Procesadores de Lenguajes

Ingeniería Técnica superior de Ingeniería Informática

Departamento de Lenguajes y Sistemas informáticos

2

Análisis léxico

Formalización y desarrollo

Javier Vélez Reyes
jvelez@lsi.uned.es
Departamento de Lenguajes Y Sistemas Informáticos
UNED

UNED

ETS de Ingeniería Informática

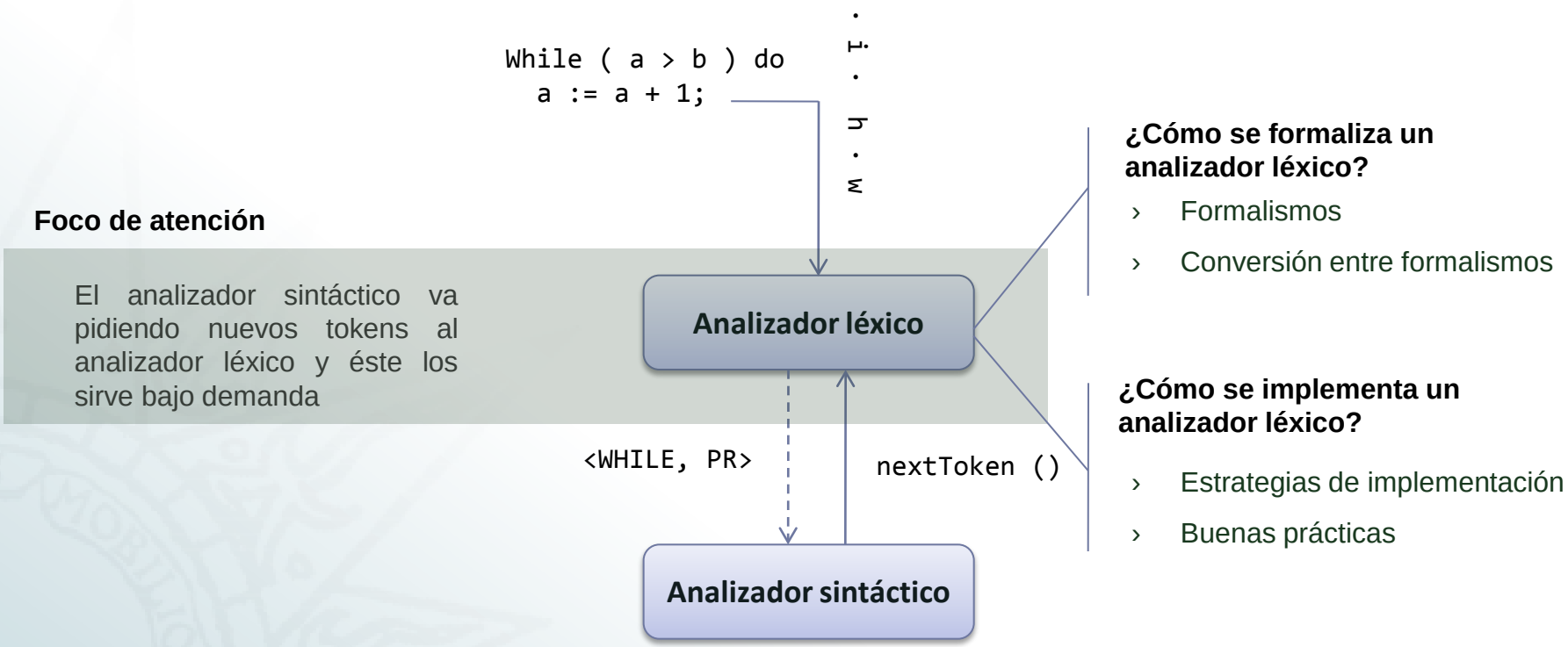
Objetivos

- Conocer las responsabilidades de un analizador léxico
  - Aprender cómo funciona
  - Entender los diferentes tipos de patrones que reconoce
- Aprender a especificar formalmente analizadores léxicos
  - A través de gramáticas formales
  - A través de expresiones regulares
  - A través de autómatas finitos
- Estudiar la implementación de analizadores léxicos
  - Conocer las distintas estrategias de implementación
  - Entender las posibles relaciones con la tabla de símbolos
- Estudiar la generación de errores léxicos

Índice

Introducción

El primer paso para procesar un programa es convertir la colección de caracteres del mismo en una colección de componentes léxicos con significado único dentro del lenguaje llamados tokens. De eso se encarga el analizador léxico o escáner



Introducción

El primer paso para procesar un programa es convertir la colección de caracteres del mismo en una colección de componentes léxicos con significado único dentro del lenguaje llamados tokens. De eso se encarga el analizador léxico o escáner

Definición de token

Un token es una unidad léxica indivisible con significado único dentro del lenguaje. Desde el punto de vista tecnológico se trata de una estructura de datos que contiene información sobre

- ID: Tipo del token
- Número de línea
- Número de columna
- Lexema
- Valor…

Categorías de token

Los tipos de tokens existentes en un lenguaje son una característica intrínseca al mismo. No obstante, pueden encontrarse categorías generales

Categoría	Ejemplos
Delimitadores	( ) , ; : [ ]
Palabras reservadas	while true do if for
Identificadores	Index f isEven
Números enteros	3 -4 55 0 7658
Números flotantes	4.5 .3 0.5 8.4e-5
Simbolos especiales	+ -* / . = < > <= >= != ++
Cadenas	“Hola mundo!”

Introducción

Cada tipo de token representa a un conjunto de tokens (construcciones léxicas diferentes) con unos mismos propósitos dentro del lenguaje. Estos tipos se definen a través de un patrón léxico al que se adscriben los lexemas del tipo

Definición de patrón léxico

Un patrón léxico es una expresión abstracta llamada expresión regular que permite identificar unívocamente un tipo de token y referenciar al conjunto de todos los tokens que se ajustan a él.

Definición de lexema

La cadena de caracteres específica de un token que se ajusta al patrón léxico de su tipo se llama lexema. Existen dos tipos de lexemas

- Cadena propia: lexema idéntica al patrón
- Cadena no propia: lexema encaja con patrón

Tipo de token	Patrón léxico	Ejemplos de lexema	Cadena propia
While	while	while	SI
Enteros	digito+	12 123 0 45	NO
División	/	/	SI
Identificador	letra (letra |digito)*	Index f isEven	NO

Introducción

Desde la perspectiva léxica un programa es una familia ordenada de tokens de varios tipos. Cada tipo, a través de su patrón léxico, define un micro lenguaje formado por todos aquellos lexemas que encajan con el patrón léxico. Este tipo de lenguajes sencillos se llaman lenguajes regulares

Definición de lenguaje regular

Un lenguaje regular describe una familia de tokens que se ajustan a un determinado patrón léxico.

Lenguaje de delimitadores de Pascal

```
;
=
,
:
[
..
]
(
)
...
```

Lenguaje de palabras reservadas de Pascal

```
program do
uses    downti
const   if
var     then
intger  End
of      ...
array
begin
for
to
```

Lenguaje de identificadores Pascal

```
burbuja
ctr
n
i
j
temp
a
readln
writeln
...
```

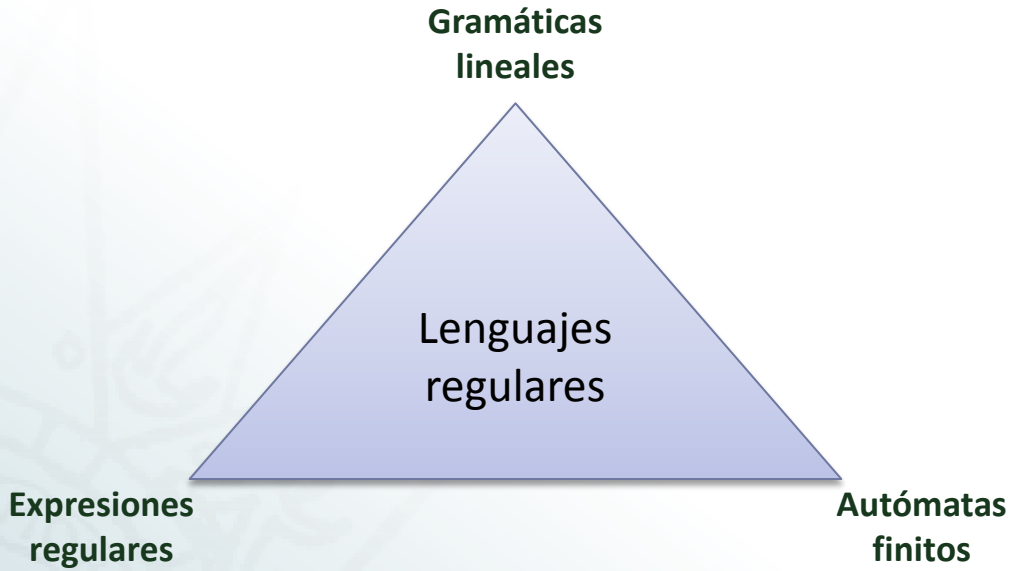
```
program burbuja;
uses crt;
const
   n = 5;
var
   i,j,temp:integer;
   a:array[1..n] of integer;
begin
   for i := 1 to n do
      readln(a[i]);

   for j := (n - 1) downto 1 do
      for i := 1 to j do
         if (a[i])>(a[i+1]) then
         begin
            temp   := a[i];
            a[i]   := a[i+1];
            a[i+1] := temp;
         end;

   writeln ('El resultado es:');
   for i := 1 to n do
      writeln (a[i]);
   readln;
end.
```

Especificación de lenguajes regulares

Además de la declaración extensiva – sólo viable en los lenguajes finitos – existen 3 diferentes maneras de definir formalmente un lenguaje regular. A lo largo de esta sección estudiaremos cada una de ellas en detalle y veremos cómo se puede pasar de cada una a las otras 2



- Elementos
  - Gramáticas lineales
  - Expresiones regulares
  - Autómatas finitos
- Transformaciones
  - De gramáticas a expresiones
  - De gramáticas a autómatas
  - De expresiones a gramáticas
  - De expresiones a autómatas
  - De autómatas a gramáticas
  - De autómatas a expresiones

Especificación mediante gramáticas lineales

Una forma de definir formalmente un lenguaje regular es utilizar una gramática lineal, que describe todas las reglas que se pueden aplicar para la construcción de lexemas que pertenecen al lenguaje regular

Definición de gramática lineal

Una gramática lineal es un conjunto de 4 elementos G = (T, N, S, P) donde:

- T es un conjunto de símbolos terminales
- N es un conjunto de símbolos no terminales
- S Є N axioma gramatical
- P un conjunto de reglas de producción de la forma
  1. A ::= B x donde A, B Є N y x Є T
  2. A ::= x B donde A, B Є N y x Є T
  3. A ::= x donde x Є T U { }

A veces T se elide, N se deduce de los antecedentes de las reglas de P y se asume que el antecedente de la primera regla es S con lo G sólo a través de P

representa la cadena vacía (ausencia de terminal)

Tipos de gramáticas lineales

Existen en realidad 2 tipos de gramáticas lineales. En función de la forma del conjunto de reglas de producción podemos distinguir entre:

- Gramáticas lineales por la izquierda
  - Usan reglas A ::= B x
  - Usan reglas A ::= x
- Gramáticas lineales por la derecha
  - Usan reglas A ::= x B
  - Usan reglas A ::= x

Dada una gramática G lineal por la izquierda siempre se puede encontrar una gramática G' lineal por la derecha y recíprocamente

Especificación mediante gramáticas lineales

Una forma de definir formalmente un lenguaje regular es utilizar una gramática lineal, que describe todas las reglas que se pueden aplicar para la construcción de lexemas que pertenecen al lenguaje regular

Derivación gramatical

Una gramática debe interpretarse como un conjunto de reglas de reescritura o transformación de elementos no terminales en elementos terminales o no terminales.

- La primera transformación parte del axioma
- En cada paso se aplica una única regla
  - Las reglas pueden extender la transformación (A::= B x / A ::= x B)
  - Las reglas pueden ser recursivas (A ::= A x / A ::= x A)
  - Las reglas pueden ser terminales (A ::= x)
- El proceso debe ser convergente

Ejemplo

L_G = {a b*}

Sea G = (T, N A, P) con

T = {a, b}

N = {A, B}

P = {

A ::= a B

A ::= a

B ::= b B

B ::= b

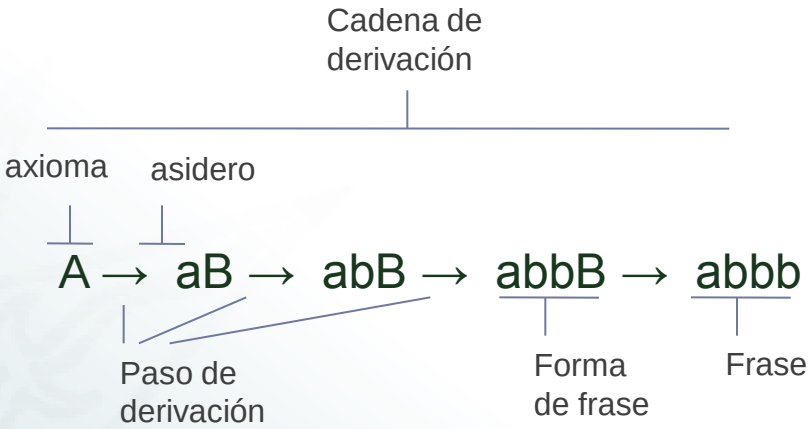
}

Especificación mediante gramáticas lineales

Una forma de definir formalmente un lenguaje regular es utilizar una gramática lineal, que describe todas las reglas que se pueden aplicar para la construcción de lexemas que pertenecen al lenguaje regular

Derivación gramatical

El problema ahora se traduce en, dada una cadena formada por una secuencia de elementos terminales de T, demostrar que dicha cadena pertenece al lenguaje definido por la gramática



Ejemplo

L_G = {a b*}

Sea G = (T, N A, P) con

T = {a, b}

N = {A, B}

P = {

A ::= a B

A ::= a

B ::= b B

B ::= b

}

Especificación mediante gramáticas lineales

Una forma de definir formalmente un lenguaje regular es utilizar una gramática lineal, que describe todas las reglas que se pueden aplicar para la construcción de lexemas que pertenecen al lenguaje regular

Ejercicios

Definir formalmente a través de una gramática lineal los siguientes lenguajes descritos informalmente

- L1 = conjunto de todas las palabras sobre {a, b} que empiezan por a
- L2 = conjunto de todas las palabras sobre {a, b} que empiezan por a y continúan con secuencias ab
- L3 = Números fraccionarios
- L4 = Identificadores

L1

GD = (T, N A, P) con

T = {a, b}

N = {A, B}

P = {

A ::= a B

B ::= a B

B ::= b B

B ::=

}

GI = (T, N A, P) con

T = {a, b}

N = {A, B}

P = {

A ::= A a

A ::= A b

A ::= a

}

Especificación mediante expresiones regulares

Otra forma de definir lenguajes regulares es a través del uso de expresiones regulares. Una expresión regular utiliza los términos del alfabeto de terminales operados a través de operaciones con una semántica especifica

Definición de expresión regular

Una expresión regular sobre un conjunto T es un conjunto ER = (T, | , · , *) que cumple las siguientes propiedades:

- Ø es una expresión regular que define el lenguaje L = Ø
- (cadena vacía) es una expresión regular que define el lenguaje L ( ) = { }
- Cualquier símbolo a Є T es una expresión regular que define el lenguaje L (a) = { a }
- Si x, y son dos expresiones regulares, x·y es una expresión regular que define L (x·y) = L (x) L (y)
- Si x, y son dos expresiones regulares, x | y es una expresión regular que define L (x | y) = L (x) U L (y)
- Si x es una expresión regular x* es una expresión regular que define $L (x^*) = \bigcup_{i=0}^{\infty} L(x)^i$
- Si x es una expresión regular x+ es una expresión regular que define el lenguaje $L (x^*) = \bigcup_{i=1}^{\infty} L(x)^i$
- No existen más expresiones regulares

Especificación mediante expresiones regulares

Otra forma de definir lenguajes regulares es a través del uso de expresiones regulares. Una expresión regular utiliza los términos del alfabeto de terminales operados a través de operaciones con una semántica especifica

Ejercicios

Definir los siguientes lenguajes regulares utilizando para ello expresiones regulares

- L1 = Lenguaje de identificadores
- L2 = Lenguaje de números naturales
- L3 = Lenguaje de números fraccionarios
- L4 = Lenguaje de números enteros con exponente
- L5 = Lenguaje de números fraccionarios con exponente
- L6 = Lenguaje de números pares
- L7 = Lenguaje de las cuentas de correo
- L8 = Lenguajes de las URL

L1 = l (l | d)* l Є {a..z} d Є {0..9}

L3 = d+ . d+ d Є {0..9}

L6 = d+ p d Є {0..9} p Є {0, 2, 4, 6, 8}

L7 = nombre @ host

nombre = L1 (. L1)*

host = L1 (. L1)*

Especificación mediante autómatas finitos

La tercera forma de definir lenguajes regulares es a través del uso de autómatas finitos. Un autómata finito reconoce una cadena de entrada si consumidos todos los caracteres de la misma acaba en un estado final de aceptación

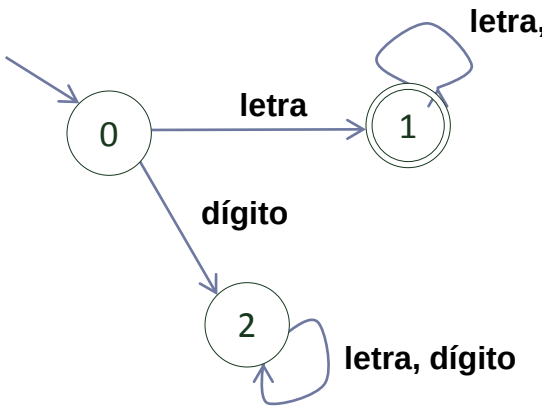
Autómata finito determinista

Un autómata finito determinista es un conjunto AFD = (T, Q, f, q, F) donde:

- T es un alfabeto de símbolos terminales de entrada
- Q es un conjunto de estados finito no vacío
- f: Q x T → Q es una función de transición
- q Є Q es un estado inicial o estado de arranque
- F C Q es un subconjunto de estados finales de aceptación

Interpretación gráfica

Gráficamente, un autómata finito determinista es una colección de estados unidos por arcos, donde para cada estado existe un arco etiquetado con cada elemento de T. El estado inicial tiene una flecha entrante y los estados finales tienen un doble borde



letra Є [a..z]
dígito Є [0..9]

Especificación mediante autómatas finitos

La tercera forma de definir lenguajes regulares es a través del uso de autómatas finitos. Un autómata finito reconoce una cadena de entrada si consumidos todos los caracteres de la misma acaba en un estado final de aceptación

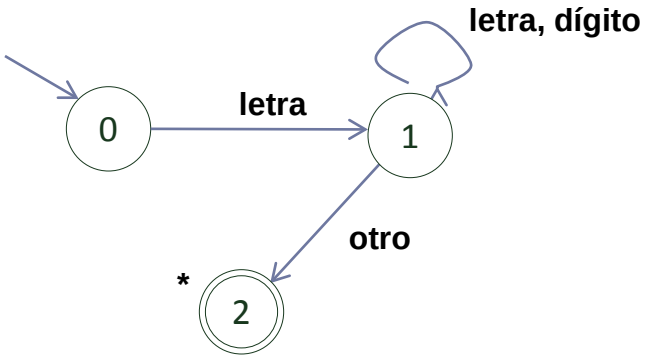
Extensión de los autómatas finitos deterministas

Con el ánimo de simplificar el trabajo con los autómatas finitos deterministas, dentro de la teoría de compiladores se utilizan las siguientes extensiones notacionales

- Las transiciones ausentes se asumen como errores
- Se omiten los estados de absorción de errores
- Se usa la etiqueta *otro* para representar el resto de alternativas
- Los estados finales paran el proceso y emiten token
- Los estados finales con * recuperan el carácter de tránsito
- Se espera a reconocer el posible toquen de lexema más largo
  - El estado final ya no es final
  - Al llegar un carácter terminador se pasa a uno final
  - Entonces se emite el token
  - Si es necesario se marca con * el estado final

Interpretación gráfica

A continuación representamos el autómata finito determinista anterior aplicando las extensiones descritas



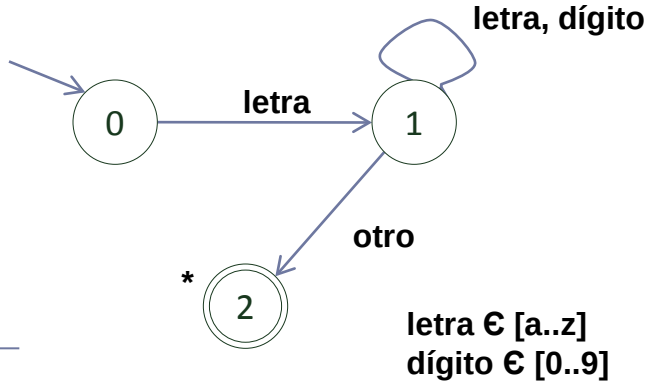
letra Є [a..z]
dígito Є [0..9]

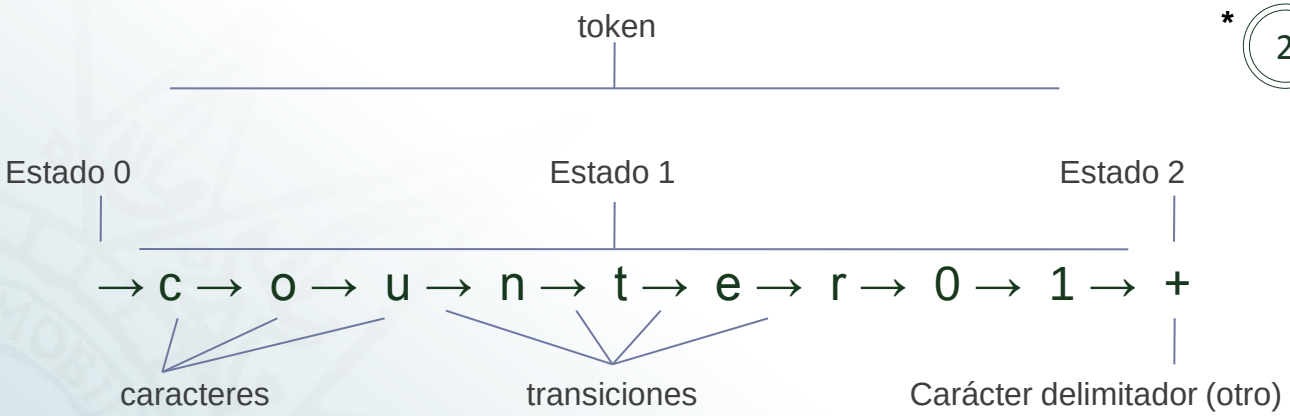
Especificación mediante autómatas finitos

La tercera forma de definir lenguajes regulares es a través del uso de autómatas finitos. Un autómata finito reconoce una cadena de entrada si consumidos todos los caracteres de la misma acaba en un estado final de aceptación

Como se usan los autómatas finitos deterministas

Los autómatas arrancan en el estado de inició y comienzan a consumir caracteres de la entrada lo que provoca una transición a un nuevo estado o un error. En cuanto se alcanza un estado de aceptación final, el autómata para y se emite el token reconocido





Especificación mediante autómatas finitos

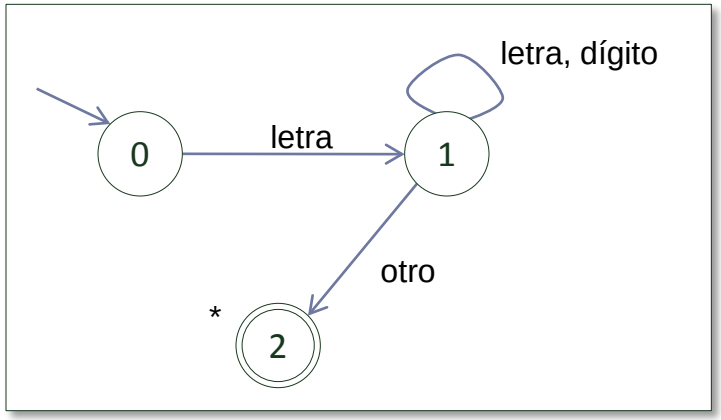
La tercera forma de definir lenguajes regulares es a través del uso de autómatas finitos. Un autómata finito reconoce una cadena de entrada si consumidos todos los caracteres de la misma acaba en un estado final de aceptación

Ejercicios

Definir los siguientes lenguajes regulares utilizando para ello autómatas finitos deterministas

- L1 = Lenguaje de identificadores
- L2 = Lenguaje de números naturales
- L3 = Lenguaje de números fraccionarios
- L4 = Lenguaje de números naturales con exponente
- L5 = Lenguaje de números fraccionarios con exponente
- L6 = When

L1



Especificación mediante autómatas finitos

La tercera forma de definir lenguajes regulares es a través del uso de autómatas finitos. Un autómata finito reconoce una cadena de entrada si consumidos todos los caracteres de la misma acaba en un estado final de aceptación

Construcción manual compositiva de autómatas

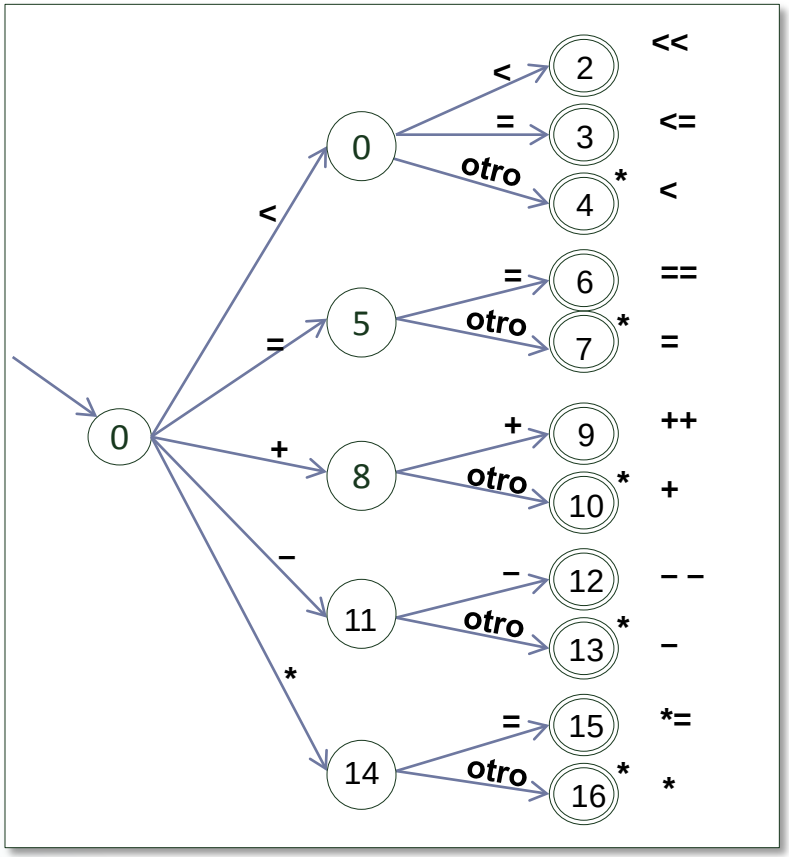
Para construir un analizador léxico completo es necesario combinar en un único autómata todos los autómatas correspondientes a los micro-lenguajes que aparecen en el lenguaje. Este proceso se puede hacer mediante una aproximación holística

Ejercicios

Definir los siguientes lenguajes regulares utilizando para ello autómatas finitos deterministas

- L7 = Números naturales y fraccionarios con exponente
- L8 = {<, <<,<=, =, ==, +, ++, -, --, *, *=}
- L9 = {While, When, Do, Downto}
- L10 = L1 U L7 U L9
- L11 = U Li con i Є [1..11]

L8



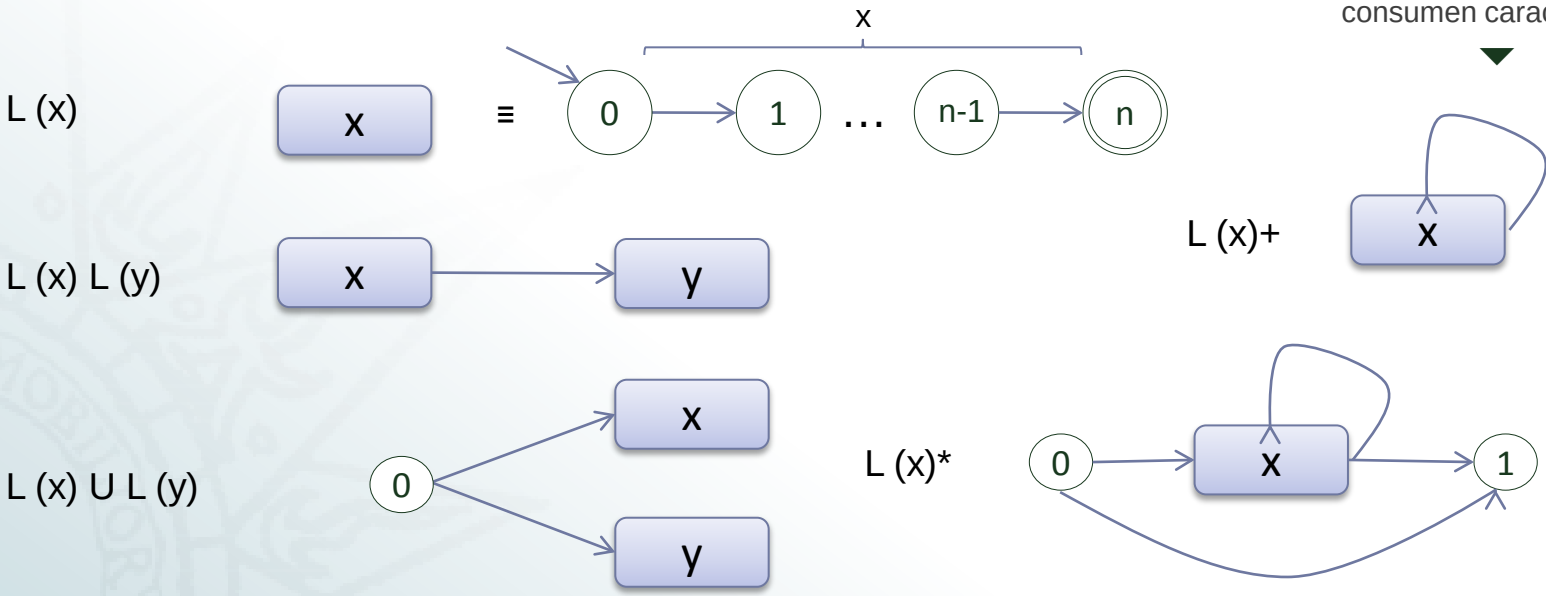
Especificación mediante autómatas finitos

La tercera forma de definir lenguajes regulares es a través del uso de autómatas finitos. Un autómata finito reconoce una cadena de entrada si consumidos todos los caracteres de la misma acaba en un estado final de aceptación

Construcción sistemática compositiva de autómatas

Cuando el número de micro-lenguajes es grande y éstos son complejos la aproximación manual se convierte en un proceso prolijo por eso conviene establecer un proceso sistemático. Este proceso se basa en definir formalmente la composición de autómatas sobre las operaciones legítimas de composición de autómatas

Las – transiciones deben interpretarse como transiciones que no consumen caracteres



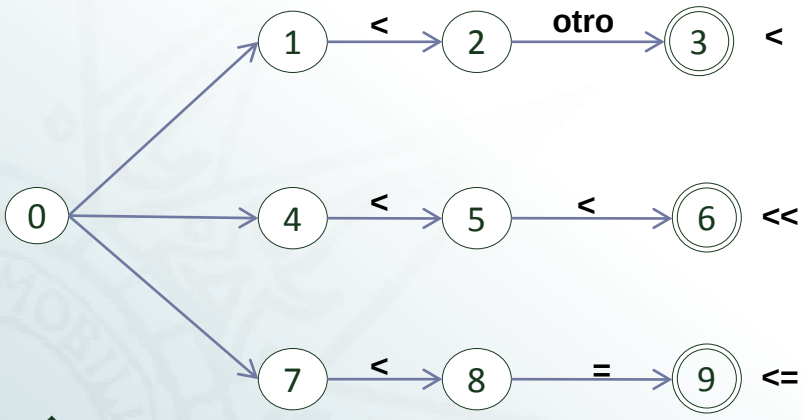
Especificación mediante autómatas finitos

La tercera forma de definir lenguajes regulares es a través del uso de autómatas finitos. Un autómata finito reconoce una cadena de entrada si consumidos todos los caracteres de la misma acaba en un estado final de aceptación

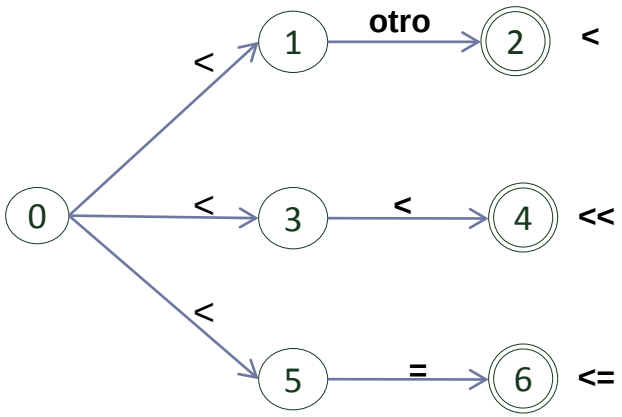
Construcción sistemática compositiva de autómatas

Cuando el número de micro-lenguajes es grande y éstos son complejos la aproximación manual se convierte en un proceso prolijo por eso conviene establecer un proceso sistemático. Este proceso se basa en definir formalmente la composición de autómatas sobre las operaciones legítimas de composición de autómatas

Ejemplo L = {<, <<, <=}



Uso de –transiciones para construir el autómata



La eliminación de las –transiciones provoca varias transiciones iguales desde el mismo nodo

Especificación mediante autómatas finitos

La tercera forma de definir lenguajes regulares es a través del uso de autómatas finitos. Un autómata finito reconoce una cadena de entrada si consumidos todos los caracteres de la misma acaba en un estado final de aceptación

Autómatas finitos no deterministas

La construcción sistemática de autómatas genera autómatas finitos no deterministas. Un autómata finito no determinista es un autómata finito donde cada estado:

- Puede tener – transiciones

  Genera indeterminación porque desde ese estado no se sabe si avanzar por otra transición normal consumiendo un carácter o hacerlo por la – transición sin consumir ningún carácter

- Puede tener varias transiciones con la misma etiqueta

  Genera indeterminación porque desde ese estado, si el carácter a la entrada tiene varias posibles transiciones no se sabe cual escoger

Los autómatas finitos deterministas no son formalismos adecuados para representar lenguajes regulares

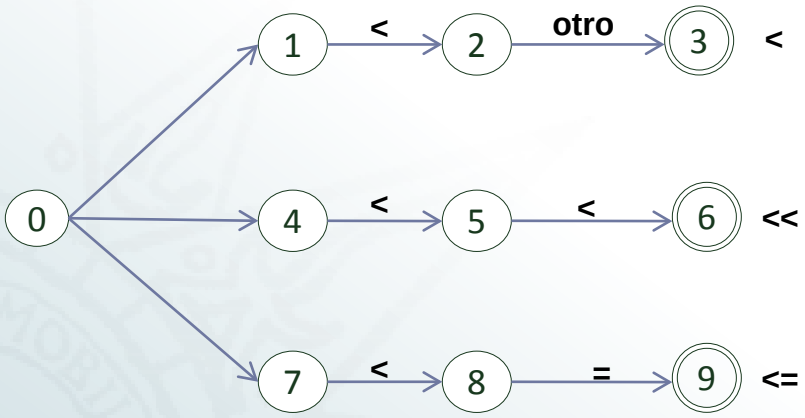
Especificación mediante autómatas finitos

La tercera forma de definir lenguajes regulares es a través del uso de autómatas finitos. Un autómata finito reconoce una cadena de entrada si consumidos todos los caracteres de la misma acaba en un estado final de aceptación

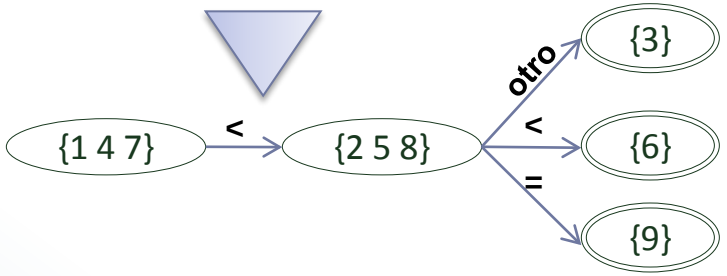
Conversión de autómatas finitos no deterministas a autómatas finitos deterministas

La construcción de un autómata finito determinista a partir de uno no determinista requiere de la identificación de todos los estados alcanzables desde cada estado a través de –transiciones y transiciones múltiples

Ejemplo L = {<, <<, <=}

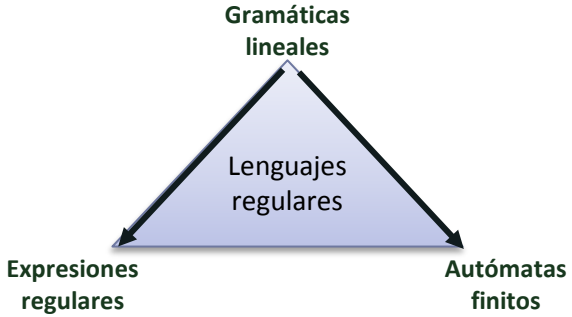


- {0} = {1 4 7}
- {1 4 7} $_{<}$ = {2 5 8}
- {2 5 8} $_{<}$ = {6}
- {2 5 8} $_{=}$ = {9}
- {2 5 8} $_{otro}$ = {3}



Conversión de formalismos

Dado cualquiera de los 3 formalismos de representación estudiados es posible encontrar un isomorfismo para expresarlo en cualquiera de los otros 2



Conversión de Gramáticas a expresiones

Cada regla se expresa en términos de los operadores de concatenación, cierre + y cierre *

A ::= l B A = l B

B ::= l B B = l B = l +

B ::= d B B = d B = d +

B ::= B =

B = (l+ | d+ |) = (l | d |)+ = (l | d)*

A = l (l | d) *

Conversión de Gramáticas a autómatas

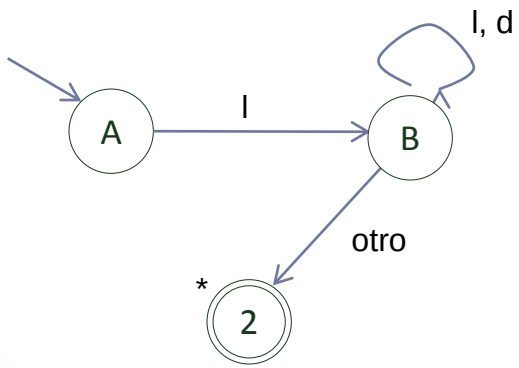
Se genera un estado por cada no terminal. El estado del axioma es el de inicio. Las reglas se traducen a transiciones. Si la regla es terminal (A ::= x) entonces se genera una transición otro a un estado de aceptación final. Si sale no determinista, convertirlo a determinista.

A ::= l B

B ::= l B

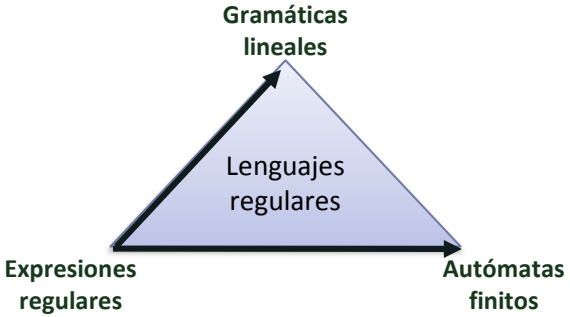
B ::= d B

B ::=



Conversión de formalismos

Dado cualquiera de los 3 formalismos de representación estudiados es posible encontrar un isomorfismo para expresarlo en cualquiera de los otros 2



Conversión de Expresiones a gramáticas

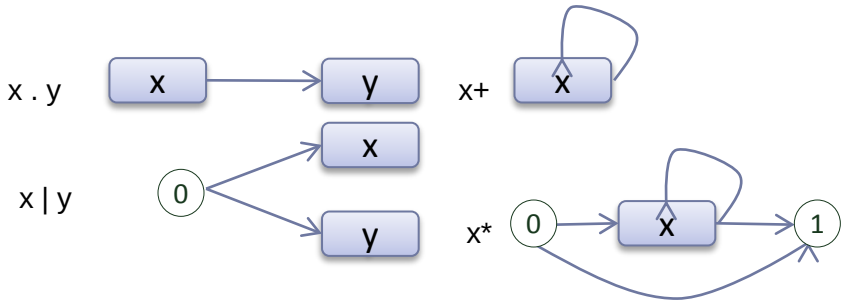
Transformar primero la expresión regular desfactorizando los cierres + y *. Después aplicar las transformaciones + a recursividad y * a recursividad con producción

L = l (l | d) * = l (l * | d *)

A ::= l B	B representa (l* | d*)
B ::= l B	Se captura l+
B := d B	Se captura d+
B ::=	Se captura l* | d*

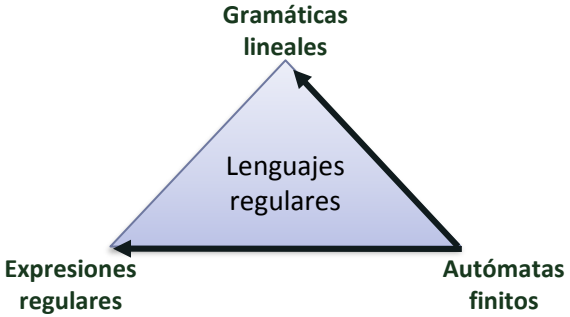
Conversión de Expresiones a autómatas

La conversión de expresiones regulares a autómatas finitos se realiza utilizando las transformaciones atómicas definidas anteriormente al discutir la composición de lenguajes



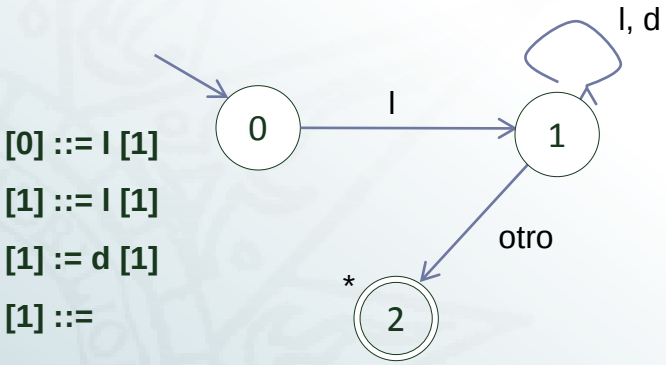
Conversión de formalismos

Dado cualquiera de los 3 formalismos de representación estudiados es posible encontrar un isomorfismo para expresarlo en cualquiera de los otros 2



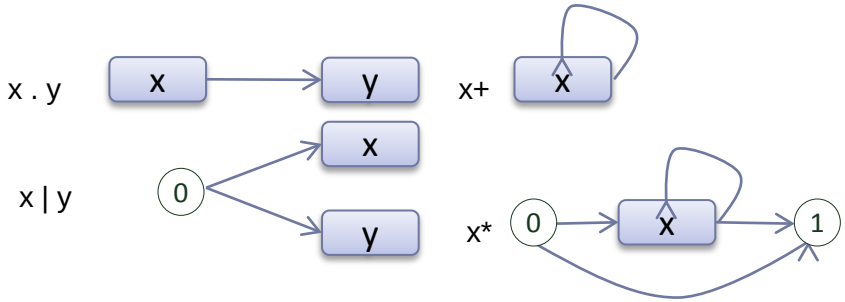
Conversión de autómatas a gramáticas

Cada estado se convierte a un no terminal. El estado de arranque corresponde con el axioma. Cada transición A → B etiquetada con x se traduce en A ::= x B. Si B es un estado final se genera A ::= x. Si x = otro se genera A ::=



Conversión de autómatas a expresión

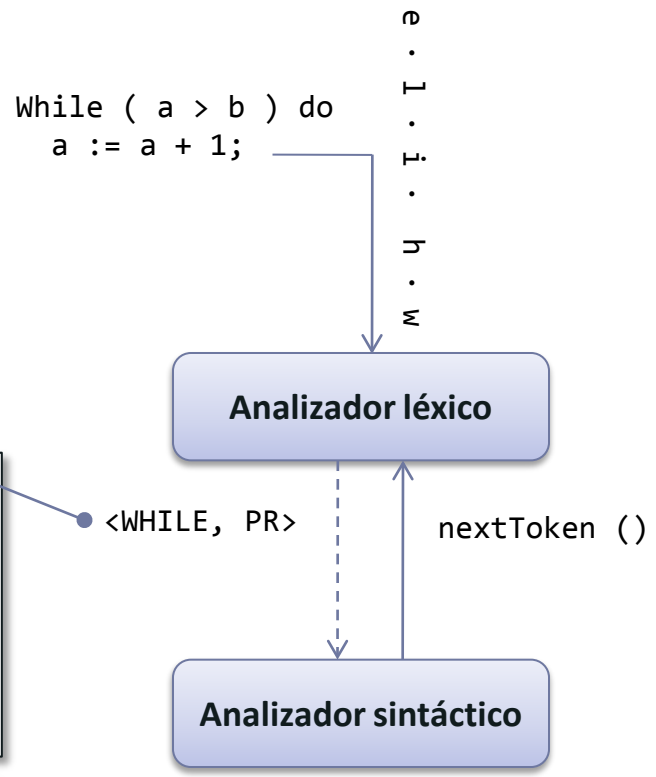
Para la transformación de autómatas a expresiones basta con hacer una lectura inversa de las transformaciones atómicas definidas anteriormente



¿Qué es un analizador léxico?

Un analizador léxico o scanner es un programa capaz de descomponer una entrada de caracteres – generalmente contenidas en un fichero – en una secuencia ordenada de tokens.

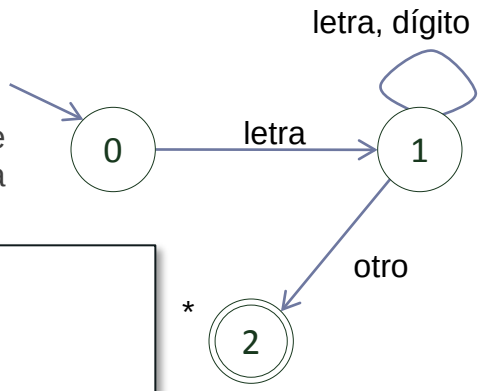
El analizador léxico responde bajo demanda a las solicitudes de siguiente token que le va haciendo el analizado sintáctico. Cada vez que éste último hace una solicitud al primero, el analizador léxico consume un número de caracteres de la entrada y retorna un artefacto computacional que representa el siguiente token en la entrada. En lo venidero utilizaremos el término token para referirnos a dicho artefacto



En la práctica el artefacto software que representa un token suele ser una estructura de datos o un objeto en los lenguajes orientados a objetos

Implementación de analizadores léxicos basada en casos

De acuerdo a esta estrategia de implementación es necesario disponer de una variable de estado para codificar el estado actual y una batería de casos que describen la lógica de transición del autómata finito determinista.

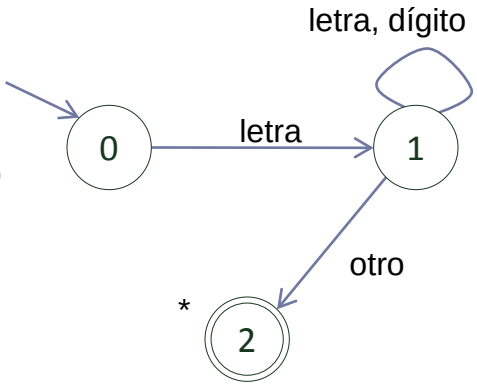



```
letra  = [a..z]
digito = [0..9]
estado = 0
fin = false
c = leerCaracter ()
while (!fin) {
  switch (estado) {
   0: switch (c) {
        letra : estado = 1
                lexema += c
                c = leerCaracter ()
                break
        digito : throw error ()
      }
```

```
      break;
   1: switch (c) {
         letra  :
         digito : estado = 1
                  lexema += c
                  c = leerCaracter ()
                  break
          default: estado = 2;
      }
      break
   2: token = crearToken (lexema...)
      fin = true
}
return token
```

Implementación de analizadores léxicos dirigida por tabla

Podemos codificar computacionalmente un autómata como una matriz bidimensional de Q x T → Q que indique, para cada entrada y estado el nuevo estado. El algoritmo se limita a leer dicha matriz




```
tTransicion = [][]
sFinales = {estados finales}
estado = 0
fin = false
c = leerCaracter ()
while (!fin) {
  s = tTransicion [s][c]
  if (s != null)
    if (!(s in sFinales)) {
       lexema += c
       c = leerCaracter ()
    } else fin = true
  else throw error ()  }
return crearToken (lexema...)
```

Implementación de analizadores léxicos guiada por herramientas

Existen herramientas para la construcción automática de analizadores léxicos [Lex] [Flex] [JFlex]. Estas herramientas se basan en la definición de una colección de reglas patrón - acción

```
letra = [a..z]
digito = [0..9]

letra (letra | digito)*  { return new Token (ID);   }
digito+                  { return new Token (ENT);  }
digito* . digito+        { return new Token (FRAC); }
“(”                      { return new Token (PI);   }
“)”                      { return new Token (PD);   }
“[”                      { return new Token (PI);   }
“]”                      { return new Token (PD);   }

...
```

- Dan prioridad al token más largo
  - DO / DOT
  - > / >=
- Ante igualdad de longitud
- Anteponer la regla mas específica
  - When
  - While
  - Id

Estrategias de implementación

I. Reconocimiento de palabras reservadas

Reconocimiento de palabras reservadas

Resolución explícita

- Se indican todas con su patrón léxico
- Se integran en el diagrama global
- Se utilizan herramientas generadoras (FLex)

Resolución implícita

- Las palabras reservadas (PR) se reconocen como identificadores
- Las k palabras reservadas se meten en la tabla de símbolos (TS)
- Se busca cada id en TS. Si su índice es < k es PR

II. Reconocimiento estructuras complejas de forma híbrida

- Implementación manual de las estructuras más sencillas
  - Operadores
  - Identificadores
  - Números
- Implementación tabular o guiada por herramientas con estructuras complejas
  - Cadenas no específicas
  - Prefijos comunes

Errores léxicos

Los errores léxicos son aquellos que se producen en el ámbito del reconocimiento de patrones para construir tokens del lenguaje. Podemos establecer una clasificación de errores de naturaleza léxica

- Errores básicos
  - Aparición de caracteres ajenos al conjunto T (ñ, Ç, etc.)
  - Ausencia de concordancia con ningún patrón (1abc, +*, etc.)
- Errores complejos
  - Cadena de caracteres sin cierre (falta “ final)
  - Cadena de caracteres sin apertura (falta “ inicial)
  - Comentario sin cierre (falta */ final)
  - Comentario sin apertura (falta /* inicial)
  - Error en el anidamiento de comentarios (/* .. /* .. */)

Son en realidad de esta categoría

No son en realidad errores capturables por un reconocedor de lenguajes regulares aunque las herramientas generadoras son capaces de reconocerlos

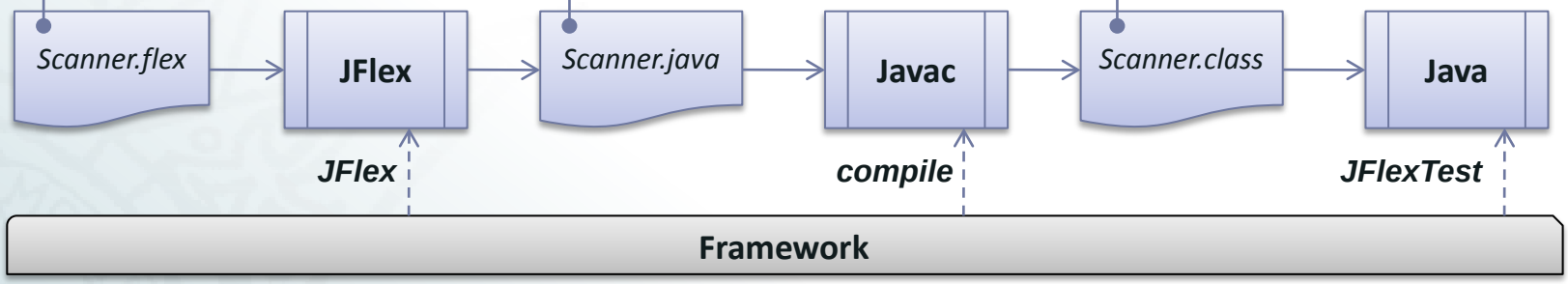
¿Qué es JFlex?

JFlex es una herramienta para la generación de analizadores léxicos escritos en java. A partir de un fichero de especificación que describe las características léxicas de un lenguaje, JFlex genera un código fuente compilable que puede ser utilizado como analizador léxico

Esta es la especificación del analizador léxico. Todo cambio en el escáner debe indicarse aquí

Esta es la clase generada a partir de la especificación. Atención! No incluir aquí ningún código manualmente ya que en la próxima generación se perderá

Esta es la clase java compilada que puede ser interpretada por la JVM



¿Cómo funciona JFlex?

El fichero de especificación de JFlex esta dividido en tres secciones separadas por el delimitador %%. Cada una de ellas tiene una semántica diferente

I. Sección de código de usuario

La sección de código de usuario se utiliza para incluir cualquier declaración java (paquete, importación o clase) que sea necesario para compilar el scanner

```
1  package compiler.lexical;
2
3  import compiler.syntax.sym;
4  import compiler.lexical.Token;
5  import es.uned.compiler.lexical.ScannerIF;
6  import es.uned.compiler.lexical.LexicalError;
7  import es.uned.compiler.lexical.LexicalErrorManager;
8
9  %%
10
```

```
1  package compiler.lexical;
2
3  import compiler.syntax.sym;
4  import compiler.lexical.Token;
5  import es.uned.compiler.lexical.ScannerIF;
6  import es.uned.compiler.lexical.LexicalError;
7  import es.uned.compiler.lexical.LexicalErrorManager;
8
9  %%
10
11 %public
12 %class Scanner
13 %char
14 %line
15 %column
16 %cup
17 %implements ScannerIF
18 %scanerror LexicalError
19
20 %{
21   LexicalErrorManager lexicalErrorManager = new LexicalErrorManager ();
22   private int commentCount = 0;
23
24 Token () {
25 n = new Token (sym.PLUS);
26 ine (yyline + 1);
27 olumn (yycolumn + 1);
28 exema (yytext ());
29 en;
30
31
32
33 -z]
34
35 b\012]
36
37 %%
38
39 <YYINITIAL>     "+"     {  return createToken (MAS);   }
40 <YYINITIAL>     "-"     {  return createToken (MENOS); }
41 <YYINITIAL>     "*"     {  return createToken (POR);   }
42 <YYINITIAL>     "/"     {  return createToken (DIV);   }
43 ...
44 <YYINITIAL>     .       {  error (); }
45
46 <COMMENT>       "/*"    { ... }
47 <COMMENT>       "*/"    { ... }
```

¿Cómo funciona JFlex?

El fichero de especificación de JFlex esta dividido en tres secciones separadas por el delimitador %%. Cada una de ellas tiene una semántica diferente

II. Sección de directivas

Se incluyen directivas de compatibilidad con cup, gestión de fila, columna y lexema, declaración de estados y macros e inclusión de código de inicialización

```
1   package compiler.lexical;
2
3   import compiler.syntax.sym;
4   import compiler.lexical.Token;
5   import es.uned.compiler.lexical.ScannerIF;
6   import es.uned.compiler.lexical.LexicalError;
7   import es.uned.compiler.lexical.LexicalErrorManager;
8
9   %%
10
11  %public
12  %class Scanner
13  %char
14  %line
15  %column
16  %cup
17  %implements ScannerIF
18  %scanerror LexicalError
19
20  %{
```

```
11  %public
12  %class Scanner
13  %char
14  %line
15  %column
16  %cup
17  %implements ScannerIF
18  %scanerror LexicalError
19
20  %{
21    LexicalErrorManager lexicalErrorManager = new LexicalErrorManager ();
22    private int commentCount = 0;
23
24    Token createToken () {
25      Token token = new Token (sym.PLUS);
26      token.setLine (yyline + 1);
27      token.setColumn (yycolumn + 1);
28      token.setLexema (yytext ());
29      return token;
30    }
31  %}
32
33  ALFA  = [A-Za-z]
34  DIGIT = [0-9]
35  SPACE = [\ \t\b\012]
36
```

¿Cómo funciona JFlex?

El fichero de especificación de JFlex esta dividido en tres secciones separadas por el delimitador %%. Cada una de ellas tiene una semántica diferente

III. Sección de reglas patrón - acción

En esta última sección se define una familia de escáneres léxicos, cada uno asociado a un *estado* distinto (YYINITIAL, COMMENT, etc.). Es posible saltar de uno a otro. La definición se basa en reglas patrón acción que indican el código java que debe ejecutarse cuando la entrada encaja con determinado patrón léxico. Si el código java no acaba con un return el scanner continua buscando un nuevo token

```
37  %%
38
39  <YYINITIAL>     "+"     {  return createToken (MAS);    }
40  <YYINITIAL>     "-"     {  return createToken (MENOS); }
41  <YYINITIAL>     "*"     {  return createToken (POR);    }
42  <YYINITIAL>     "/"     {  return createToken (DIV);    }
43  ...
44  <YYINITIAL>     .       {  error (); }
45
46  <COMMENT>       "/*"    { ... }
47  <COMMENT>       "*/"    { ... }
```

```
1   package compiler.lexical;
2
3   import compiler.syntax.sym;
4   import compiler.lexical.Token;
5   import es.uned.compiler.lexical.ScannerIF;
6   import es.uned.compiler.lexical.LexicalError;
7   import es.uned.compiler.lexical.LexicalErrorManager;
8
9   %%
10
11  %public
12  %class Scanner
13  %char
14  %line
15  %column
16  %cup
17  %implements ScannerIF
18  %scanerror LexicalError
19
20  %{
21    LexicalErrorManager lexicalErrorManager = new LexicalErrorManager ();
22    private int commentCount = 0;
23
24    Token createToken () {
25      Token token = new Token (sym.PLUS);
26      token.setLine (yyline + 1);
27      token.setColumn (yycolumn + 1);
28      token.setLexema (yytext ());
```

¿Cómo funciona JFlex?

Consúltese el documento "Manual de JFlex"

I. Sección de código de usuario

La sección de código intermedio se copia tal cual al inicio de la clase fuente que representa el escáner (Scanner.java), por tanto todo lo que va en esta sección es código puro java de carácter declarativo

Copia en vervatim

```
1   package compiler.lexical;
2
3   import compiler.syntax.sym;
4   import compiler.lexical.Token;
5   import es.uned.compiler.lexical.ScannerIF;
6   import es.uned.compiler.lexical.LexicalError;
7   import es.uned.compiler.lexical.LexicalErrorManager;
8
9   %%
10
```

Scanner.java

```
package compiler.lexical;

import compiler.syntax.sym;
import compiler.lexical.Token;
import es.uned.compiler.lexical.ScannerIF;
import es.uned.compiler.lexical.LexicalError;
import es.uned.compiler.lexical.LexicalErrorManager;

class Yylex {

      public Yylex (java.io.Reader reader) {
            ...
      }
      public Yylex (java.io.InputStream instream) {
            ...
      }

      ...
      ...
      ...

      public Yytoken yylex ()
            throws java.io.IOException {
            ...
            return YyToken(...);
      }
}
```

¿Cómo funciona JFlex?

Consúltese el documento "Manual de JFlex"

II. Sección de directivas

Directiva	Descripción
› %full	Utiliza el código ASCII extendido 8 bits
› %unicode	Utiliza codificación UNICODE de 16 bits
› %{ … %}	Se incluye el código como declaración dentro de la clase Scanner
› %init{ … %}	Incluye código dentro de los constructores
› %intthow{ … %}	Declara excepciones que el constructor puede lanzar
› %yylexthrow{ … %}	Declara excepciones que el método de escaneo puede lanzar
› %eof{ … %}	Incluye código que se ejecuta tras encontrar EOF
› %eofval{ … %}	Define el valor de retorno al encontrar EOF
› %eofthrow{… %}	Excepción de se lanza al encontrar EOF
› %ignorecase	No distinguir entre mayúsculas y minúsculas
› %char	Contabiliza el número de caracteres en la variable yychar
› %line	Contabiliza el número de líneas en la variable yyline
› %notunix	Reconoce \r\n como carácter (doble) de nueva línea

¿Cómo funciona JFlex?

Consúltese el documento "Manual de JFlex"

II. Sección de directivas

Directiva	Descripción
%class name	Da nombre a la clase del escáner
%public	Define como pública la clase del escáner
%function name	Da nombre a la función de escaneo
%interface name	Declara los interfaces que debe implementar el escáner
%type name	Define el tipo de retorno de la función de escaneo
%integer	Define el tipo de retorno de la función de escaneo como un int
%intwrap	Define el tipo de retorno de la función de escaneo como Integer
%yyeof	Define la constante Yylex.YYEOF (obligatorio uso de %integer)
%cup	Habilita la compatibilidad con Cup
%state nombre	Define el estado nombre
nombre = valor	Define una macro (LETRA = [A-Za-z])

¿Cómo funciona JFlex?

Consúltese el documento "Manual de JFlex"

III. Reglas patrón-acción

› Despliegue de macros

Se utilizan los caracteres { y } para encerrar el nombre de la macro, definida en la sección de directivas, que se quiere desplegar

```
{LETRA} ( {LETRA} |{DIGITO})*  { return crearToken (ID);  }
{DIGITO} *                     { return crearToken (NUM); }
```

› Utilización de estados

Cada estado, declarado en la sección de directivas, corresponde con un autómata distinto. Cada regla debe asociarse a un estado precediendo al patrón léxico el nombre del mismo encerrando entre < y >

Desde una regla r de un estado A se puede saltar a un estado B mediante la instrucción **yybegin (B)** invocada en la acción de r.

El estado inicial por defecto es YYINITIAL. Este es un estado que no es preciso declarar

```
<YYINITIAL>  "+"    { return crearToken (MAS);    }
<YYINITIAL>  "-"    { return crearToken (MENOS);  }
<YYINITIAL>  "*"    { return crearToken (MUL);    }
<YYINITIAL>  "/"    { return crearToken (DIV);    }
...
<YYINITIAL>  "/*"   { yybegin (COMMENT);          }
...

<COMMENT> "/*"      { nComments ++; }
<COMMENT> "*/"      { nComments --;
                      if (nComments == 0)
                          yybegin (YYINITIAL);    }
<COMMENT> {COMMENT} { }
```

¿Cómo funciona JFlex?

Consúltese el documento "Manual de JFlex"

III. Reglas patrón-acción

Secuencias de escape	Descripción
\b	Retroceso
\n	Nueva línea
\t	Tabulador
\f	Avance de página
\r	Retorno de carro
\ddd	Número octal
\xdd	Número hexadecimal
\udddd	Hexadecimal de 4 dígitos
\^C	Carácter de control
\c	Backslash seguido del c

Secuencias de escape	Descripción
yychar	Número de columna
yyline	Numero de línea
yytext ()	lexema

Metacaracteres	Descripción
\b	Retroceso
$	Fin de fichero
.	Cualquier carácter menos \n
"…"	…
{name}	Expansión de una macro
*	Clausura de Kleene
+	Una o más repeticiones
?	Opcionalidad
(…)	Agrupación de expresiones regulares
[…]	Conjunto de caracteres
[^…]	Conjunto complementario
a-b	Rango a - b

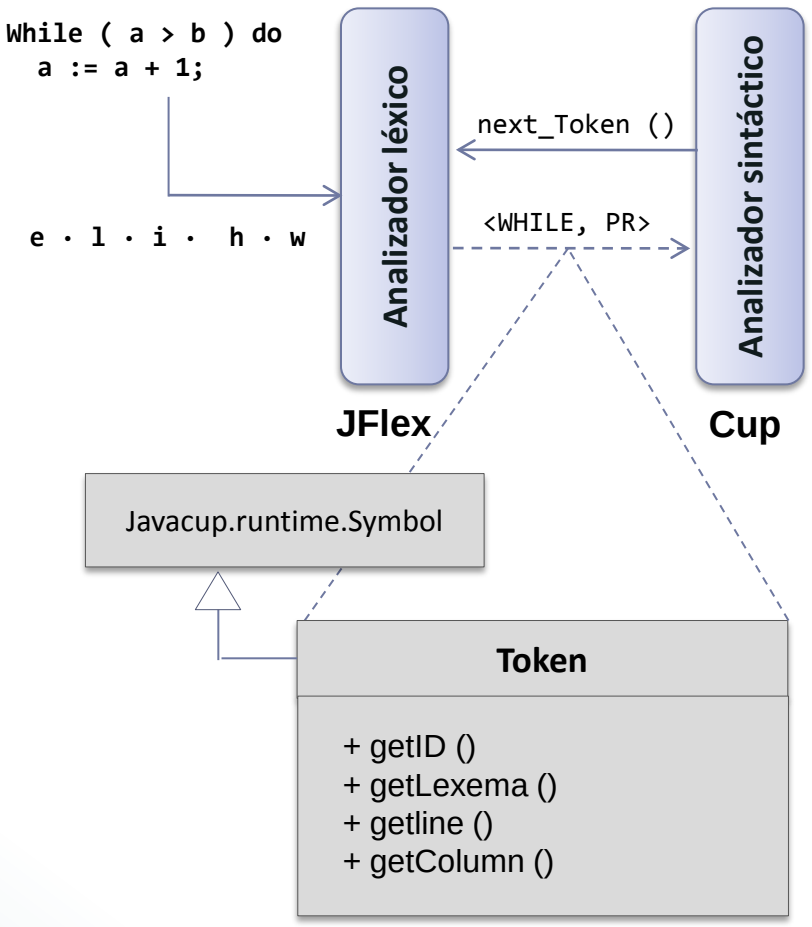
¿Cómo se usa JFlex?

Los analizadores sintácticos de JFlex están pensados para usarse bajo demanda. Se trata de artefactos con una API bien definida. El método más relevante es el de escaneo que devuelve el siguiente token a la entrada.

El framework prescribe que el método de escaneo debe devolver un objeto de la clase Token

```
Token next_Token ()
```

Consúltese el documento "Directrices de implementación"



La clase Token

Token es una clase hija que hereda de la clase que usa cup para integrarse con JFlex (javacup.runtime.Symbol). Esta clase facilita y encapsula la comunicación entre el analizador léxico y el analizador sintáctico desarrollado en Cup proporcionando una colección de métodos de interés

Análisis léxico. Formalización y desarrollo

Construcción de analizadores léxicos en la práctica

¿Cómo se usa JFlex?

Para adaptar JFlex al framework de desarrollo es necesario realizar una serie de tareas sobre el fichero de especificación. Éstas ya vienen realizadas en la plantilla Jflex que proporciona el framework por lo tanto lo aquí contado es meramente explicativo y no requiere trabajo alguno del estudiante

- El escáner pertenece al paquete compiler.lexical ❶
- Debe importarse ❷
  - La clase Token
  - El interfaz ScannerIF
  - Las clases de gestión de errores
- Debe activarse la compatibilidad con el framework ❸
  - %cup
  - %implements ScannerIF
  - %scanerror LexicalError
- Opcionalmente declararse la función crearToken ❹
- Cada acción debe acabar con un return crearToken (…); ❺

```
1  package compiler.lexical;                                          ❶
2
3  import compiler.syntax.sym;
4  import compiler.lexical.Token;                                     ❷
5  import es.uned.compiler.lexical.ScannerIF;
6  import es.uned.compiler.lexical.LexicalError;
7  import es.uned.compiler.lexical.LexicalErrorManager;
8
9  %%
10
11 %public
12 %class Scanner
13 %char
14 %line
15 %column
16 %cup                                                             ❸
17 %implements ScannerIF
18 %scanerror LexicalError
19
20 %{
21   LexicalErrorManager lexicalErrorManager = new LexicalErrorManager ();
22   private int commentCount = 0;
23
24   Token createToken () {                                         ❹
25     Token token = new Token (sym.PLUS);
26     token.setLine (yyline + 1);
27     token.setColumn (yycolumn + 1);
28     token.setLexema (yytext ());
29     return token;
30   }
31 %}
32
33 ALFA  = [A-Za-z]
34 DIGIT = [0-9]
35 SPACE = [\ \t\b\012]
36
37 %%
38
39 <YYINITIAL>     "+"     {  return createToken (MAS);   }          ❺
40 <YYINITIAL>     "-"     {  return createToken (MENOS); }
41 <YYINITIAL>     "*"     {  return createToken (POR);   }
42 <YYINITIAL>     "/"     {  return createToken (DIV);   }
43 ...
44 <YYINITIAL>     .       {  error (); }
45
46 <COMMENT>       "/*"    { ... }
47 <COMMENT>       "*/"    { ... }
```

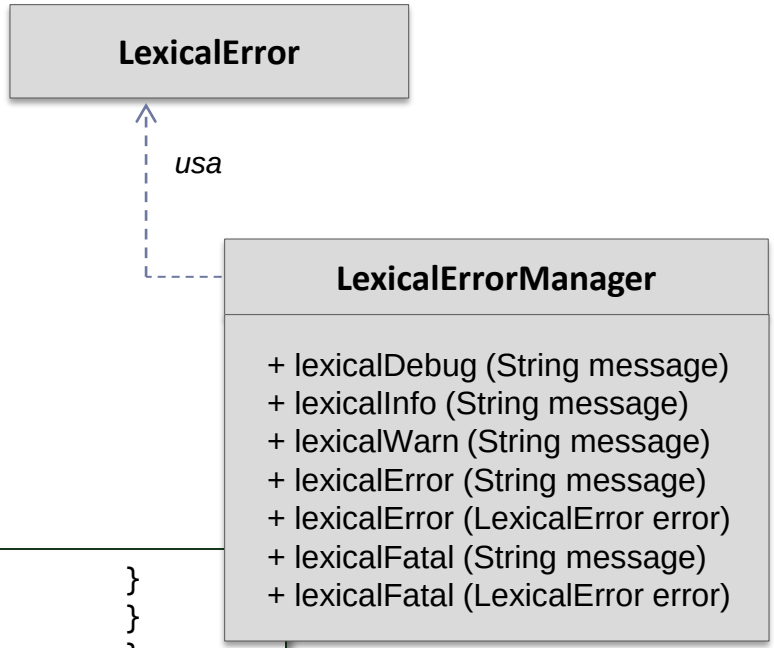
Gestión de errores en JFlex

Consúltese el documento "Directrices de implementación"

La gestión de errores léxicos está soportada dentro del framework por la clase LexicalErrorManager. Los métodos para emitir error son

- error (). Emite un mensaje de error
- fatal (). Emite un mensaje de error y para el scanner

Estos métodos están doblemente sobrecargados. Pueden recibir un mensaje de error (String) o un objeto LexicalError que representa un error léxico



Ejemplo

```
<YYINITIAL> "+"    { return crearToken (MAS);              }
<YYINITIAL> "-"    { return crearToken (MENOS);            }
<YYINITIAL> "*"    { return crearToken (MUL);              }
<YYINITIAL> "/"    { return crearToken (DIV);              }
...
<YYINITIAL> .      { LexicalErrorManager.lexicalFatal ("ufff!"); }
```

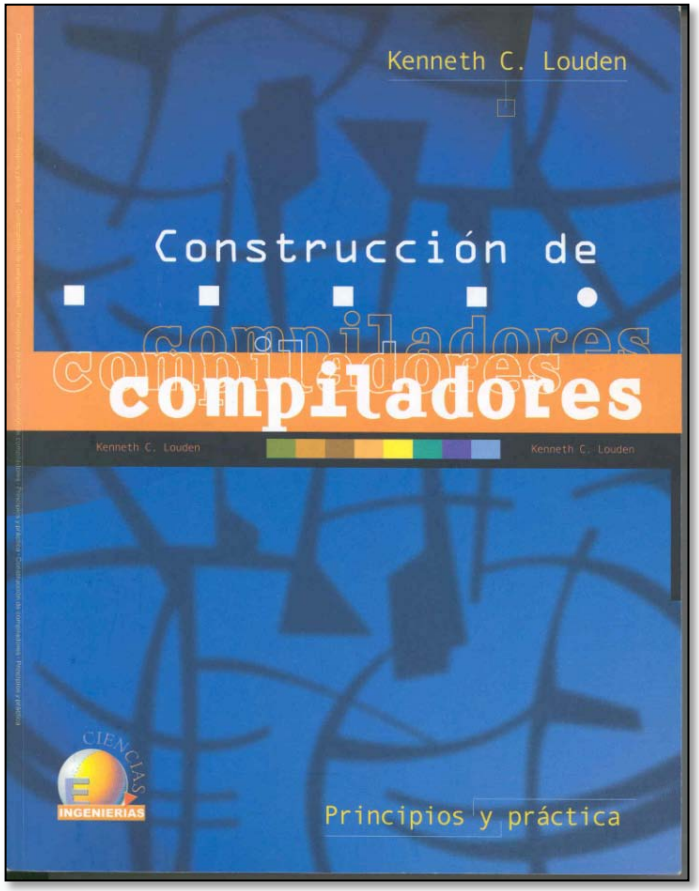
Desarrollo paso a paso

1. Especificación del escáner
   - Abrir el documento de especificaciones JFlex doc/specs/scanner.flex
   - No cambiar nada de lo que está escrito!
   - Declarar macros necesarias (ALFA, DIGIT, SPACE, NEWLINE…)
   - Declarar el estado de comentarios %%state COMMENT
   - Definir las reglas de YYINITIAL
   - Definir las reglas de COMMENT (contemplar el anidamiento si procede)
2. Generar el escáner
   - Limpiar (tarea clear)
   - Generar el escáner (tarea jflex)
   - Compilar el escáner (tarea compile)
3. Probar el escáner
   - Abrir fichero LexicalTestCase src/compiler/test/LexicalTestCase.java
   - Descomentar y comentar lo indicado en el fichero (léelo atentamente)
   - Si funciona, fin. Sino volver a 1

Material de estudio

Bibliografía básica

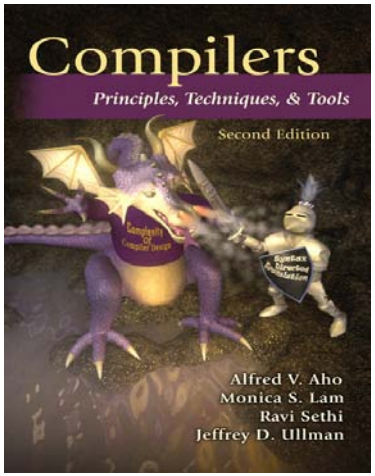
Construcción de compiladores: principios y práctica
Kenneth C. Louden International Thomson Editores, 2004 ISBN 970-686-299-4



Material de estudio

Bibliografía complementaria

Compiladores: Principios, técnicas y herramientas.
Segunda Edición Aho, Lam, Sethi, Ullman
Addison – Wesley, Pearson Educación, México 2008



Diseño de compiladores. A. Garrido, J. Iñesta, F. Moreno y J. Pérez. 2002. Edita Universidad de Alicante