Name: ______________________ *Entry:* ____________ *Gp:* ___

Indian Institute of Technology Delhi
Department of Computer Science and Engineering

CSL302	Programming Languages	Major Exam
April 29, 2008	15:30–17:30	Maximum Marks: 100

Open notes. Write your name, entry number and group at the top of each sheet in the blanks provided. Answer all questions in the space provided, in blue or black ink (no pencils, no red pens). Budget your time according to the marks. Use the last page as "overflow space".
$\mathcal{X}$ is a set of variable names, with x a typical variable. *Exp* is the inductively defined set of expressions, with e, e' typical expressions. *Types* is the collection of types with τ a typical type. $\Gamma \in \mathcal{X} \rightarrow_{fin}$ *Types* is a typical type assignment. $\Gamma \vdash e : \tau$ is the typing relation "e has type τ under type assumptions Γ on the variables", and $\Gamma \vdash c$ *ok* is the relation that says that command c is free of type errors under type assumptions Γ.

Q1 (6x 2 marks) **Run-time Data Structures.** Give short one-sentence answers for each of the following:

1. What are `static` variables in C and where are they allocated?

2. With regard to lifetime and size, what kinds of data structures can safely be allocated on stack?

3. What is the difference between *garbage* and *dangling references*?

4. Why is garbage collection rarely spoken about in languages such as Pascal and C whereas it is very important in SML and Java?

5. Why do we need to allocate values on heap in a higher-order functional language? Is it safe to assume that any data structure not reachable from the stack is garbage, even if it is on the heap?

6. Why does the execution of an infinite chain of recursive calls in C or Pascal lead to a "Stack overruns heap" error message?

Q2 (3+5 marks) **For loops.** Consider the definite iteration "for-loop" command **for** $i := m$ **upto** n **do** {c} — where i is a fresh integer variable, the scope of which is only the body of the "for loop"; c is a command that can look up the value of i but cannot change the content of i; and m, n are integer vaues. It can be described informally as "execute command c repeatedly for different values of i, taken successively from that of m to that of n (both inclusive)."

1. Write the typing rule for the command **for** $i := m$ **upto** n **do** $\{c\}$.

$$\frac{}{\Gamma \vdash \textbf{for } i := m \textbf{ upto } n \textbf{ do } \{c\}\ ok}$$

2. Write the big-step evaluation rules for this command:

$$\frac{}{\beta \vdash \langle \textbf{for } i := m \textbf{ upto } n \textbf{ do } \{c\}, \sigma\rangle \Longrightarrow_c \sigma}$$

$$\frac{}{\beta \vdash \langle \textbf{for } i := m \textbf{ upto } n \textbf{ do } \{c\}, \sigma\rangle \Longrightarrow_c \sigma'}$$

Q3 [6+2+2 marks] **Priuciple of Abstraction.** The Principle of Abstraction does not merely state that procedures and functions must exist. Rather, it suggests that any construction for a given syntactic category (e.g., commands) can be adapted to a similar construction for another syntactic category (e.g., expressions).

1. What should the corresponding "definite iteration expression abstract" $\bigodot_{i=m}^{n}(\ F\ )$ compute, if higher-order function F : **integer** $\rightarrow \tau \rightarrow \tau$ can be considered as representing a family of functions f_i indexed by i (i.e., $F(i) = f_i$)?

2. What should the expression $\bigodot_{i=m}^{n}(\ F\ )$ return if $n < m$?

3. Provide the induction hypothesis needed to show that $(\bigodot_{i=1}^{n}(\ \lambda x.(c_i + x)\ ))(z) = z + (\sum_{i=1}^{n} c_i)$.

Q4 [3+15 marks] **Parameter Passing Modes.**

1. Consider a procedure $P(x)$ with a *call-by-name* parameter x, which is invoked with actual argument e. In what environment (table) and what store should the argument e be evaluated? [*Strike out the incorrect answers*]

   The environment prevailing (i) at the time of definition of P / (ii) at the time P was called / (iii) at the current time when e is required to be evaluated.
   The store prevailing (i) at the time of definition of P / (ii) at the time P was called / (iii) at the current time when e is required to be evaluated.

2. Consider the following procedures in an imperative language, which we have extended to admit a parallel assignment statement (with the usual restrictions on LHS's being distinct, to keep the effect deterministic).

```
procedure swap(mode  x, y:
integer);
var temp:  integer;
begin
temp := x; x := y; y := temp
end (* swap *)
```

```
procedure parswap(mode  x, y:
integer);
begin
x := y  ||  y := x
end (* parswap *)
```

Suppose `i = 1`, `a[i] = 2` and `a[2] = 3`, when we call `swap(i,a[i])` or `parswap(i,a[i])`. What are the values of `i`, `a[1]` and `a[2]` on exit from the procedure call, assuming the following parameter passing *modes* for both parameters?

(a) *mode* is Call by value.

swap: i =	a[1] =	a[2] =
parswap: i =	a[1] =	a[2] =

(b) *mode* is Call by reference.

swap: i =	a[1] =	a[2] =
parswap: i =	a[1] =	a[2] =

(c) *mode* is Call by name.

swap: i =	a[1] =	a[2] =
parswap: i =	a[1] =	a[2] =

Q5 (4x4 marks) **Procedures as parameters.**

Pascal, which is implemented using stack-allocated procedure frames, allows procedures to be passed as a parameters to other procedures; for example, procedure P can call procedure Q with procedure R as an argument., *i.e.,* The code *Q(R);* can appear within the body of P. [Hint: Recall the invariant maintained about the environment for stack allocated proeedure frames].

1. What relationships should exist between P, Q, R in the text of the program? Why?

2. To ensure type correctness of the procedure call, what type information needs to be specified and checked at compile time in the procedure definitions of P and Q?
Specified in P:

   Checked in P:

   Specified in Q:

   Checked in Q:

3. At run-time, what information about R should procedure P pass to procedure Q in the call stack (i.e., how is procedure R represented as an argument)?

4. Can one use display records to implement static links, when passing procedures as parameters, and if so what information ueeds to be saved on the stack?

Q6 (8marks) **Sub-typing.** Suppose our little imperative language has a notion of "subtypes" ($\tau_1 \sqsubseteq \tau_2$, c.g., **integer** $\sqsubseteq$ **real**), and *any value of a subtype can be used wherever one of a super-type is expected, without type errors.* Modify the well-formed rclation $\Gamma \vdash c$ *ok* to account for sub-typing. Also incorporated is a rule for procedure call P(e) if P is defined to take a single formal parameter x of type τ.

$$\frac{}{\Gamma \vdash \textbf{skip}\ ok} \qquad \frac{}{\Gamma \vdash x := \mathrm{e}\ ok}$$

$$\frac{}{\Gamma \vdash \mathrm{c}_1;\ c_2\ ok} \qquad \frac{}{\Gamma \vdash \textbf{if}\ e\ \textbf{then}\ \mathrm{c}_1\ \textbf{else}\ \mathrm{c}_2\ \textbf{fi}\ ok}$$

$$\frac{}{\Gamma \vdash \textbf{while}\ e\ \textbf{do}\ \mathrm{e}\ \textbf{od}\ ok} \qquad \frac{\Gamma \vdash P : \tau\ \textbf{proc}}{\Gamma \vdash P(\mathrm{e})\ ok}$$

Q7 [8+(3+5) marks] **Scope.** In a term $\lambda x.\mathrm{e}$, bound occurrences of a variablc x are merely place holders in the body that let us refer to thc formal parameter, which is indicated by the *binding* occurrence x. However, this use of variables leads us to the tedious notion of α-conversion. De Bruijn suggested that occurrences of variables, except for free variables, can be climinated by using a notion of *indexing*, where we replace variables by numerical indexes such as $\underline{0}, \underline{1}, \ldots, \underline{i}$ which count the number of λ's that one needs to cross in the path between a bound occurrence of a variable and the λ node corresponding to its binding occurrence. For example, $\lambda x.\lambda y.(x\ (y\ \lambda x.(x\ y)))$ is represented as $\lambda.\lambda.(\underline{1}\ (\underline{0}\ \lambda.(\underline{0}\ \underline{1})))$ in DeBruijn notation.

1. Define a function that converts a closed λ-term (with no free variables) to a DeBruijn-indexed λ-term. [Hint: write a top-down function which works with a LIFO list of the binding occurrences from the root to the current term].

2. Prove the following corollaries of the Church-Rosser Theorem:
   (a) If $e_1 =_\beta e_2$ and e_1, e_2 are in β-normal form, then $e_1 \equiv_\alpha e_2$.

   (b) If $e_1 =_\beta e_2$ then either e_1 and e_2 do not have a β-normal form, or both have the same β-normal form (upto $\equiv_\alpha$).

Q8 [6+6 marks] **Type Inference.** Show your reasoning in each of the following type inference problems:

1. Any Y combinator satisfies the equation $(Y\ f) = (f\ (Y\ f))$. If the LHS and RHS must have the same type, what should the type of any Y combinator be? [Note: The Y combinators studied by you cannot be well-typed!]

2. If all the Church Numerals, e.g., $\underline{0} \overset{def}{=} \lambda f.\lambda x.x$ and $\underline{2} \overset{def}{=} \lambda f.\lambda x.(f\ (f\ x))$, should have the same type, then what is the type for such encodings of natural numbers?