Loop declaration rules:

1) Syntax of the loop is

run[][([]varname[]=[]integer[])([]varname[]conditionaloperator[]integer)(

space []varname/increment/decrement[])][]{ /n }

2) Spaces should not be missed else it would be treated as Invalid Syntax.

3) Varname must follow the rules of variable declaration (i.e. start with * and end with * and ~~must follow~~ ~~to adhere~~ can be any alphanumeric).

run[(abc X invalid variable name *abc* ✓

4) There is no need to give Data Type before var declaration.

run (Point *abc* = 1) X

↑ it is considered to be nopoint by default.

5) Variable must be ~~de~~ initialized in <initializer> part.

run (Point *ab*){ —— X

↑ no inilization only declaration is not allowed.

6) numbers should be integer float not accepted.

run (Point *abc* = 1.0) (*abc* == 10) (*abc*++){

X must be integer type.

7) All variables names must be same as declaration.

run (Point *abc* = 10)(*abe* >= 10) (*def*++)

X they both must be *abc*

8) ~~No~~ Newline character ~~before~~ a closing '}' is must.

run(——){ } X expected a /n

run(——){

}

10) We are not allowing any Statement within the Loop for now.

run(_____){

Point *abc*! ✗ → although it is correct Statement but we will throw error of ~~th~~ missing Closing Parathesis.

}

Variable declaration Rules along with test cases.

1) Variable name must start and end with '*', can contain any alphabet and digit. (can also start with digit).
   ① *abc*
   ② *9b*
   ] Varname.

2) Variable declaration | Variable initialization.
   DT [ ] Varname ! | DT [ ] Varname [ ] = [ ] value !
   space, no space. | space, space, space.

3) One line can contain only one instruction
   Point *abc*!
   noPoint *def*! ✓
   Point *abc*! noPoint *def*! ✗.

4) Once the variable is defined (declared or initialized) can never be redeclare again
   Point *abc*!
   nopoint *abc*! ✗. *abc* was declared already

5) Multiple declaration is not allowed.
   Point *abc*, *def*! ✗

6) Accepted datatypes are
   Point and nopoint. and they are case sensitive.
   Point *abc*! ✗
   Poin2t *abc*! ✗.

7) A variable can be reffered for initialization after its declaration.
   Point *abc*!
   abc = 3.0! ✓
   abc = 3.0!
   Point *abc*! ✗

8)

Date

1- VARIABLE:

R.E FOR VARIABLE:

$\Sigma = \{*, 0-9, A-Z, a-z\}$

$S = *$

$L = A-Z$

$l = a-z$

$d = 0-9$

$R.E = S(L+l+d)^+ S$ R.E = ^*[a-zA-Z0-9]**$

Example:

12a

Ab2

aAf

2- DATA TYPES:

1) point $(0-9)^+$ R.E = ^[-+]?[0-9]*\.[0-9]*$

2) nopoint $(0-9)^+ . (0-9)^+$ R.E = ^[-+]?[0-9]*$

CFG FOR VARIABLE DECLARATION:

<var> → <DT> * varname *!

<DT> → nopoint | point

example:

point *var1*!

nopoint *2abA*!

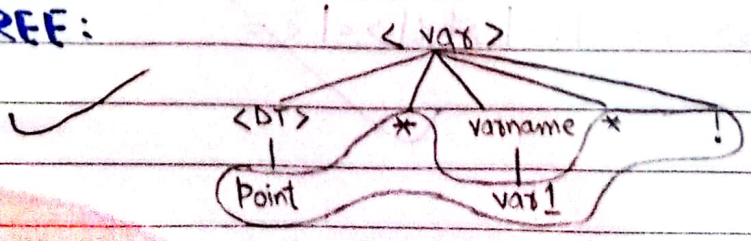
DERIVATION:

point *var1*!

<var> → <DT> * varname *!

→ point * var1 *!

PARSE TREE:



Date

CFG FOR VARIABLE INITIALIZATION:

<init> → <DT> * varname * <space> = <space> <value> ! | <already_var>

<DT> → point | nopoint

<space> → \[]

<value> → d^+ | $d^+ . d^+$

<already_var> → * varname *

Example:

nopoint *var1* [] = [] 4!

point *2abA* [] = [] 2.0!

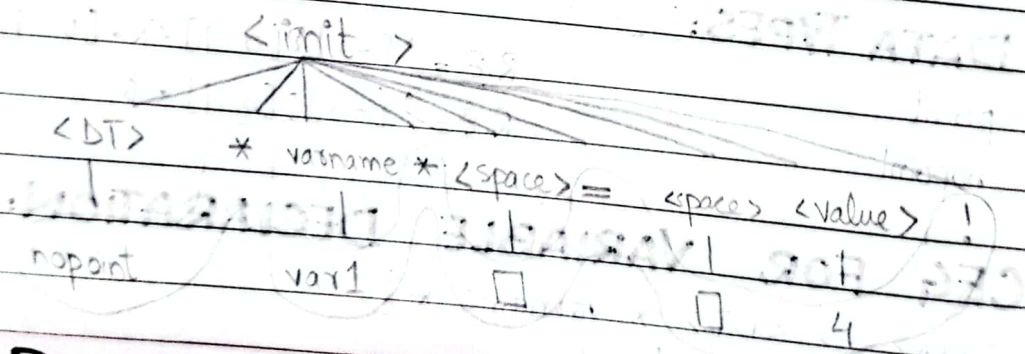
DERIVATION:

nopoint *var1* [] = [] 4!

<init> → <DT> * varname * <space> = <space> <value> !

→ nopoint * var1 * [] = [] 4!

PARSE TREE:



3- LOOP:-

<run> → run <space> [(<initializer>) (<condition>) (<inc/dec>)]

<initializer> → <space> <var> <space> = <space> d^+ <space>

<condition> → <space> <var> <space> <cond-op> <space> d^+

<inc/dec> → <space> <var> ++ <space> | <space> <var> -- <space>

<space> → \[] <space>

<var> → <DT> * varname *

<cond-op> → == | != | > | < | >= | <=

Date _____

DERIVATION: run [(var1 = 4)(var1 < 8)(var1 ++)]

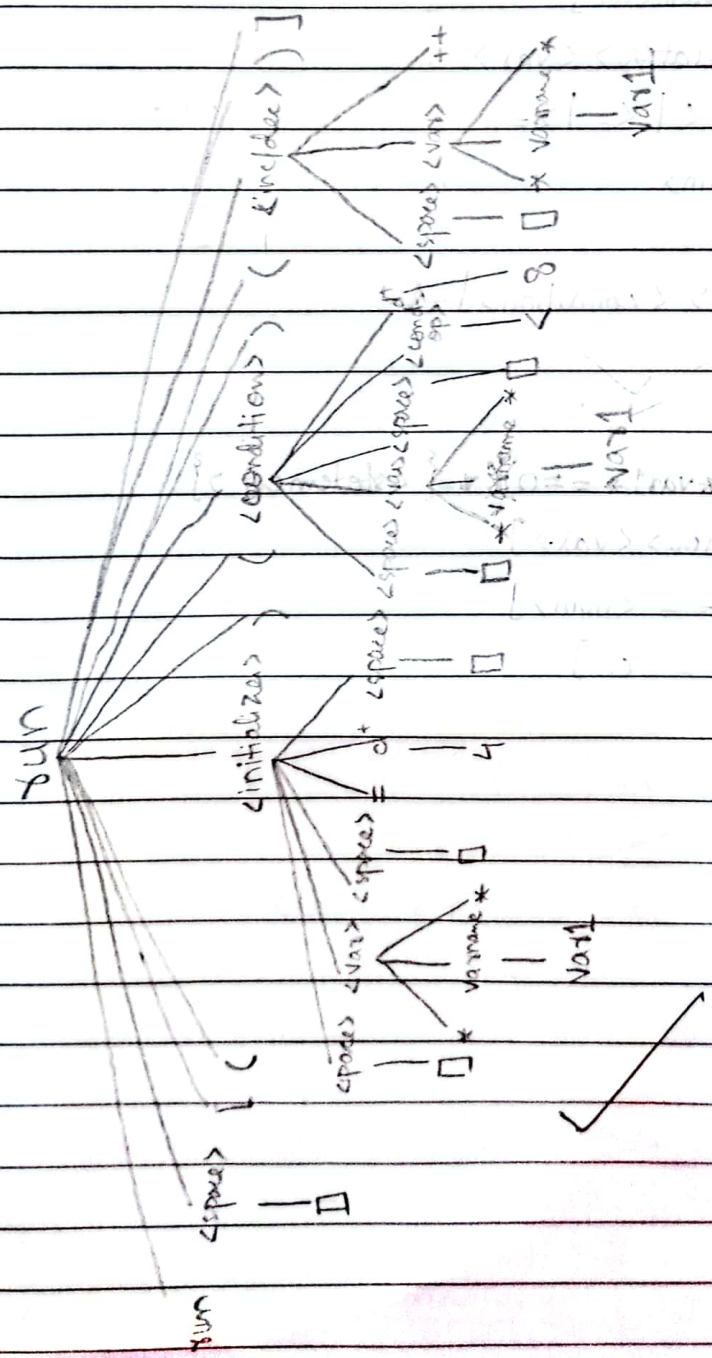
<run> → run <space> [(<initializer>)(<condition>)(<inc/dec>)]

→ run □ [(<space> <var> <space> = <space> d+ <space>)(<space> <var> <space> <cond-op> <space> d+)(<space> <var> ++ <space>)]

→ run □ [(□ * varname * □ = □ 4 □)(□ * varname * □ < □ 8)(□ * varname * ++ □)]

→ run □ [(□ * var1 * □ = □ 4 □)(□ * var1 * □ < □ 8)(□ * var1 * ++ □)]

PARSE TREE:



Date

Keywords:

point, nopoint, run, check, otherwise_check, otherwise

SYMBOLS:

!, ;, (,), {, }, [,], *

CONDITIONAL:

<NT> → check [<condition> <Next>] { <statement> }
otherwise_check [<condition> <Next>] { <statement> }!
otherwise { statement }

<condition> → <var> <operator> <val>
<operator> → == | != | > | < | <= | >=
<val> → <var> | <num>
<num> → d+
<Next> → <logical_op> <condition> | ^
<logical_op> → && | ||

DERIVATION: Check [*var1* == 0] { <statement> }

<NT> → Check [<var> <operator> <val>]
→ check [*varname* == <num>]
→ check [*var1* == 0]