C+−

Juan Diego Barrado Daganzo, Javier Saras González and Daniel González Arbelo
4th year

Índice

1. Technical specifications of the language

1.1. Identifiers and scope of definition

The language has the following characteristics:

- **Variable declaration**: simple variables of the defined types and *array* variables of these types, of any dimension, can be declared.
- **Nested blocks**: nesting is allowed in conditionals, loops, functions, etc. If two variables have the same name, the deepest one (in the nesting) overrides the outermost one.
- **Functions**: the creation of functions is allowed. Passing by value and by reference of any type to functions is guaranteed.
- **Pointers**: for each type, a pointer to a variable of that type can be declared by assigning its memory address to the pointer variable.
- **Structs and classes**: two additional types are included: structs as "data bags" —without methods— and classes, with both data and function methods.
- **Declaration of constants**: the possibility of declaring constants by the user is included.

1.2. Types

The C+−language is a strongly typed language, where type declarations must be made explicitly and prior to the use of the identifier, i.e., in order to use a variable, I must have declared it beforehand. The language consists of the following predefined types:

- Integers
- Booleans
- Classes
- Records
- Arrays
- Functions
- Pointers

which are discussed in more detail in Typing. It is also possible for the user to declare types using the `difain` statement. Again, the details are explained in more detail in the corresponding section.

1.2.1. Integers and booleans

The basic types of language are integers and Booleans. Below is the declaration syntax for both and the operations allowed for each. The identifier `var` corresponds to the identifier assigned to the variable declared with these types.

- **Integers**: `int var;` Operations enabled for the type:
  - Addition: `a + b`
  - Subtraction: `a - b`
  - Multiplication: `a * b`
  - Division: `a / b`
  - Power: `a^b`
  - Parentheses: `()`
  - Less than: `a < b`
  - Greater than: `a > b`

- Equal to: `a == b`
- Less than or equal to: `a <= b`
- Greater than or equal to: `a >= b`
- Not equal: `a != b`

The literals allowed for assigning a value to integer variables are decimal, binary, and hexadecimal, which will all be converted internally to decimal numbers.

- **Booleans**: `bul var;` Operations enabled for the type:
  - Logical *AND*: `a an b`
  - Logical *OR*: `a or b`
  - Logical *NOT*: `!a`

  The reserved words for defining the two Boolean literals that can be assigned to Boolean variables are `tru` and `fols`, indicating true and false, respectively.

1.2.2. Classes and structs

Classes and records are non-basic types composed of attributes (records) and attributes and methods (classes). In addition, each element of the class or record may have a visibility to be chosen between `public` and `praivet`, the former denoting that the field is accessible by calls from outside the type and the latter that it is only accessible through internal calls of the type. By default, classes will have all their fields private and records public.

- **Classes**: `clas var {...};`

  Options enabled for the type:
  - Access to attributes: `var.attribute`
  - Access to methods: `var.method()`
  - Constructor: `var(args)`
- **Structs**: `estrut var {...};`

  Options enabled for the type:
  - Access to attributes: `var.attribute`
  - Constructor: `var(args)`

  In both cases, the prefix `dis` in references to attributes, i.e., `dis.field`, will allow you to distinguish between the field with that name within the type or the local variable that may have the same name.

1.2.3. Arrays

All types described in this subsection can form a multidimensional array. Array declarations are static, meaning that the size must be known at compile time.

- **Array**: `Type[DIMENSION] var;`

  Options enabled for the type:
  - Access operator: `var[INDEX]`

1.2.4. Functions

The functions, which at first glance might not appear to be a type, have also been declared as one in order to be able to create lambda expressions that can then be passed as arguments to other functions or used in the execution of the program.

- **Functions**: `func foo(Type arg1, ...) : ReturnType {...; return var;};`

If the function does not return any value—which is usually called a *procedure*—the declaration syntax would be changed to

- **Functions**: `func foo(Type arg1, ...) {...; return;};`

The final `return` statement is mandatory, and only one is allowed. Furthermore, it must be the final statement in the body of the function.

As for function parameters, it is possible to pass parameters by value or by reference, with the former being the default option. To explicitly indicate that you want to pass a parameter by reference, you must add the character "&" at the end of the argument type.

Arrays can also be arguments of a function and, unlike primitive types, they are passed by reference by default. It should be noted that it is possible to pass an array of variable size as an argument. In this case, it will be written similarly to the following statement:

```
func foo(int[] array) : ReturnType {...}
```

1.2.5. Pointers

Any of the above types, including arrays themselves, can be pointed to by a specific pointer. The declaration of pointers is like the declaration of the type to be pointed to, but with the character ~ added at the end of the type.

- **Pointer**[1]: `Type~ var`

  Options enabled for the type:

  - Dynamic memory association: `var := niu Type`
  - Data access: `~var`

The value stored in the pointer is the memory address of the object it points to, which may be a stack or heap address, depending on the case.

1.2.6. User-defined types and constants

In addition to the types declared by us, we allow user-defined types to be defined using the following statement:

- **Definition of user types**: `taipdef newType typeExpression`.

[1] For clarification, the statement "The declaration of pointers is like the declaration of the type to be pointed to, but adding the character ~ at the end of the type" remains exactly the same when the type is not a simple type. For example, the declaration of a pointer to an array of any type would be written as `Type[DIMENSION]~ var;`.

However, the way to understand this user-declared type is not as a new type but as an alias for the type expression to which it was assigned.

The declaration of literal constants is permitted and implemented using the following statement:

- **Declaration of constants**: `difain NAME value`

Note that the above expression does not explicitly indicate the type to which the constant belongs, but is implicitly known through the value assigned to the constant identifier.

1.3. Language statements

The following is a list of language statements. Pay attention to those that end with the character ";" to delimit their end:

- **Assignment statement**: `:=`

```
int var = 3;
```

- **Conditional statement**: `if-els`, `suich-queis`

```
if (var > 3) {
    ...
}
els {
    ...
}
```

```
suich (var) {
    queis(val1):
        ...
        breic;
    queis(val2):
        ...
        breic;
    difolt:
        ...
        breic;
}
```

- **Loop statement**: `while`, `for`

```
guail (var > 0) {
    ...
}
```

```
for (int i = 0; i < n; i = i + 1) {
    ...
}
```

The statements `breic` and `continiu` are also included.

- **Input-output statements**: `cein()` and `ceaut()`

```
cein(var);
ceaut(var);
```

- **Function return statement**: `return`

```
func foo(Tipo arg) : ReturnType {
    ReturnType var;
    ...
    return var;
}
```

1.4. Code structure elements

The code will begin with the definition—separate or intermingled—of composite types (classes, records, functions, etc.), user types, and program constants. Finally, the main function of the program will begin, serving as the entry point, which we denote by `func mein() : int {...; return 0;}`.

It is important to note that, unlike other languages such as Python, where code execution does not require a "special" function to serve as the entry point for the program, in our language it is mandatory that the program code be inside this final function and that only the declaration of types and constants that we have explained previously be allowed outside of it. This means that any global variable or expression outside of `mein` that does not comply with these restrictions will be understood as a syntactic error.

On the other hand, it includes the possibility of writing comments in the code, that is, lines that will be ignored at compile time. Both single-line comments, preceded by "//" and terminated by a line break, and multi-line comments, preceded by "/*" and terminated by "*/" are allowed.

Below is an example of code that complies with the stipulated restrictions:

```
func mein() : int {
    // The execution of the program begins here
    int var = 3;
    /*
    The next function call computes
    the factorial of the variable var.
    We store the value in the variable res
    to show it on screen afterwards.
    */
    int res = factorial(var);
    ceaut(res);
    return 0;
}
```

2. Lexical specification of the language

During the previous explanation of the language specifications, some of the reserved words or spellings used to perform certain operations have already been presented. Below, the elements of the language and the symbols reserved for each purpose are explained in more detail.

- Variable identifier names must be alphanumeric expressions that do not begin with numbers and may contain the character "_".
- The literals allowed for integers are decimal (e.g., `var = 10`), binary (e.g., `var = 0b1001`), and hexadecimal (e.g., `var = 0x1F2BC`).
- The allowed literals for booleans are the reserved words `tru` and `fols`.
- Whitespace, tabs, and line breaks are removed internally during lexical recognition.
- Single-line comments begin with // and multi-line comments with /**/. These elements are removed during lexical recognition.
- The reserved words for defined types and their operators are those specified in the corresponding sections of Types.
- The delimiters for function bodies and statements `guail`, `if`, etc. are curly braces {}.
- The delimiter for the end of an statement or block in those that require it is ;.
- The word `niu` is reserved for heap memory allocation.
- The words `taipdef` and `difain` are reserved for the definition of types and constants.
- The words and symbols reserved for the statement repertoire are those declared in the section Language statements.
- The word reserved for the main function of the program is `main`.

The regular expressions that define the elements of the lexicon described here can be found in the document *lexicon.l*.

3. Semantics

The following shows the semantics of the language associated with the syntactic construction that generates it. Due to space limitations, not all semantic definitions are included, only the most relevant ones. For the complete list, please refer to the *Tiny.cup* document.

Constructor	Prog : Declaration × MainFunction → Program
Description	Build a program from a series of definitions and a main function. `mein`.
Syntax	`Definitions && mein`

Constructor	classDef : String × ClassBody → Class
Description	Build a class definition given a name and a class body.
Syntax	`clas id { classBody }`

Constructor	structDef : String × SturctBody → Struct
Description	Build a record definition given a name and a record body.
Syntax	`estrut id { structBody }`

Constructor	funcDef : String × Type × FunctionBody → Struct
Description	Build a function definition with return type given a name and a function body.
Syntax	`func id { cuerpoStruct }`

Constructor	typeDef : String × String × String → Type
Description	Build a type definition as an alias.
Syntax	`taipdef previousType newType;`

Constructor	decVar : Type × String → Declaration
Description	Builds a variable declaration of the specified type.
Syntax	`Type id;`

Constructor	Asign Ins : Id × Expression → Statement
Description	Builds an assignation sentence.
Syntax	`Id = Expression;`

Constructor	If Ins : Expression × Block → Statement
Description	Builds the conditional `If` statement.
Syntax	`If (Expression)` {`Block`}

Constructor	While Ins : Expression × Block → Statement
Description	Builds the iterative `While` statement.
Syntax	`guail (Expression)` {`Block`}

Constructor	Asign Ins : Id × Expression → Statement
Description	Builds the assignation statement.
Syntax	`Id = Expression;`

Constructor	New Ins : Id × Expression → Statement
Description	Builds the dynamic memory allocation `New` statement.
Syntax	`Id = niu Expression;`

3.1. Binding

Binding is the association between an identifier and the object it designates (variables, constants, functions, etc.). We distinguish between identifiers introduced with their definition (in a variable declaration, a constant, a function definition, or type aliases, etc.) and occurrences of use (all other occurrences).

We relate the occurrences of an identifier's use with its definition occurrence by traversing the AST, connecting each usage node with a reference to its corresponding definition node, using a stack of auxiliary symbol tables that stores the definitions of the symbols, taking into account the nesting levels of the program (each table associated with a scope).

As for user-defined types (typedef, class, and struct), we associate them with a table that stores the name (String) and the root definition (AST node of the definition). We apply a reserved word policy with the definitions that are declared to prevent those same type names from being used to declare new structs or classes. As for type aliases, these reserved words only prevent them from being used as aliases for another type that has already been created. In turn, we take advantage of the data structure we have so that aliases are associated with the original definition node of the type (and to also be able to take advantage of this construction with primitive types, we first put them in the definition table as if every program had those basic "user-declared" types implicitly).

We have a prior declaration policy for any use of identifiers: in order to use them, they must first be declared. In the case of recursive functions, as they are within the definition of the function, they are already considered to be declared.

C+- supports overloading by arguments, same function name, and different argument types (regardless of the return type). Therefore, during binding, we associate each use of a function identifier with a list of possible functions, and the specific function will be determined at type checking.

We have visibility of variable declarations according to the scope in which they are located: the innermost link hides the outermost one.

3.2. Typing

As mentioned in section 1.2, this language is strongly typed. Thus, the type of all elements of the code is known at compile time. As explained in that section, we allow the declaration of constants with the clause `difain`. These constants also have a type that is known at compile time: it is not a simple alias, but if it cannot be typed as one of the types allowed in the language, the compiler will interrupt and warn the user.

In general, compatibility between different types does not exist; you cannot make assignments between different types or work with them indistinguishably.

However, there is one exception to this rule, which is pointers and arrays. These two types are compatible, as long as their internal types are compatible. Thus, when declaring a variable of type `int[5]` and another of type `int~`, it is possible to assign values between them. However, this would not be possible if the array type were `bul[5]`, for example.

This design decision raises another requirement: arrays are useless if they cannot be iterated over. Therefore, a method must be enabled to "iterate over pointers" when we know that they point to an array. For this reason, the language overloads the binary operators `+` and `-`. This way, we can iterate over a pointer by using an statement such as `ptr = ptr +/- 1`, which provides behavior similar to that of an iterator as provided by higher-level languages. It should be noted that this is the only exception regarding these operators: it is allowed to operate a pointer or array with an integer (and, in the case of subtraction, the integer must be the right operand). These operators cannot be used for other cases. It should be noted that, to move forward or backward with a pointer, `+ 1` or `- 1` will always be used, regardless of the size of the internal type.

Below is an example of what we have mentioned:

Listing 1: Example of using arrays and pointers in equivalent ways.

```
func mein(): int {
    int array[5];
    for (int i = 0; i < 5; i = i + 1) {
        array[i] = i;
    }
    int~ ptr = array;
    for (int i = 0; i < 5; i = i + 1) {
        ceaut(array[i]);
        ceaut(~ptr);
        ptr = ptr + 1;
    }

    return 0;
}
```

In the previous example, a list of 5 integers is created, each position containing its own position as a value. After this, a pointer is declared that "points" to the same list, and finally another loop is iterated, printing the value of each position in the array along with the value of the pointer. The result of the execution will be the sequence `0011223344`, as expected, since what we are doing is basically printing each value of the array twice in a row.

3.2.1. Types in functions

By default, parameters are passed by value in function calls. However, passing parameters by reference is allowed by adding the `&` operator to the argument in the function declaration. Internally, this corresponds to passing a pointer instead of the value. To be clearer, when a parameter is passed by reference to a function, internally a pointer to the variable in question is being passed, but within the body of the function it will be interpreted as what it is, a pointer, and operations will be executed accordingly. In this way, we allow the user to use and modify the actual value of the parameters, but we do not ask them to handle the data differently than they normally would.

In addition to everything mentioned above, we must specify a particular case, which is that of arrays. In the case of arrays, explicit pass-by-reference is not allowed; rather, it is the default. In other words, arrays are not passed by copy. This is done for two reasons: the first is that it is generally very expensive. It involves unnecessary expenditure of time and memory, and doing it this way does not prevent the user from doing anything; they can make copies within the function. The second is that it brings consistency to the passing of dynamic arrays. In our language, arrays are allowed to have no defined size as a function parameter, and their handling is different from that of normal arrays. In particular, the most reasonable thing in this case is for modifications to the array within the function to translate into changes outside it, and forcing this to be the default behavior makes everything more homogeneous and understandable.

Finally, similar to forcing arrays to be passed to functions by reference, among other things, due to the cost involved, the return type of functions is also restricted, so that any type except arrays is allowed as a return type. If you want to do something similar to returning an array as the return value of a function, you must create it before calling the function and pass it by reference to the function. In this way, the size of the function's "return" array is not restricted; it is sufficient to declare the argument as a dynamic array, i.e., without a defined size.

A. Examples of common programs

Below are a series of simple examples of how certain programs would be written in our language. However, there is a wide variety of sample code to try out in the project's *test* folder.

Listing 2: Example of common $C+-$ program.

```
func mein() : int {
    int a;
    int b;
    cein(a);
    cein(b);
    int c = a + b;
    ceaut(c);
    return 0;
}
```

Listing 3: Example of iterative $C+-$ program.

```
func maximo(int lista[], int tam) : int {
    int maximo = -1;
    int i = 0;
    guail(i < tam) {
        if(lista[i] > maximo) {
            maximo = lista[i];
        }
        i = i + 1;
    }
    return maximo;
}
```

Listing 4: Example of $C+-$ multidimensional programs.

```
difain tam 5;

func multiplicarMatrices(int A[][], int B[][]) : int~~ {
    int~~ res;
    int C[5][5];
    res = C;
    int i = 0;
    int j;
    guail(i < tam) {
        j = 0;
        guail(j < tam) {
            C[i][j] = 0;
            j = j + 1;
        }
        i = i + 1;
    }
    int k;
    i = 0;
    guail(i < tam) {
        j = 0;
        guail(j < tam) {
            k = 0;
            guail(k < tam) {
                C[i][j] = C[i][j] + A[i][k] * B[k][j];
                k = k + 1;
            }
            j = j + 1;
        }
        i = i + 1;
    }
```

```
    return res;
}
```

Listing 5: Example of recursive program in $C+-$ using `if-els`.

```
func factorial(int num) : int {
    int res;
    if(num == 0){
        res = 1;
    }
    els {
        res = num * factorial(num - 1);
    }
    return res;
}
```

Listing 6: Example of structs and the `suich` sentence in $C+-$.

```
estrut Alumno {
    int DNI;
    int nota;

    Alumno(int DNI, int nota) {
        dis.DNI = DNI;
        dis.nota = nota;
    }
}

func aprobado(Alumno a) : bul {
    bul res;
    suich (a.nota) {
        queis(0):
        queis(1):
        queis(2):
        queis(3):
        queis(4):
            res = fols;
            breic;
        queis(5):
        queis(6):
        queis(7):
        queis(8):
        queis(9):
        queis(10):
        difolt:
            res = tru;
            breic;
    }
    return res;
}

func nota(Alumno~ a) : int {
    return (~a).nota;
}

func mein():int {
    Alumno daniel = Alumno(123456789, 6);
    Alumno~ danielptr = &daniel;
    ceaut(nota(danielptr));
    ceaut(aprobado(daniel));

    return 0;
}
```