|  |
| --- |
| The MOE Programming Language |
| Mark Jayson B. Fuentes  Nathaniel Emberga  Marlene Joyce Balajadia  Richard Glen B. Domingo  Dominique Vecina |

# Introduction

MOE is an educational experimental programming language. It has been inspired by the tactful minimalism of the C programming language, assembly (as for the short, albeit abbreviated reserved words), Perl, and a bit of creativity. Primitive in its nature, the MOE programming language attains to be a complete programming language to be used in practical and elementary programming. As a typed language, it also follows strict but natural syntax with recommendation on the implementation of structure and readability.

MOE provides scope, control structures, function and iteration. Having a built in input/output, a natural process for looping and decision making, an unambiguous assignment operator (‘:’ instead of ‘=’), every statement is straightforward.

Being a C-like programming language, identifiers are case-sensitive and blocks are indicated by curly brackets. The program logic starts in an up-down manner, and programs are executed first in the run() function (like C’s main()), therefore, a run() function is always required.

MOE aims to be a kick-starter for elementary level programmers where basic knowledge of programming is required. As MOE is a minimal language, it is easier to learn by beginners because of the limited and strict syntax of the code, but powerful enough to implement algorithms into.

Technically, a MOE program is interpreted by a fellow C compiler in which the system is mostly based on. MOE has no external libraries as all operations are done in an interpreter.

# The Syntax Elements

Character Sets

The character set or the alphabet of the MOE Programming language, like all programming language, is composed of the set of the English alphabets, set of numbers and the set of other specific special characters. To explain further:

Character set is {x|x is an є of the set of English Alphabets, an є of set of Numbers, an є of set Z}

Where

{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, 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}

English Alphabets =

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

Z = {+, -, \*, /, %, <, >, =, ?, |, &, ~, !, :, /, (, ), {, }, ;, ,}

Identifiers

Rules of identifiers in MOE Programming Language is the same as most PLs. MOE identifiers are not allowed to contain special characters except underscore ( \_ ). An identifier, though may contain numbers, is not allowed to start with it.

# Valid Identifiers

num\_Absents , a12, \_fileHold ped\_

# Invalid Identifiers

9gag percent% thisDot. ^smiles^

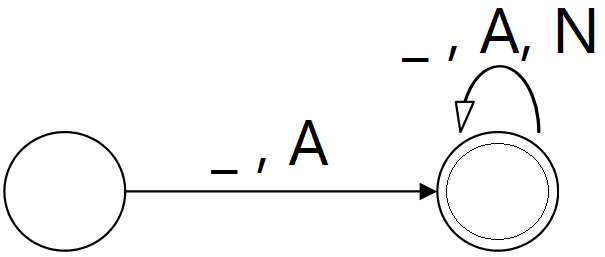
Here is the Finite Machine to simulate acceptable identifiers:

Let A be the set of all English Alphabets, including Upper and Lower cases

A = { 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, 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}

Let N be the set of all numbers

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



( \_ + A )+ (A + N)\*

Operation Symbols

Operation symbols are divided into subparts. Namely, Arithmetic operation symbols, Logical operation symbols and Relational operation symbols. Still, there are a number of symbols MOE regard as special and functions as an integral part of the language.

|  |  |
| --- | --- |
| Arithmetic operation symbols | Definition |
| + | Addition |
| - | Subtraction |
| \* | Multiplication |
| / | Division |
| % | Modulus division |
| ~ | Truncated division |
| ^ | Exponent |

|  |  |
| --- | --- |
| Relational operation symbols | Definition |
| < | Less than |
| > | Greater than |
| <= | Less than OR equal to |
| >= | Greater than OR equal to |
| = | Equal to |

|  |  |
| --- | --- |
| Logical operation symbol | Definition |
| | | OR |
| & | AND |
| ! | NOT |

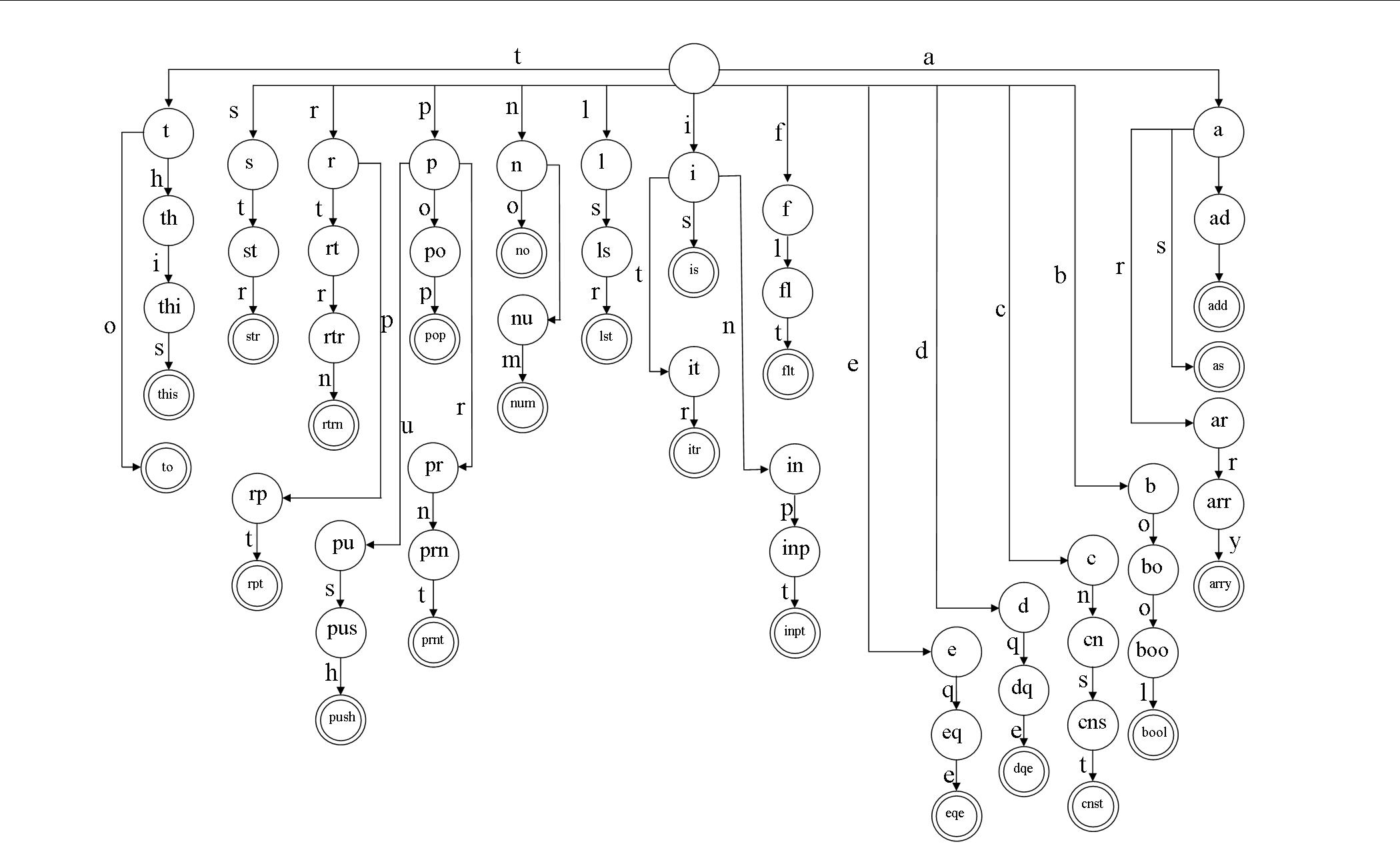
The colon symbol ( : ) , will take the place of an **assignment operator** instead of the usual equal sign ( = ).

Reserved Words

These are the set of words implemented inseparable with MOE and serves specific functions.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| add | cnst | flt | no | push | this |
| arry | dqe | inpt | num | rpt | to |
| as | eqe | is | pop | rtrn | true |
| bool | **flse** | **lst** | **prnt** | **str** |  |

At the point of reading a space character before the final state, the word or lexeme unit would be recognized as a **valid identifier.** Below is the Finite Machine, to recognize the set of reserved words.



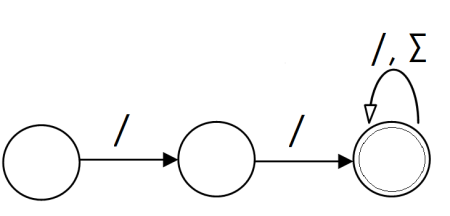
Noise Words

The only noise word in MOE programming language is the word ‘*end*’ which can appear anywhere. This noise word can improve the readability of the program marking an end to a set of statements

Comments

Any characters after the symbols, //, are ignored by the compiler; they are often used to make the source code easier to read. A comment can appear anywhere a blank, a tab, or a newline exists. As it was mentioned earlier, MOE aims to be a foundation of programming skills for elementary programmers thus, the elimination of block comments. Line comments can only be terminated by a new line. To recognize which lexeme unit is a comment, the lexical analyzer uses this Finite State Machine:

Let Σ, as mentioned in the *Character Set,* be the set of all acceptable characters in MOE programming language



// ( Σ )\*

Blanks and Spaces

Whitespaces can serve as an end of a unit and for the recognition of a certain token. For example, to recognize the keyword *num,* a series of keystroke will be pressed and must include the whitespace (“ “).

Input n|

*In this example, the keyword num, isn’t fully recognized as the unit it signifies until the spacebar is pressed.*

n

Input u

nu|

Input m

num|

Spacebar pressed

num |

But not at all times are whitespaces really necessary. Some tokens can be separated with the use of operators. The statement

num a;

a : 1 + 3\*4;

num b;

can also be properly recognized by

num a;a:1+3\*4;num b;

but the latter statements would be harder to read by the programmer. The last statement however requires a space character after the letter *m* in order to recognize the declaration of the *num* data type *b*. Otherwise, an identifier *numb* would be recognized.

|  |  |
| --- | --- |
| Whitespace symbols | Definition |
| ‘ ‘ | Single space character |
| ‘\n’ | Newline character |
| ‘\t’ | Tab character |

Delimiters and Brackets

These are the set of characters used to separated lexeme units, create *grouped statement*s or control *precedence* of expression evaluations.

Grouped statements starts with a curly bracket ( { ), followed by the bundled set of statements, each separated by the semicolon ( ; ). And then ended with a close curly bracket ( } ). The statements inside the brackets are executed as one single statement ( if it enters a region, all statements in that region must be executed respectively, unless encountering a break statement).

In the precedence of expression evaluation, PMDAS is still the rule in arithmetic operations but the availability of open ( ( )and close ( ) ) parenthesis in expressions provide readability and/or control over the mathematical equations such as forcing to operate additions before multiplication i.e. (2 +4) \*12.

|  |  |
| --- | --- |
| Brackets and Delimiters Symbols | Definition |
| ; | Statement terminator |
| ? | Condition terminator |
| { | Block start |
| } | Block end |
| ( | Parenthesis start |
| ) | Parenthesis end |
| ‘ | String literal |
| , | Comma |

Field Format

MOE programming language uses a *free field format***,** which means that lexeme units can appear and will be recognized even if it is separated by whitespaces, in **exception** with string literals. The string literal operator ( ‘ ) must read its ending pair within the same line, or else, the unit will not be recognized.

Expressions

Expressions are the basic building blocks of every programming language which represents or returns a value after an evaluation. *Arithmetic expressions* in MOE programming must be a single number constant or must contain at least two operands and one arithmetic operator since it cannot operate unary expressions, pre-increment and post-increment which only requires one operand and one operator. *Relational expressions* and *Logical expressions* are used relatively. They are also known as *Boolean expressions*. Relational expressions must contain two operands and one relational operator and returns a value of **true** or **false** after evaluation.A logical expression consists of one or more logical operators and logical or relational operands.

The unambiguous BNF grammar for arithmetic expressions:

<aexp> ::= <md> | <aexp>+<md> | <aexp>-<md>

<md> ::= <pow> | <md>\*<pow> | <md>/<pow> | <md>%<pow> | <md>~<pow>

<pow> ::= <aterm> | <pow>^<aterm>

<aterm> ::= <numcnst> | <identifier> | (<aexp>)

This BNF for arithmetic expressions follows the PEMDAS rule for mathematical evaluation. Operations which have higher precedence appear in the lower part of their parse trees.

The unambiguous BNF grammar for logical expressions:

<lexp> ::= <lterm> | !<lexp> | <lexp>|<lterm> | <lexp>&<lterm>

<lterm> ::= <booleanliteral> | <identifier> | (<lexp>) | (<rexp>)

The order of precedence of logical operator will be based solely on their positions. Logical operators on the leftmost position will have the highest precedence.

The unambiguous BNF grammar for relational expressions:

<rterm> < <rterm> | rterm> <= <rterm>

|<rterm> > <rterm> | rterm> >= <rterm>

|<rterm> = <rterm>

<rexp> ::=

<rterm> ::= <identifier> | <constants> | <literals>

Statements

MOE programming language consists of the basic programming statements

* Declaration Statements
* Assignment Statements
* I/O Statements
* Conditional Statements
* Control Statements

Declaration Statements  
In MOE, all variables are declared before they are used, usually at the beginning of a function before any executable statements. A declaration announces the properties of variables; it consists of a name and a list of variables such as

num fahr, celsius;  
num lower, upper, step;

Input and Output Statements  
If it was tricky in C, pretty easy in Java and C#, and laidback in Perl, PHP, strings, input and output are effortless in the MOE programming language. As an elementary, minimalistic programming language, there is a need to have a native support for these equally important elements.

run() {  
str answer;   
inpt answer;   
prnt answer;   
}

A larger example of the I/O statements is shown below:

// this program will demonstrate the division of two real numerals  
run() {  
flt divisor, dividend, quotient; // declaration of the  
// operands  
prnt ‘Input the dividend: ’;  
inpt dividend; // indented to clarify  
// prompts  
prnt ‘Input the divisor: ’;  
inpt divisor;  
quotient : dividend / divisor; // divides the dividend to   
// the divisor and stores it  
// in the variable quotient  
prnt quotient; // print to console  
}

Assignment  
Computation in the temperature conversion program begins with the assignment statements:

lower : 0;  
upper : 300;  
step : 20+10;  
  
which set the variables to their initial values. It is important that variables are to be initialized (except in a string variable) before they are used. Individual statements are terminated by semicolons.

But these statements are the same as most pl’s except in control statements and looping structure

# Conditional statements

is <condition> ?

<statement>;

OR

is <condition> ? {

<statement1>;

<statement2>;

…

}

OR

is <condition> ? {

<statement1>;

<statement2>;

…

} no {

<statement1>;

<statement2>;

…

}

OR

is <condition> ? {

<statement1>;

<statement2>;

…

} no is <condition> ? {

<statement1>;

<statement2>;

…

}

# Loop statements

# rpt is <condition> ?

<statement>;

OR

rpt is <condition> ? {

<statement1>;

<statement2>;

…

}

OR

rpt is <condition> ? {

<statement1>;

<statement2>;

…

} no {

<statement1>;

<statement2>;

…

}

OR

rpt is <condition> ? {

<statement1>;

<statement2>;

…

} no is <condition> ? {

<statement1>;

<statement2>;

…

}

rpt is <condition> ? {

<statement1>;

<statement2>;

…

} no is <condition> ? {

<statement1>;

<statement2>;

…

}

# The Lexical Approach

Lexical analysis is the process of converting set of characters into valid tokens. The lexical analyzer takes a text file as an input and produces a symbol table, or the set of tokens and lexemes, as an output. The lexemes are the smallest recognized units from the text file and the tokens are their equivalent category name or simply their name. Lexical analyzer can also tokenize an unrecognized unit as an error, which can serve as a first phase in catching errors within programming.

# Table of Lexemes and Tokens

|  |  |
| --- | --- |
| Lexeme | Token |
| add | ADD\_TO\_LIST |
| as | ITR\_AS |
| arry | ARRY\_TYP |
| bool | BOOL\_TYP |
| cnst | CONSTANT |
| dqe | DEQ\_LIST |
| eqe | ENQ\_LIST |
| flt | FLT\_TYP |
| flse | FALSE |
| is | IF\_ |
| inpt | INPT\_DATA |
| lst | LIST\_TYPE |
| no | ELSE\_ |
| num | INT\_TYP |
| pop | POP\_LIST |
| push | PUSH\_LIST |
| prnt | PRNT\_DATA |
| rpt | LOOP\_THIS |
| rtrn | RETURN\_VAL |
| str | STR\_TYP |
| this | THIS |
| true | TRUE |
| ( \_ + A )+ (A + N)\* | ID |
| // ( Σ )\* | COMMENT |
| ‘(Σ)’ | STRING\_LITERAL |
| [0-9]+ | INTEGER |
| [0-9]+. [0-9]+ | FLOAT |

This is the complete set of tokens which will be shown in the symbol table and will be used for grammar checking in the semantic analyzer. But before these lexemes are recognized and given an equivalent token, since MOE’s lexical analyzer approach is based on a combination of different *finite state machine,* thetext file is read character by character. Also, because of the character by character reading, **slowness of the total runtime of the lexical analyzer cannot be denied.** Still, the correctness of the program is **rest assured**.

|  |  |
| --- | --- |
| Lexeme | Token |
| + | ADD |
| - | SUB |
| \* | MUL |
| ^ | RAISE |
| / | DIV |
| % | MOD |
| ~ | TRUNC\_DIV |
| , | COM |
| | | OR |
| & | AND |
| ! | NOT |
| ? | COND\_END |
| : | ASSIGN |
| ; | STMNT\_END |
| { | B\_START |
| } | B\_END |
| ( | P\_START |
| ) | P\_END |
| < | LESSTH |
| > | GRTRTH |
| = | EQL |
| <= | LESSEQL |
| >= | GRTREQL |
| != | NOTEQL |

As was mentioned earlier, MOE’s lexical analyzer is a combination of different FS machines namely:

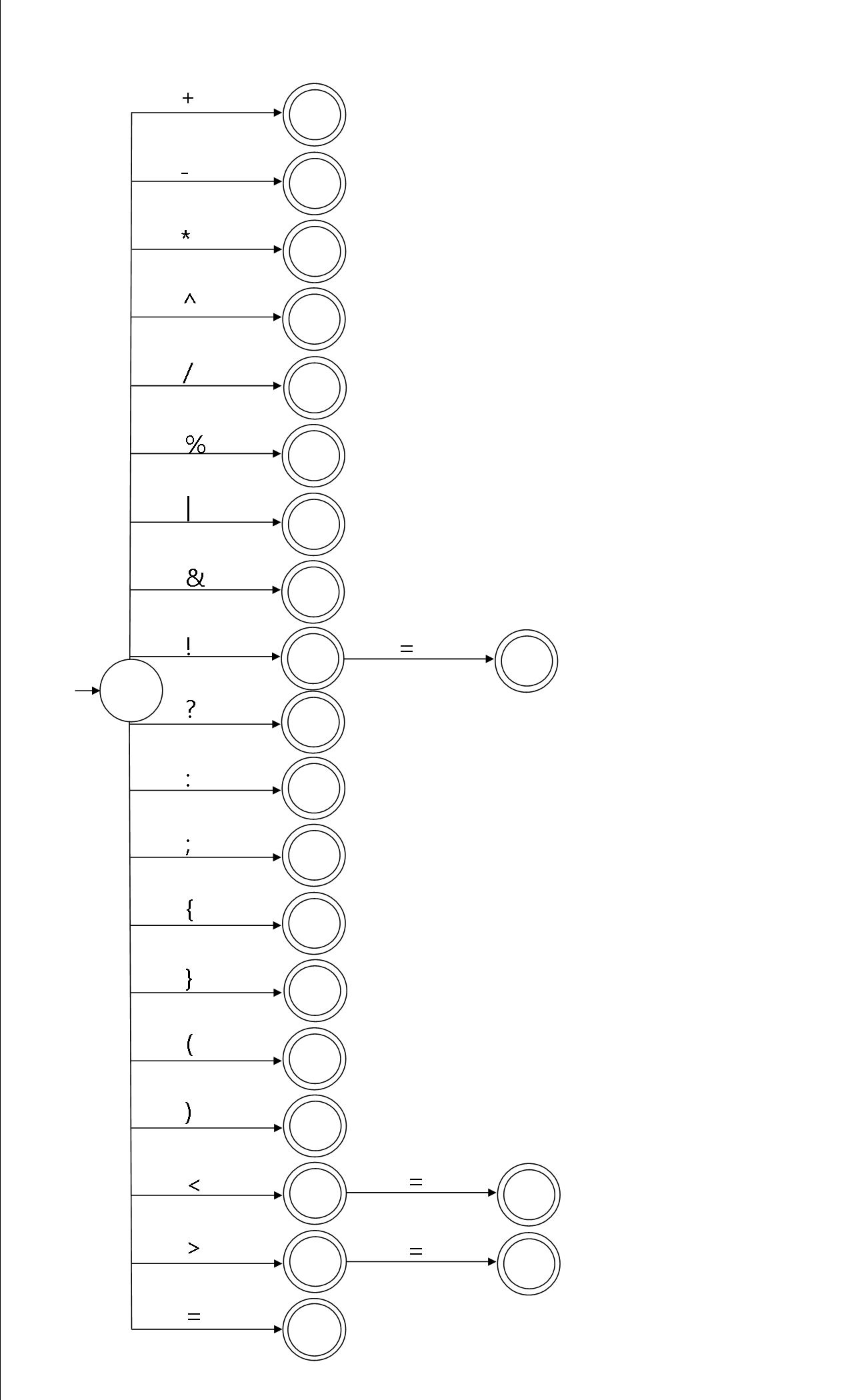
* Comment machine (defined)
* String literal machine
* Operator machine
* Number constant machine
* Keyword machine (defined)
* Identifier machine (defined)

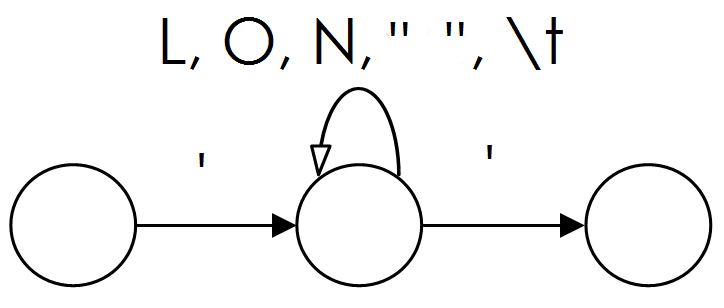
Let, L be the set of English letters

O be the set of operators

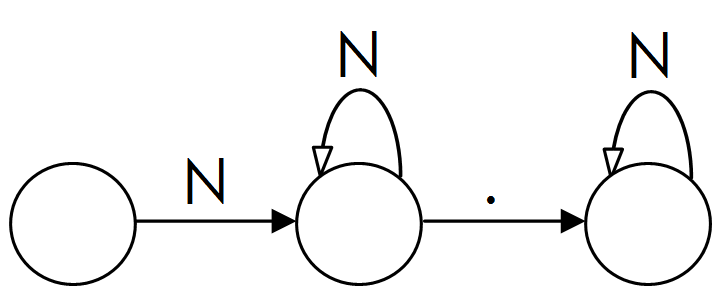
N be the set of numbers 0-9

W be the set of whitespaces





String literal machine



Number constant machine

Operator machine

Thus, combining all of these machines produces an ε-NFA and is illustrated as:

This is a summarization of the combination of the machines where, the rounded rectangle is the machine itself. The circle at the left side of the rectangle is the start state and the circle at the right side is the set of all final states. The whole specific process of each machine is abstracted. The circle in the middle of the keyword machine is the set of null states which reads a letter not in the state transition of the keyword machine, assuring that a valid identifier will not be recognized as a valid keyword and vice versa. The identifier, keyword, comment and number constant lexemes will not be easily terminated unlike the string literal and operator because the latter contains already within their machine, a terminator.

W

Σ

W, O

W, O

W, O

Comment machine

String literal machine

Keyword machine

Identifier machine

Number constant machine

Operator machine

s

s

s

s

ε

s

ε

\n

s

s

ε

ε

ε

ε

ε

# Syntactic Analyzer

Now that the Lexical Analyzer is created, we are now able to tell the recognized words from not. Next is the Syntactic Analyzer which tells whether the recognized words are properly formed or not. MOE’s syntactic analyzer parses in top-down approach which means that every terminal in the rules has its own method. Also, top-down approach also requires the grammar to be **non-left recursive** and must pass the **Pairwise Disjointness Test**. Thus making the number of earlier defined rules of the grammar (of the Expression part), and rules yet to be defined, increase.

# Formal Definition of the Grammar

Let G = {V, T, P, S}

**Where**

{<program>, <statement\_list>, <statement>, <decl>, <assign>, <cond>, <cont>, <IO>, <vars>, <decterm>, <exp>, <aexp>, <lexp>, <rexp>, <md>, <pow>, <aterm>, <lterm>, <rterm>, <outlist>, <writemem>}

V =

{+, -, \*, /, %, <, >, =, ?, |, &, ~, !, :, /, (, ), {, }, ;, ,, run, ID, rpt, is, no, inpt, prnt, flt, num, str, bool, value}

T =

S = <program>

|  |  |
| --- | --- |
| Production Rules | |
| <program> | run(){<statement\_list>} |
| <statement\_list> | <statement> |
|  | <statement\_list><statement> |
| <statement> | <decl> |
|  | <assign> |
| <cond> |
| <cont> |
| <IO> |
| {<statement\_list>} |
| <decl> | flt<vars>; |
| num<vars>; |
| str<vars>; |
| bool<vars>; |
| <vars> | <decterm> |
| <decterm>,<vars> |
| <decterm> | ID |
| ID : value |
| ID : <decterm> |
| <assign> | ID:<exp> |
| <exp> | <aexp> |
| <lexp> |
| <rexp> |
| <cond> | is<lexp>?<statement> |
| is<lexp>?<statement>no<statement> |

|  |  |
| --- | --- |
| Production Rules(*continued)* | |
| <cont> | rpt<cond> |
| <IO> | inpt ID |
|  | prnt <outlist> |
| <outlist> | <writemem>+<outlist> |
| <writemem> |
| <writemem> | <exp> |
| value |

But this grammar cannot be used in a top-down approach because some of its rules are left recursive and it does not pass the **Pairwise Disjointness Test**. Not to mention, some rules are **ambiguous**. After removing the left recursive rules, doing the left factoring method and removing ambiguity, a new grammar is created.

# The New Grammar

Let G’ = {V’, T’, P’, S}

**Where**

{<program>, <S>, <SL>, <SLP>, <D>, <V>, <DT>, <DT1>, <DT2>, <A>, <AP><ex>, <exp>, <md>, <mdp>, <pow>, <powp>, <term>, <cnd>, <cndp>, <cnt>, <io>, <outlist>, <writemem>}

V’ =

{+, -, \*, /, %, <, >, =, ?, |, &, ~, !, :, /, (, ), {, }, ;, ,, run, ID, numconst, stringliteral, boolliteral, rpt, is, no, inpt, prnt, flt, num, str, bool, cnst}

T’ =

|  |  |
| --- | --- |
| Production Rules | |
| <program> | run(){<SL>} |
| <SL> | <S><SLP> |
| <SLP> | ε |
|  | <S><SLP> |
| <S> | <D> |
|  | <A> |
| <cnd> |
|  | <cnt> |
| <io> |
| {<SL>} |
| <D> | flt<V>;|cnst flt<V>; |
|  | num<V>;|cnst num<V>; |
| str<V>;|cnst str<V>; |
| bool<V>;|cnst bool<V>; |
| <V> | <dt> |
|  | <dt>,<V> |
| <dt> | ID:<ex> |
| <dt1> | ε |
|  | :<dt2> |

|  |  |
| --- | --- |
| Production Rules(*continued)* | |
| <dt2> | numcnst |
|  | stringliteral |
| boolliteral |
| <dt> |
| <A> | ID:<AP> |
| <AP> | <ex>; |
|  | <A> |
| <ex> | !<md><exp> |
|  | <md><exp> |
| <md> < <md> |
| <md> <= <md> |
| <md> > <md> |
| <md> >= <md> |
| <md> = <md> |
| <md> != <md> |
| <exp> | ε |
| +<md><exp> |
| -<md><exp> |
| |<md><exp> |
| &<md><exp> |
| <md> | <pow><mdp> |
| <mdp> | ε |
|  | \*<pow><mdp> |
| /<pow><mdp> |
| %<pow><mdp> |
| ~<pow><mdp> |
| <pow> | <term><powp> |
| <powp> | ε |
| ^<term><powp> |
| <term> | numcnst |
| boolliteral |
| ID |
| (<ex>) |
| <cnd> | is<ex>?<S><cndp> |
| <cndp> | ε |
|  | no<S> |
| <cnt> | rpt<cnd> |
| <io> | inpt ID ; |
|  | prnt <outlist> ; |
| <outlist> | <writemem> + <outlist> |
|  | <writemem> |
| <writemem> | stringliteral |
|  | <ex> |

This is the final grammar used in the recursive descent parser of the MOE programming language. There is no ambiguity in this grammar because there is always a single distinct production (or set of production) of producing the grammar’s language.

# Proving of unambiguous grammar

To prove the unambiguous grammar, it will be shown that the parse tree of left most derivation and right most derivation is the same in every possible combination. And we shall now parse a whole MOE program.

run() {

cnst num MAX\_VAL : 100;

num incr : 1;

num itrate, incr : 2, longer;

bool isPrime : flse;

rpt is incr < MAX\_VAL ?

{

itrate : 2;

is isPrime ?

{

prnt incr;

isPrime : flse;

}

no

prnt love;

incr : incr + 1;

}

//end

}

Left Most Derivation

<program>

run(){<SL>}

run(){<S><SLP>}

run(){<D><SLP>}

run(){cnst num <V> ;<SLP>}

run(){cnst num <dt> ;<SLP>}

run(){cnst num ID <dt1> ;<SLP>}

run(){cnst num ID : <dt2> ;<SLP>}

run(){cnst num ID : numcnst ;<SLP>}

run(){cnst num ID : numcnst ;<S><SLP>}

run(){cnst num ID : numcnst ;<D><SLP>}

run(){cnst num ID : numcnst ; num <V>;<SLP>}

run(){cnst num ID : numcnst ; num <dt>;<SLP>}

run(){cnst num ID : numcnst ; num ID<dt1>;<SLP>}

run(){cnst num ID : numcnst ; num ID : <dt2>;<SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst;<SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst;<S><SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst;<D><SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num<V>;<SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num<dt>, <V> ;<SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num<dt>, <dt>, <V> ;<SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID<dt1>, <dt>, <V> ;<SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, <dt>, <V> ;<SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID<dt1>, <V> ;<SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID :<dt2>, <V> ;<SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID : numcnst, <V> ;<SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID : numcnst, <dt> ;<SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID : numcnst , ID <dt1> ;<SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID : numcnst , ID ;<SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID : numcnst , ID ; <S><SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID : numcnst , ID ; <D><SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID : numcnst , ID ; bool <V>;<SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID : numcnst , ID ; bool <dt>;<SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID : numcnst , ID ; bool ID <dt1>;<SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID : numcnst , ID ; bool ID : <dt2>;<SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID : numcnst , ID ; bool ID : bool\_literal ;<SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID : numcnst , ID ; bool ID : bool\_literal ; <S><SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID : numcnst , ID ; bool ID : bool\_literal ; <cnt><SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID : numcnst , ID ; bool ID : bool\_literal ; rpt <cnd><SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID : numcnst , ID ; bool ID : bool\_literal ; rpt is <ex>?<S><cndp><SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID : numcnst , ID ; bool ID : bool\_literal ; rpt is <md> < <md > ?<S><cndp><SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID : numcnst , ID ; bool ID : bool\_literal ; rpt is <pow><mdp> < <md > ?<S><cndp><SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID : numcnst , ID ; bool ID : bool\_literal ; rpt is <term><powp><mdp> < <md > ?<S><cndp><SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID : numcnst , ID ; bool ID : bool\_literal ; rpt is ID <powp><mdp> < <md > ?<S><cndp><SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID : numcnst , ID ; bool ID : bool\_literal ; rpt is ID <mdp> < <md > ?<S><cndp><SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID : numcnst , ID ; bool ID : bool\_literal ; rpt is ID < <md > ?<S><cndp><SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID : numcnst , ID ; bool ID : bool\_literal ; rpt is ID < <pow><mdp> ?<S><cndp><SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID : numcnst , ID ; bool ID : bool\_literal ; rpt is ID < <term><powp><mdp> ?<S><cndp><SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID : numcnst , ID ; bool ID : bool\_literal ; rpt is ID < ID <powp><mdp> ?<S><cndp><SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID : numcnst , ID ; bool ID : bool\_literal ; rpt is ID < ID <mdp> ?<S><cndp><SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID : numcnst , ID ; bool ID : bool\_literal ; rpt is ID < ID ?<S><cndp><SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID : numcnst , ID ; bool ID : bool\_literal ; rpt is ID < ID ? { <SL> }<cndp><SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID : numcnst , ID ; bool ID : bool\_literal ; rpt is ID < ID ? { <S><SLP> }<cndp><SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID : numcnst , ID ; bool ID : bool\_literal ; rpt is ID < ID ? { <A><SLP> }<cndp><SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID : numcnst , ID ; bool ID : bool\_literal ; rpt is ID < ID ? { ID : <ex> ; <SLP> }<cndp><SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID : numcnst , ID ; bool ID : bool\_literal ; rpt is ID < ID ? { ID : <md><exp> ; <SLP> }<cndp><SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID : numcnst , ID ; bool ID : bool\_literal ; rpt is ID < ID ? { ID : <pow><mdp><exp> ; <SLP> }<cndp><SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID : numcnst , ID ; bool ID : bool\_literal ; rpt is ID < ID ? { ID : <term><powp><mdp><exp> ; <SLP> }<cndp><SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID : numcnst , ID ; bool ID : bool\_literal ; rpt is ID < ID ? { ID : numcnst <powp><mdp><exp> ; <SLP> }<cndp><SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID : numcnst , ID ; bool ID : bool\_literal ; rpt is ID < ID ? { ID : numcnst <mdp><exp> ; <SLP> }<cndp><SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID : numcnst , ID ; bool ID : bool\_literal ; rpt is ID < ID ? { ID : numcnst <exp> ; <SLP> }<cndp><SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID : numcnst , ID ; bool ID : bool\_literal ; rpt is ID < ID ? { ID : numcnst ; <SLP> }<cndp><SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID : numcnst , ID ; bool ID : bool\_literal ; rpt is ID < ID ? { ID : numcnst ; <S><SLP> } <cndp><SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID : numcnst , ID ; bool ID : bool\_literal ; rpt is ID < ID ? { ID : numcnst ; <cnd><SLP> } <cndp><SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID : numcnst , ID ; bool ID : bool\_literal ; rpt is ID < ID ? { ID : numcnst ; is <ex>?<S><cndp><SLP> } <cndp><SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID : numcnst , ID ; bool ID : bool\_literal ; rpt is ID < ID ? { ID : numcnst ; is <md><exp>?<S><cndp><SLP> } <cndp><SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID : numcnst , ID ; bool ID : bool\_literal ; rpt is ID < ID ? { ID : numcnst ; is <pow><mdp><exp>?<S><cndp><SLP> } <cndp><SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID : numcnst , ID ; bool ID : bool\_literal ; rpt is ID < ID ? { ID : numcnst ; is <term><powp><mdp><exp>?<S><cndp><SLP> } <cndp><SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID : numcnst , ID ; bool ID : bool\_literal ; rpt is ID < ID ? { ID : numcnst ; is ID <powp><mdp><exp>?<S><cndp><SLP> } <cndp><SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID : numcnst , ID ; bool ID : bool\_literal ; rpt is ID < ID ? { ID : numcnst ; is ID <powp><mdp><exp>?<S><cndp><SLP> } <cndp><SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID : numcnst , ID ; bool ID : bool\_literal ; rpt is ID < ID ? { ID : numcnst ; is ID <mdp><exp>?<S><cndp><SLP> } <cndp><SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID : numcnst , ID ; bool ID : bool\_literal ; rpt is ID < ID ? { ID : numcnst ; is ID <exp>?<S><cndp><SLP> } <cndp><SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID : numcnst , ID ; bool ID : bool\_literal ; rpt is ID < ID ? { ID : numcnst ; is ID ?<S><cndp><SLP> } <cndp><SLP>}

run(){cnst num ID : numcnst ; num ID : numcnst; num ID, ID : numcnst , ID ; bool ID : bool\_literal ; rpt is ID < ID ? { ID : numcnst ; is ID ? { <S><SLP> }<cndp><SLP> } <cndp><SLP>}

( TO BE CONTINIUED)

Right Most Derivation

<program>

run(){<SL>}

run(){<S><SLP>}

run(){<S><S><SLP>}

run(){<S><S><S><SLP>}

run(){<S><S><S><S><SLP>}

run(){<S><S><S><S><S><SLP>}

run(){<S><S><S><S><S>}

run(){<S><S><S><S><cnt>}

run(){<S><S><S><S>rpt <cnd>}

run(){<S><S><S><S>rpt is<ex>?<S><cndp>}

run(){<S><S><S><S>rpt is<ex>?{<SL>}}

run(){<S><S><S><S>rpt is<ex>?{<S><SLP>}}

run(){<S><S><S><S>rpt is<ex>?{<S><S><SLP>}}

run(){<S><S><S><S>rpt is<ex>?{<S><S><S><SLP>}}

run(){<S><S><S><S>rpt is<ex>?{<S><S><S>}}

run(){<S><S><S><S>rpt is<ex>?{<S><S><A>}}

run(){<S><S><S><S>rpt is<ex>?{<S><S>ID:<ex>;}}

run(){<S><S><S><S>rpt is<ex>?{<S><cnd>ID:<ex>;}}

run(){<S><S><S><S>rpt is<ex>?{<S><cnd>ID:<md><exp>;}}

run(){<S><S><S><S>rpt is<ex>?{<S><cnd>ID:<md>+<md><exp>;}}

run(){<S><S><S><S>rpt is<ex>?{<S><cnd>ID:<md>+<md>;}}

run(){<S><S><S><S>rpt is<ex>?{<S><cnd>ID:<md>+<pow><mdp>;}}

run(){<S><S><S><S>rpt is<ex>?{<S><cnd>ID:<md>+<pow>;}}

run(){<S><S><S><S>rpt is<ex>?{<S><cnd>ID:<md>+<term><powp>;}}

run(){<S><S><S><S>rpt is<ex>?{<S><cnd>ID:<md>+<term>;}}

run(){<S><S><S><S>rpt is<ex>?{<S><cnd>ID:<md>+numcnst;}}

run(){<S><S><S><S>rpt is<ex>?{<S><cnd>ID:<pow><mdp>+numcnst;}}

run(){<S><S><S><S>rpt is<ex>?{<S><cnd>ID:<pow>+numcnst;}}

run(){<S><S><S><S>rpt is<ex>?{<S><cnd>ID:<term><powp>+numcnst;}}

run(){<S><S><S><S>rpt is<ex>?{<S><cnd>ID:<term>+numcnst;}}

run(){<S><S><S><S>rpt is<ex>?{<S><cnd>ID:ID+numcnst;}}

run(){<S><S><S><S>rpt is<ex>?{<S>is<ex>?<S><cndp>ID:ID+numcnst;}}

run(){<S><S><S><S>rpt is<ex>?{<S>is<ex>?<S>no<S>ID:ID+numcnst;}}

run(){<S><S><S><S>rpt is<ex>?{<S>is<ex>?<S>no<IO>;ID:ID+numcnst;}}

run(){<S><S><S><S>rpt is<ex>?{<S>is<ex>?<S>no prnt<outlist>;ID:ID+numcnst;}}

run(){<S><S><S><S>rpt is<ex>?{<S>is<ex>?<S>no prnt<writemem>;ID:ID+numcnst;}}

run(){<S><S><S><S>rpt is<ex>?{<S>is<ex>?<S>no prnt<ex>;ID:ID+numcnst;}}

run(){<S><S><S><S>rpt is<ex>?{<S>is<ex>?<S>no prnt<md><exp>;ID:ID+numcnst;}}

run(){<S><S><S><S>rpt is<ex>?{<S>is<ex>?<S>no prnt<md>;ID:ID+numcnst;}}

run(){<S><S><S><S>rpt is<ex>?{<S>is<ex>?<S>no prnt<pow><mdp>; ID : ID + numcnst;}}

run(){<S><S><S><S>rpt is<ex>?{<S>is<ex>?<S>no prnt<pow>;ID:ID+numcnst;}}

run(){<S><S><S><S>rpt is<ex>?{<S>is<ex>?<S>no prnt<term><powp>;ID:ID+numcnst;}}

run(){<S><S><S><S>rpt is<ex>?{<S>is<ex>?<S>no prnt<term>;ID:ID+numcnst;}}

run(){<S><S><S><S>rpt is<ex>?{<S>is<ex>?<S>no prnt ID ; ID : ID+numcnst;}}

run(){<S><S><S><S>rpt is<ex>?{<S>is<ex>?{<SL>}no prnt ID ; ID : ID+numcnst;}}

run(){<S><S><S><S>rpt is<ex>?{<S>is<ex>?{<S><SLP>}no prnt ID ; ID : ID+numcnst;}}

run(){<S><S><S><S>rpt is<ex>?{<S>is<ex>?{<S><S><SLP>}no prnt ID ; ID : ID+numcnst;}}

run(){<S><S><S><S>rpt is<ex>?{<S>is<ex>?{<S><S>}no prnt ID ; ID : ID+numcnst;}}

run(){<S><S><S><S>rpt is<ex>?{<S>is<ex>?{<S><A>}no prnt ID ; ID : ID+numcnst;}}

run(){<S><S><S><S>rpt is<ex>?{<S>is<ex>?{<S>ID : <ex>}no prnt ID ; ID : ID+numcnst;}}

run(){<S><S><S><S>rpt is<ex>?{<S>is<ex>?{<S>ID : <md><exp>}no prnt ID ; ID : ID+numcnst;}}

run(){<S><S><S><S>rpt is<ex>?{<S>is<ex>?{<S>ID : <md>}no prnt ID ; ID : ID+numcnst;}}

run(){<S><S><S><S>rpt is<ex>?{<S>is<ex>?{<S>ID : <pow><mdp>}no prnt ID ; ID : ID+numcnst;}}

run(){<S><S><S><S>rpt is<ex>?{<S>is<ex>?{<S>ID : <pow>}no prnt ID ; ID : ID+numcnst;}}

run(){<S><S><S><S>rpt is<ex>?{<S>is<ex>?{<S>ID : <term><powp>}no prnt ID ; ID : ID+numcnst;}}

run(){<S><S><S><S>rpt is<ex>?{<S>is<ex>?{<S>ID : <term>}no prnt ID ; ID : ID+numcnst;}}

run(){<S><S><S><S>rpt is<ex>?{<S>is<ex>?{<S>ID : boolliteral}no prnt ID ; ID : ID+numcnst;}}

run(){<S><S><S><S>rpt is<ex>?{<S>is<ex>?{<IO> ;ID : boolliteral}no prnt ID ; ID : ID+numcnst;}}

run(){<S><S><S><S>rpt is<ex>?{<S>is<ex>?{prnt <outlist>;ID : boolliteral}no prnt ID ; ID : ID+numcnst;}}

run(){<S><S><S><S>rpt is<ex>?{<S>is<ex>?{prnt <outlist>;ID : boolliteral}no prnt ID ; ID : ID+numcnst;}}

run(){<S><S><S><S>rpt is<ex>?{<S>is<ex>?{prnt <writemem>;ID : boolliteral}no prnt ID ; ID : ID+numcnst;}}

run(){<S><S><S><S>rpt is<ex>?{<S>is<ex>?{prnt <ex>;ID : boolliteral}no prnt ID ; ID : ID+numcnst;}}

run(){<S><S><S><S>rpt is<ex>?{<S>is<ex>?{prnt <md><exp>;ID : boolliteral}no prnt ID ; ID : ID+numcnst;}}

run(){<S><S><S><S>rpt is<ex>?{<S>is<ex>?{prnt <md>;ID : boolliteral}no prnt ID ; ID : ID+numcnst;}}

run(){<S><S><S><S>rpt is<ex>?{<S>is<ex>?{prnt <pow><mdp>;ID : boolliteral}no prnt ID ; ID : ID+numcnst;}}

run(){<S><S><S><S>rpt is<ex>?{<S>is<ex>?{prnt <pow>;ID : boolliteral}no prnt ID ; ID : ID+numcnst;}}

run(){<S><S><S><S>rpt is<ex>?{<S>is<ex>?{prnt <term><powp>;ID : boolliteral}no prnt ID ; ID : ID+numcnst;}}

run(){<S><S><S><S>rpt is<ex>?{<S>is<ex>?{prnt <term>;ID : boolliteral}no prnt ID ; ID : ID+numcnst;}}

run(){<S><S><S><S>rpt is<ex>?{<S>is<ex>?{prnt ID;ID : boolliteral}no prnt ID ; ID : ID+numcnst;}}

run(){<S><S><S><S>rpt is<md><<md>?{<S>is<ex>?{prnt ID;ID : boolliteral}no prnt ID ; ID : ID+numcnst;}}

(To Be Continue)