Practica 2.- Analizador Sintáctico de Machina

TALF 2021/22

2 de abril de 2022

Índice

1. Descripción de la práctica

Objetivo: El alumno deberá implementar un analizador sintáctico en Bison para el lenguaje de programación Machina, inventado para la ocasión. Usando el analizador léxico de la práctica 1, 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 últimas se escribirán en la salida a medida que se vayan reduciendo durante el análisis sintáctico.

Es necesario reducir todo lo que se pueda los conflictos en la gramática (o eliminarlos completamente). En caso de que queden conflictos sin eliminar, se darán por buenos si la acción por defecto del analizador asegura un análisis correcto, y el alumno es capaz de explicar como funciona esa acción por defecto.

Adicionalmente, se puntuará que se use el mecanismo de tratamiento de errores de Bison para evitar que el analizador realice el procesamiento de la entrada hasta el final, en lugar de pararse en cuando encuentre un error.

Documentación a presentar: El código fuente se enviará a través de Moovi. 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

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

Fecha de entrega y defensa: El plazo límite para subirla a Moovi es el domingo 8 de mayo de 2022, a las 23:59. La defensa tendrá lugar en las clases de aula pequeña de la semana del 9 de mayo.

Material: He dejado en Moovi (TALF → Documentos y Enlaces → Material de prácticas → Práctica 2) un directorio comprimido, `machina.bison.tar.gz`, con los siguientes archivos:

- `machina.l`, en el que se puede escribir la especificación del analizador léxico.
- `machina.y`, en el que se puede escribir la especificación del analizador sintáctico.
- `prueba.mac`, un archivo con código Machina para probar el analizador.
- `Makefile`, para compilar la especificación del analizador y generar un ejecutable, de nombre `machina`.

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

- 0'4 por las declaraciones.
- 0'4 por la definición de tipos.
- 0'3 por la definición de subprogramas.
- 0'6 por las instrucciones.
- 0'5 por las expresiones (0'25 si se emplean precedencias y asociatividades).
- 0'3 por la gestión de errores.

Para sumar 0'5 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 0'25 ptos.

Si el analizador presentado tiene conflictos sin justificar, 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 Machina

En esta sección vamos a proporcionar la especificación de la gramática de Machina. Para ello, usaremos una notación similar a BNF, con la cabeza de cada regla separada de la parte derecha por el símbolo '`::=`'. Además:

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

  ```
  instruccion_asignacion: nombre ':=' expresion ';'
  ```

  `expresion` y `nombre` son categorías sintácticas, mientras que '`:=`' y '`;`' son categorías léxicas.
- Una barra vertical separa dos reglas con el mismo símbolo cabeza. Por ejemplo:

  ```
  literal ::= CTC_CADENA | CTC_CARACTER | CTC_FLOAT | CTC_INT | 'true' | 'false'
  ```

  que también puede escribirse como:

```
literal ::= CTC_CADENA
          | CTC_CARACTER
          | CTC_FLOAT
          | CTC_INT
          | 'true'
          | 'false'
```

- Los corchetes especifican la repetición de símbolos (terminales o no terminales)[1]. Dichos símbolos pueden aparecer un número finito de veces, como mínimo cero, en cuyo caso escribiremos '`[ ]*`', o una, en cuyo caso escribimos '`[ ]+`'. De este modo, en la siguiente regla:

  ```
  tipo_registro ::= 'record' [ componente ]+ 'finish' 'record'
  ```

  un tipo registro está formado por, al menos, un `componente`, delimitado por las palabras reservadas '`record`' y '`finish`' '`record`'.

- Los símbolos '`[ ]?`' delimitan los elementos opcionales. En la regla:

  ```
  declaracion_subprograma ::= especificacion_subprograma [ cuerpo_subprograma ]? ';'
  ```

  vemos que el cuerpo del subprograma es opcional en una declaración de subprograma.

- Los paréntesis especifican la repetición de símbolos separados por comas[2]. Dichos símbolos pueden aparecer un número finito de veces, como mínimo cero, en cuyo caso escribiremos '`( )*`', o una, en cuyo caso escribimos '`()+`'. De este modo, la siguiente regla:

  ```
  declaracion_parametro ::= ( IDENTIFICADOR )+ ':' [ modo ]? especificacion_tipo
  ```

  especifica que una `declaracion_parametro` deriva uno o más identificadores, separados por comas, seguidos del delimitador '`:`', un `modo` opcional y una especificación de tipo.

2.1. Especificación de un programa Machina

Un programa Machina está compuesto por una o más declaraciones, a través de las cuales podemos definir objetos, tipos y subprogramas. A su vez, un objeto puede ser una variable o una constante. La sintaxis de declaración de los objetos consiste en una serie de identificadores (los nombres de las variables o constantes que estamos declarando) separadas por comas y seguidas por el delimitador '`:`' y el tipo de la mismas. En el caso de variables de tipo escalar (entero, real, booleano o carácter) podemos especificar los valores iniciales de las mismas incluyendo un operador de asignación ('`:=`') seguido de una o más expresiones separadas por comas. La definición de constantes es casi la misma, sólo diferenciándose en la adición de la palabra reservada '`constant`' a continuación del delimitador '`:`'. En ambos casos, la declaración termina con un delimitador '`;`'.

```
declaracion ::= declaracion_objeto
              | declaracion_tipo
              | declaracion_subprograma

declaracion_objeto ::= ( IDENTIFICADOR )+ ':' [ 'constant' ]? tipo_escalar
                       [ asignacion_escalar ]? ';'
                     | ( IDENTIFICADOR )+ ':' [ 'constant' ]? tipo_complejo
                       [ asignacion_complejo ]? ';'

tipo_escalar ::= 'integer' | 'float' | 'boolean' | 'character'

asignacion_escalar ::= ':=' ( expresion )+
```

[1] Excepto cuando aparecen entre comillas simples. En ese caso son categorías léxicas.
[2] Excepto cuando aparecen entre comillas simples.

Por ejemplo:

```
N: constant INTEGER := 10;
altura,peso: float;
```

También podemos declarar objetos correspondientes a nombres de tipo o tipos compuestos. El nombre de un tipo previamente declarado se escribirá como un identificador, mientras que un tipo compuesto puede ser un array/tabla (`tipo_tablero`), una estructura compuesta de campos (`tipo_registro`), una lista asociativa (`tipo_hashtable`), una clase (`tipo_clase`) o un tipo enumerado (`tipo_enumeracion`).

Al igual que en el caso de los objetos de tipo escalar, podemos definir constantes escribiendo la palabra reservada '`const`' antes del tipo.

Si asignamos valores a los objetos, tendremos que hacerlo en función del tipo de los mismos. En el caso de los tableros (arrays) se definirá una secuencia de simbolos `objeto_complejo`, separados por comas, y delimitados por corchetes. En el caso de tablas hash, tendermos una sucesión de pares clave-valor (ambos `objeto_complejo` con '`->`' en medio), también separados por comas y delimitados por llaves. En el caso de registros, tendremos asignaciones de valores (`objeto_complejo`) a nombres de campo (`IDENTIFICADOR`), separadas por comas, y delimitadas por paréntesis. En el resto de los casos, la constante asignada será un `literal`, que definiremos más abajo.

```
tipo_complejo ::= nombre_de_tipo | tipo_compuesto

nombre_de_tipo ::= IDENTIFICADOR

tipo_compuesto ::= tipo_tablero | tipo_registro | tipo_hashtable | tipo_clase | tipo_enumeracion

asignacion_compleja ::= ':=' objeto_complejo

objeto_complejo ::= '[' ( objeto_complejo )+ ']'
                  | '{' ( elemento_hastable )+ '}'
                  | '(' ( elemento_registro )+ ')'
                  | literal

elemento_hastable ::= objeto_complejo '->' objeto_complejo

elemento_registro ::= IDENTIFICADOR ':=' objeto_complejo
```

2.2. Tipos

En esta sección veremos cómo declarar tipos y cómo definir los diferentes tipos compuestos usados en Machina: tablas, registros, listas asociativas, clases y tipos enumerados. Una declaración de tipo está formada por la palabra reservada '`type`', seguida de un identificador (el nombre del tipo), la palabra reservada '`is`' y una especificación de tipo, terminando con el delimitador '`;`'. A su vez, una especificación de tipo puede ser un tipo escalar, otro nombre de tipo o un tipo compuesto.

```
declaracion_tipo ::= 'type' IDENTIFICADOR 'is' especificacion_tipo ';'

especificacion_tipo ::= tipo_escalar | nombre_de_tipo | tipo_compuesto
```

Por ejemplo:

```
type entero is integer;
type t_hash_cadenas is hashtable of <integer, array (1..50) of character>;
```

Con respecto a los tipos compuestos, la sintaxis de definición de una tabla es la palabra reservada '`array`' seguida de dos expresiones que especifican los valores numéricos del primer y último indice de la misma. Dichos valores se especifican entre paréntesis y separados por el delimitador '`..`'. A continuación se escribe el tipo de los elementos del

array, con la palabra reservada 'of' seguida de una especificación de tipo. Dado que una especificación de tipo puede ser un un tipo compuesto, podemos tener arrays multidimensionales, que serían declarados como arrays de arrays.

```
tipo_tablero ::= 'array' '(' expresion '..' expresion ')' 'of' especificacion_tipo
```

Ejemplos:

```
    lista: array (1..100) of integer;
    bidimensional: array (1..10) of array (1..20) of integer;
```

Por otra parte, el `tipo_registro` nos permite declarar estructuras de datos, con campos que pueden anidarse. La sintaxis para ello es la palabra reservada '`record`', seguida de uno o más componentes, y la secuencia '`finish record`'. Cada componente estará formado por uno o más identificadores separados por comas (los nombres de los campos) seguidos del delimitador '`:`' y una especificación de tipo, terminando la definición con un delimitador '`;`'.

```
tipo_registro ::= 'record' [ componente ]+ 'finish' 'record'

componente ::= ( IDENTIFICADOR )+ ':' especificacion_tipo ';'
```

Una lista asociativa (tabla hash) se declara usando la secuencia '`hashtable of`' seguida de dos especificaciones de tipo, separadas por una coma y delimitadas por '<' '>'. Dichas especificaciones corresponden, respectivamente, al tipo de las claves y al de los elementos almacenados en la tabla.

```
tipo_hashtable ::= 'hastable' 'of' '<' especificacion_tipo  ',' especificacion_tipo '>'
```

Ejemplos:

```
    type FECHA is record
        DIA : INTEGER;
        MES : INTEGER;
        ANNO : INTEGER;
    finish record;

    mi_hash: hashtable of <FECHA, integer>;
```

Para declarar una clase se usa la palabra reservada '`class`', seguida opcionalmente de un nombre de tipo entre paréntesis (la clase 'padre' de la que se está definiendo), uno o más componentes de la clase (objetos, tipos o métodos) y las palabras reservadas '`finish`' y '`class`'.

```
tipo_clase ::= 'class' [ '(' nombre_de_tipo ')' ]? [ componente_clase ]+ 'finish' 'class'

componente_clase ::= [ visibilidad ]? declaracion_componente

declaracion_componente ::= declaracion_objeto | declaracion_tipo | declaracion_metodo

visibilidad ::= 'public' | 'protected' | 'private'

declaracion_metodo ::= [ modificador ]* declaracion_subprograma

modificador ::= 'constructor' | 'destructor' | 'abstract' | 'especific' | 'final'
```

La declaración de cada componente de una clase puede estar precedida por la visibilidad del mismo ('`public`', '`protected`' o '`private`'). Si se omite, se asume que el componente es público. La sintaxis de las declaraciones de objetos y tipo ya las hemos visto anteriormente. Con respecto a los métodos, se declaran como cero o más modificadores ('`constructor`', '`destructor`', '`abstract`', '`especific`', '`final`') seguidos de la declaración de un subprograma (cuya sintaxis veremos en la siguiente subsección).

Por ejemplo:

```
TYPE esfera IS CLASS (forma)
    PRIVATE radio : float;
    -- constructor
    CONSTRUCTOR PROCEDURE esfera (radio : FLOAT) is
    START
        esfera.radio := radio;
    FINISH radio;
    -- metodo
    PUBLIC FINAL FUNCTION volumen RETURN FLOAT is
    START
        return 4%3*PI*pow(radio,3);
    FINISH volumen;
FINISH CLASS;
```

Para finalizar con los tipos, un tipo enumerado se especifica con la secuencia de palabras reservadas '`enumeration of`', seguida de la definición de un tipo escalar, y uno o más elementos de dicho tipo escalar (cada uno de ellos un `literal`), seguidos de la secuencia '`finish enumeration`'. Cada elemento del tipo pueden tener un nombre, definido como un identificador.

```
tipo_enumeracion ::= 'enumeration' 'of' tipo_escalar ( elemento )+ 'finish' 'enumeration'

elemento ::= [ IDENTIFICADOR '->' ]? literal
```

Ejemplo:

```
type dia_semana is enumeration of integer
  LUNES->1, MARTES->2, MIERCOLES->3, JUEVES->4, VIERNES->5, SABADO->6, DOMINGO->7
finish enumeration;
```

2.3. Subprogramas

Una declaración de subprograma está formada por la especificación del mismo (la firma de la función o procedimiento) seguida, opcionalmente, por su código fuente (el cuerpo del subprograma). Un subprograma puede ser un procedimiento o una función. El primer caso se especifica con la palabra reservada '`procedure`' seguida del nombre del procedimiento (un identificador) y la declaración de parámetros. Las funciones se especifican con una sintaxis similar, usando la palabra reservada '`function`' al principio de la declaración, y concluyendo ésta con la definición del tipo del valor devuelto, a través de la palabra reservada '`return`' seguida de una especificación de tipo.

```
declaracion_subprograma ::= especificacion_subprograma [ cuerpo_subprograma ]? ';'


especificacion_subprograma ::= 'procedure' IDENTIFICADOR [ '(' parte_formal ')' ]?
                             | 'function' IDENTIFICADOR [ '(' parte_formal ')' ]?
                               'return' especificacion_tipo

parte_formal ::= [ declaracion_parametros ]?

declaracion_parametros ::= declaracion_parametro [ ';' declaracion_parametro ]*

declaracion_parametro ::= ( IDENTIFICADOR )+ ':' [ modo ]? especificacion_tipo

modo ::= IN [ OUT ]?

cuerpo_subprograma ::= 'is' [ declaracion ]* 'start' [ instruccion ]+ 'finish' [ IDENTIFICADOR ]?
```

Con respecto a los parámetros, si el subprograma no tiene ninguno, se puede especificar con la secuencia '`()`', u omitiendo completamente la definición de parámetros. Si, por el contrario, el subprograma tiene parámetros, estos se declaran delimitados por paréntesis, y separados por '`;`'. Cada declaración está formada por uno o más identificadores (los nombres de los parámetros), seguidos del delimitador '`:`', el modo de paso de los parámetros y el tipo de los mismos. Para pasar los parámetros sin que estos se modifiquen dentro del subprograma, se usará la palabra reservada '`in`', o se omitirá el modo. Si se desean modificar los parámetros dentro del procedimiento o función, se pasarán en modo '`in out`'.

Ejemplos:

```
procedure METER(I,N: in out INTEGER) is
    TEMPORAL,J: INTEGER;
    TERMINAR: BOOLEAN;
start
    TEMPORAL := TABLA[I];
    J := 2*I;
    ...
    TABLA[J%2] := TEMPORAL;
finish METER;
```

2.4. Instrucciones

Vamos a considerar los siguientes tipos de instrucciones:

```
instruccion ::= instruccion_vacia
              | instruccion_asignacion
              | instruccion_exit
              | instruccion_return
              | instruccion_if
              | instruccion_case
              | instruccion_loop
              | instruccion_rise
              | instruccion_try_catch
              | llamada_procedure
```

Instrucción Vacía: Está formada por la palabra reservada '`nil`' seguida del punto y coma.

```
instruccion_vacia: 'nil' ';'
```

Instrucción asignación: Sirve para copiar el valor resultante de una expresión a una variable, elemento de array o lista asociativa, campo de un registro, etc. Está formada por un `nombre`, que definiremos más adelante cuando veamos las expresiones, seguido del operador de asignación '`:=`' y de una expresión. La instrucción termina con '`;`'.

```
instruccion_asignacion: nombre ':=' expresion ';'
```

Instrucción return: Se usa para que una función termine su ejecución devolviendo un valor. La sintaxis es la que tenemos a continuación:

```
instruccion_return: 'return' expresion ';'
```

Instrucción salida de bucle: Está formada por la palabra reservada '`exit`' seguida opcionalmente de un identificador, que sirve para especificar qué bucle (en caso de haber varios anidados) termina. Si no se especifica se asume que se trata del bucle actual. También se puede especificar una condición de obligado cumplimiento para la salida del bucle, usando la palabra reservada '`when`' seguida de una expresión. La instrucción termina con '`;`'.

```
instruccion_exit ::= 'exit' [ IDENTIFICADOR ]? [ 'when' expresion ]? ';'
```

Instrucción if: La clásica instrucción de bifurcación condicional. Está formada por la palabra reservada '`if`', seguida de la condición (una expresión), la palabra reservada '`then`' y una o más instrucciones. Opcionalmente, puede aparecer la palabra reservada '`else`' seguida de una o más instrucciones. La instrucción termina con la secuencia '`finish if ;`'.

```
instruccion_if ::= 'if' expresion 'then' [ instruccion ]+
                   [ 'else' [ instruccion ]+ ]? 'finish' 'if' ';'
```

Por ejemplo:

```
if TEMPORAL > TABLA[J] then
     TERMINAR := TRUE;
else
     TABLA[J%2] := TABLA[J];
     J := 2*J;
finish if;
```

Instrucción case: Instrucción de bifurcación que trata automáticamente varias condiciones distintas, similar a '`switch`' '`case`' en C. Básicamente especificamos una expresión que se tendrá que evaluar, y una serie de valores (o combinaciones de valores) posibles, con código asociado a cada uno de ellos, que tendrá que ejecutarse si la expresión tiene el valor correspondiente. La sintaxis de la instrucción empieza con la palabra reservada '`case`', seguida de la expresión a evaluar, la palabra reservada '`is`', los posibles casos (al menos uno) y la secuencia '`finish case ;`'.

```
instruccion_case: 'case' expresion 'is' [ caso_when ]+ 'finish' 'case' ';'

caso_when ::= 'when' entrada [ '|' entrada ]* '->' [ instruccion ]+

entrada ::= expresion [ '..' expresion ]?
          | OTHERS
```

La sintaxis de los casos a evaluar es la siguiente: cada uno comienza con la palabra reservada '`when`' seguido de una o más entradas, que especifican los valores que la expresión a evaluar puede tomar para ejecutar el caso actual. Dichas entradas están separadas por el delimitador '`|`' y están formadas por una o dos expresiones separadas por el delimitador '`..`' (si sólo aparece una expresión se trata de un valor, y si hay dos, de un rango de valores), o por la palabra reservada '`others`'. Si esto último ocurre, la entrada concordará con cualquier valor de la expresión que no aparezca en ninguno de los otros casos.

A continuación de las expresiones, separadas de éstas por el delimitador '`->`', se escriben las instrucciones que se ejecutarán si la expresión a evaluar concuerda con el caso actual. Por ejemplo:

```
case HOY.DIA is
    when LUNES .. JUEVES  -> TRABAJO();
    when VIERNES | SABADO -> TRABAJO(); DEPORTE();
    when others           -> nil;
finish case ;
```

Instrucción bucle: Un bucle puede comenzar, opcionalmente, por un identificador (que será usado en la instrucción '`exit`' para identificar el bucle) seguido del delimitador '`:`'. A continuación (o al principio, si se omite el identificador de bucle) aparece la clausula de iteración, en la que se define el índice del bucle y/o la condición de parada. Hay tres posibles cláusulas de iteración: la primera es '`for`', que especifica el rango de valores que adoptará la variable índice de bucle a partir de dos expresiones (el valor más bajo y más alto) separadas por el delimitador '`..`', así como el orden (ascendente por defecto o descendente si se usa la palabra reservada '`reverse`') en el que dichos valores serán recorridos.

```
instruccion_loop ::= [ IDENTIFICADOR ':' ]? clausula_iteracion bucle_base ';'
```

```
clausula_iteracion ::= 'for' IDENTIFICADOR 'in' [ 'reverse' ]? expresion '..' expresion
                     | 'foreach' IDENTIFICADOR 'in' expresion
                     | 'while' expresion

bucle_base ::= 'loop' [ instruccion ]+ 'finish' 'loop'
```

Por su parte, la cláusula '`foreach`' obtiene los valores de la variable índice del bucle de un array, derivado a partir de una expresión, mientras que la cláusula '`while`' especifica la condición que se tendrá que cumplir para no abandonar el bucle. A continuación de la cláusula de iteración tenemos que derivar la instrucciones del bucle a partir de `bucle_base`, delimitadas por la palabra reservada '`loop`' y la secuencia '`finish loop`'. La sintaxis de la instrucción bucle termina con el delimitador '`;`'. Por ejemplo:

```
for I in reverse 1..N_1 loop
    TEMPORAL := TABLA[I+1];
    TABLA[I+1] := TABLA[1];
    TABLA[1] := TEMPORAL;
    METER[1,I];
finish loop;
```

Instrucción rise Está formada por la palabra reservada '`raise`' seguida del nombre de la excepsión que se lanza (un idenfifocador) y el delimitador '`;`'.

```
instruccion_rise ::= 'raise' IDENTIFICADOR ';'
```

Instrucción try-catch: Está formada por la palabra reservada '`try`' seguida de una o más instrucciones (en las que se chequeará el lanzamiento de posibles excepciones) y de las cláusulas en las que se definirá el código a ejecutar en el caso de una excepción. La instrucción finaliza con la secuencia de palabras reservadas '`finish try`'.

```
instruccion_try_catch ::= 'try' [ instruccion ]+ clausulas_excepcion 'finish' 'try'

clausulas_excepcion ::= [ clausula_especifica ]* clausula_defecto
                      | [ clausula_especifica ]+

clausula_especifica ::= 'exception' '(' IDENTIFICADOR ')' [ instruccion ]+

clausula_defecto ::= 'default' '(' IDENTIFICADOR ')' [ instruccion ]+
```

Hay dos tipos de cláusulas de excepción: las cláusulas específicas, cuyo código se ejecuta cuando se lanza una excepción en particular (cuyo nombre es un identificador) y la cláusula por defecto, cuyas instrucciones se ejecutan cuando de lanza una excepción no contemplada en las cláusulas específicas. Una instrucción `try-catch` puede tener una o más cláusulas de excepción específicas, sin cláusula por defecto, o una cláusula por defecto, que puede estar precedida de cláusulas específicas.

Instrucción llamada a procedimiento: Está formada por la llamada a un subprograma: un identificador (el nombre del procedimiento) y, entre paréntesis, los parámetros del mismo. Dichos parámetros son una secuencia de cero o más expresiones, separadas por comas. La instrucción termina con un '`;`'.

```
llamada_procedure ::= llamada_suprograma ';

llamada_suprograma ::= IDENTIFICADOR '(' ( expresion )* ')'
```

2.5. Expresiones

Para definir las posibles expresiones matemáticas y lógicas (que tendrán como raíz el símbolo no terminal `expresion`), empezaremos definiendo los operandos. El símbolo que sirve como raíz para dichos operandos es `primario`, que puede

derivar un literal, nombre o expresión entre paréntesis. Por su parte, un literal será un valor constante (entero, real, carácter, cadena o booleano), mientras que un `nombre` puede ser un componente indexado, el valor asociado a una clave en una lista asociativa, el valor del campo de un registro o el valor de retorno de un método (`componente_compuesto`), el valor de retorno de una función (`llamada_suprograma`), o una variable (un identificador).

```
primario ::= literal | nombre | '(' expresion ')'

literal ::= CTC_CADENA | CTC_CARACTER | CTC_FLOAT | CTC_INT | 'true' | 'false'

nombre ::= componente_indexado
         | componente_hash
         | componente_compuesto
         | llamada_suprograma
         | IDENTIFICADOR
```

Un componente indexado puede ser un `nombre` seguido de una expresión entre corchetes (la sintaxis para acceder al valor de una posición en una tabla), mientras que un `componente_hash` está formado por un `nombre` seguido de una expresión entre llaves (la clave necesaria para acceder a un elemento de la lista asociativa). Por su parte, un componente compuesto es un `nombre` seguido de un punto '`.`' y un identificador (el campo de un registro o atributo de un objeto) o la sintaxis de llamada a procedimiento o función (el método de un objeto).

```
componente_indexado ::= nombre '[' expresion ']'

componente_hash ::= nombre '{' expresion '}'

componente_compuesto ::= nombre '.' IDENTIFICADOR
                       | nombre '.' llamada_suprograma
```

Fijaos que al derivar `nombre` desde `componente_indexado`, `componente_hash` y `componente_compuesto` tenemos una recursividad indirecta, dado que éstos los tres últimos símbolos también se pueden derivar desde `nombre` en las reglas vistas más arriba.

A partir de aquí, tenéis que implementar 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:

- '`-`' (menos unitario)
- '`**`' (potencia)
- '`*`', '`%`' y '`mod`'
- '`+`' y '`-`' ('`-`' está sobrecargado como resta y menos unitario).
- '`&`' (and binario)
- '`@`' (or binario)
- '`=`', '`/=`', '`<`', '`>`', '`<=`', '`>=`'
- '`not`'
- '`and`' (and lógico)
- '`or`' (or lógico)

Todos los operadores anteriores son binarios, excepto '`-`' (menos unitario) y '`not`', que son unarios (y prefijos). Respecto a la asociatividad, los operadores '`-`' (menos unitario), '`=`', '`/=`', '`<`', '`>`', '`<=`', '`>=`' y '`not`' **no** son asociativos, y '`**`' es asociativo por la derecha. El resto de operadores 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 (0'25 ptos en lugar de 0'5). 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:

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

Recordad que en una gramática no ambigua 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.

Para terminar con las expresiones, si llamamos al simbolo raíz de la porción de la gramática correspondiente a todas los operadores y operandos anteriores `expresion_logica`, podemos definir una `expresion` como:

```
expresion ::= expresion_logica [ 'if' expresion 'else' expresion ]?
```

es decir, una `expresion_logica`, que puede estar seguida de la palabra reservada '`if`', una condición, la palabra reservada '`else`' y una expresión. En este último caso, `expresion_logica` será el valor de la expresión si se cumple la condición a continuación del '`if`'. Si dicha condición no se cumple, se optará por el valor de la expresión a continuación de '`else`'.

2.6. 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 a la complejidad en el 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 Machina 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.
3. Especificación de subprogramas.
4. Definición de tipos y declaraciones.

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 funcione.

2.7. Ejemplo

Os he dejado un programa de ejemplo (`prueba.mac` en el archivo `machina.bison.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 `prueba.mac`, debería ser parecido a esto:

```
Linea 1 - Palabra Reservada: procedure
Linea 1 - Identificador: EJEMPLO_ADICIONES
Linea 1 - Palabra Reservada: is
```

```
   spec_suprg -> 'procedure' ID
Linea 2 - Palabra Reservada: type
Linea 2 - Identificador: FECHA
Linea 2 - Palabra Reservada: is
Linea 2 - Palabra Reservada: record
Linea 3 - Identificador: DIA
   ids -> IDENTIFICADOR
Linea 3 - Delimitador: :
Linea 3 - Palabra Reservada: INTEGER
   tipo_escalar -> INTEGER
   spec_tipo -> tp_esc
Linea 3 - Delimitador: ;
   comp -> ids ':' spec_tipo ';'
   comps -> comp
Linea 4 - Identificador: MES
   ids -> IDENTIFICADOR
Linea 4 - Delimitador: :
Linea 4 - Palabra Reservada: INTEGER
   tipo_escalar -> INTEGER
   spec_tipo -> tp_esc
Linea 4 - Delimitador: ;
   comp -> ids ':' spec_tipo ';'
   comps -> comps comp
Linea 5 - Identificador: ANNO
   ids -> IDENTIFICADOR
Linea 5 - Delimitador: :
Linea 5 - Palabra Reservada: INTEGER
   tipo_escalar -> INTEGER
   spec_tipo -> tp_esc
Linea 5 - Delimitador: ;
   comp -> ids ':' spec_tipo ';'
   comps -> comps comp
Linea 6 - Palabra Reservada: finish
Linea 6 - Palabra Reservada: record
   tp_reg -> 'record' comps 'finish' 'record'
   tp_com -> tp_reg
   spec_tipo -> tp_comp
Linea 6 - Delimitador: ;
   declr_tipo -> 'tipe' IDENTIFICADOR 'is' spec_tipo
   declr -> declr_tipo
   declrs -> declr
Linea 8 - Palabra Reservada: type
Linea 8 - Identificador: dia_semana
Linea 8 - Palabra Reservada: is
Linea 8 - Palabra Reservada: enumeration
Linea 8 - Palabra Reservada: of
Linea 8 - Palabra Reservada: integer
   tipo_escalar -> INTEGER
Linea 9 - Identificador: LUNES
Linea 9 - Delimitador: ->
Linea 9 - Entero: 1
   literal -> CTC_INT
   elem -> ID '->' literal
   elems -> elem

...
```

3. Tratamiento de errores

La idea es que, una vez tengais un analizador que funcione correctamente, introduzcais en la gramática tres o cuatro reglas para el tratamiento de errores (aunque se permiten más), usando el token `error` y la macro `yyerrok`. Dichas reglas deberían añadirse en diferentes secciones de la gramática, de modo que el analizador sea capaz de recuperarse de errores en el mayor número de casos posible, en lugar de parar el análisis de la entrada. También debéis evitar la introducción de nuevas ambigüedades derivadas de las reglas de tratamiento de error.

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

He reunido en este apéndice la especificación de la gramática, para hacer más fácil su consulta. Dicha definición es incompleta: faltan las definiciones de las reglas para los operadores de los que solo se ha definido precedencia y asociatividad.

```
**************************************DECLARACIONES*****************************************

declaracion ::= declaracion_objeto
              | declaracion_tipo
              | declaracion_subprograma

declaracion_objeto ::= ( IDENTIFICADOR )+ ':' [ 'constant' ]? tipo_escalar
                       [ asignacion_escalar ]? ';'
                     | ( IDENTIFICADOR )+ ':' [ 'constant' ]? tipo_complejo
                       [ asignacion_complejo ]? ';'

tipo_escalar ::= 'integer' | 'float' | 'boolean' | 'character'

asignacion_escalar ::= ':=' ( expresion )+

tipo_complejo ::= nombre_de_tipo | tipo_compuesto

nombre_de_tipo ::= IDENTIFICADOR

tipo_compuesto ::= tipo_tablero | tipo_registro | tipo_hashtable | tipo_clase | tipo_enumeracion

asignacion_compleja ::= ':=' objeto_complejo

objeto_complejo ::= '[' ( objeto_complejo )+ ']'
                  | '{' ( elemento_hastable )+ '}'
                  | '(' ( elemento_registro )+ ')'
                  | literal

elemento_hastable ::= objeto_complejo '->' objeto_complejo

elemento_registro ::= IDENTIFICADOR ':=' objeto_complejo

****************************************TIPOS*****************************************

declaracion_tipo ::= 'type' IDENTIFICADOR 'is' especificacion_tipo ';'

especificacion_tipo ::= tipo_escalar | nombre_de_tipo | tipo_compuesto

tipo_tablero ::= 'array' '(' expresion '..' expresion ')' 'of' especificacion_tipo

tipo_registro ::= 'record' [ componente ]+ 'finish' 'record'
```

```
componente ::= ( IDENTIFICADOR )+ ':' especificacion_tipo ';'

tipo_hashtable ::= 'hastable' 'of' '<' especificacion_tipo  ',' especificacion_tipo '>'

tipo_clase ::= 'class' [ '(' nombre_de_tipo ')' ]? [ componente_clase ]+ 'finish' 'class'

componente_clase ::= [ visibilidad ]? declaracion_componente

declaracion_componente ::= declaracion_objeto | declaracion_tipo | declaracion_metodo

visibilidad ::= 'public' | 'protected' | 'private'

declaracion_metodo ::= [ modificador ]* declaracion_subprograma

modificador ::= 'constructor' | 'destructor' | 'abstract' | 'especific' | 'final'

tipo_enumeracion ::= 'enumeration' 'of' tipo_escalar ( elemento )+ 'finish' 'enumeration'

elemento ::= [ IDENTIFICADOR '->' ]? literal

************************************SUBPROGRAMAS*************************************

declaracion_subprograma ::= especificacion_subprograma [ cuerpo_subprograma ]? ';'

especificacion_subprograma ::= 'procedure' IDENTIFICADOR [ '(' parte_formal ')' ]?
                             | 'function' IDENTIFICADOR [ '(' parte_formal ')' ]?
                               'return' especificacion_tipo

parte_formal ::= [ declaracion_parametros ]?

declaracion_parametros ::= declaracion_parametro [ ';' declaracion_parametro ]*

declaracion_parametro ::= ( IDENTIFICADOR )+ ':' [ modo ]? especificacion_tipo

modo ::= IN [ OUT ]?

cuerpo_subprograma ::= 'is' [ declaracion ]* 'start' [ instruccion ]+ 'finish' [ IDENTIFICADOR ]?



************************************INSTRUCCIONES*************************************

instruccion ::= instruccion_vacia
              | instruccion_asignacion
              | instruccion_exit
              | instruccion_return
              | instruccion_if
              | instruccion_case
              | instruccion_loop
              | instruccion_rise
              | instruccion_try_catch
              | llamada_procedure

instruccion_vacia: 'nil' ';'

instruccion_asignacion: nombre ':=' expresion ';'
```

```
instruccion_return: 'return' expresion ';'

instruccion_exit ::= 'exit' [ IDENTIFICADOR ]? [ 'when' expresion ]? ';'

instruccion_if ::= 'if' expresion 'then' [ instruccion ]+
                   [ 'else' [ instruccion ]+ ]? 'finish' 'if' ';'

instruccion_case: 'case' expresion 'is' [ caso_when ]+ 'finish' 'case' ';'

caso_when ::= 'when' entrada [ '|' entrada ]* '->' [ instruccion ]+

entrada ::= expresion [ '..' expresion ]?
          | OTHERS

instruccion_loop ::= [ IDENTIFICADOR ':' ]? clausula_iteracion bucle_base ';'

clausula_iteracion ::= 'for' IDENTIFICADOR 'in' [ 'reverse' ]? expresion '..' expresion
                     | 'foreach' IDENTIFICADOR 'in' expresion
                     | 'while' expresion

bucle_base ::= 'loop' [ instruccion ]+ 'finish' 'loop'

instruccion_rise ::= 'raise' IDENTIFICADOR ';'

instruccion_try_catch ::= 'try' [ instruccion ]+ clausulas_excepcion 'finish' 'try'

clausulas_excepcion ::= [ clausula_especifica ]* clausula_defecto
                      | [ clausula_especifica ]+

clausula_especifica ::= 'exception' '(' IDENTIFICADOR ')' [ instruccion ]+

clausula_defecto ::= 'default' '(' IDENTIFICADOR ')' [ instruccion ]+

llamada_procedure ::= llamada_suprograma ';

llamada_suprograma ::= IDENTIFICADOR '(' ( expresion )* ')'

*********************************EXPRESIONES (INCOMPLETAS)*********************************

primario ::= literal | nombre | '(' expresion ')'

literal ::= CTC_CADENA | CTC_CARACTER | CTC_FLOAT | CTC_INT | 'true' | 'false'

nombre ::= componente_indexado
         | componente_hash
         | componente_compuesto
         | llamada_suprograma
         | IDENTIFICADOR

componente_indexado ::= nombre '[' expresion ']'

componente_hash ::= nombre '{' expresion '}'

componente_compuesto ::= nombre '.' IDENTIFICADOR
                       | nombre '.' llamada_suprograma
```