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

CS 164 **P. N. Hilfinger**
Fall 2013

Project #2: Static Analyzer (version 6)

Due: Monday, 18 November 2013

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 some 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` and `type_var` 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 allocation expressions to use new AST nodes that were not produced by the parser.
   (b) Rewrite method calls into ordinary function calls.
   (c) Add `id` nodes to operators (`binop, unop, comparisons, slice, substription`.)
4. Enforce the language dialect described in subsequent sections.

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 and type variables will have an extra annotation at the end:

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

This choice of declaration indices in identifiers or type variables must reflect the scope rules of the language: two occurrences of identifier or type variable I will have the same decoration iff they both are supposed to refer to the same thing. The scope of a type variable is the class or function that uses it in its type parameters or its parameter list or return type. Furthermore, the scope of type variables defined in a class header, like that of method names and instance variables, does *not* include any **def** statements in the class. That is, the definition of `$T` at line 1, below is not available inside of `f` and `g`.

```
class Foo of [$T]:       # Line 1
    def f(self, x::$T):  # $T does not refer to the same type as Line 1
        pass
    def g(self):
        x::$T = 3        # Illegal; $T from Line 1 is not visible
```

You may not otherwise introduce type variables. For example, this trivial program is illegal, because `$a` is not defined.

```
foo::$a = 3
```

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 in the standard prelude. Not all declarations appear in the outputted list; declarations attached to new type generated by the `freshen` method generally won't appear. As a result there may be gaps in the index numbers. 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 name `__main__` is not defined within your program, and references to it is an error, even though this module certainly exists and contains 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.

4.1 Identifying types

During parsing, you can't always tell which identifiers represent types. For example, the `A` in `A(3)` could denote either a type or a value. Rewrite any `id` node that is a type with a `type` AST node containing that `id` node (if it is not already part of one, that is.)

4.2 `__init__`

For every class that does not have an `__init__` method, add a default method:

```
def __init__(self):
    pass
```

4.3 Allocators

Whenever you encounter a "call" node whose first operand is a type (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 the expression

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

and decorate the `id` node with a declaration index as if this method had actually been written explicity. The new node, `call1`, is just like `call`, but returns the value of its first argument rather than the value returned by the `__init__` function.

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
`(vardecl` N I P T`)`	Variable named I. P is the declaration index of the enclosing function (or module, for a global variable). T defines its static type (see §6, below).
`(typevardecl` N I`)`	A type variable named I.
`(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.
`(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. The parameters come first, in the order they appear in the formals.

`(classdecl` N I
`(index_list` $p_1 \cdots p_n$`)`
`(index_list` $m_1 \cdots m_{n'}$`))`
	Class declaration for class named I. The p_i are the declaration numbers of the type parameters of the class (all type variables). The m_i are the declaration numbers of the members of the class. Each should be listed in order of appearance in the source text of the class.

`(moduledecl` N `__main__` `(index_list` $m_1 \cdots m_n$`))`	Module declaration for the module `__main__` (the only one in our project). The `index_list` gives the indices of declarations in the module, in the order they appear in the source.

4.4 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) as a method, replace the `attributeref` with I, after assigning the appropriate declaration index to I. Thus, after the Python class declaration

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

The statement

```
g = A.f
```

becomes, in effect,

```
g = f
```

but with `f` decorated with the appropriate declaration of method `f`. It is an error for E_1 to denote a type that is not known to define I. E_1 can also be a parameterized type (as in `PriorityQueue of [Int].push`, but the type parameters are ignored.

4.5 Methods

During type resolution (see §8), you will encounter attribute references $E_1.x$ where E_1 is an expression having a value (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.
```

4.6 None

Rewrite all occurrences of the identifier `None` as the function call `__None__()`. Report an error if `None` is defined by `def`, `class`, or is assigned to.

Table 2: Operators and their associated function names. The standard prelude (§9) will contain definitions of these functions. For now, we translate subscription and slicing the same whether they appear as an assignment target or not. We'll deal with the difference for targets in a later project.

Operator	Function	Operator	Function
`+ (binary)`	`__add__`	`·[·] (subscription AST)`	`__getitem__`
`- (binary)`	`__sub__`	`·[·:·] (slicing AST)`	`__getslice__`
	`__mul__`	`<`	`__lt__`
`/, //`	`__floordiv__`	`>`	`__gt__`
`%`	`__mod__`	`>=`	`__ge__`
`*`	`__pow__`	`<=`	`__le__`
`- (unary)`	`__neg__`	`==`	`__eq__`
`+ (unary)`	`__pos__`	`!=`	`__ne__`
`not`	`__not__`	`notin`	`__notcontains__`
`in`	`__contains__`	`isnot`	`__isnot__`
`is`	`__is__`		

4.7 Operators

The skeleton has already incorporated this rewrite; it's here for your information. In real Python, the binary operators, unary operators, comparison operators, subscription, and slicing operators are closely related to certain methods (with names of the form $\_\_N\_\_$). Simillarly, we're going to handle these operators—type rules and all—using the same rules as for procedures. The standard prelude (see §9) will contain a bunch of definitions of functions to be called to evaluate these operators. To each of these operators, we've added an additional last child—an `id` node—containing the name of the appropriate method, as given in Table 2, which we use as the function and resolve using the usual rules for resolving function calls. (Well, not quite the same code, perhaps, since the function called and the arguments will be in different locations in the tree from an ordinary call, but proper use of virtual methods can allow you to use the same logic.)

5 Overloading

We're extending Python to allow overloaded functions (and as a result, operators). The rule is that there can be multiple definitions of functions (including methods) within a declarative region, thus overloading them. It is an error to attempt to overload any other kind of entity other than a function.

Overloaded names are disambiguated on the basis of type rules. For example, this is legal in our dialect:

```
def f():
    print "f()"
def f(x):
    print "f(x)"
```

```
f(3)
```

To keep things simple, we'll still say that *any* declaration of a function in one region hides all those in enclosing regions, so that the following is illegal, even though no definitions of `f` inside `g` can possibly satisfy the call:

```
def f():
    pass

def g():
    def f(x):
        pass
    f()       # ERROR, the outer f is not visible.
```

Since operator expressions are effectively converted into function calls (see §4.7), it follows that operators can be overloaded as well.

6 Types

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

6.1 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 type-name*`)`

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

```
class A:
    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 0))
     (type_list 0 (type 0 (id 0 A) (type_list 0))
                  (type 0 (id 0 int) (type_list 0))))
```

(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 (§8). 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`.

7 Various Restrictions

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

1. 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.
   - `tuple0(), tuple1(`T_1`), tuple2(`T_1, T_2`), tuple3(`T_1, T_2, T_3`)`: These are tuples with known, constant numbers of elements having the types T_i. Yes, we only do the ones up to 3, but that's enough to make the point.
   - `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.
2. 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.)
3. `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.

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

5. 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`.

6. 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.)

7. 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`.

8. All instances of identifiers and type variables in the program other than the identifiers denoting operators (like `+`) in `binop`, `unop`, `compare` and `compare_left` nodes must have known declarations.

9. The type rules §8 must successfully supply types for all (sub)expressions. Furthermore, for each complete statement at the outer level, there may be no free, unbound type variables; types of global variables must be completely determined by the statement that assigns them.

10. No bound method values, unless they are immediately called. See §4.5.

11. 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.

8 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, which are in the style of rules illustrated in Lecture 12. Be careful; these are definitely *not* the same as in ordinary Python, restricting or disallowing many expressions. Most of the things missing from the table are handled by the rules for calls

There is a notable complication in applying the type rules. 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 (see Lecture 12). If an identifier 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. This same rule applies to the variable `None`. But in all other cases, the declared type T is used without change. As a result, the sequence

   ```
   x = 3
   x = "foo"
   ```

   is an error.

   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.

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$n$ $(T_1, \ldots, T_n)$	$E_i : T_i$, for all $1 \leq i \leq n$, where $0 \leq n \leq 3$.
Strings	"...", r"...", ...	str	
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
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. See §8 for details.
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, $1 \leq i \leq n$.
Typed ids	x::T	T	x: T

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. Type rules for other operators (see §4.7 and Table 2) fall out from the rules for calls.

9 The standard prelude

The term *standard prelude* refers to the definitions of all built-in names in a language. In our case, these can be described by a set of ordinary declarations in our Python dialect, and handled with (mostly) the same rules (I say "mostly" because some built-in types have special significance in the language; type `str`, for example, is the type of string literals.) Your program will, in effect, prepend a standard prelude that we supply to the rest of your program. If you use your own code, you'll need a little more logic in the lexer to handle this, but it really helps avoid a lot of tedious code setting up predefined names.

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 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").

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?