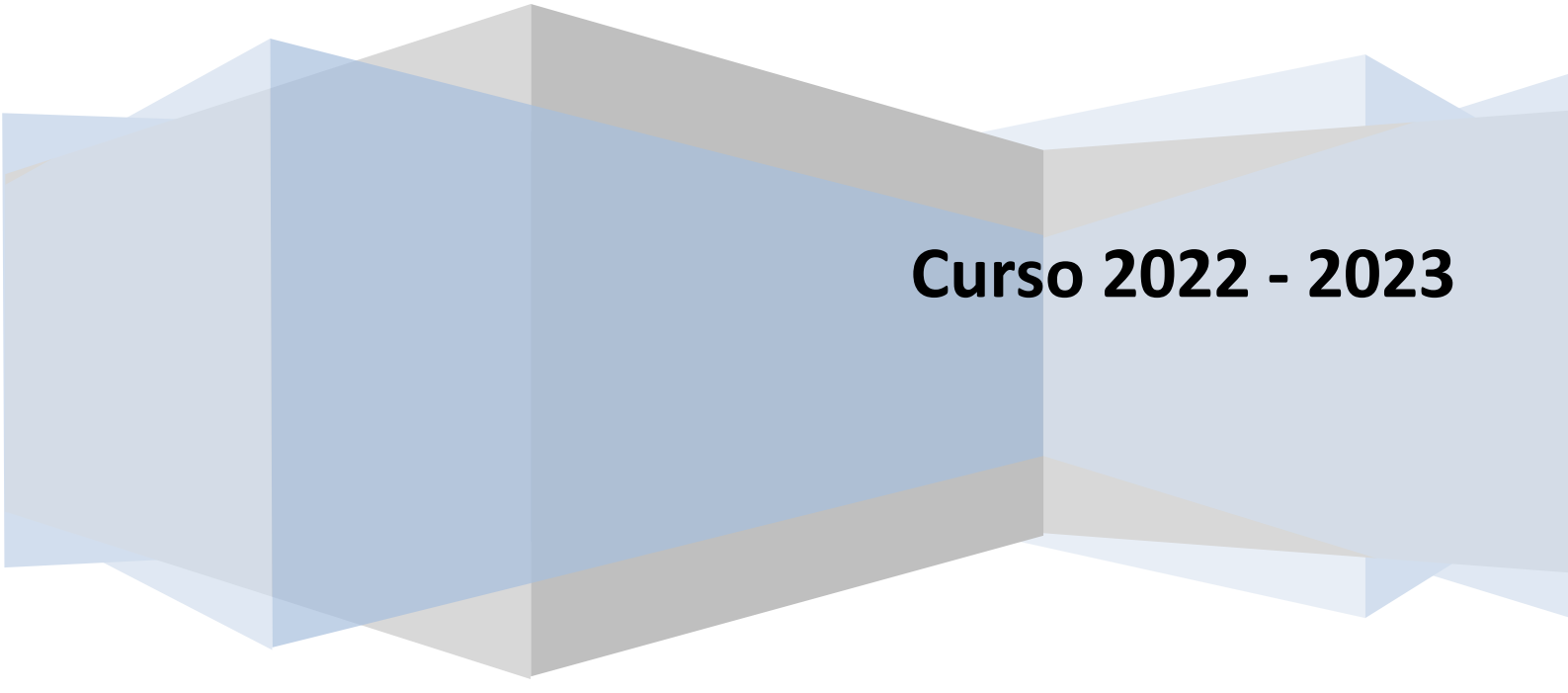
Universidad Nacional de Educación a Distancia
Departamento de Lenguajes y Sistemas Informáticos

Práctica de Procesadores del Lenguaje II

Especificación del lenguaje cES v1.0

Dpto. de Lenguajes y Sistemas Informáticos
ETSI Informática, UNED

Alvaro Rodrigo
Anselmo Peñas (coordinador)
Teresa Sastre



Curso 2022 - 2023

ÍNDICE

1 Introducción

En este documento se define la práctica de la asignatura Procesadores del Lenguaje II correspondiente al curso 2022-2023. El objetivo de la práctica es realizar un compilador del lenguaje cES, con el que se trabajó en la práctica de la asignatura Procesadores del Lenguaje I del curso 2021-2022 (salvo alguna pequeña modificación, como no permitir expresiones como sentencias).

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 junto con las herramientas a utilizar para su realización.

A lo largo de este documento se explicará la sintaxis y el comportamiento del compilador de cES, por lo que es importante que el *estudiante lo lea detenidamente y por completo*.

2 Descripción del lenguaje cES

Este apartado es una descripción técnica del lenguaje cES, una versión del lenguaje C con la característica particular de que las palabras reservadas estarán en castellano. 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 válidas.

2.1 Aspectos Léxicos

Desde el punto de vista léxico, un programa escrito en cES 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 o nombres de funciones, 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 cES 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. 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 */

     */

*/

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

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

/* Este es un comentario no cerrado
```

2.1.2 Constantes literales

En cES se pueden utilizar constantes *literales* para escribir programas. No obstante, 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 representan 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

- **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 `escribe()`(ver más adelante), pero no es necesario tratarlas en ningún otro contexto. Es decir, *no se crearán variables de este tipo*. No se tendrá en cuenta el tratamiento de caracteres especiales dentro de la cadena ni tampoco secuencias de escape tales como \t, \n, etc.

Nótese que en el lenguaje cES, de forma similar a como ocurre en C, no existen constantes lógicas (tipo boolean). Cualquier expresión numérica sirve como condición en un contexto

lógico: si el resultado de la expresión es 0 ésta se interpretará como falso (*false*), y si es distinto de cero, será verdadero (*true*) tal y como se discutirá más adelante.

2.1.3 Identificadores

Un identificador consiste, desde el punto de vista léxico, en una secuencia ordenada de letras y dígitos que comienzan obligatoriamente por una letra. Los identificadores se usan para nombrar entidades del programa tales como las variables o las funciones definidas por el programador. El lenguaje SI 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.

2.1.4 Palabras reservadas

Las palabras reservadas son entidades del lenguaje que, a nivel léxico, tienen un significado especial de manera que no pueden ser utilizadas para nombrar otras entidades como variables, constantes, funciones o 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 EN cES	DESCRIPCIÓN
`caso`	Usado en sentencias "alternativas" para delimitar una o varias sentencias
`constante`	Define una constante simbólica numérica
`corte`	Usado en sentencias "alternativas". Delimita un caso
`entero`	Tipo entero
`escribe`	Muestra por pantalla un texto
`escribeEnt`	Muestra por pantalla un entero
`alternativas`	Cabecera de una sentencia condicional "alternativas"
`mientras`	Cabecera de una sentencia de flujo "mientras"

	`pordefecto`	Caso por defecto en una sentencia “alternativas”
	`principal`	Nombre de la función principal
	`devuelve`	Instrucción de retorno en funciones
	`si`	Cabecera de una sentencia condicional “si”
	`sino`	Parte optativa de una sentencia condicional “si”
	`tipo`	Marca la declaración de un tipo
	`vacio`	Tipo vacio

Como el lenguaje es sensible al caso, las palabras reservadas solo se reconocerán como tales si se escriben en minúscula. Es decir, si se escribe *ENTERO*, se considerará como un identificador y no como una palabra reservada

2.1.5 Delimitadores

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

DELIMITADOR	DESCRIPCIÓN
“	Delimitador de constante literal de cadena.
( )	Delimitadores de expresiones y de parámetros
[ ]	Delimitador de rango en una declaración de un vector.
,	Delimitador en listas de identificadores
;	Delimitador en secuencias de sentencias.
:	Delimitador en una sentencia “alternativas”.

2.1.6 Operadores

En cES 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) * (producto)
Operadores relacionales	< (menor) == (igual)
Operadores lógicos	&& (conjunción lógica) ! (negación lógica)
Operadores especiales	++ (autoincremento)
Operadores de asignación	= (asignación) += (asignación con suma)
Operadores de acceso	[ ] (acceso a elemento de vector)

2.2 Aspectos Sintácticos

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

2.2.1 Estructura de un programa y ámbitos de visibilidad

Un programa en cES es un fichero de código fuente con extensión '.ces' que comienza por una sección[1] opcional donde se declara el conjunto de constantes simbólicas, seguido de secciones para la declaración de variables y tipos globales y la declaración de funciones no anidadas entre las que se encuentra la función principal, sin parámetros, punto de arranque del programa.

[1] El término sección es utilizado en este contexto para referirnos a un conjunto de sentencias con un mismo propósito, no existiendo ningún mecanismo sintáctico para la delimitación del mismo.

Dentro de las funciones, en primer lugar aparece una sección opcional para la declaración de variables locales y después comienza una secuencia ordenada de sentencias. Estas sentencias pueden organizarse en bloques mediante el uso de los delimitadores { y } como típicamente ocurre al definir bucles o condicionales que implican la ejecución de un bloque de sentencias.

La estructura sintáctica de los bloques es anidante y resulta idéntica a la de las funciones (excepto que estas últimas no se pueden anidar). Es decir, dentro de un bloque, es posible comenzar por una sección opcional de declaración de variables y continuar por una secuencia ordenada de sentencias que a su vez pueden contener a otros bloques.

Dado que, como se verá más adelante, la declaración de funciones también se realiza acotando el código de la función entre los delimitadores { y } podemos considerar para los efectos de la subsiguiente discusión a las funciones como un caso particular de bloque (no anidante). En breve, cada vez que en cES aparece un delimitador { puede considerarse que comienza un nuevo bloque. Cada vez que se encuentra un delimitador } el bloque abierto se cierra.

El uso de bloques en cES se utiliza para definir ámbitos de visibilidad. En cada ámbito sólo están accesibles un subconjunto de todos los identificadores (constantes, tipos, variables o subprogramas) definidos en el programa. Es decir, las reglas de ámbito de un lenguaje definen el alcance que tienen los identificadores de un programa dentro de cada punto de este. En este sentido, en PL1UnedES distinguimos 3 posibilidades:

- **Referencias globales**. Estos identificadores globales son los que se declaran directamente dentro del programa principal. Se dice que pertenecen al ámbito global del mismo y, por tanto, están accesibles desde cualquier otro ámbito, ya sea éste el programa principal o cualquiera de los subprogramas definidos.

- **Referencias locales**. Los identificadores son locales a un ámbito cuando son solamente accesibles desde dentro de dicho ámbito. Por ejemplo, todas las variables, constantes y tipos definidos dentro de un subprograma son locales al mismo, así como la colección de sus parámetros.

- **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.

En cuanto al uso de subprogramas, existen las siguientes restricciones:

- Todo subprograma que sea invocado por otro subprograma debe ser declarado previamente dentro del código fuente del programa. Es decir, si g invoca a f, f debe declararse antes que g.

- Debe darse soporte a la ejecución de subprogramas recursivos. Un subprograma f es recursivo si dentro de su código se vuelve a invocar a sí mismo. Sin embargo, no se da soporte a la recursividad indirecta. Dos subprogramas f y g guardan una relación de recursividad indirecta si dentro del cuerpo de f se invoca a g, y recíprocamente, dentro de g se invoca a f.

El siguiente listado muestra un ejemplo de la estructura de un programa en cES.

Listado 2. Ejemplo de un programa en cES

```
#constante VERDADERO 1
#constante FALSO 0
#constante TAM 7
tipo entero vector [TAM];
entero suma;
entero mayorDiez;
entero funSuma (entero x, entero y) {
   entero i;
   entero suma;
   suma = x + y;
   devuelve suma;
}
vacio principal () {
   entero a, b;
   entero i;
   suma = funSuma (a, b);
   mayorDiez = ! (suma < 10);
   si (mayorDiez == VERDADERO) { escribe ("mayor"); }
   sino {escribe ("menor");}
   devuelve;
}
}
```

El programa del listado 2, comienza por la definición de 3 constantes de tipo entero. Las dos primeras (VERDADERO y FALSO) con significado lógico y la tercera (TAM) un entero que referencia el tamaño de los vectores del programa. En el ejemplo, a continuación, aparece una declaración de tipos global llamada vector. Seguidamente aparece declarada la variable global suma que será utilizada por la función principal para almacenar el resultado de sumar dos valores enteros. Después tenemos la declaración de la variable global mayorDiez, que aunque representa un valor lógico, es especificada de tipo entero

A continuación, aparece la función funSuma, que toma como entrada dos parámetros de tipo entero. Finalmente se declara la función principal, prescriptivamente sin parámetros y con tipo de retorno vacio (se trata de un procedimiento y no devuelve valores). Dentro de ella se declaran dos variables enteras a y b en una misma línea y a continuación otra variable entera llamada i.

Ahora vamos a entrar en más detalle en cada uno de los aspectos sintácticos.

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 la ejecución del programa. Han de declararse al inicio de un programa obligatoriamente y no es posible su declaración dentro de funciones.

En cES todas las constantes simbólicas son de tipo entero. La sintaxis para la declaración de constantes simbólicas enteras es la siguiente:

```
#constante nombre valor;
```

Donde `nombre` es el nombre simbólico que recibe la constante definida dentro del programa y `valor` un valor constante de tipo entero (número) de manera que cada vez que este aparece referenciado dentro del código se sustituye por su valor establecido en la declaración. La cláusula `#constante` no marca el inicio de la sección de declaraciones de constantes, sino que para cada constante declarada es necesario utilizar esta cláusula. A continuación se muestran algunos ejemplos de declaración de constantes.

Listado 3. Ejemplo de declaración de constantes simbólicas enteras

```
#constante UNO 1;

#constante DOS 2;

#constante MIERCOLES 3;
```

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) son todos aquellos que se encuentran disponibles directamente para que el programador tipifique las variables de su programa. En concreto el lenguaje cES reconoce dos tipos primitivos: el tipo entero y el tipo especial vacío. En los apartados subsiguientes abordamos cada uno de ellos en profundidad.

Tipo entero

El tipo entero representa valores enteros positivos y negativos, aunque en cES solo vamos a tener positivos.

Desde el punto de vista sintáctico, el tipo entero se representa con la palabra reservada `entero`. La aplicación de este tipo aparece en distintos contextos de uso como en la declaración de variables, la declaración de parámetros de funciones, los tipos de retorno de las mismas, o la declaración de tipos compuestos (véase más adelante).

Es posible realizar asignación en línea al declarar variables de tipo entero. Es decir, se permiten declaraciones del tipo "`entero` a = 1;". En el listado 5 se presentan algunos ejemplos de estos usos, que se detallarán más adelante.

Listado 4. Ejemplos de uso del tipo entero

```
entero a;

entero sumar (entero x, entero y) { ...

tipo entero vector[10];
```

Tipo vacio

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 `vacio`. Un ejemplo particular de función de tipo procedimiento es la función principal del programa

Este tipo primitivo del lenguaje sólo presenta este uso por tanto NO pueden declararse variables ni parámetros ni vectores de tipo vacio. Además, la función principal deberá devolver siempre vacio. Puede considerarse que las funciones de tipo vacio representan a procedimientos ya que no devuelven ningún valor.

Listado 5. Ejemplos de uso del tipo vacio

```
vacio main () { ... }

vacio escribeEntero (entero x) {

      escribeEnt(x);

      devuelve;

}
```

2.2.3.2 Tipos Compuestos

Los tipos compuestos (también llamados tipos definidos por el usuario) 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 el código del programa (declaración de variables, parámetros, etc.). Así en cES no existen tipos anónimos. No es posible definir una estructura de datos para tipificar una variable directamente en la sección de declaración de variables. En su lugar hay que crear previamente un tipo estructurado (con nombre) y usar dicho nombre después en la declaración de variables.

En lo que respecta al sistema de tipos, la equivalencia de éstos es nominal, no estructural. Es decir, dos tipos serán equivalentes únicamente si comparten el mismo nombre. No es posible definir dos tipos con el mismo nombre aunque pertenezcan a distintos ámbitos, si desde un ámbito se puede alcanzar el otro ámbito. Esto debe controlarlo el analizador semántico.

La sintaxis para la declaración de un tipo compuesto depende de la clase de tipo de que se trate. En cES solo tenemos los vectores.

Tipo vector

Un vector es una estructura de datos que sirve para almacenar una secuencia ordenada de datos del mismo tipo. En cES sólo pueden declararse vectores de una dimensión y cuyo tipo base sea entero. Además, el tamaño del vector debe ser una constante, literal o simbólica y nunca una expresión.

La sintaxis de declaración para vectores es la siguiente:

```
tipo entero nombre[tam];
```

Donde `nombre` será el identificador del vector y `tam` una constante literal numérica o simbólica que indica el número de elementos que contiene el vector. El analizador semántico comprobará que `tam` es mayor que 0.

Listado 7. Ejemplo de uso de vectores

```
#constante MAX 7;

tipo entero vector1[MAX];

tipo entero vector2[3];

vector1 v1;

vector2 v2;
```

Obsérvese que la definición de vectores se apoya únicamente en el uso de tipos primitivos y no en el uso de otros tipos compuestos previamente definidos por el usuario. En este sentido NO existen en el lenguaje vectores de vectores (vectores multidimensionales).

2.2.4 Declaraciones de variables

En cES es necesario que las variables estén declaradas antes de utilizarlas. Estas declaraciones pueden ser globales o locales. Las variables globales se declaran en cualquier punto fuera del cuerpo de las funciones y las variables locales (a un bloque o función) se declaran *al inicio de dicho bloque*. El Analizador Semántico comprobará que dentro de un mismo ámbito una variable solo se declara una vez

Para declarar variables se utiliza la siguiente sintaxis:

```
nombre-tipo var1 [ = val1], var2 [ = val2], ...;
```

Donde `nombre-tipo` es el nombre de un tipo primitivo del lenguaje o definido por el usuario, `var1, var2,…` son identificadores para las variables creadas y `val1` y `val2`

son valores opcionales asignados (constantes literales). Nótese que cES contempla asignaciones en línea dentro de la declaración del estilo *entero a = 1*. El analizador semántico comprobará que estas asignaciones son correctas. A continuación se muestran algunos ejemplos de declaraciones de variables en cES (se supone que el tipo compuesto vector ha sido previamente declarado).

Listado 9. Ejemplo de declaración de variables en cES

```
entero a, b;

entero a = 1;

entero a = 1, b = 3;

vector miVector;
```

2.2.5 Declaración de funciones

En el lenguaje cES se pueden declarar funciones para organizar modularmente el código. Una función es una secuencia ordenada de instrucciones encapsuladas bajo un nombre y declarada con unos parámetros (puede no llevar ninguno). La sintaxis de la declaración de una función es la siguiente:

```
tipo-retorno nombre (tipo1 param1, tipo2 param2,...){
    /* declaración de tipos y variables locales */
    /* sentencias */
    devuelve expresión;
}
```

Donde nombre es el `nombre` de la función, `param1, param2,...` el nombre de cada uno de los parámetros, `tipo1, tipo2,...` los tipos de los parámetros y `tipo-retorno` el tipo del valor de retorno. Los únicos tipos permitidos en cES como valor de retorno de una función son `entero` o `vacio`.

Cuando la función no requiere ningún parámetro, ésta debe declarase haciendo uso de paréntesis vacíos. El siguiente código muestra un ejemplo de ello.

```
tipo-retorno nombre(){
    /* declaración de tipos y variables locales */
    /* sentencias */
    devuelve expresión;
}
```

Como ya dijimos con anterioridad, las funciones se comportan a todos los efectos como bloques sintácticos. Esto implica que, dentro del código de una función, primero aparece la declaración de tipos y variables locales a la misma (en este orden) y después la secuencia

ordenada de sentencias que constituyen su código. Es obligatorio el uso de las llaves para declarar el cuerpo de una función.

Dentro de este código cabe destacar el uso de la palabra clave `devuelve`. 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.

La sentencia `devuelve` puede venir seguida de una expresión cuyo tipo debe coincidir con el tipo de retorno especificado en la cabecera de la función (esta comprobación de tipo se realiza en el análisis semántico). Cuando la función ha sido declarada con `vacio` como tipo de retorno, la sentencia `devuelve` no viene acompañada de ninguna expresión (en otro caso sería un error, comprobación que se realiza en el análisis semántico). El siguiente listado muestra varios ejemplos de declaraciones de funciones.

Listado 10. Ejemplo de declaración de funciones

```
entero sumar (entero x, entero y) {
   devuelve x + y;
}
vacio escribeEntero (entero x) {
   escribeEnt (x);
   devuelve;
}

vacio saluda () {
   escribe ("Hola mundo");
}
```

2.2.5.1 Paso de parámetros a funciones

En el lenguaje cES es posible llamar a las funciones pasando expresiones, variables, constantes simbólicas y elementos de vectores como parámetros a una función. El Analizador Semántico comprobará que se pasa el número adecuado de parámetros y del tipo correcto.

En el listado 11 se incluyen ejemplos de declaraciones de funciones en cES:

Listado 11. Ejemplo de paso de parámetros

```
entero incrementa (entero x) {
  x++;
  devuelve x;
}
```

```
vacio principal () {
  entero a;
  a = 1;
  escribeEnt (a); /* Escribe 1 */
  incrementa (a);
  devuelve;
}
```

Recordamos además que la estructura de la declaración de funciones es plana. Es decir, en cES no es posible definir funciones dentro de otras funciones.

2.2.6 Sentencias y Expresiones

El cuerpo de un bloque está compuesto por un conjunto de expresiones y sentencias que manejan expresiones. En este apartado se describen detalladamente cada uno de estos elementos. Sintácticamente, cada bloque de código se organiza como una colección opcional de declaraciones de tipos y variables locales seguida de una secuencia ordenada de sentencias y/o expresiones separadas por delimitadores de punto y coma (`;`).

2.2.6.1 Expresiones

Una expresión es una construcción del lenguaje que devuelve un valor de retorno al contexto del programa donde aparece la expresión. Desde un punto de vista conceptual es posible tipificar las expresiones de un programa en cES en dos grandes grupos:

- **Expresiones aritméticas.** Las expresiones aritméticas son aquellas cuyo cómputo devuelve un valor de tipo entero al programa. 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 devuelven un valor de tipo entero. Asimismo, también son expresiones aritméticas la suma y producto de dos expresiones aritméticas, así como el autoincremento de una expresión aritmética.

- **Expresiones lógicas.** Las expresiones lógicas son aquellas que devuelven un valor de verdad (cierto o falso) al contexto sintáctico del programa llamante. En cES no existe un tipo primitivo para tipificar expresiones lógicas. 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 verdadero. A partir de ahora, llamaremos expresión lógica a toda expresión aritmética que es evaluada en un contexto lógico.

Expresiones de acceso a vectores

El acceso a los datos de un vector en cES se realiza de forma indexada. Para acceder individualmente a cada uno de sus elementos se utilizan los operadores de acceso a vector `[]`. Dentro de estos delimitadores es preciso ubicar una expresión aritmética cuyo valor debe estar comprendido entre 0 y `numero – 1`. Al poder usarse expresiones de acceso a vectores para acceder a un elemento de un vector, la comprobación de que no supere al índice de la declaración se debería hacer en tiempo de ejecución. Para simplificar el desarrollo de la práctica, no se tiene que detectar este tipo de errores. En caso de acceder mediante constantes enteras literales o simbólicas, sí es posible realizar esa comprobación en la fase de análisis semántico, y de hecho se debe de hacer en la práctica. El siguiente listado muestra distintos ejemplos de acceso a elementos de un vector.

Listado 12. Ejemplos de expresiones con acceso a elementos de un vector

```
tipo entero vector [3];

vector v;

v [0] = 1; v [1] = 2; v[2] = 3;

escribeEnt (v[0]); /* Escribe 1 */

escribeEnt (v[1+1]); /*  Escribe 3 */

escribeEnt (v[v[1]]); /* Escribe 3 */
```

Autoincrementos

Un tipo especial de expresiones aritmética son los autoincrementos. En cES es posible utilizar el operador de autoincremento (++) después de la referencia a una variable o elemento de un vector. El efecto de estos operadores es el de incrementar una unidad la variable que acompaña al operador. También se puede usar de forma aislada como si fuera una sentencia. Su sintaxis es:

```
ref++;
```

Por simplicidad, a efectos de evaluar la expresión que contiene un autoincremento, primero se incrementa la referencia y posteriormente se evalúa (ver ejemplo en el siguiente listado). A continuación se muestran unos ejemplos de uso.

Listado 14. Ejemplos de autoincremento

```
tipo entero vector [3];

vector v;

entero a, b;

v [0] = 1; v [1] = 2; v[2] = 3;

a = 0;

b = v[a++]; /*Posición a+1 del vector v*/
```

```
b = v[a];

b = v[a]++;
```

Llamadas a función

Para llamar a una función en cES ha de escribirse su nombre indicando entre paréntesis los parámetros de la llamada. Los parámetros pueden ser referencias a variables, elementos de vector, constantes o expresiones. Recordamos que el Analizador Semántico comprobará que se pasa el número adecuado de parámetros y del tipo correcto.

El uso de los paréntesis es siempre obligatorio. Así si una función no tiene argumentos en su llamada, es necesario colocar unos paréntesis vacios () tras su identificador. El analizador semántico debe detectar si la invocación de una función se realiza en el contexto adecuado. En el caso de una función que devuelve vacio, el contexto correcto es una sentencia. Si la función devuelve un entero, deberá ser llamada en el contexto de una expresión. En caso contrario, se debe generar un error.

En el siguiente listado se muestran algunos ejemplos.

Listado 15. Ejemplo llamadas a funciones

```
suma(a, b); /* Correcto si suma devuelve vacio */

operacion(centenas, v[2]);

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

datos = pideDatos ();/*correcto si pideDatos devuelve entero*/

a = funcion1();
```

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. La siguiente tabla resume la relación de precedencia y asociatividad y operadores (la prioridad decrece según se avanza en la tabla y los operadores en la misma fila tienen igual precedencia).

Precedencia	Asociatividad
()	Izquierdas
[]	Izquierdas
++ , !	Izquierdas
*	Izquierdas
+	Izquierdas

<	Izquierdas
==	Izquierdas
&&	Izquierdas

Para alterar el orden de evaluación de las operaciones en una expresión aritmética 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 18. Ejemplo de precedencia y asociatividad en expresiones aritméticas

```
2 + 2 + 3 * 2 /* Devuelve 10 */

2 + (2 + 3) * 2 /* Devuelve 12 */

1 && 0 && 1 /* Devuelve 0 (falso) */

2 < 3 < 4 /* Devuelve 1 (cierto)*/

2 < (3 < 4) /* Devuelve 0 (falso)*/
```

2.2.6.2 Sentencias

cES dispone de una serie de sentencias que permiten realizar determinadas operaciones dentro del flujo de ejecución de un programa. En concreto nos estamos refiriendo a las sentencias de asignación, las sentencias de control de flujo condicional e iterativo, y las sentencias de salida.

El cuerpo de algunas de las siguientes sentencias se compone de otro grupo de sentencias. En este caso se indica con `/* sentencias */`. *Esto quiere decir que puede tratarse de una única sentencia o un bloque de sentencias. En caso de ser un bloque se recuerda que debe estar delimitado por llaves*.

La sentencia `devuelve` no se explica en esta sección. Nos remitimos al apartado de "declaración de funciones" donde se explica su uso.

Asignaciones

Las instrucciones de asignación en cES sirven para asignar un valor a una variable o un elemento de un vector. Para ello se escribe primero una referencia a alguno de estos elementos seguido del operador de asignación (=) y a su derecha una expresión. A diferencia de lo que ocurre en C, en cES el operador de asignación no es asociativo. Es decir, no es posible escribir construcciones sintácticas del estilo a = b = c. La sintaxis es:

```
ref = expresion;
```

Donde `ref` es una referencia a una variable o elemento de un vector y expresión es una expresión del mismo tipo que la referencia (recordar que la comprobación de tipos la realiza el analizador semántico). No es posible hacer asignaciones a vectores de forma directa, de modo que son errores de compilación (de ello se encarga el analizador semántico) sentencias como `vector1 = vector2;`

El siguiente listado muestra algunos ejemplos de uso de sentencias de asignación:

Listado 16. Ejemplos de uso de la sentencia de asignación

```
i = 3 + 7;

v[i] = 10;

igual = 3==4;

a = 2 * suma(2,2);
```

Asignación con suma

En cES se contempla un tipo de operación de asignación especial: la asignación con suma (+=). Su expresión sintáctica es:

```
ref += expresion
```

Donde `ref` representa una referencia a una variable o elemento de un vector y `expresión` es una expresión. La interpretación semántica respectiva que cES hace es equivalente a la que se muestra a continuación:

```
ref = ref + expresion
```

Es decir, al ejecutarse se asignará a la referencia ref el resultado de sumar el valor de *expresion* al valor previo de ref. A continuación se muestran unos ejemplos de uso

Listado 17. Ejemplos de uso de las sentencias especiales de asignación

```
entero a = 5;

a +=2;  /* El valor de a es 5+2 = 7 */
```

Sentencia de control de flujo condicional SI – SINO

La sentencia `si – sino` 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 dos formas:

```
si (expresionLogica)

      /* sentencias1 */
```

```
si (expresionLogica)
      /* sentencias1 */
sino
      /* sentencias2 */
```

Donde `expresionLogica` es una expresión lógica (es decir aritmética en un contexto lógico) que se evalúa para comprobar si deben ejecutarse las sentencias o bloques de sentencias `sentencias1` o `sentencias2`. En concreto si el resultado de dicha expresión es cierta, es decir, es mayor que 0, se ejecuta el bloque `sentencias1`. Si existe `sino` y la condición es falsa (`0`), se ejecuta el bloque `sentencias2`. Es necesario que `expresionLogica` esté siempre entre paréntesis.

El siguiente listado ilustra un ejemplo de sentencia si.

Listado 19. Ejemplo de sentencia si – entonces – sino

```
si (a==b)
   escribe ("iguales");
si (a==0){
   escribe ("es");
   escribe ("falso");
} sino {
   escribe ("es");
   escribe ("verdadero");
}
```

Este tipo de construcciones pueden anidarse con otras construcciones de tipo `si` o con otros tipos de sentencias de control de flujo que estudiaremos a continuación.

Sentencia de control de flujo condicional ALTERNATIVAS

La sentencia `alternativas` corresponde con la sentencia switch del lenguaje C, correspondiendo en cES a la siguiente sintaxis:

```
alternativas (expresion) {
      caso constante1:  /* sentencias 1*/  corte;
      caso constante2:  /* sentencias 2*/  corte;
      pordefecto: /* sentencias*/ corte;
}
```

Primero se evalúa expresion, que corresponde con una expresión aritmética (obligatorios los paréntesis). El resultado se compara con las constantes numéricas asociadas a cada caso y si corresponde con alguna de ellas se ejecutarán el bloque de sentencias escritas a continuación de los dos puntos de el caso. Se terminará al encontrar la palabra reservada `corte`. Si un caso no contiene sentencia de corte se considerará un error sintáctico.

Si el valor devuelto por la expresión no corresponde a ningún caso se ejecutarán las sentencias correspondientes a la etiqueta *pordefecto*. No es obligatorio que aparezca esta etiqueta. Es decir, en caso de que no exista la etiqueta *pordefecto* no se generará ningún error. En el siguiente listado se muestra un ejemplo de uso.

Listado 20. Ejemplo de sentencia alternativas

```
alternativas (dia){
  caso 1: {escribe("lunes");} corte;
  caso 2: {
    escribe("es ");
    escribe("martes");
  } corte;
  pordefecto: {escribe ("no es lunes ni martes");} corte;
}
```

Sentencia de control de flujo iterativo MIENTRAS

La sentencia `mientras` se utiliza para realizar iteraciones sobre una sentencia o 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 `mientras` mientras se cumpla una determinada condición, determinada por una expresión. Su estructura es:

```
mientras (expresionLogica)
    /* sentencias */
```

Donde `expresionLogica` es la expresión lógica a evaluar en cada iteración y `sentencias` es una sentencia o 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.

Listado 21. Ejemplos de sentencia mientras

```
mientras (a<5){
   a=a+1;
   escribeEnt(a);
}
mientras (a<5)
   a=a+1;
```

Sentencias de llamada a una función vacio

Como ya se discutió con anterioridad, en cES es posible declarar funciones que devuelvan el tipo vacío 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 vacio no puede ubicarse donde se espera una expresión sino solamente donde se espera una sentencia. La comprobación sobre si esa función devuelve un tipo vacío o no la realiza el analizador semántico.

Sentencias de salida

El lenguaje cES dispone de una serie de funciones predefinidas que pueden ser utilizados para emitir por la salida estándar (pantalla) resultados de diferentes tipos. 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 no pueden declararse identificadores con dichos nombres. En concreto disponemos de 3 funciones. A continuación detallamos cada una de ellas.

- `escribe()`. Este procedimiento toma una constante literal de tipo cadena de texto y la muestra por la salida estándar. El ejemplo `escribe("Hola mundo");` mostrará el texto `Hola mundo`.

- `escribeEnt()`. Este procedimiento recibe una expresión de tipo entero y muestra su resultado por la salida estándar. Por ejemplo, `escribeEnt(12);` `escribe(a);`

Tanto *escribe* como *escribeEnt* pueden no recibir parámetros, en ese caso no mostrarían nada. Se incide en que ambas funciones realizan un salto de línea al mostrar por pantalla su resultado, tanto si reciben parámetros como si no.

A efectos prácticos estas sentencias han de considerarse como funciones vacio, es decir, no pueden ser usadas como expresiones.

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, sintáctico o semántico. Por lo demás, se valorarán intentos de aportar más información sobre la naturaleza del error semántico. Por ejemplo, los errores motivados por la comprobación explícita de tipos de acuerdo al sistema de tipos del lenguaje constituyen errores de carácter semántico. Otros ejemplos de errores semánticos son: identificador duplicado, tipo erróneo de variable, variable no declarada, función no definida, demasiados parámetros en llamada a función, etc.

3 Descripción del trabajo

En esta sección se describe el trabajo que se ha de realizar. La práctica es un trabajo amplio que exige tiempo y dedicación. De cara a cumplir los plazos de entrega, recomendamos avanzar constantemente, junto con la teoría, 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 si, 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.

La práctica ha de desarrollarse en **Java** (se recomienda utilizar la última versión disponible)**.** Para su realización se usarán las herramientas JFlex y Cup 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.

Además, se proporciona una implementación del analizador léxico (fichero scanner.flex) y del analizador sintáctico (fichero parser.cup) de la que se puede partir. No es obligatorio usar estas implementaciones. De hecho, la implementación dada del análisis léxico y sintáctico se puede modificar para adaptarla al desarrollo de la práctica que realice cada estudiante.

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 estudiante 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.

3.1 Trabajo a entregar

El trabajo a realizar contemplará el desarrollo de las siguientes etapas: análisis semántico, generación de código intermedio y código final. En ningún caso es necesario realizar optimización del código intermedio ni del código final generado.

Es **importante tener claro que la entrega debe de contener todo el proceso de compilación**. Es decir, han de incluirse las siguientes fases: análisis léxico, análisis sintáctico, análisis semántico, generación de código intermedio y código final. Para las dos primeras etapas (análisis léxico y sintáctico) se puede usar la implementación proporcionada, o se puede realizar una implementación de las mismas.

De cara a superar la asignatura se distingue la realización de una parte obligatoria y de otra opcional:

- Parte **obligatoria**: se tiene que implementar completamente el analizador semántico (incluidas las comprobaciones relativas al uso de subprogramas). Además, se tiene que implementar la generación de código intermedio y final de todo el lenguaje a excepción de lo relativo a las funciones. Si se encuentra una función, no se debe generar código asociado a ella, pero sí el resto del código. En cualquier caso, no será válido generar una excepción al encontrar funciones. La nota máxima que se podrá obtener por esta parte es de 7, en función de las decisiones tomadas y los errores detectados.
- Parte **opcional**: a la parte obligatoria se tiene que añadir la generación de código intermedio y final relativo a funciones, incluido el soporte para la recursión. En función de las decisiones tomadas y los errores detectados, la realización de esta parte puede suponer obtener una nota máxima de 10.

En las siguientes subsecciones se dan más detalles sobre el trabajo a realizar.

3.1.1 Análisis semántico

En la fase de análisis semántico se debe asignar un significado a cada construcción sintáctica del código fuente. Esto se consigue en primer lugar gestionando los elementos declarados por el programador tales como constantes, tipos, variables y funciones. Para llevar a cabo esta labor es preciso hacer uso de una colección de estructuras entre las que destaca la tabla de símbolos por su especial relevancia. La *tabla de símbolos* (TS) es una estructura disponible en tiempo de compilación que almacena información sobre los nombres definidos por el usuario en el programa fuente, llamados símbolos en este contexto teórico. Dependiendo del tipo de símbolo encontrado por el compilador durante el procesamiento del código fuente (constante, variable, función), los datos que deben ser almacenados por la TS serán diferentes. Por ejemplo:

- Para las constantes: nombre, tipo, valor…
- Para las variables: tipo, ámbito, tamaño en memoria ...
- Para las funciones: parámetros y sus tipos, tipo de retorno …

Otra estructura importante es la *tabla de tipos* (TT), cuya responsabilidad es mantener una definición computacional de todos los tipos (primitivos y compuestos) que están accesibles por el programador dentro de un programa fuente. Las entradas de la TS mantienen referencias a esta tabla para tipificar sus elementos (variables, constantes y funciones).

El trabajo de la fase de análisis semántico consiste, básicamente, en implementar, dentro de las acciones java que Cup permite insertar entre los elementos de la parte derecha de la gramática, el sistema de tipos para el lenguaje. Esto requiere añadir en la TT una entrada por cada tipo primitivo, tipo compuesto y función definida; añadir una entrada en la TS por cada símbolo declarado (variables, constantes y funciones) y finalmente comprobar en las expresiones el uso de elementos de tipos compatibles entre si. Asimismo, debe comprobarse que no existen duplicidades entre símbolos que pertenezcan al mismo ámbito o nombres de tipos. Cualquiera de estas violaciones o error de tipos constituye un error semántico que debe comunicarse al usuario.

De cara a superar la práctica, es obligatorio implementar completamente el analizador semántico.

3.1.2 Generación de código intermedio

Esta fase traduce la descripción de un programa fuente expresado en un árbol de análisis sintáctico (AST) decorado en una secuencia ordenada de instrucciones en un código cercano al ensamblador, pero aún no comprometido con ninguna arquitectura física. En este sentido, las instrucciones de código intermedio son CUÁDRUPLAS de datos formadas, a lo sumo, por: 1) un operador de operación, 2) dos operandos y 3) un operando de resultado. En esta fase, los operandos son referencias simbólicas a los elementos del programa (entradas de la TS), nombres de etiquetas o variables temporales que referencian “lugares” donde se almacenan resultados de cómputo intermedio. EN NINGUN CASO estos operandos son direcciones físicas de memoria. El trabajo de esta fase consiste básicamente en insertar en las acciones semánticas de Cup las instrucciones java pertinentes para realizar la traducción de un AST a una secuencia de CUADRUPLAS. Al final de esta etapa ya no necesitaremos más herramientas, ni tampoco el programa fuente: tendremos una TS llena y una lista de cuádruplas que describen todo el contenido del programa fuente.

De cara a superar la práctica es **obligatorio** implementar la generación de código intermedio de todo el lenguaje a excepción de lo relativo a las funciones. En cualquier caso, no será válido generar una excepción aunque se encuentre una función y no se realice la parte opcional.

Como parte **opcional**, el alumno puede implementar la generación de código intermedio relativa a las funciones.

3.1.3 Generación de código final

Para generar código final se parte del código intermedio generado y de la TS. Esta etapa es la única que no coincide en el tiempo con las anteriores, ya que se realiza posteriormente a realizar el proceso de compilación. Su objetivo es el de convertir la secuencia de CUADRUPLAS generada en la fase anterior en una secuencia de instrucciones para una arquitectura real (en nuestro caso ENS2001). Este proceso requiere convertir cada operador en su equivalente en

ENS2001 y en convertir las referencias simbólicas de los operandos en direcciones físicas reales. Nótese que en función del ámbito de las mismas (variables globales, locales, parámetros, temporales, etc.) el modo de direccionamiento puede ser diferente.

Una vez obtenido el código final, éste puede probarse utilizando la herramienta ENS2001 que permite interpretar el código generado. Así podremos ejecutar un programa compilado con nuestro compilador y probar su funcionamiento.

De cara a superar la práctica es **obligatorio** implementar la generación de código final de todo el lenguaje a excepción de lo relativo a las funciones. El código final generado debe de funcionar correctamente en la herramienta ENS2001. En cualquier caso, no será válido generar un excepción aunque se encuentre una función y no se realice la parte opcional.

Como parte **opcional**, el alumno puede implementar la generación de código final relativa a las funciones.

3.1.4 Comportamiento esperado del compilador

El compilador debe procesar archivos fuente y generar archivos ensamblador (ENS2001). Si los programas fuente incluyen errores (léxicos, sintácticos o semánticos), el compilador debe indicarlos por pantalla; en ese caso no se generará ningún código. Los programas en ensamblador generados por el compilador deben ejecutarse correctamente con la configuración predeterminada de ENS2001 (crecimiento de la pila descendente). Esta ejecución es manual. Es decir, el compilador del alumno no debe ejecutar la aplicación ENS2001, sino únicamente generar el código ensamblador. Posteriormente, el usuario podría arrancar ENS2001 y cargar el archivo ensamblador generado, ejecutándolo y comprobando su correcto funcionamiento. En todo caso, se recomienda comprobar que los archivos ensamblador generados se comportan correctamente.

IMPORTANTE: la práctica tiene que escribir la salida en un fichero con el mismo nombre (y en la misma ruta) que la entrada y extensión ens, como hace por defecto la arquitectura. No se debe modificar este comportamiento.

3.1.5 Dificultades

Es muy importante dedicar tiempo a entender el proceso completo de compilación y ejecución, especialmente a la hora de distinguir dos conceptos fundamentales: tiempo de compilación y tiempo de ejecución. La diferencia es que en tiempo de compilación tenemos a nuestra disposición la tabla de símbolos que nos dice, por ejemplo, en qué dirección de memoria está una determinada variable. Sin embargo, al ejecutar el programa ya no tenemos ningún apoyo más que el propio código generado, de forma que el código debe contener toda la información necesaria para que el programa funcione correctamente. Esto resulta especialmente complicado a la hora de manejar llamadas a funciones, especialmente si son llamadas recursivas. Para mantener esta información en tiempo de ejecución se reserva un espacio en memoria llamado *registro de activación*, que almacena elementos como la dirección de retorno, el valor devuelto, los parámetros y las variables locales.

El registro de activación y la gestión de memoria son probablemente los puntos más delicados y difíciles para el alumno, por lo que recomendamos abordarlos con cuidado y apoyarse en la teoría.

Por tanto, se recomienda ir realizando la práctica de forma conjunta al estudio de la teoría para tener un máximo aprovechamiento de la asignatura y reducir el esfuerzo global.

3.2 Fechas y forma de entrega

Las fechas límite para las diferentes entregas son las siguientes:

Febrero	11 de junio de 2023
Septiembre	10 de septiembre de 2023

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. **IMPORTANTE**: no es válido poner un enlace de descarga como entrega. En estos casos, no se corregirá la práctica ya que se tiene que subir un fichero comprimido.

Puesto que la compilación y ejecución de las prácticas de los alumnos se puede tener que realizar de forma automatizada, *el alumno debe respetar las normas de entrega indicadas en el enunciado de la práctica*. Recordamos que la práctica tiene que escribir la salida en un fichero con el mismo nombre que la entrada y extensión ens, como hace por defecto la arquitectura. No se debe modificar este comportamiento.

Se recuerda que es necesario superar una **sesión de control obligatoria** a lo largo del curso para aprobar la práctica y la asignatura. En la sesión presencial **obligatoria** el tutor comprobará que el alumno ha realizado:

- Análisis semántico
    - desarrollo tabla de tipos
    - desarrollo tabla de símbolos
    - comprobación de la unicidad de declaraciones y definiciones
    - comprobación de tipos
    - comprobación de paso de parámetros
- Código intermedio
    - se ha comenzado esta fase

El Equipo Docente requerirá un informe adicional al tutor cuando durante la corrección se detecte que una práctica que ha recibido el APTO en la sesión presencial no cumple los requisitos exigidos para superar la sesión presencial.

Nos remitimos al documento de normas de la asignatura dónde vienen explicada la normativa a aplicar.

3.3 Formato de entrega

El material a entregar, mediante el curso virtual, consiste en un único archivo comprimido en formato **zip**. Recordamos que no se admitirán entregas que consistan en un enlace de descarga.

Dicho archivo contendrá la estructura de directorios que se proporcionará en las directrices de implementación. Esta estructura debe estar en la raíz del fichero zip. **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 pdf y situado en el directorio correspondiente de la estructura dada.

El índice de la memoria será:

Portada obligatoria

1. El analizador semántico y la comprobación de tipos

    1.1. Descripción del manejo de la Tabla de Símbolos y de la Tabla de Tipos (Describir cómo se trabaja con ambas tablas y las comprobaciones realizadas)

2. Generación de código intermedio

    2.1. Descripción de la estructura utilizada

3. Generación de código final

    3.1. Descripción de hasta dónde se ha llegado

    3.2. Descripción del registro de activación implementado (en caso de realizarse)

4. Indicaciones especiales

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. En caso de que la memoria no concuerde con las decisiones tomadas en la implementación de cada alumno la práctica puede ser considerada suspensa.

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.

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 cómo 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** 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 de** la asignatura, incluido en el Curso Virtual.

Se recomienda también la utilización de los foros como medio de comunicación entre alumnos y de éstos 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**. La práctica se realiza de forma individual. Se utilizarán programas de detección de copias en el código fuente y, en caso de ser detectada, se suspenderá a los alumnos implicados en todas las convocatorias del presente curso.

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).