

Questions for the PhD Comprehensive Examination:
Programming Languages and Systems

Department of Computer Science, University of New Mexico

18 August 2003

The exam has three sections. Section 1, Programming Language Essentials, has four questions, of which you must answer two. Section 2, Programming Language Implementation, has four questions, of which you must answer two. Section 3, Programming Language Theory, has two questions, of which you must answer one.

Submit a front page indicating which problems you are solving.

Each problem solution should be on a separate sheet (or sheets) of paper, clearly marked with problem number, sheet number, and your name. Do not write on both sides of a sheet. You may type-set or write by hand. In the latter case please write legibly.

The exam is closed-book, closed-notes, closed-computer (except for type-setting). You must not discuss the questions with anyone but the proctor.

You may ask the proctor for clarification on any question; if clarification is provided, it will be provided equally to all students taking the test.

You have 8 hours to complete the exam, including any breaks for meals.

Breaks for meals will be agreed at the start of the exam.

Exam questions are confidential and must not be divulged to third parties.

Contents

1	Programming Language Essentials	3
1.1	Programming techniques	4
1.2	Programming techniques	5
1.3	Data structures and types	6
1.4	Elementary λ -calculus	7
2	Programming Language Implementation	8
2.1	Compiler optimizations	9
2.2	Unification	10
2.3	Virtual machines	11
2.4	Functional languages	12
3	Programming Language Theory	13
3.1	Fixed points	14
3.2	Semantics	15

1 Programming Language Essentials

1.1 Programming techniques

The Scheme function `append` takes two lists and returns the list consisting of the elements of the first list concatenated with the elements of the second. For example,

```
(append '(1 2 3) '(4 5 6))
```

returns the list

```
(1 2 3 4 5 6).
```

Give a purely functional and entirely tail-recursive definition (in Scheme) for the function `append`. Note: Any helper functions you define must be purely functional and entirely tail-recursive also.

1.2 Programming techniques

The following is SML code of a scanner for a simple subset of Scheme:

```
fun scan s : token list =
  let
    val slist = String.explode s
    fun f [] = []
      | f (" " :: t) = f t
      | f ("\t" :: t) = f t
      | f ("\n" :: t) = f t
      | f ("(" :: t) = LPar :: f t
      | f (")" :: t) = RPar :: f t
      | f (h :: t) = g ([h], t)
    and g (a, []) = [Symbol (String.implode (rev a))]
      | g (a, " " :: t) = Symbol (String.implode (rev a)) :: f t
      | g (a, "\t" :: t) = Symbol (String.implode (rev a)) :: f t
      | g (a, "\n" :: t) = Symbol (String.implode (rev a)) :: f t
      | g (a, "(" :: t) = Symbol (String.implode (rev a)) :: LPar :: f t
      | g (a, ")" :: t) = Symbol (String.implode (rev a)) :: RPar :: f t
      | g (a, h :: t) = g (h :: a, t)
  in
    f slist
  end
```

Here is an example showing how the scanner may be invoked:

```
val schemeSource = ...omitted...
val tokens = scan schemeSource
```

1.2.1

What is the type of `scan`? What is the type of `slist`? What is the type of `f`? What is the type of `g`?

1.2.2

What can be inferred about the type `token`?

1.2.3

Write a formal description of the lexical classes recognized by the scanner.

1.2.4

Write a version of the code in continuation-passing style. Be sure to write both the scanning function and an example of its invocation.

1.3 Data structures and types

1.3.1

Give the *type* and *value* of the following SML expression:

```
let
  datatype 'a t = A of 'a t list
  val a = A [A [A [], A []], A [A [], A []]]
  val m = foldr (fn (x,y) => if x > y then x else y) 0
  val rec d = fn A c => 1 + m (map d c)
in
  d a
end
```

1.3.2

Give at least three different signatures that the following SML structure S matches:

```
structure S =
struct
  datatype 'a t = A of 'a t list
  val a = A [A [A [], A []], A [A [], A []]]
  val m = foldr (fn (x,y) => if x > y then x else y) 0
  val rec d = fn A c => 1 + m (map d c)
end
```

1.3.3

Using SML, we define binary trees as follows:

```
datatype t = Leaf | Node of t * t
```

A path from the root to any subtree consists of a series of instructions to go left or right, which can be represented using another datatype:

```
datatype p = Left of p | Right of p | This
```

where the path `This` denotes the whole tree. Given some tree, we would like to find all paths that denote existing subtrees of the tree. Write an SML function `allpaths: t -> p list` to do so.

1.4 Elementary λ -calculus

Reduce term to normal form, or show it has none:

$$\mathbf{S}_1 \mathbf{S}_2 \cdots \mathbf{S}_n,$$

where $\mathbf{S}_i \triangleq \lambda xyz.xz(yz)$, and $n \in \mathbb{N}$.

2 Programming Language Implementation

2.1 Compiler optimizations

Describe precisely what each of the following compiler optimizations accomplishes and how it is implemented; provide an example showing each in action:

- constant propagation
- common subexpression elimination
- partial redundancy elimination

Compare these three compiler optimizations and discuss their relationship.

2.2 Unification

Unification plays a role in the definition and/or implementation of logic programming languages, and in the implementation of typed functional programming languages.

2.2.1

Describe precisely the role unification plays in these two contexts.

2.2.2

Define the unification problem, i.e., the interface to a unification algorithm.

2.2.3

Give one unification algorithm, i.e., an implementation of the interface. Efficiency is not a concern.

2.2.4

We sometimes desire unification to handle cyclic terms. What role do cyclic terms play in the two contexts? How would you solve the unification problem with cyclic terms?

2.3 Virtual machines

Suppose that the standard bytecode repertoire of Java (as defined in the Java Virtual Machine Specification) is augmented with new bytecodes **save** and **restore**. They are described as follows:

save creates a snapshot of the current state of the heap and pushes a special “save” object reference, representing this snapshot, on the operand stack.

restore pops a “save” object reference from the operand stack and resets the heap to the state represented by that “save” object, in other words, the state at the time the corresponding **save** was executed.

2.3.1

Define precisely the conditions under which the **restore** operation is legal.

2.3.2

How can **save** and **restore** be implemented? Propose one or more approaches. Discuss the efficiency of your approach(es), considering both the cost of the **save** and **restore** operations and any overhead that is imposed upon other operations. If you are making any *assumptions* about the structure of the original Java virtual machine that you are extending to accommodate **save** and **restore**, state those assumptions explicitly.

In the discussion of efficiency, purely qualitative statements will not be deemed satisfactory. Instead, use precise quantitative statements. If appropriate to your approach, provide worst-case, average-case, or amortized-cost analyses.

2.4 Functional languages

The following fragment of SML source:

```
let
  fun square x = x * x
  fun map f [] = []
    | map f (head::tail) = f head :: map f tail
in
  let
    val g = map square
  in
    g [1,2,3]
  end
end : int list
```

may be transformed by the compiler into an intermediate (“de-sugared”) form that looks as follows (written again in source SML syntax):

```
let
  val square = fn x => x * x
  val rec map = fn f =>
    let
      val h = fn l =>
        if l = nil then
          nil
        else
          f (hd l) :: (map f) (tl l)
        in
          h
        end
    in
      let
        val g = map square
      in
        g [1,2,3]
      end
    end
end
```

Define *closure conversion* and show the result of closure conversion when applied to the intermediate form above.

3 Programming Language Theory

3.1 Fixed points

3.1.1

Does the following functional defined over functions over naturals (nonnegative integers) have a least fixed point? If so, compute it and prove that it is indeed the least fixed point (i.e., it is minimal and unique). Illustrate the key steps of your derivation.

$$F(f)(x) = \text{if } (x < 3) \text{ then } x \text{ else } f(x - 2) + f(f(x - 1)).$$

How many other fixed points does this functional have? List them and show that they are indeed fixed points. Also, prove that each of these fixed points is *bigger than* or *equal to* the least fixed point. Precisely define the ordering on fixed points.

3.1.2

The following code attempts to compute the quotient and remainder of a nonnegative number when divided by another positive number. However, it has some **minor bugs**. In this problem, you have to use the axiomatic semantics method to find these bugs.

Operations on numbers, such as \geq , $+$, $-$, $*$, *div_2*, etc., are assumed to be built in; *div_2* stands for a function that gives the quotient of its argument when divided by 2.

Illustrate the use of the axiomatic approach (attempting to find the strongest invariants I_1, I_2, I_3 missing from the program) from the input-output specification of the code for finding these mistakes. In particular you should show how attempts to prove invariant conditions resulted in finding the mistakes, and then show how these mistakes can be fixed. Give the appropriate formulas for I_1, I_2, I_3 , and show a proof of correctness of the corrected code. It will be very helpful first to informally describe the algorithm being used.

```
{P : x ≥ 0, y > 0}
z := y; n := 0;
{I1}
while x > z do
  z := z * 2; n := n + 1;
endwhile;
b := x; a := 0;
{I2}
while n ≠ 0 do
  z := div_2(z); n := n - 1; a := a * 2;
{I3}
  if b > z then a := a + 1; b := b - z;
endwhile;
{Q : x = a * y + b ∧ y > b ≥ 0}
```

Hint: Try the code on some test data to get some feel for the algorithm as well as possible bugs.

3.2 Semantics

Extend a simple imperative language in which expression evaluation is free of side-effects to include another kind of assignment statement, written as

$$I ::= J$$

to mean that I and J refer to the same location. Later if I is assigned a value, say 3, then the contents of the location corresponding to J is also 3. For simplicity, assume that only arithmetic expressions are allowed in the language.

How would this change the denotational semantics of expressions and statements in the programming language? Discuss the necessary changes—semantic domains, the meaning function for expressions and statements, etc.

What would a proof rule specifying its axiomatic semantics look like?

Caution and Special Instructions: Since the first part of this problem was discussed in detail in a review session given by Kapur, the answer to this problem is expected to be precise, rigorous, and formal. Any hand-waving answer will not get credit. The grade will be based on how rigorously and precisely one is able to think about such a problem when one knows the answer at least informally, and present a solution.