Language Definitions

Sujit Kumar Chakrabarti
sujitkc@iiitb.ac.in

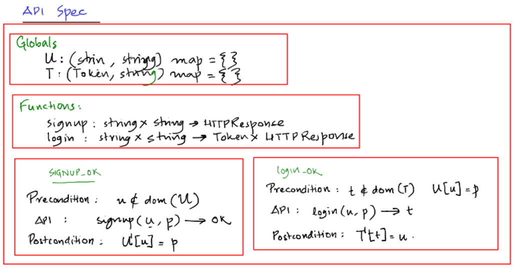
January 2025

1 API Specification Language

1.1 Abstract Syntax

1.2 Example

1.3 Example Specification – Signup-Login API



API Spec

Globals
U : (strin , string) map = { }
T : (Token, string) map = { }

Functions :
signup : string x string → HTTPResponce
login : string x string → Token x HTTPResponse

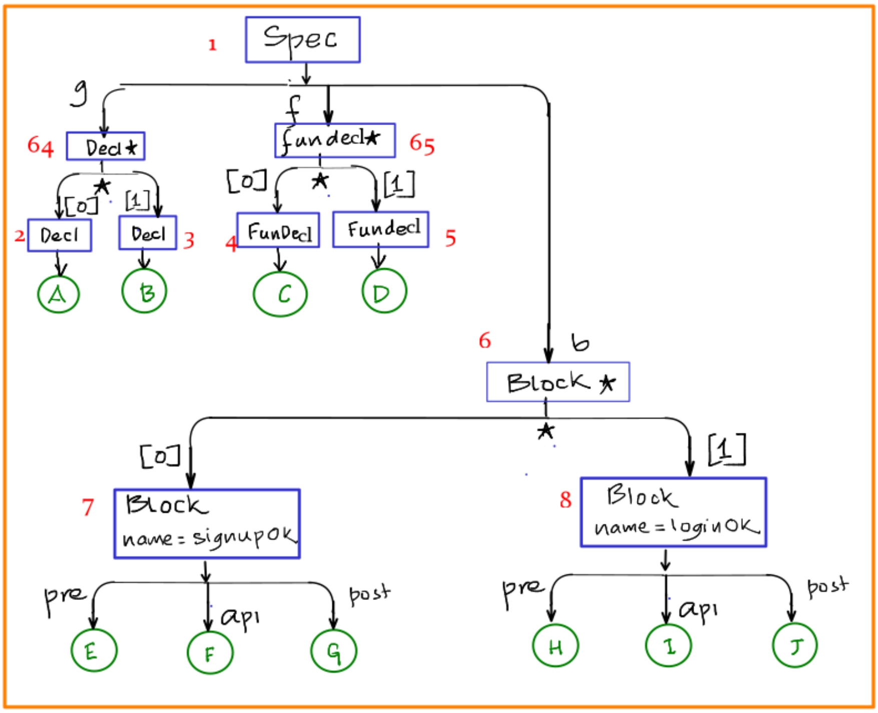
SIGNUP_OK
Precondition : $u \notin dom(U)$
API : signup(u, p) ⟶ OK
Postcondition : $U'[u] = p$

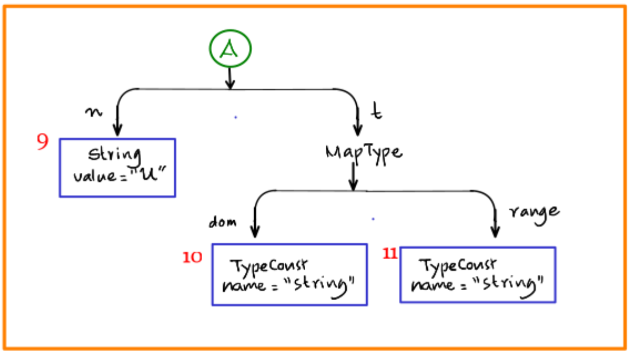
login_OK
Precondition : $t \notin dom(T)$ $U[u] = p$
API : login(u, p) ⟶ t
Postcondition : $T'[t] = u$.

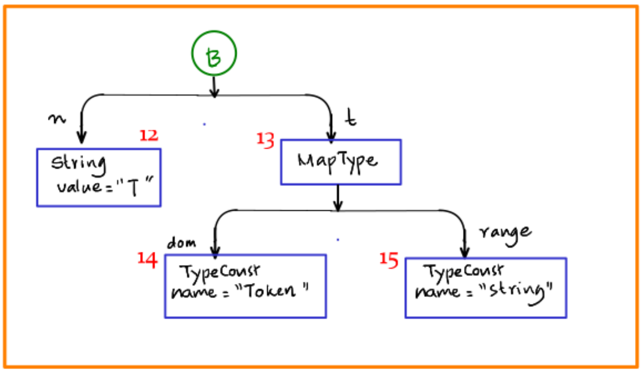
spec	→	(*g* : *Decl*⋆, *t*: *TypeDecl*⋆, *i* : *Init*, *f* : *Function*⋆, *b* : *Block*⋆)
Decl	→	(*n* : *string*, *t* : *TypeExpr*)
FunDecl	→	(*n* : *string*, *p* : *TypeExpr*, *r* : *TypeExpr*)
TypeDecl	→	*VariantDecl* | *RecordDecl*
VariantDecl	→	*VariantConstructor*+
TypeConstructor	→	(*constrname* : *String*, *TypeExpr*⋆)
RecordDecl	→	*recname* : *String*, *fields* : *Decl*+
TypeExpr	→	 *TypeConst* | *TypeVariable* | *FuncType* | *MapType* | *TupleType* | *SetType*
TypeConst	→	*String*
FuncType	→	(*args*: *TypeExpr*⋆, *ret*: *TypeExpr*)
MapType	→	(*key*: *TypeExpr*, *val*: *TypeExpr*)
TupleType	→	(*elements*: *TypeExpr*+)
SetType	→	(*elttype* : *TypeExpr*)
Init	→	(*v* : *String*, *e* : *Expr*)
	Functions	
FunctionDecl	→	(*fname* : String, *pars* : *Decl*∗, *ret* : *HTTPResponseCode*, *resp* : *TypeExpr*)
	APIs	
API	→	(*pre* : *Expr*, *call*: *FuncCall*, *resp* : (*ret* : *HTTPResponseCode*, *resp* : *post*: *Expr*)

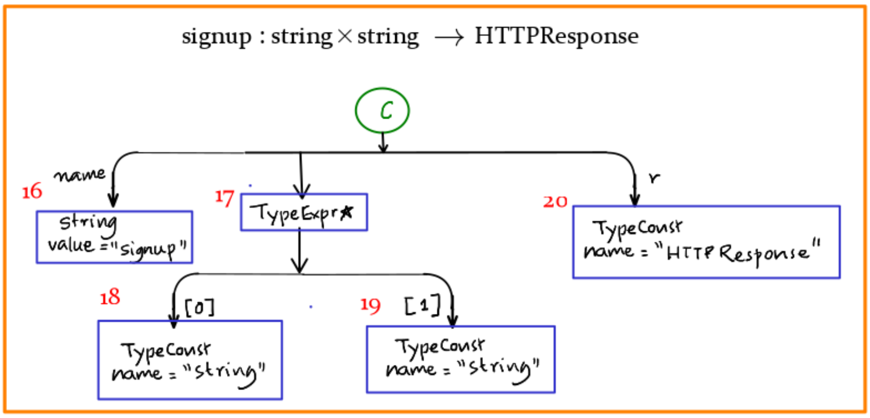
Figure 1: Abstract Syntax: (a) API Specification Language; (b) Abstract Test Case Language; (c) Common Parts

1.3.1 Abstract Syntax Tree

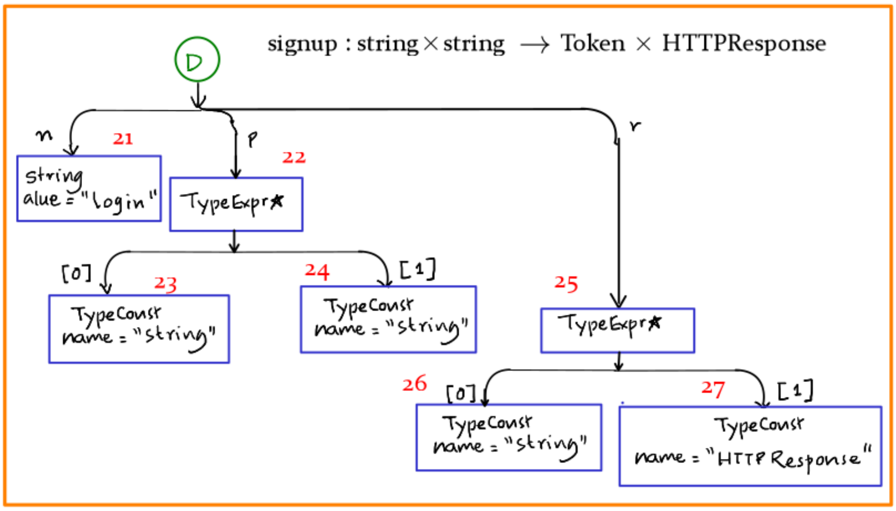



A
n
t
9
String
value="U"
MapType
dom
range
10
TypeConst
name = "string"
11
TypeConst
name = "string"

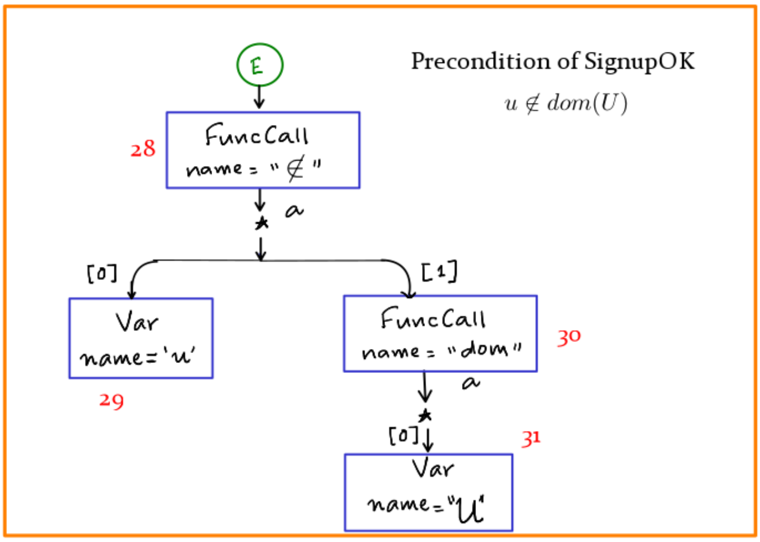

B
n
t
12
String
value="T"
13
MapType
dom
range
14
TypeConst
name = "Token"
15
TypeConst
name = "string"



signup : string × string ⟶ HTTPResponse

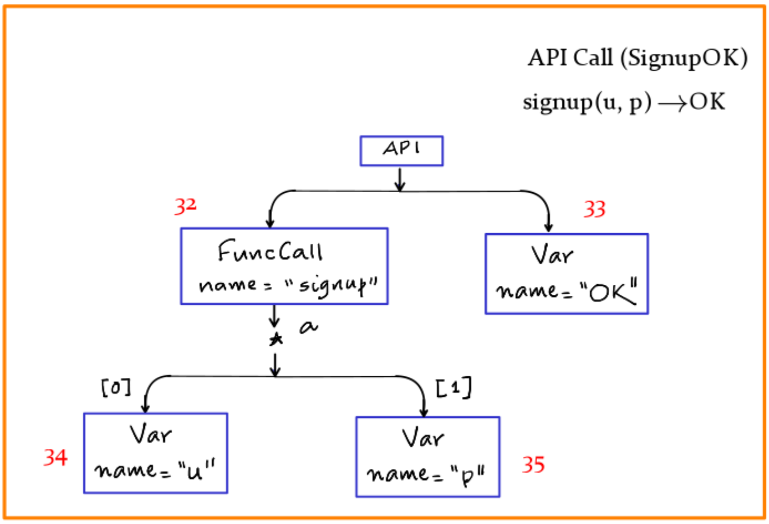


signup : string × string ⟶ Token × HTTPResponse



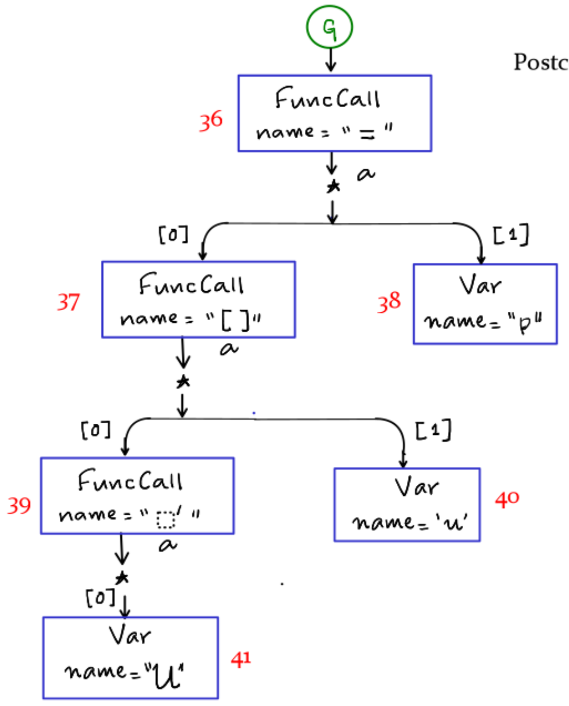
Precondition of SignupOK

$u \notin dom(U)$



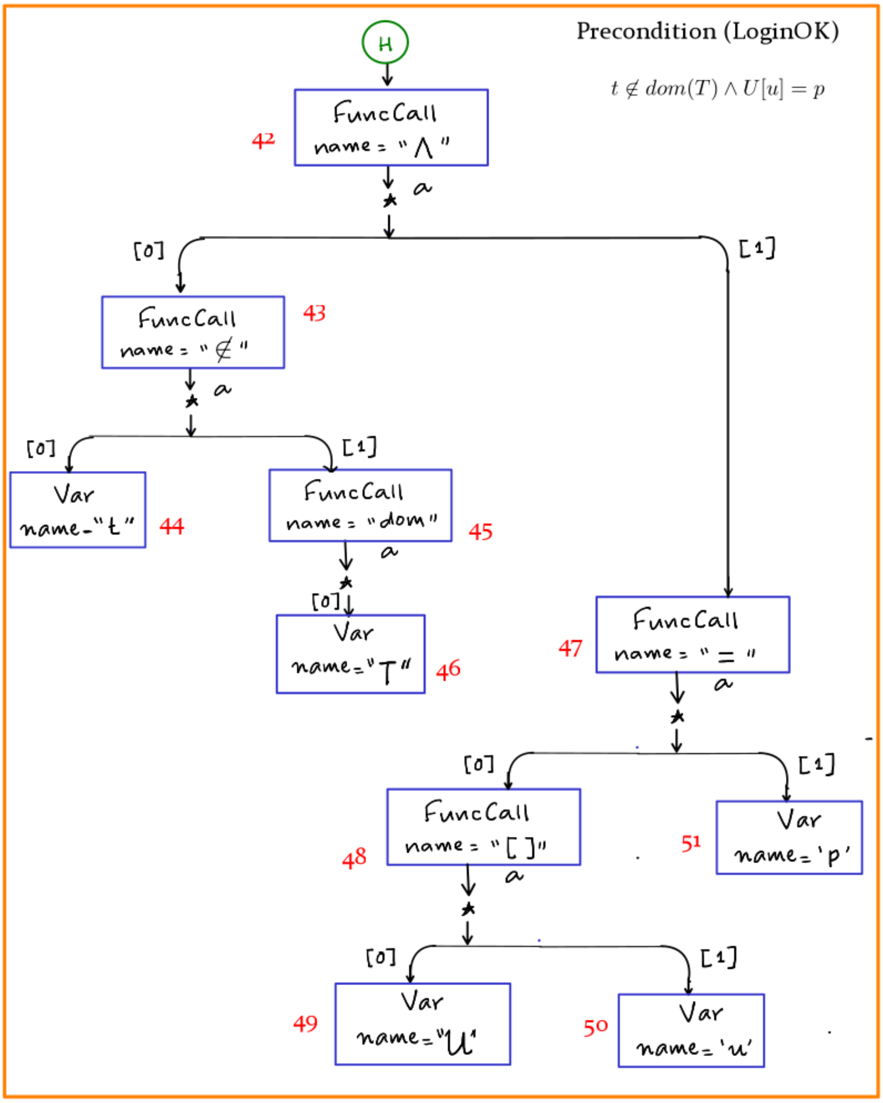
API Call (SignupOK)

signup(u, p) ⟶OK

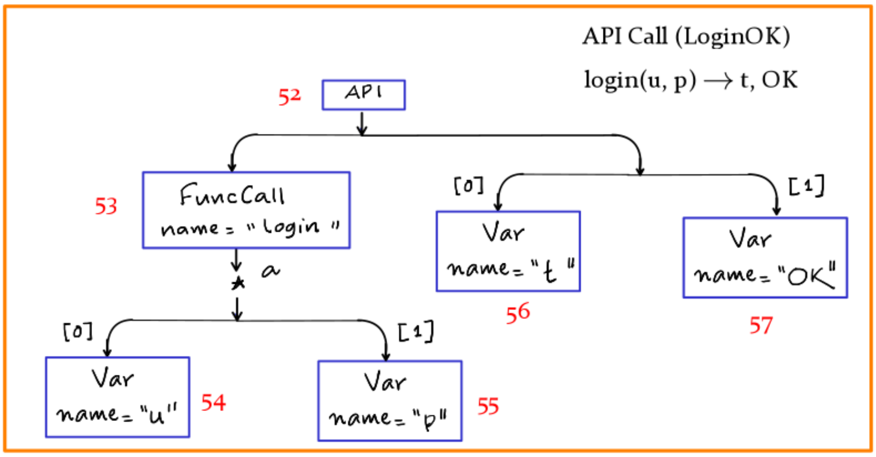


Postcondition of SignupOK

$U'[u] = p$

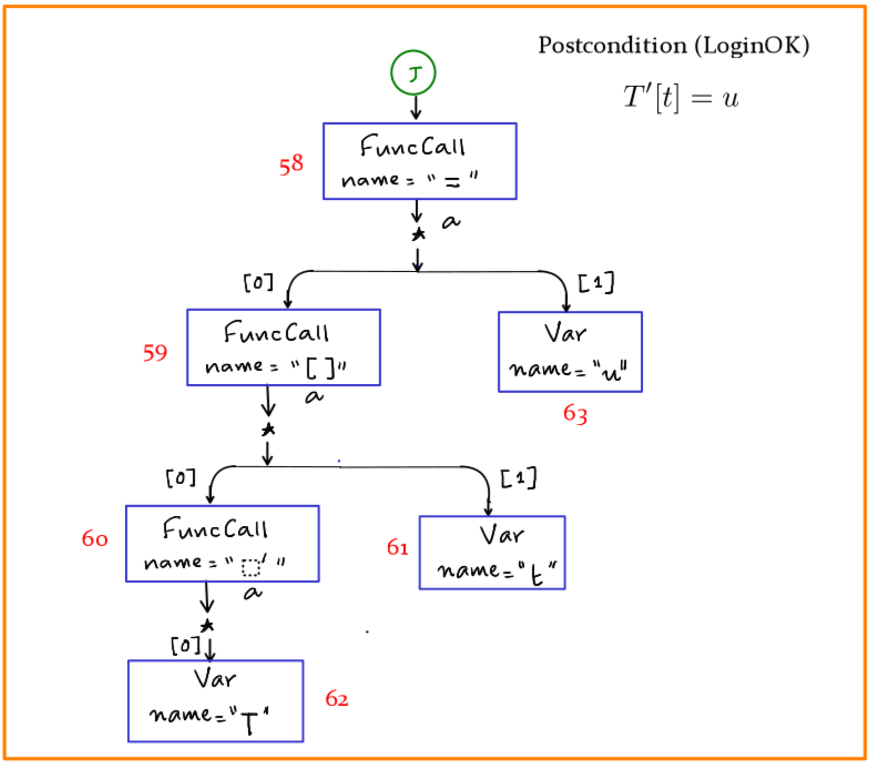

Precondition (LoginOK)

$t \notin dom(T) \wedge U[u] = p$



API Call (LoginOK)

login(u, p) $\longrightarrow$ t, OK



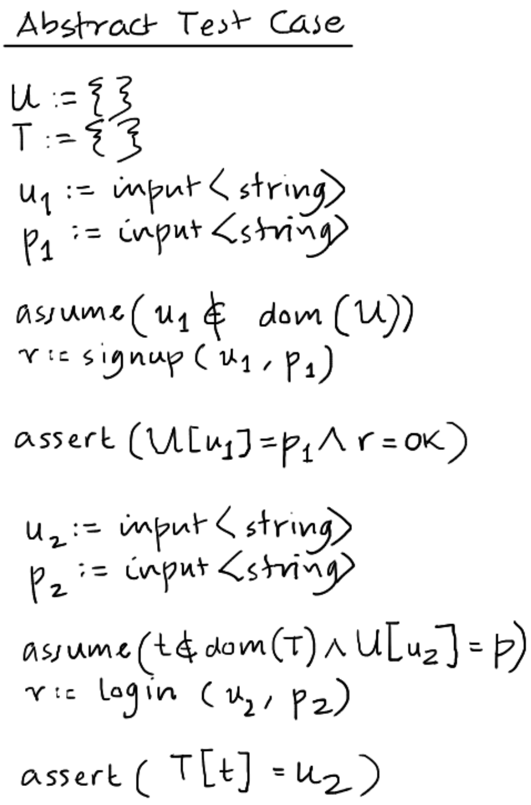
Postcondition (LoginOK)

$T'[t] = u$

2 Abstract Test Case

2.1 Example

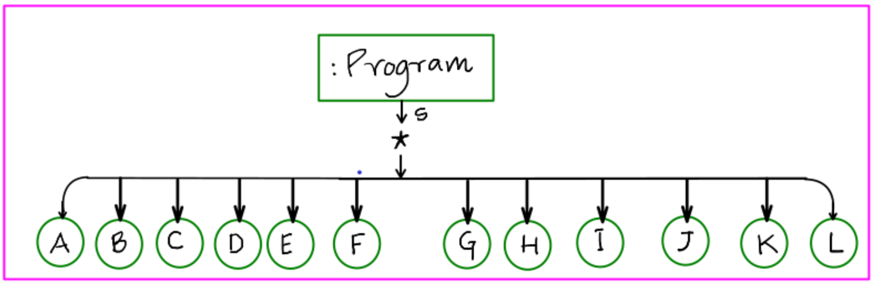
2.2 Example Abstract Test Case – Signup(OK)→Login(OK)

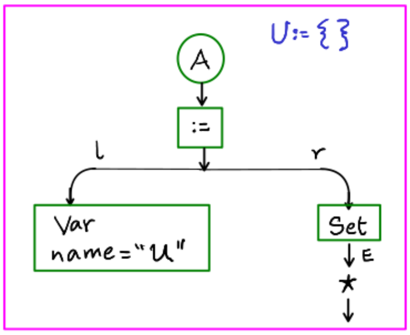


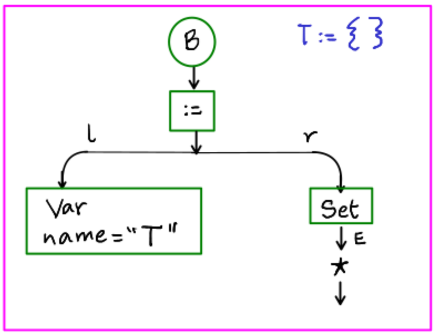
Abstract Test Case

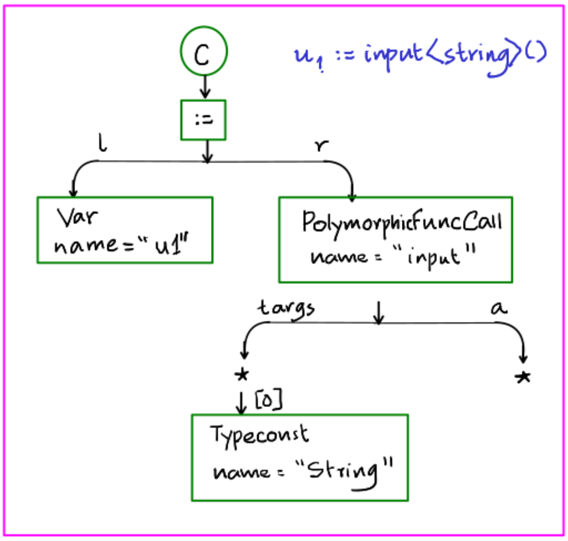
```
$\mathcal{U} := \{\}$
$T := \{\}$
$u_1 := input\langle string\rangle$
$p_1 := input\langle string\rangle$

$assume(u_1 \notin dom(\mathcal{U}))$
$r := signup(u_1, p_1)$

$assert(\mathcal{U}[u_1] = p_1 \wedge r = OK)$

$u_2 := input\langle string\rangle$
$p_2 := input\langle string\rangle$

$assume(t \notin dom(T) \wedge \mathcal{U}[u_2] = p)$
$r := login(u_2, p_2)$

$assert(T[t] = u_2)$
```

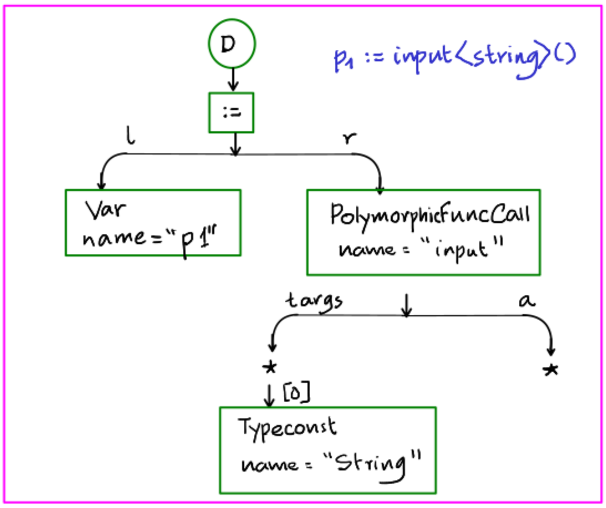
2.2.1 Abstract Syntax Tree

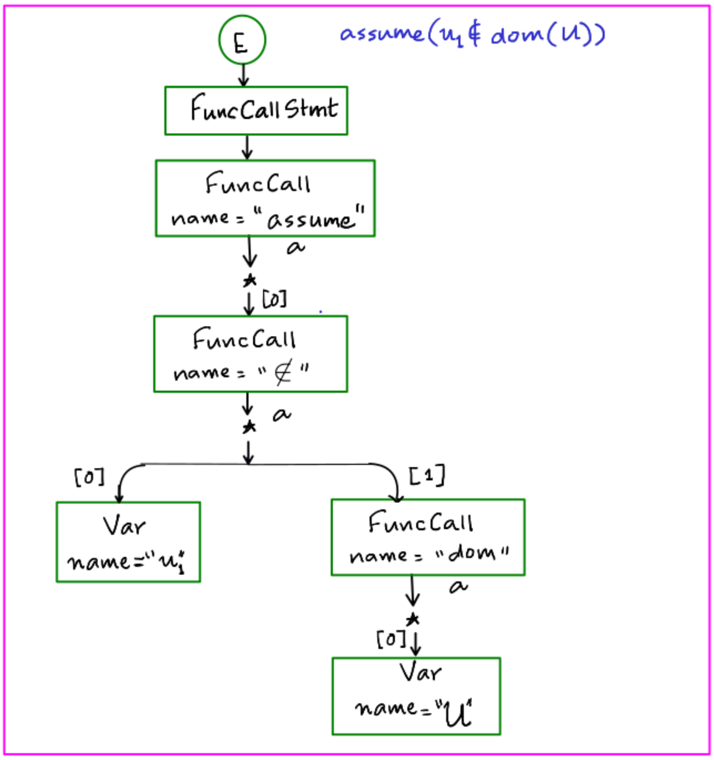


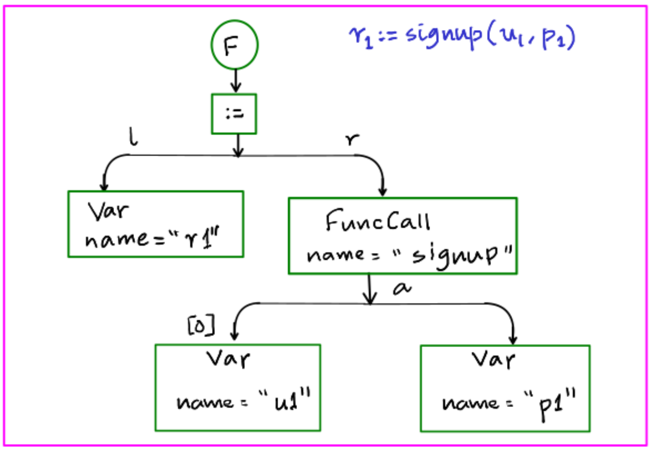


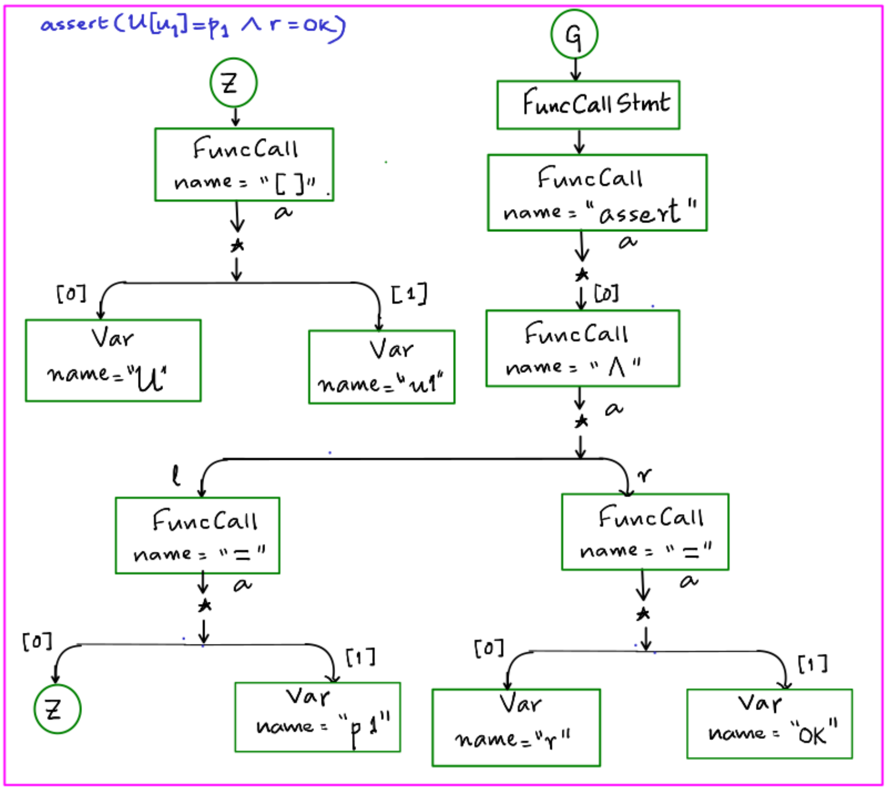


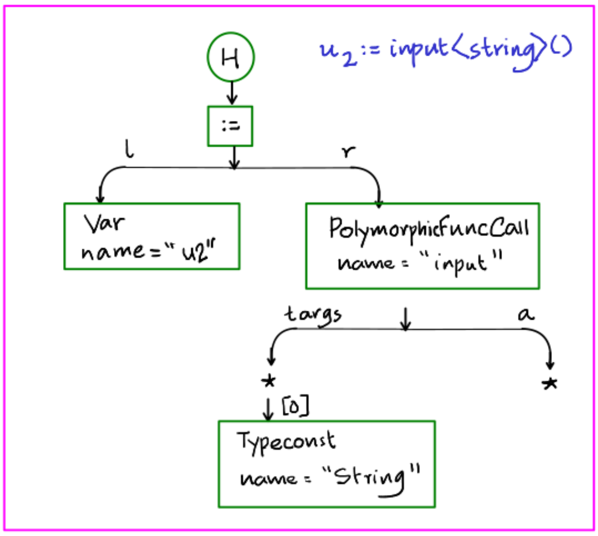


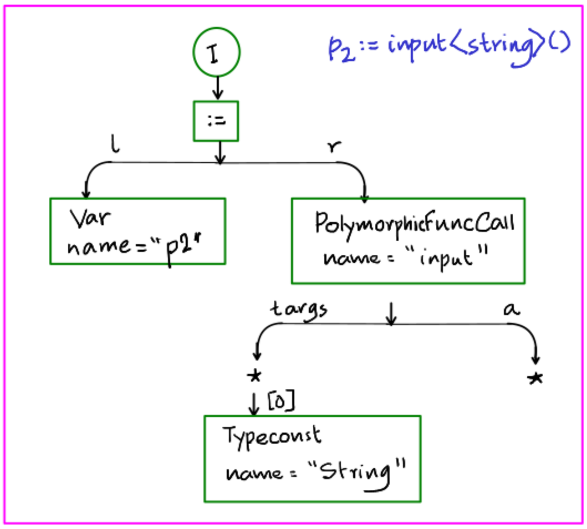

D
p1 := input<string>()
:=
l
r
Var
name="p1"
PolymorphicFuncCall
name = "input"
targs
a
*
[0]
*
Typeconst
name = "String"

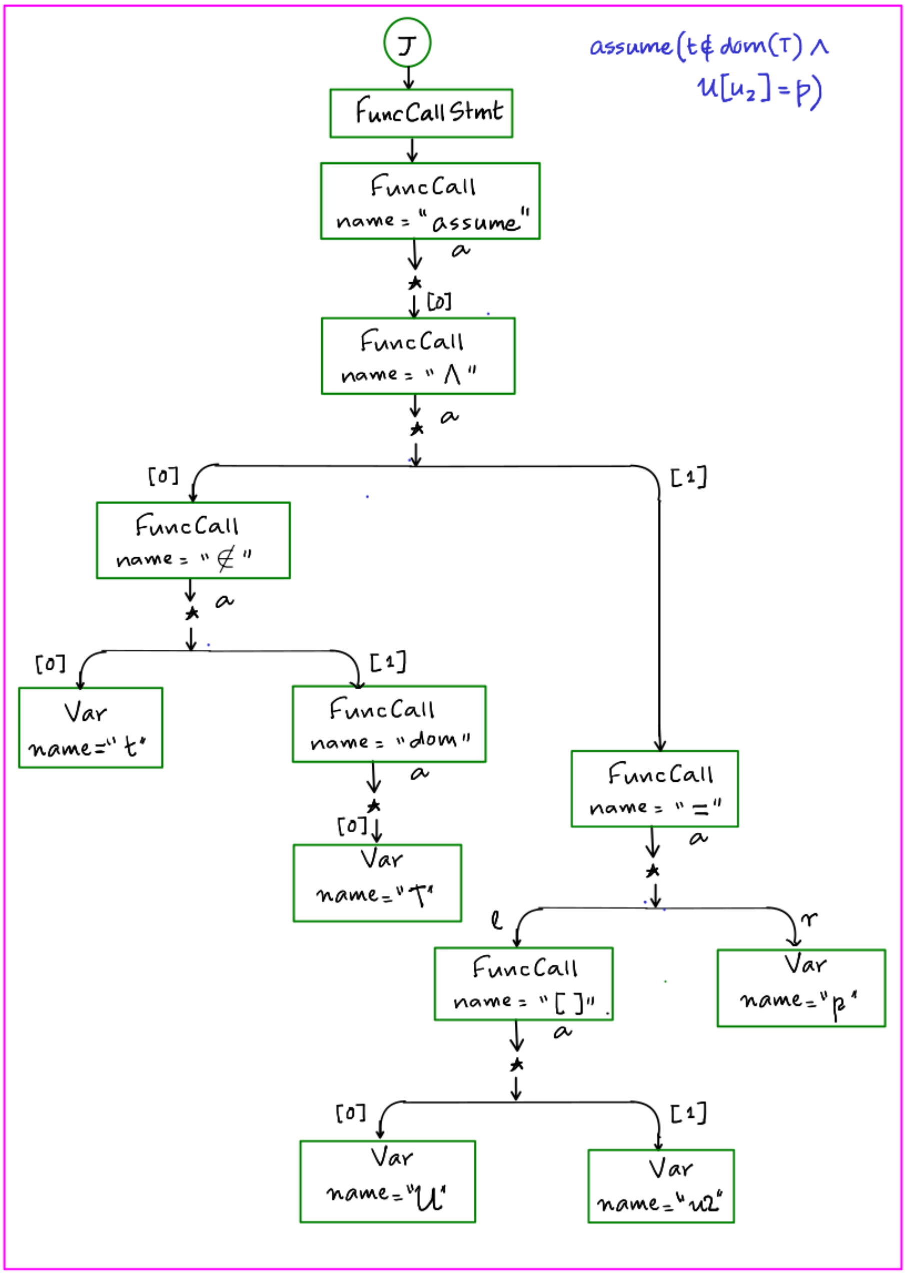

E
assume(u1 ∉ dom(U))
FuncCallStmt
FuncCall
name = "assume"
a
*
[0]
FuncCall
name = "∉"
a
*
[0]
[1]
Var
name="u1"
FuncCall
name = "dom"
a
*
[0]
Var
name="U"

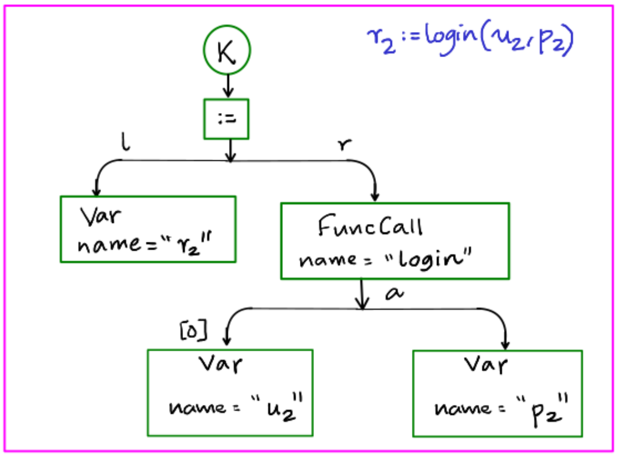

r1 := signup(u1, p1)
F
:=
l
r
Var
name="r1"
FuncCall
name = "signup"
a
[0]
Var
name = "u1"
Var
name = "p1"

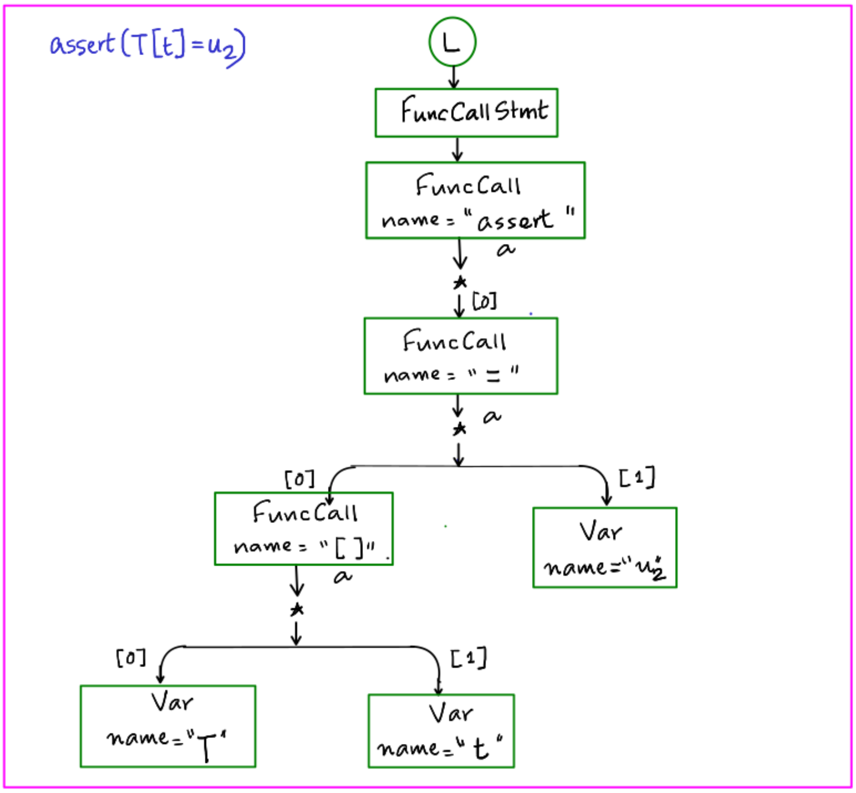

assert(U[u1]=p1 ∧ r = OK)
Z
FuncCall
name = "[ ]"
a
*
[0]
[1]
Var
name="U"
Var
name="u1"
G
FuncCallStmt
FuncCall
name = "assert"
a
*
[0]
FuncCall
name = "∧"
a
*
l
r
FuncCall
name = "="
a
*
[0]
[1]
Z
Var
name = "p1"
FuncCall
name = "="
a
*
[0]
[1]
Var
name="r"
Var
name = "OK"


u_2 := input<string>()
H
:=
l
r
Var
name = "u2"
PolymorphicFuncCall
name = "input"
targs
a
*
*
[0]
Typeconst
name = "String"


p_2 := input<string>()
I
:=
l
r
Var
name = "p2"
PolymorphicFuncCall
name = "input"
targs
a
*
*
[0]
Typeconst
name = "String"


J
assume(t∉dom(T) ∧
U[u2]=p)
FuncCallStmt
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
*
[0]
[1]
Var
name="U"
Var
name="u2"
Var
name="p"


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


assert(T[t] = u₂)
L
FuncCallStmt
FuncCall
name = "assert"
a
*
[0]
FuncCall
name = "="
a
*
[0]
[1]
FuncCall
name = "[ ]"
Var
name = "u₂"
a
*
[0]
[1]
Var
name = "T"
Var
name = "t"