THE UNIVERSITY OF CALGARY

FACULTY OF SCIENCE

FINAL EXAMINATION

COMPUTER SCIENCE 411

April, 2006 Time: 2 hrs.

Instructions

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

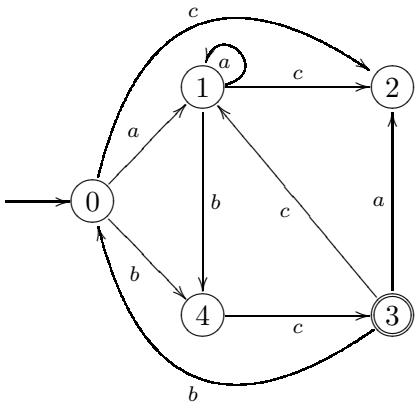
This exam is closed book. You are expected to *explain* all the answers you provide.

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

10 marks

1. Consider the following automaton (initial state 0 and final state 3):



(a) Is this automaton deterministic or non-deterministic?

(b) Can this automaton be minimized?

(c) Give a regular expression which recognizes the same set of strings as this automaton.

15 marks

2. Consider the following grammar:

```
E -> E + T
   | T.
T -> M I
   | M ( E ).
M -> { S }
   |.
S -> S A
   |.
A -> I = E ;.
I -> I [ E ]
   | id.
```

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

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

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

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

25 marks

3. Consider the following grammar (in which `0`, `1`, `2`, and `3` are terminals):

```
A -> B 2 A
   | 3.
B -> C 1 C 2 B 3
   |.
C -> 0
   |.
```

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

(b) Calculate the first sets, follows sets, nullables, and endables of this grammar;

(c) Calculate the LALR(1) (augmented) follow sets of this grammar;

(d) Determine to which of the following classes this grammar belongs: LL(1), LR(0), SLR(1), LALR(1), or LR(1);

(e) Draw a Venn diagram of the relationship between the above grammar classes.

25 marks

4. This question is concerned with the organization of stack-based run-time environments:

   (a) Describe the organization of an *activation record* for a function.

   (b) Explain how storage for multi-dimensional *local arrays* whose size is dynamically determined can be organized on the stack within the activation record of such a function.

   (c) In this organization can arrays be returned by functions or passed as arguments to functions? Do they have to be of fixed dimension? What would you have to do in order to be able to return arrays from functions?

   (d) Explain the purpose of the *static link* and how it differs from the *dynamic link*. When is a static link unnecessary? (Hint: why do some implementations of C not have a static link in the activation records of functions?)

   (e) 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*.

   (f) Explain why functions are assigned completely new code labels: what can go wrong with using their original name?

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

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

25 marks

5. (a) Write symbol table functions in Haskell for creating a new scope level, looking up the type of a symbol, and inserting a symbol with its type:

```
type SymTable = [[(String,Type)]]
data Type = INT | BOOL

new_level::SymTable -> SymTable
look_up::SymTable -> String -> Type
insert::SymTable -> String -> Type -> SymTable
```

The symbol table should report an error when one tries to insert a symbol into a level in which the symbol is already present, or when one tries to look up a symbol which is not present (in any of the levels).

(b) Given the following abstract syntax tree for "let expressions" with simultaneous declarations draw the plumbing diagrams and develop the Haskell code to determine the 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
          | NV Int
          | BV Bool
          | LET [(String,Expr)] Expr
```

A typical syntax tree (which will not type in any context) 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.