Dicho nombre (parte izquierda del punto) es un **identificador** y debe seguir las siguientes reglas:

Tener una longitud máxima de 20 caracteres.
Es sensible a mayúsculas/minúsculas.
Empezar con una letra.
Tener solo letras y números.
No usar caracteres especiales.

Ejemplos:
Correctos
CalcularInflacion.LID
CALCULARINFLACION.LID
calcularinflacion.LID

Incorrectos
Calcular Inflacion porque lleva un blanco.
1CalcularInflacion.LID porque empieza con un dígito.
CalcularInflacion_.LID porque lleva un carácter especial.
_CalcularInflacion.LID porque no empieza con letra.

2.3.2 INDEPENDENCIA FÍSICA DEL COMPILADOR

Es necesario insistir mucho en este punto: se debe entregar el archivo *.jar listo para ser ejecutado en cualquier máquina y en cualquier carpeta sin depender de todo el ambiente de desarrollo de Netbeans. En aquella carpeta que el Tutor disponga, por ejemplo: **C:\REVISION**, debe bastar con incluir el archivo *.jar, los archivos de entrada de la revisión y el compilador entregado debe generar los archivos de salida ahí mismo.

Se recalca una vez más: “ahí mismo”, no es en la raíz de C:\, ni en la raíz de D:\ ni en el escritorio de MICROSOFT WINDOWS ni en una carpeta personal de trabajo, sino en la carpeta que el Tutor disponga. En otras palabras, se desea eliminar la dependencia física de su compilador hacia su máquina y que se pueda ejecutar en cualquier parte.

Antes de hacer las entregas de los instrumentos de evaluación (tareas y proyecto), debe de hacer la prueba de su programa y confirma que se ejecuta sin problemas de la manera que se ha solicitado .

Para lograr esto se tiene que implementar que el programa reciba como parámetro el nombre del archivo a compilar; este es un buen artículo que explica el pase de parámetros desde CMD a un programa Java:

https://personales.unican.es/corcuerp/java/Slides/CommandLineArguments.pdf

2.3.3 EJEMPLO DE USO DEL COMPILADOR

Se tiene el siguiente archivo de texto que realiza el cálculo de inflaciones, llamado:

C:\REVISION\CalcInflac.LID

y cuyo contenido es el siguiente (ver próxima página; el archivo se pondrá en el Moodle):

la manera de ejecutar el compilador LIBELULA es la siguiente:

C:\REVISION\> java –jar LIBELULA.jar CalcInflac.LID

Su compilador de LIBELULA debe abrir el archivo CalcInflac.LID y empezar a leerlo línea por línea; cada una de esas líneas debe ser analizada sintáctica y semánticamente, a fin de detectar errores.

2.3.4 **ARCHIVO DE ERRORES**

LIBELULA debe crear un nuevo archivo de salida de errores, con el mismo nombre solo que con el sufijo que dice “-errores” y extensión txt

Para el ejemplo citado, se generará el archivo: CalcInflac-errores.txt

Este archivo contendrá una copia del programa en LIBELULA, con las líneas debidamente enumeradas hasta un máximo de 99,999 líneas (con ceros a la izquierda); se puede asumir que nunca ningún programa LIBELULA superará ese límite de líneas (99,999 o sea una menos que cien mil).

Entonces, se vería así (ver la siguiente página):

2.4 SINTAXIS DE LIBELULA

Es importante destacar que el lenguaje de programación LIBELULA es un subconjunto del lenguaje de programación MODULA2, de tal manera que para cualquier duda que se tenga se puede consultar un manual técnico de MODULA2 (véase el punto 2.2.1), ya que los comandos son los mismos; en particular **son sensibles a mayúsculas y minúsculas.**

Para LIBELULA vamos a precisar las siguientes reglas específicas en los siguientes aspectos:

2.4.1 FORMATO DEL ARCHIVO

- Un archivo de código fuente en LIBELULA se puede escribir en cualquier editor, siempre y cuando se grabe el contenido como “texto sin formato” (en código ASCII). Las líneas de este archivo deben terminar con ENTER o RETURN.
- El archivo puede contener líneas en blanco o bien líneas con comandos de LIBELULA o MODULA2.
- El formato de la línea debe ser el siguiente y se debe validar de manera estricta, reportando errores cuando no se cumple alguno de estos puntos del formato.
- Una línea física no puede tener más de 100 columnas o caracteres.
- Una línea física contiene una sola línea lógica; o sea que no hay multilíneas.
- Una línea lógica puede tener uno o más comandos.
- El terminador de comandos es el punto y coma; hay excepciones que se expondrán más adelante.
- Si el comando es VAR, BEGIN, REPEAT, IF, ELSE no debe terminar con punto y coma y se puede asumir que será el único comando en la línea lógica.
- Si el comando es END (el del fin del programa) no debe terminar con punto y coma sino con punto.
- Las líneas en blanco en cualquier parte del archivo se ignoran en cuanto a su procesamiento, o sea que no hay que hacer nada con ellas, pero se debe respetar su presencia y mostrarlas en el archivo de errores.

- Una línea en blanco puede ser literalmente NULA (constituida por solo un ENTER) o tener uno o más espacios, como estas que vienen a continuación con 0, 1, 7 y 35 espacios respectivamente y en donde la fecha indica dónde terminan esos blancos:

  <--

  <--

  <--

  <--

- Decimos que el carácter blanco es el que corresponde a la espaciadora; se puede asumir que otros caracteres similares como tabuladores (TAB) o caracteres invisibles "que parecen blancos" no vendrán en el archivo.
- Los blancos o espacios “redundantes” antes, en el medio o al final de una línea se ignoran en cuanto a su procesamiento, o sea que no hay que hacer nada con ellos, pero se debe respetar su presencia y mostrarlos en el archivo de errores tal y como venían en el archivo de entrada. El alumno dispondrá, a su gusto, el procesamiento que dé a estos blancos redundantes, aplicando alguna de las técnicas que se explican en el material de la asignatura.

- Un blanco es “redundante” cuando se puede eliminar y aún así procesar la línea sin ambigüedades, por ejemplo a la línea siguiente (nótese los blancos redundantes al final que terminan donde está la flecha):

  Dato := (3.2 * (4.2 - 3.1 * 2.5) / 6.35); **<--**

  se le pueden eliminar los blancos redundantes y procesarla como si en realidad fuera así (nótese que se eliminaron los blancos al final de la línea):

  Dato:=(3.2*(4.2-3.1*2.5)/6.35);

- Se podrá asumir que los caracteres propios del idioma español como los signos de apertura de exclamación y de interrogación (¡, ¿), tildes, eñes, etc. no vendrán en el archivo ni siquiera en los comentarios (para no “meter ruido”).

2.4.2 COMENTARIOS

Se debe tener claro que los comentarios como tales son un comando también.

Los comentarios se indican con **(*** para el inicio y ***)** para el fin; debe validarse que en ambos casos los dos caracteres que conforman la apertura y el cierre vengan “pegados”; los comentarios pueden ser multilíneas, o sea abarcar varias líneas físicas hasta un máximo de 10 líneas. Si después de 10 líneas físicas no se ha cerrado un comentario entonces es error y las siguientes líneas a partir de esa décima se deben procesar como si fueran otro posible comando.

Los comentarios pueden aparecer en cualquier parte del archivo.

Se puede asumir, sin embargo, que siempre empezarán en una línea nueva.

Ejemplos:
Correctos:

```
(*
  ****************************************************************
  * Asumiendo tasas de inflacion anual de 7%, 8%, y 10%,         *
  * encontrar el factor por el cual cualquier moneda, tales como *
  * el franco, el dolar, la libra esterlina, el marco, el rublo, *
  * el yen o el florin han sido devaluadas en 1, 2, ...., N anos. *
  ****************************************************************
*)

   (* Inicio del programa *)
   WriteLn;

   (* Entrada de datos *)
```

```
Incorrectos:
        ( * Inicializacion de variables *) (porque hay un espacio entre ( y *
        (* Calculos y salida de datos      (porque falta el cierre)

        END CalcularInflacion. (* fin del programa *)
                                           (porque no empieza en una nueva línea)
```

2.4.3 ESTRUCTURA DEL ARCHIVO

Los archivos de código fuente en LIBELULA siempre tendrán la siguiente estructura sin excepción; no se debe asumir y se debe verificar; si no la cumplen se deben enviar los mensajes de error respectivos:

2.4.3.1 MODULE NOMBRE_PROGRAMA;

Todo programa LIBELULA debe empezar con la palabra reservada MODULE (solo debe aparecer una vez y debe ser una aparición válida), llevar un NOMBRE_PROGRAMA (que debe ser un identificador válido, ver el punto 2.5.5) y finalizar esa línea con punto y coma.

Antes de MODULE solo pueden aparecer líneas en blanco, comentarios o líneas con comandos de MODULA2 (ver el punto 2.4.3.5).

Se puede asumir que los 3 lexemas (**MODULE NOMBRE_PROGRAMA;)** vendrán en una misma línea lógica y no incluirán más comandos.

El NOMBRE_PROGRAMA no tiene nada que ver con el nombre del archivo que contiene el programa.

2.4.3.2 VAR

Sección donde se definen las variables que utiliza el programa; es opcional.

Más adelante (ver el punto 2.5.6) se explica cómo se declaran.

Empieza luego de que aparece MODULE.

Termina donde aparece BEGIN.

Fuera de esta sección no deben permitirse más declaraciones de variables.

Se puede asumir que aparte de VAR no vendrán otros comandos en la misma línea lógica.

2.4.3.3 BEGIN

Comando con el que se se inicia el programa; siempre debe ser el primero y solo debe aparecer una vez y debe ser una aparición válida; no se deben permitir otros comandos LIBELULA mientras no haya aparecido BEGIN. Nótese que antes de BEGIN sí pueden venir líneas en blanco, comentarios o comandos MODULA2.

2.4.3.4 SECCIÓN DE COMANDOS

Sección donde se indican los comandos LIBELULA que utiliza el programa; es obligatoria.
Empieza luego de BEGIN.
Termina con END NOMBRE_PROGRAMA. ← ojo al punto final.
El NOMBRE_PROGRAMA debe ser el mismo que se puso en el comando MODULE, literalmente en cuanto a mayúsculas y minúsculas.
No pueden indicarse comandos LIBELULA o MODULA2 después de END NOMBRE_PROGRAMA. aunque sí pueden aparecer líneas en blanco o comentarios.

2.4.3.5 PALABRAS RESERVADAS DE MODULA2

Cualquier comando que no se reconozca como válido de LIBELULA (pero sí de MODULA2) se ignorará, pero en el archivo de errores saldrá el mensaje:

Advertencia: comando no es soportado por esta versión.

Estas advertencias no se considerarán errores.

Ejemplo:

```
FROM InOut     IMPORT WriteInt, WriteString, WriteLn, Write, Read, ReadInt;
```

Una vez detectados estos comandos que LIBELULA no soporta el resto de la línea se debe ignorar y por lo tanto no compilar; la idea es permitir comandos no soportados por LIBELULA pero sí por MODULA2. Para poder implementar esto, se sugiere llevar un arreglo o vector de todas las palabras reservadas de MODULA2 (el alumno, si gusta, puede hacerlo de otra manera que juzgue conveniente; al final de este documento en el **ANEXO 1** se adjuntan todas las palabras reservadas de MODULA2). Cualquier otra cosa que no esté en esta lista y que no correspondan a comandos de LIBELULA deberá aparecer un mensaje como el siguiente:

ERROR: "xxxx" no es un comando válido de LIBELULA ni de MODULA2.

2.4.3.6 PALABRAS RESERVADAS DE LIBELULA

Al final de este documento en el **ANEXO 2** se adjuntan todas las palabras reservadas de LIBELULA, las cuales corresponden a los comandos explicados en este documento.

2.4.3.7 END NOMBRE_PROGRAMA. <-- ojo al punto final

Todo programa LIBELULA debe finalizar con la palabra reservada END, llevar el NOMBRE_PROGRAMA (que debe ser un identificador válido, ver el punto 2.5.5; debe ser el mismo que apareció en el comando MODULE) y terminar con un punto.

Estos tres lexemas solo deben aparecer una vez y debe ser una aparición válida.

Como ya se indicó, deben terminar con punto.

Se puede asumir que los 3 lexemas (END NOMBRE_PROGRAMA.**)** vendrán en una misma línea lógica.

No se deben permitir otros comandos después de que aparecen, aunque sí puede haber líneas en blanco o comentarios.

Se reitera que NOMBRE_PROGRAMA debe ser el mismo que se utilizó en MODULE; no es el nombre del archivo que contiene el código fuente en LIBELULA.

2.4.3.8 MAYÚSCULAS/MINÚSCULAS

LIBELULA es sensible a mayúsculas y minúsculas en cuanto a los elementos del lenguaje. Además, lo que vaya dentro de las comillas simples (que se usan como delimitadores de textos) se debe respetar literalmente.

Ejemplos:

REPEAT es válida y se refiere al comando REPEAT.

FACTOR, factor, FACtor, facTOR, FaCtOr son válidas y se refieren a diferentes identificadores.

2.4.3.9 VARIOS COMANDOS POR LÍNEA

LIBELULA permite varios comandos por línea lógica los cuales se separan con punto y coma, con las excepciones que ya se han explicado a lo largo de este documento.

Ejemplos:

Correctos:

```
(* Inicializacion de variables *)
Ano := 0;     Factor1 := 1.0;     Factor2 := 1.0;     Factor3 := 1.0;

(* Calculos y salida de datos *)
WriteLn; WriteString ( '   Ano 7%      8%     10%' );  WriteLn;

factor2 := ( factor2 ) * 1.08;
```

Incorrecto:

```
factor2 := ( factor2 )
           *
           1.08;
```

Porque un solo comando ocupa más de una línea lógica.

2.5 ELEMENTOS DEL LENGUAJE

Se exponen a continuación los principales elementos del lenguaje LIBELULA.

2.5.1 AGRUPACIÓN

Los caracteres de agrupación válidos para LIBELULA son:

(paréntesis izquierdo
) paréntesis derecho

Ejemplo:

```
factor3 := factor3 * ( 1.10 );
```

2.5.2 CONSTANTES

En LIBELULA hay cuatro tipos de constantes: caracter, texto, enteras y reales.

Las de caracter son de un solo caracter y van delimitadas por comillas simples.

Las de texto tienen hasta 60 caracteres y van delimitadas por comillas simples.

Lo que va dentro de ambas debe respetarse en cuanto a mayúsculas y minúsculas.

Se puede asumir, “para no meter ruido”, que no llevarán caracteres del español como las tildes.

Ejemplos:

Caracter:

‘S’

‘n’

‘7’

Texto:

‘HOLA‘

‘$25,000.00‘

‘Numero de empleados‘

Las numéricas pueden ser positivas o negativas, llevar decimales (el separador de decimales es el punto) y no deben llevar comas.

Hay 2 tipos de constantes numéricas:

- enteras: números enteros positivos o negativos que no tienen punto decimal y que están en el rango
  -32768 y +32767
  Ejemplos:
  1
  -1
  10
  -10

- reales: números reales positivos o negativos que sí tienen punto decimal y al menos un dígito decimal y que están en el rango -32768.00 y +32767.99
  Ejemplos:
  1.0
  -1.0
  10.23
  -10.23
  Nótese que 1. y 10. son incorrectas porque deben tener al menos un dígito decimal.

2.5.3 OPERADORES ARITMÉTICOS

Los siguientes son los operadores aritméticos que maneja LIBELULA:

Precedencia	Operador	Operación	Ejemplo	En LIBELULA
1	*	Multiplicación	XY/Z	X * Y / Z
1	/	División	X+Y/Z	X + (Y / Z)
2	+	Suma	X+Y/Z	(X + Y) / Z
2	-	Resta	X-Z/Y	(X – Y) / Z

Las operaciones entre paréntesis se ejecutan primero.
Nótese de los ejemplos anteriores y de sus equivalentes en LIBELULA el uso adecuado de los paréntesis para cambiar las reglas de precedencia de los operadores aritméticos; cuando dos de ellos tienen la misma precedencia entonces se hacen los cálculos de izquierda a derecha.

2.5.4. OPERADORES RELACIONALES

Los siguientes son los operadores relacionales que maneja LIBELULA.
Nótese que no deben llevar blancos entre los dos caracteres especiales (si es del caso):

```
=      Igual.
#      No igual o diferente.
<      Menor que.
>      Mayor que.
<=     Menor o igual.
>=     Mayor o igual.

Estos son incorrectos porque llevan espacios:    < =    > =

Ejemplo:
    IF ( MaxAnos <= 0 ) THEN
       RETURN;
    END;
```

2.5.5 IDENTIFICADORES

En LIBELULA los identificadores se definen siguiendo las siguientes reglas:

- Tener una longitud máxima de 20 caracteres.
- Son sensibles a mayúsculas y minúsculas.
- Empezar con una letra.
- Solo tener letras o números.
- No usar otros caracteres especiales.
- No corresponder a una palabra reservada de LIBELULA o de MODULA2.

Ejemplos:

Correctos

MaxAnos
ANO
factor1
respUESTA

Incorrectos

max-anos	porque lleva un carácter especial en el medio.
1factor	porque no empieza con letra.
Factor2%	porque lleva un carácter especial al final.
_no_se_usa	porque no empieza con letra.
REAL	porque es palabra reservada de LIBELULA.
TRUE	porque es palabra reservada de MODULA2.

2.5.6 DEFINICIÓN DE VARIABLES

En LIBELULA las variables se definen siguiendo la siguiente sintaxis:

identificadores (uno o más separados por coma) : TIPO-DE-DATOS

en dónde:

- TIPO-DE-DATOS puede ser INTEGER, REAL o CHAR;
- El caracter : (dos puntos) es obligatorio y debe aparecer solo una vez antes del TIPO-DE-DATOS.

INTEGER	se usará para almacenar números enteros;
REAL	se usará para almacenar números reales;
CHAR	se usará para almacenar un caracter;

Los identificadores de variables deben ser válidos conforme a las reglas indicadas anteriormente; si hay más de un identificador de variable se deben separar por comas; pero no deben quedar comas “guindando” (al principio, en el medio o al final); tampoco deben permitirse identificadores repetidos ni siquiera con tipos de datos diferentes.

Ejemplos:
Correctos

```
MaxAnos    : INTEGER;
Ano        : INTEGER;
Respuesta  : CHAR;
Factor1, Factor2, Factor3 : REAL;
```

Incorrectos

```
maxanos ano     : INTEGER;        porque falta la coma como separador de variables
,maxanos,  ano  : INTEGER;        porque hay una coma "guindando".
Max anos, ano, : INTEGER;         porque hay una coma "guindando".
FACTOR1, factor2, factor3 REAL;   porque falta el dos puntos.
1noseusa        : CHAR;           porque no empieza con letra.
```

2.5.7 CALIDAD DE LOS ERRORES

LIBELULA debe reportar al usuario los errores que detecta cuando analiza las hileras de caracteres que conforman el archivo fuente que corresponde al programa.

Los diversos mensajes de error deben tener este formato:

ERROR 999: texto del error.

Todos los errores que se vayan a manejar deben ser identificados por un código y un texto. La enumeración de los errores queda a criterio del estudiante. Los errores deben ser claros y concisos y referirse a solo una situación de error por vez, de manera que un texto como éste:

Variable no definida o de tipo inválido.

no es correcto pues son dos errores diferentes en un mismo mensaje.

El mensaje de error debe indicar entre paréntesis cuadrados (corchetes) cuál es el lexema que provoca el error.

Ejemplos:

factor1, factor2, factor3 : RREAL;

ERROR 025: Tipo de datos inválido [RREAL].

WriteInt (Anno, 3) ;

ERROR 027: variable no definida [Anno]

Los errores deben aparecer a partir de la columna 8 de la línea respectiva en el archivo de errores, de manera que así se facilite asociar, visualmente, los errores a la línea del programa LIBELULA; no olvidar que una sola línea puede tener varios errores y que por lo tanto todos se deben mostrar, de una vez, conforme a lo que se acaba de indicar.

También es importante recalcar, una vez más, que en el archivo de errores deben aparecer **todas las líneas del programa LIBELULA, tanto las válidas como las inválidas,** debidamente enumeradas a 5 dígitos (con ceros a la izquierda) y con los mensajes de error asociados (si los hay) abajo, todos de una vez, uno por línea, tal y como se acaba de explicar. Esto es **IMPORTANTÍSIMO** para la revisión de las tareas y de la entrega final del proyecto.

Ejemplo:

```
00001 MODULE CalcularInflacion;
00002
00003    FROM InOut      IMPORT WriteInt, WriteString, WriteLn, Write, Read,
ReadInt;
00004   FROM RealInOut IMPORT WriteReal, ReadReal;
00005
00006 VAR
00007   MaxAnos    : INTEGER;
00008   Ano        : INTEGER;
00009   Respuesta : CHAR;
00010   Factor1, Factor2, Factor3 : RREAL;
       ERROR 025: Tipo de datos inválido [RREAL].
00011
00012 BEGIN
00013
00014 (*
00015  ****************************************************************
00016  * Asumiendo tasas de inflacion anual de 7%, 8%, y 10%,         *
00017  * encontrar el factor por el cual cualquier moneda, tales como  *
00018  * el franco, el dolar, la libra esterlina, el marco, el rublo,  *
00019  * el yen o el florin han sido devaluadas en 1, 2, ...., N anos. *
00020  ****************************************************************
00021 *)
00022
00023     (* Inicio del programa *)
00024     WriteLn;
00025
00026     (* Entrada de datos *)
00027     WriteString ( 'Por favor, indique la cantidad maxima de anos:' );
00028     WriteLn;
00029     WriteLn;
00030
00031     ReadInt ( MaxAnos );
00032
00033     IF ( MaxAnos <= 0 ) THEN
00034        RETURN;
00035     END;
00036
00037     (* Inicializacion de variables *)
00038     Anno := 0;  Factor1 := 1.0;  Factor2 := 1.0; Factor3 := 1.0;
       ERROR 027: variable no definida [Anno]
00039
00040     (* Calculos y salida de datos *)
00041     WriteLn;  WriteString ( '   Ano 7%   8%  10%' );  WriteLn;
```

Nótese qué fácil es notar que hay errores pues las líneas enumeradas se ven interrumpidas por líneas en blanco, las cuales muestran el mensaje de error a la derecha; esto hace que el archivo de errores sea muy legible.

2.5.8 ASIGNACIÓN DE VARIABLES

La asignación de valores a las variables se hace mediante el operador **:=** o “dos puntos igual”. Ambos caracteres deben ir pegados sin blanco intermedio entre ellos.
Ver el comando ASIGNAR en el punto 2.6.3.

2.5.9 ETIQUETAS/GOTO

A diferencia de PASCAL y ADA, LIBELULA no usa etiquetas y en consecuencia tampoco el comando GOTO.
No se van a implementar. En este punto no tienes que hacer nada, es solo informativo.

2.6 COMANDOS

Se describen a continuación cada uno de los comandos que maneja LIBELULA.

2.6.1 Read

Sintaxis: **Read (Variable de tipo CHAR);**
ReadInt (Variable de tipo INTEGER);
ReadReal (Variable de tipo REAL);

Permite ingresar valores a las variables desde el teclado.
Se puede asumir que solo se pide una variable a la vez.
Los paréntesis son obligatorios.
La variable debe haber sido definida previamente.
El tipo de datos de la variable debe corresponder al comando que se invoca.

Ejemplos:

```
ReadInt ( MaxAnos );
Read ( Respuesta );
```

NOTA

Esta es una de las grandes ventajas técnicas de MODULA2, que tratamos de aprovechar en este proyecto de LIBELULA, a saber: se deben indicar expresamente las librerías que se van a utilizar, lo cual en estos tiempos es la norma, pero en aquellos era la novedad.
¿Qué quiere decir esto?
Que si, por ejemplo, tu programa no usa variable reales o flotantes, sino solo enteras, entonces puedes quitar esta línea y de esa manera el ejecutable que se obtiene es más pequeño y por lo tanto, en teoría, más eficiente:

```
FROM RealInOut IMPORT WriteReal, ReadReal;
```

COROLARIO: siempre deben aparecer al principio los comandos MODULA2 FROM e IMPORT.

2.6.2 WriteLn

Sintaxis: **WriteLn**

Permite indicar que se avance el cursor a la línea siguiente luego de haber imprimido algo en la pantalla.

Ejemplo:

```
WriteLn;
```

2.6.3 ASIGNAR

Sintaxis: **variable := expresión**

Para asignar el valor de una expresión a una variable.

La variable debe haber sido definida previamente.

Se puede asumir que la expresión siempre es correcta y no se debe validar en cuanto a correctitud de los paréntesis, uso de los operadores aritméticos o relacionales, posición de los operandos, etc.

Sin embargo, se debe analizar cada uno de los identificadores y palabras reservadas que aparecen en la expresión y confirmar que todos correspondan a:

- variables previamente definidas; o
- palabras reservadas de MODULA2.

Es importante recordar que el orden de los análisis en todo compilador es el siguiente:

- léxico;
- sintáctico:
- semántico.

En consecuencia, los errores se deben reportar en ese orden. Veamos.

Léxico: errores de lexemas esperados y que no se encuentran o que son inválidos:

ejemplo: QeadInt (MaxAnos); es error porque QeadInt no es un comando válido ni es palabra reservada de MODULA2.

Sintáctico: lexemas mal escritos según los patrones de definición:

ejemplo: ReadInt (1MaxAnos); es error porque el identificador empieza con número.

Semántico: lexemas bien escritos pero que no corresponden a objetos definidos:

ejemplo: ReadInt (MaxAnos); es error si MaxAnos no ha sido definida.

Se recalca que este es un ejemplo; el estudiante debe determinar en todo momento qué validaciones de los tres análisis debe efectuar conforme a este enunciado.

Además, en este comando ASIGNAR se debe validar que el tipo de datos de la variable izquierda coincida con el de todas las variables que se utilicen al lado derecho, bajo las siguientes reglas:

- si la variable a la izquierda es INTEGER o REAL entonces todas las variables a la derecha deben ser INTEGER o REAL;
- si la variable a la izquierda es CHAR entonces todas las variables a la derecha deben ser CHAR.

Ejemplos:

```
      Ano := 0;      Factor1 := 1.0;      Factor2 := 1.0;      Factor3 := 1.0;
...
      Factor1 := ( Factor1 * 1.07 );
      Factor2 := ( Factor2 ) * 1.08;
      factor3 := ABS ( factor3 * ( 1.10 ) );
                 En este caso, ABS corresponde a una palabra reservada de MODULA2.
```

2.6.4 Write

Sintaxis: **Write (Variable de tipo CHAR)**

WriteInt (Variable de tipo INTEGER, tamaño)

WriteReal (Variable de tipo REAL, tamaño)

WriteString ('Texto')

Para imprimir en pantalla una variable o un texto.

El par de paréntesis siempre es obligatorio.

La variable debe existir y ser del tipo esperado según la cantidad y tipo de lexemas que trae el comando dentro de los paréntesis.

El separador de lexemas es la coma; no debe haber comas “guindando” tal y como se explicó en el punto 2.5.6.

“Tamaño” debe ser una constante entera positiva entre 0 y 20 (véase el punto 2.5.2) y siempre debe aparecer dependiendo del tipo de variable que se va a imprimir.

El texto debe ir delimitado por comillas simples tanto al inicio como al final.

El texto no puede tener más caracteres que los que almacena una constante de tipo texto (véase punto 2.5.2).

Se puede asumir que el texto a imprimir no incluirá las comillas simples o dobles.

Ejemplos:

Correctos:

```
Write ( Respuesta );
WriteInt  ( Ano,      3 ) ;
WriteReal ( Factor1, 10 ) ;
WriteString ( '*** Fin del programa ***' );
```

Incorrectos:

```
WriteInt  ( Ano      3 ) ;  porque falta la coma.
WriteReal ( Factor1, 30 ) ; porque el tamaño es mayor que 20.
Write ( Respuestaa );       porque Respuestaa no está definida.
WriteString ( '*** Fin del programa *** );
                            porque falta la comilla de cierre.
```

2.6.5 RETURN

Sintaxis: **RETURN**

Termina la ejecución del programa y regresa el control al sistema operativo.

Ejemplo:

RETURN;

2.6.6 REPEAT

Sintaxis:

REPEAT

comandos

UNTIL condición

Para hacer un ciclo de la siguiente manera:

Los comandos dentro del REPEAT se ejecutan siempre al menos una vez.

Luego se evalúa la condición y si la misma es verdadera se vuelven a repetir los comandos.

Pero si es falsa termina el ciclo y continúa en el siguiente comando luego de UNTIL.

Los comandos dentro del ciclo pueden ser cualquiera de los ya citados o bien cualquier otro que sea propio de MODULA2 tal y como ya se explicó.

Sin embargo, no se permite REPEAT dentro de otro REPEAT (o sea anidados) tampoco IF dentro de REPEAT; se puede asumir que ambos casos no se darán.

Por otro lado, la condición como tal se puede asumir que siempre va a ser correcta y no se debe validar; sin embargo, se debe validar que todos los identificadores que aparezcan en la condición estén definidos (no importa el tipo) o bien que sean palabras reservadas de MODULA2.

El REPEAT nunca debe llevar punto y coma.

Ejemplo:

```
REPEAT
  Factor1 := ( Factor1 * 1.07 );
  Factor2 := ( Factor2 ) * 1.08;
  Factor3 := Factor3 * ( 1.10 );
  WriteInt  ( Ano,      3 ) ;
  WriteReal ( Factor1, 10 ) ;
  WriteReal ( Factor2, 10 ) ;
  WriteReal ( Factor3, 10 ) ;
  WriteLn;
  WriteLn;
  Ano := Ano + 1;
UNTIL Ano > MaxAnos;
```

2.6.7 IF

```
Sintaxis: IF ( condición ) THEN
             comandos-de-IF
          ELSE
             comandos-de-ELSE
          END
```

Para hacer una decisión de la siguiente manera:

si la (condición) es cierta se ejecutan los `comandos-de-IF`;
si la (condición) es falsa se ejecutan los `comandos-de-ELSE`.

Se debe validar que se cumpla esta sintaxis: IF (condición) THEN, o sea que venga el IF, que se abran y cierren los paréntesis que rodean la condición y que el THEN también aparezca; después del THEN nunca va punto y coma.

Por otro lado, la condición como tal se puede asumir que siempre va a ser correcta y no se debe validar; sin embargo, se debe validar que todos los identificadores que aparezcan

en la condición estén definidos (no importa el tipo) o bien que sean palabras reservadas de MODULA2.

Los `comandos-de-IF` pueden ser cualquiera de los citados en esta sección de comandos o bien cualquier otro que sea propio de MODULA2 tal y como ya se explicó.

Los `comandos-de-ELSE` pueden ser cualquiera de los citados en esta sección de comandos o bien cualquier otro que sea propio de MODULA2 tal y como ya se explicó.

IF es obligatorio; debe haber al menos un comando entre el IF y el ELSE o el END, según corresponda.
ELSE es opcional; si aparece debe haber al menos un comando entre el ELSE y el END.
El ELSE nunca debe llevar punto y coma.

Sin embargo, no se permiten IF dentro de otro IF (o sea anidados), tampoco REPEAT dentro de IF; se puede asumir que ambos casos no se darán.

END debe aparecer al final, solo una vez, luego del último comando dentro del IF o del ELSE según corresponda; END es obligatorio.

Ejemplo:

```
IF ( MaxAnos <= 0 ) THEN
   RETURN;
END;
```

2.6.8 GOTO

Se recuerda que, a diferencia de PASCAL, que es el antecesor, tanto LIBELULA como MODULA2 no manejan el comando GOTO; aquí no hay que hacer nada; es solo una aclaración.

2.6.9 NULL

Se recuerda que, a diferencia de ADA, que es otro antecesor, tanto LIBELULA como MODULA2 no manejan el comando NULL; aquí no hay que hacer nada; es solo una aclaración.

2.6.10 END NOMBRE_MODULO.

Indicador de que los comandos del programa terminaron. Siempre debe ser el último y solo debe aparecer una vez; no se deben permitir otros comandos después de que aparece, aunque, como ya se indicó, sí puede haber líneas en blanco o comentarios. Véase el punto 2.4.3.7 para la sintaxis exacta.

2.7 ARCHIVO PARA PRUEBAS

Como ya mostramos al principio, un ejemplo de programa en LIBELULA es el que aparece en el punto **2.3.3** y sobre el cual hemos basado todo este enunciado. El problema es que en la vida real los programas no vienen tan "bonitos" o sea bien indentados, claramente escritos y acomodados; así que a continuación se muestra el mismo programa pero "feo", en el sentido de que no es muy ordenado; es importante que las pruebas de su compilador la haga usando este programa "feo", ya que así serán los archivos de prueba que usará el Tutor. Tomar nota, de que las primeras líneas tienen desde 0 y hasta muchos caracteres en blanco, antes de que aparezca MODULE; los mismo las líneas intermedias y las finales luego de END NOMBRE_PROGRAMA.; además, hay blancos redundantes por todo lado: antes y después de lexemas; todo eso tu programa debe procesarlo, permitirlo e ignorarlo (según sea el caso conforme a lo expuesto en este enunciado) siempre y cuando no sean errores.

Ejemplo del archivo denominado “Feo”: