

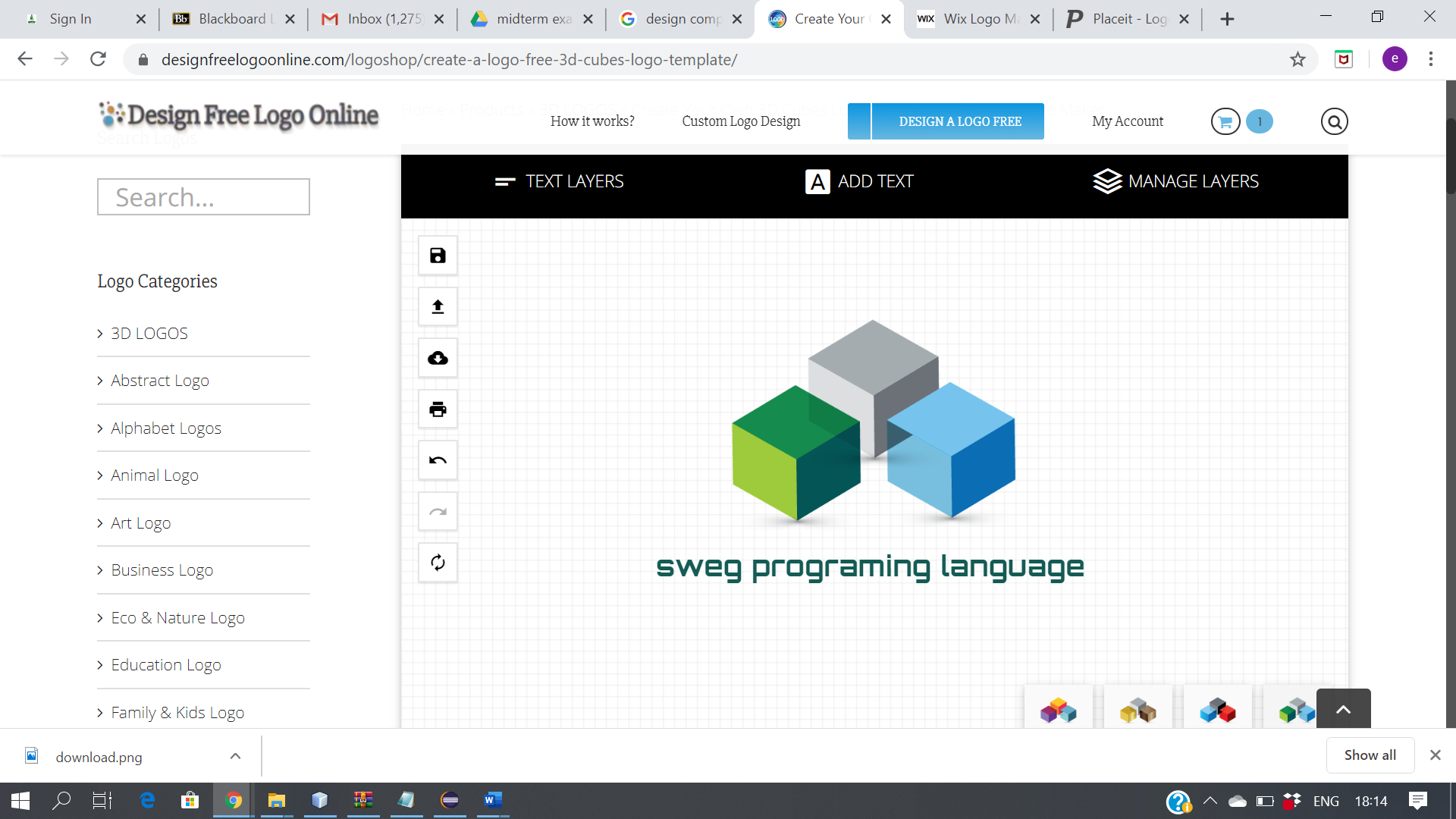
King Abdulaziz University

Faculty of Computing and Information Technology

Computer Science Department

**CPCS302, 2nd Term 2020**

**Due:** Sunday April 11th @ 11:59 pm



|  |  |  |
| --- | --- | --- |
| Name | ID | Section |
| Waad Alshanbari | **-** | **IAR** |
| Esraa Turki | **-** |
| Shahad Qumosani | **-** |
| Ghaidaa Alghamdi | **-** |

|  |  |
| --- | --- |
| Work | Student |
| Phase 1 | Esraa, Waad |
| Phase 2 | Ghaidaa, Shahad |
| Phase 3 | Shahad, Waad , Esraa |

Table of Contents

[Phase 1: 2](#_Toc37540598)

[Identification of logical pieces of the language: 2](#_Toc37540599)

[Tokens**:** 2](#_Toc37540600)

[Definition of operators: 3](#_Toc37540601)

[Phase 2: 4](#_Toc37540602)

[statements with BNF: 4](#_Toc37540603)

[1. Arithmetic Statements 4](#_Toc37540604)

[2. Logical Statements 5](#_Toc37540605)

[3. Comparison Statements 5](#_Toc37540606)

[4. Conditional Statements 5](#_Toc37540607)

[Phase 3: 6](#_Toc37540608)

[Output jj file: 6](#_Toc37540609)

[Output jjt file: 8](#_Toc37540610)

[Appendix: 11](#_Toc37540611)

[jj File: 11](#_Toc37540612)

[jjt file : 16](#_Toc37540613)

# Phase 1:

We designed a language that combines good factors of every language we have used, and we add some from our imagination

example:

* Name! = &sweg (declaration of variable name and the type is string)
* number! = 12.50 (declaration of number and the type is float)
* %....................% (anything between the symbol % is considered as a comment)
* #PI = 3.14 (declaration of a constant)
* END (end of the statement)

## Identification of logical pieces of the language:

|  |  |
| --- | --- |
| Name! = lastName! | assignment |
| Number! = 7 + 2 | Using Arithmetic operation |
| Arr! =[2] | Data structure (List) |
| if (z!>>3)  then(a!=1)  else b!=0 | If (conditional statement) |

## Tokens**:**

|  |  |
| --- | --- |
| Input | Type |
| str, int, float, list, iden , if, then, else, END | Keyword |
| [a-z] [A-Z] | Letter |
| [0-9] | Digit |
| +,-,\*,~,\*\* | Arithmetic operation |
| @ , ? , $ | Logical operation |
| >> , <<, ==, /= | Comparison operation |
| Space, tab, newline, comments | Whitespace |
| ( ),[ ] | Brackets |

## Definition of operators:

|  |  |
| --- | --- |
| **definition of Arithmetic operators:** | |
| Plus | **+** |
| Minus | **-** |
| Multiplication | **\*** |
| Division | **~** |
| Power | **\*\*** |
| Increment | **++** |
| Decrement | **--** |
| **definition of logical operation:** | |
| And | **@** |
| Or | **?** |
| Not | **$** |
| **definition of comparison operation:** | |
| Greater than | **>>** |
| Less than | **<<** |
| Equal | **==** |
| Not Equal | **/=** |
| **definition of conditional operation:** | |
| IF | if |
| Then | then |
| Else | else |

# Phase 2:

## statements with BNF:

The start symbol for all statements is:

<statements> ( < Arithmetic\_Statements >

| < Comparison\_Statements >

| < Logical\_Statements >

| < Conditional\_Statements > )

### Arithmetic Statements

الشكل< Arithmetic\_Statements > <Term> < Arithmetic\_Operator >

( < Term > | < Data\_Type > ) ( < Arithmetic\_operator > <Term > )?

| < IDEN > (< INCREMENT> | <DECREMENT> )

| (< INCREMENT> | <DECREMENT> ) < IDEN >

<exp> <Term> <Arithmetic\_Operator > <Term>

<Term> <LETTER>| <DIGITS> | < Data\_Type > | < exp >

< Data\_Type ><CONSTANT> | <FLOTE> |<STR> |<LIST>

< Arithmetic\_Operator > <PLUS> | <MINUS> | <MULTIPLY> | <DIVIDE> |<POWER>

< LETTER > [ a-z ]| A-Z]

< DIGIT > [0-9]

< CONSTANT> الشكل # (< LETTER > (< DIGIT >|< LETTER >)\*

STR الشكل(< DIGIT >|< LETTER >)+ >

INT الشكل (< DIGIT >)+ >

IDEN الشكل (< LETTER > (< DIGIT >|< LETTER >)\* "!"

FLOAT الشكل(< DIGIT >)+ "." (< DIGIT >)+

LIST الشكل "["(< DIGIT >)+"]"

### Logical Statements

<Logical\_Statements > ( < Brackets\_Operator > <Comparison\_Statements > < Brackets\_Operator > < Logical\_Operator > < Brackets\_Operator > <Comparison\_Statements > < Brackets\_Operator > )

| < Negation\_Operator > < Brackets\_Operator > < Comparison\_Statements > < Brackets\_Operator >

<Brackets\_Operator>< LEFT\_PARANTHESIS > | < RIGHT\_PARANTHESIS > |< LEFT\_BRACKET > |< RIGHT\_BRACKET >

<Logical\_Operator> @ | ?

< Negation\_Operator >$

### Comparison Statements

<comparison\_statement> <Term> <Comparison\_Operator> <Term>

<comparison\_op> << | >> | = = | /=

### Conditional Statements

<conditional\_statement> <if> < Brackets\_Operator > < Comparison\_Statements > < Brackets\_Operator >

<THEN > < Arithmetic\_Statements >

(<ELSE> < Arithmetic\_Statements >)?

Whitespace الشكل \n \r \t

Comments الشكل% ~ (\r |\n) \* %

# Phase 3:

## Output jj file:

A screenshot of a cell phone

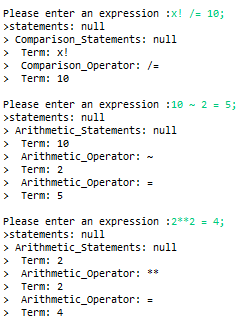
Description automatically generated

A screenshot of a cell phone

Description automatically generated

## Output jjt file:

A screenshot of a cell phone

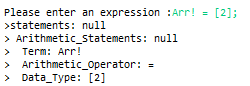
Description automatically generated

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated



# Appendix:

## jj File:

/\* ( sweg language )

Team members:

Name ID Section

Shahad Qumosani - 1708315 - IAR

Esraa Turki - 1708127 - IAR

Waad AlShanbari - 1707637 - IAR

Ghaidaa AlGhamdi - 1427159 - IAR

Instructor: Nuha AlNahdi

\*/

options

**{**

**static** = **true**;

**}**

PARSER\_BEGIN(sweg)

**package** Project;

**public** **class** sweg

**{**

**public** **static** **void** main(String args []) **throws** ParseException

{

sweg parser = **new** sweg(System.in);

System.out.println("\n~ ~ ~ ~ ~ WELCOME TO SWEG LANGUAGE ~ ~ ~ ~ ~\n");

System.out.println("Reading from standard input...\n");

**while** (**true**)

{

System.out.print("\nPlease enter an expression :");

**try**

{

**switch** (sweg.one\_line())

{

**case** 0 :

System.out.println("\n\tSyntactically Correct Statement\n");

**break**;

**case** 1 :

**break**;

**default** :

**break**;

}

}

**catch** (Exception e)

{

System.out.println("Incorrect Statement");

System.out.println(e.getMessage());

sweg.ReInit(System.in);

}

**catch** (Error e)

{

System.out.println("Oops.");

System.out.println(e.getMessage());

**break**;

}

}

}

**}**

PARSER\_END(sweg)

//--------------------------------------------------------------------------------

// Defining language statements as tokens.

SKIP :

**{**

" " // white space

| "\r" // carriage return

| "\t" // tab

| "\n" // newline

| < "%" (~[ "\n", "\r" ])\* "%" > // Comment

**}**

TOKEN :

**{**

< #LETTER : [ "a"-"z", "A"-"Z" ] >

| < #DIGIT : [ "0"-"9" ] >

**}**

TOKEN :

**{**

< STR :

"&"

(

< DIGIT >

| < LETTER >

)+ >

| < INT : (< DIGIT >)+ >

| < IDEN :

< LETTER >

(

< DIGIT >

| < LETTER >

)\*

"!" >

| < FLOAT : (< DIGIT >)+ "." (< DIGIT >)+ >

| < LIST : "[" (< DIGIT >)+ "]" >

| < END : ";" >

| < CONSTANT :

"#" (< LETTER >)

(

< DIGIT >

| < LETTER >

)\* >

**}**

TOKEN :

**{**

< LEFT\_PARANTHESIS : "(" >

| < RIGHT\_PARANTHESIS : ")" >

| < LEFT\_BRACKET : "[" >

| < RIGHT\_BRACKET : "]" >

**}**

TOKEN : /\* ARITHMETIC OPERATORS \*/

**{**

< PLUS : "+" >

| < MINUS : "-" >

| < MULTIPLY : "\*" >

| < DIVIDE : "~" >

| < POWER : "\*\*" >

| < INCREMENT : "++" >

| < DECREMENT : "--" >

| < ASSIGN : "=" >

**}**

TOKEN : /\* LOGICAL OPERATORS \*/

**{**

< AND : "@" >

| < OR : "?" >

| < NOT : "$" >

**}**

TOKEN : /\* COMPARISON OPERATORS \*/

**{**

< EQUAL : "==" >

| < NOTEQUAL : "/=" >

| < LESSTHAN : "<<" >

| < GREATERTHAN : ">>" >

**}**

TOKEN : /\* CONDITIONAL OPERATORS \*/

**{**

< IF : "if" >

| < ELSE : "else" >

| < THEN : "then" >

**}**

//--------------------------------------------------------------------------------

**int** one\_line() :

**{**

**}**

**{**

statements() < END >

{

**return** 0;

}

|< END >

{

**return** 1;

}

**}**

//--------------------------------------------------------------------------------

**void** statements() :

**{}**

**{**

(

LOOKAHEAD(2)

Arithmetic\_Statements()

| Comparison\_Statements()

| Logical\_Statements()

| Conditional\_Statements()

)

**}**

//--------------------------------------------------------------------------------

**void** Arithmetic\_Statements() :

**{}**

**{**

LOOKAHEAD(2) // to differentiate between ( \* ) multiplication and (\*\*) power

Term() Arithmetic\_Operator()

( LOOKAHEAD(2)

Term()

| Data\_Type()

)

(

Arithmetic\_Operator() Term()

)? // Example: 2 + 3 or x! ~ y! or 8 - 3 = 5 or x! = 7 + 2

// float -- > x! =2.3 or string -- > MyName! = Shahad\*\*\*\*\*\*\*\*

// List -- > x! =[2]

**|** < IDEN >

**(**

< INCREMENT >

**|** < DECREMENT >

**)** // Example: x!++ means -- > increment identifier x

**|**

**(**

< INCREMENT >

**|** < DECREMENT >

**)**

< IDEN >

// Example: ++x!

**}**

//--------------------------------------------------------------------------------

**void** Comparison\_Statements() :

**{}**

**{**

**(**

Term() Comparison\_Operator() Term()

**)** // Example: 2 << 3 or x! == y!

**}**

//--------------------------------------------------------------------------------

**void** Logical\_Statements() :

**{}**

**{**

**(**

Brackets\_Operator()

Comparison\_Statements()

Brackets\_Operator()

Logical\_Operator()

Brackets\_Operator()

Comparison\_Statements()

Brackets\_Operator() **)** // Example: AND -- > [2 << 3] @ [7 >> 4]

// OR -- > [2 << 3] ? [7 >> 4]

**|** Negation\_Operator()

Brackets\_Operator()

Comparison\_Statements()

Brackets\_Operator() // Example: NOT -- > $[ 6 >> 9 ] or $[ x! == y! ]

**}**

//--------------------------------------------------------------------------------

**void** Conditional\_Statements() :

**{}**

**{**

< IF >

Brackets\_Operator()

Comparison\_Statements()

Brackets\_Operator()

< THEN >

Arithmetic\_Statements() // Example: if (x!==5) then y!=5

**(**

< ELSE >

Arithmetic\_Statements()

**)?** // Example: if (x!==5) then y!=5 else y!=3;

**}**

//--------------------------------------------------------------------------------

**void** Term() :

**{}**

**{**

**(**

< IDEN >

**|** < INT >

**|** < CONSTANT >

**)**

**}**

//--------------------------------------------------------------------------------

**void** Arithmetic\_Operator() :

**{}**

**{**

**(**

< PLUS >

**|** < MINUS >

**|** < MULTIPLY >

**|** < DIVIDE >

**|** < POWER >

**|** < ASSIGN >

**)**

**}**

//--------------------------------------------------------------------------------

**void** Comparison\_Operator() :

**{}**

**{**

**(**

< EQUAL >

**|** < NOTEQUAL >

**|** < LESSTHAN >

**|** < GREATERTHAN >

**)**

**}**

//--------------------------------------------------------------------------------

**void** Logical\_Operator() :

**{}**

**{**

**(**

< AND >

**|** < OR >

**)**

**}**

//--------------------------------------------------------------------------------

**void** Data\_Type() :

**{}**

**{**

**(**

< FLOAT >

**|** < LIST >

**|** < STR >

**|** < CONSTANT >

**)**

**}**

//--------------------------------------------------------------------------------

**void** Negation\_Operator() :

**{}**

**{**

**(**

< NOT >

**)**

**}**

//--------------------------------------------------------------------------------

**void** Brackets\_Operator() :

**{}**

**{**

**(**

< LEFT\_PARANTHESIS >

**|** < RIGHT\_PARANTHESIS >

**|** < LEFT\_BRACKET >

**|** < RIGHT\_BRACKET >

**)**

**}**

## jjt file :

/\* ( sweg language )

Team members:

Name ID Section

Shahad Qumosani - 1708315 - IAR

Esraa Turki - 1708127 - IAR

Waad AlShanbari - 1707637 - IAR

Ghaidaa AlGhamdi - 1427159 - IAR

Instructor: Nuha AlNahdi

\*/

options

{

static = true;

}

PARSER\_BEGIN(sweg)

package sweg;

public class sweg

{

public static void main(String args []) throws ParseException

{

sweg parser = new sweg(System.in);

System.out.println("\n~ ~ ~ ~ ~ WELCOME TO SWEG LANGUAGE ~ ~ ~ ~ ~\n");

System.out.println("Reading from standard input...\n");

while (true)

{

System.out.print("\nPlease enter an expression :");

try

{

SimpleNode e = sweg.statements();

e.dump(">");

switch (sweg.one\_line())

{

case 0 :

System.out.println("\n\tSyntactically Correct Statement\n");

break;

case 1 :

break;

default :

break;

}

}

catch (Exception e)

{

System.out.println("Incorrect Statement");

System.out.println(e.getMessage());

sweg.ReInit(System.in);

}

catch (Error e)

{

System.out.println("Oops.");

System.out.println(e.getMessage());

break;

}

}

}

}

PARSER\_END(sweg)

//--------------------------------------------------------------------------------

// Defining language statements as tokens.

SKIP :

{

" " // white space

| "\r" // carriage return

| "\t" // tab

| "\n" // newline

| < "%" (~[ "\n", "\r" ])\* "%" > // Comment

}

TOKEN :

{

< #LETTER : [ "a"-"z", "A"-"Z" ] >

| < #DIGIT : [ "0"-"9" ] >

}

TOKEN :

{

< STR :

"&"

(

< DIGIT >

| < LETTER >

)+ >

| < INT : (< DIGIT >)+ >

| < IDEN :

< LETTER >

(

< DIGIT >

| < LETTER >

)\*

"!" >

| < FLOAT : (< DIGIT >)+ "." (< DIGIT >)+ >

| < LIST : "[" (< DIGIT >)+ "]" >

| < END : ";" >

| < CONSTANT :

"#" (< LETTER >)

(

< DIGIT >

| < LETTER >

)\* >

}

TOKEN :

{

< LEFT\_PARANTHESIS : "(" >

| < RIGHT\_PARANTHESIS : ")" >

| < LEFT\_BRACKET : "[" >

| < RIGHT\_BRACKET : "]" >

}

TOKEN : /\* ARITHMETIC OPERATORS \*/

{

< PLUS : "+" >

| < MINUS : "-" >

| < MULTIPLY : "\*" >

| < DIVIDE : "~" >

| < POWER : "\*\*" >

| < INCREMENT : "++" >

| < DECREMENT : "--" >

| < ASSIGN : "=" >

}

TOKEN : /\* LOGICAL OPERATORS \*/

{

< AND : "@" >

| < OR : "?" >

| < NOT : "$" >

}

TOKEN : /\* COMPARISON OPERATORS \*/

{

< EQUAL : "==" >

| < NOTEQUAL : "/=" >

| < LESSTHAN : "<<" >

| < GREATERTHAN : ">>" >

}

TOKEN : /\* CONDITIONAL OPERATORS \*/

{

< IF : "if" >

| < ELSE : "else" >

| < THEN : "then" >

}

//--------------------------------------------------------------------------------

int one\_line() :

{

}

{

statements() < END >

{

return 0;

}

|< END >

{

return 1;

}

}

//--------------------------------------------------------------------------------

SimpleNode statements() :

{}

{

(

LOOKAHEAD(2)

Arithmetic\_Statements()

| Comparison\_Statements()

| Logical\_Statements()

| Conditional\_Statements()

)

{

return jjtThis;

}

}

//--------------------------------------------------------------------------------

void Arithmetic\_Statements() :

{

Token t;

}

{

LOOKAHEAD(2) // to differentiate between ( \* ) multiplication and (\*\*) power

Term() Arithmetic\_Operator()

( LOOKAHEAD(2)

Term()

| Data\_Type()

)

(

Arithmetic\_Operator() Term()

)? // Example: 2 + 3 or x! ~ y! or 8 - 3 = 5 or x! = 7 + 2

// float -- > x! =2.3 or string -- > MyName! = Shahad\*\*\*\*\*\*\*\*

// List -- > x! =[2]

| t = < IDEN >

{

jjtThis.jjtSetValue(t.image);

}

(

t = < INCREMENT >

{

jjtThis.jjtSetValue(t.image);

}

| t = < DECREMENT >

{

jjtThis.jjtSetValue(t.image);

}

) // Example: x!++ means -- > increment identifier x

|

(

t = < INCREMENT >

{

jjtThis.jjtSetValue(t.image);

}

| t = < DECREMENT >

{

jjtThis.jjtSetValue(t.image);

}

)

t = < IDEN >

{

jjtThis.jjtSetValue(t.image);

} // Example: ++x!

}

//--------------------------------------------------------------------------------

void Comparison\_Statements() :

{}

{

(

Term() Comparison\_Operator() Term()

) // Example: 2 << 3 or x! == y!

}

//--------------------------------------------------------------------------------

void Logical\_Statements() :

{

Token t;

}

{

(

Brackets\_Operator()

Comparison\_Statements()

Brackets\_Operator()

Logical\_Operator()

Brackets\_Operator()

Comparison\_Statements()

Brackets\_Operator() ) // Example: AND -- > [2 << 3] @ [7 >> 4]

// OR -- > [2 << 3] ? [7 >> 4]

| Negation\_Operator()

Brackets\_Operator()

Comparison\_Statements()

Brackets\_Operator() // Example: NOT -- > $[ 6 >> 9 ] or $[ x! == y! ]

}

//--------------------------------------------------------------------------------

void Conditional\_Statements() :

{

Token t;

}

{

t = < IF >

{

jjtThis.jjtSetValue(t.image);

}

Brackets\_Operator()

Comparison\_Statements()

Brackets\_Operator()

t = < THEN >

{

jjtThis.jjtSetValue(t.image);

}

Arithmetic\_Statements() // Example: if (x!==5) then y!=5

(

t = < ELSE >

{

jjtThis.jjtSetValue(t.image);

}

Arithmetic\_Statements()

)? // Example: if (x!==5) then y!=5 else y!=3;

}

//--------------------------------------------------------------------------------

void Term() :

{

Token t;

}

{

(

t = < IDEN >

{

jjtThis.jjtSetValue(t.image);

}

| t = < INT >

{

jjtThis.jjtSetValue(t.image);

}

| t = < CONSTANT >

{

jjtThis.jjtSetValue(t.image);

}

)

}

//--------------------------------------------------------------------------------

void Arithmetic\_Operator() :

{

Token t;

}

{

(

t = < PLUS >

{

jjtThis.jjtSetValue(t.image);

}

| t = < MINUS >

{

jjtThis.jjtSetValue(t.image);

}

| t = < MULTIPLY >

{

jjtThis.jjtSetValue(t.image);

}

| t = < DIVIDE >

{

jjtThis.jjtSetValue(t.image);

}

| t = < POWER >

{

jjtThis.jjtSetValue(t.image);

}

| t = < ASSIGN >

{

jjtThis.jjtSetValue(t.image);

}

)

}

//--------------------------------------------------------------------------------

void Comparison\_Operator() :

{

Token t;

}

{

(

t = < EQUAL >

{

jjtThis.jjtSetValue(t.image);

}

| t = < NOTEQUAL >

{

jjtThis.jjtSetValue(t.image);

}

| t = < LESSTHAN >

{

jjtThis.jjtSetValue(t.image);

}

| t = < GREATERTHAN >

{

jjtThis.jjtSetValue(t.image);

}

)

}

//--------------------------------------------------------------------------------

void Logical\_Operator() :

{

Token t;

}

{

(

t = < AND >

{

jjtThis.jjtSetValue(t.image);

}

| t = < OR >

{

jjtThis.jjtSetValue(t.image);

}

)

}

//--------------------------------------------------------------------------------

void Data\_Type() :

{

Token t;

}

{

(

t = < FLOAT >

{

jjtThis.jjtSetValue(t.image);

}

| t = < LIST >

{

jjtThis.jjtSetValue(t.image);

}

| t = < STR >

{

jjtThis.jjtSetValue(t.image);

}

| t = < CONSTANT >

{

jjtThis.jjtSetValue(t.image);

}

)

}

//--------------------------------------------------------------------------------

void Negation\_Operator() :

{

Token t;

}

{

(

t = < NOT >

{

jjtThis.jjtSetValue(t.image);

}

)

}

//--------------------------------------------------------------------------------

void Brackets\_Operator() :

{

Token t;

}

{

(

t = < LEFT\_PARANTHESIS >

{

jjtThis.jjtSetValue(t.image);

}

| t = < RIGHT\_PARANTHESIS >

{

jjtThis.jjtSetValue(t.image);

}

| t = < LEFT\_BRACKET >

{

jjtThis.jjtSetValue(t.image);

}

| t = < RIGHT\_BRACKET >

{

jjtThis.jjtSetValue(t.image);

}

)

}