Análisis lexicográfico

Funcionamiento de un analizador léxico.

El *analizador lexicográfico*, o *explorador*, es la parte del compilador que lee el programa fuente, carácter a carácter, y construye a partir de éste unas entidades primarias llamadas *tokens*. Es decir, el analizador lexicográfico transforma el programa fuente en tiras de tokens.

Ejemplo: Sea la sentencia:

```
IF alfa< 718 THEN alfa := alfa + beta
```

El analizador lexicográfico daría como resultado la siguiente cadena de caracteres correspondiente a la misma

```
(79);(12);(28);(13);(80);(12);(65);(12);(34);(12)
```

Como puede verse, el analizador lexicográfico ha simplificado el texto de entrada, consistente en una secuencia de símbolos, en otra cadena de caracteres, representados cada uno de ellos por un número, y que corresponden a los siguientes significados:

```
12   Variable real13   Constante entera28   Comparador < 34   Signo +65   Asignador79   Palabra reservada IF80   Palabra reservac
```

La información que da esta secuencia de caracteres es suficiente para el análisis de la estructura de la sentencia, pero no basta para un análisis de su significado, para lo cual hay que tener cierta información de cuáles son las variables que entran en juego en esta sentencia. Esto se soluciona mediante un segundo elemento que compone el token, que por lo general consiste en un puntero a la tabla de símbolos en donde se hallan almacenadas las variables, quedando finalmente una secuencia de pares:

(79,-);(12,32);(28,-);(13,7);(80,-);(12,32);
(65,-);(12,32);(34,-);(12,33)

En estos pares el primer elemento nos indica el tipo de objeto que estamos procesando según una tabla que nosotros hemos diseñado previamente (tabla de tokens). El segundo miembro de estos pares es, en caso necesario, un puntero a la tabla de símbolos o a la tabla de constantes.

El analizador lexicográfico, además, realiza ciertas tareas adicionales como:

- Eliminar comentarios del programa fuente.
- Eliminar los blancos, saltos de página, tabuladores, retornos de carro, y demás caracteres propios del dispositivo de entrada.
- Reconocer las variables y asignarles una posición en la tabla de símbolos.
- Relacionar los mensajes de error que produce el compilador en sus diversas fases con la parte correspondiente del programa fuente (número de línea en que aparecen). En ciertos casos el analizador lexicográfico se encarga también de producir el listado del programa con los errores de compilación.
- Si el programa fuente incorpora metanociones o *macros*, el analizador lexicográfico puede incorporar un preprocesador.
- Avisa de los errores lexicográficos que detecta.

Hay diversas razones por las que se separa la fase de análisis de un compilador en **análisis lexicográfico** y **análisis sintáctico**. Son las siguientes:

1. El diseño del analizador sintáctico es más fácil de esta forma, ya que éste no ha de preocuparse de leer el fichero de entrada, ni de saltar blancos, ni comentarios, ni de recibir caracteres inesperados, puesto que todo ello ha sido filtrado previamente por el analizador lexicográfico. Por tanto, el diseño consiguiente se hace más claro y comprensible.
2. Se mejora la eficiencia del compilador en su conjunto. La lectura del programa fuente suele requerir gran parte del tiempo de compilación, que se ve reducido si el analizador lexicográfico incorpora técnicas especiales de lectura, o está realizado en ensamblador.
3. Aumenta la portabilidad del compilador, ya que todas las diferencias que se produzcan en el alfabeto de entrada, o en el dispositivo de almacenamiento, pueden ser reducidas al analizador lexicográfico, dejando al analizador sintáctico intacto. Supóngase que el compilador lee un texto ASCII y nos interesa construir uno que sea capaz de leer un texto escrito bajo el estándar EBCDIC. Bastaría con modificar el analizador lexicográfico.
4. Se pueden resolver fácilmente casos dudosos como la siguiente sentencia FORTRAN:

   DO5I=1.25 (compárese con DO5I=1,25)

   Hasta que no se reconoce el carácter "`.`" o el carácter "`,`" no podemos saber si se trata de una sentencia de asignación o de una instrucción `DO`.

Para comprender mejor por qué se realiza un explorador veamos el siguiente ejemplo, que corresponde a una definición muy restringida del lenguaje de programación PASCAL:

```
<SENT>    -> if <EXP-BOOL> then <SENT> [else <SENT>]<SENT>    -> <VAR> : = <EXP><EXP-BOOL>-> <VAR-BOOL> | <EXP> <COMP> <EXP><COMI
```

En esta gramática serían símbolos terminales los siguientes:

T = {, ,a,b,c,... z,(,),+,-,*,/,<,>,=,:}

y los símbolos no terminales que se han empleado son:

```
N = {<SENT>,<EXP-BOOL>,<VAR>,<EXP>,<VAR-BOOL>,<COMP>,<OP>,     <FACTOR>,<CTE>,<LETRA>,<ALFANUM>,<DIGITO>}
```

Esta gramática puede ser descompuesta o transformada en otras dos, como éstas:

- Primera:

```
<PRES-IF>   -> if<PRES-THEN> -> then<PRES-ELSE> -> else<COMP>      -> < | > | <> | <= | >= | =< | => | =<CTE>       -> <DIGITO> [
```

cuyos símbolos terminales y no terminales serían:

T = {, ,a,b,c,... z,(,),+,-,*,/,<,>,=,:}

```
N = {<PRES-IF>,<PRES-THEN>,<PRES-ELSE>,<COMP>,<CTE>,<VAR>,     <VAR-BOOL>,<LETRA>,<DIGITO>,<ALFANUM>,<OP>,<ASIG>,     <ABREPAR>,<
```

- Segunda:

  ```
  <SENT>     -> <pres-if> <EXP-BOOL> <pres-then> <SENT>              [<pres-else> <SENT>]<SENT>     -> <var> <asig> <EXP><EXP-BOOL>
  ```

  cuyos símbolos terminales y no terminales serían:

  ```
  T = {<pres-if>,<pres-then>,<pres-else>,<var>,<asig>,     <var-b>,<comp>,<op>,<abrepar>,<cierrapar>,<cte>}
  ```

  N = {<SENT>,<EXP-BOOL>,<EXP>,<FACTOR>}

La primera de estas dos gramáticas es la que utiliza el *explorador* o analizador lexicográfico, mientras que la segunda corresponde a la gramática del *parser* o analizador sintáctico.

Como puede apreciarse, la primera gramática es de *tipo 3* según la clasificación de Chomsky (es decir, es una gramática regular), mientras que la segunda es una gramática de contexto libre (o sea, de tipo 2).

Debe hacerse notar que las metanociones (símbolos no terminales) de la gramática del analizador lexicográfico, o bien no aparecen en la gramática del analizador sintáctico, o son símbolos terminales en dicha gramática.

Para la implementación de un analizador lexicográfico existen en la actualidad tres vías. La principal ventaja de cada una de ellas respecto a las siguientes es la mayor comodidad y facilidad de programación. Por el contrario, la implementación en un lenguaje más cercano a la máquina hace posible una mejor gestión de las entradas/salidas, con la consiguiente rapidez de ejecución.

- Escribir el analizador lexicográfico en lenguaje ensamblador directamente.
- Escribir el analizador lexicográfico en un lenguaje de alto nivel, usando las facilidades de entrada/salida que éste tenga.
- Usar un generador de analizadores lexicográficos, capaz de producir un analizador lexicográfico a partir de las expresiones regulares que definen los distintos tokens. En este caso, el generador incorpora sus propias rutinas de entrada/salida. Uno de los generadores de analizadores lexicográficos cuyo uso está más generalizado es el programa **Lex**.

Análisis lexicográfico

Lexemas, expresiones regulares y tokens.

Para ver las diferencias entre los conceptos de lexema, expresión regular y token, veamos algunos ejemplos de ellos para la siguiente gramática:

```
sent -> ID ASIG exp ';'exp  -> exp '+' expexp  -> exp '*' expexp  -> NUMENTEROexp  -> NUMREALexp  -> IDENT
```

Token	Lexema	Expr. regular	Expr. Reg. LEX
IDENT	perimetro radio pi hola	(a+...+z+_)(a+...+_+0+...+9)*	[a-zA-Z_][a-zA-Z0-9_]
ASIG	:=	:=	:=
NUMENTERO	47 1234567890	(0+...+9)(0+...+9)*	[0-9]+
PR_FOR	for For FOR	(F+f)(O+o)(R+r)	[Ff][Oo][Rr]

Las expresiones regulares en Lex se explicarán más adelante en el apartado II.4.

Análisis lexicográfico

LEX

Lex es un generador de analizadores léxicos. Cada vez que Lex encuentra un lexema que viene definido por una expresión regular, se ejecutan las acciones (escritas en C) que van al lado de la definición de dicha expresión.

Lex crea `yylex`, una variable que contendrá un número, el cual se corresponde con el token de cada expresión regular. También, instancia una variable global `yytext`, que contiene el lexema que acaba de reconocer.

Así, por ejemplo, para el siguiente código fuente de entrada:

```
%%expresión1    {acción1}expresión2     {acción2}...            ...expresiónn   {acciónn}%%
```

Lex genera un programa en C (generalmente denominado `lex.yy.c`) que incluye, entre otras, la función `yylex()`:

```
int yylex(){  while(!eof())    {      switch(...)      {         case -1: ... ; break;         case  0: ... ; break;         case  1: {acci
```

Esta función recorre el texto de entrada. Al descubrir algún lexema que se corresponde con alguna expresión regular, construye el token, y realiza la acción corespondiente. Cuando se alcanza el fin de fichero, devolverá `-1`. La función `yylex()` podrá ser invocada desde cualquier lugar del programa.

Cómo escribir expresiones regulares en Lex. Deberán cumplir los siguientes requisitos:

- Las expresiones regulares (*ER*, de aquí en adelante) han de aparecer en la primera columna.
- *Alfabeto de entrada*: Caracteres ASCII `0` al `127`.
- *Concatenación*: Sin carácter especial, se ponen los caracteres juntos.
- *Caracteres normales*: Se representan a ellos mismos.
- *Caracteres especiales*: Se les pone la barra '\' delante. Éstos son:

  * + ? | [ ] ( ) " \ . { } ^ $ / < >

- *Caracteres especiales dentro de los corchetes* ('`[`' y '`]`'):

  - \ ^

Las equivalencias entre ER normales y las que se usan en Lex se muestran con ejemplos en esta tabla:

Caracteres	Ejemplo	Significado
Concatenación	xy	El patrón consiste en `x` seguido de `y`.
Unión	x|y	El patrón consiste en `x` o en `y`.
Repetición	x*	El patrón consiste en `x` repetido cero a más veces.
Clases de caracteres	[0-9]	Alternancia de caracteres en el rango indicado, en este caso `0|1|2|...|9`. Más de un rango se puede especificar, como por ejemplo: `[0-9A-Za-z]`para caracteres alfanuméricos.
Operador negación	[^0-9]	El primer carácter en una clase de caracteres deberá ser ^ para indicar el complemento del conjunto de caracteres especificado. Así, `[^0-9]` especifica cualquier carácter que no sea un dígito.
Carácter arbitrario	.	Con un único carácter, excepto `\n`.
Repetición única	x?	Cero o una ocurrencia de `x`.
Repetición no nula	x+	Una o más ocurrencias de `x`.
Repetición especificada	`x{n,m}`	x repetido entre n y m veces.
Comienzo de línea	^x	Unifica `x` sólo al comienzo de una línea
Fin de línea	x$	Unifica `x` sólo al final de una línea
Sensibilidad al contexto (operador "look ahead")	ab/cd	Unifica `ab`, pero sólo seguido de `bc`.
Cadenas de literales	"x"	Cuando `x` tenga un significado especial.
Caracteres literales	\x	Cuando `x` es un operador que se representa a él mismo. También para el caso de `\n`, `\t`, etc.
Definiciones	{nombrevar}	Pueden definirse subpatrones. Esto significa incluir el patrón predefinido llamado `nombrevar`.

Definiciones en Lex. La sección de definiciones permite predefinir cadenas que serán útiles en la sección de las reglas. Por ejemplo:

```
comentario      "//".*limitador [ \t\n]espblanco        {limitador}+letramay    [A-Z]letramin   [a-z]letra              {letramay
```

Cada regla está compuesta de un *nombre* que se define en la parte izquierda, y su definición se coloca en la derecha. Así, podemos definir `comentario` como `//` (con la barra puesto que es un carácter especial), seguido por un número arbitrario de caracteres excepto el de fin de línea. Un `limitador` será un espacio, tabulador o fin de línea, y un `espblanco` será uno o más limitadores. Nótese que la definición de `espblanco` usa la definición anterior de `limitador`.

Reglas para eliminar ambigüedades. Se producen cuando varias ER's son aplicables a un mismo lexema reconocido. Lex seleccionará siempre la ocurrencia más larga posible. Si dos ocurrencias tienen la misma longitud, tomará la primera.

Notas complementarias sobre Lex.

- Cada vez que se realice una de las acciones, la variable `char *yytext` contendrá el lexema reconocido.
- La variable `int yyleng` contiene la longitud del lexema reconocido.
- *Entrada y salida*: `FILE *yyin, *yyout`. Por defecto, se usan los predefinidos en C.
- Cuando Lex reconoce el carácter de fin de fichero, llama a la función `int yywrap()`, que por defecto devuelve `1`. Si devuelve `0`, significará que está disponible una entrada anterior, con lo cuál aún no se habrá terminado la lectura.
- *Contextos*: Permiten especificar cúando se usarán ciertas reglas. Veremos mediante un ejemplo de eliminación de comentarios cómo se usan los contextos:

```
%start COMENTARIO%%\/\*           {BEGIN COMENTARIO;} /* activa COMENTARIO */<COMENTARIO>\*\/  {BEGIN 0;}              /* desact
```