CSL202: PROGRAMMING PARADIGMS & PRAGMATICS

Semester II, 2014 – 2015

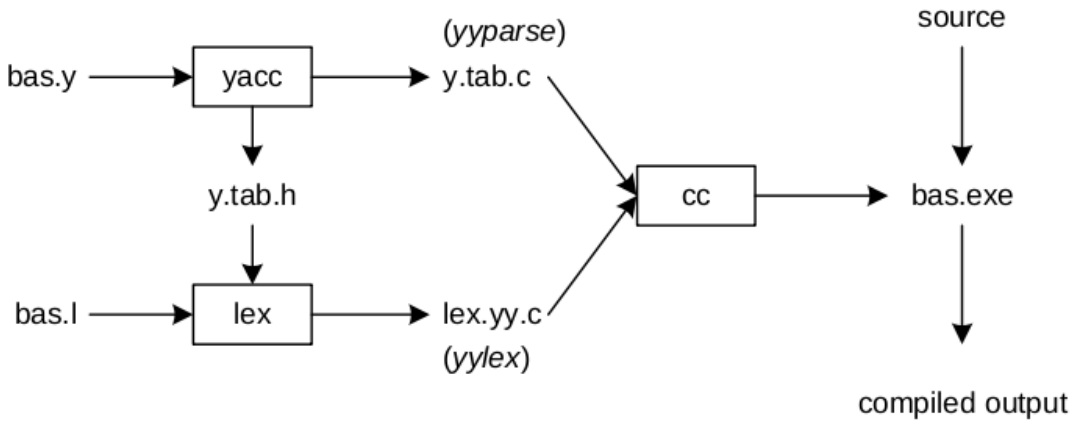
Lab 9: Lex & Yacc for Compiler Writing

• Introduction

Some of the most time consuming and tedious parts of writing a compiler involve the lexical scanning and syntax analysis. Luckily there is freely available software to assist in these functions. While they will not do everything for you, they will enable faster implementation of the basic functions. Lex and Yacc are the most commonly used packages with Lex managing the token recognition and Yacc handling the syntax.

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. They work well together, but conceivably can be used individually as well and each serves a different purpose.

Both operate in a similar manner in which instructions for token recognition or grammar are written in a special file format. The text files are then read by Lex and/or Yacc to produce C code. This resulting source code is compiled to make the final application. In practice the lexical instruction file has a ".l" suffix and the grammar file has a ".y" suffix. This process is shown here:



In the above example first, we need to specify all pattern matching rules for lex (`bas.l`) and grammar rules for yacc (`bas.y`). Commands to create our compiler, `bas.exe`, are listed below:

```
yacc -d bas.y                    # create y.tab.h, y.tab.c
lex bas.l                        # create lex.yy.c
cc lex.yy.c y.tab.c -o bas.exe   # compile/link
```

Yacc reads the grammar descriptions in `bas.y` and generates a syntax analyzer (parser) that includes function `yyparse`, in file `y.tab.c`. Included in file `bas.y` are token declarations. The `-d` option causes Yacc to generate definitions for tokens and place them in file `y.tab.h`.

Lex reads the pattern descriptions in `bas.l`, includes file `y.tab.h`, and generates a lexical analyzer that includes function `yylex`, in file `lex.yy.c`. Finally, the lexer and parser are compiled and linked together to create executable bas.exe. From main we call `yyparse` to run the compiler. Function `yyparse` automatically calls `yylex` to obtain each token.

- **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. The file format for a Lex file consists of (4) basic sections:

```
%{
/* C includes */
}%
/* Definitions */
%%
/* Rules */
%%
/* user subroutines */
```

- The first section, in between the **%{** and **%}** pair, is an area for C code that will be place directly/verbatim at the beginning of the generated source code (ouput program). Typically is will be used for things like `#include`, `#defines`, and variable declarations.

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

- The next section is for definitions of token types to be recognized. These are not mandatory, but in general makes the next section easier to read and shorter. Definitions have the form: `Name  Definition`

  Examples:
```
DIGIT    [0-9]
ID       [a-z][a-z0-9]*
```

- The third or Rules section is separated from the above sections using **%%.** This section defines the pattern for each token that is to be recognized, and can also include C code to be called when that token is identified. Rules have the form: `Pattern  Action`

  Examples:
```
{ID}                printf( "An identifier: %s\n", yytext );
[a-z][a-z0-9]*      { return ID; }
```

  - If action has more than one statement, enclose it within { }

- The last section, again separated from earlier sections using **%%,** is for more C code (generally subroutines) that will be appended to the end of the generated C code. This would typically include a main function if Lex is to be used by itself.

- A very simple example (`Example1.l`):

```
%{
  #include <stdio.h>
%}
%%
stop     printf("Stop command received\n");
start    printf("Start command received\n");
%%
```

- We need to include `stdio.h` because `printf` is used later on, which is defined in `stdio.h`
- Rules section includes two definitions, one for ‘stop’ key and other for ‘start’ key
- Whenever 'stop' or ‘start’ key is encountered in the input, the rest of the line (a `printf` () call) is executed

- 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 the program 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.

- Another simple example (`Counting.l`)

  - This program simply counts in the number of lines and characters in the input.

```
%{
  int num_lines = 0, num_chars = 0;
%}
%%
\n      ++num_lines; ++num_chars;
.       ++num_chars;
%%
main()   {
  yylex();
  printf("# of lines = %d, # of chars = %d\n", num_lines, num_chars);
}
```

- Lex also provides a bunch of predefined functions and variables as follows:

Name	Function
`int yylex(void)`	call to invoke lexer, returns token
`char *yytext`	pointer to matched string
`yyleng`	length of matched string
`yylval`	value associated with token
`int yywrap(void)`	wrapup, return 1 if done, 0 if not done
`FILE *yyout`	output file
`FILE *yyin`	input file
`INITIAL`	initial start condition
`BEGIN`	condition switch start condition
`ECHO`	write matched string

- Play around with them and try using them!

- **Yacc (Yet Another Compiler Compiler)**

  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.

  Grammars for Yacc are described using a Backus Naur Form (BNF) discussed in class! Note: Yacc does not cope with ambiguity and will complain about shift/reduce or reduce/reduce conflicts. It is your responsibility to provide unambiguous grammar to Yacc.

  The file format for a Yacc file is similar to Lex and is as follows:

```
%{
/* C includes */
}%
/* Other Declarations */
%%
/* Rules */
%%
/* user subroutines */
```

- The first section, in between the **%{** and **%}** pair, is an area for C code that will be place directly/verbatim at the beginning of the generated source code (output program). Typically it will be used for things like `#include`, `#defines`, and variable declarations.

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

- The next section, Declarations, consists of token declarations. Although not mandatory when used on its own, makes communication easier when Yacc is used along with Lex! Token definitions have the form: `%token TokenName`

  Example: `%token INTEGER`

  - All terminal symbols should be declared through `%token`
  - Yacc produces `y.tab.h` with `%token` definitions for communication with Lex

- The third or Rules section is separated from the above sections using **%%**. The rules section is made up of one or more grammar rules. A grammar rule has the form: `A : BODY ;`
  - Each rule contains LHS and RHS, separated by a colon and end by a semicolon. White spaces or tabs are allowed
  - Where '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.

  Examples:
```
A       :       B   C   D
        |       G
        ;

Statement   : name EUQALSIGN expression
            | expression ;
```

- 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:

```
Example:   A        :        '('   B   ')'
                               {        hello( 1, "abc" );   }

           XXX      :        YYY   ZZZ
                               {        printf("a message\n");
                               flag = 25;    }
```

- 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 following form frequently need not have an explicit action.

```
A        :        B     ;
```

- 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.

- The last section is for more C code (generally subroutines) that will be appended to the end of the generated c code. This would typically include a main function.

Example: 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 (`Heat.l`) will be:

```
%{
#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 make sure to return the names of tokens instead of simply printing stuff. 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 (`Heat.y`) 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 we 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'.

- 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.
- 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.
- 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.
- 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.
- Compiling & running the thermostat controller

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

- 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;
```

- A sample session:

```
$ ./Heat
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
$
```

- **PRACTICE:**
  - This was a very basic crash course on Lex and Yacc. Refer to the Tutorial posted on Moodle and other documents online for more information on any aspects that have not been dealt with in detail here.
  - LEX
    - Write a program to recognize and count the number of identifiers in a given input file.
    - Write a program to count the number of 'scanf' and 'printf' statements in a C program. Replace them with 'readf' and 'writef' statements respectively.
    - *Hint: Open the file for reading the main function and then call yylex from within main*
  - YACC
    - Write a program to recognize the grammar ($a^n b$, n>=10)
    - Write a Program to recognize nested IF control statements and display the levels of nesting.
  - COMPILER
    - Write a Lexer and a Parser for a simple Calculator that takes numbers and calculates results of operators (+, -, * and /) with proper precedence and associativity.

- **No Submission!**
  - As this lab was supposed to give some time to practice working with Lex and Yacc, you do not need to submit anything.
  - Next lab session, you will have to write a complete compile from scratch for a custom made language!