Manual: C- - From Zero to Hero

Mendez Sevín Melissa
Palmerin Morales David Gabriel

Presentamos un pequeño manual para el lenguaje de programación `C-`.
Se detallan **especificaciones** sobre la escritura en este lenguaje y además su **instalación**.
La estructura global de un programa en este lenguaje se define con *declaraciones*, *funciones* y *sentencias*

1 Declaraciones

Una declaración es una expresión que permite asignar memoria y además podemos identificar tal segmento de memoria con un nombre y un tipo.
En este lenguajes tenemos los siguientes tipos para variables:

- `char`
  Nos permite identificar un caracter.
  El tamaño de este tipo ocupa `8 bits`.
  Ejemplo: `char` c;
- `int`
  Permite identificar variables para números enteros de hasta `32 bits`.
  Ejemplo: `int` entero;
- `float`
  Permite identificar variables para números flotantes de hasta `32 bits`.
  Ejemplo: `float` float_32;
- `double`
  Permite identificar variables para números flotantes de hasta `64 bits`.
  Ejemplo: `double` float_64;
- `struct`
  Permite crear conjuntos de declaraciones de variables las cuales todas ellas pueden ser accesadas por medio del nombre definido para el `struct` en cuestión. Ejemplo:

```
struct{
    int red;
    int green;
    int blue;
} rgb;
```

  En este caso definimos un `struct` llamado rgb; la variable **red** la podemos obtener por medio de: `rgb.red`. En su forma general la obtenemos como: `{nombre_struct}.{nombre_variable}`

Estas son las declaraciones que podemos realizar para nombres de variables con sus respectivos tipos.
Los nombres de las varibales asignado deberán ser parte del lenguaje generado por la siguiente expresión regular:

$$[\_a-zA-Z][\_a-zA-Z0-9]$$

Y además, deberán tener a lo más **32 caracteres**.

2 Funciones

Una parte muy importante de este lenguaje de programación son las funciones.
Una función se encarga de realizar ciertas operaciones y permiten ser identifcadas por un **nombre**, además pueden recibir **argumentos** los cuales pueden ser usados dentro de la definición de la función. Finalmente, también puede **devolver** valores de los tipos `void, char, int, float, double`, donde `void` indica que la función no devolverá ningún valor, y únicamente es válido usarlo en definiciones de funciones.
La definición de una función es de la siguiente forma:

$$\texttt{func tipo nombre } (\texttt{tipo } arg_1, \ldots, \texttt{tipo } arg_n)\{\}$$

Donde la **cantidad de argumentos** es mayor o igual a cero.
Es muy importante notar que los **nombres son únicos**, es decir el nombre declarado no se repite con ninguna otra función declarada así como con ninguna otra variable declarada anteriormente.

Cada función tiene la estructura de colocar **declaraciones al inicio** y **posteriormente colocar sentencias**. Una función puede no tener declaraciones dentro, sin embargo siempre debe existir al menos una sentencia. Siempre debe existir la función `main` y debe ser la última en definirse.

Expresiones

Existen **expresiones** básicas para operar, las cuales son las siguientes:

- **Suma**
  Reliza la suma de dos expresiones
  `expresión` + `expresión`
- **Resta**
  Reliza la resta de dos expresiones
  `expresión` − `expresión`
- **Multiplicación**
  Reliza la multiplicación de dos expresiones
  `expresión` ∗ `expresión`
- **División**
  Reliza la división de dos expresiones
  `expresión` / `expresión`
- **Módulo**
  Reliza la suma de dos expresiones
  `expresión` % `expresión`
- **Arreglos**
  El i-ésimo elemento de un arreglo. `arr`$[i]$
- **Caracteres**
  Se pueden respesentar caracteres por medio de apóstrofes: $'c'$, $'h'$, $'a'$, $'r'$
- **Cadenas**
  Son concatenaciones de caracteres las cuales se representan con comillas: "$Cadena$"
- **Número**
  Como se mecionó anteriormente, este lenguaje puede usar distintos `tipos` de represntaciones de números. Cualquier tipo de éstos es una expresión válida.
- **Identificador**
  Un identificador se refiere al nombre asignado a alguna variable o alguna función.
  Cuando queremos usar un identificador de variable como una expresión se deben colocar paréntesis frente al nombre: `var`()
  SI se desea usar el nombre de una función como expresión se deben colocar los argumentos que recibe (si es que recibe alguno): `foo`$(arg_1, \ldots, arg_n)$

Condiciones

Tenemos la posiblidad de poder definir comportamiento por medio de expresiones booleanas, las cuales son definidas como **condiciones**, existen distintos tipos y son las siguientes:

- **Or**
  Realiza un **or lógico** de dos condiciones:
  `condición` || `condición`
- **And**
  Realiza un **and lógico** de dos condiciones:
  `condición` && `condición`
- **Negación**
  Realiza **negación lógica** de una condición:
  ! `condición`
- **Parentización**
  Realiza la parentización de una condición:
  (`condición`)
- **Relacionales**
  Realiza la operación booleana de dos expresiones:
  `expresión op expresión`
  Donde `op` $\in \{<, >, \leq, \geq, !=, ==\}$
- **Verdadero**
  Es la representación para una condición verdadera:
  `true`
- **Falso**
  Es la representación para una condición falsa: `false`

3 Sentencias

Una sentencia permite realizar la manipulación de nuestros elementos definidos, podemos hacer modificaciones en éstos, así como definir comportamiento a nuestro programa.
En este lenguaje de programación existen múltiples tipos de sentencias las cuales serán clasificadas como **condicionales**, **estructuras de control, escapes** y **asignaciones**

3.1 Sentencias Condicionales

Este tipo de sentencia permite dividir el flujo de programa, de esta forma podemos realizar distintas operaciones dependiendo de cierta condición. En este lenguaje tenemos **if** y **switch**.

If

En esta sentencia podemos usar una condición para dividir el flujo del programa. Por ahora, **no se implementa su versión if else**. La estructura es la siguiente:

`if` (condición) sentencias

Switch

De igual forma, permite dividir el flujo del programa implementando múltiples casos. La estructura es la siguiente:

`switch`(expresión){
`case`$_1$: `Número`
sentencias
`case`$_n$: `Número`
sentencias
`default` : sentencias }

El número que haga *match* con la expresión es el caso que se ejecutará, saliendo así del switch. En caso de que no existe ningun *macth*, entonces entra al caso default.

3.2 Sentencias de Asignación

Una parte muy importante es poder realizar asignaciones a nuestras variables, de esta forma podemos llevar un mejor manejo de nuestro programa. Las asignaciones tienen la siguiente estructura:

`parte_izquierda` = `expresión`

Donde la parte izquierda está conformada por:

- **Id**
  En este caso podemos tener el identificador de una variable, por lo que podemos tener la sentencia:
  `miVar` = `expresión`
- **Elemento de Arrgelo**
  Como se mencionó anteriormente, podemos acceder a un elemento de un arreglo por medio de índices, por lo que es conveniente poder modificar estas regiones de memoria, teniendo así:
  `arr`$[i]\cdots[j]$ = `expresión`
- **Elemento de struct**
  De igual forma, podemos asignar el valor de un elemento de un `struct`:
  `rgb.red` = 144;

3.3 Sentencias de Estructuras de Control

Este tipo de sentencia nos permite realizar iteraciones para poder realizar sentencias de forma repetitiva mientras se cumpla una `condición`. Existen distintos tipos como: **while, do while, for**

While

Realiza iteraciones basadas en que la condición asignada sea verdadera. La estructura es la siguiente:

`while` (condición) `sentencias`

Es importante notar que en esta estructura podemos realizar ciclos infinitos.

Do while

Es la estructura similar al while, sin embargo las sentencias que contiene siempre se ejecutan al menos una vez, la condición se verifica hasta después de la primer ejecución de las sentencias. La estructura es la siguiente:

`do sentencias while` (condición);

Nuevamente, en esta estructura podemos generar ciclos infinitos.

For

Esta estructura de control pide escoger un rango de iteraciones a realizar. La estructura es la siguiente:

`for` (`sentencia` ; `condición` ; `sentencia`) `sentencias`

Donde la primer sentencia corresponde a una asignación. La condición es la que se encarga de determinar si se itera nuevamente, esta condición se relaciona con la primer sentencia. Finalmente, a última sentencia se encarga de modificar la primer sentencia para poder llegar al caso en el que la condición es falsa. Por ejemplo:

`for` (i = 0; ; i() < 10 ; i = i() + 1) `sentencias`

3.4 Sentencias de Escape

Tenemos dos sentencias de escape: `return` expresión; y `return`;

Return con Expresión

Se usa en funciones para especificar que queremos terminar la ejecución de la función en ese punto y además regresa al contexto padre la expresión recibida.
Una condición importante es que la expresión que regresa debe coincidir con la especificación de la función. Ejemplo:

func `int` foo() { `return` 1; }

Claramente, los tipos coinciden en las declaraciones.

Return

En este caso podemos terminar la función sin regresar ningún valor al contexto padre. Esta sentencia se usa para funciones definidas con tipo `void` de regreso. Ejemplo:

func `void` foo(`int` n){
`if` (`n` $<$ 10) `return;`

```
        while (true);
    }
```

En esta función si el argumento recibido es menor a 10, entonces termina la ejecución de la función, en otro caso, el programa se cicla para siempre. De esta forma vemos cómo es útil `return`, ya sea con expresión de regreso o sin expresión.

4 Comentarios

Algo importante es poder realizar comentarios. Este lenguaje de programación lo permite y existen dos tipos de comentarios: **Linea** y **Multilínea**.

4.1 Comentarios de línea

Para poder realizar comentarios de este tipo basta con anteponer los caracteres `//` seguido de cualquier comentario que queramos. Se detendrá la detección de este comentario hasta encontrar un salto de línea.

```
a = a() + 1 //Actualiza variable
```

4.2 Comentarios multilínea

Podemos realizar comentarios para varias líneas, estos comentarios se obtienen anteponiendo los caracteres /* y colocando al final */. El comentario puede contener cualquier caracter que deseemos, excepto */

```
/*
* Función para actualizar el color rojo.
* Recibe color, el nuevo color a remplazar.
*/
```

5 Instalación

Se debe contar con el siguiente software:

1. `bison` En su versión 3+
2. `flex` En su versión más reciente.
3. `gcc` En su versión más reciente.

Se proporciona un archivo `Makefile`. Para poder usarlo es necesario acceder a una consola **$** y ejecutar:

```
$ make ejec
```

Se proporcionan además diversos comandos, para poder visualizarlos puede ejecutar nuevamente en su consola **$** el siguiente comando:

```
$ make
```

Una vez que realizamos la compilación de algún programa y resultó exitoso, entonces encontraremos los siguientes archivos en nuestra carpeta fuente:

- `codigo.ci`
  Es el código intermedio para una arquitectura `MIPS`, generado por el compilador.
- `errores.txt`
  Se encuentran los errores generados por el programa fuente.
- `producciones.txt`
  Están todas las producciones usadas para el análisis semántico.
- `contexts.txt`
  Están todos los contextos de las funciones, así como el contexto global del programa fuente.
- `comentarios.txt`
  Están todos los comentarios encontrados en el programa fuente.
- `tokens_output.txt`
  Están todos los tokens encontrados durante el análisis léxico.
- `errores_lexicos.txt`
  Están los errores encontrados en el proceso del análisis léxico.

6 Código Fuente

El código fuente de este compilador puede ser encontrado en la siguiente liga:

https://github.com/DavidPalmerin/Proyecto-Compiladores

Se proporciona un ejemplo escrito en `C--`