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.

Next Previous Contents