CS202: PROGRAMMING PARADIGMS & PRAGMATICS

Semester II, 2020 – 2021

Lab 8: Introduction to Lex/Yacc (Flex/Bision)

- **Aim:** The goal of this lab is to introduce the use of Lex (Flex) and Yacc (Bison) for building scanners and parsers

- **Let's get started!**

  a. Create a directory structure to hold your work for this course and all the subsequent labs:
     - Suggestion: `CS202/Lab8`

  b. A bit more detailed description of Lex and Yacc is provided in `LexAndYaccTutorial.pdf` on Moodle

- **Introduction**

  When properly used, Lex & Yacc 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. Although these programs shine when used together, they each serve a different purpose. The next two will explain what each part does.

- **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 (`example1.l`):

Example 1:

```
%{
#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 this example 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.

This will generate the file `Ex1`. 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.

- **Regular Expression in matches**

The above 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 (`example2.l`).

<u>Example 2:</u>

```
%{
#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. You should be quite comfortable with Regular Expressions by now and realize that `NUMBER` matches:

`[0123456789]+` : A sequence of one or more characters from the group 0123456789. Same as: `[0-9]+`

And, the `WORD` matches:
`[a-zA-Z][a-zA-Z0-9]*` : A letter(small or capital) followed by zero or more characters which are either a letter or a digit. Basically this rule constitutes legal variable names in many languages.

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.

- **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:

- `WORD`s, like 'zone' and 'type'
- `FILENAME`s, like '/etc/bind/db.root'
- `QUOTE`s, like those surrounding the filename
- `OBRACE`s, `{`
- `EBRACE`s, `}`
- `SEMICOLON`s, `;`

The corresponding Lex file is **Example 3** `(example3.l):`

```
%{
#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.

- **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 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 (`example4.l`) 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 (`example4.y`) we are about to create. As our language is very basic, so is the BNF 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` for the thermostat, 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.

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`.

- **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 (this part goes before the grammar shown above in `example4.y` file):

```
        %{
        #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 we can 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 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;
```

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
$
```

- **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`. Below (`example5.l`), 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 teach `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 expand 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 compile and run example5, it properly outputs what you entered.

- **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. We modify the lexer in example3.l to the following (example6.l):

```
%{
#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 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 `quotedname` 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 `QUOTE`s. 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
```

- **Exercises**

  1. Write a program using LEX to count the number of characters, words, spaces and lines in a given input file. (`Count.l`)
  2. Write a program using LEX to count the numbers of comment lines in a given C program. Also eliminate them and copy the resulting program into separate file (`Comments.l`)
  3. Write a YACC program to recognize strings 'aaab', 'abbb','ab' and 'a' using the grammar (a^nb, n>= 10) (`ab.y`)
  4. Write a program using LEX to recognize a valid arithmetic expression and to recognize the identifiers and operators present. Print them separately (`calc.l`)
  5. Write a program using YACC to recognize and evaluate valid arithmetic expression that uses operators +, -, * and /. (`calc.y`)

- **Submitting your work:**

  - All source files and class 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!!!)
    - Should include:
      - ✓ `count.l` [3 points]
      - ✓ `Comments.l` [3 points]
      - ✓ `ab.l` [3 points]
      - ✓ `calc.l` [3 points]
      - ✓ `calc.y` [3 points]
      - ✓ README file

  - ***Negative marks for any problems/errors in running your programs***

  - *If any aspects of tasks are confusing, make an assumption and state it clearly in your* ***README*** *file!*

  - **README** file should also have instructions on how to use/run your program!

  - Submit/Upload it to Google Classroom