Faculty of Computing and Information Technology

Department of Computer Science

CPCS302 - Compiler Construction

Project – Group#: 4

ID	Name
2010269	Hanin Suleiman Omer Alhaj
1905483	Dhay Mohammed Alharbi
1905536	Hanin Mohammed Alharbi
2005863	Roaa Ahduallah Alzahrani
2010304	Alaa Emad Alhamzi

Instructors: Dr. Sultanah M. Alshammari & I.Tahani S. Almoshadak

Table of Contents

Assignment task

ID	Name	Task
2010269	Hanin Suleiman Omer Alhaj	- Regular expressions of the tokens - Statements - Coding - JJTree
1905483	Dhay Mohammed Alharbi	- Regular expressions of the tokens - Statements - BNF & grammar explanation
1905536	Hanin Mohammed Alharbi	- Regular expressions of the tokens - Statements - Coding - JJTree
2005863	Roaa Ahduallah Alzahrani	- Regular expressions of the tokens - Statements - BNF & grammar explanation
2010304	Alaa Emad Alhamzi	- Regular expressions of the tokens - Statements - Coding

Phase 1

1.1 Introduction

Star is a high-level programming language created using JavaCC, based on a specific BNF. The main objective of the Star project is to develop a language that offers a wide range of statements for various expressions, including arithmetic, relational, conditional, logical, and iterative, as well as different data types and data structures.

To support the Star language, we have constructed a compiler. The compilation process comprises three phases: lexical analysis, syntactic analysis, and parse Tree. The first phase, lexical analysis, converts a sequence of characters into a set of lexical tokens. These tokens are then passed to the syntax analysis phase, which employs a parser to examine the input's syntactic structure and assess its compliance with the programming language's syntax. The parser generates a parse tree, subsequently used as input for the third phase, semantic analysis. In semantic analysis, the syntax tree and symbol table determine if the provided program is semantically consistent with the language definition.

1.2 Regular Expressions of Tokens

Token Name	Description	Regular Expression
Arithmetic Operation	Includes plus, minus, multiply, divide, reminder, increment, decrement, and assignment.	Plus: "*+" Minus: "*-" Multiply: "**" Divide: "*/"

		Reminder: "*%" Increment: "*++" Decrement: "*--" Assign: "*="
Relational operation	Includes equality, inequality, greater than, less than, greater than or equal, less than or equal.	Equal: "*==" NotEqual: "*=!" GreaterThan: "*<" LessThan: "*>" GreaterOrEqual: "*=<" LessOrEqual: "*=>"
Logical operators	Includes and, or, not	AND: "*&" OR: "*|" NOT: "*!"
Punctuation marks	Marks that are reserved in the language.	Semi – Colon: ";" Colon :":" Question Mark: "?" Exclamation-Mark: "!" Dot :"." L-Bracket: "(" R-Bracket: ")" Double_Qutation" " : " Dash"-" : Comma: "," L-CBracket: "{" R-CBracket: "}"
Identifiers	Must start with exclamation mark followed by combination of letters and digit	("!") (Letters |Digits)
Digits	Integer is at least one digit. Fraction is at least one digit followed by dot followed by at least one digit.	Integer: ["0"-"9"]+ Fraction: ["0"-"9"]+ "." ["0"-"9"]+
Letters	Contains at least one small or capital letters.	["a-z" ,"A-Z"]+
White Spaces	One or more of white spaces	(" ")+

Keywords	Set of reserved words(IF, THEN, ELSE, INTEGER, FRACTION, LETTER, TEXT, BOOLEAN, CONSTANT, CLASS, STATIC, PUBLIC, PRIVATE, RETURN, END, CONTINUE, PRINT, VECTOR, FOR, DO, WHILE, LOOP)	"if" | "then" | "else" | "int" | "frac" | "letter" |"bool"|"const" | "class" | "static" | "public" | "private" | "return" | "end" | "continue" | "print" | "vector" | "do" | "while" | "table" | "loop" |"break"
Comment	Comment start with (*%) and end with (*%)	"*%"(Letters |Digits| Punctuation marks) + "*%"
DATA_ STRUCT	Different structures to describe data: - Vector (1D array) - Table (2D array)	VECTOR: (vector)(!)(\w|\d) (*=) (\w|\d|,)+ Table: (table) (!)(\w|\d) (*=) ("{") (\w|\d|,)+ ("}")(,)*)+(
DataType	The data types are Integer, Letters, And Fraction	"int" "frac" "letter" "bool"
Constant	Start with "const" followed by data type followed by an identifier	("const") (DataType) (Identifiers)
Iterative statement	Start with "loop"followed digits then Stmts. more than one them "End"	("loop ") ((Digits)+): (Stmts.)+ END
Boolean Statement	Start with token bool followed by logical operators	("bool")(Identifiers) (Logical operators) (Boolean)
Print Statement	Start with "print" followed by angle brackets then letter or digit or punctuation marks	("print")(":") (')((Letters|Digits| Punctuation_Marks)+ (')
Boolean	Contains two values which are true and false	("true" | "false")

1.3 Statements

Statement Type	Code
Arithmetic Statement	!R *= 65 *+ 2.
Relational Statement	!X *= 18 *> 7.
Logical statement	!a *= !b *& !c.
Boolean statements	(true)(*|) (true)

Conditional statements	if-then:
	If (!x *> 5 :(print("Hi ") End.
	
	If-then-else:
	If (!y *== 7):
	print(“hello”)
	else:
	print(“error”).

Iterative Statement	print (" Stars :(Loop (x*<5 language ") end.
Variable declaration	x! *= 5. !y *= 5.5. !str *="Hi".
Constant	Const Integer !PI *= 3.14.
Data structure	**vector** !num *= 1,2,3,4. **Table** !num *= {1,1,1},{2,2,2}.

Phase 2

2.1 BNF

Start ➔ STMTS. | Comment

STMTS ➔Assignment| Constant | PrintSTMT | Conditional_Stmt | DataStructures | Iteration

Assignment ➔ Identifier *= STMT

Constant ➔ const (DataType)

PrintSTMT ➔ print (":")(" ' ") (STMT+)(" ' ")

Conditional_Stmt➔ If (Condition) (":") (STMTS+) End

Iteration ➔ Loop (Condition)(":") (STMTS.) End

Condition ➔ (Identifier | Digits)

(RelationalOperation| LogicalOperation) (Identifier | Digits)

STMT ➔ (Identifier | Letters | Digits) (ArithmeticStmt | LogicalStmt | RelationalStmt)?

ArithmeticStmt ➔ ArithmeticOperation (Identifier | Digits)

LogicalStmt ➔ LogicalOperation (Identifier | Digits)

RelationalStmt ➔ RelationalOperation (Identifier | Digits)

RelationalOperation ➔ *==|*!= |*> |*< | *>= | *<=

LogicalOperation ➔ *& | *| | *!

ArithmeticOperation ➔ *+ | *- | **| */ | *% | *++ |*--| *=

DataStructures ➔ Vector | Table

Vector ➔ vector Identifier *= (Digits | Letters)+ (, (Digits | Letters))*

Table ➔ table Identifier *= {(Digits | Letters)+ (, (Digits | Letters))*} , {(Digits | Letters)+ (, (Digits | Letters))*}

Identifier ➔ ! (Digits | Letters)+

DataType➔(intDeclare | FractionDeclare |LetterDeclare| BooleanDeclare)

intDeclare ➔ int Identifier *= Integer

FractionDeclare ➔ frac Identifier *= Fraction

BooleanDeclar ➔bool Identifier *= Boolean

LettersDeclare ➔ letter Identifier *= Letters

Comment ➔ (*%) (Digit | Letters | PunctuationMarks)+(*%)

PunctuationMarks ➔ (; | : | ?| ! | . |) | (| " | - | ,{|})

Digits ➔ Integer | Fractions

Integer ➔ ["0" - "9"] +

Fraction ➔ (["0" - "9"])+ . (["0" - "9"])+

Letters ➔ (["A"-"Z", "a"-"z"])+

Boolean ➔ ("true " | "false ")

2.2 BNF explanation

Start ➔ STMTS. | Comment

The "Start" symbol represents the starting point of the language. It can statements (STMTS) ending with a dot.It can also start with a comment rather than STMTS .

STMTS ➔ Assignment | Constant | PrintSTMT | Conditional_Stmt | DataStructures | Iteration

The "STMTS" non-terminal represents 6 types of statements in star language. It can be one of the seven types of statements.

Assignment ➔ Identifier *= STMT

The "Assignment" rule represents an assignment statement. It assigns a statement (STMT) to an identifier (Identifier) in the left-hand side (LHS) using the "*=" operator.

Constant ➔ const (DataType)

The "Constant" rule represents a constant declaration statement. It starts by the keyword "const" followed by the data type (DataType).

PrintSTMT ➔ print (":")(" ' ") (STMT)*(" ' ")

The "PrintSTMT" rule represents a print statement. It starts with the keyword "Print" followed by colon and sequence of digits, letters, or punctuation marks between 2 single quotations.

Conditional_Stmt➔ If (Condition) : (STMTS+) End

The "Conditional_Stmt" rule represents a conditional statement. It starts with the keyword "If" followed by a condition (Condition) enclosed in parentheses. The statements (STMTS) within the conditional block are enclosed in colons and can be repeated one or more times. The block is terminated with the keyword "End".

Iteration ➔ Loop (Condition): (STMTS.) End

The "Iteration" rule represents a loop statement. It starts with the keyword "Loop" followed by a condition (Condition) enclosed in parentheses. The statements (STMTS) within the loop block

are enclosed in colons and can be repeated one or more times. The block is terminated with the keyword "End".

Condition ➔ (Identifier | Digits) (RelationalOperation| LogicalOperation) (Identifier | Digits)

The "Condition" rule represents a condition used in conditional and loop statements. It consists of an identifier or digits followed by a relational operation or logical operation and another identifier or digits.

STMT ➔ (Identifier | Letters | Digits) (ArithmeticStmt | LogicalStmt | RelationalStmt)?

The "STMT" rule represents a general statement in the language. It can start with an identifier, letters, or digits and can optionally be followed by an arithmetic statement, logical statement, or relational statement.

ArithmeticStmt ➔ ArithmeticOperation (Identifier | Digits)

The "ArithmeticStmt" rule represents an arithmetic statement. It consists of an arithmetic operation followed by an identifier or digits.

LogicalStmt ➔ LogicalOperation (Identifier | Digits)

The "LogicalStmt" rule represents a logical statement in the language. It consists of a logical operation followed by either an identifier or digits.

RelationalStmt ➔ RelationalOperation (Identifier | Digits)

The "RelationalStmt" rule represents a relational statement in the language. It consists of a relational operation followed by either an identifier or digits.

RelationalOperation ➔ *==|*!= |*> |*< | *>= | *<=

The "RelationalOperation" rule represents different relational operations available in the language.

LogicalOperation ➔ *& | *| | *!

The "LogicalOperation" rule represents different logical operations available in the language.

ArithmeticOperation ➔ *+ | *- | **| */ | *% | *++ |*--| *=

The "ArithmeticOperation" rule represents different arithmetic operations available in the language.

DataStructures ➔ Vector | Table

The "DataStructures" rule represents different data structures available in the language. It can be either a "Vector" or a "Table".

Vector ➔ vector Identifier *= (Digits | Letters)+ (, (Digits | Letters))*

The "Vector" rule represents the declaration of a vector in the language. It starts with the keyword "vector" followed by an identifier, then "*=" and a list of digits or letters. Additional elements can be added by separating them with commas.

Table ➔ table Identifier *= {(Digits | Letters)+ (, (Digits | Letters))*} , {(Digits | Letters)+ (, (Digits | Letters))*}

The "Table" rule represents the declaration of a table in the language. It starts with the keyword "table" followed by an identifier, then "*=" and two sets of curly braces. Each set contains a list of digits or letters separated by commas. The two sets are separated by a comma.

Identifier ➔ ! (Digits | Letters)+

The "Identifier" rule represents an identifier in the language. It starts with an exclamation mark "!" followed by a combination of digits or letters.

DataType➔(intDeclare | FractionDeclare |LetterDeclare | BooleanDeclare)

The "DataType" rule represents different data types available in the star language. It can be one of the following: "intDeclare", "FractionDeclare", "LettersDeclare", or " BooleanDeclare".

intDeclare ➔ int Identifier *= Integer

The "intDeclare" rule represents the declaration of an integer variable in the star language. It starts with the keyword "int" followed by an identifier and "*=" and an integer value.

FractionDeclare ➔ frac Identifier *= Fraction

The "FractionDeclare" rule represents the declaration of a fraction variable in the star language. It starts with the keyword "frac" followed by an identifier and "*=" and a fraction value.

LettersDeclare ➔ letter Identifier *= Letters

The "LettersDeclare" rule represents the declaration of a letter's variable in the star language. It starts with the keyword "letter" followed by an identifier and "*=" and a combination of letters.

BooleanDeclar ➔bool Identifier *= Boolean

The "BooleanDeclare" rule represents the declaration of a boolean variable in the language. It starts with the keyword "bool" followed by an identifier and "*=" and a boolean value.

PunctuationMarks ➔ (; | : | ?| ! | . |) | (| " | - | ,{|})

The "PunctuationMarks" rule represents various punctuation marks reserved in the language.

Digits ➔ Integer | Fractions

The "Digits" rule represents a digit in the language. It can be either an integer or a fraction.

Integer ➔ ["0" - "9"] +

The "Integer" rule represents a sequence of one or more digits in the range from 0 to 9.

Fraction ➔ (["0" - "9"])+ . (["0" - "9"])+

The "Fraction" rule represents a fraction in the language. It consists of one or more digits followed by a dot (.) and one or more digits.

Letters ➔ (["A"-"Z", "a"-"z"])+

The "Letters" rule represents a sequence of one or more letters, either uppercase or lowercase.

Boolean ➔ (true | false)

The "Boolean" rule represents a boolean value in the language. It can be either "true" or "false", indicating the two possible boolean states.

Comment ➔ (*%) (Digit | Letters | PunctuationMarks)+ (*%)

The "Comment" rule represents a comment in the language. It starts and ends with (*%) and can contain any combination of letters, digits, and punctuation marks.

2.3 jj Grammar Output

```
------------------Welcome to Star Programming Language-----------------

Enter your input:
print : 'Star Language is interesting' .

Syntactically correct statement
```

```
------------------Welcome to Star Programming Language-----------------

Enter your input:
vector ! theVector *= 78 , 10 .
Found Data Structure: Vector

Syntactically correct statement
```

```
------------------Welcome to Star Programming Language-----------------

Enter your input:
loop ( 64 *< 93 ): ! v *= 765 . end .

Syntactically correct statement
```

```
------------------Welcome to Star Programming Language-----------------

Enter your input:
if ( 5 *> 3 ) : ! variable *= 10 end .

Syntactically correct statement
```

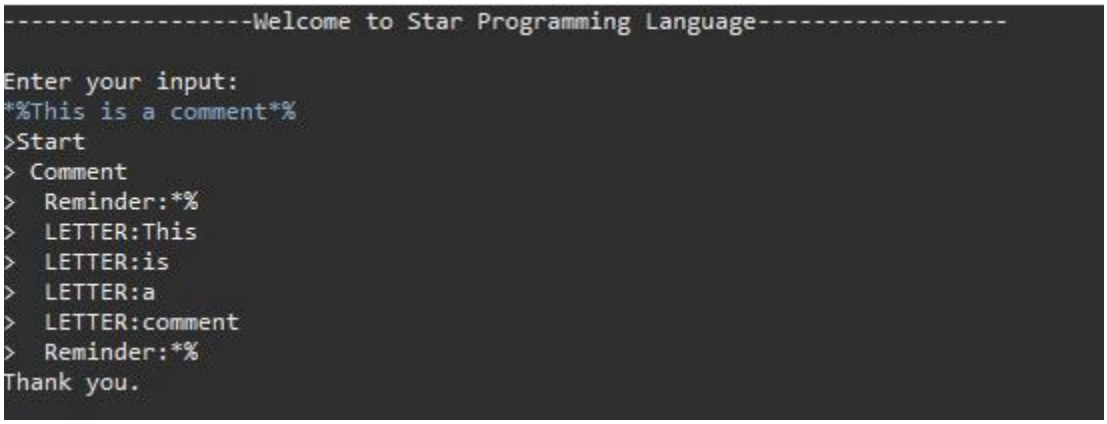
```
------------------Welcome to Star Programming Language------------------

Enter your input:
! x *= 1 *+ 7 .

Syntactically correct statement
```

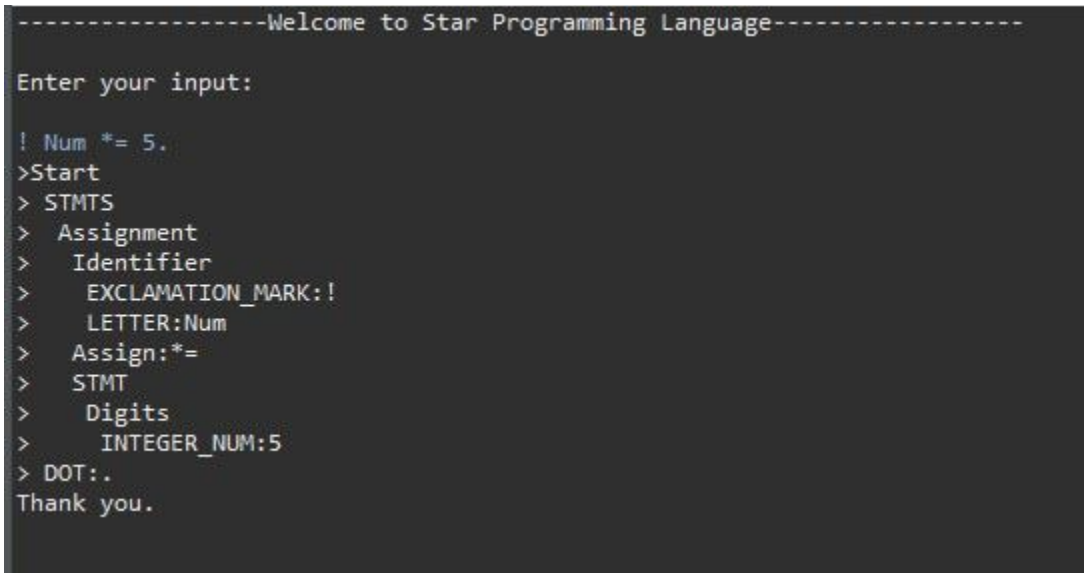
```
------------------Welcome to Star Programming Language------------------

Enter your input:
const int ! x *= 5 .
Found Integer Declaration

Syntactically correct statement
```

```
------------------Welcome to Star Programming Language------------------

Enter your input:
*% Star Language is interesting! *%

Syntactically correct statement
```

Phase 3

3.1 jjt Output





```
------------------Welcome to Star Programming Language------------------

Enter your input:

! F *= 5 *+ 5.
>Start
> STMTS
>  Assignment
>   Identifier
>    EXCLAMATION_MARK:!
>    LETTER:F
>   Assign:*=
>   STMT
>    Digits
>     INTEGER_NUM:5
>    ArithmeticStmt
>     ArithmeticOperation
>      Plus:*+
>     Digits
>      INTEGER_NUM:5
> DOT:.
Thank you.
```

```
------------------Welcome to Star Programming Language------------------

Enter your input:

const bool ! C *= true .
Found Boolean Declaration
>Start
> STMTS
>  Constant
>   Constant:const
>   DataType
>    BooleanDeclar
>     Boolean:bool
>     Identifier
>      EXCLAMATION_MARK:!
>      LETTER:C
>     Assign:*=
>     Boolean:true
> DOT:.
Thank you.
```

```
------------------Welcome to Star Programming Language------------------

Enter your input:
const int ! C *= 10 .
Found Integer Declaration
>Start
> STMTS
>  Constant
>   Constant:const
>   DataType
>    intDeclare
>     INTEGER:int
>     Identifier
>      EXCLAMATION_MARK:!
>      LETTER:C
>     Assign:*=
>     INTEGER_NUM:10
> DOT:.
Thank you.
```

```
------------------Welcome to Star Programming Language------------------

Enter your input:
print : 'Hello World'.
>Start
> STMTS
>  PrintSTMT
>   PRINT:print
>   COLON::
>   SINGLE_QUTATION:'
>   STMT
>    LETTER:Hello
>   STMT
>    LETTER:World
>   SINGLE_QUTATION:'
> DOT:.
Thank you.
```

```
------------------Welcome to Star Programming Language------------------

Enter your input:
if ( true *== true ) : ! c *= 0 end .
>Start
> STMTS
>  Conditional_Stmt
>   IF:if
>   R_BRACKET:(
>   Condition
>    Boolean:true
>    RelationalOperation
>     Equal:*==
>    Boolean:true
>   L_BRACKET:)
>   COLON::
>   STMTS
>    Assignment
>     Identifier
>      EXCLAMATION_MARK:!
>      LETTER:c
>     Assign:*=
>     STMT
>      Digits
>       INTEGER_NUM:0
>   END:end
> DOT:.
Thank you.
```

```
------------------Welcome to Star Programming Language------------------

Enter your input:

if ( 5 *> 5 ) : const bool ! C *= true end .
Found Boolean Declaration
>Start
> STMTS
>  Conditional_Stmt
>   IF:if
>   R_BRACKET:(
>   Condition
>    Digits
>     INTEGER_NUM:5
>    RelationalOperation
>     GreaterThan:*>
>    Digits
>     INTEGER_NUM:5
>   L_BRACKET:)
>   COLON::
>   STMTS
>    Constant
>     Constant:const
>     DataType
>      BooleanDeclar
>       Boolean:bool
>       Identifier
>        EXCLAMATION_MARK:!
>        LETTER:C
>       Assign:*=
>       Boolean:true
>   END:end
> DOT:.
Thank you.
```

```
------------------Welcome to Star Programming Language------------------

Enter your input:
loop (5 *<= 10 ) : print :'True Statement' . end .
>Start
> STMTS
>  Iteration
>   LOOP:loop
>   R_BRACKET:(
>   Condition
>    Digits
>     INTEGER_NUM:5
>    RelationalOperation
>     LessOrEqual:*<=
>    Digits
>     INTEGER_NUM:10
>   L_BRACKET:)
>   COLON::
>   STMTS
>    PrintSTMT
>     PRINT:print
>     COLON::
>     SINGLE_QUTATION:'
>     STMT
>      LETTER:True
>     STMT
>      LETTER:Statement
>     SINGLE_QUTATION:'
>   DOT:.
>   END:end
> DOT:.
Thank you.
```


```
------------------Welcome to Star Programming Language------------------

Enter your input:
loop (5 *<= 10 ) : print :'True Statement' . end .
>Start
> STMTS
>  Iteration
>   LOOP:loop
>   R_BRACKET:(
>   Condition
>    Digits
>     INTEGER_NUM:5
>    RelationalOperation
>     LessOrEqual:*<=
>    Digits
>     INTEGER_NUM:10
>   L_BRACKET:)
>   COLON::
>   STMTS
>    PrintSTMT
>     PRINT:print
>     COLON::
>     SINGLE_QUTATION:'
>     STMT
>      LETTER:True
>     STMT
>      LETTER:Statement
>     SINGLE_QUTATION:'
>   DOT:.
>   END:end
> DOT:.
Thank you.
```

Appendix

jj file

```
/**
 * JavaCC template file created by SF JavaCC plugin 1.5.28+ wizard for JavaCC 1.5.0+
 */
options
{
  static = true;
}

PARSER_BEGIN(MyNewGrammar)
package CPCS302_2023_3_Project;
import java.io.*;
import java.util.*;

public class MyNewGrammar
{
  public static void main(String args []) throws ParseException
  {
      MyNewGrammar parser = new MyNewGrammar(System.in);
   System.out.println("------------------Welcome to Star Programming Language------------------");
   while (true)
   {
     System.out.println("\nEnter your input: ");
     try
     {
       MyNewGrammar.Start();
```

```
      System.out.println("\nSyntactically correct statement");
    }
    catch (Error e)
    {
      System.out.println("Syntactically NOT correct statement");
      System.out.println(e.getMessage());
      break;
    }
   }
  }
}

PARSER_END(MyNewGrammar)

SKIP :
{
  " "
| "\r"
| "\t"
| "\n"
}

TOKEN : /* Arithmatic op. */
{
  < Plus : "*+" >
| < Minus : "*-" >
| < Multiply : "**" >
| < Divide : "*/" >
| < Reminder : "*%" >
```

```
| < Increment : "*++" >
| < Decrement : "*--" >
| < Assign : "*=" >
}

TOKEN : /* Rational op. */
{
  < Equal : "*==" >
| < NotEqual : "*!=">
| < GreaterThan : "*>" >
| < LessThan : "*<" >
| < GreaterOrEqual : "*>=" >
| < LessOrEqual : "*<=" >
}


TOKEN : /* Logical op. */
{
  < AND : "*&" >
| < OR : "*|">
| < NOT : "*!" >
}

TOKEN : /*Punctuation marks*/
{
< SEMI_COLON : ";" >
| < COLON : ":" >
| < QUESTION_MARK : "?" >
| < EXCLAMATION_MARK:"!" >
```

```
| < DOT :"." >
| < L_BRACKET: ")" >
| < R_BRACKET: "(" >
| < SINGLE_QUTATION: "'" >
| < DASH: "-" >
| < COMMA: "," >
| < RCBRACKET: "}" >
| < LCBRACKET: " { " >
}

TOKEN : //digit
{
< #Digits: ["0" - "9"] >
| < INTEGER_NUM: (["0"-"9"])+ >
| < FRACTION_NUM: (["0"-"9"])+(< DOT >)(["0"-"9"])+ >
}

TOKEN : //identifier
{
< IDENTIFIER: ("!")(["a"-"z"]|["A"-"Z"]|["0"-"9"])+ >
| < LETTER: (["A" - "Z" , "a" - "z"])+ >
}

TOKEN : //boolean
{
< Boolean: ("true ") | ("false ") >

}
```

```
TOKEN : // Keywords
{
<  IF: "if " >
|< THEN: "then " >
|< ELSE: "else " >
|< INTEGER: "int " >
|< FRACTION: "frac ">
|< LETTERS:"letter " >
| < BOOLEAN: "bool " >
| < CONSTANT: "const " >
| < CLASS:"class " >
| < STATIC:"static " >
| <PIBLIC:"public " >
| < PRIVATE:"private " >
| < RETURN:"return " >
| < END: "end " >
| < CONTINUE:"continue " >
| < PRINT:"print " >
| < DO:"do " >
| < WHILE:"while " >
| < VECTOR:"vector " >
| < TABLE:"table " >
| < LOOP: "loop " >
}

/*
TOKEN ://comment
{
  < COMMENT: ("*%")(["a"-"z"]|["A"-"Z"])+("*%") >
```

```
}
*/

//all statements must end with dot.
void Start(): { }
{
   STMTS() <DOT > | Comment()
}



//There are 6 types of statements in the language
void STMTS(): { }
{



 LOOKAHEAD(3)Assignment()
 |Constant()
 |LOOKAHEAD(3)PrintSTMT()
 |Conditional_Stmt()
 |Iteration()
 |DataStructures()


}



//rule to assign a statement (STMT) to an identifier (Identifier) in the left-hand side (LHS) using the "*=" operator.
void Assignment(): { }
{
```

```
  Identifier() (<Assign >) STMT()
}

// The "Constant" rule represents a constant declaration statement. It starts by the keyword "const" followed by the data type (DataType).
void Constant(): { }
{
   < CONSTANT > DataType()
}

//The "PrintSTMT" rule represents a print statement.
// It starts with the keyword "Print" followed by colon and sequence of digits, letters, or punctuation marks between 2 single quotations.
void PrintSTMT (): { }
{

    (< PRINT >) (<COLON>)  (< SINGLE_QUTATION>) ( STMT() )* (< SINGLE_QUTATION>)

}

//The "Conditional_Stmt" rule represents a conditional statement. It starts with the keyword "If" followed by a condition (Condition) enclosed in parentheses.
// The statements (STMTS) within the conditional block are enclosed in colons and can be repeated one or more times.
//The block is terminated with the keyword "End".
void Conditional_Stmt(): { }
{
  (< IF >) (<  R_BRACKET >) Condition() (< L_BRACKET >) (<COLON>) (STMTS())+ (< END >)
}
```

```
//The "Iteration" rule represents a loop statement.

//It starts with the keyword "Loop" followed by a condition (Condition) enclosed in parentheses.

// The statements (STMTS) within the loop block are enclosed in colons and can be repeated one or more times.

//The block is terminated with the keyword "End".

void Iteration(): { }

{

  (< LOOP >) (<  R_BRACKET >)  Condition() (< L_BRACKET >) (<COLON>) STMTS() (< DOT >) (< END >)

}


//The "Condition" rule represents a condition used in conditional and loop statements.

// It consists of an identifier or digits followed by a relational operation or logical operation and another identifier or digits.

void Condition(): { }

{

  ( Identifier() | Digits() |  < Boolean > ) ( RelationalOperation ()  | LogicalOperation() ) (Identifier() | Digits() | < Boolean >)

}


//The "STMT" rule represents a general statement in the language.

void STMT(): { }

{

  ( Identifier() | < LETTER > | Digits() ) ( ArithmeticStmt() | LogicalStmt () | RelationalStmt () )?

}


//The "ArithmeticStmt" rule represents an arithmetic statement

void ArithmeticStmt(): { }

{

  ArithmeticOperation() (Identifier() | Digits())
```

```
}

//The "LogicalStmt" rule represents a logical statement in the language
void LogicalStmt (): { }
{
  LogicalOperation() (Identifier() | Digits())
}

//The "RelationalStmt" rule represents a relational statement in the language
void RelationalStmt (): { }
{
  RelationalOperation () (Identifier() | Digits())
}

//The "RelationalOperation" rule represents different relational operations available in the
language.
void RelationalOperation (): { }
{
  (< Equal > | < NotEqual > | < GreaterThan > | < LessThan > | < GreaterOrEqual > | <
LessOrEqual >)
}

//The "LogicalOperation" rule represents different logical operations available in the language.
void LogicalOperation(): { }
{
  (< AND > | < OR > | < NOT >)
}

//The "ArithmeticOperation" rule represents different arithmetic operations available in the
language.
```

```
void  ArithmeticOperation(): { }
{
  (< Plus > | < Minus > | < Multiply > | < Divide > | < Reminder > | < Increment > | < Decrement >
  |< Assign >)
}

//there are two types of data structures in Star language
void DataStructures():{ }
{
  Vector()
| Table()
}

/*
To define a vector, it must start with vector followed by an identifier
than a sign operator, and elements could be digits or litters.
*/
void Vector(): { }
{
  < VECTOR > Identifier() <Assign> (Digits ()|< LETTER >)+ (< COMMA > (Digits ()|< LETTER >))*
  { System.out.println("Found Data Structure: Vector");}
}

/*
To define a table, it must start with table followed by an identifier
than a sign operator, and elements could be digits or litters.
*/
void Table(): { }
{
```

```
< TABLE > Identifier() <Assign> <LCBRACKET >(Digits () |< LETTER >)+ (< COMMA > (Digits () |< LETTER >))*

<RCBRACKET > <COMMA >  <LCBRACKET >(Digits () |< LETTER >)+ (< COMMA > (Digits () |< LETTER >))* <RCBRACKET >


{

    System.out.println("Found Data Structure: Table");

  }

}



/*An identifier must start with ! followed by any combinations of letters and digits

*/

void Identifier (): { }

{

  (<EXCLAMATION_MARK> ) ( Digits ()|< LETTER >)+

}


//Data Type can be integer declaration, fraction declaration, or letter declaration

void DataType (): { } {

  (intDeclare() | FractionDeclare() | LettersDeclare() | BooleanDeclar() )

}


//To declare an integer, it starts with <INTEGER> followed by an identifier then *= followed by an integer

void intDeclare ():{}{

  <INTEGER> Identifier() <Assign> <INTEGER_NUM>

  {

    System.out.println("Found Integer Declaration");

  }
```

```
}

//To declare a fraction, it starts with <FRACTION> followed by an identifier then *= followed by a fraction
void FractionDeclare  ():{}{
  <FRACTION> Identifier() <Assign> <FRACTION_NUM>
  {
    System.out.println("Found Fraction Declaration");
  }
}

//To declare a letter, it starts with <LETTERS> followed by an identifier then *= followed by a letter
void LettersDeclare ():{}{
  <LETTERS> Identifier() <Assign> <LETTER>
  {
    System.out.println("Found Letter Declaration");
  }
}

//To declare a boolean, it starts with <BOOLEAN> followed by an identifier then *= followed by a true or false
void BooleanDeclar ():{}{
  <BOOLEAN> Identifier() <Assign> <Boolean>
  {
    System.out.println("Found Boolean Declaration");
  }
}
```

```
/*
Comments start and end with an Asterisk and it can be containing any combination
of digits, letters, and punctuation marks.
*/
void Comment(): { } {
 < Reminder > (Digits()| <LETTER> | PunctuationMarks())+ < Reminder >
}

//The "PunctuationMarks" rule represents various punctuation marks reserved in the language.
void PunctuationMarks (): { } {
  <SEMI_COLON>
| <COLON>
| <QUESTION_MARK>
| <EXCLAMATION_MARK>
| <DOT>
| <L_BRACKET>
| <R_BRACKET>
| <SINGLE_QUTATION>
| <DASH>
| <COMMA>
}

//The "Digits" rule represents a digit in the language. It can be either an integer or a fraction.
void Digits (): { } {
 <INTEGER_NUM> | < FRACTION_NUM >
}
```

jjt file

```
/**
 * JJTree template file created by SF JavaCC plugin 1.5.28+ wizard for JavaCC 1.5.0+
 */
options
{
  static = true;
}


PARSER_BEGIN(MyNewGrammar)
package tree;


public class MyNewGrammar
{
  public static void main(String args [])
  {
    System.out.println("------------------Welcome to Star Programming Language------------------");
    System.out.print("\nEnter your input: ");
    new MyNewGrammar(System.in);
    try
    {
      SimpleNode n = MyNewGrammar.Start();
      n.dump(">");
      System.out.println("Thank you.");
    }
    catch (Exception e)
    {
      System.out.println("Oops.");
      System.out.println(e.getMessage());
    }
  }
}

PARSER_END(MyNewGrammar)
SKIP :
```

```
{
  " "
| "\r"
| "\t"
| "\n"
}

TOKEN : /* Arithmatic op. */
{
  < Plus : "*+" >
| < Minus : "*-" >
| < Multiply : "**" >
| < Divide : "*/" >
| < Reminder : "*%" >
| < Increment : "*++" >
| < Decrement : "*--" >
| < Assign : "*=" >
}

TOKEN : /* Rational op. */
{
  < Equal : "*==" >
| < NotEqual : "*!=">
| < GreaterThan : "*>" >
| < LessThan : "*<" >
| < GreaterOrEqual : "*>=" >
| < LessOrEqual : "*<=" >
}
```

```
TOKEN : /* Logical op. */
{
  < AND : "*&" >
| < OR : "*|">
| < NOT : "*!" >
}

TOKEN : /*Punctuation marks*/
{
< SEMI_COLON : ";" >
| < COLON : ":" >
| < QUESTION_MARK : "?" >
| < EXCLAMATION_MARK:"!" >
| < DOT :"." >
| < L_BRACKET: ")" >
| < R_BRACKET: "(" >
| < SINGLE_QUTATION: "'" >
| < DASH: "-" >
| < COMMA: "," >
| < RCBRACKET: "}" >
| < LCBRACKET: "{" >
}

TOKEN : //digit
{
< #Digits: ["0" - "9"] >
| < INTEGER_NUM: (["0"-"9"])+ >
| < FRACTION_NUM: (["0"-"9"])+(< DOT >)(["0"-"9"])+ >
```

```
}

TOKEN : //identifier
{
< IDENTIFIER: ("!")(["a"-"z"]|["A"-"Z"]|["0"-"9"])+ >
| < LETTER: (["A" - "Z" , "a" - "z"])+ >
}

TOKEN : //boolean
{
< Boolean: ("true ") | ("false ") >

}

TOKEN : // Keywords
{
<  IF: "if " >
|< THEN: "then " >
|< ELSE: "else " >
|< INTEGER: "int " >
|< FRACTION: "frac ">
|< LETTERS:"letter " >
| < BOOLEAN: "bool " >
| < CONSTANT: "const " >
| < CLASS:"class " >
| < STATIC:"static " >
| <PIBLIC:"public " >
| < PRIVATE:"private " >
| < RETURN:"return " >
```

```
| < END: "end " >
| < CONTINUE:"continue " >
| < PRINT:"print " >
| < DO:"do " >
| < WHILE:"while " >
| < VECTOR:"vector " >
| < TABLE:"table " >
| < LOOP: "loop " >
}



//all statements must end with dot.
SimpleNode Start() :{}
{
  ( STMTS() DOT() | Comment() )

 {
   return jjtThis;
 }

}

//There are 6 types of statements in the language
void STMTS(): { }
{


 LOOKAHEAD(3)Assignment()
```

```
  |Constant()
  |LOOKAHEAD(3)PrintSTMT()
  |Conditional_Stmt()
  |Iteration()
  |DataStructures()

}

//rule to assign a statement (STMT) to an identifier (Identifier) in the left-hand side (LHS) using the "*=" operator.
void Assignment(): { }
{
  Identifier() (Assign()) STMT()
}

// The "Constant" rule represents a constant declaration statement. It starts by the keyword "const" followed by the data type (DataType).
void Constant(): { }
{
   CONSTANT() DataType()
}

//The "PrintSTMT" rule represents a print statement.
// It starts with the keyword "Print" followed by colon and sequence of digits, letters, or punctuation marks between 2 single quotations.

void PrintSTMT (): { }
{
```

(PRINT()) (COLON()) (SINGLE_QUTATION()) (STMT())* (SINGLE_QUTATION())

}

//The "Conditional_Stmt" rule represents a conditional statement. It starts with the keyword "If" followed by a condition (Condition) enclosed in parentheses.

// The statements (STMTS) within the conditional block are enclosed in colons and can be repeated one or more times.

//The block is terminated with the keyword "End".

void Conditional_Stmt(): { }

{

(IF()) (R_BRACKET()) Condition() (L_BRACKET()) (COLON()) (STMTS())+ (END())

}

//The "Iteration" rule represents a loop statement.

//It starts with the keyword "Loop" followed by a condition (Condition) enclosed in parentheses.

// The statements (STMTS) within the loop block are enclosed in colons and can be repeated one or more times.

//The block is terminated with the keyword "End".

void Iteration(): { }

{

(LOOP()) (R_BRACKET()) Condition() (L_BRACKET()) (COLON()) STMTS() (DOT()) (END())

}

//The "Condition" rule represents a condition used in conditional and loop statements.

// It consists of an identifier or digits followed by a relational operation or logical operation and another identifier or digits.

```
void Condition(): { }
{
  ( Identifier() | Digits() |  Boolean()) ( RelationalOperation ()  | LogicalOperation() ) (Identifier() | Digits() | Boolean())
}

//The "STMT" rule represents a general statement in the language.
void STMT(): { }
{
  ( Identifier() | LETTER() | Digits() ) ( ArithmeticStmt() | LogicalStmt () | RelationalStmt () )?
}

//The "ArithmeticStmt" rule represents an arithmetic statement
void ArithmeticStmt(): { }
{
  ArithmeticOperation() (Identifier() | Digits())
}

//The "LogicalStmt" rule represents a logical statement in the language
void LogicalStmt (): { }
{
  LogicalOperation() (Identifier() | Digits())
}

//The "RelationalStmt" rule represents a relational statement in the language
void RelationalStmt (): { }
{
  RelationalOperation () (Identifier() | Digits())
}
```

```
//The "RelationalOperation" rule represents different relational operations available in the language.
void RelationalOperation (): { }
{
  (Equal() | NotEqual() | GreaterThan() | LessThan()| GreaterOrEqual() | LessOrEqual())
}

//The "LogicalOperation" rule represents different logical operations available in the language.
void LogicalOperation(): { }
{
  (AND() | OR() | NOT())
}

//The "ArithmeticOperation" rule represents different arithmetic operations available in the language.
void  ArithmeticOperation(): { }
{
  (Plus() | Minus() | Multiply() | Divide() | Reminder() | Increment() | Decrement()
  | Assign() )
}

//there are two types of data structures in Star language
void DataStructures():{ }
{
  Vector()
| Table()
}

/*
```

```
To define a vector, it must start with vector followed by an identifier
than a sign operator, and elements could be digits or litters.
*/
void Vector(): {Token t; }
{
  t = <VECTOR> {jjtThis.jjtSetValue(t.image); }
  Identifier() Assign() (Digits ()| LETTER() )+ (COMMA()  (Digits ()|LETTER()))*
  { System.out.println("Found Data Structure: Vector");}
}

/*
To define a table, it must start with table followed by an identifier
than a sign operator, and elements could be digits or litters.
*/
void Table(): {Token t; }
{
  t = <TABLE> {jjtThis.jjtSetValue(t.image); }
  Identifier() Assign() LCBRACKET() (Digits () |LETTER())+ (COMMA() (Digits() | LETTER() ))*
  RCBRACKET() COMMA()   LCBRACKET() (Digits () | LETTER() )+ ( COMMA()  (Digits() |LETTER()))*
RCBRACKET()

{
    System.out.println("Found Data Structure: Table");
  }
}


/*An identifier must start with ! followed by any combinations of letters and digits
*/
```

```
void Identifier (): { }

{

  (EXCLAMATION_MARK() ) ( Digits ()| LETTER() )+

}


//Data Type can be integer declaration, fraction declaration, or letter declaration

void DataType (): { } {

  (intDeclare() | FractionDeclare() | LettersDeclare() | BooleanDeclar() )

}


//To declare an integer, it starts with <INTEGER> followed by an identifier then *= followed by an integer

void intDeclare ():{}{

  INTEGER() Identifier() Assign() INTEGER_NUM()

     { System.out.println("Found Integer Declaration"); }

  }



//To declare a fraction, it starts with <FRACTION> followed by an identifier then *= followed by a fraction

void FractionDeclare  ():{}{

  FRACTION() Identifier() Assign() FRACTION_NUM()

  {

     System.out.println("Found Fraction Declaration");

  }

}


//To declare a letter, it starts with <LETTERS> followed by an identifier then *= followed by a letter

void LettersDeclare ():{}{
```

```
 LETTERS() Identifier() Assign() LETTER()
 {
   System.out.println("Found Letter Declaration");
 }
}

//To declare a boolean, it starts with <BOOLEAN> followed by an identifier then *= followed by a true or false
void BooleanDeclar ():{}{
 BOOLEAN() Identifier() Assign() Boolean()
 {
   System.out.println("Found Boolean Declaration");
 }
}

/*
Comments start and end with an Asterisk and it can be containing any combination
of digits, letters, and punctuation marks.
*/
void Comment(): { } {
 Reminder()  (Digits()| LETTER() | PunctuationMarks())+  Reminder()
}

void PunctuationMarks (): { } {
  SEMI_COLON()
| COLON()
| QUESTION_MARK()
| EXCLAMATION_MARK()
```

```
| DOT()
| L_BRACKET()
| R_BRACKET()
| SINGLE_QUTATION()
| DASH()
| COMMA()
}

void Digits (): { } {
 INTEGER_NUM() | FRACTION_NUM()
}

void DOT(): { Token t; }
{
  t = <DOT> {jjtThis.jjtSetValue(t.image); }
}

void Assign(): {Token t;}
{
   t = <Assign> {jjtThis.jjtSetValue(t.image); }
}

void CONSTANT():{ Token t; }
{
  t = <CONSTANT> {jjtThis.jjtSetValue(t.image); }
}

void PRINT(): { Token t; }
{
```

```
  t = <PRINT> {jjtThis.jjtSetValue(t.image); }
}

void COLON():{ Token t; }
{
  t = <COLON> {jjtThis.jjtSetValue(t.image); }
}

void SINGLE_QUTATION() : { Token t; }
{
  t = <SINGLE_QUTATION> {jjtThis.jjtSetValue(t.image); }
}

void IF(): { Token t; }
{
  t = <IF> {jjtThis.jjtSetValue(t.image); }
}

void R_BRACKET(): { Token t; }
{
  t = <R_BRACKET> {jjtThis.jjtSetValue(t.image); }
}


void L_BRACKET():{ Token t; }
{
  t = <L_BRACKET> {jjtThis.jjtSetValue(t.image); }
}
```

```
void END(): { Token t; }
{
  t = <END> {jjtThis.jjtSetValue(t.image); }
}

void LOOP(): { Token t; }
{
  t = <LOOP> {jjtThis.jjtSetValue(t.image); }
}

void Boolean(): { Token t; }
{
  t = <Boolean> {jjtThis.jjtSetValue(t.image); }
}

void LETTER(): { Token t; }
{
  t = <LETTER> {jjtThis.jjtSetValue(t.image); }
}

void Equal(): { Token t; }
{
  t = <Equal> {jjtThis.jjtSetValue(t.image); }
}

void NotEqual():{ Token t; }
{
  t = <NotEqual> {jjtThis.jjtSetValue(t.image); }
}
```

```
void GreaterThan(): { Token t; }
{
  t = <GreaterThan> {jjtThis.jjtSetValue(t.image); }
}

void LessThan():{ Token t; }
{
  t = <LessThan> {jjtThis.jjtSetValue(t.image); }
}

void GreaterOrEqual(): { Token t; }
{
  t = <GreaterOrEqual> {jjtThis.jjtSetValue(t.image); }
}

void LessOrEqual():{ Token t; }
{
  t = <LessOrEqual> {jjtThis.jjtSetValue(t.image); }
}

void AND():{ Token t; }
{
  t = <AND> {jjtThis.jjtSetValue(t.image); }
}

void OR(): { Token t; }
{
  t = <OR> {jjtThis.jjtSetValue(t.image); }
```

```
}

void NOT():{ Token t; }
{
  t = <NOT> {jjtThis.jjtSetValue(t.image); }
}

void Plus():{ Token t; }
{
  t = <Plus> {jjtThis.jjtSetValue(t.image); }
}

void Minus():{ Token t; }
{
  t = <Minus> {jjtThis.jjtSetValue(t.image); }
}

void Multiply():{ Token t; }
{
  t = <Multiply> {jjtThis.jjtSetValue(t.image); }
}

void Divide():{ Token t; }
{
  t = <Divide> {jjtThis.jjtSetValue(t.image); }
}

void Reminder():{ Token t; }
{
```

```
  t = <Reminder> {jjtThis.jjtSetValue(t.image); }
}

void Increment():{ Token t; }
{
  t = <Increment> {jjtThis.jjtSetValue(t.image); }
}

void Decrement():{ Token t; }
{
  t = <Decrement> {jjtThis.jjtSetValue(t.image); }
}

void COMMA():{ Token t; }
{
  t = <COMMA> {jjtThis.jjtSetValue(t.image); }
}

void LCBRACKET():{ Token t; }
{
  t = <LCBRACKET> {jjtThis.jjtSetValue(t.image); }
}

void EXCLAMATION_MARK():{ Token t; }
{
  t = <EXCLAMATION_MARK> {jjtThis.jjtSetValue(t.image); }
}

void RCBRACKET(): { Token t; }
```

```
{
  t = <RCBRACKET> {jjtThis.jjtSetValue(t.image); }
}

void INTEGER():{ Token t; }
{
  t = <INTEGER> {jjtThis.jjtSetValue(t.image); }
}

void INTEGER_NUM():{ Token t; }
{
  t = <INTEGER_NUM> {jjtThis.jjtSetValue(t.image); }
}

void FRACTION():{ Token t; }
{
  t = <FRACTION> {jjtThis.jjtSetValue(t.image); }
}

void FRACTION_NUM():{ Token t; }
{
  t = <FRACTION_NUM> {jjtThis.jjtSetValue(t.image); }
}

void LETTERS():{ Token t; }
{
  t = <LETTERS> {jjtThis.jjtSetValue(t.image); }
}
```

```
void BOOLEAN(): { Token t; }
{
  t = <BOOLEAN> {jjtThis.jjtSetValue(t.image); }
}

void SEMI_COLON():{ Token t; }
{
  t = <SEMI_COLON> {jjtThis.jjtSetValue(t.image); }
}

void QUESTION_MARK():{ Token t; }
{
  t = <QUESTION_MARK> {jjtThis.jjtSetValue(t.image); }
}

void DASH():{ Token t; }
{
  t = <DASH> {jjtThis.jjtSetValue(t.image); }
```