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

Práctica de Procesadores del Lenguaje I

Especificación del lenguaje PL1UnedES v1.02

Dpto. de Lenguajes y Sistemas Informáticos
ETSI Informática, UNED

Alvaro Rodrigo
Anselmo Peñas (coordinador)

Curso 2019 - 2020

ÍNDICE

1 INTRODUCCIÓN

En este documento se define la práctica de la asignatura de Procesadores del Lenguaje I correspondiente al curso 2019-2020. El objetivo de la práctica es realizar un compilador del lenguaje PL1UnedES.

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

A lo largo de este documento se explicará la sintaxis y el comportamiento del compilador de PL1UnedES, 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 PL1UnedES, una versión convenientemente reducida resultado de la combinación de otros lenguajes y que usa palabras reservadas en castellano. En los siguientes apartados presentaremos la estructura general de los programas escritos en dicho lenguaje describiendo primero sus componentes léxicos y discutiendo después cómo éstos se organizan sintácticamente para formar construcciones del lenguaje.

2.1 Aspectos Léxicos

Desde el punto de vista léxico, un programa escrito en PL1UnedES 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 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. Los comentarios pueden aparecen en la misma línea después de una sentencia. Algunos ejemplos de comentarios se muestran en el listado 1.

2.1.2 Constantes literales

En PL1UnedES 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 constantes 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 `CIERTO` y `FALSO` e indican el valor cierto y falso respectivamente.

- **Enteras**. Las constantes enteras permiten representar valores enteros no negativos. Por ejemplo: `0, 32, 127`, etc (no es válido un número con más de un dígito y que empiece por cero como por ejemplo 007). En este sentido, no es posible escribir expresiones como `-2`, ya que el operador unario "–", no existe en este lenguaje.

- **Cadenas de caracteres**. Las constantes literales de tipo cadena consisten en una secuencia ordenada de caracteres ASCII. Están delimitadas por las comillas dobles, por ejemplo: "ejemplo de cadena". Las cadenas de caracteres se incluyen en la práctica únicamente para poder escribir mensajes de texto por pantalla mediante la instrucción `escribir` (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. Para mostrar un salto de línea se utilizará la instrucción `escribir` sin parámetros (se explicará más adelante).

2.1.3 Identificadores

Un identificador consiste, desde el punto de vista léxico, en una secuencia ordenada de letras y dígitos que comienzan obligatoriamente por una letra o un guion bajo. 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 (lo mismo ocurre con las palabras reservadas del lenguaje). 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

PALABRA CLAVE	DESCRIPCIÓN
`booleano`	Tipo lógico.
`cierto`	Constante lógica que representa un valor verdadero.
`comienzo`	Delimitador de comienzo de bloque de sentencias.
`constantes`	Declaración de constantes.
`de`	Asignación del tipo base en vectores
`en`	Rango de bucle para
`devolver`	Retorno de una función.
`entero`	Tipo entero.
`entonces`	Comienzo del cuerpo de una condicional IF.
`escribir`	Procedimiento predefinido que muestra contenido por pantalla
`falso`	Constante lógica que representa falso
`fin`	Delimitador de final de bloque de sentencias
`funcion`	Comienzo de una función
`no`	Negación lógica
`para`	Comienzo de un bucle para

sino	Comienzo del cuerpo de alternativa de una condicional si
Subprogramas	Comienzo de declaración de subprogramas
tipos	Comienzo de la declaración de tipos.
var	Parámetro por referencia
variables	Comienzo de la declaración de variables.
vector	Declaración de una estructura tipo vector.
Y	Y lógica.

2.1.5 Delimitadores

PL1UnedES 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 vectores.
#	Delimitador de comentario
,	Delimitador de identificadores
;	Delimitador de sentencias.

2.1.6 Operadores

En PL1UnedES existen diferentes tipos de operadores que son utilizados para construir expresiones por combinación de otras más sencillas como se discutirá más adelante. En concreto podemos distinguir los siguientes tipos:

Operadores aritméticos	+ (suma) * (producto)
Operadores relacionales	< (menor) == (igual)
Operadores lógicos	y (conjunción lógica) no (negación lógica)
Operador de asignación	=
Operadores de acceso	[ ] (acceso a elemento de vector)

Obsérvese que, como se advirtió con anterioridad, en PL1UnedES 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).

2.2 Aspectos Sintácticos

A lo largo de esta sección describiremos detalladamente las especificaciones sintácticas que permiten escribir programas correctos en PL1UnedES. Comenzaremos presentado la estructura general de un programa en dicho lenguaje y, posteriormente, iremos describiendo cada una de las construcciones que aparecen en detalle.

2.2.1 Estructura y ámbitos de un programa

Desde el punto de vista sintáctico, un programa en PL1UnedES es un fichero de código fuente con extensión ".pl1" que contiene una colección de declaraciones. En el código fuente del listado 2 se muestra un esquema general de la estructura de un programa PL1UnedES.

Listado 2. Estructura general de un programa en PL1UnedES

```
programa nombrePrograma:

        constantes

        #Declaración constantes globales

        tipos

        #Declaración tipos globales

        variables

        #Declaración variables globales

        subprogramas

        #Declaración subprogramas

comienzo

        #Lista sentencias

fin.
```

En PL1UnedES un programa consiste en una secuencia ordenada de construcciones sintácticas que empiezan por la declaración del programa con la instrucción `programa` seguida del nombre del programa y dos puntos (:). 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 `comienzo` y `fin`. 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.

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. Las constantes simbólicas pueden ser de tipo entero positivo o lógico. La sintaxis para la declaración de constantes simbólicas es la siguiente:

NOMBRE_CONSTANTE = valor;

Donde `NOMBRE_CONSTANTE` es el nombre simbólico que recibe la constante definida dentro del programa y `valor` su valor constante de tipo entero (número) o lógico (`cierto` o

Listado 3. Ejemplo de declaración de constantes en PL1UnedES

```
constantes

        TAMANO = 10;

        MESES = 12;

        ANYO = 2007;

        ABIERTO = cierto;
```

2.2.3 Declaraciones de Tipos

PL1UnedES 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 PL1UnedES 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. El tipo entero se representa con la palabra reservada `entero`.

Tipo lógico

El tipo lógico representa valores de verdad (verdadero o falso). Estos valores están representados por las constantes literales `cierto` y `falso`. Para referirse en PL1UnedES a este tipo de datos se utiliza la palabra reservada `booleano`.

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 PL1UnedES 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.

Tipo vector

Un vector es una estructura de datos que puede almacenar varios valores de un mismo tipo primitivo. La definición de vectores se apoya únicamente en el uso de tipos primitivos y no en el uso de otros tipos compuestos previamente definidos por el usuario. En este sentido NO existen en el lenguaje vectores de vectores.

Para definir un tipo vector se utiliza la palabra reservada `vector` 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 `de` 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 = vector [n1..n2] de 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. En el listado 4 se muestran ejemplos de declaración de vectores.

Listado 4. Ejemplo de declaración de vectores en PL1UnedES

```
tipos

      TipoVectorEnteros = vector [1..3] de entero;

      TipoVectorBooleanos = vector [1..3] de booleano;
```

2.2.4 Declaraciones de Variables

En PL1UnedES 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 `variables` (dentro de cada ámbito sólo puede existir una cláusula `variables`). Por su parte, una declaración de variables consiste en una lista de identificadores separados 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:

```
variables

    nombre11, nombre 12, ..., nombre1N : Tipo1;

    nombre21, nombre 22, ..., nombre2N : Tipo2;

      ...
```

Listado 6. Ejemplo de declaración de variables en PL1UnedES

```
variables

    x, z : entero;

    abierto : booleano;

    v1, v2 : TipoVector;

    v : TipoVectorEnteros;

    b : TipoVectorBooleanos;
```

2.2.5 Declaración de subprogramas

En PL1UnedES 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 sintaxis de un procedimiento en PL1UnedES es:

```
procedimiento nombre (param11, param12,..., param1N : Tipo1;
                      param21, param22,..., param2N : Tipo2;
                         ...
                      paramM1, paramM2,..., paramMN : TipoM):
        constantes
              #declaración constantes
        tipos
              #declaración tipos
        variables
              #declaración variables
        subprogramas
```

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 es opcional. Por tanto, la cabecera de un procedimiento sin parámetros tiene la siguiente estructura:

```
procedimiento nombre ():
```

Por su parte, la cabecera es seguida de un cuerpo con una estructura idéntica a la del programa principal (sección `constantes`, `tipos`, `variables`, `subprogramas` y sentencias encerradas entre los delimitadores `comienzo` y `fin;`). En el listado 7 presentamos un ejemplo de procedimiento.

Listado 7. Ejemplo de declaración de procedimiento en PL1UnedES

```
procedimiento ejemplo (a : entero):

variables b,c: entero;

comienzo

      c = a + 2;

      b = a + c;

fin;
```

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 PL1UnedES 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 vectores) y 2) dentro del cuerpo de una función debe aparecer al menos una vez el uso de la palabra reservada `devolver` 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 (la comprobación de que existe la palabra reservada `devolver` se delega para el análisis semántico, que no se realiza en esta práctica). La sintaxis de la declaración de una función es la siguiente:

```
funcion nombre (param11, param12,..., param1N : Tipo1;
                param21, param22,..., param2N : Tipo2;
                   ...
                paramM1, paramM2,..., paramMN : TipoM): Tipo:
    constantes
        #declaración constantes
    tipos
        #declaración tipos
    variables
        #declaración variables
    subprogramas
    #Declaración subprogramas
comienzo
        #Lista sentencias
fin;
```

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:

```
funcion nombre () : Tipo:
```

En el listado 8 presentamos un ejemplo de función

Listado 8. Ejemplo de declaración de una función en PL1UnedES

```
funcion ejFuncion (a,b : entero): entero:
variables suma: entero;
comienzo
        suma = a + b;
```

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.

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 y elementos de vectores, pero **no** vectores 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 PL1UnedES el paso de vectores completos como parámetros a un subprograma se realiza siempre por referencia. Para indicar que un parámetro 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 referencia:

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

```
procedimiento ejemplo (a : entero):

comienzo

      a = a + 1;

fin;

funcion ejFuncion (var a,b : entero): entero:

variables suma: entero;

comienzo

      suma = a + b;
```

En PL1UnedES 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 del comienzo de sección marcado por la palabra reservada `subprogramas` y antes de la palabra reservada `comienzo`.

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 PL1UnedES 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 y el producto 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

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 de dos expresiones lógicas. PL1UnedES 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 (`Y` y `NO`) 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
`NO`	Derechas
`Y`	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

Listado 11. Ejemplo de precedencia y asociatividad en expresiones lógicas

```
CIERTO Y (b < c) # se opera < y después Y

NO (a < b) Y (c < d) #se opera el primer <, NO, segundo <, Y

(a < b) Y (b < c) Y (c < d) #Asociatividad a izquierdas

(a < b) Y ((b < c) Y (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
NO
* Y
+
< ==

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. El listado 12 muestra un ejemplo de uso de vectores en un programa en PL1UnedES.

Listado 12. Ejemplo de uso de vectores en PL1UnedES

```
tipos TipoVectorEnteros = vector [1..3] de entero;

      TipoVectorBooleanos = vector [1..3] de booleano;


variables v : TipoVectorEnteros;

    b : TipoVectorBooleanos;

comienzo
```

```
    b[3] = b[1] y b[2];

fin
```

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 elementos de un vector o la llamada a una función.

En caso de no necesitar parámetros 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(10, v[2]);

a = funcion1();
```

2.2.6.2 Sentencias

PL1UnedES 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 o elemento de un vector. Para ello se escribe primero una referencia a alguno de estos elementos seguido del operador de asignación y a su derecha una expresión.

```
ref = expresión;
```

Donde `ref` es una referencia a una variable o elemento de un vector y expresión es una expresión del mismo tipo que la referencia. El listado 15 muestra algunos ejemplos de uso de sentencias de asignación:

Listado 15. Ejemplos de uso de la sentencia de asignación en PL1UnedES

```
i = 3 + 7;

v[i] = 10;

igual = 3==4;

a = suma(2,2); #llamada a función
```

Sentencia de control de flujo condicional si – entonces– sino

La sentencia `si – entonces – sino` permite alterar el flujo normal de ejecución de un programa en virtud del resultado de la evaluación de una determinada expresión lógica. Sintácticamente esta sentencia puede presentarse de dos formas:

```
si expresionLogica entonces: sentencias1 fin si;

si expresionLogica entonces: sentencias1 sino: sentencias2 fin si;
```

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, se ejecuta el bloque `sentencias1`. Si existe `sino` y la condición es falsa, 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. El listado 16 ilustra un ejemplo de sentencia si.

Listado 16. Ejemplo de sentencia si – entonces – sino

```
si (a < b) entonces:
      a = b;
sino:
      a = a + b;
      b = a;
fin si;
si (a<b) entonces:
      a =b;
      si a<c entonces:
            a =c;
      sino:
            c =a;
      fin si;
sino:
```

Sentencia de control de flujo iterativo `para`

La estructura sintáctica de una sentencia `para` es:

```
para indice en expresionComienzo .. expresionFinal :
    sentencias;
fin para;
```

Donde `indice` es un identificador de tipo entero 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 de 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 `para` 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 `para`.

Listado 18. Ejemplo de sentencia `para`

```
b=5;
para a en 1 .. b:
      escribir(a);
fin para;
```

Sentencia de salida

PL1UnedES dispone de un procedimiento predefinido que puede ser utilizado para emitir por la salida estándar (pantalla) resultados de diferentes tipos. Este procedimiento está implementado dentro del código del propio compilador, lo que implica que constituye una palabra reservada del lenguaje. El procedimiento se usa de la siguiente forma:

```
escribir(parámetro);
```

Este procedimiento puede mostrar por pantalla el valor de un parámetro, que puede ser un identificador o expresión, así como una constante literal de tipo cadena de texto (demarcada entre comillas dobles), seguido de un salto de línea. El ejemplo `escribir` (“Hola mundo”); mostrará el texto `Hola mundo`. `escribir` (x); mostrará el valor de la variable x . El

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 elementos de un vector o la llamada a una función.

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

Listado 19. Ejemplo llamadas a procedimientos

```
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 o sintáctico. 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 deben ser postergados al análisis sintáctico donde el contexto de análisis es mayor. En la medida de lo posible deben, en esta fase, detectarse el mayor número de errores. Son ejemplos de errores léxicos: literal mal construido, identificador mal construido, carácter no admitido, etc.

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

No se debe realizar una recuperación de errores. Así por ejemplo, si el compilador encuentra un carácter extraño en el código fuente, éste emitirá un mensaje de error y abortará el proceso de compilación. Del mismo modo, si se encuentra una construcción incorrecta desde el punto de vista sintáctico, se debe de abortar la 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 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 y Cup además de *seguir la estructura de directorios y clases que se proporcionará*. Más adelante se detallan estas herramientas.

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

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

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.

3.1 Trabajo a entregar

La entrega cubrirá únicamente la parte de análisis léxico y sintáctico. Es decir, sólo se pide que el compilador procese archivos fuente e identifique y escriba por pantalla los errores léxicos y sintácticos encontrados en el archivo fuente, así como que identifique programas correctos.

3.1.1 Análisis léxico

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

permite integrar el análisis léxico de JFlex, de forma que al finalizar esta etapa el compilador debe reconocer las sentencias del programa fuente y detectar posibles errores sintácticos, indicando por pantalla el número de línea en que ha ocurrido el error.

3.1.3 Comportamiento esperado del compilador

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

3.2 Fechas y forma de entrega

Las fechas límite para las diferentes entregas son las siguientes:

Febrero	16 de febrero de 2020
Septiembre	13 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. Nos remitimos al documento de normas de la asignatura, dónde viene explicada la normativa a aplicar. No obstante, recordamos que para superar la sesión el tutor comprobará que el alumno ha realizado el analizador léxico y está dando comienzo al analizador sintáctico. **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.**

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 (El modelo estará disponible en el curso virtual).

1. El analizador léxico

2. El analizador sintáctico

3. Conclusiones

4. Gramática

En este punto se ha de incluir un esquema con las producciones de la gramática generada

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

4 HERRAMIENTAS

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

4.1 JFlex

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

Web JFlex: http://jflex.de/

4.2 Cup

Esta herramienta permite especificar gramáticas formales facilitando el análisis sintáctico para obtener un analizador ascendente de tipo LALR.

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

4.3 Jaccie

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

4.4 Ant

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

5 AYUDA E INFORMACIÓN DE CONTACTO

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

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

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

Esto no significa que la práctica pueda hacerse en común, por tanto **no debe compartirse código**. La práctica se realiza de forma individual. Se utilizarán programas de detección de copias en el código fuente y, en caso de ser detectada, se suspenderá a los alumnos implicados en todas las convocatorias del presente curso.

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

Listado 20. Ejemplo de un programa en PL1UnedES

```
programa Ejemplo:
      constantes MAX = 5;
      tipos Mivector = vector [1..MAX] de entero;
      variables  i : entero;
            x : booleano;
            v1 : Mivector;
      subprogramas
      funcion EsMenorQueDos (a:entero): booleano:
            variables dos: entero;
                  result: entero;
                  esMayor : booleano;
            subprogramas
                  procedimiento Imprime (numero: entero):
                  comienzo
                        escribir("imprimiendo");
                        escribir (numero);
                  fin;
            comienzo
                  dos=2;
                  result=a+dos;
                  si result<2 entonces:
                        Imprime(result);
                        esMenor=cierto;
                  sino:
                        esMenor=falso;
```

```
comienzo
        v1[1] = 1;
        v1[2] = 2;
        v1[3] = 3;
        para i en 1..3:
                x = EsMenorQueDos(v1[i]);
        fin para;
fin.
```