Theory of Programming Languages

Phd. Qualifying Exam (January 9, 2015)

Answer two questions out of four. Only top two scores will be considered.

Problem 1. Static scoping and dynamic scoping:

This question concerns programs written in a simple language *L*, a variant of Pascal(or C). A program in *L* consists of a main program containing a body, some variable declarations and a collection of procedure definitions, each of which can introduce local variables. Blocks can introduce local variables as you know. Nested procedures are not permitted in *L*, there are no procedure valued parameters, and the languages is not strongly typed. Recursion among the procedures is permitted.

Let *P* be a program in language *L*. Define $M_{\text{stat}}[P]$ to be the result (for example, the final values of variables) of execution of *P* using static scope rules (also known as lexical scoping,) and $M_{\text{dyn}}[P]$ to be the result of execution of *P* using dynamic scope rules.

(a) Explain static scope and dynamic scope.

(b) Give a program *P* for which $M_{\text{stat}}[P] \neq M_{\text{dyn}}[P]$, i.e. which gives different results under different scope rules.

A preprocessor *G* for *L* translates each program *P* in *L* to a (possibly different) program *G*(*P*) in *L*. For our purposes, a preprocessor can make at most a constant number of passes over the input program.

(c) Describe a preprocessor *G* that for all programs *P* in *L* satisfies

$M_{\text{dyn}}[G(P)] = M_{\text{stat}}[P]$.

Give an example with the program in (b).

State your argument why your procedure will work.

(d) Is it necessary to modify the preprocessor *G* if nested procedures **are permitted** in *L*? Why or why not?

Problem 2

Different programming languages provide different degrees of block structure, in which the scope of an identifier can be limited to a portion of the program text. Blocks are here defined always to be properly nested, i.e. either one block is completely contained within another or the two blocks are disjoint.

For example, Pascal allows one procedure definition to be nested within another; it also allows one **record** to be nested within another. C allows nesting of a record(**struct**), but not of a procedure. Fortran provides disjoint blocks (e.g. **common** and **subroutine**), but no nesting.

Consider a language with block structure, and a compiler for the language. For each of the five following assertions state to what extent you agree or disagree with it, and why. Where appropriate, be careful to consider both lifetime (temporal duration) and scope (spatial extent of visibility).

a. Block structure makes it easier to implement a variable that continues to live, keeping its value, between invocations of the procedure in which it is declared. (An example application is the seed for a pseudo-random number generator. Example constructs are **own** variables in Algol which introduced the concept and **static** variables in PL/I and C.)

b. Block structure makes a program harder to understand, because it permits identifiers to be reused for different purposes.

c. Block structure permits modularization, hence faster recompilation and easier error identification.

d. Consider a **goto** to a label in an enclosing procedure block. The code compiled for the **goto** is not appreciably more complex than the code compiled for a **goto** to a label in the same block.

e. Block structure reduces the main storage space requirement at both compilation time and execution time, permitting the use to run with a smaller machine.

Problem 3

Consider the expressions defined by the following abstract syntax

$$e \ ::= \ n \mid x \mid e + e \mid e \times e$$

where "+" and "×" are addition and multiplication, respectively. Note that "*x*" is a variable.

(1) Define denotational semantics **or** natural semantics for *e* precisely. Don't forget to define domains of your functions and values.

(2) We will implement this language E with <*S,E,C*>-machine. The <*S,E,C*>-machine is an abstract machine. *S* is a stack of values (ordered sequence). *E* is an environment (function : Variable → Value). *C* is a command sequence defined as following:

C	$\rightarrow$	add $\cdot$ C	; pop and add two values and push back the result
	$\mid$	mul $\cdot$ C	; pop and multiply two values and push back the result
	$\mid$	push(x) $\cdot$ C	; push the value of x on the stack
	$\mid$	push(n) $\cdot$ C	; push the integer n on the stack
	$\mid$	ε	; empty command

Define the transition semantics for command C.

(3) Define the compilation rules from programs e to command sequences C .

Problem 4

Consider a language with expressions defined by the following abstract syntax

$$e \quad ::= \quad 0 \mid 3 \mid x \mid e + e \mid e \times e \mid \text{let } x = e \text{ in } e$$

where "+" and "×" are addition and multiplication, respectively. Note that "x" is a variable. We want to design a type system (i.e. to define inference rules) to check an expression e is the multiple of 3 with static semantics without evaluating the value of e with dynamic semantics. For example, you can clearly say that $0+3$, $3*3+$**3** and let $x=3$ in $x+3$ are the multiple of 3 without evaluating expressions.

(1) Design your type system for e precisely.

(2) What can your type system guarantee for programmers?