CS202: PROGRAMMING PARADIGMS & PRAGMATICS

Semester II, 2019 – 2020

Lab 6: Lex (Flex) & Yacc (Bison)

Aim: Use BNF and Regular expressions for language design via Flex and Bison.

- **Let's get started!**
  - Create a directory structure to hold your work for this course and all the subsequent labs:
    - Suggestion: `CS202/Lab6`

- **Introduction**
  - Today we will be using two open source tools, *Flex* and *Bison*, which together allow their user to create a program that "tokenizes" an input stream. The names for these tools comes from their predecessors *lex* and *yacc*. Lex is short for *lexical analyzer*; flex is the *Fast lexical analyzer*; yacc stands for *Yet Another Compiler Compiler*; and since yacc sounds like *yak*, bison is sort of a pun on yacc.
  - The flex program works by reading a scanner-language specification from an input file. By convention, this file uses a `.lex` (or just `.l`) file suffix. As output, flex then produces a C source-code program that, when compiled, will serve as a scanner or *tokenizer* for the specified language.
  - The bison program works by reading a parser-language specification from an input file. By convention, this file uses a `.y` file suffix. As output, bison produces a C source-code program that, when compiled, will serve as a parser for the language.
  - We will examine each of these in turn.

- **Flex Example 1**
  - Let's start with a very simple example.

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

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

  - This file is divided into sections, where '`%%`' indicates the dividing lines.
  - The first section, consisting of the directives between the `%{` and the `%}` is typically used to include any libraries needed by the other sections. Whatever we place here will also be incorporated into the program flex outputs. In this example, we need `stdio.h` because we use `printf()` in the second section, which is defined in `stdio.h`. The 'f' of '`printf`' is for formatted, and this command prints formatted text to `stdout`.
  - In the second section, we tell flex that whenever the word 'start' is encountered in the input, the scanner should recognize that and print the string `Start command received`; and whenever the scanner encounters the word 'stop' in the input, it should recognize it and print the string `Stop command received.`
  - We terminate the second section with '`%%`' again. (A third section follows, but ours is empty in this simple example.)

- Save this code in a file called example1.lex. You can then compile this first example as follows:

```
flex example1.lex
gcc lex.yy.c -o example1 -lfl
```

- The first command (flex) generates a C file named `lex.yy.c`. This file contains the source code for a scanner for the language we specified in `example1.lex`
- The second command uses `gcc` to compile that C source code and generate an executable called `example1`. That executable will then act as a scanner / lexical analyzer for the language specified in `example1.lex`.
- You can then run this example in a terminal by entering:

```
./example1
```

- (assuming you are in your Lab6 directory). Since the program reads from standard input (i.e., the keyboard), it waits for you to enter something.
- If you enter 'start' it will output 'Start command received', and if you enter 'stop' it will output 'Stop command received'. Whenever you type something that is not matched by any of the defined symbols (ie, 'stop' and 'start') the program simply echoes that symbol.
- To end your session, you can either send `EOF (^D)`, or (less cleanly) you can kill the program with `^C`.
- Take a minute to open the C file generated by flex (`lex.yy.c`). While there is a great deal of code generated for you by flex, take a closer look at the main loop and check out the interesting parts!

- **Flex Example 2**
  - In class we have considered EBNF, which allows us to write BNF expressions more compactly. Flex also supports some shortcuts by letting us use regular expressions to specify symbols.
  - For example, suppose we wanted to be able to categorize non-negative integers like *0*, *42*, or *99999* as NUMBERs; and categorize words like *hello*, *a*, *I*, and *R2D2* as WORDs. Here is a specification file for this "language". :

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

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

  - This specification file describes two kinds of symbols (tokens): WORDs and NUMBERS. Let's examine the first one:

```
[0123456789]+     printf("NUMBER\n");
```

  - This tells flex to print the string *NUMBER* whenever it sees a sequence of one or more characters from the group 0123456789. The brackets indicate the group of characters to be matched, the + symbol indicates flex should match one or more characters from the group. We could also have written it more concisely as:

```
[0-9]+
```

  - The second one is somewhat more involved:

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

- This tells flex to print the string *WORD* for symbols where:
  1. The first part matches 1 and only 1 character from the group 'a' through 'z' or 'A' through 'Z'. In other words, a letter.
  2. The second part matches zero or more characters that can be letters or digits. Where the '+' in the first line signifies one or more characters, the '*' in this second line signifes zero or more characters. We must use it because a WORD might consist of only one character, which we've already matched. Since the second part may have zero matches, we use a '*'.
- With the second part, we've mimicked the behavior of many programming languages in which an identifier *must* start with a letter, but can contain digits afterwards. In other words, R2D2 is a valid identifier, but 2R2D is not.
- Save this specification in a new file called example2.lex and then use it to generate a new scanner, as you did with example1. Feed this new tokenizer a variety of inputs and see what happens. Have one of your test inputs include an underscore. What happens? Why? Extend this language specification so that an underscore is a valid first (and subsequent) letter, as was the case for the 'identifier' grammar we used in class. Make sure to test your extension of the grammar by giving input values that include an underscore.

- **Flex Example 3**
  - Now suppose we want to be able to tokenize files that are more complicated, such as the one that follows:

```
class StudentInfo {
public:

   StudentInfo() { myGPA = 0.0; myHours = 0; }

   StudentInfo(double gpa, int hours) {
      myGPA = gpa;
      myHours = hours;
   }

private:
   double myGPA;
   int    myHours;
};
```

  - If we look carefully, we can see a number of categories of symbols (tokens) in this file:
    - KEYWORDs, like `class`, `public`, `private`, `int` and `double`
    - IDENTIFIERs, like `StudentInfo`, `myGPA`, `myHours`, `gpa`, and `hours`
    - INTEGER literals, like `0`
    - DOUBLE literals, like `0.0`
    - PUNCTUATION, like `{`, `}`, parentheses, colons, commas, and semicolons.
    - OPERATORs, like `=`
  - A corresponding specification file for flex is as follows:

```
%{
#include <stdio.h>     /* printf() */
#include <stdlib.h>    /* atof()   */
%}

DIGIT    [0-9]
ID       [a-zA-Z][a-zA-Z0-9]*

%%

{DIGIT}+ {
     printf( "INTEGER: %s (%d)\n", yytext,
          atoi( yytext ) );
}

{DIGIT}+"."{DIGIT}* {
     printf( "DOUBLE: %s (%g)\n", yytext,
            atof( yytext ) );
}

class|int|double|public|private {
     printf( "KEYWORD: %s\n", yytext );
     }

{ID} printf( "IDENTIFIER: %s\n", yytext );

":"|"{"|"}"|"("|")"|";"|"," printf( "PUNCTUATION: %s\n", yytext );

"="  printf( "OPERATOR: %s\n", yytext );

[ \t\n]+ /* eat up whitespace */

.    printf( "Unrecognized character: %s\n", yytext);

%%
```

- Note that with curley-braces, you can use several statements or lines to specify the action flex should take when it recognizes a symbol.

- Save this specification in a file named example3.lex, and use it to create a scanner named `example3`. Record the output of running it on the example *StudentInfo* class. Note: If you store the *StudentInfo* class in a file named `StudentInfo.h`, you can tell your scanner to read from that file instead of standard input by entering:

```
./example3  < StudentInfo.h
```

- Writing a scanner / tokenizer from scratch is a non-trivial project; flex lets us create a scanner / tokenizer with a lot less work.

- **Bison: Example 4**

  - Flex lets us turn an input file into a stream of tokens. Bison lets us parse a stream of tokens to see if it follows the grammar rules of a language. Now that we've seen how Flex can produce a stream of tokens, we will see how to combine that with Bison to check that that stream of tokens for syntax errors.

- Suppose 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!
set temperature 22
        New temperature set!
```

- The tokens we need to recognize are: heat, on/off (STATE), set, temperature, and NUMBERs.
- We can specify this language for Flex as follows:

```
%{
#include <stdio.h>
#include "example4.tab.h"
%}
%%
[0-9]+                  return NUMBER;
heat                    return TOKHEAT;
on|off                  return STATE;
set                     return TOKSET;
temperature             return TOKTEMPERATURE;
\n                      /* ignore end of line */;
[ \t]+                  /* ignore whitespace */;
%%
```

- Save this specification file as example4.lex. There are two important changes to note:
  - We include the file `example4.tab.h`.
  - We no longer print strings; we return names of tokens.

- The latter change is because where before we were outputting strings to the screen, now we want to feed the output from flex to bison as its input token-stream. The file `example4.tab.h` has definitions for these tokens.
- But where does `example4.tab.h` come from? It is generated by Bison from the grammar File we are about to create. As our language is pretty simple, so is the grammar:

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

command:
        heat_switch
        | temperature_set
        ;

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

```
temperature_set:
        TOKSET TOKTEMPERATURE NUMBER
        {
                printf("\tTemperature set\n");
        }
        ;
```

- Study this grammar: What differentiates a terminal from a non-terminal?
- The Bison file provides the grammar, as well as some additional information, called the *header*, which is also required:

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

/* declarations */
int yylex (void);
void yyerror (char const *);
int yyparse(void);

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

int yywrap() {
        return 1;
}

int main() {
        yyparse();
}

%}

%token NUMBER TOKHEAT STATE TOKSET TOKTEMPERATURE
%%
```

- The `yyerror()` function is called by bison if it finds an error. For simplicity, we just output the message bison passes. Don't worry about the `yywrap()` function, as it is useful for reading multiple files, which we won't be doing in this class. The `main()` function is what sets everything in motion, by calling `yyparse()`. The last line defines the tokens we will be using.
- Put the header followed by the grammar information together to form a single file, named `example4.y`
- Now can use flex and bison (and the C compiler) to create a combined scanner-parser executable that will let us run our temperature controller:

```
flex example4.lex
bison -d example4.y
gcc lex.yy.c example4.tab.c -o example4
```

- Run your temperature controller, giving it some valid commands that use valid syntax. Do you see anything strange?

- **Bison: Example 5**

  - We can now parse the thermostat commands correctly. But as you might have guessed by the imprecise output, the program has no idea of what it should do, since Flex does not pass it the actual values you enter.
  - Let's add the ability to read the new target temperature. To do this, we need to tell the NUMBER match in our scanner to convert itself into an integer value that Bison can read.
  - Whenever flex recognizes a symbol, it puts the actual text of the match in a character string variable named `yytext`. Bison in turn expects to find a value in the variable `yylval`. The following language specification shows one way to use this:

```
%{
#include <stdio.h>
#include <stdlib.h>              /* atoi() */
#include <string.h>              /* strcmp() */
#include "example5.tab.h"
%}
%%
[0-9]+                      yylval = atoi(yytext); return NUMBER;
heat                        return TOKHEAT;
on|off                      yylval = !strcmp(yytext,"on"); return STATE;
set                         return TOKSET;
temperature                 return TOKTEMPERATURE;
\n                          /* ignore end of line */;
[ \t]+                      /* ignore whitespace */;
%%
```

  - In this specification, we use `atoi()` (ascii to int) on `yytext` to convert its string to a number, then we assign the resulting number to `yylval`, so that bison can see it. When we return the `NUMBER` token, flex will attach the value of `yylval` to that token for bison to use.
  - We do something similar for the `STATE` symbol: we use the C `strcmp()` function to compare the string in `yytext` to the string "`on`", and set `yylval` to 1 if they are equal; 0 if they are not.
  - Now we need to tell Bison how to deal with this. What is called `yylval` in Flex has a different name in Bison. Here is a replacement rule for setting the temperature to a new value:

```
temperature_set:
        TOKSET TOKTEMPERATURE NUMBER
        {
                printf("\tTemperature set to %d\n",$3);
        }
        ;
```

  - In the rule above, the production (right side) has 3 parts: `TOKSET`, `TOKTEMPERATURE`, and `NUMBER`. To access the value of the third part of the rule (ie, `NUMBER`), we must use `$3`. Whenever `yylex()` returns, the contents of `yylval` are attached to the terminal symbol, and the value can be accessed using the $-mechanism.

- To further illustrate this mechanism, let's modify the 'heat_switch' rule to indicate whether the heat is to be turns on or off:

  ```
  heat_switch:
          TOKHEAT STATE
          {
                  if ($2) {  /* if STATE == 1 */
                          printf("\tHeat turned on\n");
                  } else {
                          printf("\tHeat turned off\n");
                  }
          }
          ;
  ```

- Make copies of `example4.lex` and `example4.y` named `example5.lex` and `example5.y` respectively, and incorporate these changes. Then verify that it properly responds to your input.

- **Exercise: Example 6**

  - Make copies of `example5.lex` and `example5.y` named `example6.lex` and `example6.y`, respectively. In these copies, extend the temperature controller to let the user update the humidity correctly. The user should be able to enter a command like

    ```
    set humidity to 45
    ```

    and receive an *informative* message in return.

  - Use Flex, Bison, and gcc to create an `example6` scanner-parser.

  - Show that your `example6` scanner-parser correctly recognizes and accepts all of the temperature-control commands described above.

- **Exercise: Example 7**

  - Writer a Scanner-Parser that counts the number of characters, words and the number of lines of the input.
  - Example of session:

    ```
    ./exemple7
    How long is this input?
    Let's find out!
    ^D

    # of lines = 2, # of chars = 40
    ```

- **Submitting your work:**

  - All source files as one tar-gzipped archive.
    - When unzipped, it should create a directory with your ID. Example: **2008CS1001** (NO OTHER FORMAT IS ACCEPTABLE!!! Case sensitive!!!)
    - ***Negative marks if the TA has to manually change this to run his/her scripts!!***
  - Source files should include the following: (Case-Sensitive file names!!)
    - `.lex` and `.y` files for all examples from 1 - 7

- ***Negative marks for any problems/errors in running your programs***
- Submit/Upload it to Moodle: