Practica 2.- Analizador Sintáctico de Micro

TALF 2020/21 (Julio)

25 de junio de 2021

Índice

1. Descripción de la práctica

Objetivo: El alumno deberá implementar un analizador sintáctico en Bison para el lenguaje de programación Micro, 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). 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 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-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: El plazo límite para subirla a Moovi es el mismo día del examen, 2 de julio de 2021, a las 23:59.

Material: He dejado en Moovi (TALF → Documentos y Enlaces → Material de prácticas → Práctica 1) un directorio comprimido, `micro.flex.tar.gz`, con los siguientes archivos:

- `micro.l`, en el que se puede escribir la esecificación del analizador léxico.
- `micro.y`, en el que se puede escribir la esecificación del analizador sintáctico.
- `prueba1.mi` y `prueba2.mi`, archivos con código Micro para probar el analizador.
- `Makefile`, para compilar la especificación del analizador y generar un ejecutable, de nombre `micro`.

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

- 0'5 por declaraciones y tipos.
- 0'3 por la definición de subprogramas.
- 0'5 por las instrucciones.
- 0'9 por las expresiones (0'45 si se emplean precedencias y asociatividades).
- 0'3 por la gestión de errores.

Para sumar 0'9 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 0'45 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 Micro

En esta sección vamos a proporcionar la especificación de la gramática de Simple. 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, con `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_objeto ::= ( IDENTIFICADOR )+ ':' tipo_compuesto
  ```

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

2.1. Especificación de un programa Micro

Un programa Micro 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. Si estamos definiendo variables de tipo escalar (entero, real booleano o carácter) podemos especificar el valor inicial 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 ]? ';'
                     | ( IDENTIFICADOR )+ ':' nombre_de_tipo ';'
                     | ( IDENTIFICADOR )+ ':' tipo_compuesto ';'

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

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

También podemos declarar objetos (variables) con tipos compuestos o tipos definidos previamente, sólo que en este caso no podremos dar valores iniciales a dichas variables. 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 (`tipo_registro`) o una lista asociativa (`tipo_hashtable`).

```
nombre_de_tipo ::= IDENTIFICADOR

tipo_compuesto ::= tipo_tablero | tipo_registro | tipo_hashtable
```

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

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. A su vez, una especificación de tipo puede ser un tipo escalar, un nombre de tipo o un tipo compuesto. Esto último nos permite tener arrays multidimensionales, que serían declarados como arrays de arrays.

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

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

Ejemplos:

```
N: constant INTEGER := 10;
lista : array (1..100) 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 ';'

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

declaracion_tipo ::= 'type' IDENTIFICADOR 'is' 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.

Finalmente, una declaración de tipo está formada por un la palabre reservada '`type`', seguida de un identificador (el nombre del tipo), la palabra reservada '`is`' y una especificación de tipo, y terminando en el delimitador '`;`'.Por ejemplo:

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

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

2.2. 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 esta con la definición del tipo del valor de retorno, lo que se consigue con 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;
```

2.3. Instrucciones

Vamos a considerar los siguientes tipos de instrucciones:

```
instruccion ::= instruccion_vacia
              | instruccion_asignacion
              | instruccion_exit
              | instruccion_return
              | instruccion_if
              | instruccion_case
              | instruccion_loop
              | 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           -> null;
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 llamada a procedimiento: Está formada por un identificador (el nombre del procedimiento) seguido, opcionalmente, de los parámetros del mismo. Dichos parámetros son una secuencia de una o más expresiones, separadas por comas. La instrucción termina en el delimitador '`;`'.

```
llamada_procedure ::= IDENTIFICADOR [ parametros ]? ';

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

2.4. 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 una llamada a función, un componente indexado, el valor asociado a una clave en una lista hash, el valor del campo de un registro o una variable (un identificador).

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

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

nombre ::= llamada_funcion
         | componente_indexado
         | componente_hash
         | componente_campo
         | IDENTIFICADOR
```

Una llamada a función se escribe como un identificador (el nombre de la función) seguido de los parámetros de la misma: cero o más expresiones separadas por comas, entre paréntesis. Por su parte, un componente indexado puede ser un `componente_recursivo` seguida de una expresión entre corchetes (la sintaxis para acceder al valor de una posición en una tabla). Dicho `componente_recursivo`, a su vez, puede ser una llamada a función, componente indexado o el campo de un registro, dado que podemos almacenar tablas en todas esas estructuras.

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

componente_indexado ::= componente_recursivo '[' expresion ']'

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

componente_recursivo ::= llamada_funcion | componente_indexado | componente_campo

componente_campo ::= nombre '.' IDENTIFICADOR
```

Por su parte, un `componente_hash` permite derivar un `nombre` seguido de una expresión entre llaves, lo que constituye la sintaxis de acceso al elemento de una tabla hash a partir de su clave. Finalmente, un `componente_campo` está formado por un `nombre` seguido de un punto y un identificador (el nombre del campo al que se está accediendo).

Fijaos que al derivar `nombre` desde `componente_hash` y `componente_campo` tenemos recursividad indirecta, dado que éstos dos últimos símbolos también se pueden derivar desde `nombre` en las reglas vistas más arriba.

A partir de aquí, teneis 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, mientras que '`**`', '`=`', '`/=`', '`<`', '`>`', '`<=`' y '`>=`' sin asociativos 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 (0'45 en lugar de 0'9). 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:

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

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`'.

2.5. 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 Micro 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.

Con respecto 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       s : Aa C
                     | A B C  o      ;
                     ;              a : B
                                      |
                                      ;
```

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

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

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

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

2.6. 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;

     ...
     }
   ```

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 '`micro.output`' para ver en qué parte del autómata está el fallo, y hacer las correcciones correspondientes en la gramática.

2.7. Ejemplo

Os he dejado dos programas de ejemplo (`prueba1.mi` y `prueba2.mi`) en el archivo `micro.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.mi`, debería ser parecido a esto:

```
Linea 1 - Palabra Reservada: procedure
Linea 1 - Identificador: ORDENAR_POR_MONTONES
Linea 1 - Palabra Reservada: is
   spec_suprg -> 'procedure' ID
Linea 2 - Identificador: N
   ids -> IDENTIFICADOR
Linea 2 - Delimitador: :
Linea 2 - Palabra Reservada: constant
   const -> constant_token
Linea 2 - Palabra Reservada: INTEGER
```

```
   tipo_escalar -> INTEGER
Linea 2 - Operador: :=
Linea 2 - Entero: 10
   literal -> CTC_INT
   primario -> literal
   negacion -> primario
Linea 2 - Delimitador: ;
   factor -> negacion
   expr_mult -> factor
   expr_ad -> expr_mult
   expr_bin -> expr_ad
   expr_rel -> expr_bin
   expr_neg -> expr_rel
   expr_conj -> expr_not
   expr_dis -> expr_conj
   expr -> expr_dis
   exprs -> expr
   asig -> ':=' exprs
   declr_obj -> ids ':' const tipo_escalar asig ';'
   declr -> declr_obj
Linea 3 - Identificador: TABLA
   ids -> IDENTIFICADOR
Linea 3 - Delimitador: :
Linea 3 - Palabra Reservada: array
Linea 3 - Delimitador: (
Linea 3 - Entero: 1
   literal -> CTC_INT
   primario -> literal
   negacion -> primario
Linea 3 - Delimitador: ..
   factor -> negacion
   expr_mult -> factor
   expr_ad -> expr_mult
   expr_bin -> expr_ad
   expr_rel -> expr_bin
   expr_neg -> expr_rel
   expr_conj -> expr_not
   expr_dis -> expr_conj

...
```

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 debeis evitar la introducción de nuevas ambigüedades derivadas de las reglas de tratamiento de error.

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

He reunido en un 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 Y TIPOS*********************************

declaracion ::= declaracion_objeto
              | declaracion_tipo
              | declaracion_subprograma

declaracion_objeto ::= ( IDENTIFICADOR )+ ':' [ 'constant' ]? tipo_escalar [ asignacion ]? ';'
                     | ( IDENTIFICADOR )+ ':' nombre_de_tipo ';'
                     | ( IDENTIFICADOR )+ ':' tipo_compuesto ';'

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

asignacion ::= ':=' ( expresion )+

nombre_de_tipo ::= IDENTIFICADOR

tipo_compuesto ::= tipo_tablero | tipo_registro | tipo_hashtable

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

especificacion_tipo ::= tipo_escalar | nombre_de_tipo | tipo_compuesto

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

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

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

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

**************************************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
              | llamada_procedure

instruccion_vacia: 'nil' ';'

instruccion_asignacion: nombre ':=' expresion ';'

instruccion_return: 'return' expresion ';'

llamada_procedure ::= IDENTIFICADOR [ parametros ]? ';

parametros ::= '(' ( 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'

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

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

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

nombre ::= llamada_funcion
         | componente_indexado
         | componente_hash
         | componente_campo
         | IDENTIFICADOR

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

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

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

componente_recursivo ::= llamada_funcion | componente_indexado | componente_campo

componente_campo ::= nombre '.' IDENTIFICADOR
```