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

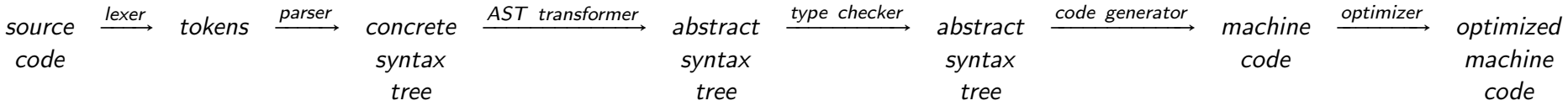
Christiano Braga

Instituto de Computação,
Universidade Federal Fluminense, Niterói, Brazil

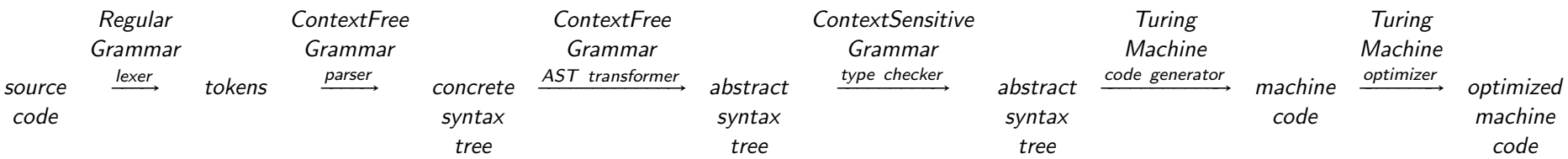
August 28, 2018

http://github.com/ChristianoBraga/BPLC

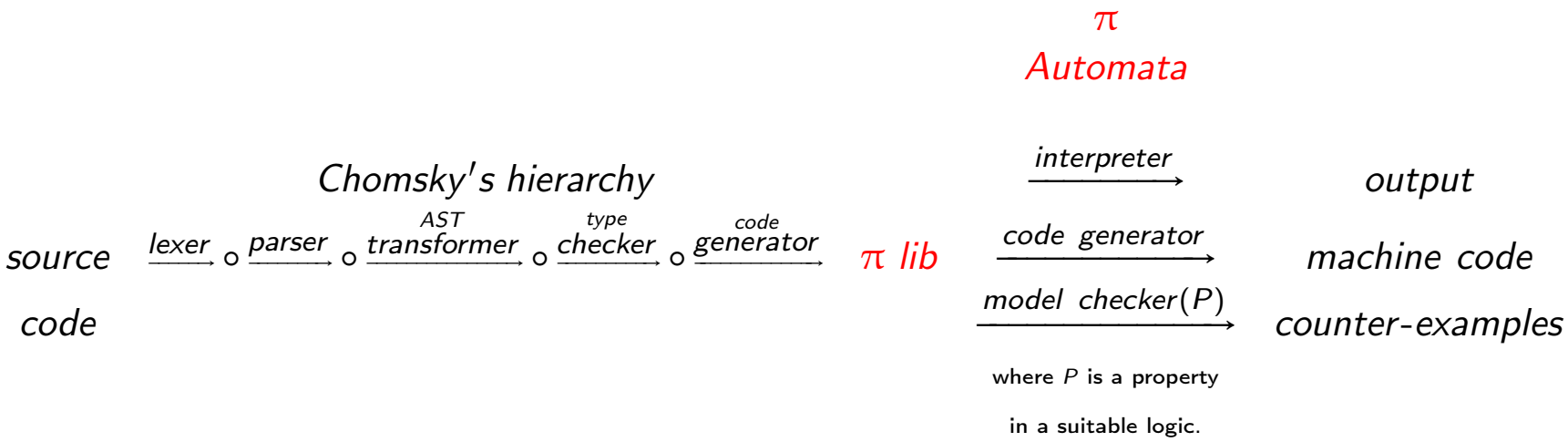
# Compiler pipeline



# Compiler pipeline and formal languages



# Compiler pipeline with the π Framework




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

Instituto de Computação

# A calculator

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

# A calculator: Lexer

$\langle digit \rangle \quad ::= [0..9]$

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

$\langle boolean \rangle \quad ::= \text{`}\texttt{true}\text{'} \mid \text{`}\texttt{false}\text{'}$

Instituto de Computação

# A calculator: concrete syntax

$\langle exp \rangle$ ::= $\langle aexp \rangle \mid \langle bexp \rangle$

$\langle aexp \rangle$ ::= $\langle aexp \rangle$ '+' $\langle term \rangle \mid \langle aexp \rangle$ '-' $\langle term \rangle \mid \langle term \rangle$

$\langle term \rangle$ ::= $\langle term \rangle$ '*' $\langle factor \rangle \mid \langle term \rangle$ '/' $\langle factor \rangle \mid \langle factor \rangle$

$\langle factor \rangle$ ::= '(' $\langle aexp \rangle$ ')' $\mid \langle digits \rangle$

$\langle bexp \rangle$ ::= $\langle boolean \rangle \mid$ '~' $\langle bexp \rangle \mid \langle bexp \rangle \langle boolop \rangle \langle bexp \rangle$
$\mid \langle aexp \rangle \langle iop \rangle \langle aexp \rangle$

$\langle boolop \rangle$ ::= '=' | '/\' | '\/'

$\langle iop \rangle$ ::= '<' | '>' | '<=' | '>='

Instituto de Computação UFF

# A calculator: abstract syntax

$\langle exp \rangle \quad ::= \langle digits \rangle \mid \langle boolean \rangle \mid \langle exp \rangle \ \langle bop \rangle \ \langle exp \rangle$

$\langle bop \rangle \quad ::= \text{`+'} \mid \text{`-'} \mid \text{`*'} \mid \text{`/'} \mid \text{`='} \mid \text{`\/'} \mid \text{`/\'} \mid \text{`<'} \mid \text{`>'} \mid \text{`<='} \mid \text{`>='}$

Instituto de computação

# A calculator: $\pi$ denotations I

Let $D$ in <digits>, $B$ in <boolean> and $E_1, E_2$ in <exp>,

$$[\![D]\!]_\pi = Num(D) \tag{1}$$
$$[\![B]\!]_\pi = Boo(B) \tag{2}$$
$$[\![E_1 + E_2]\!]_\pi = Sum([\![E_1]\!]_\pi, [\![E_2]\!]_\pi) \tag{3}$$
$$[\![E_1 - E_2]\!]_\pi = Sub([\![E_1]\!]_\pi, [\![E_2]\!]_\pi) \tag{4}$$
$$[\![E_1 * E_2]\!]_\pi = Mul([\![E_1]\!]_\pi, [\![E_2]\!]_\pi) \tag{5}$$
$$[\![E_1 / E_2]\!]_\pi = Div([\![E_1]\!]_\pi, [\![E_2]\!]_\pi) \tag{6}$$
$$[\![E_1 < E_2]\!]_\pi = Lt([\![E_1]\!]_\pi, [\![E_2]\!]_\pi) \tag{7}$$
$$[\![E_1 <= E_2]\!]_\pi = Le([\![E_1]\!]_\pi, [\![E_2]\!]_\pi) \tag{8}$$
$$[\![E_1 > E_2]\!]_\pi = Gt([\![E_1]\!]_\pi, [\![E_2]\!]_\pi) \tag{9}$$
$$[\![E_1 >= E_2]\!]_\pi = Ge([\![E_1]\!]_\pi, [\![E_2]\!]_\pi) \tag{10}$$

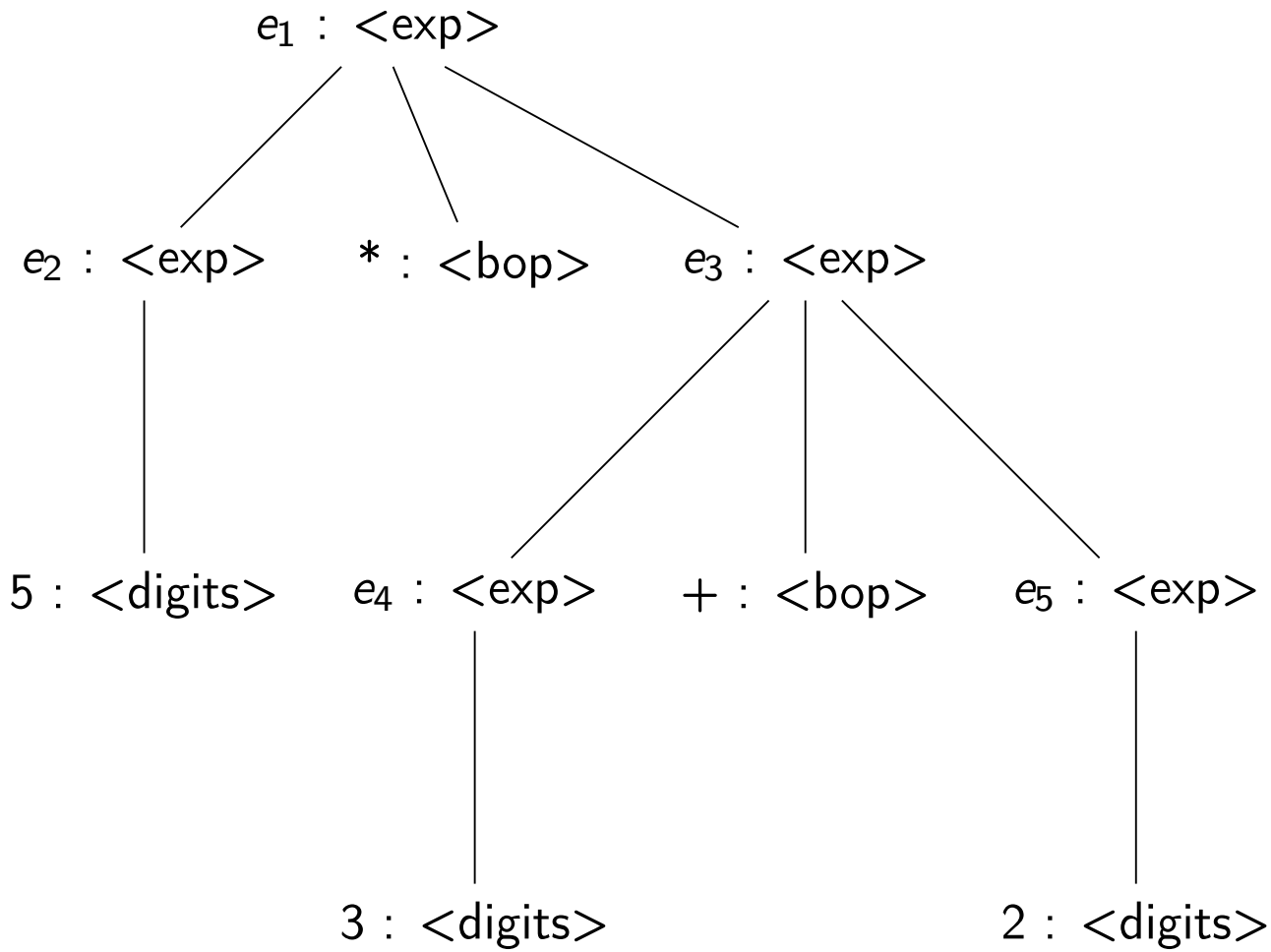
Instituto de Computação

# A calculator: $\pi$ denotations II

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

Instituto de computação

# A calculator: $\pi$ denotations III



# A calculator: $\pi$ denotations IV

$$[\![5 * (3+2)]\!]_\pi = Mul([\![5]\!]_\pi, [\![(3+2)]\!]_\pi) \quad \text{by Equation 5}$$

$$Mul([\![5]\!]_\pi, [\![(3+2)]\!]_\pi) = Mul(Num(5), [\![(3+2)]\!]_\pi) \quad \text{by Equations 1}$$

$$Mul(num(5), [\![(3+2)]\!]_\pi) = Mul(Num(5), Sum([\![3]\!]_\pi, [\![2]\!]_\pi) \quad \text{by Equation 3}$$

$$Mul(Num(5), Sum([\![3]\!]_\pi, [\![2]\!]_\pi) = Mul(Num(5), Sum(Num(3), [\![2]\!]_\pi) \quad \text{by Equation 1}$$

$$Mul(Num(5), Sum(Num(3), [\![2]\!]_\pi) = Mul(Num(5), Sum(Num(3), Num(2)) \quad \text{by Equation 1}$$

Instituto de Computação

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

A $\pi$ automaton is a 5-tuple $\mathscr{A} = (G, Q, \delta, q_0, F)$, 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 $[s_1, s_2, \ldots, s_n]$.)

$$\delta([Mul(Num(5), Sum(Num(3), Num(2)]], \emptyset, \emptyset) = \delta([Num(5), Sum(Num(3), Num(2)), \#MUL], \emptyset, \emptyset)$$
$$\delta([Num(5), Sum(Num(3), Num(2)), \#MUL], \emptyset, \emptyset) = \delta([Sum(Num(3), Num(2)), \#MUL], [Num(5)], \emptyset)$$
$$\delta([Sum(Num(3), Num(2)), \#MUL], [Num(5)], \emptyset) = \delta([Num(3), Num(2), \#SUM, \#MUL], [Num(5)], \emptyset)$$
$$\delta([Num(3), Num(2), \#SUM, \#MUL], [Num(5)], \emptyset) = \delta([Num(2), \#SUM, \#MUL], [Num(3), Num(5)], \emptyset)$$
$$\delta([Num(2), \#SUM, \#MUL], [Num(3), Num(5)], \emptyset) = \delta([\#SUM, \#MUL], [Num(2), Num(3), Num(5)], \emptyset)$$
$$\delta([\#SUM, \#MUL], [Num(2), Num(3), Num(5)], \emptyset) = \delta([\#MUL], [Num(5), Num(5)], \emptyset)$$
$$\delta([\#MUL], [Num(5), Num(5)], \emptyset) = \delta(\emptyset, [Num(25)], \emptyset)$$
$$\delta(\emptyset, [Num(25)], \emptyset) = Num(25)$$

Instituto de Computação

# Excerpt of $\pi$ lib expressions

⟨*Statement*⟩ ::= <Exp>

⟨*Exp*⟩ ::= <ArithExp> | <BoolExp>

⟨*ArithExp*⟩ ::= 'Num'(<digits>) | 'Sum'(<Exp> , <Exp> ) |
'Sub'(<Exp>, <Exp>) | 'Mul'(<Exp>, <Exp>)

⟨*BoolExp*⟩ ::= 'Eq'(<Exp>, <Exp>) | 'Not'(<Exp>)

Instituto de computação

# $\pi$ automata semantics for $\pi$ lib expressions

- Recall that $\delta : L(G)^* \times L(G)^* \times Store \rightarrow Q$, and let $N, N_i \in \mathbb{N}$, $C, V \in L(G)^*$, $S \in Store$,

$$\delta(Num(N) :: C, V, S) = \delta(C, Num(N) :: V, S) \quad (11)$$

$$\delta(Sum(E_1, E_2) :: C, V, S) = \delta(E_1 :: E_2 :: \#SUM :: C, V, S) \quad (12)$$

$$\delta(\#SUM :: C, Num(N_1) :: Num(N_2) :: V, S) = \delta(C, Num(N_1 + N_2) :: V, S) \quad (13)$$

$$\ldots$$

$$\delta(Not(E) :: C, V, S) = \delta(E :: \#NOT :: C, V, S) \quad (14)$$

$$\delta(\#NOT :: C, Boo(true) :: V, S) = \delta(C, Boo(false) :: V, S) \quad (15)$$

$$\delta(\#NOT :: C, Boo(false) :: V, S) = \delta(C, Boo(true) :: V, S) \quad (16)$$

- Notation $h :: ls$ denotes the concatenation of element $h$ with the list $ls$.
- $C$ represents the *control* stack. $V$ represents the *value* stack. $S$ denotes the memory store.
- $\delta(\emptyset, V, S)$ denotes an *accepting state*.

Instituto de Computação UFF

# $\pi$ lib commands

- Commands are language constructions that require both an *environment* and a *memory* store to be evaluated.

  ⟨*Statement*⟩ ::= <Cmd>

  ⟨*Exp*⟩ ::= 'Id'(<String>)

  ⟨*Cmd*⟩ ::= 'Assign'(<Id>, <Exp>)
  | 'Loop'(<BoolExp>, <Cmd>)
  | 'CSeq'(<Cmd>, <Cmd>)

- From a syntactic standpoint, they extend both statements and expressions, as an identifier is an expression.

Instituto de computação

# $\pi$ automata semantics for $\pi$ lib commands I

- A location $l \in Loc$ denotes a memory cell.
- Storable and Bindable sets denote the data that may be mapped to by identifiers and locations on the memory and environment respectively.
- $Store = Id \mapsto Storable$, $Env = Loc \mapsto Bindable$, $Loc \subseteq Store$, $\mathbb{N} \subseteq Loc, Bindable$.
- Now the transition function is $\delta : L(G)^* \times L(G)^* \times Env \times Store \rightarrow Q$, and let $W \in String$, $C, V \in L(G)^*$, $S \in Store$, $E \in Env$, $B \in Bindable$, $l \in Loc$, $T \in Storable$, $X \in < Exp >$, $M, M_1, M_2 \in < Cmd >$,and

Instituto de computação

# $\pi$ automata semantics for $\pi$ lib commands II

expression $S' = S/[I \mapsto N]$ means that $S'$ equals to $S$ in all indices but $I$ that is bound to $N$,

$$\delta(Id(W) :: C, V, E, S) = \delta(C, B :: V, E, S), \quad (17)$$
$$\text{where } E[W] = I \text{ and } S[I] = B,$$
$$\delta(Assign(W, X) :: C, V, E, S) = \delta(X :: \#ASSIGN :: C, W :: V, E, S'), \quad (18)$$
$$\delta(\#ASSIGN :: C, T :: W :: V, E, S) = \delta(C, V, E, S'), \quad (19)$$
$$\text{where } E[W] = I \text{ and } S' = S/[I \mapsto T],$$
$$\delta(Loop(X, M) :: C, V, E, S) = \delta(X :: \#LOOP :: C, Loop(X, M) :: V, E, S), \quad (20)$$
$$\delta(\#LOOP :: C, Boo(true) :: Loop(X, M) :: V, E, S) = \delta(M :: Loop(X, M) :: C, V, E, S), \quad (21)$$
$$\delta(\#LOOP :: C, Boo(false) :: Loop(X, M) :: V, E, S) = \delta(C, V, E, S), \quad (22)$$
$$\delta(CSeq(M_1, M_2) :: C, V, E, S) = \delta(M_1 :: M_2 :: C, V, E, S). \quad (23)$$

Instituto de Computação

# $\pi$ 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
```

Instituto de Computação

# $\pi$ 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 ...
```

Instituto de Computação

# $\pi$ 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 ...
```

Instituto de Computação

# $\pi$ 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

```
exp2 =Mul(2, 1)
---------------------------------------------------------------------------
                                          ---
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"
```

Instituto de Computação

# $\pi$ 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 ...
```

Instituto de Computação

# $\pi$ 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 0x11851470>, <
                                __main__.Mul object at 0x1118516d8>]
                                }
3 {'val': [], 'cnt': ['#SUM', <__main__.Num object at 0x11851470>, '#MUL'
                                , <__main__.Num object at
                                0x11851630>, <__main__.Num object
                                at 0x1118516a0>]}
4 {'val': [4], 'cnt': ['#SUM', <__main__.Num object at 0x11851470>, '#MUL
                                ', <__main__.Num object at
                                0x11851630>]}
5 {'val': [4, 2], 'cnt': ['#SUM', <__main__.Num object at 0x11851470>, '#
                                MUL']}
6 {'val': [8], 'cnt': ['#SUM', <__main__.Num object at 0x11851470>]}
7 {'val': [8, 1], 'cnt': ['#SUM']}
8 {'val': [9], 'cnt': []}
```

Instituto de Computação

# $\pi$ lib commands I

```
1  class Cmd(Statement): pass
2  class Id(Exp):
3      def __init__(self, s):
4          assert(isinstance(s, str))
5          Exp.__init__(self, s)
6  class Assign(Cmd):
7      def __init__(self, i, e):
8          assert(isinstance(i, Id) and isinstance(e, Exp))
9          Cmd.__init__(self, i, e)
10 class Loop(Cmd):
11     def __init__(self, be, c):
12         assert(isinstance(be, BoolExp) and isinstance(c, Cmd))
13         Cmd.__init__(self, be, c)
14 class CSeq(Cmd):
15     def __init__(self, c1, c2):
16         assert(isinstance(c1, Cmd) and isinstance(c2, Cmd))
17         Cmd.__init__(self, c1, c2)
```

Instituto de computação

# $\pi$ lib commands II

```
1 cmd =Assign(Id("x"), Num(1))
2 print(type(cmd))
3 print(cmd)
4 <class '__main__.Assign'>
5 Assign(Id(x)Num(1))
```

Instituto de Computação

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

Environment, Location, Store and commands opcodes.

```
1 ## Commands
2 class Env(dict): pass
3 class Loc(int): pass
4 class Sto(dict): pass
5 class CmdKW:
6     ASSIGN ="#ASSIGN"
7     LOOP ="#LOOP"
```

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

$\pi$ automaton for commands extends the $\pi$ automaton for expressions.

```
1  class CmdPiAut(ExpPiAut):
2      def __init__(self):
3          self["env"] =Env()
4          self["sto"] =Sto()
5          ExpPiAut.__init__(self)
6      def env(self):
7          return self["env"]
8      def getLoc(self, i):
9          en =self.env()
10         return en[i]
11     def sto(self):
12         return self["sto"]
13     def updateStore(self, l, v):
14         st =self.sto()
15         st[l] =v
```

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

## $\pi$ semantics for assignment.

$$\delta(Assign(W,X)::C,V,E,S) = \delta(X::\#ASSIGN::C,W::V,E,S'),$$
$$\delta(\#ASSIGN::C,T::W::V,E,S) = \delta(C,V,E,S'),$$
$$\text{where } E[W] = I \text{ and } S' = S/[I \mapsto T].$$

```
1     def __evalAssign(self, c):
2         i =c.opr[0]
3         e =c.opr[1]
4         self.pushVal(i.opr[0])
5         self.pushCnt(CmdKW.ASSIGN)
6         self.pushCnt(e)
7     def __evalAssignKW(self):
8         v =self.popVal()
9         i =self.popVal()
10        l =self.getLoc(i)
11        self.updateStore(l, v)
```

# $\pi$ automaton for $\pi$ lib commands IV

$\pi$ semantics for identifiers.

$$\delta(Id(W) :: C, V, E, S) = \delta(C, B :: V, E, S),$$
$$\text{where } E[W] = I \text{ and } S[I] = B.$$

```
1     def __evalId(self, i):
2         s =self.sto()
3         l =self.getLoc(i)
4         self.pushVal(s[l])
```

# $\pi$ automaton for $\pi$ lib commands V

$\pi$ semantics for loop: recursive step.

$$\delta(Loop(X,M)::C,V,E,S) = \delta(X::\#LOOP::C,Loop(X,M)::V,E,S).$$

```
1    def __evalLoop(self, c):
2        be =c.opr[0]
3        bl =c.opr[1]
4        self.pushVal(Loop(be, bl))
5        self.pushVal(bl)
6        self.pushCnt(CmdKW.LOOP)
7        self.pushCnt(be)
```

Instituto de Computação

# $\pi$ automaton for $\pi$ lib commands VI

## $\pi$ semantics for loop: basic steps.

$$\delta(\#LOOP :: C, Boo(true) :: Loop(X, M) :: V, E, S) = \delta(M :: Loop(X, M) :: C, V, E, S),$$
$$\delta(\#LOOP :: C, Boo(false) :: Loop(X, M) :: V, E, S) = \delta(C, V, E, S).$$

```
1     def __evalLoopKW(self):
2         t =self.popVal()
3         if t:
4             c =self.popVal()
5             lo =self.popVal()
6             self.pushCnt(lo)
7             self.pushCnt(c)
8         else:
9             self.popVal()
10            self.popVal()
```

# $\pi$ automaton for $\pi$ lib commands VII

$\pi$ semantics for command composition.

$$\delta(CSeq(M_1, M_2) :: C, V, E, S) = \delta(M_1 :: M_2 :: C, V, E, S).$$

```
1     def __evalCSeq(self, c):
2         c1 =c.opr[0]
3         c2 =c.opr[1]
4         self.pushCnt(c2)
5         self.pushCnt(c1)
```

# $\pi$ automaton for $\pi$ lib commands VIII

Commands are now on the top of the food chain.

```
1     def eval(self):
2         c =self.popCnt()
3         if isinstance(c, Assign):
4             self.__evalAssign(c)
5         elif c ==CmdKW.ASSIGN:
6             self.__evalAssignKW()
7         elif isinstance(c, Id):
8             self.__evalId(c.opr[0])
9         elif isinstance(c, Loop):
10            self.__evalLoop(c)
11        elif c ==CmdKW.LOOP:
12            self.__evalLoopKW()
13        elif isinstance(c, CSeq):
14            self.__evalCSeq(c)
15        else:
16            self.pushCnt(c)
17            ExpPiAut.eval(self)
```

# IMP grammar for expressions and commands

$$
\begin{array}{lll}
\langle Prog \rangle & ::= & \langle Cmd \rangle * \\
\langle Cmd \rangle & ::= & \text{`}\texttt{while}\text{'}\ \text{`(`}\langle BoolExp \rangle \text{`)'}\ \text{`}\texttt{do}\text{'}\ \text{`\{'}\ \langle Cmd \rangle\ \text{`\}'} \\
 & | & \langle Id \rangle\ \text{`:='}\ \langle Exp \rangle \mid \langle Cmd \rangle\ \text{`;'}\ \langle Cmd \rangle \\
 & | & \text{`}\texttt{print}\text{'}\ \text{`('}\ \langle Exp \rangle\ \text{`)'} \\
\langle Exp \rangle & ::= & \langle Id \rangle \mid \langle AExp \rangle \mid \langle BExp \rangle \\
\langle AExp \rangle & ::= & \langle AExp \rangle\ \langle AOp \rangle\ \langle AExp \rangle \\
\langle AOp \rangle & ::= & \text{`+'} \mid \text{`-'} \mid \text{`*'} \mid \text{`/'} \\
\langle BExp \rangle & ::= & \text{`}\texttt{true}\text{'} \mid \text{`}\texttt{false}\text{'} \mid \text{`\~{}'}\ \langle BExp \rangle \mid \langle BExp \rangle\ \langle BOp \rangle\ \langle BExp \rangle \\
 & | & \langle AExp \rangle\ \langle IOp \rangle\ \langle AExp \rangle \\
\langle BOp \rangle & ::= & \text{`/\textbackslash'} \mid \text{`\textbackslash/'} \mid \text{`==`} \\
\langle IOp \rangle & ::= & \text{`<'} \mid \text{`<='} \mid \text{`>'} \mid \text{`>='}
\end{array}
$$

Instituto de Computação

# Example: iterative factorial

```
1 while (~ (x == 0))
2 do {
3       y := y * x ;
4       x := x - 1
5 } ;
6 print(y)
```