

Notes on Formal Compiler Construction with the π Framework

Christiano Braga

Instituto de Computação,
Universidade Federal Fluminense, Niterói, Brazil

August 30, 2018

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



1. Introduction

Example

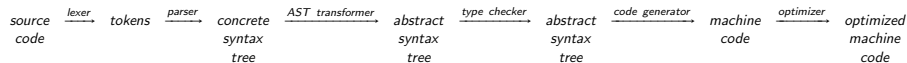
2. π expressions

3. π commands

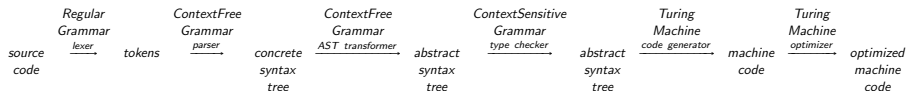
4. π^2 : π Framework in Python

5. IMP language

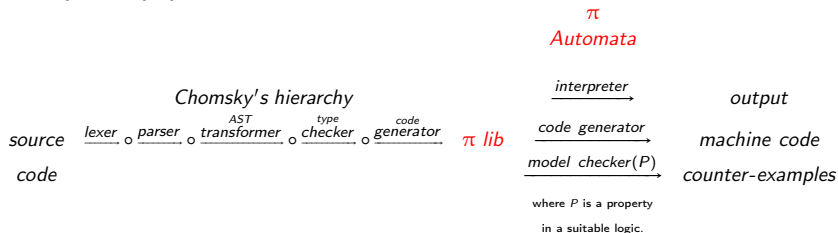
Compiler pipeline



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>.

A calculator

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

A calculator: Lexer

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

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

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

A calculator: concrete syntax

$\langle \text{exp} \rangle ::= \langle \text{aexp} \rangle \mid \langle \text{bexp} \rangle$

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

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

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

$\langle \text{bexp} \rangle ::= \langle \text{boolean} \rangle \mid '\sim' \langle \text{bexp} \rangle \mid \langle \text{bexp} \rangle \langle \text{boolop} \rangle \langle \text{bexp} \rangle$
 $\mid \langle \text{aexp} \rangle \langle \text{iop} \rangle \langle \text{aexp} \rangle$

$\langle \text{boolop} \rangle ::= '=' \mid '/\backslash' \mid '\backslash/'$

$\langle \text{iop} \rangle ::= '<' \mid '>' \mid '<=' \mid '>='$

A calculator: abstract syntax

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

A calculator: π denotations I

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

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

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

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

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

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

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

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

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

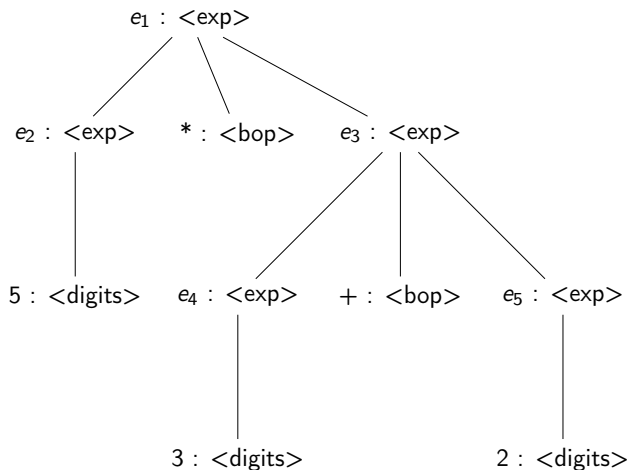
$$\llbracket E_1 > E_2 \rrbracket_{\pi} = \text{Gt}(\llbracket E_1 \rrbracket_{\pi}, \llbracket E_2 \rrbracket_{\pi}) \quad (9)$$

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

A calculator: π denotations II

- π 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: π denotations III



A calculator: π denotations IV

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

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

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

$$\text{Mul}(\text{Num}(5), \text{Sum}(\llbracket 3 \rrbracket_{\pi}, \llbracket 2 \rrbracket_{\pi})) = \text{Mul}(\text{Num}(5), \text{Sum}(\text{Num}(3), \llbracket 2 \rrbracket_{\pi})) \quad \text{by Equation 1}$$

$$\text{Mul}(\text{Num}(5), \text{Sum}(\text{Num}(3), \llbracket 2 \rrbracket_{\pi})) = \text{Mul}(\text{Num}(5), \text{Sum}(\text{Num}(3), \text{Num}(2))) \quad \text{by Equation 1}$$

A calculator: executing π lib with π automata

A π 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 $[s_1, s_2, \dots, s_n].$)

$$\begin{aligned} \delta([Mul(Num(5), Sum(Num(3), Num(2))), \emptyset, \emptyset) &= \delta([Num(5), Sum(Num(3), Num(2)), \#MUL], \emptyset, \emptyset) \\ \delta([Num(5), Sum(Num(3), Num(2)), \#MUL], \emptyset, \emptyset) &= \delta([Sum(Num(3), Num(2)), \#MUL], [Num(5)], \emptyset) \\ \delta([Sum(Num(3), Num(2)), \#MUL], [Num(5)], \emptyset) &= \delta([Num(3), Num(2), \#SUM, \#MUL], [Num(5)], \emptyset) \\ \delta([Num(3), Num(2), \#SUM, \#MUL], [Num(5)], \emptyset) &= \delta([Num(2), \#SUM, \#MUL], [Num(3), Num(5)], \emptyset) \\ \delta([Num(2), \#SUM, \#MUL], [Num(3), Num(5)], \emptyset) &= \delta([\#SUM, \#MUL], [Num(2), Num(3), Num(5)], \emptyset) \\ \delta([\#SUM, \#MUL], [Num(2), Num(3), Num(5)], \emptyset) &= \delta([\#MUL], [Num(5), Num(5)], \emptyset) \\ \delta([\#MUL], [Num(5), Num(5)], \emptyset) &= \delta(\emptyset, [Num(25)], \emptyset) \\ \delta(\emptyset, [Num(25)], \emptyset) &= Num(25) \end{aligned}$$

Excerpt of π lib expressions

$\langle \text{Statement} \rangle ::= \langle \text{Exp} \rangle$

$\langle \text{Exp} \rangle ::= \langle \text{ArithExp} \rangle \mid \langle \text{BoolExp} \rangle$

$\langle \text{ArithExp} \rangle ::= \text{'Num'}(\langle \text{digits} \rangle) \mid \text{'Sum'}(\langle \text{Exp} \rangle, \langle \text{Exp} \rangle) \mid$
 $\text{'Sub'}(\langle \text{Exp} \rangle, \langle \text{Exp} \rangle) \mid \text{'Mul'}(\langle \text{Exp} \rangle, \langle \text{Exp} \rangle)$

$\langle \text{BoolExp} \rangle ::= \text{'Eq'}(\langle \text{Exp} \rangle, \langle \text{Exp} \rangle) \mid \text{'Not'}(\langle \text{Exp} \rangle)$

π automata semantics for π lib expressions

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

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

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

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

...

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

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

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

- 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*.

π lib commands

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

$\langle \text{Statement} \rangle ::= \langle \text{Cmd} \rangle$

$\langle \text{Exp} \rangle ::= \text{'Id'}(\langle \text{String} \rangle)$

$\langle \text{Cmd} \rangle ::= \begin{array}{l} \text{'Assign'}(\langle \text{Id} \rangle, \langle \text{Exp} \rangle) \\ | \text{'Loop'}(\langle \text{BoolExp} \rangle, \langle \text{Cmd} \rangle) \\ | \text{'CSeq'}(\langle \text{Cmd} \rangle, \langle \text{Cmd} \rangle) \end{array}$

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

π automata semantics for π 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 $W \in String$, $C, V \in L(G)^*$, $S \in Store$, $E \in Env$, $B \in Bindable$, $l \in Loc$, $T \in Storable$, $X \in \langle Exp \rangle$, $M, M_1, M_2 \in \langle Cmd \rangle$, and

π automata semantics for π 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 ,

$$\delta(Id(W) :: C, V, E, S) = \delta(C, B :: V, E, S), \quad (17)$$

where $E[W] = I$ and $S[I] = B$,

$$\delta(Assign(W, X) :: C, V, E, S) = \delta(X :: \#ASSIGN :: C, W :: V, E, S'), \quad (18)$$

$$\delta(\#ASSIGN :: C, T :: W :: V, E, S) = \delta(C, V, E, S'), \quad (19)$$

where $E[W] = I$ and $S' = S/[I \mapsto T]$,

$$\delta(Loop(X, M) :: C, V, E, S) = \delta(X :: \#LOOP :: C, Loop(X, M) :: V, E, S), \quad (20)$$

$$\delta(\#LOOP :: C, Boo(true) :: Loop(X, M) :: V, E, S) = \delta(M :: Loop(X, M) :: C, V, E, S), \quad (21)$$

$$\delta(\#LOOP :: C, Boo(false) :: Loop(X, M) :: V, E, S) = \delta(C, V, E, S), \quad (22)$$

$$\delta(CSeq(M_1, M_2) :: C, V, E, S) = \delta(M_1 :: M_2 :: C, V, E, S). \quad (23)$$

π 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
```

π 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 ...
```

π 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 ...
```

π 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)))
```

π 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:

π automaton for π 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"
```

π automaton for π 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    ...
```

π automaton for π lib expressions III

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

π automaton for π 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',
    MUL']}
6 {'val': [8], 'cnt': ['#SUM', <__main__.Num object at 0x111851470>]}
7 {'val': [8, 1], 'cnt': ['#SUM']}
8 {'val': [9], 'cnt': []}

```

π 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)
```

π 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))
```

π automaton for π 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"
```

π automaton for π lib commands II

π automaton for commands extends the π 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
```


π automaton for π lib commands III

π semantics for assignment.

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

```

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)

```

π automaton for π lib commands IV

π semantics for identifiers.

$$\delta(Id(W) :: C, V, E, S) = \delta(C, B :: V, E, S),$$

where $E[W] = I$ and $S[I] = B$.

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

π automaton for π lib commands V

π semantics for loop: recursive step.

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

```
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)
```

π automaton for π lib commands VI

π semantics for loop: basic steps.

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

```
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()
```

π automaton for π lib commands VII

π 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)
```

π automaton for π 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 *concrete syntax* for expressions and commands

$\langle prog \rangle ::= \langle cmd \rangle^*$

$\langle cmd \rangle ::= \text{'while' '('} \langle bexp \rangle \text{'')} \text{'do' '{' } \langle cmd \rangle \text{' } \text{'}$
 $\quad \mid \langle id \rangle \text{' := ' } \langle exp \rangle \mid \langle cmd \rangle \text{' ; ' } \langle cmd \rangle$

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

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

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

$\langle factor \rangle ::= \text{'(' } \langle aexp \rangle \text{') ' } \mid \langle digits \rangle \mid \langle id \rangle$

$\langle bexp \rangle ::= \langle id \rangle \mid \langle boolean \rangle \mid \text{' ~ ' } \langle bexp \rangle \mid \langle bexp \rangle \langle boolop \rangle \langle bexp \rangle$
 $\quad \mid \langle aexp \rangle \langle iop \rangle \langle aexp \rangle$

$\langle boolop \rangle ::= \text{' = ' } \mid \text{' /\ ' } \mid \text{' \ / '}$

$\langle iop \rangle ::= \text{' < ' } \mid \text{' > ' } \mid \text{' < = ' } \mid \text{' > = '}$

Example: iterative factorial

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