UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTY OF INFORMATICS

LANGUAGE PROCESSORS

PANA

Rubén de Mora Losada
Natalia Rodríguez Caballero
Double Degree in Computer Science and Mathematics

March 2024

Contents

1 Introduction **2**

2 Language specification **3**

2.1 Comments . . . 3

2.2 Identifiers and areas of definition . . . 3

2.3 Types . . . 5

2.4 Language instructions . . . 7

2.5 Expressions in our lenguaje . . . 9

3 Examples **11**

3.1 Fibonacci . . . 11

3.2 Read - write . . . 11

3.3 Person . . . 11

3.4 Arrays . . . 12

3.5 Student . . . 12

1 Introduction

Welcome to the new and revolutionary programming language PANA. This innovative programming language is aimed at those people interested in getting started in the world of programming, without the need to have any previous knowledge of this interesting world. For the design of PANA we have been inspired by the C++ programming language and it could be observed that there is a great similarity with it in the general structure of a correctly written program in our language. However, as it is aimed at beginners in the world of programming, we have modified instructions, use of operators and declarations to make it, in our opinion, more intuitive. The following are some of the most relevant features of our language:

- Whenever a new function or variable is declared, the type should be specified first. The type of the variable indicates the type of the value that will be associated with the variable in question, while the type of the function will determine the expected type of the value returned by the function. If the variables and functions are defined correctly, they will be used to be instantiated during the execution of our program. More specific aspects of function and variable declaration will be discussed throughout this document.
- In our language, variables of integer (num), real (rial) and boolean (fact and infact) types may be declared. If a variable is declared and not initialised, it is assigned the default value of 0, regardless of the type of the variable.
- In every piece of program, a function named 'main' must be defined. As in C++, it will be the function that is initially executed, it will not receive parameters and will always return the value 0. In addition, we will have console write and read functions for boolean variables, integers and real numbers.
- The detection and display of errors shall be made as simple as possible. The row, the column and the type in which the detected error is found shall be indicated. We shall detect errors by means of grammar in any type of declaration and instruction (we shall recover at the next semicolon).

2 Language specification

2.1 Comments

In our language, single-line comments are indicated by '**'. For separate multi-line comments, '/**' will be placed at the beginning of the first commented line. to end the block of multi-line comments, ' **/' will be placed at the end of the first commented line. The following are examples of single and multi-line comments:

```
**This is a comment
/**
This is also a comment
**/
```

2.2 Identifiers and areas of definition

Declaration of simple variables

To declare a single variable, it is sufficient to indicate the type of the variable and then the variable identifier. For example:

```
type id;
```

To declare a variable and initialise it at the same time we will use the operator '− >', an example of this is:

```
 type id -> value;
```

Declaration of arrays

For the case of array declaration, we will make use of the keyword 'list' and we will change the usual C++ syntax, in fact, the syntax for the declaration of arrays will be similar to the one we use to declare simple variables. First, we will indicate the type of the array (in our language, all the elements of the same array must be of the same type), then we will use the reserved word mentioned above followed by the initial size between parentheses and finally the name of our array. Furthermore, in the case of an n-dimensional array we can indicate the size of each dimension by separating each value by ':' (which must be integers, expressions are not allowed).

```
type list (size) id1;
type list (size1 : size2 : ...) id2;
```

On the other hand, to declare arrays with a given initial value, we will use the operator '− >' as we use for simple variables, followed by the initial value in square brackets. Using this syntax, we obtain an n-dimensional array with the same value in each of the positions.

If we wanted to assign different values to each position (although it can be very cumbersome) we can use, after '− >', the operators '{ }'. Inside these operators we will put the value that we will assign to each position sequentially, that is to say, the first value will correspond to the first element of the first dimension, the second value to the second element of the first dimension and so on until the first dimension is filled, once we have finished with the first dimension we will continue with the same procedure until we finish with all the positions of each dimension. Again, to separate the values of different positions we will use ':'.

```
type list (size) id1 -> [value];
type list (size1 : size2 : ...) id2 -> [value];
type list (size1 : size2 : ... : sizeN) id3 -> {value0 : value1 : ... :valueM};
```

Declaration of functions

Similarly to the declaration of arrays and simple variables, to declare a function we first indicate its type and then use the reserved word 'fun' followed by the name of the function. In addition, we include a new type in our language, used only for functions, the void type. These functions simply do not return anything, unlike the rest that use the reserved word 'return' followed by the value to be returned (which has to be of the type of the function).

To start the code of each declared function, we will use the operator '{', after which all the functionality of the function will be written. We will indicate the end of the code of our function with the opposite operator '}'

On the other hand, to distinguish the input of parameters by value and by reference, we will use the keyword 'ref, for the case of parameter by reference. Again, to separate several parameters in the function input, we will use ':'.

```
type fun id (type1 ref param1 : type2 param2 : ... : typeN paramN){
    **function body
}
```

On the other hand, our language needs to have a special main function, which would be of type num, and in case there is no failure in execution it should return the value 0. This function will be in charge of calling the rest of the declared auxiliary functions.

```
num main(){
    **main code
    return 0;
}
```

Declaration of structs

The usual C++ structs are created using the reserved word 'reg' followed by the name of the struct. In this case, the type is not put before the reserved word as a record itself has no type. The identifiers declared in the struct are declared between braces and the same syntax as above will be used, whether we give the variables initial value or not.

```
reg regName {type1 id1 : type2 id2 -> value : ...};
```

To access each of the struct variables we will use the '.' as in C++, being able to give a new value to the variables, as follows:

```
type2 x -> regName.id2 ** this will set the variable x to the value of the id2 field of
    reg Name
regName.id1 -> 3 **this will set the id1 field of reg Name to the value of 3 (this is
    only possible if id1 has the same type as 3).
```

Declaration of alias

For the alias declaration we use the following syntax, making use of the reserved words 'typedef' and 'as' together with the type and alias chosen for it:

```
typedef num list (2:3) as array
```

Declaration of pointers

PFor the declaration of pointers we use the reserved word pointer to avoid confusion with the binary operator *. First, we indicate the type of the variable we want to point to and then we use pointer followed by the variable name.

To initialise the dynamic memory we use the allocation operator we have been using and the reserved word new followed by the type of the variable..

Finally, to access the address of a variable, we use the infix operator $ before the variable identifier.

```
bool pointer x;
num list (10 : 7) pointer ar;

ar -> new num list (10 : 7);
pointer x -> fact;

num a -> 4;
num pointer b;
b -> $a;
num pointer c -> $a;
```

In addition, we allow the initialisation of these pointers, of any of the types mentioned above. The initialisation is performed in a manner consistent with non-pointer variables, the only difference being that these values are stored on the heap instead of on the stack.

```
bool pointer x -> new bool (fact);
num list (2:3) pointer arr -> new num list ({1 : 2 : 3 : 4 : 5 : 6});
rial list (4) pointer arr2 -> new rial list ([2,7]);
```

2.3 Types

Basic types(Integer, real, array and boolean)

The basic **integer** type shall be declared by the reserved word 'num'.**Real** numbers shall be created by the word 'rial'. Real numbers are distinguished from integers by the fact that they are decimal. To separate the integer part from the decimal part we use the ',' operator.

Moreover, the **boolean** type shall be declared by the reserved word 'bool'. In addition, the possible boolean values shall be 'fact' for True and 'infact' for False.

```
num id -> 14;
bool id -> fact;
bool id -> infact;
rial id -> 12,75;
```

As mentioned in the section on declaring simple types, the array type has a reserved word 'list'. To declare lists of lists, it would not be necessary to concatenate the word 'list' several times as in C++. It would be enough to put the keyword only once and it would be in the declaration of the size where the dimensions of our array would be indicated, since it would have as many dimensions as values provided. In our language, the indexing of the arrays will start with 1.

Infix operators

The infix operators used in arithmetic and Boolean expressions will be the usual ones except for division, power and AND and logical OR. For division, we will distinguish the use of '/' for division resulting in an integer from the use of '//' for division resulting in a real number. For the power of numbers we will use 'ˆ'. The logical OR operator could be used by using '@' while the AND operator would use a single '&'.

To access array positions, we use the '[ ]' operator. operator, putting between the two square brackets the index of the position we want to access. As mentioned above, the indexing of arrays starts with 1, so for a one- dimensional array of size n we can use the '[ ]' operator in the following way:

```
miArray[i] **access to position i of the array, 1 <= i <= n
```

For n-dimensional arrays we use the same operator and separate the different elements with a colon to access a specific position in the array. In addition, it is mandatory to put as many values as the array has dimensions because we treat them as a matrix:

```
miArray[i:j] **access to element j of row i
```

To declare negative numbers we use the unary operator '-' before the number in question.

For completeness with the use of '− >' we will change the usual form of C++ operators '+=', ' -= ', etc and replace it with '+− >', '−− >', etc.

Finally, we include as an infix operator ':' to separate elements in arrays, structs, function arguments, etc.

The priority between operators is shown in the table below.

La prioridad entre operadores se muestra en la tabla siguiente.

Operator	Priority	Type	Associativity
:	0	Binary	Left
− > +− > −− > ∗− > /− > %− >	1	Binary	Non-associative
@	2	Binary	Left
&	3	Binary	Left
!	4	Unary	Non-associative
== !=	5	Binary	Non-associative
<= >= < >	6	Binary	Non-associative
+ −	7	Binary	Left
∗ / // %	8	Binary	Left
$	9	Unary	Non-associative
pointer	9	Unary	Non-associative
.	10	Binary	Right

Table 1: Infix operators

2.4 Language instructions

Read and write

To read and write to the screen, we create some specific functions. As we have 3 types of variables (bool, num and rial), we create 3 different functions to read and write respectively: one for booleans (readBool() and writeBool()), one for integers (readNum() and writeNum()) and one for reals (readRial() and writeRial()). For the write type, we enter by parameter what we want to be written to the screen in the following way:

```
num id1 -> readNum();
bool id2 -> readBool();
rial id3 -> 12,56;
writeRial(id3);
```

ToInt and toFloat

In order to change the type of a numeric expression from integer to decimal or vice versa, we have created these two functions that will allow us to perform operations between integers and decimals that may be of interest to us. However, it is important to emphasize that even if these functions are applied to a variable, the value of the variable will only change in that instruction, subsequently it will maintain the type it had before doing anything.

```
num x -> 40;
rial y -> toFloat(x);
rial nota -> 7,3;
num notaTruncada -> toInt(nota); ** will have a 7
```

Conditionals

As in any language, we will have conditional instructions to evaluate boolean expressions and, if the condition evaluates to true, execute the next block of code or, if it evaluates to false, perform a jump to the end of the conditional instruction.

We will use the usual names to refer to conditional statements. Whenever you want to use conditional statements, you should start with 'if. After the if, the boolean expression that we want to evaluate to decide whether to continue with the block of instructions or to make a jump must appear between parentheses. Finally, some braces will appear and inside them will be the content of the block of instructions to be executed in case the evaluation of the 'if' is true.

Optionally, we can use the 'else if' and 'else' clauses. They will be used in the same way as the 'if' in terms of the use of parses for the Boolean expression (actually in the case of the 'else' we will see that it is not necessary to use parses, since no Boolean expression is evaluated) and the use of braces for the delimitation of the corresponding code block. If the evaluation of the 'if' is true, neither an 'else if' nor an 'else' immediately following that 'if' will ever be executed. The 'else if' is executed when the 'if' has evaluated to false and every 'else if' (if any) are also executed to false. If the condition of the 'else if' evaluates to true, the code in the braces will be executed and no subsequent 'else if' or 'else' will ever be executed. Finally, if the 'if' and all the else if's are false, we will reach the else (if it exists) and execute the code between braces.

```
if (cond1){
    **If cond1 is true, we never get to evaluate cond2 or execute else
}
else if(cond2){
    **** We get to evaluate cond2 if cond1 evaluates to false

}
else if(cond3){
    **** cond3 is evaluated if cond2 ( and thus cond1 ) are false

}
else{
    ** If we get this far, it is because cond1 , cond2 and cond3 are false. We then
    execute the code
between braces
}
```

It should be noted that the 'else if' and the 'else' are "coordinated" with the previous 'if' to them, i.e. if we have two 'if' in a row and then an 'else if' and an 'else', we will arrive at the 'else if' or the 'else' if the Boolean expression of the second 'if evaluates to false (regardless of the value of the evaluation of the first 'if').

```
if(cond1){

}
if(cond2){**always executed, regardless of cond1

}
else if(cond3){
/**executed only if cond2 evaluates to false and independently of the value of cond1**/

}
else{**if cond2 and cond3 evaluate to false, irrespective of the value of cond1

}
```

Loops

Our language is going to have three types of loops, 'while' and 'for' loops (usual ones), and one additional type:

- **While loops**. For the creation of these loops we only need the condition that must be fulfilled during the duration of the loop and the body that will be executed while this happens. As in C++, this condition is checked at the beginning of the loop, if it is fulfilled, the body is executed and the loop condition is returned to the loop condition, and if not, the loop execution is terminated. It would be written as follows:

```
while (condition) {
    **while body
}
```

- **For loops**. To create the for loops, we follow the scheme of the for of C++, we will have an iterator to which we will give an initial value, a condition to fulfil and a change will be applied in each turn of the loop.

  These three fields will be separated by ':'. In the first one we can use a variable that already had a value before or create a new one by giving it the desired initial value as in the example.

  In the second field we will have a condition that must be fulfilled during the loop execution, being checked at the beginning of each loop. An example of this could be to check that our iterator has not reached a certain limit value.

  Finally, in the third field there will be an arithmetic expression that will normally affect our iterator.

```
for (num it -> 0 : bool condition: it +-> 1){
    **for body
}
```

- **Loop loops**. Finally, we create a type of loop that we do not have in C++. These loops simply simplify something that can be done with the previous ones, but because it is something very recurrent, it is quite considerable to make this type more simplified. It is a loop in which we simply indicate the number of iterations to be performed, without any additional condition other than not having performed those iterations yet. We will write it as follows:

```
loop (num iterations){
    **loop body
}
```

2.5 Expressions in our lenguaje

Expressions in our language include the following:

- **Constants**. In our language only numbers, both integers and reals, are understood as constants, and in the case of booleans `fact` and `infact`.
- **Variables**. Variable identifiers with or without initial value. In the case of arrays, subindices may be used to indicate a specific position in the array. These subindices are indicated by square brackets.

- **Infix operators**. To perform all kinds of arithmetic operations, on the same variable or on several variables, we have the infix operators mentioned above.
- **Function calls**. This shall consist of the identifier of the desired function followed by parentheses containing the various arguments necessary for the development of the function body, separated by ':'.

```
functionId (param1 : param2 : ... : paramN);
```

3 Examples

3.1 Fibonacci

In this example we calculate the Fibonacci function for all numbers between 0 and 10, so we can see one way of looping in our language (for).

```
num fun fibonacci (num x) {
    num sol;
    if (x == 0 @ x == 1) {
        sol -> 1;
    }
    else{
        sol -> fibonacci(x-1) + fibonacci (x-2);
    }
    return sol;
}

num main(){
    for (num it -> 0 : it <= 10 : it+->1){
        writeNum(fibonacci(it));
    }

    return 0;
}
```

3.2 Read - write

In this example we pass to the readWrite function a real and we read by console the number of times we will write the real on the screen. Here we can see how we read an integer on the screen and another type of loops (while), as well as showing an infix operator that differs from those defined in C++.

```
void fun readWrite (rial r){
    num count -> 0;
    num n -> readNum();
    while (count < n){
        writeRial(r);
        count +-> 1;
    }
}

num main() {
    readWrite(3,14);

    return 0;
}
```

3.3 Person

Here we show how to deal with structs in our language. We create the struct with one of the variables it contains initialised and for the rest we take the values by console.

```
num main() {
    reg person {num id -> 52489263 : num phone : num age : rial height : rial weight :
    bool married};

    person.phone -> readNum();
    person.age -> readNum();
    person.height -> readRial();
    person.weight -> readRial();
    person.married -> readBool();

    return 0;
}
```

3.4 Arrays

In this example we show how arrays work in our language. To do so, we create a function to which we pass an integer, which will be the number of positions of the array. For each of the positions of the array we will calculate the square of the number of the position, that is to say, in the i-th position we will have the number i^2. The way to calculate it is with the special loop of our language that sums i times the quantity i.

```
num main() {
   num size -> 10;
   num list (size) array;

   for(num i -> 0 : i < size : i +-> 1){
        num x -> 0;
        loop (i){
            x +-> i;
        }
        array[i] -> x;
   }

   return 0;
}
```

3.5 Student

In this last example we want to show the functionality of the pointers as well as other new functions that we have defined. As we can see, the record of type student only occupies 12 bytes on the stack when using the heap to store the list of notes. From main we create a new instance of this type of record and we use the initializeNotes function to enter the student's notes by keyboard, in addition this function returns the average truncated to an integer

```
reg alumno {num edad : bool genero /** 0 if boy, 1 if girl**/ : rial list (10) pointer
    notas};

typedef alumno as aliasAlumno;

typedef rial list(10) pointer as aliasArray;

rial fun mejoresN(rial list(10) lista : num N){
    rial sol;
    rial less -> 100,1;
    for(num i -> 1 : i <= (N + 1) : i +-> 1){
```

```
    sol +-> lista[i];
    if (less > lista[i]){
      less -> lista[i];
    }
    writeRial(lista[i]);
      }
      sol --> less;
      return sol/toFloat(N);
  }

  rial fun inicializarNotas(aliasArray calificaciones){
      num i -> 1;
      rial media;
      loop(4){
          rial x -> readRial();
          (pointer calificaciones)[i] -> x;
    writeRial((pointer calificaciones)[i]);
          media +-> x;
          i +-> 1;
      }
      return mejoresN( (pointer calificaciones) : 3);
  }

  num main(){
      aliasAlumno ruben;
      aliasArray list(6) array;
      readNum();
      (array[1]) -> new rial list(10);
      (pointer (array[1]))[6] -> 10,8;
      ruben.genero -> infact;
      ruben.notas -> new rial list(10);
      rial z -> inicializarNotas(ruben.notas);
      writeRial(z);
      return toInt(z);
  }
```