**Grubby**

**Programming Language Proposal**

Prepared by:

**Bengco, Dana Kirstie C.**

**Chua, Jericka A.**

**Tablanza, Krys Roger P.**

**Waquez, Emerson T.**

BSCS 3 – 2

Prepared for:

**Sagum, Ria**

Principles of Programming Language

December 2020

1. **Introduction**

**Description**

Creating a mini language is one of most important part to learn more about the principles of programming language. The programmers understand that going deeper and learning more about the principles of programming language will help in terms of understanding the syntax and how programming languages run including its theory.

The language that the programmers will be making is going to use common language rules and set of attributes to keep the language coherent and familiar to the user. Although the language is modelled from an existing language, the programmers intend to make it a properly functioning one. The mini programming language that we will make will be based on the existing programming language and it is C and Java but with our own twist and rules for syntax. The mini programming language will have limitations, but it can program and run like a normal programming language.

**Inspiration**

Our group decided that we would be modeling our language based on Java language because most of our subjects this year use Java. Our language name is called Grubbi, which came from a bot on discord called Groovy which plays songs while on a call in a discord server where our group meets.

The programmers intend to take advantage of the learning opportunity about programming languages, specifically C and Java. They intend to make themselves more familiar with Syntax Elements of C and Java that can prove to be useful by giving the programmers a perspective on how a programming language works. It can also benefit by sharpening their skills in terms of problem solving and logical thinking that can be applied while coding. Lastly, the programmers want to discover and learn new things while overcoming new challenges along their way.

1. **Syntactic Elements of Language**
2. **Character Set**

CharacterSet = {Uppercase, Lowercase, Digits, Symbols}

Uppercase = {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}

Lowercase = {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 = { +, - , \* , /, ^, &, |, !, >, <, =, {, }, [, ], (, ), “, ‘, \, ;}

1. **Identifiers**

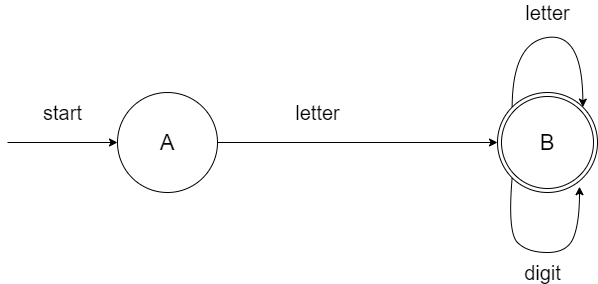
Rules in naming identifiers:

* + An identifier should start with a letter and can either be uppercase or lowercase
  + A valid identifier should contain letters or digits
  + Words that fall into the category of being a reserve word or a keyword should not be used as an identifier
  + Identifiers are case sensitive
    - E.g., GROOVY, Groovy, and groovy are three different identifiers

**Regular Expression**

|  |  |
| --- | --- |
| letter | = [a - zA - Z] |
| digit | = [0 - 9] |
| identifier | = letter (digit\* | letter\*)\* |
|  |  |

**Machine for the Identifier**

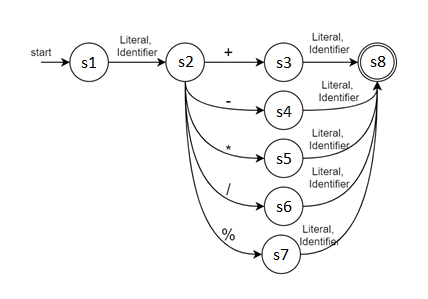


1. **Operation Symbols**
   1. **Arithmetic Operators**

|  |  |  |
| --- | --- | --- |
| **Operator** | **Operation** | **TOKEN** |
| + | Addition | PLUS |
| - | Subtraction | MINUS |
| \* | Multiplication | MULTI |
| / | Division | DIV |
| % | Modulo Operation (Remainder after division) | MOD |
| ^ | Exponential | EXP |

|  |  |  |
| --- | --- | --- |
| literal | = | digit\* |
| **Regular Expression** | = | (Literal | Identifier) (+ | - | \* | / | %| ^) (Literal | Identifier) |

**Machine for the Arithmetic Operators**

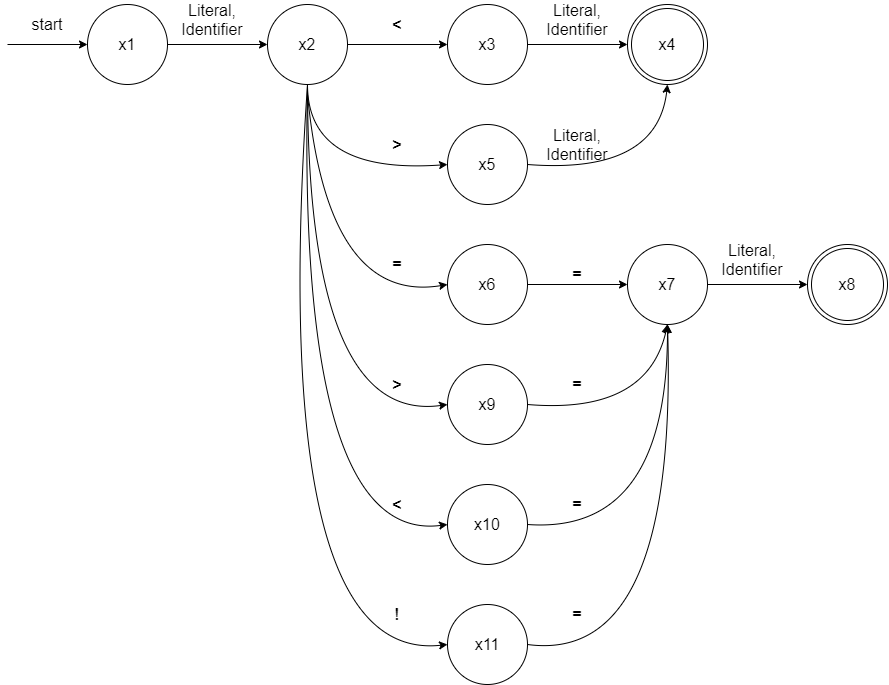


* 1. **Boolean Operators**
     1. Relational Operators

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **TOKEN** |
| == | is equal to | EQ |
| != | not equal to | NEQ |
| > | greater than | GT |
| < | less than | LT |
| >= | greater than or equal to | GTEQ |
| <= | less than or equal to | LTEQ |

|  |  |  |
| --- | --- | --- |
| **Regular Expression** | = | (Literal | Identifier) (< | > | (==) | (>=) | (<=) | (!=)) (Literal | Identifier) |

**Machine for the Boolean Operators**

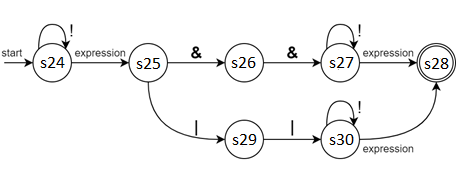


* + 1. Logical Operators

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Example** | **Meaning** | **TOKEN** |
| && (Logical AND) | expression1 && expression2 | true only if both expression1 and expression2 are true | AND |
| || (Logical OR) | expression1 || expression2 | true if either expression1 or expression2 is true | OR |
| ! (Logical NOT) | !expression | true if expression is false and vice versa | NOT |

|  |  |  |
| --- | --- | --- |
| **Regular Expression** | = | (!)? Expression ((&&) | (||)) (!)? (Expression)) |

**Machine for the Logical Operators**



* 1. **Assignment Operators**

|  |  |  |
| --- | --- | --- |
| **Operator** | **Example** | **TOKEN** |
| = | x = y | EQUALS |

|  |  |  |
| --- | --- | --- |
| **Regular Expression** | = | (Literal | Identifier) (= (Literal | Identifier) |

**Machine for the Assignment Operators**

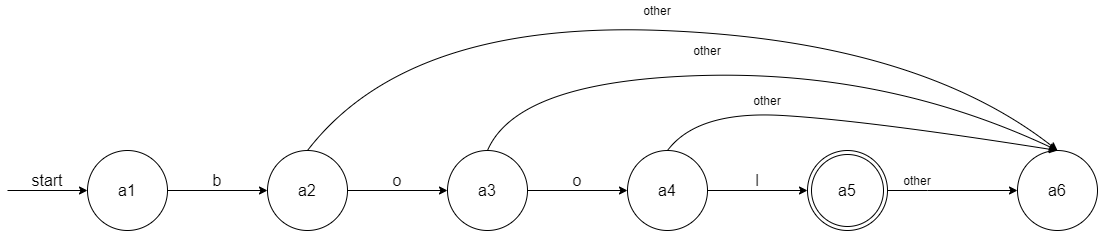


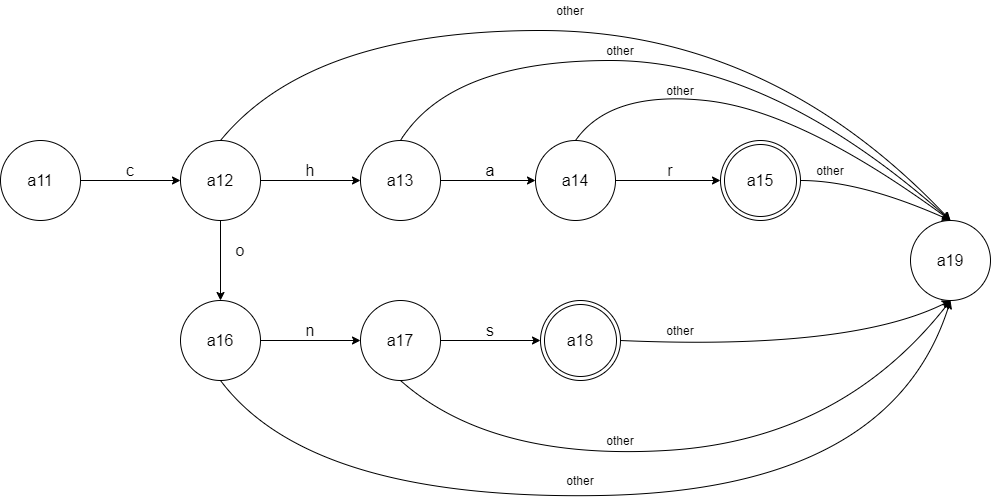
1. **Keywords and Reserved Words**

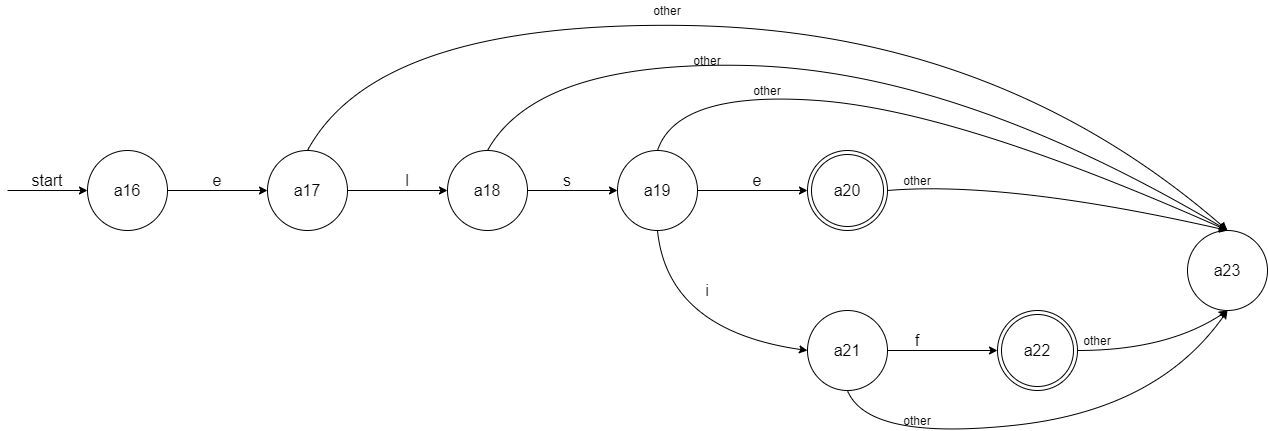
|  |  |  |
| --- | --- | --- |
| **Keywords** | **Description** | **TOKEN** |
| int | A data type that can store whole numbers from -2147483648 to 2147483647 | INT |
| char | A data type that is used to store a single character | CHAR |
| real | A data type that can store whole numbers from 3.4e−038 to 3.4e+038 | REAL |
| str | A data type that is used to store a string | STR |
| bool | A data type that can only store true and false values | BOOL |
| stop | Breaks out of a loop | STOP |
| if | Makes a conditional statement | IF |
| else | Used in conditional statements | ELSE |
| elsif | Used for multiple conditional statements | ELSIF |
| while | Creates a while loop | WHILE |
| length |  | LENGTH |
| true | Indicates that the statement is true | TRUE |
| false | Indicates that the statement is false | FALSE |
| **Reserved Words** | **Description** |  |
| input | Reads the input of a user; acts as the scanf command to the console | INPUT |
| output | Used to display the output. | OUTPUT |

|  |  |  |
| --- | --- | --- |
| **Regular Expression** | = | Letter (Letter | Digit)\* other RECONSTRUCT |

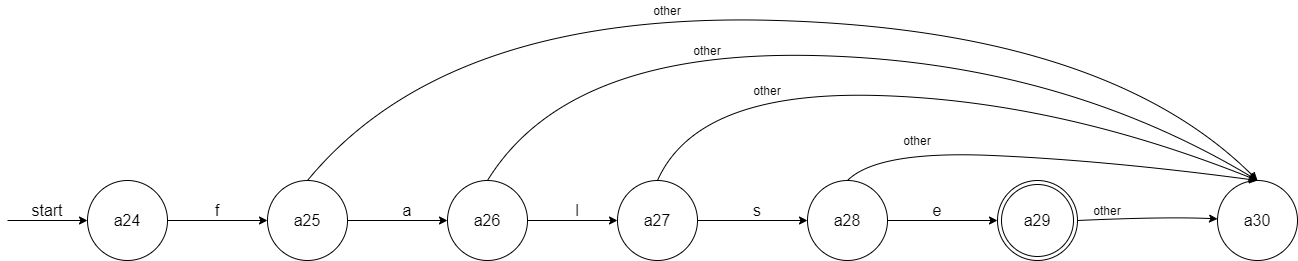
**Machine for Each Key Words**

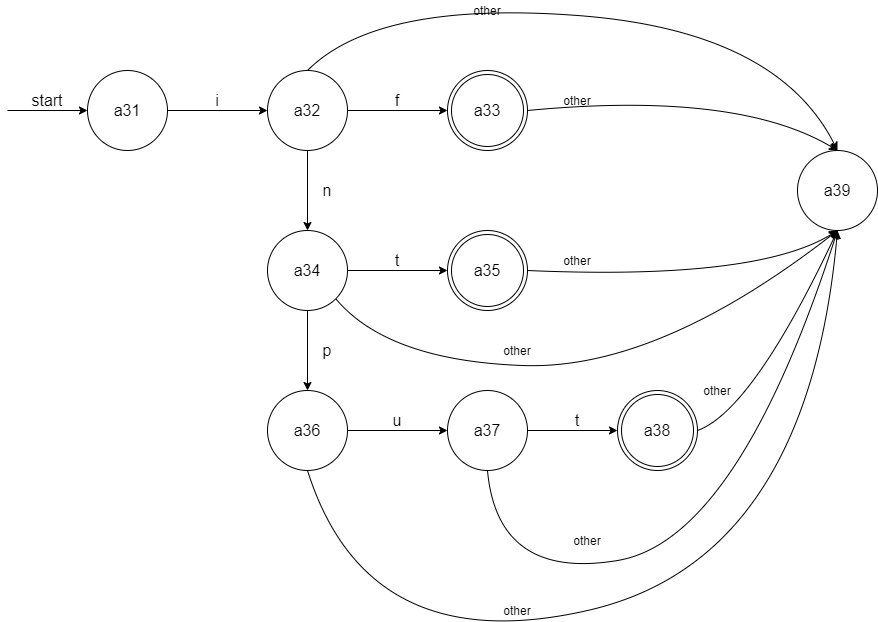
**bool** 

**char & cons**

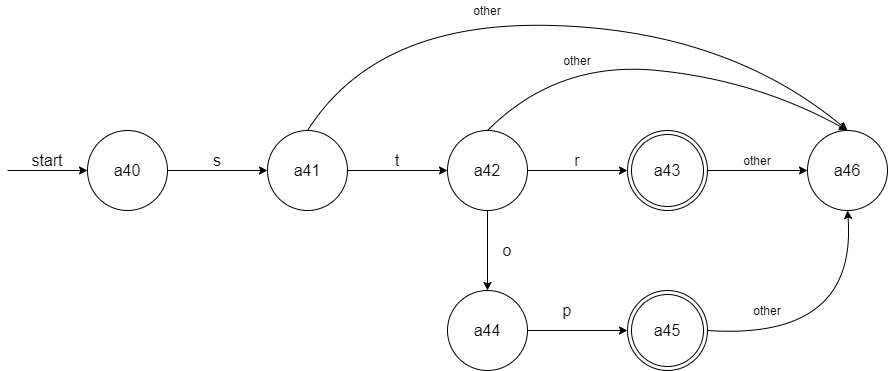
**else & elsif**

**false**

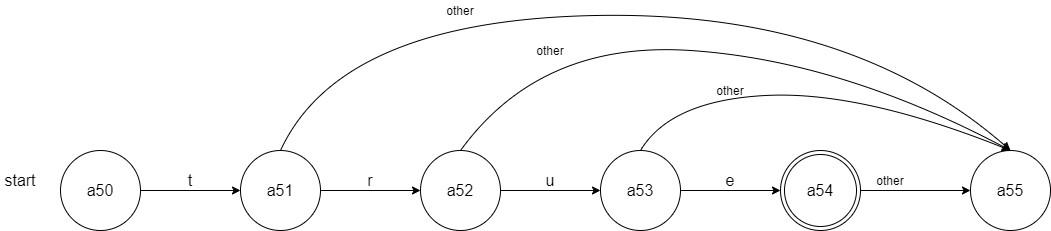
****

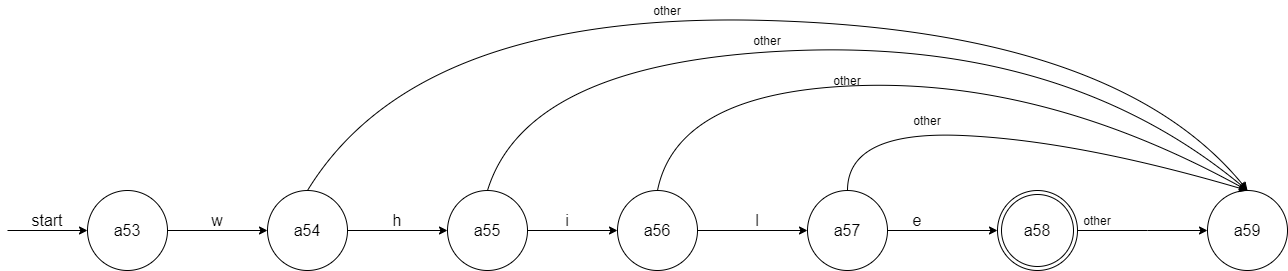
**if & int & input**

**str & stop**

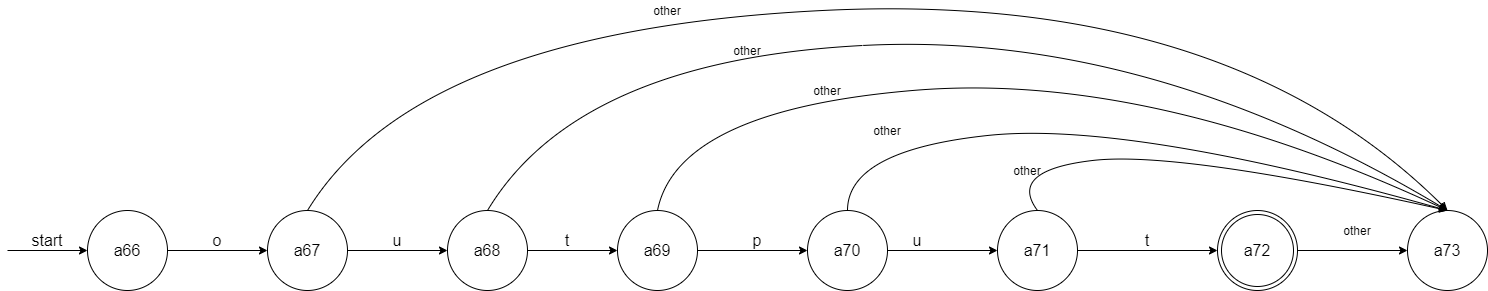
****

**true**

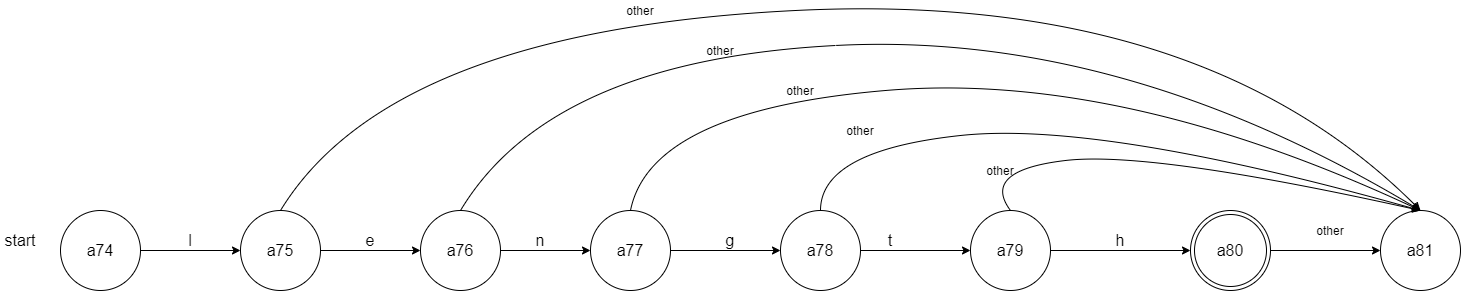
****

**while** 

**output**

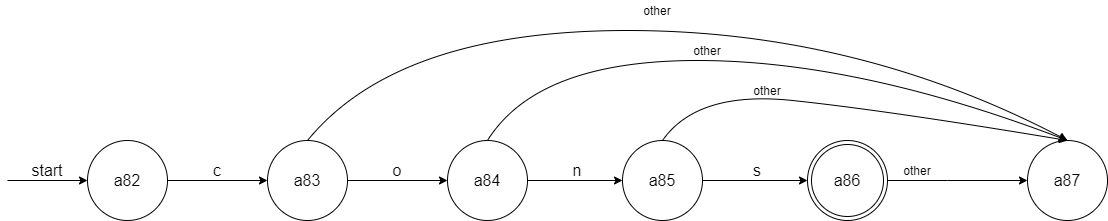
****

**length**

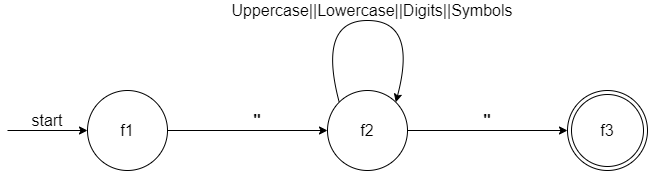
****

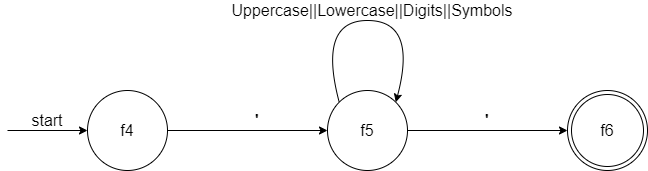
1. **Noise Words**

* Cons



**Constant Values**





**Regular Expression** = Letter (Letter | Digit)\* other

1. **Comments**

CharacterSet = {Uppercase, Lowercase, Digits, Symbols}

Uppercase = {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}

Lowercase = {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}

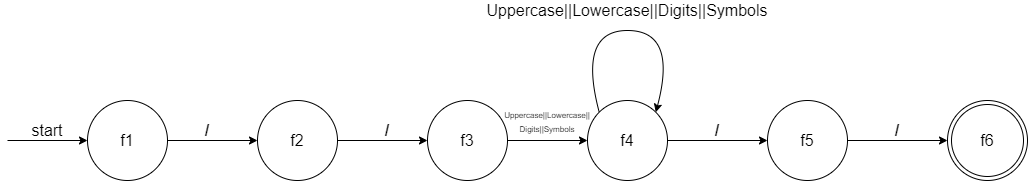
Digits = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

Symbols = { +, - , \* , /, ^, &, |, !, >, <, =, {, }, [, ], (, ), “, ‘, \, ;}

* Multiple-line Comments: // //

**Regular Expression** = (//) (Uppercase || Lowercase || Digits || Symbols)\* (//)

**Machine for the Multiple-Line Comments**



1. **Blank**

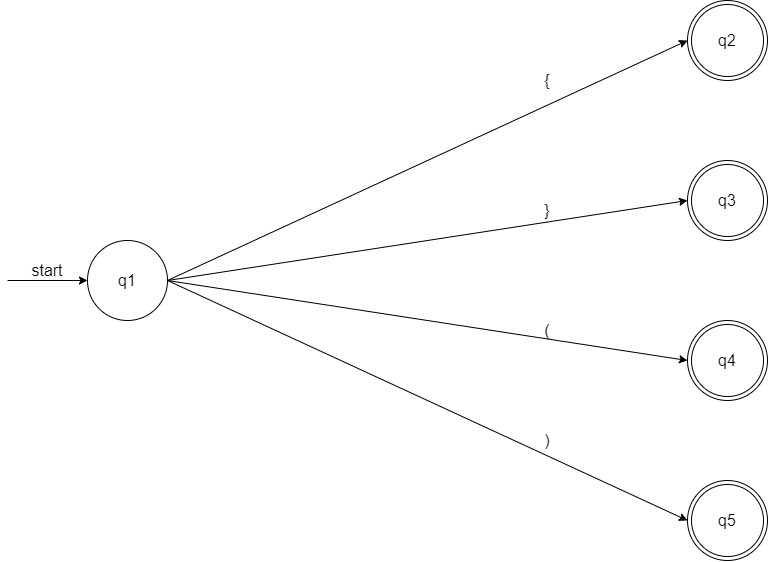
* White spaces are used to describe blanks and tabs
* White spaces are required between keywords and identifiers
* White spaces are used to separate keywords and identifiers
* ^ln **–** Used to create a new line TOKEN: NEWLINE
* **^**ht **– Used** to create tab TOKEN: TAB

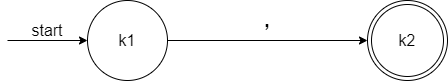
1. **Delimiters and Brackets**

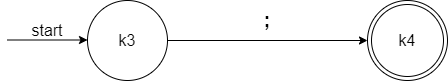
|  |  |  |
| --- | --- | --- |
| **Lexeme** | **Symbol Name** | **TOKEN NAME** |
| ( | open parentheses | LPRN |
| ) | close parentheses | RPRN |
| { | open curly brace | LBRC |
| } | close curly brace | RBRC |
| , | coma | COMA |
| ; | Semicolon | SEMI |

**Regular Expression =** {(,\* )\*}; | {[] {};};

**Machine for Delimiters and Brackets**







1. **Free-and-Fixed-Field Formats**

* Free-field format will be used in the programming language to allow program statements to be written anywhere on a line without regards to position on the line or to line breaks.

1. **Expressions**

**Rules in Evaluating Arithmetic Expressions**

1. For a statement to be evaluated to an arithmetic expression, it should consist of at least two operands and at least one operator.
2. The arithmetic operators are addition (+), subtraction (-), multiplication (\*), division (/) and remainder/modulus (%).
3. The operands could be a constant value, an identifier, an expression, and identifier in parenthesis, and/or an expression in parenthesis.
4. The use of open parenthesis (() as a grouping symbol for an expression is allowed as long as the expression is enclosed by a close parenthesis ()).
5. The order of arithmetic expressions follows the MDAS (Multiplication-Division-Addition-Subtraction) rule.
6. When it comes to the associativity of arithmetic expressions, certain evaluation is utilized.

**Rules in Evaluating Relational Expressions**

1. A relational expression returns a logical value of either TRUE or FALSE on evaluation and comparison of two arithmetic.
2. A relational expression can be only used in selection or conditional statements.
3. The values that are evaluated in the relational expression must be of the same type.
4. Relational operators "==" and "!=" have the same precedence.
5. Relational operators "<", "<=", ">", and ">=" have the same precedence but relational operators "==" and "!=" have lower precedence.
6. The associativity of evaluating relational operators with the same precedence is from left to right.

**Rules in Evaluating Logical Expressions**

1. A logical expression specifies a logical computation that returns a Boolean value of either TRUE or FALSE on evaluation and comparison of two arithmetic.
2. A relational expression can be only used in selection or conditional statements and also with Boolean values.
3. The logical operators in grubby is logical AND "&&", logical OR "||", or logical Not "!". A Logical OR and logical AND must have at least two (2) operands and at least one operator; Logical NOT requires only one operand.
4. Logical AND combines two boolean values and returns a boolean which is true if and only if both of its operands are true.
5. Logical OR combines two boolean variables or expressions and returns a result that is true if either or both of its operands are true.
6. Logical NOT return reverses the value of a boolean expression. Logical NOT and operand true returns false. Logical NOT and operand false returns true.
7. The precedence of logical operators in order is Logical NOT, Logical AND, and Logical OR.
8. In complex expressions with logical and relational operators, relational operators are evaluated first.
9. The associativity of evaluating logical operators with the same precedence is from left to right.

**Rules in Evaluating Assignment Expressions**

1. An assignment expression in grubby uses the assignment operator (=) to assign the result of an expression to a variable.
2. Assignment operator returns the value of the left operand.
3. Assignment operator cannot be used on any other expressions than the assignment expressions.
4. **Statements**
   1. Declaration Statements

Identifiers Declaration

Example:

int Gruvvy;

float gruvvy;

Identifiers Declaration with Initialization

Example:

char a = ‘b’;

int love = 143;

* 1. Input/Output Statement

Input Statement

Example:

input( Gruvvy1);

input(gruvvy2);

Output Statement

output(“Gruvvy Programming Language”);

output(“Welcome to our own Languange”);

* 1. Assignment Statements

Example 1:

Gruvvy1 = 99.05;  
 gruv2 = ‘a’

Gruvvy99 = “Programming”

* 1. Conditional Statements

Example:

if (gruv < 100) {

write (“Congrats the program is working well”);

}

Example 2:

if (sheep < 50) {

write (“the number of sheep in the cage is under control”);

}else {

write (“the number of sheep in the cage is overloaded”);

}

Example 3:

if (feb == 14) {

write (“Happy Valentines Day”);

}elsif (feb == 30){

write(“The correct person will come in the right time”);

* 1. Iterative Statement

Example:

while (person != 1) {

// you can put here any statements

}