C+−

Juan Diego Barrado Daganzo, Javier Saras González y Daniel González Arbelo
4º de Carrera

16 de abril de 2024*

Índice

*Este documento se actualiza, para consultar las últimas versiones entrar en el enlace https://github.com/JuanDiegoBarrado/PracticaPL

1. Especificaciones técnicas del lenguaje

1.1. Identificadores y ámbitos de definición

El lenguaje posee las siguientes características:

- **Declaración de variables**: se pueden declarar variables sencillas de los tipos definidos y variables *array* de estos tipos, de cualquier dimensión.
- **Bloques anidados**: se permiten las anidaciones en condicionales, bucles, funciones, etc. Si dos variables tienen el mismo nombre, la más profunda (en la anidación) tapa a la más externa.
- **Funciones**: se permite la creación de funciones. El paso por valor y por referencia de cualquier tipo a las funciones está garantizado.
- **Punteros**: para cada tipo se puede declarar un puntero a una variable de ese tipo, mediante la asignación de su dirección de memoria a la variable puntero.
- **Registros y clases**: se incluyen dos tipos adicionales: los registros como "saco de datos" —sin métodos— y las clases, tanto con datos como con métodos de función.
- **Declaración de constantes**: se incluye la posibilidad de declarar constantes por parte del usuario.

1.2. Tipos

El lenguaje C+−es un lenguaje fuertemente tipado, donde la declaración de tipos ha de hacerse de manera explícita y de forma previa al uso del identificador, es decir, que para poder usar una variable tengo que haberla declarado antes.

El lenguaje consta de los siguientes tipos predefinidos:

- Enteros
- Booleanos
- Clases
- Registros
- Arrays
- Funciones
- Punteros

sobre los que se habla más en profundidad en su respectiva sección. Asimismo, existe la posibilidad de declarar tipos por parte del usuario a través de la instrucción `difain`. Los detalles, de nuevo, se exponen de forma más precisa en su correspondiente apartado.

1.2.1. Enteros y booleanos

Los tipos básicos del lenguaje son los enteros y los booleanos y son los únicos permitidos para ser retornados explícitamente[1] por funciones. A continuación, se expone la sintaxis de declaración de ambos y las operaciones permitidas para cada uno de ellos. El identificador `var` corresponderá al identificador asignado a la variable declarada con estos tipos.

[1] Si bien es cierto que la instrucción `return` está restringida a tipos enteros y booleanos, una función puede "devolver" un valor de otro tipo pasando por referencia como argumento una variable de ese tipo.

- **Enteros**: `int var;`

  Operaciones habilitadas para el tipo:

  - Suma: `a + b`
  - Resta: `a - b`
  - Multiplicación: `a * b`
  - División: `a / b`
  - Potencia: `a^b`
  - Paréntesis: `()`
  - Menor: `a < b`
  - Mayor: `a > b`
  - Igual: `a == b`
  - Menor o igual: `a <= b`
  - Mayor o igual: `a >= b`
  - Distinto: `a != b`

  Los literales permitidos para la asignación de un valor a las variables de tipo entero son decimales, binarios y hexadecimales, que serán convertidos internamente todos a números decimales.

- **Booleanos**: `bul var;`

  Operaciones habilitadas para el tipo:

  - *Y* lógico: `a an b`
  - *O* lógico: `a or b`
  - *No* lógico: `!a`

  Las palabras reservadas para definir los dos literales booleanos que pueden asignarse a variables booleanas son `tru` y `fols`, indicando respectivamente verdadero y falso.

1.2.2. Clases y registros

Las clases y registros son tipos no básicos compuestos por atributos (registros) y por atributos y métodos (clases). Además, cada elemento de la clase o del registro podrá tener una visibilidad a elegir entre `public` y `praivet`, denotando el primero que el campo es accesible por llamadas desde fuera del tipo y el segundo, que solo es accesible a través de llamadas internas del tipo. Por defecto, las clases tendrán todos sus campos privados y los registros públicos.

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

  Opciones habilitadas para el tipo:

  - Acceso a atributos: `var.atributo`
  - Acceso a métodos: `var.metodo()`
  - Constructor: `var(args)`

- **Registros**: `estrut var {...};`

  Opciones habilitadas para el tipo:

  - Acceso a atributos: `var.atributo`
  - Constructor: `var(args)`

  En ambos casos, el prefijo `dis` en las referencias a atributos, es decir, `dis.campo` permitirá distinguir entre el campo con dicho nombre dentro del tipo o la variable local que pudiese tener el mismo nombre.

1.2.3. Arrays

Todos los tipos descritos en esta subsección pueden formar un array multidimensional. La declaración de arrays es estáctica, esto es, que el tamaño se debe conocer en tiempo de compilación.

- **Array**: `Tipo[DIMENSION] var;`
  Opciones habilitadas para el tipo:
  - Operador de acceso: `var[INDEX]`

1.2.4. Funciones

Las funciones, que a priori podría parecer que no son un tipo, se han declarado también como uno para poder hacer expresiones lamdba que luego poder pasar como argumento a otras funciones o utilizar en la ejecución del programa.

- **Funciones**: `func foo(Tipo arg1, ...) : TipoRetorno {...; return var;};`

En caso de que la función no devuelva ningún valor —lo que se suele llamar *procedimiento*—, la sintaxis de declaración se cambiaría por

- **Funciones**: `func foo(Tipo arg1, ...) {...; return;};`

La instrucción `return` final es obligatoria y solo está permitido que haya una y, además, que sea la instrucción final del cuerpo de la función.

En cuanto a los parámetros de la función, existe tanto la posibilidad de pasar los parámetros por valor como por referencia, siendo la primera la opción por defecto. Para indicar explícitamente que quiere pasar un parámetro por referencia, es necesaria añadir el caracter "&" al final del tipo del argumento.

Los arrays también pueden ser argumentos de una función y, a diferencia de los tipos primitivos, su paso por defecto es por referencia. Cabe destacar que existe la posibilidad de pasar como argumento un array de dimensión variable. En tal caso se escribirá similar a la siguiente sentencia:

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

1.2.5. Punteros

Cualquiera de los tipos anteriores, incluyendo los propios arrays, son susceptibles de ser apuntados por un puntero específico. La declaración de punteros es como la declaración del tipo al que se pretende apuntar, pero añadiendo el caracter ˜ al final del tipo.

- **Puntero**[2]: `Tipo˜ var`
  Opciones habilitadas para el tipo:
  - Asociación a memoria dinámica: `var := niu Tipo`
  - Acceso al dato: `˜var`

El valor guardado en el puntero es la dirección de memoria del objeto al que apunta, pudiendo ser esta una dirección de la pila o del heap en función del caso.

[2] A modo de aclaración, la afirmación "La declaración de punteros es como la declaración del tipo al que se pretende apuntar, pero añadiendo el caracter ˜ al final del tipo" se mantiene exactamente igual para cuando el tipo no es un tipo simple. Por ejemplo, la declaración de un puntero a un array de cualquier tipo, se haría como `Tipo[DIMENSION]˜ var;`.

1.2.6. Tipos definidos por el usuario y constantes

Además de los tipos declarados por nosotros, permitimos la definición de tipos por parte del usuario a través de la siguiente instrucción:

- **Definición de tipos de usuario**: `taipdef nuevoTipo expresionDeTipos`.

Sin embargo, la manera de entender este tipo declarado por el usuario no es como un nuevo tipo sino como un alias de la expresión de tipos a la que se asignó.

La declaración de constantes literales está permitida e implementada a través de la siguiente instrucción:

- **Declaración de constantes**: `difain NOMBRE valor`

Nótese que la expresión anterior no indica de manera explícita el tipo al que pertenece la constante, sino que se conoce de manera implícita a través del valor asignado al identificador de la constante.

1.3. Instrucciones del lenguaje

A continuación se presenta el repertorio de instrucciones del lenguaje. Préstese atención a aquellas que terminan con el caracter ";" para delimitar su final:

- **Instrucción de asignación**: `:=`

```
int var = 3;
```

- **Instrucciones condicionales**: `if-els`, `suich-queis`

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

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

- **Instrucción de bucle**: `while`

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

Se incluyen además las instrucciones `breic` y `continiu`.

- **Instrucciones de entrada salida**: `cein()` y `ceaut()`

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

- **Instrucción de retorno de función**: `return`

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

1.4. Elementos de la estructura del código

El código comenzará con la definición —separada o entremezclada— de los tipos compuestos (clases, registros, funciones, etc.), de los tipos de usuario y de las constantes del programa. Finalmente, comenzará la función principal del programa y que sirve como punto de entrada, que es la que denotamos por `func mein() : int ...; return 0;`.

Es importante destacar que, a diferencia de otros lenguajes como Python, donde la ejecución del código no requiere de una función "especial" que sirva como punto de entrada para el programa, en nuestro lenguaje es obligatorio que el código del programa esté dentro de esta función final y que fuera solo se permita la declaración de tipos y constantes que hemos explicado previamente. Esto quiere decir que cualquier variable global o expresión fuera del `mein` que no se ajuste a estas restricciones será entendida como un error sintáctico.

Por otro lado, se incluye la posibilidad de escribir comentarios en el código, esto es, líneas que en el momento de compilación serán ignoradas. Se permiten tanto comentarios de una línea, encabezados por "//" y terminados en un salto de línea, como comentarios multilínea, encabezados por "/*" y terminados con "*/".

A continuación, un ejemplo de código que se ajusta a las restricciones estipuladas:

```
func mein() : int {
    // La ejecucion del programa empieza aqui
    int var = 3;
    /*
    La llamada a la funcion siguiente calcula el
    factorial de la variable var.
    Almacenamos el valor en la variable res para
    mostrarla por pantalla a continuacion.
    */
    int res = factorial(var);
    ceaut(res);
    return 0;
}
```

2. Especificación léxica del lenguaje

Durante la explicación previa de las especificaciones del lenguaje, ya se han presentado algunas de las palabras reservadas o grafías destinadas a realizar determinadas operaciones. A continuación,

se explica de forma más detallada los elementos del lenguaje y los símbolos reservados para cada fin.

- Los nombres de los identificadores de las variables han de ser expresiones alfanuméricas que no comiencen por números y que posiblemente tengan el caracter "_".
- Los literales permitidos para los enteros son decimales (p. ej. `var = 10`), binarios (p.ej. `var = 0b1001`) y hexadecimales (p. ej. `var = 0x1F2BC`).
- Los literales permitidos para los booleanos son las palabras reservadas `tru` y `fols`.
- Los espacios en blanco, tabuladores y saltos de línea, se eliminan internamente durante el reconocimiento léxico.
- Los comentarios monolínea comienzan por // y los multilínea /**/. Estos elementos se eliminan durante el reconocimiento léxico.
- Las palabras reservadas para los tipos definidos y para sus operadores son las especificadas en los correspondientes apartados de Tipos.
- Los delimitadores de los cuerpos de funciones e instrucciones `guail`, `if`, etc. son las llaves {}.
- El delimitador de final de instrucción o bloque en aquellas que lo requieren es el ;.
- La palabra `niu` es la reservada para la reserva de memoria del heap.
- Las palabras `taipdef` y `difain` son las palabras reservadas para la definición de tipos y constantes.
- Las palabras y símbolos reservados para el repertorio de instrucciones son los declarados en la sección Instrucciones del lenguaje.
- La palabra reservada para la función principal del programa es `mein`.

Las expresiones regulares que definen los elementos del léxico aquí descrito pueden consultarse en la sección **??** del anexo.

3. Especificación sintáctica del lenguaje

Constructora	If Ins : Expresión × Bloque → Instrucción
Descripción	Construye la instrucción condicional `If`.
Sintaxis	`If (Expresión) {Bloque}`

Constructora	While Ins : Expresión × Bloque → Instrucción
Descripción	Construye la instrucción iterativa `While`.
Sintaxis	`While (Expresión) {Bloque}`

Constructora	Asign Ins : Id × Expresión → Instrucción
Descripción	Construye la instrucción de asignación
Sintaxis	`Id = Expresión`

A. Ejemplos de programas habituales

Listing 1: Ejemplo de programa típico en $C+-$.

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

Listing 2: Ejemplo de programa iterativo en $C+-$.

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

Listing 3: Ejemplo de arrays multidimensionales en $C+-$.

```
func multiplicarMatrices(int[][] A, int[][] B, int tam) : int[] {
    int C[tam][tam];
    int i := 0;
    int j;
    guail(i < tam) {
        j := 0;
        guail(j < tam) {
            C[i][j] := 0;
            j++;
        }
        i++;
    }
    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++;
            }
            j++;
        }
        i++;
    }
    return C;
}
```

Listing 4: Ejemplo de programa recursivo y sentencia `if-els` en $C+-$.

```
func factorial(int num) : int {
    if(num == 0){
```

```
        return 1;
    }
    els {
        return num * factorial(num - 1);
    }
}
```

Listing 5: Ejemplo de uso de registros y sentencia `suich` en $C+-$.

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

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

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

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

    return 0;
}
```

B. Expresiones regulares del léxico

```
%eofval}

%init{
  ops = new ALexOperations(this);
%init}

character  = ([A-Z]|[a-z])
positiveDigit = [1-9]
digit = ({positiveDigit}|0)
integerNumber = (({positiveDigit}{digit}*)|0)
binary = 0b[0-1]+
```

```
hexadecimal = 0x([0-9]|[a-f])+
separator = [ \t\r\b\n]
comment = \/\/[^\n]*
// multiLineComment = \/\*[^(\*\/)]*
multiLineComment = \/\*[\s\S]*?\*\/
identificator = ({character}|_)({character}|{digit}|_)*
integerType = int
sumOperator = \+
substractionOperator = \-
multiplicationOperator = \*
divisionOperator = \/
modulusOperator = %
powerOperator = \^
openParenthesis = \(
closeParenthesis = \)
lessOperator = <
greaterOperator = >
equalOperator = \=\=
lessOrEqualOperator = <\=
greaterOrEqualOperator = >\=
notEqualOperator = \!\=
booleanType = bul
andOperator = and
orOperator = or
notOperator = \!
trueValue = tru
falseValue = fols
comma  = ,
colon = :
semicolon = ;
classType = clas
openBracket = \{
closeBracket = \}
fieldAccessOperator = \.
structType = estrut
openSquareBracket = \[
closeSquareBracket = \]
functionType = func
pointerOperator = \~
referenceOperator = \&
newOperator = niu
typedef = taipdef
define = difain
assignationOperator = \=
if = if
else = els
switch = suich
case = queis
break = breic
default = difolt
```

C. Gramática de la sintaxis