IACS CS 207 | Systems Development for Computational Science

PyPE: a Python Pipeline Engine, Part 1

Part 1: The Front End

This is a group project. Please work with your team.

There is no google form for this. It will be graded along with the rest of your codebase at the course Milestones.

Overview

A Domain-Specific Language (DSL) are a way of trading off generality for expressive power. The basic idea is to create a restricted language that can only really express a few tasks, but to make it fast and easy to complete those tasks.

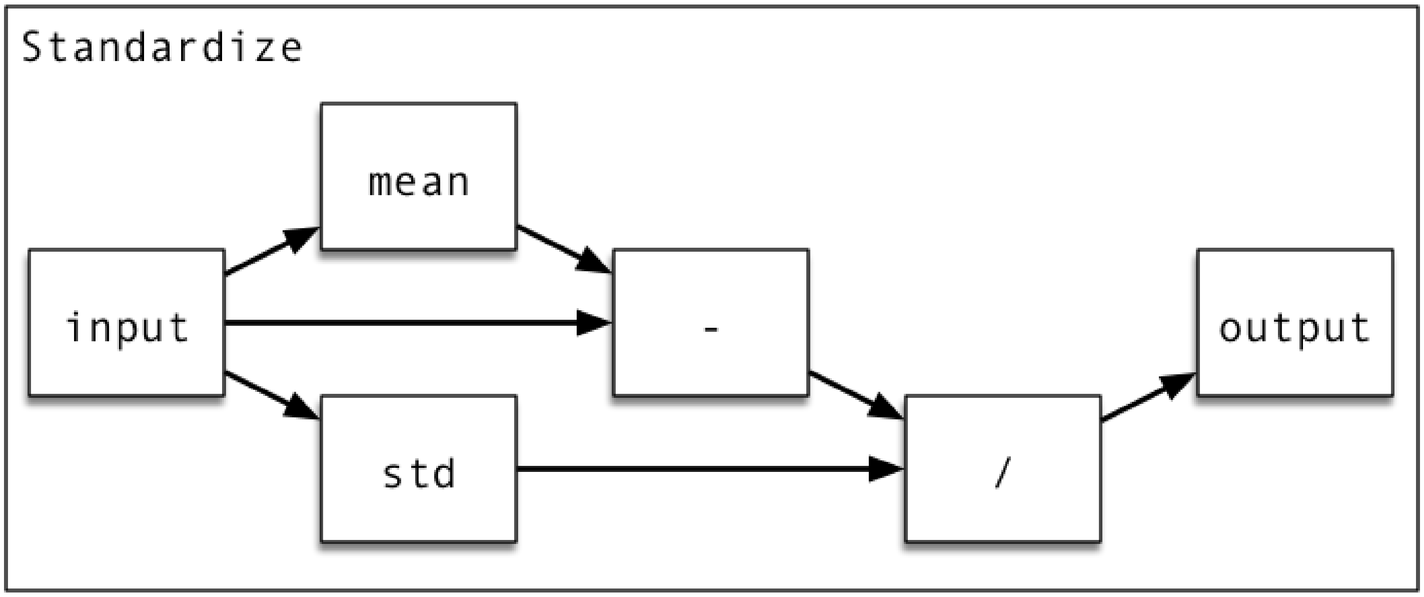
In the past couple weeks, you've had some exposure to simple DSLs: the infix calculator language from the tree lectures, and the `stupidlang` from the packaging lecture. These languages took care to step around some of the tricky areas that crop up in real compilers, but today you'll be attacking those areas head-on.

Today you'll be building the front-end of a compiler for PyPE, a DSL built around the idea of computational pipelines. Eventually, we'll be hooking this up around your TimeSeries class, and you'll be able to build a distributed computation system with it. PyPE is a declarative dataflow language. If you recall from the programming methodology lecture, declarative programming means that statements describe *what* should be done, not necessarily *how* it should be done.

Here's a simple example of a PyPE program:

```
(import timeseries)

{ standardize
  (input (TimeSeries t))
  (:= mu (mean t))
  (:= sig (std t))
  (:= new_t (/ (- t mu) sig))
  (output new_t)
}
```

At the highest level, PyPE programs are built from *components*. Components represent reusable computational pipelines, and they serve a similar purpose to functions in Python. In the example program, we have one component named `standardize` , and we enclose it in braces. A component is built out of expressions, which represent the actual operations that it carries out. You can think of these expressions as nodes in a computational graph: incoming edges to a node represent the use of some input, the node represents the function which computes a value, and outgoing edges represent the use of that value in other expressions (nodes). It might look like this:



So the expression `(:= mu (mean t))` , for instance, is shown in the graph as the node labeled `mean` . It uses data from the `input` node, and it produces output for the subtraction node.

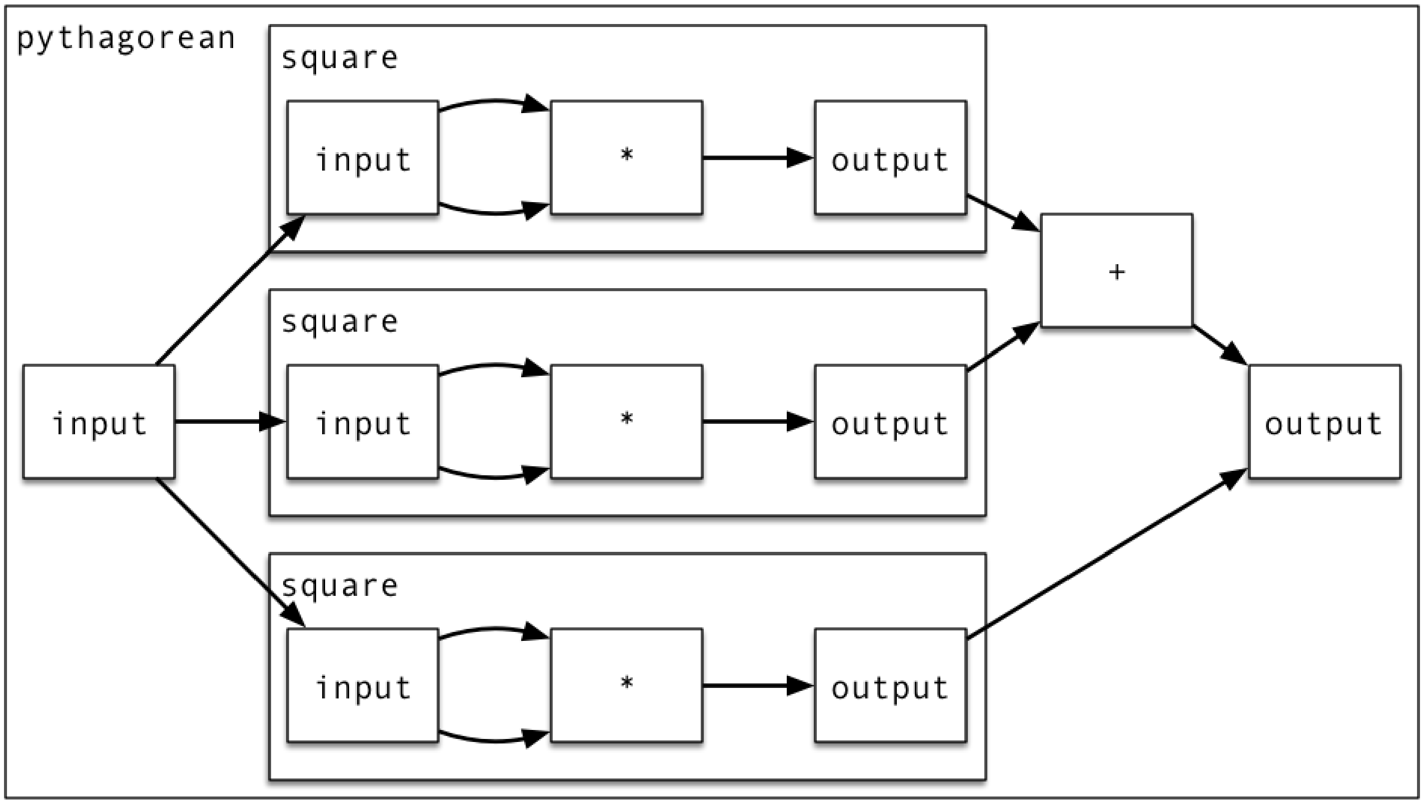
Because PyPE is declarative, this is an equivalent version of the program above:

```
(import timeseries)

{ standardize
  (:= new_t (/ (- t mu) sig))
  (:= mu (mean t))
  (:= sig (std t))

  (input (TimeSeries t))
  (output new_t)
}
```

PyPE also supports composition, so if you write a component, you can use it in other components. For instance:

```
{ square
  (input v)
  (:= new_v (* v v))
  (output new_v)
}

{ pythagorean
  (input a b c)
  (:= rhs (+ (square a) (square b)))
  (:= lhs (square c))
  (output lhs rhs)
}
```

Here, the `square` component is reused several times, and each time, the internal values in the component are bound to different inputs. The graph for this might look something like this:



Since you're dealing with a much more complicated language, you'll be using higher-powered tools today. For the lexer and parser, you'll be using a Python library called Ply (http://www.dabeaz.com/ply/), which is itself a port of the industry-standard tools lex (http://flex.sourceforge.net/) and yacc (https://www.gnu.org/software/bison/). These tools have a learning curve, so we encourage you to use Ply's documentation (http://www.dabeaz.com/ply/ply.html) and any other resources you can find on the web.

One last note before we begin: this lab is *long* (if you hadn't realized). Because it's a team project, we encourage you to be proactive about distributing the workload.

Back to top ↑

Downloading the Skeleton

Language design and compiler construction are hard problems, and some of the design decisions that go into building them are beyond the scope of this class. (For instance, choosing a language structure which lends itself to unambiguous grammars, or deciding on a restricted type system which can still support powerful composite operations.) As a result, we're going to give you a skeletal compiler. For each piece, you'll have more or less of a headstart, depending on how hard the task is. For instance, for your parser, we're giving you a full BNF definition of the PyPE language, along with some example production rules. For your lexer, however, you'll just get a list of tokens and a very empty-looking file. (Parsers tend to be much more involved than lexers.)

You can download the skeleton here: https://github.com/rdadolf/pype-skeleton (https://github.com/rdadolf/pype-skeleton)

You'll also need to install ply. You should be able to use `conda` or `pip` . Your choice:

```
conda install ply
```

or

```
pip install ply
```

Back to top ↑

Building a Lexer

Before you write anything, start by reading at least sections 4 and 4.1 of the Ply documentation: read this first (http://www.dabeaz.com/ply/ply.html#ply_nn3). It would not a bad idea to copy their calclex.py file into a scratch directory and make sure you can get it to work.

Now look at the `lexer.py` file in the skeleton. At the top of the file, you'll see a list of tokens. Your first job is to implement production rules for each token. Many of these will be simple literals (for instance, `LPAREN` , `RPAREN` , ect.),

If you need a guide, try section 4.3 (http://www.dabeaz.com/ply/ply.html#ply_nn6). If you aren't comfortable with simple regular expressions, you can try looking at the Python HOWTO (https://docs.python.org/3/howto/regex.html).

It's worth noting that Ply operates a little differently than most python libraries. Ply implicitly builds your lexer by scraping the file and finding all of the variables and functions which start with `t_` . So writing a production rule for the `LPAREN` token means either defining a variable or function named `t_LPAREN` .

If you go the variable route, you'll need to assign it a regular expression which tells ply what to match this token to. For instance:

```
t_LPAREN = r'\('
```

If you go the function route, it gets a little weird. The first aspect is that the function needs a regular expression *as its docstring*. This regex will define what the token matches. The rest of the function defines what your lexer will *do* with your token. It would look like this:

```
def t_LPAREN(t):
  r'\('
  # do something with t, if necessary
  return t
```

The function form is generally used when you need to do something more complicated with the token. For instance, in Ply's `calclex.py` example, they convert their `NUMBER` tokens into actual numeric values instead of strings.

The rest of the functionality you need to write is in the skeleton file.

If you get stuck, *consult the documentation*. This is your first line of defense, and basically everything we're asking you to do has descriptions in this file. Also, it may help to add `debug=True` as an argument to the ply.lex.lex() function call in `lexer.py` —this will print out a bunch of debug information.

If you need to test the lexer independently, try dumping the list of tokens like they do in section 4.1.

Back to top ↑

Building a Parser

Next up is constructing a parser. Copy over the `parser.py` file and take a look. Again, keep the ply documentation (http://www.dabeaz.com/ply/ply.html#ply_nn22) handy. You will almost certainly need it.

Back to top ↑

Understanding the EBNF Specification

As we discussed in class, formal grammars can be tricky. We won't make you devise your own BNF specification for PyPE, but deterministic context-free grammars are somewhat central to *why* the production rules look the way they look. Take for example the production rule we supply you for `statement_list`:

```
r'''statement_list : statement_list component
                   | statement_list import_statement
                   | import_statement
                   | component'''
```

Why not say this?

```
r'''statement_list : statement_list statement_list
                   | component
                   | import_statement
                   | NULL'''
```

The answer is that it's ambiguous. There's more than one way to get several strings. For instance, there are an infinite number of ways of producing the empty string. When writing your parser, if you run into error messages from Ply complaining about infinite recursion, this is what they're talking about.

Back to top ↑

Writing parser rules

Note that because we've already given you a complete formal language specification, you don't need to add or remove any of the BNF production rules. Still, we need to talk about what parser rules actually *do*.

Up to this point, our compiler has largely centered around recognizing symbols and statements. You had a little taste of rules which had actions in the rules for `t_ID` and `t_newline` in your lexer, but the parser will go much further.

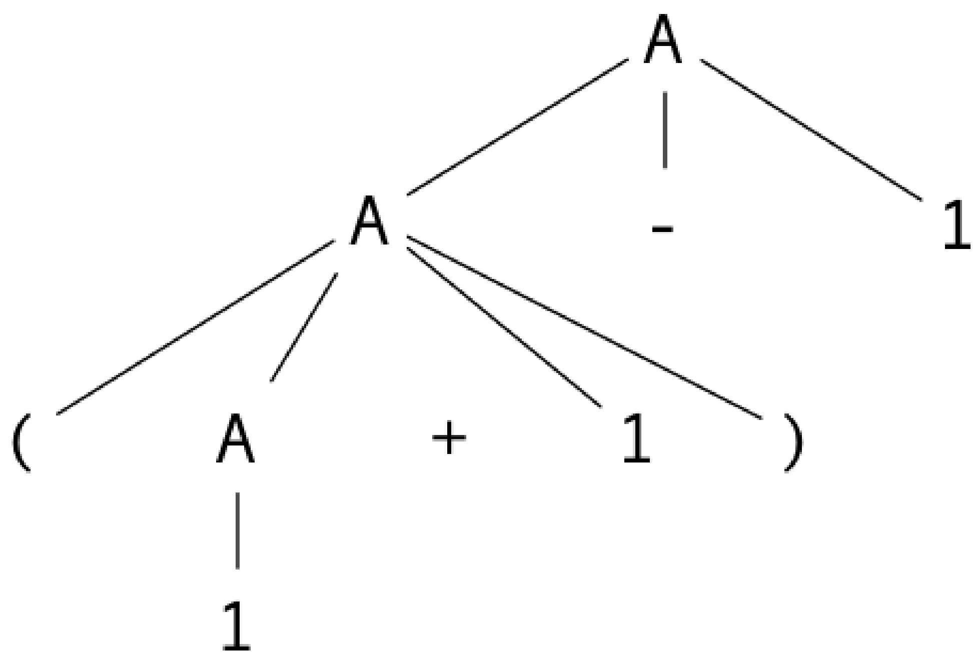
Typically, a parser's job is to convert a concrete syntax tree into an abstract one. A concrete syntax tree is a tree where the leaves are made up of lexer tokens and the internal nodes represent production rules from our formal grammar. As an example, given a language consisting only of the single production rule:

```
A := ( A + 1 ) | ( A - 1 ) | 1
```

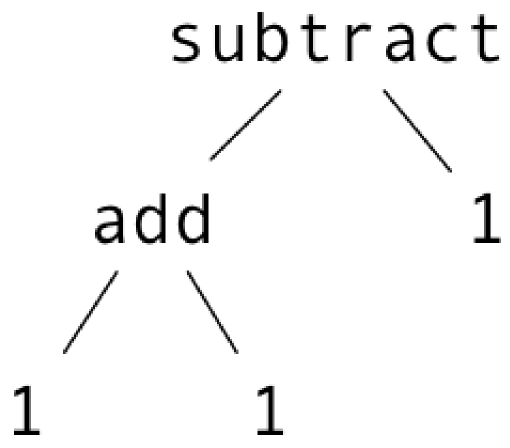
and the input string

```
( 1 + 1 ) - 1
```

the concrete syntax tree might look something like this:



Of course, the tricky part is that we typically never actually construct the concrete syntax tree. It's embedded in the production rules, but instead of actually generating it, we execute small bits of code in each production rule to produce a reduced form (without all the syntactic fluff).



This is the abstract syntax tree.

Back to top ↑

AST Construction

AST Data Structures

Back to top ↑

The first component of an AST is a tree data structure. This will be very similar to the trees you've seen in previous lectures. We've given you a basic tree node class called `ASTNode` in `ast.py`.

The first thing to note is that we've asked you to implement a rather large number of subclasses. This is somewhat of a well-known characteristic. In fact, ASTs are often held up as an example of how type-specialization can make a night-and-day difference in software maintainability.

For the AST data structure itself, we've asked you to implement to fairly straightforward functions: a pretty-printer and a tree-walker. The first is mostly so you can get some visual feedback on what your parser is doing. (Note that the samples in the skeleton come with dumps of a correct AST from our reference implementation of the AST class. Yours should be more or less the same, ignoring formatting differences.) The second is a pre-order traversal, just like you've implemented before. We'll use this later.

Syntax-Directed Translation

The next step is to flesh out both the parser and the AST subclass definitions. This is a hefty task, but it's best done in tandem. The reason is that the code for production rules in your parser has to match up with your AST class definitions. For instance, the `p_program` rule we've given you in the parser corresponds directly to the `ASTProgram`. The constructor takes a single argument which corresponds to the list of AST nodes that are passed through the `p[1]` variable in `p_program` from the production rule `p_statement`.

It's okay if you're a little confused here. When built, your parser will automatically call production rule functions in your parser file based on what state it is currently in while trying to recognize input from a source file. You can think of it like the parser is calling a production rule function for every node in the concrete syntax tree and returning the result to its parent. Your job is to write those functions such that as they build up an AST as they go. (Think about how you would translate the concrete/abstract example from earlier using this approach.)

This method is called "syntax-directed translation", because you are literally using the syntax of the input file as a guide for when to call the functions that will translate your source into an AST piece by piece.

As always, if you get stuck, the ply documentation (http://www.dabeaz.com/ply/ply.html#ply_nn34) can help you out to some extent.

Final note: when you're done, you should be able to print your AST and check it against the reference trees in the sample directory.

Good luck!

Back to top ↑