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. $\Gamma \in \mathcal{X} \to_{fin}$ *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)."

Name: Entry: Gp:

Indian Institute of Technology Delhi
Department of Computer Science and Engineering

CS 232 F	Programming Languages	Major Test
May 12, 2004	10:30–12:30	Maximum Marks: 100

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 for rough work or as overflow space.

Q1 (3+6+3+6 marks) **Language Design Principles.**
Consider a language $\mathcal{L}_0$ where the only expressions are variables, i.e, $e \in Exp ::= x$ where $x \in \mathcal{X}$.

1. Suppose by the Principle of Abstraction, we extend $\mathcal{L}_0$ to $\mathcal{L}_1$ by admitting 1-parameter abstracts on expressions (which are not first-class entities). What are the various syntactic classes in $\mathcal{L}_1$? (Give abstract grammars)

   $e ::=$ $a \in Abs ::=$

2. Now if we extend $\mathcal{L}_1$ to $\mathcal{L}_2$ by having the Principle of Qualification applied to each of the syntactic classes (including any new classes introduced by this principle), what are the various syntactic classes? (Give their abstract grammars)

   $e ::=$ $a ::=$

3. Suppose we modify $\mathcal{L}_2$ to $\mathcal{L}_3$ such that 1-parameter abstracts on expressions are now considered first-class expressions, how would the syntactic classes (and their abstract grammars) change?

4. Give typing rules for the language $\mathcal{L}_3$.

Q2 (10+2+2+2 marks) **Big-step semantics for Logic Programming.**
Assume *MGU* is an algorithm that computes a most general unifier of two terms if it exists, returning failure otherwise. Fill in the blanks in a big-step specification of execution of a pure logic programming language, based on Prolog, but in which rule order doesn't matter. Assume that the program $\mathcal{P}$ consists of a collection of *clauses* C_i, each of the form p_i :- $q_{i,1}, \ldots, q_{i,k_i}$ for some $k_i \geq 0$. Let $G_1, \ldots, G_m$ be the goal.

$$\frac{\mathcal{P} \vdash \qquad\qquad\qquad\qquad \Longrightarrow}{\mathcal{P} \vdash G_1, \ldots, G_m \Longrightarrow \sigma}$$

if for some clause $C_i \in \mathcal{P}$, $\sigma_0 = MGU(p_i, \_)$.

What is the simplified form of this inference rule if the matching clause C_i is a fact?

What is σ_0 if $G_i \equiv p_i$?

What happens if at any stage, no C_i is found such that p_i unifies with the first subgoal?

Q3 (6+4+2+6 marks) **Imperative Paradigm.**

1. Consider the command **for** x := e_1 **to** e_2 **do** c **rof** with the informal semantics that the lower and upper limits e_1 and, e_2 are first evaluated, and then the body c is executed for different values of x which takes consecutively the values ranging from the lower limit to the upper limit. The body is not executed if the value of x exceeds that of the upper limit e_2. Express this command in terms of an equivalent command using the **while** construct.

   Give a block diagram showing how assembly code generated for this command will be laid out for execution on a typical architecture, clearly indicating the jumps, tests and code blocks.

Name: Entry: Gp:

ing
ing
sts
be

Which principle suggests that, in analogy with the **for** command, there should be expression iteration constructs such as $\sum_{i=e_1}^{e_n} e(i)$ or $\prod_{i=e_1}^{e_n} e(i)$?

2. The command **repeat** c **until** e **taeper** (where c is a command and e is a boolean expression) can be described informally as "execute command c repeatedly, stopping when the condition e becomes *true*". Give Big-step rule(s) for this command

-).

Q4 (4+4+4 marks) **Parameter passing.**

Apart from the usual call-by-value parameter passing mechanism, the language Algol had a *call-by-name* parameter-passing mechanism which involved a (lazy) evaluation of the argument each time it was required in the body of the called procedure. It was based on the idea of substitution in the λ-calculus, that is, replacing the formal parameter of the procedure by the text of the argument expression, taking care not to capture any free variables (by suitably renaming local variables in the procedure) and maintaining the bindings of the free variables in the argument. A *function* in Algol was nothing but a procedure that returned a value. Consider the following procedure *sum* in the following program fragment, where $\langle ppmode\rangle$ will be either *call-by-value* or *call-by-name*.

```
 : : : : :
var
   i: integer;
   result: real;
   B, C: array[1..100] of real;
function sum(lo,hi: integer; <ppmode> x: real): real;
var
   sum: real;
begin
     sum := 0.0;
     for i:= lo to hi do
         sum := x+sum;
     return(sum)
end;
: : : :
begin
(* initialization of arrays B, C omitted *)
: : : :
i := 1;
result := sum(1,100, B[i]*C[i]);
: : : :
end.
```

What is the *result* of the function call to *sum* if the parameter passing mode is *call-by-value*?

What is the *result* of the function call to *sum* is the parameter passing mode is *call-by-name*?

In 2-3 short sentences, suggest how call by name should be implemented.

Q5 (10+6 marks) **α-equivalence and DeBruijn indices.**
Let $e ::= x \mid \lambda x.e \mid (e_1\ e_2)$. The notion of α-equivalence essentially says that local variables and/or parameters can be systematically renamed without altering the meaning of the program. However, manipulating programs into α-equivalent forms can be a nuisance, and one approach is to eliminate bound variables altogether by a clever indexing scheme, which replaces each *bound occurrence* of a variable by the number of λ's crossed to reach its *binding* occurrence (free occurrences are left unchanged). This is exactly the number of levels outward that the variable is "declared". Provide a function *trans* that converts expressions to indexed-forms $\varepsilon ::= n \mid x \mid \lambda\varepsilon \mid (\varepsilon_1\ \varepsilon_2)$. For example, the expression $\lambda x.((\lambda y.(x\ y))(x\ z))$ translates to $\lambda(\lambda((1\ 0)\ (0\ z)))$. [Hint: to determine the index, use the list of variables whose binding occurrences are at the λ-nodes which are encountered on the path up to the root].

Sketch a proof that if $e_1 \equiv_\alpha e_2$ then $trans(e_1) \equiv trans(e_1)$. Here $\equiv$ denotes syntactic identity. Mention only what technique you will use, and the main lemmas if any.

Name:

Q6 (

Q6 (4+4+4+8 marks) **Implementation issues.**

1. In C, an array is considered the same as a pointer to its first value. Name one advantage and one disadvantage to this idea.

2. Give one advantage of basing equivalence of types on their structure. Why does C use structural equivalence for all types and type constructions except for records (`structs`)? [Hint: consider the type of pointers to nodes in a linked list]

3. Since *square* can be a sub-type (sub-class) of both *rhombus* and *rectangle*, there arises the possibility of inheritance from multiple super-classes. Mention what problems may arise due to multiple inheritance (1 sentence).

4. The language Pascal allows a limited facility for defining procedures that can take a procedure as a parameter (the latter's formal parameter types must be specified so that all calls to it can be type-checked). In Pascal's implementation, all closures are implicit on the stack due to the calling discipline. So what information about the argument procedure should be passed by the caller to the called procedure, so that the stack-based implicit closure property is maintained? [Hint: Consider a procedure P passing its child R in a call to P's sibling Q]

Indian Institute of Technology Delhi
Department of Computer Science and Engineering

SL302 Programming Languages Major Exam

ay 4, 2007 15:30–17:30 Maximum Marks: 100

pen book and notes. Write your name, entry number and group at the top of <u>each sheet</u> in the blanks ovided. Answer all questions in the space provided, in blue or black ink (no pencils, no red pens). Budget your e according to the marks. Do rough work on separate sheets.

Q1. (4x3 marks) **Unification.** For each of the following pairs of terms, if the *most general unifier* exists, present it as a *simultaneous substitution*, or else state why it does not exist:

1. $g(h(X,a),h(b,X))$ and $g(h(b,a),h(X,Z))$.

2. $g(h(X,a),Y)$ and $g(Z,h(b,X))$.

3. $g(h(X,a),Y)$ and $g(Y,h(b,X))$.

4. $g(h(a,b),X)$ and $g(Y,h(a,X))$.

Q2. (3+3+4 marks) **Prolog.** A list L_1 (which may even be empty) is a *prefix* of L_2 if some initial part of L_2 is exactly L_1. Likewise list L_3 is a *suffix* of L_4, if some final part of L_4 is L_3. L_5 is a *sublist* of L_6 if it appears somewhere within L_6. Suppose we are given a Prolog program append(L_1, L_2, L_3) for appending lists. Write Prolog programs for

1. `prefix`$(L1, L2)$:-

2. `suffix`$(L1, L2)$:-

3. `sublist`$(L1, L2)$:-

Q3. (16 marks) **Subtyping.** Type τ_1 is called a *subtype* of type τ_2 if wherever a value of type τ_2 is expected, a value of type τ_1 can safely be used. (This notion is at the core of the idea of subclass used for inheritance in Object-oriented programming). Suppose we have the following abstract grammar for types.

$$\tau ::= \textbf{unit} \mid \textbf{int} \mid \textbf{bool} \mid \textbf{real} \mid \tau_1 * \tau_2 \mid \tau_1 \rightarrow \tau_2$$

and wish to define a subtype relation between types defined inductively using the following rules: (a) **unit** is a subtype of any type; (b) **int** is a subtype of **real**; (c) τ_1 is a subtype of $\tau_1 * \tau_2$; (d) if τ_1' is a subtype of τ_1 and τ_2 is a subtype of τ_2', then $\tau_1 \rightarrow \tau_2$ is a subtype of $\tau_1' \rightarrow \tau_2'$; (e) the subtyping relation is reflexive and transitive.

Define a Prolog program $\mathtt{subtype}(T_1, T_2)$ to express the subtyping relation.

Q4. (8 marks) **Σ-homomorphisms.** Consider signature $\Sigma = \{0^{(0)}, 1^{(0)}, +^{(2)}\}$. Let $\mathcal{A}$ be the Σ-algebra with natural numbers as the carrier set, 0 interpreted as *zero*, 1 as *one* and + as addition on natural numbers. What should the Σ-algebra $\mathcal{B}$ be, if the function *odd* : $\mathbb{N} \rightarrow \mathbb{B}$ is to be a Σ-homomorphism?

Carrier Set =

Interpretation of 0 is

Interpretation of 1 is

Interpretation of + is

Q5. (8+6 marks) **Call-by-value-result.** The *call-by-reference* parameter passing mechanism is necessary for writing procedures (such as swap) in which the procedure is intended to change the contents of its arguments. However, this mechanism is not a"clean abstraction" in that it suffers from the problem that any assignment to a reference argument immediate changes the corresponding global variable *before the completion of the execution of the procedure.* A better variant of this method is *call-by-value-result,* in which the contents of the argument variables are copied into fresh storage in the called procedure, the assignments are made to these locations, and finally, just before exit, the contents of these new locations are copied back into the argument variables. Suppose P is a procedure with a single call-by-value-result parameter y and whose body is command c.

1. Provide big-step operational semantics for calling procedure P with (global) variable x as argument.

$$\frac{}{\beta \vdash \langle P(x), \sigma\rangle \implies \_} \quad \beta(P) = \ll \beta_1, \lambda y.c \gg, \beta(x) = l,$$

2. Suppose we have an implementation of procedure calls in an imperative language for the *call by value* parameter-passing mechanism. How will you modify this by adding some code before and after call-by-value procedure calls so that call-by-value-result can be simulated? *Explain precisely by providing pseudocode.*

Q6. (4x3 marks) **Lifetime and scope.** Suppose during the runtime execution of a program, an object o is created at time t_1 and is destroyed at time t_2, whereas it is associated via a *binding* to name x from time t_3 (when the binding is created) to t_4 (when the binding is destroyed). Express the relation that should hold (in terms of some of t_1, t_2, t_3, t_4) if

1. object o is garbage:

2. object o is always available (via name x):

3. name x is uninitialized for some time:

4. name x is a dangling reference:

Q7. (2+10 marks) **Definite iterations.** In mathematics, we have expressions such as $\sum_{i=a}^{i=b} f(i)$ and $\prod_{i=a}^{i=b} f(i)$. By which principle should we consider a special iterator command **foreach** i **from** a **to** b **do** C **od**, where C is a command in which i appears as an integer variable?

Provide Big-step semantics for the construct **foreach** i **from** e_1 **to** e_2 **do** C **od**, assuming that (i) e_1 and e_2 are evaluated first to values v_1 and v_2 respectively; (ii) if $v_2 < v_1$, the command C is not executed, (iii) otherwise, C is serially executed $v_2 - v_1 + 1$ times with a fresh variable i taking successive values from v_1 to v_2 for each iteration of C; (iv) assume that i is only read but not otherwise changed within C, and that (v) variable i is not accessible after the end of the loop.

Q8. (8+8 marks) **λ-terms and their types.** Recall that Church numerals (encodings of the natural numerals in the λ-calculus) are $\lambda f.\lambda x.x, \lambda f.\lambda x.(f\ x), \lambda f.\lambda x.(f\ (f\ x)), \ldots$. If all these λ-terms were given the same type according to the typing rules given for abstraction and application, what would that type be?

Type of Church Numerals: ____

Any Y combinator satisfies the equation $Y\ f\ =_\beta\ f(Y\ f)$ for any function f. Assuming that the left and right sides of the equations must have the same type, what type should any Y combinator have? (*Make a suitable assumption about the type of f, and solve the constraints*).

Type of Y should be: ____