CSE 428, Spring 1999. Midterm 1, 24 February 1999

The exam consists of 5 exercises, totalling 100 points. Write your answers directly on these pages. **Please remember to write your name and SSN at the top of each sheet.**

Exercise 1 (20 pts)

Consider the following grammar, defining a fragment of an imperative language:

```
<Statement> ::= <Ide> := <Exp>
              | if <Exp> then <Stat_list> end
              | if <Exp> then <Stat_list> else <Stat_list> end
              | while <Exp> do <Stat_list> end
              | begin <Stat_list> end

<Stat_list> ::= <Statement> | <Statement> ; <Stat_list>
```

where `<Exp>` and `<Ide>` stand for the syntactic categories of expressions and identifiers. Their productions are not shown here: for the purpose of this exercise, just consider `<Exp>` and `<Ide>` like terminals.

1. Say whether this grammar is ambiguous or not, and justify your answer (If your answer is "yes", prove that it is indeed ambiguous. If your answer is "no", just give an informal motivation.)
2. Say what kind of associativity is enforced for ";".
3. For each of the following strings, say whether it is generated by the above grammar or not. In case it is generated, show the corresponding derivation tree.
   (a) `begin <Ide> := <Exp> end ; <Ide> := <Exp> end`
   (b) `while <Exp> do begin <Ide> := <Exp> end end`
   (c) `if <Exp> then <Ide> := <Exp> end else <Ide> := <Exp> end`
   (d) `if <Exp> then if <Exp> then <Ide> := <Exp> end else <Ide> := <Exp> end`

Exercise 2 (20 pts)

Consider the logical operator "nand", defined as follows:

$$\frac{r \vdash e_1 \ \mathit{eval}\ \texttt{false}}{r \vdash (e_1 \ \texttt{nand}\ e_2)\ \mathit{eval}\ \texttt{true}}$$

$$\frac{r \vdash e_1 \ \mathit{eval}\ \texttt{true} \quad r \vdash e_2 \ \mathit{eval}\ \texttt{false}}{r \vdash (e_1 \ \texttt{nand}\ e_2)\ \mathit{eval}\ \texttt{true}}$$

$$\frac{r \vdash e_1 \ \mathit{eval}\ \texttt{true} \quad r \vdash e_2 \ \mathit{eval}\ \texttt{true}}{r \vdash (e_1 \ \texttt{nand}\ e_2)\ \mathit{eval}\ \texttt{false}}$$

1. Is `nand` (with the above semantics) strict or non-strict? Remember that an operator `op` is strict if the evaluation of an expression of the form e_1 `op` e_2 always requires to evaluate both e_1 and e_2 before computing `op`. It is non-strict (or short-circuit) otherwise.

2. Consider the expression

   ```
   x < 0 nand (x = 0 nand 1/x > 0)
   ```

   where `x` is an identifier of type integer. Say what are the results (`true`, `false`, or `error`) of the above expression, depending on the possible values of `x`.

3. Suppose that we want to implement the above operator `nand`, by using a function.

   (a) What kind of parameter-passing method do we need?

   (b) Define the function in Pascal-like or C-like code. You can assume that the `if_then_else` construct is available, with the usual semantics.

Exercise 3 (20 pts)

Consider the ADT "simple list" described in Lecture 10 (see last page for the definition). Given a list of integers l, define $sum\_list(l)$ as the function that returns the sum of all nodes of the list l. For instance, if the sequence of nodes is

1 5 2 4 8

then the result should be 20.

1. Give a recursive definition of $sum\_list(l)$ in C-like or Pascal-like code. The function should be independent from the implementation of lists, i.e. it should use (as operations on lists) only the functions `is_empty`, `emptylist`, `cons`, `head` and `tail` of the specification of the ADT. In particular, it should not use the names `info` and `next`.
2. Say whether the function you have defined is tail-recursive or not.

Exercise 4 (20 pts)

Consider the following procedure declaration:

```
procedure p;
   var x,y: integer;                               /* variables x,y local to p */
   procedure q(y: integer);                          /* procedure q local to p */
      begin if x = y then r else print(x) end;                    /* body of q */
   procedure r;                                      /* procedure r local to p */
      var x : integer;                                /* variable x local to r */
      begin x := 2; if y = 2 then q(x) else print(x) end;         /* body of r */
   begin
   x:= 1;
   y:= 2;
   q(x)
   end;
```

1. What is the output produced by calling `p` in:
   (a) a language with dynamic scope
   (b) a language with static scope
2. For the case of static scope, show the activation records at the moment of the first execution of an instruction `print`. In particular, show the static and the control links, and the allocation of the variables and parameters. You can use a drawing.

Exercise 5 (20 pts)

Consider the following program in C or C++:

```
#include<stdio.h>
int *p;
void foo1(){
...
}
void foo2(){
...
}
main(){
   foo1();
   printf("%d",*p);
   foo2();
   printf("%d",*p);
}
```

Assume that this is the only program or process running on the system, that there is no procedure or function call in either `foo1` or `foo2`, and that `foo2` does not contain occurrences of `p`.

Assume that the first `printf` produces output 5.

1. (a) Is it possible that the second `printf` produces an output different from 5?

   (b) If we assume additionally that the body of `foo2` does not make any dynamic allocation (i.e. it does not use the instruction `new`), is it possible that the second `printf` produces an output different from 5?

   Justify your answers. In particular, in the cases in which a different output is produced, say what we can deduce about `foo1` (does it leave a memory leak? A dangling pointer to the stack? A dangling pointer to the heap? ...) and what, during the execution of `foo2`, might cause the modification of `*p`.

2. Consider the same program written in Pascal. What are, in this case, the answers to the questions (a) and (b) above? Justify your answers.

   Hint: remember that one difference between Pascal and C is that Pascal is designed so that it is not possible to have dangling references to the stack.