Programming Languages, Introduction

CSCI 312, Winter 2023

January 11, 2023

“The Perils of JavaSchools,” *Joel Spolsky*

I have never met anyone who can do Scheme, Haskell, and C pointers who can’t pick up Java in two days, and create better Java code than people with five years of experience in Java, but try explaining that to the average HR drone.

New languages are everywhere:

Java, PHP, Perl, Python, TCL, Javascript, C#, Rust, Go, Zig, ...
Scripting languages for games, spreadsheets, editors ...
Data description languages for HTML, typesetting, spreadsheets, music, graphics ...

Modeling Languages

Learning a new programming languages means learning:

- a peculiar syntax for the language,
- behavior associated with each syntax,
- numerous useful libraries, and
- a collection of idioms that programmers of that language use.

Syntax doesn't tell us about a program's *behavior*

Which two of these are most similar?

1. `a [25]`
2. `(vector-ref a 25)`
3. `a [25]`
4. `a [25]`

Syntax doesn't tell us about a program's *behavior*

Which two of these are most similar?

1. `a [25]`
2. `(vector-ref a 25)`
3. `a [25]`
4. `a [25]`

The first and the second, obviously!

- The first is in Python; it's an array reference and will signal an error if the vector has less than 25 entries.
- The second is in Scheme; it's an array reference and will signal an error if the vector has less than 25 entries.
- The third is in C; God help you if the array isn't big enough.
- The fourth is in Haskell; it's a function applied to a list.

Syntax Matters

- Syntax is important:
- Bad syntax can hide errors (Fortran).
- Good syntax is easy to read and understand.
- Good syntax makes it easy to express your ideas.
- Try doing long division with Roman numerals sometime.

Syntax Matters, but not here

- Syntax is irrelevant to semantics.
- We're interested in semantics.
- Programs are trees, not strings, anyway.

```
((2**3)/sqrt(x))/(5 - 2x)+3
```

```
Add(
  Div(
    Div(
      Exp(Num(2), Num(3))
      Call(Name(sqrt), Args([Name(x)])
    )
    Sub(Num(5), Mul(Num(2), Name(x)))
  )
  Num(3)
)
```

Parsing in Python

- Python has a builtin module `ast` that provides access to Python parser tools.
- I have written some convenience tools in `ast_utilities.py`.

```
>>> from ast import parse, dump
>>> from ast_utilities import *
>>> parse('x=x+1')
<_ast.Module object at 0x0000022032093308>
>>> dump(parse('x=x+1'))
"Module(body=[Assign(targets=[Name(id='x', ctx=Store())],
    value=BinOp(left=Name(id='x', ctx=Load()), op=Add(),
    right=Num(n=1)))])"
```

Pretty-printed Python Parses

```
>>> ppp('x=x+1',pruning=False)
x=x+1
Module(
  body=[
    Assign(
      targets=[
        Name(
          id='x',
          ctx=Store(
            ))],
      value=BinOp(
        left=Name(
          id='x',
          ctx=Load(
            )),
        op=Add(
          ),
        right=Num(
          n=1))))])
```

With tree pruning:

```
>>> ppp('x=x+1')
x=x+1
Module(
  body=[
    Assign(
      targets=[
        Name(
          id='x')],
      value=BinOp(
        left=Name(
          id='x'),
        op=Add(
          ),
        right=Num(
          n=1))))])
```

```
class Module: ...
class Assign: ...
class BinOp: ...
```

prune: an example of processing an AST

Dispatching on type, recursion through subtrees:

```
def prune(tree):
    if type(tree) is Module:
        return Module(body=prune(tree.body))
    elif type(tree) is list:
        return [prune(x) for x in tree]
    elif type(tree) is Name:
        return Name(id=tree.id)
    elif type(tree) is BinOp:
        return BinOp(left=prune(tree.left),
                     op=tree.op,
                     right=prune(tree.right))
    ...
    else:
        return tree
```

Exploring Python Parse Trees

```
>>> dump(parse('x'))
"Module(body=[Expr(value=Name(id='x', ctx=Load()))])"
>>> ppp('x')
x
Module(
  body=[
    Expr(
      value=Name(
        id='x'))])
------------------------------------------------------------
>>> dump(parse('x').body[0])
"Expr(value=Name(id='x', ctx=Load()))"
>>> ppd(parse('x').body[0])
Expr(
  value=Name(
    id='x'))
------------------------------------------------------------
```

Homework # 1

Use Python parse trees to translate arithmetic expressions into prefix, postfix, and fully parenthsized infix forms.
Also build a calculator!

```
>>> s = '13+4**(5+6)/7'
>>> prefix(s)
'+ 13 / ** 4 + 5 6 7'
>>> postfix(s)
'13 4 5 6 + ** 7 / +'
>>> infix(s)
'(13+((4**(5+6))/7))'
>>> calc(s)
599199.2857142857
>>> eval(s)
599199.2857142857
```

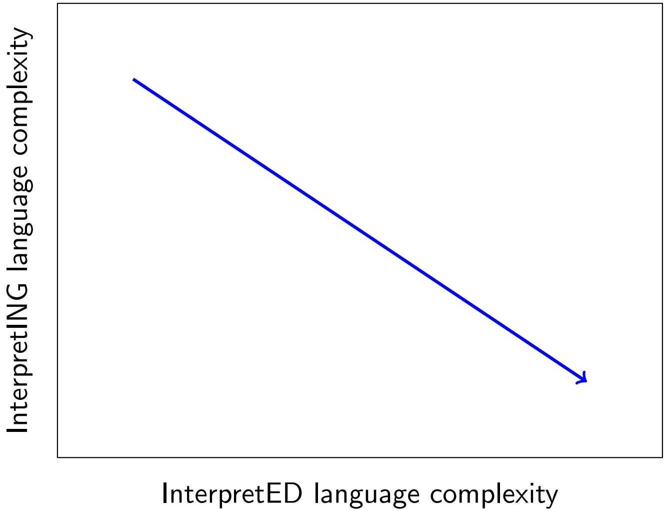
Finish the unit test as well

```python
import unittest
from ast_homework import *

class ast_Test(unittest.TestCase):
    def test_ast_calc(self):
        self.assertAlmostEqual(calc('9 + 3 * 4 / 2'),
                               9 + 3 * 4 / 2)
        self.assertAlmostEqual(calc('13+4**(5+6)/7'),
                               13+4**(5+6)/7)

    def test_ast_postfix(self):
        self.assertEqual(postfix('2+3+4'),
                         '2 3 + 4 +')


if __name__ == '__main__':
    unittest.main()
```

In this course: trees and their semantics

- How do we express semantics?
- English (natural language) is notoriously sloppy.
- Mathematics is difficult and abstract:
    - denotational semantics
    - operational semantics
    - axiomatic semantics
- We will use **interpreter semantics**:
  to explain a language, build an interpreter for it.

Interpreting language *vs.* interpreted language



Our InterpretING Language: Scheme

- `https://racket-lang.org/`
  - Our implementation
- `https://docs.racket-lang.org/index.html`
  - Work through the *Quick* tutorial
- Also do one or more of the following:
  - `https://ds26gte.github.io/tyscheme/`
    - Work through Chapters 1 to 6: Recursion
    - Note that we use Racket instead of mzscheme
  - `https://www.scheme.com/tspl4/`
    - Work through Chapters 1 and 2
    - Note that we use Racket instead of chez scheme
  - `https://htdp.org/`
    - Very careful exposition.