#define YYSTYPE char *
char mode[20];
char state[20];
%}

%token SET STATE MODE GET STRING EOLN
%%
```

```
list:   /* nothing */
        | list getorset
        ;
getorset: getvalue
         | setvalue
         ;
setvalue:
        SET STATE STRING  { strcpy(state,$3); printf("State set\n"); }
        | SET MODE STRING { strcpy(mode,$3); printf("Mode set\n"); }
        ;
getvalue:
        GET STATE  { printf("State: %s\n",state); }
        | GET MODE { printf("Mode: %s\n",mode); }
        ;

%%

#include <stdio.h>
#include <ctype.ha>
char *progname;

main( argc, argv )
char *argv[];
{
  progname = argv[0];
  yyparse();
}

yyerror( s )
char *s;
{
  fprintf( stderr ,"%s: %s\n" , progname , s );
}
```

Unlike the calculator example, where the input text had to be converted to a number, you can actually use the input string directly without conversion (see Listing 23).

Listing 23. Using the input string directly

```
$ textparser
get state
 State:

set state bob
  State set

get state
 State: bob
```

More complex text parsers build on the same basic principles and can be used for anything from configuration file parsers, to building your own scripting language, or even a compiled alternative.

Language compilers

Language compilers (for example, a C compiler) take your input and convert that input into raw assembly language for a given target platform. This is why, in the first instance, compilers are architecture-specific and why it is comparatively easy to write a compiler that generates code for an alternative platform (cross compiling). The compiler just generates assembly code for an alternative platform. For example, to add two numbers together, you might alter the ruleset used earlier for the purpose to generate Intel x86 assembly code like this:

```
add_expr: mul_expr
        | add_expr PLUS mul_expr
          { printf("MOV ax,%d\nADD ax,%d\n",$1,$3); }
```

Other structures, like loops, iteration, functions, variable names, and other elements, can be added to the system in order to build up a complete language to assembler conversion process.

Summary

Through the combination of lex and yacc, you can generate the code that builds a parser. The lex tool provides a method for identifying text fragments and elements in order to return tokens. The yacc tool provides a method of taking the structure of those tokens using a series of rules to describe the format of the input and defines a method for dealing with the identified sequences. In both cases, the tools use a configuration file, which when processed generates the C source code to build the appropriate parsing application.

In this tutorial, you've seen examples of how to use these applications to build a simple parser for a calculator application. Through that process, you've seen the importance of the tokenizing, the rulesets, and how rulesets interact to provide a complete parsing solution. You've also seen how the basic principles can be used for other processing, such as text parser and even for building your own programming language.

Related topics

- Lex & Yacc Page: This site provides links to a number of different resources.
- GNU: Get versions of lex and yacc from the GNU site.
- Want more? The developerWorks AIX and UNIX zone hosts hundreds of informative articles and introductory, intermediate, and advanced tutorials.