PyPE: a Python Pipeline Engine, Part 2

Part 2: Analysis Passes

- Skeleton Updates
- Semantic Analysis
  - Walking an AST
  - Single-Assignment Checking
- Symbol Tables
  - Importing External Libraries
  - Populating the Symbol Table

Skeleton Updates

For today's lab, you'll need a couple new files. We've put these files in the same skeleton repository as before, and you'll need to grab them before continuing: https://github.com/rdadolf/pype-skeleton (https://github.com/rdadolf/pype-skeleton)

Because you've (hopefully) already worked on the previous lab, you don't want to copy over all the files (because you'd overwrite your work). These are the new files you'll want to copy over:

- `lib_import.py`
- `symtab.py`
- `translate.py`

In addition, there's a couple small tweaks you'll need to make to your existing files:

- We've added a second method to the `ASTVisitor` class: `return_value()`. This will simplify non-analysis passes on your AST traversals somewhat. You can just copy the definition of the method into your `ASTVisitor` class.

- You'll need your `ASTNode.walk()` function to return the result of this new value. So the last line of `walk()` should now be `return visitor.return_value()`. The `ast.py` in the skeleton has been updated to reflect these two changes. (But remember not to copy `ast.py` because you'll overwrite your changes!)
- You'll also want the new `pipeline.py` and `semantic_analysis` files. You already have one of these, but the new versions have minor updates for today's lab. If you haven't changed the reference version, you can just copy the new version.

Back to top ↑

Semantic Analysis

As we discussed in class, semantic analysis is the process of understanding and enforcing rules about the behavior of a language. Today we'll be implementing a very simple semantic check to enforce a language invariant which the parser cannot do.

As we mentioned in the first lab, PyPE is a declarative dataflow language. One corollary of this is that variables cannot be re-assigned. So this statement, while syntactically correct, is meaningless:

```
{ multiple_assignment
  (input x)
  (:= y 1)
  (:= y 2)
  (output y)
}
```

An easy way to see why this is a bug is to recall that the order of expressions in PyPE is irrelevant (as in most declarative languages). So this program is equivalent to the one above:

```
{ multiple_assignment
  (input x)
  (:= y 2)
  (:= y 1)
  (output y)
}
```

If we allowed multiple assignment, these two "equivalent programs" would actually produce different results—a *Bad Thing*™.

Today, we'll let our compiler check for this condition and raise an error.

Back to top ↑

Walking an AST

In the previous lab, you implemented a `walk()` function in your AST class. As we talked about in class, most modern semantic analysis is done using this technique. To get familiar with, let's implement a version of the AST printer using an AST walker.

In `semantic_analysis.py`, there's a stub for a `PrettyPrint` class. All we want you to do is print out the class name of every node in the AST as it's traversed. If you recall, given some object, you can get a string of its class name: `obj.__class__.__name__`. (The ability to get information about an object's type programmatically like this is usually called "reflection". If you were curious.)

To run an AST walker, you need two things: an AST, and an *instance* of the visitor you want to run. It will look something like this: `ast.walk( YourVisitor() )`

Note: this part of the lab is just as simple as it sounds. You're literally writing a single line which does nothing tricky. The main goal here is just to get familiar with the interaction between an `ASTVisitor` and the `walk()` method.

Back to top ↑

Single-Assignment Checking

Now let's write a semantic analysis pass for our compiler. As we said above, the goal is to enforce a behavioral *invariant* about our PyPE language, namely:

Within a single component, two assignment expressions should never bind an expression to the same name.

Implement a visitor class (`CheckSingleAssignment`, in `semantic_analysis.py`) to enforce this. If you detect a double assignment, `raise` an exception with an appropriate error message.

A couple of hints regarding how to do this:

(1) Since you only have a single `visit()` method, you're going to have to make it do different things depending on what type of node it's visiting. This is basically *why* our AST class has so many subclasses—so we can distinguish semantically distinct objects and treat them accordingly. Your `visit()` function will likely end up with a bunch of `if isinstance(node, ...)` statements. (Side note: while it might seem plausible, you cannot use the `@singledispatch` approach here. You can prove this to yourself by looking at its implementation (https://hg.python.org/cpython/file/f6f691ff27b9/Lib/functools.py#l707). It would be possible to implement your own decorator to do something similar in this case, but it's probably more trouble than it's worth, and it diminishes readability.)

(2) It's absolutely okay for two *different* components to assign the same variable. So this code should not raise an error:

```
{ comp1 (input) (output) (:= a 1) }
{ comp2 (input) (output) (:= a 2) }
```

This means that your visitor is going to have to track which component it is currently in.

Back to top ↑

Symbol Tables

In class, we talked about how compilers are based around iterated transformations to analyze, simplify, augment, and optimize code. The "augment" part of that often takes the form of auxiliary data structures built as part of some pass. These auxiliary structures can then be reused over and over.

A symbol table is one such structure which centralizes information about all the names in a program. In its simplest form, a symbol table maps names to a set of attributes. For instance, in a typed language, a symbol table entry might contain the type of a variable as one attribute.

Look in `symtab.py`. Near the top you'll see our definition of a `Symbol`. If you're not familiar with named tuples, you can check the docs here: collections.namedtuple (https://docs.python.org/3/library/collections.html#collections.namedtuple) Basically, a

named tuple is just an object with named attributes that can either be accessed as attributes or by index. Just like normal tuples, they are immutable.

```
Symbol = collections.namedtuple('Symbol','name type ref')
s = Symbol('my-name','my-type','my-ref')
# These all reference the symbol's name:
print(s.name, s[0], s['name'])
```

Additionally, we define a set of symbol types which will describe what kind of object the name was declared as. This uses the enum.Enum (https://docs.python.org/3/library/enum.html) class, which is just a concise way of saying "values of this class can only take one of a small set of values, which have these names." Underneath, most enumerations are just integers with fancy names. So in our case, a `SymbolType.component` has the value 1, `SymbolType.var` has value 2, and so on. When we say, `x = SymbolType.var`, in essence we're saying `x = 2`, but Python keeps track of some extra information so that later we can say `if x==SymbolType.var`. Enumerations make your code more readable than using plain integers.

One last wrinkle is that PyPE programs have scopes. Just like we saw in our single assignment checker, the same name can be declared in two different components. This means that a PyPE symbol table actually has to be a collection of tables, each corresponding to a different scope. (In practice, most languages require even more complicated organizations to deal with symbols, since they usually support things like nested scopes, static values, and/or shadowing.)

Looking in the `SymbolTable` class, you see that its initializer creates a dictionary (`T`). This table maps scope names to sub-tables which actually contain mappings from names to `Symbol` tuples. You'll also see that we've pre-initialized the `global` scope, which is for names that are visible program-wide (like component names and external library functions imported from Python modules).

We'd like you to implement the `addsym` method. This should be straightforward: all you're doing is adding an entry for a symbol into the appropriate scope table. For instance, someone could call:

```
symtab = SymbolTable()
symtab.addsym( ('varname', SymbolType.var, None), 'component-name' )
```

And a symbol for 'varname' should appear in the table for `component-name`.

This is not the tricky part. It's just to get you acquainted with the data structure.

Back to top ↑

Importing External Libraries

Our compiler will be *far* more useful if we can use functions implemented in Python as pieces of our pipelines. To do this, we're going to use some magic to turn a PyPE import statement `(import module_name)` into symbol table entries for selected functions within the Python module `module_name`.

The first problem is how to tell PyPE *which* functions we want to import. We'll do this with a decorator. Write the `component` decorator which adds a `_attributes` dictionary to a function and adds a key `'_pype_component'` to it with the value `True`. So this should print `True`:

```
import pype
@pype.component
def external_function(x):
  ...

print(external_function._attributes['_pype_component'])
```

We'd also like you to write a `is_component` which takes an object and determines whether it was decorated with your `component` decorator. So, like before, this should print `True`:

```
import pype
@pype.component
def external_function(x):
  ...

# Note that this is the function itself, not a string:
print(is_component(external_function))
```

Now look at the `LibraryImporter` class. This is a little tricky, since we use a library called `importlib` to dynamically import Python modules on demand. Basically, `importlib` allows us the write an normal Python `import` statement, except with a string as an argument. So

these two statements are effectively the same:

```
import timeseries as ts
ts = importlib.import_module('timeseries')
```

Then, we get a list of all of the elements of that Python module using the `inspect` module. This is *very* similar to the `dir` function you're already used to (in fact, it's a wrapper around it). If you'd like to understand how it works, check its documentation (https://docs.python.org/3/library/inspect.html).

Now we do two things:

1. We iterate through all the top-level objects in the module and check if any are functions AND they have been decorated with our `component` decorator.
2. We iterate through all the classes in the module, then iterate through their methods and check for the decorator again.

For each of these, we need to add globally-scoped symbols to the symbol table.

Finally, go ahead add a `mean` and `std` function to your `TimeSeries` class and decorate them with the PyPE component decorator. (`mean` should compute the mean of the values of a time series, and `std` should compute the standard deviation. You can use numpy functions to do this.)

Back to top ↑

Populating the Symbol Table

Now we're all ready to generate a symbol table from an AST. As you've probably guessed, since we're operating on the AST, we'll be writing another `ASTVisitor` class. We've given you a stub for this in `translate.py` .

Your job is to flesh out the `SymbolTableVisitor` class. This should be very similar to the checker you wrote earlier, and you'll need to pull some of the same tricks (scope-tracking and class type checking) as well as some new ones (adding symbols).

Once you're done, you should be able to call the `SymbolTable.pprint()` method to print out your symbol table. (Note that you'll need to be able to import your timeseries class to get this to work. You should've had this done in Monday's lab.)

We've provided you with reference output in the samples directory to make sure you're on the right track.

Back to top ↑