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

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

Project #2: Static Analyzer for Pyth[*]

Due: Wednesday, 6 April 2005

The second project picks up where the last left off. Beginning with the AST we produced in Project #1, you are to perform a number of static checks on its correctness, and "decorate" it with information about the meanings of identifiers.

When the directory `~cs164/hw/proj2` becomes readable, you can copy the files we have placed there to help with this project. We will also update the `pyc.ast` package (and its C++ counterpart) to give you places to "hang" decorations on the tree. Specifically, you will be able to connect every variable to a definition that contains (among other information) what you know about its type.

1 Static Semantics Rules

The semantics of Pyth are largely those of Python, but with some significant changes that make a certain amount of static checking possible. Assume the rules of Python, therefore, except as otherwise indicated. Since this is the first offering of the project, and we may yet make clarifications and simplify things to avoid problems we discover, you should be sure to check the entry for this project on the homework page for updates (and watch the newsgroup, of course).

1.1 Various Restrictions

The context-free grammar from Project #1 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.

[*]Updated 5 April 2005. Corrections shown in red.

1. Class definitions may appear only at the outer level—that is, a program is a sequence of classdefs and other statements, none of which may contain a classdef nested within it.

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

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

4. 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`()`. In this version, we deal only with parameterless constructors.

   d. As a type name in a type assertion.

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

5. An inheritance clause in a class must reference a class defined previously in the program. There may not be a circular chain of inheritance: a class may not inherit directly or indirectly from itself.

1.2 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** or **class** may not be redefined in the same declarative region. 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 lambda or 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 or lambda, 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.

2 Types

The types in Pyth are as follows:

Any The supertype of all types. To say that `x` has type `Any` is to say we know nothing about its type.

Unit The type of `None`. Unit is a subtype of all types except `Int`.

Int The type of Pyth integers (like Java's `int` type).

String All kinds of Pyth strings.

List Mutable sequences, as created by the `[...]` construct.

Tuple Tuples, as created by `(...)`.

Dict Dictionaries, as created by `{...}`.

classes Each class name functions also as the name of a type. Class types are supertypes of Unit, and subtypes of Any and of their parent type.

function types As given by the syntax for function_type in the grammar. A function type $(D_1, \cdots, D_n) - > C$ is a subtype of $(D'_1, \cdots, D'_n) - > C'$ iff C is a subtype of C' and each D'_i is a subtype of D_i (Yes, I got that right; see if you can figure out why. Hint to you mathematicians: the word is “contravariant”).

The names **String**, etc., are not reserved; they are meaningful as types only in Pyth type assertions.

By default, any defined entity in Pyth has type `Any`. A **def**ed function has (by default) a type `(Any,Any,...)->Any`, where the number of Any’s to the left of the arrow is the number of parameters declared for the function. 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.

The AST definitions are set up so that each type is represented by an object that tells you what attributes (things fetched by the dot operator) that type has, and whether it is a sub- or supertype of another.

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, dictionary displays of type Dict, and lambdas have a function type `(Any,...,Any)->Any`. 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
    3[1]            # ERROR: Int doesn't define __getitem__
```

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.
    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`, or `List`, or `Dict`.
5. In an attribute reference $X.y$, either X has type `Any`, or the attribute y is known to be an attribute of X (an instance variable for a class, e.g.).

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

3 The Standard Prelude

The term *standard prelude,* is used to describe the set of predefined things that a language provides. You don't have to worry about this; it will be our job to arrange that the AST that is input to your program contains all the necessary definitions, including information about the methods defined on built-in types. You won't have to make any special arrangements for checking that `x[3]` is legal; in the tree, it will look like a function call (to `__getitem__`) and the rules for function call will cover it.

4 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 "decorated unpything" 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.

5 What to turn in

You will be turning in three things:

- Source files (in Java or C++).
- A testing subdirectory containing Pyth source files and corresponding files with the correct output.
- A Makefile that provides (at least) two targets:
  - The default target (built with a plain `gmake` command) should compile your program, producing an executable program called `pythc` (we will provide instructions for how to accomplish this in Java, so that you don't need the `java` command to run your program.
  - The command `gmake check` should run all your tests against your compiler and check the results.

6 What We Supply

We will shortly update the `pyc.ast` package for your use. We'd like to keep control of the `AST` hierarchy to make changes where needed, so we will be using a version of the *Decorator Pattern* to allow you to add whatever you need. That is, each AST node will have a *delegate* pointer to an object that you will control and that will contain any additional information and methods that you need to add. With this, and judicious use of inheritance, you can add things to AST, in effect, without rewriting the types.

7 Assorted Advice

What, you haven't started yet? First, get to know the Python language better, compare it to Pyth, 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?