CS 164, Spring 2010 **CS 164: Final Examination Solutions** **P. N. Hilfinger**

1. [10 points] For each of the following possible modifications to a fully functional system for compiling and executing our Python dialect, tell which components of the compiler and run-time system would have to be modified: lexical analyzer, parser and tree-generator, static semantic analyzer, standard prelude, intermediate-code generator, machine-language code generator, 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. Change the scoping rules so that the bodies of compound statements (**if**, **while**, **for**) are declarative regions. For example, the following program is no longer legal:

```
def f(x, y):
  if x < y:
    a = x
  else:
    a = y
  return a
```

because the **return** statement is outside of the **if** construct and therefore outside the scope of `a`. However, an assignment inside a compound statement does not create a new declaration if a variable with that name is declared in an enclosing scope.

> **Answer:** It should suffice to change to name-lookup and variable declaration procedures in static semantics.

b. Allow the "infinite" looping construct:

```
while:
    𝒮
```

which repeats the statements $\mathcal{S}$ until they exit with a `break`, `return`, etc.

> **Answer:** Since this is equivalent to
>
> ```
> while True:
>     𝒮
> ```
>
> you can actually do it by modifying the parser to produce the tree for this expanded **while** loop when it sees an empty condition.

c. Make identifiers partially case-insensitive, so that, e.g., "`foo`" and "`Foo`" are treated as the same identifier, but make it an error to be inconsistent in the use of case for a given variable, as in

```
foo = 13
def f(x):
   Foo = 3 * x # OK (Foo hides the outer foo, and they are different
               # variables, which need not have the same capitalization).
   print foo   # ERROR (this refers to the local variable Foo, but
               # is capitalized differently).
```

Answer: Best handled in semantic analysis. Look variables up case-insensitively, but keep original spelling. Signal an error when looking up a use of a variable finds a different capitalization from the definition of the variable.

c. As for item (b), make identifiers partially case-insensitive, but this time make it an error to use two different capitalizations of the same identifier *anywhere in the source file,* even if they refer to two different variables:

```
def f(x):
    Foo = 3 * x
    print Foo
def g(x):
    foo = x / 3  # ERROR: the identifier 'foo' is used elsewhere,
                 # but capitalized.
```

Answer: Conveniently handled in the lexer. Keep a set of all identifiers (ignoring scope) and signal an error when a case-insensitive match finds a variable with a different capitalization.

e. Detect attempts to use the values of uninitialized variables for boxed types. By "use" here, we mean "try to use as a function" (as in `v(...)`), "select a method from" (as in `v.f(...)`), or "select an instance variable from" (as in `v.a`). You need not (but may) consider assignments of an undefined value to a variable as uses. The detection should *not* be conservative: detect exactly those cases where an uninitialized variable would actually be used.

> **Answer:** Since we can't be conservative, we must handle this at runtime and in code generation. Arrange to initialize all boxed variables to an invalid pointer value (e.g., 0). Either rely on the hardware to fault when this value is dereferenced, or (if your machine just produces random results in that case rather than segfaulting), insert runtime checks against the invalid value at each use.

2. [4 points] Consider the following grammar:

```
p : s ⊣
s : '[' E ']' s '/' s
s : V '=' E
```

Suppose that we have a shift-reduce parser for this grammar. Write a regular expression that describes the possible contents of the parsing stack just after a shift or a reduce. You do not need to include the final shifting of ⊣.

Answer: `("[E]"(s/)?)*(s|"["|"[E"|"[E]"|V|V=|V=E)|("[E]"(s/)?)+`

3. [8 points] GCC allows you to print out assembly code (the '-S' option) and then to read it back in and assemble it. In fact, this is what it does internally, even when you don't ask to retain the actual code. I'd like to do the same thing with our IL code (at the moment, we can only dump it with '-dIL' and cannot read it back in). In this problem, we ask you to do part of the job. Write a Bison parser for the following types of IL statements, separated by newlines:

```
MEM := REGIMM
REG := MEM
REG := REGIMM OP REGIMM     # OP is + or *
jump LABEL
if REGIMM CMP REGIMM jump LABEL  # CMP is > or ==
LABEL:
```

where

- `LABEL` is LN
- `MEM` is `*`M`(REG)`;
- `REG` is `%r`N;
- `REGIMM` is `$`M or `REG`.

Where M indicates any integer literal and N any positive integer literal. Don't bother with other instructions or types of operands (in particular, there is no `ENTRY` or `EXIT`, so we translate only the body of a single function). The actions in this Bison grammar should call the appropriate functions from the project: `vm.IMM(`M`)` for integer literals, `vm.MEM(`R`, `M`)` for memory accesses, `vm.allocateRegister()`, `vm.newLabel()`, and `vm.defineInstLabel()` for statement labels, and `vm.emitInst(...)`. You should not need any other methods from `vm.h` for this subset, but feel free to use standard C++ libraries as you see fit (or make them up within reason, if you don't remember the details). Also feel free to introduce global variables as needed. When you translate a register (such as `%r7`), or local statement label (such as `L7`), it doesn't matter what register 2number or label number gets used in the IL, as long as it is the same for each appearance of the same register or label, and different for different registers or labels.

Fill in the grammar rules and actions on the next page. Indicate what tokens you assume from the lexer (which you don't have to write) and what semantic values you assume each token or non-terminal has.

Put `%token` *and* `%type` *definitions here, as needed*

```
%token<int> LABEL REG IMM
%token ":="

%type<Operand*> reg regimm mem
%type<Label*> label
%type<Opcode> op cmp

%{
    VM vm = ...;
    map<int, Register*> regMap;
    map<int, Label*> lblMap;
%}

%%

program :
          /* EMPTY */
        | program inst NEWLINE
        ;

inst : mem ":=" regimm     { vm.emitInst(VM::MOVE, $1, $3); }
     | reg ":=" mem        { vm.emitInst(VM::MOVE, $1, $3); }
     | reg ":=" regimm op regimm { vm.emitInst(op, $1, $3, $5); }
     | "jump" label        { vm.emitInst(VM::JUMP, $2); }
     | "if" regimm cmp regimm "jump" label
                           { vm.emitInst($3, $2, $4, $6); }
     | label ':'           { vm.defineInstLabel(); }
     ;

mem : '*' DISP '(' reg ')' ;  { $$ = vm.MEM ($4, $2); }

reg : REG ;                { if (regMap[$1] == NULL)
                                 regMap[$1] = vm.allocatedRegister ()
                             $$ = regMap[$1]; }
imm : IMM ;                { $$ = vm.IMM($1); }
regimm : reg | imm ;
label : LABEL ;            { if (lblMap[$1] == NULL)
                                 lblMap[$1] = vm.newLabel();
                             $$ = lblmap[$1]; }
op : '+' { $$ = VM::ADD; } | '*' { $$ = VM::MULT; } ;
cmp : '>' { $$ = VM::IFGT; } | "==" { $$ = VM::IFEQ; } ;
```

4. [6 points] We'd like to determine what statements in a program *might* blow up from dereferencing a null pointer. We'll assume a vastly simplified, statically typed programming language. The only expressions are **new** (indicating a storage allocation of some appropriate type), v (for v a pointer variable), $v.a$ (for v a pointer variable and a the name of a field), and **null**. The only statements are

Assignments: $v = E$ or $v.a = E$, where E is an expression;

Conditionals: `if` E_1 `==` E_2 `then` S_1 `else` S_2 `fi`, where S_1 and S_2 are lists of (zero or more) statements and E_1 and E_2 are expressions;

Loops: `while` E_1 `!= null do` S_1 `od`.

Whenever the program evaluates an expression $v.m$ and v is `null`, the program "blows up" and control never reaches the next statement.

At any point in the program, we'd like to know (conservatively) what variables might be **null**. For example, your analysis should be able to annotate the following program as shown in its comments:

```
# 1. x, y, L, q, z might be null
while L != null do
    # 2. x, y, q, z might be null
    x = new
    # 3. y, q, z might be null
    y.a = x
    # 4. q, z might be null
    z = x
    # 5. q might be null
    L = L.n
    # 6. L, q might be null
    if L == null then
       # 7. L, q might be null
       q = new
       # 8. L might be null
    else
       # 9. q might be null
       q = L
       # 10. no variable can be null
    fi
    # 11. L might be null
od
# 12. x, y, L, q, z might be null
```

We'll assume that we never know anything about the contents of fields (such as `y.a`); that is, your analysis need not track their contents.

Questions begin on next page.

a. [1 point] Why does comment 4 no longer include `y` as possibly null, even though the pointer `y` itself is not changed by the preceding assignment to `y.a`?

Answer: According to the problem statement, the program will never get to #4 if `y` is null; it will "blow up" instead.

b. [4 points] Describe an appropriate analysis along the lines of the flow analyses described in lecture. One slight complication: in our lecture examples, a statement s always had one output (e.g., $C_{out}(X, s)$ for constant propagation). On this problem, for nodes denoting conditional branches, which have more than one output, you'll want to specify outputs for the true branch and the false branch. For example, $N_{out/t}(\cdots)$ and $N_{out/f}(\cdots)$.

Answer: Assume we transform the program into a CFG (**while** loops and **if** statements then become collections of basic blocks in the usual fashion). We'll choose true and false branches so that all tests have the form E_1 `==` E_2. We compute $N_{\text{in}}(s)$ and $N_{\text{out}}(s)$ to be sets of possibly null variables. The value of $N_{\text{in}}(s)$ is simply the union of the N_{out} values of s's predecessors (which may be $N_{\text{out/t}}$ or $N_{\text{out/f}}$ values for predecessors that are decision nodes). We define the transfer functions:

When s **is**	**Value of** $N_{\textbf{out}}(s)$
v_1 `:=` v_2	$N_{\text{in}}(s) \cup \{v_1\}$ if $v_2 \in N_{\text{in}}(s)$
	$N_{\text{in}}(s) - \{v_1\}$ if $v_2 \notin N_{\text{in}}(s)$
$v.a$ `:=` E	$N_{\text{in}}(s) - \{v\}$
v `:= new`	$N_{\text{in}}(s) - \{v\}$
v `:=` E	for other expressions E: $N_{\text{in}}(s) \cup \{v\}$

And for tests:

When s **is**	**Value of** $N_{\textbf{out/t}}(s)$	**Value of** $N_{\textbf{out/f}}(s)$
v_1 `==` v_2	$N_{\text{in}}(s) - \{v_1\}$ if $v_2 \notin N_{\text{in}}(s)$	$N_{\text{in}}(s)$
	$N_{\text{in}}(s) - \{v_2\}$ if $v_1 \notin N_{\text{in}}(s)$	$N_{\text{in}}(s)$
v `== null`, `null ==` v	$N_{\text{in}}(s) \cup \{v\}$	$N_{\text{in}}(s) - \{v\}$

and for all other cases, $N_{\text{out/t}}(s) = N_{\text{out/f}}(s) = N_{\text{in}}(s)$. Now we initialize $N_{\text{in}}(s)$ for the entry node to the set of all variables and to the null set for other nodes, and iterate to a fixed point as for other analyses.

c. [1 point] Give an example in which your analysis gives an overly conservative bound on a variable (that is, says that a variable might be null when it can't possibly be null).

Answer: Consider

```
x.a = new
y = x.a
# Analysis says y might be null.
```

5. [4 points] Ingrid Hackersdottir noticed that she could almost get the effect of dynamic variable binding in Python by converting definitions such as

```
def f(x):                          into     def f(x0):
    ...                                         global x
    h()                                         x = x0
    ...                                         ...
def h()                                         h()
    print x                                     ...
                                            def h()
                                                global x
                                                print x
```

(where `x0` is some variable name that is not used elsewhere in `f`.) This does cause `h`, when called, to see and print the right value for `x` (namely, the one established by the call to `f`). Unfortunately, this transformation does not quite implement dynamic scoping; it is missing a vital piece.

a. Give an example where this transformation does not work.

> **Answer:** The problem is that Ingrid forgot to restore the values of variables at the ends of functions. Thus, for the above, after `x = 3; f(5)`, `x` will be 5, whereas it should only be 3 during execution of `f`.

b. Describe how to fix the problem—that is, how to complete the implementation of dynamic scoping within Python. Give sufficient detail for Ingrid to implement the procedure. For simplicity, assume that all functions have a single exit point (either a single `return` statement or the end of the function) and that there are no `try` blocks.

> **Answer:** An easy approach is to introduce new local variables to save/restore previous values, as in

```
def f(x0):
    global x
    x1 = x
    x = x0
    ...
    h()
    ...
    x = x1
```

> This causes a slight problem with uninitialized variables (oddly, nobody asked about that). For this problem, we didn't worry about that (assume all variables get initialized to None, e.g.). For local variables other than parameters, use a slight modification:

```
def f():
    global x
    x1 = x
    x = None
    ...
    h()
    ...
    x = x1
```

6. [1 point] Who, responding to the Earl of Sandwich's taunt, "Egad sir, I do not know whether you will die on the gallows or of the pox," said "That will depend, my Lord, on whether I embrace your principles or your mistress."?

> **Answer:** John Wilkes. (Gladstone and Disraeli repeated this exchange pretty much word for word in the next century.)

7. [4 points] In a statically typed language without subtypes (that is, without **extends** and **implements** in Java), consider the following looping construct:

```
for V = E0 while C0 incr E1 do
    S
od
```

meaning that we set V to the value of E_0, and then execute S as long as C_0 is true, evaluating E_1 and assigning it to V after each iteration of S (and before re-testing C_0). Unlike C or Python, all expressions used for tests have type `bool`. The type of S is irrelevant, as long as it has a proper type. The type of the **for** as a whole is `void`. This statement defines variable V, whose scope then includes C_0, E_1, and S. Provide a typing rule for this construct, using the Prolog predicates `typeof` and `defn` as described in the lecture notes: `typeof(`V, T, E`)` means the static type of expression V is T in environment E (a list of the form `[def(`V_1, T_1`),...,def(`V_n, T_n`)]` for variables V_i and types T_i); and `defn(`I, T, E`)` means that identifier I has static type T in environment E. Assume that there are already other rules describing the other constructs in the language. The AST of these loops in Prolog syntax has the form

`for(`V', E_0', C_0', E_1', S'`)`

where the primed quantities are the ASTs for V, E_0, etc.

Answer:

```
typeof(for(V,E0, C0, E1, S), void, Env) :-
    typeof(E0, T, Env), Env1 = [def(V,T)|Env],
    typeof(C0, bool, Env1), typeof(E1, T, Env1), typeof(S, _, Env1).
```

(the clause `Env1 = [def(V,T)|Env]` just provides a shorthand; you can also just replace the three uses of `Env1` with `[def(V,T)|Env]`.)

8. [6 points] Consider the following class definitions:

```
abstract class A {
    void f(T t);
}
class B extends A {
    void f(T t) { t.g(this); } // #1
}
class C extends A {
    A x;
    C(A a) { x = a; } // #2
    void f(T t) {     // #3
        t.g(this);
        x.f(t);
    }
}
abstract class T {
    void g(B b);
    void g(C c);
}
class U extends T {
    void g(B b) { println("Hello!"); }    // #4
    void g(C c) { println("Bonjour!"); }  // #5
}
```

On the next page, draw a diagram of the objects pointed to by the variables in the following snippet, showing their instance variables and other data structures needed to make the code work. Label function code pointers with the numbers given in the comments above.

```
A a = new C(new B());

T t = new U();

a.f(t);
```

Put diagram here

Answer: See Nathan.

9. [8 points] For each of the following questions, provide a short, succinct answer.

a. Demonstrate that a grammar can be ambiguous, even though there are strings with exactly one parse according to the grammar.

   Answer: For the grammar

   ```
   e : e '-' e | 'x';
   ```

   there is only one parse of the input "`x`," but multiple parses of "`x-x-x`".

b. We chose (as do most implementations of similar OOP features) to put virtual table pointers in objects. But it is also possible to keep virtual table pointers in variables instead. That is, objects could contain only instance variables and each variable (local variable, instance variable, parameter, or global variable) could then contain a pointer to an object and another pointer to the virtual table pointer for that object. What advantages or disadvantages in performance can you see for such a scheme?

   Answer: Advantage: One fewer dereferences to get the virtual table pointer. Disadvantages:

   – Instance variables are bigger: total extra space is one word for each null variable and for each variable pointing to the same object as another.
   – When doing garbage collections, we can no longer tell the type of an object by just looking at the object, so sweeping gets a bit harder.

c. Our implementation of Python makes no provisions for garbage collection. Suppose that we were interested in changing that. Assuming we want to do "real" garbage collection (rather than a conservative heuristic, such as mentioned briefly in lecture), why is the current `vm.h` interface (used for IL code generation) insufficient? That is, why couldn't we just change the implementations of the IL routines, interpreter, low-code generation routines (in `assemble.cc`), and runtime to add garbage collection?

   Answer: We also need a little cooperation from the code generator to make sure that we initialize all local variables to some "neutral" value (like **null**) that the garbage collector can recognize, rather than leaving them uninitialized. Most importantly, the VM has no notion of type, so we don't know which variables are supposed to contain pointers.

d. In full Python, integers have no set limits on their values; it depends on the size of virtual memory. How does this language feature affect even the performance of programs that never compute values outside the range $[-2^{31}..2^{31}-1]$?

   Answer: The runtime can't *know* that all values will be in range, so we must add checks for integer overflow if we want an unboxed (and thus fast) representation.