# →Script - Language definition

María Alejandra Pestana Viso A00824367

Armando Roque Villasana A01138717

> 💡 *Sections in italics are optional*
> **Bold words** are **reserved** for the language
> Comments are indicated by /→

## General language structure

```
program Prog_name;

<class declarations>
<global vars>
<function declarations>

/-> comments

main()
{
  <statements>
}
```

### Variable declarations (can be local or global)

```
var <-
  <type> ids;
  <type> ids;
  [...]
->
```

→ For the type

→ Acceptable types:

```
int | float | char | Class_name
```

→ For the ids

- List of id separated by commas

→ For each id

- Can have 1 or 2 dimensions
- Must start with lower case letter, followed by letters (lower and capital) and numbers
- Can follow camelCase or snake_cas

→ Examples:

```
Person id1, id2;   /-> two objects of the Person class
int id3, id4[2];   /-> a single-valued variable and an array of size 2
float id5[3][5];   /-> matrix of 3 rows and 5 columns
```

→ Multi-dimensional structures (arrays and matrixes) can only be declared with *int_cte* (integer constants)

- But they can be indexed or calles using variables (see below)

## Class declarations (0-n)

```
class Class_name extends Father_class_name
{
  attributes
    <vars>

  methods <-
    <funcs>
  ->
}
```

→ *Father_class_name* → is the class from which inherits

→ All attributes and methods are public

→ For each Class_name

- Must start with capital case letter, followed by letters (lower and capital) and numbers
- Can follow CamelCase or snake_case

## Function declarations (0-n)

```
<return_type> func func_name( <params> )
<vars>
{
  <func_statements>
}
```

→ Support recursion

→ For func_name

- Must start with lower case letter, followed by letters (lower and capital) and numbers
- Can follow camelCase or snake_case

→ Examples:

→ For vars

- Same format specified in Variable declarations

→ For the return_type

- If it is not provided, then the function does not return anything (that is, it is void)

→ Accept any of:

```
int | float | char | void
```

→ For the params

- Same format as *<vars>*
- Only provide variables with initial values

## Function Statements

```
<statements>

<return_statement>
```

→ For return_statement

```
return <expression>;
```

# Statements

## Assignment

```
<var> = <expression>;
```

Assignment with return value from function

```
<var> = func_name( <params> ) <expression>;

<var> = object_name.func_name( <params> ) <expression>;
```

→ For var

→ Accept any of:

```
var_name | var_name[2] | var_name[n][m] |
object_name.attribute
```

→ For params

- List of param separated by commas
- For each param

→ Accept any of:

```
<expression>
```

## Functions

### Calling void functions

```
func_name( <params> );

object_name.func_name( <params> );
```

→ For params

- List of param separated by commas
- For each param

  → Accept any of:

```
<expression>
```

**As long as the expression resolved to the specified variable type, then it is accepted*

## I/O

### Input (Read)

```
read(<var_names>);
```

→ For var_names

- List of var_name separated by commas
- For each var_name

  → Accept any of:

```
var_name | var_name[2] | var_name[n][m] |
object_name.attribute
```

**Output (Write)**

```
print(<output_vars>);
```

→ For output_vars

- List of output_var separated by commas
- For each output_var

  → Accept any of:

```
string_cte

<expression>

<var_name>
```

→ For string_cte

- Letters, digits, and spaces (tab, newline, or whitespace) enclosed by **" "**

→ For var_name

→ Accept any of:

```
var_name | var_name[2] | var_name[n][m] |
object_name.attribute
```

## Control *statements*

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

```
{
  <statements>
}
```

## Iteration *statements*

### While loop

```
while(<expression>)
{
  <statements>
}
```

→ Repeat *statements* while expression evaluates to true

### For loop

```
for(<numeric_assignment> until <expression>)
{
  <statements>
}
```

→ Repeat *statements* while expression evaluates to true

→ Adds 1 to the variable inside numeric_assignment on each iteration

→ For numeric_assignment

```
<numeric_var_name> = <numeric_expression>
```

→ For numeric_var_name

- Any <var_name> that is of <numeric_type>

→ For numeric_expression

- Any <expression> that returns or evaluates to a number of <numeric_type>

→ For numeric_type

→ Accepted any of:

```
int
```

### Expressions

Traditional expressions like those in C and Java are accepted.

Can contain:

- Arithmetic operators

  ```
  + | - | * | /
  ```

- Relational operators

  ```
  > | < | == | !=
  ```

- Logical operators

  ```
  & | |
  ```

→ Can be any var_name

→ Accept any of:

```
var_name | var_name[2] | var_name[n][m] |
object_name.attribute
```

## → Sample Program

```
program AS_Program;

class Person
{
  attributes <-
    int age;
```

```
    char name[30];
  ->

  methods <-
    int func one(var <- int x; ->)
    {
      return (age - x);
    }
  ->
}

var <-
  int i, j, p;
  Person student;
->

int func fact(var <- int x; ->)
var <-
  int y;
->
{
  y = x + (p - x * 2 +  j);
  if (x == 1) {
    return (x);
  }
  else {
    return (x * fact(x - 1));
  }
}

func pelos(var <- int z; ->)
var <-
  int k;
->
{
  read(student.age);
  k = z;
  while(k < 10) {
    print(student.one(k));
    k = k + 1;
  }
}

main()
{
  read(p);
  j = p * 2;
  i = fact(p);
  read(student.name);
  hairs(p);
  for (i = 1 until 10)
  {
    print("HelloWorld", student.name, fact(student.age));
  }
}
```