Compiladores 24-2
Análisis Léxico

Lourdes del Carmen González Huesca
luglzhuesca@ciencias.unam.mx

Facultad de Ciencias, UNAM

7 febrero 2024

Análisis léxico

- ✓ primera fase de un compilador
- ✓ incluye un escaneo del código fuente para identificar cadenas sintácticamente correctas
- ✓ clasifica **tokens** generando una secuencia de símbolos para ser procesada por la siguiente fase
- ✓ detecta posibles errores de escritura o errores léxicos
- ✓ interacción con la Tabla de Símbolos para almacenar los valores de los tokens

palabras / cadenas

estructura que vive sólo en la compilación

& posición en el código
tipo
valor

Análisis léxico

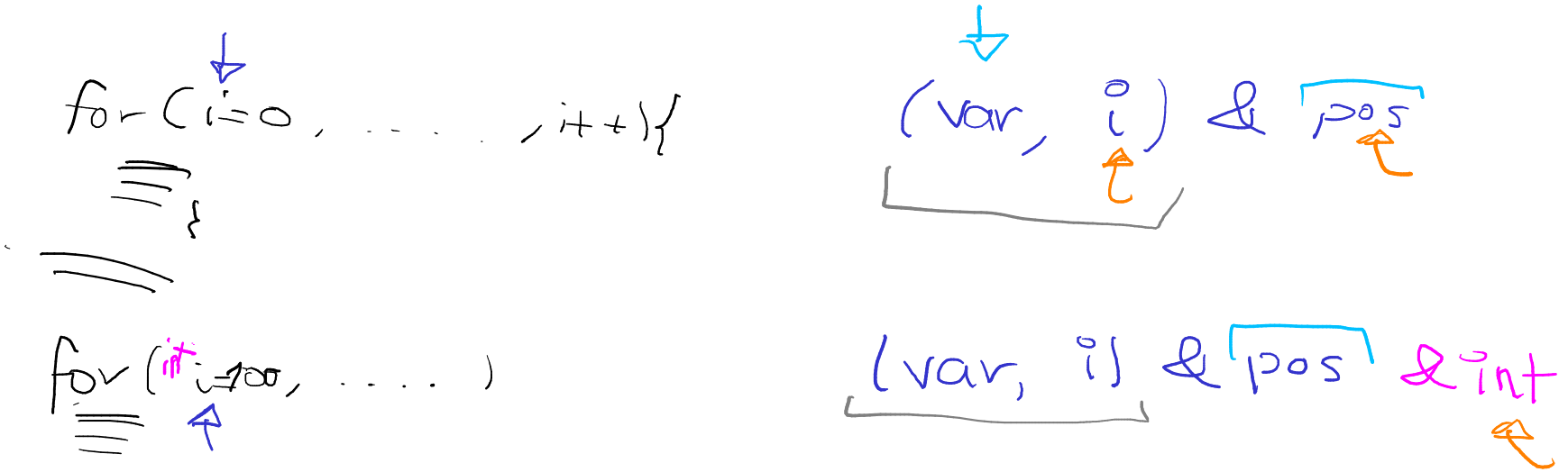
- ✓ primera fase de un compilador
- ✓ incluye un escaneo del código fuente para identificar cadenas sintácticamente correctas
- ✓ clasifica **tokens** generando una secuencia de símbolos para ser procesada por la siguiente fase
- ✓ detecta posibles errores de escritura o errores léxicos
- ✓ interacción con la Tabla de Símbolos para almacenar los valores de los tokens

La entrada es un código fuente que será transformado en una secuencia de tokens para recuperar la información relevante del código, esta representación es útil para análisis posteriores.

Tokens

Un token es una palabra significativa en el lenguaje de programación

- Consta de dos partes, el nombre y el valor (opcional) del atributo.
- El nombre es la representación abstracta del tipo de unidad léxica.
- El atributo es el valor de dicha unidad (cada token tiene a lo más un atributo).



Tokens

Un token es una palabra significativa en el lenguaje de programación

- Consta de dos partes, el nombre y el valor (opcional) del atributo.
- El nombre es la representación abstracta del tipo de unidad léxica.
- El atributo es el valor de dicha unidad (cada token tiene a lo más un atributo).
- El **lexema** son los caracteres del programa fuente que son instancia del token.
- Los tokens (sólo los nombres) serán los símbolos de entrada para el análisis sintáctico.

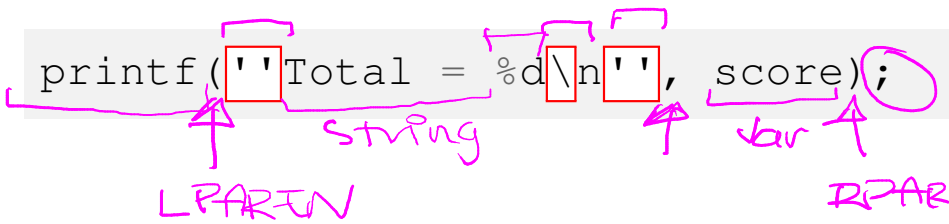
Tokens

Un token es una palabra significativa en el lenguaje de programación

- Consta de dos partes, el nombre y el valor (opcional) del atributo.
- El nombre es la representación abstracta del tipo de unidad léxica.
- El atributo es el valor de dicha unidad (cada token tiene a lo más un atributo).
- El **lexema** son los caracteres del programa fuente que son instancia del token.
- Los tokens (sólo los nombres) serán los símbolos de entrada para el análisis sintáctico.

token	descripción informal	lexema	atributo
if	caracteres i f	`if`	–
else	caracteres e l s e	`else`	–
id	una letra seguida de letras o dígitos	`myvariable`	entrada en la tabla
num	cualquier valor numérico	`4.235`	entrada en la tabla
relop	cualquier operador de comparación	`<=`	`LE`

```
printf(''Total = %d\n'', score);
```



Analizador léxico

i=5+j

i _ = _ 5 _ + _ j

Para obtener un mejor diseño en el analizador, opcionalemente se puede dividir en dos procesos:

1. ***Scanner***: proceso que elimina los comentarios y los espacios en blanco.
2. ***Lexer***: proceso para producir la secuencia de tokens.

se almacenan en la Tabla de símbolos

for(i=0, ...) — — — —

init

forvar=0

Autómatas para Leng. Regulares [mecanismo para identificar tokens

Scanner

Espacios

- Los espacios en blanco, los saltos de línea y demás espacios para sangría (en inglés *indentation*) separan lexemas pero no son relevantes.
- Son convenciones de alto nivel, particulares a cada lenguaje de programación.
- Ayudan a identificar tokens y estructuras de control.

Comentarios

- Los comentarios son tratados como espacios en blanco.
- Sirven para la generación de documentación del código fuente pero no representan información significativa para el compilador.

Analizador léxico

lexer

El analizador léxico debe identificar espacios y comentarios a través del *scanner*.
El *lexer* identificará a los valores que representan las unidades léxicas de un programa (palabras reservadas del lenguaje, identificadores, etc.,) por medio de *tokens*.

Analizador léxico

lexer

El analizador léxico debe identificar espacios y comentarios a través del *scanner*.
El *lexer* identificará a los valores que representan las unidades léxicas de un programa (palabras reservadas del lenguaje, identificadores, etc.,) por medio de *tokens*.

¿cómo hacerlo de la mejor manera?

Analizador léxico

lexer

El analizador léxico debe identificar espacios y comentarios a través del *scanner*.
El *lexer* identificará a los valores que representan las unidades léxicas de un programa (palabras reservadas del lenguaje, identificadores, etc.,) por medio de *tokens*.

¿cómo hacerlo de la mejor manera?

- Ad-hoc lexers: escritos a mano...

Analizador léxico

lexer

El analizador léxico debe identificar espacios y comentarios a través del *scanner*.
El *lexer* identificará a los valores que representan las unidades léxicas de un programa (palabras reservadas del lenguaje, identificadores, etc.,) por medio de *tokens*.

¿cómo hacerlo de la mejor manera?

- Ad-hoc lexers: escritos a mano...
  - ✓ utilizar la técnica de *look-ahead character* para determinar el tipo de token que se está procesando o un posible espacio;

Analizador léxico

lexer

El analizador léxico debe identificar espacios y comentarios a través del *scanner*.
El *lexer* identificará a los valores que representan las unidades léxicas de un programa (palabras reservadas del lenguaje, identificadores, etc.,) por medio de *tokens*.

¿cómo hacerlo de la mejor manera?

- Ad-hoc lexers: escritos a mano...
  - ✓ utilizar la técnica de *look-ahead character* para determinar el tipo de token que se está procesando o un posible espacio;
  - ✗ difícil anticipar caracteres para identificar comentarios y palabras reservadas
  - ✗ el diseño no está estandarizado y es difícil darle mantenimiento

Analizador léxico

lexer

El analizador léxico debe identificar espacios y comentarios a través del *scanner*.
El *lexer* identificará a los valores que representan las unidades léxicas de un programa (palabras reservadas del lenguaje, identificadores, etc.,) por medio de *tokens*.

¿cómo hacerlo de la mejor manera?

- Ad-hoc lexers: escritos a mano...
  - ✓ utilizar la técnica de *look-ahead character* para determinar el tipo de token que se está procesando o un posible espacio;
  - ✗ difícil anticipar caracteres para identificar comentarios y palabras reservadas
  - ✗ el diseño no está estandarizado y es difícil darle mantenimiento
- Generadores automáticos: automatizan la obtención de un analizador léxico a partir de un conjunto de expresiones regulares

especificación

Analizador léxico

lexer

El analizador léxico debe identificar espacios y comentarios a través del *scanner*.
El *lexer* identificará a los valores que representan las unidades léxicas de un programa (palabras reservadas del lenguaje, identificadores, etc.,) por medio de *tokens*.

¿cómo hacerlo de la mejor manera?

- Ad-hoc lexers: escritos a mano...
  - ✓ utilizar la técnica de *look-ahead character* para determinar el tipo de token que se está procesando o un posible espacio;
  - ✗ difícil anticipar caracteres para identificar comentarios y palabras reservadas
  - ✗ el diseño no está estandarizado y es difícil darle mantenimiento
- Generadores automáticos: automatizan la obtención de un analizador léxico a partir de un conjunto de expresiones regulares
  - ✓ construir un autómata con la especificación dada

Analizador léxico

lexer

Un analizador léxico debe procesar una cadena de símbolos y devolver los tokens reconocidos:

if	(	b	==	0	)	{	a	=	0	;	}

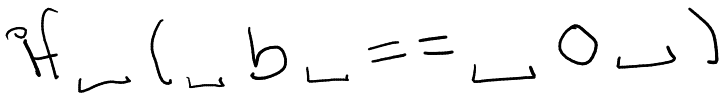
Analizador léxico

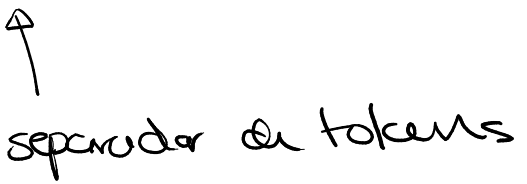
lexer

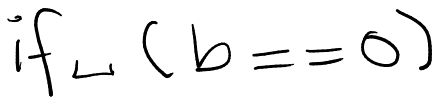
Un analizador léxico debe procesar una cadena de símbolos y devolver los tokens reconocidos:

if	(	b	==	0	)	{	a	=	0	;	}

```
IF; LPAREN; Ident("b"); RELOP(EQEQ); Int(0); RPAREN; LBRAC;
        Ident("a"); EQ; Int(0); SEMI; RBRACE
```







Analizador léxico

lexer

Un analizador léxico debe procesar una cadena de símbolos y devolver los tokens reconocidos:

if	(	b	==	0	)	{	a	=	0	;	}

```
IF; LPAREN; Ident("b"); RELOP(EQEQ); Int(0); RPAREN; LBRAC;
Ident("a"); EQ; Int(0); SEMI; RBRACE
```

Para su implementación:

1. especificar los tipos de tokens a ser reconocidos usando expresiones regulares para cada uno;
2. convertir las expresiones regulares en autómatas finitos, éstos pueden ser no deterministas incluso con transiciones épsilon pero es deseable que sean deterministas;
3. unir los autómtas en uno más grande que tenga la opción de escoger entre los diferentes autómatas de las expresiones regulares;
4. transformar el autómata anterior en uno determinista y opcionalmente minimizarlo;
5. implementar el autómata respetando la regla de coincidencia más extensa posible al almacenar el índice del último token reconocido y el índice de mayor alcance en la cadena de entrada, regresando error si no es posible reconocer alguna palabra.

Especificación para generación automática de un lexer

- ✓ Usar formalismos como Teoría de Cadenas y Lenguajes y Autómatas Finitos para describir el procesamiento de cadenas (un programa fuente tratado como cadena).
- ✓ La especificación establece las cadenas que se aceptarán y el tipo de tokens asociados mediante expresiones regulares.

Especificación para generación automática de un lexer

- ✓ Usar formalismos como Teoría de Cadenas y Lenguajes y Autómatas Finitos para describir el procesamiento de cadenas (un programa fuente tratado como cadena).
- ✓ La especificación establece las cadenas que se aceptarán y el tipo de tokens asociados mediante expresiones regulares.

Input: especificación de tokens del lenguaje de programación
lista de expresiones regulares en orden de prioridad y una acción asociada a cada expresión (generar token, guardar info en la tabla de símbolos, etc.)

Especificación para generación automática de un lexer

- ✓ Usar formalismos como Teoría de Cadenas y Lenguajes y Autómatas Finitos para describir el procesamiento de cadenas (un programa fuente tratado como cadena).
- ✓ La especificación establece las cadenas que se aceptarán y el tipo de tokens asociados mediante expresiones regulares.

Input: especificación de tokens
lista de expresiones regulares en orden de prioridad y una acción asociada a cada expresión (generar token, guardar info en la tabla de símbolos, etc.)

Output: lexer
un programa que lee un *stream* y lo disecta en tokens de acuerdo a la especificación de las expresiones regulares

Especificación para generación automática de un lexer

- ✓ Usar formalismos como Teoría de Cadenas y Lenguajes y Autómatas Finitos para describir el procesamiento de cadenas (un programa fuente tratado como cadena).
- ✓ La especificación establece las cadenas que se aceptarán y el tipo de tokens asociados mediante expresiones regulares.

Input: especificación de tokens
lista de expresiones regulares en orden de prioridad y una acción asociada a cada expresión (generar token, guardar info en la tabla de símbolos, etc.)

Output: lexer
un programa que lee un *stream* y lo disecta en tokens de acuerdo a la especificación de las expresiones regulares

Posible efecto secundario Reportar errores léxicos. del lexer

escritura
sangría/indentación
salto de línea

Especificación para generación automática de un lexer

- ✓ Usar formalismos como Teoría de Cadenas y Lenguajes y Autómatas Finitos para describir el procesamiento de cadenas (un programa fuente tratado como cadena).
- ✓ La especificación establece las cadenas que se aceptarán y el tipo de tokens asociados mediante expresiones regulares.

Input: especificación de tokens
lista de expresiones regulares en orden de prioridad y una acción asociada a cada expresión (generar token, guardar info en la tabla de símbolos, etc.)

Output: lexer
un programa que lee un *stream* y lo disecta en tokens de acuerdo a la especificación de las expresiones regulares

stream tokens

Posible efecto secundario Reportar errores léxicos.

Este programa es la implementación de un autómata para reconocer tokens.

Especificación para generación automática de un lexer

- ★ Reciben una lista de expresiones regulares $R_1, R_2, \ldots, R_n$ que definen cada token del lenguaje fuente
- ★ Junto con una acción correspondiente al token, es decir un código para ser ejecutado cuando la expresión regular coincide.

Especificación para generación automática de un lexer

- ★ Reciben una lista de expresiones regulares $R_1, R_2, \ldots, R_n$ que definen cada token del lenguaje fuente
- ★ Junto con una acción correspondiente al token, es decir un código para ser ejecutado cuando la expresión regular coincide.
- ★ Al final, genera un código escaneado donde se decide si una cadena de entrada es de la forma $(R_1 \mid R_2 \mid \cdots \mid R_n)^\star$

Especificación para generación automática de un lexer

- ★ Reciben una lista de expresiones regulares $R_1, R_2, \ldots, R_n$ que definen cada token del lenguaje fuente
- ★ Junto con una acción correspondiente al token, es decir un código para ser ejecutado cuando la expresión regular coincide.
- ★ Al final, genera un código escaneado donde se decide si una cadena de entrada es de la forma $(R_1 \mid R_2 \mid \cdots \mid R_n)^\star$
- ★ Cada vez que se **reconoce** el token o la coincidencia de la cadena más larga de algún token, se ejecuta la acción asociada.

  Regla de coincidencia más extensa posible: dar prioridad a las transiciones posibles, si no se llega a un estado final se reporta error léxico.

Especificación para generación automática de un lexer

- ★ Reciben una lista de expresiones regulares $R_1, R_2, \ldots, R_n$ que definen cada token del lenguaje fuente
- ★ Junto con una acción correspondiente al token, es decir un código para ser ejecutado cuando la expresión regular coincide.
- ★ Al final, genera un código escaneado donde se decide si una cadena de entrada es de la forma $(R_1 \mid R_2 \mid \cdots \mid R_n)^\star$

  Ri es una categoría sintáctica definida
- ★ Cada vez que se **reconoce** el token o la coincidencia de la cadena más larga de algún token, se ejecuta la acción asociada.

  Regla de coincidencia más extensa posible: dar prioridad a las transiciones posibles, si no se llega a un estado final se reporta error léxico.

1. considerar cada expresión regular R_i y su acción asociada A_i;
2. calcular un autómata finito no-determinista para $(R_1 \mid R_2 \mid \cdots \mid R_n)^\star$
3. transformar el autómata anterior en uno determinista y minimizarlo;
4. producir la tabla de transiciones que define al autómata.

Analizador léxico

lexer para operadores de comparación

Categorías sintácticas

definidas en la especificación de un leng. programación

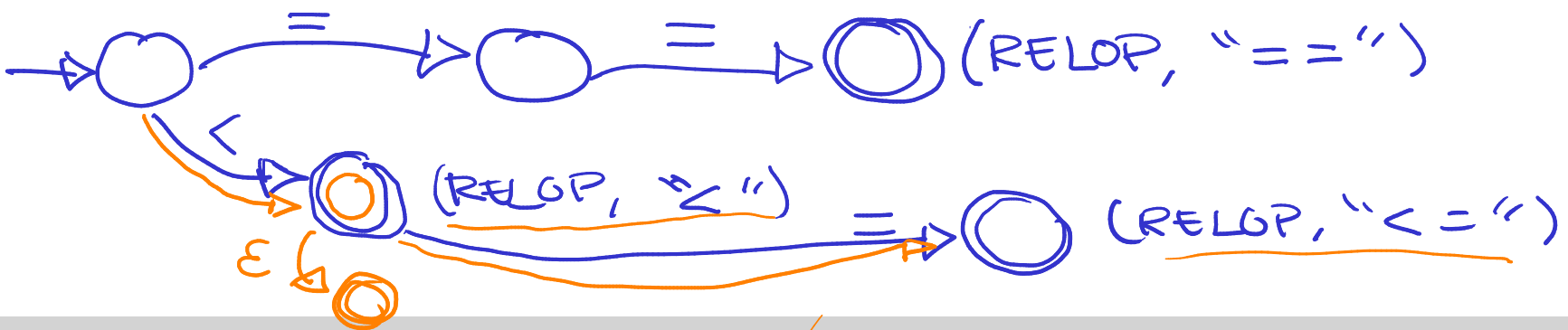
- palabras reservadas
- signos de puntuación
- operadores de algunos tipos
- nombres de variables / funciones
- valores específicos (Int, Bool, Char, String ...)

Operadores de comparación (para números)

< <= == >= > !=

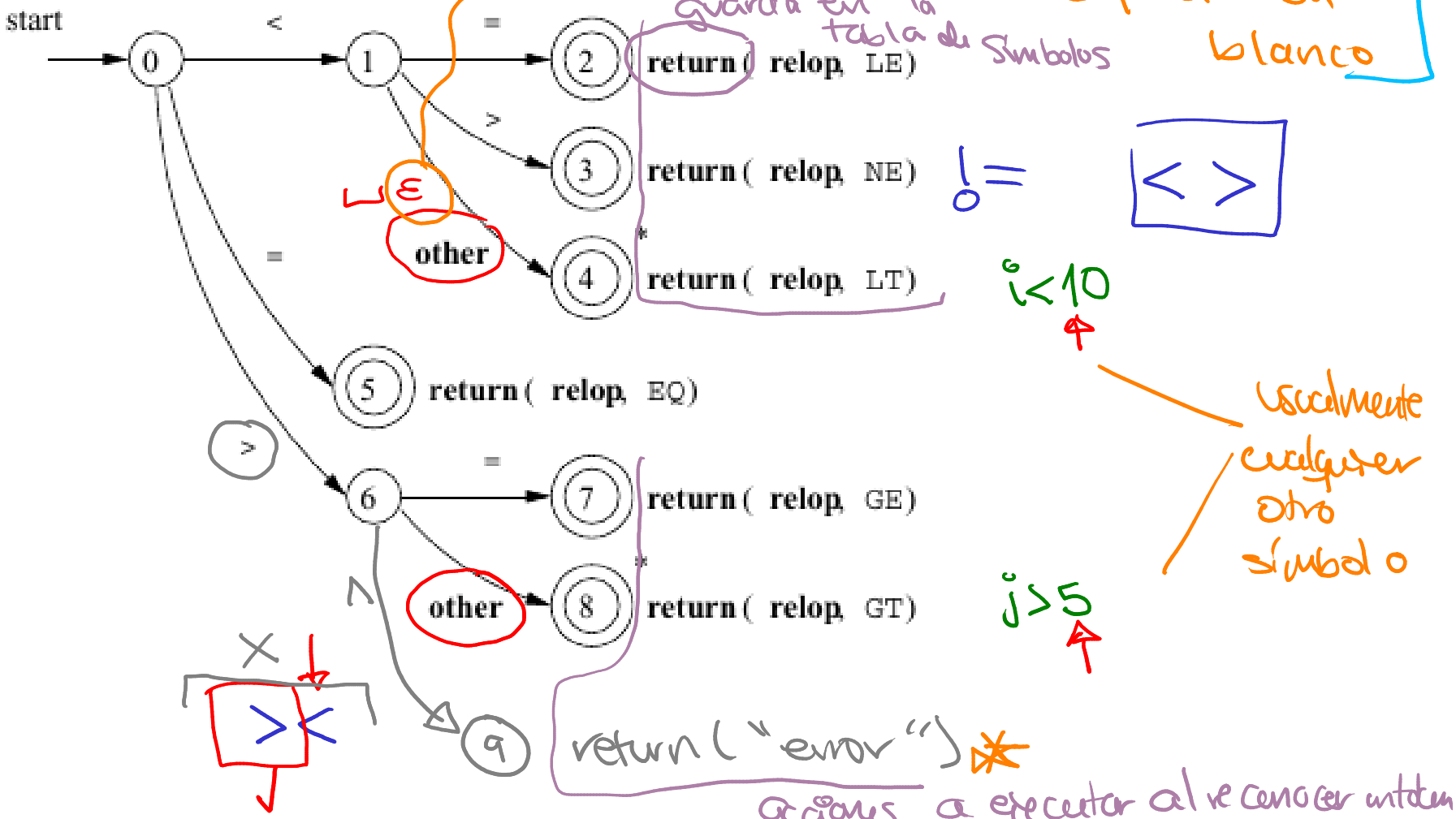
Tipo de token: RELOP

valores (atributo)



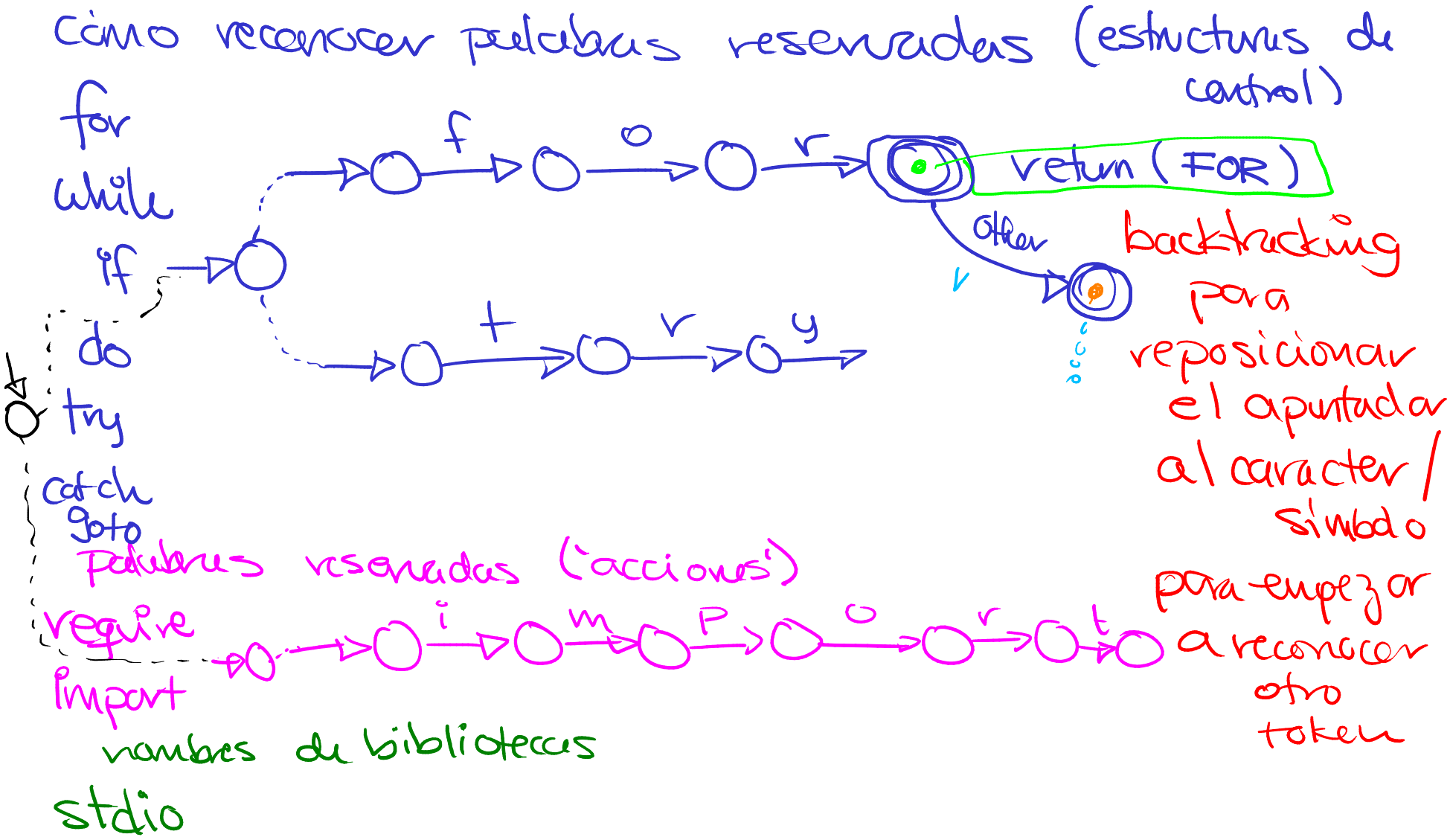
Analizador léxico

lexer para operadores de comparación



Analizador léxico

lexer



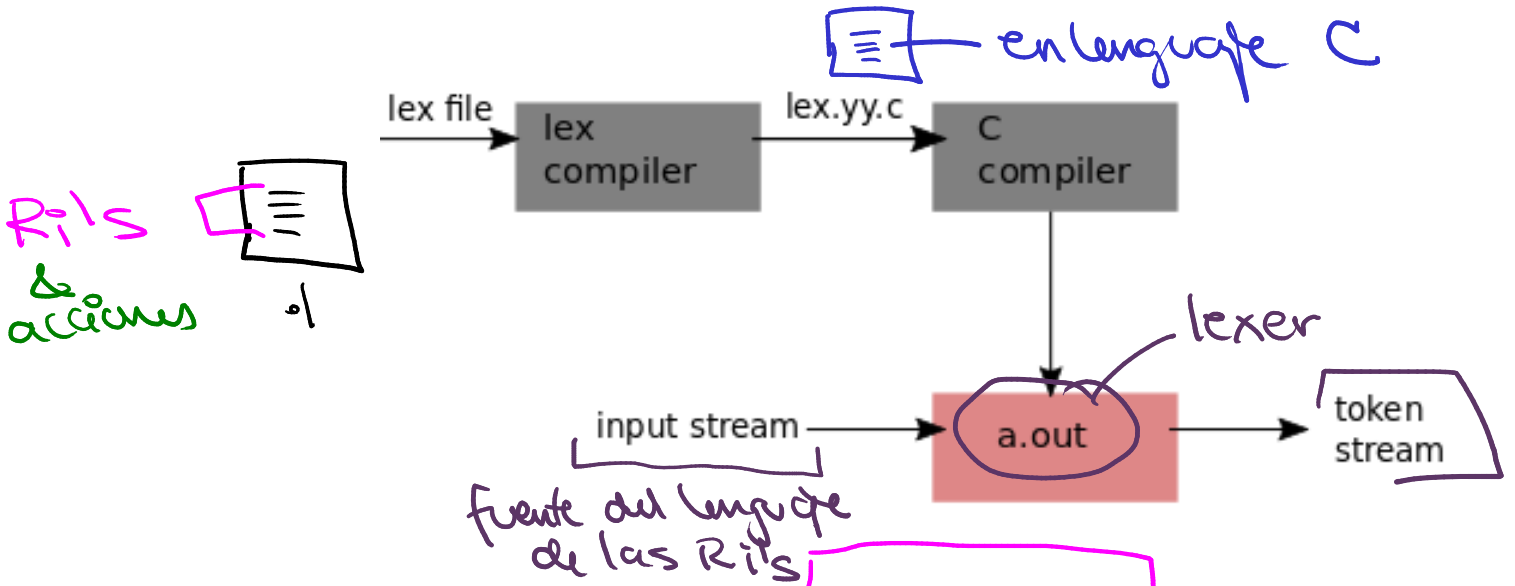
Generadores léxicos

- `Lex`: generador léxico para Unix que obtiene un lexer en C.
  https://en.wikipedia.org/wiki/Lex_(software)
  http://dinosaur.compilertools.net/lex/index.html https://cse.iitkgp.ac.in/~bivasm/notes/LexAndYaccTutorial.pdf
- `Flex`: *Fast LEXical analyzer generator*
  https://www.cs.princeton.edu/~appel/modern/c/software/flex/flex.html#SEC1
- `ocamllex`: generador léxico en Ocaml.
  https://courses.softlab.ntua.gr/compilers/2015a/ocamllex-tutorial.pdf
  https://caml.inria.fr/pub/docs/manual-ocaml/lexyacc.html
- `JLex`: generador para Java escrito en Java.
  https://www.cs.princeton.edu/~appel/modern/java/JLex/
- Python Lex-Yacc
  https://www.dabeaz.com/ply/

Usualmente van de la mano con el analizador sintáctico correspondiente.

Generadores léxicos

Lex / Flex (Unix)



- Lista de expresiones regulares $R_1, R_2, \ldots, R_n$ que definen cada token del lenguaje fuente junto con una acción correspondiente al token.
- Calcular un AFN, transformarlo en un AFD y minimizarlo .
- Producir la tabla de transiciones del autómata.
- Usar la técnica de coincidencia más larga:
  - seguir transiciones y recordar el último estado de aceptación;
  - procesar la cadena hasta que no sea posible realizar una transición;
  - realizar la acción asociada al estado de aceptación cuando se llega a alguno, sino enviar un error.

return/ guardar el token

$\delta : Q \times \Sigma \to Q \times A$

Ejemplo: Flex

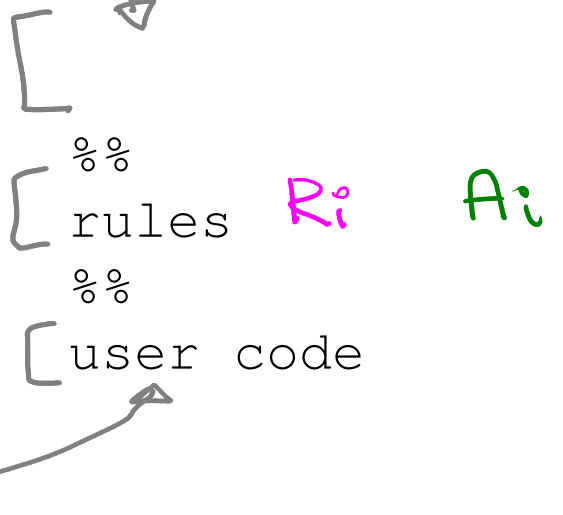
- ★ `lex` fue creado en 1975 por M. Lesk y E. Schmidt como un estándar para analizar léxicamente expresiones en Unix.
- ★ Usaremos `flex` (*Fast LEXical analyzer generator*) creado en 1987.

Ejemplo: Flex

- ★ `lex` fue creado en 1975 por M. Lesk y E. Schmidt como un estándar para analizar léxicamente expresiones en Unix.
- ★ Usaremos `flex` (*Fast LEXical analyzer generator*) creado en 1987.

Tiene la siguiente estructura:

- **definiciones**: incluyen las bibliotecas a usarse en el código del usuario o en las acciones
- **reglas**: expresiones regulares junto con la acción correspondiente a cada una de ellas
- **código**: funciones auxiliares



Ejemplo: Flex

lenguaje `lex`

Regla = Ri ⇒ Ai
expreg acción

Lenguaje de especificación para analizadores léxicos que define esquemas con expresiones regulares y acciones.

Regex

`ab`	concatenación ✓ Ri
`a | b`	disyunción
$a^\star$	cerradura de Kleene ✓
`a{k}`	repetición *k* veces ✓
`a?`	opción
a^+	cerradura positiva ✓
`.`	cualquier caracter excepto `\n`
`[]`	conjunto de caracteres
`[^]`	complemento ✓
`\`	escape
`".."`	literalmente ese patrón
`^`	inicio de línea
`$`	fin de línea
`a/b`	reconoce *a* si es seguido de *b*

acciones

- Las acciones están en código C que será usado por el autómata generado, es decir pueden describir estados.
- Se aplican las reglas en orden de aparición en el archivo.
- Si hay dos o más reglas se aplica la primera.
- Los caracteres no reconocidos se almacenan en la salida estándar.

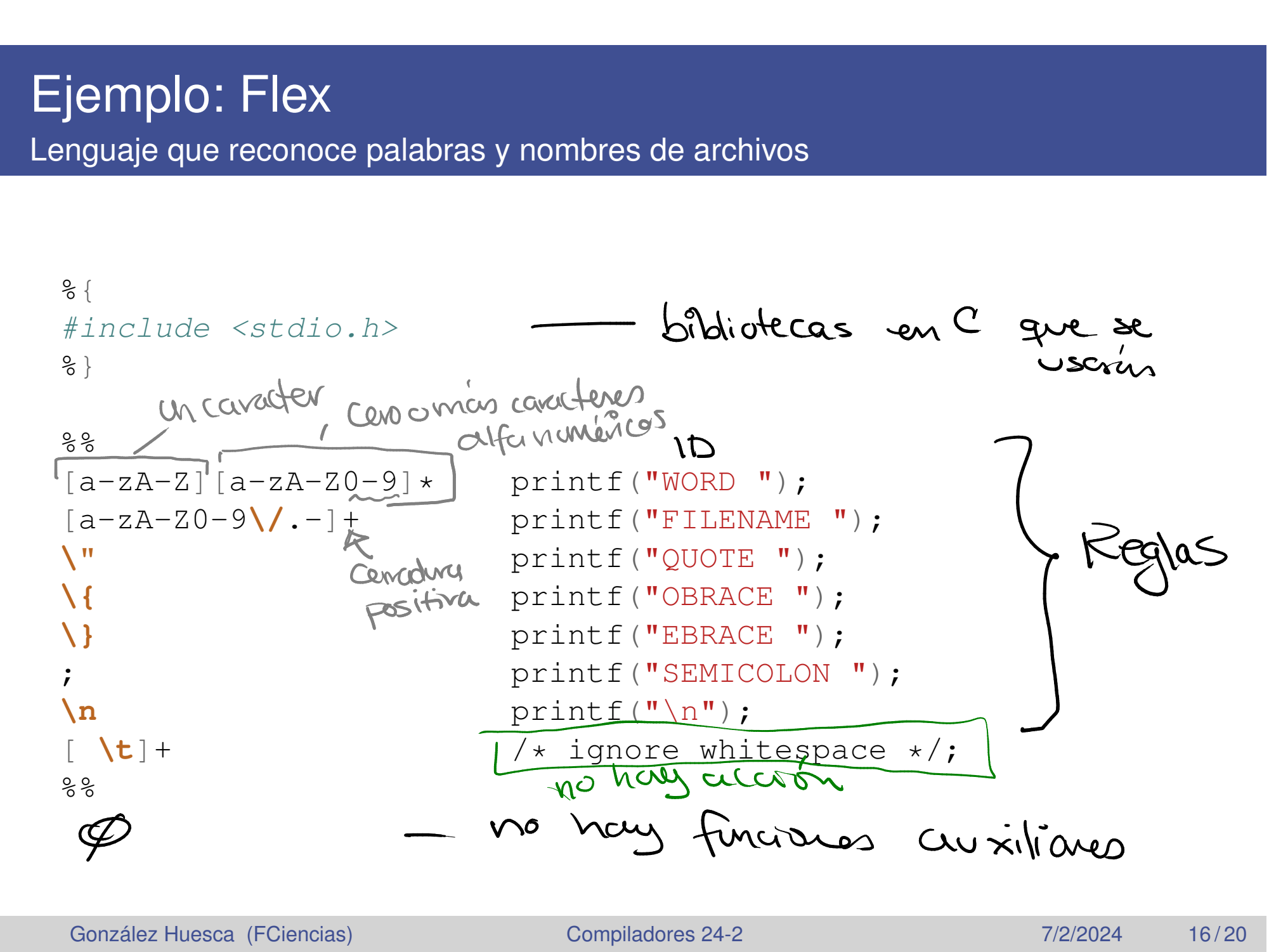
siempre se imprime en consola.

Ejemplo: Flex

Lenguaje que reconoce palabras y nombres de archivos

```
%{
#include <stdio.h>
%}

%%
[a-zA-Z][a-zA-Z0-9]*        printf("WORD ");
[a-zA-Z0-9\/.-]+            printf("FILENAME ");
\"                          printf("QUOTE ");
\{                          printf("OBRACE ");
\}                          printf("EBRACE ");
;                           printf("SEMICOLON ");
\n                          printf("\n");
[ \t]+                      /* ignore whitespace */;
%%
```



Ejemplo: Flex

manejo de calefacción

```
%{
#include <stdio.h>
#include "y.tab.h"
%}
%%
[0-9]+                  yylval=atoi(yytext); return NUMBER;
heat                    return TOKHEAT;
on|off                  yylval=!strcmp(yytext,"on"); return STAT
target                  return TOKTARGET;
temperature             return TOKTEMPERATURE;
\n                      /* ignore end of line */;
[ \t]+                  /* ignore whitespace */;
%%
```

Ejemplo tomado de https://tldp.org/HOWTO/Lex-YACC-HOWTO-4.html

Ejemplo: Flex

- Se tiene un programa en C que define la función `yylex(void)` que implementa el analizador léxico.
- `yylex` hace uso de las acciones definidas.
- Cada que se reconoce una unidad léxica, la cadena reconocida es un valor en `yytext` y la variable `yylen` indica la logitud de la unidad.
- La función `yywrap(void)` es el proceso de decisión:
  - devuelve 1 si el programa se detiene al final del flujo de entrada
  - devuelve 0 si se continua procesando una entrada

Errores léxicos

- Originados cuando el caracter de entrada al lexer no pede ser aceptado como parte de un token que se está procesando y tampoco como un nuevo token.

Errores léxicos

- Originados cuando el caracter de entrada al lexer no pede ser aceptado como parte de un token que se está procesando y tampoco como un nuevo token.
- Ejemplos de errores léxicos: errores de escritura de palabras reservadas, operadores o identificadores; uso incorrecto de comillas simples o dobles.

Errores léxicos

- Originados cuando el caracter de entrada al lexer no pede ser aceptado como parte de un token que se está procesando y tampoco como un nuevo token.
- Ejemplos de errores léxicos: errores de escritura de palabras reservadas, operadores o identificadores; uso incorrecto de comillas simples o dobles.
- Errores que no son muy usuales, por lo general es algún caracter fuera de lugar.

Errores léxicos

- Originados cuando el caracter de entrada al lexer no pede ser aceptado como parte de un token que se está procesando y tampoco como un nuevo token.
- Ejemplos de errores léxicos: errores de escritura de palabras reservadas, operadores o identificadores; uso incorrecto de comillas simples o dobles.
- Errores que no son muy usuales, por lo general es algún caracter fuera de lugar.
- Manejo de errores:
  1. ignorar el token con error o inválido
  2. recuperar el stream de entrada en el inicio de lo que sería el siguiente token a analizar
  3. reiniciar el proceso de análisis
  4. utilizar un mecanismo de recuperación de errores para devolver información útil al usuario.

Referencias

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman.
Compilers, Principles, Techniques and Tools.
Pearson Education Inc., Second edition, 2007.

[2] H. R. Nielson and F. Nielson.
Semantics with Applications: An Appetizer (Undergraduate Topics in Computer Science).
Springer-Verlag, Berlin, Heidelberg, 2007.

[3] F. Pfenning.
Notas del curso (15-411) Compiler Design.
https://www.cs.cmu.edu/~fp/courses/15411-f14/, 2014.

[4] M. L. Scott.
Programming Language Pragmatics.
Morgan-Kauffman Publishers, Third edition, 2009.

[5] Y. Su and S. Y. Yan.
Principles of Compilers, A New Approach to Compilers Including the Algebraic Method.
Springer-Verlag, Berlin Heidelberg, 2011.

[6] T. Teitelbaum.
Introduction to compilers.
http://www.cs.cornell.edu/courses/cs412/2008sp/, 2008.

[7] L. Torczon and K. Cooper.
Engineering A Compiler.
Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2011.

[8] S. Zdancewic.
Notas del curso (CIS 341) - Compilers, Universidad de Pennsylvania, Estados Unidos.
https://www.cis.upenn.edu/~cis341/current/, 2018.