UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Escuela de Ingeniería Informática

Procesadores de Lenguaje:

WiC

Curso 2018 / 2019

invalid utf-8 byte sequence spanish characters listing latex

Realizado por:
Medina Medina, David Alberto
Brito Ramos, Christian
Hernández Delgado, Christopher
López González, Néstor

Índice

Resumen

`WiC` (Winter is Coming) es un lenguaje de programación diseñado por estudiantes de la *ULPGC* con el objetivo de establecer el diseño y desarrollo de un compilador de lenguaje. El objetivo de esta experiencia es desarrollar un compilador del lenguaje que pase por las diferentes etapas de análisis léxico, sintáctico y semántico con el objetivo de generar un código intermedio en lenguaje ensamblador `IA-32`

1. Definición del lenguaje

En esta sección se describen todos los detalles que describen las características propias del lenguaje `WiC`.

1.1. Tipos de datos

Cualquier leguaje de programación necesita definir un conjunto de *tipos de datos*, esto es, la batería de valores y operaciones que puede adquirir una variable. Cada tipo de dato está definido en el lenguaje por un *literal* único que lo representa, lo que permite que cada tipo de dato tenga un representación física específica.

Los tipos de datos definidos en el lenguaje son los que figuran en el *cuadro 1*. Las características críticas de implementación que define a cada tipo son:

Entero Representa a todas y cada una de las variables enteras que sean declaradas en el lenguaje. Este tipo de dato presenta un tamaño de 4 bytes (32 bits) y permite representar números enteros con signo. El *complemento a 2* es el sistema elegido para definir el signo del número entero. Este dato se representa por el literal `int`. El rango de valores que puede tomar es

$$\left[-2^{N-1},\, 2^{N-1}-1\right] = \left[-2^{32-1},\, 2^{32-1}-1\right] = [-2147483648,\, 2147483647] \tag{1}$$

donde, N es el número de bits disponibles para representar el número entero (32 bits).

Coma flotante de simple precisión Este tipo de dato representa número reales en coma flotante de simple precisión con un tamaño de 4 bytes (32 bits) siguiendo el estándar *IEEE 754*. En la figura 1.1 puede observarse como esta representación binaria los bits se organizan en tres sectores principales:

- **Signo** (1 bit). Se trata de un sólo bit que define el signo del número: positivo (0) o negativo (1).
- **Exponente (8bits)**. Se trata de un número entero con signo de complemento a 2 ($[-128,\, 127]$)
- **Mantisa (23 bits)**. Conforma la fracción a la derecha de la coma binaria y un bit de encabezado implícito.

Este tipo de dato se representa con el literal `real`. Su rango de valores es de $\left[1{,}18 \cdot 10^{\tilde{}38},\ 3{,}4 \cdot 10^{38}\right]$.

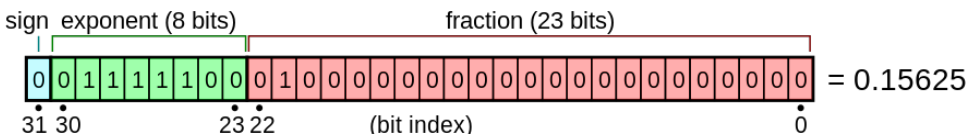

Figura 1: Representación binaria de número en coma flotante de simple precisión (*IEEE 754*)

Caracter Este tipo de dato es usado para representar caracteres con la codificación *ASCII*, es por este motivo que el tamaño de que ocupan las variables de tipo caracter son de 1 byte de tamaño, siendo `char` el literal que lo representa.

Booleano Se trata de un tipo de dato utilizado para representar representar valores booleanos. Su tamaño es de 1 bit, por lo que tan solo puede tomar dos valores: `1` (verdadero) ó `0` (falso). El literal que lo representa es `bool`.

Cuadro 1: Tipos de datos

Tipo	**Literal**	**Tamaño**	**Rango**
Entero	int	4 Bytes	$[-2147483648,\ 2147483647]$
Coma flotante de simple precisión	real	4 Byte	$[1{,}18 \cdot 10^{-38},\ 3{,}4 \cdot 10^{38}]$
Caracter	char	1 Byte	[`0x00 - 0xFF`]
Lógico	bool	1 Bit	$[0,\ 1]$

En el siguiente ejemplo se muestra cómo se declaran las variables con los literales de los tipos descritos anteriormente:

```
int entero      .. Esto es un entero
real flotante   .. Esto es un numero en coma flotante
char caracter   .. Esto es un caracter
bool booleano   .. Esto es un booleano
```

1.2. Colecciones de datos: `Arrays`

Las variables pueden ser agrupadas en colecciones de datos de una dimensión denominados `arrays`. En este lenguaje, cualquier tipo de dato puede formar parte de un `array`.

Para declarar un `array` del tipo que se desee, debe usar la gramática 2:

$$tipo[< int >]\ nombre\_variable \tag{2}$$

donde el *tipo* define el tipo de dato e $< int >$ un valor entero opcional que define el tamaño del `array`. Cuando no se indica este último valor entero, no se lleva a término la reserva en memoria del `array` declarado. En caso contrario, se reservará en memoria tantos bytes/bits como fueren necesarios para generar una colección de tipos de datos del tamaño indicado por $< int >$. El número de bytes/bits a reservar está determinado por el tipo de dato y el tamaño del `array`,

$$Tamano_{memoria} = Tamano_{tipo\,dato} \times Tamano_{array} \tag{3}$$

Un `array` que ha sido declarado con anterioridad puede ser redefinido haciendo uso de la gramática 4,

$$tipo[\,]\ nombre\_variable = new\ tipo[< int >]\ \{expresion\_1, expresion\_2, \dots\} \tag{4}$$

Esta notación es alternativa a la gramática 2, donde podremos inicializar el `array` a un conjunto de expresiones separados por comas encerrados dentro de los caracteres `{` y `}`. Estas expresiones son

opcionales. El tamaño máximo del vector y, por tanto, de expresiones es el indicado por $< int >$ el cual es un número entero de caracter obligatorio.

Si se declara un `array` utilizando la gramática 2 o 4 sin expresiones en esta última, el array es inicializado en todas sus posiciones al valor `0` para tipos de datos enteros y reales. Para caracteres el valor por defecto es el caracter `nulo`. Y para tipos booleanos el valor por defecto es `FALSE`.

La gramática 5 es necesaria para acceder al valor de un elemento del `array` en una posición arbitraria del mismo,

$$nombre\_variable\ [< int >] \tag{5}$$

donde $< int >$ es un entero obligatorio que indica la posición del `array` a la que se desea acceder.

El `array` de caracteres constituyen los denominados `string`, los cuales requieren una atención especial ya que es posible cargar una variable con una secuencia de caracteres sin ser necesaria la declaración dada por la gramática 4. La gramática 6 define la instancia de un `string` con un conjunto de caracteres localizados entre los caracteres comillas doble,

$$nombre\_variable\ = " < char >< char > ..." \tag{6}$$

El siguiente listado muestra algunos ejemplos de uso de los `array` definidos en este lenguaje:

Listing 1: Ejemplo de uso de arrays

```
,.
   En este bloque se declara un array de cada tipo de variable sin asignacion de espacio en
       memoria
.,

int [] entero
real [] real
char [] string
bool [] booleano


,.
   En este bloque se declara un array de cada tipo con asignacion de espacio en memoria (de
       tamano 5 cada uno)
.,

int [5] entero5
real [5] real5
char [5] string5
bool [5] booleano5


,.
   En este bloque se declara un array de cada tipo y se instancia cada uno de los valores de
       los elementos que lo componen
.,

int [] entero2 = new int[2]{ 0, 1}
real [] real2 = new real[2]{0.0, 1.0}
char [] string2_ab = "ab" .. Declaracion con asignacion alternativo al tipo string
char [] string2_cd = new char[2]{'c','d'}
bool [] booleano2 = new bool[2]{true, false}
```

1.3. Palabras reservadas y cambios de contexto

Este lenguaje utiliza una serie de palabras reservadas que se utilizan para desempeñar las funciones aquí descritas:

`continue` La sentencia de continue es de tipo de control de bucles. Dentro de la iteracion en un bucle, de cualquiera de los tipos (while, do-while, for), el uso de esta sentencia rompe la iteracion de dicho bucle. Provocando que se ejecute la siguiente iteracion de dicho bucle, ignorando las sentencias posteriores a çontinue".

`break` Dentro de la iteracion en un bucle, de cualquiera de los tipos (while, do-while, for), el uso de esta sentencia rompe la iteracion de dicho bucle.

`return` Palabra empleada para retornar el resultado de un método o función, además de interrumpir la ejecución del mismo.

`void` Es utilizado para indicar que una función no debe devolver ningún valor.

`fun` Palabra especial utilizada al principio de cada declaración o definición de funciones.

Los **cambios de contexto** de bloques de instrucciones son aplicables cuando se hace uso de un caracter `0x09` (`TAB`) o de 4 espacios consecutivos (`0x20`).

1.4. Comentarios

Aquí va el texto. Poner siempre un código de ejemplo.

- **'Comentario de línea'** . Para indicar un comentario de línea, basta con comenzar con '..', siendo el resultado final '.. TEXTO'.

Listing 2: Ejemplo de comentario de línea

```
int a = 7
.. Variable que almacena una suma.
int suma = 0
suma = 30 + a
```

- **'Comentario de bloque'** . Los comentarios de varias líneas se establecen utilizando ',.' y '.,' quedando como resultado ',. VARIAS LÍNEAS DE TEXTO .,'.

Listing 3: Ejemplo de comentario de bloque

```
int a = 7
int suma = 0
suma = 30 + a
,.
En este punto, la variable suma toma el
valor de 37.
.,
```

1.5. Tipos de operadores

El conjunto de posibles operadores aplicables a nuestro lenguaje pueden clasificarse en:

- Operadores aritméticos

- Operadores lógicos
- Operadores bit a bit
- Operadores de array

1.5.1. Operadores aritméticos

Estos operadores son necesarios para realizar operaciones matemáticas sencillas. La descripción detallada de cada uno de estos operadores queda reflejado en la tabla 2.

Cuadro 2: Operadores aritméticos

Operación	**Tipo operación**	**Lexema**	**Estructura**	**Tipo expresión**	**Tipo evaluado** [1]
Suma	Binaria	`+`	`{Expr} + {Expr}`	`int/real`	`int/real`
Resta	Binaria	`-`	`{Expr} - {Expr}`	`int/real`	`int/real`
Multiplicación	Binaria	`*`	`{Expr} * {Expr}`	`int/real`	`int/real`
División	Binaria	`/`	`{Expr} / {Expr}`	`int/real`	`int/real`
Módulo	Binaria	`%`	`{Expr} % {Expr}`	`int/real`	`int/real`
Potencia	Binaria	`**`	`{Expr} ** {Expr}`	`int/real`	`int/real`
Raíz	Binaria	`#`	`{Expr} # {Expr}`	`int/real`	`int/real`
Pre-incremento [2]	Unaria	`++`	`++{Expr}`	`int/real`	`int/real`
Pre-decremento [2]	Unaria	`--`	`--{Expr}`	`int/real`	`int/real`
Post-incremento [3]	Unaria	`++`	`{Expr}++`	`int/real`	`int/real`
Post-decremento [3]	Unaria	`--`	`{Expr}--`	`int/real`	`int/real`

[1] Todas las operaciones binarias permiten entremezclar el uso de un valor entero y otro real al mismo tiempo aunque esto conllevaría el *casteo* del resultado de la operación a un valor de tipo real.
[2] El valor de la variable es incrementado/decrementado una unidad previa asignación del mismo.
[3] El valor de la variable es incrementada/decrementada una unidad después de asignar el valor del misma.

Listing 4: Ejemplo de uso los operadores artiméticos

```
int a = 2
int b = 4

int c = 2#4          .. c = 2

int d
d = c++              .. d = 2, c = 3

int e
e = ++c              .. e = 4, c = 4
```

1.5.2. Operadores lógicos

Definimos los símbolos que identificarán a los operadores lógicos de forma análoga a los utilizados por gran parte de otros lenguajes de programación, separando cada operador 2 expresiones a comparar por los operadores lógicos a utilizar, con la excepción del propio *NOT*, que podrá utilizarse directamente para determinar si el valor devuelto por una expresión de tipo `booleano` es directamente verdadero o no.

Como dato adicional, también consideramos la posibilidad de hacer uso de comparadores mediante dichos operadores lógicos n-arios, comparando pares de expresiones entre sí de forma sucesiva siguiendo el orden de lectura de izquierda a derecha (que a efectos prácticos es lo mismo que hacer sucesivos *AND* pero de forma más breve) y la posibilidad de prioridades para ciertos operadores (como el *AND*).

En relación a las expresiones de comparación, consideramos hacer que de momento solo se puedan utilizar entre tipos de datos del mismo tipo o similar (por ejemplo, entre enteros y reales podría hacerse). Dejamos la tabla de operadores como sigue:

Cuadro 3: Operadores lógicos

Operación	Tipo operación	Lexema	Estructura	Tipo expresión[1]	Tipo evaluado
`AND`	Binaria	`&&`	`{Expr} && {Expr}`	`int/real/bool`	`bool`
`OR`	Binaria	`||`	`{Expr} || {Expr}`	`int/real/bool`	`bool`
`NOT`	Unaria	`!`	`!{Expr}`	bool	`bool`
Igual	Binaria	`==`	`{Expr} == {Expr}`	`int/real/bool`	`bool`
No Igual	Binaria	`!=`	`{Expr} != {Expr}`	`int/real/bool`	`bool`
Mayor	Binaria	`>`	`{Expr} >{Expr}`	`int/real/bool`	`bool`
Mayor o igual	Binaria	`>=`	`{Expr} >= {Expr}`	`int/real/bool`	`bool`
Menor	Binaria	`<`	`{Expr} <{Expr}`	`int/real/bool`	`bool`
Menor o igual	Binaria	`<=`	`{Expr} <= {Expr}`	`int/real/bool`	`bool`

[1] El tipo de dato utilizado como expresión deben ser del mismo tipo para operadores lógicos binarios.

Listing 5: Ejemplo de uso los operadores lógicos

```
int a = 7
int suma = 0
5 <= a < 10 ?:
    suma = 4 * a

suma > 20 > a && 2*suma != 32 < a
    suma++
```

1.5.3. Operadores bit a bit

Estos operadores permiten manipular los valores de una expresión a nivel de bits. El cuadro 4 refleja la descripción detallada de cada uno de estos operadores.

Cuadro 4: Operadores bit a bit

Operación	Tipo operación	Lexema	Estructura	Tipo expresión[1]	Tipo evaluado[2]
AND	Binaria	&	{Expr} & {Expr}	Cualquiera	Cualquiera
OR	Binaria	|	{Expr} | {Expr}	Cualquiera	Cualquiera
XOR	Binaria	^	{Expr} ^ {Expr}	Cualquiera	Cualquiera
Desp. izq.	Unaria	«	{Expr} « {Expr}[3]	Cualquiera	Cualquiera
Desp. dcha.	Unaria	»	{Expr} » {Expr}[3]	Cualquiera	Cualquiera

[1] El segundo operando debe ser siempre de tipo `int`.
[2] El tipo de dato asignado a la variable debe ser del mismo tipo que el del primer operando.
[3] Rota los bits de la expresión de la izquierda tantas unidades a la izquierda/derecha como indique la expresión de la derecha.

Listing 6: Ejemplo de uso los operadores bit a bit

```
char c = ' '          .. c ==> Caracter ASCII 0x20
int  i = 2

char rc = c<<2        .. rc = '@' ==> Caracter ASCII 0x40
int  ri = i<<2        .. ri = 8
```

1.5.4. Operadores de array

Los operadores de arrays nos permiten realizar operaciones con arrays, tratándolos como a subconjuntos. Los operadores a definir son los siguientes:

Unión La unión de dos arrays consiste en crear un nuevo array formado por los elementos del primer array y los del segundo. La unión no permite elementos repetidos.

Diferencia La diferencia de un array con otro consiste en crear un nuevo array con todos los elementos del primer array, descartando aquellos que aparecen en el segundo.

Intersección La intersección de dos arrays consiste en crear un nuevo array formado por todos los elementos comunes entre el primer y el segundo array.

Concatenación Nuevo array formado por los elementos del primero seguido de los del segundo. La concatenación permite crear el nuevo array con elementos repetidos.

Cuadro 5: Operadores de array

Operación	Tipo operación	Lexema	Estructura	Tipo expresión	Tipo evaluado
Unión	Binaria	`U`	`{Expr} U {Expr}`	Cualquiera	Cualquiera
Diferencia	Binaria	`D`	`{Expr} D {Expr}`	Cualquiera	Cualquiera
Intersección	Binaria	`I`	`{Expr} I {Expr}`	Cualquiera	Cualquiera
Concatenación	Binaria	`+`	`{Expr} + {Expr}`	Cualquiera	Cualquiera

[1] Rota los bits de la expresión de la izquierda tantas unidades a la izquierda/derecha como indique la expresión de la derecha.

A continuación mostramos un ejemplo de uso de los operadores de array para un caso trivial meramente ejemplificativo:

Listing 7: Ejemplo de uso de operadores de array

```
int [6]  a = new int [6]{0,1,2,3,4,5}
int [7]  b = new int [7]{3,4,5,6,7,8,9}

int []  c = a U b ..c =  {0,1,2,3,4,5,6,7,8,9}
int []  d = a D b ..d = {0,1,2}
int []  e = a I b  .. e = {3,4,5}
int []  f = (a U b) D (a I b)  .. f =  {0,1,2,6,7,8,9}
int []  g = d + f + a ..g =   {0,1,2,0,1,2,6,7,8,9,0,1,2,3,4,5}
```

1.6. Estructuras de control

El conjunto de instrucciones del lenguaje no tienen por qué ejecutarse en una secuencia lineal sino que es posible que el programador establezca un cierto control en esa secuencia de ejecución. Las diferentes estructuras de control que se definen en nuestro lenguaje se clasifican en los siguiente grupos:

- Sentencias `if-ifelse-else`
- Bucle `for-forelse-else`
- Bucle `while-whileelse-else`

1.6.1. Sentencias `if-ifelse-else`

La estructura de control if-ifelse-else sentencia condicional que está compuesta de los siguientes bloques:

Bloque `if` Este bloque está identificado por la siguiente estructura:

```
{Expr} ?:
    {Instr}
    ...
```

En caso de ser verdadera, se ejecutan las instrucciones de este bloque. Sólo puede existir un único bloque `if` al principio de una sentencia `if-ifelse-else`.

Bloque `ifelse` Si no se cumple la condición del bloque anterior, se comprueba si la condición de este bloque se cumple para posteriormente ejecutar el conjunto de instrucciones que se encuentran en su contexto. En caso contrario, pasa a ejecutarse el siguiente bloque `elseif` La estructura de este bloque es la siguiente:

```
. {Expr} ?:
    {Instr}
    ...
```

Bloque `else` Si ninguna de la expresiones de los bloques anteriores cumple la condición, se ejecutarán las instrucciones contenidas en este bloque. La estructura de este bloque es la siguiente:

```
.?:
    {Instr}
    ...
```

Sólo puede existir un único bloque `else` al final de una sentencia `if-ifelse-else`.

Listing 8: Ejemplo de uso de la sentencia `if-ifelse-else`

```
int a = 25
int b = 0

a > 20 ?:
    b = a - 10
. a < 10 ?:
    b = a + 5
.?:
    b = -1
```

1.6.2. Bucle `for-forelse-else`

Este tipo de bucle permiten modificar el valor de una variable mientras se realiza la evaluación lógica de uno de sus bloques. El bucle for-forelse-else se estructura en los siguientes bloques:

Bloque `for` Este bloque presenta la siguiente estructura:

```
{Expr} ?? {Expr}:
    {Instr}
    ...
```

La primera expresión no exige que figure el parámetro utilizado en la iteración. Si, por ejemplo, se definen otras formas de salida dentro del contexto del bucle, y si se cumple la condición evaluada se ejecutarán las instrucciones dentro del contexto en caso de que el conjunto de condiciones del bucle se cumpla. Además, es legal no indicar una expresión lógica a evaluar, en cuyo caso, el resultado de la evaluación siempre será `true`, iterándose indefinidamente las instrucciones de este contexto.

La segunda expresión se usará comúnmente como modificador del valor de iteración empleado en la condición.

Bloque `forelse` Se ha considerado hacer que exista el bloque `for-else`, que actuaría de tal manera que si no se cumple inicialmente la condición del primer bucle, comprobará la expresión del siguiente, y si su condición se cumple, estará evaluando repetidamente las instrucciones de este último (sin volver a comparar con las condiciones del bucle anterior al `else`). De esta manera, podemos hacer una estructura combinada de lo que en otros casos serían `if-for-else-for`, de forma directa con solo un `for-else`. Su estructura es la siguiente:

```
. {Expr} ?? {Expr}:
    {Instr}
    ...
```

Bloque `else` De forma análoga a los `else` de las expresiones condicionales, la representación para este bloque es la siguiente:

```
.??:
    {Instr}
    ...
```

Este bloque es siempre terminal al bucle `for-forelse-else` y siempre iterará por el conjunto de instrucciones de su contexto (bucle infinito) siempre y cuando las expresiones lógicas de los bloques anteriores evalúen a `false`.

Listing 9: Ejemplode uso del bucle for-forelse-else

```
int a = 7
int b = 9
int suma = 0
int contador = 0

a > contador ?? contador++:
    suma = suma + a * contador + b
. b > contador ?? contador++:
    suma = suma + b*contador + a
.??:
    suma = a + b
.. Como 'a' es mayor que 'contador' inicialmente, solo se ejecutan instrucciones del primer
    bucle
```

1.6.3. Bucle `while-whileelse-else`

Este bucle permite ejecutar un bloque de instrucciones de manera iterativa siempre y cuando la condición de la expresión sea `true`. En caso contrario, se irá probando las diferentes condiciones de cada bloque del bucle. Este bucle se puede estructura en los siguientes bloques:

Bloque `while` Este bloque está identificado por la siguiente estructura:

```
{Expr} ??:
    {Instr}
    ...
```

Mientras la expresión del bloque siga sienda `true`, se ejecuta el bloque de instrucciones del bucle. Si la expresión es `false`, se evalúa la expresión lógica del siguiente bloque. Sólo puede existir un único bloque `while` al principio de un bucle `while-whileelse-else`.

Bloque `whileelse` Si no se cumple la condición del bloque anterior, se comprueba si la expresión lógica de este bloque es `true`. Si es así, se itera por el conjunto de instrucciones que se encuentran en su contexto hasta que la expresión sea `false`. Si la evaluación de la expresión lógca es `false`, se pasa a ejecutar el siguiente bloque `whileelse` o el bloque terminal `else`. La estructura de este bloque es la siguiente:

```
. {Expr} ??:
    {Instr}
    ...
```

Bloque `else` Si ninguna de la expresiones de los bloques anteriores cumple la condición, se iterarán las instrucciones contenidas en este bloque indefinidamente, generándose un bucle infinito termnial. La estructura de este bloque es la siguiente:

```
.??:
    {Instr}
    ...
```

Sólo puede existir un único bloque `else` al final de un bucle `while-ifelse-else`.

En cualquier caso, la expresión lógica es opcional. Si no se indica una existe una expresión al principio del bucle, se itera por el conjunto de instrucciones de ese bloque de manera indefinida.

1.7. Declaración, definición y uso de funciones

Las funciones componen una estructura fundamental en cualquier lenguaje de programación que son útiles para estructurar funcionalmente el código en diferentes contextos, aportando una mayor expresividad del lenguaje.

Una función puede ser declarada sin especificar el código que se ejecutará dentro de su contexto. La forma general de declaración puede observarse en el listado 10.

Listing 10: Declaracion de funciones

```
fun {tipo} {nombre_funcion} ({parametro1}, [{parametro2}], ...)
```

La definición de cualquier función sigue la estructura del listado 11. El cuerpo de la función debe comenzar por el caracter de cambio de contexto (caracter `0x09` o cuatro caracteres `0x20` seguidos).

Listing 11: Declaracion de funciones

```
fun {tipo} {nombre_funcion} ({parametro1}, [{parametro2}], ...)
    {instruccion1}
    [{instruccion2}]
    ...
```

El listado 12 ilustra la estructura general de llamada a una función.

Listing 12: Declaracion de funciones

```
{nombre_funcion}({parametro1}, [{parametro2}], ...)
```

Es importante recalcar que la referencia en memoria de un array es leída cuando este es usado como parámetro de una función. Si un array es el valor de retorno de una función, se devuelve su referencia en memoria.

1.8. Funciones primitivas

En nuestro lenguaje, Van a existir predefinidas un conjunto de funciones primitivas que facilitarán al programador su trabajo en el desarrollo del programa. Las funciones están divididas en las siguientes categorías:

Interacción entrada-salida Las funciones básicas de E/S son:

`bool print(T salida)` Esta función permite mostrar por pantalla el valor de `salida`, cuyo tipo `T` puede ser `int, real, bool, char` o una string (`char[]`). Si la operación termina correctamente la primitiva devolverá `true`, en caso contrario, devuelve `false`.

`bool scan(char[] entrada)` Esta rutina permite leer el valor pasado por teclado una vez se pulse `ENTER` (`0x0D`). Si la operación termina satisfactoriamente la primitiva devolverá `true`, en caso contrario, devuelve `false`.

Listing 13: Ejemplo de uso de la función `print()` y `scan()`

```
int number
char[] name
print("Introduzca el primer número: ")
a = scan()
print("¿Cuál es su nombre? ")
name = scan()
print("El usuario " + name + " ha introducido el número " + number)
```

Funciones para arrays numéricos Las funciones para arrays numéricos permiten operaciones que impliquen el uso de arrays de enteros, números en coma flotante o combinar ambos tipos.

El tipo `T` de las siguientes funciones indica que el tipo del parámetro de entrada o el tipo de retorno de la función puede ser `int` o `real`:

`int length(array)` Devuelve el número de elementos del array.

`T sum(array)` Devuelve la suma de todos los elementos contenidos en el array.

`T res(T[] array)` Devuelve el resultado de restar todos los elementos del array.

`real prod(T[] array)` Devuelve el producto de todos los elementos contenidos en el array.

`int prod(T[] array1, T[] array2)` Devuelve el producto escalar de los dos arrays.

`real coc(T[] array)` Devuelve el cociente de todos los elementos contenidos en el array.

`T[] append(T[] array, T[] element, int position)` Coloca en la posición `position` del array un elemento que debe ser del mismo tipo que el resto del array. Los demás elementos a partir de la posición especificada serán desplazados una posición a la derecha. Se devuelve la referencia al array resultante.

`T[] delete(T[] array, int position)` Elimina el elemento de la posición `position` del array y devuelve la referencia al array resultante. Los demás elementos a partir de la posición especificada serán desplazados una posición a la izquierda.

`T[] find(array,element)` Devuelve una referencia a un array de enteros, que representan las posiciones donde se encuentra un elemento en el array.

`T[] sort(T[] array, T[] ascendent)` Devuelve la referencia a un array con los elementos ordenados de forma ascendente si `ascendent` es igual a `true`; en caso contrario, los ordenará de forma descendente.

`bool equal(T[] array1, T[] array2)` Devuelve `true` si los dos arrays son iguales en tamaño y los elementos están dispuestos en el mismo orden. En caso contrario, devuelve `false`.

Listing 14: Ejemplo de uso de funciones primitivas para arrays numéricos

```
int [] v1 = {2,5,9,5}
int [] v2 = {5,4,8,9,3}
int tam = length(v1)      .. tam = 4
int n1 = sum(v1)          .. n1 = 21
int n2 = sum(v2)          .. n2 = 29
int [] v3 = sum(v1,v2)        .. v3 = {7,9,17,14,3}
int [] v4 = res(v1,v2)        .. v4 = {-3,1,1,-4,-3}
int n3 = prod(v1)         .. n3 = 450
int n4 = prod(v2)         .. n4 = 4320
int [] v5 = prod(v1,v2)       .. v5 = {10,20,72,45,0}
int [] v6 = coc(v3,v2)        .. v6 = {2,5,9,5,0}
float [] v7 = coc(v1,v2)      .. v7 = {0.4,1.25,1.125,0.5555555556,0.0}
append(v1,6,3)            .. v1 = {2,5,6,9,5}
delete(v1,3)              .. v1 = {2,5,9,5}
int [] found1 = find(v1,5)    .. found1 = {2,4}
int [] found2 = find(v1,8)    .. found2 = {}
sort(v2,true)             .. v2 = {3,4,5,8,9}
sort(v2,false)            .. v2 = {9,8,5,4,3}
boolean b1 = equal(v2,v3)  .. b1 = false
boolean b2 = equal(v1,v1)  .. b2 = true
```

Funciones para strings Las funciones para strings permiten realizar operaciones sobre strings. Las funciones principales son:

`int length(char[] string)` Devuelve el número de caracteres de la string.

`char[] substring(char[] string, int position)` Devuelve la referencia a la substring que se encuentra entre `position` y el final de `string`.

`char[] substring(char[] string, int beginning, int end)` Devuelve la referencia a un substring ubicado entre la posición `beginning` y la posición anterior al índice `end`.

`char[] trim(char[] string)` Elimina todos los espacios (`0x20`) de la string.

`bool equal(char[] string1, char[] string2)` Devuelve `true` si las dos strings son iguales en tamaño y los caracteres que lo conforman están dispuestos en el mismo orden. En caso contrario, devuelve `false`.

Listing 15: Ejemplo de uso de funciones primitivas para strings

```
char[] s1 = "Hola"
char[] s2 = " mundo"
char[] s3 = s1 + s2                    .. s3 = "Hola mundo"
int tam1 = length(s1)                  .. tam1 = 4
int tam2 = length(s3)                  .. tam1 = 10
char[] s4 = substring(s3,6)               .. s4 = "mundo"
char[] s5 = substring(s3,4,8)          .. s5 = "a mu"
char[] s6 = " " + s1 + " " + s2 + "    " .. s6 = " Hola   mundo   "
int tam3 = length(s6)                  .. tam3 = 17
trim(s6)                            .. s6 = "Hola mundo"
int tam4 = length(s6)                  .. tam4 = 10
boolean b1 = equal(s3,s6)              .. b1 = true
boolean b2 = equal(s1,s3)              .. b2 = false
```

1.9. Código ejemplo

Aquí va el código de ejemplo con el que probaremos nuestro compilador.

2. Análisis léxico

En esta sección se describen todos los detalles que describen al analizador léxico del lenguaje `WiC`.

2.1. Introducción

El analizador léxico es la parte del compilador que se encarga de leer el flujo de caracteres que componen el programa fuente y agruparlos en tokens, que son agrupaciones de secuencias que guardan cierto significado; produciendo, para cada lexema, un token con la estructura $< nombre-token, valor-atributo >$. Cada token pasará a la fase del análisis sintáctico, a partir de la cuál se generará la gramática y el *Árbol Sintáctico Abstracto* (*AST*) asociado.

En cada token, el primer componente (nombre-token) se utiliza para el análisis sintáctico, y el segundo (valor-atributo), el cual es opcional, apunta a una entrada de la tabla de símbolos con el valor de dicho token.

Listing 16: Ejemplo de conversión de un conjunto de caracteres de un programa WiC en tokens

```
int doble = total * 2

(int)    ->  INT_TYPE
(doble) ->  ID
=     ->  ASSIGN
(total) ->  ID
*     ->  PRODUCT
2     -> INT_VAL
```

2.2. Tabla de tokens

Nombre-Token	Valor-Atributo	Expresión-Regular
NEW	NO	(new)
CONTINUE	NO	(continue)
BREAK	NO	(break)
RETURN	NO	(return)
VOID	NO	(void)
FUN	NO	(fun)
INT_VAL	SI	[0-9]+
REAL_VAL	SI	([0-9]+\.)[0-9]*|\.[0-9]+
BOOL_VAR	SI	(true|false)
CHAR_VAL	SI	\'.?\'
STRING_VAL	SI	\".*\"
INT_TYPE	NO	(int)
REAL_TYPE	NO	(real)
BOOL_TYPE	NO	(bool)
CHAR_TYPE	NO	(char)
IF_CLAUSE	NO	(\?){1}
FOR_WHILE_CLAUSE	NO	(\?){2}
HEADER_ON	NO	:
ELSE_IF_FOR_WHILE_CLAUSE	NO	(\.){1}
BLANK_SPACE	NO	[ ]{1,3}
CONTEXT_TYPE	NO	[\t]|[ ]{4}
ID	SI	[a-zA-Z]([a-zA-Z]|[0-9])+
ASSIGN	NO	-"
SUM	NO	\+
SUBSTRACT	NO	-
PRODUCT	NO	*
DIVIDE	NO	/
MODULUS	NO	%
POWER	NO	**
RADICAL	NO	#
INCREMENT	NO	++
DECREMENT	NO	−−

Nombre-Token	Valor-Atributo	Expresión-Regular
AND_BIT	NO	&
OR_BIT	NO	|
XOR_BIT	NO	^
LEFT_SHIFT	NO	<<
RIGHT_SHIFT	NO	>>
AND	NO	&&
OR	NO	||
NOT	NO	!
EQUALS	NO	-="
NOT_EQUALS	NO	!=
GREATER	NO	<
GREATER_EQUALS	NO	>=
LESS	NO	<
LESS_EQUALS	NO	<=
UNION	NO	U
DIFFERENCE	NO	D
INTERSECTION	NO	I
SQUARE_BRACKET_OPEN	NO	[
SQUARE_BRACKET_CLOSE	NO	]
CURLY_BRACKET_OPEN	NO	{
CURLY_BRACKET_CLOSE	NO	}
ELEM_SEPARATOR	NO	,
PARETHESES_OPEN	NO	\(
PARETHESES_CLOSE	NO	\)
END_OF_INSTR	NO	\n
CHAR_QUOTE	NO	\'
STRING_QUOTE	NO	\"

2.3. Estados