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

CS 164 **P. N. Hilfinger**
Spring 2011

Project #2: Static Analyzer (revision 15)

Due: Wednesday, 6 April 2011

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 two rewrites. 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 §4.
2. Decorate each `id` node by adding a declaration index that links it to a declaration in the list. This is also described in §4.
3. Perform several small rewrites of the AST, described in §5:
   (a) Add an extra (decorated) `id` node as the last child of `subscription` and `slicing` nodes to indicate which function to use to perform the operation.
   (b) Rewrite all lambda expressions into explicit functions.
   (c) Rewrite allocation expressions to use new AST nodes that were not produced by the parser.
4. Enforce the restrictions described in §8.

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 AST Node Types

We introduce the following node type for denoting allocation, which the parser could not generate since it lacked the necessary semantic information:

`(new` N T `(id` N `__init__) (expr_list` N E_1 E_2 `...))`

is the translation of

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

when T turns out to be a type (it will generally be an `id` node). The extra `id` node for `__init__` is to be decorated with the declaration for the relevant `__init__` procedure for that type (there should always be one). (In Python, when `A` denotes a type, the construct `A(`*arguments*`)` means "allocate a new object of type `A` (call it x), then call x`.__init__(`*arguments*`)`, and then yield the value x as the value of the expression.")

In addition, the semantics deals with two new function types. They never appear in the output from your program, but for internal purposes, you are likely to want to have AST nodes to represent them. See §6.1. First,

`(bound_function_type` N *return-type* *arg-type*$_1$ ... *arg=type*$_n$`)`

represents the type of `x.f` when `x` denotes an object and `f` denotes a method of its class. Similarly,

`(bound_function_type_star` N *return-type* *arg-type*$_1$ ... *arg=type*$_n$`)`

is the same, but for methods with a trailing `*` argument.

4 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.

- Nodes representing subscripting and slicing get an extra `id` node at the end to provide a place for the semantic analyzer to indicate what function should perform the operation. See §5.

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

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.

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. T defines its static type (see §6, below).
`(globaldecl` N I P T`)`	A global variable named I, defined outside any function. P is the declaration index of the module containing it. T is its static type.
`(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 module (actually, this is always the `__builtin__` module) whose declaration index is P. This is for unassignable values such as `None`, and is only used in the standard prelude. All definitions of "variables" in the standard prelude should produce `constdecl` nodes (and only those declarations).
`(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. Only the builtin class `any` has $P = 0$ (see §9) (for purely syntactic reasons, however, In the standard prelude it is written to take itself as its parent.) The m_i are the declaration numbers of the class members that are introduced or overridden in the class (not the ones that are just inherited, but not overridden). They should be listed in order of appearance in the source text of the class.
`(moduledecl` N I `(index_list` $m_1 \cdots m_n$`))`	Module declaration. The main module of a program has the name `__main__`. However, that name is not visible in the program, nor can it be imported. The builtin definitions of the standard prelude are contained in a module with the name `__builtin__` (also not explicitly visible or importable). The **import** statement introduces other modules. The second and subsequent imports of the same module do not create new module declarations. The `index_list` gives the indices of declarations in the module, in the order they appear in the source.

5 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 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 `(expr_list` E_1 E_2 `...))`,

convert it to the expression

`(new` N T `(id` N `__init__) (expr_list` N E_1 E_2 `...))`.

With an appropriate declaration index on the `id` node.

Attributes of classes. Whenever you encounter a node of the form

`(attributeref` N E_1 I`)`,

where E_1 denotes a known type or module that defines I (an `id` node), replace it with I, after assigning the appropriate declaration index to I. Thus, after the Python class declaration

```
class A(object):
   x = 13
   def f (self): ...
```

The statement

```
a, g = A.x, A.f
```

becomes, in effect,

```
a, g = x, f
```

but with `x` and `f` decorated with the appropriate declarations of instance variable `x` and method `f`. It is an error for E_1 to denote a type or module that is not known to define I.

Slices and Subscripting In our Python dialect, slicing and subscripting are actually function calls. The effect of

```
print x[3], s[1:7]
```

is identical to that of

```
print x.__getitem__(3), s.__getslice__(1, 7)
```

The statements

```
x[3] = y
s[1:7] = q
```

are equivalent to

```
x.__setitem__(3, y)
s.__setslice__(1, 7, q)
```

To help out the code generator, add an extra `id` node to the ends of `subscription`, and `slicing` AST nodes, decorated with the appropriate declarations, as in:

```
(subscription N (id N x) (int N 3) (id N __getitem__))
(slicing N (id N s) (int N 1) (int N 7) (id N __getslice__))
(subscription N (id N x) (int N 3) (id N __setitem__))
(slicing N (id N s) (int N 1) (int N 7) (id N __setslice__))
```

where the trailing *id* nodes are decorated with the appropriate `methoddecl` indices (and, as usual, the other `id` nodes are eventually decorated with indices as well).

6 Types

For this project, the possible types are either classes, function types, or bound-method types.

6.1 Type representation

Class and function types are represented as in project #1:

(`type` N (`id` N *type-name*))

(`function_type` N *return-type* *arg-type*$_1$ ... *arg=type*$_n$)

(`function_type_star` N *return-type* *arg-type*$_1$ ... *arg=type*$_n$)

and in addition, we introduce two new function types:

(`bound_function_type` N *return-type* *arg-type*$_1$ ... *arg=type*$_n$)

(`bound_function_type_star` N *return-type* *arg-type*$_1$ ... *arg=type*$_n$)

(All `id` nodes here and below should also have appropriate declaration indices attached.) If we have the Python statements:

```
class A(B):
    def f(self): ...
    def g(self,*x): ...
x::A = A()
```

then the expressions `A.f` and `A.g` have, respectively, the types

```
(function_type 0 (type 0 (id 0 any)) (type 0 (id 0 A)))
(function_type_star 0 (type 0 (id 0 any)) (type 0 (id 0 A)))
```

and the expressions `x.f` and `x.g` have, respectively, the types

```
(bound_function_type 0 (type 0 (id 0 any)))
(bound_function_type_star 0 (type 0 (id 0 any)))
```

(the line-number attributes here are irrelevant). There are restrictions on what you can do with values of the bound types (see §8).

With a couple of exceptions, unless defined otherwise with an explicit type assignment (via '`::`'), any variable or parameter has static type `any`. Likewise, `any` is the default return type of a method or function. This type, defined in the standard prelude, is an extension to Python. It's not a real type, in that no object of that type can be created (attempting to call `any(`*E*`)` will always cause a run-time error). There are two exceptions to this default rule:

- The first parameter of any method has the type of the enclosing class.
- A '`*`' parameter (such as `extra` in

  ```
  def open(name, *extra):
      etc.
  ```

  always has type `list` (AST: `(type 0 (id 0 list))`) (where the `id` is decorated with the declaration index for the builtin class `list`).

Besides `any`, the standard prelude provides several classes that represent built-in types:

```
bool int str list tuple dict type object file NoneType module xrange
```

These types do not inherit from `object`, as user-defined types must. In the text of the standard prelude, the special type `any` inherits from itself (a kludgy special case.) 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`. The standard prelude also defines the two constants `True` and `False` of type `bool` (in Python 2.5, these aren't constants, but it's convenient for us to make them constant anyway), and the constant `None`, of type `NoneType`. If *A* is a type (the name of a class), then it has type "`type`". We've also introduced the type `module` (not in Python), which is the type of all modules.

6.2 Subtyping Rules

The subtyping rules in our dialect of Python are more than a little odd because of the status of the type `any`. In fact, any decent type theorist would probably murder me in my sleep were they to become public, so be aware that things will be a bit different after you leave this course.

- All non-function types are subtypes of `any`.
- `NoneType` (the type of `None`) is a subtype of all types except `int` and `bool`.
- A class type is a subtype of its parent type (if any).
- Function types and function-star types (i.e., those represented by `function_type` and `function_type_star`) are subtypes of `any`. Bound-function types are not (this is a kludge to explain a restriction on the use of bound functions (see §8.)
- Apart from `any`, a function type is not a subtype of any non-function types.
- A function type represented by

  `(`$V\ \ N\ \ R\ \ T_1 \cdots T_n$`)`,

  where V is `function_type`, `function_type_star`, etc. and R and T_i are types, is a subtype of

  `(`$V\ \ N'\ \ R'\ \ T'_1 \cdots T'_n$`)`,

  if

  - R is a subtype of R'.
  - For each i, either T'_i is `any` or T'_i is a subtype of T_i*.

  Any other combination of function types is erroneous. So,

```
class A(B):
   pass

def f(x): ...
def f0(x)::A: ...
def f1(x::A)::A: ...
def f2(x::A)::B: ...
def f3(x::B)::A: ...
def f4(x::B)::B: ...
```

*This is the part that could get me killed. There is not usually supposed to be an exception for `any` here.

```
# In the following, x1 must have a supertype of the right side's type
x1::(A)->B = f   # WRONG (return-type mismatch)
x1 = f0          # OK
x1 = f1          # OK
x1 = f2          # OK
x1 = f3          # OK
x1 = f4          # OK
x2::(B)->A = f0  # OK
x2 = f1          # WRONG (argument-type mismatch)
x2 = f2          # WRONG (argument and return-type mismatch)
x2 = f3          # OK
x2 = f4          # WRONG (return-type mismatch)
```

As usual, the "is a subtype (supertype) of" relation is reflexive and transitive.

Table 2: Method names for operators.

Unary operators	
Operator	**Method**
`+`	`__pos__`
`-`	`__neg__`
`~`	`__invert__`
`not`	`__not__`

Binary operators	
Operator	**Method**
`+`	`__add__`
`-`	`__sub__`
`*`	`__mul__`
`/`	`__div__`
`//`	`__div__`
`%`	`__mod__`
`**`	`__pow__`
`<<`	`__lshift__`
`>>`	`__rshift__`
`&`	`__and__`
`^`	`__xor__`
`|`	`__or__`

Container operators	
Operator	**Method**
`a[x]`	`__getitem__, __setitem__`
`s[i:j]`	`__getslice__, __setslice__`

Comparisons	
Operator	**Method**
`<`	`__lt__`
`>`	`__gt__`
`<=`	`__le__`
`>=`	`__ge__`
`==`	`__eq__`
`!=`	`__ne__`
`in`	`__contains__`
`notin`	`__notcontains__`
`is`	`__is__`
`is not`	`__isnot__`

Augmented Assignment	
Operator	**Method**
`+=`	`__iadd__`
`-=`	`__isub__`
`*=`	`__imul__`
`/=`	`__idiv__`
`//=`	`__idiv__`
`%=`	`__imod__`
`**=`	`__ipow__`
`<<=`	`__ilshift__`
`>>=`	`__irshift__`
`&=`	`__iand__`
`^=`	`__ixor__`
`|=`	`__ior__`

Notes.

a. `x in y` has the same effect as `y.__contains__(x)` (the arguments are reversed, in other words). Likewise, `x not in y` corresponds to `y.__notcontains__(x)`.

b. The `__notcontains__`, `__is__`, and `__isnot__` are not standard Python.

c. The `__set...__` methods are for targets (left sides of assignments) and the `__get...__` methods are for other contexts.

7 Special Methods for Operators

Several AST node types contain identifiers for operators: specifically, `unop`, `binop`, `comparison`, and `aug_assign`. All of these correspond to special operator names that are defined in the pseudo-class `any`. Thus, an expression such as `x+y` is treated *exactly* as if it were the method call `x.__add__(y)`. Accordingly, you should decorate the identifier node for `+` with the declaration index for `__add__`. Table 2 gives the equivalent method names for all the operators. As a result of this approach, there is nothing special you have to do to check the validity of operator expressions—the problem is essentially reduced to that of checking that calls are valid.

The standard prelude defines these methods in type `any`, so that (as in normal Python), you are free to write things like

```
x = y = 3
print x+y
```

and have it work. The default definitions of the operators are native methods that cause run-time errors.

8 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. All methods (defined by `def`s that occur immediately within a class definition) are instance methods (there are no static methods), and all therefore have at least one parameter. The first parameter of a method has the enclosing class as its static type. (The first parameter of a Python method corresponds to `this` in a Java program.)
2. An identifier that is defined as a class, function, method, constant, or module may not be assigned to in the same declarative region.
3. All variables defined to be local to module `__builtin__` are constants.
4. Except in the standard prelude, an inheritance clause in a class must reference a class completely defined previously in the program, and that class must be a subtype of the predefined class `object`. In the standard prelude, the inheritance clause for the type `any` is `any`.
5. In a call, the expression for the called function must have a function type (and in particular, not type `any`) that is *type-compatible with* the arguments to the call. A function type is type-compatible with a list of arguments iff
   - The number of arguments is the same as the number in the function type, or greater than or equal to that number for a `..._star` type.

   - For each of the formal parameter types in the function type, the corresponding argument has a (static) subtype that is *assignment compatible* to the formal type. We say that a type T is assignment compatible to T' if T is a subtype of T' or T is type `any`[†]. It is a consequence of the subtyping rules that bound function values are not assignment-compatible to anything, and so cannot be assigned to any variable, passed as parameters, or even used in tuples or lists.

6. 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 (thus becoming a local variable or instance variable), its name may not then be defined by **def** or **class** statements immediately within that same region (and vice-versa).

7. 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     # ERROR: static type of x is any, not A
A().b = 2           # ERROR: no b in class A instances
A.b = 3             # ERROR: No b in class A itself
x = A()
x.b = 2             # ERROR: static type of x (any) has no b.
```

   Your compiler must catch these errors.

8. If a class inherits a method, then it may override (redefine) that method only with another having a compatible signature. That is, a method M' may override a method M only if M''s type, minus the first parameter, is a subtype of M's type, minus the first parameter. It may not redefine an inherited method by assignment (i.e., as an instance variable). It may not define an inherited instance variable in any way (either by assignment or **def**).

[†]Because of this rule, an argument may have a dynamic type at execution time that is incompatible with T'. Therefore our system (in Project #3) will have to check and cause an exception if that is the case. Such is the price of dynamic typing.

9. The static type of the first parameter of a method is the enclosing class.
10. 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.
11. It is illegal to introduce a variable, parameter, function, method, class, or module named `None`.
12. All identifiers that are used must be defined.
13. The right-hand side expressions of an assignment must have types that are assignment compatible to those of the target (left side). Properly speaking, in an assignment such as

    ```
    x, y = 1, 2
    ```

    the right-hand side value is a tuple, whose elements have type `any`, so this kind of assignment will always work.
14. The expressions in a list display (`[...]`), tuple, or dictionary display (`{...}`) must all be subtypes of `any`.
15. The expression in a **return** statement must be assignment compatible to the return type of the enclosing function. A **return** statement without a value is equivalent to '**return** `None`'. A **return** statement with a list of values (or a single value followed by comma) returns a tuple.
16. The file argument to a **print** statement, if present, must be assignment-compatible to the standard-prelude type `file.`
17. If a variable (local variable, parameter, or instance variable) is ascribed a type using the '`::`' notation, it may not be ascribed a different type within its scope.

It follows from these rules that the indicated statements in the following Python code are all errors that the compiler must catch:

```
class A(object):
   z::int = 3
   def f(self, x)::int:
       return x

def f(c::A, y::int): return y

x = f
print x(A(), 3)        # ERROR: x has type any, not a function type.
x = A()
```

```
print x.z              # ERROR: x has type any, not a class type

g::(A,int)->any = f
print g(A(),3)         # OK.
g = A.f                # OK (A.f has a subtype of g's type).
y::A = A()
print y.z              # OK (y's static type A).
x = y.f                # ERROR: y.f has bound function type, not a
                       # subtype of any.
```

9 Predefined Names and the Standard Prelude

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__`. Unlike other modules, all of its definitions are visible in the module `__main__` (the one containing your program), as if you had a statement

```
from __builtin__ import *
```

at the top of your program (that is, if this were legal in our subset). Its definitions are visible in any module, unless hidden by a definition in that module

We will supply a file, `__builtin__.py`, containing our standard prelude (but a very small subset of what's provided in real Python, of course). You just parse and process it as for other modules, with the exception that certain restrictions don't apply to it. For your own testing purposes, you'll be able to use cut-down versions of `__builtin__`. The definitions from `__builtin__`—at least those used in your program—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.

You'll see prolific use of `native` methods in the standard prelude, which should come as no surprise.

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 [ --library=DIR ] FILE.py
./apyc --phase=2 [ --library=DIR ] FILE.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 directory *DIR* contains `.py` files for imported modules and for the special module `__builtin__.py`, which contains the standard prelude. To process imported module *M*, your parser should read and process *M*`.py`. You'll need to output only the AST for the main module (`__main__`), together with all the declaration nodes for `__main__` and for the modules you import (you don't output AST nodes for the imported modules).

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 and revising 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 partners' phone numbers 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. In fact, this project is structured in such a way that you can break it down into a set of small problems, each implemented by a few methods that traverse the ASTs.

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?