Principles of Programming Languages

http://www.di.unipi.it/~andrea/Didattica/PLP-14/

Prof. Andrea Corradini
Department of Computer Science, Pisa

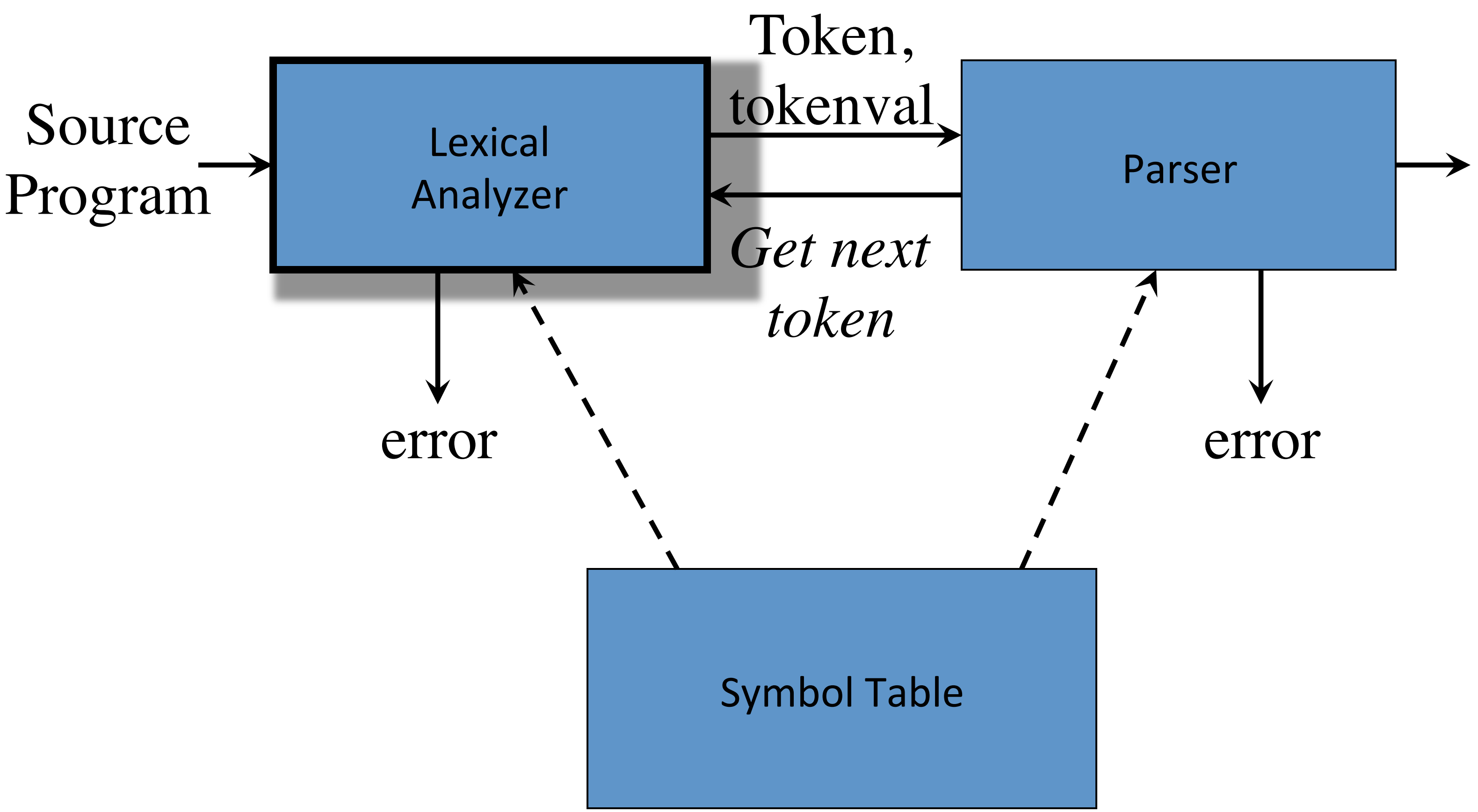
Lesson 4

- Lexical analysis: implementing a scanner

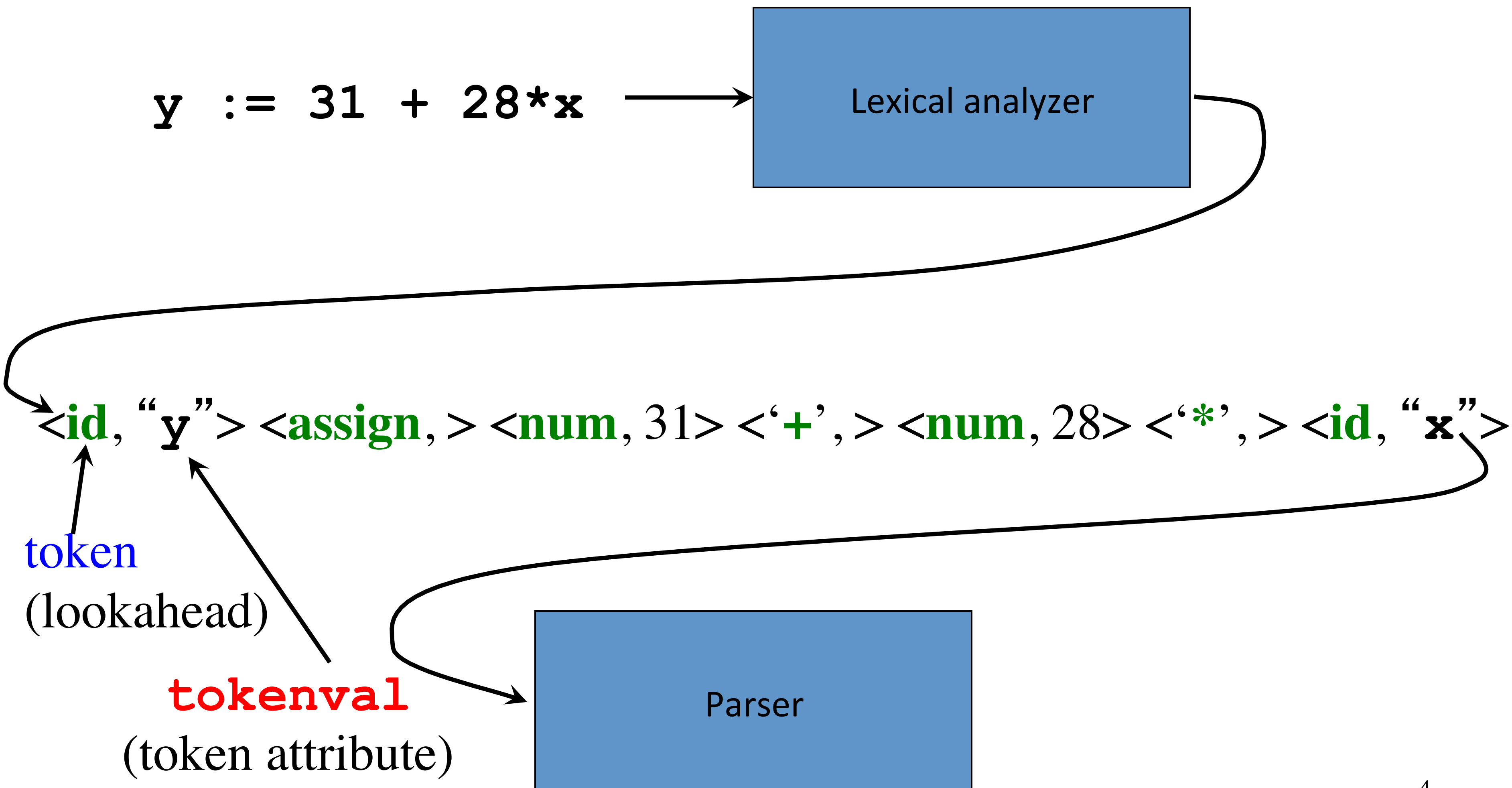
The Reason Why Lexical Analysis is a Separate Phase

- Simplifies the design of the compiler
  - LL(1) or LR(1) parsing with 1 token lookahead would not be possible (multiple characters/tokens to match)
- Provides efficient implementation
  - Systematic techniques to implement lexical analyzers by hand or automatically from specifications
  - Stream buffering methods to scan input
- Improves portability
  - Non-standard symbols and alternate character encodings can be normalized (e.g. UTF8, trigraphs)

Interaction of the Lexical Analyzer with the Parser



Attributes of Tokens



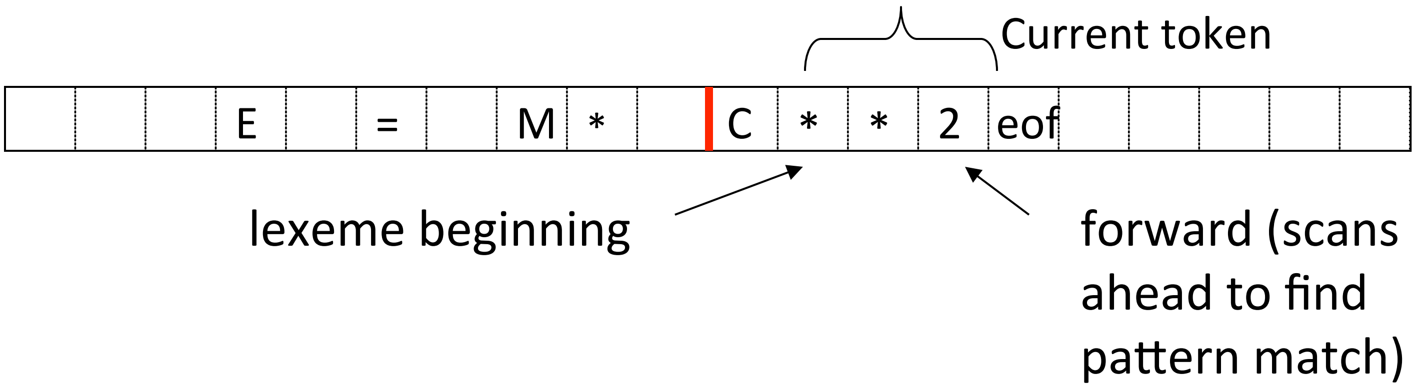
Tokens, Patterns, and Lexemes

- A *token* is a classification of lexical units
  - For example: **id** and **num**
- *Lexemes* are the specific character strings that make up a token
  - For example: **abc** and **123**
- *Patterns* are rules describing the set of lexemes belonging to a token
  - For example: "*letter followed by letters and digits*" and "*non-empty sequence of digits*"
- The scanner reads characters from the input till when it recognizes a lexeme that matches the patterns for a token

Example

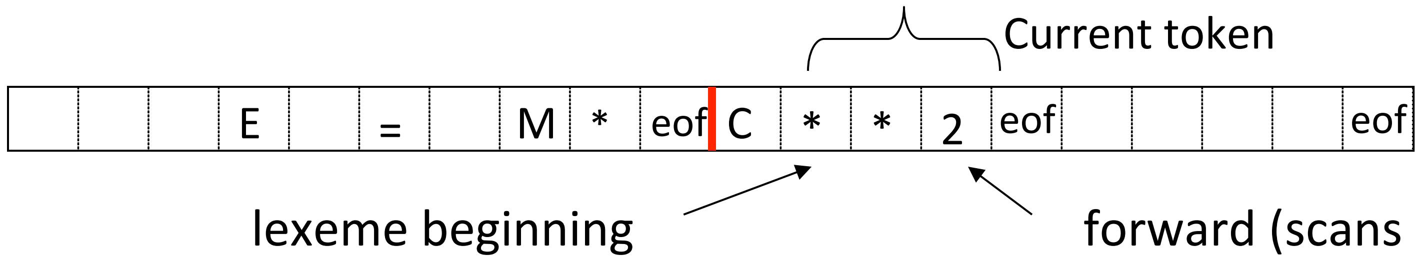
Token	Informal description	Sample lexemes
if	Characters i, f	if
else	Characters e, l, s, e	else
relation	< or > or <= or >= or == or !=	<=, !=
id	Letter followed by letter and digits	pi, score, D2
number	Any numeric constant	3.14159, 0, 6.02e23
literal	Anything but “ sorrounded by “	“core dumped”

Using Buffer to Enhance Efficiency



if *forward* at end of first half then begin
    reload second half ; ← Block I/O
    forward := *forward* + 1
end
else if *forward* at end of second half then begin
    reload first half ; ← Block I/O
    move *forward* to beginning of first half
end
else *forward* := *forward* + 1 ;

Algorithm: Buffered I/O with Sentinels




```
forward : = forward + 1 ;
if forward is at eof then begin
   if forward at end of first half then begin
      reload second half ;  ← Block I/O
      forward : = forward + 1
   end
   else if forward at end of second half then begin
      reload first half ;  ← Block I/O
      move forward to beginning of first half
   end
   else / * eof within buffer signifying end of input * /
      terminate lexical analysis
end      2nd eof ⇒ no more input !
```

Specification of Patterns for Tokens: *Definitions*

- An *alphabet* Σ is a finite set of symbols (characters)
- A *string s* is a finite sequence of symbols from Σ
  - $|s|$ denotes the length of string *s*
  - ε denotes the empty string, thus $|\varepsilon| = 0$
- A *language* is a specific set of strings over some fixed alphabet Σ

Specification of Patterns for Tokens: *String Operations*

- The *concatenation* of two strings ***x*** and ***y*** is denoted by ***xy***
- The *exponentation* of a string s is defined by

$$s^0 = \varepsilon$$
$$s^i = s^{i-1}s \quad \text{for } i > 0$$

note that $s\varepsilon = \varepsilon s = s$

Specification of Patterns for Tokens: *Language Operations*

- *Union*
  $L \cup M = \{s \mid s \in L \text{ or } s \in M\}$
- *Concatenation*
  $LM = \{xy \mid x \in L \text{ and } y \in M\}$
- *Exponentiation*
  $L^0 = \{\varepsilon\}$; $L^i = L^{i-1}L$
- *Kleene closure*
  $L^* = \cup_{i=0,\dots,\infty} L^i$
- *Positive closure*
  $L^+ = \cup_{i=1,\dots,\infty} L^i$

Language Operations: Examples

$L = \{A, B, C, D\}$ $D = \{1, 2, 3\}$

- $L \cup D = \{A, B, C, D, 1, 2, 3\}$
- $LD = \{A1, A2, A3, B1, B2, B3, C1, C2, C3, D1, D2, D3\}$
- $L^2 = \{ AA, AB, AC, AD, BA, BB, BC, BD, CA, \dots DD\}$
- $L^4 = L^2 \; L^2 = ??$
- $L^* = \{$ All possible strings of L plus $\varepsilon \}$
- $L^+ = L^* - \{ \varepsilon \}$
- $L (L \cup D) = ??$
- $L (L \cup D)^* = ??$

Specification of Patterns for Tokens: *Regular Expressions*

- Basis symbols:
  - ε is a regular expression denoting language $\{\varepsilon\}$
  - $a \in \Sigma$ is a regular expression denoting $\{a\}$
- If r and s are regular expressions denoting languages $L(r)$ and $M(s)$ respectively, then
  - $(r) \mid (s)$ is a regular expression denoting $L(r) \cup M(s)$
  - $(r)(s)$ is a regular expression denoting $L(r)M(s)$
  - $(r)^*$ is a regular expression denoting $L(r)^*$
  - (r) is a regular expression denoting $L(r)$
- To avoid too many brackets we impose:
  - Precedence of operators: $(\_)^*$ > $(\_)(\_)$ > $(\_)\mid(\_)$
  - Left-associativity of all operators
- Example: $(a) \mid ((b)^*(c))$ can be written as $a \mid b^*c$

EXAMPLES of Regular Expressions

$L = \{A, B, C, D\}$ $D = \{1, 2, 3\}$

$A \mid B \mid C \mid D = L$

$(A \mid B \mid C \mid D)(A \mid B \mid C \mid D) = L^2$

$(A \mid B \mid C \mid D)^* = L^*$

$(A \mid B \mid C \mid D)((A \mid B \mid C \mid D) \mid (1 \mid 2 \mid 3)) = L(L \cup D)$

- A language defined by a regular expression is called a *regular set*

Algebraic Properties of Regular Expressions

AXIOM	DESCRIPTION
r | s = s | r	| is commutative
r | (s | t) = (r | s) | t	| is associative
(r s) t = r (s t)	concatenation is associative
r (s | t) = r s | r t (s | t) r = s r | t r	concatenation distributes over |
εr = r rε = r	ε Is the identity element for concatenation
r* = (r | ε)*	relation between * and ε
r** = r*	* is idempotent

Specification of Patterns for Tokens: *Regular Definitions*

- Regular definitions introduce a naming convention with name-to-regular-expression bindings:

  $d_1 \rightarrow r_1$
  $d_2 \rightarrow r_2$
  …
  $d_n \rightarrow r_n$

  where each r_i is a regular expression over
  $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$
- Any d_j in r_i can be textually substituted in r_i to obtain an equivalent set of definitions

Specification of Patterns for Tokens: *Regular Definitions*

- Example:

$$\textbf{letter} \to \texttt{A} \mid \texttt{B} \mid \ldots \mid \texttt{Z} \mid \texttt{a} \mid \texttt{b} \mid \ldots \mid \texttt{z}$$
$$\textbf{digit} \to \texttt{0} \mid \texttt{1} \mid \ldots \mid \texttt{9}$$
$$\textbf{id} \to \textbf{letter} \; (\; \textbf{letter} \mid \textbf{digit} \;)^*$$

- Regular definitions cannot be recursive:

$$\textbf{digits} \to \textbf{digit digits} \mid \textbf{digit} \quad \textit{wrong!}$$

Specification of Patterns for Tokens: *Notational Shorthand*

- The following shorthands are often used:

$$r^+ = rr^*$$
$$r? = r \mid \varepsilon$$
$$[\mathtt{a}\text{-}\mathtt{z}] = \mathtt{a} \mid \mathtt{b} \mid \mathtt{c} \mid \ldots \mid \mathtt{z}$$

- Examples:

digit $\to$ [`0-9`]

num $\to$ $\textbf{digit}^+ \; (\textbf{.}\ \textbf{digit}^+)? \; (\ \textbf{E}\ (\textbf{+} \mid \textbf{-})?\ \textbf{digit}^+\ )?$

Context-free Grammars and Tokens

- Given the context-free grammar of a language, *terminal symbols* correspond to the tokens the parser will use.
- Example:
- The tokens are:

`if, then, else, relop, id, num`

$$
\begin{aligned}
\textit{stmt} &\rightarrow \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{stmt} \\
&\mid \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{stmt}\ \textbf{else}\ \textit{stmt} \\
&\mid \varepsilon \\
\textit{expr} &\rightarrow \textit{term}\ \textbf{relop}\ \textit{term} \\
&\mid \textit{term} \\
\textit{term} &\rightarrow \textbf{id} \\
&\mid \textbf{num}
\end{aligned}
$$

Informal specification of tokens and their attributes

Pattern of lexeme	**Token**	**Attribute-Value**
Any ws	-	-
if	if	-
then	then	-
else	else	-
Any id	id	pointer to table entry
Any num	num	pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	EQ
< >	relop	NE
>	relop	GT
>=	relop	GE

Regular Definitions for tokens

- The specification of the patterns for the tokens is provided with regular definitions

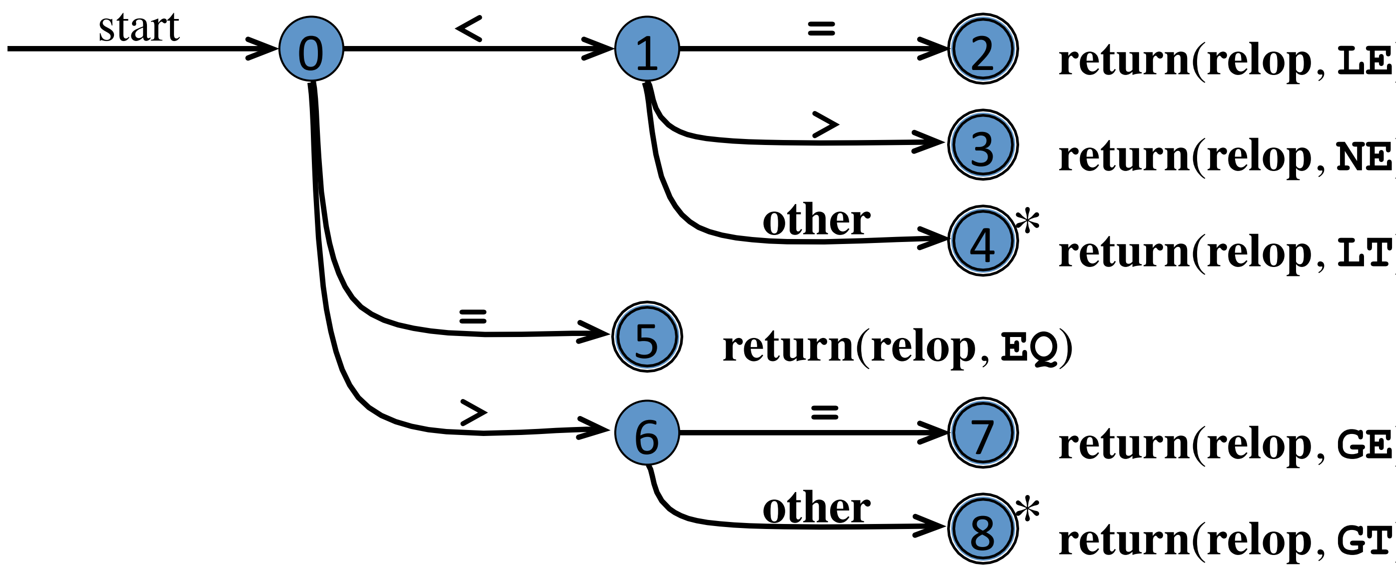
$$
\begin{aligned}
\textbf{if} &\rightarrow \texttt{if} \\
\textbf{then} &\rightarrow \texttt{then} \\
\textbf{else} &\rightarrow \texttt{else} \\
\textbf{relop} &\rightarrow \texttt{<} \mid \texttt{<=} \mid \texttt{<>} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{=} \\
\textbf{id} &\rightarrow \textbf{letter} \; (\, \textbf{letter} \mid \textbf{digit} \,)^{*} \\
\textbf{num} &\rightarrow \textbf{digit}^{+} \; (\textbf{.}\; \textbf{digit}^{+})? \; (\, \textbf{E} \; (\textbf{+} \mid \textbf{-})? \; \textbf{digit}^{+} \,)?
\end{aligned}
$$

From Regular Definitions to code

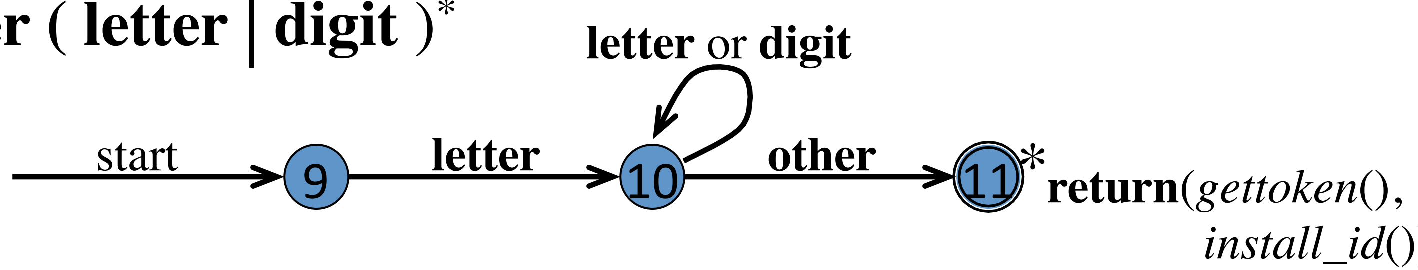
- From the regular definitions we first extract a *transition diagram*, and next the code of the scanner.
- We do this by hand, but it can be automatized.
- In the example the lexemes are recognized either when they are completed, or at the next character. In real situations a longer lookahead might be necessary.
- The diagrams guarantee that the longest lexeme is identified.

Coding Regular Definitions in *Transition Diagrams*

relop → < | <= | <> | > | >= | =



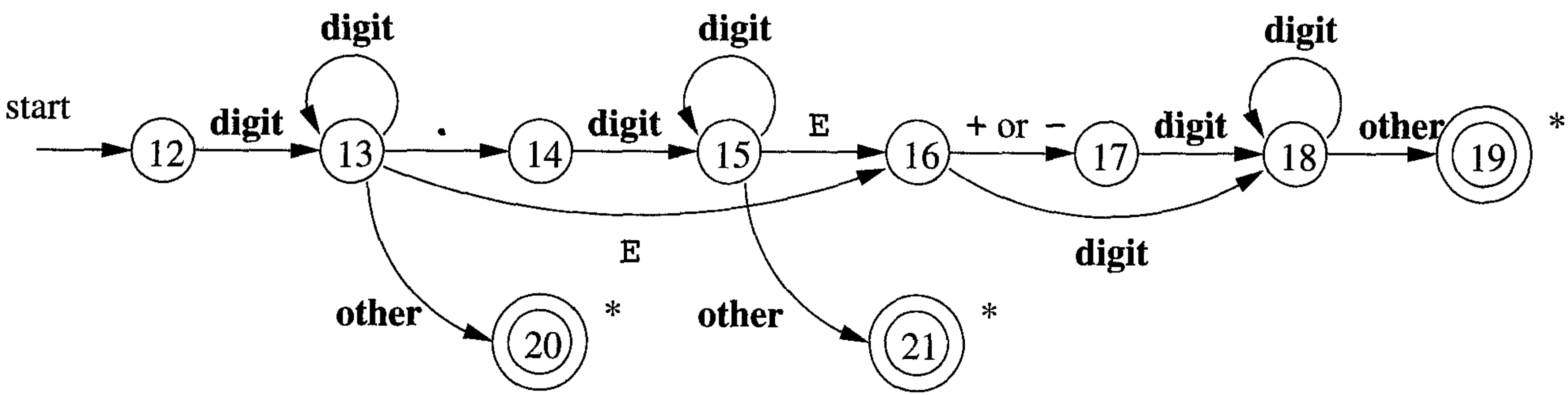
id → **letter** (**letter** | **digit**)*



Coding Regular Definitions in *Transition Diagrams* (cont.)

Transition diagram for unsigned numbers

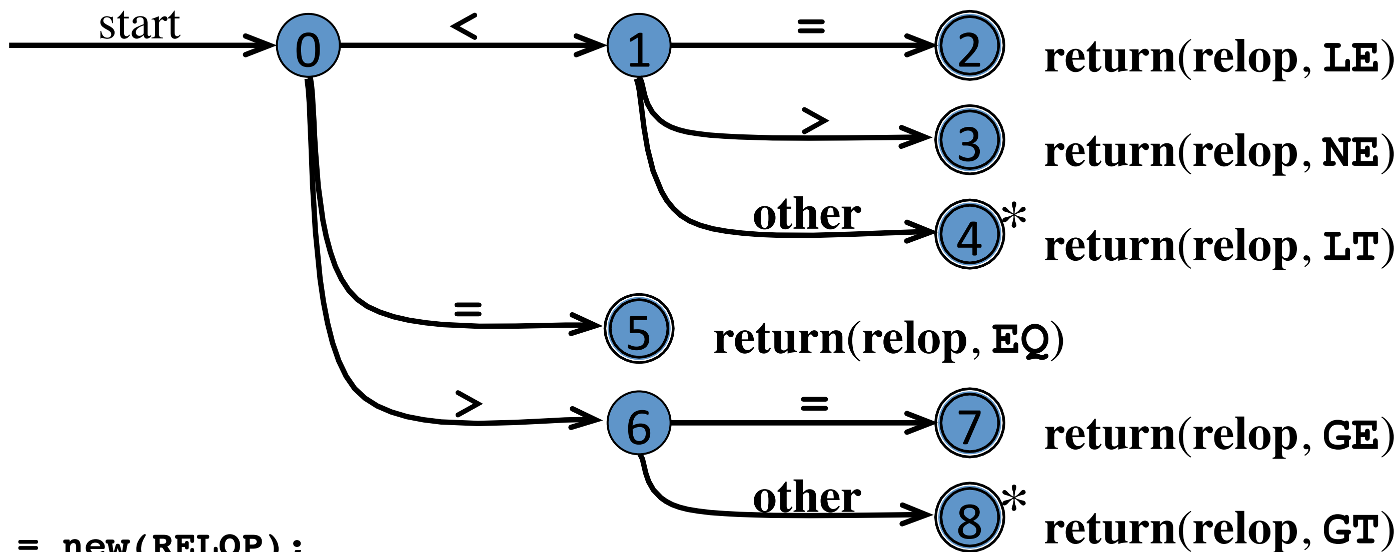
num $\rightarrow$ **digit**$^{+}$ (**. digit**$^{+}$)? (**E** (**+** | **-**)? **digit**$^{+}$)?



From Individual Transition Diagrams to Code

- Easy to convert each Transition Diagram into code
- Loop with multiway branch (switch/case) based on the current state to reach the instructions for that state
- Each state is a multiway branch based on the next input channel

Coding the Transition Diagrams for Relational Operators




```
TOKEN getRelop()
{    TOKEN retToken = new(RELOP);
     while(1) { /* repeat character processing
          until a return or failure occurs */
          switch(state) {
               case 0:  c = nextChar();
                        if(c == '<') state = 1;
                        else if (c == '=') state = 5;
                        else if (c == '>') state = 6;
                        else fail() ; /* lexeme is not a relop */
                        break;
               case 1: ...
               ...
               case 8:  retract();
                        retToken.attribute = GT;
                        return(retToken);
}    }    }
```

Putting the code together

```
token nexttoken()
{ while (1) {
    switch (state) {
    case 0: c = nextchar();
      if (c==blank || c==tab || c==newline) {
        state = 0;
        lexeme_beginning++;
      }
      else if (c=='<') state = 1;
      else if (c=='=') state = 5;
      else if (c=='>') state = 6;
      else state = fail();
      break;
     case 1:
      …
     case 9: c = nextchar();
      if (isletter(c)) state = 10;
      else state = fail();
      break;
     case 10: c = nextchar();
      if (isletter(c)) state = 10;
      else if (isdigit(c)) state = 10;
      else state = 11;
      break;
     …
```

The transition diagrams for the various tokens can be tried sequentially: on failure, we re-scan the input trying another diagram.

```
int fail()
{ forward = token_beginning;
  switch (state) {
  case  0: start =  9; break;
  case  9: start = 12; break;
  case 12: start = 20; break;
  case 20: start = 25; break;
  case 25: recover(); break;
  default: /* error */
  }
  return start;
}
```

Putting the code together: Alternative solutions

- The diagrams can be checked in parallel
- The diagrams can be merged into a single one, typically *non-deterministic*: this is the approach we will study in depth.

Lexical errors

- Some errors are out of power of lexical analyzer to recognize:

```
fi (a == f(x)) …
```

- However, it may be able to recognize errors like:

```
d = 2r
```

- Such errors are recognized when no pattern for tokens matches a character sequence

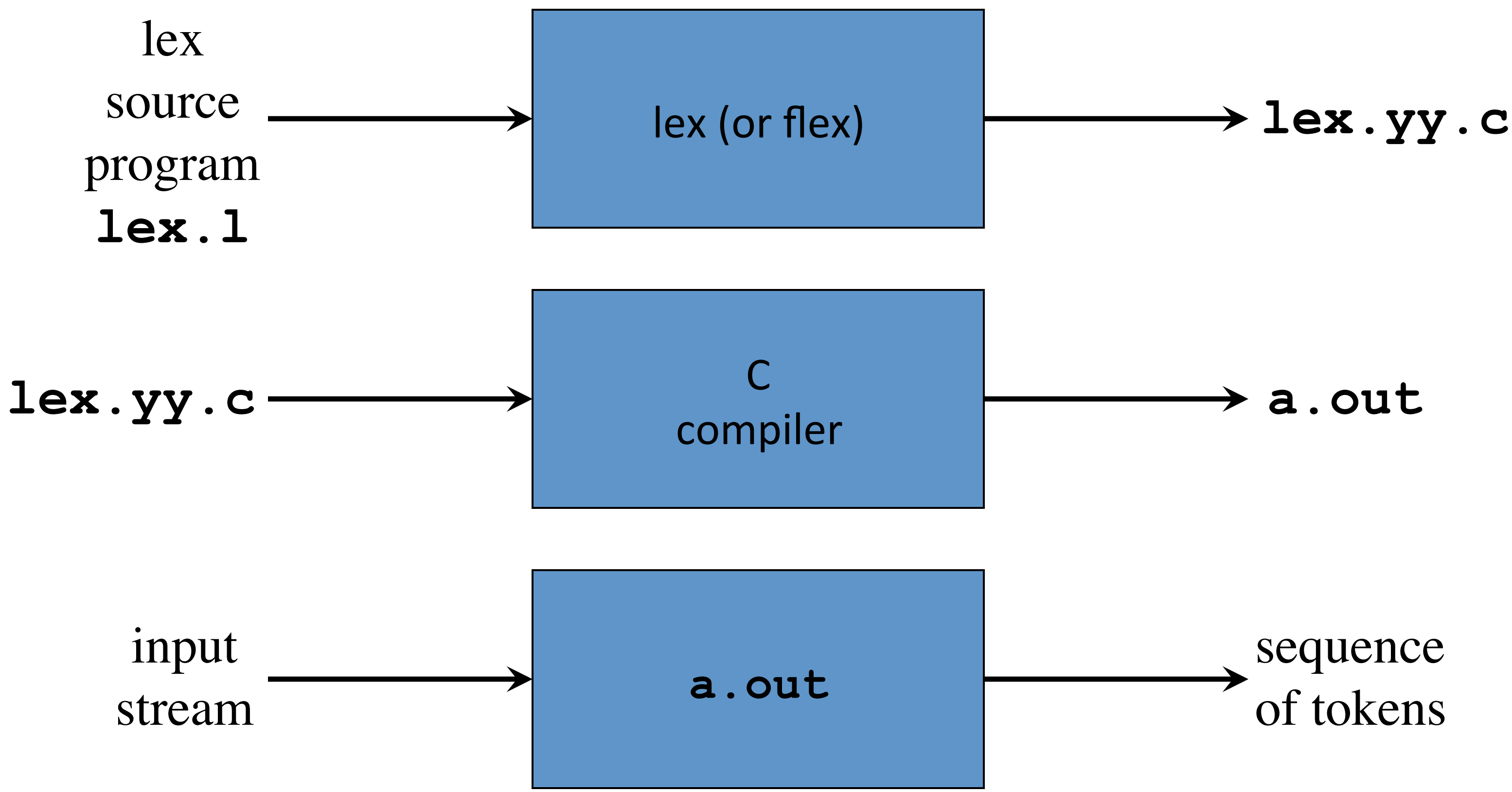
Error recovery

- Panic mode: successive characters are ignored until we reach to a well formed token
- Delete one character from the remaining input
- Insert a missing character into the remaining input
- Replace a character by another character
- Transpose two adjacent characters
- Minimal Distance

The Lex and Flex Scanner Generators

- *Lex* and its newer cousin *flex* are *scanner generators*
- Scanner generators systematically translate regular definitions into C source code for efficient scanning
- Generated code is easy to integrate in C applications

Creating a Lexical Analyzer with Lex and Flex



Lex Specification

- A *lex specification* consists of three parts:
  regular definitions, C declarations in `%{ %}`
  `%%`
  translation rules
  `%%`
  user-defined auxiliary procedures
- The *translation rules* are of the form:
  p_1 { $action_1$ }
  p_2 { $action_2$ }
  …
  p_n { $action_n$ }

Regular Expressions in Lex

x	match the character **x**
\\.	match the character **.**
"*string*"	match contents of string of characters
.	match any character except newline
^	match beginning of a line
$	match the end of a line
[xyz]	match one character **x**, **y**, or **z** (use **** to escape **-**)
[^xyz]	match any character except **x**, **y**, and **z**
[a-z]	match one of **a** to **z**
r*****	closure (match zero or more occurrences)
r**+**	positive closure (match one or more occurrences)
r**?**	optional (match zero or one occurrence)
r_1r_2	match r_1 then r_2 (concatenation)
r_1**|**r_2	match r_1 or r_2 (union)
(r **)**	grouping
r_1****r_2	match r_1 when followed by r_2
{d**}**	match the regular expression defined by d

Example Lex Specification 1

Translation rules

Contains the matching lexeme

```
%{
#include <stdio.h>
%}
%%
[0-9]+  { printf("%s\n", yytext); }
.|\n    { }
%%
main()
{ yylex();
}
```

Invokes the lexical analyzer

```
lex spec.l
gcc lex.yy.c -ll
./a.out < spec.l
```

Example Lex Specification 2

Translation rules

Regular definition

```
%{
#include <stdio.h>
int ch = 0, wd = 0, nl = 0;
%}
delim     [ \t]+
%%
\n        { ch++; wd++; nl++; }
^{delim}  { ch+=yyleng; }
{delim}   { ch+=yyleng; wd++; }
.         { ch++; }
%%
main()
{ yylex();
  printf("%8d%8d%8d\n", nl, wd, ch);
}
```

Example Lex Specification 3

Regular definitions

Translation rules

```
%{
#include <stdio.h>
%}
digit      [0-9]
letter     [A-Za-z]
id         {letter}({letter}|{digit})*
%%
{digit}+   { printf("number: %s\n", yytext); }
{id}       { printf("ident: %s\n", yytext); }
.          { printf("other: %s\n", yytext); }
%%
main()
{ yylex();
}
```

Example Lex Specification 4

```
%{ /* definitions of manifest constants */
#define LT (256)
…
%}
delim       [ \t\n]
ws          {delim}+
letter      [A-Za-z]
digit       [0-9]
id          {letter}({letter}|{digit})*
number      {digit}+(\.{digit}+)?(E[+\-]?{digit}+)?
%%
{ws}        { }
if          {return IF;}
then        {return THEN;}
else        {return ELSE;}
{id}        {yylval = install_id(); return ID;}
{number}    {yylval = install_num(); return NUMBER;}
“<“         {yylval = LT; return RELOP;}
“<=“        {yylval = LE; return RELOP;}
“=“         {yylval = EQ; return RELOP;}
“<>“        {yylval = NE; return RELOP;}
“>“         {yylval = GT; return RELOP;}
“>=“        {yylval = GE; return RELOP;}
%%
int install_id()
…
```

Return token to parser

Token attribute

Install **`yytext`** as identifier in symbol table