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

- Identifica unidades léxicas "con significado" (**tokens**)

  Calcula el valor de los atributos de los tokens
- Filtra la parte de la entrada que no tiene "valor sintáctico"

  Comentarios

  Blancos: espacios, tab, \n
- Detecta y trata los errores léxicos

2.1 Introducción

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

Tokens básicos y sus atributos (I) EJEMPLOS

- 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) EJEMPLOS

- separadores (nombre: string; tipo: enum)

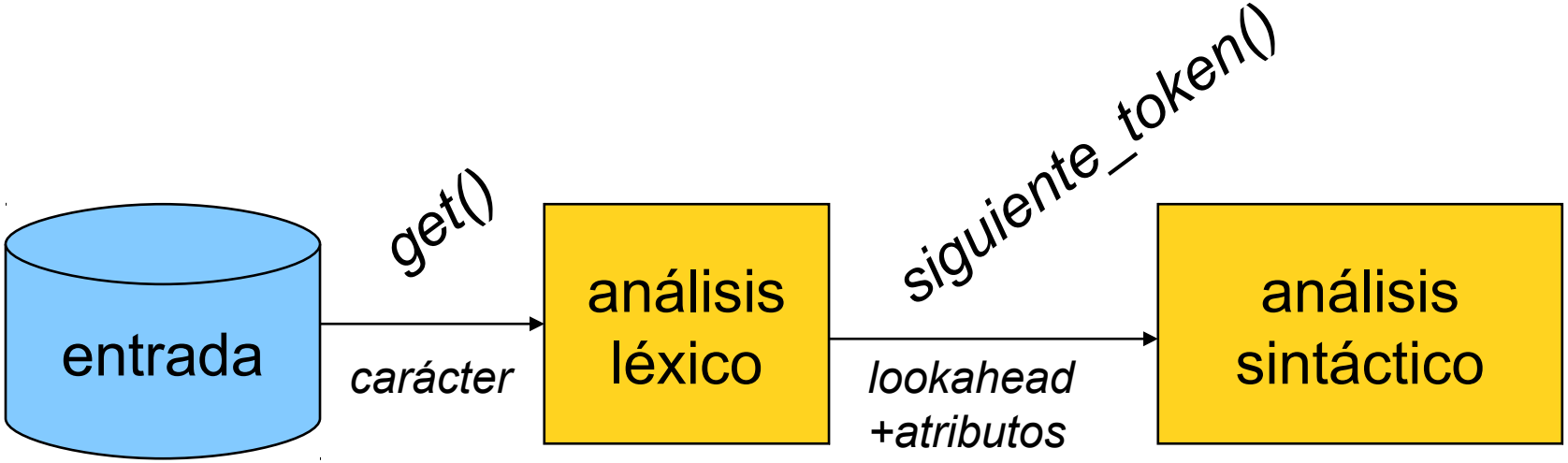
  , ; : ( ) [ ]
- Operadores (nombre: string; tipo: enum)

  + - / *

  := =
- Comentarios

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

2.2 Conceptos básicos

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

Lenguajes regulares

Sea Σ un alfabeto:

I. $\{\varepsilon\}$ es un lenguaje regular

II. $\{\mathbf{a}\}$ es un lenguaje regular para cada $\mathbf{a} \in \Sigma$

Sean **A** y **B** dos lenguajes regulares sobre Σ:

III. $\mathbf{A} \cup \mathbf{B}$ es un lenguaje regular

IV. $\mathbf{A} \cdot \mathbf{B}$ es un lenguaje regular

V. $\mathbf{A}^*$ 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. $\mathbf{a}$ es una ER para cada $\mathbf{a} \in \Sigma$ $L(a) = \{a\}$

Sean $\mathbf{e}_1$ y $\mathbf{e}_2$ dos ERs sobre Σ:

III. $(\mathbf{e}_1) \mid (\mathbf{e}_2)$ es una ER $L((\mathbf{e}_1) \mid (\mathbf{e}_2)) = L(\mathbf{e}_1) \cup L(\mathbf{e}_2)$

IV. $(\mathbf{e}_1) \cdot (\mathbf{e}_2)$ es una ER $L((\mathbf{e}_1) \cdot (\mathbf{e}_2)) = L(\mathbf{e}_1) \cdot L(\mathbf{e}_2)$

V. $(\mathbf{e}_1)^*$ es una ER $L((\mathbf{e}_1)^*) = L(\mathbf{e}_1)^*$

VI. Nada más es una ER sobre Σ

Definiciones regulares

- Notación para facilitar la escritura de ERs

$$def_1 \rightarrow er_1$$

$$def_2 \rightarrow er_2$$

...

Donde def_i son identificadores y er_i expresiones regulares que pueden contener más 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 palabras (cadenas) de L

  rechaza el resto

- Un programa reconocedor consta de:

  a) Cinta de entrada $\rightarrow$ secuencia de símbolos de un alfabeto

  b) Cabeza de lectura $\rightarrow$ apunta a un elemento de la entrada

  c) Control finito $\rightarrow$ dirigido por una función de transición que define el lenguaje a reconocer

Autómatas finitos

- Proceso para obtener un programa reconocedor:
  a) definir una expresión regular para el lenguaje
  b) obtener la función de transición
  c) escribir un programa reconocedor

- Autómatas finitos:
  deterministas (AFD)
  no deterministas (AFN)

Autómatas finitos

- AFD: $A = (Q, \Sigma, \delta, q_0, F)$ donde

  Q: conjunto de estados

  Σ: alfabeto

  δ: $Q \times \Sigma \rightarrow Q$ función de transición

  q_0: estado inicial

  F: conjunto de estados finales

Autómatas finitos

- Representación de un AFD: grafo de transiciones
- Configuración: (q, w#)

  donde *q* es un estado y *w#* es lo que queda por leer

- Movimiento:

  $(q, aw\#) \vdash (q_1, w\#)$ si $q_1 = \delta(q, a)$

- Palabra aceptada: $(q_0, w\#) \overset{*}{\vdash} (q_f, \#)$ y $q_f \in F$
- AFD: cada símbolo tiene para cada estado una transición (o no está definida)

Autómatas finitos

- ε-AFN: $A = (Q, \Sigma, \delta, q_0, F)$ donde

  Q: conjunto de estados

  Σ: alfabeto

  δ: $Q \times \Sigma \cup \{\varepsilon\} \rightarrow \wp(Q)$ función de transición

  q_0: estado inicial

  F: conjunto de estados finales

Autómatas finitos

- Representación de un ε-AFN: grafo de transiciones
- Configuración: $(q, w\#)$
- Movimiento:

$$(q, aw\#) \vdash (q_1, w\#) \text{ si } q_1 \in \delta(q, a)$$

$$(q, w\#) \vdash (q_1, w\#) \text{ si } q_1 \in \delta(q, \varepsilon)$$

- Palabra aceptada: $(q_0, w\#) \overset{*}{\vdash} (q_f, \#)$ y $q_f \in F$
- AFN: cada símbolo tiene para cada estado un conjunto de transiciones

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 $\Rightarrow$ ε-AFN $\Rightarrow$ AFD

2.3 Diseño e implementación de un analizador léxico

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

Otras cuestiones de implementación:

Un único AFD (¿?)

Buffers

Palabras clave/reservadas

Tabla de transiciones eficiente

Acciones asociadas a los estados finales

Algoritmo de un AFD

- *Para un autómata A cualquiera*

```
q := estado_inicial(A);
obtener sobre c el primer carácter;
mientras existe_transicion(A,q,c) hacer
      q := transicion(A,q,c);
      c := siguiente caracter;
fin mientras;
si final(A,q) y c == EOF entonces devolver "si"
                                 si no devolver "no"
fin si;
```

siguiente_token(out t)

{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(out t)

Obtiene el siguiente token a partir del carácter en curso

```
{c contiene el carácter en curso del fichero de entrada}
q := estado_inicial(A);
mientras existe_transicion(A, q, c) hacer
    q := transicion(A,q,c);
    c := siguiente carácter;
    -- siguiente carácter no "casca" si se alcanza fin de fichero
fin mientras;
si final(A, q) entonces devolver token del estado q
si no devolver "token erróneo"
fin si;
```

Tipo Autómata

- **Operaciones constructoras del Autómata**

  Crear

  Añadir_estado

  Añadir_símbolo

  Añadir_transición

  Definir_estado_inicial

  Añadir_estado_final

Tipo Autómata

- **Otras operaciones**

  existe_transición:

  `Autómata × estado × símbolo → boolean`

  transición: `Autómata × estado × símbolo → estado`

  estado_inicial: `Autómata → estado`

  final: `Autómata × estado → boolean`

- **Una posible representación para el tipo Autómata**

  Registro con 4 componentes (estado inicial, estados, estados finales, transiciones)

  Las transiciones se representan por medio de una tabla de dos dimensiones (estados, símbolos)

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:

  `palres: string` $\rightarrow$ `boolean`

- 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 q

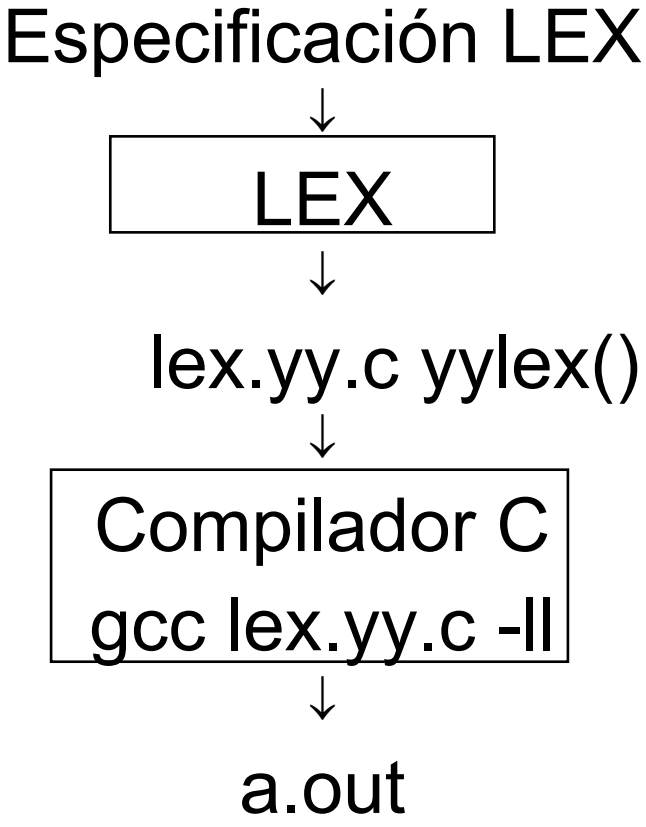
EJEMPLO
Especificación (tabla de símbolos)

Tipo token	Descripción	Atributos	Acciones

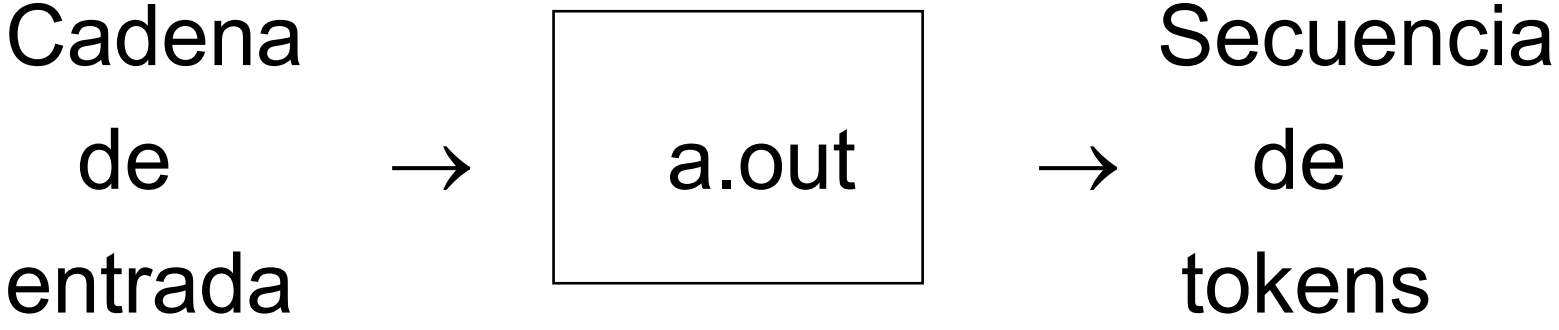
ident	l(l|d)*	posición: ref_T_S	*si* palres(str) *entonces*
			tipo:=PALRES;
			posición:=buscar_PR(T_S,str);
			sino
			tipo:=IDENT;
			str:=normalizar(str);
			posición:=añadir(T_S,str);

pal. res.	program| var| ...	posición: ref_T_S	
cte_ent	d+	valor: entero	tipo:=CTEENT; valor:=str2int(str);
cte_real	d+.d+	valor: real	tipo:=CTEREAL; valor:=str2real(str);

2.4 Un lenguaje de especificación de analizadores léxicos: LEX



2.4 Un lenguaje de especificación de analizadores léxicos: LEX



Especificaciones LEX (I)

DECLARACIONES

%%

REGLAS

%%

SUBRUTINAS DE USUARIO

Especificaciones LEX (II)

DECLARACIONES

Variables/constantes

Definiciones regulares

REGLAS

patrón$_i$ {acción$_i$}

Especificaciones LEX (III)

SUBPROGRAMAS DE USUARIO

- Procedimientos auxiliares
- Redefinición de procedimientos con los que trabaja LEX

Expresiones regulares en LEX (I)

- Caracteres de texto vs. Caracteres distinguidos
- ESCAPE \ y entre comillas "

  \n fin de línea \t tabulador

  \[o "[" corchete \a o "a" símbolo
- 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 de todos los símbolos excepto a, b y c

  Es el complementario de un conjunto, por tanto dentro de un conjunto

Expresiones regulares en LEX (II)

- EXPRESION OPCIONAL **?**
  ab?c {abc,ac}
- EXPRESIONES REPETIDAS * +
- ALTERNATIVAS |
  a|b {a,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

- No existe: copiar cadena de entrada
- ECHO: copiar cadena de entrada
- ACCION NULA: ;
- yymore()
- ...
- yywrap()

Estructuras que proporciona LEX al usuario

- **yytext**

  array de caracteres externo

- **yyleng**

  contador del número de caracteres emparejados

Tratamiento de la ambigüedad

Emparejamiento más largo

↓

Empate: 1ª 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 LEX
- Copia en lex.yy.c el fragmento de entrada LEX comprendido entre **%{** y **%}**

Ejemplo

```
%%
[bv]        {printf("he leido una b o una v");}
[^b]        {nob();}
.           ;
%%
void nob(){
  printf("<> de b");
}
```