Practica 2.- Analizador Sintáctico de Morónico

TALF 2022/23

2 de mayo de 2023

Índice

1. Descripción de la práctica

Objetivo: El alumno deberá implementar un analizador sintáctico en Bison para el lenguaje de programación Morónico[1], inventado para la ocasión. Partiendo del analizador léxico escrito en la primera práctica, el programa resultante deberá recibir como argumento el path del fichero de entrada conteniendo el código fuente que se quiera analizar, y escribirá en la consola (o en un fichero) la lista de tokens encontrados en dicho fichero de entrada (omitiendo los comentarios) y las reglas reducidas. Estas ultimas se volcarán a la salida a medida que se vayan reduciendo durante el análisis sintáctico.

Se valorará reducir todo lo que se pueda los conflictos en la gramática (o eliminarlos completamente). También puntuará que, en caso de que la entrada contenga errores, el analizador prosiga el análisis hasta el final de la misma.

Documentación a presentar: El código fuente se enviará a través de Faitic. Para ello, primero se creará un directorio formado por los apellidos de los autores en orden alfabético, separados por un guión, y sin acentos ni eñes.

Ej.- DarribaBilbao-VilaresFerro

Dentro del directorio se copiará el código fuente de la práctica: los archivos de Flex y Bison y cualquier otro archivo fuente (Makefile incluído) que se haya usado. A continuación, el directorio se comprimirá en un archivo (tar, tgz, rar o zip) con su mismo nombre.

Ej.- DarribaBilbao-VilaresFerro.zip

[1] Un lenguaje de programación tan estúpido que hasta el nombre está mal escrito.

Grupos: Se podrá realizar individualmente o en grupos de dos personas.

Defensa: Consistirá en una demo al profesor, que calificará tanto los resultados como las respuestas a las preguntas que realice acerca de la implementación de la práctica.

Fecha de entrega y defensa: El código se subirá a Faitic como muy tarde el 8 de mayo a las 14:00. La defensa tendrá lugar en las horas de aula pequeña del martes 9 de mayo en adelante.

Material: Hemos dejado en Faitic (TALF → documentos y enlaces → material de prácticas → Práctica 2) un fichero (`moronico.tar.gz`). Dicho fichero contiene la especificación incompleta[2] en Flex del analizador léxico (`moronico.l`), la especificación incompleta[3] del analizador sintáctico en Bison (`moronico.y`, sólo con las reglas para definir un programa o paquete), dos ficheros de ejemplo (`ordenar.mor` y `formas_geometricas.mor`), y un fichero `Makefile`.

Nota máxima: 2'5 ptos. Se evaluará al alumno por las partes del analizador que se hayan hecho satisfactoriamente:

- 0'25 puntos por las declaraciones de programas, paquetes y cargas.
- 0'6 puntos por la declaración de tipos (incluyendo clases)
- 0'25 puntos por las declaraciones de constantes, variables e interfaces.
- 0'25 puntos por los subprogramas.
- 0'5 puntos por las instrucciones.
- 0'4 por las expresiones (0'2 si se emplean precedencias y asociatividades).
- 0'25 por la gestión de errores.

Para sumar 0'4 por las expresiones, es necesario haberlas implementado correctamente usando una gramática no ambigua. Si se usan precedencias y asociatividades, la nota máxima por ese apartado se reduce a 0'2 ptos.

Si el analizador presentado tiene conflictos, la nota de la práctica bajará 0'1 ptos por cada conflicto, hasta un máximo de 1'5 ptos.

2. Gramática del analizador sintáctico de Morónico

En esta sección vamos a proporcionar la especificación de la gramática de Morónico. Para ello, usaremos una notación similar a la BNF:

- Las palabras con caracteres en minúscula sin comillas denotan las categorías sintácticas (símbolos no terminales).
- Las secuencias de caracteres en mayúscula o entre comillas denotan las categorías léxicas (símbolos terminales). Por ejemplo, en:

  ```
  expresion_primaria ::= '(' expresion ')'
  ```

  los símbolos `expresion` y `expresion_primaria` son categorías sintácticas, mientras que '`(`' y '`)`' son categorías léxicas.
- Una barra vertical separa la parte derecha de dos reglas con el mismo símbolo cabeza. Por ejemplo:

  ```
  expresion_constante ::= CTC_ENTERA | CTC_REAL | CTC_CADENA | CTC_CARACTER | CTC_BOOLEANA
  ```

  que también puede escribirse como:

[2] En realidad, prácticamente vacía.
[3] De nuevo, prácticamente vacía.

```
expresion_constante ::= CTC_ENTERA
                      | CTC_REAL
                      | CTC_CADENA
                      | CTC_CARACTER
                      | CTC_BOOLEANA
```

son cinco reglas diferentes con el mismo símbolo cabeza, `expresion_constante`.

- Los corchetes especifican la repetición de símbolos (terminales o no terminales)[4] . Dichos símbolos pueden aparecer cero o más veces, en cuyo caso escribiremos '`[ ]*`', o de una o más, en cuyo caso escribimos '`[ ]+`'. De este modo, en la siguiente regla:

  ```
  bloque_instrucciones ::= '{' [ instruccion ]+ '}'
  ```

  tenemos que un bloque de intrucciones está formada por una llave de apertura '`{`', al menos una instrucción, y una llave de cierre '`}`'.

- Los paréntesis especifican la repetición de símbolos separados por comas[5]. Dichos símbolos pueden aparecer un cero o más veces, en cuyo caso escribiremos '`( )*`', o una o más veces, en cuyo caso escribimos '`( )+`'. De este modo, la siguiente regla:

  ```
  llamada_subprograma ::= nombre '(' ( expresion )* ')'
  ```

  especifica una llamada a subprograma está formada por un `nombre` seguido de una lista de cero o más expresiones, separadas por comas, entre dos paréntesis (de apertura y cierre).

- Los delimitadores '`[ ]?`' especifican los elementos opcionales. En la regla:

  ```
  seccion_cuerpo ::= CUERPO
                     [ declaracion_tipos ]?
                     [ declaracion_constantes ]?
                     [ declaracion_variables ]?
                     [ declaracion_subprograma ]+
  ```

  sólo la palabra reservada '`cuerpo`' y `declaracion_subprograma` son obligatorios, el resto de declaraciones son opcionales.

2.1. Especificación de programas, paquetes y cargas

Un fichero Morónico puede ser un programa o un paquete. En el primer caso, un programa está compuesto por la palabra reservada '`programa`' seguida de un `nombre`, un punto y coma, y un bloque de programa.

```
programa ::= definicion_programa
           | definicion_paquete

definicion_programa ::= PROGRAMA nombre ';' bloque_programa

nombre ::= [ IDENTIFICADOR '::' ]* IDENTIFICADOR

bloque_programa ::= [ declaracion_cargas ]?
                    [ declaracion_tipos ]?
                    [ declaracion_constantes ]?
                    [ declaracion_variables ]?
                    bloque_instrucciones

bloque_instrucciones ::= '{' [ instruccion ]+ '}'
```

[4]Excepto cuando aparecen entre comillas. En ese caso son categorías léxicas.
[5]Excepto cuando aparecen entre comillas.

A su vez, un `nombre` está formado por uno o más identificadores separados por '`::`', mientras que un bloque de programa está formado por cero o una subsecciones de declaracion de cargas, tipos, constantes y variables, y un bloque de instrucciones.

Por otra parte, un paquete Morónico está formado por la palabra reservada '`paquete`' seguida de un `nombre`, punto y coma, secciones obligatorias de cabecera y cuerpo.

La sección de cabecera de un paquete Morónico está formada declaraciones opcionales de tipos, constantes y variables, y una o más declaraciones de subprogramas, mientras que la sección de cuerpo está formada por declaraciones opcionales de tipos, constantes, variables, y una o más declaraciones de subprogramas. La idea detrás de esta división es que la sección de cabecera contendrá declaraciones de objetos (tipos, constantes, variables, clases e interfaces de funciones) visibles para otros paquetes, mientras que el cuerpo del paquete se usará para implementar subprogramas (incluyendo los métodos de los objetos declarados previamente) así como las estructuras de datos necesarias para esos subprogramas que no queremos exportar fuera de nuestro paquete.

```
definicion_paquete ::= PAQUETE nombre ';' seccion_cabecera seccion_cuerpo

seccion_cabecera ::= CABECERA
                     [ declaracion_cargas ]?
                     [ declaracion_tipos ]?
                     [ declaracion_constantes ]?
                     [ declaracion_variables ]?
                     [ declaracion_interfaces ]?

seccion_cuerpo ::= CUERPO
                   [ declaracion_tipos ]?
                   [ declaracion_constantes ]?
                   [ declaracion_variables ]?
                   [ declaracion_subprograma ]+
```

Las cargas se utilizan, como su nombre indica, para cargar elementos (subprogramas, tipos, constantes, etc) implementados en otros paquetes. En el siguiente ejemplo:

```
  CARGA formas_geometricas (circulo,cuadrado), solidos en "/home/ejemplos/";
```

estamos cargando dos definiciones (en este caso de clases) llamadas '`circulo`' y '`cuadrado`' del paquete previamente existente '`formas_geometricas`', situado en el fichero "`formas_geometricas.mor`"de alguno de los directorios del path de Morónico, y todas las definiciones exportables del paquete `solidos`, sito en '`/home/ejemplos/solidos.mor`'.

Para implementar esta subsección definimos la declaración de cargas como la palabra reservada '`carga`' seguida de una o mas declaraciones de carga separadas por comas. Dichas declaraciones de cargas están, a su vez, formadas por un `nombre` seguido por dos construcciones opcionales: a) la palabra reservada '`en`' y un path, y b) una lista de uno o más nombres de objetos cargados, entre paréntesis y separados por comas.

```
declaracion_cargas ::= CARGA ( declaracion_carga )+ ';'

declaracion_carga ::= nombre [ EN PATH ]? [ '(' ( nombre )+ ')' ]?
```

2.2. Declaración de tipos

La subsección de declaración de tipos comienza con la palabra reservada '`tipo`', seguida de una o más declaraciones de tipos. A su vez, una declaración de tipo está formada por un `nombre` seguido del operador '`=`' y la definición del tipo. Si el tipo en cuestión es un tipo no estructurado o un nombre de tipo, agregamos el delimitador '`;`', lo que no es necesario si se trata de un tipo estructurado (como veremos, los tipos estructurados usan '`{ }`' como delimitadores).

```
declaracion_tipos ::= TIPO [ declaracion_tipo ]+

declaracion_tipo ::= nombre '=' tipo_no_estructurado_o_nombre_tipo ';'
                   | nombre '=' tipo_estructurado
```

El meollo de esta subsección está en la declaraciones de los posibles tipos. Para empezar, como ya hemos mencionado más arriba, un tipo puede ser un tipo no estructurado, un nombre de tipo o un tipo estructurado. Vamos a agrupar los dos primeros porque tienen usos comunes, pudiendo ser `tipo_no_estructurado_o_nombre_tipo` un `nombre` (de tipo), o un tipo escalar, fichero, enumerado, lista, lista asociativa o conjunto. Por su parte, un tipo estructurado puede ser un tipo registro o una declaración de clase (debido a su complejidad, esta última la veremos en una sección posterior de este documento).

```
tipo_no_estructurado_o_nombre_tipo ::= nombre
                                     | tipo_escalar
                                     | tipo_fichero
                                     | tipo_enumerado
                                     | tipo_lista
                                     | tipo_lista_asociativa
                                     | tipo_conjunto

tipo_estructurado ::= tipo_registro | declaracion_clase
```

En el resto de la sección vamos a detallar la declaración de los diferentes tipos que acabamos de enumerar, a excepción de las clases, que examinaremos posteriormente.

Tipos Escalares y Fichero Un `tipo_escalar` puede ser uno de los siguientes tipos predefinidos: '`entero`', '`real`', '`booleano`', '`caracter`' o '`cadena`'. También tenemos un tipo '`fichero`' para especificar descriptores de fichero (el resultado de abrir un fichero), que podremos usar para leer o escribir datos en un fichero de texto o binario previamente abierto.

```
tipo_escalar ::= ENTERO | REAL | BOOLEANO | CARACTER | CADENA

tipo_fichero ::= FICHERO
```

Tipo enumerado está formado por la enumeración de elementos pertenecientes al tipo, que serán expresiones constantes[6], separadas por comas y encerradas entre paréntesis.

```
tipo_enumerado ::= '(' ( expresion_constante )+ ')'
```

Por ejemplo, si quisiéramos declarar un tipo '`dias_laborables`', compuesto por los nombres de los días laborables de la semana, haríamos algo parecido a esto.

```
  dias_laborables = ("lunes","martes","miércoles","jueves","viernes");
```

Listas y listas asociativas La definición de un tipo lista estará definido por la palabra reservada '`lista`' seguida de las dimensiones de la misma (rangos de expresiones separados por comas y encerrados entre corchetes), la palabra reservada '`de`' y un nombre de tipo o tipo no estructurado.

```
tipo_lista ::= LISTA [ '[' ( rango )+ ']' ]? DE tipo_no_estructurado_o_nombre_tipo

rango ::= expresion '..' expresion [ '..' expresion ]?
```

Un rango se especifica escribiendo los límites inferior y superior de los elementos del tipo, separados por '`..`'. Opcionalmente, se podrá definir el incremento que separa a elementos consecutivos del rango, escribiendo de nuevo '`..`' y la cuantía de dicho incremento. Los rangos establecen los posibles índices de las dimensiones del array. Si se omiten, se considerará que el array es unidimensional con indices en el rango '`0..65535`'. En el siguiente ejemplo:

```
  tridim = lista [1..100, 2..200..2, -50..50] de entero;
```

[6] Veremos como se escribe una `expresion_constante` en la sección de expresiones de este documento.

el tipo 'tridim' es un array de enteros de tres dimensiones de 100 elementos cada una, la primera de 1 a 100, la segunda de 2 a 200 contando de 2 en 2 (2,4,6,...,200), y la tercera, de -50 a +50.

Por su parte, una lista asociativa se define como un array unidimensional en el que los índices son cadenas de caracteres. Por lo tanto, sólo es necesario especificar el tipo de los elementos de la lista.

```
tipo_lista_asociativa ::= LISTA ASOCIATIVA DE tipo_no_estructurado_o_nombre_tipo
```

Si queremos tener listas asociativas de más dimensiones, podemos anidar el tipo tantas veces como sea necesario. Por ejemplo, para tener una lista asociativa bidimensional de enteros, escribiríamos:

```
lista_anidada = lista asociativa de lista asociativa de entero;
```

Conjuntos El tipo conjunto se define aplicando mediante las palabras reservadas 'conjunto' y 'de' escritas antes de un nombre de tipo o tipo no estructurado.

```
tipo_conjunto = CONJUNTO DE tipo_no_estructurado_o_nombre_tipo
```

Esto nos permite definir una lista de elementos sin repetición, pertenecientes a un mismo tipo. Por ejemplo, si queremos definir conjuntos formados por dias de la semana laborables, podríamos recurrir a la siguiente definición:

```
conjunto_dias_laborables = conjunto de dias_laborables;
```

Registros Los registros son tipos estructurados, pero, a diferencia de las clases no tienen asociados métodos. Se definen de la siguiente manera.

```
tipo_registro ::= REGISTRO '{' [ declaracion_campo ]+ '}'

declaracion_campo ::= ( nombre )+ ':' tipo_no_estructurado_o_nombre_tipo ';'
```

El tipo se define mendiante la palabra reservada 'registro' seguida de una o más declaraciones de campos entre llaves. Cada declaración de campo está formada por uno o más nombres, separados por comas, seguidos por el delimitador ':' y un nombre de tipo o tipo no estructurado, con ';' finalizando la declaración. Por ejemplo:

```
datos_personales = registro { nombre,direccion : cadena; edad,peso : entero; altura : real; }
```

2.3. Declaración de constantes, variables e interfaces

La subsección de constantes comienza con la palabra reservada 'constante', seguida de una o más declaraciones de constantes. Dichas declaraciones relacionan un nombre de constante con un tipo no estructurado o nombre de tipo (que puede ser el de un tipo estructurado), y el valor de dicha constante (o valores, si el tipo en cuestión es una lista, conjunto o tipo estructurado). El nombre está separado del tipo por ':' y el tipo esta separado del valor constante por '='. El delimitador ';' se utiliza para terminar la declaración de la constante.

```
declaracion_constantes ::= CONSTANTE [ declaracion_constante ]+

declaracion_constante ::= nombre ':' tipo_no_estructurado_o_nombre_tipo '=' valor_constante ';'

valor_constante ::= expresion
                  | '[' ( valor_constante )+ ']'
                  | '[' ( clave_valor )+ ']'
                  | '[' ( campo_valor )+ ']'

clave_valor ::= CTC_CADENA '=>' valor_constante

campo_valor ::= nombre '=>' valor_constante
```

Por su parte, un valor constante puede ser una expresión o una lista con elementos separados por comas y delimitados por '`[ ]`'. Dichos elementos pueden ser valores constantes (esto permite definir listas anidadas de constantes), pares clave+valor (para listas asociativas) o pares campo+valor (para registros). El primero de estos tipos de pares está formado por una constante cadena, seguida de la flecha doble '`=>`' y de un valor constante, mientras que un par campo+valor está formado por un `nombre` (de campo), '`=>`' y un valor constante. Por ejemplo:

```
CONSTANTE
  dias_semana : lista [1..7] de cadena = ["lunes","martes","miércoles","jueves,"viernes",
                                          "sabado", "domingo"];
  datos_pepe : datos_personales = [nombre => "pepe" , direccion => "", edad => 24,
                                   peso => 80; altura => 1.80];
```

La definición de variables es similar a la de constantes. Primero tenemos la palabra reservada '`variable`' seguida de una o más declaraciones de variables. Cada variable se define especificando su nombre, seguido de '`:`', el tipo de la variable (un nombre de tipo o la definición de un tipo no estructurado), y opcionalmente, un '`=`' seguido del valor inicial de la variable. Varias variables del mismo tipo pueden especificarse juntas, separando sus nombres con comas, aunque en ese caso no se podrán especificar sus valores iniciales (no es necesario que implementeis esta ultima restricción).

```
declaracion_variables ::= VARIABLE [ declaracion_variable ]+

declaracion_variable ::= ( nombre )+
                         ':'
                         tipo_no_estructurado_o_nombre_tipo
                         [ '=' valor_constante ]?
                         ';'
```

Finalmente, la declaración de interfaces comienza con la palabra clave '`interfaz`', seguida de una o más cabeceras de subprogramas, con '`;`' como delimitador. Aunque veremos como se define un subprograma en una sección posterior, os adelanto que la cabecera de un subprograma está formada por el tipo de subprograma, nombre del mismo, argumentos y, opcionalmente, el tipo del valor de retorno.

```
declaracion_interfaces ::= INTERFAZ [ cabecera_subprograma ';' ]+
```

2.4. Declaración de clases

Aunque las clases se declaran junto al resto de los tipos, hemos dejado la explicación de las mismas hasta haber visto las declaraciones de constantes, variables e interfaces, dado que una clase puede tener sus propias declaraciones de tipos y constantes.

Una declaración de clase está compuesta por los siguientes símbolos: la palabra reservada '`clase`', la palabra reservada '`final`' (que es opcional e indica que la clase no puede tener subclases), una lista de uno o más nombres entre paréntesis (que de nuevo, es opcional, e indica los nombres de las clases de las que hereda la clase actual), y las declaraciones de de componentes de la clase, públicas, semipúblicas y privadas entre llaves '`{}`'.

```
declaracion_clase ::=  CLASE
                       [ FINAL ]?
                       [ '(' ( nombre )+ ')' ]?
                       '{'
                       declaraciones_publicas
                       [ declaraciones_semi ]?
                       [ declaraciones_privadas ]?
                       '}'

declaraciones_publicas ::= [ PUBLICO ]? [ declaracion_componente ]+

declaraciones_semi ::= SEMIPUBLICO [ declaracion_componente ]+

declaraciones_privadas ::= PRIVADO [ declaracion_componente ]+
```

A su vez, las declaraciones publicas, que son las únicas obligatorias, están precedidas de forma opcional por la palabra reservada '`publico`', mientras que las declaraciones de componentes semipúblicos y privados están precedidas, respectivamente, por las palabras reservadas '`semipublico`' y '`privado`'.

Una declaración de componente puede ser una declaración anidada de tipo, de constante de atributos y la cabecera (es decir, el interfaz) de un subprograma. Una declaración anidada de tipo (respectivamente constante) está formada por la palabra reservada '`tipo`' (respectivamente '`constante`') seguida de la consiguiente declaración, tal y como se ha descrito en las secciones anteriores.

```
declaracion_componente ::= declaracion_tipo_anidado
                         | declaracion_constante_anidada
                         | declaracion_atributos
                         | cabecera_subprograma ';' [ modificadores ';' ]?

declaracion_tipo_anidado ::= TIPO declaracion_tipo

declaracion_constante_anidada ::= CONSTANTE declaracion_constante

declaracion_atributos ::= ( nombre )+ ':' tipo_no_estructurado_o_nombre_tipo ';'

modificadores ::= ( modificador )+

modificador ::= GENERICO | ABSTRACTO | ESPECIFICO | FINAL
```

Por su parte, una declaración de atributos está formada por uno o más nombres de atributos, separados por comas, seguidos de ':' y la especificación de un tipo no estructurado (o nombre de un tipo definido), con un ';' al final, mientras que una cabecera de subprograma puede estar seguida, opcionalmente, por una lista de modificadores. Los modificadores pueden ser:

- '`generico`' define un método que debe ser redefinido por métodos con el mismo nombre de las subclases de la clase actual.
- '`abstracto`' define un método cuyo código no se escribe para la clase actual, teniendo que ser obligatoriamente implementado en sus subclases.
- '`final`' define un método que no puede ser redefinido por otro método con el mismo nombre en ninguna de las subclases de la clase actual.
- '`especifico`' redefine un método genérico o abstracto de una superclase de la clase actual.

Por ejemplo, si queremos definir una superclase figura geométrica de un determinado color y material, y, a partir de ella, las clases cubo, esfera, cono, con sus dimensiones y volumenes, podríamos hacerlo de la siguiente manera.

```
TIPO
  forma = CLASE
  {
    PUBLICO
      constructor crear (material : tipos_material; color : colores);
      funcion volumen => real; abstracto;
      destructor destruir;
    PRIVADO
      material : tipos_material;
      color : colores;
  }

  esfera = CLASE (forma)  // hereda los atributos y metodos de forma
  {
    PUBLICO
      constructor crear (radio : real; material : tipos_material; color : colores);  final;
```

```
    funcion volumen => real; especifico, final;
    destructor destruir;
  PRIVADO
    radio : real;
}

cubo = CLASE (forma)  // hereda los atributos y metodos de forma
{
  PUBLICO
    constructor crear (lado : real; material : tipos_material; color : colores);  final;
    funcion volumen => real; especifico, final;
    destructor destruir;
  PRIVADO
    lado : real;
}

cono = CLASE (forma)  // hereda los atributos y metodos de forma
{
  PUBLICO
    constructor crear (radio,altura: real; material : tipos_material; color : colores); final;
    funcion volumen => real; especifico, final;
    destructor destruir;
  PRIVADO
    radio,altura : real;
}
```

Los métodos se implementarían en la sección del cuerpo del paquete.

2.5. Subprogramas

Cuando empezamos a describir los componentes de un paquete Morónico, dijimos que el cuerpo del paquete termina con la declaración de uno o más subprogramas. En esta sección, vamos a ver como implementar una gramática para los subprogramas de Morónico.

En primer lugar, cuando declaramos un subprograma, tenemos dos partes diferenciadas, la cabecera (o interfaz) del subprograma y el bloque del subprograma (la cabecera era lo que escribíamos en la subsección de interfaces de la cabecera de un paquete o en la definición de un método en una clase). El formato de la cabecera del subprograma comienza con la palabra clave que especifica el tipo de subprograma que estemos escribiendo ('`función`', '`procedimiento`', '`constructor`' o '`destructor`') seguida de un `nombre` (del subprograma en cuestión). A continuación tendremos la declaración de parámetros (detalles más abajo), excepto en el caso de los destructores, que no tienen parámetros. Las funciones se diferencian de los procedimientos y los constructores en que en ellas tenemos que especificar el tipo del valor de retorno, lo que hacemos con la secuencia '`=>`' seguida del tipo no estructurado o nombre de tipo en cuestión (los procedimientos no tienen valor de retorno y en los constructores es un objeto de la clase del constructor).

```
declaracion_subprograma ::= cabecera_subprograma bloque_subprograma

cabecera_subprograma ::= cabecera_funcion
                       | cabecera_procedimiento
                       | cabecera_constructor
                       | cabecera_destructor

cabecera_funcion ::= FUNCION
                     nombre
                     [ declaracion_parametros ]?
                     '=>'
                     tipo_no_estructurado_o_nombre_tipo
```

```
cabecera_procedimiento ::= PROCEDIMIENTO nombre [ declaracion_parametros ]?

cabecera_constructor ::= CONSTRUCTOR nombre [ declaracion_parametros ]?

cabecera_destructor ::= DESTRUCTOR nombre
```

Los parámetros de una función, procedimiento o constructor son opcionales. De haberlos, aparecerán entre paréntesis, y separados por el delimitador ';'. En cada declaración de parámetros tenemos uno o más nombres (de los parámetros que se están definiendo) separados por comas, seguidos de ':', el tipo de los parámetros (que puede ser un nombre de tipo o tipo no estructurado) y, opcionalmente, la palabra reservada '`modificable`'. Dicha palabra indica que, si el contenido del parámetro se modifica en el subprograma, el nuevo valor se conservará a la salida del mismo (es decir, el parámetro se pasa por referencia). Si no se adjunta, los cambios realizados en valor del parámetro durante la ejecución del programa se pierden a la salida del mismo (el parámetro se pasa por valor).

```
declaracion_parametros ::= '(' lista_parametros_formales ')'

lista_parametros_formales ::= parametros_formales
                            | lista_parametros_formales ';' parametros_formales

parametros_formales ::= ( nombre )+ ':' tipo_no_estructurado_o_nombre_tipo [ MODIFICABLE ]?
```

Respecto al bloque de programa, está formado por subsecciones opcionales de declaraciones de tipos, constantes y variables, en ese orden, que nos permiten especificar estructuras de datos internas al procedimiento, seguidas de un bloque de instrucciones (una o más instrucciones entre llaves '`{ }`').

```
bloque_subprograma ::= [ declaracion_tipos ]?
                       [ declaracion_constantes ]?
                       [ declaracion_variables ]?
                       bloque_instrucciones
```

Por ejemplo, si queremos implementar los métodos de cálculo de volumen en las tres formas geométricas descritas en el apartado anterior, lo haríamos de la siguiente manera:

```
CUERPO

  funcion esfera::volumen => real
  {
    devolver (4/3*PI*actual.radio**3);
  }

  funcion cubo::volumen => real
  {
    devolver (actual.lado**3);
  }

  funcion cono::volumen => real
  {
    devolver (PI*actual.radio**2*actual.altura)/3;
  }
```

2.6. Instrucciones

Vamos a considerar los siguientes tipos de intrucciones.

```
instruccion ::= ';'
              | instruccion_asignacion
              | instruccion_salir
              | instruccion_devolver
              | instruccion_llamada
              | instruccion_si
              | instruccion_casos
              | instruccion_bucle
              | instruccion_probar_excepto
              | instruccion_lanzar
```

La primera es la instrucción vacia, que, como su nombre indica, no ejecuta ninguna acción. Detallamos las siguientes a continuación.

Instrucción de asignación. Asocia el valor resultante de una expresión a una variable, elemento de una tabla o campo de un registro (las reglas que tienen al no terminal `objeto` como cabeza se verán en la sección dedicada a expresiones).

```
instruccion_asignacion ::= objeto '=' expresion ';'
```

Instrucción salir. Se usa para terminar la ejecución de un bucle inmediatamente. En su forma más simple está formada únicamente por la palabra reservada '`salir`', que puede estar acompañada, opcionalmente, por una condición que tiene que cumplirse para efectuar la salida del bucle, y que está señalada por la palabra reservada '`si`'.

```
instruccion_salir ::= SALIR [ SI expresion ]? ';'
```

Instrucción devolver. Especifica la salida inmediata de un subprograma. En el caso de funciones también establece el valor de retorno, a través de una `expresion` opcional.

```
instruccion_devolver ::= DEVOLVER [ expresion ]? ';'
```

Instrucción llamada a subprograma. Esta formada por el nombre del subprograma seguido de los parámetros del mismo entre paréntesis, especificados como expresiones separadas por comas. Si no hay parámetros de entrada, los paréntesis estarán vacíos.

```
instruccion_llamada ::= llamada_subprograma ';'

llamada_subprograma ::= nombre '(' ( expresion )* ')'
```

Instrucción si...entonces...sino. Especifica la instrucción de bifurcación `if..then..else` de toda la vida. Si la `expresion` a continuación del '`si`' es booleana y cierta (cosa que vosotros no teneis que comprobar), se ejecutan las instrucciones asociadas al '`entonces`'. Si la condición es falsa, se ejecutarán las instrucciones a continuación del '`sino`', si este ha sido escrito.

```
instruccion_si ::= SI expresion
                   ENTONCES bloque_instrucciones
                   [ SINO bloque_instrucciones ]?
```

Instrucción casos. Es similar al `switch...case` de C. En primer lugar, tras la palabras reservadas '`en caso`', se especifica la expresión cuyo valor será evaluado y la palabra reservada '`sea`'. A continuación, se enumerarán los posibles valores que puede tomar la `expresion` (especificados mediante el símbolo no terminal `entradas`). Para cada uno de ellos, se escribe, después del operador '`=>`', el bloque de instrucciones que será ejecutado si la expresión tiene el valor correspondiente.

```
instruccion_casos ::= EN CASO expresion SEA [ caso ]+ ';'

caso ::= entradas '=>' bloque_instrucciones

entradas ::= [ entrada '|' ]* entrada

entrada ::= expresion | rango | OTRO
```

Los posibles valores de `entradas` que puede tomar la `expresion` en `instruccion_casos` pueden ser especificados a través de una o varias expresiones, separadas por el operador '`|`' (análogo a la disyunción lógica, pero específico para esta instrucción), por un rango de valores (`expresion '..' expresion`), o usando la palabra reservada '`otros`'. Este último caso será elegido cuando la expresión de partida no corresponda a ninguno de los valores especificados en el resto de casos. Por ejemplo:

```
en caso hoy sea
  "lunes" .. "jueves"   => { trabajo(); }           // de lunes a juevas
  "viernes" | "sabado"  => { trabajo(); fiesta(); } // viernes o sabado
  otros                 => { ; }                    // domingo
;
```

Bucles. La sintaxis de un bucle en Morónico está formada por una cláusula de iteración, especificando el tipo de bucle y la variable índice del mismo, y un bloque de instrucciones especificando el código que se ejecutará en cada iteración.

```
instruccion_bucle ::= clausula_iteracion bloque_instrucciones

clausula_iteracion ::= PARA nombre EN objeto
                     | REPITE ELEMENTO nombre EN rango [ DESCENDENTE ]?
                     | MIENTRAS expresion
                     | REPITE HASTA expresion
```

Hay cuatro tipos de bucle en Morónico. El más simple es '`mientras`', para el cual sólo hay que especificar la condición, en forma de `expresion`, que tiene que cumplirse para continuar en el bucle. El bucle '`repite hasta`' funciona de (casi) de la misma manera, excepto que cuando la `expresion` es cierta, se sale del bucle. Por su parte, el bucle '`para`' es similar al `foreach` presente en algunos lenguajes de programación. El programador especifica una variable que, en cada iteración del bucle, tomará el valor de uno de los elementos de una tabla, especificada como un `objeto`. Por ejemplo:

```
tabla : lista [1..25] de entero;
valor : entero;
...
para valor en tabla {
  escribir(valor*valor);
  }
```

También tenemos un bucle '`repite elemento`', similar al bucle `for` de muchos lenguajes de programación. En su sintaxis se establece primero la variable índice que se incrementará o decrementará, como un `nombre`, seguida de la palabra reservada '`en`', y el rango de valores que se van a recorrer. Estos últimos se establecen especificando el primer valor, seguido del delimitador '`..`' y el ultimo valor del rango. Opcionalmente, se puede especificar si el rango es descendente, usando la palabra reservada '`descendente`' antes de especificar el primer valor del rango, y

el incremento/decremento dentro del mismo, escribiendo de nuevo el delimitador '..' después del último valor. Por ejemplo:

```
tabla : lista [1..25] de entero;
i : entero;
...
repite elemento i en  n..1..5 descendente { // de n a 1 saltando de 5 en 5
  tabla[i] = aleatorio(1,100);
  }
```

Instruccion probar...excepto. Es la instrucción equivalente a try...catch en java y otros lenguajes orientados a objetos. Primero se escribe la palabra reservada '`probar`' seguida del bloque de instrucciones que se ejecuta sin incidente si no se lanza ninguna excepción[7]. A continuación, se escribe la palabra reservada '`excepto`' seguida de una o más cláusulas de excepción, y, opcionalmente, la palabra reservada '`finalmente`' seguida de un nuevo bloque de instrucciones. La idea es que si se lanza una excepción se ejecutará el código asociado a la cláusula de excepción correspondiente, y, de haberlo, el código asociado a la cláusula '`finalmente`'.

```
instruccion_probar_excepto ::= PROBAR bloque_instrucciones
                               EXCEPTO [ clausula_excepcion ]+
                               [ FINALMENTE bloque_instrucciones ]?

clausula_excepcion ::= CUANDO nombre EJECUTA bloque_instrucciones
```

Por su parte, una clausula de excepción está formada por la palabra reservada '`cuando`' seguida de un `nombre` (el de la excepción correspondiente), la palabra reservada '`ejecuta`' y un bloque de instrucciones.

Instrucción lanzar. Se usa para lanzar una excepción, por lo que está compuesta de la palabra reservada '`lanzar`' seguida de un `nombre` (de la excepción) y ';'.

```
instruccion_lanzar ::= LANZAR nombre ';'
```

2.7. Expresiones

Al contrario que en las secciones previas, vamos a describir como se calculan las expresiones de abajo a arriba, partiendo de los operandos hasta llegar al simbolo `expresion`, que será el símbolo raíz de esta parte de la gramática. En primer lugar, un objeto puede ser un nombre, elemento de una tabla, o campo de registro, mientras que una expresión constante puede ser una constante numérica, cadena, carácter o booleana.

```
objeto ::= nombre
         | objeto '[' ( expresion )+ ']'
         | objeto '.' IDENTIFICADOR

expresion_constante ::= CTC_ENTERA | CTC_REAL | CTC_CADENA | CTC_CARACTER | CTC_BOOLEANA
```

De este modo una expresión primaria podrá ser un objeto, una expresión constante, una llamada a subprograma o una expresión entre paréntesis.

```
expresion_primaria ::= expresion_constante | objeto | llamada_subprograma | '(' expresion ')'
```

A continuación se implementará el resto de operadores de manera similar a como hemos hecho con los operandos, teniendo en cuenta sus precedencias y asociatividades. De mayor a menor precedencia:

- '-' unario.

[7] Una excepción es un objeto que tiene que haber sido declarado previamente.

- '`**`' (potencia)
- '`*`', '`/`' y '`%`' (módulo)
- '`+`' y '`-`'.
- '`<-`' y '`->`' (operadores de desplazamiento)
- '`&`' (and binario)
- '`^`' (or binario)
- '`@`' (xor binario)
- '`<`', '`>`', '`=<`', '`>=`', '`:=`' y '`!=`'
- '!' (negación lógica)
- '`/\`' (and lógico)
- '`\/`' (or lógico)

Los operadores definidos en la misma línea tienen la misma precedencia. Todos los operadores anteriores son binarios, excepto '`-`' unario, y '!', que son unarios (y prefijos). Respecto a la asociatividad, '`**`', '`/\`' y '`\/`' son asociativos por la derecha, mientras que '`-`' unario y '!' no son asociativos y el resto son asociativos por la izquierda.

A la hora de diseñar las reglas para los operadores binarios, podeis implementar la precedencia y asociatividad diseñando una gramática determinista, o podeis implementar esta porción de la gramática como ambigua y definir las precedencias y asociatividades a través de la definición de los operadores en la zona de declaraciones. Como se dice más arriba, esta última opción se valorará con la mitad de la nota que la primera (0'2 en lugar de 0'4). Si elegís la primera posibilidad, podeis usar como ejemplo la gramática de las expresiones aritméticas que se usa repetidamente en los ejemplos sobre gramáticas LR(k) que hemos visto en clase:

$$\begin{array}{llll} E' \to E & E \to E + T & T \to T * F & F \to (E) \\ & \mid T & \mid F & \mid id \end{array}$$

También debeis recordar que la asociatividad de un operando determina el tipo de recursividad que vais a usar en las reglas para implementar dicho operando. Así, si es asociativo por la izquierda, la regla o reglas correspondientes serán recursivas por la izquierda, como ocurre con la suma y el producto en la gramática anterior. Por otra parte, si el operador es asociativo por la derecha, se implementará mediante reglas con recursividad derecha. Finalmente, si el operador no es asociativo se implementará a través de reglas no recursivas.

2.8. Implementación

En las secciones anteriores he intentado presentar la especificación de la gramática de la manera más clara posible. En general he presentado las reglas de arriba (más cerca del axioma) a abajo, excepto en el caso de las expresiones, en el que debido al número de operadores potenciales, y a la necesidad de implementar la precedencia y asociatividad de los mismos, me pareció mejor idea empezar por la parte más baja, la definición de operandos, antes de pasar a los operadores.

Ahora bien, que haya definido la especificación de Morónico en un cierto orden, no quiere decir que sea el mejor orden para escribir las reglas del analizador. En vuestro lugar, yo intentaría escribir la gramática por partes, y hacer pruebas sobre lo ya escrito antes de pasar a la siguiente parte. El orden que yo seguiría es:

1. Expresiones.
2. Instrucciones (empezando por `instruccion_asignacion`).
3. Declaración de subprogramas.
4. Declaración de tipos (menos las clases).

5. Declaración de constantes y variables.
6. Declaración de clases.
7. Declaración de cargas, paquetes y programa.
8. Tratamiento de errores.

Este procedimiento tiene la ventaja de que, si no os da tiempo a implementar toda la gramática, podeis presentarme una porción de la misma que más o menos funcione. Además, en caso de hacer la práctica por parejas, el escribir las reglas de esta manera os permite trabajar en paralelo en diferentes partes de la especificación. Por ejemplo, podeis hacer las expresiones y las instrucciones al mismo tiempo. El encargado de escribir las instrucciones puede tener una regla para que la categoría `expresion` derive, por ejemplo, una `CTC_REAL`, mientras su compañero no escriba las reglas correspondientes.

2.9. Ejemplo

Os he dejado un programa de ejemplo (`ordenar.mor`) en el archivo `moronico.tar.gz`. Recordad que la salida del analizador tiene que ser un volcado de los tokens que se van leyendo y de las reglas que se van reduciendo. El resultado de aplicar vuestro analizador a `ordenar.mor`, debería ser parecido a esto:

```
linea 1, palabra reservada: paquete
linea 1, identificador: ordenar
  nombre_decl -> IDENTIFICADOR
linea 1, delimitador: ;
linea 3, palabra reservada: CABECERA
linea 4, palabra reservada: CONSTANTE
linea 5, identificador: N
  nombre_decl -> IDENTIFICADOR
linea 5, delimitador: :
linea 5, palabra reservada: entero
  tipo_escalar -> ENTERO
  tipo_no_str_o_nom -> tipo_escalar
linea 5, operador: =
linea 5, constante entera: 10
  expr_ctc -> CTC_ENTERA
  expr_prim -> expr_ctc
  expr_neg -> expr_prim

...

linea 157, delimitador: }
  blq_instrs -> '{' instrs '}'
  instr_bucle -> claus_iter blq_instrs
  instr -> instr_bucle
  instrs -> instr
linea 158, delimitador: }
  blq_instrs -> '{' instrs '}'
  blq_subprg -> declr_tipos declr_ctcs declr_vars blq_instrs
  declr_subprg -> cab_subprg blq_subprg
  declrs_subprgs -> declrs_subprgs declr_subpr
  seccion_cuerpo -> declr_tipos declr_cons declr_vars declrs_subprgs
  def_paq -> PAQUETE nom ';' seccion_cab seccion_cuerpo
EXITO: programa -> def_paq
```

A. Definición (casi) completa de la gramática de Morónico

Para que no tengais que ir buscando cacho a cacho en el texto. Obviamente, faltan las definiciones de las reglas para operadores en las expresiones, dado que sólo os he descrito sus precedencias y asociatividades.

```
****************************************PROGRAMA****************************************
programa ::= definicion_programa
           | definicion_paquete

definicion_programa ::= PROGRAMA nombre ';' bloque_programa

nombre ::= [ IDENTIFICADOR '::' ]* IDENTIFICADOR

bloque_programa ::= [ declaracion_cargas ]?
                    [ declaracion_tipos ]?
                    [ declaracion_constantes ]?
                    [ declaracion_variables ]?
                    bloque_instrucciones

bloque_instrucciones ::= '{' [ instruccion ]+ '}'

definicion_paquete ::= PAQUETE nombre ';' seccion_cabecera seccion_cuerpo

seccion_cabecera ::= CABECERA
                     [ declaracion_cargas ]?
                     [ declaracion_tipos ]?
                     [ declaracion_constantes ]?
                     [ declaracion_variables ]?
                     [ declaracion_interfaces ]?

seccion_cuerpo ::= CUERPO
                   [ declaracion_tipos ]?
                   [ declaracion_constantes ]?
                   [ declaracion_variables ]?
                   [ declaracion_subprograma ]+

declaracion_cargas ::= CARGA ( declaracion_carga )+ ';'

declaracion_carga ::= nombre [ EN PATH ]? [ '(' ( nombre )+ ')' ]?

****************************************TIPOS****************************************
declaracion_tipos ::= TIPO [ declaracion_tipo ]+

declaracion_tipo ::= nombre '=' tipo_no_estructurado_o_nombre_tipo ';'
                   | nombre '=' tipo_estructurado

tipo_no_estructurado_o_nombre_tipo ::= nombre
                                     | tipo_escalar
                                     | tipo_fichero
                                     | tipo_enumerado
                                     | tipo_lista
                                     | tipo_lista_asociativa
                                     | tipo_conjunto

tipo_estructurado ::= tipo_registro | declaracion_clase

tipo_escalar ::= ENTERO | REAL | BOOLEANO | CARACTER | CADENA
```

```
tipo_fichero ::= FICHERO

tipo_enumerado ::= '(' ( expresion_constante )+ ')'

tipo_lista ::= LISTA [ '[' ( rango )+ ']' ]? DE tipo_no_estructurado_o_nombre_tipo

rango ::= expresion '..' expresion [ '..' expresion ]?

tipo_lista_asociativa ::= LISTA ASOCIATIVA DE tipo_no_estructurado_o_nombre_tipo

tipo_conjunto = CONJUNTO DE tipo_no_estructurado_o_nombre_tipo

tipo_registro ::= REGISTRO '{' [ declaracion_campo ]+ '}'

declaracion_campo ::= ( nombre )+ ':' tipo_no_estructurado_o_nombre_tipo ';'

*************************************CONSTANTES*****************************************
declaracion_constantes ::= CONSTANTE [ declaracion_constante ]+

declaracion_constante ::= nombre ':' tipo_no_estructurado_o_nombre_tipo '=' valor_constante ';'

valor_constante ::= expresion
                  | '[' ( valor_constante )+ ']'
                  | '[' ( clave_valor )+ ']'
                  | '[' ( campo_valor )+ ']'

clave_valor ::= CTC_CADENA '=>' valor_constante

campo_valor ::= nombre '=>' valor_constante

*************************************VARIABLES******************************************
declaracion_variables ::= VARIABLE [ declaracion_variable ]+

declaracion_variable ::= ( nombre )+
                         ':'
                         tipo_no_estructurado_o_nombre_tipo
                         [ '=' valor_constante ]?
                         ';'

*************************************INTERFACES*****************************************
declaracion_interfaces ::= INTERFAZ [ cabecera_subprograma ';' ]+

*************************************CLASES*********************************************
declaracion_clase ::= CLASE
                      [ FINAL ]?
                      [ '(' ( nombre )+ ')' ]?
                      '{'
                      declaraciones_publicas
                      [ declaraciones_semi ]?
                      [ declaraciones_privadas ]?
                      '}'

declaraciones_publicas ::= [ PUBLICO ]? [ declaracion_componente ]+

declaraciones_semi ::= SEMIPUBLICO [ declaracion_componente ]+
```

```
declaraciones_privadas ::= PRIVADO [ declaracion_componente ]+

declaracion_componente ::= declaracion_tipo_anidado
                         | declaracion_constante_anidada
                         | declaracion_atributos
                         | cabecera_subprograma ';' [ modificadores ';' ]?

declaracion_tipo_anidado ::= TIPO declaracion_tipo

declaracion_constante_anidada ::= CONSTANTE declaracion_constante

declaracion_atributos ::= ( nombre )+ ':' tipo_no_estructurado_o_nombre_tipo ';'

modificadores ::= ( modificador )+

modificador ::= GENERICO | ABSTRACTO | ESPECIFICO | FINAL

************************************SUBPROGRAMAS****************************************
declaracion_subprograma ::= cabecera_subprograma bloque_subprograma

cabecera_subprograma ::= cabecera_funcion
                       | cabecera_procedimiento
                       | cabecera_constructor
                       | cabecera_destructor

cabecera_funcion ::= FUNCION
                     nombre
                     [ declaracion_parametros ]?
                     '=>'
                     tipo_no_estructurado_o_nombre_tipo

cabecera_procedimiento ::= PROCEDIMIENTO nombre [ declaracion_parametros ]?

cabecera_constructor ::= CONSTRUCTOR nombre [ declaracion_parametros ]?

cabecera_destructor ::= DESTRUCTOR nombre

declaracion_parametros ::= '(' lista_parametros_formales ')'

lista_parametros_formales ::= parametros_formales
                            | lista_parametros_formales ';' parametros_formales

parametros_formales ::= ( nombre )+ ':' tipo_no_estructurado_o_nombre_tipo [ MODIFICABLE ]?

bloque_subprograma ::= [ declaracion_tipos ]?
                       [ declaracion_constantes ]?
                       [ declaracion_variables ]?
                       bloque_instrucciones

************************************INSTRUCCIONES***************************************
instruccion ::= ';'
              | instruccion_asignacion
              | instruccion_salir
              | instruccion_devolver
              | instruccion_llamada
              | instruccion_si
              | instruccion_casos
```

```
              | instruccion_bucle
              | instruccion_probar_excepto
              | instruccion_lanzar

instruccion_asignacion ::= objeto '=' expresion ';'

instruccion_salir ::= SALIR [ SI expresion ]? ';'

instruccion_devolver ::= DEVOLVER [ expresion ]? ';'

instruccion_llamada ::= llamada_subprograma ';'

llamada_subprograma ::= nombre '(' ( expresion )* ')'

instruccion_si ::= SI expresion
                   ENTONCES bloque_instrucciones
                   [ SINO bloque_instrucciones ]?

instruccion_casos ::= EN CASO expresion 'es' [ caso ]+ ';'

caso ::= CUANDO entradas '=>' bloque_instrucciones

entradas ::= [ entrada '|' ]* entrada

entrada ::= expresion | rango | OTRO

instruccion_bucle ::= clausula_iteracion bloque_instrucciones

clausula_iteracion ::= PARA nombre EN objeto
                     | REPITE ELEMENTO nombre EN rango [ DESCENDENTE ]?
                     | MIENTRAS expresion
                     | REPITE HASTA expresion

instruccion_probar_excepto ::= PROBAR bloque_instrucciones
                               EXCEPTO [ clausula_excepcion ]+
                               [ FINALMENTE bloque_instrucciones ]?

clausula_excepcion ::= CUANDO nombre EJECUTA bloque_instrucciones

instruccion_lanzar ::= LANZAR nombre ';'

*****************************EXPRESIONES (SOLO OPERANDOS)*****************************
expresion_primaria ::= expresion_constante | objeto | llamada_subprograma | '(' expresion ')'

objeto ::= nombre
         | objeto '[' ( expresion )+ ']'
         | objeto '.' IDENTIFICADOR

expresion_constante ::= CTC_ENTERA | CTC_REAL | CTC_CADENA | CTC_CARACTER | CTC_BOOLEANA
```