THE UNIVERSITY OF CALGARY

FACULTY OF SCIENCE

FINAL EXAMINATION

COMPUTER SCIENCE 411

April, 2003 Time: 2 hrs.

Instructions

The exam contains questions totaling 100 points. Answer all questions.

This exam is closed book.

Please note that all grammars are provided using the CFG format: non-terminals are in lower-case and terminals are in upper-case. The productions associated with a non-terminal are separated by bars and ended with a period.

. 2

15 marks

1. Consider the following grammar:

```
exp -> exp ADD term
     | term.

term -> ID
      | ID index
      | LET stmts IN exp END.

stmts -> stmts stmt
       |.

stmt -> lexp ASSIGN exp SEMI.

lexp -> lexp index
      | ID.

index -> LPAR exp RPAR.
```

3 marks

(a) Give three examples of lists of terminals recognized by this grammar.

5 marks

(b) Calculate the *first* and *follow* sets for the above grammar. Indicate which non-terminals are nullable and which are endable.

2 marks

(c) Explain why this grammar is not LL(1).

5 marks

(d) Transform the grammar to remove left recursion. Is the result LL(1)? Can you transform the grammar to become LL(1)?

.....3

35 marks

2. For each of these grammars you are expected to calculate the vital statistics (that is nullable, endable, first, and follow properties) and to give the appropriate item automaton.

20 marks

(a) Draw a Venn diagram describing the relationship between LL(1), SLR(1), LALR(1), and LR(1) *grammars*. Now draw the Venn diagram of the *languages* these grammars recognize.

Where do the following two grammars fit into these diagrams?

```
s -> a A a B | b B b A.
a -> .
b -> .
```

and

```
s -> A a D | B b D | c.
c -> A b E | B a E.
a -> C.
b -> C.
```

15 marks

Where would an ambiguous grammar lie in these diagrams?

(b) Show that the following grammar is neither LL(1) nor SLR(1), but that it is LALR(1). Transform the grammar into an SLR(1) grammar (which is not LL(1)) then transform the grammar into an LL(1) grammar.

```
s -> ID
   | v ASSIGN e.
v -> ID.
e -> v
   | NUM.
```

25 marks

3. This question is concerned with the organization of a stack-based run-time environment:

   (a) Describe the organization of an *activation record* for a function which has local arrays.

   (b) Explain the purpose of the *dynamic link* and how it differs from the *static link*.

   (c) Explain how storage for *local arrays* can be organized on the stack. In this organization can arrays be returned by functions or passed as arguments to functions?

   (d) Explain how, in a program, the "static distance" of an occurrence of an identifier from its place of declaration is calculated. Explain why is it necessary to know this distance for an occurrence of a *function identifier*.

   (e) Explain how one uses activation records to access *non-local variables*.

   (f) Describe the *caller* and *callee* responsibilities when a function is called.

   (g) Describe what has to be done on a *return* from a function call.

25 marks

4. Given the following abstract syntax tree for expressions with "simultaneous" typed let declarations draw the plumbing diagrams and develop pseudo-code to determine the type of the expression given an initial symbol table:

```
Expr = ADD(Expr,Expr)
     | OR(Expr,Expr)
     | LE(Expr,Expr)
     | ID String
     | LET(Decls,Expr)
Decls = DCL(Decl)
     | DCLS(Decls,Decl)
Decl = DEC(String,Type,Expr)
Type = INT
     | BOOL
```

A typical abstract syntax tree (which will not type) could then be:

```
ADD(ID x,LET(DCLS(DCL(DEC(y,BOOL,OR(ID a,ID b)))
                     ,DEC(z,BOOL,ADD(ID c,ID b)))
            ,OR(ID y,ID z)))
```

You may assume that you have the following operations on your symbol table and types:

```
lookup(st:Symtab,str:String ; t:Type)
insert(st:Symtab,str:String,t:Type ; st':Symtab)
eq(t:Type,t':Type ; )
```

for respectively looking up the type of a variable (as most recently inserted), inserting the name of a variable and its type into the symbol table, and testing the equality of types. Note that both the first and last operation can fail and raise an exception.

`OR` is an operation on booleans, while `ADD` and `LE` are operations on integers with the latter producing a boolean:

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

* JRBC