Lab 2
Lexing and Parsing with Flex and Bison - 2 labs

Objective

- Understand the software architecture of flex/bison.
- Be able to write simple grammars in bison.
- Be able to correct grammar issues in bison.

First download and uncompress the archive available here.

2.1 Overview of a flex/bison parser

In this section we give a quick outline on how to develop a parser using `flex` and `bison` and how to compile it. The next section will provide you with little examples.

2.1.1 Structure of the files

In most parsers, the analysis is done in two steps:

- First lexical analysis, that transforms the input file into a stream of tokens, which are an abstraction over the language constructs (a token per keyword, a token for identifiers, ...). This phase is very simple and only uses regular expressions to do very shallow transformation on the input.
  This analysis is described by abstract rules in the **lexer** and resides in a `.lex` file. The rules are then transformed to C code by `flex`.
- The stream of tokens is then parsed by an automaton corresponding to a context-free grammar. The grammar is spelled out in a `.y` file and the `bison` program transforms it into an automaton written in C.

Lexical analysis – the lexer The typical structure of a `.lex` file is as follows:

```
%{
  /* Prelude C code that will be prepended to the code of the lexer
  generated by ‘flex‘ */
%}

%%
<regular expression>
  { <action run on every part of the input matching the regular expression> }

/* Another form: <regular expression> ;

   It means ignore the parts of the input matching the regexp (for
instance spaces) and continue */
%%

/* Postlude C code that will be appended to the code of the lexer
(like a main function for instance) */
```

The lexer then tries every regular expression in the order specified by the file on the input, and execute the code corresponding to the first expression that matches a prefix of the input. It then continues with the rest of the input.

Actions can also return tokens to the parser instead of consuming the all the input.

Syntactic analysis – the parser The typical structure of a `.y` file is as follows:

```
%{
  /* Prelude C code that will be prepended to the code of the parser
  generated by ‘bison‘ */
%}

/* Definition of the different tokens */
%token TOK_1 TOK_2

/* Name of the non-terminal that is the starting point of the grammar. */
%start prog
%%

/* Actual grammar, composed of rules of the form */
symbol:
  TOK_1 other_symbol ;
| TOK_2 symbol ;

/* Or with code */
other_symbol:
 TOK_3 { puts ("other_symbol"); }
%%

/* Postlude C code that will be appended to the code of the parser
(like a main function for instance) */
```

2.1.2 Compilation

We describe here the basic sequence of commands to compile your flex and bison files (should be done in this order):

- Compile a parser into C: `bison -d -o parser.c parser.y -defines=parser.h`
  The input is the file `parser.y` (written by you). The main output is `parser.c`, which is a syntactical analyzer.
  Note that the file `parser.h` is also generated; it is used in the following command (`flex`) to know the tokens that are used by the lexer and the parser.
  After compilation, we can call a function `yyparse`, that does the syntactical analysis.
- Compile a lexer into C: `flex -noyywrap -o lexer.c lexer.lex`
  The input is the file `lexer.lex` (written by you), and the output is the file `lexer.c`.
  After compilation, we can call a function `yylex`, that does the lexical analysis.
- Compile all the C files: `cc -o prog parser.c lexer.c ...`
  The files generated above, `parser.c` and `lexer.c`, are fed to the compiler, together with other source C files (this is the "..." above | it can be, for instance, a `main.c` file).
  Finally, this generated the file called "`prog`", which can be executed by invoking `./prog`.

In some examples of this TP, we only use the lexer generator. In other examples, there is no `main.c` file, because all the necessary C/C++ code is in the lexer and parser files.

You will find in several directories a file called `Makefile`, whose role is to make the managment of the above mentioned compilation steps automatic. But it is useful to have an idea of what's going on when invoking `make` (thus exploiting the `Makefile` file).

2.2 Simple examples

EXERCISE #1 ▶ **Demo files**
Work your way through the five examples in the directory `demo_files`:

- `exemple.lex`: A very simple lexical analysis for simple arithmetic expressions of the form `x+3`. To compile, run:
  `make exemple`
  This generates a binary (`exemple`) that you can run using `./exemple`. To signal the program you have finished entering the input, use **Control-D**.
  Examples of run: [`^D` means that I pressed Control-D, what I typed is in boldface].
  ```
  % ./exemple
  1+1
  ^D
  This is a valid expression.
  % ./exemple
  ()
  Invalid expression!
  %
  ```
  Questions:
  - Read and understand the code.
  - Allow for parentheses to appear in the expressions.
  - Make the lexer terminate on a newline instead of the EOF (Control-D) signal.
  - What is an example of a recognized expression that looks odd? To fix this problem we need a syntactic analyzer (cf. exemple5)
- `exemple2.lex`: A little variation to show how to use the text that is matched by the rule (`yytext`). To compile, type `make exemple2` and to run it, type `./exemple2`. Example of session:
  ```
  % ./exemple2
  foo
  Your identifier is: foo
  ()
  not recognized!
  ```
  Question: Add a rule to match numbers and print their successor in the action (*Hint:* Use the `atoi` function to convert the string into an integer).
- `exemple3.lex`: We wish now to count the number of characters and the number of lines of the input. To compile, run `make exemple3`. Example of session:
  ```
  % ./exemple3
  How long is this input?
  Let's find out!
  ^D
  # of lines = 2, # of chars = 40
  ```
  Question: When encountering a letter, print the previous letter encoutered if any.
- `exemple4.y` and `exemple4.lex`: First syntactic analysis. The goal is to make a program that recognizes inputs that are well-parenthesized.
  The lexer extracts the tokens (opening and closing parentheses) from the input (understand what it does with the other characters), whereas the parser checks that they are matching. To compile, run `make exemple4`. Example of session:
  ```
   % ./exemple4
  (1+3)*(2+5)
  ^D
  Well-parenthesized.
   % ./exemple4
  (1+3*(2+5)
  ^D
  *** Parsing error: syntax error
  Not well-parenthesized.
  ```

  Question: Add the support for square brackets (`[` and `]`). (`Hint:` add two new tokens, and one more rule.)
- `exemple5.y` and `exemple5.lex`: Parsing (and evaluating) simple arithmetic expressions. This parser recognizes the grammar for arithmetic expressions with + and * (without parenthesis). To compile, run

`make exemple5`. Example of session:
`% ./exemple5`
`2+3*5;`
`Result: 17`
(Do not forget the semicolon at the end for expressions! | can you see where, in the source files, we make the presence of the semicolon mandatory?)
Question: What happens if we remove the `%left` directives in `exemple5.y`?

If these files read like a novel, go on with the other exercises. Otherwise, make sure that you understand what is going on. You can ask the Teaching Assistant, or another student, for advice.

2.3 Warmup, mini-exercises

For these exercises, first do them on paper, then confirm your solution on the machine – see directory `ex_3`, type `make` to compile the grammars.

EXERCISE #2 ▶ **Flex**
What will be the behavior of this flex file after compilation?

mystere1.lex

```
%%
[a-z] printf ( "%c", yytext [ 0 ] - 'a' + 'A' );
\n printf ( "%s", yytext );
. printf ( "%s", yytext );
%%

int main ( int argc, char** argv )
{
   yylex ();
}
```

EXERCISE #3 ▶ **Flex/Bison**
We give you the following code:

mystere2.lex

```
%{
#include "parser.h"
#include <stdio.h>
%}
%%
"("  {return PO;}
")"  {return PF;}
"["  {return CO;}
"]"  {return CF;}
\n
.
%%
```

mystere2.y

```
%{
#include <stdio.h>
  extern int yyerror (const char *s);
  extern int yylex();
%}
```

```
%token PO PF CO CF
%start F
%%

F : L        {     printf("F->L\n"); }
L : LI       {    printf("L->LI\n") ; }
|            {  printf("L->eps\n"); }
LI : LI I    {  printf("LI->LI␣I\n"); }
| I          {   printf("LI->I\n"); }
I : PO L PF  {    printf("I->(L)\n"); }
| CO L CF    {     printf("I->[L]\n"); }

%%

int yyerror(const char *s) {
  printf("Parsing␣error:␣%s\n", s);
  return 1;
}
int main(int argc, char** argv) {
  return yyparse();
}
```

- What is the behavior of the `flex` file? (guess, then check).
- Give the output of flex+bison on the following input: `(3+a)*b[8-c]`, write the derivation and draw the parse tree.
- Test.

2.4 Solving conflicts with bison

EXERCISE #4 ▶ **Shift/Shift**

In the directory `ex_4` the file `ifthenelse.y` contains a (partial) specification of the following grammar:

```
Z -> S
S -> if E then S else S
S -> if E then S
S - > inst
E -> id
```

- Compile using `make` (write `make`, not `make ifthenelse`). C code of the analyzer is produced using the following command:
  `bison --defines=parser.h -o parser.c parser.y`. *Bison* also generates `parser.h` which contains the declarations.
- Complete the grammar, `inst` being the constant string "inst", and `id` any other string, and test your grammar.
  Remark that there is now a warning about a shift/reduce conflict.
- **Open `Makefile`** and add the options `--report=lookahead --report-file=bison_report` in the bison command line. This will tell bison to generate a report describing the conflict.
- **Go to the state where the shift/reduce conflict arises**. What is the default choice of the analyzer? What is the ambiguity in the grammar causing this conflict?
- When there is a shift/reduce conflict, it means that the parser hesitates between shifting (reading) one more specific token or reducing using a specific rule.
  What is the token and the rule in conflict here?
- Bison has a mechanism allowing one to solve this conflict just by annotating the grammar: we need to tell which from the token or the rule has the priority. The priority of a rule is given by the priority of its rightmost-token. The priority of a token is declared together with its associativity (left, right, none). Associativity is used to tell bison what to do when a rule has the same priority than a token (i.e., the situation of a shift/reduce conflict):

- If the priority is `left`, then reduce the rule
- If the priority is `right`, shift
- If it is `nonassoc`, report a syntax error.

The declaration is made by the directives `%left`, `%right` and `%nonassoc` followed with a list of tokens that will share the same priority.
Finally the directives:
```
%left tok1
%left tok2
```
declares `tok2` to have higher priority (or precedence) than `tok1`.
Give a set of priority directives so that "if then else" is "right-associative" in the following sense: `if x then if y then inst else inst` should be parsed `if x then (if y then inst else inst)`.

- (*optional*) Give a set of priorities for it to be left-associative.

EXERCISE #5 ▶ **Reduce/reduce**
Reduce/reduce conflicts are real bugs in the grammar that need correction. The idea is that a reduce/reduce conflict means that a word has two different syntactic interpretations. For example: the string `int*x` could be a declaration of a variable `x` of type `int*`, or the computation of the product between the variable `int` and `x`.

In the `ex_5` directory of the archive, look at the grammar. Where does the reduce/reduce conflict come from? How does bison solve it? (Look at the `bison_report` file).

2.5 Grammar Attributes (action)

Until now, our analyzers are passive oracles. We would like to execute code during the analysis and produce the intermediate representation. This is what *attribute grammars* are made for. We associate to each production a piece of code that will be executed each time the production will be reduced. This piece of code is called *semantic action* and computes the attributes of non-terminals. Let us consider the following grammar (the end of an expression is a semicolon):

$$
\begin{aligned}
Z &\rightarrow E; \\
E &\rightarrow E + T \\
E &\rightarrow T \\
T &\rightarrow T * F \\
T &\rightarrow F \\
F &\rightarrow id \\
F &\rightarrow (E)
\end{aligned}
$$

EXERCISE #6 ▶ **Evaluating arithmetic expressions**
In the `ex_6` directory of the archive, you will find this grammar. Then :

- Attribute the grammar to evaluate arithmetic expressions.
- Execute your grammar against `1+(2*3);`.

EXERCISE #7 ▶ **Constructing the AST for an arithmetic language**
In the `ex_7` directory of the archive, you will find the grammar for an extension of this arithmetic language with constants.

- Complete the functions in `ast.cpp` for the creation and evaluation of the syntax tree.
- Complete the action for the grammar in `expr.y`.
- Compile (`make`) and test the toplevel using `./expr`. Example of valid input: `pi+3`, `3*5+e/2`
- *(this question is optional)* Add in the grammar the support for unary subtraction (to support for instance `-pi`). Hint: try using the `%prec` modifier, together with priority `nonassoc`.

EXERCISE #8 ▶ **From Scratch**
Consider prefixed expressions like `* + * 3 4 5 7` and assigments of such expressions to variables: `a=* + * 3 4 5  7`. Identifiers are allowed in expressions.

- Give a grammar that recognizes lists of such assigments. Encode it in flex/bison. Test.

- Use grammar rules to construct infix assigments during parsing: the former assigment will we transformed into the *string* `a=(3 * 4 + 5)*7`. Be careful to avoid useless parentheses.
- Modify the attribution to verify that the use of each identifier is done after his first definition.

2.6 Bonus

EXERCISE #9 ► **Grammar + attribution for pseudo-XML files**
Consider the following grammar for XML files:

```
L ->E L
|
E -> A L B
|    ident
A -> < ident >
B -> </ ident >
```

- Write lex + bison files to recognize this grammar.
- Add rules to check during the parsing that opening and closing tags are refering to the same identifiers.

2.7 C++ warmup - Mandatory (2nd lab)

Carefully read Appendix **??**. Make the exercises.