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

CS 164 **P. N. Hilfinger**
Spring 2005

CS 164: Final Examination

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 65+ points (out of the total of 200), distributed as indicated on the individual questions.

You may use any notes, books, or computers you please—anything inanimate. 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 6 problems on 10 pages.

1. ____________/13

2. ____________/15

3. ____________/8

4. ____________/15

5. ____________/

6. ____________/14

TOT ____________/65

1. [13 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 modules that might reasonably be changed, choose the one that makes for the simplest change or is earlier in the list (e.g., prefer changing the lexical analyzer to the parser, if either change would be about equally difficult).

a. Implement an "include *FILE*" directive (as in C and C++), whose effect is to substitute the contents of *FILE* verbatim in place of the directive.

b. Allow for multiple inheritance, as in full Python, where you can write:

```
class A (B, C):
   etc.
```

c. Introduce the 'range' function from Python, allowing one to write

```
for i in range (0, N):
   etc.
```

and have `i` take on the values $0, 1, \ldots, N-1$.

d. Propagate types to lambdas. For example, given the sequence

```
t = lambda x,n: x[n]
t: (List,Int) -> Any
```

cause the static type of the lambda expression to be the same as that of `t`, and the static types of the parameters `x` and `n` to be `List` and `Int` respectively.

e. Instead of passing the static link to functions as the first parameter, always pass it in register EBX.

f. Allow the following usage:

```
class A:
   x = 3
   def foo (self, n):  x = n
```

That is, allow the programmer to refer to `self.x` as `x` (or more generally, to qualify any "bare" mention of an attribute of class `A` in method `foo` with the first parameter of `foo`.)

2. [15 points] Consider this simple language in which all quantities are integers:

program ::=*stmts*	{ *#1* }
stmts ::=	
ϵ	{ *#2* }
| *stmts stmt* **;**	{ *#3* }
stmt ::=	
print *expr*	{ *#4* }
| *var* **:=** *expr*	{ *#5* }
| *while-part* **do** *stmts* **od**	{ *#6* }
while-part ::=**while** *expr*	{ *#6a* }
expr ::=	
var	{ *#7* }
| *intlit*	{ *#8* }
| *expr op expr*	{ *#9* }
| **(** *expr* **)**	{ *#10* }
op ::=+ | - | * | /	

The meanings of all sentences should be obvious. In the case of the **while** construct, the *stmts* repeat until the *expr* is 0. The lexical analyzer supplies semantic values for the terminal symbols *var* and *intlit* (integer literal). The semantic value of an instance of *var* or *op* is simply its text (a string), and that 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
- `call print`, which prints the value in the special register a_0.
- `je` r_1`,`g_2`,`L, which jumps to L if $r_1 = g_2$.
- `jmp` L, which jumps to L.

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.

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()` returns a new virtual register.

Pseudo-code is fine. To indicate the output of an instruction, feel free to write things like

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

You'll find it convenient to make the semantic values of your non-terminal symbols be operands (registers or immediate operands), or in some cases pairs of operands, allowing you to communicate where to find the result of each subexpression. You may also introduce a global variable to use as a symbol table (for whatever purpose you might find you need it).

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

3. [8 points] *Receiver-class analysis* attempts to determine the possible dynamic types of the expressions "to left of dots" in languages like Java and Pyth. Consider a language that contains just class and type declarations and the following kinds of statements:

```
x = T()       // #1. T is a type; x now has exactly type T
x = y
x = y.m()     // #2. m is a method of y (an object pointer)
x = T.m(y)    // #3. m is a method of T, of which y must be a subtype
if b goto L1 // #4. b is a boolean variable, L1 is a label
goto L        // #5. L is a label
```

In addition, any statement can have a label. As in Pyth, we use single inheritance, and all possible methods of any given class are determined at compile time. To implement #2, you fetch `y`'s virtual table pointer, and then from it fetch the pointer to the function code of `m`. Implementing #3, on the other hand, will be faster—it requires no dereferences, because the code to be called is determined at compile time. If we could determine that the possible dynamic types of a variable `y` all share exactly the same code for `m`, we could implement line #2 the same way we implement #3.

Thus, in the following:

```
class T:
  def m (self): ... /*A*/
  m: T -> U
  def p (self): ... /*B*/
  p: T -> T
class U (T):
  def m (self): ... /*C*/
  m: U -> U
class V (T):
  pass
```

```
if b1 goto L1
y = T ()
goto L2
L1: if b2 goto L3
y = V ()
L2: z = y.m ()
goto L4
L3: y = U ()
L4: x = y.m ()
L5: x = y.p()
L6: x = x.p()
```

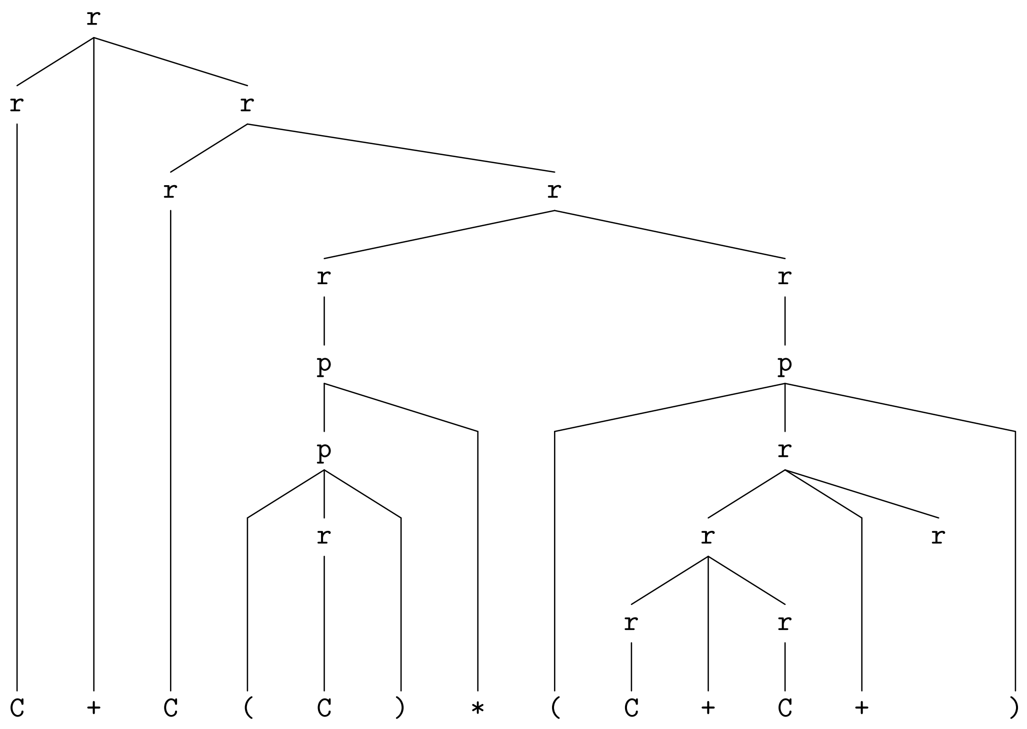
we can implement `L5` and `L6` as if they were `T.p(y)` and `T.p(x)` (that is, just do an ordinary function call to the label for the code implementing B above), and we can implement `L2` as if it were `T.m(y)`, since the only possible values of `y` at that point have dynamic types `T` or `V`. But we have to implement `L4` as a full method call, loading its virtual table first (since `y` might have type `U`). In this problem, we'll ignore entirely the possibility that a value might be None.

Initially, all you know about a variable (or parameter) is its static type, given by a type declaration (or Any if there is no declaration). All you know about the result of a method is that its return value has a subtype of the static return type declared for it.

On the next page, provide a global flow-analysis procedure that takes a segment of code such as that on the right above (a complete method or main procdure), complete information on the set of all classes and their methods, and static types (if specified) of variables, methods, and parameters, and computes the set of possible dynamic types for all variables at each point in the program. To make our notation consistent, show how to compute $\text{TYPES}_{\text{in}}(s, v)$ and $\text{TYPES}_{\text{out}}(s, v)$ for all statements s and variables v, which are the set of all possible dynamic types of v just before and just after the statement s.

Answer goes here.

4. [15 points] Consider the following parse tree. The terminal symbols are '`C`', '`+`', '`(`', '`)`', and '`*`'.



(Yes, that really is an 'r' with no children.)

a. Of what string is this a parse tree?

b. Give at least 9 steps in the reverse rightmost derivation that corresponds to this parse tree.

continued

c. Reconstruct as much of the BNF grammar corresponding to this parse as possible.

d. 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 `-|` denotes the end of file.

State	`C`	`(`	`)`	`*`	`+`	`-|`	`r`	`p`
0.	s1	s2	rE		rE	rE	s3	s4
1.	rA	rA	rA		rA	rA		
3.	s1	s2	rE		s6	s11	s7	s4
6.	s1	s2	rE		rE	rE	s10	s4
10.	s1	?	rD		?	rD	s7	s4

Show what symbols could be on the parsing stack that would cause state 10 to be the state of the top of the stack (Hint: examples arise in the parse tree at the beginning of this problem). Should the '?' entry in state 10 under '`(`' be filled in with a shift or with a reduce? (I am not asking for *which* shift or reduce.) Should the '?' under '`+`' be filled in with a shift or with a reduce? In both cases, why?

5. [1 point] Which of the following does not belong?

Aeolian, Corinthian, Dorian, Ionian, Locrian, Lydian, Mixolydian, Phrygian.

6. [14 points] For each of the following questions, provide a short, succinct answer.

a. In Pyth, we represent values of type Int differently from those of other types. Why?

b. Assuming that we wish avoid using the `__callattr__` routine whenever possible, how does the different representation of Int complicate code generation?

c. The runtime specification for the third Pyth project said that just before a function call occurs, the stack must consist of an integral number of PythValues, plus possibly a static link (at the top of the stack). What is the purpose of this requirement?

d. We included the runtime functions `__callattr__`, `__callfunc__`, `__getattr__`, and `__setattr__` in Pyth to deal with cases where the static type of an entity is `Any`. But Java has the type `Object`, which serves much the same purpose as `Any`. So why don't we need equivalent functions in the Java runtime? For this question, assume that the Int type and function types don't exist.