CMPT 379 - Summer 2016 - Final Exam

Fill in your name and student id on your Exam Booklet. Write "Final Exam" next to your name.
Provide answers in the Exam Booklet provided to you. Do not answer the questions on this paper.
When you have finished, return your Exam Booklet along with this question booklet.

(1) The following CFG describes regular expressions:

$$R \rightarrow R\,\text{‘}\texttt{|}\text{’}\,R \mid R\,R \mid R\,\text{‘}\texttt{*}\text{’} \mid \text{‘}\texttt{(}\text{’}\,R\,\text{‘}\texttt{)}\text{’} \mid a \mid b$$

a. (3pts) Provide all the leftmost derivations for the input string `a|b*b`. Show all the steps in each leftmost derivation.

b. (4pts) Convert this grammar into an unambiguous CFG that resolves ambiguity by assuming that Kleene closure, **‘*’** has the highest priority, followed by concatenation, *RR*, followed by alternation, **‘|’** .

Assume that each operation associates to the left, e.g. *RRR* should be treated as (*RR*)*R* and *R*|*R*|*R* should be treated as (*R*|*R*)|*R*.

To make grading easier, for any new non-terminals that you introduce to solve this question you must use numeric subscripts, e.g. $R_1, R_2, R_3, \ldots$.

c. (3pts) For the input string *abba* provide the lexemes that would be returned by a greedy longest match lexical analyzer assuming that the only token is defined by the (unambiguous) regular expression `a|b*b|a`.

d. (1pt) We want to add a new operator **?** which denotes a match of 0 or 1 repetitions to the regular expression syntax. For example `a?b` matches the string `ab` or `b`. This new operator should have the same precedence as Kleene closure * and should be left associative wrt multiple * and **?** operators. Add a rule to the grammar for the operator **?** that will keep it unambiguous.

e. (1pt) Provide the leftmost derivation for `a*?` using your augmented grammar.

f. (3pts) Provide a context free grammar that generates the same language as the following regular expression.

```
( abb*(c | d*) ) | ( (a | c)*b )
```

(2) Consider the augmented CFG *G* with *S′* as the start symbol:

$$\begin{array}{rcll} S' & \rightarrow & S & (1) \\ S & \rightarrow & A\,a\,A\,b & (2) \\ S & \rightarrow & B\,b\,B\,a & (3) \\ S & \rightarrow & \epsilon & (4) \\ A & \rightarrow & \epsilon & (5) \\ B & \rightarrow & \epsilon & (6) \end{array}$$

a. (8pts) Use the canonical LR(1) set-of-items construction and create an action/goto table for LR parsing for grammar *G*. Use the rule numbers that follow each rule in *G* above in your table.

   Write down the itemsets and table clearly and legibly.

b. (1pt) Is the CFG *G* is SLR(1)? Yes or no is sufficient.

c. (3pts) Provide the LL(1) parsing table for *G*.

d. (3pts) Trace the LL(1) parser (show the stack and input for each step) for input *ab*.

(3) **Code Generation**: Instead of an **if** statement what if we had an **if** expression? Remember that, in procedural programming languages, a statement ends with a semicolon and cannot be used inside expressions, while an expression can be used inside statements and other expressions. There are at least two ways to implement an **if** expression (simplified to use boolean constants). We will call these two ways alternative 1 and 2.

1.

Rule	Syntax-directed definition
Expr → **if (***B* **,** *Expr* **,** *Expr* **)**	\$3.true := “eval” \$5.code; \$3.false := “eval” \$7.code; \$0.code := \$3.code;
B → **true**	\$0.code = \$0.true; *// true is inherited*
B → **false**	\$0.code = \$0.false; *// false is inherited*

2.

Rule	Syntax-directed definition
Expr → **if (***B* **,** *Expr* **,** *Expr* **)**	\$3.true := true := newlabel(); \$3.false := false := newlabel(); \$0.code := \$3.code + label(true) + “return” \$5.code + label(false) + “return” \$7.code;
B → **true**	\$0.code = “goto” \$0.true; *// true is inherited*
B → **false**	\$0.code = “goto” \$0.false; *// false is inherited*

In the above definition, *true* and *false* are inherited attributes, inherited by the left hand side `B` non-terminal. *“eval”* produces an *r*-value from the code generated for its argument.

The operator + concatenates instructions and labels and creates a list of instructions.

The function *newlabel()* creates a new label each time it is called (returning `L1`, `L2`, . . .) and *label(L)* attaches label *L* to the next instruction, e.g. *label(L1)* would result in `L1:` generated in the output.

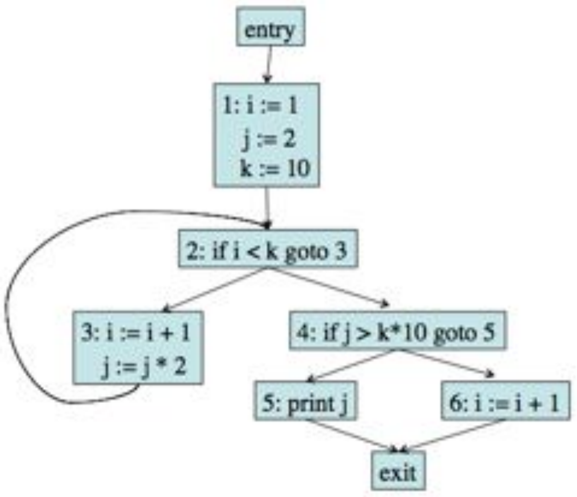
a. (6pts) For the input **if** expression: `if(true,0,1)` provide the value of \$0.code for the **if** expression for both alternatives.

b. (5pts) While making testcases to try out this new **if** expression, the professor discovers that following recursive function goes into an infinite loop for the first definition, but works correctly for the second definition. Provide a brief but precise (one sentence) answer about why this happens.

```
int factorial (int n)
{
  bool b = (n == 0);
  return (if (b, 1, n*factorial(n-1)));
}
```

c. (4pts) Fix the syntax directed definition in alternative 1 in order to avoid the infinite loop for the code in question 3b. Provide a new code generation definition only for the *Expr* rule in alternative 1.

(4) **Static Single Assignment Form**: Consider the flowgraph below:



a. (3pts) For each basic block *X* provide *D*(*X*) which is the set of basic blocks strictly dominated by *X* in the above flowgraph. Ignore the entry and exit blocks.

b. (4pts) Provide the dominance frontier *DF*(*X*) for each basic block *X* in the flowgraph.

c. (8pts) Using the dominance frontier *DF*(·) construct the flowgraph in minimal Static Single Assignment (SSA) form. A *minimal* SSA form has no redundant static variable definitions.