Javier Mulero Martín y Ángela Ruiz Ribera

Resumen

Especificación y funcionamiento del lenguaje `C-`.
Léxico, sintaxis, vinculación, tipos, generación de código y ejemplos.

Índice

1. Introducción

En esta entrega vamos a especificar nuestro lenguaje C-, así como algunos ejemplos típicos de programas escritos en él. Para desarrollarlo, nos hemos basado en C++.

Veremos su léxico y sintaxis, el proceso de vinculación y comprobación de tipos y cómo hemos implementado la generación de código.

2. Estructura de un programa

Como nos hemos basado en C++, nuestros programas tendrán una lista de declaraciones de tipos enumerados, de tipos struct y de funciones. Además, necesariamente tendrá una función principal llamada main.

Los comentarios en C- comienzan con el operador // y terminan con un salto de línea.

- **Enumerados**:

```
enum NombreEnumerado = {Nombre1, Nombre2,..., NombreN};
```

- **Structs**:

```
struct nombreReg {
    // lista de declaraciones [5.1]
};
```

- **Funciones**:

```
tipoRetorno nombreFun(tipo1 arg1, ..., tipoN argN) {
    // lista de instrucciones [5]
    return valorRetorno;
}
```

- **Función `main`:**

```
void main() {
    // lista de instrucciones [5]
    return;
}
```

3. Tipos

Hay declaración explícita de tipo. Veamos a continuación los distintos tipos que hay.

3.1. Tipos básicos predefinidos

- `int`: para representar enteros.
- `float`: para representar números decimales.
- `bool`: para representar `true` o `false`.
- `char`: para representar caracteres.
- `void`: para representar el tipo vacío.
- `string`: para representar cadena de caracteres.

3.2. Tipos definidos por el usuario

- enum: representa un conjunto de valores constantes.

  enum NombreEnumerado = {Nombre1, Nombre2,..., NombreN};

  Por ejemplo:
  enum DiasSemana = {Lunes, Martes, Miercoles, Jueves, Viernes, Sabado, Domingo};

- struct: define una estructura de datos formada por *campos*, que son una o más variables de uno o distintos tipos.

```
struct nombreReg {
    // lista de declaraciones [5.1]
};
```

  Por ejemplo:

```
struct Tiempo{
    int hora;
    int minuto;
    float segundo;
};
```

3.3. Tipo array

Representa una serie de elementos del mismo tipo.
Se identifica por tipo<> o tipo<tamaño>.
Pueden ser multidimensionales: tipo<tamaño1><tamaño2>...<tamañoN>.

3.4. Tipo puntero

Representa una dirección de memoria que almacena un valor.
Se identifica por tipo *.

4. Expresiones y operadores

4.1. Expresiones

Las expresiones en nuestro lenguaje pueden ser:

- Un **acceso** a:
  - Variables: con el nombre de la variable formado por letras, dígitos y barrabajas. Tienen que comenzar por una letra y estar declaradas.
  - Punteros: Si queremos acceder al valor usamos *nombre y si queremos acceder a la dirección de memoria basta con poner el nombre de la variable.
  - Arrays: nombre[indice1][indice2]..[indiceN]
  - Structs: nombre.campo

    Por ejemplo:

```
Tiempo contador;
contador.hora = 12; contador.minuto = 15; contador.segundo = 59.9;
```

Observación 1. *Se permite el anidamiento de distintos tipos de accesos.*

Observación 2. *El acceso más prioritario es el acceso a punteros, y es el único que permite asociación por paréntesis. Ejemplo:*

```
*figura.circulo = 3;
```

```
*(figura.circulo) = 3;
```

El primer ejemplo es equivalente a ~~`(*figura).circulo = 3;`~~ *pero no permitimos esta notación.*

- Una **constante**:
  - de tipo `int`: dígitos, por ejemplo: `20`.
  - de tipo `float`: dígitos separados por punto (.), por ejemplo: `20.03`.
  - de tipo `bool`: `true` y `false`.
  - de tipo `char`: caracteres entre comillas simples (`' '`), por ejemplo `'a'`.
  - de tipo `string`: cadena de caracteres entre comillas dobles (`" "`), por ejemplo `"Adiós mundo"`.
- Una **llamada a una función**: se escribe el nombre de la función seguido de los parámetros entre paréntesis, por ejemplo: `devuelveDia(20, variable_deEjeMpLo)`.
- Un `new tipo()`: para guardar memoria (dinámica) al iniciar los punteros, por ejemplo: si `variable_deEjeMpLo` es de tipo `int*`, `variable_deEjeMpLo = new int();`.

4.2. Operadores

Operador	Tipo	Prioridad	Asociatividad
||	Binario infijo	0	De izquierda a derecha
`&&`	Binario infijo	1	De izquierda a derecha
`==, !=`	Binario infijo	2	De izquierda a derecha
`<, >, <=, >=`	Binario infijo	3	De izquierda a derecha
`+, -`	Binario infijo	4	De izquierda a derecha
`*, /, %`	Binario infijo	5	De izquierda a derecha
`-, !`	Unario prefijo	6	Asociativo
`exp`	Binario infijo	7	De izquierda a derecha

Cuadro 1: Operadores de `C-`. 7 indica la máxima prioridad.

5. Instrucciones

A continuación mostramos las instrucciones de nuestro lenguaje.

Observación 3. ***(Bloques anidados)*** *Se permite anidamiento de instrucciones utilizando* { ... } .

5.1. Declaración

Podemos declarar variables de dos formas:

- No inicializadas: `Tipo nombre;`
  - Ejemplo: `int<7> vector;`
- Inicializadas: `Tipo nombre = expresión;`
  - Ejemplo: `char caracter = 'a';`

5.2. Asignación

Podemos asignar a las variables una expresión: `acceso = expresion;`

5.3. If

Puede ser de una rama:

```
if(expresion_bool) {
    // lista de instrucciones
}
```

o de dos ramas:

```
if(expresion_bool) {
    // lista de instrucciones
} else {
    // lista de instrucciones
}
```

5.4. While

```
while(expresion_bool) {
    // lista de instrucciones
}
```

5.5. For

La declaración del `for` solo puede ser de tipo `int` y esa variable será local al bucle.

```
for(int nombre = exp; exp_bool; asig) {
    // lista de instrucciones
}
```

5.6. Switch

```
switch(nombreVarAComparar){
    case valor1: lista de instrucciones; break;
    case valor2: lista de instrucciones;break;
    default: lista de instrucciones;
}
```

5.7. Llamada a función

Como vimos antes, podemos realizar llamadas a funciones como parte de una expresión. También vamos a permitir que sea una instrucción en sí misma a la que no le interesa el valor de retorno.

```
nombreFun(param1, ... paramN);
```

Los valores se pasan por valor, no por referencia.

5.8. Print

Para mostrar por consola una expresión:

```
print(expresión);
```

Observación 4. *Fácilmente de forma análoga podríamos añadir una función read que leyese de teclado.*

5.9. Return

Para salir de una función devolviendo un valor o ninguno en las funciones `void`:

```
return expresión;
```

```
return;
```

Puede haber varios `return` en una misma función.

6. Gestión de errores léxicos y sintácticos

- Errores léxicos: Se detectan *tokens* no declarados y se cuentan como error, mostrando la fila y la columna donde se ha producido. Vamos a continuar analizando por si hubiese más.
- Errores sintácticos: Indicamos la fila y la columna del mismo y nos recuperamos de él para proseguir la compilación y detectar más. La recuperación de los errores se hace tras el punto y coma.

  Los principales errores sintácticos que detectamos son:

  - Cuando el programa no tiene función `main`.
  - En los `enumerados` y `structs` controlamos cualquier tipo de error y nos recuperamos en en el siguiente ;.
  - En las `funciones` controlamos los errores en la cabecera y el bloque de instrucciones.
  - En las instrucciones controlamos los ; finales y cualquier otro error que pueda haber en ellas. Por ejemplo, cuando faltan paréntesis, cuando hay tipos indefinidos o hay errores de construcción en las condiciones del `if`, `for`, `while` y `switch`.

  Observación 5. *La falta de llaves al crear bloques no se controla en funciones e instrucciones.*

7. Vinculación y tipos

Una vez hecho el análisis léxico y sintáctico, pasamos a hacer el análisis semántico. Lo vamos a realizar en dos fases:

- **Vinculación**: Consiste en asociar un identificador al objeto que designa. Esto se lleva a cabo mediante recorriendo el AST con una pila de tablas que representa los ámbitos. Lo primero que se hace es la vinculación de los tipos `struct` y `enum` y posteriormente de las `funciones`. Para estas últimas, se recorren sus parámetros y luego sus instrucciones.

  Garantizamos lo siguiente:

  - Nos aseguramos de que las variables, y en general los accesos, que utiliza el programa se han definido antes de su uso y se vinculan a sus declaraciones.
  - Nos aseguramos de que los valores de retorno estén vinculados a sus funciones.
  - Nos aseguramos de que las llamadas a funciones estén vinculadas a la función a la que hacen referencia.

  Sobre variables tenemos en cuenta que:

  - Las variables declaradas son visibles en su ámbito.
  - Tomamos una variable declarada en un ámbito en el cual hay un bloque con otra declarada con el mismo nombre. En este caso, en el bloque será visible la más interna, es decir, la que se ha sobreescrito. Si esta última no existiese, en el bloque podríamos ver la externa.
  - En un mismo bloque no se permite la declaración de dos variables con el mismo nombre.

- **Comprobación de tipos**: Nos aseguramos de que los tipos se usan de forma adecuada. Para ello, comenzamos comprobando que la definición de los `structs` sea correcta. Posteriormente, comprobamos las `funciones` chequeando que el tipo de la función y el de sus parámetros son correctos y que cada instrucción también lo es.

  La comprobación de tipos la hacemos de la siguiente manera:

  - Tenemos una serie de tipos básicos que se comparan de forma directa.
  - En los arrays comprobamos que el tamaño y tipo de sus elementos coincide.
  - En los structs realizamos equivalencia estructural y los enumerados los consideramos todos del mismo tipo.
  - Los punteros comprueban que el tipo del elemento que guardan es igual.

  Observación 6. *Hemos realizado la vinculación y comprobación de tipos de* ***todo*** *nuestro lenguaje. No obstante, por falta de tiempo no hemos podido controlar todas las excepciones que saltan en Java cuando hay tipos null.*

8. Generación de código

Antes de generar código realizamos dos cosas:

- Calcular la **etiqueta** de cada declaración, es decir, asignar a cada una un entero que representa el inicio de su posición de memoria. Esto lo realizamos teniendo en cuenta el tamaño que ocupan.
- Calcular el **tamaño máximo de memoria** que ocupa una función. Esto es obtener el tamaño de todas las declaraciones de una función y devolver la suma de ellas.

En los dos pasos anteriores, para ahorrar memoria, tenemos en cuenta que el espacio que ocupan las variables declaradas dentro de un bloque se puede reutilizar cuando se sale de él.

Una vez finalizados esos pasos, generamos el código WebAssembly.
Aceptamos programas constituidos por:

- Una única función `main`.
- Genera código para tipos enteros, booleanos, arrays y structs.
- Puede contener todas las instrucciones que hemos especificado con anterioridad.

Observación 7. *Por falta de tiempo, nuestro compilador tiene algunas limitaciones:*

- *Los programas no pueden tener varias funciones. Por lo tanto, no hemos implementado las llamadas a funciones ni los retornos de las mismas.*
- *No hemos implementando generación de código para punteros, memoria dinámica ni enumerados.*
- *En los arrays, solamente podemos acceder a elementos individuales.*

9. Ejemplos de programas en C-

9.1. Ejemplos de comprobación

En el proyecto hemos incluido 5 ficheros *.txt* para probar todas las partes de la práctica:

- **input1LexSint**
- **input2Errores**
- **input3Vinculacion**
- **input4Tipos**
- **input5Codigo**

Están explicados y se pueden usar para comprobar cosas adicionales.

9.2. Ejemplos generales

Incluimos dos ejemplos generales para que se pueda visualizar mejor la apariencia y funcionamiento de nuestro lenguaje.

- Cálculo del máximo y mínimo de un array.

```
struct Solucion { // definición del struct
        int max; // declaración de variables
        int min;
};

Solucion calcular(int<> vector, int tam) { //declaro función
        Solucion sol;
        sol.max = 0; // Accedo a sus campos y los inicializo
        sol.min = vector[0]; // Accedo a posición del vector e inicializo

        int i = 0;
        while (i < tam) { // Instrucciones while, if y anidamiento de bloques
                if (vector[i] > sol.max) {
                        sol.max=vector[i];
                }
                if (vector[i] < sol.min) {
                        sol.min = vector[i];
                }
                i = i + 1;
        }
        return sol; // valor retorno
}

void main() {
    // declaración de un array de enteros con 7 elementos
        int<7> vector;
        int i = 0;
        while ( i < 7 ){
            vector[i] = i+1;
            i = i + 1;
        }

        Solucion sol; // declaro un struct
    // llamada a función y asigno su valor de retorno a sol
        sol = calcular(vector,7);
        print(sol.min);
        print(sol.max);
        return; // valor retorno de la función
}
```

- Utilización de enumerados.

```
enum TipoFigura = { Cuadrado, Triangulo, Circulo };

void main() {
        TipoFigura figura; // Definimos Figura de tipo enumerado TipoFigura
        figura = Circulo;
        switch (figura) {
            case Cuadrado: print(Cuadrado); break;
            case Triangulo: print(Triangulo); break;
            case Circulo: print(Circulo); break;
            default: print(-1);
        }
        return;
}
```