Language Definitions

Sujit Kumar Chakrabarti
sujitkc@iiitb.ac.in

January 2025

spec	→	(g : $Decl*$, t: $TypeDecl*$, i : $Init$, f : $Function*$, b : $Block*$)
$Decl$	→	(n : `string`, t : $TypeExpr$)
$FunDecl$	→	(n : `string`, p : $TypeExpr$, r : $TypeExpr$)
$TypeDecl$	→	$VariantDecl$ | $RecordDecl$
$VariantDecl$	→	$VariantConstructor+$
$RecordDecl$	→	$recname$: `string`, $fields$: $Decl+$
$TypeExpr$	→	$TypeConst$ | $TypeVariable$
	|	$FuncType$
	|	$MapType$
	|	$TupleType$
	|	$SetType$
	|	(More will be added as we get more examples)
$TypeConst$	→	(n : `string`)
$FuncType$	→	(p : $TypeExpr*$, r: $TypeExpr$)
$MapType$	→	(d : $TypeExpr$, r: $TypeExpr$)
$TupleType$	→	(et : $TypeExpr+$)
$SetType$	→	(et : $TypeExpr$)
$Init$	→	(v : `string`, e : $Expr$)
Functions		
$FuncDecl$	→	(n : `string`, p : $Decl*$, r : `pair`< $HTTPResponseCode$, $TypeExpr$ >)
APIs		
API	→	(pre : $Expr$, $call$: $FuncCall$, $resp$: (ret : $HTTPResponseCode$, $resp$: $post$: $Expr$))

Figure 1: Abstract Syntax: API Specification Language

$Expr$	→	Var |$FuncCall$ |Num |Set |Map |$Tuple$
	|	(More will be added as we get more examples)
Var	→	(n : `string`)
FuncCall	→	(n : `string`, a : $Expr*$)
Num	→	(v : `int`)
Set	→	(e : $Expr*$)
Map	→	(v : `pair`< Var, $Expr$ >)
Tuple	→	(e : $Expr$)

Figure 2: Abstract Syntax: Expressions

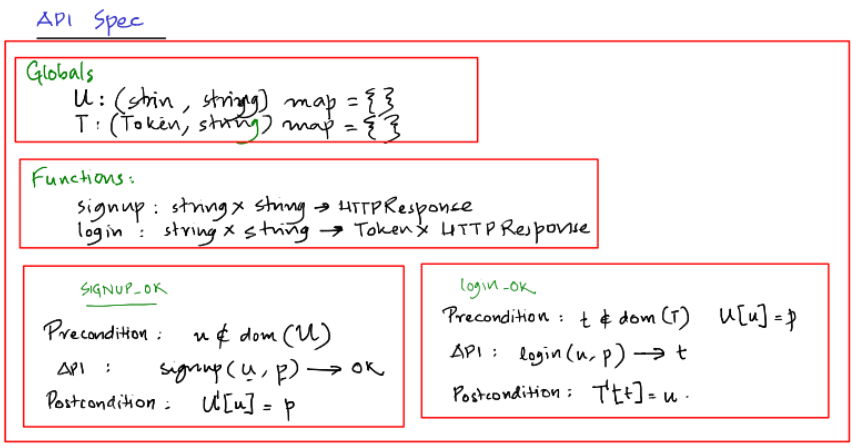


Figure 3: Example: API specification

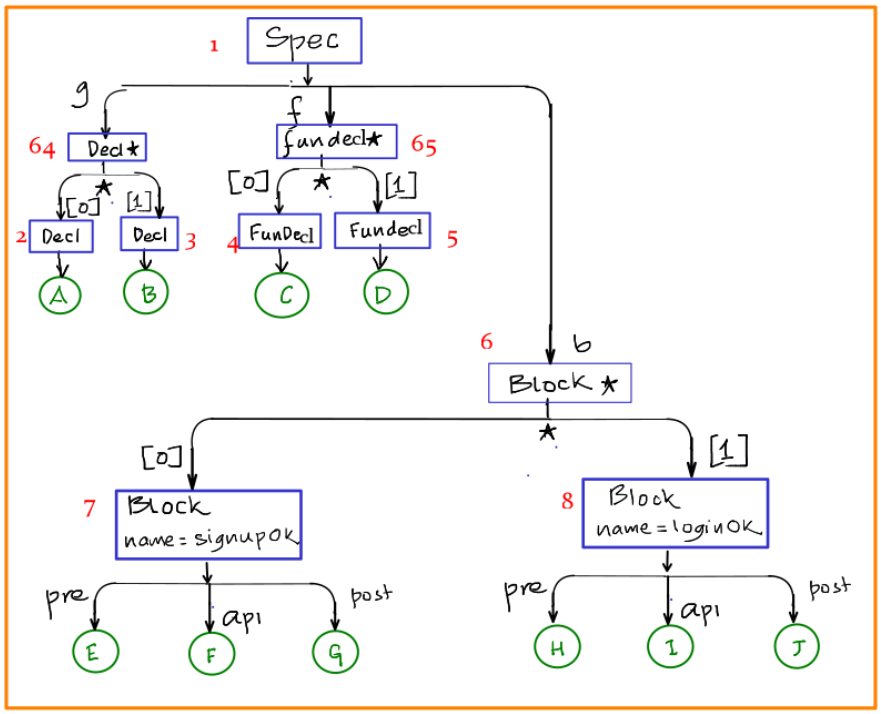
1 API Specification Language

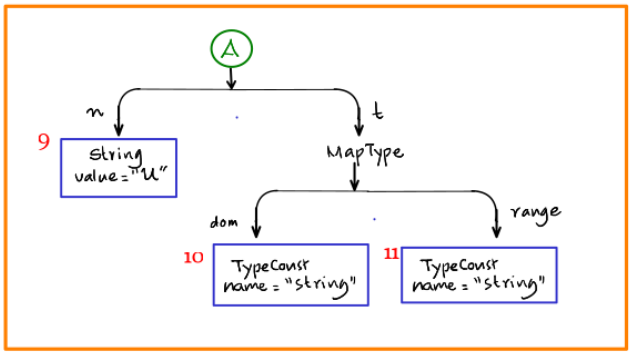
1.1 Abstract Syntax

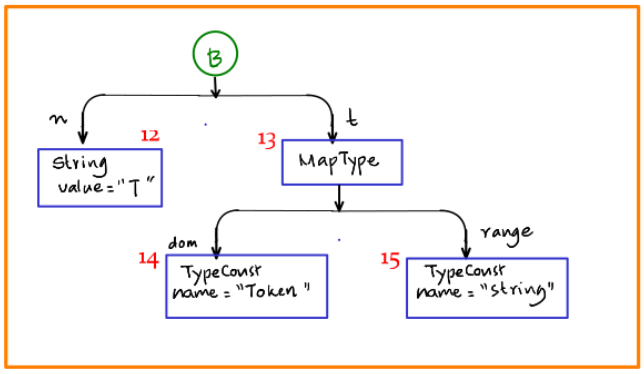
1.2 Example

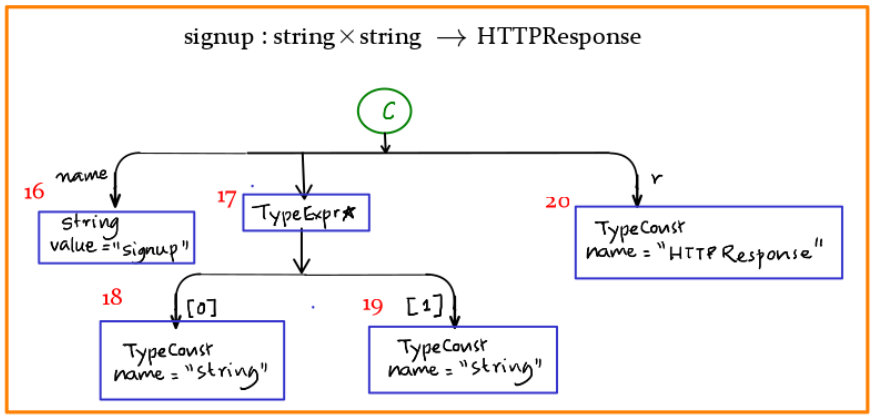
1.3 Example Specification – Signup-Login API

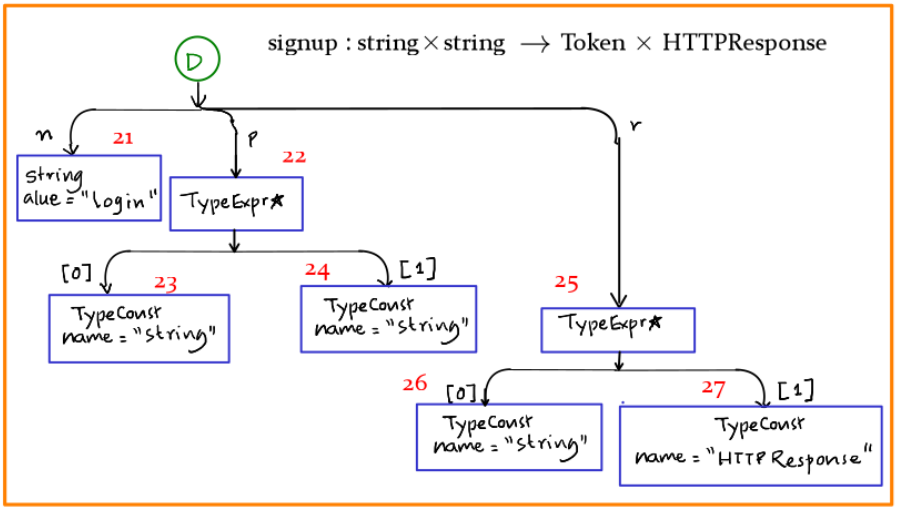
1.3.1 Abstract Syntax Tree

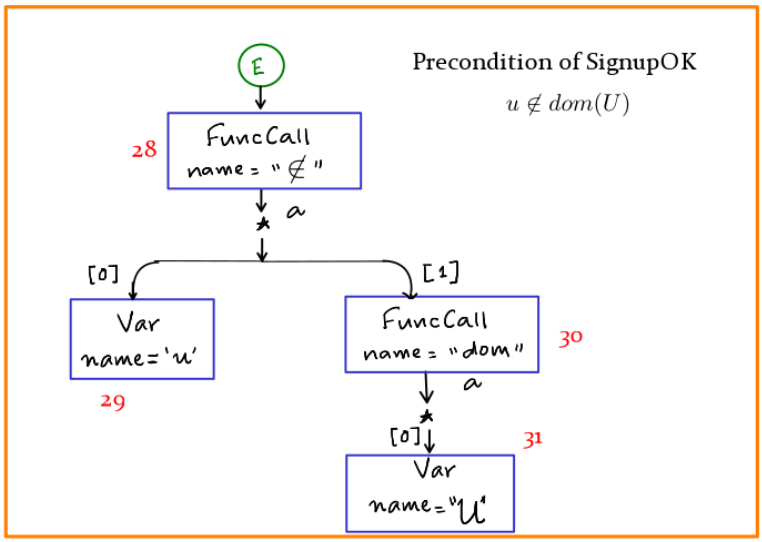


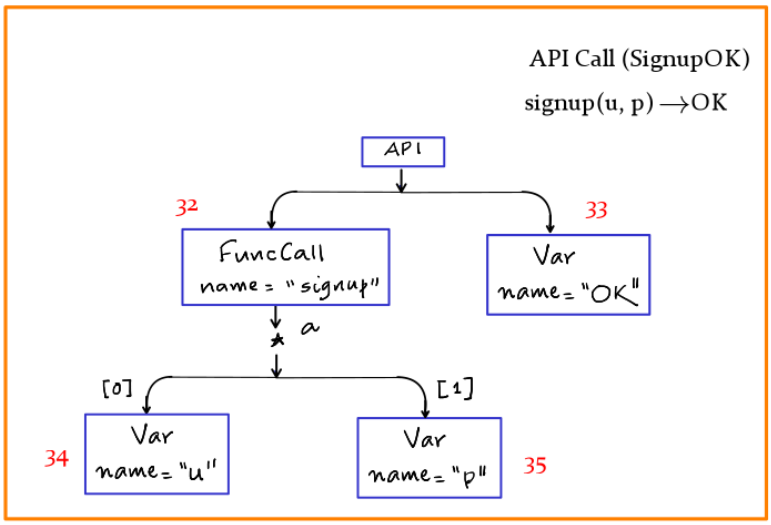

A
n
t
9
String
value = "U"
MapType
dom
range
10
TypeConstr
name = "string"
11
TypeConstr
name = "string"

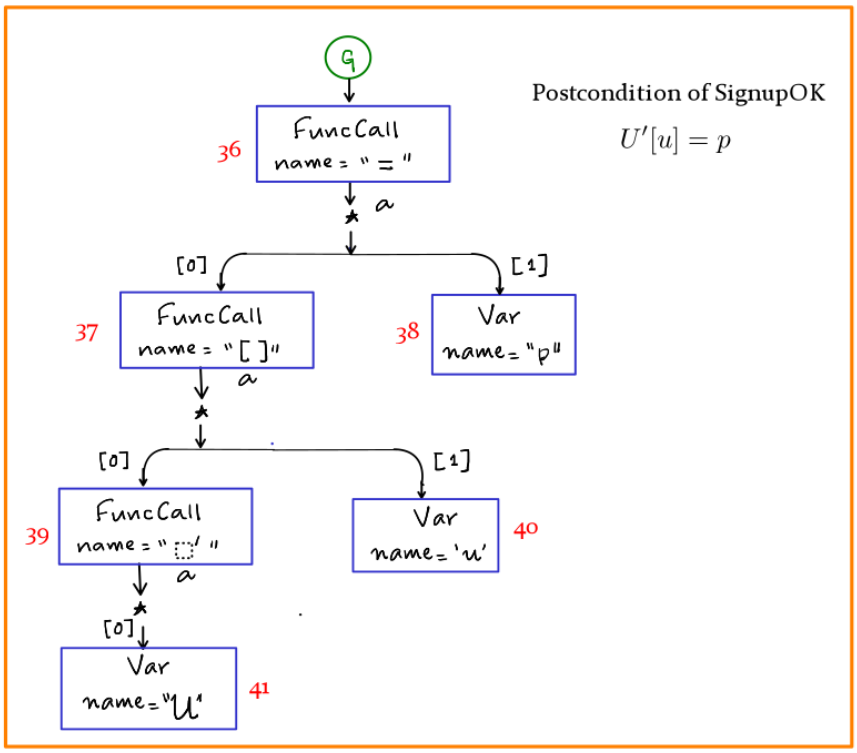

B
n
t
12
String
value = "T"
13
MapType
dom
range
14
TypeConstr
name = "Token"
15
TypeConstr
name = "string"

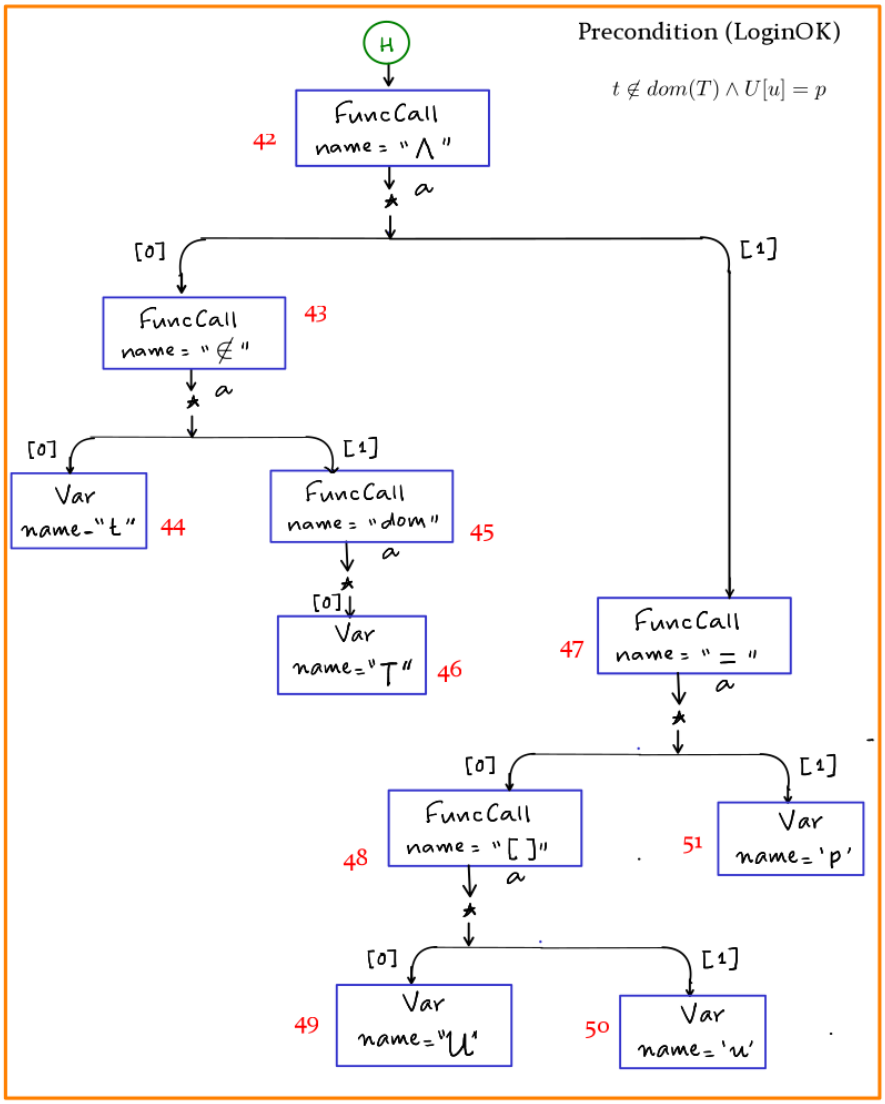

signup : string × string → HTTPResponse
C
name
r
16
17
20
String
value = "signup"
TypeExpr*
TypeConstr
name = "HTTPResponse"
18
[0]
19
[1]
TypeConstr
name = "string"
TypeConstr
name = "string"

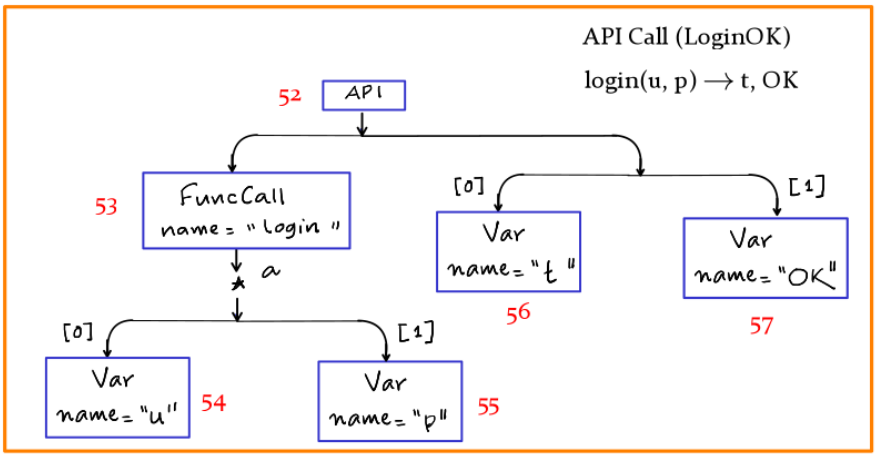

signup : string × string → Token × HTTPResponse
D
n
21
p
22
r
String
alue = "login"
TypeExpr*
[0]
23
24
[1]
25
TypeConstr
name = "string"
TypeConstr
name = "string"
TypeExpr*
26
[0]
27
[1]
TypeConstr
name = "string"
TypeConstr
name = "HTTPResponse"

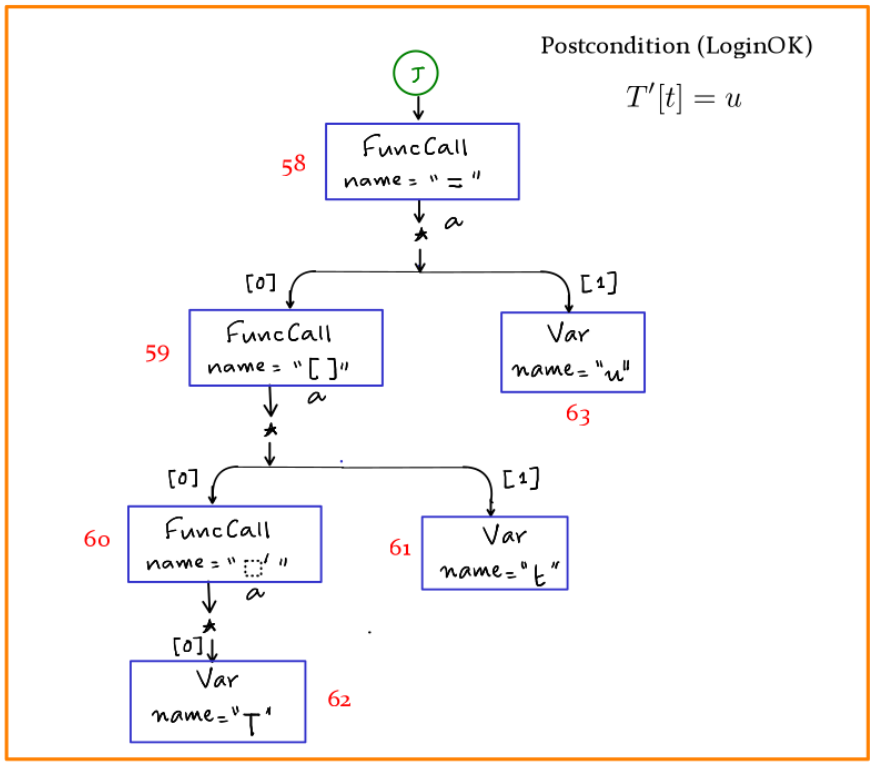

Precondition of SignupOK
$u \notin dom(U)$
E
28
FuncCall
name = "∉"
a
[0]
[1]
Var
name='u'
29
FuncCall
name = "dom"
30
a
[0]
31
Var
name="U"


API Call (SignupOK)
signup(u, p) ⟶OK
API
32
33
FuncCall
name = "signup"
Var
name="OK"
a
[0]
[1]
34
Var
name="u"
Var
name="p"
35


G
36
FuncCall
name = " = "
a
[0]
[1]
37
FuncCall
name = " [ ] "
a
38
Var
name = "p"
[0]
[1]
39
FuncCall
name = " □' "
a
40
Var
name = 'u'
[0]
Var
name = "u"
41
Postcondition of SignupOK
$U'[u] = p$


Precondition (LoginOK)
$t \notin dom(T) \wedge U[u] = p$
H
FuncCall
name = "∧"
42
a
[0]
[1]
FuncCall
name = "∉"
43
a
[0]
[1]
Var
name="t"
44
FuncCall
name = "dom"
45
a
[0]
Var
name="T"
46
FuncCall
name = "="
47
a
[0]
[1]
FuncCall
name = "[ ]"
48
a
Var
name='p'
51
[0]
[1]
Var
name="U"
49
Var
name='u'
50


API Call (LoginOK)
login(u, p) ⟶ t, OK
52
API
53
FuncCall
name = "login"
a
[0]
[1]
Var
name = "u"
54
Var
name = "p"
55
[0]
[1]
Var
name = "t"
56
Var
name = "OK"
57


Postcondition (LoginOK)
$T'[t] = u$
T
58
FuncCall
name = "="
a
[0]
[1]
59
FuncCall
name = "[]"
a
Var
name = "u"
63
[0]
[1]
60
FuncCall
name = "□'"
a
61
Var
name = "t"
[0]
Var
name = "T'"
62

$Program$	$\rightarrow$	$(s : Stmt*)$
	$\mid$	(More will be added as we get more examples)
$Stmt$	$\rightarrow$	$Assign \mid FuncCallStmt$
$Assign$	$\rightarrow$	$(l : Var, r : Expr)$
$FuncCallStmt$	$\rightarrow$	$(f : FuncCall)$

Figure 4: Abstract Syntax: Abstract Test Cases

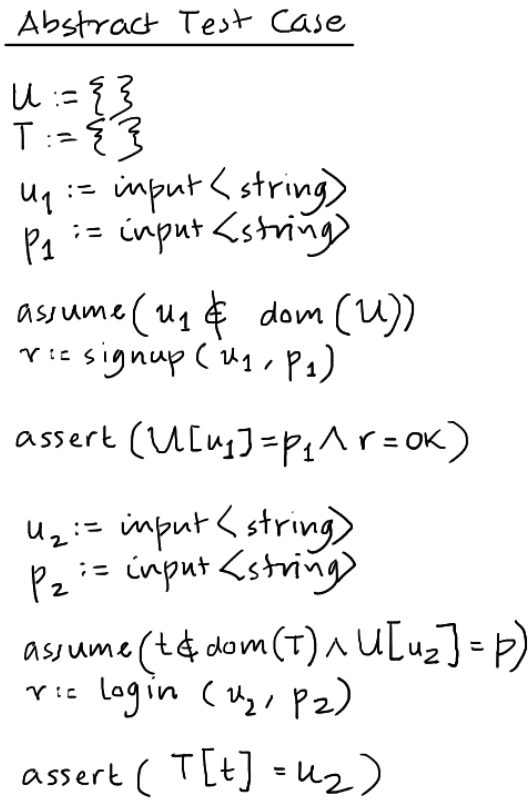


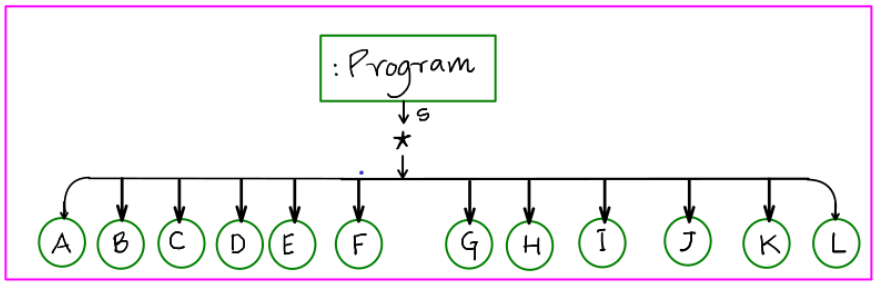
Figure 5: Example – Abstract test case

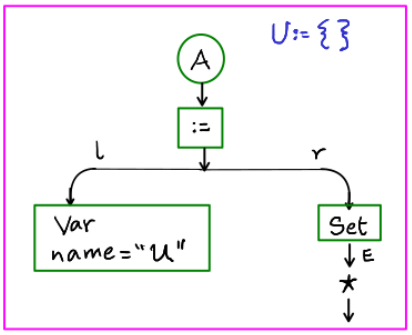
2 Abstract Test Case

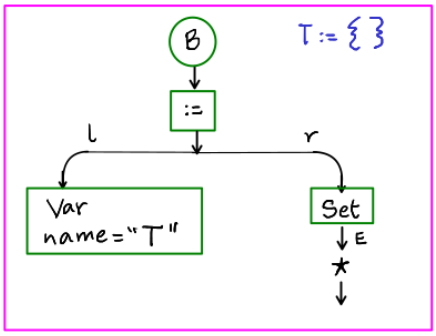
2.1 Abstract Syntax

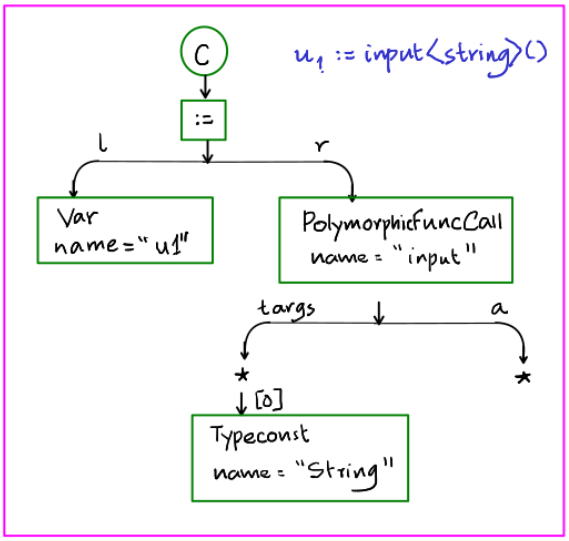
2.2 Example Abstract Test Case – Signup(OK)→Login(OK)

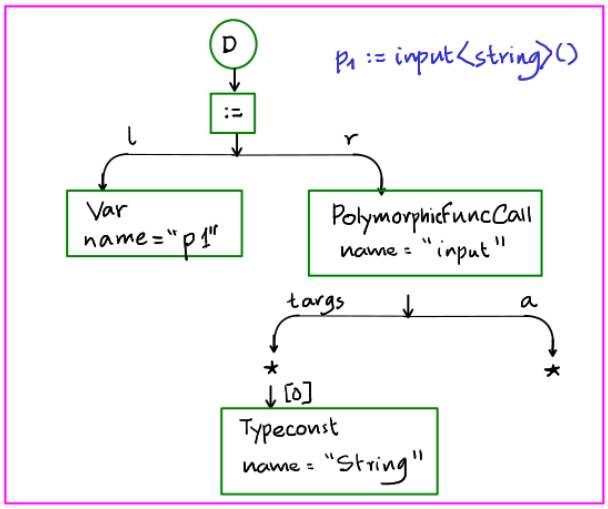
2.2.1 Abstract Syntax Tree

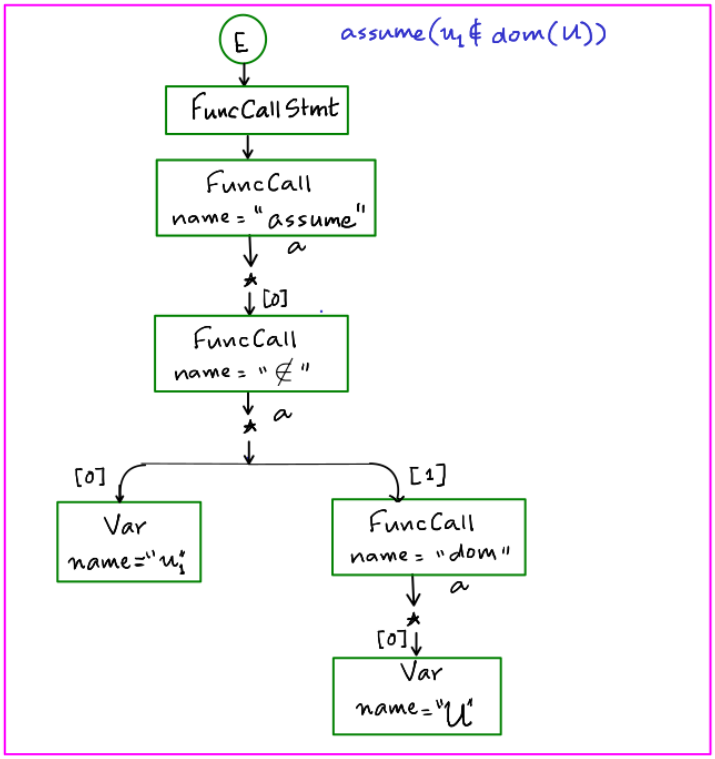


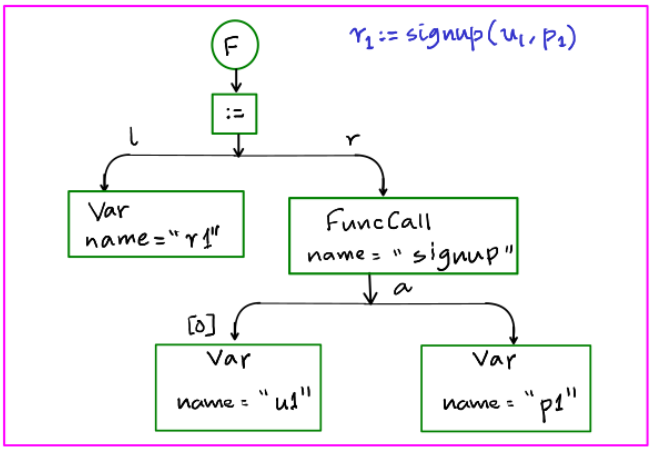


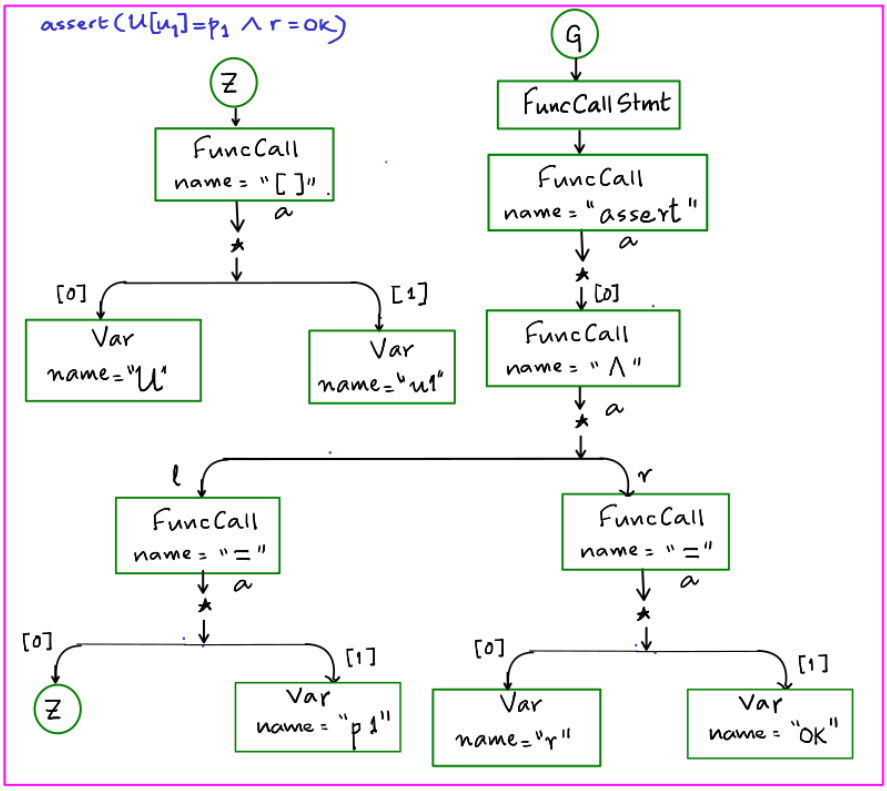


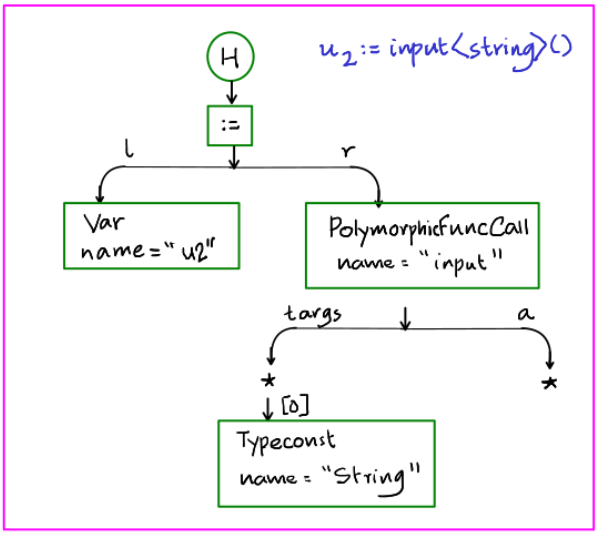


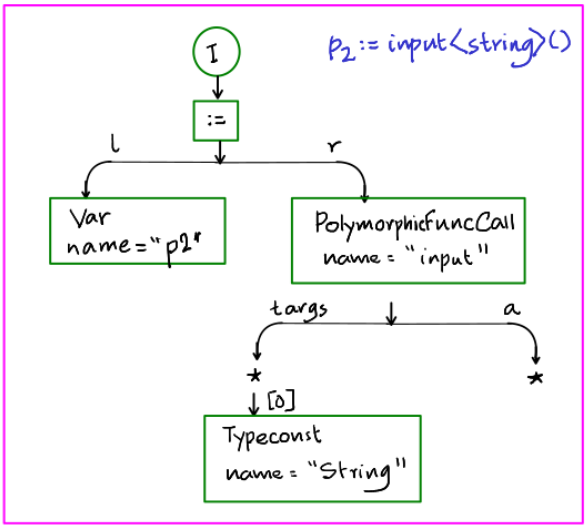

D
p_1 := input<string>()
:=
l
r
Var
name = "p1"
PolymorphicFuncCall
name = "input"
targs
a
[0]
Typeconst
name = "String"

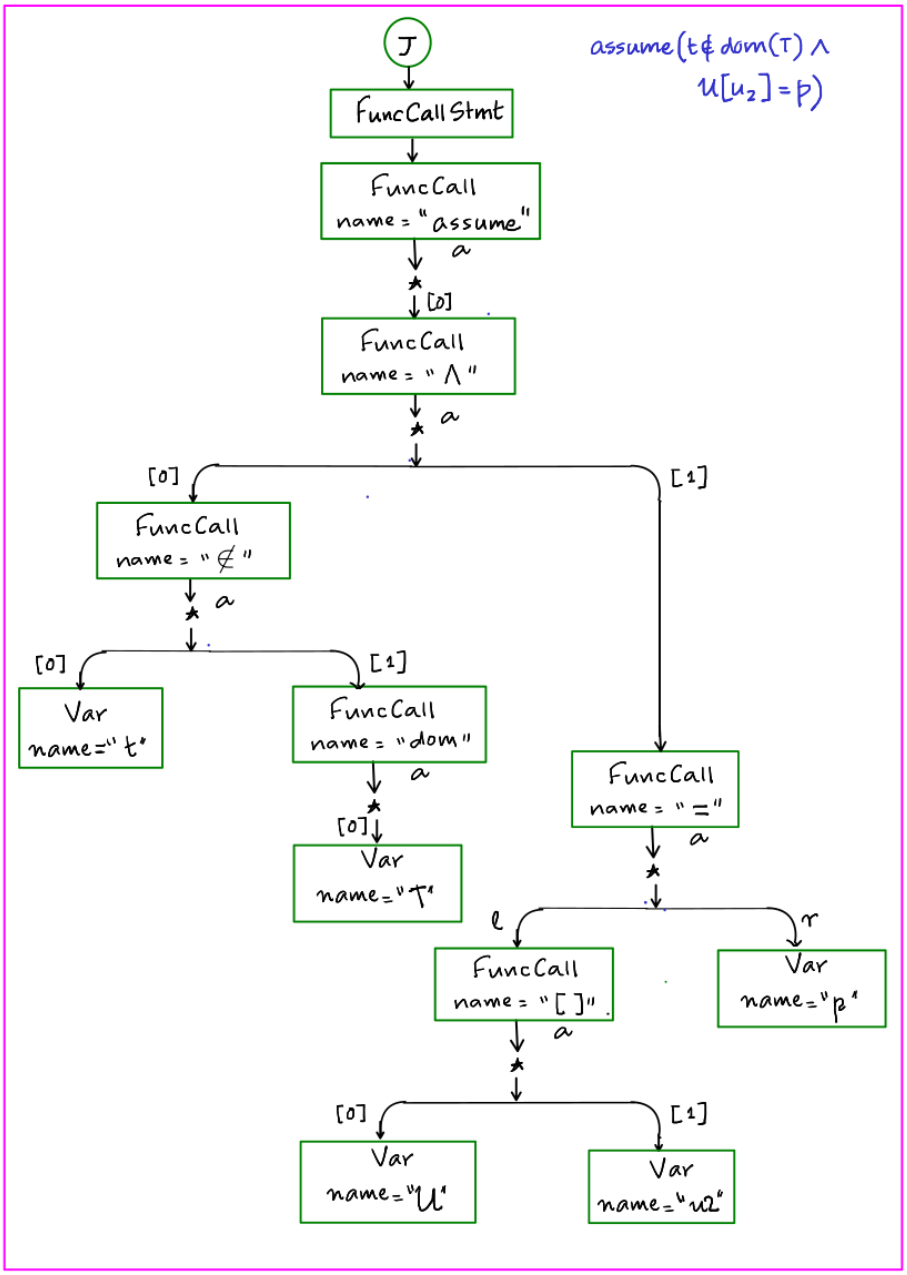

E
assume($u_1 \notin dom(U)$)
FuncCallStmt
FuncCall
name = "assume"
a
[0]
FuncCall
name = "∉"
a
[0]
[1]
Var
name = "u_1"
FuncCall
name = "dom"
a
[0]
Var
name = "U"

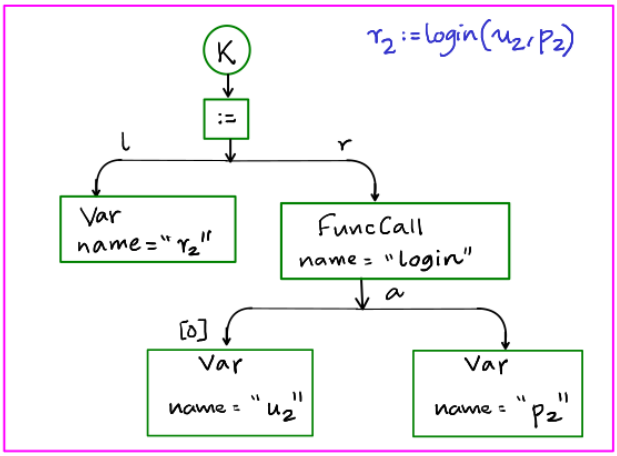

$r_1 := signup(u_1, p_1)$
F
:=
l
r
Var
name = "r1"
FuncCall
name = "signup"
a
[0]
Var
name = "u1"
Var
name = "p1"

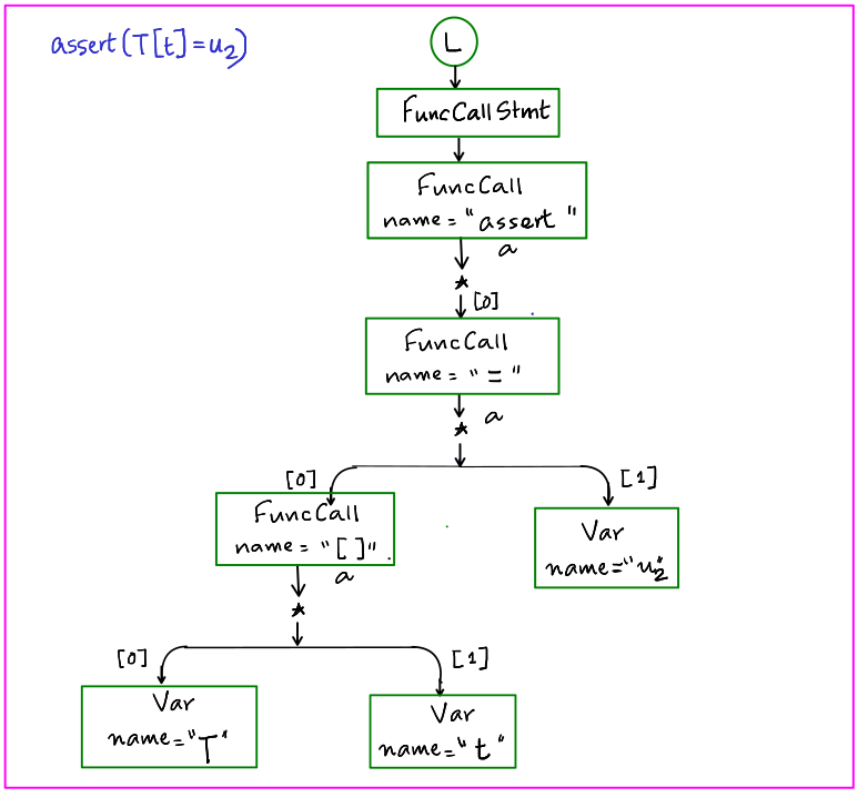

assert(U[u_1]=p_1 ∧ r = OK)
Z
FuncCall
name = "[ ]"
a
[0]
Var
name = "U"
[1]
Var
name = "u1"
G
FuncCall Stmt
FuncCall
name = "assert"
a
[0]
FuncCall
name = "∧"
a
l
r
FuncCall
name = "="
a
[0]
Z
[1]
Var
name = "p1"
FuncCall
name = "="
a
[0]
Var
name = "r"
[1]
Var
name = "OK"


H
u2 := input<string>()
:=
l
r
Var
name = "u2"
PolymorphicFuncCall
name = "input"
targs
a
*
*
[0]
Typeconst
name = "String"


I
p2 := input<string>()
:=
l
r
Var
name = "p2"
PolymorphicFuncCall
name = "input"
targs
a
*
*
[0]
Typeconst
name = "String"


J
assume(t ∉ dom(T) ∧ U[u2] = p)
FuncCall Stmt
FuncCall
name = "assume"
a
[0]
FuncCall
name = "∧"
a
[0]
[1]
FuncCall
name = "∉"
a
[0]
[1]
Var
name="t"
FuncCall
name = "dom"
a
[0]
Var
name="T"
FuncCall
name = "="
a
l
r
FuncCall
name = "[]"
a
Var
name="p"
[0]
[1]
Var
name="U"
Var
name="u2"


K
:=
l
r
Var
name = "r2"
FuncCall
name = "login"
a
[0]
Var
name = "u2"
Var
name = "p2"


assert(T[t] = u2)
L
FuncCall Stmt
FuncCall
name = "assert"
a
[0]
FuncCall
name = "="
a
[0]
[1]
FuncCall
name = "[]"
a
Var
name = "u2'"
[0]
[1]
Var
name = "T"
Var
name = "t"