Programming Language Qualifying Exam

January 13, 1998

There are 5 problems and 5 pages to this exam. Please take a minute to make sure that you have all of the pages, and skim each question before beginning the exam. This is a closed book exam. Please write all of your answers in an exam booklet. Do *not* put your name on the booklet but do put your id number.

1. **Parsing (30 points):**

   This questions tests your knowledge of parsing theory and practice.

   The syntax of a simple imperative, while-loop language is given by the following BNF grammar:

(Id)	X		
(Exp)	E	$::=$	$X \mid i \mid E_1 + E_2 \mid E_1 * E_2 \mid (E)$
(Com)	C	$::=$	$X{:=}E \mid C_1;C_2 \mid$ `if0` E `then` C_1 `else` $C_2 \mid$ `while0` E `do` $C \mid \{C\}$

   The grammar above is suitable only as the *abstract* syntax for the language because it is ambiguous in many places.

   (a) (10 points) Modify the grammar to serve as a *concrete* syntax by making the grammar unambiguous. However, the grammar should accept the same language. As usual "$*$" should have higher precedence than "+".

   (b) (20 points) Using your favorite language (*e.g.*, C, ML, Java), write a recursive-descent parser for your concrete syntax for *commands* only. The parser should return true if the input stream yields a valid command (terminated by an end of stream) and false otherwise. You may assume the following built in functions:

   - `parse_exp`: takes no arguments and returns true if the current input stream yields a valid expression and false otherwise. Leaves the rest of the input stream intact.
   - `lookahead`: takes no arguments and returns an integer code representing the next token in the input stream. Does *not* remove the token from the input stream. The possible return values and the tokens they represent are as follows:

0	an identifier	6	"`while0`"
1	assignment operator (":=")	7	"`do`"
2	"`if0`"	8	left brace ("{")
3	"`then`"	9	right brace ("}")
4	"`else`"	10	end of stream ("")
5	semicolon (";")		

   - `advance`: takes no arguments and returns an integer code representing the next token, *removing* the token from the input stream.

2. **Language Implementation (15 points):**

   (a) (5 points) In the compilation of block-structured, lexically-scoped programming languages, the implementation of (recursive) procedures and functions is a delicate issue. Briefly explain the role of static and dynamic links in a conventional implementation of activation records for Pascal-like languages.

   (b) (5 points) C compilers do not need to use static links. Why?

   (c) (5 points) Automatic memory management (garbage collection) is provided by a variety of language implementations. List two garbage collection strategies and in a sentence or two, discuss their relative strengths and weaknesses.

3. **λ-Calculus and Semantics (20 points):**

This question tests your knowledge of λ-calculus, types, and semantics of imperative programming languages.

We begin by giving a denotational semantics for the commands of the language in Part 1. Assume we have an auxiliary function $\mathcal{E}[\![\,]\!] : \mathit{Exp} \to \mathit{State} \to \mathit{Int}$ where $\mathit{State} = \mathit{Id} \to \mathit{Int}$ for assigning meaning to expressions.

To model the behavior of commands, we define a function $\mathcal{C}[\![\,]\!]$ using typed λ-calculus notation in *continuation-passing style*:

$$\begin{array}{lcl}
\mathcal{C}[\![\{C\}]\!] & = & \mathcal{C}[\![C]\!] \\
\mathcal{C}[\![X{:=}E]\!] & = & \lambda\kappa.\lambda\sigma.\kappa\,(\lambda y.\text{if } y = X \text{ then } \mathcal{E}[\![E]\!]\,\sigma \text{ else } \sigma\, y) \\
\mathcal{C}[\![C_1;C_2]\!] & = & \lambda\kappa.\mathcal{C}[\![C_1]\!]\,(\mathcal{C}[\![C_2]\!]\,\kappa) \\
\mathcal{C}[\![\texttt{if0 } E \texttt{ then } C_1 \texttt{ else } C_2]\!] & = & \lambda\kappa.\lambda\sigma.\text{if } \mathcal{E}[\![E]\!]\,\sigma = 0 \text{ then } \mathcal{C}[\![C_1]\!]\,\kappa\,\sigma \text{ else } \mathcal{C}[\![C_2]\!]\,\kappa\,\sigma \\
\mathcal{C}[\![\texttt{while0 } E \texttt{ do } C]\!] & = & \lambda\kappa.\text{Fix}(\lambda f.\lambda\sigma.\text{if } \mathcal{E}[\![E]\!]\,\sigma = 0 \text{ then } \kappa\,\sigma \text{ else } \mathcal{C}[\![C]\!]\,f\,\sigma)
\end{array}$$

where Fix is the usual fixed-point combinator. The type of $\mathcal{C}[\![\,]\!]$ is $\mathit{Com} \to (\mathit{State} \to \mathit{State}) \to \mathit{State} \to \mathit{State}$.

Given an input state σ_0, the output state calculated by a program C is given by evaluating $\mathcal{C}[\![C]\!]\,(\lambda\sigma.\sigma)\,\sigma_0$. That is, we start the computation by giving it the identity function as the initial continuation.

We wish to add a `break` command to the language with the intended semantics that a `break` should cause control to transfer to the command following the nearest enclosing `while0`-loop (if any). If there is no enclosing `while0`-loop, or there is no following command, the command should terminate the program in the current state.

You are to complete the denotational semantics (below) for the new language by adding definitions for `while0` and `break`. The key idea is to add another continuation argument to each clause so that the semantics has type $\mathit{Com} \to (\mathit{State} \to \mathit{State}) \to (\mathit{State} \to \mathit{State}) \to \mathit{State} \to \mathit{State}$ and the output state for command C and input state σ_0 is obtained by evaluating $\mathcal{C}[\![C]\!]\,(\lambda\sigma.\sigma)(\lambda\sigma.\sigma)\,\sigma_0$.

$$\begin{array}{lcl}
\mathcal{C}[\![\{C\}]\!] & = & \mathcal{C}[\![C]\!] \\
\mathcal{C}[\![X{:=}E]\!] & = & \lambda\kappa_b.\lambda\kappa.\lambda\sigma.\kappa\,(\lambda y.\text{if } y = X \text{ then } \mathcal{E}[\![E]\!]\,\sigma \text{ else } \sigma\, y) \\
\mathcal{C}[\![C_1;C_2]\!] & = & \lambda\kappa_b.\lambda\kappa.(\mathcal{C}[\![C_1]\!]\,\kappa_b)(\mathcal{C}[\![C_2]\!]\,\kappa_b\,\kappa) \\
\mathcal{C}[\![\texttt{if0 } E \texttt{ then } C_1 \texttt{ else } C_2]\!] & = & \lambda\kappa_b.\lambda\kappa.\lambda\sigma.\text{if } \mathcal{E}[\![E]\!]\,\sigma = 0 \text{ then } \mathcal{C}[\![C_1]\!]\,\kappa_b\,\kappa\,\sigma \text{ else } \mathcal{C}[\![C_2]\!]\,\kappa_b\,\kappa\,\sigma \\
\mathcal{C}[\![\texttt{while0 } E \texttt{ do } C]\!] & = & ??? \\
\mathcal{C}[\![\texttt{break}]\!] & = & ???
\end{array}$$

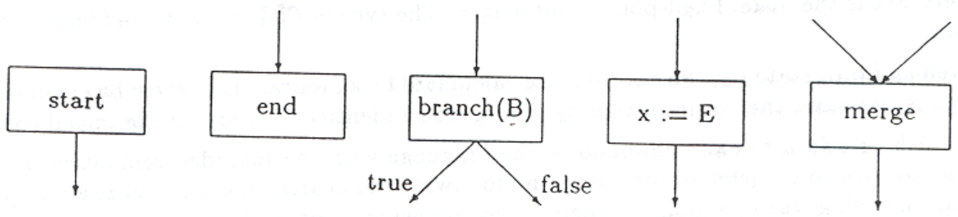
4. **Dataflow Analysis (45 points):**

(a) (20 points) Let D be a partially ordered set with a least element and consider a montonic function $f : D \to D$.

i. Suppose D is *finite*. Does f necessarily have a least fixed-point? Explain your answer by either proving the assertion or giving a counter example.

ii. Suppose D is *infinite*. Does f necessarily have a least fixed-point? Explain your answer by either proving the assertion or giving a counter example.

(b) (25 points) An expression E is said to be *very busy* at a node p in a control-flow graph if no matter what path is taken from p, the expression E is evaluated before any of its operands are [re]defined. Formulate the problem of finding all very busy expressions in a program using dataflow equations. You may assume that the control-flow graph has the following types of nodes:



assume array sorted says Greg

5. **Programming Methodology (30 points):**

This question concerns the methodology for developing a loop. Given is an array $b[0..n]$, where $0 < n$, and a variable x that satisfy the relation

$$P : b[0] > x \geq b[n] \quad .$$

Desired is an algorithm that stores in i a value to establish

$$R : 0 \leq i < n \;\wedge\; b[i] > x \geq b[i+1].$$

Develop an iterative algorithm for this task, using the techniques described in the text *The Science of Programming*; Describe your method for finding the loop invariant. The invariant should lead to an $O(\log n)$ algorithm —which is binary search. The more work you show, the easier it will be to give partial credit. Your grade to a large extent depends on suitable application of the methods for developing loops, and not only on the final algorithm.