|  |  |  |
| --- | --- | --- |
| **2ª Fase**  **Analizador Léxico con FLEX** | 2010 | |
|  | |  |

**Autores**

Pablo Eduardo Ojeda Vasco

Roberto Marco Sánchez

# A Índice

# Expresiones regulares del lenguaje …………………………………….. *Página 3*

1. Analizador Léxico ………………………………………………………………. *Página 5*
   * *Definición*
   * *Conceptos básicos de un Analizador Léxico*
   * *Tareas de un Analizador Léxico*

# *Prioridad de los tokens*

* + *Formas de construir un Analizador Léxico*
  + *Motivo de la independencia de un Analizador Léxico*

1. Lex / Flex …………………………………………………………………………… *Página 7*
   * + *Definición*
     + *Formato de un fichero Flex*
     + *Generación*
     + *Contenido de lex.yy.c*
     + *Tratamiento de errores en Flex*

# 1 Expresiones regulares del lenguaje

Tipos de datos

* **letra** [a-za-z\_]
* digito0 [0-9]
* digito [1-9]
* **letdig** {letra}|{digito0}
* hexa [a-fa-f0-9]
* oct [0-7]
* **decimal** {digito}{digito0}\*
* **octal** 0{oct}\*
* **hexadecimal** [0x|0x]{hexa}{hexa}\*
* integer\_constant {decimal}|{octal}|{hexadecimal}
* decimal\_floating1 {({digito0}{digito0}\*)?"."{digito0}{digito0}\*([Ee]([+-])?{digito0}{digito0}\*)?
* decimal\_floating2

{digito0}{digito0}\*"."({digito0}{digito0}\*)?([Ee]([+-])?{digito0}{digito0}\*)?

* decimal\_floating3 {digito0}{digito0}\*([Ee]([+-])?{digito0}{digito0}\*)?
* **Número:** {integer\_constant}
* **Número:** {decimal\_floating1}|{decimal\_floating2}|{decimal\_floating3}
* **Indentificador:** {letra}{letdig}\*

Ristras

* **Carácter:** '{letra}'
* **String:** \"{letra}{letdig}\*\"
* **String\_C:** \"(.)\*\"

Comentarios

* comentario1 "/\*"{comentario}\*"\*/"
* comentario2 "//"{comentario\_2}\*
* **Comentario:** {comentario1}|{comentario2}

Palabras reservadas

* break | case | char | const | default | do | double | else | float | float | for | goto | if | int | return | struct | switch | typedef | union | void | while | printf | scanf

Caracteres de escape

* simple\_escape\_sequence \' | \" | \? | \\ | \a | \b | \f | \n | \r | \t | \v
* octal\_escape\_sequence \{oct}({oct}{oct}?)?
* hexadecimal\_escape\_sequence \x{hexa}{hexa}?
* **Carácter de escape:** {simple\_escape\_sequence} | {octal\_escape\_sequence} | {hexadecimal\_escape\_sequence}|.

Separadores

* [ | ] | ( | ) | { | } | . | : | ; | ,

Operadores

* **Operadores aritméticos:** + | - | \* | / | %
* **Operadores relacionales:** < | > | <= | >= | == | !=
* **Operadores lógicos:** && | “||” | !
* **Operadores a nivel de bits:** & | “|” | ^ | ~ | << | >>
* **Operadores asignación:** = | ++ | -- | \*= | /= | %= | += | -=

# 2 Analizador Léxico

**2.1.- Definición**

Un **analizador léxico** o **analizador lexicográfico** (en inglés *scanner*) es un programa que recibe como **entrada el código fuente de otro programa (secuencia de caracteres)** y produce una **salida compuesta de *tokens* (componentes léxicos)** o símbolos, a los que asocia, si es necesario, **unos atributos**. Estos *tokens* sirven para una posterior etapa del proceso de traducción, **siendo la entrada para el analizador sintáctico** (en inglés *parser*). Un analizador léxico es, por tanto, **un programa que reconoce patrones en un texto**.

**2.2.- Conceptos básicos en un analizador léxico**

1. **Token** (Componente léxico): Secuencia de caracteres con significado sintáctico propio. Los tokens suelen ser palabras reservadas, operadores, identificadores, constantes, signos de puntuación (paréntesis, punto y coma, etc.), marcas de comienzo y fin de bloque.
2. **Lexema**: secuencia de caracteres cuya estructura se corresponde con el patrón de un token.
3. **Patrón**: regla que describe los lexemas correspondientes a un token.

Ejemplo:

|  |  |  |
| --- | --- | --- |
| **Token** | **Lexema** | **Patrón** |
| Identificador | x, y, valor, etc | <letra>(<letra>|<digito>)\* |

**2.3.- Tareas de un analizador léxico**

* Reconocer los componentes léxicos de la entrada.
* Eliminar los espacios en blanco, caracteres de tabulación, saltos de línea y de página, etc.
* Eliminar comentarios de entrada.
* Detectar errores léxicos.
* En lenguajes de programación debe:
  + Reconocer identificadores de variable, tipo, constante y guardarlo en una tabla de símbolos.
  + Relacionar los mensajes de error del compilador con el lugar en el que aparecen en el programa fuente.

**2.4.- Prioridad de los tokens**

Se da prioridad al token con el lexema más largo. Ejemplo:

“>=” ó “>” 🡪 reconocería el primero “>=”

Si el mismo lexema se puede asociar a dos tokens, estos patrones estarán definidos en un orden determinado, por tanto, elegirá el que esté más arriba en código.

**2.5.- Formas de construir un Analizador Léxico**

**Usando generadores de analizadores léxicos**. Es la forma más sencilla, pero el código generado por el analizador léxico es más difícil de mantener y puede resultar menos eficiente. Recibe la especificación de las expresiones regulares de los patrones que representan a los tokens del lenguaje y las acciones a tomar cuando los detecte.

Ventaja: Comodidad de desarrollo. Permite concentrarnos en el diseño y

acciones asociadas.

Desventaja: El mantenimiento del código generado resulta complicado.

La eficiencia del código depende del generador.

**Escribiendo el analizador léxico en un lenguaje de alto nivel**. Exige más esfuerzo pero es más eficiente y sencillo de mantener.

**2.6.- Motivos de la independencia del Analizador Léxico**

La independencia del Analizador Léxico simplifica el diseño del Analizador Sintáctico. Además se consigue un compilador más eficiente, puesto que un sistema de entrada optimizado aumenta la velocidad de lectura.

Por otro lado añade portabilidad al compilador gracias a la independencia del alfabeto.

# 3 Lex / Flex

**3.1.- Definición**

**Lex** es un programa para generar analizadores léxicos (en inglés *scanners* o *lexers*). Lex se utiliza comúnmente con el programa yacc que se utiliza para generar análisis sintáctico. Lex, escrito originalmente por Eric Schmidt y Mike Lesk, es el analizador léxico estándar en los sistemas Unix, y se incluye en el estándar de POSIX. Lex toma como entrada una especificación de analizador léxico y devuelve como salida el código fuente implementando el analizador léxico en C.

Aunque tradicionalmente se trata de software propietario, existen versiones libres de lex basadas en el código original de AT&T en sistemas como OpenSolaris y Plan 9 de loslaboratorios Bell. Otra versión popular de software libre de lex es Flex.

**Flex** es una herramienta para generar **escáneres**: programas que reconocen patrones léxicos en un texto. Flex lee los ficheros de entrada dados, o la entrada estándar si no se le ha indicado ningún nombre de fichero, con la descripción de un escáner a generar. La descripción se encuentra en forma de parejas de expresiones regulares y código C, denominadas **reglas**. Flex genera como salida un fichero fuente en C, `lex.yy.c', que define una rutina `yylex()'. Este fichero se compila y se enlaza con la librería `-lfl' para producir un ejecutable. Cuando se arranca el fichero ejecutable, este analiza su entrada en busca de casos de las expresiones regulares. Siempre que encuentra uno, ejecuta el código C correspondiente.

**3.2.- Formato de un fichero Flex**

A grosso modo, la estructura es la siguiente:

*definiciones*

*%%*

*reglas*

*%%*

*código de usuario*

De forma más detallada:

Declaraciones en C entre %{ y }%

Declaraciones especiales comenzadas por %

Reglas de “no-terminales” auxiliares

%%

Secuencia de expresiones reguales (patrones) de símbolos a reconocer y entre

llaves { }, acciones en C asociadas, normalmente terminadas en return “código\_del\_símbolo”.

%%

Rutinas auxiliares y/o main si es un programa independiente.

**3.3.- Generación**

Editamos un fichero de especificación “lexico.l”.

Lo procesamos con $ *flex lexico.l* y obtenemos lex.yy.c

Normalmente lo compilaremos junto con un Analizador Sintáctico producido por bison. Sin embargo, en este caso lo vamos a utilizar cómo un Analizador Léxico independiente, por lo que introducimos un *main()* que llame a *yylex()*. Compilamos con:

$ *gcc –o lexico lex.yy.c –lfl*

**3.4.- Contenido de lex.yy.c**

* Declaraciones iniciales copiadas.
* Función ***yylex()*** con bucle de reconocimiento de patrones:
  + Cada vez que reconoce uno, almacena el texto en **yytext** y ejecuta el código asociado.
  + Si hay más de uno, da preferencia al texto de mayor longitud.
  + Si tienen igual longitud, da preferencia al patrón indicado en primer lugar.
* Rutinas auxiliares.

\* Si detectamos un error léxico 🡪 Si no encuentra patrón alguno (error), se deja atrás (y visualiza) el primer carácter y vuelve a intentarlo.

**3.5.- Tratamiento de errores en Flex**

Podemos y debemos:

* Detectar caracteres no permitidos utilizando “.”. Son caracteres ilegales, es decir, no recogidos en el alfabeto.
  + Ejemplo “ñ” , “é”…
* Detectar patrones no permitidos. Combinaciones ilegales de caracteres.
  + Ejemplo “:=” “:::”…
* Visualizar los correspondientes mensajes explicativos. Se debe mostrar un mensaje de error claro y exacto.
  + Ejemplo: “Error 124: línea 85, columna 6, carácter no válido”

Posibles acciones del analizador léxico para detectar errores, recuperarse y seguir trabajando:

* Modo de pánico: ignora los caracteres no válidos hasta un carácter en el cuál se considera que podría empezar un lexema correcto. **Éste es el que utiliza Flex.**
* Distancia mínima de corrección de un error. Recuperación inteligente con correcciones cómo: borrar los caracteres extraños, insertar un carácter incorrecto por uno correcto, insertar un carácter que pudiera faltar, conmutar las posiciones de dos caracteres adyacentes.