

Notes on Formal Compiler Construction with the Π Framework

Christiano Braga

Instituto de Computação,
Universidade Federal Fluminense, Niterói, Brazil

August 7, 2020

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



1. Introduction

Example

2. Π IR expressions

Grammar

Automaton

3. Π commands

Grammar

Automaton

4. Π IR declarations

Grammar

Automaton

5. Π IR abstractions

Grammar

Automaton

6. Π IR recursive abstractions

Grammar

Automaton

7. Π^2 : Π Framework in Python

Expressions

Commands

Declarations

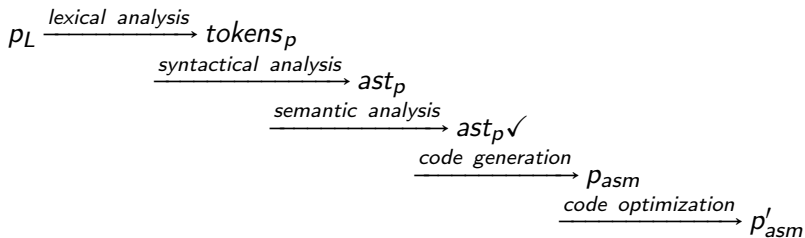
Abstractions

8. IMP language

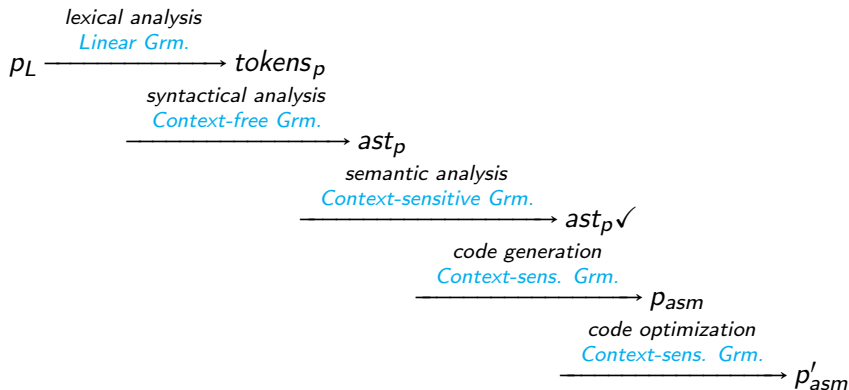
Grammar

Examples

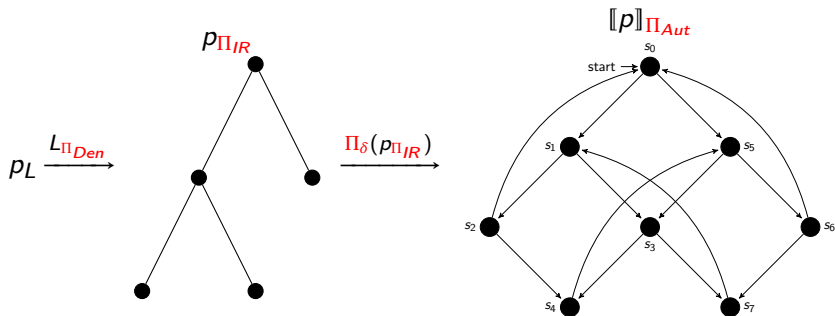
Compiler pipeline



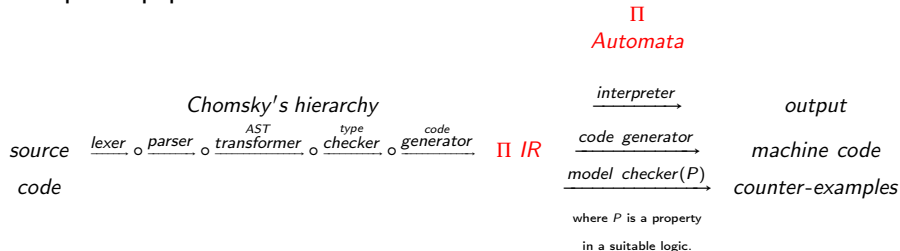
Compiler pipeline and formal languages



Compiler pipeline with the Π Framework I



Compiler pipeline with the Π Framework II



- Π IR defines a set of constructions common to many programming languages.
- Π IR 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 IR program.

Compiler pipeline with the Π Framework III

- Π Framework:
<http://github.com/ChristianoBraga/PiFramework>
- Notes on Formal Compiler Construction with the Π Framework:
<https://github.com/ChristianoBraga/PiFramework/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

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

A calculator: Π denotations I

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

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

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

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

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

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

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

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

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

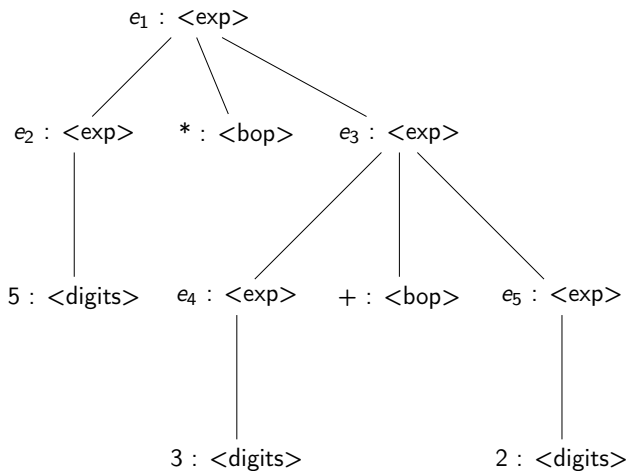
$$\llbracket E_1 > E_2 \rrbracket_{\Pi} = \text{Gt}(\llbracket E_1 \rrbracket_{\Pi}, \llbracket E_2 \rrbracket_{\Pi}) \quad (9)$$

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

A calculator: Π denotations II

- Π denotations are *functions* $\llbracket \cdot \rrbracket_{\Pi} : AST \rightarrow \Pi \text{ IR}$, where *AST* denotes the *datatype* for the abstract syntax tree and $\Pi \text{ IR}$ denotes the datatype for $\Pi \text{ IR}$ 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

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

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

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

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

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

A calculator: executing Π IR 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 IR expressions
$$\langle \textit{Statement} \rangle ::= \langle \textit{Exp} \rangle$$
$$\langle \textit{Exp} \rangle ::= \langle \textit{ArithExp} \rangle \mid \langle \textit{BoolExp} \rangle \langle \textit{Exp} \rangle$$
$$\langle \textit{ArithExp} \rangle ::= \text{'Num'}(\langle \textit{digits} \rangle) \mid \text{'Sum'}(\langle \textit{Exp} \rangle, \langle \textit{Exp} \rangle) \mid \\ \text{'Sub'}(\langle \textit{Exp} \rangle, \langle \textit{Exp} \rangle) \mid \text{'Mul'}(\langle \textit{Exp} \rangle, \langle \textit{Exp} \rangle)$$
$$\langle \textit{BoolExp} \rangle ::= \text{'Eq'}(\langle \textit{Exp} \rangle, \langle \textit{Exp} \rangle) \mid \text{'Not'}(\langle \textit{Exp} \rangle)$$

Π automaton semantics for Π IR expressions I

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

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

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

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

...

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

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

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

- 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 automaton semantics for Π IR expressions II

- On a particular implementation of the Π Framework, as in Python, `<digits>` denote built-in numbers in implementation language, such that all arithmetic operations, such as `+`, are defined. That's why N_i are in \mathbb{N} .

Π IR commands

- Commands are language constructions that require a *memory* store to be evaluated.

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

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

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

Π automaton semantics for Π IR 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.
- $Store = Loc \mapsto Storable$, $Env = Id \mapsto Bindable$, $Loc \subseteq Storable$, $\mathbb{N} \subseteq Loc, Bindable$.
- 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 automaton semantics for Π IR commands II

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

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

$$\text{where } E[W] = I \text{ and } S[I] = B,$$

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

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

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

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

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

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

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

Π IR declarations

- Declarations are statements that create an environment, binding identifiers to (bindable) values.
- In Π IR, a bindable value is either a Boolean value, an integer or a location.
- From a syntactic standpoint, all classes are monotonically extended.

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

$\langle \text{Exp} \rangle ::= \text{'Ref'}(\langle \text{Exp} \rangle) \mid \text{'DeRef'}(\langle \text{Id} \rangle) \mid \text{'ValRef'}(\langle \text{Id} \rangle)$

$\langle \text{Dec} \rangle ::= \text{'Bind'}(\langle \text{Id} \rangle, \langle \text{Exp} \rangle) \mid \text{'DSeq'}(\langle \text{Dec} \rangle, \langle \text{Dec} \rangle)$

$\langle \text{Cmd} \rangle ::= \text{'Blk'}(\langle \text{Dec} \rangle, \langle \text{Cmd} \rangle)$

Π automaton semantics for Π IR declarations I

Let $BlockLocs = \mathcal{P}(Loc)$, now the transition function is $\delta : L(G)^* \times L(G)^* \times Env \times Store \times BlockLocs \rightarrow Q$, and let $L, L' \in BlockLocs$, $Loc \subseteq Storable$, and S/L means the store S without the locations in L ,

Π automaton semantics for Π IR declarations II

$$\delta(\text{Ref}(X) :: C, V, E, S, L) = \delta(X :: \#REF :: C, V, E, S, L), \quad (24)$$

$$\delta(\#REF :: C, T :: V, E, S, L) = \delta(C, I :: V, E, S', L'), \text{where } S' = S \cup [I \mapsto T], I \notin S, L' = L \cup \{I\}, \quad (25)$$

$$\delta(\text{DeRef}(\text{Id}(W)) :: C, V, E, S, L) = \delta(C, I :: V, E, S, L), \text{where } I = E[W], \quad (26)$$

$$\delta(\text{ValRef}(\text{Id}(W)) :: C, V, E, S, L) = \delta(C, T :: V, E, S, L), \text{where } T = S[S[E[W]]], \quad (27)$$

$$\delta(\text{Bind}(\text{Id}(W), X) :: C, V, E, S, L) = \delta(X :: \#BIND :: C, W :: V, E, S, L), \quad (28)$$

$$\delta(\#BIND :: C, B :: W :: H :: V, E, S, L) = \delta(C, [W \mapsto B] :: V, E, S, L), \quad (29)$$

$$\delta(\text{DSeq}(D_1, D_2), X) :: C, V, E, S, L) = \delta(D_1 :: D_2 :: \#DSEQ :: C, V, E, S, L), \quad (30)$$

$$\delta(\#DSEQ :: C, E :: E' :: H :: V, E, S, L) = \delta(C, (E \cup E') :: H :: V, E, S, L), \text{where } E' \in \text{Env}, H \notin \text{Env}, \quad (31)$$

$$\delta(\text{Blk}(D, M) :: C, V, E, S, L) = \delta(D :: \#BLKDEC :: C, M :: L :: V, E, S, \emptyset), \quad (32)$$

$$\delta(\#BLKDEC :: C, E' :: V, E, S, L) = \delta(M :: \#BLKCMD :: C, E :: V, E \setminus E', S, L), \quad (33)$$

$$\delta(\#BLKCMD :: C, E :: L :: V, E', S, L') = \delta(C, V, E, S', L), \text{ where } S' = S/L. \quad (34)$$

Π IR abstractions

- Abstractions extend Bindables by allowing a name to be bound to a list of formal parameters, a list of identifiers, and a block in the environment.
- Such names can be called and applied to actual parameters, a list of expressions.

$\langle Dec \rangle \quad ::= \text{'Bind'}(\langle Id \rangle, \langle Abs \rangle)$

$\langle Abs \rangle \quad ::= \text{'Abs'}(\langle Formals \rangle, \langle Blk \rangle)$

$\langle Formals \rangle \quad ::= \langle Id \rangle^*$

$\langle Cmd \rangle \quad ::= \text{'Call'}(\langle Id \rangle, \langle Actuals \rangle)$

$\langle Actuals \rangle \quad ::= \langle Exp \rangle^*$

Π automaton semantics for Π IR abstractions — Closures I

We chose a *static binding* semantics for abstractions. Therefore, we interpret abstractions as *closures* formed by an abstraction together with its declaration environment which defines the context in which the abstraction will be evaluated.

$$\text{Closure} : \text{Formals} \times \text{Blk} \times \text{Env} \rightarrow \text{Bindable}$$

Π automaton semantics for Π IR abstractions — Example I

```
1 # Recursive factorial
2 var z = 1 ;
3 def fat (x) {
4     if (x > 0) {
5         z := z * x ;
6         fat(x - 1)
7     }
8 }
9 fat(2)
```

Π automaton semantics for Π IR abstractions — Example II

1 Π IR syntax tree:

```
2 Blk(DSeq(Bind(Id(z), Ref(1)),  
3     BindAbs(Id(fat), Abs(Id(x),  
4         Blk(Cond(Gt(Id(x), 0),  
5             Blk(CSeq(Assign(Id(z),  
6                 Mul(Id(z), Id(x))),  
7                 Call(Id(fat), [Sub(Id(x), 1)]))), Nop())))),  
8     Call(Id(fat), [2]))
```

Π automaton semantics for Π IR abstractions I

Let $F \in \text{Formals}$, $B \in \text{Blk}$, $I \in \text{Id}$, $A \in \text{Actuals}$, $V_i \in \text{Value}$, $1 \leq i \leq n$, $n \in \mathbb{N}$,

$$\delta(\text{Abs}(F, B) :: C, V, E, S, L) = \delta(C, \text{Closure}(F, B, E) :: V, E, S, L) \quad (35)$$

$$\begin{aligned} \delta(\text{Call}(I, [X_1, X_2, \dots, X_n])) :: C, V, E, S, L) = \\ \delta(X_n :: X_{n-1} :: \dots :: X_1 :: \# \text{CALL}(I, n) :: C, V, E, S, L) \end{aligned} \quad (36)$$

$$\begin{aligned} \delta(\# \text{CALL}(I, n) :: C, (V_1 :: V_2 :: \dots V_n :: V), E, S, L) = \\ \delta(B :: \# \text{BLKCMD} :: C, E :: V, E', S, L) \end{aligned} \quad (37)$$

where $E = \{I \mapsto \text{Closure}(F, B, E_1)\} \cup E_2$,
 $E' = E / E_1 / \text{match}(F, [V_1, V_2, \dots, V_n])$

Π automaton semantics for Π IR abstractions II

$$match : Id^* \times Values^* \rightarrow Env$$

$$match(fl, al) = \text{if } |fl| \neq |al| \text{ then } \{\} \text{ else } _match(fl, al, \{\})$$

$$_match : Id^* \times Values^* \times Env \rightarrow Env$$

$$_match([], [], E) = E$$

$$_match(f, a, E) = \{f \mapsto a\} \cup E$$

$$_match(f :: fl, a :: al, E) = _match(fl, al, \{f \mapsto a\} \cup E)$$

Π IR recursive abstractions

- Abstractions can be recursive to allow for the declaration of recursive functions.

$\langle Dec \rangle \quad ::= \text{'Rbnd'}(\langle Id \rangle, \langle Abs \rangle)$

Π automaton semantics for Π IR recursive abstractions — Recursive closures I

In the context of *static binding* semantics for abstractions, in a call to a recursive function, the evaluation of identifiers needs to be reminded about the binding of the function name to a closure.

$$Rec : Formals \times Blk \times Env \times Env \rightarrow Bindable$$
$$unfold : Env \rightarrow Env$$
$$reclose_E : Env \rightarrow Env$$

Π automaton semantics for Π IR recursive abstractions — Recursive closures II

$$\text{unfold}(E) = \text{reclose}_E(E) \quad (38)$$

$$\text{reclose}_E(I \mapsto \text{Closure}(F, B, E')) = (I \mapsto \text{Rec}(F, B, E', E)) \quad (39)$$

$$\text{reclose}_E(I \mapsto \text{Rec}(F, B, E', E'')) = (I \mapsto \text{Rec}(F, B, E', E)) \quad (40)$$

$$\text{reclose}_E(I \mapsto v) = (I \mapsto v) \text{ if } v \neq \text{Closure}(F, B, E) \quad (41)$$

$$\text{reclose}_E(E_1 \cup E_2) = \text{reclose}_E(E_1) \cup \text{reclose}_E(E_2) \quad (42)$$

$$\text{reclose}_E(\emptyset) = \emptyset \quad (43)$$

Π automaton semantics for Π IR recursive abstractions — Recursive closures I

$$\begin{aligned} \delta(Rbnd(I, Abs(F, B)) :: C, V, E, S, L) = \\ \delta(C, unfold(I \mapsto Closure(F, B, E)) :: V, E, S, L) \end{aligned} \quad (44)$$

$$\begin{aligned} \delta(\#CALL(I, n) :: C, V_1 :: V_2 :: \dots :: V_n :: V, E, S, L) = \\ \delta(B :: \#BLKCMD :: C, E :: V, E', S, L) \end{aligned} \quad (45)$$

where $E = \{I \mapsto Rec(F, B, E_1, E_2)\} \cup E_3,$
 $E' = E / E_1 / unfold(E_2) / match(F, [V_1, V_2, \dots, V_n])$

Π IR expressions in Python I

```
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 IR 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 IR 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 IR 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 IR 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 IR 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 IR 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 IR 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 IR 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',
    <__main__.Num object at 0x111851630>]}
6 {'val': [8], 'cnt': ['#SUM', <__main__.Num object at 0x111851470>]}
7 {'val': [8, 1], 'cnt': ['#SUM']}
8 {'val': [9], 'cnt': []}
```

Π IR 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 IR 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 IR 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 IR 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 IR 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)
```

Π automaton for Π IR commands IV

```
11 self.updateStore(l, v)
```

Π automaton for Π IR commands V

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 IR commands VI

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 IR commands VII

Π semantics for loop: basic steps.

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

```
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 IR commands VIII

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 IR commands IX

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)
```

Π automaton for Π IR commands X

Π IR declarations in Python I

DeRef and ValRef not implemented yet.

```
1 ## Declarations
2 class Dec(Statement): pass
3 class Bind(Dec):
4     def __init__(self, i, e):
5         assert (isinstance(i, Id) and isinstance(e, Exp))
6         Dec.__init__(self, i, e)
7 class Ref(Exp):
8     def __init__(self, e):
9         assert (isinstance(e, Exp))
10        Exp.__init__(self, e)
11 class Blk(Cmd):
12     def __init__(self, d, c):
13         assert (isinstance(d, Dec) and isinstance(c, Cmd))
14        Cmd.__init__(self, d, c)
```

Π IR declarations in Python II

```
1 class DSeq(Dec):  
2     def __init__(self, d1, d2):  
3         assert (isinstance(d1, Dec) and isinstance(d2, Dec))  
4         Dec.__init__(self, d1, d2)
```

Π IR declarations in Python III

```
1  ## Declarations
2  class DecExpKW(ExpKW):
3      REF = "#REF"
4
5  class DecCmdKW(CmdKW):
6      BLKDEC = "#BLKDEC"
7      BLKCMD = "#BLKCMD"
8
9  class DecKW:
10     BIND = "#BIND"
11     DSEQ = "#DSEQ"
12
13  class DecPiAut(CmdPiAut):
14     def __init__(self):
15         self["locs"] = []
16         CmdPiAut.__init__(self)
17
18     def locs(self):
```

Π IR declarations in Python IV

```
19     return self["locs"]
20
21 def pushLoc(self, l):
22     ls =self.locs()
23     ls.append(l)
24
25 def __evalRef(self, e):
26     ex =e.opr[0]
27     self.pushCnt(DecExpKW.REF)
28     self.pushCnt(ex)
29
30 def __newLoc(self):
31     sto =self.sto()
32     if sto:
33         return max(list(sto.keys())) +1
34     else:
35         return 0.0
36
37 def __evalRefKW(self):
```

Π IR declarations in Python V

```
38     v =self.popVal()
39     l =self.__newLoc()
40     self.updateStore(l, v)
41     self.pushLoc(l)
42     self.pushVal(l)
43
44 def __evalBind(self, d):
45     i =d.opr[0]
46     e =d.opr[1]
47     self.pushVal(i)
48     self.pushCnt(DecKW.BIND)
49     self.pushCnt(e)
50
51 def __evalBindKW(self):
52     l =self.popVal()
53     i =self.popVal()
54     x =i.opr[0]
55     self.pushVal({x: l})
56
```

Π IR declarations in Python VI

```
57 def __evalDSeq(self, ds):
58     d1 =ds.opr[0]
59     d2 =ds.opr[1]
60     self.pushCnt(DecKW.DSEQ)
61     self.pushCnt(d2)
62     self.pushCnt(d1)
63
64 def __evalDSeqKW(self):
65     d2 =self.popVal()
66     d1 =self.popVal()
67     d1.update(d2)
68     self.pushVal(d1)
69
70 def __evalBlk(self, d):
71     ld =d.opr[0]
72     c =d.opr[1]
73     l =self.locs()
74     self.pushVal(list(l))
75     self.pushVal(c)
```

Π IR declarations in Python VII

```
76     self.pushCnt(DecCmdKW.BLKDEC)
77     self.pushCnt(1d)
78
79     def __evalBlkDecKW(self):
80         d =self.popVal()
81         c =self.popVal()
82         l =self.locs()
83         self.pushVal(l)
84         en =self.env()
85         ne =en.copy()
86         ne.update(d)
87         self.pushVal(en)
88         self["env"] =ne
89         self.pushCnt(DecCmdKW.BLKCMD)
90         self.pushCnt(c)
91
92     def __evalBlkCmdKW(self):
93         en =self.popVal()
94         ls =self.popVal()
```

Π IR declarations in Python VIII

```
05     self["env"] =en
06     s =self.sto()
07     s ={k: v for k, v in s.items() if k not in ls}
08     self["sto"] =s
09     # del ls
00     ols =self.popVal()
01     self["locs"] =ols
02
03 def eval(self):
04     d =self.popCnt()
05     if isinstance(d, Bind):
06         self.__evalBind(d)
07     elif d ==DecKW.BIND:
08         self.__evalBindKW()
09     elif isinstance(d, DSeq):
10         self.__evalDSeq(d)
11     elif d ==DecKW.DSEQ:
12         self.__evalDSeqKW()
13     elif isinstance(d, Ref):
```


Π IR declarations in Python IX

```
14         self.__evalRef(d)
15     elif d == DecExpKW.REF:
16         self.__evalRefKW()
17     elif isinstance(d, Blk):
18         self.__evalBlk(d)
19     elif d == DecCmdKW.BLKDEC:
20         self.__evalBlkDecKW()
21     elif d == DecCmdKW.BLKCMD:
22         self.__evalBlkCmdKW()
23     else:
24         self.pushCnt(d)
25         CmdPiAut.eval(self)
```

Π IR declarations in Python X

```
1 dc =DecPiAut()
2 fac =Loop(Not(Eq(Id("y"), Num(0))),
3           CSeq(Assign(Id("x"), Mul(Id("x"), Id("y"))),
4               Assign(Id("y"), Sub(Id("y"), Num(1))))))
5 dec =DSeq(Bind(Id("x"), Ref(Num(1))),
6           Bind(Id("y"), Ref(Num(200))))
7 fac_blk =Blk(dec, fac)
8 dc.pushCnt(fac_blk)
9 while not dc.emptyCnt():
10     aux =dc.copy()
11     dc.eval()
12     if dc.emptyCnt():
13         print(aux)
```

Π IR abstractions in Python I

```
1
2 class Formals(list):
3     def __init__(self, f):
4         if isinstance(f, list):
5             for a in f:
6                 if not isinstance(a, Id):
7                     raise IllFormed(self, a)
8             self.append(f)
9         else:
10            raise IllFormed(self, f)
11
12 class Abs:
13     def __init__(self, f, b):
14         if isinstance(f, list):
15             if isinstance(b, Blk):
16                 self._opr = [f, b]
17             else:
18                 raise IllFormed(self, b)
```

Π IR abstractions in Python II

```
19     else:
20         raise IllFormed(self, f)
21
22     def formals(self):
23         return self._opr[0]
24
25     def blk(self):
26         return self._opr[1]
27
28     def __str__(self):
29         ret =str(self.__class__.__name__) +"("
30         formals =self.formals()
31         ret +=str(formals[0]) # First formal argument
32         for i in range(1, len(formals)):
33             ret +=", "
34             ret +=str(formals[i]) # Remaining formal arguments
35         ret +=", "
36         ret +=str(self.blk()) # Abstraction block
37         ret +=")"
```

Π IR abstractions in Python III

```
38         return ret
39
40 class BindAbs(Bind):
41     '''
42     BindAbs is a form of bind but that receives an Abs instead of an
43     expression.
44     '''
45     def __init__(self, i, p):
46         if isinstance(i, Id):
47             if isinstance(p, Abs):
48                 Dec.__init__(self, i, p)
49             else:
50                 raise IllFormed(self, p)
51         else:
52             raise IllFormed(self, i)
53
54 class Actuals(list):
55     def __init__(self, a):
56         if isinstance(a, list):
```

Π IR abstractions in Python IV

```
57     for e in a:
58         if not isinstance(e, Exp):
59             raise IllFormed(self, e)
60     self.append(a)
61 else:
62     raise IllFormed(self, a)
63
64 class Call(Cmd):
65     def __init__(self, f, actuals):
66         if isinstance(f, Id):
67             if isinstance(actuals, list):
68                 Cmd.__init__(self, f, actuals)
69             else:
70                 raise IllFormed(self, actuals)
71         else:
72             raise IllFormed(self, f)
73
74     def caller(self):
75         return self.operand(0)
```

Π IR abstractions in Python V

```
76
77 def actuals(self):
78     return self.operand(1)
79
80 class Closure(dict):
81     def __init__(self, f, b, e):
82         if isinstance(f, list):
83             if isinstance(b, Blk):
84                 # I wanted to write assert(isinstance(e, Env)) but it
85                                     fails.
86
87                 if isinstance(e, dict):
88                     self["for"] = f # Formal parameters
89                     self["env"] = e # Current environment
90                     self["block"] = b # Procedure block
91                 else:
92                     raise IllFormed(self, e)
93             else:
94                 raise IllFormed(self, b)
95         else:
```

Π IR abstractions in Python VI

```
04         raise IllFormed(self, f)
05
06     def __str__(self):
07         ret =str(self.__class__.__name__) +"("
08         formals =self.formals()
09         fst_formal =formals[0] # First formal argument
10         ret +=str(fst_formal)
11         for i in range(1, len(formals)):
12             ret +=", "
13             formal =formals[i] # Remaining formal arguments
14             ret +=str(formal)
15         ret +=", "
16         ret +=str(self.blk()) # Closure block
17         ret +=")"
18         return ret
19
20     def formals(self):
21         return self['for']
```


Π IR abstractions in Python VII

```
13 def env(self):
14     return self['env']
15
16 def blk(self):
17     return self['block']
18
19 class AbsPiAut(DecPiAut):
20     def __evalAbs(self, a):
21         if not isinstance(a, Abs): # p must be an abstraction
22             raise EvaluationError(self, "Function __evalAbs called with
                                         no abstraction but with "
                                         , a, " instead.")
23         else:
24             f =a.formals() # Formal parameters
25             b =a.blk() # Body
26             e =self.env() # Current environment
27             # Closes the given abs. with the current env
28             c =Closure(f, b, e)
29             # Closure c is pushed to the value stack such that
```

Π IR abstractions in Python VIII

```
30         self.pushVal(c)
31         # a BIND may create a new binding to a given identifier.
32
33     def __match(self, f, a):
34         '''
35         Given a list of formal parameters and a list of actual parameters
36         it returns an environment relating the elements of the former
37         with the latter.
38         '''
39         if isinstance(f, list):
40             if isinstance(a, list):
41                 if len(f) == 0:
42                     return {}
43                 if len(f) == len(a) and len(f) > 0:
44                     # For some reason, f[0] is a tuple, not an Id.
45                     f0 = f[0]
46                     a0 = a[0]
47                     b0 = {f0.id(): a0.num()}
```

Π IR abstractions in Python IX

```
47     if len(f) == 1:
48         return b0
49     else:
50         # For some reason, f[0] is a tuple, not an Id.
51         f1 = f[1]
52         a1 = a[1]
53         b1 = {f1.id(): a1.num()}
54         e = b0.update(b1)
55         for i in range(2, len(f)):
56             fi = f[i][0]
57             ai = a[i][0]
58             e.update({fi.id(): ai.num()})
59         return e
60     else:
61         raise EvaluationError("Call to '.__match' on " + str(self) +
62                                ": " + "formals and"
63                                "actuals differ in"
64                                "size.")
65 else:
```

Π IR abstractions in Python X

```
raise EvaluationError("Call to '.__match' on " +str(self) +":  
                        " + " no formals, but with  
                        ", f, " instead.")
```

```
def __evalCall(self, c):  
    '''
```

Essentially, a call is translated into a block.

If we were programming π in a symbolic language,

*we could simply create a proper block and push it to the control
stack.*

*However, the environment is not symbolic: is a dictionary of
objects.*

*To create a block we would need to "pi-IR-fy" it, that is,
recreate the*

*pi IR tree from the concrete environment and joint it with
matches created*

*also at pi IR level. These would be pushed back into the control
stack and*

Π IR abstractions in Python XI

```
74     reobjectified. Thus, to avoid pi-IRfication and reevaluation
75         of the
76     environment we manipulate it at the object level, which is
77         dangerous but
78     seems to be correct.
79
80     In this implementation, actual parameters are already evaluated.
81     '''
82     if not isinstance(c, Call): # c must be a Call object
83         raise EvaluationError("Call to __evalCall with no Call object
84                                but with ", c, " instead
85                                .")
86
87     else:
88         # Procedure to be called
89         caller =c.caller()
90         # Retrieves the current environment.
91         e =self.env()
92         # Retrieves the closure associated with the caller function.
93         clos =e[caller.id()]
```

Π IR abstractions in Python XII

```

89 # Retrieves the actual parameters from the call.
90 a =c.actuals()
91 # Retrieves the formal parameters from the closure.
92 f =clos.formals()
93 # Matches formals and actuals, creating an environment.
94 d =self.__match(f, a)
95 # Retrives the closure's environment.
96 ce =clos.env()
97 # The caller's block must run on the closures environment
98 # overwritten with the matches.
99 d.update(ce)
100 self["env"] =d
101 self.pushVal(self.locs())
102 # Saves the current environment in the value stack.
103 self.pushVal(e)
104 # Pushes the keyword BLKCMD for block completion.
105 self.pushCnt(DecCmdKW.BLKCMD)
106 # Pushes the body of the caller function into the control
      stack.

```

Π IR abstractions in Python XIII

```
07         self.pushCnt(clos.blk())
08
09     def eval(self):
10         d =self.popCnt()
11         if isinstance(d, Abs):
12             self.__evalAbs(d)
13         elif isinstance(d, Call):
14             self.__evalCall(d)
15         else:
16             self.pushCnt(d)
17             DecPiAut.eval(self)
```

Complete Imp grammar in EBNF notation I

```

1 @@grammar::IMP
2 @@eol_comments ::/#.*?/start = @:cmd ;
3
4
5
6 cmd =nop | let | assign | loop | call ;
7
8 call =i:identifier '(' { a:actual }* ')' ;
9
10 actual =e1:expression { ',' e2:expression }* | {} ;
11
12 nop ='nop' ;
13
14 loop =op:'while' ~ e:expression 'do' { c:cmd }+ ;
15
16 assign =id:identifier op:':=' ~ e:expression ;
17
18 let =op:'let' ~ d:dec 'in' { c:cmd }+ ;

```


Complete Imp grammar in EBNF notation II

```

19 dec =var | fn ;
20
21
22 var =op:'var' ~ id:identifier '=' e:expression ;
23
24 fn =op:'fn' ~ id:identifier '(' f:formal ')' '=' c:cmd ;
25
26 formal =i1:identifier { ',' i2:identifier }* | {} ;
27
28 expression =@:bool_expression ;
29
30 bool_expression =negation | equality | conjunction | disjunction
31                  | lowereq | greatereq | lowerthan | greaterthan
32                  | add_expression ;
33
34 equality =left:add_expression op:"==" ~ right:bool_expression ;
35
36 conjunction =left:add_expression op:"and" ~ right:bool_expression ;
37

```

Complete Imp grammar in EBNF notation III

```

38 disjunction =left:add_expression op:"or" ~ right:bool_expression ;
39
40 lowereq =left:add_expression op:"<=" ~ right:add_expression ;
41
42 greatereq =left:add_expression op:">=" ~ right:add_expression ;
43
44 lowerthan =left:add_expression op:"<" ~ right:add_expression ;
45
46 greaterthan =left:add_expression op:">" ~ right:add_expression ;
47
48 parenthesisexp ='(' ~ @:bool_expression ')';
49
50 negation =op:'not' ~ b:bool_expression ;
51
52 add_expression =addition | subtraction | @:mult_expression ;
53
54 addition =left:mult_expression op:"+" ~ right:add_expression ;
55
56 subtraction =left:mult_expression op:"- " ~ right:add_expression ;

```

Complete Imp grammar in EBNF notation IV

```
57
58 mult_expression =multiplication | division
59                 | atom
60                 | parenthesisexp ;
61
62 multiplication =left:atom op:"*" ~ right:mult_expression ;
63
64 division =left:atom op:"/" ~ right:mult_expression ;
65
66 atom =number | truth | identifier ;
67
68 number =/\d+/ ;
69
70 identifier =/(?!\d)\w+/ ;
71
72 truth ='True' | 'False' ;
```

Example: iterative factorial

```
1 # The classic iterative factorial example
2 let var z = 1
3 in
4   let var y = 10
5   in
6     while not (y == 0)
7     do
8       z := z * y
9       y := y - 1
```

Example: iterative factorial within a function

```
1 # In this example we encapsulate the iterative calculation
2 # of the factorial within a function call.
3 let var z = 1
4 in
5   let fn f(x) =
6     let var y = x
7     in
8       while not (y == 0)
9       do
10         z := z * y
11         y := y - 1
12   in f(10)
```