Documentation for PL Proposal

Elway Cortez, James Marck Maranon, King Red Sanchez, Briel Aldous Viola
BSCS 3-1
2/23/21

LANGUAGE PROPOSAL

1. INTRODUCTION

Describe the proposed PL:

The main aim of our proposed language is to make a programming language based on C that would be very easy to learn the basic principles of programming, such as assigning variables, making inputs and outputs for the user, and control flows. The language does this without being too verbose, allowing a person with minimal programming language experience, may it be old folks or young kids with little computer literacy, to actually be able to write code in a manner that most reflects what actual programmers do.

Most programming languages that do this (such as MIT's Scratch or the open source Emojicode) accomplish this by making the interface as user friendly as possible, but for us the act of "writing code" should also be a given feel in teaching programming. The idea of sitting down in front of a screen, writing and debugging code in a language you are just starting to learn can seem like a tall order. By using our language, it would make the daunting task to some, a more manageable and easier to learn format that can be picked up by all, allowing users to focus on learning the concepts behind programming and developing the mindset of problem solving needed for a programmer.

Inspiration for you PL.

The language was inspired by the fact that there are currently no easy ways or shortcuts in learning programming. We endeavour to create a programming language that would make everything in a symbolic, easy to categorise, and recognizable format. Allowing the use of purely symbolic code with the most basic usages be explained as simply as possible, even if the other users don't have the same linguistic acuity as you.

2. SYNTACTIC ELEMENTS OF LANGUAGE (WITH MACHINE)

Character Set -

MEZ= {LETTERS,DIGITS,SYMBOLS}

LETTERS={CAP,SM}

CAP={A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z}

SM= {a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z}

DIGITS= {0,1,2,3,4,5,6,7,8,9}

SYMBOLS={ARITHMETIC,!,",#,$,&,', (,),,,.,:,;,<,=,>,?,@,[,\,],_,`,{,|,},~}

ARITHMETIC={+,-,*,/,%,^}

BOOL={<,>,==,<=,>=,&&,!=,||,!}

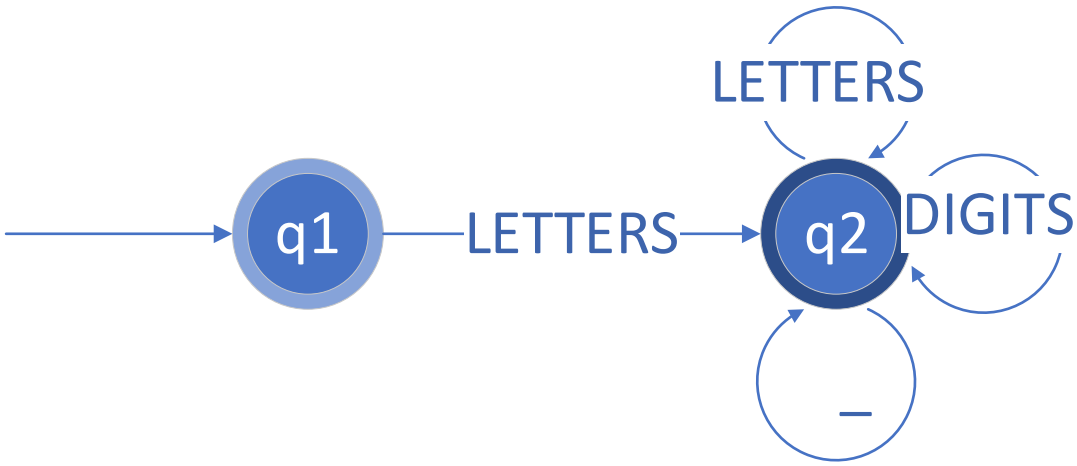
Identifiers:

1. Identifiers can only contain alphanumeric characters (a-z, A-Z, 0-9) and underscores(_).
2. The first letter of an identifier must be a letter(a-z,A-Z).
3. Keywords cannot be used as an identifier.
4. Identifiers are case sensitive.

Regular Expressions

Set of strings that starts with a character from the set LETTERS that may or may not be followed by a combination of characters from set LETTERS, set DIGITS, and underscores.

IDENTIFIERS=LETTERS(LETTERS|DIGITS|_)*

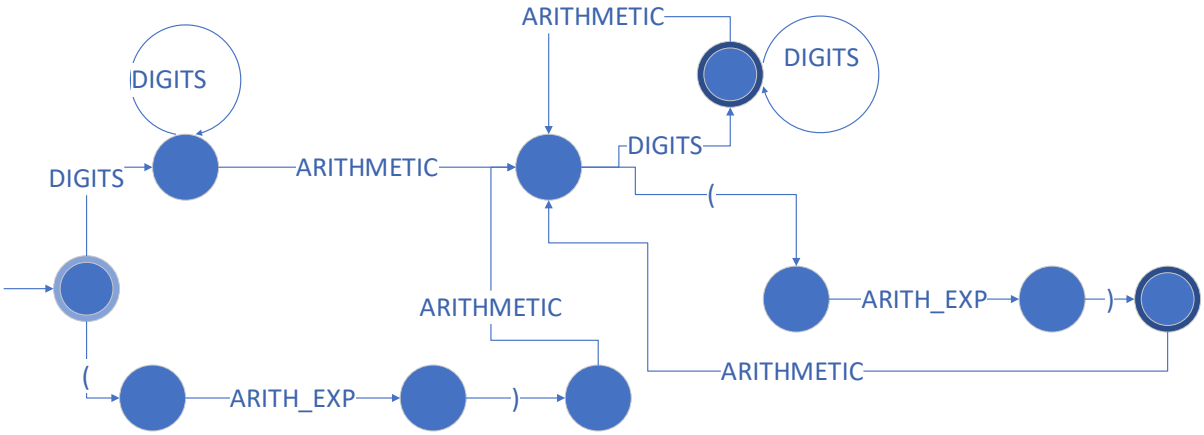


3. OPERATION SYMBOLS (WITH MACHINE)

a. Arithmetic operations {+,-,*,/,%,^}

Regular Expression

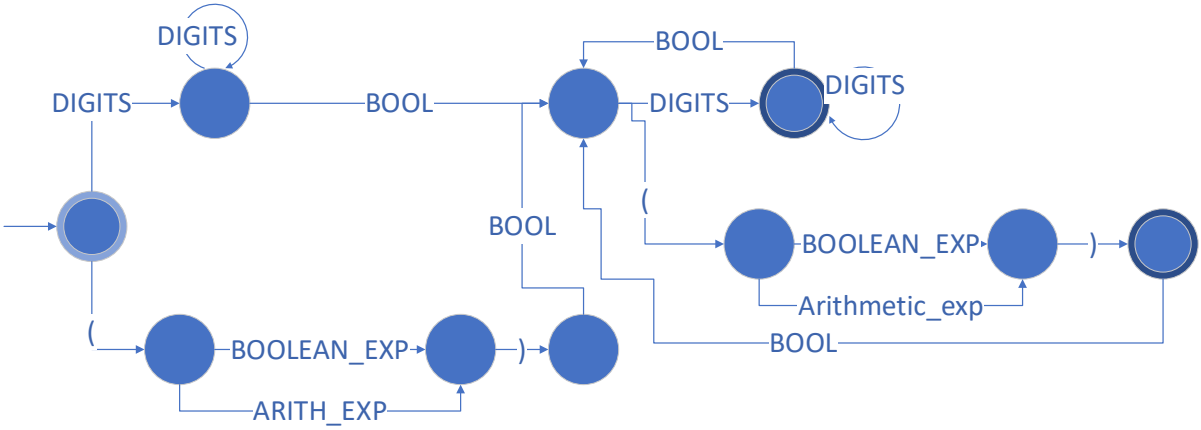
ARITH_EXP=(DIGIT+|((ARITH_EXP)))(ARITHMETIC(DIGIT+|((ARITH_EXP))))+



b. Boolean operations {<,>,<=,>=,==,!=,&&,||,}

Regular Expression

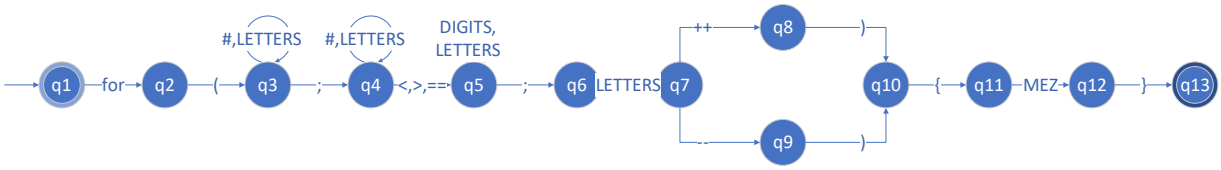
BOOLEAN_EXP=(DIGIT+|((ARITH_EXP|BOOLEAN_EXP)))(BOOL(DIGIT+|((ARITH_EXP|BOOLEAN_EXP))))+



4. KEYWORDS AND RESERVED WORDS (WITH MACHINE)

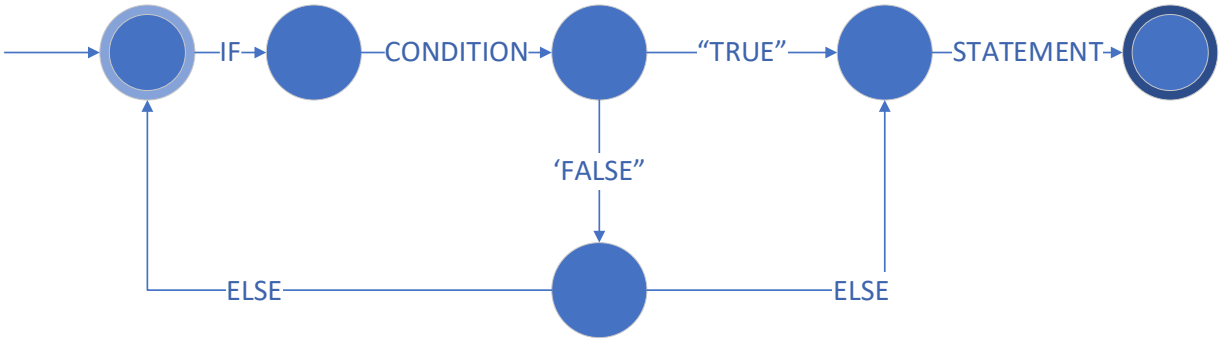
1.) If
2.) for
3.) else
4.) break
5.) unsigned
6.) Num - would identify all numerical data types.
7.) Str - would identify all character and string data types.
8.) print
9.) scan

ITERATIVE LOOP MACHINE (FOR LOOP)



DATA TYPE MACHINE (ASSIGNMENT)

CONDITIONAL STATEMENT MACHINE (IF STATEMENT)



5. NOISE WORDS (WITH MACHINE)

The noise words that would appear in the language would be similar to C in the aspect that it is designed to be functionally similar. Noise words that would appear from it would be auto, register, signed. The cases in which these words can be used are for managing how the assignment is treated, and how to use it, yet removing these keywords would make it function just as fine.

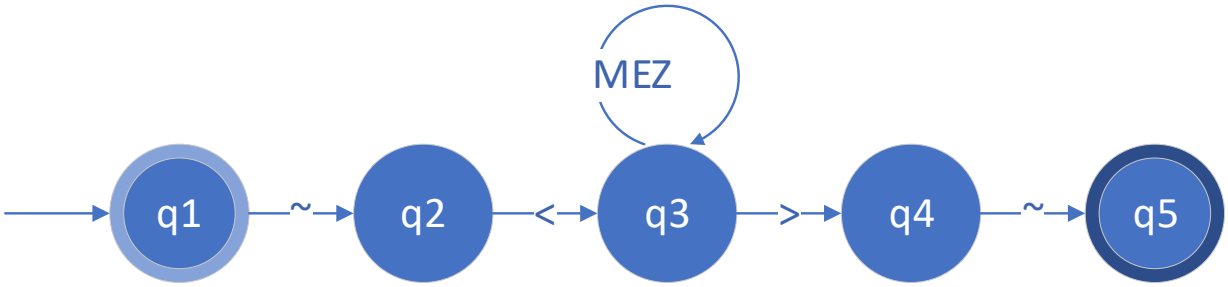
6. COMMENTS (WITH MACHINE)

~< symbols would be a starting point of a comment. >~ would be the end of a comment. Anything found within the symbols ~< and >~ will be treated as a comment.

Regular Expression

Set of strings that starts with ~< followed by a combination of characters from the set MEZ including empty string and ends with >~

~<(MEZ)*>~



7. BLANKS (SPACES)

Blank spaces or whitespaces will be ignored during compilation. It can be used to make the code readable. Spaces, sentence breaks, and tabs can be used but will not have any functional effect on the program.

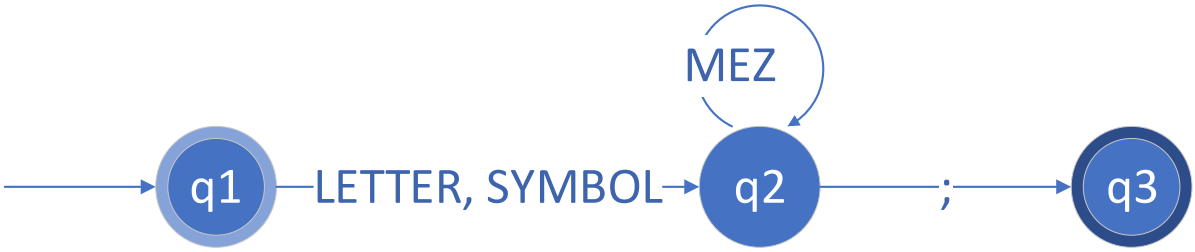
8. DELIMITERS AND BRACKETS (WITH MACHINE)

Delimiter ={;}

We used ";"as our delimiter so that our programming language instills the core habit of putting a semicolon after each line. We retained this practice because we believe that instilling this core practice helps the future users of our programming language to be robust and have a solid foundation of practice that will help them in the long run

Regular Expression

DELIMITER= ((LETTERS,SYMBOLS(MEZ)*);)



Brackets

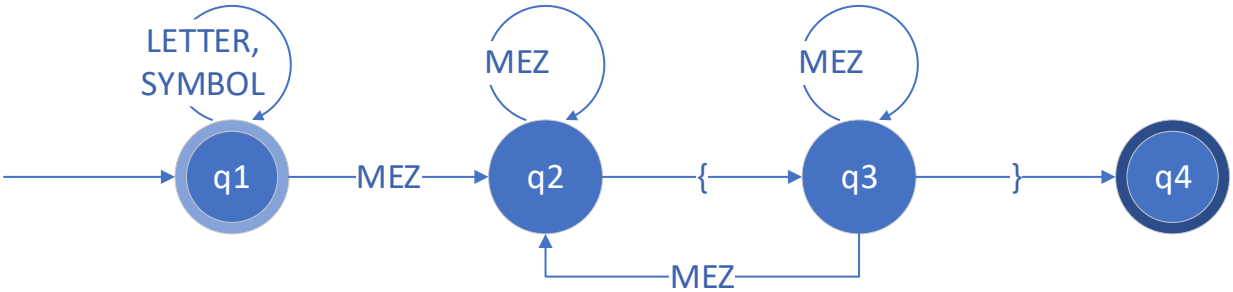
() – Parenthesis this is used for enclosing the parameters of method as well for the main condition of the conditional statements

{}- used for initializing an array and used for enclosing the body of the methods and classes

[]- used for initializing an array

Regular Expression

BRACKETS "{}"=(MEZMEZ*{(MEZ+MEZMEZ*{)*})



9. FREE-AND-FIXED-FIELD FORMATS

The proposed PL would be following the free field format, meaning the positioning of the characters would be irrelevant for the compilation of the program. Akin to c where we based our language, putting spaces, tabs and sentence brakes are purely for the ease of readability and writability.

10. EXPRESSION

Arithmetic Expression

Order of Precedence: (), *, /, %, +, -

Rules of Arithmetic Operation:

1. The operators in expressions contained within pairs of parentheses (‘()’) are evaluated first. In case of nested parentheses, the operators in the innermost are applied first.
2. The Multiplication (‘*’), division (‘/’), and remainder (‘%’) operations are applied next. If an expression contains several multiplication, division and remainder operations, evaluation proceeds from left to right.
3. The Addition and subtraction operations are evaluated next. If an expression contains several addition and subtraction operations, evaluation proceeds from left to right.
4. If there are several operators of the same precedence, they will be evaluated from left to right.
5. The assignment operator (=) is evaluated last.

Boolean Expression

Order of Precedence:

Relational: >, >=, <, <=, ==, !=

Logical: !, &, |

Rules of Boolean Expressions:

1. The relational operators will be evaluated first before the logical operators.
2. Relational operators have left to right associativity. the operator with the highest precedence is grouped with its operand(s) first. Then the next highest operator will be grouped with its operands, and so on.
3. Logical Operators also have left to right associativity. the operator with the highest precedence is grouped with its operand(s) first. Then the next highest operator will be grouped with its operands, and so on.
4. If there are several operators of the same precedence, they will be evaluated from left to right.

11. STATEMENTS

Declaration Statement

- Num x
- Num x, y, z;
- Num num;
- Str string;
- Str char;

Assignment Statement

- Num num = 100;
- Num num = 4.20;
- Num num = 201869420;
- Num x = y + z;
- Str string = “Hello World”;
- Str letter = ‘a’;

Conditional Statement

- if statement

```
1. if (condition){
2.          <statement/s>
3. }
```

- if else statement

```
1. if (condition){
2.   <statement/s>
3. }
```

- else if statement

```
1. if (condition){
2.   <statement/s>
3. } else if (condition){
4. <statement/s>
5. }
```

Iterative Statement

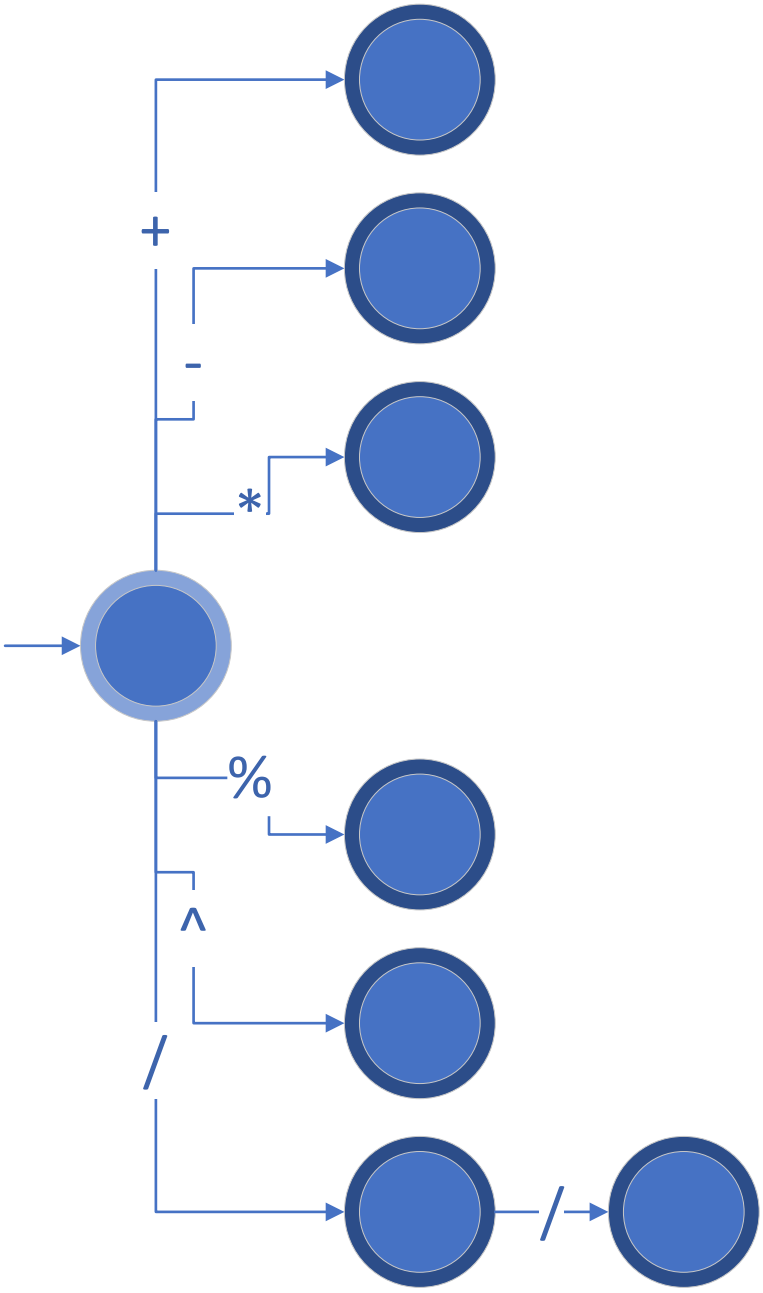
- for statement

```
1. for(condition1, condition2, condition3){
2.   <statement/s>
3. }
```

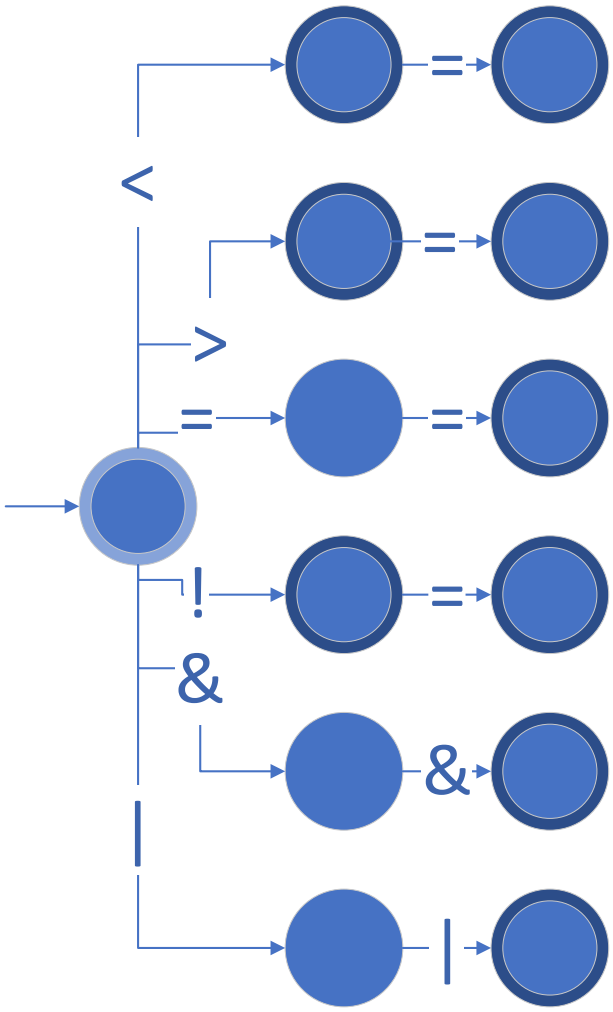
LEXICAl ANALYZER

1. OPERATION SYMBOLS (WITH MACHINE)

a. Arithmetic operations {+,-,*,/,%,^,//}



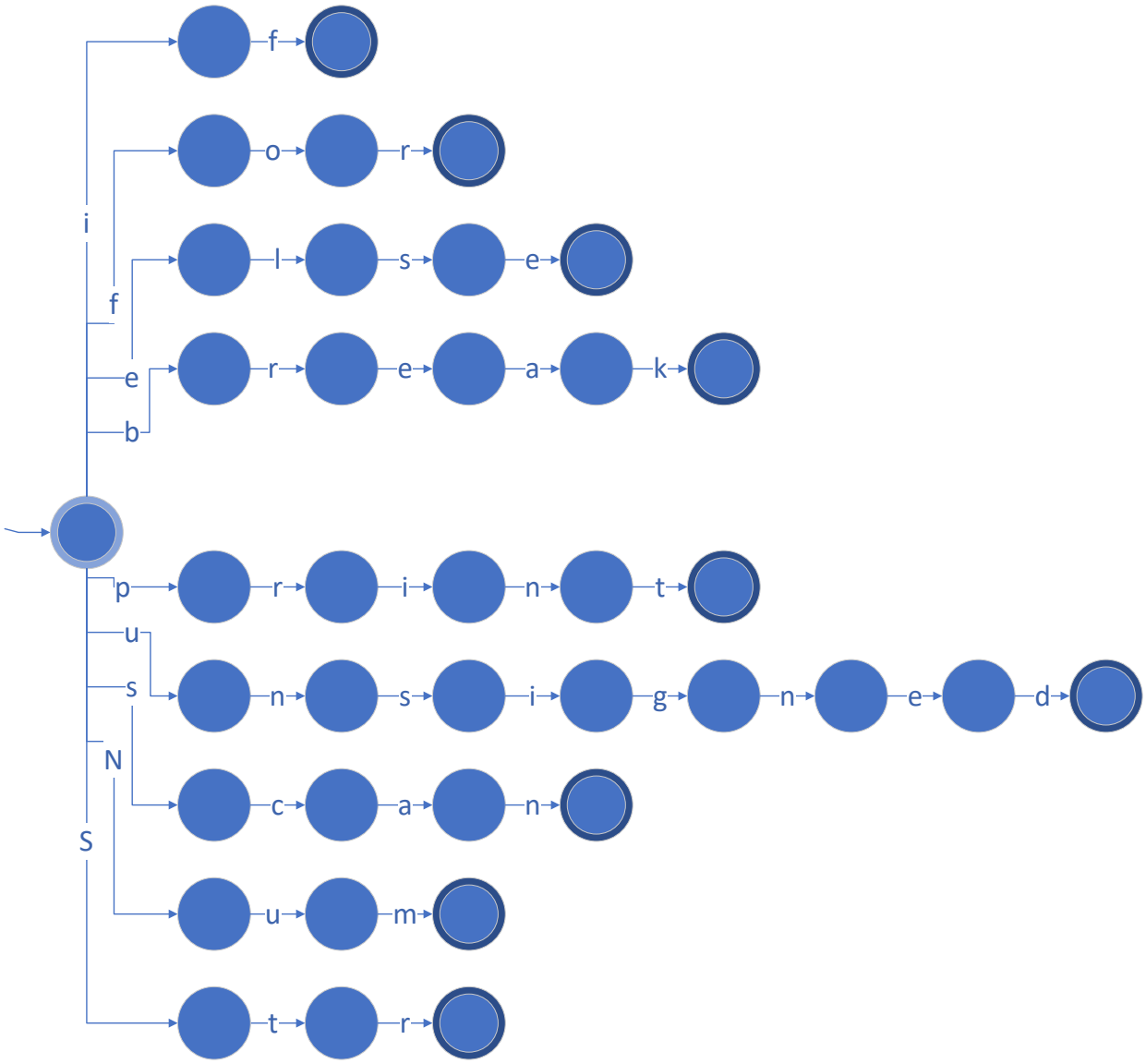
b. Boolean operations {<,>,<=,>=,==,!=,&&,||,!}



2. KEYWORDS AND RESERVED WORDS (WITH MACHINE)

Keywords

1.) if
2.) for
3.) else
4.) break
5.) unsigned
6.) Num - would identify all numerical data types.
7.) Str - would identify all character and string data types.
8.) print
9.) scan

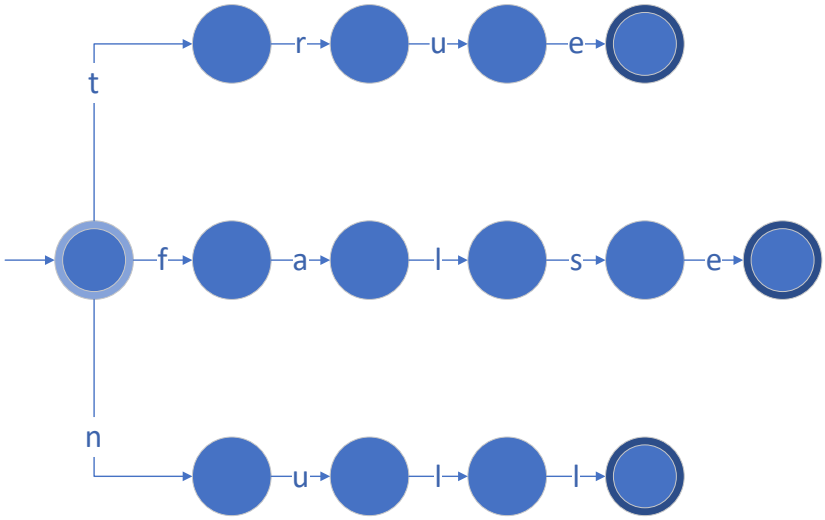


Reserved words

1.) true

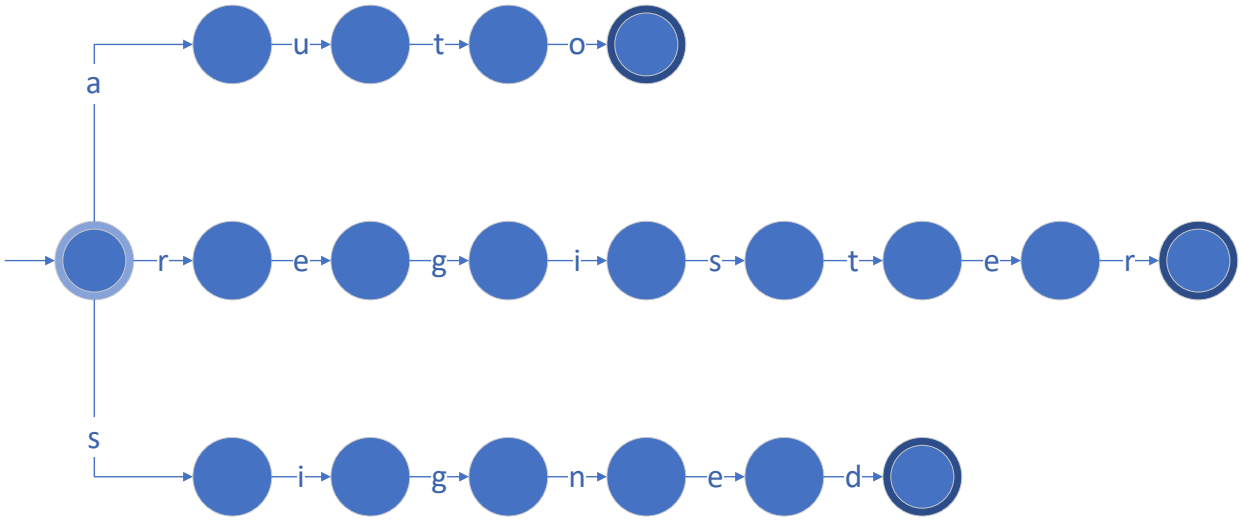
2.) false

3.) null


t
r
u
e
f
a
l
s
e
n
u
l
l

3. NOISE WORDS (WITH MACHINE)

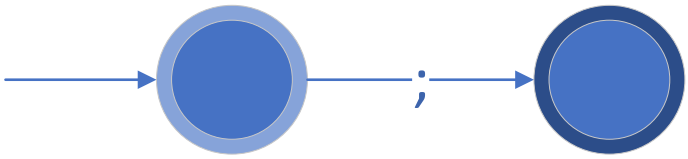
1. auto
2. register
3. signed



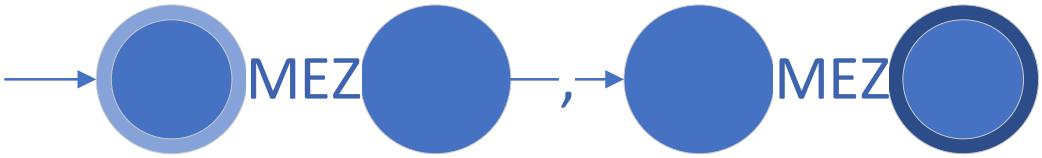
4. DELIMITERS AND BRACKETS (WITH MACHINE)

Delimiter ={;}

We used “;”as our delimiter so that our programming language instills the core habit of putting a semicolon after each line. We retained this practice because we believe that instilling this core practice helps the future users of our programming language to be robust and have a solid foundation of practice that will help them in the long run



Delimiter ={,}

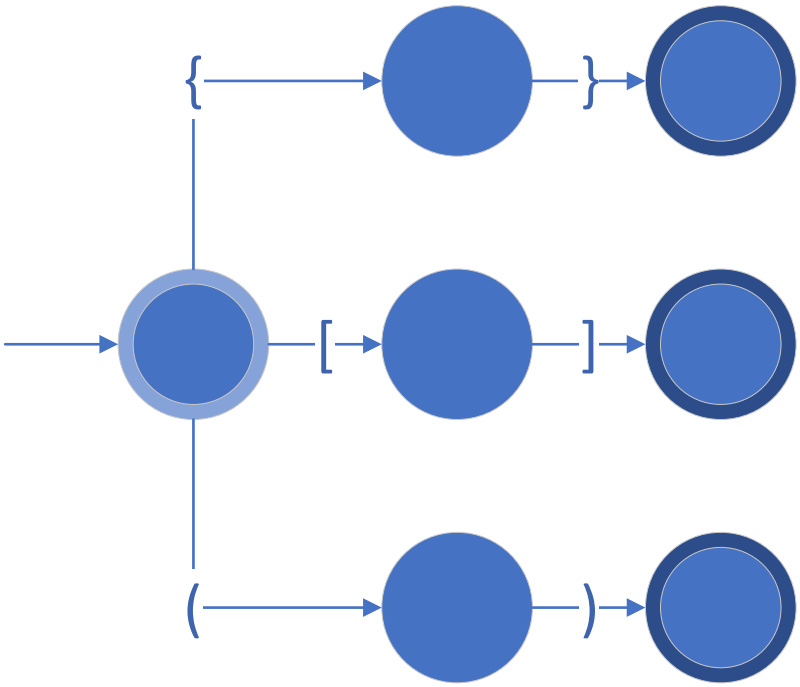


Brackets

() – Parenthesis this is used for enclosing the parameters of method as well for the main condition of the conditional statements

{}- used for initializing an array and used for enclosing the body of the methods and classes

[]- used for initializing an array



SYNTAX ANALYZER

I. SYNTAX

1. CONTEXT FREE GRAMMAR OF THE PROPOSED PL WITH EXAMPLES

<MEZ_STATEMENT> ::=

<STATEMENT>

|Comment

<STATEMENT> ::=

<Compound_Stmt>

|<Declaration_Stmt>

|<Iteration_Stmt>

|<Selection_Stmt>

|<Expression_Stmt>

|<Input_Stmt>

|<Output_Stmt>

|break ;

<Compound_STMT> ::=

<STATEMENT> <STATEMENT>

|<STATEMENT> <Compound_Stmt>

<Declaration_Stmt> ::=

<DATA_TYPE> <Identifiers> ;

|<DATA_TYPE> <id_list> ;

|<DATA_TYPE> <Assignment_Expression> ;

|<STRG_CLASS_SPECIFIER> <DATA_TYPE> <Identifiers> ;

|<STRG_CLASS_SPECIFIER> <DATA_TYPE> <id_list> ;

|<STRG_CLASS_SPECIFIER> <DATA_TYPE> <Assignment_Expression> ;

Example:

Num var; (<DATA_TYPE> <id> ;)

Num x, y, z; (<DATA_TYPE> <id_list> ;)

Num var = 100; (<DATA_TYPE> <Assignment_Expression> ;)

unsigned Num var; (<STRG_CLASS_SPECIFIER> <DATA_TYPE> <Identifiers> ;)

auto Num x, y, z; (<STRG_CLASS_SPECIFIER> <DATA_TYPE> <id_list> ;)

register Num var = 100; (<STRG_CLASS_SPECIFIER> <DATA_TYPE> <Assignment_Expression> ;)

<Identifiers> ::=

id

|id [integer_Const]

<id_list> ::=

<Identifiers> , <Identifiers>

|<Identifiers> , <id_list>

<Assignment_Expression> ::=

<Identifiers> Assignment_Op <Single_Expression>

|<Identifiers> Assignment_Op <Operation_Expression>

|<Identifiers> Assignment_Op { <Array_contents> }

Example:

x = 100 (<Identifiers> Assignment_Op <Single_Expression>)

x = 2 + 15 (<Identifiers> Assignment_Op <Operation_Expression>)

<Array_contents> ::=

<CONSTANT> , <CONSTANT>

|<Array_contents> , <CONSTANT>

|<CONSTANT>

<Single_Expression> ::=

<Identifiers>

|<CONSTANT>

<CONSTANT> ::=

integer_Const

|float_Const

|string_Const

|character_Const

|boolean_Const

<Operation_Expression> ::=

<Arithmetic_Exp>

|<Boolean_Exp>

<Arithmetic_Exp> ::=

<Single_Expression> <Arithmetic_Op> <Single_Expression>

|<Single_Expression> <Arithmetic_Op> (<Operation_Expression>)

|(<Operation_Expression>) <Arithmetic_Op> <Single_Expression>

|(<Operation_Expression>) <Arithmetic_Op> (<Operation_Expression>)

<Boolean_Exp> ::=

<Single_Expression> <Boolean_Op> <Single_Expression>

|<Single_Expression> <Boolean_Op> (<Operation_Expression>)

|(<Operation_Expression>) <Boolean_Op> <Single_Expression>

|(<Operation_Expression>) <Boolean_Op> (<Operation_Expression>)

<Iteration_Stmt> ::=

for (<Expression> ; <Expression> ; <Expression>) { <STATEMENT> }

Example:

for (i = 0 ; i < 10; i = i + 1) { x = x + 2; } (for (<Expression> ; <Expression> ; <Expression>) { STATEMENT })

<Expression> ::=

<Assignment_Expression>

|<Operation_Expression>

|<Single_Expression>

<Selection_Stmt> ::=

if (<Expression>) { <STATEMENT> }

|if (<Expression>) { <STATEMENT> } else { <STATEMENT> }

|if (<Expression>) { <STATEMENT> } else <Selection_Stmt>

Example:

if (x > 10) { var = 0; } (if (<Expression>) { STATEMENT })

if (x == true) { x = 10; } else { x = 0; } (if (<Expression>) { STATEMENT } else {STATEMENT})

if (x == 1) { x = 10; } else if (x > 20) { x = 1; } else {x = 0}; (if (<Expression>) { STATEMENT } else <Selection_Stmt>)

<Expression_Stmt> ::=

<Expression> ;

<Input_Stmt> ::=

scan (<DATA_TYPE> , id) ;

Example:

scan (Num , x); (scan (<DATA_TYPE> , <id>) ;)

<Output_Stmt> ::=

print (<output>) ;

Example:

print (“Hello”); (print (<output>) ;)

<output> ::=

string_Const

|id

|<output> , <output>

2. DERIVATION THROUGH INSTANTANEOUS DESCRIPTION AND PARSE TREES

Input Statement

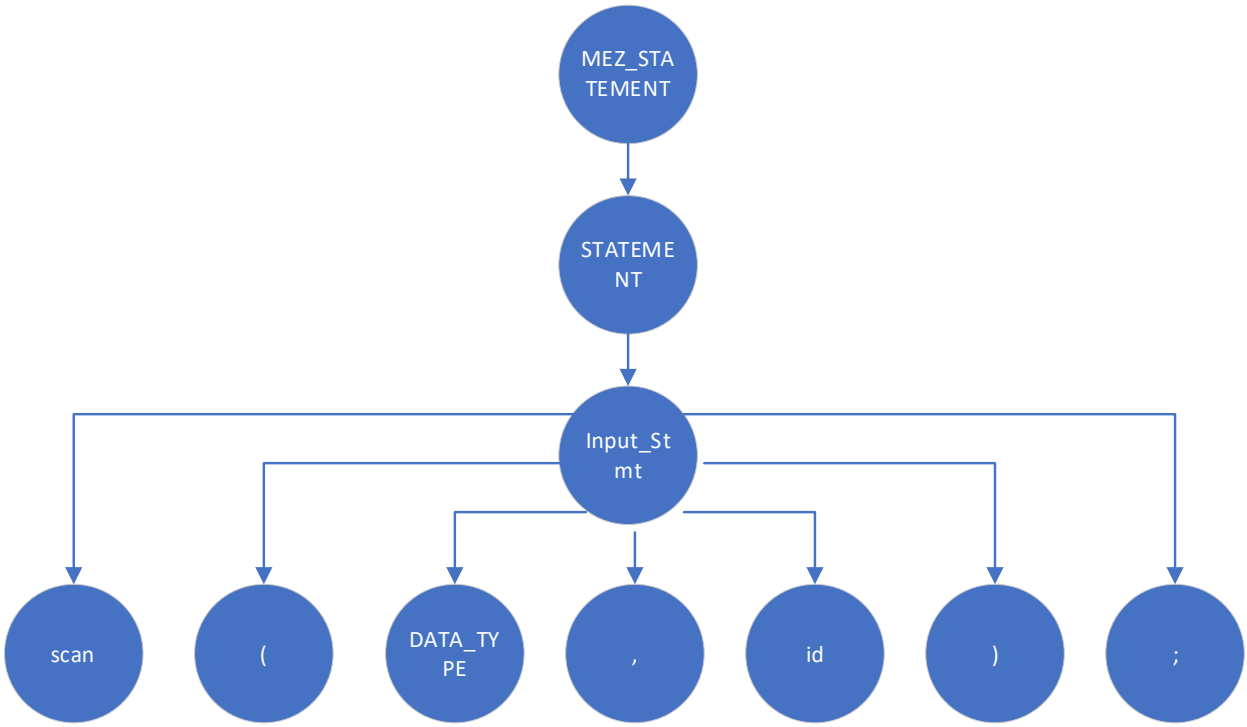
scan (<DATA_TYPE> , <id>) ;

scan (Num , x);

<Input_Stmt>-> scan (<DATA_TYPE> , <id>) ;

-> scan (Num , <id>) ;

-> scan (Num , x) ;



Output Statement

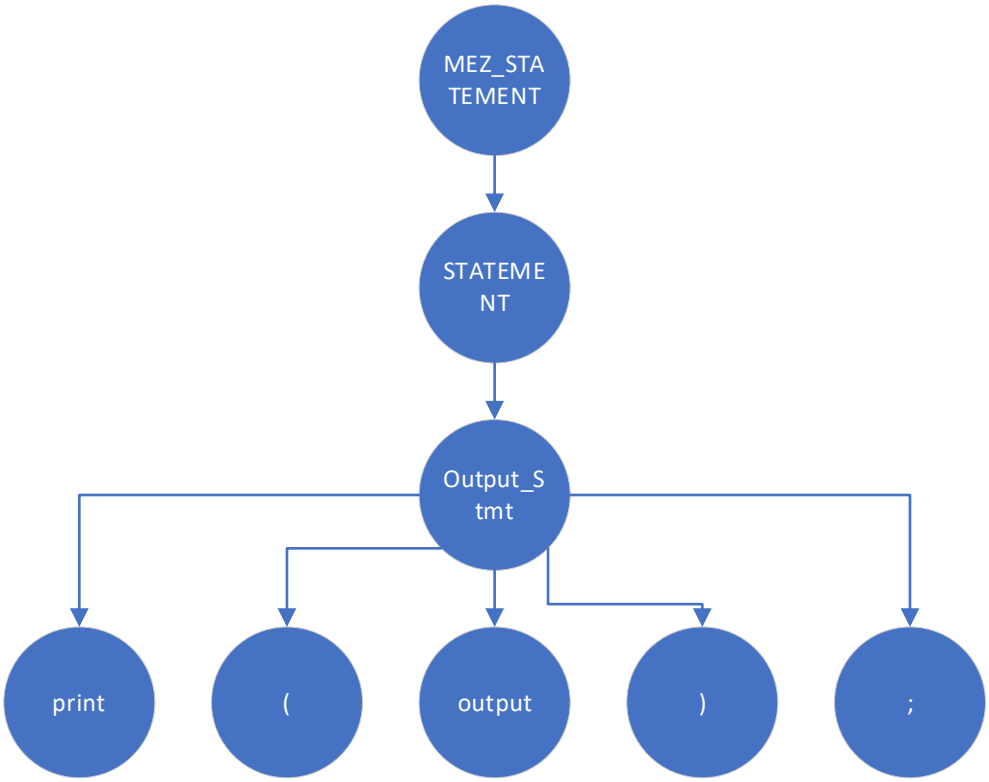
print (<output>) ;

print(x);

<Output_Stmt>-> print (<output>) ;

-> print (id) ;

-> print (x)



Assignment Statement

x = 100

(<Identifiers> Assignment_Op <Single_Expression>)

<Assignment_Expression> -> <Identifiers> Assignment_Op <Single_Expression>

-> id Assignment_Op <Single_Expression>

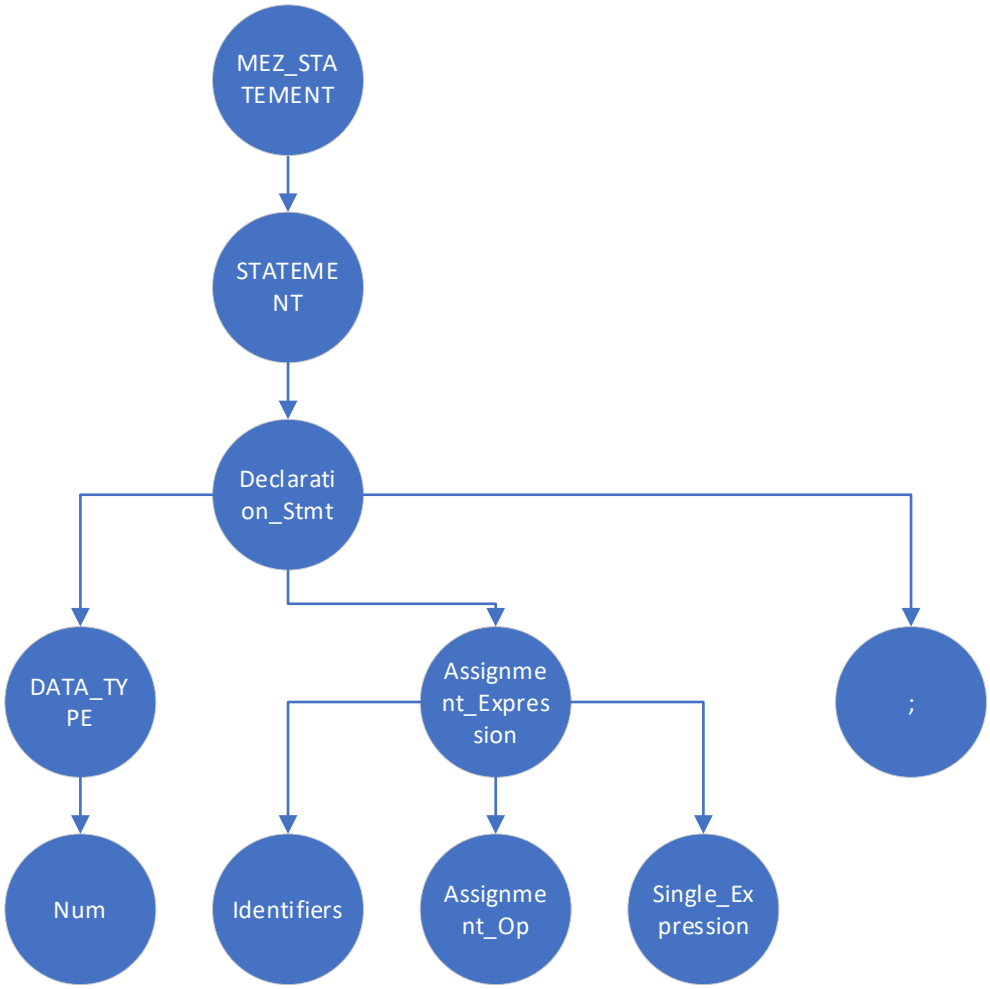
-> x Assignment_Op <Single_Expression>

-> x = <Single_Expression>

-> x = <CONSTANT>

-> x = integer_Const

-> x = 100



Condition Statement

if (x == 1) { x = 10; } else if (x > 20) { x = 1; } else {x = 0}

if (<Expression>) { STATEMENT } else <Selection_Stmt>

<Selection_Stmt> -> if (<Expression>) { <STATEMENT> } else <Selection_Stmt>

-> if (<Operation_Expression>) else <Selection_Stmt>

-> if (<Boolean_Exp>) else <Selection_Stmt>

-> if (<Single_Expression> <Boolean_Op> <Single_Expression>) { <STATEMENT> } else <Selection_Stmt>

-> if (<Identifiers> <Boolean_Op> <Single_Expression>) { <STATEMENT> } else <Selection_Stmt>

->if (id <Boolean_Op> <Single_Expression>) { <STATEMENT> } else <Selection_Stmt>

->if (x <Boolean_Op> <Single_Expression>) { <STATEMENT> } else <Selection_Stmt>

->if (x == <Single_Expression>) { <STATEMENT> } else <Selection_Stmt>

->if (x == <CONSTANT>) { <STATEMENT> } else <Selection_Stmt>

->if (x == integer_Const) else <Selection_Stmt>

->if (x == 1) { <STATEMENT> } else <Selection_Stmt>

->if (x == 1) { <Expression_Stmt> } else <Selection_Stmt>

->if (x == 1) { <Expression> ; } else <Selection_Stmt>

->if (x == 1) { <Assignment_Expression> ; } else <Selection_Stmt>

->if (x == 1) { <Identifiers> Assignment_Op <Single_Expression> ; } else <Selection_Stmt>

->if (x == 1) { id Assignment_Op <Single_Expression> ; } else <Selection_Stmt>

->if (x == 1) { x Assignment_Op <Single_Expression> ;} else <Selection_Stmt>

->if (x == 1) { x = <Single_Expression> ; } else <Selection_Stmt>

->if (x == 1) { x = <CONSTANT> ; } else <Selection_Stmt>

->if (x == 1) { x = integer_Const ; } else <Selection_Stmt>

->if (x == 1) { x = 10 ; } else <Selection_Stmt>

->if (x == 1) { x = 10 ;} else if (<Expression>) { <STATEMENT> } else { <STATEMENT> }

->if (x == 1) { x = 10 ; } else if (<Operation_Expression>) { <STATEMENT> } else { <STATEMENT> }

->if (x == 1) { x = 10 ; } else if (<Boolean_Exp>) { <STATEMENT> } else { <STATEMENT> }

->if (x == 1) { x = 10 ; } else if (<Single_Expression> <Boolean_Op> <Single_Expression>) { <STATEMENT> } else { <STATEMENT> }

->if (x == 1) { x = 10 ; } else if (<Identifiers> <Boolean_Op> <Single_Expression>) { <STATEMENT> } else { <STATEMENT> }

->if (x == 1) { x = 10 ; } else if (id <Boolean_Op> <Single_Expression>) { <STATEMENT> } else { <STATEMENT> }

->if (x == 1) { x = 10 ; } else if (x <Boolean_Op> <Single_Expression>) { <STATEMENT> } else { <STATEMENT> }

->if (x == 1) { x = 10 ; } else if (x > <CONSTANT>) { <STATEMENT> } else { <STATEMENT> }

->if (x == 1) { x = 10 ; } else if (x > integer_Const) { <STATEMENT> } else { <STATEMENT> }

->if (x == 1) { x = 10 ; } else if (x > 20) { <STATEMENT> } else { <STATEMENT> }

->if (x == 1) { x = 10 ; } else if (x > 20) { <Expression_Stmt> } else { <STATEMENT> }

->if (x == 1) { x = 10 ; } else if (x > 20) { <Expression> ; } else { <STATEMENT> }

->if (x == 1) { x = 10 ; } else if (x > 20) { <Assignment_Expression> ; } else { <STATEMENT> }

->if (x == 1) { x = 10 ; } else if (x > 20) { <Identifiers> Assignment_Op <Single_Expression> ; } else { <STATEMENT> }

->if (x == 1) { x = 10 ; } else if (x > 20) { id Assignment_Op <Single_Expression> ; } else { <STATEMENT> }

->if (x == 1) { x = 10 ; } else if (x > 20) { x Assignment_Op <Single_Expression> ; } else { <STATEMENT> }

->if (x == 1) { x = 10 ; } else if (x > 20) { x = <Single_Expression> ; } else { <STATEMENT> }

->if (x == 1) { x = 10 ; } else if (x > 20) { x = integer_Const ; } else { <STATEMENT> }

->if (x == 1) { x = 10 ; } else if (x > 20) { x = 1 ; } else { <STATEMENT> }

->if (x == 1) { x = 10 ; } else if (x > 20) { x = 1 ; } else { <Expression_Stmt> }

->if (x == 1) { x = 10 ; } else if (x > 20) { x = 1 ; } else { <Expression> ;}

->if (x == 1) { x = 10 ; } else if (x > 20) { x = 1 ; } else { <Assignment_Expression> ; }

->if (x == 1) { x = 10 ; } else if (x > 20) { x = 1 ; } else { <Identifiers> Assignment_Op <Single_Expression> ; }

->if (x == 1) { x = 10 ; } else if (x > 20) { x = 1 ; } else { id Assignment_Op <Single_Expression> ; }

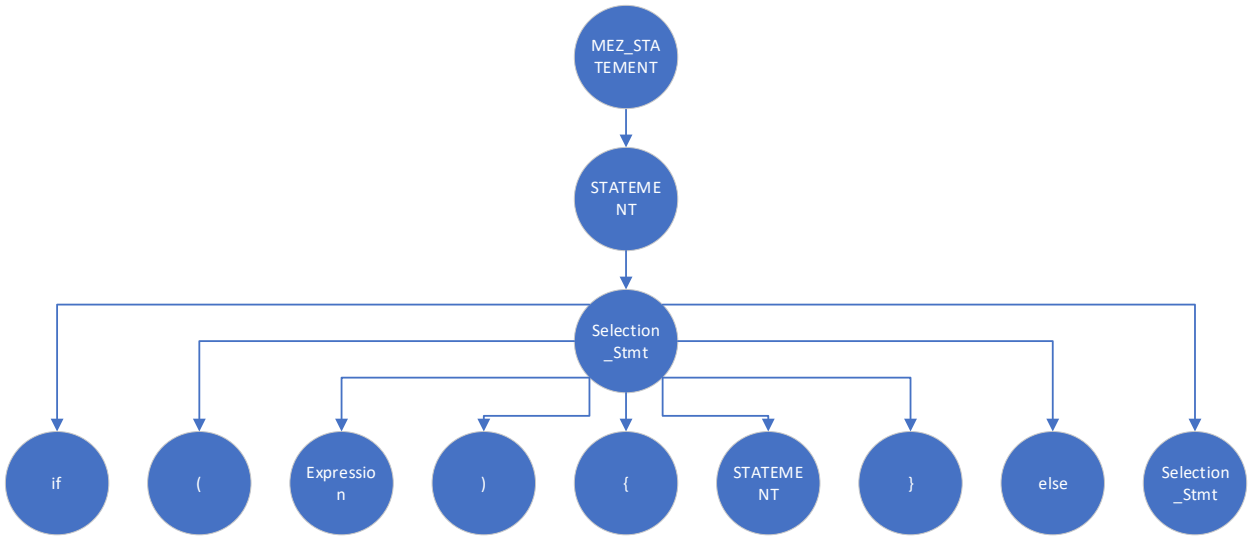
->if (x == 1) { x = 10 ; } else if (x > 20) { x = 1 ; } else { x Assignment_Op <Single_Expression> ; }

->if (x == 1) { x = 10 ; } else if (x > 20) { x = 1 ; } else { x = <Single_Expression> ; }

->if (x == 1) { x = 10 ; } else if (x > 20) { x = 1 ; } else { x = <CONSTANT> }

->if (x == 1) { x = 10 ; } else if (x > 20) { x = 1 ; } else { x = integer_Const> }

->if (x == 1) { x = 10 ; } else if (x > 20) { x = 1 ; } else { x = 0 ; }



Iterative Statement

for (i = 0 ; i < 10; i = i + 1) { x = 2 ; }

(for (<Expression> ; <Expression> ; <Expression>) { STATEMENT })

<Iteration_Stmt> -> for (<Expression> ; <Expression> ; <Expression>) { <STATEMENT> }

-> for (<Assignment_Expression> ; <Expression> ; <Expression>) { <STATEMENT> }

-> for (<Identifiers> Assignment_Op <Single_Expression> ; <Expression> ; <Expression>) { <STATEMENT> }

-> for (id Assignment_Op <Single_Expression> ; <Expression> ; <Expression>) { <STATEMENT> }

-> for (i Assignment_Op <Single_Expression> ; <Expression> ; <Expression>) { <STATEMENT> }

-> for (i = <Single_Expression> ; <Expression> ; <Expression>) { <STATEMENT> }

-> for (i = <CONSTANT> ; <Expression> ; <Expression>) { <STATEMENT> }

-> for (i = integer_Const ; <Expression> ; <Expression>) { <STATEMENT> }

-> for (i = 0 ; <Expression> ; <Expression>) { <STATEMENT> }

-> for (i = 0 ; <Boolean_Exp> ; <Expression>) { <STATEMENT> }

-> for (i = 0 ; <Identifiers> <Boolean_Op> <Single_Expression> ; <Expression>) { <STATEMENT> }

-> for (i = 0 ; id <Boolean_Op> <Single_Expression> ; <Expression>) { <STATEMENT> }

-> for (i = 0 ; i <Boolean_Op> <Single_Expression> ; <Expression>) { <STATEMENT> }

-> for (i = 0 ; i < <Single_Expression> ; <Expression>) { <STATEMENT> }

-> for (i = 0 ; i < <CONSTANT> ; <Expression>) { <STATEMENT> }

-> for (i = 0 ; i < integer_Const ; <Expression>) { <STATEMENT> }

-> for (i = 0 ; i < 10 ; <Expression>) { <STATEMENT> }

-> for (i = 0 ; i < 10 ; <Assignment_Expression>) { <STATEMENT> }

-> for (i = 0 ; i < 10 ; <Identifiers> Assignment_Op <Operation_Expression>) { <STATEMENT> }

-> for (i = 0 ; i < 10 ; id Assignment_Op <Operation_Expression>) { <STATEMENT> }

-> for (i = 0 ; i < 10 ; i Assignment_Op <Operation_Expression>) { <STATEMENT> }

-> for (i = 0 ; i < 10 ; i = <Operation_Expression>) { <STATEMENT> }

-> for (i = 0 ; i < 10 ; i = <Single_Expression> <Arithmetic_Op> <Single_Expression>) { <STATEMENT> }

-> for (i = 0 ; i < 10 ; i = <Identifiers> <Arithmetic_Op> <Single_Expression>) { <STATEMENT> }

-> for (i = 0 ; i < 10 ; i = id <Arithmetic_Op> <Single_Expression>) { <STATEMENT> }

> for (i = 0 ; i < 10 ; i = i <Arithmetic_Op> <Single_Expression>) { <STATEMENT> }

-> for (i = 0 ; i < 10 ; i = i + <Single_Expression>) { <STATEMENT> }

-> for (i = 0 ; i < 10 ; i = i + <CONSTANT>) { <STATEMENT> }

-> for (i = 0 ; i < 10 ; i = i + integer_Const) { <STATEMENT> }

-> for (i = 0 ; i < 10 ; i = i + 1) { <STATEMENT> }

-> for (i = 0 ; i < 10 ; i = i + 1) { <Expression_Stmt> }

-> for (i = 0 ; i < 10 ; i = i + 1) { <Expression> ; }

-> for (i = 0 ; i < 10 ; i = i + 1) { <Assignment_Expression> ; }

-> for (i = 0 ; i < 10 ; i = i + 1) { <Identifiers> Assignment_Op <Single_Expression> ; }

-> for (i = 0 ; i < 10 ; i = i + 1) { id Assignment_Op <Single_Expression> ;
}

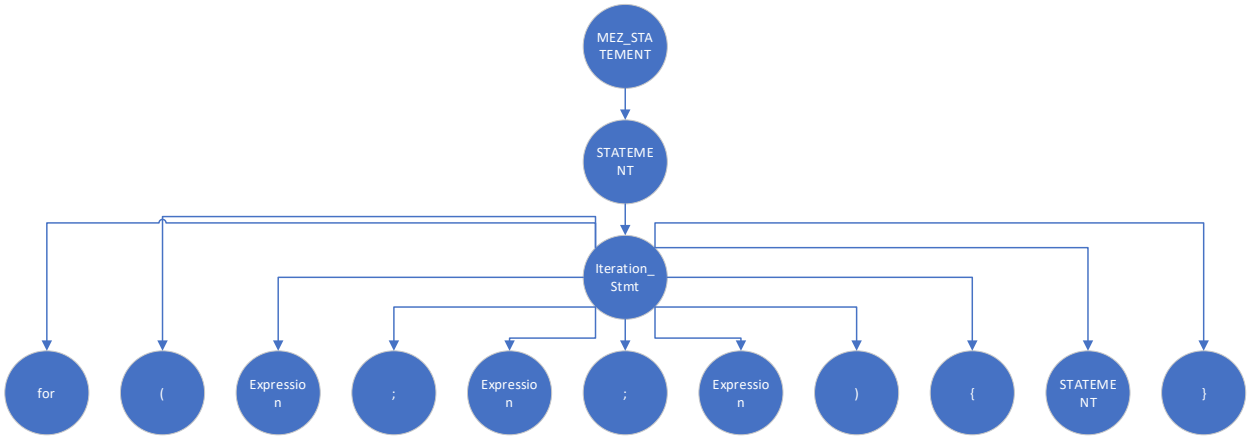
-> for (i = 0 ; i < 10 ; i = i + 1) { x Assignment_Op <Single_Expression> ;
}

-> for (i = 0 ; i < 10 ; i = i + 1) { x = <Single_Expression> ; }

-> for (i = 0 ; i < 10 ; i = i + 1) { x = <CONSTANT> ; }

-> for (i = 0 ; i < 10 ; i = i + 1) { x = integer_Const ; }

-> for (i = 0 ; i < 10 ; i = i + 1) { x = 2 ; }

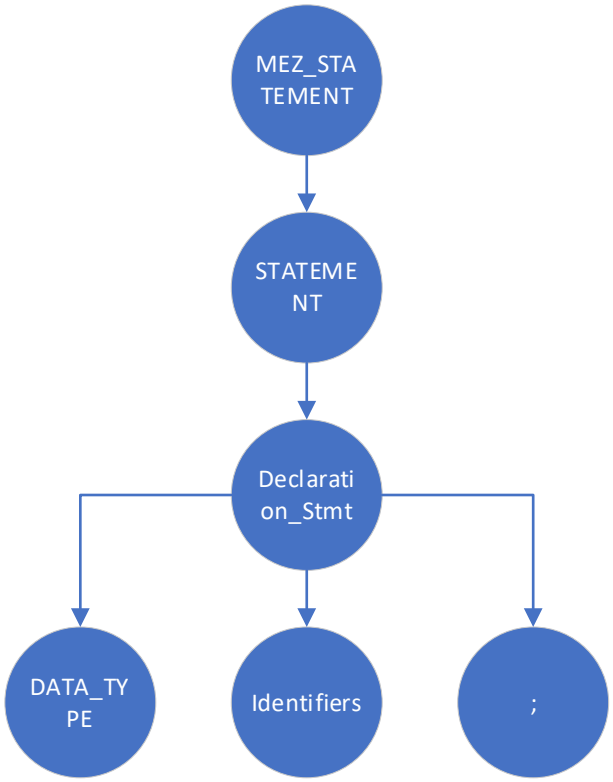


Declaration Statement

Num x;

(<DATA_TYPE> <id> ;)

<Declaration_Stmt> -> <DATA_TYPE> <id> ;


MEZ_STATEMENT
STATEMENT
Declaration_Stmt
DATA_TYPE
Identifiers
;

SEMANTIC RULES

1. Input Statement

Data Type should match

```
Num x ;
scan( Str , x );  <- Data type mismatch
scan( Num , x); <- Accepted
```

Variable should be declared

```
scan (Num , x); <- Variable “x” not declared
```

2. Output Statement

Variable should be declared

```
Num x;
print(“Hello”, name); <- Variable “name” not declared
```

3. Assignment Statement

Data Type should match

```
Str b;
b = 1; <- Data type mismatch
```

Variable should be declared

4. Condition Statement

Data Type should match

```
Str a;
if( a ==  2 ){ <- Data type mismatch
        2+2;
}
```

Variable should be declared

```
Num b;
if( b ==  2 ){
        c=c+1; <- Variable “c” not declared
}
```

5. Iterative Statement

Data Type should match

```
Str s = “”;
Num x = 0;
for ( s = 0 ; s < 10; s = s + 1 ) {  <- Data Type mismatch
x = x + 2;
```

}

Variable should be declared

```
Num x = 0;
for ( i = 0 ; i < 10; i = i + 1 ) {  <- variable “i” not declared
x = x + 2;
}
```

6. Declaration Statement

Data Type should match

```
Str s = 12; <- Data Type mismatch
```

Variable should be declared

```
print(x); <- variable “x” not declared
```

Only single declarations allowed

```
Num x , x; <- Double declaration
```

Warning: Variable declared but not used

Num x; <- variable “x” declared but not used

INTERPRETER

The interpreter would work as a command line program. By passing an input file with the MEZ extension, the exe file will pass it through the following modules:

Lexical analyzer

Syntax analyzer

Semantic analyzer

After passing the final module, equivalent C code will be made and executed by the exe file itself, showing input and output prompts on the command line.