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

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

Project #2: Static Analyzer (revision 8)

Due: Wednesday, 9 November 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 §3.
2. Decorate each `id` node by adding a declaration index that links it to a declaration in the list. This is also described in §3.
3. Perform several rewrites of the AST, described in §4:
   (a) Rewrite all lambda expressions into explicit functions.
   (b) Rewrite allocation expressions to use new AST nodes that were not produced by the parser.
   (c) Rewrite method calls into ordinary function calls.
4. Enforce the language subset described in §6 (including the type inference rules 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.

Full 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 greatly restrict the language to give it static typing, and your project will infer those types in most places.

3 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 class definition, local variable, parameter, method definition, and instance variable. There are also declarations for the built-in types and functions. 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.

4 Rewriting

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

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 §5, 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. This is for unassignable values such as `None`, and is only used for predefined names. P is the declaration for the containing module (always `__builtin__`).
`(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 `(index_list` $m_1 \cdots m_n$`)`	A **def**ed function (including instance methods for this project, since we don't use inheritance) named I of type T, defined in a function, class, or module with declaration index P. The m_i are the declaration numbers of local variables, parameters, and local **def**s defined in the body of the function.
`(classdecl` N I P `(index_list` $m_1 \cdots m_n$`))`	Class declaration for class named I. The m_i are the declaration numbers of the class members of the class. They should be listed in order of appearance in the source text of the class. P is the declaration index of the module containing the class declaration.
`(moduledecl` N I `(index_list` $m_1 \cdots m_n$`))`	Module declaration for the module I. There are two of these, one for module `__builtin__`, which contains declarations for all the predefined names, and one for module `__main__`, which contains all the declarations for the user's program. The `index_list` gives the indices of declarations in the module, in the order they appear in the source (or alphabetical order for `__builtin__`).

4.1 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.

4.2 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_n`))`,

convert it to either the expression

`(new` N T`)`,

if $n = 0$ and class N has no `__init__` method, or else

`(call1 (id` N `__init__) (expr_list` N `(new` N T`)` E_1 ... E_n`))`.

if there is an `__init__` method, with an appropriate declaration index on the `id` node. It is an error if $n > 0$ without an `__init__` method. The new node `call1` is just like call, but returns the value of its first argument.

4.3 Attributes of classes

Whenever you encounter a node of the form

`(attributeref` N E_1 I`)`,

where E_1 denotes a known class that defines I (an `id` node), replace the `attributeref` 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.

4.4 Methods

During type resolution (see §7), you will encounter attribute references $E_1.x$ where E_1 is an expression (as opposed to a class), and the type of E_1 resolves to a specific class. When this happens in the context of a method call, $E_1.x(E_2, \ldots, E_n)$, and x is the name of a method in E_1's class, convert the expression to the ordinary function call $x(E_1, \ldots, E_n)$, decorating x with the appropriate declaration index for the method it names. It is an error if type resolution does not eventually compute a specific class as the type of E_1. It is also an error if this $E_1.x$ (again where x denotes a method) occurs in a context other than a method call. That is,

```
class A(object):
    def f(self):
        ...
x = A()
y = x.f    # ERROR: x.f denotes a "bound method" that is not
           #        called immediately.
```

5 Types

For this project, the possible types are either builtin types, user classes, or function types.

5.1 Type representation

Type variables, class, and function types are represented as in project #1:

`(type` N `(id` N *type-name*`) (type_list` N *types*`))`.

`(function_type` N *return-type* `(type_list` N *types*`))`.

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

(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, x::int)::bool: ...
x::A = A()
```

then the expression `A.f` has the type

```
(function_type 0 (type 0 (id 0 bool))
     (type_list (type 0 (id 0 A)) (type 0 (id 0 int)))
```

(the line-number attributes here are irrelevant).

Each identifier and expression has the *most general* static type that is consistent with the type rules of the language (§7). As discussed in lecture, the most general type is one that is compatible with all choices of types that obey the type rules and incompatible with all others. For example, the function

```
def id(x):
    return x
```

has type `($t)->$t`, since `id` can take any type of argument and returns a value of the same type. On the other hand, the functions

```
def sub(x,y):
    return x-y
def intid(z::int):
    return z
```

have types `(int,int)->int` and `(int)->int`, because '`-`' in our subset operates only on integers and the type rule for :: notations requires that `z` have the type `int`.

6 Various Restrictions

Our Python dialect is a rather violent restriction of Python designed, among other things, to make the language statically typed.

1. There is no inheritance. In class declarations, we'll simply ignore the supertype parameter.
2. For this project, we will also ignore `import` statements.
3. We restrict ourselves to the following types:
   - `int`.
   - `bool`.
   - `file`.
   - `str` (string).
   - `range` (type of `xrange`'s result. This is not the standard Python type name.)
   - `list(`T`)`: that is, lists all of whose elements have the same type.
   - `tuple(), tuple(`T_1`), tuple(`T_1, T_2`), ...`: These are tuples with known, constant numbers of elements having the types T_i.

- `dict(`K`,`V`)`: Maps from a type K to a type V. The type K is restricted to be `int`, `bool`, or `str`.
- User-defined classes.
- Function types, but *without* the trailing '`*`' arguments.

4. 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 type. (The first parameter of a Python method corresponds to `this` in a Java program.)

5. `class` and `def` statements declare constants, which may not be assigned to. 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.

6. Likewise, classes, methods, and functions may not be redefined immediately within the same declarative region (function, class, or file).

7. The only attributes of a class (things referenced 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. Thus, the only attributes of class C:

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

   are `a` and `f`.

8. The scope of parameters, local variable declarations (assignments to local variables) and **def**s that are nested inside other function bodies or 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 (so that, for example,

```
class A(object):
    x = 3
    def f(self):
        if self.x > 0:   # OK
            return x     # ERROR: x is unknown here.
```

   This is as in regular Python.)

9. The scope of outer-level declarations (those that are not nested inside a **def** or **class** declaration) begins with the declaration and continues to the end of the program (except where hidden). Thus, at the outer level, you may not use identifiers before their definition, so that the program

```
def f():
    print y
y = 3
```

is erroneous (`y` is used before it is declared by assignment.) However,

```
def g():
    def h():
        print y
    y = 3
```

is fine, because in this case, `y` is nested in `g`.

10. `None`, `True`, and `False` are predefined and those names may not be used in any other declaration (and therefore may not be assigned to).

11. All instances of identifiers in the program (other than in constructs that are ignored as indicated in items 1 and 2 above) must have known declarations.

12. The type rules §7 must successfully supply types for all (sub)expressions.

13. No bound method values, unless they are immediately called. See §4.4.

14. Classes may not be used as values. The only valid uses are for allocators, type designators (after '::'), or for fetching attributes (as in `A.f`). Builtin classes may not be used for allocators.

7 Type Rules

The language subset is chosen so that type inference can assign types to all expressions and statically check the validity of all constructs. A correct program obeys the rules in Figures 1 and 2, which are in the style of rules illustrated in Lecture 11. Be careful, these are definitely *not* the same as in ordinary Python, restricting or disallowing many expressions.

There are a couple of complications in applying the type rules:

- Some rules (such as that for '`+`') are overloaded to allow several different types of arguments, but not all. A simple extension of the type inference algorithm from lecture addresses this.

- Consider an expression such as `x.y`. Until we know the type of `x`, we cannot determine which method or instance variable to use for `y`, and therefore we don't know which type to use for it.

The solution is to use an iterative process to resolve types.

1. First, determine the declarations attached to all "outer" instances of identifiers (identifiers that don't occur immediately after a '.').

2. Assign a fresh type variable as the type of each variable, parameter, and **def**'ed name indicating that as far as we know initially, its type could be anything.

3. On each statement at the outer level of the program (including each **def** and **class**, repeatedly

   a. Perform type inference (described below).

   b. Find all qualified subexpressions, $E.x$, for which E's type is now known to be a specific class, and resolve x.

   until step b yields no further change. At this point, any remaining unresolved identifiers to the right of '.' are errors.

Type inference is as described in Lecture 11 with one extension to handle cases such as '`+`' or slicing. Implement type variables that can be restricted so that they match only particular types (such as `int`, `str`, and `list`) or (as usual) other type variables. When these are bound to non-variables, check that the binding is allowed. When bound to other type variables, change the set of allowed bindings to the intersection of the sets from the two variables. If this results in a type variable with an empty set of allowed bindings, it indicates a type error.

8 Running the program

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

```
./apyc --phase=2 -o OUTFILE FILE.py
./apyc --phase=2 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").

9 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,

Name	Construct	Type	Conditions
Lists	[]	list($a)	
	[$E_1, E_2, \ldots$]	list($a)	E_i: $a, for all i
Tuples	$(E_1, E_2, \ldots, E_n)$	tuple($T_1, \ldots, T_n$)	$E_i : T_i$, for all $1 \leq i \leq n$.
Strings	"...", r"...", ...	str	
Selection	$E_1[E_2]$	$a	E_1: $b, E_2: $c, where $b is either list($a), str, or dict($c,$a), and one of: $b is list($a) and $c is int, or $b is str, $a is str, and $c is int, or $b is dict($c,$a).
Slice	$E_1[E_2{:}E_3]$	$a	E_1: $a, E_2: int, E_3: int where $a is one of str or list($b).
Operators	E_1 + E_2	$a	E_1: $a, E_2: $a where $a is one of int, list($b), or str.
	E_1 * E_2	$a	E_1: $a, E_2: int, $a is one of int or str.
	$\pm E$, ~E	int	E: int
	$E_1 \oplus E_2$	int	E_1: int, E_2: int, where $\oplus$ is `-, /, //, %, <<, >>, &, |, ^`.
Comparisons	$E_1 \prec E_2 \ldots \prec E_n$	bool	where $\prec$ are comparisons.
	Furthermore, for each individual comparison:		
	E_1 [not] in E_2	bool	E_1: $a, E_2: $b, where $b is one of list($a) or dict($a,$c)
	$E_1 \prec E_2$	bool	E_1: $a, E_2: $a where $a is one of int, str, or bool, and $\prec$ is a comparison operation other than in, not in, `==`, or `!=`.
	E_1 == E_2, or E_1 != E_2	bool	E_1: $a, E_2: $a
Logical	E_1 and E_2	$a	E_1: $a, E_2: $a
	E_1 or E_2	$a	E_1: $a, E_2: $a
	not E	bool	E: $a
Call	$E_0(E_1, \ldots, E_n)$	$a	E_0:($a1,...,$an)->$a, E_1:$a1,...,E_n:$an.
Call1	__init__(E_1,...,E_n)	$a1	__init__: ($a1,...,$an)->$b, E_1:$a1,...,E_n:$an. This is used only by the special `call1` node described in §4.
Allocate	$C()$	C	where C is a class (this is the `new` node described in §4.)
Identifier	I	T	T is the declared type of I. If I is defined by a **def** and all type processing for I and its enclosing **defs** and **classes** (if any) is complete, then T has all type variables replaced by fresh ones.

Figure 1: Type rules for the subset, part I. In general, type variables $a, $b, etc., refer to fresh type variables for each instance of the construct. When a rule calls for "one of" several alternatives, type inference must determine unambiguously which of the alternatives applies.

Name	Construct	Type	Conditions
Constants	None	$a	
	True	bool	
	False	bool	
	stdout	file	
	stdin	file	
	stderr	file	
	argv	list(str)	
Predefined Functions	append(E_1, E_2)	$b	E_1: list($a), E_2: $a.
	len(E)	int	E: $a, where $a is one of list($b), dict($b, $c), or s
	open(E_1, E_2)	file	E_1: str, E_2: str.
	close(E)	$a	E: file.
	xrange(E_1, E_2)	range	E_1: int, E_2: int.
Assignment	$L = R$	$a	L: $a, R: $a.
For	for T in E: ...	—	E: $a, T: $b, where $a is one of list($b), range, or str and: $a is list($b) or $a is range and $b is int or $a and $b are str.
Control	while C: ...	—	C: $a
	if C: ...	—	C: $a
	elif C: ...	—	C: $a
	E_1 if C else E_2	$b	C: $a, E_1: $b, E_2: $b
	return E	—	E: T, where T is the enclosing function's return type.
Print	print $E_1, \ldots, E_n$[,]	—	E_i: $ai, $1 \leq i \leq n$.
	print >> F,$E_1, \ldots, E_n$[,]	—	F: file, E_i: $ai, $1 \leq i \leq n$.
Type declarations	x::T	T	x: T

Figure 2: Type rules for the subset, part II. The notation '—' in the Type column indicates constructs that have no value.

```
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.

10 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?