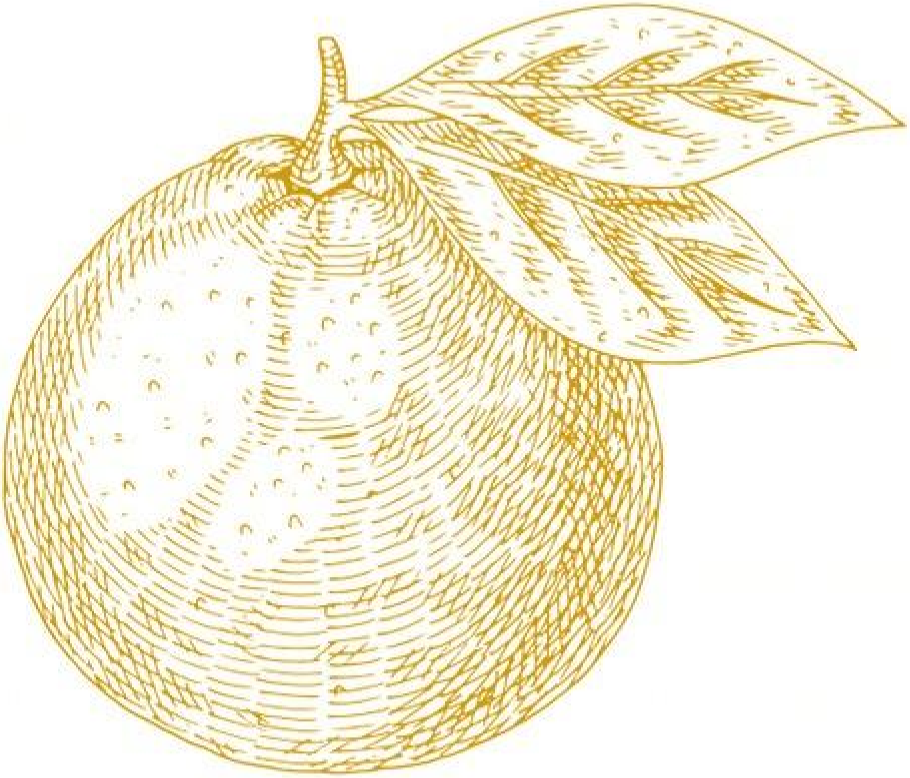


Project Name: Orange Compiler
Due Date: **Month Day Year**
Student Name: Humberto Rendon Ruiz
Student Number: A01039636

Firma___________________________

Vision/Purpose

This compiler will be the main focus of the book “The Orange Book of Compilers”, which is a book I’m making about compiler development (without the boring stuff). The creation and usage of this compiler should be simple, educational and also enjoyable, where even new programmers can get the hang of it without too many complications.

Main Objective

The language’s main objective is simplifying data analysis. The language will be simple, straightforward and serve as an introduction to data analysis and certain modern ways of analyzing data while also learning about compilers and programming languages in general.

Requirements

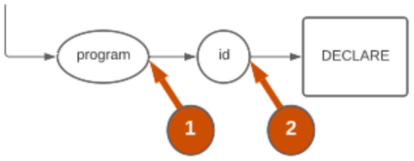
- **Basic elements (Tokens) like keywords, id’s, etc.**

program	‘PROGRAM’	colon	\:	id	[a-zA-Z_][a-zA-Z_0-9]*
main	‘MAIN’	semicolon	\;	cte_float	\-?\d*\.\d+
func	‘FUNC’	comma	\,	cte_int	\-?\d+
void	‘VOID’	lparen	\(	cte_string	\".*\"
return	‘RETURN’	rparen	\)	new_line	\n+
mean	‘MEAN’	lbracket	\[	comment	\#.*
mode	‘MODE’	rbracket	\]	ignore	\t
variance	‘VARIANCE’	lcurly	\{		
histogram	‘HISTOGRAM’	rcurly	\}		
random	‘RANDOM’	assignment	\=		
while	‘WHILE’	equal	\=\=		
for	‘FOR’	not equal	\!\=		
do	‘DO’	gt	\>		
vars	‘VARS’	gte	\>\=		
id	‘ID’	lt	\<		
int	‘INT’	lte	\<\=		
float	‘FLOAT’	and	\&\&		
bool	‘BOOL’	or	\\|\\|		
true	‘TRUE’	plus	\+		
false	‘FALSE’	minus	\-		
input	‘INPUT’	times	*		
print	‘PRINT’	divide	\/		
if	‘IF’				
else	‘ELSE’				

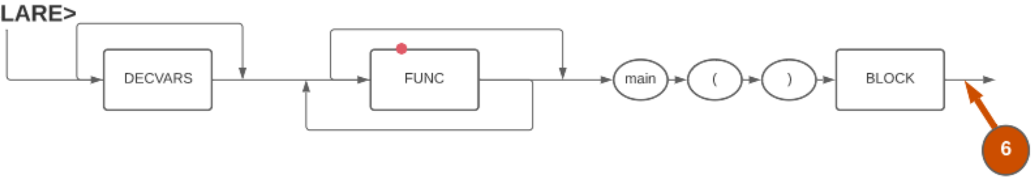
- Syntax Diagrams for all the structures in your language

1. Create DirFunc
2. Add id-name and type program a DirFunc
3. If current Func doesn't have a VarTable then
    a. Create VarTable and link it to current Func
4. Current-type = type
5. Search for id-name in current VarTable
    a. if found -> Error "multiple declaration"
    b. if not, add id-name and current-type to current VarTable
6. Delete DirFunc and current VarTable (Global)

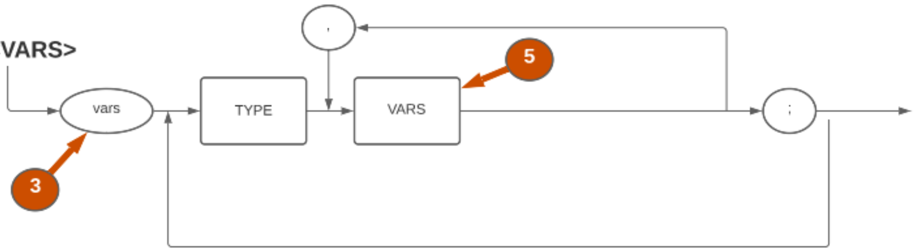
<PROGRAM>



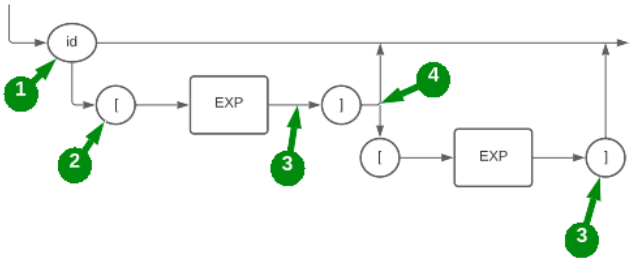
<DECLARE>



<DECVARS>

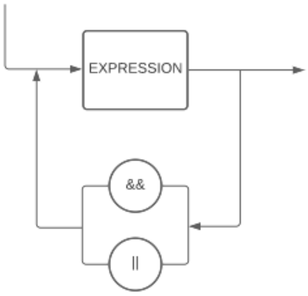


<VARS>

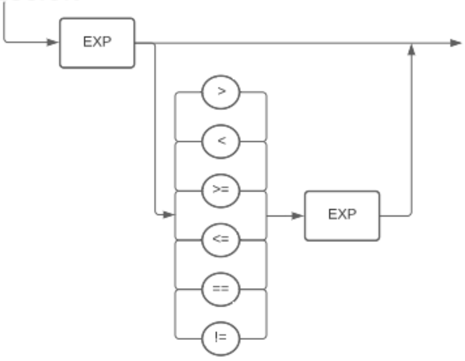


```
1. PilaO.push(id) and PTipos.push(type)
2. id = PilaO.pop(); type = PTipos.pop()
   verify that id has dimensions
   DIM = 1
   PilaDim.push(id, DIM)
   Get the first node of dimensions (list)
   POper.push(FakeBottom)
3. Create Quadruple:
      Verify PilaO.Top LiDIM LsDIM ** NO Pop at this time
   If NextPointer(List)
   { aux = PilaO.Pop()
   Create quadruple: * aux mDIM Tj | where Tj is next available temp
   PilaO.Push(Tj)}
   If DIM > 1
   { aux2 = PilaO.Pop(); aux1 = PilaO.Pop();
   Create quadruple: + aux1 aux2 Tk | where Tk is next available temp
   PilaO.Push(Tk)}
4. DIM = DIM + 1
   Update DIM in PilaDIM
   Move to next node in List
5. Aux1 = PilaO.Pop()
   Create quadruples;
      + aux1 K Ti | where Ti is next available temp. K is the offset
      + Ti VirtualAddress Tn | where Tn is next available temp
   PilaO.Push((Tn)). | Contains an Address, thats why we use () to
   distinguish it
   POper.Pop() | Eliminates fake bottom
```

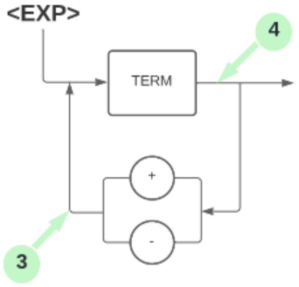
<SUPER_EXP>



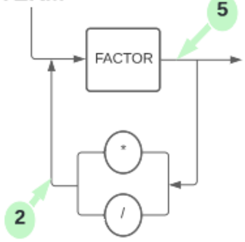
<EXPRESSION>



<EXP>

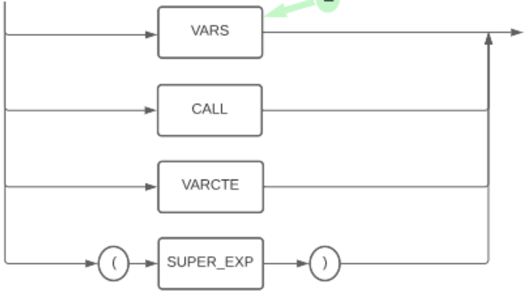


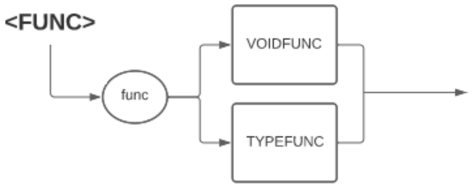
<TERM>

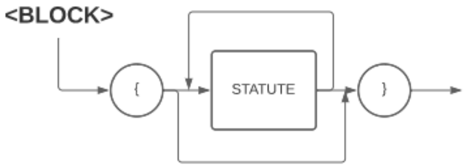


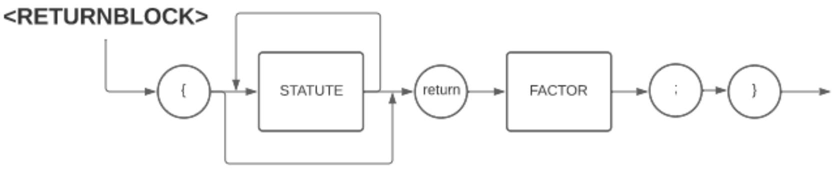
```
1. PilaO.Push(id.name) and PTypes.Push(id.type)
2. POper.Push(+ or -)
3. POper.Push(* or /)
4. If POper.top() == '+' or '-' then
   a.
      i.  right_operand = PilaO.Pop()
      ii. right_Type = PTypes.Pop()
   b.
      i.  left_operand = PilaO.Pop()
      ii. left_Type = PTypes.Pop()
   c. operator = POper.Pop()
   d. result_Type = Semantics[left_type,
      right_type, operator]
   e. if (result_Type != ERROR)
      i.  result <-- AVAIL.next()
      ii. generate quad
         1. (operator, left_op, right_op, result)
      iii. Quad.Push(quad)
      iv.
         1. PilaO.Push(result)
         2. PTypes.Push(result_Type)
      v.  If any operand were a temporal space,
          return it to AVAIL
   f. Else
      i.  ERROR ("Type mismatch")
5. If POper.top()== '*' or '/' then
   a. Sames as 4., but with * and /
```

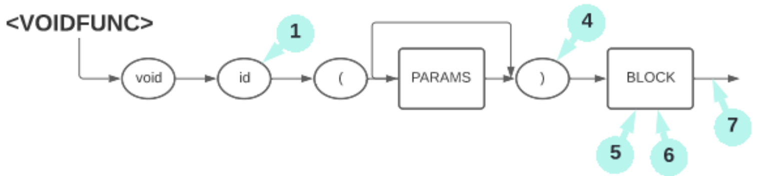
<FACTOR>

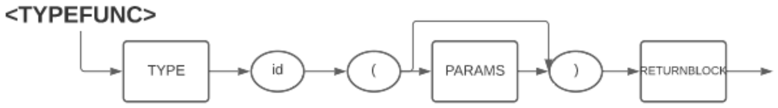




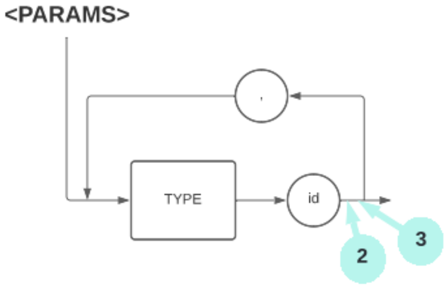




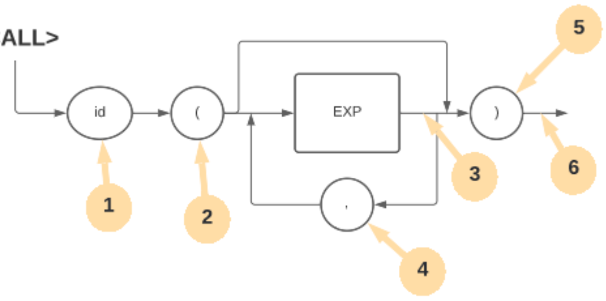




1. Insert Function name into the DirFunc table (and its type, if any), verify semantics
2. Insert every parameter into the current (local) VarTable
3. Insert the type to every parameter uploaded into the VarTable
   At the same time into the ParameterTable (to create the Function's signature)
4. Insert into DirFunc the number of parameters defined. **to calculate the workspace required for execution
5. Insert into DirFunc the current quadruple counter (CONT), **to establish where the function starts
6. Insert into DirFunc the current quadruple counter (CONT), **to establish where the function starts
7. Release the current VarTable (local)
   Generate an action to end the function (ENDFunc)
   Insert into DirFunc the number of temporal vars used. **to calculate the workspace required for execution

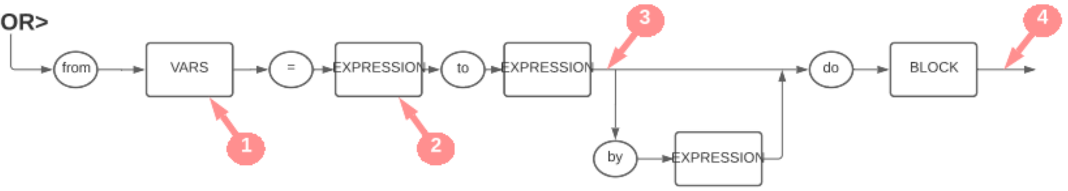


<CALL>



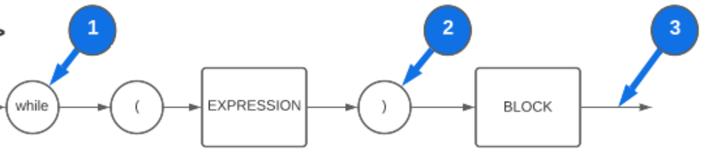
```
1. Verify that the function exists into the DirFunc
2. Generate action ERA size (Activation Record
   expansion -NEW- size)
   Start the parameter counter (k) in 1
   Add a pointer to the first parameter tpye in the
   ParameterTable
3. Argument = PilaO.Pop() ArgumentType =
   PTypes.Pop()
   Verify ArgumentType against current
   Parameter (#k) in ParameterTable
   Generate action PARAMETER, Argument,
   Argument #k
4. K = K + 1, move to next parameter
5. Verify that the last parameter points to null
   (coherence in number of parameters)
6. Generate action GOSUB, procedure-name, ,
   initial-address
```

<FOR>



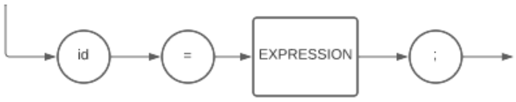
```
1. PilaO.push(id) PTypes.push(tipo id)
   Validar que tipo sea numerico,
   si no es ERROR (Type-mismatch)
2. Exp_type = PTypes.Pop()
   if(Exp_type != numerico)
   ERROR(Type-mismatch)
   else
      Exp = PilaO.Pop()
      VControl = PilaO.Top();
      Control_type = PType.pop()
      Tipo_res = Semantica[=, Control_type,
   Exp_type]
   Si Tipo_Res == ERR --> ERROR
   (Type-mismatch)
   else
      Genera (=, Exp, VControl)
3. Exp_type = PTypes.Pop()
   if (Exp_type != numerico)
   ERROR(Type-mismatch)
   else
      Exp = PilaO.Pop()
      Genera(=, Exp, , VFinal)
      Genera(<, VControl, VFinal, Tx)
      PJumps.push(cont-1);
      Genera(GotoF, Tx, , __)
      PJumps.push(cont-1);
4. Genera (+, VControl, 1, Ty)
   FIN = PJumps, Pop()
   RET = PJumps.Pop()
   Genera (Goto, RET)
   FILL (FIN, cont)
   idOriginal: PilaO.Pop();
   Genera(=, VControl, , idOriginal)
```

<WHILE>

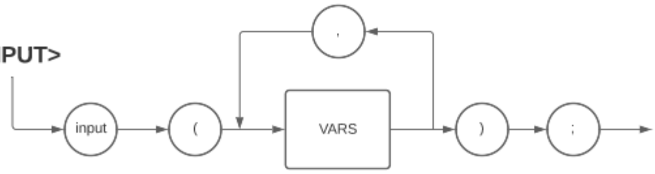


```
1. PJumps.Push(cont)
2. exp_type = PTypes.Pop()
   if (exp_type != bool) ERROR (Type-mismatch)
   else
      result = PilaO.Pop()
      Generate quad: GotoF, result, , ___
      PJumps.Push(cont-1)
3. end = PJumps.Pop()
   return = PJumps.Pop()
   Generate quad: GOTO return
      FILL(end, cont)
```

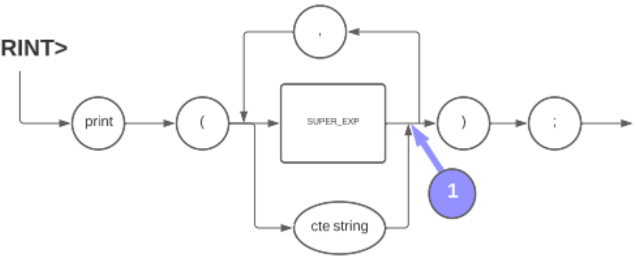
<ASSIGNMENT>



<INPUT>

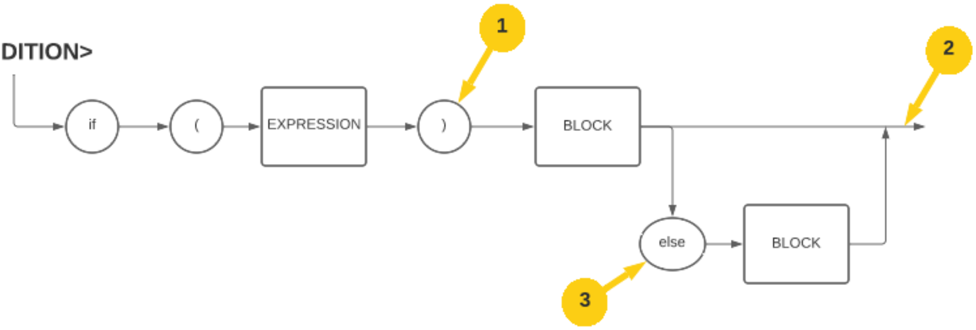


<PRINT>



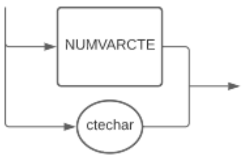
```
1. Resultado = pop(PilaO)
   a. genera cuadruplo < print, , , resultado >
```

<CONDITION>

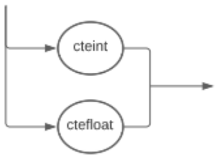


```
1. exp_type = PTypes.Pop()
   if(exp_type != bool) ERROR(type-mismatch)
   else
       result = PilaO.Pop()
       Generate quad: GotoF, result, , ___
                              PJumps.Push(cont-1)
2. end = PJumps.Pop()
   FILL (end, cont)
3. Generate quad: GOTO ___
   false = PJumps.Pop()
   PJumps.Push(cont-1)
   FILL(false, cont)
```

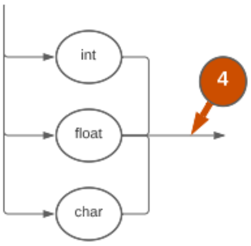
<VARCTE>



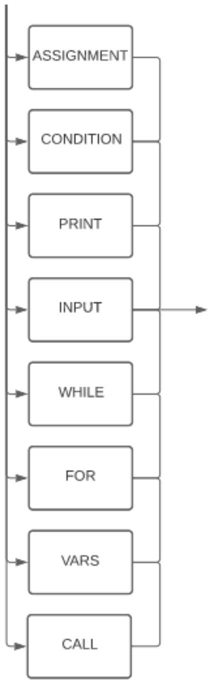
<NUMVARCTE>



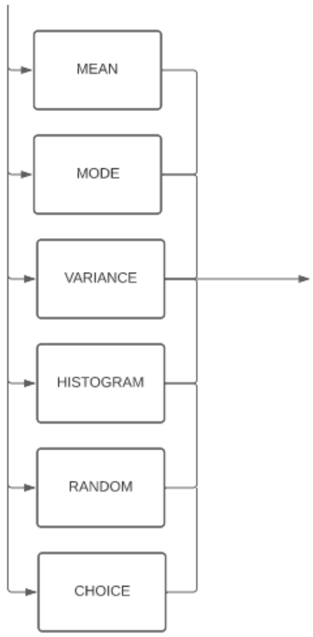
<TYPE>



<STATUTE>



<SPECIALFUNCS>



- **Main Semantic characteristics**

- Italic indicate optional sections
- Bold indicates a reserved word
- # indicates a comment

The general structure for the Orange Language is:

program program_name
<*Global variable declaration*>
<*Function declaration*>

# Main section
main()
{
        <*Statutes*>
}

Variable declaration (it has global and local variables):

vars
        type id ;
        type id, id2, id3, id4 ;
        type id[5] ;
        type id[5][5];
        type id, id2[5], id3[5][5], id4 ;
# type could be int, float, char

Function declaration (0 or more):

```
func <return type> function_id (<parameters>)
{
        <Local variable declaration>
        <Statutes>
        <Return block>
}
# return block is optional depending on the function type
# recursive function calling will be supported
# return type depends on type or void (no value returned)
```

Assignment:

```
id = expression ;
id = function_name(<parameters>)  ;
id = function_name(<parameters>) + id - cte_int ;
# an id could be assigned the value of an expression
# an id could be assigned the returned value of a function
# an id could be assigned a combination of both
```

Void function call:

```
function_name(<parameters>) ;
# a function without a return value is called
```

Function return:

```
return expression ;
# Return value only if the function has a return type
```

Read:

```
input (<id, id2, id3, ...>) ;
# Read one or more IDs separated by commas
```

Write:

```
print (expression | ”string”, expression | ”string”, ...) ;
# Print a combination of an expression or a string separated by commas
```

Decision statute:

```
if (expression)
{
        <Statutes>
}
<else
{
        <Statutes>
}
```

```
>
# A conditional decision with an optional else statement
```

Conditional repetition:

```
while(expression)
{
          <Statutes>
}
# A conditional cycle based on a met or unmet expression

do
{
          <Statutes>
} while (expression) ;
# Triggers first repetition cycle regardless of the expression being met
```

Unconditional repetition:

```
from id = exp to expression <by num> do  {
          <Statutes>
}
# Repeats statutes from N to M in increases of 1
```

Arithmetic expressions:

```
+, -, * , /
# Sum
# Substraction
# Multiplication
# Division
```

Logical expressions:

```
&&, ||
# And
# Or
```

Relational expressions:

```
>, >=, <, <=, ==, !=
# (GT) Greater than
# (GTE) Greater than or equal
# (LT) Less than
# (LTE) Less than or equal
# (EQ) Equal
# (NEQ) Not equal

# These expressions are handled with traditional priorities
# Priorities could be altered by using parenthesis
```

- **Brief description of every special functions as well as rarely used instructions in your language (most of them are related to the area of your language)**

  Mean: mean(array[25], 5, 7.78)
  A function that returns the mean of a one dimensional array and/or constant numerical values (integers or floating point numbers)

  Mode: mode(array[25], 5, 7.78)
  A function that finds the most frequent value or values in an array

  Variance: variance(array[25], 5, 7.78)
  A function that returns the variance of an array of numeric values and/or individual numeric values

  Histogram: histogram(array[25])
  A function that plots a histogram from an array of values determining the frequency of said values

  Random: random(start, finish)
  A function that generates a random number from a starting point to a finishing one

  Choise: choice(subset[5], reps)
  From a subset of possible choices, we extract random repetitions of said choices

- **Data types**

  Id (Usable):
  Users can define an ID for their variables

  Reserved word (Usable)
  Reserved words can be used for their intended purpose

  Int (Usable)
  Integers can be used as constants, return types, etc.

  Float (Usable)
  Floating point numbers can be used as constants, return types, etc.

  String (Partially Usable)
  Strings can only be used in a print statement

  Bool (Usable)
  Booleans can be declared and used for conditions, but they will not represent 0s and 1s

Language and OS that will be used for development

The Orange Compiler will be built using Python as a programming language and Linux (Ubuntu 22.04) as the operating system.

Bibliography

https://numpy.org/
https://pandas.pydata.org/
https://www.dabeaz.com/ply/
https://ply.readthedocs.io/en/latest/
Compilers : principles, techniques, and tools / Alfred V. Aho ... [et al.]. -- 2nd ed.
p. cm.
Rev. ed. of: Compilers, principles, techniques, and tools / Alfred V. Aho, Ravi
Sethi, Jeffrey D. Ullman. 1986.
ISBN 0-321-48681-1 (alk. paper)
1. Compilers (Computer programs) I. Aho, Alfred V. II. Aho, Alfred V.
Compilers, principles, techniques, and tools.
QA76.76.C65A37 2007
005.4'53--dc22