Designing a Sukoshi python compiler

Project submitted to the
SRM University - AP, Andhra Pradesh
for the partial fulfillment of the requirements to award the degree of

Bachelor of Technology
in
Computer Science and Engineering
School of Engineering and Sciences

Submitted by
Seera Bhaskar Rao (AP20110010614)
Harsha Vardhan Reddy Gurrala (AP20110010588)
Lakshmana Teja Kapuganti (AP20110010633)
Brahmendra Raavi (AP20110010629)
Nehal Sampath Kumar (AP20110010618)
Venu gopal Kalluri (AP20110010573)
Srikanth Moparthi (AP20110010617)
Sriram Singamaneni (AP20110010630)
Sridhar GVS (AP20110010615)

SRM University-AP
Neerukonda,Mangalagiri,Guntur
Andhra Pradesh - 522 240

December, 2022

1) Overview of Source and target language

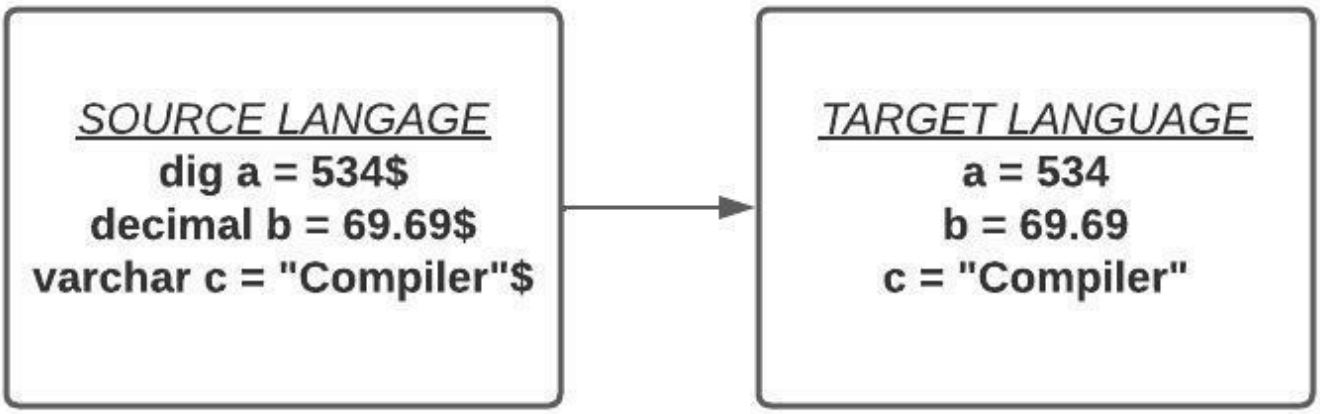
a) Description of source language

In this project, the design of a compiler is being implemented where the self-defined source language, known as sukoshi Python, is used. This is a versioned language of Python where the syntax for data types, conditional and loop statements were implemented. Sukoshi-Python is not a realistic language for use in production. It is large enough to allow for a core compiler project, though, which demonstrates every stage of compilation. Additionally, it serves as a foundation for several extensions. This source language was created entirely from the grammar that we use in the real world; it has many parallels to Python and several of our own languages, such C and C++.

b) Description of target language

The target language for our source language is python language. Python is a vastly packed language where all the real world simulations like machine-learning, deep-learning, etc. can be implemented. The main reason we chose this language is that it is simple to understand as well as simple to implement.

Example:



2) Conceptualization and Design of source language:

a) Data types available in the source language

- dig (both positive and negative numbers eg:- 1,1000,-2,0,-586 e.t.c)
- decimal (all decimal integers having any number of places after a point. eg:- 5.00, 2.8528, -18.5582, 0 e.t.c.)
- alpha (all single characters . Eg:- a, A, b, Z, e, P, e.t.c.)
- varchar (group of characters including spaces within the quotes. Eg:- "word", "pen", "A book is a good friend." e.t.c.)
- flag (it is a boolean type either true(1) or false(0) .)

b) Syntax of variable declaration if your language needs pre declaration before its use. Otherwise, the assumptions.

Syntax for variable declaration:-

dig a=-52$

alpha c="t"$ (here "$" is the terminator for each statement)

Variables should start with lower or upper case letters except with numbers and special characters.

c) Syntax of decision-making statement

***Format*:- (indentation should be followed).**

1. ***if(open bracket)(condition)(closed bracket)***

   then(space)(statement)

2. ***if(open bracket)(condition)(closed bracket)***

   then(space)(statement)

   otherwise(comma) statement

3. ***if(open bracket)(condition)(closed bracket)***

then(space)(statement)

otherif(open bracket)(condition)(closed bracket)

then(space)(statement)

otherwise(comma) statement

Example :

```
dig a = 2$
      if[a==2]
        then show(a|" is shown.")$
       otherwise,
        show(a|" is not shown")$
```

d) Syntax of iterative statements

From Loop Statement :-

Loop statement such as from loop statements.

from(open bracket)(start ,range , increment)(closed bracket)

statement(s)$

Example :

```
dig i=0$
from[i=0$i<=3$i++]
      show(i|" is the number.")$
```

Upto Loop Statement :-

Loop statement such as upto loop.

upto(open bracket)(condition)(closed bracket)

statement(s)$

Example :

```
dig i=2$
upto[i<5]
    show(i|" is the value.")$
    i++$
```

e) CFG for at least five constructs in your language

CFG for Variable Declaration

•Declaration —> <type><var-list>$

•<type> —> dig|decimal|alpha|varchar |flag

•<var-list> —> IdA

•A —> , <var-list>|#

CFG for Variable Initialization

•Declaration —> <type><Id>D'$

•D' —> =<init>

•<type> —> dig | decimal | varchar | flag | alpha | #

•<init> —> BA

•B —> - | #

•A —> Number | Decimals | Strings | Bool

•Bool —> 0 | 1

•Number —> num

•Decimals —> Num. Num

•String —> id

CFG for Arithmetic/Relational Expression

•Expression —> <type><Id>D'$

•<type> —> dig | decimal | #

•D' —> =Limit | #

•Limit —> <Id> C | #

•C —> A Limit | #

•A —> + | - | * | / | ++ | = | - - | % |==| != | < | > | <= |>=

•<Id> → <identifier> | <number>

CFG for IF Statement

- Condition —> Statement.Alt
- Statement —> if.[.Expression.].then .Y
- Alt —> otherwise, Y | other.Statement.Alt | #
- Expression —> <type><Id>D'
- <type> —> dig | decimal | #
- D' —> BLimit | #
- Limit —> <Id> D'| #

- <Id> —> id
- Var —> id
- Y —> show.(.L.).$ | Expression
- L —> “identifier” | varK | #
- K —> |.L | L
- B —> + | - | * | / | ++ | = | - - | % |==| != | < | > | <= |>=

CFG for From Loop

- S —> f.r.o.m.[Declaration $ Exp $ Exp]Y
- Exp —>id C
- C —> F E
- E —> id | #
- F —> ++ | - -| < | > | <= |>=
- Declaration —> <type>idD’
- D’ —> =id
- <type> —> dig | decimal | #
- Y —> show.(.L.).$ | Expression
- L —> “id” | idK | #
- K —> |.L | L
- Expression —> <type>idG$
- G —> =Limit | #
- Limit —> id H
- H —> A Limit | #
- A —> + | - | * | / | ++ | = | - - | % |==| != | < | > | <= | >=

CFG for Upto Loop

- S —> u.p.t.o.[.X.].Y. Exp.
- X —> id | Exp
- Exp —>id C |#
- C —> AE
- E —> id | #

- Y —> show.(.L.).$ | Expression
- L —> “id” | idK | #
- K —> |.L | L
- <type> —> dig | decimal | #
- Expression —> <type>idG$
- G —> =Limit | #
- Limit —> id H
- H —> A Limit | #
- A —> + | - | * | / | ++ | = | - - | % |==| != | < | > | <= | >=

f) Design of Parser

We designed with LL(1) parser:

For variable declaration

D → TV$

T→ dig

T→decimal

T→alpha

T→varchar

T→flag

V→idA

A→,V|#

First:

FIRST(D)= { dig, decimal, alpha, varchar, flag }

FIRST(T)= { dig, decimal, alpha, varchar, flag }

FIRST(V)= { id }

FIRST(A)= { , # }

Follow: -

follow(D) = {~}

follow(T) = {id}

follow(V) = {$}

follow(A) = {$}

	~	$	dig	decimal	alpha	varchar	flag	,	id	#
D			D → TV$	D → TV$	D → TV$	D → TV$	D → TV$			
T			T→ dig	T→decimal	T → alpha	T→varchar	T→flag			
V									V→idA	
A		A→#						A→,V		

```
FIRST OF D: b
FIRST OF T: b
FIRST OF V: d
FIRST OF A: , ^

FOLLOW OF D: ~
FOLLOW OF T: d
FOLLOW OF V: $
FOLLOW OF A: $

FIRST OF D->TV$: b
FIRST OF T->b: b
FIRST OF V->dA: d
FIRST OF A->,V: ,
FIRST OF A->^: ^


        ***************** LL(1) PARSING TABLE ********************
        ------------------------------------------------------------
          $          ,          b          d          ~
D                                D->TV$
T                                T->b
V                                          V->dA
A         A->^       A->,V       |
```

Valid input = decimal id,id,id $

Stack	Input	Action
~D	decimal id,id,id $ ~	D → TV$
~$VT	decimal id,id,id $ ~	T →decimal

~$Vdecimal	decimal id,id,id $ ~	Match decimal
~$V	id,id,id $~	V → id A
~$A id	id,id,id $~	Match id
~$A	,id,id $~	A → ,V
~$V,	,id,id $~	Match ,
~$V	id,id $~	V → id A
~$ A id	id,id $~	Match id
~$ A	,id $~	A → ,V
~$ V ,	,id $~	Match ,
~$ V	id $~	V → id A
~ $ A id	id $~	Match id
~$ A	$ ~	A → #
~$	$ ~	Match $
~	~	Succeeded

First and Follow for the Variable Initialization

•Declaration —> <type><Id>D'$
•D' —> =<init>
•<type> —> dig | decimal | varchar | flag | alpha | #
•<init> —> BA
•B —> - | + | #
•A —> Number | Decimals | Strings | Bool
•Bool —> 0 | 1
•Number —> num
•Decimals —> Num. Num
•String —> id

First :

First(D) = (dig,decimal ,varchar,flag ,alpha , #,Id)
First(D') = (=)
First(type) = (dig,decimal ,varchar,flag ,alpha , #)
First(init)=(-,+,#,num,num.num,id,0,1)
First(B)=(-,+,#)
First(A)=(num,num.num,id,0,1)
First(Bool)=(0,1)
First(Number)=(num)
First(Decimals) = (num.num)
First(String) = (id)

Follow:

Follow(D)=(**$**)
Follow(D')=()
Follow(type)=(Id)
Follow(init)=()
Follow(B)=(num,num.num,id,0,1)
Follow(A)=()
Follow(Bool)=()
Follow(Number)=()
Follow(Decimals)=()
Follow(String)=()

	~	=	id	Id	dig	decimal	varchar	flag	alpha	-	+	0	1	num	num.num
D	Declaration —> <type><Id>D'$			Declaration —> <type><Id>D'$	Declaration —> <type><Id>D'$	Declaration —> <type><Id>D'$	Declaration —> <type><Id>D'$	Declaration —> <type><Id>D'$	Declaration —> <type><Id>D'$						
D'		D' —> =<init>													
Type				<type> —> #	<type> —> dig	<type> —> decimal	<type> —> varchar	<type> —> flag	<type> —> alpha						
init			<init> —> BA							<init> —> BA	<init> —> BA	<init> —> BA	<init> —> BA	<init> —> BA	<init> —> BA
B		B —> #								B —> -	B —> +	B —> #	B —> #	B —> #	B —> #
A		A —> Strings										A —> Bool	A —> Bool	A —> Number	A —> Decimals
Bool												Bool —> 0	Bool —> 1		
Number														Number —> num	
Decimals															Decimals —> Num.Num
String		String —> id													

```
FIRST OF I: b d
FIRST OF D: =
FIRST OF T: ^ b

FOLLOW OF I: ~
FOLLOW OF D: $
FOLLOW OF T: d

FIRST OF I->TdD$: b d
FIRST OF D->=d: =
FIRST OF T->b: b
FIRST OF T->^: ^


        *************** LL(1) PARSING TABLE ******************
        ------------------------------------------------------
          $         =         b         d         ~
I                             I->TdD$   I->TdD$
D                 D->=d
T                             T->b      T->^
```

Valid Input: dig i = 0

Stack	**Input**	**Action**
~D	Dig Id=0$~	D→tIdD'$
~$D'Idt	DigId=0$~	t→Dig
~$D'IdDig	DigId=0$~	Matched
~$D'Id	Id=0$~	Matched
~$D'	=0$~	D' → =<init>
~$<init>=	=0$~	Matched
~$<init>	0$~	<init> → BA
~$AB	0$~	B→#
~$A	0$~	A→ Bool
~$Bool	0$~	Bool→0
~$0	0$~	Matched
~$	$~	matched
~	~	Accepted

For Arithmetic/Relational Expression

E →T.id.D'.$
T →dig | decimal | #
D' →=.L. | #
L →id.C | #
C →A.L | #
A →+ | - | * | / | ++ | = | -- | % | == | != | < | > | < = | >=

First :-

FIRST(E) = {dig,decimal,id}
FIRST(T) = {dig,decimal,#}
FIRST(D') = {=,#}
FIRST(L) = {id,#}
FIRST(C) = {+, -, *, /, ++, =, --, %, ==, !=, <, >, <=, >=, #}
FIRST(A) = {+, -, *, /, ++, =, --, %, ==, !=, <, >, <=, >=}

Follow :-

FOLLOW(E) = {~}
FOLLOW(T) = {id}
FOLLOW(D') = {$}
FOLLOW(L) = {$}
FOLLOW(C) = {$}
FOLLOW(A) = {id}

	~	$	dig	decimal	id	+	-	*	/	++	=	--	%	==	!=	<	>	<=	>=
E			E→TidD'$	E→TID'$	E→TidD'$														
T			T→dig	T→decimal	T→#														
D'		D'→#									D'→=L								
L		L→#			L→idC														

C		C → #				C → AL	C→ AL	C→ AL	C → AL	C→ AL	C → AL	C→ AL	C→ AL	C→ AL	C→ AL	C→ AL	C→ AL	C→A L	C→AL
A						A →+	A→ -	A →*	A →/	A→ ++	A →=	A→- -	A → %	A→ ==	A→ !=	A→ <	A →>	A→< =	A→>=

Valid Input : dig id=id + id$

Stack	**Input**	**Action**
~E	dig id = id + id $ ~	E→T.id.D'.$
~$.D'.id.T	dig id = id + id $ ~	T→dig
~$.D'.id.dig	dig id = id + id $ ~	Match dig
~$.D'.id	id = id + id $ ~	Match id
~$.D'	= id + id $~	D' →=.L.
~$.L.=	= id + id $~	Match =
~$.L	id + id $~	L →id.C
~$.C.id	id + id $~	Match id
~$.C	+ id $~	C→A.L
~$.L.A	+ id $~	A→+
~$.L.+	+ id $~	Match +
~$.L	id $~	L→id.C
~$.C.id	id $~	Match id
~$.C	$~	C→#
~$	$~	Match $
~	~	Match ~

For if statement

First :-

FIRST(C) = {if}
FIRST(S) = {if}
FIRST(E) = {dig,decimal,#}
FIRST(T) = {dig,decimal,#}
FIRST(X) = {+, -, *, /, ++, =, --, %, ==, !=, <, >, <=, >=}
FIRST(L) = {#,id}
FIRST(I) = {id}
FIRST(V) = {id}
FIRST(Y) = dig,decimal,#,show}
FIRST(Z) = {#,"}
FIRST(K) = {#,id,|}
FIRST(B) = {+, -, *, /, ++, =, --, %, ==, !=, <, >, <=, >=}

Follow :-

FOLLOW(C) = {|}
FOLLOW(S) = {|}
FOLLOW(E) = {],|}
FOLLOW(T) = {id}
FOLLOW(X) = {],),|}
FOLLOW(L) = {],),|}
FOLLOW(I) = {+, -, *, /, ++, =, --, %, ==, !=, <, >, <=, >=}
FOLLOW(V) = {#,id,"}
FOLLOW(Y) = {|}
FOLLOW(Z) = {}
FOLLOW(K) = {}
FOLLOW(B) = {#,id}

Let the grammar be:-

1. C→S
2. S→if [E] then Y
3. E→T I X
4. T→dig
5. T→decimal
6. T→#
7. X→B L
8. X→#
9. L→I X
10. L→#
11. I→id
12. V→id
13. Y→show (L) $
14. Y→E
15. Z→" id "
16. Z→V K
17. K→| L
18. K→L
19. B→+
20. B→-
21. B→*
22. B→/
23. B→=
24. B→++
25. B→--
26. B→%
27. B→==
28. B→!=
29. B→<
30. B→>
31. B→<=
32. B→>=

	if	[	]	then	dig	decimal	#	id	show	(	)	$	“	|	+	-	*	/	=	++	--	%	==	!=	<	>	<=	>=	|
C	1																												
S	2																												
E					3	3	3																						
T					4	5	6																						
X							8								7	7	7	7	7	7	7	7	7	7	7	7	7	7	
L							9	10																					
I								11																					
V								12																					
Y					14	14	14		13																				
Z								16					15																
K							18	18						17															
B															19	20	21	22	23	24	25	26	27	28	29	30	31	32	

For FROM Loop

FIRST :-

FIRST(S) = {from}
FIRST(Exp) = {id}
FIRST(C) = {+, -, *, /, ++, =, --, %, ==, !=, <, >, <=, >=}
FIRST(E) = {id, #}
FIRST(F) = {+, -, *, /, ++, =, --, %, ==, !=, <, >, <=, >=}
FIRST(Dec) = {dig, decimal, id}
FIRST(D') = {=}
FIRST(T) = {dig, decimal, #}
FIRST(Y) = {show, dig, decimal, id}
FIRST(L) = {", id, #}
FIRST(K) = {|}
FIRST(Expression) = {dig, decimal, id}
FIRST(G) = {=,#}
FIRST(Limit) = {id}
FIRST(H) = {+, -, *, /, ++, =, --, %, ==, !=, <, >, <=, >=, #}
FIRST(A) = {+, -, *, /, ++, =, --, %, ==, !=, <, >, <=, >=, #}

FOLLOW :-
FOLLOW(S) = {~}
FOLLOW(Exp) = {$,]}
FOLLOW(C) = {$,]}
FOLLOW(E) = {$,]}
FOLLOW(F) = {id, $,]}
FOLLOW(Dec) = {$}
FOLLOW(D') = {$}
FOLLOW(T) = {id}
FOLLOW(Y) = {~}
FOLLOW(L) = {)}
FOLLOW(K) = {)}
FOLLOW(Expression) = {~}

FOLLOW(G) = {$}
FOLLOW(Limit) = {$}
FOLLOW(H) = {$}
FOLLOW(A) = {id}

CFG

1. S —> f.r.o.m.[Declaration $ Exp $ Exp]Y
2. Exp —>id C
3. C —> F E
4. E —> id
5. E —> #
6. F —> ++
7. F —> --
8. F —> <
9. F —> >
10. F —> <=
11. F —> >=
12. Declaration —> TidD’
13. D’ —> =id
14. T —> dig
15. T —> decimal
16. T —> #
17. Y —> show.(.L.).$
18. Y —> Expression
19. L —> “id”
20. L —> idK
21. L —> #
22. K —> |.L
23. K —> L
24. Expression —> TidG$
25. G —> =Limit
26. G —> #
27. Limit —> id H
28. H —> A Limit

29. H —> #
30. A —> +
31. A —> -
32. A —> *
33. A —> /
34. A —> ++
35. A —> =
36. A —> --
37. A —> %
38. A —> ==
39. A —> !=
40. A —> <
41. A —> >
42. A —> <=
43. A —> >=

	~	from	[	$	]	id	dig	decimal	show	(	)	“	|	+	-	*	/	%	=	==	++	--	!=	<	>	<=	>=
S		**1**																									
Exp						**2**																					
C																					**3**	**3**		**3**	**3**	**3**	**3**
E				**5**	**5**	**4**																					
F																					**6**	**7**		**8**	**9**	**10**	**11**
Dec						**12**	**12**	**12**																			
D’																			**13**								
T						**16**	**14**	**15**																			
Y						**18**	**18**	**18**	**17**																		

L						20					21	19															
K						23						23	22														
Expression							24	24																			
G				26															25								
Limit						27																					
H				29										28	28	28	28	28	28	28	28	28	28	28	28	28	28
A														30	31	32	33	34	35	36	37	38	39	40	41	42	43

For Upto Loop

First :-

FIRST(S) = {upto}
FIRST(X) = {id, #}
FIRST(Exp) = {id, #}
FIRST(C) = {+, -, *, /, ++, =, --, %, ==, !=, <, >, <=, >=}
FIRST(A) = {+, -, *, /, ++, =, --, %, ==, !=, <, >, <=, >=}
FIRST(E) = {id,#}
FIRST(Y) = {show, dig, decimal, #}
FIRST(L) = {“, id, #}
FIRST(K) = {|, “, id, #}
FIRST(<type>) = {dig, decimal, #}

FIRST(Expression) = {dig, decimal, #}
FIRST(G) = {=,#}
FIRST(Limit) = {id}
FIRST(H) = {+, -, *, /, ++, =, --, %, ==, !=, <, >, <=, >=}

Follow :-

FOLLOW(S) = {~}
FOLLOW(X) = {[}
FOLLOW(Y) = {id, ~}
FOLLOW(Exp) = {~, [}
FOLLOW(C) = {~, [}
FOLLOW(A) = {id, ~, [}
FOLLOW(E) = {~, [}
FOLLOW(L) = {)}
FOLLOW(K) = {)}
FOLLOW(<type>) = {id}
FOLLOW(Expression) = {id, ~}
FOLLOW(G) = {$}
FOLLOW(Limit) = {$}
FOLLOW(H) = {$}

CFG for Upto Loop

1. S —> u.p.t.o.[.X.].Y. Exp.
2. X —> id
3. X —> Exp
4. Exp —> id C
5. Exp —> #
6. C —> AE
7. E —> id
8. E —> #
9. Y —> show.(.L.).$
10. Y —> Expression

11. L —> “id”
12. L —> idK
13. L —> #
14. K —> .L
15. K —> L
16. <type> —> dig
17. <type> —> decimal
18. <type> —> #
19. Expression —> <type>idG$
20. G —> =Limit
21. G —> #
22. Limit —> id H
23. H —> A Limit
24. H —> #
25. A —> +
26. A —> -
27. A —> *
28. A —> /
29. A —> ++
30. A —> =
31. A —> --
32. A —> %
33. A —> ==
34. A —> !=
35. A —> <
36. A —> >
37. A —> <=
38. A —> >=

	~	upto	[	]	id	Number	show	(	)	$	“	”	dig	decimal	.	+	-	*	/	%	=	==	++	--	!=	<	>	<=	>=
S		**1**																											
X					**2**																								
Exp	**5**		**5**																										
C																**6**	**6**	**6**	**6**	**6**	**6**	**6**	**6**	**6**	**6**	**6**	**6**	**6**	**6**
E	**8**	**8**			**7**																								
Y							[9]						**10**	**10**															
L					**12**				**13**		**11**																		
K					**15**				**15**		**15**				**14**														
<type>										**18**			**16**	**17**															
Expression										[19]			[19]	[19]															
G										[21]												[20]							
Limit					[22]																								
H										[24]						[23]	[23]	[23]	[23]	[23]	[23]	[23]	[23]	[23]	[23]	[23]	[23]	[23]	[23]
A																[25]	[26]	[27]	[28]	[29]	[30]	[31]	[32]	[33]	[34]	[35]	[36]	[37]	[38]

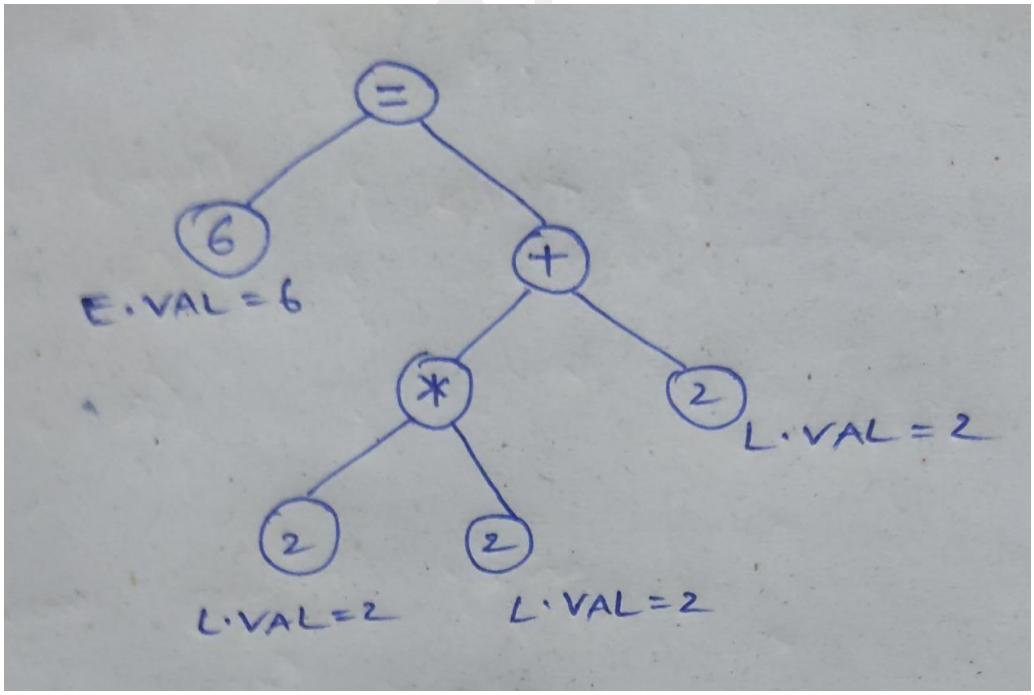
g) Semantic Actions

By using syntax-directed translation, semantic actions—code fragments—are inserted into production bodies.

They are usually enclosed within curly braces ({ }).

Semantic Actions for Arithmetic Expressions : 6=2*2+2

Production	Semantic Actions
E→T.id.D.$	{E.val=Node(T.val, Leaf(id), D.val)}
D' →=.L.	{D'.val=Node(=,L.val)}
L→id.C	{L.val=Node(Leaf(id),C.val)}
C→A.L	{C.val=Node(A.val,L.val)}
A→*	{A.val=Node(*)}
A→+	{A.val=Node(+)}



3) Implementation:

a) Lexical Analyzer

Lex code :-

https://drive.google.com/file/d/1i3saW6Zm4E6jWkh3lOIQNJvWz4QZISR2/view?usp=share_link

Input Code: -

https://drive.google.com/file/d/16UqUo04ZqydQUlkHBWdYJ51aOIhJ9M7u/view?usp=share_link

Output Code :-

https://drive.google.com/file/d/1jntlQwuD1Lga-f7SozjfzTxPmE43qc0f/view?usp=share_link

b) Parser

- **YACC stands for Yet Another Compiler Compiler.**
- **YACC provides a tool to produce a parser for a given grammar**

Parser for Variable Declaration :-

Lex:https://drive.google.com/file/d/1Bo9CsL0NnP3500GNZdu3KgUej52dFOWQ/view?usp=share_link

Yacc : https://drive.google.com/file/d/14iyD4A28As2PzR9dop_naEvkt-inQQX_/view?usp=share_link

Validation :

```
C:\Users\seera\5th semister>a
dig a,b$
dig     is a datatype
a       is a variable
,       is a comma
b       is a variable
$       is a terminator
Valid declaration
```

Parser for Variable Initialization :-

Lex:https://drive.google.com/file/d/1p4CCfZDCf6ceTPv_kg456p-40LjjZ-hS/view?usp=share_link
Yacc:https://drive.google.com/file/d/1dFbcBXHNTipm3XWtXkhA3za3dxsX-j-7/view?usp=share_link

Validation :

```
decimal a=345.2345$
decimal  is a datatype
a        is a variable
=        is a assignment operator
345.2345          is a decimal number
$        is a terminator
Valid initialisation
```

Parser for syntax of Expression statement :-

Lex:https://drive.google.com/file/d/1Bdu1mm4qIkk2fBQOVE_vK5-agprcdPH5/view?usp=share_link
Yacc: https://drive.google.com/file/d/14LWXQNRyQI5tlcHdNocDatlXj280mnhQ/view?usp=share_link
Validation :

```
C:\Users\seera\5th semister>a
max=3*7+2/23+max-1$
max      is a variable
=        is a Assignment operator
3        is a whole number
*        is a  operator
7        is a whole number
+        is a  operator
2        is a whole number
/        is a  operator
23       is a whole number
+        is a  operator
max      is a variable
-        is a  operator
1        is a whole number
$        is a terminator
Valid Expression
```

Parser for syntax of If statement :-

Lex:https://drive.google.com/file/d/1SCvv8W5b_RxCP4rpV5WTzTW8YqU_wAQL/view?usp=share_link

Yacc:https://drive.google.com/file/d/1SVPz_oda9o93Gn7oQrk1BE0pnCdtHH_5/view?usp=share_link
Validation :

```
C:\Users\seera\5th semister>a
if[2+3==5]then show(5)$
if        is a if keyword
[         is the open bracket
2         is a whole number
+         is a operator
3         is a whole number
==        is the relation op
5         is a whole number
]         is the close bracket
then      is a keywork next to if
show      is a print statement
(         is the open parenthesis
5         is a whole number
)         is the closed parenthesis
$         is a terminator
Valid condition
```

Parser for syntax of From statement :-

Lex:https://drive.google.com/file/d/1WJxvAoygBQ760CvyCSEhX231dvVuwO8h/view?usp=share_link
Yacc:https://drive.google.com/file/d/1q-6gSPNhe6M4K7xOEENLXwVpzDfpTai1/view?usp=share_link
Validation :

```
C:\Users\seera\5th semister>a
from[dig a=0$ a<5$a=a+1$]show(a)$
from      is a if keyword
[         is the open bracket
dig       is a datatype
a         is a variable
=         is the assignmet operator
0         is a whole number
$         is a terminator
a         is a variable
<         is the relation op
5         is a whole number
$         is a terminator
a         is a variable
=         is the assignmet operator
a         is a variable
+         is a operator
1         is a whole number
$         is a terminator
]         is the close bracket
show      is a print statement
(         is the open parenthesis
a         is a variable
)         is the closed parenthesis
$         is a terminator
Valid condition
```

4) Our insights on how we design a compiler/interpreter for the chosen language,what are the issues we have faced and how we resolved them.

Implementation

- The data types for characters,decimals,bool,integers, as well as their unique syntax, which includes programming statements like "for loop," "if statement" and "while loop," are all implemented in accordance with the source language that we used to create it.
- For each conditional statement, loop statement, variable declaration, and arithmetic expression a complex CFG was built that is totally free of left recursion and left factorization.
- The source program that was constructed with the aid of CFG, that was able to have its lexemes generated by the lexical analyzer.
- We generated the FIRST and FOLLOW for each CFG that we declared and because the language we designed was free from left factorization and left recursion, the LL(1)parser is also successfully constructed during the syntactic analysis step for each CFG grammar
- The semantic actions for the arithmetic expressions are successfully generated during the semantic analysis.

Issues faced

- It took time to design a CFG grammar for the self-declaration syntaxes; we had hoped for a simple grammar, but the final product was much more complex than we had anticipated.
- It is difficult for us to create a CFG grammar that is free from left recursion and left factorization for self-declaring syntaxes since the parser that we wish to implement for this CFG in the syntax analysis is the LL(1) parser and generating the FIRST and FOLLOW for each non-terminal was quite challenging.
- Implementation of the YACC was more difficult than using the real programming language when building the parse tree from the CFG.
- We only succeeded in implementing the semantic actions for the arithmetic expressions because of the complexity and syntax of the example programmes,

which made it difficult to implement the other items. Generating the semantic actions is a rather difficult task.

Solution for issues

- We tried to split the effort across the team to build the CFG for each syntax we specified that is suited for LL 1 parser since creating a CFG grammar for one's own syntaxes is a very hard process.
- Through several open-source websites, we validated the first, follow and LL(1) table generated by our example software for the CFG.
- After consulting the lectures that were given in the Google Classroom and the provided examples, and referring the websites, the implementation of the LEX and YACC tool was fairly evident.