C+−

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

25 de marzo de 2024*

Índice

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. Los nombres de los identificadores han de ser expresiones alfanuméricas que no comiencen por números y que posiblemente tengan el caracter "_".

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

- **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 y la declaración implícita dentro de otras. 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

La declaración de tipos ha de hacerse de manera explícita y de forma previa al lugar donde se emplee el identificador, es decir, que para poder usar una variable tengo que haberla declarado antes.

1.2.1. Enteros y booleanos

Los tipos básicos del lenguaje son los enteros y los booleanos. La sintaxis de declaración de estos tipos es la siguiente:

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

  Entre las 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`

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

  Entre las operaciones habilitadas para el tipo:

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

  Además, las palabras reservadas para indicar el valor verdadero o falso son `tru` y `fols`.

1.2.2. Clases y registros

Como tipos adicionales hemos incluido los registros, las clases y las funciones. La sintaxis de declaración es la siguiente:

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

  Entre las operaciones habilitadas para el tipo:

  - Acceso a campos: `var.campo`
  - Acceso a métodos: `var.metodo`
  - Constructor: `var(args)`

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

  Entre las operaciones habilitadas para el tipo:

  - Acceso a campos: `var.campo`
  - Constructor: `var(args)`

  En ambos casos, el prefijo `dis` permitirá acceder específicamente a los atributos de la clase o registro.

  En el caso particular de las clases que, a diferencia de los registros, pueden métodos definidos internamente, permitirá el acceso a estos atributos cuando haya ocultación por el nombre de los argumentos de los métodos.

  Aparte de esto, en ambos casos los atributos podrán tener una visibilidad a elegir entre `public` y `praivet`, con el mismo significado que tienen en C++.

1.2.3. Arrays

Todos los tipos pueden formar un array multidimensional, la sintaxis de declaración es la siguiente:

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

  Entre las operaciones habilitadas para el tipo:

  - Operador de acceso: `var[INDEX]`

1.2.4. Funciones

Las funciones se han declarado también como un tipo para poder hacer expresiones lambda y pasar funciones como argumento. La sintaxis de declaración de una función es la siguiente:

- **Funciones**: `func var(Tipo arg1, Tipo arg2, ...) : TipoRetorno {...};`

El paso de parámetros por defecto es por valor, pero puede cambiarse a por referencia añadiendo el caracter "&" al final del tipo del argumento. Cabe mencionar que se permite como argumento un array de dimensión variable. En tal caso se escribirá similar a la siguiente sentencia:

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

Se añade una restricción al cuerpo de las funciones: se impone que la última sentencia debe ir encabezada por `return`.

1.2.5. Punteros

Pueden declarase punteros a cualquiera de los tipos definidos. La sintaxis de declaración es:

- **Puntero**[1]: `Tipo~ var`
  Entre las operaciones habilitadas para el tipo:
  - Asociación a memoria dinámica: `var := niu Tipo`
  - Acceso al dato: `~var`

1.2.6. Tipos definidos por el usuario y constantes

Adicionalmente, permitimos la definición de tipos por parte del usuario a través de la siguiente palabra reservada:

- **Definición de tipos de usuario**: `taipdef nombre expresion`.

Cabe mencionar que esta expresión no crea ningún tipo nuevo, sino que funciona como un "alias". Se puede entender como una sentencia que reemplaza cada aparición de `nombre` en el código por `expresion`.

Por último, la declaración de constantes se ejecuta a través de la expresión siguiente:

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

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**: `:=`

```
1 int var := 3;
```

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

```
1 if (var > 3) {
2     ...
3 }
4 els {
5     ...
6 }
```

[1] Como nota especial, la declaración de un puntero a estructuras de tipo array, se haría como `Tipo[DIMENSION] var;`.

```
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

De forma idéntica a C++ el código se empezará a ejecutar a partir de la funcíon mein.

De forma también idéntica, 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 "*/".

Se muestra a continuación un ejemplo de código que incluye estos elementos:

```
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;
}
```

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;
}
```