

# Notes on Formal Compiler Construction with the $\pi$ Framework

Christiano Braga

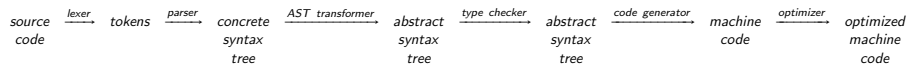
Instituto de Computação,  
Universidade Federal Fluminense, Niterói, Brazil

August 14, 2018

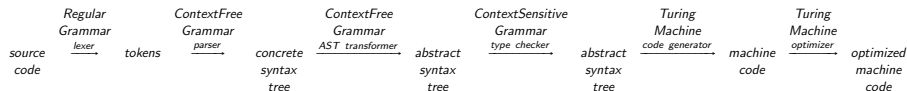
<http://github.com/ChristianoBraga/BPLC>



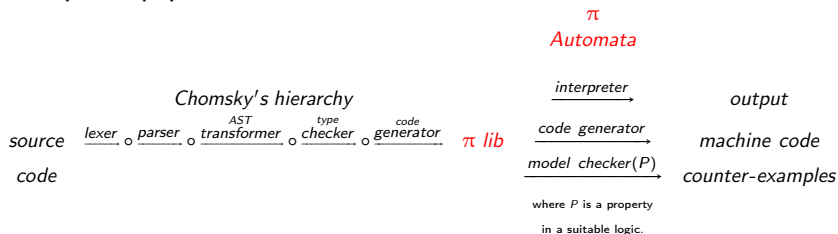
# Compiler pipeline



# Compiler pipeline and formal languages



# Compiler pipeline with the $\pi$ Framework



- $\pi$  lib defines a set of constructions common to many programming languages.
- $\pi$  lib constructions have a formal automata-based semantics in  $\pi$  automata.
- One may execute (or validate) a program in a given language by running its associated  $\pi$  lib program.
- $\pi$  Framework: <http://github.com/ChristianBraga/BPLC>
- Notes on Formal Compiler Construction with the  $\pi$  Framework: <https://github.com/ChristianBraga/BPLC/blob/master/notes/notes.pdf>.

# A calculator

We wish to compute simple arithmetic expressions such as  $5 * (3 + 2)$ .

# A calculator: Lexer

$\langle \textit{digit} \rangle ::= [0..9]$

$\langle \textit{digits} \rangle ::= \langle \textit{digit} \rangle^+$

$\langle \textit{boolean} \rangle ::= \text{'true'} \mid \text{'false'}$

# A calculator: concrete syntax

$$\langle exp \rangle ::= \langle aexp \rangle \mid \langle bexp \rangle$$
$$\langle aexp \rangle ::= \langle aexp \rangle '+' \langle term \rangle \mid \langle aexp \rangle '-' \langle term \rangle \mid \langle term \rangle$$
$$\langle term \rangle ::= \langle term \rangle '*' \langle factor \rangle \mid \langle term \rangle '/' \langle factor \rangle \mid \langle factor \rangle$$
$$\langle factor \rangle ::= '(' \langle aexp \rangle ')' \mid \langle digits \rangle$$
$$\langle bexp \rangle ::= \langle boolean \rangle \mid \sim \langle bexp \rangle \mid \langle bexp \rangle \langle boolop \rangle \langle bexp \rangle \\ \mid \langle aexp \rangle \langle iop \rangle \langle aexp \rangle$$
$$\langle iop \rangle ::= '=' \mid '<' \mid '>' \mid '<=' \mid '>='$$

# A calculator: abstract syntax

$$\langle exp \rangle ::= \langle digits \rangle \mid \langle boolean \rangle \mid \langle exp \rangle \langle bop \rangle \langle exp \rangle$$
$$\langle bop \rangle ::= '+' \mid '-' \mid '*' \mid '|' \mid '/' \mid '=' \mid '<' \mid '>' \mid '<=' \mid '>='$$



# A calculator: $\pi$ denotations I

Let  $D$  in  $\langle \text{digits} \rangle$ ,  $B$  in  $\langle \text{boolean} \rangle$  and  $E_1, E_2$  in  $\langle \text{exp} \rangle$ ,

$$\llbracket D \rrbracket_{\pi} = \text{num}(D) \quad (1)$$

$$\llbracket B \rrbracket_{\pi} = \text{boo}(B) \quad (2)$$

$$\llbracket E_1 + E_2 \rrbracket_{\pi} = \text{add}(\llbracket E_1 \rrbracket_{\pi}, \llbracket E_2 \rrbracket_{\pi}) \quad (3)$$

$$\llbracket E_1 - E_2 \rrbracket_{\pi} = \text{sub}(\llbracket E_1 \rrbracket_{\pi}, \llbracket E_2 \rrbracket_{\pi}) \quad (4)$$

$$\llbracket E_1 * E_2 \rrbracket_{\pi} = \text{mul}(\llbracket E_1 \rrbracket_{\pi}, \llbracket E_2 \rrbracket_{\pi}) \quad (5)$$

$$\llbracket E_1 / E_2 \rrbracket_{\pi} = \text{div}(\llbracket E_1 \rrbracket_{\pi}, \llbracket E_2 \rrbracket_{\pi}) \quad (6)$$

$$\llbracket E_1 < E_2 \rrbracket_{\pi} = \text{lt}(\llbracket E_1 \rrbracket_{\pi}, \llbracket E_2 \rrbracket_{\pi}) \quad (7)$$

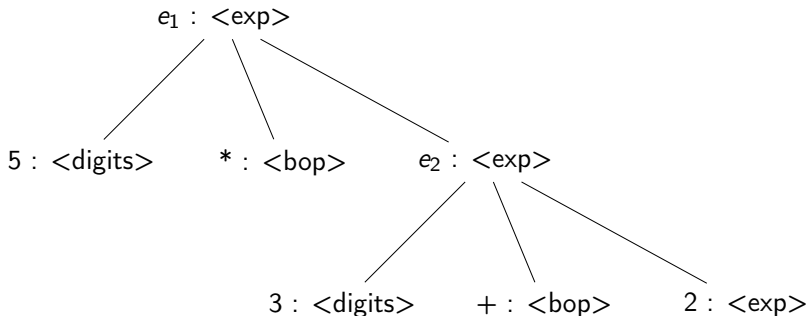
$$\llbracket E_1 \leq E_2 \rrbracket_{\pi} = \text{le}(\llbracket E_1 \rrbracket_{\pi}, \llbracket E_2 \rrbracket_{\pi}) \quad (8)$$

$$\llbracket E_1 > E_2 \rrbracket_{\pi} = \text{gt}(\llbracket E_1 \rrbracket_{\pi}, \llbracket E_2 \rrbracket_{\pi}) \quad (9)$$

$$\llbracket E_1 \geq E_2 \rrbracket_{\pi} = \text{ge}(\llbracket E_1 \rrbracket_{\pi}, \llbracket E_2 \rrbracket_{\pi}) \quad (10)$$

## A calculator: $\pi$ denotations II

- $\pi$  denotations are *functions*  $\llbracket \cdot \rrbracket_{\pi} : AST \rightarrow \pi \text{ lib}$ , where *AST* denotes the *datatype* for the abstract syntax tree and  $\pi \text{ lib}$  denotes the datatype for  $\pi \text{ lib}$  programs.
- Note that  $\llbracket \cdot \rrbracket_{\pi}$  has *trees* as parameters, instances of *AST*. The example expression  $5 * (3 + 2)$  becomes



# A calculator: $\pi$ denotations III

$$\llbracket 5 * (3 + 2) \rrbracket_{\pi} = \text{mul}(\llbracket 5 \rrbracket_{\pi}, \llbracket (3 + 2) \rrbracket_{\pi}) \quad \text{by Equation 5}$$

$$\text{mul}(\llbracket 5 \rrbracket_{\pi}, \llbracket (3 + 2) \rrbracket_{\pi}) = \text{mul}(\text{num}(5), \llbracket (3 + 2) \rrbracket_{\pi}) \quad \text{by Equations 1}$$

$$\text{mul}(\text{num}(5), \llbracket (3 + 2) \rrbracket_{\pi}) = \text{mul}(\text{num}(5), \text{add}(\llbracket 3 \rrbracket_{\pi}, \llbracket 2 \rrbracket_{\pi})) \quad \text{by Equation 3}$$

$$\text{mul}(\text{num}(5), \text{add}(\llbracket 3 \rrbracket_{\pi}, \llbracket 2 \rrbracket_{\pi})) = \text{mul}(\text{num}(5), \text{add}(\text{num}(3), \llbracket 2 \rrbracket_{\pi})) \quad \text{by Equation 1}$$

$$\text{mul}(\text{num}(5), \text{add}(\text{num}(3), \llbracket 2 \rrbracket_{\pi})) = \text{mul}(\text{num}(5), \text{add}(\text{num}(3), \text{num}(2))) \quad \text{by Equation 1}$$

# A calculator: executing $\pi$ lib with $\pi$ automata

A  $\pi$  automaton is a 5-tuple  $\mathcal{A} = (G, Q, \delta, q_0, F)$ , where  $G$  is a context-free grammar,  $Q$  is the set of states,  $q_0$  is the initial state,  $F \subseteq Q$  is the set of final states and

$$\delta : L(G)^* \times L(G)^* \times Store \rightarrow Q,$$

where  $L(G)$  is the language generated by  $G$  and  $Store$  represents the memory. (Elements in a set  $S^*$  are represented by terms  $[s_1, s_2, \dots, s_n]$ .)

$$\begin{aligned} \delta([mul(num(5), add(num(3), num(2))), \emptyset, \emptyset) &= \delta([num(5), add(num(3), num(2)), \#MUL], \emptyset, \emptyset) \\ \delta([num(5), add(num(3), num(2)), \#MUL], \emptyset, \emptyset) &= \delta([add(num(3), num(2)), \#MUL], [num(5)], \emptyset) \\ \delta([add(num(3), num(2)), \#MUL], [num(5)], \emptyset) &= \delta([num(3), num(2), \#SUM, \#MUL], [num(5)], \emptyset) \\ \delta([num(3), num(2), \#SUM, \#MUL], [num(5)], \emptyset) &= \delta([num(2), \#SUM, \#MUL], [num(3), num(5)], \emptyset) \\ \delta([num(2), \#SUM, \#MUL], [num(3), num(5)], \emptyset) &= \delta([\#SUM, \#MUL], [num(2), num(3), num(5)], \emptyset) \\ \delta([\#SUM, \#MUL], [num(2), num(3), num(5)], \emptyset) &= \delta([\#MUL], [num(5), num(5)], \emptyset) \\ \delta([\#MUL], [num(5), num(5)], \emptyset) &= \delta(\emptyset, [num(25)], \emptyset) \\ \delta(\emptyset, [num(25)], \emptyset) &= num(25) \end{aligned}$$

# $\pi$ lib expressions in Python I

<https://github.com/ChristianoBraga/BPLC/blob/master/python/pi.ipynb>

```
1 class Statement:
2     def __init__(self, *args):
3         self.opr =args
4     def __str__(self):
5         ret =str(self.__class__.__name__)+ "("
6         for o in self.opr:
7             ret +=str(o)
8         ret +=")"
9         return ret
10 class Exp(Statement): pass
11 class ArithExp(Exp): pass
```

## $\pi$ lib expressions in Python II

```
1 class Num(ArithExp):
2     def __init__(self, f):
3         assert(isinstance(f, int))
4         ArithExp.__init__(self, f)
5 class Sum(ArithExp):
6     def __init__(self, e1, e2):
7         assert(isinstance(e1, Exp) and isinstance(e2, Exp))
8         ArithExp.__init__(self, e1, e2)
9 ...
```

## $\pi$ lib expressions in Python III

```
1 class BoolExp(Exp): pass
2 class Eq(BoolExp):
3     def __init__(self, e1, e2):
4         assert(isinstance(e1, Exp) and isinstance(e2, Exp))
5         BoolExp.__init__(self, e1, e2)
6 ...
```

# $\pi$ lib expressions in Python IV

```
1 exp = Sum(Num(1), Mul(Num(2), Num(4)))  
2 print(exp)  
3  
4 Sum(Num(1)Mul(Num(2)Num(4)))
```



# $\pi$ lib expressions in Python V

```

1 exp2 =Mul(2, 1)
2 -----
3
4
5
6
7
8
9
10
11
12
13
14

```

AssertionError Traceback (most recent call last)

<ipython-input-7-00fd40a79a54> in <module>()

---->1 exp2 =Mul(2, 1)

<ipython-input-5-42a82e58862f> in \_\_init\_\_(self, e1, e2)

```

28 class Mul(ArithExp):
29     def __init__(self, e1, e2):
30     assert(isinstance(e1, Exp) and isinstance(e2, Exp))
31     ArithExp.__init__(self, e1, e2)
32     class BoolExp(Exp): pass

```

AssertionError:

# $\pi$ automaton for $\pi$ lib expressions I

```
1  ## Expressions
2  class ValueStack(list): pass
3  class ControlStack(list): pass
4  class ExpKW:
5      SUM = "#SUM"
6      SUB = "#SUB"
7      MUL = "#MUL"
8      EQ = "#EQ"
9      NOT = "#NOT"
```

## $\pi$ automaton for $\pi$ lib expressions II

```
1 class ExpPiAut(dict):
2     def __init__(self):
3         self["val"] = ValueStack()
4         self["cnt"] = ControlStack()
5     def __evalSum(self, e):
6         e1 = e.opr[0]
7         e2 = e.opr[1]
8         self.pushCnt(ExpKW.SUM)
9         self.pushCnt(e1)
10        self.pushCnt(e2)
11    def pushCnt(self, e):
12        cnt = self.cnt()
13        cnt.append(e)
14    ...
```

# $\pi$ automaton for $\pi$ lib expressions III

```
1 ea =ExpPiAut()  
2 print(exp)  
3 ea.pushCnt(exp)  
4 while not ea.emptyCnt():  
5     ea.eval()  
6     print(ea)
```

$\pi$  automaton for  $\pi$  lib expressions IV

```

1 Sum(Num(1)Mul(Num(2)Num(4)))
2 {'val': [], 'cnt': ['#SUM', <__main__.Num object at 0x111851470>, <
    __main__.Mul object at 0x1118516d8>]}
3 {'val': [], 'cnt': ['#SUM', <__main__.Num object at 0x111851470>, '#MUL',
    , <__main__.Num object at
    0x111851630>, <__main__.Num object
    at 0x1118516a0>]}
4 {'val': [4], 'cnt': ['#SUM', <__main__.Num object at 0x111851470>, '#MUL',
    , <__main__.Num object at
    0x111851630>]}
5 {'val': [4, 2], 'cnt': ['#SUM', <__main__.Num object at 0x111851470>, '#MUL']}
6 {'val': [8], 'cnt': ['#SUM', <__main__.Num object at 0x111851470>]}
7 {'val': [8, 1], 'cnt': ['#SUM']}
8 {'val': [9], 'cnt': []}

```