Homework #2: CMPT-379

Anoop Sarkar – anoop@cs.sfu.ca

- Only submit answers for questions marked with †.
- Checkout your group svn repository:
  `svn co https://punch.cs.sfu.ca/svn/CMPT379-1127-g-YOUR-GROUP-NAME`
- Copy the files for this homework (and then add and commit the files using svn):
  `scp -r fraser.sfu.ca:/home/anoop/cmpt379/hw2 CMPT379-1127-g-YOUR-GROUP-NAME/.`
- Put your solution programs in the `hw2/answer` directory. Use the `makefile` provided. There are strict filename requirements. Read the file `readme.txt` in the `hw2` directory for details.
- Create a file called `HANDLE` in your `hw2` directory which contains your group handle (no spaces).
- The `hw2/testcases` directory contains useful test cases; you will need to consult `readme.txt` for the mapping between the homework questions and test cases and instructions on how to run the auto check program.
- Always use `exit(0);` as the last line of your `main` function. Use `exit(1);` to indicate failure.
- The `hw2` directory contains files `usingcpp*.*` which is a working example of using C++ with flex and bison. Run `make usingcpp` to compile the binary and `make test` to run it.
- Reading for this homework includes Chp 4 of the Dragon book. If needed, refer to:

  `http://tldp.org/HOWTO/Lex-YACC-HOWTO.html`

(1) The file `simple-expr.y` contains yacc code for a very simple (and incomplete) expression interpreter. The `%{ ... %}` section contains arbitrary C/C++ code and the `%token` definitions is a list of tokens provided by the lexical analyzer. `bison` can be used to convert this parser definition into a parser implementation by using the following command:

```
bison -osimple-expr.tab.c -d simple-expr.y
```

The `-d` option produces a header file called `simple-expr.tab.h` to convey information about the tokens to the lexical analyzer. Examine the contents of this file. The lexical analyzer is defined as lex code in the file `simple-expr.lex`. The lex file can be compiled to a C program using `flex`:

```
flex -osimple-expr.lex.c simple-expr.lex
```

The final binary is created by compiling the output from flex and bison with a C/C++ compiler as follows. Note that these steps are automated in the provided `makefile` so running `make simple-expr` will run bison and then flex and then use `gcc` to compile the generated C code.

```
gcc -o ./simple-expr simple-expr.tab.c simple-expr.lex.c -ly -lfl
echo "a=2+3+5" | ./simple-expr
```

Convert the above yacc and lex code so that it can handle multiple expressions, exactly one per line. You will need a recursive context-free rule in the yacc definition in order to handle multiple lines of input. Try different ways of writing this recursive rule. Note that we can assign a value to a variable, e.g. `b=5+10-5` but we cannot yet use `b` in a following expression, e.g. `a=b+10` (which fails with a syntax error).

(2) The yacc and lex code in Q. (1) does not yet handle assignments to variables. In order to implement this, we need two different kinds of values to be returned from the lexical analyzer: one for numbers, and another for variable names. The code in `simple-varexpr.lex` shows you how to do that. For numbers it returns `yylval.rvalue` and for variable names it returns `yylval.lvalue`. The two types of return value, `rvalue` and `lvalue` are defined in the yacc program `simple-varexpr.y` using the `%union` declaration. The `%union` declaration can include complex datatypes. The yacc code defines a type not just for the tokens, but also for nonterminals, which is specified in the `%type` definition. This allows yacc to check that the type of the non-terminal expression is `rvalue`, an integer type.

Extend this code to properly handle assignments of values to variables. The variable on the left hand side of an equation will be an *ℓ-value* and the variable used on the right hand side of an equation will be a *r-value*.

(3) Extend your expression interpreter to include constants of type `double`, and variables that can hold either integer or double types. Finally, add the functions: `exp`, `sqrt`, `log` so that you can interpret the following types of input:

```
a = 2.0
b = exp(a)
b
```

To avoid issues with precedence of operators use the grammar provided in `expr-grammar.y` as the basis for your expression interpreter.

(4) † Write a **`Decaf`** program that implements the quicksort algorithm to sort a list. Create an array variable `list` with 100 elements. Then initialize it using the following **`Decaf`** loop:

```
  void initList() {
        int i;
        for (i = 0; i < 100; i = i + 1) {
            list[i] = (i * 2382983) % 100;
        }
    }
```

Sort this list using quicksort and then print the sorted list by iteratively calling the `print_int` library function in the format shown in the test cases directory.

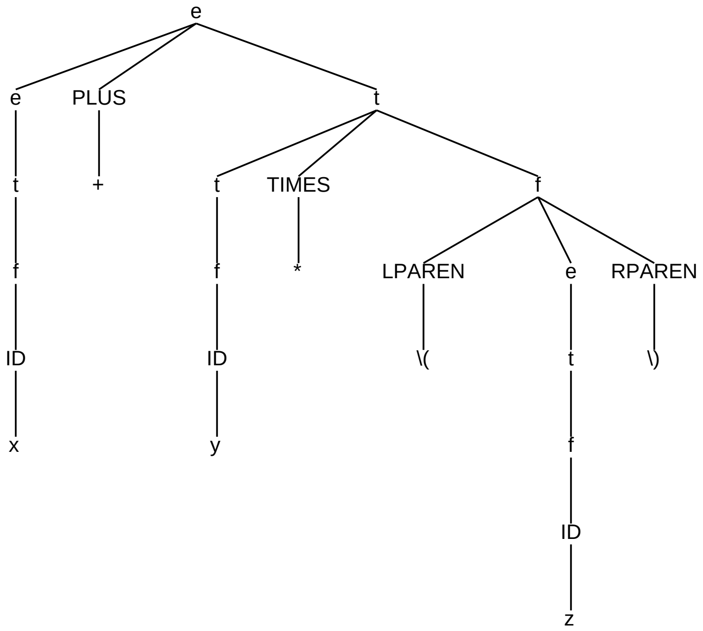
(5) † Write down a yacc parser for the following context-free grammar:

$$\begin{array}{lcl} \text{e} & \rightarrow & \text{e PLUS t} \\ \text{e} & \rightarrow & \text{t} \\ \text{t} & \rightarrow & \text{t TIMES f} \\ \text{t} & \rightarrow & \text{f} \\ \text{f} & \rightarrow & \text{LPAREN e RPAREN} \\ \text{f} & \rightarrow & \text{ID} \end{array}$$

The tokens PLUS, TIMES, LPAREN, RPAREN are defined to be `+`, `*`, `(`, `)` respectively. And the token ID is defined to be an identifier as in the **`Decaf`** specification. These tokens should be defined using a lexical analyzer produced using lex.

For the input string `x + y * ( z )` the output produced by the yacc parser should be the parse tree for the input string produced in the format shown in the left column below. Note the backslash preceding each instance of a literal parenthesis to avoid confusion with the parentheses used to denote the tree structure. Note that you may need to augment the grammar to produce the right output. Do not bother to indent your tree, just print out your parse tree in a single line of output text (`perl indentrees.pl` will auto-indent a tree in parenthesis format). A graphical view, using the Tcl/Tk program `viewtree.tk`, is shown in the right column below:

```
(e (e (t (f (ID x))))
   (PLUS +)
   (t (t (f (ID y)))
      (TIMES *)
      (f (LPAREN \()
         (e (t (f (ID z))))
         (RPAREN \)))))
```



(6) Write down a context-free grammar for **Decaf** that is a LR(1) grammar accepted by yacc without any shift/reduce conflicts based on the reference grammar in the **Decaf** language definition (make sure that the non-terminal and terminal symbols used in the CFG correspond as much as possible to the symbols used in the reference grammar).

Use the following text format: For the CFG: ⟨start⟩ → A ⟨start⟩ B | ϵ the text format you should use is:

```
start: A start B
     | /* empty string */
     ;
```

You can optionally have C-style comments as above. Follow the convention that the non-terminals in this text format are written in the same format as identifiers in **Decaf** but are in lowercase (e.g. `start`, and for hyphenated non-terminals like *method-name* replace the hyphen with an underscore, e.g. `method_name`) and write the terminal symbols in the same format as identifiers but entirely in uppercase (e.g. `A`).

Writing a context-free grammar so that we can ensure it is a LR(1) grammar is the most challenging part of writing down the syntax specification of a programming language. Consider the following fragment of a **Decaf** program:

```
class foo {
  int bar
```

Note that we could continue the above fragment with a field declaration, *or* a method declaration. This issue will not be a problem for a LR parser if the CFG for **Decaf** can be written as an LR(1) grammar. Compare the two CFGs provided in `decafSmallGrammar.y` and `decafSmallLRGrammar.y`. The both represent the same fragment of **Decaf** syntax corresponding to the **Decaf** examples above. Experiment with these two fragments to understand that one is not an LR(1) grammar and results in shift/reduce errors for `bison` while the second grammar is an LR(1) grammar and `bison` does not report any shift/reduce errors.

Create a LR(1) grammar in yacc for the entire **`Decaf`** syntax specification.

(7) † **`Decaf` Parse Trees**: Write a yacc program that prints out the abstract syntax tree for any input **`Decaf`** program.

- Use the lexical analyzer you have already built based on the **`Decaf`** specification.
- The first step after a working lexer for **`Decaf`** is to make sure that the yacc program uses a context-free grammar for **`Decaf`** that is a complete LR grammar for **`Decaf`** so that you can parse any input **`Decaf`** program without any shift/reduce conflicts.
- The abstract syntax tree specification for **`Decaf`** is provided as part of the **`Decaf`** language specification. Also the abstract tree syntax is specified in ASDL format in the file `Decaf.asdl`.
- The abstract syntax tree must be printed out by first creating an abstract syntax tree data structure. There **must** be exactly one print statement in your program for printing out the string representation of the entire abstract syntax tree from this data structure.
- If the input is not a valid **`Decaf`** program you must emit a syntax error message.

The **input specification** is the the **`Decaf`** reference grammar, and the **output specification** is the `Decaf.asdl` file.

(8) Write a JSON reader (`http://www.json.org`) using lex and yacc. The program should accept a JSON file from standard input (stdin) and represent it as a nested hash table data structure in C/C++ that is analogous to the Javascript data structure and then print out the entire JSON tree *after* reading in the entire input by printing out the internal data structure.