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

CS 164 **P. N. Hilfinger**
Spring 2008

CS 164: Final Examination (with corrections)

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 50+ 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 8 problems on 15 pages.

1. _____/10
2. _____/5
3. _____/5
4. _____/
5. _____/5
6. _____/10
7. _____/5
8. _____/10

TOT _____/50

1. [10 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, standard prelude, and run-time libraries. In each case, indicate a *minimal* set of components 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. When using character sets that support it, allow programmers to write $\leq$ in place of `<=`, $\geq$ in place of `>=`, and $\rightarrow$ in place of `->`.

b. Introduce a statement "`until` *E*: *suite*" that is just like the `while` construct, but loops until *E* becomes true rather than false.

c. Cause a fatal error when an integer multiplication overflows.

d. Allow **for** statements to take multiple variables, as in full Python:

```
for x, y in L: print x, y
```

e. Allow a selection expression `a.x` even when `a` has not been given a type declaration, as long as all assignments to `a` are from expressions whose static type is a class that defines instance variable `x`.

2. [5 points] Consider the language described by this grammar:

expr →
    term
  | *expr OP term*
term →
    primary
  | *term primary*
primary →
    ID
  | *'(' expr ')'*
  | *'λ' VAR '.' '(' expr)'*

Here, everything in single quotes or capital letters is a terminal symbol, and everything else is a non-terminal. Fill in a recursive-descent parser for this language. The function `next()` returns the lexer's current token, and `scan(`T`)` checks that `next()` is T (causing an error if not) and then reads the next token in the input (thus changing the value of `next()`). The lexer returns the token *EOF* when it runs out of tokens.

The grammar as written is not quite suitable for direct conversion to recursive descent. Feel free to add additional functions or take other steps to correct for this. We are interested only in recognizing correct programs, not in translating them

```
def expr ():
```

```
def term ():
```

```
def primary ():
```

Continue on next page, if needed.

Continue here, if needed.

3. [5 points] Consider the following partial grammar for a typical imperative programming language:

program → *stmts*	{ *#1* }
stmts →	
stmt ';'	{ *#2* }
| *stmts stmt* ';'	{ *#3* }
stmt →	
if *expr* **then** *stmts* **else** *stmts* **fi**	{ *#4* }
| **for** *ID* = *INTLIT* **to** *INTLIT* **do** *stmts* **od**	{ *#5* }
| *ID* = *expr*	{ *#6* }
| **pass**	{ *#7* }

Assume that the nonterminal *expr* is defined elsewhere. The *INTLIT* token has an integer as its semantic value (supplied by the lexer). As you might expect, the construct

for $i = N_1$ **to** N_2 **do** S **od**

repeatedly executes S, with i set in turn to each value between N_1 and N_2, inclusive. The other kinds of statements have their obvious meanings.

The problem is to fill in the actions for this grammar so that they conservatively estimate the maximum number of assignment statements that the program can execute (not including assignments to the control variable of **for**), and make that the semantic value of the *program* non-terminal. "Conservative" here means worst case, because your analysis will have no knowledge of how **if** tests will go. For example, for the program

```
x = f(2);
for i = 1 to 10 do
   y = 3;
   if x > y then
       y = y+x;
       z = x;
   else
       y = y-1;
   fi;
od;
```

would compute the value 31 for *program* (each pass through the the **for** executes at most 3 statements, the loop executes 10 times, and there is one other assignment before the loop starts.

Fill in actions for the grammar on the next page to do this computation. Use any of the usual notations for writing the semantic actions.

```
#1:

#2:

#3:

#4:

#5:

#6:

#7:
```

4. [1 point] If the intersection of a set, $\mathcal{S}$, of open sets is not open, what can you say about $\mathcal{S}$?

5. [5 points] The Java Virtual Machine executes what is essentially an intermediate form (called "Java bytecode") that assumes a stack machine *and* an unlimited supply of registers used for local variables and parameters. Java interpreters don't "trust" these bytecode files, since they can come from anywhere, and therefore perform certain consistency checks on them before execution (a process called "bytecode verification"). One check has to do with stack consistency. The stack must have a fixed, statically known size at each point in the program. Therefore, intermediate code fragments such as the two examples below are illegal:

```
 if x < y goto L1
 push 3
 push 4
 call(1) f
 goto L2
L1:
 push 5
 push 6
 call(2) g
L2:
```

```
L3:
 if x > 0 goto L4
 push 3
 push 4
 call(1) f
 goto L3
L4:
```

Here, `push` C means "push the constant C on the stack, " and `call(`n`)` h means "call h, popping the top n elements of the stack and passing them as its actual parameters, and leaving the value on the stack upon return." The left-hand example results in two different possible stack sizes at L2, depending on whether the program branches to L1. The right-hand example causes the stack size to increase by two on each iteration through the loop.

Describe how to use global flow analysis to check this property of a bytecode program. That is, show a modification of one of the methods we used in lecture for constant propagation or dead code elimination to solve the problem of computing the size of the stack at each point in a procedure (where '$\top$', meaning inconsistent, is one of the possible "sizes"). We assume that the stack has size 0 at the start of every procedure (that is, we give each procedure its own private stack). We're interested in a reasonably high-level description, so give sufficient detail to convince us that you know what you're talking about.

Continue on next page, if needed.

Continue your answer here, if needed.

6. [10 points] Consider the following derivation.

```
p
p / s
p / e
p / i < e >
p / i < i >
s / i < i >
d / i < i >
i f / i < i >
i < f > / i < i >
i < # f > / i < i >
i < # i > / i < i >
```

a. [1 point] Read top to bottom, is this a leftmost, rightmost, or reverse rightmost derivation?

b. [1 point] Which symbols are terminal symbols and which are non-terminals?

c. [2 points] What is the parse tree corresponding to this derivation?

d. [2 points] Reconstruct as much of the BNF grammar corresponding to this parse as possible.

e. [1 point] Show an alternative parse tree for the same sentence, thus showing that the grammar is ambiguous.

f. [2 points] Here are the entries for the states in a shift-reduce table for this grammar. State 0 is the start state, and `$end` denotes the end of file.

State	'i'	'/'	'<'	'>'	'#'	$end	p	s	d	e	f
0	s1						s2	s3	s4	s5	
1	s6	r9	s7	r9	s8	r9					s9
2											
3	r1	r1	r1	r1	r1	r1					
4	r3	r3	r3	r3	r3	r3					
5	r4	r4	r4	r4	r4	r4					
6	r6	r6	r6	r6	r6	r6					
7	s11		s12		s8					s14	s13
8	s6		s12		s8						s15
9	r5	r5	r5	r5	r5	r5					
10	s1							s16	s4	s5	
11	r6	r6	s17	r6	r6	r6					
12	s6		s12		s8						s13
13				s18							
14				s19							
15	r8	r8	r8	r8	r8	r8					
16	r2	r2	r2	r2	r2	r2					
17	s20									s14	
18	r7	r7	r7	r7	r7	r7					
19	r10	r10	r10	r10	r10	r10					
20	r9	r9	s17	r9	r9	r9					
21											
22	acc	acc	acc	acc	acc	acc					

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 19 to be the state of the top of the stack.

g. [1 point] Referring again to part f, what reduction must r10 be? After taking that reduction, what will be the next top state on the stack?

7. [5 points] In Pyth, we have types `List` and `Tuple` whose elements have static type `Any`. Suppose we wanted to be more specific, and have types `List(`T`)` and `Tuple(`T`)`, meaning a list (or tuple) of items each of which is a T (that is, has type T or some subtype of it).

a. Assume also that we introduce a construct (similar to one in Python) that allows Tuple-valued expressions of this form:

`(` E `for` v `in` L `)`

meaning "the tuple whose elements are computed by evaluating expression E with variable v set to each item in L (a sequence-valued expression) in turn." The scope of variable v is limited to this construct (it is independent of any v declared outside). For example,

```
(2*i for i in xrange (0, 5)) == (0, 2, 4, 6, 8)
(x[1:] for x in ("the", "quick", "brown", "fox")) ==
   ("he", "uick", "rown", "ox")
```

The value of L may be `Tuple(`T`)` or `List(`T`)` (in which case, v will take on values of type T), or `Xrange` (in which case v will take on integer (Int) values).

Provide typing rules for this construct, using the Prolog notation from lecture. We are looking for rules for `typeof(tuplegen(E, V, L), T, Env)`, where `tuplegen(E, V, L)` is the AST for this type of expression. Use $\leq$ to denote the subtyping relation (as in Int $\leq$ Any).

Your rules (combined with rules for other constructs in Pyth), should allow us to conclude that `(2*i for i in xrange(0,5))` is both a `Tuple(Int)` and a `Tuple(Any)`.

b. The typing rule

```
Tuple(X) ≤ Tuple(Y) :- X ≤ Y.
```

(where, again, ≤ is intended to mean "is a subtype of") turns out to work without any problem. However, the analogous rule

```
List(X) ≤ List(Y) :- X ≤ Y.
```

will cause problems. That is, we will eventually run into a little trouble if we allow an assignment `x = E` whenever `x` is declared to have static type `List(A)` and expression `E` has dynamic type `List(B)` where `B` is a subtype of `A` but not the same type. Why is this? Why is it a problem for Lists, but not Tuples?

8. [10 points] For each of the following questions about the project, provide a short, succinct answer.

a. What exactly might go wrong if you failed to initialize the space reserved to hold return values to some valid Pyth value, such as None?

b. We required that all variables on the stack be initialized to valid Pyth values. As it turns out, any values will do, not just None. So why not save time and initialize the entire stack to None values, so that all the variables came pre-initialized? Since the other code in Pyth programs always write only valid Pyth values, wouldn't this guarantee that all variables always have valid values without having to initialize each time? Assume that we know an upper limit to the stack size and that we change Pyth semantics to say that until we assign into a local variable, its value is some arbitrary (but valid) Pyth value.

c. How would I have to change the Pyth language if I did not want the code generator to have to create any function descriptors?

d. If the lexical analyzer did not compute line-number and file information for each token, what would be the effect on the Pyth compiler?

e. Give as many reasons as you can think of as to why we didn't simply have the Pyth parser module from Project 1 call the functions in Assembler directly in the action parts of its rules.