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

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

Project #3: Static Analyzer for Pyth (version of 3 November 2006)

Due: Thursday, 9 November 2006

The third project picks up where the last left off. Beginning with the AST we produced in Project #2, you are to perform a number of static checks on its correctness, "decorate" it with information about the meanings of identifiers, and perform some transformations to forms that would have been inconvenient to produce in the parser.

You may implement your solution in either C++ or Java. Your job is to hand in a program (the parser and its testing harness), including adequate internal documentation (comments), and a thorough set of test cases, which we will run both against your program and everybody else's.

1 Input and Output

We'll provide a glue script that takes input arguments, generates an AST out of the specified input plus a *standard prelude,* described below, and feeds this AST to your semantic analyzer. The output from your analyzer will be a revised tree with "decorations" that indicate which declaration applies to which identifier and also give the type of each subexpression.

We'll provide the tedious software for reading and outputting the tree. Since it is difficult to test the output of your project directly against any kind of standard (many different ASTs encode the same program after all), we'll also provide a *decompiler* that converts the ASTs you output into Pyth-like programs in a language that is essentially Pyth extended with notations that reflect the type and definition decorations.

2 The Standard Prelude

The term *standard prelude* is used to describe the set of predefined things that a language provides. To vastly simplify your task in this project, we'll provide the standard prelude as a file of Pyth definitions that get prepended to the source program (that's why we've had that provision for inputting multiple source files, in fact). It will define predefined types (like `Any` and `Dict`), methods (like `__add__`), and constants (like `None`). Since the parser will already have turned operators into method calls, you won't have to do anything special about them.

Indeed, you can *almost* treat the standard prelude as just part of the source program text. There are just a few unavoidable little glitches, in which the handling of the standard prelude differs from elsewhere:

- The built-in types will be defined with `class` declarations, which they normally can't be.
- The superclass of `Any` is ignored, since `Any` has no proper superclass.
- The built-in types can use other built-in types as supertypes (normally, you have to use subtypes of `Object`).
- The type `Void` is a subtype of everything, which normal types cannot be.

For the purposes of this project, let's just recognize these special-case types by name.

3 Output Format

The output ASTs differ from input ASTs in these respects:

- Most identifiers will have an extra decoration at the end:

  `(id` F N `"`*name*`" (decl` D`))`

  where $D \geq 1$ is an integer *declaration index.*

- Nodes other than id nodes that denote expressions (things with values) will have a type decoration of the form `(stype` T`)` added to the end, where T is a type node such as produced in Project #2 for type declarations, with decorations. For example,

  ```
  (not 0 10 (id 0 10 "x" (decl 42))
       (stype (type (id "Bool" (decl 5)))))
  ```

  The expression operators are

  ```
  and call compare dict float int list not or select string tuple
  ```

  plus a few new ones introduced below.

- A list of all declaration nodes (in order by index) will follow the list of AST nodes (and the list of files, unchanged, will follow that).
- The analyzer will apply several re-writes to the tree so as to produce a tree that would have been inconvenient for the parser to produce.

There is one declaration index for each distinct declaration in the program: each class definition (including built-in types), local variable, parameter, constant definition, method definition, instance variable, and class variable. The output from your program will consist of the decorated AST followed by a list of all these declarations. The output declarations have the formats shown in Table 1.

Table 1: Declaration nodes. In each case, N is the declaration index, unique to each declaration node instance. Types are represented as in Project #2.

Node	Meaning
`(localdecl` N `"`I`"` P T`)`	Local variable named I of type T. P is the declaration index of the enclosing function, if any, or 0 for variables at the outer level.
`(paramdecl` N `"`I`"` P T`)`	Parameter named I of type T defined in a function whose declaration index is P.
`(constdecl` N `"`I`"` P T`)`	A constant named I of type T and defined inside the class or function whose declaration index is P (0 for constants at the outer level).
`(instancedecl` N `"`I`"` P T`)`	Instance variable named I of type T defined in the class with declaration index P.
`(funcdecl` N `"`I`"` P T `[`$f_1 \cdots f_n$`]` `[`$v_1 \cdots v_m$`])`	A class method or ordinary function named I of type T, defined in class or function with declaration index P (0 for the outer level), whose formal parameters have declaration indices $f_1, \ldots, f_n$ and that contains local declarations (of local variables, nested functions, constants) whose declaration indices are $v_1, \ldots, v_m$.
`(methoddecl` N `"`I`"` P T `[`$f_1 \cdots f_n$`]` `[`$v_1 \cdots v_m$`])`	An instance method. The arguments are the same as for `funcdecl`.
`(classdecl` N `"`I`"` P `[`$m_1 \cdots m_n$`])`	Class declaration for class named I. P is the declaration index of the parent type. The m_i are the declaration numbers of the class members.
`(nativeclassdecl` N `"`I`"` P `[`$m_1 \cdots m_n$`])`	Built-in class declaration. Same as class declaration, but indicates that this is one of the built-in classes from the standard prelude.

4 Rewriting

Because the parser lacks the necessary information (or finds it inconvenient to do the right thing), the semantic analyzer must do some re-writing of the trees that come from the parser. This comes in two parts: those rewrites that occur *before* doing any scope or type computations (i.e., that don't depend on the meanings of any identifiers or the types of any expressions), and those that occur *after* decoration with declarations and types. Table 2 shows rewrites that happen before decoration, and Table 3 shows rewrites that happen after decoration.

Beware: the application of some rewriting enables others. For example, Table 2 shows that when you translate the AST for

```
x[i], y = E
```

you will get a node that looks this (ignoring file and line number information):

```
(multiassign $\hat{E}$
   [(assign (call (id "__getitem__") [(id "x") (id "i")]) (getrhs (int "0")))
    (assign (id "y") (getrhs (int "1")))])
```

This in turn creates a new rewrite possibility, ultimately yielding:

```
(multiassign $\hat{E}$
   [(call (id "__setitem__") [(id "x") (id "i") (getrhs (int "0"))])
    (assign (id "y") (getrhs (int "1")))])
```

4.1 Various Restrictions

The context-free grammar from Project #2 allows certain things that Pyth disallows. Some of these *could* be enforced in the grammar, but weren't for convenience. In any case, they must be enforced in the static semantic analyzer.

1. A **break** or **continue** statement may occur only in a loop (**for** or **while**).

2. A **return** statement may only occur in a **def**.

3. An identifier, C, that refers to a class (in the scope of a a **class** definition) may be used only in the following ways:

   a. In the inheritance clause of another class definition;
   b. On the left of an attribute reference: C`.x`;
   c. As the function name in a call (i.e., in an object creation): C`()`.
   d. As a type name in a type assertion.

   Thus, constructs like `x =` C, `f (`C`)`, or `if` C are all illegal.

4. An inheritance clause in a class must reference a class completely defined previously in the program.

5. One may only assign to a variable, indexing operation, slicing operation, selection operation, or tuple. Augmented assignments to tuples are illegal.

4.2 Scope rules

In Pyth, the declarative regions (called *namespaces* in Python documentation) are as follows:

- The *global region,* consisting of the source file that is input to the compiler and the built-in definitions or *standard prelude* (in Python, these are two nested regions);
- A *class region* for each class definition;
- A *local region* including the parameters and body of each function or method definition (**def**) and each lambda.

Local regions nest inside each other and inside the global region. Class regions nest in the global region.

The scope of a **def** is the entire declarative region in which it occurs, except where hidden in a nested scope. Likewise, the scope of a class definition is the entire global region. A name defined by **def** may not be redefined in the same declarative region, and classes may not be redefined at all. Since an assignment in Pyth, as in Python, causes a declaration, this means the following are illegal:

```
f = 3
def f (x): ...  # Illegal redefinition.
def f (y): ...  # Another illegal redefinition
class f: ...    # And still another
```

The scope of a parameter is the entire function definition in which it occurs, except where hidden by inner declarations of the same name.

If there is an assignment or augmented assignment to a variable in a declarative region, or if it is implicitly assigned by being used as a control variable in a **for** statement, then there is (one) implicit definition of that variable in the innermost region containing the assignment, *unless* there is a **global** declaration of that variable in the region. The scope of such an implicit definition is the entire innermost region containing it (before and after the assignment), except where hidden by inner declarations, as usual. There is at most one implicit definition of a variable generated by this rule; multiple assignments have the same effect as one. If there is a **global** definition of a name in a certain region, then all uses of that name in the region (again, except where hidden) refer to the definition of the name in the global region. A name defined as global must also be defined by **def** or (implicitly) by assignment in the global region. Thus, the following program is illegal:

```
def f ():
   global a   # Illegal: no assignment to a in the global region
   a = 3
# a = 0       # Program WOULD be legal if this statement were uncommented.
f ()
print a       # Illegal: a is not defined.
```

The following are illegal:

```
class foo:
   global a       # Illegal: no assignment or def a in the global region
   global b
   def b (): ...  # Illegal: b already declared as global.
def b (): ...
# def a (): ...   # global a WOULD be legal if this were uncommented
```

The nesting rules already given imply that if a name is *not* assigned to or **def**ed in some function body, then any use of it in that body refers to a **def**, or to an implicit definition caused by an assignment, in some surrounding region; there must be one for the program to be legal.

A variable is implicitly defined by the presence of an assignment statement even if, during execution, that assignment never happens. Thus, the following is a statically *legal* program that may cause a runtime error when executed (if `a` is used but never initialized)

```
def f (x):
   if x > 3:
      a = "Answer is %d" % x
   print a
```

In other words, your static analyzer never considers whether a variable will be initialized (assigned to) before it is used when a program is run.

When the innermost declarative region surrounding an assignment is a class definition, the assigned-to variable is an *instance variable* of the class. The class also inherits all instance variables of its parent (if any). An assignment to an instance variable of the parent does not create a second instance variable of the same name; it refers to the parent's variable. These are the only legal instance variables (in Python, you can introduce new instance variables into a class or class object by assignment to an attribute at any time). In addition, the assigned-to variable is also defined as a *class variable* (static in Java). Thus,

```
class Stuff:
   x = 13
   def g (self, y): self.x = y

z = Stuff ()
z.g (42)
print Stuff.x  # Prints 13
print z.x      # Prints 42
```

When your static checker sees `Stuff.x`, it will know that `Stuff` is the name of a class. It should also be able to determine whether `x` is defined as a class variable of that class. On the other hand, when you see `z.x`, your checker *won't* generally know whether the expression is valid unless it happens to have a type declaration for `z`.

A class may **def** a function that has a **def** in its parent, which has the effect of overriding the parent's definition.

5 Types

By default, any defined entity in Pyth has type `Any`. A **def**ed instance method defined in class T has (by default) a type `(`*T*`,Any,...)->Any`, where the number of types to the left of the arrow is the number of parameters declared for the function. Other **def**ed functions have by default type `(Any,...)->Any`. A Pyth type assertion ascribes a type to a defined name. The name must be declared somewhere in the same innermost declarative region that contains its type assertion, and successive assertions for the same name must be compatible. When you ascribe a type to a `def`ed function, the number of parameters must be the same. The types of the parameters are ascribed to the formal parameters. Formal parameters may not be given types in the body of the function or method.

The job of your analysis is to determine the most specific type you can for each expression and declaration in the program, and to find any uses of a name that *must* be illegal. For this purpose, you only need to use information gleaned from type assertions, **defs**, and the language rules about certain basic constructs of the language: string literals are of type `String`, integer literals of type `Int`, tuples of type Tuple, list displays of type List, and dictionary displays of type Dict. If you know the type of a function, you can determine the type of a call to the function. Since the AST translates many Pyth constructs into calls, this fact will take care of most type checking for you.

As an example, your analyzer should be able catch these illegalities:

```
def f (x, y): x + y
print f (3)           # ERROR: number of arguments doesn’t match f’s type
class Foo:
   x: Int
   x = 3
S: String
S = "a"
S = 3           # ERROR: wrong type
Foo.x = S       # ERROR: wrong type
```

Your program is *not* required to catch this:

```
f = 3
f(2)     # There is no type assertion or def for f, so you don’t
         # know whether it’s a function.
3[1]     # __getitem__ is defined for all types.
a: List
a = [ "x", "y", "z" ]
a[0] + 1   # Only know that a is a List, not what’s in it
a[-1]      # Don’t understand about index bounds.
```

The specific legality rules we want checked are:

1. In a call $f(E_1, \ldots, E_n)$, either f must have type `Any` (i.e., unknown) or it must have a function type with n arguments.

2. Also, if f has the type $(T_1, \ldots, T_n) \rightarrow T_0$, then the known type of each E_i must be *feasible* for type T_i. We say that type A is feasible for type F if either A is a subtype of F or F is a subtype of A. Unlike Java, this allows many programs where the compiler cannot tell for certain that a particular call or assignment is allowed. What it does not allow is passing something known to be an `Int` as a `String` parameter.
3. In an assignment $x = E$, the type of E must be feasible for that of x.
4. In an assignment $x_1, \ldots, x_n = E$, where $n > 1$, the type of E must be `Any`, `Tuple`, `List`, or `Dict`.
5. In an attribute reference $X.y$, the attribute y is known to be an attribute of X (an instance variable for a class, e.g.).
6. No assignment is allowed to a **def**ed entity.

Most other interesting cases are handled by the function-call rule.

6 Output and Testing

The output of the program is again a textual representation of the AST (plus error messages). It's even more difficult than in the first project to check that your program's output is correct. Error tests will be of particular importance: you must make sure that breaking any of the rules causes an error. In addition, however, we will be augmenting the AST to provide a "decompiler" that reconstructs the source program with little annotations indicating which definition connects to each use, and giving type information for defined quantities. Once again, testing will be an important part of your grade.

7 What to turn in

The directory you turn in (under the name `release-3.`n in your `project/tags` directory should contain a file `Makefile` that is set up so that

```
gmake
```

(the default target) compiles your program and

```
gmake check
```

runs all your tests against your program. We'll put sample Makefiles (for C++ and Java) in the `~cs164/hw/proj3` directory; feel free to modify at will as long as these two commands continue to work.

8 What We Supply

We'll supply a framework to read and write ASTs and declarations (which, of course you may modify however you wish). We'll also supply a decompiler to turn the output of your program back into an annotated Pyth program for checking purposes.

9 Assorted Advice

What, you haven't started yet? First, review the Pyth language, and start writing test cases

Again, be sure to ask us for advice rather than spend your own time getting frustrated over an impasse. By now, you should have your partner's phone number at least. Keep in regular contact.

Be sure you understand what we provide. The `pyc.ast` package actually does quite a bit for you. Make sure you don't reinvent the wheel.

Do not feel obliged to cram all the checks that are called for here into one method! Keep separate checks in separate methods. To the extent possible, introduce and test them one at a time.

Keep your program neat at all times. Keep the formatting of your code correct at all times, and when you remove code, remove it; don't just comment it out. It's much easier to debug a readable program. Afraid that if you chop out code, you'll lose it and not be able to go back? That's what CVS or PRCS is for. Archive each new version when you get it to compile. Either of these version-control systems will allow you to go back to earlier versions at will.

Write comments for classes and functions before you write bodies, if only to clarify your intent in your own mind. Keep comments up to date with changes. Remember that the idea is that one should be able to figure how to use a function from its comment, without needing to look at its body.

You *still* haven't started?

Table 2: Pre-decoration rewriting rules. Source-position information is elided for brevity. New nodes should be given the source positions of the nodes they replace.

Notes.

- The new `getrhs` operator is an expression that returns type `Any`.
- The new `foritem` operator is an expression that returns type `Any`.

Original Node	Rewrite
(assign (call (id "__getitem__") [A K]) V)	(call (id "__setitem__") [A K V])
(assign (call (id "__getslice__") [A J K]) V)	(call (id "__setslice__") [A J K V])
(assign (tuple [$A_1 \cdots A_n$]) E)	(multiassign E [(assign A_1 (getrhs (int "0"))) $\vdots$ (assign A_n (getrhs (int "$n-1$")))])
(for v e S_1 S_2)	(for (assign v (foritem)) e S_1 S_2)

Table 3: Post-decoration rewriting rules. Decorations and source-position information are elided for brevity. New nodes should be given the source positions of the nodes they replace.

Original Node	Rewrite
(call (id i) [$E_1 \cdots E_n$]) *where i denotes an instance method*	(callmethod (id i) [$E_1 \cdots E_n$])
(call (select (id i) E_0) [$E_1 \cdots E_n$]) *where i denotes an instance method and E_0 denotes a value (as opposed to a type).*	(callmethod (id i) [E_0 $E_1 \cdots E_n$])
(select (id i) T) *where either T denotes a type (as opposed to a value), or i denotes a class method or constant.*	(selectstatic (id i) T)