Homework #2: CMPT-379

Distributed on Sep 30; due on Oct 18

Anoop Sarkar – anoop@cs.sfu.ca

Reading for this homework includes Chp 4 of the Dragon book. If needed, refer to:

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

Only submit answers for questions marked with †. You must use a `makefile` such that `make` compiles all your programs, and `make test` runs each program, and supplies the necessary input files. You have been provided a file `makefile.answer` which you should use as your makefile. The stdout when you run `make test` must match `makefile.answer.out`, and stderr must match `makefile.answer.err`.

(1) The following program is yacc code for a very simple (and incomplete) expression interpreter.

```
%{
#include <stdio.h>
%}
%token NAME NUMBER
%%
statement: NAME '=' expression { printf("%c = %d\n", $1, $3); }
     | expression  { printf("%d\n", $1); }
     ;

expression: expression '+' NUMBER { $$ = $1 + $3; }
     | expression '-' NUMBER { $$ = $1 - $3; }
     | NUMBER { $$ = $1; }
     ;
%%
```

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 can be defined in lex code as follows:

```
%{
#include "simple-expr.tab.h"
#include <stdlib.h>
extern int yylval;
%}
%%
[0-9]+   { yylval = atoi(yytext); return NUMBER; }
[a-z]    { yylval = yytext[0]; return NAME; }
[ \t\n]  ;
.        return yytext[0];
%%
```

The lex file can be compiled to a C program using `flex`:

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

The final interpreter binary is created by compiling the output from flex and bison with a C/C++ compiler:

```
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. For example, it should print out a value for each line in the following input:

```
a = 10
b = 5 + 10 - 5
2 + 5
2 + 3 + 5
```

You will need a recursive rule in order to handle multiple lines of input. Try different ways of writing this recursive rule.

(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 following fragment of lex code now returns two different types of values to yacc: for numbers it returns `yylval.rvalue` and for variable names it returns `yylval.lvalue`.

```
... /* removed code common with previous lex code */
%%
  /* convert NUMBER token value to integer */
[0-9]+    { yylval.rvalue = atoi(yytext); return NUMBER; }

  /* convert NAME token into index */
[a-z]     { yylval.lvalue = yytext[0] - 'a'; return NAME; }

 ... /* removed code common with previous lex code */
%%
```

The two types of return value, `rvalue` and `lvalue` are defined in the yacc code using the `%union` declaration, as shown in the fragment below:

```
%{
int symtbl[26];
%}

%union {
  int rvalue; /* value of evaluated expression */
  int lvalue; /* index into symtbl for variable name */
}

%token <rvalue> NUMBER
%token <lvalue> NAME

%type <rvalue> expression

%%
statement: NAME '=' expression { symtbl[$1] = $3; }
     | expression  { printf("%d\n", $1); }
     ;

expression: expression '+' NUMBER { $$ = $1 + $3; }
     | expression '-' NUMBER { $$ = $1 - $3; }
     | NUMBER { $$ = $1; }
     ;
%%
```

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 above. This allows yacc to check that the type of the non-terminal expression is `rvalue`, an integer type.

Use the above code fragments, and add the necessary lex and yacc code in order to handle the following input and output:

Input:	Output:

`a = 2 + 3`
`a`
`b = a + a - 2 + 3 + a`
`b`
	`5`
	`16`

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 input:

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

(3) † Write a **Decaf** program that implements the quicksort algorithm to sort a list. Your program should print the sorted list by iteratively calling the `print_int` library function. Submit the program as the file `quicksort.decaf`

(4) Write down a context-free grammar for the structure of **Decaf** programs 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). Submit a file called `decafGrammar.txt` which contains the CFG in the following text format: For the CFG:
$\langle \text{start} \rangle \rightarrow A \ \langle \text{start} \rangle \ B \mid \epsilon$ the text format you should use is:

```
start A start B
start
```

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`).

You should verify the correctness of your CFG either by examining it closely or you can verify aspects of the CFG by writing some simple code for checking whether non-terminals are used in the right-hand side of rules but not defined on the left-hand side anywhere else in the CFG, whether the terminal symbols are valid tokens, etc.

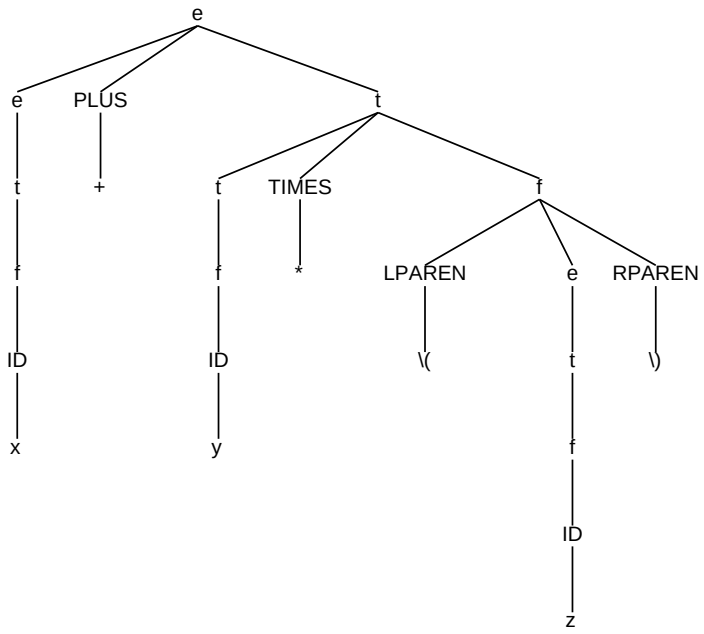
(5) † Write down a yacc parser for the following context-free grammar:

e	→	e PLUS t
e	→	t
t	→	t TIMES f
t	→	f
f	→	LPAREN e RPAREN
f	→	ID

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. A graphical view, using the Tcl/Tk program `viewtree`, 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) **Grammar Conversion**: 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 grammar. Convert the following CFG, which represents a small fragment of **`Decaf`** syntax, into an LR grammar. As a result, the LR parsing table for such a grammar will have no shift/reduce or reduce/reduce conflicts.

`program`	$\rightarrow$	`CLASS ID LCB field_decl_list method_decl_list RCB`
`field_decl_list`	$\rightarrow$	`field_decl field_decl_list`
`field_decl_list`	$\rightarrow$	ϵ
`method_decl_list`	$\rightarrow$	`method_decl method_decl_list`
`method_decl_list`	$\rightarrow$	ϵ
`field_decl`	$\rightarrow$	`type ID ASSIGN INTCONSTANT SEMICOLON`
`method_decl`	$\rightarrow$	`return_type ID LPAREN RPAREN`
`return_type`	$\rightarrow$	`type`
`return_type`	$\rightarrow$	`VOID`
`type`	$\rightarrow$	`INT`
`type`	$\rightarrow$	`BOOL`

You can test if your revised grammar is an LR grammar by writing down the grammar as yacc code with no actions and simply check if you get any shift/reduce conflicts. Note that in this skeleton yacc code, the token definitions do not need to be linked with a lexical analyzer in order to check for shift/reduce conflicts.

(7) † **`Decaf` Parse Trees**: Write a yacc program that prints out a parse tree for any input **`Decaf`** program. You will need to make sure that the context-free grammar you use is an LR grammar. The parse tree format should be the same as in Q. (5). Use the lexical analyzer you have already built based on the **`Decaf`** specification.