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

Práctica de Procesadores del Lenguaje II

Especificación del lenguaje ModulUNED v1.0

Dpto. de Lenguajes y Sistemas Informáticos
ETSI Informática, UNED

Alvaro Rodrigo
Anselmo Peñas (coordinador)
Laura Plaza

Curso 2019 - 2020

ÍNDICE

1 INTRODUCCIÓN

En este documento se define la práctica de la asignatura Procesadores del Lenguaje II correspondiente al curso 2019-2020. El objetivo de la práctica es realizar un compilador del lenguaje modulUNED, con el que se trabajó en la práctica de la asignatura Procesadores del Lenguaje I del curso 2017-2018.

Primero se presenta una descripción del lenguaje 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 modulUNED, 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 ModulUNED, una versión convenientemente reducida del lenguaje de programación Modula-2. 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 escrito en ModulUNED 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 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 ModulUNED 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 *)

(* Este 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 ModulUNED 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 literales de 3 tipos:

- **Lógicas.** Las constantes lógicas representan valores de verdad (cierto o falso) que son utilizadas dentro de expresiones lógicas como se verá más adelante. Únicamente existen `2` que quedan representadas por las palabras reservadas `TRUE` y `FALSE` e indican el valor cierto y falso respectivamente.

- **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 **– 2** debería escribirse `0 - 2`.

- **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 `WRITESTRING` (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 tales como /t /n etc.

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 variables o los subprogramas definidos por el programador. El lenguaje NO 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 iguales. Por ejemplo, `Abc` y `ABC` representan el mismo identificador. 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	DESCRIPCIÓN
AND	Y lógica.
ARRAY	Declaración de una estructura tipo vector.
BEGIN	Delimitador de comienzo de bloque de sentencias.
BOOLEAN	Tipo lógico.
CONST	Declaración de constantes.
DO	Comienzo del cuerpo de un bucle WHILE o FOR.
ELSE	Comienzo del cuerpo de alternativa de una condicional IF.
END	Delimitador de final de bloque de sentencias.
FALSE	Constante lógica que representa falso.

FOR	Comienzo de un bucle FOR.
IF	Comienzo de un condicional IF.
INTEGER	Tipo entero.
MODULE	Comienzo de programa.
NOT	Negación lógica.
OF	Asignación del tipo base en ARRAY.
OR	O lógica.
PROCEDURE	Comienzo de un subprograma.
RECORD	Declaración de una estructura tipo registro.
RETURN	Retorno de una función.
THEN	Comienzo del cuerpo de una condicional IF.
TO	Asignación de rango en un bucle FOR.
TRUE	Constante lógica que representa un valor verdadero.
TYPE	Comienzo de la declaración de tipos.
VAR	Comienzo de la declaración de variables.
WHILE	Comienzo de un bucle iterativo WHILE.
WRITESTRING	Procedimiento predefinido que muestra una cadena de caracteres.
WRITEINT	Procedimiento predefinido que muestra un entero.
WRITELN	Procedimiento predefinido que muestra un salto de línea.

2.1.5 Delimitadores

ModulUNED define una colección de delimitadores que utiliza 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 ARRAY.
(* *)	Delimitadores de comentario
,	Delimitador de identificadores
;	Delimitador de sentencias.
:	Delimitador de tipo en una declaración de variable.

2.1.6 Operadores

En ModulUNED 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)
	− (resta)
	* (producto)
	/ (división entera)
Operadores relacionales	< (menor)
	> (mayor)
	= (igual)
	<> (distinto)

Operadores lógicos	`AND` (conjunción lógica) `OR` (disyunción lógica) `NOT` (negación lógica)
Operador de asignación	`:=`
Operadores de acceso	`.` (acceso a campo de registro) `[]` (acceso a elemento de vector)

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

2.2 Aspectos Sintácticos

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

2.2.1 Estructura y ámbitos de un programa

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

Listado 2. Estructura general de un programa en ModulUNED

```
MODULE nombrePrograma;

      (*Declaración constantes*)

      (*Declaración tipos*)

      (*Declaración variables*)

      (*Declaración subprogramas*)

BEGIN

      (*Lista sentencias*)


END nombrePrograma;
```

En ModulUNED un programa consiste en una secuencia ordenada de construcciones sintácticas que empiezan por la declaración del programa con la instrucción `MODULE` seguida del nombre del módulo y PUNTO Y COMA (;). A continuación se declaran opcionalmente, pero necesariamente siguiendo este orden, constantes, tipos, variables y subprogramas. Posteriormente se encuentra una secuencia ordenada de sentencias (secuencia que puede ser vacía) encerradas entre los delimitadores `BEGIN` y `END` seguidos del nombre del módulo y terminado en PUNTO Y COMA (;). Además, se pueden insertar comentarios en cualquier punto del programa. En sucesivas secciones se describirá en detalle la estructura sintáctica de cada uno de estos bloques.

Cada subprograma constituye un bloque que determina un ámbito 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 del mismo. En este sentido, en ModulUNED distinguimos 3 posibilidades:

- **Referencias globales**. Estos identificadores globales son los que se declaran directamente dentro del módulo del programa. 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 procedimiento 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 subprogramas.

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.

2.2.2 Declaración de constantes simbólicas

Las constantes simbólicas, como 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. La sintaxis para la declaración de constantes simbólicas es la siguiente:

```
CONST NOMBRE = valor;
```

Donde `NOMBRE` es el nombre simbólico que recibe la constante definida dentro del programa y `valor` su valor constante de tipo entero (número) o lógico (TRUE o FALSE), de manera que cada vez que la constante aparece referenciada dentro del código se sustituye por el valor establecido en la declaración. A continuación se muestran algunos ejemplos de declaración de constantes. Como puede apreciarse en el listado 3, **la palabra reservada CONST sólo puede aparecer una vez dentro de cada ámbito**.

Listado 3. Ejemplo de declaración de constantes en ModulUNED

```
CONST TAMANO = 10;

      MESES = 12;

      ANYO = 2007;

      ABIERTO = TRUE;
```

2.2.3 Declaraciones de Tipos

ModulUNED articula dos mecanismos para trabajar con tipos: los tipos primitivos del lenguaje y los constructores para declarar tipos compuestos definidos por el usuario. En las dos siguientes subsecciones describimos 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 dentro de ModulUNED se establecen 2 tipos predefinidos:

Tipo entero

El tipo entero representa valores enteros positivos y negativos. Por tanto, a las variables declaradas de tipo entero se les puede asignar el resultado de evaluar una expresión aritmética, ya que dicho resultado puede tomar valores positivos y negativos. El tipo entero se representa con la palabra reservada `INTEGER`.

Tipo lógico

El tipo lógico representa valores de verdad (verdadero o falso). Estos valores están representados por las constantes literales `TRUE` y `FALSE`. Para referirse en ModulUNED a este tipo de datos se utiliza la palabra reservada `BOOLEAN`.

2.2.3.2 Tipos Compuestos

Los tipos compuestos (también llamados tipos definidos por el usuario) permiten al programador definir estructuras de datos compuestas y establecerlas como un tipo más del lenguaje. Estas estructuras reciben un nombre (identificador) que sirve para referenciarlas posteriormente en la declaración de variables de ese tipo. Así en ModulUNED no existen tipos anónimos. Además, 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) en la sección de declaración de tipos 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. Esto debe controlarlo el Analizador Semántico.

Para declarar tipos en ModulUNED se utiliza la palabra clave TYPE seguida de una secuencia de una o más declaraciones de tipos separadas por un punto y coma (;). Cada declaración de tipos consiste en un identificar de tipo seguido de un igual (=), seguido de una declaración de un tipo. La palabra reservada TYPE solamente puede aparecen una vez dentro de cada ámbito.

La sintaxis para la declaración de un tipo depende de la clase de tipo de que se trate. En concreto se distinguen 2 tipos de estructuras compuestas en ModulUNED: los vectores y los registros. La definición de cada uno de estos tipos 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 registros, registros cuyos campos son vectores, vectores de vectores (vectores multidimensionales) ni registros que tienen a otros registros como campos. A continuación describimos en detalle cada uno de estos tipos.

Tipo vector

Un vector es una estructura de datos que puede almacenar varios valores de un mismo tipo primitivo. Para definirlo se utiliza la palabra clave `ARRAY` seguida de un rango numérico encerrado entre los delimitadores de rango donde se indican el índice mínimo y máximo del vector, a continuación la palabra reservada `OF` y, finalmente, un tipo primitivo. Esta definición debe igualarse a un nombre, que será el nombre del tipo definido. La sintaxis tiene la siguiente forma:

```
nombreTipo = ARRAY [n1..n2] OF TipoPrimitivo;
```

Donde `nombreTipo` será el nombre del tipo, `n1` y `n2` dos constantes numéricas (literales o simbólicas) que determinan el rango de posiciones disponibles en el vector, y `TipoPrimitivo` indica el tipo de sus elementos. El Analizador Semántico comprobará que `n2` es mayor o igual que `n1`. En el listado 4 se muestran ejemplos de declaración de vectores.

Listado 4. Ejemplo de declaración de vectores en ModulUNED

```
TYPE TipoVectorEnteros = ARRAY [1..3] OF INTEGER;

     TipoVectorBooleanos = ARRAY [1..3] OF BOOLEAN;
```

Tipo registro

Un registro es una estructura de datos compuesta que permite agrupar elementos de diferentes tipos. Para definir un registro se utiliza la palabra reservada RECORD seguida de una lista de declaraciones de campos y termina con la palabra reservada END. Cada declaración de campos consiste en un nombre (identificador) seguido del delimitador de tipo seguido de un tipo primitivo. Concretamente la declaración de una estructura sigue la siguiente sintaxis:

```
nombreTipo = RECORD
                campo1 : TipoPrimitivo;
                campo2 : TipoPrimitivo;
```

```
                    ...
                  campoN : TipoPrimitivo;
                END;
```

Donde `nombreTipo` será el nombre del tipo, y `campo1`, `campo2`, … `campoN`, representan los nombres que reciben los distintos campos del registro. El listado 5 muestra un ejemplo de declaración de registros.

Listado 5. Ejemplo de declaración de registros en ModulUNED

```
TYPE TipoPersona = RECORD
                    edad   : INTEGER;
                    dni    : INTEGER;
                    casado : BOOLEAN;
                   END;
```

2.2.4 Declaraciones de Variables

En ModulUNED el uso de variables requiere previamente de la declaración de las mismas para asignarles un tipo. Desde el punto de vista sintáctico, la sección de declaración de variables contiene una lista de declaraciones de variables que comienza por la palabra reservada `VAR` (dentro de cada ámbito sólo puede existir una cláusula `VAR`). Por su parte, una declaración de variables consiste en una lista de identificadores separadas por comas (opcionalmente solo un identificador) seguido del delimitador de tipo `:` y de un nombre de tipo (primitivo o compuesto) y acabado por el delimitador `;` . Esto es:

```
VAR nombre11, nombre 12, ..., nombre1N : Tipo1;
    nombre21, nombre 22, ..., nombre2N : Tipo2;
      ...
    nombreM1, nombre M2, ..., nombreMN : TipoM;
```

Donde `nombreIJ` es un identificador de variable y Tipo1, Tipo2, … TipoM son nombres de tipos primitivos o identificadores de tipos compuestos. El Analizador Semántico comprobará que dentro de un mismo ámbito una variable solo se declara una vez. El listado 6 muestra algunos ejemplos de declaraciones de variables.

Listado 6. Ejemplo de declaración de variables en ModulUNED

```
VAR x, y : INTEGER;
    abierto : BOOLEAN;
    v1, v2 : TipoVector;
    c1, c2 : TipoComplejo;
    v : TipoVectorEnteros;
```

```
b : TipoVectorBooleanos;

juan, Maria : TipoPersona;
```

2.2.5 Declaración de subprogramas

En ModulUNED se pueden declarar subprogramas para organizar modularmente el código. Un subprograma es una secuencia de instrucciones encapsuladas bajo un nombre y opcionalmente declarada con unos parámetros. En concreto existen 2 tipos de subprogramas: procedimientos y funciones. A continuación describimos cada uno de ellos.

2.2.5.1 Procedimientos

Los procedimientos son rutinas encapsuladas bajo un nombre que realizan una determinada operación para que el programador las invoque, convenientemente parametrizadas, desde distintos puntos del programa. La declaración de un procedimiento se realiza en ModulUNED con la cabecera:

```
PROCEDURE nombre (param11, param12,..., param1N : Tipo1;
                  param21, param22,..., param2N : Tipo2;
                     ...
                  paramM1, paramM2,..., paramMN : TipoM);
```

Donde `nombre` es el nombre del procedimiento, `paramIJ` el nombre de cada uno de los parámetros y `tipo1, tipo2,... TipoM` los tipos de dichos parámetros. Como puede verse, la declaración de los parámetros formales es una lista de declaraciones de parámetros separadas por un delimitador de punto y coma. Cada declaración de parámetros tiene la forma de una lista de identificadores separados por comas, seguido del delimitador de tipo, dos puntos (:) y seguido de un nombre de tipo primitivo o un identificador de tipo compuesto.

La declaración de parámetros (incluidos los paréntesis) es opcional. Por tanto, la cabecera de un procedimiento sin parámetros tiene dos estructura sintácticas equivalentes:

```
PROCEDURE nombre;

PROCEDURE nombre ();
```

Por su parte, la cabecera es seguida de un cuerpo con una estructura idéntica a la del módulo (sección `CONST`, `TYPE`, `VAR`, `PROCEDURE`, y sentencias encerradas entre los delimitadores `BEGIN` y `END identificador;`). En el listado 7 presentamos un ejemplo de procedimiento.

Listado 7. Ejemplo de declaración de procedimiento en ModulUNED

```
CONST MAX = 10;

      MIN = 1;

TYPE TipoVector = ARRAY [MIN..MAX] OF INTEGER;
```

```
PROCEDURE invertir (v : TipoVector);

VAR aux, index: INTEGER;

BEGIN

      FOR index := MIN TO MAX DO

            aux := v[index];

            v[index] := v[MAX – index + MIN] ;

            v[MAX – index + MIN] := aux;

      END;

END invertir;
```

2.2.5.2 Funciones

Las funciones son rutinas encapsuladas bajo un nombre que realizan un determinado cómputo y cuyo resultado devuelven al contexto de invocación. Desde el punto de vista sintáctico, una función en ModulUNED solo se diferencia de un procedimiento en dos aspectos: 1) después de la declaración de los parámetros le sigue un delimitador de tipo, dos puntos `(:)`, seguido de un tipo primitivo (las funciones NO pueden devolver estructuras de datos como resultado de su invocación) y 2) dentro del cuerpo de una función debe aparecer al menos una vez el uso de la palabra reservada `RETURN` seguido de una expresión y un delimitador de punto y coma para indicar al compilador el valor que deberá emitir como resultado de su invocación (esta comprobación se realiza dentro del análisis semántico). La sintaxis de la cabecera en la declaración de una función es la siguiente (obsérvese que comienza con la palabra reservada PROCEDURE al igual que los procedimientos):

```
PROCEDURE nombre (param11, param12,..., param1N : Tipo1;
                  param21, param22,..., param2N : Tipo2;
                     ...
                  paramM1, paramM2,..., paramMN : TipoM): Tipo;
```

Donde `nombre` es el nombre de la función, `paramIJ` el nombre de cada uno de los parámetros, `tipo1, tipo2,... TipoM` los tipos de dichos parámetros y `Tipo`, el tipo de retorno. Como en el caso de los procedimientos, el uso de parámetros en una función es opcional. La cabecera de una función sin parámetros tiene la siguiente estructura sintáctica en su declaración:

```
PROCEDURE nombre : Tipo;

PROCEDURE nombre () : Tipo;
```

En el listado 8 presentamos un ejemplo de función que suma los elementos de un vector.

Listado 8. Ejemplo de declaración de una función en ModulUNED

```
TYPE TipoVector = ARRAY [1..5] OF INTEGER;

PROCEDURE sumaVector ( VAR v : TipoVector): INTEGER;

VAR suma, index: INTEGER;

BEGIN

      suma := 0;

      FOR index := 1 TO 5 DO

         suma := suma + v[index];

      END;

      RETURN suma;

END sumaVector;
```

2.2.5.3 PASO DE PARÁMETROS A SUBPROGRAMAS

Como se ha comentado anteriormente, tanto los procedimientos como las funciones pueden recibir **parámetros** en cada invocación. En ModulUNED es posible llamar a subprogramas pasando expresiones, variables, constantes simbólicas, elementos de vectores, campos de estructuras, arrays completos y registros completos como parámetros a una función o procedimiento.

El Analizador Semántico comprobará que se pasa el número adecuado de parámetros y del tipo correcto.

El paso de parámetros actuales a una función o procedimiento puede llevarse a cabo de dos formas diferentes:

- **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. Este modelo de paso de parámetros se usa para pasar argumentos de entrada a un subprograma. Los parámetros actuales pasados por valor pueden ser expresiones, variables, elementos de vectores y campos de registros pero **no** vectores o registros completos.

- **Paso por referencia**. En este caso el compilador transmite al subprograma la dirección de memoria donde está almacenado el parámetro actual, de forma que las modificaciones que se hagan dentro del subprograma tendrán efecto sobre el argumento una vez terminada la ejecución del mismo. El paso de parámetros por referencia es utilizado para pasar al subprograma en la invocación argumentos de salida o de entrada / salida. En ModulUNED el paso de vectores y registros completos como parámetros a un subprograma se realiza siempre por referencia. Para indicar que una variable, elemento de vector o campo de registro se pasa por referencia se utiliza la palabra reservada `VAR`.

En el listado 9 se incluyen ejemplos de declaraciones de funciones y procedimientos que utilizan paso por valor y por referencia:

Listado 9 Ejemplo de pasos de parámetros por valor y por referencia

```
CONST MAX = 10;
      MIN = 1;
TYPE TipoVector = ARRAY [MIN..MAX] OF INTEGER;
      TipoPunto = RECORD
                        x : INTEGER;
                        y : INTEGER;
                  END;
PROCEDURE suma (a, b : INTEGER) : INTEGER; (* por valor *)
BEGIN
    RETURN a + b;
END suma;
PROCEDURE cambia (VAR a, b : INTEGER); (* por referencia *)
VAR aux : INTEGER;
BEGIN
    aux := a;
    a := b;
    b := aux;
END cambia;
PROCEDURE sumaVector (VAR v : TipoVector): INTEGER; (*
referencia *)
VAR suma, index: INTEGER;
BEGIN
      suma := 0;
      FOR index := MIN TO MAX DO
         suma := suma + v[index];
      END;
      RETURN suma;
END sumaVector;
```

```
PROCEDURE asigna (VAR p1, p2 : TipoPunto); (* por referencia *)
BEGIN
    p1.x := p2.x;
    p1.y := p2.y;
END asigna;
```

En ModulUNED se admite el **anidamiento** de subprogramas. Es decir, pueden declararse subprogramas locales a un subprograma dado. Para definir un subprograma dentro de otro ha de hacerse después de declarar las variables y antes del BEGIN.

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 del programa donde aparece la expresión. En ModulUNED existen los siguientes tipos de expresiones:

Expresiones aritméticas

Las expresiones aritméticas son aquellas cuyo cómputo devuelve un valor de tipo entero al programa. Sintácticamente 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, resta, producto, resto o división entera de dos expresiones aritméticas. Cuando se trabaja con expresiones aritméticas 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. Los operadores en la misma fila tienen igual precedencia).

Precedencia	Asociatividad
. ( ) [ ]	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 un expresión aritmética. En el listado 10 se exponen algunos ejemplos sobre precedencia y asociatividad y cómo la parentización puede alterar la misma.

Listado 10. Ejemplo de precedencia y asociatividad en expresiones aritméticas

```
3 * 4 + 2  (* 14 ya que * > + *)

3 * (4 + 2) (* 18 ya que los paréntesis alteran la evaluación *)

3 + 2 + 1  (* asociatividad a izquierdas *)

3 + (2 + 1) (* asociatividad alterada por paréntesis *)
```

Expresiones lógicas

Las expresiones lógicas son aquellas cuyo computo devuelve un valor de tipo lógico al programa. Sintácticamente puede afirmarse que son expresiones lógicas las constantes literales de tipo lógico (valores de verdad cierto y falso), las constante simbólicas de tipo lógico, los identificadores (variables o parámetros) de tipo lógico y las funciones que devuelven un valor de tipo lógico. Asimismo también son expresiones lógicas la negación de una expresión lógica y la conjunción y disyunción de dos expresiones lógicas. ModulUNED incluye una serie de operadores relacionales que permite comparar expresiones aritméticas entre si. El resultado de esa comparación es una expresión lógica. Es decir, la comparación con los operadores `>`, `<`, `=` o `<>` de dos expresiones aritméticas es también una expresión lógica.

Igual que antes los operadores lógicos (`AND`, `OR` y `NOT`) y los relacionales (`>`, `<`, `=` y `<>`) definen una relación de precedencia para determinar el orden de evaluación. La siguiente tabla resume la relación de precedencia y asociatividad y operadores (la prioridad decrece según se avanza en la tabla. Los operadores en la misma fila tienen igual precedencia).

Precedencia	Asociatividad
`. ( ) [ ]`	Izquierdas
`NOT`	Derechas
`AND`	Izquierdas
`OR`	Izquierdas
`>, <, = <>`	Izquierdas

Nuevamente, para alterar el orden de evaluación de las operaciones en una expresión lógica prescrita por las reglas de prelación se puede hacer uso de los paréntesis. Esto implica que toda expresión lógica encerrada entre paréntesis es también una expresión lógica. En el listado 11 se exponen algunos ejemplos sobre precedencia y asociatividad y cómo la parentización puede alterar la misma.

Listado 11. Ejemplo de precedencia y asociatividad en expresiones lógicas

```
a > b > c (* Error *)

a > b AND b > c (* Evalúa primero AND, ERROR de tipos *)

FALSE AND b > c (* ERROR de tipos *)

TRUE OR (b > c) (* se opera > y después OR *)

NOT (a > b) AND (c > d) (* se opera el primer >, NOT, segundo >
y AND *)

NOT (a > b AND c > d) (* Error de tipos *)

(a > b) AND (b > c) AND (c > d) (* Asociatividad a izquierdas *)

(a > b) AND ((b > c) AND (c > d)) (* Alteración *)

(a > b) OR (b > c) OR (c > d) (* Asociatividad a izquierdas *)

(a > b) OR ((b > c) OR (c > d)) (* Alteración *)
```

Si en una expresión se mezclan operadores aritméticos y lógicos, el orden de precedencia es el mostrado en la siguiente tabla:

Precedencia
NOT
* / AND
+ - OR
> < = <>

Expresiones de acceso a vectores

Una vez declarado un vector es posible acceder individualmente a cada uno de sus elementos de forma indexada. Para ello se utilizan los operadores `[]` de acceso a vector. Al poder usarse expresiones aritméticas 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, sí es posible realizar esa comprobación en la

fase de análisis semántico, y de hecho se debe de hacer en la práctica. El listado 12 muestra un ejemplo de uso de vectores en un programa en ModulUNED.

Listado 12. Ejemplo de uso de vectores en ModulUNED

```
TYPE TipoVectorEnteros = ARRAY [1..3] OF INTEGER;

     TipoVectorBooleanos = ARRAY [1..3] OF BOOLEAN;

VAR v : TipoVectorEnteros;

    b : TipoVectorBooleanos;

BEGIN

   v[1] := 3;

   v[2] := 3 * v[1];

   v[3] := v[3] + v[2];

   b[1] := v[1] > 0;

   b[2] := TRUE;

   b[3] := b[1] AND b[2];

END ejemplo;
```

Expresiones de acceso a campos de registros

Para acceder a los campos de un registro se utiliza el operador de acceso a registro **.** (punto). Una vez declarado un registro se pueden crear variables de ese tipo. El analizador semántico debe comprobar que el campo al que se trata de acceder existe realmente. El listado 13 muestra un ejemplo de uso de registros en ModulUNED.

Listado 13. Ejemplo de uso de registros en ModulUNED

```
PROCEDURE persona;

TYPE TipoPersona = RECORD

                    edad   : INTEGER;

                    dni    : INTEGER;

                    casado : BOOLEAN;

                   END;

VAR juan : TipoPersona;

    a: INTEGER;

BEGIN

      juan.edad := 23;

      juan.dni := 1234567;
```

```
        juan.casado := FALSE;

        a := juan.edad;

END persona;
```

Invocación de funciones

Para llamar a una función ha de escribirse su nombre indicando entre paréntesis los parámetros de la llamada. Los parámetros pueden ser constantes numéricas, lógicas o expresiones, incluyendo valores de un registro, elementos de un vector o la llamada a una función.

El analizador semántico debe detectar si la invocación de una función se realiza en el contexto de una expresión. Si se realiza en el contexto de una sentencia, debe generarse error.

En caso de no necesitar parámetros no es necesario incluir los paréntesis en la llamada. En el listado 14 se muestran algunos ejemplos.

Listado 14. Ejemplo llamadas a funciones

```
c:= suma(a,b);

d:= resta(n.centenas, v[2]);

a: = funcion1;
```

2.2.6.2 Sentencias

ModulUNED dispone de una serie de sentencias que permiten realizar determinadas operaciones dentro del flujo de ejecución de un programa. Todas las sentencias han de terminar con el delimitador ";". A continuación describimos en detalle cada una de ellas.

Sentencia de asignación

Las sentencias de asignación sirven para asignar un valor a una variable, elemento de un vector 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.

```
ref := expresión;
```

Donde `ref` es una referencia a una variable, elemento de un vector o campo de un registro y expresión es una expresión del mismo tipo que la referencia (el Analizador Semántico realizará esta comprobación de tipos). El listado 15 muestra algunos ejemplos de uso de sentencias de asignación:

Listado 15. Ejemplos de uso de la sentencia de asignación en ModulUNED

```
i := 3 + 7;

v[i] := 10;

distinto := 3<>4;

juan.casado := true;
```

```
a := suma(2,2); (*llamada a función*)
```

En ModulUNED no es posible hacer asignaciones a vectores ni estructuras de forma directa. Así son errores de compilación (de ello se encarga el Analizador Semántico) sentencias como `vector1 := vector2` o `registro1 := registro2;`

Sentencia de control de flujo condicional IF – THEN – ELSE

La sentencia `IF – THEN – ELSE` 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:

```
IF expresionLogica THEN sentencias1 END;

IF expresionLogica THEN sentencias1 ELSE sentencias2 END;
```

Donde `expresionLogica` es una expresión lógica que se evalúa para comprobar si debe ejecutarse el bloque de sentencias `sentencias1` o `sentencias2`. En concreto si el resultado de dicha expresión es cierta, es decir, resulta igual a `TRUE`, se ejecuta el bloque `sentencias1`. Si existe `ELSE` y la condición es falsa (`FALSE`), se ejecuta el bloque `sentencias2`.

No es necesario que la expresión lógica esté delimitada entre paréntesis. En caso de no existir se aplicará la precedencia de operadores. Esto se aplica también a las sentencias WHILE y FOR.

El listado 16 ilustra un ejemplo de sentencia IF.

Listado 16. Ejemplo de sentencia IF – THEN – ELSE

```
IF (a < b) THEN
      A := b;
ELSE
      A := a + b;
      B := a;
END;
IF (a>b) THEN (* IF anidante *)
      a:=b;
      IF a>c THEN (* IF anidado y sin paréntesis *)
            a:=c;
      ELSE
            c:=a;
      END;
ELSE
```

```
        b:=a;
    END;
```

Como se observa en el listado 16, este tipo de construcciones pueden anidarse con otras construcciones IF o con otros tipos de sentencias de control de flujo que estudiaremos a continuación.

Sentencia de control de flujo iterativo WHILE

La sentencia `WHILE` 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`. Su estructura es:

```
WHILE expresionLogica DO

    sentencias;

END;
```

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 (TRUE), 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. Además una estructura `WHILE` puede contener otras estructuras de control anidadas dentro del bloque de sentencias. En el listado 17 se muestra un ejemplo de bucle `WHILE`.

Listado 17. Ejemplo de sentencia WHILE – DO

```
WHILE a<5 DO

   a:=a+1;

   WRITEINT(a);

END;
```

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 diferencia estriba en que la primera está especialmente recomendada para recorrer rangos de valores dirigidos por un índice, mientras que el control de la interacción de la segunda queda dirigido por una expresión lógica. La estructura sintáctica de una sentencia FOR es:

```
FOR indice := expresionComienzo TO expresionFinal

DO

   sentencias;

END;
```

Donde `indice` es el índice que regula la iteración, `expresionComienzo` y `expresionFinal` las expresiones aritméticas que indican el valor inicial y final que deberá alcanzar el índice y `sentencias` el bloque se sentencias a ejecutar en cada iteración. Este bloque se ejecutará tantas veces como sea necesario para que el índice tome por orden todos los valores comprendidos dentro del rango acotado por `expresionComienzo` y `expresionFinal`. Es decir, el bloque de sentencias se repetirá mientras el valor de `indice` sea menor o igual a `expresionFinal` tomando como valor inicial el definido en `expresionComienzo`, incrementándose automáticamente en una unidad en cada iteración. La estructura FOR permite que otras estructuras de control formen parte del bloque de sentencia a iterar de forma que se creen anidamientos. El listado 18 muestra un ejemplo de sentencia FOR.

Listado 18. Ejemplo de sentencia FOR – DO

```
b:=5;

FOR a:=1 TO b DO

      WRITEINT(a);

END;
```

Sentencias de Entrada / Salida

ModulUNED dispone de una serie de procedimientos predefinidos que pueden ser utilizados para emitir por la salida estándar (pantalla) resultados de diferentes tipos. Estos procedimientos están implementados dentro del código del propio compilador, lo que implica que: 1) son subprogramas especiales que están a disposición del programador y 2) constituyen palabras reservadas del lenguaje. En concreto disponemos de 3 procedimientos. A continuación detallamos cada uno de ellos.

- WRITESTRING. Este procedimiento toma una constante literal de tipo cadena de texto y la muestra por la salida estándar. Indicar que ***se debe generar un salto de línea*** después de mostrar el resultado por pantalla. El ejemplo `WRITESTRING ("Hola mundo");` mostrará el texto `Hola mundo`.

- WRITEINT. Este procedimiento recibe una expresión de tipo entero (`INTEGER`) y muestra su resultado por la salida estándar. Indicar que ***se debe generar un salto de línea*** después de mostrar el resultado por pantalla. Por ejemplo, `WRITEINT(12); WRITEINT(a);`

- WRITELN. Este es un procedimiento sin parámetros que provoca la impresión de la secuencia de escape de salto de línea por la salida estándar. Por ejemplo, `WRITELN();`

Sentencias de llamada a procedimientos

Para llamar a un procedimiento ha de escribirse su nombre indicando entre paréntesis los parámetros de la llamada. Los parámetros pueden ser constantes numéricas, lógicas o expresiones, incluyendo valores de un registro, elementos de un vector o la llamada a una función.

El analizador semántico debe detectar si la llamada a un procedimiento se realiza en el contexto de una sentencia. Si se realiza en el contexto de una expresión, debe generarse error.

En caso de no necesitar parámetros no es necesario incluir los paréntesis en la llamada. En el listado 19 se muestran algunos ejemplos.

Listado 19. Ejemplo llamadas a procedimientos

```
procedimiento1;

procedimiento1();

escribeVector(vectorEnteros);
```

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. Los errores léxicos y sintácticos los genera la estructura proporcionada. 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, subprograma no definido, campo de registro inexistente, campo de registro duplicado, demasiados parámetros en llamada a subprograma, etc.

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 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.** Para su realización se usarán las herramientas JFlex, Cup y ENS2001 además de *seguir la estructura de directorios y clases que se proporciona*. Más adelante se detallan estas herramientas.

En este documento no se abordarán las directrices de implementación de la práctica que serán tratadas en otro documento 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.

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 alumno. Para cada especificación del lenguaje (ver siguiente apartado con la división del trabajo) se proporciona una de estas implementaciones.

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 junio 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 complejos	Vector	-	**X**
	Registro	**X**	-
Paso de parámetros	Por valor	-	**X**
	Por referencia	**X**	-
Operadores aritméticos	+	**X**	-
	-	-	**X**
	*	-	**X**
	/	**X**	-
Operadores relacionales	<	**X**	-
	>	-	**X**
	=		**X**
	<>	**X**	-
Operadores lógicos	AND	**X**	-
	OR	-	**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:

Junio	15 de junio de 2020
Septiembre	14 de septiembre de 2020

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. 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.

3.2.2 Formato de entrega

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

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

(Ejemplo: a-gonzalez1.zip)

Dicho archivo contendrá la estructura de directorios que se describe 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.

3.2.3 Trabajo a entregar

El trabajo a realizar contemplará la realización 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 los subprogramas. 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 subprogramas, 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.

3.2.3.1 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, procedimientos 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 los subprogramas: parámetros formales 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, funciones y procedimientos).

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 subprograma definido; añadir una entrada en la TS por cada símbolo declarado (variables, constantes y subprogramas) y finalmente comprobar en las expresiones el uso de elementos de tipos compatibles entre si. Asimismo debe comprobarse que no existen duplicaciones 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.

3.2.3.2 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 los subprogramas.

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

3.2.3.3 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 los subprogramas. El código final generado debe de funcionar correctamente en la herramienta ENS2001.

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

3.2.3.4 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 arrancará ENS2001 y cargará el archivo ensamblador generado, ejecutándolo y comprobando su correcto funcionamiento.

3.2.3.5 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.

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 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** 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 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**. 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).

Listado 20. Ejemplo de un programa en ModulUNED

```
MODULE Ejemplo;
      CONST MAX = 5;
      TYPE Mivector = ARRAY [1..MAX] OF INTEGER;
      VAR  i : INTEGER;
            x : BOOLEAN;
      v1 : Mivector;

      PROCEDURE EsMayorQueDos (a:INTEGER): BOOLEAN;
            VAR dos: INTEGER;
                  result: INTEGER;
            esMayor : BOOLEAN;
      PROCEDURE Imprime (numero:INTEGER);

            BEGIN
                  WRITESTRING("imprimiendo");
                  WRITEINT(numero);
            END Imprime;

            BEGIN
                  dos:=2;
                  result:=a+dos;
                  IF result>2 THEN
                        Imprime(result);
                        esMayor:=TRUE;
            ELSE
                  esMayor:=FALSE;
            END;
            RETURN esMayor;
            END EsMayorQueDos;
```

```
BEGIN
      v1[1] := 1;
      v1[2] := 2;
      v1[3] := 3;
      FOR i:=1 TO 3 DO
            x := EsMayorQueDos(v1[i]);
      END;
END Ejemplo;
```