# THE UNIVERSITY OF CALGARY

## FACULTY OF SCIENCE

## FINAL EXAMINATION

## COMPUTER SCIENCE 411

April, 2008 Time: 2 hrs.

### Instructions

**The exam contains questions totalling 100 points. Answer all questions.**

This exam is closed book. You are expected to *explain* all the answers you provide.

Please note that all grammars have non-terminals in upper-case and terminals in lower-case. The productions associated with a non-terminal are separated by bars and ended with a period.

. . . . . . 2

25 marks

1. Consider the following grammar:

```
S -> id ass E semi
   | E.
E -> id
   | ( E D .
D -> B ) id
   | ) .
B -> , E B
   | .
```

   (a) Determine whether the following strings can be recognized by the grammar and, if so draw their parse trees

      i. `id ass ( ( id ) ) id`

      ii. `id ( id )`

      iii. `(( id , id )) id`

   (b) Calculate “vital statistics” of the grammar, that is the *first* and *follow* sets. Indicate which non-terminals are nullable and which are endable.

   (c) What must be satisfied by the grammar to ensure it is LL(1). Explain why this grammar is not LL(1).

   (d) Transform this grammar into an LL(1) grammar.

   (e) What does it mean for a grammar to be unambiguous? Is this grammar ambiguous? Explain.

. . . . . . 3

25 marks

2. Consider the following grammar:

```
A -> A ( B C A )
   |.
B -> B [ A D B ]
   |.
C -> x
   | .
D -> y
   | .
```

(a) Using all the terminals, give three examples of strings which are recognized by this grammar.

(b) Calculate the first sets, follow sets, nullables, and endables of this grammar.

(c) Draw a Venn diagram of the relationship between the grammar classes: LL(0), LL(1), LR(0), LR(1), SLR(1), and LALR(1).

(d) Calculate the LR(0) item automaton for this grammar.

(e) Is the grammar LR(0), SLR(1), or LALR(1).

(f) Give the LALR(1) parse table for this grammar.

25 marks

3. This question is concerned with the organization of stack-based run-time environment for the **M** language:

   (a) Describe the organization of an *activation record* for a function. In particular, describe how local arrays are organized on the stack.

   (b) Describe the *caller* and *callee* responsibilities when a function is called. Describe what has to be done on a *return* from a function call.

   (c) Explain how arrays are passed into functions and how they are accessed.

   (d) Explain the purpose of the *static link* (or *access link*) and how it differs from the *dynamic link* (or *control link*).

   (e) Explain why a static link is unnecessary for blocks.

   (f) Explain how, in a program, the "static distance" of an occurrence of an identifier from its place of declaration is calculated. Explain how one uses this information to access *non-local variables*. Explain why it is necessary to know this distance for an occurrence of a *function identifier*.

   (g) Describe how user defined datatypes are implemented. In particular describe what happens when a constructor is applied to its arguments and how a case statement is implemented.

. . . . . . 5

25 marks

4. Given the following abstract syntax tree for "let expressions" with sequential declarations (where earlier defined terms in a declaration sequence can be used in later definitions) draw the plumbing diagrams and develop pseudo code to determine the legallity and type of such a let expression.

   The definition of the abstract syntax tree for let expressions is:

```
data Expr = ADD Expr Expr
          | OR Expr Expr
          | LE Expr Expr
          | ID String
          | IV Int
          | BV Bool
          | LET [(String,Expr)] Expr
```

   A typical syntax tree (which does not type) is:

```
    ADD (ID x)
        (LET [(y,OR(ID a,ID b)),(z,ADD(ID c,ID b))]
             (OR(ID y,ID z)))
```

   which may be viewed as representing the code:

```
      x + (let y=a||b; z=c+b; in y||z end)
```

   The operations should be assumed to have the following types:

```
    OR::(BOOL,BOOL)->BOOL,
   ADD::(INT,INT)->INT,
    LE::(INT,INT)->BOOL
```

   Note that in a given `let` declaration list at most one declaration for any symbol is allowed. Furthermore, as the declarations are simultaneous the body of each declaration can only use variables which have already been defined in an outer scope.

   You may assume you have an implementation of a symbol table which allows look up and insertion:

```
look_up(st: ST, id: String ; t: Type)
insert(st: ST, id: String ; st': St)
```

. . . . . . 6