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

CS 164 **P. N. Hilfinger**
Spring 2018

Project #2: Static Analyzer

Due: Friday, 6 April 2018

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. Rewrite allocation expressions to use new AST nodes that were not produced by the parser.
4. Enforce the language dialect described in subsequent sections.

The remaining sections describe these in more detail.

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. We have collected differences between Python 2.5 and our dialect in
`https://inst.eecs.berkeley.edu/~cs164/sp18/hw/dialect.pdf`.

2 Input and Output

You can start either from a parser that we provide, or you can augment your own parser from Project 1. 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.

The program will run phase 1 (the parser) phase twice: once to parse the source program, and once to parse the *standard prelude,* which defines the built-in functions and classes and is included in the skeleton.

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` and `type_var` 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
```

The programmer 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, type variable, method definition, and instance variable. There are also declarations for the built-in types and functions in the standard prelude. Only the (single) module declaration and declarations reachable from the AST need to appear in the outputted list. 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 are errors, 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), the program must perform several rewrites. Some of these may already have been done in the first project (as in our solution), or are already done in the skeleton.

4.1 Initial "post-phase 1" rewrites

There are some initial rewrites that we have done for you, since they are straightforward. The skeleton file `proj2/src/post1.cc` implements these:

1. The AST from parsing the standard prelude is inserted at the front of the module containing the source program. This is to avoid having to deal with module constructs in this project.
2. All "block" ASTs are removed and their children inserted directly into the surrounding class or function definition. (Our project 1 solution does this already.)
3. All "type_list" ASTs are removed and their children inserted directly into the surrounding "type" or "function_type" node. (Our project 1 solution does this already.)
4. All function definitions in which the return type is defaulted are supplied with a new type variable as the return type. This gives an explicit return type without beihg explicit about what it is. Thus, the function header

   ```
   def f():
       ...
   ```

   might become

   ```
   def f()::$#12:
       ...
   ```

(we use `$#`n to notate anonymous type variables produced by the compiler as opposed to being written by the programmer.) (Our project 1 solution does this already.)

5. Likewise, all formal function parameters that do not have explicit types are also given anonymous type variables, so that every parameter has an explicit formal type. For example:

   ```
   def f(x):
       ...
   ```

   might become

   ```
   def f(x: $#14)::$#15:
       ...
   ```

6. All "def" nodes that represent methods in a class are changed to "method" nodes (with the same format). This makes it a little easier to write code that handles the slight differences between them.

Again, we've written these transformations for you. The rest you supply.

4.2 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.)

4.3 Allocators

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

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

convert it to the expression

(`call1` N (`id` N `__init__`) (`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. If a class does not have an `__init__` method, one can't allocate anything with it using the allocator syntax. Hence, for user-defined classes, you'll usually want to explicitly include an `__init__` method (it is not defaulted). This means that one cannot allocate using the built-in types. You will not need to check specifically that a given type is a user-defined class, however, since if it isn't, the `__init__` method won't be defined, which should cause an error without any special provisions.

4.4 Attributes of classes

A node of the form

`(attributeref` N E_1 I`)`,

corresponds to $E_1.I$. When E_1 denotes a known class that defines I (an `id` node) as a method, assign the appropriate declaration index (indicies in the case of an overloaded method) to I. Thereafter, E_1 itself is irrelevant. 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 in that case (since methods are not included in the scope of E_1's type parameters).

If E_1 denotes a class and I denotes an instance variable rather than a method, the attribute reference is erroneous in our dialect.

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 within a declarative region, thus overloading them. It is an error to attempt to overload any other kind of entity 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.
```

As you saw in Project 1, operator expressions are converted into nodes with the same format as function calls, and in fact are treated just like function calls in the semantics. The function names in these expressions (e.g., `__add__`) are defined by ordinary `defs` (usually in the standard prelude) and can therefore 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, but with `type_list` nodes replaced by their children, as described in §4.1:

> `(type` *N* `(id` *N* *type-name*`)` *types*`)`
>
> `(function_type` *N* *return-type* *argument-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 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 (§??). 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 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.) The rewrite that we supply (see §4.1) prepends the AST for the standard prelude to the rest of your program.

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 SOURCE.py
./apyc --phase=2 SOURCE.py
```

We've extended the main program `apyc` to call phase 1 (the parser, previously written) phase 2 programs. The command lines from project 1 will 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 file is *SOURCE*`.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 the subdirectory, `proj2`. This should contain a file `Makefile` that is set up, again as in the skeleton we supply, so that

```
make
```

(the default target) compiles your program;

```
make check
```

runs all your tests against your program; and finally,

```
make APYC=PROG check
```

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

```
make clean
```

should remove all files that are regeneratable or unnecessary. The Makefiles in the `proj2` skeleton do all this. You may have to modify them to make them continue to work as required in the face of certain changes you make.

10 Using Git to Get Started and to Submit

If you have set up your local copy of your GIT team repository as in our setup directions, one member of your team can start your project from this point with the command:

```
$ cd team-repo
$ git fetch shared
$ git checkout -b proj2 shared/proj2
```

(As usual, first be sure that you have first committed any changes in the current working directory.) This starts and checks out a branch called `proj2` and puts a single subdirectory called `proj2` in it. Alternatively, if you would prefer to keep all your projects in your team's master directory, use instead

```
$ cd team-repo
$ git fetch shared
$ git merge shared/proj2
```

which simply adds our `proj2` directory to what's already there.

The initial skeleton includes our solution to `proj1`. You can replace it or parts of it with your own version as you see fit.

Should we make modifications to our skeleton, you can incorporate them into your files (if desired) with the following steps, assuming you are in your `proj2` branch:

```
$ git commit -a    # If needed.
$ git fetch shared
$ git merge shared/proj2  # To incorporate get changes to proj2 skeleton
```

We will test your program by first using it to translate a suite of correct Python programs (checking that your program exits cleanly with an exit code of 0), and we will check the translations by unparsing them (using `pyunparse` with switches that check that you've gotten the declarations right), running the resulting Python programs, and checking their output. Next, we will run your program against a suite of erroneous programs, and check that you produce appropriate error messages (their contents are not important as long as the form is as specified) and that your program exits with a standard error code (as produced by `exit(1)` in C++).

Not only must your program work, but it must also be well documented internally. At the very least, we want to see *useful and informative* comments on each method you introduce and each class.

11 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 start immediately by augmenting your project 1 tests. We won't give credit for tests that are simply duplicates of ours.

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 Git 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?

Table 1: Declaration nodes. The list of the declaration nodes for a program in order by index follows the AST for the program. In each case, N is the declaration index, unique to each declaration node instance. T is a type (represented as usual by an AST).

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 of which are 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.