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

Práctica de Procesadores del Lenguaje II

Especificación del lenguaje HAda

Dpto. de Lenguajes y Sistemas Informáticos
ETSI Informática, UNED

Alvaro Rodrigo Yuste
Anselmo Peñas (coordinador)

Curso 2012 - 2013

Contenido

1 Introducción

En este documento se define la práctica de la asignatura de Procesadores del Lenguaje II correspondiente al curso 2012/2013. El objetivo de la práctica es realizar un compilador del lenguaje HAda[1], que implementa un lenguaje similar a PASCAL usando la sintaxis del lenguaje Ada.

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

A lo largo de este documento se intentará clarificar todo lo posible la sintaxis y el comportamiento del compilador de HAda, 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 HAda. 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 semánticamente válidas.

2.1 Aspectos Léxicos

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

2.1.1 Comentarios

Un comentario es una secuencia de caracteres que se encuentra a continuación del limitador de principio de comentario "--". Todos los caracteres desde el inicio de comentario hasta el fin de línea 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. Los comentarios pueden aparecen en la misma línea después de una sentencia. Algunos ejemplos de comentarios correctos e incorrectos son los siguientes:

[1] Esta especificación se basa en la realizada por Emilio Julio Lorenzo y Anselmo Peñas en la asignatura Procesadores de lenguajes durante el curso 2011-2012.

Listado 1. Ejemplos de comentarios

```
-- Este es un comentario correcto

-- Este comentario

-- ocupa varias

-- líneas

- - Este es un comentario erroneo

x:=x+2 -- Comentario despues de una sentencia
```

2.1.2 Constantes literales

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

- **Enteras**. Las constantes enteras permiten 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. Si se pretende representar una cantidad negativa será necesario hacerlo mediante una expresión cuyo resultado será el valor deseado. Por ejemplo para representar – 2 podría escribirse `0 - 2`

- **Lógicas. .** Las constantes lógicas representan valores de verdad (cierto o falso) que son utilizados 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.

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

2.1.3 Identificadores

Un identificador consiste, desde el punto de vista léxico, en una secuencia ordenada de caracteres y dígitos que comienzan obligatoriamente por una letra. Los identificadores se usan para nombrar entidades del programa tales como las constantes, los tipos definidos por el usuario, las variables o los subprogramas definidos por el programador. El lenguaje **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` son identificadores iguales. La longitud de los identificadores no está restringida, pero el alumno es libre de hacerlo en caso de que lo considere necesario.

2.1.4 Palabras reservadas

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

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	Operador lógico
array	Declaración de un vector
begin	Comienzo de un bloque de sentencias en un subprograma
Boolean	Tipo lógico
constant	Declaración de constantes simbólicas
else	Comienzo del cuerpo de alternativa de una condicional if
end	Final de subprograma o de una estructura de control
False	Constante lógica que representa falso

	for	Comienzo de un bucle for
	function	Comienzo de una función
	if	Comienzo de una sentencia condicional if
	in	Inicio del rango en bucle for
	Integer	Tipo entero
	is	Comienzo del cuerpo de un subprograma y parte de la declaración de un registro
	loop	Comienzo /fin del bloque de sentencias de un bucle for y while
	of	Parte de la declaración de un vector
	or	Operador lógico
	out	Indica paso por referencia
	procedure	Comienzo de un procedimiento
	Put_line	Procedimiento predefinido para mostrar información por pantalla
	record	Declaración de un registro
	return	Parte de la declaración de una función y sentencia de retorno
	then	Comienzo de bloque de sentencias en un if
	True	Constante lógica
	type	Declaración de un tipo estructurado
	while	Comienzo de un bucle while

2.1.5 Delimitadores

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

DELIMITADOR	DESCRIPCIÓN
"	Delimitador de constante literal de cadena
( )	Delimitadores de expresiones, de parámetros, rango y acceso a vectores
..	Delimitador entre valores de un rango de un vector y de sentencia for
-- (Salto línea)	Delimitadores de comentario
,	Delimitador en listas de identificadores
;	Delimitador en secuencias de sentencias y secuencias de parámetros
:	Delimitador de tipo en una declaración de variable o constante y en parámetros

2.1.6 Operadores

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

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

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

2.2 Aspectos Sintácticos

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

2.2.1 Estructura de un programa y ámbitos de visibilidad

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

Listado 2. Estructura general de un programa en HAda

```
procedure nombrePrograma () is

      -- Declaración de constantes simbólicas

      -- Declaración de tipos globales

      -- Declaración de variables globales

      -- Declaración de subprogramas

begin

      -- Lista sentencias

end nombrePrograma;
```

Como puede apreciarse un programa en HAda comienza con la declaración de un procedimiento principal *nombrePrograma* que no recibe ningún parámetro. A continuación se declaran *opcionalmente*, pero *en este orden*, las constantes simbólicas, los tipos y las variables globales, terminando con los subprogramas (funciones y procedimientos). A partir de la palabra reservada **begin** se insertará la lista de sentencias por las que comenzará a ejecutarse el programa. Respecto a los subprogramas existen una serie de restricciones estructurales importantes:

- 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 del de g se invoca a f.

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

2.2.2 Declaraciones de constantes simbólicas

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

Algunos lenguajes resuelven el procesamiento de las constantes en la fase de análisis léxico haciendo una pasada sobre el fichero fuente y sustituyendo cada aparición de una constante simbólica por su correspondiente valor literal. En esta práctica se pretende que esta tarea sea llevada a cabo por las fases posteriores (análisis sintáctico, semántico y generación de código).

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

```
nombre: constant := valor;
```

Donde *nombre* es el nombre simbólico que recibe la constante definida dentro del programa **constant** es una palabra reservada y *valor* un valor constante literal de tipo entero positivo (número sin signo) o lógico (True, False) de manera que cada vez que éste aparece referenciado dentro del código se sustituye por su valor establecido en la declaración. Para cada constante declarada es necesario utilizar esta cláusula, terminando obligatoriamente en ";". Por convenio el nombre de la constante suele expresarse en mayúsculas, lo que ayuda a la hora de leer el código final, pero no es obligatorio y pueden declararse también en minúsculas.

A continuación se muestran algunos ejemplos de declaración de constantes.

Listado 3. Ejemplos de declaración de constantes simbólicas

```
CIERTO: constant := True;

FALSO: constant := False;

UNO: constant := 1;

TRES:constant := 2+1; -- ERROR, valor no puede ser una expresion

DOS: constant := +2;  -- ERROR, el entero no ha de llevar signo

CINCO: constant : =5  -- ERROR, falta punto y coma
```

2.2.3 Declaración de tipos

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

2.2.3.1 Tipos primitivos

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

Tipo entero (Integer)

El tipo entero representa valores enteros positivos y negativos. Por tanto, a las variables de este tipo se les puede asignar el resultado de evaluar una expresión aritmética, ya que dicho

resultado puede tomar valores enteros positivos y negativos. El rango de valores admitido para variables de este tipo oscila entre -32728 y +32727 ya que el espacio en memoria que es reservado para cada variable de este tipo es de 1 palabra (16 bits).

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

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

Listado 4. Ejemplos de uso del tipo Integer

```
x : Integer;

procedure (x,y: Integer) is ...
```

Tipo lógico (Boolean)

El tipo lógico representa valores de verdad, representados por las constantes literales *True* y *False*. Para referirse a este tipo de datos se utiliza la palabra reservada *Boolean*. A continuación se muestran unos ejemplos de declaración de tipo lógico

Listado 5. Ejemplos de uso del tipo Boolean

```
esMayor : Boolean;

procedure (esCierto: Boolean) is ...
```

2.2.3.2 Tipos Estructurados

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

En lo que respecta a la equivalencia de tipos se considera que dos tipos son equivalentes únicamente si tienen el mismo nombre. Por ejemplo dos variables serán del mismo tipo si su tipificación hace referencia a una misma declaración de tipo (o tipo primitivo) independientemente de la estructura interna de los mismos. Es decir, la tipificación en HAda es nominal.

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

Tipo Vector (Array)

Un vector es una estructura de datos que puede almacenar un rango de valores de un mismo tipo primitivo. Este rango se define en su declaración a través de dos constantes enteras

(literales o simbólicas) conocidas en tiempo de compilación, lo que significa que esos valores nunca pueden ser una expresión. La sintaxis de declaración para matrices es la siguiente:

```
type nombre is array (n1..n2) of Tipo
```

Donde **type** es una palabra reservada para indicar que se va a definir un nuevo tipo. *Nombre* es identificador del vector. Los índices del vector se definen por n1 y n2. *Tipo* indica el tipo de los elementos del array, que pueden ser *únicamente enteros o lógicos*.

El acceso a los datos de un vector es una expresión y como tal se explicará más adelante.

El listado 6 presenta algunos ejemplos de declaración de matrices.

Listado 6. Ejemplos de uso de vectores

```
MIN: constant:= 1;

MAX: constant:= 5;

type Vector1 is array(1..10) of Integer;

type Vector2 is array(MIN..MAX) of Integer;

type Vector3 is array(3..5) of Boolean;

-- En el caso de Vector3 el rango empieza con el índice 3

-- y termina en 5. Es decir, puede tener 3 elementos.

type Vector1 is array(MIN..15) of Integer;

-- A continuación se muestran varios ejemplos de errores:

type V4 is array(1..2+1) of Integer; -- error. No expresiones

type V2 is array(4..6) of Vector3; -- error. Tipo no permitido
```

Tipo Registro (Record)

Un registro es una estructura de datos compuesta que permite agrupar varios elementos de distinto tipo. Para definir un registro se utiliza la palabra reservada *record* seguida de una lista de declaraciones de campos. La declaración de un registro sigue la siguiente sintaxis:

```
type nombre is record
      nombreCampo1: TipoCampo1;
      nombreCampo2: TipoCampo2;
      ...
end record;
```

Donde `nombre` será el identificador del tipo registro, y `nombreCampoi`, son los nombres de los distintos campos del registro. Los tipos de los campos *pueden ser entero, lógicos u otros registros*.

El acceso a los datos de un registro es una expresión y como tal se explicará más adelante.

A continuación se muestran varios ejemplos de uso de registros.

Listado 7. Ejemplos de uso de registros

```
type Tpersona is record
      dni: Integer;
      edad : Integer;
      casado : Boolean;
end record;
type Tfecha is record
      dia: Integer;
      mes : Integer;
      anyo : Integer;
end record;
type Tcita is record
      usuario:Tpersona;
      fecha:Tfecha;
end record;
```

2.2.4 Declaraciones de variables

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

```
nombre1, nombre2 : tipo;
```

Donde `tipo` es el nombre de un tipo primitivo del lenguaje o el de un tipo definido por el usuario y `nombre1, nombre2,...` son identificadores para las variables declaradas. Como se ve en la sintaxis, si se desean declarar varias variables de un mismo tipo en la misma línea ha de hacerse utilizando el delimitador ",".Nótese que **no** se contempla la posibilidad de realizar asignaciones en línea. Es decir, declaraciones del tipo "*nombre : tipo := valor*" son incorrectas.

En el siguiente listado se muestran algunos ejemplos de declaraciones de variables. Se supone que los tipos compuestos utilizados (Tpersona y Vector1) han sido previamente declarados y corresponden con los expuestos en los listados 7 y 6 :

Listado 8. Ejemplos de declaración de variables

```
x : Integer;

x,y : Integer;

a : Boolean;

p1,p2: Tpersona;

ve1 : Vector1;

a: Integer := 1; -- error. Asignacion no permitida
```

2.2.5 Declaración de subprogramas

En el lenguaje propuesto se pueden declarar subprogramas para organizar modularmente el código. Un subprograma es una secuencia de instrucciones encapsuladas bajo un nombre con un conjunto ordenado de parámetros tipificados (opcionalmente ninguno) y, en el caso de funciones, un valor de retorno también tipificado. Los procedimientos no devuelven ningún valor. El uso de funciones con valor de retorno se emplea en el contexto de una expresión mientras que las llamadas a procedimientos se consideran sentencias.

2.2.5.1 Funciones

En la sección 2.2.6.1 (Invocación de funciones) abordaremos la invocación de funciones. Por ahora nos ocuparemos de describir la sintaxis de su declaración:

```
function nombre (param1, param11…:tipo1; para2, param21…:tipo2)
            return tipoRetorno is
      -- Declaración de tipos locales
      -- Declaración de variables
      -- Declaración de subprogramas anidados

begin
      -- Bloque de sentencias. Debe incluir la sentencia return
end nombre;
```

Donde *nombre* es el nombre de la función. Los parámetros se indican entre paréntesis siendo una lista de identificadores separados por coma, correspondientes a un determinado tipo. Los parámetros de diferente (también pueden ser iguales) tipo han de separarse con punto y coma ";" . Si la función no tiene parámetros, al nombre de la función le siguen paréntesis vacíos "()".A continuación *tipoRetorno* indica el tipo de retorno de la función. Éste debe ser necesariamente un tipo primitivo (Integer, Boolean).

Tras la declaración de la cabecera aparece la declaración tipos y variables locales, así como la de otros subprogramas (funciones o procedimientos) anidados (Ver sección 2.2.6.2 Anidamiento de subprogramas y ámbitos de visibilidad). Por último, después de la palabra reservada `begin` comienza el bloque de sentencias. Hay que destacar que no es posible declarar tipos ni variables en la parte de sentencias.

Una sentencia relevante dentro del cuerpo de la función es la sentencia **`return`** (no confundir con la usada en la cabecera de la función). Esta instrucción indica un punto de salida de la función y provoca que el flujo de control de ejecución pase de la función llamada a la siguiente instrucción dentro de la función llamante. Nótese que una función debe tener una instrucción de retorno por cada punto de salida. La sentencia `return` puede venir seguida de una expresión que indique el valor de retorno. El tipo de esta expresión debe coincidir, naturalmente, con el tipo de retorno especificado en la cabecera de la función. En casos como este, la sentencia `return` evalúa la expresión y devuelve su valor al contexto del programa llamante.

Algunos de los errores en funciones son de tipo sintáctico, como por ejemplo construir mal la cabecera, y deben ser tratados en la fase de análisis sintáctico. Existen, sin embargo, otro tipo de errores que deben ser comprobados en otras fases. Por ejemplo, la comprobación de la existencia de `return` debería delegarse al análisis semántico. A efectos de simplificar el desarrollo de la práctica basta con asegurar que existe al menos un return en el cuerpo de la función, independientemente de su ubicación.

A continuación se muestran varios ejemplos de declaraciones de funciones.

Listado 9. Ejemplos de declaración de funciones

```
function uno() return Integer is
begin
      return 1;
end uno;


function suma (x, y: Integer) return Integer is
      z: Integer;
begin
      z:= x+y;
```

```
      return z;
end suma;

function mayor (x, y: Integer) return Boolean is
begin
      return x>y;
end resta;

-- Ejemplo declaración de subprogramas anidados
function padre (x:Integer; y:Integer) return Integer is
      z: Integer;
      procedure escribe (a: Integer) is
      begin
            Put_line (a); -- Muestra el valor de a
      end hijo;
begin
      z:= x+y;
      escribe (z);
      return z;
end padre;

-- ERROR: function dos faltan los paréntesis ()
function dos return Integer is
begin
      return 2;
end dos;

-- ERROR: function tres debe devolver un tipo primitivo
function tres() return Tpersona is
begin
```

```
        return p1;
    end tres;


    -- ERROR: function cuatro falta sentencia return
    function cuatro(a:Integer; b:Integer) return Integer is
        y:Boolean
    begin
        y:= a and b;
    end cuatro;
```

2.2.5.2 Procedimientos

Los procedimientos son similares a las funciones descritas anteriormente. Su única diferencia es que no devuelven ningún valor y por tanto no deben contener la sentencia return. Su sintaxis es:

```
    procedure nombre (param1,param11…:tipo1;para2,param21…:tipo2) is
        -- Declaración de tipos locales
        -- Declaración de variables
        -- Declaración de subprogramas anidados
    begin
        -- Bloque de sentencias
    end nombre;
```

A continuación se muestran varios ejemplos de declaración de procedimientos:

Listado 10. Ejemplos de declaración de procedimientos

```
    procedure vacio()is
    begin
    end vacio;


    procedure suma (x, y, z: out Integer) is
        -- parametros pasados por referencia
    begin
        z:= x+y;
    end suma;
```

```
-- ERROR. Procedure devuelve. No puede tener return
procedure devuelve () is
begin
      return 1;
end devuelve;
```

2.2.5.3 Paso de parámetros a subprogramas

En el lenguaje propuesto es posible invocar a subprogramas pasando expresiones o referencias de alguno de los tipos definidos en la sección 2.2.3 como parámetros a un subprograma. El paso de parámetros es posicional. Por ello, el compilador debe asegurar que el tipo, orden y número de parámetros actuales de la invocación de un subprograma coincidan con el tipo, orden y número de los parámetros formales definidos en la cabecera de la declaración de dicho subprograma. En caso de no ser así deberá emitirse un mensaje de error. En concreto, el paso de parámetros actuales a un subprograma en HAda puede llevarse a cabo de dos formas diferentes: paso por valor y paso por referencia.

Paso por valor

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

Paso por referencia

En este caso, el parámetro sólo podrá ser una referencia (no una expresión) y el compilador transmite al subprograma la dirección de memoria donde está almacenado el parámetro actual, de forma que las modificaciones que se hagan dentro de éste tendrán efecto sobre el argumento una vez terminada la ejecución del mismo. El paso de parámetros por referencia es utilizado para pasar argumentos de salida o de entrada / salida. Para indicar que un parámetro se pasa por referencia debe precederse su tipo con la palabra reservada **out**. Se considera un error de compilación pasar expresiones como argumentos actuales a un subprograma donde se ha declarado un parámetro formal de salida o entrada/salida. Hay que aclarar que la semántica de esta instrucción es propia de HAda y no corresponde con el lenguaje Ada.

A continuación se incluyen ejemplos de funciones que utilizan paso por valor y por referencia:

Listado 11. Ejemplos de pasos de parámetros por valor y por referencia

```
-- Paso por valor

function suma (a,b:Integer) return Integer is

begin

      z:= x+y;

      return z;

end suma;


-- Paso por referencia

procedure mayor (x, y: out Integer; a: out Boolean) is

begin

      a:=x>y

end suma;
```

2.2.6 Sentencias y Expresiones

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

2.2.6.1 Expresiones

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

Expresiones aritméticas

Las expresiones aritméticas son aquellas cuyo cómputo devuelve un valor de tipo entero (Integer) al contexto de evaluación. Puede afirmarse que son expresiones aritméticas las constantes literales de tipo entero, las constantes simbólicas de tipo entero, los identificadores (variables o parámetros) de tipo entero y las funciones que tienen declarado un valor de retorno de tipo entero. Asimismo también son expresiones aritméticas la suma y resta de dos expresiones aritméticas. No es posible realizar operaciones aritméticas sobre tipos compuestos enteros. Por ejemplo, no es posible sumar dos vectores o registros enteros, del tipo “registro1 + registro2”.

Expresiones lógicas

Las expresiones lógicas son aquellas cuyo cómputo devuelve un valor de tipo lógico (Booelan) al contexto de evaluación. Son expresiones lógicas las constantes literales y simbólicas de tipo lógico, los identificadores de tipo entero, las funciones con un valor de retorno de tipo lógico y el resultado de operadores relacionales y lógicos. Es decir, la comparación con los operadores relacionales >, <, =, /= de dos expresiones aritméticas es también una expresión lógica. Los operadores relacionales sólo pueden comparar expresiones aritméticas, por tanto expresiones del tipo "true = true" deberán generar un error de tipo semántico. Los operadores lógicos (and , or) sólo pueden comparar expresiones lógicas. Es decir, expresiones del tipo "1 and 1" también deberán generar un error de tipo semántico. Además, no es posible aplicar los operadores relacionales ni lógicos con tipos estructurados. Por ejemplo, no es posible comparar dos vectores o arrays enteros, del tipo "vector1=vector2".

Expresiones de acceso a campos de registros

Para acceder a los campos de un registro se utiliza el operador de acceso a registro "." (punto). Dado que los campos de un registro pueden ser a su vez registros previamente declarados es posible concatenar el uso del operador de punto. Es decir si el resultado de la evaluación de una expresión que utiliza el operador de acceso a registros es otro registro, entonces es posible volver a aplicar nuevamente el operador de punto para profundizar en los campos del registro interno.

Listado 12. Ejemplo expresiones con acceso a campos de un registro

```
x:Integer;

mayorEdad:Boolean;

Tpersona p1;

Tfecha f1;

Tcita c1;

p1.dni := 1234;

p1.edad := 23;

f1.dia := 12;

f1.mes := 3;

c1.usuario := p1;

c1.fecha :=f1;

c1.fecha.anyo := 2012;

x := c1.usuario.edad;

mayorEdad := c1.usuario.edad > 18;

-- En este último ejemplo se compara un campo de un registro.

-- Esto es correcto. Lo que no es posible es comparar registros
```

```
-- enteros
```

Expresiones de acceso a vectores

El acceso a los datos de un vector se realiza de forma indexada. Para acceder individualmente a cada uno de sus elementos se utilizan los paréntesis "`()`". Dentro de estos delimitadores es preciso ubicar una expresión aritmética cuyo valor debe estar comprendido entre el rango definido al declarar el vector. Al poder usarse expresiones aritméticas para acceder a un elemento de una matriz, 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 será obligatorio 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.

El siguiente listado muestra distintos ejemplos de accesos a elementos de un vector.

Listado 13. Ejemplos de expresiones con acceso a elementos de un vector

```
v1: Vector1;

v2: Vector1;

v3: Vector3;

x: Integer;

v1(1):=12;

v3(3):= True;

v2(2):=v1(1);

x:=v1(1);

v2(2):=x;

x:= v1(1) + v2(2);

x:=v1(1) + v3(3) -- ERROR Semántico. Intenta sumar entero y

                 -- lógico
```

Invocación de funciones

Para llamar a un subprograma ha de escribirse su identificador indicando entre paréntesis los parámetros actuales de la llamada, que deben coincidir en número, tipo y orden con los definidos en la declaración de la función. Los parámetros pueden ser referencias a variables, campos de registros, elementos de un vector o expresiones. El uso de los paréntesis es siempre obligatorio. Así, si un subprograma no tiene argumentos, en su llamada es necesario colocar unos paréntesis vacios "()" tras su identificador.

Nótese que las llamadas a funciones actúan cómo expresiones pero las llamadas a procedimientos son sentencias y nunca una expresión. Por tanto, si se utilizan en el lugar de una expresión debe generarse un error. Esta comprobación es propia del análisis semántico.

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

Listado 14. Ejemplos de llamadas a funciones

```
x := resta(a, b);

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

a := funcion1();

b := suma (r.campo1, r.campo2); -- r es de tipo registro

c := suma (v(2), 3);          -- v es de tipo vector

funcion2(z) := 3;  -- ERROR.
```

Evaluación de expresiones

El modelo de evaluación de expresiones lógicas es en cortocircuito o modo de *evaluación perezosa*. Este modo de evaluación prescribe que 1) si en la evaluación de una conjunción (`and`) de dos expresiones la primera de las evaluadas es falsa la otra no debe evaluarse y 2) si en la evaluación de una disyunción (`or`) de dos expresiones la primera de las evaluadas es verdadera la otra no debe evaluarse.

Precedencia y asociatividad de operadores

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

Precedencia	Asociatividad
`. ( )`	Izquierda
`+ -`	Izquierda
`< >`	Izquierda
`= /=`	Izquierda
`and or`	Izquierda

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

Listado 15. Ejemplos de precedencia y asociatividad en expresiones

```
true or false and true  -- resultado: True

true or (false and true) -- resultado : True

2 + (2 - 3)   -- resultado : 1
```

2.2.6.2 Sentencias

El lenguaje propuesto dispone de una serie de sentencias que sirven a distintos propósitos dentro de un programa. Las sentencias se escriben dentro del cuerpo de los subprogramas (o procedimiento principal) declarados en el mismo. En efecto, tal como se describió anteriormente, tras la declaración de tipos, variables y subprogramas anidados, comienza una secuencia de sentencias que constituyen el código de la función. A lo largo de esta sección describiremos todos los tipos de sentencias existentes a excepción de la sentencia return que ya fue explicada en la sección 2.2.5.1.

Anidamiento de subprogramas y ámbitos de visibilidad

Las declaraciones de tipos y variables realizadas dentro de un subprograma sólo son visibles por las sentencias del mismo. Como dentro de un subprograma se pueden declarar otros anidados; en los hijos se verán las declaraciones locales y las realizadas en cada uno de los subprogramas de la jerarquía de anidamiento. El siguiente listado muestra un ejemplo de uso de subprogramas anidados.

Listado 16. Ejemplo de subprogramas anidados

```
procedure padre(a:Integer; b:Integer)is

      x: Integer;

      -- procedimiento anidado

      procedure hijo (x: Integer) is

      z: Integer;

      begin

            -- variable x es no local a procedimento hijo

            -- variable z es local a procedimiento hijo

            z:=x;

      end hijo;

begin  -- de padre

      x:=z; -- ERROR. Variable z no alcanzable por padre

end padre;
```

Esta estructura sintáctica articula una colección anidada de **ámbitos de visibilidad**. En este sentido podemos distinguir entre, 1) el ámbito de visibilidad global asociado a las declaraciones realizadas al inicio del programa, que es visible por todas las funciones declaradas y 2) el ámbito de visibilidad asociado a cada subprograma. Cuando hacemos referencia a una variable desde cualquier punto del programa podemos caer en uno de los 3 casos siguientes:

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

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

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

Cabe destacar que debido a la estructura de un programa en HAda no hay variables sólo locales al procedimiento principal, ya que las declaradas en ese ámbito son de tipo global a todo el programa.

Sentencias de asignación

Las expresiones 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. El compilador deberá comprobar en primer lugar la compatibilidad entre el tipo de la expresión a la derecha del operador de asignación con el de la referencia a la izquierda. El operador de asignación no es asociativo. Es decir no es posible escribir construcciones sintácticas del estilo a := b := c. Tampoco es posible hacer asignaciones a estructuras (vectores o registros) de forma directa. La sintaxis de la expresión de asignación es a:

```
ref := expresion;
```

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

Listado 17. Ejemplos de uso de asignación

```
i := 3 + 7;

m(i) := 10;

distinto := 3/=4; -- distinto es variable lógica

cita1.usuario.edad := 30;

a := 2 + suma(2,2);

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

```
                    -- aparecer en la parte izquierda de una
                    -- asignación
vector1 := vector2; -- ERROR. La asignación directa de un
                       -- vector a otro vector no está permitido
                       -- aunque ambas sean del mismo tipo */
```

Sentencia de control de flujo condicional if – then – else

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

```
if expresiónLogica then
   -- sentencias1

else
   -- sentencias2

end if;
```

Donde `expresionLogica` es una expresión lógica que se evalúa para comprobar si debe ejecutarse las sentencias1 o sentencias2. En concreto si el resultado de dicha expresión es cierta, se ejecuta el bloque de sentencias1.

La parte correspondiente al `else` es optativa y no tiene necesariamente que aparecer (ver ejemplos del listado 18). Si existe `else` y la condición es falsa, se ejecuta el bloque de sentencias2. `expresionLogica` no necesita llevar paréntesis, a no ser que sean parte de la expresión lógica. La sentencia ha de terminarse siempre con "`end if;`" , tenga o no tenga la parte else.

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

Listado 18. Ejemplos de sentencia if –else

```
-- if simple sin parte else
if a=b then
      a:=a+1;
end if;
-- expresión lógica  más compleja
if a=b and x>y then
   a:=a+1;
   b:=b+1;
end if;
-- expresión lógica con paréntesis
if a=b and (x>y or x>z) then
   a:=a+1;
   b:=b+1;
end if;
-- uso de else
if esCierto then
      valor:=true;
else
      valor:=false;
end if;
```

Sentencia de control de flujo condicional while

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

```
while expresionLogica loop
    -- sentencias
end loop;
```

Donde `expresionLogica` es la expresión lógica a evaluar en cada iteración, siguiendo las mismas normas expuestas en el caso de if-then-else. S`entencias` es el bloque de sentencias a ejecutar en cada vuelta. Es decir, si la expresión lógica es cierta, se ejecuta el bloque de sentencias y después se comprueba de nuevo la expresión. Este proceso se repite una y otra vez hasta que la condición sea falsa. En el siguiente listado se muestra un ejemplo de este tipo de bucle.

Listado 19. Ejemplo de sentencia while

```
while a<5 loop
   a:=a+1;
   Put_line (a);
end loop;
```

Sentencia de control de flujo iterativo for

La sentencia `for` es otra sentencia de control de flujo iterativo. La estructura sintáctica de una sentencia `for` es:

```
for indice in inicio..fin loop
      -- sentencias
end loop;
```

Donde `indice` es un identificador correspondiente a una variable entera que tomará el valor definido en la expresión numérica *`inicio`* y en cada iteración sumará uno hasta llegar al valor definido en la expresión numérica *`fin`*.

Hay que hacer notar que no es posible declarar la variable `indice` dentro de la sentencia for. La declaración de variables ha de ser anterior a la de las sentencias. En caso de asignarse un valor a la variable índice previo a su incialización en el bucle for, este será sustituido por el valor de *`inico`*. El siguiente listado muestra unos ejemplos de sentencia for.

Listado 20. Ejemplos de sentencia for

```
-- Ejemplo complete. Muestra por pantalla 1 2 3 4 5
procedure procedimiento () is
a:Integer;
begin
      a:= 10;
      for a in 1..5 loop -- a empieza en 1 aunque fuera 10
            Put_line(a);
      end loop
end procedimiento;
```

```
-- Ejemplo con expresiones númericas. La variable i debe
-- haber sido declarada previamente
for i in a..b+1 loop
      Put_line(a);
end loop
```

Debido a la estructura de esta sentencia es posible modificar el valor de la variable índice dentro del bloque de sentencias. Durante la ejecución de un programa se debería salir de este bucle en caso de que el valor de la variable índice sea igual o superior al indicado en fin. Es decir, **no** debe generarse un error (ni en compilación, ni en ejecución) en caso de sobrepasarse el valor final del bucle. No es necesario comprobar que el valor de inicio es menor al de fin, ni de que se trata de valores positivos.

Sentencias de invocación a un procedimiento

Como ya se discutió con anterioridad, la invocación de procedimientos es una sentencia y no una expresión. Por tanto la llamada a un procedimiento no puede ubicarse en un contexto sintáctico donde se espera una expresión sino solamente donde se espera una sentencia. El compilador debe detectar esta propiedad de la función y emitir un error en caso de que se use una llamada a procedimiento donde se esperaba una función. Recordamos que la diferenciación entre las llamadas a funciones y procedimientos puede delegarse al análisis semántico.

En lo que respecta a los demás aspectos (sintácticos, semánticos y funcionales) la invocación de procedimientos es idéntica a la que se describió para el caso de las funciones.

Sentencias de Entrada / Salida

El lenguaje HAda dispone de un procedimiento predefinido que puede ser utilizado para emitir mensajes de distinto tipo por la salida estándar (pantalla). Este procedimiento está implementado dentro del código del propio compilador lo que implica: 1) que es un procedimiento especial que está a disposición del programador y 2) constituye una palabra reservada del lenguaje y por tanto debe considerarse un error la declaración de identificadores con dicho nombre. Su sintaxis es la siguiente:

```
Put_line(parametro);
```

Put_line es la palabra reservada para invocar la sentencia de salida. Sólo puede admitir un parámetro que podrá ser una expresión numérica, lógica o una cadena de caracteres. En caso de una expresión lógica si es cierta debe mostrarse el texto: true o false (todo en minúsculas) según sea cierto o falso. En caso de intentar mostrar un vector o registro completo se debe generar un error. En caso de no recibir ningún parámetro no debe mostrar nada por pantalla, pero no sería un error. Por último, indicar que ***se debe generar un salto de línea*** después de mostrar el resultado por pantalla (se considerará un error en la práctica no hacerlo). No han de considerarse el uso de caracteres especiales ni secuencias de escape. No está permitido el uso de /n para saltos de línea. El listado 20 muestra unos ejemplos de uso.

Listado 21. Ejemplos de entrada / salida

```
Put_line (1);  -- Muestra 1

Put_line (x);  -- Muestra el valor de la variable x

Put_line (x+1);

Put_line (cita1.usuario.edad);

Put_line (True); -- Muestra el texto true

Put_line (1>2); -- Muestra el texto false

Put_line ("Hola mundo"); -- Muestra el texto Hola mundo

Put_line (a,b); -- ERROR. Sólo se admite un parámetro

Put_line (cita1); -- ERROR. No se muestran tipos compuestos
```

Se vuelve a insistir en que tras imprimir por pantalla el resultado se ha de generar un único salto de línea.

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. Debe indicarse obligatoriamente el número de línea en que ha ocurrido. Por lo demás, se valorarán intentos de aportar más información sobre la naturaleza del error, por ejemplo:

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

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

- Errores semánticos: Los errores motivados por la comprobación explícita de tipos de acuerdo al sistema de tipos del lenguaje constituyen errores de carácter semántico. Algunos 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.

El compilador generado deberá recuperarse de los errores sintácticos que se encuentre durante el proceso de análisis de un programa. Esto implica que deben utilizarse los

mecanismos pertinentes para que cuando se encuentre un contexto sintáctico de error el compilador genere un mensaje y continúe con el análisis con el ánimo de emitir la mayor cantidad de mensajes de error posible y no solamente el primero. Sin embargo, no se debe realizar una recuperación de errores a nivel léxico ni semántico. Así por ejemplo, si el compilador encuentra un carácter extraño en el código fuente o un error de concordancia de tipos, éste emitirá un mensaje de error y abortará el proceso de compilación.

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 empezar a trabajar lo antes posible y avanzar constantemente sin dejar todo el trabajo para el final. Se debe abordar etapa por etapa, pero hay que saber que todas las etapas están íntimamente ligadas entre sí, de forma que es complicado separar unas de otras. De hecho, es muy frecuente tener que revisar en un punto decisiones tomadas en partes anteriores, especialmente en lo que concierne a la gramática.

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

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

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

3.1 División del trabajo

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

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	Vectores	X	-
	Registros	-	X
Paso de parámetros	Por valor	-	X
	Por referencia	X	-
Operadores aritméticos	+	-	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	10 de junio de 2013
Septiembre	9 de septiembre de 2013

Si una vez entregada desea corregir algo y entregar una nueva versión, puede hacerlo las veces que sea necesario 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**. La práctica que no cumpla estas normas podrá ser suspendida.

Se recuerda que es necesario superar una **sesión de control obligatoria** a lo largo del curso para aprobar la práctica y la asignatura. Nos remitimos al documento de normas de la asignatura, dónde viene explicado las fechas y normativa a aplicar.

3.2.2 Formato de entrega

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

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

(Ejemplo: a-gonzalez1.rar)

Donde el identificador hace referencia al identificador del usuario en el curso virtual.

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

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

El índice de la memoria para la entrega será:

Portada obligatoria

1. El analizador semántico y la comprobación de tipos

1.1. Descripción de la Tabla de Símbolos implementada

Describir la solución tecnológica elegida para implementarla

2. Generación de código intermedio

    2.1. Descripción de la estructura utilizada

3. Generación de código final

    3.1. Descripción del registro de activación implementado

4. Indicaciones especiales

    En este punto se han de incluir aquellas apreciaciones que el alumno quiera hacer sobre su práctica. Por ejemplo: partes incompletas, interpretaciones dudosas del enunciado y soluciones tomadas, apuntes sobre necesidades de la ejecución del compilador, etc.

5. Conclusiones

6. Gramática

    En este punto se ha de incluir un esquema con las producciones de la gramática utilizada, sin incluir acciones semánticas ni atributos.

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

3.2.3 Trabajo a entregar

Debido a que en esta práctica se trabaja sobre el mismo lenguaje que en la práctica de la asignatura Procesadores del Lenguaje I, se sugiere partir del trabajo realizado en tal asignatura, que contemplaba el analizador léxico y el analizador sintáctico.

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. Las dos primeras etapas (análisis léxico y sintáctico) se han de reutilizar de la práctica de la asignatura Procesadores del Lenguaje I o ser implementadas puesto que son fundamentales para el compilador final.

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: nombres, tipo, valor…
- Para las variables: tipo, ámbito, tamaño en memoria...
- Para las funciones: 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 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.

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 CUADRUPLAS 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.

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.

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 ejecutar 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.

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. Además Cup también permite asociar acciones java entre los elementos de la parte derecha de la gramática de forma que éstas se vayan ejecutando a medida que se va construyendo el árbol de análisis sintáctico. Estas acciones permitirán implementar las fases de análisis semántico y generación de código intermedio.

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

4.3 Ensamblador ENS2001

ENS2001 es una máquina virtual que proporciona un sencillo entorno de ejecución basado en registros, memoria direccionada a nivel de byte y un sencillo juego de instrucciones. Además proporciona 2 entornos (uno en línea de comandos y otro con interfaz gráfica de usuario) que permite ejecutar los programas escritos en ENS2001 y depurarlos convenientemente.

4.4 Ant

Ant es una herramienta muy útil para automatizar la compilación y ejecución de programas escritos en java. Ant permite evitar situaciones habituales en las que, debido a configuraciones particulares del proceso de compilación, la práctica sólo funciona en el ordenador del alumno.

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

5 Ayuda e información de contacto

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

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

Se recomienda también la utilización de los foros como medio de comunicación entre alumnos y de estos con el Tutor de Apoyo en Red (TAR). 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, por el TAR o por el equipo docente sea más eficiente.

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

El alumno puede plantear sus dudas al tutor del Centro o a los profesores (procleng@lsi.uned.es). El profesor encargado de la práctica es Alvaro Rodrigo (alvarory@lsi.uned.es). *El alumno debe comprobar si su duda está resuelta en la sección de Preguntas Frecuentes (FAQ) o en los foros de la asignatura antes de contactar con el tutor o profesor. Por otra parte, si el alumno tiene problemas relativos a su tutor o a su Centro Asociado, debe contactar con el coordinador de la asignatura Anselmo Peñas* (anselmo@lsi.uned.es).

Anexo A. Programa de ejemplo completo

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

Listado 21. Ejemplo de un programa

```
procedure ejemplo () is

-- constantes simbólicas

MAYORDEEDAD: constant := 17;

-- tipos globales

type Tpersona is record

      dni: Integer;

      edad: Integer;

end record;

-- variables globales

Tpersona persona1;

-- subprogramas

function mayorDeEdad (edad: Integer) return Boolean is

begin

      if edad>MAYORDEEDAD then

            return True;

      else

            return False;

      end if;

end mayorDeEdad;


procedure imprimePersona(dni:Integer; edad:Integer)is

      -- variable local

      debug: Boolean;

      -- procedimiento anidado

      procedure escribe (x: Integer) is

      begin
```

```
            Put_line(x);
      end escribe;
begin  -- de imprimePersona
      escribe(dni);
      debug:=True;
end imprimePersona;


-- procedimiento principal
begin
      persona1.dni:=1234;
      persona1.edad:=23;
      if mayorDeEdad(persona1.edad)then
            Put_line("Persona:");
            imprimePersona(persona.dni, edad);
      end if;
end ejemplo;
```