

Tema 2: La fase de análisis léxico

2.1 Introducción

2.2 Conceptos básicos

2.3 Diseño e implementación de un
analizador léxico

2.4 Un lenguaje de especificación de
analizadores léxicos: LEX

2.1 Introducción

El análisis léxico:

- Filtra la parte de la entrada que no tiene “valor sintáctico”
 - » Comentarios
 - » Blancos:espacios, tab, \n
- Detecta (y trata) errores léxicos
- Identifica unidades léxicas “con significado”
- Calcula el valor de los atributos de los símbolos terminales (tokens)

2.1 Introducción

- Los tokens y sus atributos
- Su relación con el resto del compilador
- Problemas que conllevan algunas características del lenguaje fuente
- Errores léxicos

Tokens básicos y sus atributos(I)

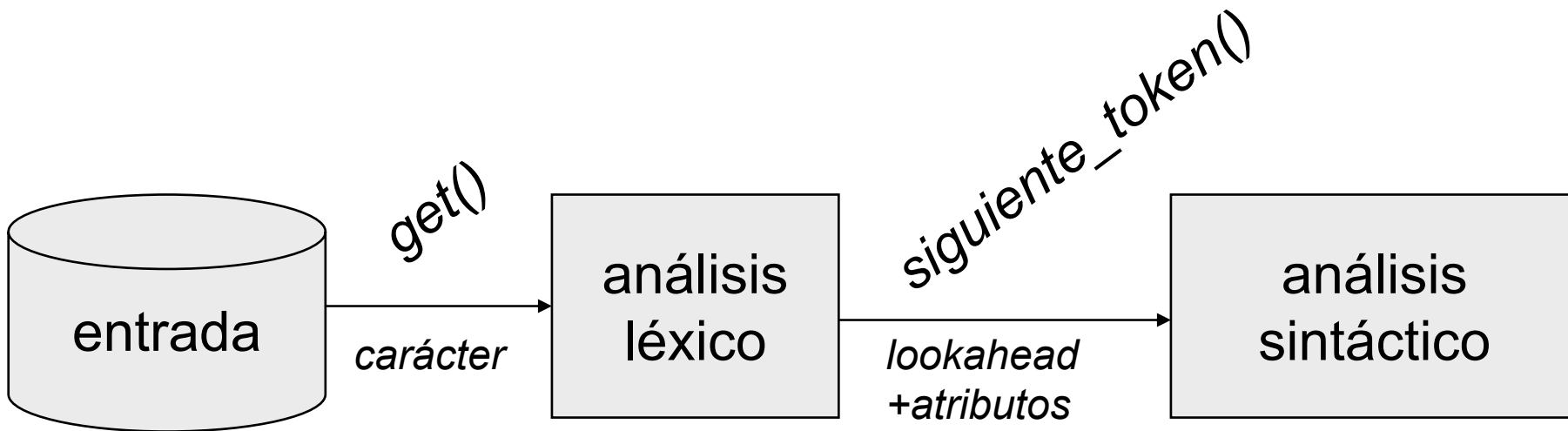
- Identificadores (nombre: string)
- Palabras clave/reservadas
(nombre: string; tipo: enumerado)
- Constantes
 - » Enteros (nombre: string; valor: entero)
 - » Reales (nombre: string; valor: real)
 - » strings, caracteres ...

Tokens básicos y sus atributos (II)

- separadores (nombre: string; tipo: enum)
 - » , :: () []
- Operadores (nombre: string; tipo: enum)
 - » + - / *
 - » := =
- Comentarios (atributo)

Interfaz con el analizador sintáctico

- lookahead
- siguiente_token()



Errores léxicos

- 3.a
- 3.5e-
- "string sin cerrar
- 3ident := 5
- Carácter erróneo

2.2 Conceptos básicos

- Especificación:
 - Expresiones regulares y definiciones regulares
- Mecanismos reconocedores: Autómatas finitos
- Equivalencia y conversión entre expresiones regulares y autómatas finitos

Especificación

- Alfabeto Σ -> conjunto de símbolos
- Palabra -> secuencia de símbolos de un alfabeto Σ
- Σ^* -> conjunto de palabras sobre Σ
- Operaciones sobre palabras
- Lenguaje -> subconjunto de Σ^*
- Operaciones sobre lenguajes -> unión, intersección, concatenación, potencia, clausura, ...

Lenguajes regulares

Sea Σ un alfabeto:

- I. $\{\epsilon\}$ es un lenguaje regular
- II. $\{a\}$ es un lenguaje regular para cada a perteneciente a Σ

Sean A y B dos lenguajes regulares sobre Σ :

- III. $A \cup B$ es un lenguaje regular
- IV. $A \cdot B$ es un lenguaje regular
- V. A^* es un lenguaje regular
- VI. Nada más es un lenguaje regular sobre Σ

Expresiones regulares

Sea Σ un alfabeto:

I. ε es una ER. $L(\varepsilon) = \{\varepsilon\}$

II. a es una ER para cada $a \in \Sigma$. $L(a) = \{a\}$

Sean $e1$ y $e2$ dos ER sobre Σ :

III. $(e1) \mid (e2)$ es una ER. $L((e1) \mid (e2)) = L(e1) \cup L(e2)$

IV. $(e1) \cdot (e2)$ es una ER. $L((e1) \cdot (e2)) = L(e1) \cdot L(e2)$

V. $(e1)^*$ es una ER. $L((e1)^*) = L(e1)^*$

VI. Nada más es una ER sobre Σ .

Prioridad (de menor a mayor): $\mid \cdot ^*$

Definiciones regulares

- Notación para facilitar la escritura de E.R.

$$\text{def}_1 \rightarrow \text{er}_1$$
$$\text{def}_2 \rightarrow \text{er}_2$$
$$\dots$$

Donde def_i son identificadores y er_i expresiones regulares que pueden contener def_i

Ejemplo de especificación (1er intento)

Tipo token	Descripción	Atributos	Ejem.
Constante entera	(0 1 2 3 4 5 6 7 8 9)+	Nombre: string Valor: entero	00 1245
...

Autómatas finitos

› Un autómata A define un lenguaje L :

- acepta las cadenas de L
- rechaza el resto

› Un programa reconocedor consta de:

- a) Cinta de entrada -> secuencia de símbolos de un alfabeto
- b) Cabeza de lectura -> apunta a un elemento de la entrada
- c) Control finito -> dirigido por una función de transición que define el lenguaje a reconocer

Autómatas finitos

- Proceso para obtener un programa reconocedor:
 - » a) definir una exp. regular para el lenguaje
 - » b) obtener un diagrama de transición (A.F.D.)
 - » c) escribir un programa reconocedor
- Autómatas finitos:
 - » deterministas (AFD)
 - » no deterministas (AFND)

Autómatas finitos

- AFND: $A = (S, \Sigma, \delta, S_0, F)$ donde
 - » S : conjunto de estados
 - » Σ : alfabeto
 - » $\delta : S \times \Sigma \cup \{\xi\} \rightarrow P(S)$ función de transición
 - » S_0 : estado inicial
 - » F : conjunto de estados finales

Autómatas finitos

Representación de un AFND: grafo de transiciones

Configuración: $(q, w\#)$

Movimiento:

- $(q, aw\#) \vdash (q1, w\#)$ si $q1$ pertenece a $\delta(q, a)$
- $(q, w\#) \vdash (q1, w\#)$ si $q1$ pertenece a $\delta(q, \varepsilon)$

Palabra aceptada: $(S_0, w\#) \vdash^* (qf, \#)$ qf pertenece a F

A.F.D.: cada símbolo tiene para cada estado una transición como máximo

Equivalencia y conversión

- ¿Podemos decidir si $x \in L(\alpha)$ donde α es una ER?
- Podríamos construir a mano un autómata que decida si $x \in L(\alpha)$.
- Por suerte hay algoritmos de conversión que garantizan equivalencia
 - » $ER \Rightarrow AFND \Rightarrow AFD$

2.3 Diseño e implementación de un analizador léxico

- Construcción de un analizador léxico a partir de un autómatata finito determinista: `siguiente_token()`

Otras cuestiones de implementación:

- Un único AFD (¿?)
- Buffers
- Palabras reservadas
- Tabla de transiciones eficiente
- Acciones asociadas a los estados finales

Algoritmo de un AFD

- *Para un autómata A cualquiera*
s:=estado_inicial(A);
obtener sobre c el primer carácter;
mientras existe (transicion(A,s,c)) hacer
 s:=transicion(A,s,c);
 c:=siguiente caracter;
fin mientras;
si final(A,s) entonces devolver "si"
si no devolver "no"

Siguiente_token

{c contiene el carácter en curso del fichero de entrada}

repetir

 obtener_token (t);

hasta que

 t sea un token interesante para el
 analizador sintáctico

fin repetir

Obtener_token

Obtiene el siguiente token a partir del carácter en curso

contiene el carácter en curso del fichero de entrada}

=estado_inicial(A);

mientras existe (transicion(A, s, c)) hacer

s:=transicion(A,s,c);

c:=siguiente carácter;

-- siguiente carácter no "casca" si se alcanza el fin de fichero

mientras;

final(A, s) entonces devolver token del estado s

no devolver "token erróneo"

Tipo Autómata

- **Operaciones de creación del Autómata**
 - » Crear
 - » Añadir_estado
 - » Añadir_símbolo
 - » Añadir_transición
 - » define_estado_inicial
 - » añadir_estado_final

Tipo Autómata

Otras operaciones

- » $\text{existe_transición: Aut\acute{o}mata} \times \text{estado} \times \text{s\acute{ı}mbolo} \rightarrow \text{boolean}$
- » $\text{transici\acute{o}n: Aut\acute{o}mata} \times \text{estado} \times \text{s\acute{ı}mbolo} \rightarrow \text{estado}$
- » $\text{estado_inicial: Aut\acute{o}mata} \rightarrow \text{estado}$
- » $\text{final: Aut\acute{o}mata} \times \text{estado} \rightarrow \text{boolean}$

Una posible representaci3n para el tipo Aut3mata

- » Registro con 4 componentes (estado inicial, estados, estados finales, transiciones).
- » Las transiciones se representan por medio de una tabla de $n \times m$ dimensiones (estados, s3mbolos).

Otras cuestiones (I)

- Uno por AFD?
- Juntar todos en uno sólo?
- Leer carácter a carácter puede ser ineficiente:
 - » Línea a línea: cuidado con fin de línea.
¿Comentarios multilínea?
 - » Usar un buffer

Otras cuestiones (II)

- Palabras reservadas:
 - » Un automata por cada palabra reservada
 - » Hacer una excepción: función `palres(str)`
- Tabla de transiciones:
 - » Hay 256 caracteres ASCII \Rightarrow 256 columnas
 - » Muchas columnas son idénticas:
columna etiquetada por un conjunto de caracteres
(nuevo tipo de datos)

Otras cuestiones (III)

- Acciones asociadas a los estados:
 - » Si el autómata acaba en un estado no final:
ERROR
 - » Si acaba en un estado final:
 - Ejecutar acciones asociadas
 - Devolver tipo de token correspondiente
 - » Posible implementación: case s

Otras cuestiones (IV)

Acciones: especificación del an. lex

Tipo token	Descripción	Atributos	Acciones
ident	$l(l d)^*$	nombre: ref_T_S	si palres(str) entonces tipo:=PALRES; nombre:=buscar_PR(str); sino tipo:=IDENT; str:=normalizar(str); nombre:=añadir(T_S,str);
pal. res.	program var ...	nombre: enumerado	
cte_ent	d+	valor: entero	tipo:=CTEENT; valor:=str2int(str);
cte_real	d+.d+	valor: real	tipo:=CTERREAL; valor:=str2real(str);

2.4 Un lenguaje de especificación de analizadores léxicos: LEX

Especificación LEX



LEX



lex.yy.c

yylex()



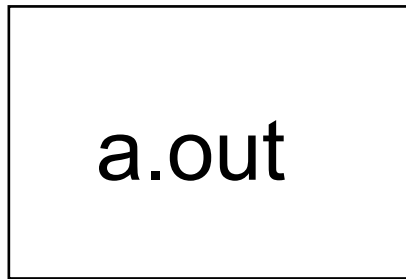
Compilador C
cc lex.yy.c -ll



a.out

2.4 Un lenguaje de especificación de analizadores léxicos: LEX

cadena
de
entrada



Secuencia
de
tokens

Especificaciones LEX (I)

DECLARACIONES

%

REGLAS

%

SUBROUTINAS DE USUARIO

Especificaciones LEX (II)

ECLARACIONES

Variables/constantes

Definiciones regulares

EGLAS

pi {acción i}

Especificaciones LEX (III)

UBPROGRAMAS DE USUARIO

Procedimientos auxiliares

Redefinición de procedimientos con los que
trabaja LEX

Expresiones regulares en LEX(I)

Caracteres de texto / Caracteres distinguidos

ESCAPE " \

PUNTO. Cualquier carácter menos el fin de línea

CLASES DE CARACTERES

- » [] [AB] conjunto con los símbolos A y B
- » - Rango [A-Z] conjunto con los símbolos de A a Z
 - El rango sólo se puede utilizar en un conjunto
- » ^ Complementario [^abc] conjunto con cualquier símbolo excepto a, b o c
 - Es el complementario de un conjunto, por tanto dentro de un conjunto
- » \ Escape \n salto de línea \t tabulación
 - \[o "[" corchete \a o "a" el símbolo a

Expresiones regulares en LEX(II)

EXPRESIÓN OPCIONAL

? ab?c -> abc ac

EXPRESIONES REPETIDAS * +

ALTERNATIVAS | a|b "a" o bien "b"

CONTEXTO:

» ^ \$

^ representa el comienzo de línea y

\$ representa el final de línea:

^[A-Z].*[A-Z]\$ Línea que comienza y acaba con
mayúscula

No existe la palabra vacía en LEX

Acciones LEX

Por defecto → Copiar

» Cuando no existe emparejamiento

ACCION NULA ;

ECHO

yymore()

...

yywrap()

Estructuras que proporciona LEX al usuario

yytext

» array de caracteres externo

yylen

» contador del número de caracteres emparejados

Tratamiento de la ambigüedad

Emparejamiento más largo



Primera regla en el orden de entrada LEX

Ejemplo:

if	{accion 1}
[a-z] +	{accion 2}

LEX

Convierte las reglas en código

Copia cualquier línea que comience por blanco en `lex.yy.c`

- » Declaración de variables externas a LEX
- » Declaración de variables a nivel de la función LEX
- » Copia en `lex.yy.c` el fragmento de entrada LEX comprendido entre `%{` y `%}`

Ejemplo

%

```
bv]
v" ); }
```

```
{printf("he leído una b o una
```

```
^b]
```

```
{nob( ); }
```

```
;
```

%

```
ob( )
```

```
{printf("<> de b" ); }
```