Lex and YACC primer/HOWTO

PowerDNS BV (bert hubert <bert@powerdns.com>)
v0.8 $Date: 2004/09/20 07:14:23 $

This document tries to help you get started using Lex and YACC

Contents

1 Introduction

Welcome, gentle reader.

If you have been programming for any length of time in a Unix environment, you will have encountered the mystical programs Lex & YACC, or as they are known to GNU/Linux users worldwide, Flex & Bison, where Flex is a Lex implementation by Vern Paxson and Bison the GNU version of YACC. We will call these programs Lex and YACC throughout - the newer versions are upwardly compatible, so you can use Flex and Bison when trying our examples.

These programs are massively useful, but as with your C compiler, their manpage does not explain the language they understand, nor how to use them. YACC is really amazing when used in combination with Lex, however, the Bison manpage does not describe how to integrate Lex generated code with your Bison program.

1.1 What this document is NOT

There are several great books which deal with Lex & YACC. By all means read these books if you need to know more. They provide far more information than we ever will. See the 'Further Reading' section at the end. This document is aimed at bootstrapping your use of Lex & YACC, to allow you to create your first programs.

The documentation that comes with Flex and BISON is also excellent, but no tutorial. They do complement my HOWTO very well though. They too are referenced at the end.

I am by no means a YACC/Lex expert. When I started writing this document, I had exactly two days of experience. All I want to accomplish is to make those two days easier for you.

In no way expect the HOWTO to show proper YACC and Lex style. Examples have been kept very simple and there may be better ways to write them. If you know how to, please let me know.

1.2 Downloading stuff

Please note that you can download all the examples shown, which are in machine readable form. See the *homepage* `http://ds9a.nl/lex-yacc` for details.

1.3 License

2 What Lex & YACC can do for you

When properly used, these programs allow you to parse complex languages with ease. This is a great boon when you want to read a configuration file, or want to write a compiler for any language you (or anyone else) might have invented.

With a little help, which this document will hopefully provide, you will find that you will never write a parser again by hand - Lex & YACC are the tools to do this.

2.1 What each program does on its own

Although these programs shine when used together, they each serve a different purpose. The next chapter will explain what each part does.

3 Lex

The program Lex generates a so called 'Lexer'. This is a function that takes a stream of characters as its input, and whenever it sees a group of characters that match a key, takes a certain action. A very simple example:

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

%%
stop    printf("Stop command received\n");
start   printf("Start command received\n");
%%
```

The first section, in between the %{ and %} pair is included directly in the output program. We need this, because we use printf later on, which is defined in stdio.h.

Sections are separated using '%%', so the first line of the second section starts with the 'stop' key. Whenever the 'stop' key is encountered in the input, the rest of the line (a printf() call) is executed.

Besides 'stop', we've also defined 'start', which otherwise does mostly the same.

We terminate the code section with '%%' again.

To compile Example 1, do this:

```
lex example1.l
cc lex.yy.c -o example1 -ll

NOTE: If you are using flex, instead of lex, you may have to change '-ll' to '-lfl' in
the compilation scripts.  RedHat 6.x and SuSE need this, even when you invoke 'flex' as
'lex'!
```

This will generate the file 'example1'. If you run it, it waits for you to type some input. Whenever you type something that is not matched by any of the defined keys (ie, 'stop' and 'start') it's output again. If you enter 'stop' it will output 'Stop command received';

Terminate with a EOF (^D).

You may wonder how the program runs, as we didn't define a main() function. This function is defined for you in libl (liblex) which we compiled in with the -ll command.

3.1 Regular expressions in matches

This example wasn't very useful in itself, and our next one won't be either. It will however show how to use regular expressions in Lex, which are massively useful later on.

Example 2:

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

%%
[0123456789]+           printf("NUMBER\n");
[a-zA-Z][a-zA-Z0-9]*    printf("WORD\n");
%%
```

This Lex file describes two kinds of matches (tokens): WORDs and NUMBERs. Regular expressions can be pretty daunting but with only a little work it is easy to understand them. Let's examine the NUMBER match:

[0123456789]+

This says: a sequence of one or more characters from the group 0123456789. We could also have written it shorter as:

[0-9]+

Now, the WORD match is somewhat more involved:

[a-zA-Z][a-zA-Z0-9]*

The first part matches 1 and only 1 character that is between 'a' and 'z', or between 'A' and 'Z'. In other words, a letter. This initial letter then needs to be followed by zero or more characters which are either a letter or a digit. Why use an asterisk here? The '+' signifies 1 or more matches, but a WORD might very well consist of only one character, which we've already matched. So the second part may have zero matches, so we write a '*'.

This way, we've mimicked the behaviour of many programming languages which demand that a variable name *must* start with a letter, but can contain digits afterwards. In other words, 'temperature1' is a valid name, but '1temperature' is not.

Try compiling Example 2, lust like Example 1, and feed it some text. Here is a sample session:

```
$ ./example2
foo
WORD

bar
WORD

123
NUMBER

bar123
WORD

123bar
NUMBER
WORD
```

You may also be wondering where all this whitespace is coming from in the output. The reason is simple: it was in the input, and we don't match on it anywhere, so it gets output again.

The Flex manpage documents its regular expressions in detail. Many people feel that the perl regular expression manpage (perlre) is also very useful, although Flex does not implement everything perl does.

Make sure that you do not create zero length matches like '[0-9]*' - your lexer might get confused and start matching empty strings repeatedly.

3.2 A more complicated example for a C like syntax

Let's say we want to parse a file that looks like this:

```
logging {
        category lame-servers { null; };
        category cname { null; };
};

zone "." {
        type hint;
        file "/etc/bind/db.root";
};
```

We clearly see a number of categories (tokens) in this file:

- WORDs, like 'zone' and 'type'
- FILENAMEs, like '/etc/bind/db.root'
- QUOTEs, like those surrounding the filename
- OBRACEs, {
- EBRACEs, }
- SEMICOLONs, ;

The corresponding Lex file is Example 3:

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

%%
[a-zA-Z][a-zA-Z0-9]*    printf("WORD ");
[a-zA-Z0-9\/.-]+        printf("FILENAME ");
\"                      printf("QUOTE ");
\{                      printf("OBRACE ");
\}                      printf("EBRACE ");
;                       printf("SEMICOLON ");
\n                      printf("\n");
[ \t]+                  /* ignore whitespace */;
%%
```

When we feed our file to the program this Lex file generates (using example3.compile), we get:

```
WORD OBRACE
WORD FILENAME OBRACE WORD SEMICOLON EBRACE SEMICOLON
WORD WORD OBRACE WORD SEMICOLON EBRACE SEMICOLON
EBRACE SEMICOLON

WORD QUOTE FILENAME QUOTE OBRACE
WORD WORD SEMICOLON
WORD QUOTE FILENAME QUOTE SEMICOLON
EBRACE SEMICOLON
```

When compared with the configuration file mentioned above, it is clear that we have neatly 'Tokenized' it. Each part of the configuration file has been matched, and converted into a token.

And this is exactly what we need to put YACC to good use.

3.3 What we've seen

We've seen that Lex is able to read arbitrary input, and determine what each part of the input is. This is called 'Tokenizing'.

4 YACC

YACC can parse input streams consisting of tokens with certain values. This clearly describes the relation YACC has with Lex, YACC has no idea what 'input streams' are, it needs preprocessed tokens. While you can write your own Tokenizer, we will leave that entirely up to Lex.

A note on grammars and parsers. When YACC saw the light of day, the tool was used to parse input files for compilers: programs. Programs written in a programming language for computers are typically *not* ambiguous - they have just one meaning. As such, YACC does not cope with ambiguity and will complain about shift/reduce or reduce/reduce conflicts. More about ambiguity and YACC "problems" can be found in 'Conflicts' chapter.

4.1 A simple thermostat controller

Let's say we have a thermostat that we want to control using a simple language. A session with the thermostat may look like this:

```
heat on
        Heater on!
heat off
        Heater off!
target temperature 22
        New temperature set!
```

The tokens we need to recognize are: heat, on/off (STATE), target, temperature, NUMBER.

The Lex tokenizer (Example 4) is:

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

```
%%
[0-9]+                  return NUMBER;
heat                    return TOKHEAT;
on|off                  return STATE;
target                  return TOKTARGET;
temperature             return TOKTEMPERATURE;
\n                      /* ignore end of line */;
[ \t]+                  /* ignore whitespace */;
%%
```

We note two important changes. First, we include the file 'y.tab.h', and secondly, we no longer print stuff, we return names of tokens. This change is because we are now feeding it all to YACC, which isn't interested in what we output to the screen. Y.tab.h has definitions for these tokens.

But where does y.tab.h come from? It is generated by YACC from the Grammar File we are about to create. As our language is very basic, so is the grammar:

```
commands: /* empty */
        | commands command
        ;

command:
        heat_switch
        |
        target_set
        ;

heat_switch:
        TOKHEAT STATE
        {
                printf("\tHeat turned on or off\n");
        }
        ;

target_set:
        TOKTARGET TOKTEMPERATURE NUMBER
        {
                printf("\tTemperature set\n");
        }
        ;
```

The first part is what I call the 'root'. It tells us that we have 'commands', and that these commands consist of individual 'command' parts. As you can see this rule is very recursive, because it again contains the word 'commands'. What this means is that the program is now capable of reducing a series of commands one by one. Read the chapter 'How do Lex and YACC work internally' for important details on recursion.

The second rule defines what a command is. We support only two kinds of commands, the 'heat_switch' and the 'target_set'. This is what the |-symbol signifies - 'a command consists of either a heat_switch or a target_set'.

A heat_switch consists of the HEAT token, which is simply the word 'heat', followed by a state (which we defined in the Lex file as 'on' or 'off').

Somewhat more complicated is the target_set, which consists of the TARGET token (the word 'target'), the TEMPERATURE token (the word 'temperature') and a number.

4.1.1 A complete YACC file

The previous section only showed the grammar part of the YACC file, but there is more. This is the header that we omitted:

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

void yyerror(const char *str)
{
        fprintf(stderr,"error: %s\n",str);
}

int yywrap()
{
        return 1;
}

main()
{
        yyparse();
}

%}

%token NUMBER TOKHEAT STATE TOKTARGET TOKTEMPERATURE
```

The yyerror() function is called by YACC if it finds an error. We simply output the message passed, but there are smarter things to do. See the 'Further reading' section at the end.

The function yywrap() can be used to continue reading from another file. It is called at EOF and you can than open another file, and return 0. Or you can return 1, indicating that this is truly the end. For more about this, see the 'How do Lex and YACC work internally' chapter.

Then there is the main() function, that does nothing but set everything in motion.

The last line simply defines the tokens we will be using. These are output using y.tab.h if YACC is invoked with the '-d' option.

4.1.2 Compiling & running the thermostat controller

```
lex example4.l
yacc -d example4.y
cc lex.yy.c y.tab.c -o example4
```

A few things have changed. We now also invoke YACC to compile our grammar, which creates y.tab.c and y.tab.h. We then call Lex as usual. When compiling, we remove the -ll flag: we now have our own main() function and don't need the one provided by libl.

```
NOTE: if you get an error about your compiler not being able to find 'yylval', add this
to example4.l, just beneath #include <y.tab.h>:

extern YYSTYPE yylval;
```

```
This is explained in the 'How Lex and YACC work internally' section.
```

A sample session:

```
$ ./example4
heat on
        Heat turned on or off
heat off
        Heat turned on or off
target temperature 10
        Temperature set
target humidity 20
error: parse error
$
```

This is not quite what we set out to achieve, but in the interest of keeping the learning curve manageable, not all cool stuff can be presented at once.

4.2 Expanding the thermostat to handle parameters

As we've seen, we now parse the thermostat commands correctly, and even flag mistakes properly. But as you might have guessed by the weasely wording, the program has no idea of what it should do, it does not get passed any of the values you enter.

Let's start by adding the ability to read the new target temperature. In order to do so, we need to learn the NUMBER match in the Lexer to convert itself into an integer value, which can then be read in YACC.

Whenever Lex matches a target, it puts the text of the match in the character string 'yytext'. YACC in turn expects to find a value in the variable 'yylval'. In Example 5, we see the obvious solution:

```
%{
#include <stdio.h>
#include "y.tab.h"
%}
%%
[0-9]+                  yylval=atoi(yytext); return NUMBER;
heat                    return TOKHEAT;
on|off                  yylval=!strcmp(yytext,"on"); return STATE;
target                  return TOKTARGET;
temperature             return TOKTEMPERATURE;
\n                      /* ignore end of line */;
[ \t]+                  /* ignore whitespace */;
%%
```

As you can see, we run atoi() on yytext, and put the result in yylval, where YACC can see it. We do much the same for the STATE match, where we compare it to 'on', and set yylval to 1 if it is equal. Please note that having a separate 'on' and 'off' match in Lex would produce faster code, but I wanted to show a more complicated rule and action for a change.

Now we need to learn YACC how to deal with this. What is called 'yylval' in Lex has a different name in YACC. Let's examine the rule setting the new temperature target:

```
target_set:
        TOKTARGET TOKTEMPERATURE NUMBER
```

```
        {
                printf("\tTemperature set to %d\n",$3);
        }
        ;
```

To access the value of the third part of the rule (ie, NUMBER), we need to use $3. Whenever yylex() returns, the contents of yylval are attached to the terminal, the value of which can be accessed with the $-construct.

To expound on this further, let's observe the new 'heat_switch' rule:

```
heat_switch:
        TOKHEAT STATE
        {
                if($2)
                        printf("\tHeat turned on\n");
                else
                        printf("\tHeat turned off\n");
        }
        ;
```

If you now run example5, it properly outputs what you entered.

4.3 Parsing a configuration file

Let's repeat part of the configuration file we mentioned earlier:

```
zone "." {
        type hint;
        file "/etc/bind/db.root";
};
```

Remember that we already wrote a Lexer for this file. Now all we need to do is write the YACC grammar, and modify the Lexer so it returns values in a format YACC can understand.

In the lexer from Example 6 we see:

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

%%

zone                    return ZONETOK;
file                    return FILETOK;
[a-zA-Z][a-zA-Z0-9]*    yylval=strdup(yytext); return WORD;
[a-zA-Z0-9\/.-]+        yylval=strdup(yytext); return FILENAME;
\"                      return QUOTE;
\{                      return OBRACE;
\}                      return EBRACE;
;                       return SEMICOLON;
\n                      /* ignore EOL */;
[ \t]+                  /* ignore whitespace */;
%%
```

If you look carefully, you can see that yylval has changed! We no longer expect it to be an integer, but in fact assume that it is a char *. In the interest of keeping things simple, we invoke strdup and waste a lot of memory. Please note that this may not be a problem in many areas where you only need to parse a file once, and then exit.

We want to store character strings because we are now mostly dealing with names: file names and zone names. In a later chapter we will explain how to deal with multiple types of data.

In order to tell YACC about the new type of yylval, we add this line to the header of our YACC grammar:

```
#define YYSTYPE char *
```

The grammar itself is again more complicated. We chop it in parts to make it easier to digest.

```
commands:
        |
        commands command SEMICOLON
        ;


command:
        zone_set
        ;

zone_set:
        ZONETOK quotedname zonecontent
        {
                printf("Complete zone for '%s' found\n",$2);
        }
        ;
```

This is the intro, including the aforementioned recursive 'root'. Please note that we specify that commands are terminated (and separated) by ;'s. We define one kind of command, the 'zone_set'. It consists of the ZONE token (the word 'zone'), followed by a quoted name and the 'zonecontent'. This zonecontent starts out simple enough:

```
zonecontent:
        OBRACE zonestatements EBRACE
```

It needs to start with an OBRACE, a {. Then follow the zonestatements, followed by an EBRACE, }.

```
quotedname:
        QUOTE FILENAME QUOTE
        {
                $$=$2;
        }
```

This section defines what a 'quotedname' is: a FILENAME between QUOTEs. Then it says something special: the value of a quotedname token is the value of the FILENAME. This means that the quotedname has as its value the filename without quotes.

This is what the magic '$$=$2;' command does. It says: my value is the value of my second part. When the quotedname is now referenced in other rules, and you access its value with the $-construct, you see the value that we set here with $$=$2.

```
NOTE: this grammar chokes on filenames without either a '.'  or a '/' in them.

zonestatements:
        |
        zonestatements zonestatement SEMICOLON
        ;

zonestatement:
        statements
        |
        FILETOK quotedname
        {
                printf("A zonefile name '%s' was encountered\n", $2);
        }
        ;
```

This is a generic statement that catches all kinds of statements within the 'zone' block. We again see the recursiveness.

```
block:
        OBRACE zonestatements EBRACE SEMICOLON
        ;

statements:
        | statements statement
        ;

statement: WORD | block | quotedname
```

This defines a block, and 'statements' which may be found within.

When executed, the output is like this:

```
$ ./example6
zone "." {
        type hint;
        file "/etc/bind/db.root";
        type hint;
};
A zonefile name '/etc/bind/db.root' was encountered
Complete zone for '.' found
```

5 Making a Parser in C++

Although Lex and YACC predate C++, it is possible to generate a C++ parser. While Flex includes an option to generate a C++ lexer, we won't be using that, as YACC doesn't know how to deal with it directly.

My preferred way to make a C++ parser is to have Lex generate a plain C file, and to let YACC generate C++ code. When you then link your application, you may run into some problems because the C++ code by default won't be able to find C functions, unless you've told it that those functions are extern "C".

To do so, make a C header in YACC like this:

```
extern "C"
{
        int yyparse(void);
        int yylex(void);
        int yywrap()
        {
                return 1;
        }

}
```

If you want to declare or change yydebug, you must now do it like this:

```
extern int yydebug;

main()
{
        yydebug=1;
        yyparse();
}
```

This is because C++'s One Definition Rule, which disallows multiple definitions of yydebug.

You may also find that you need to repeat the #define of YYSTYPE in your Lex file, because of C++'s stricter type checking.

To compile, do something like this:

```
lex bindconfig2.l
yacc --verbose --debug -d bindconfig2.y -o bindconfig2.cc
cc -c lex.yy.c -o lex.yy.o
c++ lex.yy.o bindconfig2.cc -o bindconfig2
```

Because of the -o statement, y.tab.h is now called bindconfig2.cc.h, so take that into account.

To summarize: don't bother to compile your Lexer in C++, keep it in C. Make your Parser in C++ and explain your compiler that some functions are C functions with extern "C" statements.

6 How do Lex and YACC work internally

In the YACC file, you write your own main() function, which calls yyparse() at one point. The function yyparse() is created for you by YACC, and ends up in y.tab.c.

yyparse() reads a stream of token/value pairs from yylex(), which needs to be supplied. You can code this function yourself, or have Lex do it for you. In our examples, we've chosen to leave this task to Lex.

The yylex() as written by Lex reads characters from a FILE * file pointer called yyin. If you do not set yyin, it defaults to standard input. It outputs to yyout, which if unset defaults to stdout. You can also modify yyin in the yywrap() function which is called at the end of a file. It allows you to open another file, and continue parsing.

If this is the case, have it return 0. If you want to end parsing at this file, let it return 1.

Each call to yylex() returns an integer value which represents a token type. This tells YACC what kind of token it has read. The token may optionally have a value, which should be placed in the variable yylval.

By default yylval is of type int, but you can override that from the YACC file by re#defining YYSTYPE.

The Lexer needs to be able to access yylval. In order to do so, it must be declared in the scope of the lexer as an extern variable. The original YACC neglects to do this for you, so you should add the following to your lexter, just beneath #include <y.tab.h>:

```
extern YYSTYPE yylval;
```

Bison, which most people are using these days, does this for you automatically.

6.1 Token values

As mentioned before, yylex() needs to return what kind of token it encountered, and put its value in yylval. When these tokens are defined with the %token command, they are assigned numerical id's, starting from 256.

Because of that fact, it is possible to have all ascii characters as a token. Let's say you are writing a calculator, up till now we would have written the lexer like this:

```
[0-9]+          yylval=atoi(yytext); return NUMBER;
[ \n]+          /* eat whitespace */;
-               return MINUS;
\*              return MULT;
\+              return PLUS;
...
```

Our YACC grammer would then contain:

```
        exp:    NUMBER
                |
                exp PLUS exp
                |
                exp MINUS exp
                |
                exp MULT exp
```

This is needlessly complicated. By using characters as shorthands for numerical token id's, we can rewrite our lexer like this:

```
[0-9]+          yylval=atoi(yytext); return NUMBER;
[ \n]+          /* eat whitespace */;
.               return (int) yytext[0];
```

This last dot matches all single otherwise unmatched characters.

Our YACC grammer would then be:

```
        exp:    NUMBER
                |
                exp '+' exp
                |
                exp '-' exp
                |
                exp '*' exp
```

This is lots shorter and also more obvious. You do not need to declare these ascii tokens with %token in the header, they work out of the box.

One other very good thing about this construct is that Lex will now match everything we throw at it - avoiding the default behaviour of echoing unmatched input to standard output. If a user of this calculator uses a ^, for example, it will now generate a parsing error, instead of being echoed to standard output.

6.2 Recursion: 'right is wrong'

Recursion is a vital aspect of YACC. Without it, you can't specify that a file consists of a sequence of independent commands or statements. Out of its own accord, YACC is only interested in the first rule, or the one you designate as the starting rule, with the '%start' symbol.

Recursion in YACC comes in two flavours: right and left. Left recursion, which is the one you should use most of the time, looks like this:

```
commands: /* empty */
        |
        commands command
```

This says: a command is either empty, or it consists of more commands, followed by a command. They way YACC works means that it can now easily chop off individual command groups (from the front) and reduce them.

Compare this to right recursion, which confusingly enough looks better to many eyes:

```
commands: /* empty */
        |
        command commands
```

But this is expensive. If used as the %start rule, it requires YACC to keep all commands in your file on the stack, which may take a lot of memory. So by all means, use left recursion when parsing long statements, like entire files. Sometimes it is hard to avoid right recursion but if your statements are not too long, you do not need to go out of your way to use left recursion.

If you have something terminating (and therefore separating) your commands, right recursion looks very natural, but is still expensive:

```
commands: /* empty */
        |
        command SEMICOLON commands
```

The right way to code this is using left recursion (I didn't invent this either):

```
commands: /* empty */
        |
        commands command SEMICOLON
```

Earlier versions of this HOWTO mistakenly used right recursion. Markus Triska kindly informed us of this.

6.3 Advanced yylval: %union

Currently, we need to define *the* type of yylval. This however is not always appropriate. There will be times when we need to be able to handle multiple data types. Returning to our hypothetical thermostat, perhaps we want to be able to choose a heater to control, like this:

```
heater mainbuiling
        Selected 'mainbuilding' heater
target temperature 23
        'mainbuilding' heater target temperature now 23
```

What this calls for is for yylval to be a union, which can hold both strings and integers - but not simultaneously.

Remember that we told YACC previously what type yylval was supposed to by by defining YYSTYPE. We could conceivably define YYSTYPE to be a union this way, by YACC has an easier method for doing this: the %union statement.

Based on Example 4, we now write the Example 7 YACC grammar. First the intro:

```
%token TOKHEATER TOKHEAT TOKTARGET TOKTEMPERATURE

%union
{
        int number;
        char *string;
}

%token <number> STATE
%token <number> NUMBER
%token <string> WORD
```

We define our union, which contains only a number and a string. Then using an extended %token syntax, we explain to YACC which part of the union each token should access.

In this case, we let the STATE token use an integer, as before. Same goes for the NUMBER token, which we use for reading temperatures.

New however is the WORD token, which is declared to need a string.

The Lexer file changes a bit too:

```
%{
#include <stdio.h>
#include <string.h>
#include "y.tab.h"
%}
%%
[0-9]+                  yylval.number=atoi(yytext); return NUMBER;
heater                  return TOKHEATER;
heat                    return TOKHEAT;
on|off                  yylval.number=!strcmp(yytext,"on"); return STATE;
target                  return TOKTARGET;
temperature             return TOKTEMPERATURE;
[a-z0-9]+               yylval.string=strdup(yytext);return WORD;
\n                      /* ignore end of line */;
[ \t]+                  /* ignore whitespace */;
%%
```

As you can see, we don't access the yylval directly anymore, we add a suffix indicating which part we want to access. We don't need to do that in the YACC grammar however, as YACC performs the magic for us:

```
heater_select:
```

```
        TOKHEATER WORD
        {
                printf("\tSelected heater '%s'\n",$2);
                heater=$2;
        }
        ;
```

Because of the %token declaration above, YACC automatically picks the 'string' member from our union. Note also that we store a copy of $2, which is later used to tell the user which heater he is sending commands to:

```
target_set:
        TOKTARGET TOKTEMPERATURE NUMBER
        {
                printf("\tHeater '%s' temperature set to %d\n",heater,$3);
        }
        ;
```

For more details, read example7.y.

7 Debugging

Especially when learning, it is important to have debugging facilities. Luckily, YACC can give a lot of feedback. This feedback comes at the cost of some overhead, so you need to supply some switches to enable it.

When compiling your grammar, add –debug and –verbose to the YACC commandline. In your grammar C heading, add the following:

int yydebug=1;

This will generate the file 'y.output' which explains the state machine that was created.

When you now run the generated binary, it will output a *lot* of what is happening. This includes what state the state machine currently has, and what tokens are being read.

Peter Jinks wrote a page on *debugging* `http://www.cs.man.ac.uk/~pjj/cs2121/debug.html` which contains some common errors and how to solve them.

7.1 The state machine

Internally, your YACC parser runs a so called 'state machine'. As the name implies, this is a machine that can be in several states. Then there are rules which govern transitions from one state to another. Everything starts with the so called 'root' rule I mentioned earlier.

To quote from the output from the Example 7 y.output:

```
state 0

    ZONETOK     , and go to state 1

    $default    reduce using rule 1 (commands)

    commands    go to state 29
```

```
command     go to state 2
zone_set    go to state 3
```

By default, this state reduces using the 'commands' rule. This is the aforementioned recursive rule that defines 'commands' to be built up from individual command statements, followed by a semicolon, followed by possibly more commands.

This state reduces until it hits something it understands, in this case, a ZONETOK, ie, the word 'zone'. It then goes to state 1, which deals further with a zone command:

```
state 1

    zone_set  ->  ZONETOK . quotedname zonecontent   (rule 4)

    QUOTE       , and go to state 4

    quotedname  go to state 5
```

The first line has a '.' in it to indicate where we are: we've just seen a ZONETOK and are now looking for a 'quotedname'. Apparently, a quotedname starts with a QUOTE, which sends us to state 4.

To follow this further, compile Example 7 with the flags mentioned in the Debugging section.

7.2 Conflicts: 'shift/reduce', 'reduce/reduce'

Whenever YACC warns you about conflicts, you may be in for trouble. Solving these conflicts appears to be somewhat of an art form that may teach you a lot about your language. More than you possibly would have wanted to know.

The problems revolve around how to interpret a sequence of tokens. Let's suppose we define a language that needs to accept both these commands:

```
delete heater all
delete heater number1
```

To do this, we define this grammar:

```
delete_heaters:
        TOKDELETE TOKHEATER mode
        {
                deleteheaters($3);
        }

mode:   WORD

delete_a_heater:
        TOKDELETE TOKHEATER WORD
        {
                delete($3);
        }
```

You may already be smelling trouble. The state machine starts by reading the word 'delete', and then needs to decide where to go based on the next token. This next token can either be a mode, specifying how to delete the heaters, or the name of a heater to delete.

The problem however is that for both commands, the next token is going to be a WORD. YACC has therefore no idea what to do. This leads to a 'reduce/reduce' warning, and a further warning that the 'delete_a_heater' node is never going to be reached.

In this case the conflict is resolved easily (ie, by renaming the first command to 'delete heaters all', or by making 'all' a separate token), but sometimes it is harder. The y.output file generated when you pass yacc the –verbose flag can be of tremendous help.

8 Further reading

GNU YACC (Bison) comes with a very nice info-file (.info) which documents the YACC syntax very well. It mentions Lex only once, but otherwise it's very good. You can read .info files with Emacs or with the very nice tool 'pinfo'. It is also available on the GNU site:

BISON Manual `http://www.gnu.org/manual/bison/` .

Flex comes with a good manpage which is very useful if you already have a rough understanding of what Flex does. The

Flex Manual `http://www.gnu.org/manual/flex/` is also available online.

After this introduction to Lex and YACC, you may find that you need more information. I haven't read any of these books yet, but they sound good:

Bison-The Yacc-Compatible Parser Generator

By Charles Donnelly and Richard Stallman. An *Amazon* `http://www.amazon.com/exec/obidos/ASIN/0595100325/qid=989165194/sr=1-2/ref=sc_b_3/002-7737249-1404015`

user found it useful.

Lex & Yacc

By John R. Levine, Tony Mason and Doug Brown. Considered to be the standard work on this subject, although a bit dated. Reviews over at *Amazon* `http://www.amazon.com/exec/obidos/ASIN/1565920007/ref=sim_books/002-7737249-1404015` .

Compilers : Principles, Techniques, and Tools

By Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. The 'Dragon Book'. From 1985 and they just keep printing it. Considered the standard work on constructing compilers.

Amazon `http://www.amazon.com/exec/obidos/ASIN/0201100886/ref=sim_books/002-7737249-1404015`

Thomas Niemann wrote a document discussing how to write compilers and calculators with Lex & YACC. You can find it *here* `http://epaperpress.com/lexandyacc/index.html` .

The moderated usenet newsgroup comp.compilers can also be very useful but please keep in mind that the people there are not a dedicated parser helpdesk! Before posting, read their interesting *page* `http://compilers.iecc.com/` and especially the *FAQ* `http://compilers.iecc.com/faq.txt` .

Lex - A Lexical Analyzer Generator by M. E. Lesk and E. Schmidt is one of the original reference papers. It can be found

here `http://www.cs.utexas.edu/users/novak/lexpaper.htm` .

Yacc: Yet Another Compiler-Compiler by Stephen C. Johnson is one of the original reference papers for YACC. It can be found

here `http://www.cs.utexas.edu/users/novak/yaccpaper.htm` . It contains useful hints on style.

9 Acknowledgements & Thanks

- Pete Jinks <pjj%cs.man.ac.uk>
- Chris Lattner <sabre%nondot.org>
- John W. Millaway <johnmillaway%yahoo.com>
- Martin Neitzel <neitzel%gaertner.de>
- Sumit Pandaya <sumit%elitecore.com>
- Esmond Pitt <esmond.pitt%bigpond.com>
- Eric S. Raymond
- Bob Schmertz <schmertz%wam.umd.edu>
- Adam Sulmicki <adam%cfar.umd.edu>
- Markus Triska <triska%gmx.at>
- Erik Verbruggen <erik%road-warrior.cs.kun.nl>
- Gary V. Vaughan <gary%gnu.org> (read his awesome *Autobook* `http://sources.redhat.com/autobook`)
- *Ivo van der Wijk* `http://vanderwijk.info` (

  Amaze Internet `http://www.amaze.nl`)