PRÁCTICA DE PROCESADORES DEL LENGUAJE I

Curso 2021 – 2022

Entrega de Febrero

APELLIDOS Y NOMBRE: ubikitina

IDENTIFICADOR: ubikitina

Índice

1. El analizador léxico

Está compuesto por el código de usuario, directivas JLex y reglas patrón-acción.

1.1. Código de Usuario

Esta sección contiene los paquetes e importaciones proporcionados por el Equipo Docente por defecto. No se ha añadido más contenido para el desarrollo de esta práctica.

1.2. Directivas JLex

En esta sección se han incluido los estados generados aparte de YYINITIAL:

```
%state COMENTARIO
%state CADENA_SIN_COMILLAS
```

Además, entre el código encerrado en %{ y }%, el cual es el código que se copiará al inicio de yyLex, se incluye el contador stringCount, además del commentCount que ya venía por defecto, para controlar que las aperturas y cierres de las cadenas de caracteres estén balanceados:

```
private int stringCount = 0;
```

Adicionalmente, también en esta sección entre %{ y }%, se incluye la función createToken, de modo que pueda ser posteriormente utilizado en la sección de reglas de patrón-acción, y de este modo, la escritura de las posteriores reglas sea más simple:

```
Token createToken (int t) {
     Token token = new Token(t);
     token.setLine (yyline + 1);
     token.setColumn (yycolumn + 1);
     token.setLexema (yytext ());
     return token;
  }
```

Así mismo, se ha añadido un apartado entre %eof{ y %eof}para incluir las sentencias que serán ejecutadas tras encontrar EOF (*end of file*):

```
%eof{
       if(commentCount != 0){
             LexicalError error = new LexicalError ("Comentario mal balanceado o
anidamiento incorrecto.");
         error.setLine (yyline + 1);
         error.setColumn (yycolumn + 1);
         error.setLexema (yytext ());
         lexicalErrorManager.lexicalError (error);
         System.exit(-1); //detenemos ejecución
       }

       if(stringCount != 0){
             LexicalError error = new LexicalError ("Apertura de cadena sin cierre.");
         error.setLine (yyline + 1);
         error.setColumn (yycolumn + 1);
         error.setLexema (yytext ());
         lexicalErrorManager.lexicalError (error);
         System.exit(-1); //detenemos ejecución
       }
%eof}
```

Estas sentencias son comprobaciones de los dos contadores definidos. Si el contador commentCount es diferente a 0, emitirá un error con el mensaje `"Comentario mal balanceado o anidamiento incorrecto."`

y detendrá la ejecución. Por otra parte, si el contador stringCount es diferente a 0, emitirá un error con el mensaje `"Apertura de cadena sin cierre."` y detendrá la ejecución.

Siguiendo con la sección de directivas JLex, en esta sección también se han añadido las definiciones de macros:

```
LETRA = [A-Za-z]
DIGITO=[0-9]
NUMERO = 0|[1-9]({DIGITO})*
NUMERO_ERRONEO = (-{NUMERO}) | (-?(0+{NUMERO}) | (-?{NUMERO}*\.{NUMERO}+))
IDENTIFICADOR = {LETRA}({LETRA}|{DIGITO})*
IDENTIFICADOR_ERRONEO = {NUMERO}({LETRA}|_)({LETRA}|_|{NUMERO})*
ESPACIO_BLANCO=[ \t\r\n\f]
CADENA = ({LETRA}|{ESPACIO_BLANCO}|{DIGITO})*
fin = "fin"{ESPACIO_BLANCO}
```

Para la definición de estas macros se han seguido las directivas del enunciado.

1.3. Reglas patrón-acción

Para el desarrollo de esta práctica se han definido tres estados, y dentro de cada una, se han incluido las correspondientes expresiones regulares y acciones java a ejecutar.

Estado <YYINITIAL>
Se trata del estado inicial. Contiene el desarrollo de las siguientes expresiones regulares o patrones y sus correspondientes acciones:

Patrón	Acción
"("	Crea el token PARENTESISIZQ
")"	Crea el token PARENTESISDCH
"{"	Crea el token LLAVEIZQ
"}"	Crea el token LLAVEDCH
"["	Crea el token CORCHETEIZQ
"]"	Crea el token CORCHETEDCH
"/*"	Suma 1 al contador de apertura de comentarios y va al estado <COMENTARIO>.
"*/"	Crea un error con el mensaje “Cierre de comentario antes de apertura.” y detiene la ejecución.
":"	Crea el token DOSPUNTOS
","	Crea el token COMA
";"	Crea el token PUNTOCOMA
"*"	Crea el token PRODUCT
"<"	Crea el token MINOR
"=="	Crea el token EQUAL
"&&"	Crea el token AND
"!"	Crea el token NOT
"++"	Crea el token AUTOINCREMENT
"="	Crea el token ASSIGN
"+="	Crea el token ASSIGNWITHSUM
"+"	Crea el token PLUS
"caso"	Crea el token CASE
"corte"	Crea el token BREAK
"entero"	Crea el token INT
"escribe"	Crea el token WRITE
"escribeEnt"	Crea el token WRITEINT
"alternativas"	Crea el token ALTERNATIVE
"mientras"	Crea el token WHILE
"pordefecto"	Crea el token DEFAULT

Patrón	Acción
"principal"	Crea el token MAIN
"devuelve"	Crea el token RETURN
"si"	Crea el token IF
"sino"	Crea el token ELSE
"tipo"	Crea el token TYPE
"vacio"	Crea el token VOID
"#constante "	Crea el token CTE
"\""	Suma 1 al contador de comillas de cadenas y va al estado <CADENA_SIN_COMILLAS>.
{IDENTIFICADOR}	Crea el token ID
{IDENTIFICADOR_ERRONEO}	Crea un error con el mensaje "Identificador incorrecto." y detiene la ejecución.
{NUMERO_ERRONEO}	Crea un error con el mensaje "Número incorrecto." y detiene la ejecución.
{NUMERO}	Crea el token NUM
{ESPACIO_BLANCO}	Es ignorado.
{fin}	Es ignorado.
[^]	Es para el caso de coincidir con ningún patrón. Crea un error con el mensaje "No coincide con ningún patrón." y detiene la ejecución.

Estado <COMENTARIO>

Se trata de un estado que se alcanza tras reconocer una apertura de comentario /*. Este estado ignora el contenido del comentario y, además, gestiona el anidamiento de comentarios. Contiene el desarrollo de las siguientes expresiones regulares o patrones y sus correspondientes acciones:

Patrón	Acción
"/*"	Suma 1 al contador de apertura de comentarios.
"*/"	Resta 1 del contador de apertura de comentarios. Además, si el contador es igual a 0, redirige el flujo de ejecución al estado YYINITIAL.
[^]	Ignora todo lo que no coincida con los dos patrones anteriores (es decir, ignora el contenido del comentario).

Estado <CADENA_SIN_COMILLAS>

Se trata de un estado que se alcanza tras reconocer la primera comilla " de una cadena de caracteres. Este estado crea el token STRING con el contenido de la cadena de caracteres, garantiza que las comillas de las cadenas de caracteres estén correctamente balanceadas y en caso de detectar caracteres no permitidos, lanza error. Contiene el desarrollo de las siguientes expresiones regulares o patrones y sus correspondientes acciones:

Patrón	Acción
{CADENA}*([^A-Za-z0-9 \t\f\"])+([^A-Za-z0-9 \t\f\"]|{CADENA})*	Es para el caso de que una cadena de caracteres contenga caracteres no admintidos. Crea un error con el mensaje "Carácter(es) no permitido(s) en la cadena." y detiene la ejecución.
{CADENA}	Crea el token STRING
"\""	Resta 1 del contador de comillas de cadenas. Además, si el contador es igual a 0, redirige el flujo de ejecución al estado YYINITIAL.
[^]	Ignora todo lo que no coincida con los patrones anteriores.

2. El analizador sintáctico

El trabajo ha consistido en completar el fichero parser.cup, en el cual en total se han establecido:

- 38 terminales
- 41 no-terminales
- 113 producciones

Todas ellas producen 251 estados únicos en el autómata. El axioma de la gramática se ha mantenido axiom, tal y como venía establecido en el código de partida de la práctica.

Además, se han añadido reglas de precedencia y asociatividad de operadores siguiendo las directrices del enunciado:

```
precedence left     AND;
precedence left     EQUAL;
precedence left     MINOR;
precedence left     PLUS;
precedence left     PRODUCT;
precedence left     AUTOINCREMENT, NOT;
precedence left     LLAVEIZQ, LLAVEDCH;
precedence left     PARENTESISIZQ, PARENTESISDCH;
precedence right    ELSE;
```

Y también se han añadido reglas de gestión de errores, para mostrar más detalle sobre la construcción sintáctica que no se ha ajustado a las especificaciones gramaticales del lenguaje. A continuación, un ejemplo de los errores añadidos:

```
funcionPrincipal ::= VOID MAIN PARENTESISIZQ PARENTESISDCH bloque
                   | VOID error {: syntaxErrorManager.syntaxInfo("No se ha especificado la
palabra principal en la función principal."); :} PARENTESISIZQ PARENTESISDCH bloque;
```

Para más detalle sobre los terminales, no-terminales y producciones del analizador sintáctico, ver sección *4. Gramática* de esta memoria.

3. Conclusiones

Desde mi punto de vista como alumna, esta práctica es un ejercicio muy completo que ayuda a asimilar conceptos básicos de la asignatura Procesadores de Lenguaje I en cuanto a la estructura de un compilador y sus fases iniciales de análisis léxico y sintáctico. Cabe destacar que desde el inicio de la práctica se trabaja el concepto token como entidad léxica indivisible, y la descripción de la gramática mediante el flujo de tokens para la parte del sintáctico.

El enunciado está muy detallado, de modo que se puedan listar la gran mayoría de las especificaciones léxicas y sintácticas concretas del lenguaje cES como punto de partida, y después realizar el desarrollo del analizador léxico y sintáctico correspondientes. Para aquellos detalles en los que el enunciado podía resultar ambiguo, a continuación listo algunas de las interpretaciones que he realizado:

- En la sección *2.2.1 Estructura de un programa y ámbitos de visibilidad* habla sobre el comienzo del programa. Detalla que un programa comienza con el conjunto de constantes simbólicas necesarias, seguido de secciones para la declaración de variables y tipos globales y la declaración de funciones. En este caso, el enunciado no especifica el orden de aparición de la declaración de variables y tipos globales. Sin embargo, más adelante en la sección *2.2.5 Declaración de funciones* indica que dentro de las funciones primero aparece la declaración de tipos y variables locales a la misma (en este orden), haciendo hincapié en el orden. Desde mi punto de vista, creo que tendría sentido respetar este orden en ambos casos o en ninguno. Sin embargo, para la implementación del código me he ceñido al enunciado y he programado de modo que el analizador sintáctico permita un orden indiferente al comienzo del programa, y sin embargo, respete el orden en las funciones, primero apareciendo la declaración de tipos y después variables.
- En cuanto al uso de las llaves en si-sino y en alternativas-caso, he supuesto que las llaves se pueden omitir siempre y cuando a continuación de la llave venga una sola sentencia. Si hay más de una sentencia, las llaves son necesarias. Me he basado en la sintaxis de C para tomar esta decisión, dado que cES es una variante de C.
- En cuanto al constante de los casos de la sentencia alternativas-caso, el enunciado indica que tienen que ser constantes numéricas. En este caso, me surgía la duda de qué token tenía que ser. Tenía claro

que el token número (NUM) era válido y que el token identificador (ID) también, ya que es el nombre que se asigna a las constantes. Sin embargo, me surgía la duda de si las expresiones (como por ejemplo 3+5) eran válidas, debido a que dan como resultado un valor numérico. He decidido incluir estas expresiones también, optando por no limitar la gramática y seguir adelante con la solución menos restrictiva.

Durante el desarrollo de la práctica me gustaría destacar que aparte de la estructura general y funcionamiento de un analizador léxico y sintáctico, también he profundizado en otros conceptos como las relaciones de precedencia y asociatividad y cómo implementarlos en una gramática, o la diferencia entre los conflictos (reducción-reducción y reducción-desplazamiento) y los errores sintácticos. He aprendido y trabajado la teoría de estos conceptos, además de su implementación en la práctica.

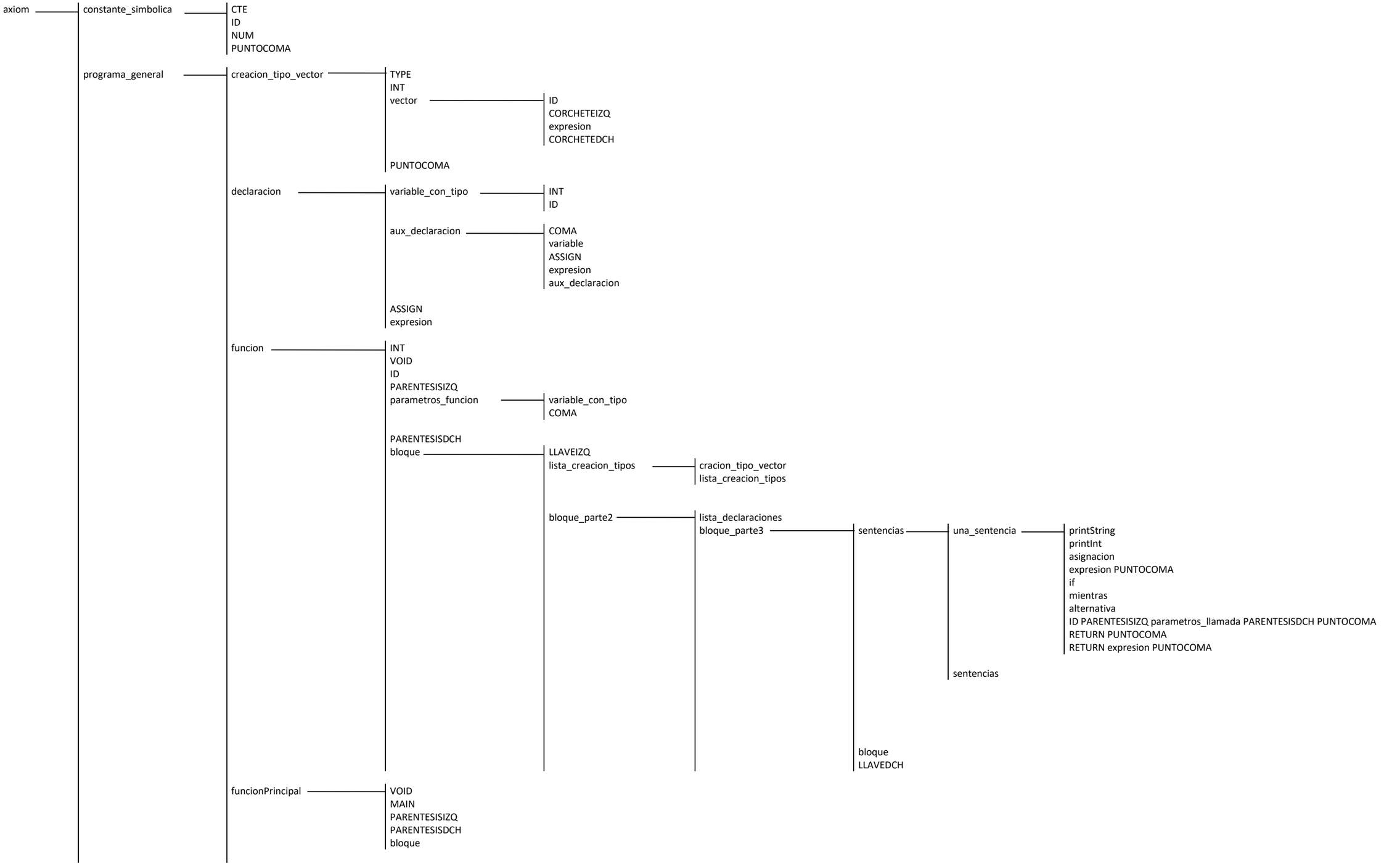
Me gustaría destacar que la implementación del tratamiento de errores en el analizador sintáctico me ha resultado costosa. Siguiendo las recomendaciones de mi tutor, he abordado esta parte al final del desarrollo, ya que era necesario ampliar la gramática para implementarlos. Durante la ampliación de mi gramática con errores he comprendido que la palabra "error" al fin y al cabo es un no terminal más que hay que incorporarlo a la gramática y que puede generar conflictos al igual que otro no terminal. En cada regla existen varias opciones de errores diferentes y la implementación de todos los errores posibles resulta muy larga y laboriosa. Por lo que he decidido realizar una implementación parcial, para demostrar que he trabajado con el concepto. Como futuras líneas de trabajo o mejoras, se podría seguir trabajando en esto y realizar una implementación exhaustiva de los errores.

4. Gramática

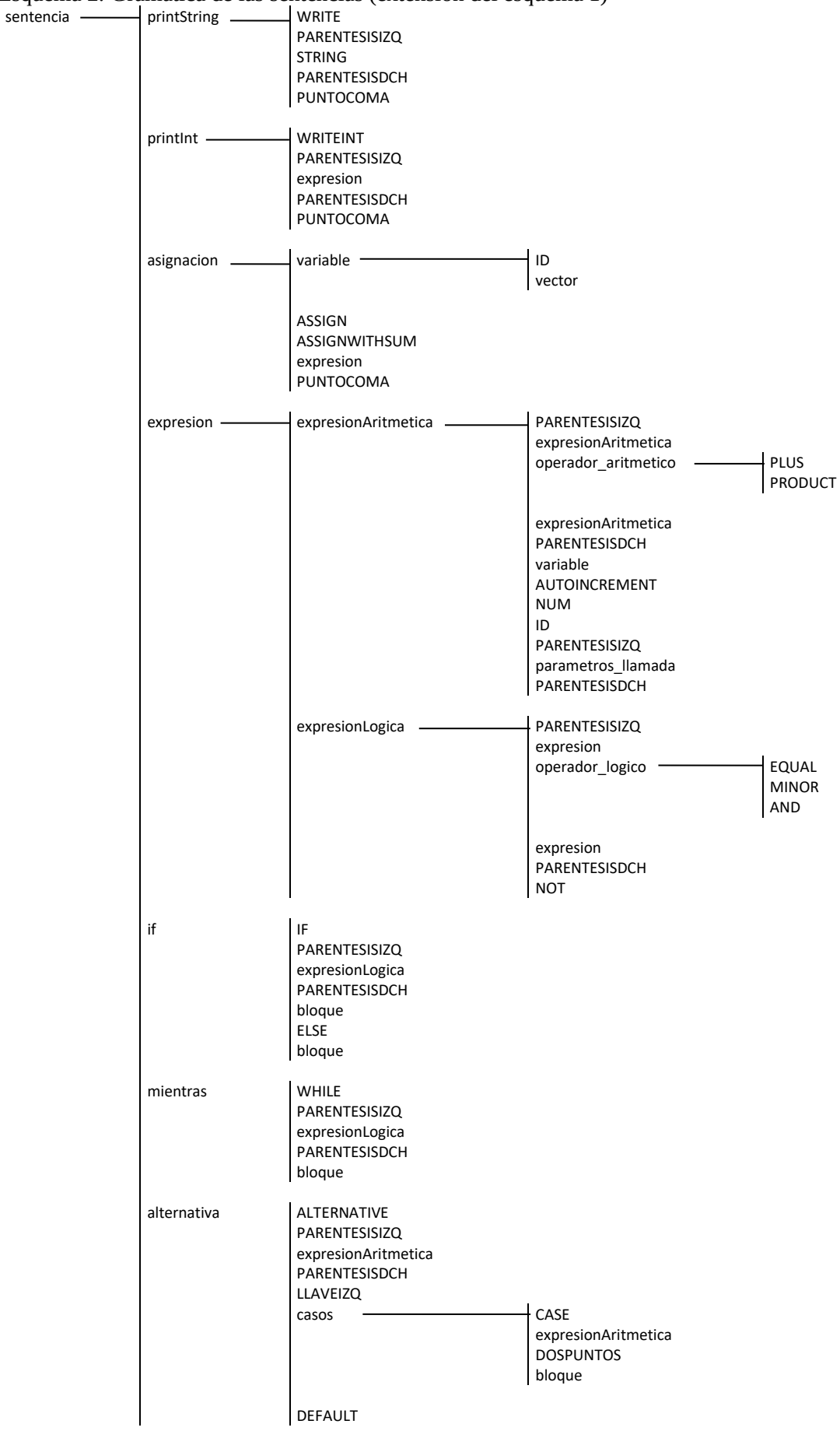
A continuación se muestra la representación de las las producciones de la gramática generada en dos esquemas. El primer esquema corresponde a la estructura general del programa, y el segundo esquema profundiza en las sentencias.

Esto esquemas son aproximaciones del código que compone la gramática completa del fichero parser.cup, ya que la recursividad y alternativas opcionales de la gramática completa dificultan su exposición al completo. Para la consulta de la gramática completa en detalle, referirse al archivo parser.cup.

Esquema 1: Estructura general del programa con terminales en mayúscula y no terminales en minúscula.



Esquema 2: Gramática de las sentencias (extensión del esquema 1)



	bloque BREAK PUNTOCOMA LLAVEDCH