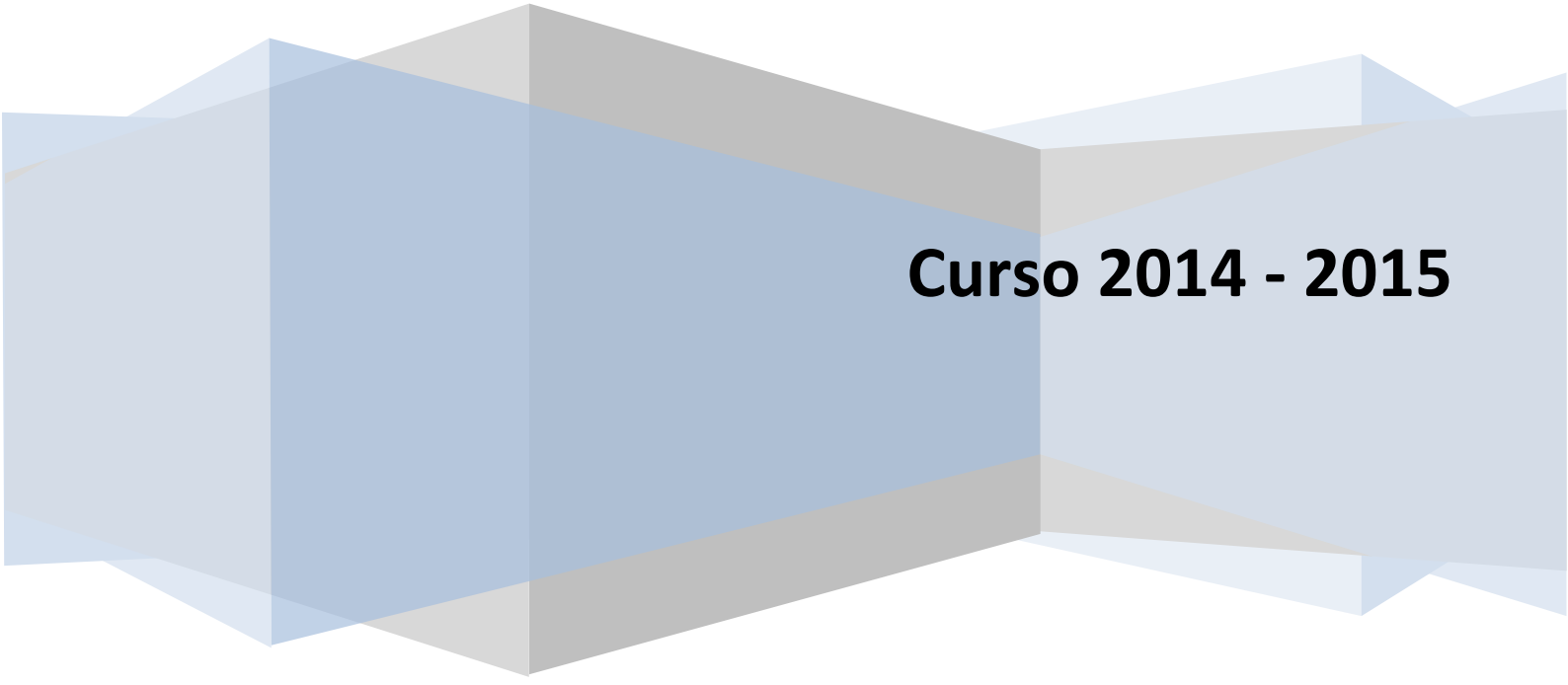
Universidad Nacional de Educación a Distancia
Departamento de Lenguajes y Sistemas Informáticos

Práctica de Procesadores del Lenguaje I

Especificación del lenguaje cUNED

Dpto. de Lenguajes y Sistemas Informáticos
ETSI Informática, UNED

Alvaro Rodrigo
Anselmo Peñas (coordinador)



Curso 2014 - 2015

Contenido

1 Introducción

En este documento se define la práctica de la asignatura de Procesadores del Lenguaje I correspondiente al curso 2014-2015. El objetivo de la práctica es realizar un compilador del lenguaje cUNED, que es una variación del conocido lenguaje de programación C.

Primero se presenta una descripción del lenguaje elegido y las características especiales que tiene. A continuación se indicará el trabajo a realizar por los alumnos en diferentes fases, junto con las herramientas a utilizar para su realización.

A lo largo de este documento se intentará clarificar todo lo posible la sintaxis y el comportamiento del compilador de cUNED, por lo que es importante que el *estudiante lo lea detenidamente y por completo*.

2 Descripción del lenguaje

Este apartado es una descripción técnica del lenguaje cUNED, una versión reducida y modificada del lenguaje C. En los siguientes apartados presentaremos la estructura general de los programas escritos en dicho lenguaje describiendo primero sus componentes léxicos y discutiendo después cómo éstos se organizan sintácticamente para formar construcciones del lenguaje.

2.1 Aspectos Léxicos

Desde el punto de vista léxico, un programa es una secuencia ordenada de TOKENS. Un TOKEN es una entidad léxica indivisible que tiene un sentido único dentro del lenguaje. En términos generales es posible distinguir diferentes tipos de TOKENS: Los operadores aritméticos, relacionales y lógicos, los delimitadores como los paréntesis o los corchetes, los identificadores utilizados para nombrar variables, constantes, tipos definidos por el usuario, nombres de procedimientos, o las palabras reservadas del lenguaje son algunos ejemplos significativos. A lo largo de esta sección describiremos en detalle cada uno de estos tipos junto con otros elementos que deben ser tratados por la fase de análisis léxico de un compilador.

2.1.1 Comentarios

Un comentario es una secuencia de caracteres que se encuentra encerrada entre los delimitadores de principio de comentario y final de comentario: “/*” y “*/”, respectivamente. Todos los caracteres encerrados dentro de un comentario deben ser ignorados por el analizador léxico. En este sentido su procesamiento *no debe generar TOKENS* que se comuniquen a las fases posteriores del compilador. En el caso de este compilador **sí** es posible realizar *anidamiento* de comentarios. De esta manera, dentro de un comentario pueden aparecer los delimitadores de comentario “/*” y “*/” para acotar el comentario anidado. El analizador léxico deberá gestionar apropiadamente el anidamiento de comentarios para garantizar que los delimitadores de principio y fin de comentario están adecuadamente balanceados. En caso de que esto no se produzca el proceso de análisis debe finalizar

emitiendo un mensaje de error léxico. Algunos ejemplos de comentarios correctos e incorrectos son los siguientes:

Listado 1. Ejemplo de comentarios

```
/* Este es un comentario correcto */

/* Es comentario contiene varias líneas

     Esta es la primera línea

     Esta es la segunda línea   */

/* Este es un comentario con comentarios anidados correctamente

    /* Comentario Anidado 1

          /* Comentario Anidado 1.1 */

          /* Comentario anidado 1.2 */

     */

*/

/* ERROR. Este es un comentario mal balanceado */ */

/* ERROR. Este es un comentario /* mal balanceado */

/* ERROR. Este es un comentario no cerrado
```

2.1.2 Constantes literales

Estas constantes no deben confundirse con la declaración de constantes simbólicas, que permiten asignar nombres a ciertas constantes literales, para ser referenciadas por nombre dentro del programa fuente, tal como se verá más adelante. En concreto, se distinguen los siguientes tipos de constantes literales:

- **Enteras**. Las constantes enteras permiten representar valores enteros no negativos. Por ejemplo: `0`, `32`, `127`, etc. En este sentido, no es posible escribir expresiones como `-2`, ya que el operador unario "–", no existe en este lenguaje. Si se pretende representar una cantidad negativa será necesario hacerlo mediante una expresión cuyo resultado será el valor deseado. Por ejemplo para representar – 2 podría escribirse `0 - 2` (si la especificación asignada al alumno lo permite)

- **Cadenas de caracteres**. Las constantes literales de tipo cadena consisten en una secuencia ordenada de caracteres ASCII. Están delimitadas por las comillas dobles, por ejemplo: "ejemplo de cadena". Las cadenas de caracteres se incluyen en la práctica únicamente para poder escribir mensajes de texto por pantalla mediante la instrucción `printc()` (ver más adelante), pero no es necesario tratarlas en ningún otro contexto. Es decir, *no se crearán variables de este tipo*. No se tendrán en cuenta el tratamiento de caracteres especiales dentro de la cadena ni tampoco secuencias de escape.

2.1.3 Identificadores

Un identificador consiste, desde el punto de vista léxico, en una secuencia ordenada de caracteres y dígitos que comienzan obligatoriamente por una letra. Los identificadores se usan para nombrar entidades del programa tales como las constantes, los tipos definidos por el usuario, las variables o los subprogramas definidos por el programador. El lenguaje **sí** es sensible a las mayúsculas (case sensitive), lo que significa que dos identificadores compuestos de los mismos caracteres y que difieran únicamente en el uso de mayúsculas o minúsculas se consideran diferentes. Por ejemplo, `Abc` y `ABC` son identificadores diferentes. La longitud de los identificadores no está restringida, pero el alumno es libre de hacerlo en caso de que lo considere necesario.

2.1.4 Palabras reservadas

Las palabras reservadas son tokens del lenguaje que, a nivel léxico, tienen un significado especial de manera que no pueden ser utilizadas para nombrar otras entidades como variables, constantes, tipos definidos por el usuario o funciones y procedimientos.

A continuación se muestra una tabla con las palabras reservadas del lenguaje así como una breve descripción aclarativa de las mismas. Su uso se verá en más profundidad en los siguientes apartados.

PALABRA CLAVE	DESCRIPCIÓN
`#define`	Comienzo de bloque de declaración de constantes
`else`	Comienzo del cuerpo de alternativa de una condicional if
`for`	Comienzo de un bucle for
`if`	Comienzo de una sentencia condicional if
`int`	Tipo entero
`main`	Nombre función principal
`printc`	Procedimiento predefinido que muestra por pantalla una cadena de texto con salto de línea al final
`printi`	Procedimiento predefinido que muestra por pantalla un entero con salto de línea al final

`return`	Sentencia de retorno de una función
`struct`	Comienzo de la declaración de un tipo registro
`void`	Tipo vacio
`while`	Comienzo de un bucle while

2.1.5 Delimitadores

El lenguaje define una colección de delimitadores que se utilizan en diferentes contextos. A continuación ofrecemos una relación de cada uno de ellos:

DELIMITADOR	DESCRIPCIÓN
"	Delimitador de constante literal de cadena
( )	Delimitadores de expresiones y de parámetros
[ ]	Delimitadores de rango de una matriz
/* */	Delimitadores de comentario
,	Delimitador en listas de identificadores
;	Delimitador en secuencias de sentencias

2.1.6 Operadores

Existen diferentes tipos de operadores que son utilizados para construir expresiones por combinación de otras más sencillas como se discutirá más adelante. En concreto podemos distinguir los siguientes tipos:

Operadores aritméticos	+ (suma aritmética) − (resta aritmética)
Operadores relacionales	< (menor) > (mayor) == (igual) != (distinto)
Operadores lógicos	&& (conjunción lógica) || (disyunción lógica)
Operadores de asignación	= (asignación)
Operadores de acceso	. (acceso al campo de un registro)

Obsérvese que, como se advirtió con anterioridad, no se consideran los operadores unarios + y −, de forma que los literales numéricos que aparezcan en los programas serán siempre sin signo (positivos). Así, los números negativos no aparecerán en el lenguaje fuente, pero sí pueden surgir en tiempo de ejecución como resultado de evaluar una expresión aritmética de resultado negativo, como por ejemplo 1 - 4.

2.2 Aspectos Sintácticos

A lo largo de esta sección describiremos detalladamente las especificaciones sintácticas que permiten escribir programas correctos. Comenzaremos presentado la estructura general de un programa y, posteriormente, iremos describiendo cada una de las construcciones que aparecen en detalle.

2.2.1 Estructura de un programa y ámbitos de visibilidad

Desde el punto de vista sintáctico, un programa en cUNED es un fichero de código fuente con extensión '.cuned' que contiene una colección de declaraciones. En el código fuente del listado 2 se muestra un esquema general de la estructura de un programa cUNED.

Listado 2. Estructura general de un programa en cUNED

```
<< Declaración de constantes simbólicas >>

<< Declaración de tipos globales>>

<< Declaración de variables globales >>

<< Declaración de funciones >>
```

```
<< Declaración de la función principal >>
```

Como puede apreciarse un programa en cUNED comienza con la declaración opcional de las constantes simbólicas que se van a utilizar en el programa. En caso de existir, éstas solamente pueden ir al inicio del programa. A continuación deben declararse primero los tipos y después las variables que se utilizarán globalmente en todo el programa. Un programa se completa con una colección ordenada de declaración de funciones. A este respecto existe una restricción importante:

- Existe una función principal, denominada *main*, con una cabecera fija y bien definida cuyo código constituye el punto de arranque del programa y que debe ser declarada obligatoriamente siempre al final del programa. La función main se explicará más detalladamente a lo largo de este documento.

La estructura de un programa en cUNED consiste exactamente en todas estas declaraciones con posibles comentarios entremezclados entre ellas. En las siguientes subsecciones pasamos a describir cada una de ellas en detalle. En el anexo A se muestra un ejemplo completo en cUNED.

2.2.2 Declaraciones de constantes simbólicas

Las constantes simbólicas, cómo se ha comentado antes, constituyen una representación nombrada de datos constantes, cuyo valor va a permanecer inalterado a lo largo de toda la ejecución del programa. Únicamente pueden declararse al inicio del programa y no es posible su declaración dentro de otros ámbitos distintos al global.

En esta práctica las constantes simbólicas representan literales de valor entero y positivo por lo que son entidades de tipo entero (consulte la descripción de tipos en la sección 2.2.3.1 de este documento). La sintaxis para su declaración es la siguiente:

```
#define nombre valor;
```

Donde *nombre* es el nombre simbólico que recibe la constante definida dentro del programa y *valor* un valor constante literal de tipo entero positivo (número sin signo) de manera que cada vez que éste aparece referenciado dentro del código se sustituye por su valor establecido en la declaración. La cláusula `#define` no marca el inicio de la sección de declaraciones de constantes, sino que se usa en cada declaración de constante simbólica. Para cada constante declarada es necesario utilizar esta cláusula, terminando obligatoriamente en ";". Por convenio el nombre de la constante suele expresarse en mayúsculas, lo que ayuda a la hora de leer el código final, pero no es obligatorio y pueden declararse también en minúsculas.

A continuación se muestran algunos ejemplos de declaración de constantes.

Listado 3. Ejemplos de declaración de constantes simbólicas enteras

```
#define FALSE 0;

#define TRUE 1;

#define LUNES 1;
```

```
#define TRES 2+1;  /* ERROR, valor no puede ser una expresión */

#define CUATRO +4; /* ERROR, el entero no ha de llevar signo  */

#define CINCO 5    /* ERROR, falta punto y coma */
```

2.2.3 Declaración de tipos

La familia de tipos de un lenguaje de programación estructurado puede dividirse en 2 grandes familias: tipos primitivos del lenguaje y tipos estructurados o definidos por el usuario. A continuación describimos la definición y uso de cada uno de ellos.

2.2.3.1 Tipos primitivos

Los tipos primitivos del lenguaje (también llamados tipos predefinidos o tipos simples) son todos aquellos que se encuentran disponibles directamente para que el programador tipifique las variables de su programa. En concreto en esta práctica se reconocen dos tipos primitivos: el tipo entero y el tipo especial vacio. En los apartados subsiguientes abordamos cada uno de ellos en profundidad.

Tipo entero (int)

El tipo entero representa valores enteros positivos y negativos. Por tanto, a las variables de este tipo se les puede asignar el resultado de evaluar una expresión aritmética, ya que dicho resultado puede tomar valores enteros positivos y negativos. El rango de valores admitido para variables de este tipo oscila entre -32728 y +32727 ya que el espacio en memoria que es reservado para cada variable de este tipo es de 1 palabra (16 bits). Sin embargo, en las fases que se desarrollan en esta práctica no se tiene que comprobar que el entero que se desea asignar a una variable está dentro del rango mencionado.

El tipo entero se representa con la palabra reservada `int`. La aplicación de este tipo aparece en distintos contextos sintácticos como en la declaración de variables, la declaración de parámetros y tipo de retorno de funciones o la definición de tipos compuestos.

En el listado 4 se presentan algunos ejemplos donde aparecen elementos tipificados con el tipo entero. No se preocupe si no entiende el detalle de estas construcciones, más adelante en este documento se describirá cada una de ellas.

Listado 4. Ejemplos de uso del tipo int

```
int a;

int a=3;

int a [5][2];

struct Punto {

   int x;

   int y;

}

int sumar (int x, int y) { ... }
```

Tipo vacio (void)

Dentro del paradigma de programación estructurada existe una clara diferenciación conceptual entre funciones (construcciones sintácticas que al ser invocadas devuelven un valor de retorno al contexto del programa llamante) y procedimientos (construcciones sintácticas que ejecutan una secuencia de instrucción pero no devuelven ningún resultado al programa llamante). Sin embargo en nuestro caso no existe el concepto de procedimiento en si mismo. Se entiende que todos los subprogramas son funciones. Cuando se desea que una función se comporte como un procedimiento (esto es como un bloque de código cuya invocación no produce ningún valor de retorno al programa llamante) es preciso tipificar su tipo de retorno en la cabecera de la función con la palabra reservada `void`. Esto justifica el uso del tipo void y permite exponer algunos ejemplos donde aparece utilizado (ver un ejemplo en el listado 5, donde no se pretende que el lector comprenda la sintaxis asociada a la declaración de funciones en cUNED). El uso del tipo void queda circunscrito únicamente a la declaración de este tipo de funciones, no pudiéndose utilizar en ningún otro contexto.

Listado 5. Ejemplos de uso del tipo void

```
void main () { ... }

void escribeEntero (int x) {

      printi (x);

}
```

2.2.3.2 Tipos Estructurados

Los tipos estructurados (también llamados tipos *definidos por el usuario* o tipos *compuestos*) permiten al programador definir estructuras de datos complejas y establecerlas como un tipo más del lenguaje. Estas estructuras reciben un nombre (identificador) que sirve para referenciarlas posteriormente en la declaración de variables, parámetros, campos de registros, tipos base de matrices, etc. Por eso, es preciso declararlos siempre antes que los elementos donde aparecen referidos.

La sintaxis para la declaración de un tipo compuesto depende de la clase de tipo de que se trate. En el caso de esta práctica existen dos tipos estructurados: matrices y registros.

Tipo Matriz

Una matriz es una estructura de datos bidimensional que sirve para almacenar una secuencia ordenada de valores de tipo entero. Es decir, en cUNED sólo pueden declararse matrices de dos dimensiones y cuyo tipo base sea entero. El tamaño de una matriz limita el número de elementos que pueden ser almacenados en la misma en tiempo de ejecución. Éste se define en su declaración a través de dos constantes (literales o simbólicas) conocidas en tiempo de compilación. Esto significa que el tamaño de cada dimensión nunca puede ser una expresión. La sintaxis de declaración para matrices es la siguiente:

```
int nombre [NUMFILAS][NUMCOLUMNAS];
```

Donde `nombre` será el identificador de la matriz y `NUMFILAS y NUMFILAS` constantes enteras y positivas, delimitadas entre corchetes (en este caso son obligatorios, no indican

elemento opcional), que indica el número de elementos que contiene la matriz en sus dos dimensiones. El listado 6 presenta algunos ejemplos de declaración de matrices.

Listado 6. Ejemplo de uso de matrices

```
#const MAXROW 7;

#const MAXCOL 7;

int matriz1[MAXROW][MAXCOL];

int matriz2[3][2];

int matriz3[MAXROW][3];

int vectorf[0][5]    /* Declaración de un vector fila */

int vectorc[4][0]    /* Declaración de un vector columna */

int matriz4[2+1][3]; /* ERROR. El tamaño no puede ser una expre-
                        sión en ninguna de sus dimensiones */
```

Como se ha descrito, en cUNED no se pueden definir vectores ni arrays multidimensionales. A todos los efectos un vector se podría declarar asignando un tamaño 0 al número de filas o columnas de una matriz. El acceso a los datos de una matriz es una expresión y como tal se explicará más adelante.

Tipo Registro

Un registro es una estructura de datos compuesta que permite agrupar varios elementos de distinto tipo. Para definir un registro se utiliza la palabra reservada *struct* seguida de una lista de declaraciones de campos encerradas entre llaves. Cada declaración de campos comienza por un identificador de tipo (primitivo o compuesto) seguido de una lista de identificadores de campos separadas por comas y finalizadas en punto y coma. Concretamente la declaración de un registro sigue la siguiente sintaxis:

```
struct nombre {

      tipo1 campo11 [, campo12, ...];

      [tipo2 campo21 [, campo22, ...]];

      ...

}
```

Donde `nombre` será el identificador del tipo registro, y `campoij`, son los nombres de los distintos campos del registro. Los tipos de los campos sólo pueden ser enteros o registros. No es posible declarar matrices como campos de un registro. Para declarar un registro cuyos campos son registros es preciso declarar primero los registros de los campos y a continuación declarar el registro compuesto de otros registros (aunque la comprobación de este requisito no se realiza en esta práctica). El nivel de anidación de registros no puede ser mayor que dos.

Es decir, no puede existir un registro que contenga un campo de tipo registro y que éste, a su vez, contenga un campo de tipo registro. A continuación se muestran varios ejemplos de uso de registros.

Listado 7. Ejemplo de uso de registros

```
struct Tpersona {
      int dni;
      int edad;
}
struct Tfecha {
      int dia, mes, anyo;
}
struct Tcita {
      Tpersona empleado;
      Tfecha fecha;
      int urgente;
}
/* El siguiente ejemplo es erróneo. Tcita ya contiene un
registro en su declaración, por lo que no puede ser usado en la
declaración de un nuevo registro */
struct TCitaDoble {
      TCita cita;
}
```

2.2.4 Declaraciones de variables

En el lenguaje propuesto es necesario declarar las variables antes de utilizarlas. El propósito de la declaración de variables es el de registrar identificadores dentro de los ámbitos de visibilidad establecidos por la estructura sintáctica del código fuente y tipificarlos adecuadamente para que sean utilizados dentro de éstos. Para declarar variables se utiliza la siguiente sintaxis, dentro de las áreas de declaración de variables de un programa:

```
nombre-tipo nombre1 [=valor1], nombre2 [=valor2],...;
```

Donde `nombre-tipo` es el nombre de un tipo primitivo del lenguaje o el de un tipo definido por el usuario, `nombre1, nombre2,...` son identificadores para las variables declaradas y `valor1, valor2,...` son los valores asignados a esas variables, siendo su uso opcional. Como se ve en la sintaxis, si se desean declarar varias variables de un mismo tipo en la misma línea ha de hacerse utilizando el delimitador ",".Nótese que en nuestro caso se contempla la

posibilidad de realizar asignaciones en línea dentro de la declaración del estilo "`int a = 1;`". Esto está sólo permitido en caso de que las variables sean de tipo int y los valores han de ser constantes enteras literales o simbólicas, pero nunca una expresión o función. Además la asignación no es asociativa, no pudiéndose dar casos como: "`int a = b =1`".

En el siguiente listado se muestran algunos ejemplos de declaraciones de variables. Se supone que los tipos compuestos utilizados (TPersona y matriz1) han sido previamente declarados y corresponden con los expuestos en los listados 7 y 6 :

Listado 8. Ejemplos de declaración de variables

```
int a;

int a,b;

int a=1, b=2;

int a=1, b;

int a=b=1; /* ERROR. El operador = no es asociativo */

TPersona p1, p2;

matriz1 m1, m2;

matriz1 m3[1][2]=2; /* ERROR. La asignación en línea solo está
                        permitida para variables de tipo int */
```

2.2.5 Declaración de funciones

En el lenguaje propuesto se pueden declarar funciones para organizar modularmente el código. Una función es una secuencia de instrucciones encapsuladas bajo un nombre con un conjunto ordenado de parámetros tipificados (opcionalmente ninguno) y un valor de retorno también tipificado. En este lenguaje no existen los procedimientos propiamente dichos (subprogramas que no devuelven valor). Para declarar una función que se comporte como un procedimiento debe tipificarse el retorno de la misma con el tipo especial void.

El uso de funciones con y sin valor de retorno se aplica en contextos sintácticos diferentes. Las funciones con valor de retorno se emplean en el contexto de una expresión mientras que las funciones sin valor de retorno se consideran sentencias. En las secciones 2.2.6.1 (Invocación de funciones) y 2.2.6.2 (Sentencias de invocación a una función void) abordaremos la invocación de funciones en esos dos casos. Por ahora nos ocuparemos de describir la sintaxis de su declaración:

```
tipo-retorno nombre ([tipo1 param1, tipo2 param2,...]) {
    [<< declaración de tipos locales >>]
    [<< declaración de variables locales >>]
    [<< sentencias >>]
    return expresión;
```

```
}
```

En esta definición los corchetes indican opcionalidad. Como puede apreciarse, la declaración de la función comienza por `tipo-retorno` que indica el tipo de retorno de la función. Éste debe ser necesariamente un tipo primitivo (int o void). A esto le sigue el nombre de la función y después, opcionalmente, una lista de parámetros con nombre separados por comas. Si la función no tiene parámetros al nombre de la función le siguen paréntesis vacios "()". Nótese que cada declaración de parámetro debe ir emparejado con su tipo correspondiente.

Tras la declaración de la cabecera aparece la declaración del cuerpo de la función. Esta es en realidad un bloque de sentencias y por tanto postergaremos su explicación para la sección 2.2.6.2 (Bloques de sentencias) donde explicaremos su sintaxis en detalle. Sí precisaremos aquí no obstante que dentro de un bloque de sentencias **No** es posible declarar nuevas funciones lo que significa que en este lenguaje no está permitido declarar funciones anidadas (funciones cuya declaración aparece dentro del cuerpo de la función).

Una sentencia relevante dentro del cuerpo de la función es la sentencia `return`. Esta instrucción indica un punto de salida de la función y provoca que el flujo de control de ejecución pase de la función llamada a la siguiente instrucción dentro de la función llamante. Nótese que una función debe tener una instrucción de retorno por cada punto de salida. Sin embargo, esta comprobación queda fuera del ámbito de esta práctica, por lo que no debe realizarse. La sentencia `return` puede venir seguida de una expresión que indique el valor de retorno (se permite que no vaya seguida de ninguna expresión para utilizarse en funciones declaradas con void).

Listado 9. Ejemplo de declaración de funciones

```
void nada () {

}

int sumar (int x, int y) {

   return x + y;

}

void escribeEntero (int x) {

   printi (x);

   return;

}

void saluda () {

   printc ("Hola mundo!");

}
```

```
int devuelve2 () {
   return 2;
}
```

Algunos de los errores en funciones son de tipo sintáctico, como por ejemplo construir mal la cabecera, y deben ser tratados en la fase de análisis sintáctico. Existen, sin embargo, otro tipo de errores que deben ser comprobados en otras fases. Por ejemplo, la comprobación de la existencia de `return` debería delegarse al análisis semántico, que no se aborda en esta práctica.

2.2.5.1 La función principal main

Todo programa en cUNED debe obligatoriamente declarar una función principal ubicada al final del código fuente. La declaración de la función principal tiene la siguiente estructura sintáctica:

```
void main () {
    [<< declaración de tipos locales >>]
    [<< declaración de variables locales >>]
    [<< sentencias >>]
}
```

Notese que se trata de una función sin valor de retorno ni parámetros y cuyo cuerpo es estructuralmente idéntico a la de cualquier función sin valor de retorno.

2.2.5.2 Paso de parámetros a funciones

En el lenguaje propuesto es posible invocar a funciones pasando expresiones o referencias de alguno de los tipos definidos en la sección 2.2.3 como parámetros a una función. El paso de parámetros actuales a un subprograma puede llevarse a cabo de dos formas diferentes: paso por valor y paso por referencia.

Paso por valor

En este caso el compilador realiza una copia del argumento a otra zona de memoria para que el subprograma pueda trabajar con él sin modificar el valor del argumento tras la ejecución de la invocación. Los parámetros actuales pasados por valor **no** pueden ser matrices ni registros enteros.

Paso por referencia

En este caso, el parámetro sólo podrá ser una referencia (no una expresión). El paso de parámetros por referencia es utilizado para pasar argumentos de salida o de entrada / salida. Para indicar que un parámetro se pasa por referencia debe precederse su nombre con el

carácter &. En el listado 10 se incluyen ejemplos de funciones que utilizan paso por valor y por referencia:

Listado 10. Ejemplos de pasos de parámetros por valor y por referencia

```
int incrementa (int x) { /* Paso por valor */
  x=x+1;
  devuelve x;
}
void decrementa (int &x) { /* Paso por referencia */
  x=x-1;
}
void main () {
  int a=1;
  printi (a); /* Escribe 1 */
  incrementa (a);
  printi (a); /* Escribe 1 */
  a = incrementa (a);
  printi (a); /* Escribe 2 */
  decrementa (a);
  printi (a); /* Escribe 1 */
  return;
}
```

2.2.6 Sentencias y Expresiones

El cuerpo de un programa o subprograma está compuesto por sentencias que, opcionalmente, manejan internamente expresiones. En este apartado se describen detalladamente cada uno de estos elementos.

2.2.6.1 Expresiones

Una expresión es una construcción del lenguaje que devuelve un valor de retorno al contexto sintáctico del programa donde se evaluó la expresión. Las expresiones no deben aparecer de *forma aislada* en el código. Es decir, han de estar incluidas como parte de una sentencia allá donde se espere una expresión. Desde un punto de vista conceptual es posible clasificar las expresiones de un programa en cUNED en varios grupos:

Expresiones aritméticas

Las expresiones aritméticas son aquellas cuyo cómputo devuelve un valor de tipo entero (int) al contexto de evaluación. Puede afirmarse que son expresiones aritméticas las constantes literales de tipo entero, las constantes simbólicas de tipo entero, los identificadores (variables o parámetros) de tipo entero y las funciones que tienen declarado un valor de retorno de tipo entero. Asimismo, también son expresiones aritméticas la suma y resta de dos expresiones aritméticas.

Expresiones lógicas

Las expresiones lógicas son aquellas cuyo cómputo devuelve un valor de tipo lógico al contexto de evaluación. En nuestro caso no existe un tipo primitivo para tipificar expresiones de verdad (cierto / falso). En su lugar, se utiliza una convención para codificar numéricamente valores lógicos de verdad o falsedad. En concreto ha de suponerse que cuando una expresión aritmética se evalúa en un contexto lógico (como en el caso de la expresión condicional de un bucle mientras) y devuelve un valor igual a 0 este resultado se debe interpretar con el significado de falso, mientras que si el resultado es distinto de 0 la interpretación será de cierto. A partir de ahora, llamaremos expresión lógica a toda expresión aritmética que es evaluada en un contexto lógico. Asimismo también son expresiones lógicas la conjunción "&&" y disyunción "||" de dos expresiones aritméticas. Se incluyen una serie de operadores relacionales que permite comparar expresiones aritméticas y lógicas entre sí. El resultado de esa comparación es también una expresión lógica. Es decir, la comparación con los operadores >, <, == o != de dos expresiones aritméticas o lógicas es también una expresión lógica.

Expresiones de asignación

Las expresiones de asignación sirven para asignar un valor a una variable, elemento de una matriz o campo de un registro. Para ello se escribe primero una referencia a alguno de estos elementos seguido del operador de asignación "=" y a su derecha una expresión. El operador de asignación no es asociativo. Es decir no es posible escribir construcciones sintácticas del estilo a = b = c. Tampoco es posible hacer asignaciones a estructuras (matrices o registros) de forma directa. La sintaxis de la expresión de asignación es a:

```
ref = expresion;
```

Donde ref es una referencia a una posición de memoria (variable, parámetro, campo de registro o elemento de una matriz y expresión una expresión que le da valor. El siguiente listado muestra algunos ejemplos de uso de sentencias de asignación:

Listado 12. Ejemplos de uso de asignación

```
i = 3 + 7;

m[i][j] = 10;

distinto = 3!=4;

cita1.urgente = 0;

a = 2 + suma(2,2);

suma(2,3) = 5; /* ERROR. La llamada a una función no puede
```

```
                    aparecer en la parte izquierda de una
                    asignación */
```

Las asignaciones pueden aparecen en forma de sentencias, es decir de manera independiente en el código, pero a diferencia de cómo ocurre en otros lenguajes, en cUNED las instrucciones de asignación se comportan también cómo expresiones. Así, la parte derecha de la asignación es una expresión que se evalúa y se asigna a la referencia a la izquierda del operador de asignación (=), pero adicionalmente devuelve el valor del resultado de la asignación al contexto sintáctico donde esta instrucción se ejecuta. Esto permite que las asignaciones puedan escribirse en lugares donde típicamente se esperaría una expresión. Por ejemplo, en el fragmento de sentencia condicional "if (a=3>1)" se evalúa primero el valor de la expresión 3>1. Como el resultado es verdadero, el valor que devuelve al contexto es 1 (cierto) y ese valor es asignado a la variable a y simultáneamente devuelto al contexto de la sentencia condicional que, al ser un valor distinto de 0 provoca que se entre a ejecutar el cuerpo del condicional.

Expresiones de acceso a campos de registros

Para acceder a los campos de un registro se utiliza el operador de acceso a registro "." (punto). Dado que los campos de un registro pueden ser a su vez registros previamente declarados, es posible concatenar el uso del operador de punto. Es decir, si el resultado de la evaluación de una expresión que utiliza el operador de acceso a registros es otro registro, entonces es posible volver a aplicar nuevamente el operador de punto para profundizar en los campos del registro interno. Hay que recordar que en cUNED el nivel de anidamiento máximo en registros es dos. En el siguiente listado se puede ver un ejemplo de estas expresiones.

Listado 13. Ejemplo expresiones con acceso a campos de un registro

```
int x;

TCita cita1;

TCita cita2;

cita1.urgente = 1;

cita1.empleado.dni = 1234;

cita1.empleado.edad = 23;

cita1.fecha.dia = 2;

cita1.fecha.mes = 11;

cita1.fecha.anyo = 2014;

x = cita1.empleado.edad;

cita2.empleado.edad = x;

cita2.urgente = cita2.urgente;
```

Expresiones de acceso a matrices

El acceso a los datos de una matriz se realiza de forma indexada. Para acceder individualmente a cada uno de sus elementos se utilizan los operadores de acceso a matriz `[][]`. Dentro de estos delimitadores es preciso ubicar una expresión aritmética. El siguiente listado muestra distintos ejemplos de accesos a elementos de una matriz.

Listado 14. Ejemplos de expresiones con acceso a elementos de una matriz

```
int matrizVector [0][3];

int matrizDos [2][2];

matrizVector m1;

matrizDos m2;

m1[0][0]=1;

m1[0][1]=2;

m1[0][2]=3;

m2[1][0]=5;

m2[0][0]=m1[0][2];

m2[0][1]=0;

printi (m1[0][0]);        /* Escribe 1 */

printi (m1[0][1+1]);      /* Escribe 3. m1[0][2] */

printi (m1[m2[0][1]][1]); /* Escribe 2. m1[0][1] */
```

Invocación de funciones

Para llamar a una función ha de escribirse su identificador indicando entre paréntesis los parámetros actuales de la llamada. Los parámetros pueden ser referencias a variables, campos de registros o elementos de matriz o expresiones. El uso de los paréntesis es siempre obligado. Así, si una función no tiene argumentos, en su llamada es necesario colocar unos paréntesis vacios "()" tras su identificador.

Nótese que **las funciones que devuelven un tipo void no son expresiones**. Por tanto, si se utilizan en el lugar de una expresión debe generarse un error. Esta comprobación es propia del análisis semántico, por lo que no se tratan en esta práctica. Posteriormente, en la sección 2.2.6.2 (Sentencias de invocación a una función void) abordaremos este tipo de sentencias.

En el siguiente listado se muestran algunos ejemplos de llamadas a funciones.

Listado 15. Ejemplos de llamadas a funciones

```
X = resta(a, b);

y = suma(a, resta(b, c));

a = funcion1();
```

```
b = suma (r.campo1, r.campo2); /* r es de tipo registro */

c = suma (m[0][1], 2);         /* m es de tipo matriz */
```

Precedencia y asociatividad de operadores

Cuando se trabaja con expresiones aritméticas o lógicas es muy importante identificar el orden de prioridad (precedencia) que tienen unos operadores con respecto a otros y su asociatividad. Ahora que hemos descrito el uso sintáctico de todos los operadores conviene resumir la relación de precedencia y asociatividad de los mismos. En la siguiente tabla la prioridad decrece por filas. Los operadores en la misma fila tienen igual precedencia.

Precedencia	Asociatividad
. ( )	Izquierda
[]	Derecha
+ -	Izquierda
< >	Izquierda
== !=	Izquierda
&&	Izquierda
||	Izquierda
=	Derecha

Para alterar el orden de evaluación de las operaciones en una expresión aritmética o lógica prescrita por las reglas de prelación se puede hacer uso de los paréntesis. Como puede apreciarse los paréntesis son los operadores de mayor precedencia. Esto implica que toda expresión aritmética encerrada entre paréntesis es también una expresión aritmética. En el siguiente listado se exponen algunos ejemplos sobre precedencia y asociatividad y cómo la parentización puede alterar la misma.

Listado 11. Ejemplos de precedencia y asociatividad en expresiones

```
1 || 0 && 1   /* Devuelve 1 (cierto) */

1 || (0 && 1) /* Devuelve 1 (cierto) */

2 + (2 - 3)   /* Devuelve 1 */

(2 + 2) - 3   /* Devuelve 1 */

2 > 3 > 1 /* Devuelve 0 (falso), ya que se evalúa 2 > 3 como
             falso, que se considera cero para comparar con 1*/
```

2.2.6.2 Sentencias

El lenguaje propuesto dispone de una serie de sentencias que sirven a distintos propósitos dentro de un programa. Las sentencias se escriben dentro del cuerpo de las funciones declaradas en el mismo (ya sea la función principal o funciones declaradas por el usuario). En efecto, tal como se describió en la sección 2.2.5, tras la declaración de tipos y variables comienza una secuencia de sentencias que constituyen el código de la función. A lo largo de esta sección describiremos todos los tipos de sentencias existentes en cUNED a excepción de la sentencia return que ya fue explicada en la sección 2.2.5.

Bloques de sentencias

Un bloque de sentencias es una colección de declaraciones y sentencias encerradas entre los delimitadores de llaves "{" y "}" . La sintaxis de un bloque de sentencias es la siguiente:

```
{
    [<< declaración de tipos locales >>]
    [<< declaración de variables locales >>]
    [<< sentencias >>]
}
```

Como puede apreciarse, dentro de un bloque aparece inicialmente una colección opcional de declaraciones de tipos definidas por el usuario. Seguidamente aparece una colección opcional de declaraciones de variables y finalmente una colección de sentencias también opcional. A todos los efectos, un bloque de sentencias se comporta como una sentencia ordinaria. Esto implica varias cosas. En primer lugar que los bloques de sentencia pueden aparecer unos dentro de otros sin ninguna restricción en cuanto al nivel de anidamiento. En segundo lugar, que en cualquier punto, a lo largo de este documento, donde indiquemos el uso de una sentencia puede aparecer perfectamente un bloque de sentencias en su lugar (a no ser que se indique expresamente lo contrario). No obstante, la única diferencia sintáctica entre un bloque de sentencias y una sentencia es que el primero no lleva un punto y coma ";" tras la llave "}" mientras que las sentencias simples obligatoriamente si lo llevan.

El propósito de crear bloques de sentencias es definir ámbitos de visibilidad para las declaraciones. Esto significa que las declaraciones de tipos y variables realizadas dentro de un bloque sólo son visibles por las sentencias del mismo. Como dentro de un bloque se pueden declarar nuevos bloques anidados, en los bloques hijos se verán las declaraciones locales y las realizadas en cada uno de los bloques de la jerarquía de anidamiento. Sin embargo, la visibilidad de declaraciones no se trata en esta práctica, aunque sí se debe permitir construir estos bloques anidados. El listado 16 muestra un ejemplo de uso de bloques anidados.

Listado 16. Ejemplo de bloques anidados

```
int x = 0;

void f () {

   int a=1, b = 1;

   { int a=2, c = 2;

     { int d = 3

       printi (a); /* Escribe 2, referencia no local */

       printi (b); /* Escribe 1, referencia no local */

       printi (c); /* Escribe 2, referencia no local */

       printi (d); /* Escribe 3, referencia local */

       printi (x); /* Escribe 0, referencia global */

     }

   }

}
```

Conviene ahora describir dónde se da potencialmente el uso de los bloques de sentencias. En concreto podemos reconocer 3 situaciones gramaticales diferentes:

- **Bloques anónimos**. Como acabamos de describir un bloque de sentencias se comporta como una sentencia ordinaria y puede aparecer en cualquier parte donde se espere una sentencia. Este tipo de situaciones, poco comunes en la práctica, se identifica con bloques anónimos ya que no aparecen vinculadas a ninguna construcción gramatical adjunta.

- **Bloques de control de flujo**. Las sentencias de control de flujo condicionales e iterativas, que se estudiaran más adelante en esta sección, hacen uso potencial de bloques de sentencias para definir la colección de sentencias que deben ejecutarse si se verifican las condiciones de evaluación pertinentes. Pueden verse ejemplos de este tipo de bloques en los listados que ejemplifican las sentencias de control.

- **Bloques de funciones**. Tal y como se describió en la sección 2.2.5, la declaración de una función adjunta a la cabecera un bloque de sentencias que describe el cuerpo de la misma.

Esta estructura sintáctica articula una colección anidada de **ámbitos de visibilidad**. En este sentido podemos distinguir entre, 1) el ámbito de visibilidad global asociado a las declaraciones realizadas al inicio del programa, que es visible por todas las funciones declaradas; 2) el ámbito de visibilidad asociado a cada función y 3) los ámbitos de visibilidad asociados a bloques anónimos declarados dentro del cuerpo de cada función. Cuando

hacemos referencia a una variable desde cualquier punto del programa podemos caer en uno de los 3 casos siguientes:

- **Referencias globales.** Las referencias globales son aquellas referencias a variables que han sido declaradas en el ámbito global del programa.

- **Referencias locales**. Las referencias locales son aquellas variables que se encuentran declaradas dentro del ámbito local donde se realiza la referencia (véase ejemplo del listado 16).

- **Referencias no locales.** Las referencias no locales son referencias a variables que no son globales y tampoco son locales estando declaradas en algún punto dentro de la jerarquía de anidamiento de bloques (véase ejemplo del listado 16).

Sentencia de control de flujo condicional if – then – else

Esta sentencia permite alterar el flujo normal de ejecución de un programa en virtud del resultado de la evaluación de una determinada expresión lógica. Sintácticamente esta sentencia puede presentarse de esta forma:

```
if (expresiónLogica)
  << sentencia o bloque de sentencias 1 >>

[else
  << sentencia o bloque de sentencias 2 >>]
```

Los corchetes en este caso indican una parte opcional, es decir, puede aparecer o no. Donde `expresionLogica` es una expresión lógica, es decir aritmética, que se evalúa para comprobar si debe ejecutarse la sentencia o bloques de sentencias `1` o `2`. En concreto si el resultado de dicha expresión es cierta, es decir, es mayor que 0, se ejecuta el bloque de sentencias `1`. Si existe `else` y la condición es falsa (`0`), se ejecuta el bloque de sentencias `2`. Es necesario que `expresionLogica` esté siempre entre paréntesis.

Este tipo de construcciones puede anidarse con otras construcciones de tipo if-then-else o con otros tipos de sentencias de control de flujo que estudiaremos a continuación. Hay que destacar que esta sentencia es inherentemente ambigua debido al problema del "*else ambiguo*". Esta ambigüedad se da cuando se declaran varios if de forma consecutiva y un único else al final, ya que no queda claro con que sentencia if debe asociarse el else. Se asume para resolverla que el else viene siempre asociado al if sintácticamente más cercano, forzando una máxima precedencia al else. El listado 17 ilustra varios ejemplos de uso.

Listado 17. Ejemplos de sentencia if –else

```
/* Una sola sentencia. No es necesario usar llaves de bloque */

if (a==b)

      a=a+1;
```

```
/* Una sola sentencia, pero usando un bloque */
if (a==b) {
      a=a+1;
}
/* varias sentencias */
if (a==b) {
   a=a+1;
   b=b+1;
}
/* uso de else */
if (esCierto==1)
      valor=1;
else
      valor=0;

/* varias sentencias en if y una sola en else */
if (esCierto==1){
      valor=1;
      a=a+1;
}
else
      valor=0;
/* varias sentencias en if y else */
if (esCierto==1){
      valor=1;
      a=a+1;
}
else  {
      valor=0;
      a=a-1;
```

```
}
if (esCierto==1){
      valor=1;
}
else  {
      valor=0;
}
```

Sentencia de control de flujo condicional while

Esta sentencia se utiliza para realizar iteraciones sobre un bloque de sentencias alterando así el flujo normal de ejecución del programa. *Antes* de ejecutar en cada iteración el bloque de sentencias, el compilador evalúa una determinada expresión lógica para determinar si debe seguir iterando el bloque o continuar con la siguiente sentencia a la estructura `while` mientras se cumpla una determinada condición, determinada por una expresión. Su estructura es:

```
while (expresionLogica)
    << sentencia o bloque de sentencias >>
```

Donde `expresionLogica` es la expresión lógica a evaluar en cada iteración y `sentencias` el bloque de sentencias a ejecutar en cada vuelta. Es decir, si la expresión lógica es cierta (distinto de 0), se ejecuta el bloque de sentencias. Después se comprueba de nuevo la expresión. Este proceso se repite una y otra vez hasta que la condición sea falsa (0). En el siguiente listado se muestra un ejemplo de este tipo de bucle.

La sentencia `while` permite que otras estructuras de control formen parte del bloque de sentencia a iterar de forma que se creen anidamientos. En el listado 18 se muestran unos ejemplos de uso.

Listado 18. Ejemplos de sentencia while

```
/* while con bloque de sentencias */
while (a<5){
   a=a+1;
   escribeEnt (a);
}
/* while con una sola sentencia simple */
while (a<5)
      a=a+1;
```

```
/* while con bloque y con una única sentencia simple dentro */

while (a<5) {

      a=a+1;

}
```

Sentencia de control de flujo iterativo for

La sentencia `for` es otra sentencia de control de flujo iterativo funcionalmente similar a la sentencia `while`. La estructura sintáctica de una sentencia `for` es:

```
for (indice = expresionComienzo; condicion; actualizacion)

      << sentencia o bloque de sentencias >>
```

Donde `indice` es el índice que regula la iteración, `expresionComienzo` es una expresión aritmética que indica el valor inicial del índice. `condicion` es una expresión lógica que indica cuando han de terminar las iteraciones. Es decir, en cada iteración se evaluará la condición y si es verdadera se realizará una nueva. `actualizacion` será una expresión aritmética y debería incrementar el valor del índice para un correcto funcionamiento del bucle. Esta actualización se llevará a cabo al terminar las sentencias y antes de evaluar la condición de nuevo.

Hay que hacer notar que no es posible declarar la variable `indice` dentro de la sentencia `for`. La declaración de variables ha de ser anterior a la de las sentencias. La estructura for permite que otras estructuras de control formen parte del bloque de sentencia a iterar de forma que se creen anidamientos, de forma análoga a lo explicado en la sentencia while. El listado 19 muestra unos ejemplos de sentencia for.

Listado 19. Ejemplos de sentencia for

```
/* sin bloque. El for sólo iterará sobre la sentencia b=b+1,
cuando termine ejecutará la siguiente asignación x=2 */

for (a=0;a<5;a=a+1)

      b=b+1;

x=2;

/* con bloque */

for (a=0;a<5;a=a+1) {

      printi(a);

      b=b+1;

}
```

Sentencias de invocación a una función void

Como ya se discutió con anterioridad, en cUNED es posible declarar funciones que devuelvan el tipo void para que se comporten como procedimientos. Esto implica que su invocación no es una expresión sino una sentencia. Por tanto una función con tipo de retorno void no puede ubicarse en un contexto sintáctico donde se espera una expresión sino solamente donde se espera una sentencia. El compilador debe detectar esta propiedad de la función y emitir un error en caso de que se use una llamada a procedimiento donde se esperaba una función. Como la diferenciación entre las llamadas a funciones y procedimientos se puede delegar al análisis semántico, para esta práctica no se va a tratar. Por tanto, el compilador de esta práctica debe permitir la innovación de funciones tanto como expresiones como sentencias.

Sentencias de Entrada / Salida

El lenguaje cUNED dispone de una serie de procedimientos predefinidos que pueden ser utilizados para emitir mensajes de distinto tipo por la salida estándar (pantalla). Estas funciones están implementadas dentro del código del propio compilador lo que implica 1) que son funciones especiales que están a disposición del programador y 2) constituyen palabras reservadas del lenguaje y por tanto debe considerarse un error la declaración de identificadores con dichos nombres. En concreto disponemos de 2 procedimientos. A continuación los detallamos.

- **printi**(*parametro*). Este procedimiento puede mostrar por pantalla el valor de un parámetro, que debe ser un identificador o expresión. En caso de intentar mostrar una matriz o registro completo, etc., se debe generar un error.

  En caso de no recibir ningún parámetro no escribiría nada por pantalla, pero no sería un error. Este procedimiento sólo admite un único parámetro.

  Un caso especial es la impresión de expresiones lógicas. Al ser en verdad expresiones aritméticas se permiten en esta sentencia.

- **printc**(*parametro*). Es similar al printi pero se utiliza para imprimir cadenas de texto únicamente. Sólo debe recibir un parámetro, correspondiente a una cadena de texto entre comillas.

No han de considerarse el uso de caracteres especiales ni secuencias de escape. No está permitido el uso de \n para saltos de línea. El listado 20 muestra unos ejemplos de uso.

Listado 20. Ejemplos de entrada / salida

```
printi (1);  /* Escribe 1 */

printi (x);  /* Escribe el valor de la variable x */

printi (x+1);

printi (cita1.urgente);

printi (a<b);

printi (a=2);
```

```
printi (“hola”); /* ERROR. No admite cadenas de caracteres */

printi (a,b);    /* ERROR. Sólo se admite un parámetro */

printc (“hola”);

printc (a);      /* ERROR. Sólo acepta cadenas de caracteres */

printc (“hola”\n); /* ERROR. No se admiten caracteres
                      especiales*/
```

2.3 Gestión de errores

Un aspecto importante en el compilador es la gestión de los errores. Se valorará la cantidad y calidad de información que se ofrezca al usuario cuando éste no respete la descripción del lenguaje propuesto.

Como mínimo se exige que el compilador indique el tipo de error: léxico o sintáctico. Debe indicarse obligatoriamente el número de línea en que ha ocurrido. Por lo demás, se valorarán intentos de aportar más información sobre la naturaleza del error, por ejemplo:

- Errores léxicos: Aunque algunos errores de naturaleza léxica no puedan ser detectados a este nivel y deban ser postergados al análisis sintáctico donde el contexto de análisis es mayor, en la medida de lo posible deben, en esta fase, detectarse el mayor número de errores. Son ejemplos de errores léxicos: literal mal construido, identificador mal construido, carácter no admitido, etc.

- Errores sintácticos: todo tipo de construcción sintáctica que no se ajuste a las especificaciones gramaticales del lenguaje constituye un error sintáctico.

No se debe realizar una recuperación de errores a nivel léxico. Así por ejemplo, si el compilador encuentra un carácter extraño en el código fuente, éste emitirá un mensaje de error y abortará el proceso de compilación. Si el alumno lo desea, el compilador generado puede recuperarse de los errores sintácticos que se encuentre durante el proceso de análisis de un programa. Esto implica que deben utilizarse los mecanismos pertinentes para que cuando se encuentre un contexto sintáctico de error el compilador genere un mensaje y continúe con el análisis con el ánimo de emitir la mayor cantidad de mensajes de error posible y no solamente el primero.

3 Descripción del trabajo

En esta sección se describe el trabajo que ha de realizar el alumno. La práctica es un trabajo amplio que exige tiempo y dedicación. De cara a cumplir los plazos de entrega, recomendamos avanzar constantemente sin dejar todo el trabajo para el final. Se debe abordar etapa por etapa, pero hay que saber que todas las etapas están íntimamente ligadas entre sí, de forma que es complicado separar unas de otras. De hecho, es muy frecuente tener que revisar en un punto decisiones tomadas en partes anteriores, especialmente en lo que concierne a la gramática.

La práctica ha de desarrollarse en **Java** (se recomienda utilizar la última versión disponible)**.** Para su realización se usarán las herramientas JFlex, Cup y ENS2001 además de *seguir la estructura de directorios y clases que se proporcionará*. Más adelante se detallan estas herramientas.

En este documento no se abordarán las directrices de implementación de la práctica que serán tratadas en otro diferente. El alumno ha de ser consciente de que se le proporcionará una estructura de directorios y clases a implementar que ha de seguir fielmente.

Es responsabilidad del alumno visitar con asiduidad el *tablón de anuncios* y el foro del Curso Virtual, donde se publicarán posibles modificaciones a este y otros documentos y recursos.

3.1 División del trabajo

A la hora de desarrollar la práctica se distinguen **dos especificaciones** diferentes sobre la misma que denominaremos A y B. Cada una de ellas supone una carga de trabajo equivalente y prescribe la implementación de un subconjunto de la especificación descrita en los apartados anteriores de este documento

Para las entregas de febrero y septiembre, cada alumno deberá implementar *solamente* una de las dos especificaciones. La especificación que debe realizar depende de su número de DNI. Asi:

```
Si DNI es par → Especificación A

Si DNI es impar → Especificación B.
```

El compilador debe respetar la descripción del lenguaje que se hace a lo largo de esta sección. Incorporar características no contempladas no sólo no se valorará, sino que se considerará un error y **puede suponer un suspenso en la práctica**.

A continuación se detallan las funcionalidades que incorporan cada una de las especificaciones A y B. Para cada funcionalidad, la "X" indica que esa característica debe implementarse mientras que el "-" indica que no debe implementarse.

Todas las funcionalidades que no se incluyan en esta tabla pertenecen a ambas especificaciones.

FUNCIONALIDAD		A	B
Tipos de datos	Matrices	-	X
	Registros	X	-
Paso de parámetros	Por valor	X	-
	Por referencia	-	X
Operadores aritméticos	+	-	X
	-	X	-
Operadores relacionales	<	-	X
	>	X	-
	==	X	-
	!=	-	X
Operadores lógicos	&&	-	X
	||	X	-
Sentencias de control de flujo	while	-	X
	for	X	-

3.2 Entregas

Antes de empezar, nos remitimos al documento "Normas de la asignatura" que podrá encontrar en el entorno virtual para más información sobre este tema. Es fundamental que el alumno conozca en todo momento las normas indicadas en dicho documento. Por tanto en este apartado se explicará únicamente el contenido que se espera en cada entrega.

3.2.1 Fechas y forma de entrega

Las fechas límite para las diferentes entregas son las siguientes:

Febrero	16 de febrero
Septiembre	10 de septiembre

Para entregar su práctica el alumno debe acceder a la sección Entrega de Trabajos del Curso Virtual (los enlaces a cada entrega se activan automáticamente unos meses antes). Si una vez entregada desea corregir algo y entregar una nueva versión, puede hacerlo hasta la fecha límite. Los profesores no tendrán acceso a los trabajos hasta dicha fecha, y por tanto no

realizarán correcciones o evaluaciones de la práctica antes de tener todos los trabajos. En ningún caso se enviarán las prácticas por correo electrónico a los profesores.

Puesto que la compilación y ejecución de las prácticas de los alumnos se realiza de forma automatizada, *el alumno debe respetar las normas de entrega indicadas en el enunciado de la práctica*.

Se recuerda que es necesario superar una **sesión de control obligatoria** a lo largo del curso para aprobar la práctica y la asignatura. Nos remitimos al documento de normas de la asignatura, dónde viene explicado las fechas y normativa a aplicar. No obstante, recordamos que para superar la sesión, el tutor comprobará que el alumno ha realizado el analizador léxico y está dando comienzo al analizador sintáctico.

3.2.2 Formato de entrega

El material a entregar, mediante el curso virtual, consiste en un único archivo comprimido en formato **rar** cuyo nombre debe construirse de la siguiente forma:

Grupo de prácticas + "-" + Identificador de alumno + "." + extensión

(Ejemplo: a-gonzalez1.rar)

Dicho archivo contendrá la estructura de directorios que se proporcionará en las directrices de implementación. Esta estructura debe estar en la raíz del fichero rar. **No** se debe de incluir dentro de otro directorio, del tipo, por ejemplo: “pdl”, “practica”, “arquitectura”, etc.

En cuanto a la memoria, será un breve documento llamado "memoria" con extensión .doc o .pdf y situado en el directorio correspondiente de la estructura dada.

El índice de la memoria será:

> Portada obligatoria (El modelo estará disponible en el curso virtual).
>
> 1. El analizador léxico
>
> 2. El analizador sintáctico
>
> 3. Conclusiones
>
> 4. Gramática
>
> En este punto se ha de incluir un esquema con las producciones de la gramática generada

En cada apartado habrá que incluir únicamente comentarios relevantes sobre cada parte y no texto “de relleno” ni descripciones teóricas, de forma que la extensión de la memoria esté comprendida aproximadamente entre 2 y 5 hojas (sin incluir el esquema de las producciones de la gramática). En caso de que la memoria no concuerde con las decisiones tomadas en la implementación de cada alumno la práctica puede ser considerada suspensa.

3.2.3 Trabajo a entregar

La entrega cubrirá únicamente la parte de análisis léxico y sintáctico: sólo se pide que el compilador procese el archivo fuente e identifique y escriba por pantalla los errores léxicos y sintácticos encontrados en el archivo fuente.

3.2.3.1 Análisis léxico

Para realizar esta fase se usará la herramienta *JFLex*. El primer paso es familiarizarse con la herramienta a través de los ejemplos básicos proporcionados en la página de la asignatura y después realizar la especificación léxica del lenguaje, compilarla y probarla. En esta fase, es importante identificar el número de línea y columna en el que aparece un símbolo, de cara a proporcionar información de contexto, dentro del código fuente al analizador sintáctico. Al finalizar esta etapa, se debe obtener el código fuente en java de un scanner capaz de identificar todos los TOKENS de un programa fuente en el lenguaje pedido así como detectar los posibles errores léxicos que éste pudiera contener.

3.2.3.2 Análisis sintáctico

Para la etapa de análisis sintáctico se utilizará la herramienta *Cup*. En primer lugar hay que dedicar tiempo a escribir la gramática del lenguaje. Esta gramática es el eje de la práctica y debe estar cuidadosamente diseñada, abarcando todas las posibles sentencias que pueden aparecer en un programa fuente. Especial atención merece la precedencia de operadores, un problema que se puede solucionar utilizando la directiva “precedence” de Cup. El código Cup permite integrar el análisis léxico de JFlex, de forma que al finalizar esta etapa el compilador debe reconocer las sentencias del programa fuente y detectar posibles errores sintácticos, indicando por pantalla el número de línea en que ha ocurrido el error.

3.2.3.3 Comportamiento esperado del compilador

El compilador debe procesar archivos fuente. Si aparecen errores léxicos o sintácticos, debe notificarlos por pantalla. En caso contrario se emitirá un mensaje por pantalla indicando que el código fuente no tiene errores sintácticos y que el proceso de compilación ha terminado con éxito (pese a que aún no se haya generado un fichero de salida).

4 Herramientas

Para el desarrollo del compilador se utilizan herramientas de apoyo que simplifican enormemente el trabajo. En concreto, se utilizarán las indicadas en los siguientes apartados. Para cada una de ellas se incluye su página web e información relacionada. En el curso virtual de la asignatura pueden encontrarse una versión de todas estas herramientas junto con manuales y ejemplos básicos.

4.1 JFlex

Se usa para especificar analizadores léxicos. Para ello se utilizan reglas que definen expresiones regulares como patrones en que encajar los caracteres que se van leyendo del archivo fuente, obteniendo tokens.

Web JFlex: http://jflex.de/

4.2 Cup

Esta herramienta permite especificar gramáticas formales facilitando el análisis sintáctico para obtener un analizador ascendente de tipo LALR. Además Cup también permite asociar acciones java entre los elementos de la parte derecha de la gramática de forma que éstas se vayan ejecutando a medida que se va construyendo el árbol de análisis sintáctico. Estas acciones permitirán, de cara a la segunda entrega de la práctica implementar las fases de análisis semántico y generación de código intermedio.

Web Cup: http://www2.cs.tum.edu/projects/cup/

4.3 Jaccie

Esta herramienta consiste en un entorno visual donde puede especificarse fácilmente un analizador léxico y un analizador sintáctico y someterlo a pruebas con diferentes cadenas de entrada. Su uso resulta muy conveniente para comprender como funciona el procesamiento sintáctico siguiendo un proceso ascendente. Desde aquí recomendamos el uso de esta herramienta para comprobar el funcionamiento de una expresión gramatical. Además, puede ayudar también al estudio teórico de la asignatura, ya que permite calcular conjuntos de primeros y siguientes y comprobar conflictos gramaticales. Pero **No es necesaria** para la realización de la práctica

4.4 Ant

Ant es una herramienta muy útil para automatizar la compilación y ejecución de programas escritos en java. La generación de un compilador utilizando JFlex y Cup se realiza mediante una serie de llamadas a clases java y al compilador de java. Ant permite evitar situaciones habituales en las que, debido a configuraciones particulares del proceso de compilación, la práctica sólo funciona en el ordenador del alumno.

Web Ant: http://ant.apache.org/

5 Ayuda e información de contacto

Es **fundamental y obligado** que el alumno consulte regularmente el Tablón de Anuncios y el foro de la asignatura, accesible desde el Curso Virtual para los alumnos matriculados. En caso de producirse errores en el enunciado o cambios en las fechas siempre se avisará a través de este medio. El alumno es, por tanto, responsable de mantenerse informado.

También debe estudiarse bien el documento que contiene **las normas y el calendario** de la asignatura, incluido en el Curso Virtual.

Se recomienda también la utilización de los foros como medio de comunicación entre alumnos y de estos con el Equipo Docente. Se habilitarán diferentes foros para cada parte de la práctica. Se ruega elegir cuidadosamente a qué foro dirigir el mensaje. Esto facilitará que la respuesta bien por otros compañeros o por el Equipo Docente sea más eficiente.

Esto no significa que la práctica pueda hacerse en común, por tanto **no debe compartirse código**. Se utilizará un programa de detección de copias al corregir la práctica.

El alumno debe comprobar si su duda está resuelta en la sección de Preguntas Frecuentes de la práctica (FAQ) o en los foros de la asignatura antes de contactar con el tutor o profesor. Por otra parte, si el alumno tiene problemas relativos a su tutor o a su Centro Asociado, debe contactar con el coordinador de la asignatura Anselmo Peñas (anselmo@lsi.uned.es).

Anexo A. Programa de ejemplo completo

El siguiente listado muestra un ejemplo de un programa complejo en cUNED que contiene varios ejemplos de todas las estructuras sintácticas que se han definido en este documento.

Listado 21. Ejemplo de un programa

```
/* constantes simbólicas */

#define FALSE 0;

#define TRUE 1;

#define MAYORDEEDAD 17;


/* tipos globales */

struct Tpersona {

      int dni;

      int edad;

}

struct Tfecha {

      int dia, mes, anyo;

}


/* variables globales */

Tpersona persona1;


/* funciones */

void excribe (int x) {

      printi (x);

}

int mayorDeEdad (int edad) {


      if (edad>MAYORDEEDAD){

            return TRUE;

      }
```

```
	else{
		return FALSE;
	}
}
void imprimePersona(int dni, int edad){
	int debug;
	escribe(dni);

	debug=1;
	/* nuevo bloque */
	{
		/* variable local al bloque*/
		int x;

		/* debug y edad referencias no locales*/
		if (debug=1){
			x=edad;
			escribe(x);
		}
	}

}
/* Función principal */
void main (){
	/* tipos locales*/
	struct Tfecha {
		int dia, mes, anyo;
	}
	struct Tcita {
		Tpersona empleado;
```

```
        Tfecha fecha;
        int urgente;
    }
    /* variables locales*/
    Tfecha f;
    Tcita c1, c2;
    int numeroCitas;
    int urge=1;

    /* sentencias y bloques*/
    persona1.dni=1234;
    persona1.edad=23;

    if (mayorDeEdad(persona1.edad){
        /* edad variable local del bloque if*/
        int edad=persona1.edad;
        printc("Persona:");
        imprimePersona(persona.dni, edad);
        printc("Mayor de edad");
    }
    f.dia=1;
    f.mes=12:
    f.anyo=2014;
    c1.empleado=persona1;
    c1.fecha=f;
    c1.urgente=urge;
}
```