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

CS 164 **P. N. Hilfinger**
Spring 2010

Project #2: Static Analyzer (revision 8, 4/6/2010)

Due: Tuesday, 6 April 2010

The second project picks up where the last left off. Beginning with the AST you produced in Project #1, you are to perform a number of static checks on its correctness, annotate it with information about the meanings of identifiers, and perform one rewrite. Your job is to hand in a program (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 Summary

Your program is to perform the following processing:

1. Add a list of indexed declarations, as described in §5.
2. Add a declaration index to the end of each `id` nodes, linking it to a declaration. This is also described in §5. There are a few exceptions to this rule: don't bother attaching declarations to the identifiers in "binop," "unop," "aug_assign," or "comparison" nodes.
3. Rewrite all lambda expressions into explicit functions, as described in §6.
4. Rewrite method calls and allocation expressions to use new AST nodes not produced by the parser, as described in §6 and §4.
5. Enforce the restrictions described in §7.

The remaining sections describe these in more detail.

2 Input and Output

You can start either from a parser that we provide, or you can augment your own parser. In either case, the output from your program will look essentially like that from the first project, but with some additional annotations. We'll augment `pyunparse` to show your annotations.

Python is a very dynamic language; one may insert new fields and methods into classes or even into individual instances of classes at any time. One may redefine functions, methods, modules, and classes at will. For this project, we will introduce a few restrictions, but there will be many places where we can't definitively say that something is an error.

You will add information to identifiers indicating their type. In Python, the compiler will, in general, know very little about the types of things, so that the best we can usually say is that "variable α has static type `any`," where `any` denotes the supertype of all types. Sometimes, however, as in the case of functions, you will be able to at least check parameter counts.

3 New Syntax

We've added a small piece of syntax to the language. If you want to use your own parser, you'll find it is not difficult to add.

A target of any assignment (which includes the control variables of **for** nodes) may have the form "I`::`T", where '`::`' is a new lexeme and I and T are identifiers. This corresponds to the AST node

`(typed_id` N `(id` N I`) (id` N T`)),`

and indicates that the variable denoted by I here has type T (which must be a class). This typing applies to *all* instances of I that refer to the same variable as this one (so that in

```
a = "Hello"

def f(x):
    a::Int = x
    print a + 3
```

the '`a`' defined in '`f`' has integer type, but the one defined outside '`f`' has unknown type.

4 New AST Nodes

Besides "`typed_id`" (see §3), we introduce the following new nodes, which you'll use to rewrite certain parts of the tree (but which the parser could not generate since it lacked the necessary semantic information).

`(call_method` N I `(`E_1 E_2 `...))` represents $E_1.I(E_2, \ldots)$ when E_1 denotes an object (as opposed to a module or type).

`(call_method_ident` N I `(`E_1 E_2 `...))` is the same as `call_method`, except that it denotes a construct that returns the value of E_1 rather than the value returned by the method.

`(new` N T`)` When the first child of a "`call`" node denotes a type, T, that call actually creates a new object of the type. This node denotes a construct that creates and returns a new, uninitialized instance of T.

5 Output Format

The output ASTs differ from input ASTs in these respects:

- Identifier nodes will have an extra annotation at the end:

  `(id` N *name* D`)`

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

- Compilations will now have the syntax

  ```
  Compilation : '(' "module" N Stmt* ')' Decl*
  ```

  The `Decls`, described in Table 1, represent declarations. They are indexed by the declaration indices used in `id` nodes, and appear in order according to their index.

There is one declaration index (and corresponding declaration node) for each distinct declaration in the program: each imported module, class definition, local variable, parameter, method definition, and instance variable. Table 1 shows the formats of the declaration nodes.

Use the special declaration `unknowndecl` for the (many) cases where Python is too dynamic to give static information about what kind of entity one is dealing with. So, in

```
import foo, bar
j = foo.A()
k = bar.A()
j.i = foo.i + bar.i + k.i
```

there could be the following declaration nodes:

```
(moduledecl 99 __main__)
(moduledecl 100 foo)  (moduledecl 101 bar)
(localdecl 102 j 99 (type 0)) (localdecl 103 k 99 (type 0))
(unknowndecl 104 A 100) (unknowndecl 105 A 101)
(unknowndecl 106 i 100) (unknowndecl 107 i 101)
(unknowndecl 108 i 0)
```

Declaration 108 here handles both `j.i` and `k.i`. (We started numbering declarations at 99 here to suggest the existence of the declarations in the standard prelude, described in §9.)

The set of declarations is *not* the same as a symbol table (or environment). It is an undifferentiated set of *all* declarations without regard to scopes, declarative regions, etc. You'll need some entirely separate data structure (which you'll never output) to keep track of the mappings of identifiers to declarations at various points in the program. Some declarations don't correspond to anything you can point to or name in the program. For example, under our rules, the module names `__main__` and `__builtin__` are not defined within your program, and references to them are errors, even though those modules certainly exist and contain lots of definitions you *can* reference.

6 Rewriting

For the sake of the code generator (and to some extent, to simplify parts of semantic analysis), your program must perform several rewritings.

Lambda expressions. In the output tree, replace all lambda expressions with explicit functions. For example, an input program that looks like this:

```
def f (x, L):
    return map (lambda y: y+x, L)
```

would produce the same kind of tree that would be generated by

```
def f (x, L):
    def __lambda1__ (y):
        return y+x
    return map (__lambda1__, L)
```

Place the `def`s for all lambda expressions in a function (or the main module) at the beginning of the body of that function (or the main module). For simplicity, we'll just assume that names of the form `__lambda`N`__` are not explicitly allowed in source programs.

Allocators. Whenever you encounter a "call" node whose first operand denotes a class (which is Python's way of writing the Java or C++ **new** operator):

`(call` N T `(`E_1 E_2 `...))`,

convert it to the expression

`(call_method_ident` N `(id` N `__init__) ((new` N T`)` E_1 E_2`...))`,

thus making explicit what happens when you construct a new object.

Method calls. Whenever you encounter a "call" node of the form

`(call` N `(attributeref` N E_1 I`) (`E_2 `...))`,

where E_1 denotes an object (not a type or module), convert it to the form

Table 1: Declaration nodes. The list of the declaration nodes for a program in order by index follows the AST. In each case, N is the declaration index, unique to each declaration node instance.

Node	Meaning
`(localdecl` N I P T`)`	Local variable named I. P is the declaration index of the enclosing function or module. T defines what is known about I's type (see §8, below).
`(paramdecl` N I P K T`)`	Parameter named I of type T defined as the K^{th} parameter (numbering from 0) of the function whose declaration index is P.
`(constdecl` N I P T`)`	Constant value named I of type T defined in a function or module whose declaration index is P. This is for unassignable values such as `None`, and is only used in the standard prelude.
`(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`)`	An ordinary function (as opposed to an instance method) named I of type T, defined in a function or module with declaration index P.
`(methoddecl` N I P T`)`	An instance method. The arguments are the same as for `funcdecl`, except that P refers to the enclosing class.
`(classdecl` N I M P `(index_list` $m_1 \cdots m_n$`))`	Class declaration for class named I. M is the declaration index of the containing module. P is the declaration index of the parent type. The predefined classes, including 'object' (see §9), are the only classes with $P = 0$.The m_i are the declaration numbers of the class members that are introduced or overridden in the class (not the ones that are inherited but not overridden).
`(moduledecl` N I`)`	Module declaration. The main module of a program has the name `__main__`. However, that name is not visible in the program (the module named `__main__` is not imported). The **import** statement introduces other modules. The second and subsequent imports of the same module do not create new mdoule declarations.
`(unknowndecl` N I P`)`	An unknown entity. Use this for members of imported modules (e.g., `re.match` or `sys.argv`), and for names that are selected from objects of unknown type. If I is imported from a module (using the "**from** *module* **import...**" syntax), or if it is selected from a module named in an **import**, then P is the index of that module. Otherwise, it is 0. Create one `unknowndecl` declaration for each *distinct* combination of I and P.

`(call_method` N I `(`E_1 $E_2 \ldots$`)`.

On the other hand, if E_1 turns out to denote a type or module, replace the "attributeref" node with I (do this, however, *after* determining the declaration index to attach to the identifier I). Thus, when E_1 above denotes a type, we replace the "`call`" with

`(call` N I `(`E_2 `...))`.

Attributes of classes. As with method calls, whenever you encounter a node of the form

`(attributeref` N E_1 I`)`,

where E_1 is known to denote a type or module, replace it with I, after assigning the appropriate declaration index to I.

7 Various Restrictions

Our Python dialect is going to place certain restrictions on programs that are not official Python, but that make it possible to perform a few simple checks.

1. The first parameter of a method (that is, of a `def` that occurs immediately within a class definition) has the enclosing class as its static type. The first parameter of a Python method corresponds to `this` in a Java program. All methods must have at least one parameter.

2. An identifier that is defined as a class, function, method, constant, or module may not be assigned to. Programs can't explicitly define constants, but they can appear in the standard prelude, which uses one for the identifier `None`.

3. An inheritance clause in a class must reference a class completely defined previously in the program, or the predefined class `object` (which is defined in the standard prelude: §9).

4. If an identifier, f, resolves to a function or method definition (as opposed to simply having the universal supertype `any`), then a call to it must have the right number of parameters.

5. Names of classes, methods, and functions may not be redefined immediately within the same declarative region (function, class, or module). If a variable is assigned to in some declarative region, its name may not then be defined by **def** or **class** statements immediately within that same region (and vice-versa).

6. The only attributes of a class (things referenceable with '.') defined by a **class** declaration in the program are instance variables explicitly assigned to in the body of the class (outside of any methods), or methods defined by **def** immediately within the class body, or inherited attributes. Thus, the only attributes of class C:

```
class C(A):
    a = 3
    def f(self): ...
```

are `a`, `f`, and anything inherited from `A` (other than `a` or `f`). That means that the following are illegal in our subset:

```
class A(object):
   a = 3
   def f(self, x):
       self.b = 10  # ERROR: no b in class A instances
       x.b = 10     # OK: static type of x is any, not A
A().b = 2           # ERROR: no b in class A itself
A.b = 3             # ERROR: No b in class A itself
x = A()
x.b = 2             # OK: static type of x is any
```

Your compiler must catch these errors.

7. If a class inherits a method, it may not override (redefine) that method with another having a different number of parameters. It may not redefine an inherited method by assignment or define an attribute defined by assignment as a method.

8. The static type of the first parameter of a method is the enclosing class. The compiler must check that any attribute fetched from that parameter is indeed an attribute defined (or inherited) by that class (see point 6 above).

9. The scope of declarations other than classes includes the entire declarative region that contains them (before and after the declaration, in other words). In the case of classes, this declarative region does not include the bodies of methods within those classes. This is the same as for regular Python except at the outer level.

10. It is illegal to introduce a variable, parameter, function, method, class, or module named `None`.

11. Except for identifiers that appear immediately after the dot (.) operator, all identifiers that are used must be defined.

12. In a call such as

    E`.f(3)`,

    where E denotes an object (as opposed to a type or module), `f` must be a method (defined by **def**), not an instance variable. In actual Python, you can have situations like this:

```
def g(x):
    return x+1
```

```
y::A = A()
y.f = g
z = y.f(4)
```

where `f` denotes an instance variable. We won't allow that (you can still write

```
q = y.f
z = q(4)
```

Similarly, we don't allow selection of a method defined by **def** *except* in a call, so this is illegal:

```
class A(object):
    def f(self): ...

y::A = A()
g = y.f
```

We assumed these rules when we indicated that you should rewrite some call nodes into "`call_method`" nodes (§6). You must enforce them whenever the selected-from object has a known user-defined class (that is, a subtype of `object`). You don't (indeed often can't) enforce these rules when the selected-from object has some other type.

13. While most identifiers have type `any`, a few will have a non-trivial static type (those declared by `::` and the first parameters of methods). Your program must insure that any simple assignments to such variables come either from expressions whose static type is `any` or from a subtype of the assigned variable's type. Thus, it should be illegal to write

```
x::Int = 2
x = []
```

Likewise, in those few cases where a formal parameter has a known type other than 'any' in a call (which only happens in a few methods defined in the standard prelude and for the first parameters of user-defined methods), your program should complain if the actual has the wrong type:

```
x::String = "X"
y::Int = 3
z = "X"
print chr(x)     # ERROR: parameter must have type Int, and x is a String
print chr(y)     # OK: y has the right type
print chr(z)     # OK: z has the wrong dynamic type, but its static
                 # type is any.  Detection must wait until run time.
```

8 Types

For this project, the possible types are either classes or function types, represented as follows:

Node	Meaning
`(type 0)`	The type 'any' (meaning basically the unknown type).
`(type` C`)`	Where $C > 0$: the class whose declaration index is C.
`(functype` T_0 $T_1 \cdots T_n$ `)`	Where the T_i are types: the type of a function that takes $n \geq 0$ arguments of types $T_1, \ldots, T_n$ and returns a value of type T_0. For this project, T_0 and T_j for $j > 1$ will always be `(type 0)` for functions and methods the user introduces (but not necessarily for methods or functions introduced in the standard prelude). For methods introduced by the user, T_1 will always be `(type` D`)`, where D is the index of the enclosing class for methods. For other user-defined functions, T_1, if present, will be `(type 0)`.

By default, any defined identifier in Python has static type `any` (represented `(type 0)`). An instance method defined in a class C whose declaration index is k and which has $n > 0$ parameters has a type $\underbrace{C \times \text{any} \times \ldots \text{any}}_{n} \rightarrow \text{any}$, which is represented by

`(functype (type 0) (type` k`) (type 0) ...)`.

Other **def**ed n-argument functions have type $\underbrace{\text{any} \times \ldots \times \text{any}}_{n} \rightarrow \text{any}$, which is represented `(functype (type 0) (type 0) ...)`.

Using the `::` syntax from §3, the programmer can attach a static type to any variable identifier. Doing so anywhere in the variable's scope defines the type for all instances. If the same variable is given a type twice, your compiler must check that it is the same type.

The standard prelude provides several classes that represent built-in types:

```
Int String List Tuple Dict
```

These types all inherit from type `any` (which user-defined classes can't do). All integer literals have type `Int`, string literals have type `String`, list displays (`[...]`) have type `List`, tuples have type `Tuple`, and dictionary displays (`{...}`) have type `Dict`. (We are not going to worry about floating-point numbers for the rest of the semester). Make sure that you give these constructs the indicated types in your output trees.

9 Predefined Names

Python has a large set of predefined classes, functions, and variables, collectively referred to as "the standard library," or in other languages as "the standard prelude." These live in a module called `__builtin__`, which you may think of as a declarative region that surrounds that of the module `__main__`, so that all of its definitions are visible in any program, unless hidden by a definition in that program.

We will supply a file containing the AST for a `__builtin__` module (but not even close to full Python), and our framework will contain a small AST-parsing section for reading it in (which you are free to borrow should you choose not use our framework). You will have to fill in this parser with statements to create declaration and type objects, however you want to represent them. For your own testing purposes, you'll be able to use cut-down versions of `__builtin__`. The definitions from `__builtin__` should be included at the beginning of the declaration list in your output. All of the names there, with the exception of `__builtin__` itself, should be visible in your program.

10 Running the program

For this project, the command line looks like one of these (square brackets indicate optional arguments):

```
./apyc --phase=2 -o OUTFILE [ --prelude=PRELUDE ] FILE1.py
./apyc --phase=2 [ --prelude=PRELUDE ] FILE1.py FILE2.py ...
```

The command lines from project 1 should still do the same thing. That is, `phase=1` should just parse your program and not do semantic analysis. The `-o` switch indicates the output file. By default (the second form), the output files are $FILE_i$`.dast` (".dast" for "decorated ast"). The file *PRELUDE* contains the standard prelude definitions in S-expression notation (as described in §9). By default, this will be a file `standard-prelude-2.dast` that we supply in a standard location.

11 What to turn in

The directory you turn in (under the name `proj2-`n in your `tags` directory) should contain a file `Makefile` that is set up so that

```
gmake
```

(the default target) compiles your program,

```
gmake check
```

runs all your tests against your program, and finally,

```
gmake APYC=PROG check
```

runs all your tests against the program *PROG* (by default, in other words, *PROG* is your program, `./apyc`). Finally,

```
gmake clean
```

should remove all files that are regeneratable or unnecessary. We'll put a sample Makefile in the staff proj2 repository directory and in the file `~cs164/hw/proj2` directory; feel free to modify at will as long as these commands continue to work.

12 Assorted Advice

What, you haven't started yet? First, review the Python language, and start writing test cases. You get points for thorough testing and documentation, and it should not be difficult to get them, so don't put this off to the last minute!

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 skeleton classes actually do 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 Subversion is for. Archive each new version when you get it to compile (or whenever you take a break, for that matter). This 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?