TEMA 2: ANALIZADOR LÉXICO

Índice

Definición del analizador léxico

- Desde el pto. de vista funcional:
  <token, atributos> = analizador_lexico (fichero_fuente)
- Lexema
  - secuencia de caracteres
- Token:
  - conjunto asociado de lexemas (forman un lenguaje). Este lenguaje es descrito por gramáticas regulares o por expresiones regulares.
- Especificar un token:
  - Expresión regular que describe sus lexemas
  - Atributos
  - Algoritmos que calculan los atributos

Definición del analizador léxico

- Especificar un analizador léxico:
  - Especificar todos los tokens del lenguaje
    ({t1,t2,...tn}).
  - Puede haber varios tokens que correspondan a una misma expresión regular
    (ej / {e1,e2...ej,..en} ,, ej ➔{t1,t2,..,tj,..,tn}).
  - En este caso el algoritmo asociado a la exp. regular deberá diferenciar a partir del lexema el token.
    (aj➔ tj).

Definición del analizador léxico

- Funcionamiento:
  - hay una variable (texto) que contiene el lexema encontrado al reconocer un determinado patrón.
  - Cada vez que se llama a la función analizador_lexico, éste va leyendo caracteres hasta encontrar un lexema que pertenece a un lenguaje descrito por una expresión regular (sea ej).
  - En ese momento en texto está el lexema completo y se ejecuta el algoritmo asociado a ej (sea aj).

<token, atributos> = aj(texto)

- El objetivo del algoritmo aj es determinar el token y sus atributos.

Generación automática de analizadores léxicos (Lex)

- •Es un generador de analizadores léxicos a partir de expresiones regulares.
- Se define el comportamiento del analizador mediante una especificación basada en E.R.
- Se genera un fichero en un código fuente de algún lenguaje de programación con un interfaz bien definido (función yylexx())
- • Esquema de la especificación:

(zona de declaraciones) [código C o macros]
%%
(zona de reglas y acciones)
%%
(zona de rutinas de usuario)

Generación automática de analizadores léxicos (Lex)

- •Es un generador de analizadores léxicos a partir de expresiones regulares.
- • Esquema de zona de declaraciones:

```
%{
----código C---- /*declaración variables, funciones, etc.*/
%}
macro1
macro2
```


Generación automática de analizadores léxicos (Lex)

- Ej. Zona de declaraciones:

```
%{
#include <stdio.h>
int y;
%}
digito (0|1|2|3|4|5|6|7|8|9)
letraAB (a|b)
```

Generación automática de analizadores léxicos (Lex)

- Ej. De una especificación:

```
%{
#include <stdio.h>
int cuenta=0;
%}
digito (0|1|2|3|4|5|6|7|8|9)
%%
{digito}        {++cuenta}
%%
```


Generación automática de analizadores léxicos (Lex)

- (zona de reglas y acciones):
  - reglas: 1ª columna
  - acciones: a partir de la 7ª columna. No pasar de la 75ª columna:
    - regla rj { acc1;
      acc2;
      accn;}
    - O bien usar \ retorno de carro.

EXPRESIONES REGULARES (E.R.)

- Operadores:

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

- Operador como literal o carácter ordinario:

  entre comillas dobles o precedido de \ ej.: "[" o \[

- No hay una E.R. específica para λ ni para ϕ
- Comillas dobles: encierran cadenas de caracteres
- Secuencias de escape:

  \n reconoce un retorno de carro
  \t reconoce tabulación horizontal
  \v reconoce tabulación vertical
  \b reconoce un retroceso de un carácter hacia atrás
  .....

EXPRESIONES REGULARES (E.R.)

- Clases de caracteres
  - Positiva: [a1a2....an], [a1-an] (todos los caracteres máquina existentes entre a1 hasta an)

  = [a1a2....an] ➔ $L(\alpha) = \{a \in \sum M \mid a = a1,a2,...,an\}$

  = [a1-an] ➔ $L(\alpha) = \{a \in \sum M \mid a1 <= a <= a2\}$

  - Negativa: [^a1a2..an], cualquier carácter que no es a1,a2,...,an
  - Dentro de una clase pierde el sentido de operador (por lo tanto no necesito precederlo de "" o \ para denotar carácter):

    ? . * + | ( ) $ / { } % < >

  ej.: [0-9], reconoce un dígito entre 0 y 9.

EXPRESIONES REGULARES (E.R.)

- Operador cualquier carácter: **.**

  . ➔ L(α) = {a ∈∑M | a ≠ \n}

- Unión: **|** (prefijo de mayor longitud)

β|φ ➔ L(α) = {ρ ∈Σ_M^* | (ρ ∈ L(α) o ρ ∈ L(β)) y |ρ| es máxima}

Ej. α= ab|abc

abcdef, reconoce abc.

para reconocer todo: (.|\n)

- Clausura: *
- Clausura positiva: +

Ej: [0-9][a-z0-9]*

EXPRESIONES REGULARES (E.R.)

- Repetición acotada: expresión_regular{n1,n2} (muy costoso)

  Ej.: α = (a{1,3}) ➔ L(α) = {a, aa, aaa}

- Opción: ? (una o ninguna vez)

  β? ➔ L(α) = {λ} ∪ L(β))

  ej: [0-9]+(.[0-9]+)? = un nº con parte decimal o no

- Macros y paréntesis:
  - La definición de macros debe ir encerrada entre ( ) ya que la sustitución es literal y concatenada.
  - Def. de macro: nombre_macro exprRegular
  - Referencia a una macro: {nombre_macro}
  - Ej: letra a|b|c....|z

  Ident {letra}+ esto es igual a a|b|c....|z+ = {a,b,c..,z,zz,zzz,...}

Dependencias del contexto

- Dependencias simples: tiene en cuenta el contexto solo dentro de una línea
  - ^α reconoce si α está al principio de la línea
  - $\alpha$$ .....al final de la línea (no incluye retorno de carro)
  - ^$\alpha$$ .....en la línea solo hay α (no incluye retorno de carro)
  - ^$ .....línea no tiene nada (sí incluye retorno de carro)
- α/β: contiene α pero solo si va seguida de β (muy costoso)

  ej. α=ab/cd entrada= abcd y reconoce ab y deja cd en la entrada.

Dependencias del contexto

- Ejemplo:

Un fuente en lex que elimine todas las líneas completamente en blanco, todos los blancos al principio de una línea, líneas con solo blancos, todos los blancos al final de la línea y entre dos palabras consecutivas solo haya un espacio en blanco

```
^$              return;
^" "+\n         return;
^" "+           return;
" "+$           return;
" "+            putchar(' ');
.|\n echo;
```

Dependencias del contexto

- Dependencias complejas: reconoce cualquier expresión dependiendo del contexto en que se encuetra.
  - Definir contexto: <ci,cj...cn> α acci reconoce α y se ejecuta solo si estoy en el contexto ci,cj,...cn.
  - Declarar contexto: %s c1 c2 ....cn Entrar en un contexto: BEGIN(ci). Según versión, para ejecutarse debe ir solo en la acción o bien encerrado entre llaves con el resto de acciones.
- Por defecto se está en el contexto INITIAL, también denominado 0 (referirse a 0 por compatibilidad)
- Las reglas de INITIAL se reconocen en cualquier contexto. ➔ el orden de las reglas es importante.

Dependencias del contexto

- OBSERVACIÓN: en PCLEX
  - los contexto son de evaluación exclusivo, por lo tanto no se tienen en cuenta las reglas INITIAL.
  - la declaración de contextos, en lugar de %s es %x)
  - los operadores ^y $ no se pueden utilizar en la definición de macros (se interpretan como caracteres normales)

Ej: eliminar los comentarios C de un fichero

```
%x COM COD
%%
BEGIN(COD)
<COD>"/*" BEGIN(COM);
<COD>.|\n ECHO;
<COM>"*/"BEGIN(COD);
<COM>.|\n ;
```

Generación de un fichero y variables predefinidas

- Generación del analizador:
  - Fichero fuente de texto con extensión .l
  - Ejecutar lex.
  - Genera un fichero con extensión .c
  - Compilar con una rutina que llame a yylex()
- El fichero con extención .c:

Prototipos (todas las funciones de lex)
Código global (zona de declaraciones)
int yylex(void){
    Código local (código inicial en sección reglas no asociado a E.R.)
    Simulación autómata
}
rutinas usuarios (zona de rutinas de usuario)
tablas de transición del autómata AFD

Generación de un fichero y variables predefinidas

- Variables predefinidas:
  - Char *yytext: cadena de caracteres (prefijo) validada por la última regla
  - Int yyleng: la longitud del último prefijo
  - FILE *yyin: puntero al fichero de entrada (teclado)
  - FILE *yyout: puntero la fichero de salida (pantalla)
- Acciones predefinidas:
  - ECHO: muestra el último prefijo por pantalla
    #define ECHO fprintf(yyout, "%s", yytext)
  - BEGIN (C): cambia de contexto
  - output(c): manda el carácter c al fichero yyout
  - input(c): lee un carácter del fichero yyin
  - unput(c): devuelve el carácter al fichero yyin
  - yyless(n): deja los n primeros caracteres en yytext y devuelve los restantes al fichero de entrada

Generación de un fichero y variables predefinidas

- Ej:"=-"[a-zA-Z]+ {
  fprintf (yyout, "devolvemos el identificador);
  yyless(2);
  }
- yymore(): mantiene el prefijo anterior
  Ej. ab12 ➔ yytext="ab"
  yytext="12";

  con yymore:
  yytext="ab"
  yymore();
  yytext="ab12";

Generación de un fichero y variables predefinidas

- Ej. Un reconocedor de cadenzas en C (caracteres entre comillas dobles y dentro puede aparecer comillas dobles precedida de \)

```
%{
#include <stdio.h>
%}
\"[^\"]*\"   { if(yytext[yyleng-2]=`\`){
                        yyless(yyleng-1);
                        yymore();
                 }else  ECHO;
               }
.|\n           ;
```

Generación de un fichero y variables predefinidas

- REJECT: desactiva una regla temporalmente hasta que se reconozca un prefijo. No es compatible con BEGIN. Desactiva las reglas que tiene a la derecha.

  α1 {a1; REJECT;}
  α2 a2;
  α3 a3;
  α1 no se valida hasta que se reconozca un prefijo

- int yywrap(void): programa acciones ante el fin de fichero. Lex verifica el final de fichero de la siguiente forma:

  Utilidad: definir acciones finales y cambiar de fichero fuente (preprocesador).

  ```
  if (c=input()) = = EOF && yywrap() )
  return 0;
  else {
  ……..continua.....
  }
  ```


Generación de un fichero y variables predefinidas

- Observaciones:
  - Los caracteres que se leen y no hacen coincidencia con ninguna E.R. van por defecto a yyout. Si no quiero esto, lo acepto y no hago ninguna acción (. {;}
  - La llamada a yylex termina cuando encuentra eof (and yywarp()) o cuando ejecuta un return
  - Da igual en E.R. texto entre comillas o no, (texto = "texto")

Tratamiento de las ambigüedades

Ej.: identificar palabras reservadas e identificadores

```
%{
#define PAL_RES 1
#define IDEN    2
%}
ident ([a-zA-Z]([a-zA-Z0-9])*
%%
if          |
Then        |
else        |       return PAL_RES;
{ident}             return IDEN;
%%
```

- En la entrada: then

  Si coinciden con dos E.R. se ejecuta el aj de la primera desde arriba hacia abajo (devuelve PAL_RES)
- Entrada: ifthenelse

  Si coinciden con varias E.R. se toma la que reconoce entrada de mayor longitud. (devuelve IDEN)

ATRIBUTOS DE LOS TOKENS

- Debe haber un compromiso entre el léxico y el sintáctico.

- <u>Principales tokens:</u>
- Identificadores
  1. devolver el lexema ➔ problemática
  2. devolver una entrada a la tabla de símbolos. ➔ métodos de manejo de la tabla.
- Números
  1. Devolver el lexema ➔ problemática
  2. Valor numérico asociado al lexema (hilera de dígitos) ➔ funciones de conversión
  3. Crear una tabla de constantes y devolver la entrada a la misma.

ATRIBUTOS DE LOS TOKENS

- Operadores
  1. No hay atributo
  2. Agruparse todos los operadores del mismo tipo (binario, ternario, etc.) en un solo token y el atributo distingue cual es.

- Hileras de caracteres
  1. no tiene atributo
  2. crear un buffer de hileras y se devuelve el puntero a su inicio.

DECISIONES EN EL DISEÑO

- Palabras reservadas
  - Las palabras reservadas pueden o no ser identificadores. Lo más sencillo es que las palabras reservadas no sean identificadores, en este caso ocurre que el mismo patrón suele definir a ambos. Soluciones:
  - expresión regular para cada lexema que define la palabra reservada
  - tabla de palabras reservadas con sus tokens. El lexema se busca y si no existe es un identificador.
- Mayúsculas y minúsculas
  - Una función que transforma de minúsculas a mayúsculas. Si se quiere que el lenguajes no distinga ➔ invocar siempre a dicha función. Si se quiere que se distinga no se hace nada.

DECISIONES EN EL DISEÑO

- Longitud de los identificadores
  - Un buffer que permita toda la longitud que se quiera. Si el lenguaje quiere limitar ➔ una función que trunca el lexema a un máximo y este es con el que se trabaja.
- Números
  - Si hay varios tipos (entero, real, doble, etc.), un token para cada uno de ellos. Determinar la notación de los mismos. Utilizar funciones para determinar el valor a partir del lexema.
- Comentarios
  - El problema es detectar los fines de líneas dentro de ellos para mantener información de la línea actual. Crear un entorno para gestionarlo.
- Hilera de caracteres
  - Definir muy bien cómo es una hilera y encontrar su expresión regular que la defina.

Tabla de Símbolos (TS)

- Es una estructura de datos que contiene un registro por cada identificador, con los campos para los atributos del identificador.
- La estructura de datos permite
  - encontrar rápidamente el registro de cada identificador
  - y almacenar o consultar rápidamente datos de ese registro.
- En la fase de análisis léxico se crean las distintas entradas en la tabla
- El resto de las fases de compilación introducen y completan atributos

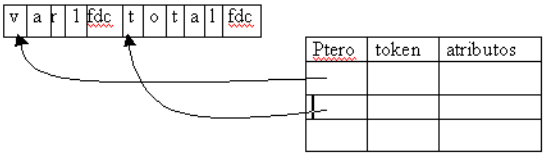
(el sintáctico y semántico: tipo de datos, generación de código: detalle sobre la memoria.)

Tabla de Símbolos (TS)

- El interfaz de la TS son dos funciones:
  - inserta (lexema, token): inserta el nuevo símbolo en la tabla y devuelve la entrada donde lo ha insertado y su token.
  - Busca(lexema): devuelve el índice donde se encuentra el lexema o 0 si no lo encuentra.
- Funcionamiento:
  - El analizador en primer lugar buscará el lexema encontrado en la tabla y si no lo encuentra lo inserta.
  - En caso de que lo encuentre no debe hacer nada en la tabla.

Tabla de Símbolos (TS)

- Implementación:
  - Un campo: un puntero apunta a la tabla de lexemas.
  - En la tabla de lexemas, para diferenciarlos: un carácter de fin de cadena (FDC) (se resuelve longitudes máximas)
  - El segundo campo contiene el token asociado al lexema.
  - Resto de campos: distintos atributos que dependen del tipo de token (dirección de memoria, tipo de datos, nº de parámetros línea del fuente de su declaración, línea de fuente de su referencia y un puntero para ordenar por el símbolo)



Implementación del analizador léxico

- Estructura de ficheros:
  - ansic.h: contiene la definición de los tokens y sus atributos

```
typedef union {
int i;
float r;
char *s;
}YYSTYPE;
extern YYSTYPE yylval;
#define C_CARACTER 258
#define C_FLOAT 259
#define C_ENTERA 260
#define STRING 261
#define ID 262
……
```


Implementación del analizador léxico

- lex.l: el fuente en lex (includes de los ficheros, expresiones regulares y sus acciones (llamada a funciones) retornando siempre el token)
- reservadas.c: tabla de palabras reservadas e implementación e procedimiento de búsqueda. Cada entrada de la tabla será un registro con el lexema y el número de token:

```
static const struct {
char *nombre;
int token;
} palabras_clave [] = {
{"auto", Auto},
{"break", Break},
….
}
int busca_palabra (void){ …. }
```

Implementación del analizador léxico

- acciones.c: la implentación de las funciones y procedimientos que constituyen las acciones de las expresiones regulares. Retornan el token y en yylval deja el valor del atributo.
- global.h: contiene la definición de todas las variables que son globales a todas las partes del compilador. Se utilizan para comunicar distintas partes del mismo (declaramos todas como extern):
  - yyin
  - yyfuente
  - token
  - yylineno
  - yycoluno
  - yycaracter
  - yytext
  - yylval

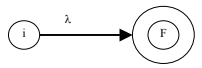
CONSTRUCCIÓN MANUAL DEL ANALIZADOR LÉXICO

- Tres formas de realización de un analizador léxico:
  - Herramientas como Lex
  - Escribir el analizador léxico en un lenguaje clásico ➔ arrastra sus limitaciones
  - Escribirlo en lenguaje ensamblador ➔ gestionar los buffers directamente

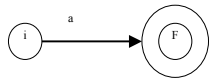
CONSTRUCCIÓN MANUAL DEL ANALIZADOR LÉXICO

•**Construcción de Thompson:**
Generar autómatas a partir de E.R.:

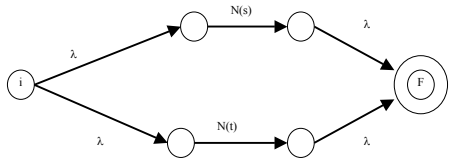
- λ



- a

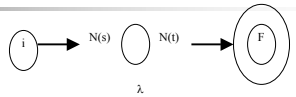


- s|t

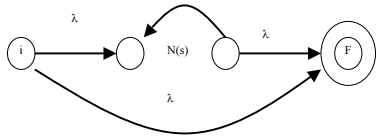


CONSTRUCCIÓN MANUAL DEL ANALIZADOR LÉXICO

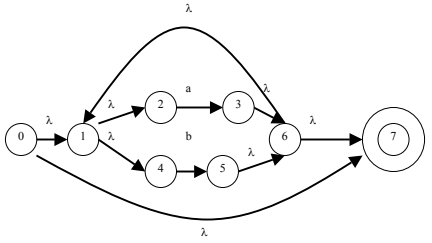
- st



- s*



•Ejemplo: (a|b)*



Implementación mediante tabla de transiciones

- Un simulador que gestiona el cambio de estado mediante una tabla de transiciones.
- La tabla de transiciones etiqueta las filas con estados y las columnas con los caracteres de entrada.
- En la intersección fila-columna se escribe el cambio de estado.

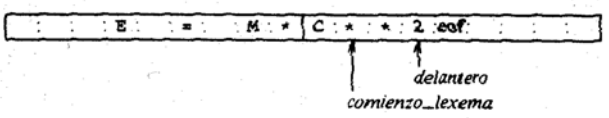
Implementación con reglas y acciones de un fuente lex

- Un grafo con estados y arcos:
  - Cada arco lo etiquetamos con un predicado (disjuntos) y una acción.
  - Solo puede haber un arco para un estado que parte con *else*
  - Hay un estado inicial (I) y varios finales (F) y entre medias los etiquetados con (M).
  - En los estados finales se reconoce el lexema y se ejecuta la acción asociada al patrón (Un predicado es el lexema (o lexemas) de la ER y su acción es el algoritmo).
- Hay que tener en cuenta que al llegar a un estado final, la variable yytext debe tener el lexema y que es posible tener que devolver al fichero de entrada posibles caracteres de adelanto.

Gestión de los buffers

- Si se prescinde de un lenguaje de programación que gestiona la entrada fichero ➔ debo gestionar el buffer (más rápido)
- Opciones:
  1. Si se lee carácter a carácter
     - ➔ va muy lento (el A.L. es lo más fácil y lento del compilador).
     - No tengo preanálisis: devolver carácter ➔ muy lento.
  2. Uso de buffers:
     - Parejas de buffers:

Gestión de los buffers



- Se divide en dos mitades de tamaño N (N= un bloque del disco (1024 o 4096)
- Dos punteros: comienzo_lexema y delantero.
- Al principio los dos igual y avanzo delantero hasta que encuentro un token
- Devuelvo token y coloco comienzo_lexema y delantero en el primer carácter del token
- Cando termina una mitad, recarga la otra y continua hasta encontrar token.

Gestión de los buffers

- Algoritmo para recargar los buffers

```
if delantero está al final de la primera mitad then
begin
recargar la segunda mitad;
delantero: = delantero + 1
end
else
if delantero está al final de la segunda mitad then
   begin
    recargar la primera mitad;
    pasar delantero al principio de la primera mitad
end
else
delantero := delantero + 1;
```


Gestión de los buffers

- Algoritmo para recargar los buffers
  - Problema 1: limito tamaño de preanálisis al tamaño del buffer.
  - Problema 2: el algoritmo en la mayoría de los moviento de punteros (cuando no hay que cargar ninguna mitad) comprueba dos veces
- Solución: Pareja de buffers con centinelas:
  - Un carácter centinela (suele ser eof): no existe en le fichero fuente
  - Se pone al final de las mitades ➔ en cada movimiento pregunto por él y solo compruebo final de la correspondiente mitad si estoy en un eof.
  - La mayoría de las veces solo hago una comprobación para mover el puntero.

Gestión de los buffers

- Algoritmo de recarga de Pareja de buffers con centinelas:

```
delantero: = delantero + 1:
if delantero = eof then  begin
   if delantero está al final de la primera mitad then begin
     recargar la segunda mitad;
     delantero: = delantero + 1
   end
   else if delantero está al final de la segunda mitad then begin
              recargar la primera mitad;
              pasar delantero al principio de la primera mitad
     end
     else /* eof dentro de un buffer significa el final de la entrada */
   terminar el análisis léxico
   end
```