UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

CS 164 **P. N. Hilfinger**
Fall 2006

CS 164: Final Examination (updated)

Name: ______________________________ Login: ________

You have three hours to complete this test. Please put your login on each sheet, as indicated, in case pages get separated. Answer all questions in the space provided on the exam paper. Show all work (but be sure to indicate your answers clearly.) The exam is worth a total of 55+ points (out of the total of 200), distributed as indicated on the individual questions.

You may use any notes or books you please, but not computers, cell phones, etc.—anything inanimate and unresponsive. We suggest that you read all questions before trying to answer any of them and work first on those about which you feel most confident.

You should have 7 problems on 12 pages.

1. ________/12	5. ________/12
2. ________/12	6. ________/
3. ________/3	7. ________/12
4. ________/4	TOT ________/55

1. [12 points] For each of the following possible modifications to a fully functional Pyth system, tell which components of the compiler and run-time system would have to be modified: lexical analyzer, parser and tree-generator, static semantic analyzer, code generator, and run-time libraries. In each case, indicate a *minimal* set of modules from this list that would have to be changed, and indicate *very briefly* what change would be needed. When you have a choice of two equal-sized sets of modules that might reasonably be changed, choose the one that makes for the simplest change or whose modules appear earlier in the list (e.g., prefer changing the lexical analyzer to the parser, if either change would be about equally difficult).

a. Require that, as in Java, actual parameters in a call must have static types that are subtypes of the formal parameters

b. Require that when the type declared for a formal parameter is a "primitive type"—Int, Float, or Bool—the actual parameter's value may not be None. Likewise, require that when the type declared for any kind of variable is a primitive type, it may not be assigned the value None (although it is still initialized to None).

c. Introduce a do...while construct, as in C:

```
do:
   print x
   x += 1
while x < N
```

This executes its body first and then performs the test.

d. Introduce lambda expressions. That is, allow expressions (as in full Python) such as

```
sum = reduce (lambda x,y: x+y, L)
```

e. Change the allowed escape sequences in string literals.

f. Infer the types of local variables. A variable without a type declaration takes as its static type the static type of the right-hand side expression in the first assignment to it in the program text (the one that's written first, not necessarily the one that executes first).

2. [12 points] Consider this simple language in which all quantities are integers:

program $\rightarrow$ *stmts*	{ *#1* }
stmts $\rightarrow$	
ϵ	{ *#2* }
| *stmts stmt* **;**	{ *#3* }
stmt $\rightarrow$	
INTLIT	{ *#4* }
| **pop**	{ *#5* }
| *if-part stmts else-part stmts* **fi**	{ *#6* }
| *OP*	{ *#7* }
| **read**	{ *#8* }
if-part $\rightarrow$ **if**	{ *#9* }
else-part $\rightarrow$ **else**	{ *#10* }

This language describes a simple stack-based calculator. There is an expression stack of integer values, which is initially empty. Each occurrence of *INTLIT* pushes the value denoted by that literal onto the stack. Each occurrence of **pop** pops off the top value of the stack (and is erroneous if the stack is empty). Each occurrence of an operator (*OP*) pops the top two values on the stack (say x_1 and x_0, where x_0 is the top value), performs the operation $x_1 \oplus x_0$, where $\oplus$ is the operator, and pushes the result on the stack. Operators are erroneous when the stack has fewer than two elements on it. Each occurrence of **read** reads a value from the user and pushes it on the stack (as a result, in general you cannot know at compilation time exactly what values are on the stack). Finally, the **if** construct pops a value off the stack and compares it to 0. If it is 0, the program executes the statements after **else**, and otherwise the statements before **else**. It is an error to execute **if** on an empty stack or for the two statement lists (before and after **else**) to leave different numbers of items on the stack.

The lexical analyzer supplies semantic values for integer literals (*INTLIT*), and for operators (*OP*). The semantic value of *OP* is simply its text (a string), which can be one of `"+"`, `"-"`, `"*"`, or `"/"`. The semantic value of an *INTLIT* is its denoted value (an integer).

The problem is to build a one-pass compiler for this language into virtual machine code. The virtual machine has an infinite supply of registers and the following instructions:

- r_1 `:=` $g_2 \ \oplus \ g_3$, where $\oplus$ is one of `+`, `-`, `*`, or `/`.
- r_1 `:=` g_2
- `jz` r_1`,`L, which jumps to L if $r_1 = 0$.
- `jmp` L, which jumps to L.
- r_1 `:= call read`, which calls a subprogram that reads a value from the user and puts the result in r_1.

Here, r_1, r_2, …refer to a virtual registers (they don't have to be distinct); g_1, g_2 are each either a register or an immediate integer constant (`$0`, `$-1`, etc.); and L is a statement label. We will assume that all intermediate results and all variables will be stored in virtual registers. There is no way to create a stack in memory using these instructions (since there is no way to access memory).

Fill in actions for the grammar that produce this virtual machine language. Your actions may use the following functions:

- `label()` returns a new label.
- `reg(`k`)` returns the k^{th} virtual register.

Pseudo-code is fine. You may wish to introduce auxiliary functions that you can call from the semantic actions. To indicate the output of an instruction, feel free to write things like

```
aReg = reg (i); aLabel = label ();
N = some integer;
...
emit ‘je aReg, $N, aLabel’
```

The final stack must have at least one value. The program is supposed to leave the final result of the computation (i.e., the top of the stack) in register `reg(0)`. The compiler should catch all cases where the stack is handled illegally so that no run-time checks relating to the stack are required to find errors.

You may choose any semantic values you want for nonterminals of your grammar (i.e., the values of $\$n$ in Bison notation).

Fill in the actions here (label them `#1, #2, ..., #10`).

More space on next page.

Continue your solution here, if needed.

3. [3 points] Here are a few type rules from a Pyth-like language. In all these rule templates, N is to be replaced by any integer constant, V_i by any identifier, E_i by any expression, and T_i by any type. The clause $T_1 \leq T_2$ means "type T_1 is a subtype of T_2."

$$\frac{}{O \vdash N : \text{Int}} \qquad \frac{O \vdash E_0 : T_0 \rightarrow T_1, \quad O \vdash E_1 : T_0}{O \vdash E_0(E_1) : T_1} \qquad \frac{O \vdash E_0 : T_0, \quad T_0 \leq T_1}{O \vdash E_0 : T_1}$$

We add a lambda expression to this language and give it the following rule:

$$\frac{O \vdash E_0 : T_0}{O \vdash \lambda V_1.E_0 : \text{Any} \rightarrow T_0}$$

The intent is to say that the formal parameter of a lambda expression has static type Any, and that it represents a function that takes type Any and returns a value having the static type of E_0, its body. However, this last rule is flawed. Which of the following problems does it have? Indicate all that apply.

a. The type judgment $O \vdash (\lambda x.3)(k) : \text{Int}$ is intended to be correct but *can't* be proven.

b. The type judgment $O \vdash (\lambda x.3)(k) : \text{Int}$ is intended to be incorrect but *can* be proven.

c. The type judgment $O \vdash (\lambda x.x)(3) : \text{Int}$ is intended to be correct but can't be proven.

d. The type judgment $O \vdash (\lambda x.x)(3) : \text{Int}$ is intended to be incorrect but can be proven.

e. The type judgment $0[\text{String}/x] \vdash (\lambda x.x)(3) : \text{String}$ is intended to be correct, but can't be proven.

f. The type judgment $0[\text{String}/x] \vdash (\lambda x.x)(3) : \text{String}$ is intended to be incorrect, but can be proven.

4. [4 points] Java implicitly assigns default values to all local variables before executing the function that contains them. However, Java also requires that at each place where a local variable is used, it must first have been *definitely assigned*—that it must have been *explicitly* assigned to by a previously executed statement (one that the programmer actually wrote) in the same function. Describe a way to use global flow analysis to check this property of a program. Give sufficient detail to convince us that you know what you're talking about. We're interested in a reasonably high-level description; you can feel free to use any of the analyses you've encountered in this course as part of your solution (i.e., without having to regurgitate their details). However, we are looking for a solution that involves the kinds of analyses we've talked about; just saying "you look at all possible paths from the beginning of the function to each use of x and check that there's an assignment to x in there somewhere" is not going to cut it.

5. [12 points] Consider the following leftmost derivation. The terminal symbols are '`*`', '`/`', '`(`', '`)`', and '`!`'. Lower-case letters denote non-terminals.

```
q
t
r
r s
r s s
s s s
* s s
* ( q ) s
* ( / * ! q ) s
* ( / * ! t ) s
* ( / * ! r / * ! q ) s
* ( / * ! s / * ! q ) s
* ( / * ! * / * ! q ) s
* ( / * ! * / * ! t ) s
* ( / * ! * / * ! r ) s
* ( / * ! * / * ! s ) s
* ( / * ! * / * ! * ) s
* ( / * ! * / * ! * ) *
```

a. [3 points] What is the parse tree corresponding to this derivation?

continued

b. [3 points] Give the first 10 steps of the corresponding rightmost derivation.

c. [3 points] Reconstruct as much of the BNF grammar corresponding to this parse as possible.

continued

d. [1 point] Here are the entries for a few of the states in a shift-reduce table for this grammar. State 0 is the start state, and `$end` denotes the end of file.

State	'*'	'/'	'('	')'	'!'	`$end`	q	r	s	t
0	s2	s1	s3				s4	s6	s7	s5
1	s8									
2	r7	r7	r7	r7	r7	r7				
3	s2	s1	s3				s9	s6	s7	s5
4						acc				
5				r1		r1				
6	s2	s11	s3	r3		r3			s12	
7	r6	r6	r6	r6		r6				
8					s13					
9				s14						
11	s15									
12	r5	r5	r5	r5		r5				
13	s2	s1	s3				s16	s6	s7	s5
14	r8	r8	r8	r8		r8				
15					s17					
16				r2		r2				
17	s2	s1	s3				s18	s6	s7	s5
18				r4		r4				

Shift or goto entries are denoted sn and reducing by rule number k by rk; 'acc' means "accept". Show what symbols could be on the parsing stack that would cause state 16 to be the state of the top of the stack.

e. [1 point] Referring again to part d, what reduction must r2 be? After taking that reduction, what will be the next top state on the stack?

f. [1 point] If the parse tree that results from parsing a program with this grammar contains N nodes and the input program contains M characters, what is the asymptotic running time of the parse?

6. [1 point] Where is Olympus Mons (the place that is, not the blogger)?

7. [12 points] For each of the following questions about the project, provide a short, succinct answer.

a. What is the point of having an exemplar object of type Int, given that Int values don't go on the heap?

b. Suppose that Pyth did not require you to give `x` a static type that defined instance variable `a` before you could use `x.a` in a program. What effect would this have on the implementation?

c. What is the purpose `__registerVar` and `__registerObj`? What would happen if your code didn't call them properly, and why?

d. If my program declares `x: Int`, why can't I just store the variable `x` as a 4-byte quantity? If I have to pass it to something that expects type Any, can't I just create the usual 16-byte value on the stack by pushing the pointer to `__obj_Int` and the integer value from the variable?