Indian Institute of Technology Delhi
Department of Computer Science and Engineering

CSL 302 Programming Languages Major Test

May 11, 2005 13:00–15:00 Maximum Marks: 120 (12 Bonus)

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 (12 x 1 marks) **Language Design.**

1. Which principle/idea suggests the introduction of block structure?
   Principle of Qualification
2. Which principle/idea suggests that there can be (polymorphic, generic) type constructions such as lists, binary trees, etc. ?
   Principle of Abstraction
3. Which principle/idea suggests that functions can be returned as results of function application?
   First-class Functions/Abstractions
4. If we were to introduce a notion of scoped definitions into Prolog, which principle/idea would suggest its operational semantics (and implementation)?
   Principle of Correspondence
5. What principle/idea 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 $\prod_{i=e_1}^{e_n} e(i)$?
   Principle of Abstraction
6. Which principle/idea suggests that evaluating an expression several times always yields the same result?
   Referential Transparency
7. Which parameter passing mechanism evaluates arguments exactly once?
   Call by value
8. What principle/idea suggests that operations on a particular data type should be bundled together with the data objects?
   Encapsulation
9. What is the notion of having a common symbol for different operations called?
   Overloading or ad hoc polymorphism
10. What principle/idea suggests that control features such as "go to" should not be included in a programming language?
    Structured programming
11. Which notion of type equivalence is based on the idea that type definitions are merely abbreviations for canonical type descriptions?
    Structural equivalence of types
12. What idea allows code developed for a method of a superclass to be applicable to objects of a subclass?
    Inheritance, or subtype polymorphism or extensible polymorphism

Q2 (6+2+2+2 marks) **Big-step semantics for Logic Programming**.
Let *MGU* be an algorithm that computes a most general unifier of two terms if it exists, reporting 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 goal order matters, but rule order doesn't matter. Assume that the program $\mathcal{P}$ consists of a collection of *clauses* $\mathcal{C}_i$, 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.

$$\frac{\mathcal{P} \vdash q_{i,1}\sigma_0, \ldots, q_{i,k_i}\sigma_0, G_2\sigma_0, \ldots, G_m\sigma_0 \Longrightarrow \sigma}{\mathcal{P} \vdash G_1, \ldots, G_m \Longrightarrow \sigma}$$

if for some clause $C_i \in \mathcal{P}$, $\sigma_0 = MGU(p_i, G_1)$.

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

$$\frac{\mathcal{P} \vdash G_2\sigma_0, \ldots, G_m\sigma_0 \Longrightarrow \sigma}{\mathcal{P} \vdash G_1, \ldots, G_m \Longrightarrow \sigma}$$

if for some clause $C_i \in \mathcal{P}$, $\sigma_0 = MGU(p_i, G_1)$.

What is σ_0 if $G_i \equiv p_i$?
identity substitution

What happens if at any stage, no C_i is found such that p_i unifies with the first subgoal? (One sentence)
The rule fails, leading to the program failing if all instances of the rule fail.
Note: Big-step semantics specifies all possible answers. Backtracking is not a concept that is relevant in the big-step semantics, since it is a strategy to enumerate the possible answers.

Q3 (2x3+8 marks) **Relational and functional paradigm.**

Since in mathematics, a function is a special case of a relation, it is possible to convert any functional program into a logic program in a systematic way. First outline how to solve the sub-problems in *one short phrase each* Be precise.

- How should a function $f : \alpha_1 \to \alpha_2 \to \ldots \to \alpha_k \to \beta$ be represented? (Assume α_i, β are simple types).
  As a relation $fR \subseteq \alpha_1 \times \ldots \times \alpha_k \times \beta$.
- How should sub-expressions and in particular, recursive calls be expressed?
  By introducing a new variable for the result of a subexpression or recursive call, and using that name in place of the sub-expression in the term standing for the answer.
- How should a clause of a function definition $f\ x\ y\ z\ =\ e$ be represented?
  As a Prolog clause $fR(X, Y, Z, E)$, where E is the name introduced to represent expression e.

Now using these hints, express the following "first-order" functional program (written in ML) as a logic program (use Prolog syntax)

```
fun zip [] [] = []
  | zip (x::xs) (y::ys) = (x,y) :: (zip xs ys);

zip([],[],[]).
zip([X|XS], [Y|YS], [(X,Y)|Z]) :- zip(XS,YS,Z).
```

This is a cleaned-up form of the version obtained from the rules above:

```
zip([],[],L) :- L is [].
zip([X|XS], [Y|YS], ZS) :- zip(XS,YS,Z), ZS is [(X,Y)|Z].
```

Q4 ((2+8)+6 marks, 8 bonus) **Typed functional programs.**

For each of the following expressions, give their most general type. (Suggestion: use different type variables for the types of S and K).

- `fun K x y = x`

  `val K = fn - :` $\forall\alpha\forall\beta(\alpha \to \beta \to \alpha)$.
  Assume $x : \alpha, y : \beta, rhs : \gamma$. Therefore $K : \alpha \to \beta \to \gamma$; but since since rhs is x, $\gamma = \alpha$.

– `fun S x y z = ((x z) (y z))`

`val S = fn - :` $\forall\gamma\forall\delta\forall\epsilon((\gamma \to \delta \to \epsilon) \to (\gamma \to \delta) \to (\gamma \to \epsilon))$.
Assume $x : \alpha, y : \beta, z : \gamma, rhs : \epsilon$. Therefore $S : \alpha \to \beta \to \gamma \to \epsilon$. Let $(y\ z) : \delta$ (1). Therefore $(x\ z) : \delta \to \epsilon$ (2). (1) means $y : \gamma \to \delta$, whence $\beta = (\gamma \to \delta)$ (3). Also from (2), $x : \gamma \to \theta$, where $theta = \delta \to \epsilon$. Thus $\alpha = (\gamma \to \delta \to \epsilon)$.

Bonus
```
let
   fun K x y = x
   fun S x y z = (x z) (y z)
in
   ((S K) K)
end;
```

`val it = fn - :` $\forall\gamma.\gamma \to \gamma$.
Generalization is used for the let rule. So different instances for the type of K are chosen. The first instance is chosen to have type $\gamma \to (\gamma \to \delta) \to \gamma$ by instantiating α to γ, β to $\gamma \to delta \to \gamma$. The second instance of K is chosen to have type $\gamma \to \delta \to \gamma$.

(b) Two function expressions are equivalent if for each input they return the same result. Show that $((S\ K)\ K)$ is operationally equivalent to the identity function.

Let e be any expression with v its canonical value (weak head normal form). If e does not have a canonical value, both expressions do not terminate. Note that $(I\ e)$ evaluates to v. Also $(((S\ K)\ K)\ e)$ simplifies to $(((S\ K)\ K)\ v)$ which is then simplified to $((K\ v)\ (K\ v))$, by the definition of S. This simplifies to v by the definition of K.

Q5 (4 x 4 marks) **Parameter passing.**
The *call-by-name* parameter-passing mechanism in Algol 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 function procedure *sum* in the following program fragment, where ⟨*ppmode*⟩ 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.
```

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

   100(A[1]*B[1])*

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

   $\sum_{j=1}^{100}(A[j] * B[j])$

3. According to the Principle of Correspondence, what code will be equivalent to the block (in a Pascal-like syntax) **var** $x : \tau := e$ **begin** c **end**.

   procedure $P(x : \tau)$; **begin** c **end**
   $\vdots$
   $P(e)$

4. In 2-3 short sentences, suggest how the closure for a call by name argument can be implemented and passed in an Algol-like language.
   A closure for a call by name argument e is implemented in Algol-like languages by 2 pieces of information which are passed together to the called procedure: (1) a pointer to the compiled code for the expression e and (2) a pointer to the caller's frame. The latter information captures the "environment" (bindings of variables) with respect to which the expression code is to be run. (These two pieces of information are quite like the closure of an anonymous function with no arguments).

Q6 (6+10 marks) **Imperative paradigm.**

Dijkstra's guarded choice language includes two non-deterministic construct with the following informal descriptions. You have to give Big-step Operational formalizations of the semantics.

1. A conditional command **if** $\square_{i=1}^{n}\ e_i \triangleright c_i$ **fi** with the interpretation that if in the current state, any of the e_i evaluates to **true** then the corresponding c_i *may* be executed – the choice is made non-deterministically if more than one e_i evaluates to true. If no e_i evaluates to true, then the command is "stuck" (does not return).
   For each $i \in \{1, \ldots, n\}$ *there is a rule*

$$\frac{\langle \gamma, \sigma\rangle \vdash e_i \Longrightarrow \underline{\textbf{true}} \quad \gamma \vdash \langle \sigma, c_i\rangle \Longrightarrow_c \sigma'}{\gamma \vdash \langle \sigma, \textbf{if } \square_{i=1}^{n}\ e_i \triangleright c_i \textbf{ fi}\rangle \Longrightarrow_c \sigma'}$$

   Note there is no need for a rule if all e_i *evaluate to* **false**, *since absence of a rule implies that the input-output relationship is not specified in that case.*

2. An iteration command **do** $\square_{i=1}^{n}\ e_i \triangleright c_i$ **do** with the interpretation that if in the current state, any of the e_i evaluates to **true** then the corresponding c_i *may* be executed and then the loop in executed again from the resulting state. (Again, the choice of c_i is made non-deterministically if more than one e_i evaluates to **true**). If no e_i evaluates to true in the current state, then the loop terminates in that state.

$$\frac{\forall i \in \{1, \ldots, n\} \quad \langle \gamma, \sigma\rangle \vdash e_i \Longrightarrow \underline{\textbf{false}}}{\gamma \vdash \langle \sigma, \textbf{do } \square_{i=1}^{n}\ e_i \triangleright c_i \textbf{ do}\rangle \Longrightarrow_c \sigma}$$

   All e_i *evaluating to false terminates the loop, yielding the same store in which execution began.*

$$\frac{\langle \gamma, \sigma\rangle \vdash e_i \Longrightarrow \underline{\textbf{true}} \quad \gamma \vdash \langle \sigma, c_i\rangle \Longrightarrow_c \sigma' \quad \gamma \vdash \langle \sigma', \textbf{do } \square_{i=1}^{n}\ e_i \triangleright c_i \textbf{ do}\rangle \Longrightarrow_c \sigma''}{\gamma \vdash \langle \sigma, \textbf{do } \square_{i=1}^{n}\ e_i \triangleright c_i \textbf{ do}\rangle \Longrightarrow_c \sigma''}$$

If any e_i evaluates to true, execute the corresponding c_i and the retry the loop from the store reached after executing the c_i. Some marks were allocated for explaining the two rules.

Q7 (2x3 marks, 4 bonus) **λ-Calculus.**
Let Δ stand for $\lambda x.(x\ x)$, Ω stand for $(\Delta\ \Delta)$ and K stand for $\lambda x.\lambda y.x$.

1. Show that $\Omega\ \rightarrow_{1\beta}\ \Omega$

$$\Omega\ \equiv\ (\lambda x.(x\ x)\ \Delta)\ \rightarrow_{1\beta}\ (x\ x)[\Delta/x]\ \equiv\ (\Delta\ \Delta)\ \equiv\ \Omega$$

2. To what normal form does $((K\ K)\ \Omega)$ reduce if we follow the
   - call by value (eager, applicative order) evaluation strategy
     Does not reach a normal form (loops indefinitely).
   - the call by name (lazy, normal order) evaluation strategy
     Normal form is K.

Bonus Do your answers agree with or differ from the Church-Rosser confluence property? Justify in 1 sentence.
Agrees with the C-R Theorem. Both strategies are sub-relations of β-reduction. While one reaches normal form, since the other does not yield a different normal form, there is no disagreement with CR Theorem. In fact, if one abandons the c-b-v strategy at any stage and adopts the other strategy, the normal form can be reached.

Q8 (6+6+8+8 marks) **Implementation issues.**

1. For each of the following, state whether the action is done by the caller or the called procedure, with a 1-phrase justification.
   (a) saving static link in stack frame
   Caller. Called function cannot know the location of the latest frame instance of its parent on the stack.
   (b) saving the k^{th} display register
   Called. While both can in principle save the display, it is better for the called to do so both for efficiency, and because it knows better where to store it in the frame, and why it needs to be saved.
   (c) saving dynamic link in stack frame (frame pointer)
   Called. In principle, both can. But it is more efficient for called procedure to do so (code for this appears once) – copy fp onto stack.
   (d) allocating space for return value
   Caller. It uses the result, which must therefore be on top of stack on return. Size and type are known by both.
   (e) changing frame pointer to point to the new frame
   Called. It may be possible for caller to do so, but both for efficiency – cp sp to fp – and since called "knows" its frame better, it is preferable to do so in called.
   (f) restoring other saved registers
   Called. Of course, it depends on who saved the registers; but generally the called saves registers since only it "knows" which ones need to be reused.
2. Write 1 sentence each on the following
   (a) How is a variable size argument stored in a stack frame?
   Note it is a variable-sized argument (not a variable number of arguments). We should assume that the size is not known at compile time. The argument is stored in part of the stack where variable size data can be placed (e.g., below the fixed argument part). In the fixed size part a pointer to the argument and information on size and how to interpret it ("dope vector") are stored. (Note: The variable-szied argument could have been in heap, but that was not asked.)

(b) Where are "static" variables in C allocated?
In a fixed part of memory (typically below stack). Some could be in registers, but these must be "dedicated".

(c) Why are function closures in ML allocated on heap?
Their lifetimes may exceed from those of the creating "context", so closures cannot be represented in a data structure with a LIFO discipline. Also closures can be of different sizes.

3. The language Pascal allows a limited facility for defining procedures that can take a procedure as a parameter. 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]

Two pieces of information need to be passed: (1) the address of the code for the argument procedure R; (2) a pointer to the latest frame of the parent of R (at time of calling). Note that this frame exists since P can only know of the names of its children or those of its ancestors and all these procedures have an instance on stack. Note: subtly, when R is eventually called, the pointer may not point to the last instance of the parent on stack, since later instances may have been placed on stack afterwards (Q calls P passing R as an argument!).

4. Consider a sequential imperative object-oriented language such as C++, with no nested procedure/method declarations but *with dynamic dispatch* (virtual functions in C++). Suppose in a method of class A, a method f is invoked on an object b expected to be of class B, but which may actually be of a subclass C of B. Explain what information should be placed on the stack frame, and how the correct stack frame (of C not of B) is determined.

What's on top of stack initially is a frame corresponding to the layout of an object of class A. Then a reference or a frame of object b is pushed onto stack. Also pushed are arguments to f. The shape of object b's frame is given by its actual class C. In each object's frame, there is a pointer (typically at the first position) to a table ("some of you call it "vtable") that gives for each method, the code to be executed. This table is used to determine which version of f to run ($B.f$ or $C.f$). The appropriate code is invoked, pushing onto stack the return address etc.