

Notes on Formal Compiler Construction with the π Framework

Christiano Braga

Instituto de Computação,
Universidade Federal Fluminense, Niterói, Brazil

June 6, 2019

<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

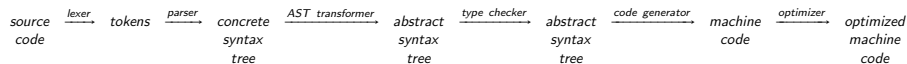
Abstractions

8. IMP language

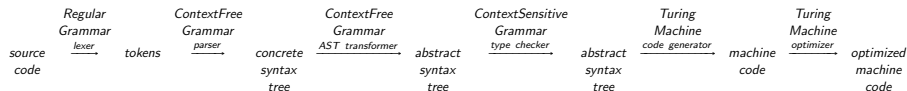
Grammar

Examples

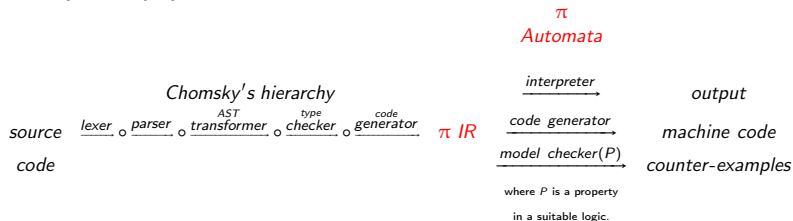
Compiler pipeline



Compiler pipeline and formal languages



Compiler pipeline with the π Framework



- 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.
- 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

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

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

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

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 $\langle \text{exp} \rangle$,

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

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

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

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

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

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

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

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

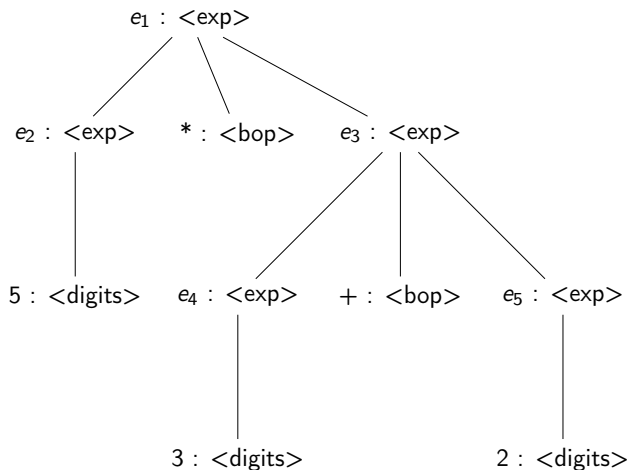
$$\llbracket E_1 > E_2 \rrbracket_{\pi} = \text{Gt}(\llbracket E_1 \rrbracket_{\pi}, \llbracket E_2 \rrbracket_{\pi}) \quad (9)$$

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

A calculator: π denotations II

- π denotations are *functions* $\llbracket \cdot \rrbracket_{\pi} : AST \rightarrow \pi \text{ IR}$, 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

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

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

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

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

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

A calculator: executing π 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

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

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

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

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

π 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.

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

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

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

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.
- $Store = Id \mapsto Storable$, $Env = Loc \mapsto Bindable$, $Loc \subseteq Store$, $\mathbb{N} \subseteq Loc, Bindable$.
- 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 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 ,

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

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 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 $BlockLocs = Set\{Loc\}$, 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

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

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

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

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

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

$$\delta(\#BIND :: C, B :: W :: E' :: V, E, S, L) = \delta(C, ([W \mapsto B] \cup E') :: V, E, S, L), \text{ where } E' \in \text{Env}, \quad (29)$$

$$\delta(\#BIND :: C, B :: W :: H :: V, E, S, L) = \delta(C, [W \mapsto B] :: H :: V, E, S, L), \text{ where } H \notin \text{Env}, \quad (30)$$

$$\delta(\text{DSeq}(D_1, D_2), X) :: C, V, E, S, L) = \delta(D_1 :: D_2 :: C, V, E, S, L), \quad (31)$$

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

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

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

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.

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

π automaton semantics for π IR abstractions — Example I

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

π automaton semantics for π IR abstractions — Example II

```
1  $\pi$  IR AST:  
2 Blk(Bind(Id(z), Ref(Num(1))),  
3   Blk(BindAbs(Id(f), Abs(Id(x),  
4     Blk(Bind(Id(y), Ref(Id(x))),  
5       Loop(Not(Eq(Id(y), Num(0))),  
6         CSeq(Assign(Id(z), Mul(Id(z), Id(y))),  
7           Assign(Id(y), Sub(Id(y), Num(1)))))))))  
8   Call(Id(f), Num(10)))
```

π automaton semantics for π IR abstractions I

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

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

$$\delta(\text{Call}(I, [X_1, X_2, \dots, X_n])) :: C, V, E, S, L) = \quad (36)$$

$$\delta(X_n :: X_{n-1} :: \dots :: X_1 :: \# \text{CALL}(I, n) :: C, V, E, S, L)$$

$$\delta(\# \text{CALL}(I, n) :: C, [V_1, V_2, \dots, V_n] :: V, [I \mapsto \text{Closure}(F, B, E_1)] E_2, S, L) = \quad (37)$$

$$\delta(B :: \# \text{BLKCMD} :: C, E_2 :: V, (E_1 / \text{match}(F, [V_1, V_2, \dots, V_n])), S, L)$$

$$\text{match} : \text{Id}^* \times \text{Values}^* \rightarrow \text{Env}$$

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

$$_ \text{match} : \text{Id}^* \times \text{Values}^* \times \text{Env} \rightarrow \text{Env}$$

$$_ \text{match}([], [], E) = E$$

$$_ \text{match}(f, a, E) = \{f \mapsto a\} E$$

$$_ \text{match}(f :: fl, a :: al, E) = _ \text{match}(fl), al, \{f \mapsto a\} E$$

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

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

$$\begin{aligned} \delta(\#CALL(I, n) :: C, [V_1, V_2, \dots, V_n] :: V, [I \mapsto Rec(F, B, E_1, E_2)] \cup E_3, S, L) = \\ \delta(B :: \#BLKCMD :: C, E_2 :: V, E_1 / match(F, [V_1, V_2, \dots, V_n]), S, L) \end{aligned} \quad (45)$$

π IR expressions in Python I

<https://nbviewer.jupyter.org/github/ChristianoBraga/PiFramework/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 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)
11    self.updateStore(l, v)
```

π automaton for π IR 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 IR 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 IR 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 IR 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 IR 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)
```

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 Cns(Exp):
12      def __init__(self, e):
13          assert (isinstance(e, Exp))
14         Exp.__init__(self, e)
15  class Blk(Cmd):
16      def __init__(self, d, c):
```

π IR declarations in Python II

```
17 assert (isinstance(d, Dec) and isinstance(c, Cmd))  
18 Cmd.__init__(self, d, c)
```

π IR declarations in Python III

```
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 IV

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

π IR declarations in Python V

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


π IR declarations in Python VI

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

π IR declarations in Python VII

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

π IR declarations in Python VIII

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

π IR declarations in Python IX

```
95     ls =self.popVal()
96     self["env"] =en
97     s =self.sto()
98     s ={k: v for k, v in s.items() if k not in ls}
99     self["sto"] =s
100     # del ls
101     ols =self.popVal()
102     self["locs"] =ols
103
104 def eval(self):
105     d =self.popCnt()
106     if isinstance(d, Bind):
107         self.__evalBind(d)
108     elif d ==DecKW.BIND:
109         self.__evalBindKW()
110     elif isinstance(d, DSeq):
111         self.__evalDSeq(d)
112     elif d ==DecKW.DSEQ:
113         self.__evalDSeqKW()
```

π IR declarations in Python X

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

π IR declarations in Python XI

```
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

```
94         raise IllFormed(self, f)
95
96     def __str__(self):
97         ret =str(self.__class__.__name__) +"("
98         formals =self.formals()
99         fst_formal =formals[0] # First formal argument
100         ret +=str(fst_formal)
101         for i in range(1, len(formals)):
102             ret +=", "
103             formal =formals[i] # Remaining formal arguments
104             ret +=str(formal)
105         ret +=", "
106         ret +=str(self.blk()) # Closure block
107         ret +=")"
108         return ret
109
110     def formals(self):
111         return self['for']
112
```

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
23                                     no abstraction but with "
24                                     , a, " instead.")
25
26         else:
27             f = a.formals() # Formal parameters
28             b = a.blk() # Body
29             e = self.env() # Current environment
30             # Closes the given abs. with the current env
31             c = Closure(f, b, e)
32             # 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]
```

```
         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) +
                                ": " +"formals and
                                actuals differ in
                                size.")
62 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

reobjectified. Thus, to avoid pi-IRfication and reevaluation of the environment we manipulate it at the object level, which is dangerous but seems to be correct.

In this implementation, actual parameters are already evaluated.
'''

```
if not isinstance(c, Call): # c must be a Call object
    raise EvaluationError("Call to __evalCall with no Call object
                           but with ", c, " instead
                           .")
else:
    # Procedure to be called
    caller = c.caller()
    # Retrieves the current environment.
    e = self.env()
    # Retrieves the closure associated with the caller function.
    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)
```