Practica 2.- Analizador Sintáctico de Chusco

ALF 2023/24 - Evaluación Contínua

17 de marzo de 2024

Índice

1. Descripción de la práctica

Objetivo: El alumno deberá implementar un analizador sintáctico en Bison para el lenguaje de programación Chusco, 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 volcarán a 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, si es posible). En caso de que queden conflictos sin eliminar, se darán por buenos si la acción por defecto del analizador asegura un análisis correcto, y sois capaces de explicarme 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 caso de errores.

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-OuteirinoCid

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,

zip o similares) con su mismo nombre.

Ej.- DarribaBilbao-OuteirinoCid.zip

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 Moovi como muy tarde el 7 de mayo a las 23:59. La defensa tendrá lugar en las clases de aula pequeña de los días 8 y 10 de mayo.

Material: He dejado en Moovi (ALF → documentos y enlaces → material de prácticas) un fichero (`chusco.bison.tar.gz`). Dicho fichero contiene la especificación incompleta[1] en Flex del analizador léxico (`chusco.l`), la especificación incompleta[2] del analizador sintáctico en Bison (`chusco.y`, sólo con las reglas para definir un programa), tres ficheros de ejemplo (`misc.chu`, `ordenar.chu` y `program.chu`), y un Makefile.

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

- 0'3 ptos por programas y objetos
- 0'5 ptos por tipos y clases
- 0'2 ptos por subprogramas
- 0'7 ptos por instrucciones
- 0'5 ptos por expresiones
- 0'3 ptos por el tratamiento de errores

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 Chusco

En esta sección vamos a proporcionar la especificación de la gramática de Chusco. 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 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:

  ```
  definicion_parametro ::= [ IDENTIFICADOR ':=' ]? expresion
  ```

  `definicion_parametro` y `expresion` son categorías sintácticas, mientras que `IDENTIFICADOR` y '`:=`' son categorías léxicas.

- Una barra vertical separa la parte derecha de dos reglas con el mismo símbolo cabeza. Por ejemplo:

  ```
  literal ::= VERDADERO | FALSO | CTC_ENTERA | CTC_REAL | CTC_CARACTER | CTC_CADENA
  ```

  que también puede escribirse como:

  ```
  expresion_constante ::= VERDADERO
                        | FALSO
                        | CTC_ENTERA
                        | CTC_REAL
                        | CTC_CARACTER
                        | CTC_CADENA
  ```

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

- Los corchetes especifican la repetición de símbolos (terminales o no terminales)[3] . 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:

  ```
  codigo_libreria ::= [ libreria ]* [ exportaciones ]? [ declaracion ]+
  ```

  tenemos que un `codigo_librería` está formado por 0 o más importaciones de librerías y 1 o más declaraciones.

- Los símbolos '`[ ]?`' delimitan los elementos opcionales. En el ejemplo anterior el no terminal `exportaciones` es opcional, por lo que una cadena generada por ese símbolo puede aparecer o no en el código de una librería.

- Los paréntesis especifican la repetición de símbolos separados por comas[4]. 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:

  ```
  exportaciones ::= 'exportar' ( nombre )+ ';'
  ```

  especifica que `exportaciones` deriva el símbolo terminal '`exportar`', así como 1 o más símbolos `nombre`, separados por comas, y un '`;`'.

2.1. Especificación de un programa Chusco

Un fichero de código Chusco puede contener un programa o una librería. En el primer caso estamos hablando de código que se puede ejecutar directamente una vez compilado, mientras que en el segundo tendremos una colección de declaraciones de objetos (variables o constantes), tipos o librerías.

Por lo tanto, la sintaxis de un programa Chusco está formada por la palabra reservada '`programa`' seguida de un identificador especificando el nombre del programa, el delimitador '`;`' y el código del programa. A su vez el código del programa, contendrá 0 o más llamadas a librerías seguidas del cuerpo de un subprograma (como se verá más abajo, 0 o más declaraciones seguidas de un bloque de instrucciones).

```
programa :: = definicion_programa | definicion_libreria

definicion_programa ::= 'programa' IDENTIFICADOR ';' codigo_programa

codigo_programa ::= [ libreria ]* cuerpo_subprograma

libreria ::= 'importar' 'libreria' nombre [ 'como' IDENTIFICADOR ]? ';'
           | 'de' 'libreria' nombre 'importar' ( IDENTIFICADOR )+ ';'

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

En Chusco existen dos posibilidades a la hora de usar una librería. En la primera de ellas, importaremos la totalidad de los objetos definidos en la librería, especificando la secuencia de palabras reservadas '`importar libreria`' seguidas del nombre de la misma. Dicho `nombre` es una secuencia de 1 o más identificadores separados por el delimitador '`::`', especificando la jerarquía de librerías que debemos seguir para llegar al código que nos interesa importar. Opcionalmente, a través de la palabra reservada '`como`', podemos especificar un identificador que se usará para nombrar la librería en el código de nuestro programa. La importación termina con un '`;`'.

La segunda posibilidad consiste en importar sólo parte de los objetos/tipos/subprogramas declarados en la librería especificando en primer lugar la secuencia '`de libreria`' seguida del nombre de la librería de la que se quiere importar y una lista de identificadores separados por comas. Para delimitar dicha declaración, usaremos el '`;`'. Por ejemplo:

```
importar libreria principal::sistema_operativo::entrada_salida como e_s;

de principal::sistema_operativo::entrada_salida importar abrir,cerrar;
```

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

Por otra parte, para escribir el código de una librería, con definiciones que deseemos exportar a otros programas, utilizaremos la palabra reservada '`libreria`' seguida de un identificador (el nombre de la librería), y el delimitador '`;`'. A continuación escribiremos el código de la librería, comenzando con 0 o más importaciones de objetos de otras librerías y la lista opcional de nombres de objetos/tipos/subprogramas a exportar[5], especificada con la palabra reservada '`exportar`' seguida de 1 o más nombres separados por comas. Finalmente, se escribirán una o más declaraciones de objetos (variables o constantes), tipos y/o subprogramas.

```
definicion_libreria ::= 'libreria' IDENTIFICADOR ';' codigo_libreria

codigo_libreria ::= [ libreria ]* [ exportaciones ]? [ declaracion ]+

exportaciones ::= 'exportar' ( nombre )+ ';'

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

2.2. Declaración de objetos

Para declarar constantes o variables, primero se escriben sus nombres, como identificadores separados por comas, seguidos del delimitador '`:`'. A continuación especificaremos su tipo, precedido de la palabra reservada '`constante`' si estamos definiendo valores constantes. De manera obligatoria en el caso de las constantes, y opcionalmente en el caso de las variables, definiremos el valor del objeto con el operador '`:=`' seguido de una expresión. Terminaremos la declaración con el delimitador '`;`'.

```
declaracion_objeto ::= ( IDENTIFICADOR )+ ':' 'constante' especificacion_tipo ':=' expresion ';'
                     | ( IDENTIFICADOR )+ ':' especificacion_tipo [ ':=' expresion ]? ';'

especificacion_tipo ::= nombre | tipo_no_estructurado
```

La especificación de tipo de una variable o constante puede ser el `nombre` de un tipo declarado previamente (o importado de una librería), o una declaración de tipo no estructurado, que veremos en la siguiente subsección. Por ejemplo:

```
n : constante entero := 20;
lista : tabla(1..n) de entero;
limIzdo : entero := 0;
limDcho : entero := n-1;
```

2.3. Declaración de tipos

Una declaración de tipo está formada por la palabra reservada '`tipo`', seguida del nombre del mismo (un identificador), la palabra reservada '`es`' y el tipo en cuestión. En el caso de un tipo no estructurado, habrá que acabar la declaración con '`;`', a diferencia de lo que se hace con un tipo estructurado.

```
declaracion_tipo ::= 'tipo' IDENTIFICADOR 'es' tipo_no_estructurado ';'
                   | 'tipo' IDENTIFICADOR 'es' tipo_estructurado

tipo_no_estructurado ::= tipo_escalar | tipo_tabla | tipo_diccionario

tipo_escalar ::= [ 'signo' ]? tipo_basico [ longitud ]? [ 'rango' rango ]?

longitud ::= 'corto' | 'largo'

tipo_basico ::= 'booleano' | 'caracter' | 'entero' | 'real'

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

[5] Si no especificamos nada, se exportarán todas las definiciones contenidas en el código de la librería

Un tipo estructurado puede ser un tipo escalar, tabla o diccionario. A su vez, un tipo escalar estará compuesto por un tipo básico (booleano, carácter, entero o real) opcionalmente precedido por la palabra reservada '`signo`' y sucedido por la especificación de si el tipo escalar es '`corto`' o '`largo`'. Ambas definiciones son válidas para los tipos entero y real[6]. La primera determina si los elementos del tipo pueden ser positivos o negativos, mientras que la segunda determina el número de bytes que ocupará el tipo básico en memoria (y por lo tanto su intervalo de valores posibles). De manera opcional, se puede especificar el rango de valores que puede tomar un tipo caracter, entero y real, con la palabra reservada '`rango`' seguida de dos o tres expresiones separadas por el delimitador '`..`'. Las dos primeras determinan los límites mínimo y máximo del rango, mientras que la tercera especifica la diferencia entre valores consecutivos. Por ejemplo:

```
tipo entero_corto_sin_signo es entero corto;
tipo enteros_8bits es signo entero corto rango -255 .. 255;
tipo reales_10000 es real largo rango 0 .. 1^4;
```

Por otra parte, un tipo tabla puede designar dos tipos de estructuras: arrays estáticos, siempre con el mismo tamaño predefinido en memoria, y listas dinámicas, a las que se pueden añadir elementos durante la ejecución del código. La primera de ellas se especifica con la palabra reservada '`tabla`' seguida por los índices más bajo y más alto a los que se puede acceder, especificados entre paréntesis y separados por el delimitador '`..`'. A continuación escribiremos la palabra reservada '`de`' y una especificación de tipo, que puede ser un nuevo tipo tabla, para estructuras de más de una dimensión.

```
tipo_tabla ::= 'tabla' '(' expresion '..' expresion ')' 'de' especificacion_tipo
             | 'lista' 'de' especificacion_tipo
```

Con respecto a las listas dinámicas, se especificarán con las palabras reservadas '`lista de`' seguidas de una especificación de tipo, que, de nuevo, puede ser una tabla. Por ejemplo:

```
tabla_ejemplo : tabla (1..10) de tabla (1..25) de entero;
tabla_de_listas : tabla (1..n) de lista de entero;
lista_ejemplo : lista de real;
lista_3d : lista de lista de lista de real;
cadena_estatica : tabla (0..255) de caracter;
cadena_dinamica : lista de caracter;
```

(Nota: una cadena puede definirse como una lista estática o dinámica de caracteres)

Respecto al tipo diccionario, es una lista asociativa que nos permite acceder a un valor a través de una clave especificada como una cadena. Es una estructura de datos dinámica, por lo que su definición es igual a la de las listas dinámicas, usando con la palabra reservada '`diccionario`' en lugar de '`lista`'.

```
tipo_diccionario ::= 'diccionario' 'de' especificacion_tipo
```

En otro orden de cosas, vamos a considerar posibles tres tipos estructurados: registros, enumeraciones y clases. El primero de ellos es una estructura de datos que puede estar compuesta de múltiples campos, cada uno de los cuales tendrá un nombre y un tipo determinado, que a su vez puede ser un tipo no estructurado o un nombre de tipo. Su definición comienza por la palabra reservada '`registro`', seguida de la definición de uno o más campos, y termina con la secuencia '`fin registro`'.

```
tipo_estructurado ::= tipo_registro | tipo_enumerado | clase

tipo_registro ::= 'registro' [ campo ]+ 'fin' 'registro'

campo ::= ( IDENTIFICADOR )+ ':' especificacion_tipo [ ':= expresion ]? ';'
```

[6]Lo que se chequeará en el análisis semántico. Vosotros no os teneis que ocupar de ello.

A su vez, los campos se especifican como una lista de identificadores, seguidos del delimitador `:` y una especificación de tipo. Opcionalmente, podremos asignar un valor por defecto al campo, con el operador '`:=`' y una expresión. Finalizaremos la definición del campo o campos con el delimitador '`;`'. Por ejemplo:

```
tipo persona es registro
  nombre : lista de caracter;
  edad : entero;
  casada : booleano := Falso;
fin registro
```

Por su parte, un tipo enumerado estará formado por una serie de elementos, que pueden tener valores correspondientes a un tipo escalar, y que se especifican durante la definición del tipo. La sintaxis de dicha definición empieza con la palabra reservada '`enumeración`', seguida opcionalmente por la palabra reservada '`de`' y el tipo escalar al que pertenecerán los valores de los elementos del tipo[7]. A continuación, se especifican dichos elementos, separados por comas, terminando con la secuencia de palabras reservadas '`fin enumeracion`'.

```
  tipo_enumerado ::= 'enumeracion' [ 'de' tipo_escalar ]? ( elemento_enumeracion )+
                     'fin' 'enumeracion'

  elemento_enumeracion ::= IDENTIFICADOR [ ':=' expresion ]?
```

Los nombres de los elementos de la enumeración se escribirán como identificadores, y se podrá asignar a cada uno de ellos el resultado de una expresión, que constituirá su valor, perteneciente al tipo escalar declarado previamente[8]. Por ejemplo:

```
tipo color es enumeracion de entero
  Rojo := 1,
  Verde := 2,
  Azul := 3
fin enumeracion
```

2.4. Declaración de clases

Aunque técnicamente son tipos de datos, las clases tienen mayor complejidad que el resto de tipos, por lo que usaremos una subsección separada para describirlas. Su declaración comienza con la palabra reservada '`clase`', seguida opcionalmente por la palabra reservada '`ultima`'[9]. A continuación se declaran las superclases de las que la clase actual hereda, como una secuencia de nombres separados por comas y delimitados por paréntesis. Finalmente, se declaran los componentes (tipos, atributos y métodos) de la clase, terminando con la secuencia de las palabras reservadas '`fin clase`'.

```
  clase ::= 'clase' [ 'ultima' ]? [ superclases ]? [ declaracion_componente ]+ 'fin' 'clase'

  superclases ::= '(' ( nombre )+ ')'

  declaracion_componente ::= [ visibilidad ]? componente

  visibilidad ::= 'publico' | 'protegido' | 'privado'

  componente ::= declaracion_tipo
               | declaracion_objeto
               | ( modificador )* declaracion_subprograma

  modificador ::= 'constructor' | 'destructor' | 'generico' | 'abstracto' | 'especifico' | 'final'
```

[7] Si no se especifica, Chusco asumirá un entero sin signo.
[8] Lo que se comprobará en el análisis semántico, por lo que no tenemos que preocuparnos ahora.
[9] Que especifica una clase que no puede ser superclase.

Los componentes de la clase se declaran de la misma manera que un tipo, objeto (ya vistos) o subprograma, precedidos por uno de los modificadores de visibilidad: '`publico`', '`protegido`' o '`privado`'[10]. En el caso de los métodos (`declaracion_subprograma`), pueden ser precedidos de 0 o más de los siguientes modificadores: '`constructor`', '`destructor`', '`generico`', '`abstracto`', '`especifico`' o '`final`'[11]. Por ejemplo:

```
tipo esfera es clase (forma)
  privado radio : real largo;                                    ## atributo
  constructor subprograma esfera (radio : real) devolver esfera  ## metodo constructor
    principio
      esfera::radio := radio;
    fin subprograma
  publico subprograma volumen devolver real                      ## metodo
    principio
      devolver 4/3*PI*esfera::radio^3;
    fin subprograma
  fin clase
```

2.5. Declaración de subprogramas

La declaración de un subprograma está formada por la cabecera del mismo y su cuerpo, antecedidos y sucedidos por la palabra reservada '`subprograma`'. La cabecera está formada por el nombre del subprograma (un identificador) sus parámetros y el tipo del resultado devuelto por el subprograma. Tanto la parametrización como el tipo del resultado son opcionales. De aparecer este último, estaremos ante una función, y, en caso contrario, ante un procedimiento.

```
declaracion_subprograma ::= 'subprograma' cabecera_subprograma cuerpo_subprograma 'subprograma'

cabecera_subprograma ::= IDENTIFICADOR [ parametrizacion ]? [ tipo_resultado ]?

parametrizacion ::= '(' [ declaracion_parametros ';' ]* declaracion_parametros ')'

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

modo ::= 'valor' | 'referencia'

tipo_resultado ::= 'devolver' especificacion_tipo

cuerpo_subprograma ::= [ declaracion ]* 'principio' [ instruccion ]+ 'fin'
```

Si el subprograma no tiene ningún argumento de entrada, podemos omitir la parametrización completamente. En caso contrario, listaremos los argumentos de entrada entre paréntesis, separándolos mediante '`;`'. La especificación de un argumento está formado por su nombre (un identificador) seguido de '`:`', que puede estar seguido del modo de paso de parámetros ('`valor`' o '`referencia`') y la especificación del tipo correspondiente. Opcionalmente, podemos definir un valor por defecto mediante un '`:=`' seguido de una expresión. Si el paso del parámetro es por valor, se realizará una copia del mismo que será usada dentro del subprograma, de manera que el valor original se recupera cuando termina la ejecución de dicho subprograma. Si se hace por referencia, las modificaciones al parámetro en cuestión dentro del subprograma se conservan una vez termina la llamada al mismo. Si tenemos varios parámetros con el mismo tipo, su definición se puede agrupar, separándolos por comas y escribiendo a continuación '`:`' y el tipo común. Por ejemplo:

```
subprograma quicksort (datos : tabla(1..n) de entero; limIzdo,limDcho: valor entero)
```

Respecto al cuerpo del programa, comienza por 0 o más declaraciones de objetos, tipos o subprogramas y un bloque de una o más instrucciones, delimitadas por las palabras reservadas '`principio`' y '`fin`'.

[10] Si no se especifica ninguno, se asume '`publico`'.

[11] De comprobar que los modificadores se combinan correctamente ya se ocupará el analizador semántico.

2.6. Instrucciones

Vamos a considerar los siguientes tipos de intrucciones.

```
instruccion ::= instruccion_asignacion
              | instruccion_devolver
              | instruccion_llamada
              | instruccion_si
              | instruccion_casos
              | instruccion_bucle
              | instruccion_interrupcion
              | instruccion_lanzamiento_excepcion
              | instruccion_captura_excepcion
              | ';'
```

La última es la instrucción vacia, que, como su nombre indica, no ejecuta ninguna acción, y se escribe simplemente como el delimitador ';'. Detallamos las siguientes a continuación.

Instrucción de asignación. Asocia el valor resultante de una expresión a un objeto[12] (variable con un tipo básico, instancia de una clase, elemento de una tabla, campo de un registro...).

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

op_asignacion ::= ':=' | ':+' | ':-' | ':*' | ':/' | ':\' | ':^' | ':<' | ':>'
```

Además de la asignación simple, también podemos usar el resto de operadores de asignación, que realizan un cálculo antes de copiar el valor resultante al objeto[13].

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`.

```
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. Si no hay parámetros de entrada, los paréntesis estarán vacíos.

```
instruccion_llamada ::= llamada_subprograma ';'

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

definicion_parametro ::= [ IDENTIFICADOR ':=' ]? expresion
```

De haber parámetros, estos se separan por comas, y estarán formados por, al menos, una expresión (el valor del parámetro). Si deseamos escribir los parámetros en un orden distinto al de la firma de la función, podemos escribir el nombre de cada parámetro como un identificador seguido del operador `:=` antes de la expresión que especifica el valor del mismo.

Instrucción si .. entonces .. sino. Especifica el `if-then-else` de toda la vida. Si la expresión a continuación de la palabra reservada '`si`' es booleana y cierta[14], se ejecutan las instrucciones asociadas al '`entonces`'. Si, por el contrario, la condición es falsa, se ejecutarán las instrucciones a continuación del '`sino`'. Para evitar ambigüedades, la instrucción temina con la secuencia '`fin si`'.

[12] Dado que generalmente será un operando en expresiones, he definido `objeto` en la subsección dedicada a describir las expresiones.
[13] De verificar que el tipo de la expresión sea el apropiado para el operador, se ocuparía el analizador semántico.
[14] Cosa que vosotros no teneis que comprobar.

```
instruccion_si ::= 'si' expresion 'entonces' [ instruccion ]+
                   [ 'sino' [ instruccion ]+ ]? 'fin' 'si'
```

Instrucción casos. Es similar al `switch-case` de C. En primer lugar, tras la palabra reservada '`casos`', se especifica la expresión cuyo valor será evaluado y la palabra reservada '`es`'. A continuación, se enumerarán los posibles valores de la expresión (especificados mediante el símbolo no terminal `entradas`). Para cada uno de ellos, se escribe, después del delimitador '`=>`', la instrucción o instrucciones que serán ejecutadas si la expresión toma el valor correspondiente.

```
instruccion_casos ::= 'casos' expresion 'es' [ caso ]+ 'fin' 'casos'

caso ::= 'cuando' entradas '=>' [ instruccion ]+

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

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

Como se acaba de indicar, los posibles valores que puede tomar la expresión evaluada pueden ser especificados a través de una o varias entradas, separadas por el operador '`:`', análogo en este caso a la disyunción lógica. Dichas entradas pueden ser expresiones, rangos de valores (`expresion '..' expresion`), o la palabra reservada '`otro`'. 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:

```
casos hoy es
  cuando lunes .. jueves   => trabajo() ;           ## de lunes a juevas
  cuando viernes : sabado  => trabajo(); fiesta();  ## viernes o sabado
  cuando otro              => ;                     ## domingo
fin casos
```

Bucles. La sintaxis de un bucle en Chusco está formada por una cláusula de iteración, especificando el tipo de bucle y la variable índice del mismo, y un bucle base especificando las instrucciones que se ejecutarán en cada iteración. Para poder usar una instrucción '`salir`' con un nombre de bucle determinado, puede asociarse un identificador a cada bucle, escribiéndolo al principio del mismo, seguido del delimitador '`:`'.

```
instruccion_bucle ::= [ IDENTIFICADOR ':' ]? clausula_iteracion [ instruccion ]+ 'fin' 'bucle'

clausula_iteracion ::= 'para' IDENTIFICADOR [ ':' especificacion_tipo ]? 'en' expresion
                     | 'repetir' IDENTIFICADOR [ ':' especificacion_tipo ]?
                       'en' rango [ 'descendente' ]?
                     | 'mientras' expresion
```

Hay tres tipos de bucle en Chusco. El más simple es '`mientras`', para el cual sólo hay que especificar una condición en forma de expresión. Por su parte, el bucle '`para`' es similar al `foreach` presente en algunos lenguajes de programación. A continuación de la palabra reservada '`para`', se especifica una variable que, en cada iteración del bucle, tomará el valor de uno de los elementos de una tabla. Si no se ha declarado previamente, deberá asignarse un tipo a dicha variable a través del delimitador '`:`' seguido de una especificación de tipo. A continuación, la palabra reservada '`en`' se usa para especificar la tabla cuyos elementos se recorren, obtenida a partir de una expresión. Por ejemplo:

```
array : tabla (1..25) de entero;
...
para numero : entero en array
  escribir(numero*numero);
fin bucle
```

Finalmente, también tenemos un bucle '`repetir`', similar al bucle `for` de muchos lenguajes de programación. En su sintaxis, a continuación de la palabra reservada '`repetir`', escribiremos la variable índice que se incrementará o

decrementará, seguida de su especificación de tipo si no ha sido declarada anteriormente. A continuación, escribiremos la palabra reservada '`en`', seguida del rango de valores que se van a recorrer. Estos últimos se establecen a través de un `rango`, que, como se especificó más arriba, consiste en dos o tres expresiones, separadas por el delimitador '`..`'. Las dos primeras serían los límites del rango, mientras que la tercera sería la magnitud del cambio en la variable índice en cada iteración del bucle. Opcionalmente, se puede especificar si el rango es descendente, usando la palabra reservada '`descendente`' a continuación del mismo. Por ejemplo:

```
array : tabla (1..25) de entero;
i : entero;
...
repetir i en n .. 1 .. 5 descendente    ## de n a 1 saltando de 5 en 5
  array[i] := aleatorio(1,100);
fin bucle
```

Instrucción interrupción. Tenemos dos posibles instrucciones para interrumpir el flujo normal de un bucle. La primera de ellas nos permite pasar a la siguiente iteración del bucle, escribiendo la palabra reservada '`siguiente`' seguida opcionalmente de una condición de obligado cuplimiento, expresada como una expresión con valor booleano. La instrucción termina con el delimitador '`;`'.

```
  instruccion_interrupcion ::= 'siguiente' [ cuando ]? ';'
                             | 'salir' [ 'de' IDENTIFICADOR ]? [ cuando ]? ';'

  cuando ::= 'cuando' expresion
```

Por otra parte, la instrucción '`salir`' comienza por dicha palabra reservada, seguida opcionalmente de '`de`' y el nombre (un identificador) del bucle del que se quiere salir. Además, también es posible restringir la ruptura del flujo normal del bucle al cumplimiento de una condición, antes de llegar al delimitador de fin de instrucción, '`;`'.

Instrucción de lanzamiento de excepción. Para lanzar excepciones simplemente escribimos la palabra reservada '`lanza`' seguida del nombre de la excepción y del delimitador '`;`'.

```
  instruccion_lanzamiento_excepcion ::= 'lanza' nombre ';'
```

Instrucción de captura de excepción. Dado que podemos lanzar excepciones, necesitaremos también una instrucción para capturarlas. La sintaxis es similar a la de lenguajes como C#, formada por la palabra reservada '`prueba`' seguida de un 1 o más instrucciones, y de 1 o más cláusulas de excepción o una cláusula '`finalmente`' (o ambas). A su vez, una cláusula de excepción puede ser específica o general. En el primer caso, acompañanando a la palabra reservada '`excepción`', tendremos un nombre de excepción entre paréntesis (que se omite en el caso de una cláusula de excepción general) seguida de una o más instrucciones.

```
  instruccion_captura_excepcion ::= 'prueba' [ instruccion ]+ clausulas 'fin' 'prueba'

  clausulas ::= clausulas_excepcion [ clausula_finalmente ]?
              | clausula_finalmente

  clausulas_excepcion ::= [ clausula_excepcion_especifica ]* clausula_excepcion_general

  clausula_excepcion_especifica ::= 'excepcion' '(' nombre ')' [ instruccion ]+

  clausula_excepcion_general ::= 'excepcion' [ instruccion ]+

  clausula_finalmente ::= 'finalmente' [ instruccion ]+
```

Es posible tener varias clausulas específicas, una sola cláusula general o ambas. En este último caso, la cláusula general debe aparecer después de las específicas. Por su parte, la cláusula '`finalmente`' está compuesta de dicha

palabra reservada seguida de un bloque de instrucciones. Por último, cualquiera que sea la combinación de cláusulas que escribamos, la instrucción termina con la secuencia de palabras reservadas '`fin prueba`'. Por ejemplo:

```
prueba
  mientras indice < longitud(valores)
    E_S::escribir_consola(valores[indice]);
    indice :+ 1;
    fin bucle
  excepcion (longitud_excedida)
    E_S::escribir_consola("Lista vacia%n");
  excepcion
    E_S::escribir_consola("Excepcion indefinida%n");
fin prueba
```

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 y siguiendo con los operadores hasta llegar al no terminal `expresion`, que será el símbolo raíz de esta parte de la gramática. De este modo, un `primario` puede ser un literal, objeto, una llamada a subprograma (precedida por la palabrareservada '`objeto`' si estamos ejecutando el constructor de una clase para crear un nuevo miembro de dicha clase), una enumeración o una expresión entre paréntesis[15].

```
primario ::= literal
           | objeto
           | [ 'objeto' ]? llamada_subprograma
           | enumeraciones
           | '(' expresion ')'

literal ::= VERDADERO | FALSO | CTC_ENTERA | CTC_REAL | CTC_CARACTER | CTC_CADENA

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

A su vez, un `literal` puede ser un booleano ('`verdadero`' o '`falso`'), una constante entera, real, carácter o cadena. Un objeto puede ser un `nombre` (variable o constante previamente definida o importada), el campo de un registro (un `objeto` seguido de '`.`' y un identificador), un elemento de lista o array (un `objeto` seguido de una o más expresiones separadas por comas entre corchetes) o un elemento de un diccionario (un `objeto` seguido de una o más cadenas entre llaves).

Por su parte, una enumeración es una lista, diccionario o registro. En el primer caso podemos generar sus elementos mediante una o más expresiones separadas por comas, o mediante una expresión calculada dentro de uno o más bucles[16]. En este último caso, la expresión puede tener una condición asociada. La lista estará delimitada mediante corchetes.

```
enumeraciones ::= '[' expresion_condicional [ clausula_iteracion ]+ ']'
                | '[' ( expresion )+ ']'
                | '{' ( clave_valor )+ '}'
                | '{' ( campo_valor )+ '}'

expresion_condicional ::= expresion
                        | 'si' expresion 'entonces' expresion [ 'sino' expresion ]?
```

[15] Fijaos que `objeto` y '`objeto`' son símbolos diferentes.
[16] De manera similar al concepto de *list comprehension* en Python.

```
clave_valor ::= CTC_CADENA '=>' expresion

campo_valor ::= IDENTIFICADOR '=>' expresion
```

Si lo que estamos constuyendo es un diccionario, escribiremos los pares clave-valor entre llaves, separados por comas e intercalando el símbolo '=>' entre la cadena que forma la clave y la expresión de la que se obtiene el valor correspondiente. Finalmente, en el caso de un registro, enumeraremos los identificadores de los campos del registro, seguido cada uno de ellos del terminal '=>' y del valor del campo. Los diferentes campos estarán separados por comas, y la estructura estará delimitada por llaves. Por ejemplo:

```
valores : lista de entero corto := [ 88, 56, 100, 2, 25 ];
nuevos_valores : lista de entero corto :=
                    [ si elemento < 80 entonces elemento^2 para elemento : entero en valores ];
pepito : persona := { nombre => "Pepe Perez", edad => 33, casada => Verdadero };
```

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

- '-' unario.
- '++' y '--' (posincremento y posdecremento)
- '^' (potencia)
- '*', '/' y '\'
- '+' y '-'.
- '<-' y '->' (operadores de desplazamiento)
- '<', '>', '<=', '>=', '=' y '~='
- '~' (negación lógica)
- '/\' (and lógico)
- '\/' (or lógico)

Todos los operadores anteriores son binarios, excepto '-' unario, '++', '--' y '~', que son unarios. Respecto a la asociatividad, '^' es asociativo por la derecha, mientras que '-' unario, '++', '--', '~', '~' '<', '>', '<=', '>=', '=' y '~=' **no** son asociativos. El resto son asociativos por la izquierda.

A la hora de diseñar las reglas para los operadores, podeis implementar la precedencia y asociatividad diseñando una gramática no ambigua; alternativamente, podeis implementar una gramática ambigua y definir las precedencias y asociatividades a través de la definición de los operadores en la zona de declaraciones[17]; o podeis combinar ambas aproximaciones. Si elegís la primera posibilidad, podeis seguir como ejemplo la gramática de las expresiones aritméticas que hemos usado en alguna ocasión en los ejemplos sobre gramáticas LR(k) vistos 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. De este modo, el '-' unario, '++' y '--' podrían implementarse, manteniendo también el orden de precedencia expresado más arriba, como:

[17] En ese caso puede que sea necesario que declarar un nuevo token para el '-' unario. Podeis hacerlo.

```
expresion_potencia ::= expresion_posfija [ '^' expresion_potencia ]?

expresion_posfija ::= expresion_unaria [ operador_posfijo ]?

operador_posfijo ::= '++' | '--'

expresion_unaria ::= [ '-' ]? primario
```

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 Chusco 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. Subprogramas
4. Tipos, clases y objetos
5. Programas y librerías

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. 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 vacía para el símbolo `expresion`, mientras su compañero no escriba las reglas correspondientes.

Con respcto a la implementación de las reglas de la gramática, a estas alturas deberíais saber como implementar una gramática independiente del contexto con las indicaciones que os he dado. Por si no es el caso, a continuación teneis ejemplos de como implementar las diferentes estructuras que hemos usado en nuestra notación pseudo-BNF.

Símbolos que aparecen opcionalmente:

```
s ::= A [ B ]? C  ⇒  s : A C
                        | A B C
                        ;
```

o

```
s : A a C
  ;
a : B
  |
  ;
```

Símbolos que aparecen 1 (respectivamente 0) o más veces:

```
s ::= [ B ]+  ⇒  s : s B
                   | B
                   ;
```

```
s ::= [ B ]*  ⇒  s : s B
                   |
                   ;
```

Símbolos que aparecen 1 (respectivamente 0) o más veces, pero separados por ',':

```
s ::= ( A )+  ⇒  s : s ',' A
                   | A
                   ;
```

```
s ::= ( A )*  ⇒  s : a
                   |
                   ;
                 a : a ',' A
                   | A
                   ;
```

2.9. Depuración de la gramática

Dado que, como se establece en la siguiente sección, teneis que volcar en la consola (o fichero) las reglas que se van reduciendo, podeis usar esa información para depurar la gramática. Si ello no es suficiente, os recomiendo que useis la macro `YYDEBUG`, para lo que teneis que seguir dos pasos:

1. Declarar dicha macro en la sección de declaraciones de Bison.

```
%{

  #include <stdio.h>
  extern FILE *yyin;
  extern int yylex();

  #define YYDEBUG 1

%}
```

2. Dar a la variable `yydebug` un valor distinto a 0 en alguna parte del código C del analizador, por ejemplo en el programa principal.

```
int main(int argc, char *argv[]) {

  yydebug = 1;

  if (argc < 2) {
    printf("Uso: ./chusco NombreArchivo\n");
    }
  else {
    yyin = fopen(argv[1],"r");
    yyparse();
    }
  }
```

Activando la macro `YYDEBUG`, el analizador sintáctico listará por la consola, a medida que realiza el análisis de la entrada, los tokens que va leyendo de `yylex()`, las reglas que va reduciendo, los estados por los que va transitando y el contenido de la pila del analizador. Con esa información podeis ir al fichero '`chusco.output`' para ver en qué parte del autómata está el fallo, y hacer las correcciones correspondientes en la gramática.

2.10. Ejemplos

Os he dejado tres archivos de código Chusco en `chusco.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. Por ejemplo, el resultado de aplicar vuestro analizador a `ejemplo.chu`, debería ser parecido a esto[18]:

```
linea 1, palabra reservada: programa
linea 1, identificador: ejemplo
linea 1, delimitador: ;
  librerias -> epsilon
linea 3, palabra reservada: importar
linea 3, palabra reservada: libreria
linea 3, identificador: principal
  nombre -> ID
linea 3, operador: ::
linea 3, identificador: sistema_operativo
```

[18] Yo he abreviado los nombres de los símbolos en los `printf()` de las reglas en `chusco.y`, pero vosotros no teneis que hacerlo.

```
  nombre -> nombre '::' ID
linea 3, operador: ::
linea 3, identificador: entrada_salida
  nombre -> nombre '::' ID
linea 3, palabra reservada: como
linea 3, identificador: E_S
  como_id -> COMO ID
linea 3, delimitador: ;
  libreria -> LIBRERIA IMPORTAR nombre como_id ';'

...

linea 17, delimitador: ;
  instr_llamada -> llam_subpr ';'
  instr -> instr_llamada
  instrs -> instr
linea 18, palabra reservada: fin
  claus_excp_gen -> EXCEPCION instrs
  clauss_excp -> list_claus_excp_espec claus_excp_gen
  clauss -> clauss_excp
linea 18, palabra reservada: prueba
  instr_capt_excpt -> PRUEBA blq_instr clauss FIN PRUEBA
  instr -> instr_capt_excpt
  instrs -> instr
linea 19, palabra reservada: fin
  cuerpo_subpr ->  declrs PR instrs FIN
  cod_prog -> librerias cuerpo_subpr
  def_prog -> PROG ID cod_prog
EXITO: programa -> def_prog
```

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

Para que no tengáis que ir buscando fragmento a fragmento en el texto. Obviamente, faltan las definiciones de las reglas para operadores binarios en las expresiones, dado que sólo los he enumerado con sus precedencias y asociatividades.

```
****************************************PROGRAMA****************************************

  programa ::= definicion_programa | definicion_libreria

  definicion_programa ::= 'programa' IDENTIFICADOR ';' codigo_programa

  codigo_programa ::= [ libreria ]* cuerpo_subprograma

  libreria ::= 'importar' 'libreria' nombre [ 'como' IDENTIFICADOR ]? ';'
             | 'de' 'libreria' nombre 'importar' ( IDENTIFICADOR )+ ';'

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

  definicion_libreria ::= 'libreria' IDENTIFICADOR ';' codigo_libreria

  codigo_libreria ::= [ libreria ]* [ exportaciones ]? [ declaracion ]+

  exportaciones ::= 'exportar' ( nombre )+ ';'

  declaracion ::= declaracion_objeto | declaracion_tipo | declaracion_subprograma

****************************************OBJETOS*****************************************

  declaracion_objeto ::= ( IDENTIFICADOR )+ ':' 'constante' especificacion_tipo ':=' expresion ';'
                       | ( IDENTIFICADOR )+ ':' especificacion_tipo [ ':=' expresion ]? ';'

  especificacion_tipo ::= nombre | tipo_no_estructurado

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

  declaracion_tipo ::= 'tipo' IDENTIFICADOR 'es' tipo_no_estructurado ';'
                     | 'tipo' IDENTIFICADOR 'es' tipo_estructurado

  tipo_no_estructurado ::= tipo_escalar | tipo_tabla | tipo_diccionario

  tipo_escalar ::= [ 'signo' ]? tipo_basico [ longitud ]? [ 'rango' rango ]?

  longitud ::= 'corto' | 'largo'

  tipo_basico ::= 'booleano' | 'caracter' | 'entero' | 'real'

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

  tipo_tabla ::= 'tabla' '(' expresion '..' expresion ')' 'de' especificacion_tipo
               | 'lista' 'de' especificacion_tipo

  tipo_diccionario ::= 'diccionario' 'de' especificacion_tipo

  tipo_estructurado ::= tipo_registro | tipo_enumerado | clase

  tipo_registro ::= 'registro' [ campo ]+ 'fin' 'registro'
```

campo ::= (IDENTIFICADOR)+ ':' especificacion_tipo [':= expresion]? ';'

tipo_enumerado ::= 'enumeracion' ['de' tipo_escalar]? (elemento_enumeracion)+
'fin' 'enumeracion'

elemento_enumeracion ::= IDENTIFICADOR [':=' expresion]?

**********CLASES**********

clase ::= 'clase' ['ultima']? [superclases]? [declaracion_componente]+ 'fin' 'clase'

superclases ::= '(' (nombre)+ ')'

declaracion_componente ::= [visibilidad]? componente

visibilidad ::= 'publico' | 'protegido' | 'privado'

componente ::= declaracion_tipo
| declaracion_objeto
| (modificador)* declaracion_subprograma

modificador ::= 'constructor' | 'destructor' | 'generico' | 'abstracto' | 'especifico' | 'final'

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

declaracion_subprograma ::= 'subprograma' cabecera_subprograma cuerpo_subprograma 'subprograma'

cabecera_subprograma ::= IDENTIFICADOR [parametrizacion]? [tipo_resultado]?

parametrizacion ::= '(' [declaracion_parametros ';']* declaracion_parametros ')'

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

modo ::= 'valor' | 'referencia'

tipo_resultado ::= 'devolver' especificacion_tipo

cuerpo_subprograma ::= [declaracion]* 'principio' [instruccion]+ 'fin'

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

instruccion ::= instruccion_asignacion
| instruccion_devolver
| instruccion_llamada
| instruccion_si
| instruccion_casos
| instruccion_bucle
| instruccion_interrupcion
| instruccion_lanzamiento_excepcion
| instruccion_captura_excepcion
| ';'

instruccion_asignacion ::= objeto op_asignacion expresion ';'

op_asignacion ::= ':=' | ':+' | ':-' | ':*' | ':/' | ':\' | ':^' | ':<' | ':>'

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

```
  instruccion_llamada ::= llamada_subprograma ';'

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

  definicion_parametro ::= [ IDENTIFICADOR ':=' ]? expresion

  instruccion_si ::= 'si' expresion 'entonces' [ instruccion ]+
                     [ 'sino' [ instruccion ]+ ]? 'fin' 'si'

  instruccion_casos ::= 'casos' expresion 'es' [ caso ]+ 'fin' 'casos'

  caso ::= 'cuando' entradas '=>' [ instruccion ]+

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

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

  instruccion_bucle ::= [ IDENTIFICADOR ':' ]? clausula_iteracion [ instruccion ]+ 'fin' 'bucle'

  clausula_iteracion ::= 'para' IDENTIFICADOR [ ':' especificacion_tipo ]? 'en' expresion
                       | 'repetir' IDENTIFICADOR [ ':' especificacion_tipo ]?
                         'en' rango [ 'descendente' ]?
                       | 'mientras' expresion

  instruccion_interrupcion ::= 'siguiente' [ cuando ]? ';'
                             | 'salir' [ 'de' IDENTIFICADOR ]? [ cuando ]? ';'

  cuando ::= 'cuando' expresion

  instruccion_lanzamiento_excepcion ::= 'lanza' nombre ';'

  instruccion_captura_excepcion ::= 'prueba' [ instruccion ]+ clausulas 'fin' 'prueba'

  clausulas ::= clausulas_excepcion [ clausula_finalmente ]?
              | clausula_finalmente

  clausulas_excepcion ::= [ clausula_excepcion_especifica ]* clausula_excepcion_general

  clausula_excepcion_especifica ::= 'excepcion' '(' nombre ')' [ instruccion ]+

  clausula_excepcion_general ::= 'excepcion' [ instruccion ]+

  clausula_finalmente ::= 'finalmente' [ instruccion ]+

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

  expresion_potencia ::= expresion_posfija [ '^' expresion_potencia ]?

  expresion_posfija ::= expresion_unaria [ operador_posfijo ]?

  operador_posfijo ::= '++' | '--'

  expresion_unaria ::= [ '-' ]? primario
```

```
primario ::= literal
           | objeto
           | [ 'objeto' ]? llamada_subprograma
           | enumeraciones
           | '(' expresion ')'

literal ::= VERDADERO | FALSO | CTC_ENTERA | CTC_REAL | CTC_CARACTER | CTC_CADENA

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

enumeraciones ::= '[' expresion_condicional [ clausula_iteracion ]+ ']'
                | '[' ( expresion )+ ']'
                | '{' ( clave_valor )+ '}'
                | '{' ( campo_valor )+ '}'

expresion_condicional ::= expresion
                        | 'si' expresion 'entonces' expresion [ 'sino' expresion ]?

clave_valor ::= CTC_CADENA '=>' expresion

campo_valor ::= IDENTIFICADOR '=>' expresion
```