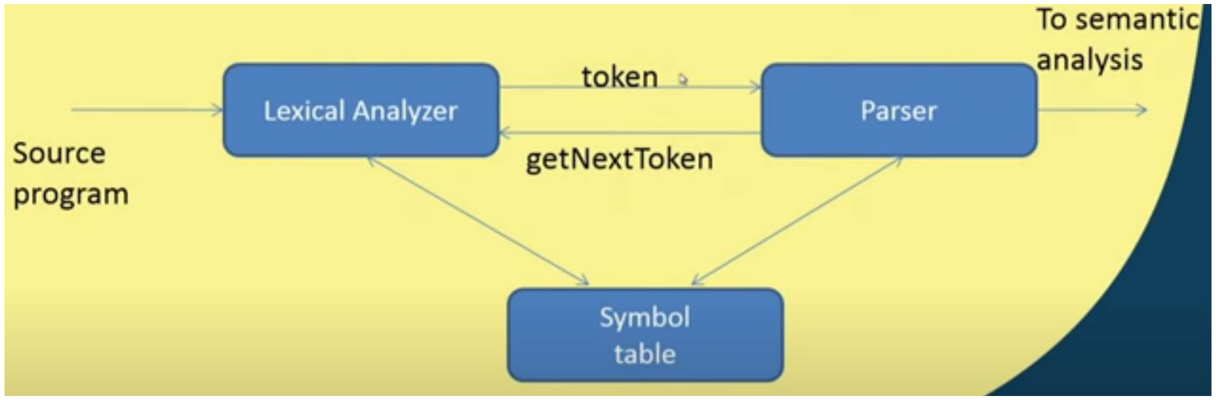
Lexical Analysis Phase

19CSE401 Compiler Design

Overview



- ***Main task***: to read input characters and group them into "*tokens*."
- ***Secondary tasks***:
  - Skip comments and whitespace;
  - Correlate error messages with source program (e.g., line number of error).

Lexical Analysis

- Lexical analyzer: reads input characters and produces a sequence of tokens as output (nexttoken()).
  - Trying to understand each element in a program.
  - *Token*: a group of characters having a collective meaning.

```
const pi = 3.14159;
```

Tokens identified:

```
Token 1: (const, -)
Token 2: (identifier, ‘pi’)
Token 3: (=, -)
Token 4: (realnumber, 3.14159)
Token 5: (;, -)
```

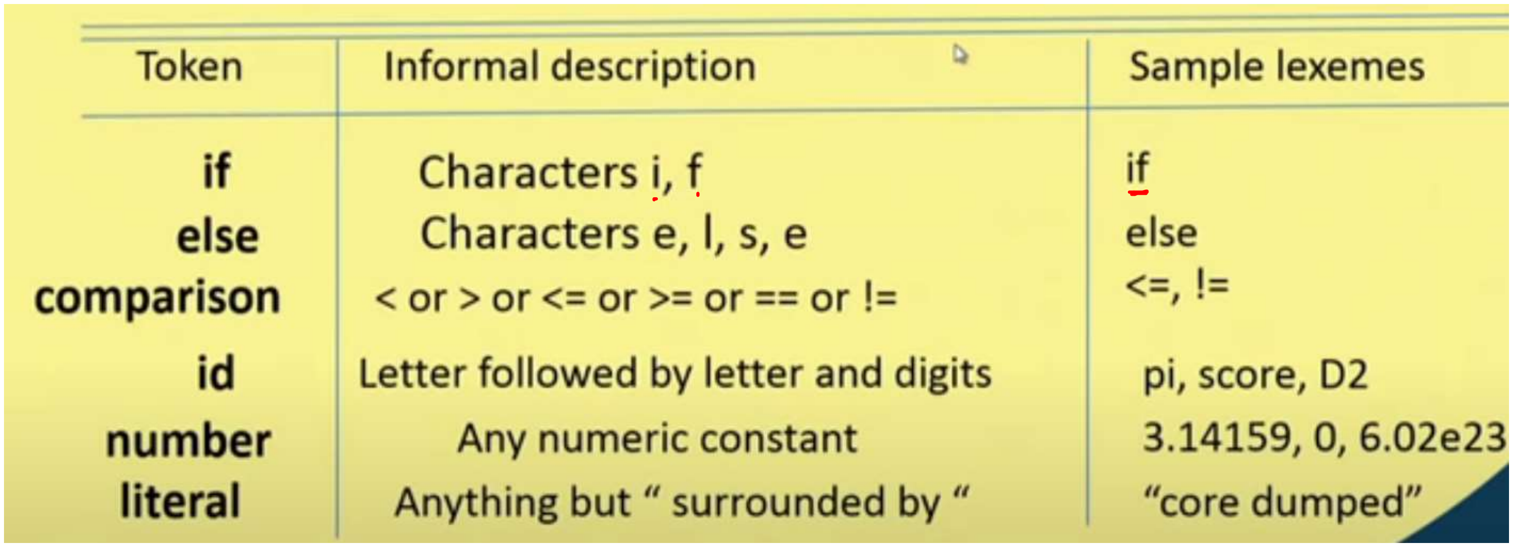
Separate Lexical analysis and Parsing

- Simplicity of design
  - simplify both the lexical analysis and the syntax analysis
- Improving compiler efficiency
  - specialized techniques can be applied to improve lexical analysis.
- Enhancing compiler portability
  - only the scanner needs to communicate with the outside

Token, Pattern, Lexeme

- *Token*:
  - a group of characters having a collective meaning.
  - A token is a pair <*token name, optional token value*>
    - E.g. token: identifier, lexeme: pi, etc.
- *Pattern:*
  - The rule describing how a token can be formed.
  - A pattern is a description of the form that the lexemes of a token may take.
    - E.g: identifier: ([a-z] | [A-Z]) ([a-z] | [A-Z] | [0-9])*
- *Lexeme*
  - A lexeme is a sequence of characters in the source program that matches the pattern for a token.
  - A *lexeme* is a particular instant of a token.

Example



Token	Informal description	Sample lexemes
if	Characters i, f	if
else	Characters e, l, s, e	else
comparison	< 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 " surrounded by "	"core dumped"

- Two issues in lexical analysis.
  - How to specify tokens (patterns)?
  - How to recognize the tokens giving a token specification (how to implement the nextToken() routine)?
- How to specify tokens:
  - all the basic elements in a language must be tokens so that they can be recognized.

```
main( ) {
    int i, j;
    for (I=0; I<50; I++) {
        printf("I = %d", I);
    }
}
```

    - Token types: constant, identifier, reserved word, operator and misc. symbol.
  - Tokens are specified by **regular expressions**.

What exactly is lexing?

Consider the code:

```
if (i==j);
     z=1;
else;
     z=0;
endif;
```

This is really nothing more than a string of characters:

i	f	_	(	i	=	=	j	)	;	\n	\t	z	=	1	;	\n	e	l	s	e	;	\n	\t	z	=	0	;	\n	e	n	d	i	f	;

During our lexical analysis phase we must divide this string into meaningful sub-strings.

Tokens

- The output of our lexical analysis phase is a streams of tokens.
- A token is a syntactic category.

- In English this would be types of words or punctuation, such as a "noun", "verb", "adjective" or "end-mark".
- In a program, this could be an "identifier", a "floating-point number", a "math symbol", a "keyword", etc…

Identifying Tokens

- A sub-string that represents an instance of a token is called a lexeme.
- The class of all possible lexemes in a token is described by the use of a pattern.
- For example, the pattern to describe an identifier (a variable) is a string of letters, numbers, or underscores, beginning with a non-number.
- Patterns are typically described using regular expressions.

Implementation of LA

A lexical analyzer must be able to do three things:

1. Remove all whitespace and comments.

2. Identify tokens within a string.

3. Return the lexeme of a found token, as well as the line number it was found on.

Example

i	f	_	(	i	=	=	j	)	;	\n	\t	z	=	1	;	\n	e	l	s	e	;	\n	\t	z	=	0	;	\n	e	n	d	i	f	;

Line	Token	Lexeme
1	BLOCK_COMMAND	if
1	OPEN_PAREN	(
1	ID	i
1	OP_RELATION	==
1	ID	j
1	CLOSE_PAREN	)
1	ENDLINE	;
2	ID	z
2	ASSIGN	=
2	NUMBER	1
2	ENDLINE	;
3	BLOCK_COMMAND	else

- Etc…

Attributes For Tokens

- The lexical analyzer collects information about tokens into their associated attributes.
- As a practical matter ,a token has usually only a single attribute, a pointer to the symbol-table entry in which the information about the token is kept; the pointer becomes the attribute for the token.
- **Example:** Let **num** be the token representing an integer. when a sequence of digits appears in the input stream, the lexical analyzer will pass **num** to the parser. The value of the integer will be passed along as an attribute of the token **num**.
- Logically, the lexical analyzer passes both the token and the attribute to the parser.
- If we write a token and its attribute as a tuple enclosed b/w < >, the input

33 + 89 – 60

is transformed into the sequence of tuples

< **num**, 33 > <+, > <**num**, 89 > <-, > <**num**, 60>

The token "+" has no attribute ,the second components of the tuples ,the attribute ,play no role during parsing,but are needed during translation.

Attributes For Tokens (cont’d)

The token and associated attribute values in the “ C ” statement.

E = M + C * 2

Tell me the token and their attribute value ?

Attributes for Tokens

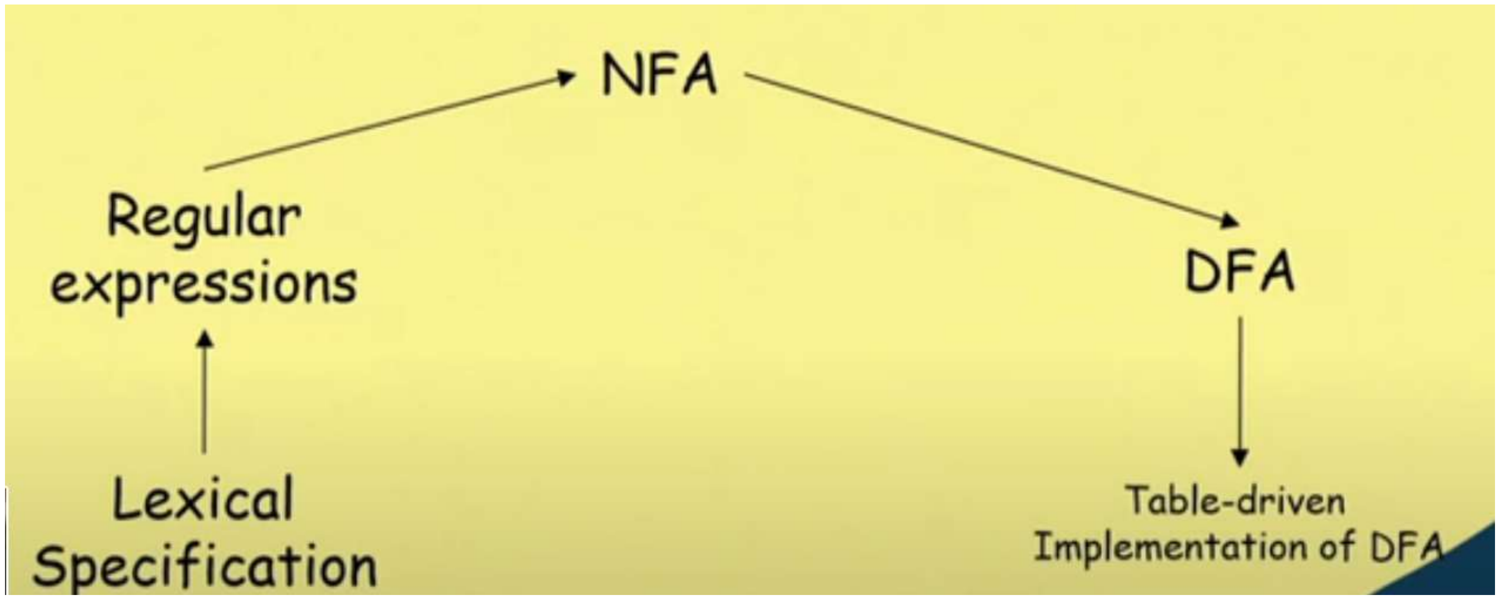
- E = M * C ** 2
  - <id, pointer to symbol table entry for E>
  - <assign-op>
  - <id, pointer to symbol table entry for M>
  - <mult-op>
  - <id, pointer to symbol table entry for C>
  - <exp-op>
  - <number, integer value 2>

Specification of Token Lexical Phase

Scanner from the Specification

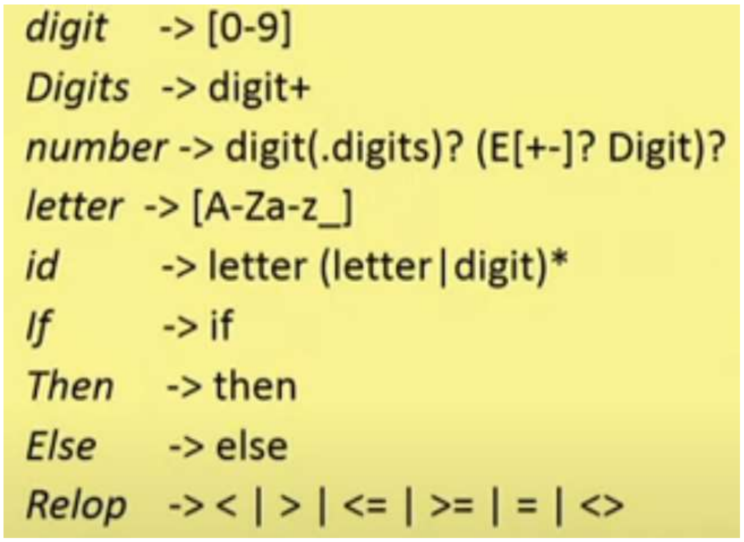
- The collection of tokens of a programming language can be specified by a set of regular expressions.
- A **scanner or lexical analyzer** for the language uses a DFA (recognizer of regular languages) in its core.
- **Different final states** of the DFA identifies different tokens.
- A scanner is a big DFA, essentially the "aggregate" of the automata for the individual tokens.

Lexical Analyser Tool



Recognition of Tokens

- Formalize the pattern for tokens



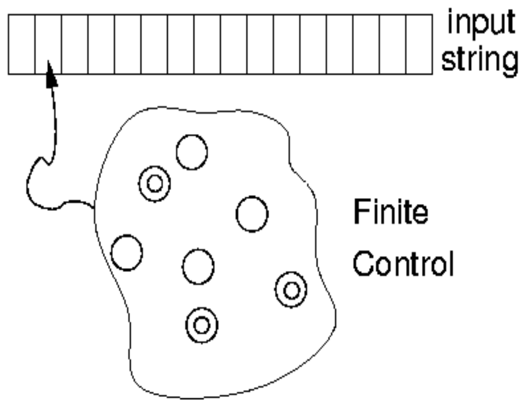
```
digit    -> [0-9]
Digits   -> digit+
number -> digit(.digits)? (E[+-]? Digit)?
letter  -> [A-Za-z_]
id       -> letter (letter|digit)*
If       -> if
Then     -> then
Else     -> else
Relop    -> < | > | <= | >= | = | <>
```

- We also need to handle whitespace

Recognizing Tokens: Finite Automata

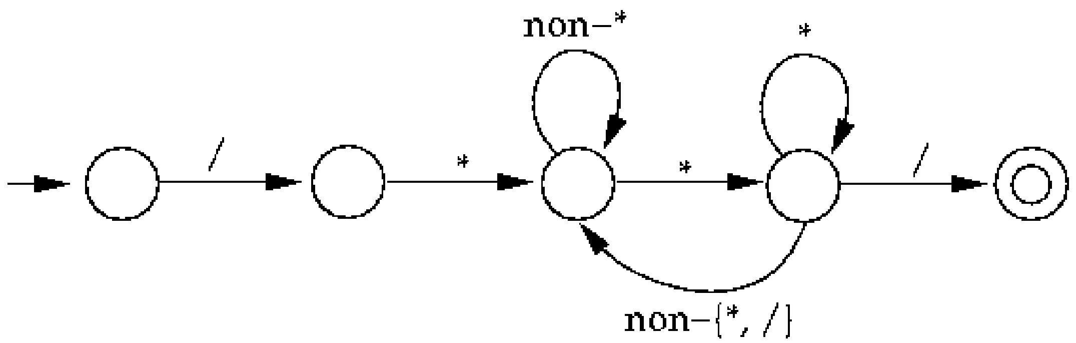
A *finite automaton* is a 5-tuple (Q, Σ, T, q_0, F), where:

- Σ is a finite alphabet;
- Q is a finite set of states;
- T: $Q \times \Sigma \rightarrow Q$ is the transition function;
- $q_0 \in Q$ is the initial state; and
- $F \subseteq Q$ is a set of final states.

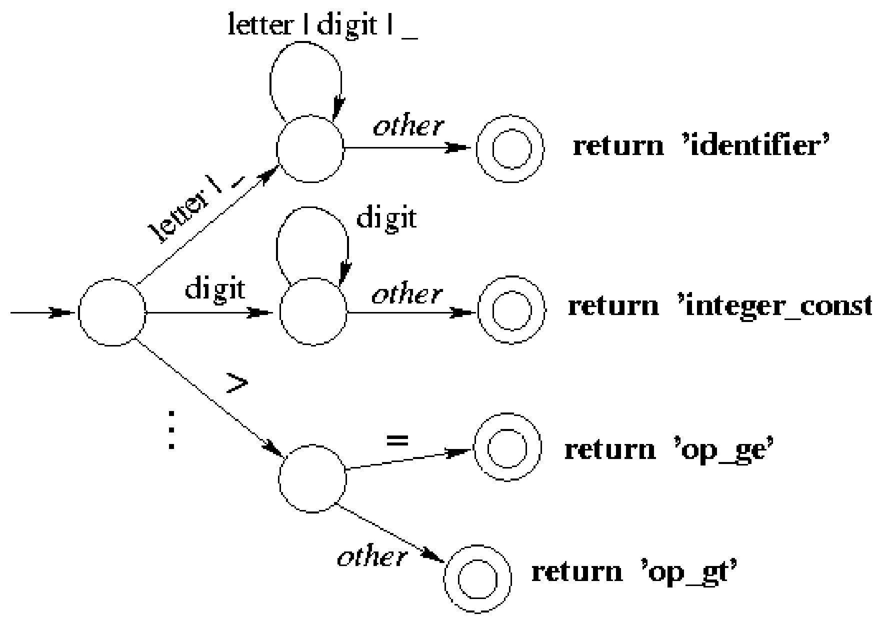


Finite Automata: An Example

A (deterministic) finite automaton (DFA) to match C-style comments:



Structure of a Scanner Automaton



How much should we match?

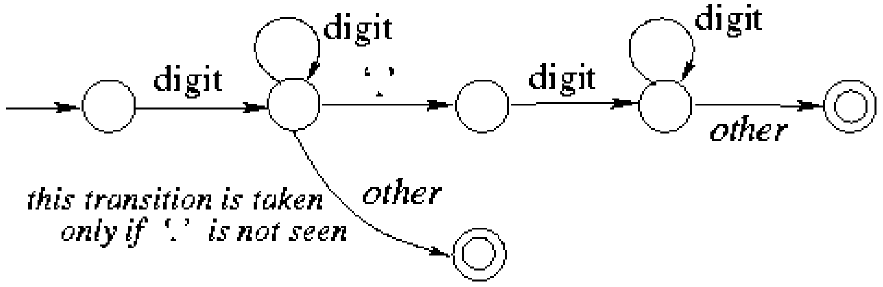
In general**, find the longest match possible**.

E.g., on input 123.45, match this as

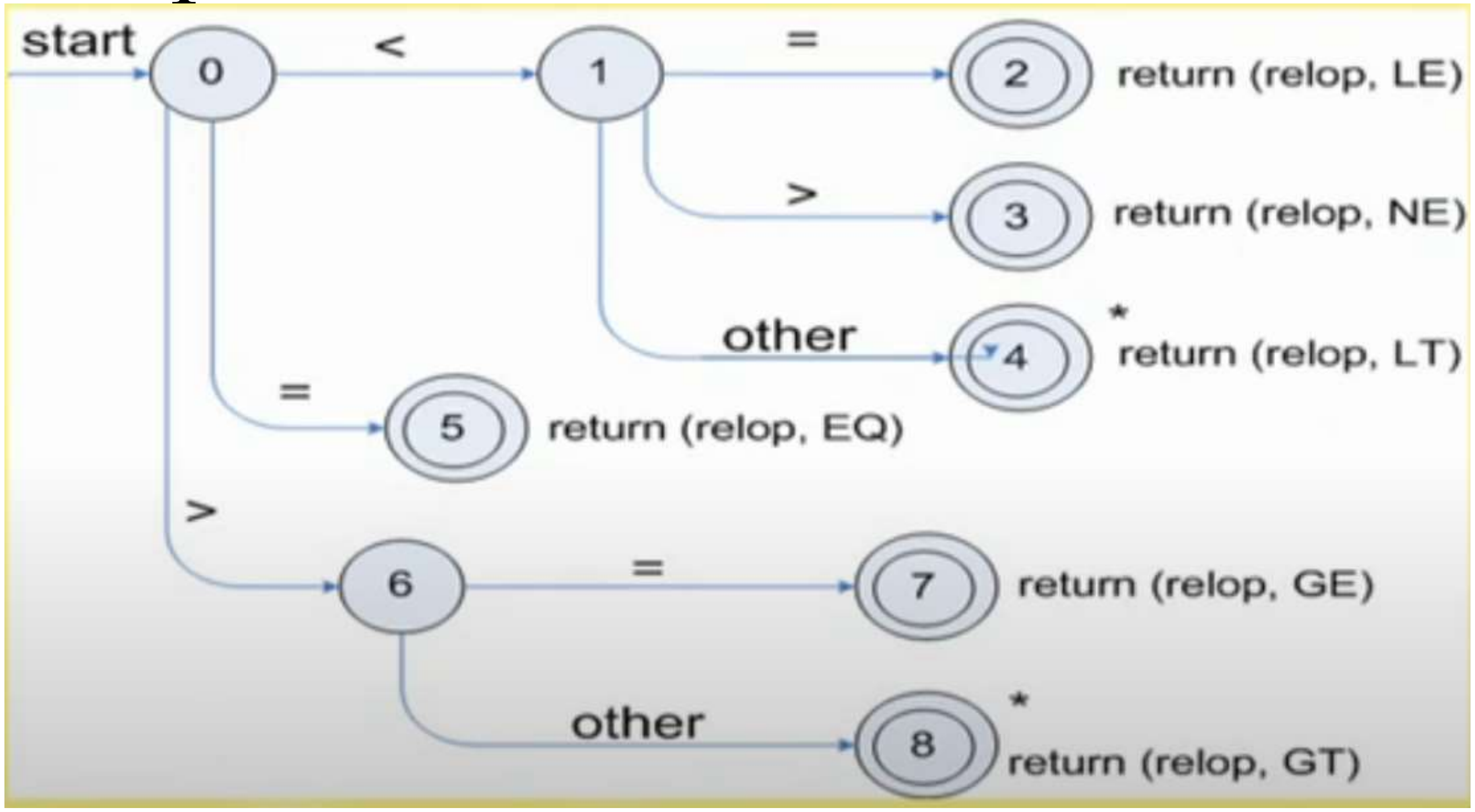
num_const(123.45)

rather than

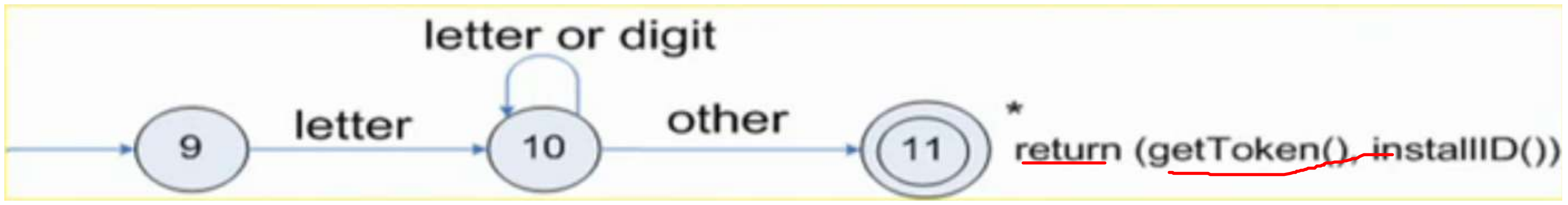
num_const(123), “.”, num_const(45).



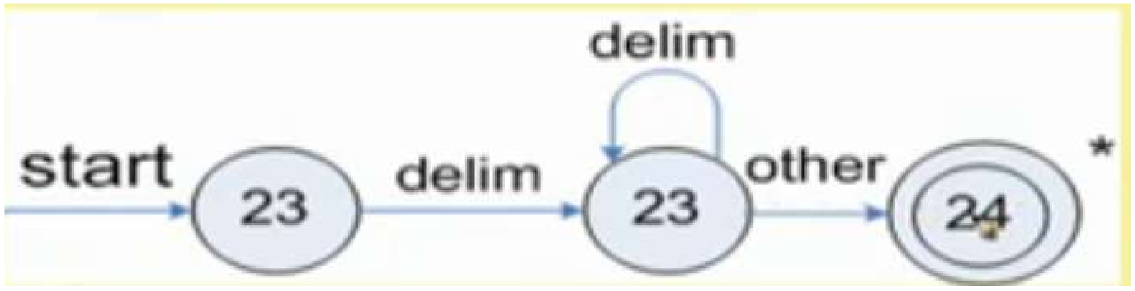
Transition Diagram for Relational Operator



Transition Diagram for Identifiers



Transition of whitespaces



Example

if	{ return IF; }
[a-z][a-z0-9]*	{ return ID; }
[0-9]+	{ return NUM; }
[0-9]"."[0-9]*|[0-9]*"."[0-9]+	{ return REAL; }
(\-\-[a-z]*\n)|(" "|\n|\t)	{ ; }
.	{ error(); }

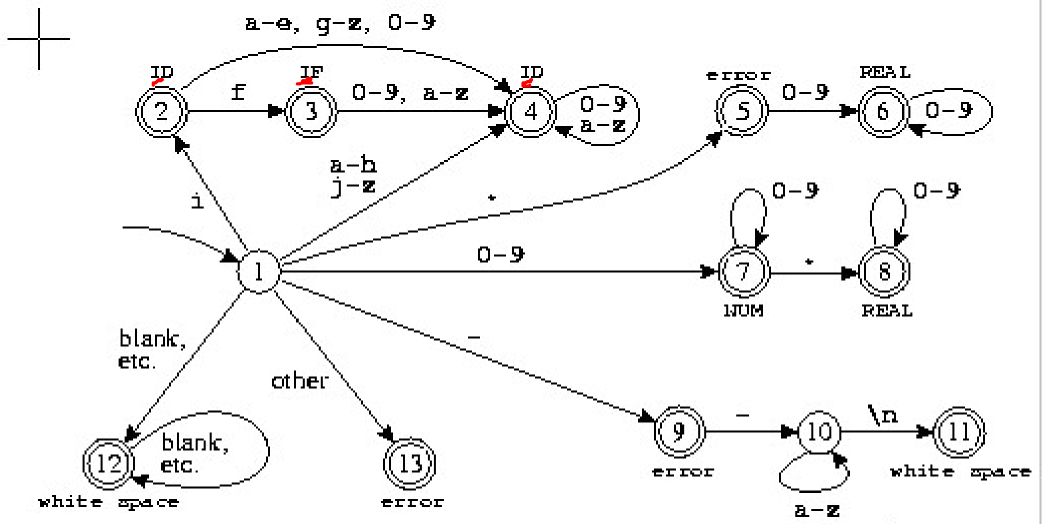


FIGURE 2.4. Combined finite automaton.
From *Modern Compiler Implementation in ML,*
Cambridge University Press, ©1998 Andrew W. Appel

Theory vs. Practice

- Two differences:
- DFAs recognize lexemes. A lexer must return a type of acceptance (token type) rather than simply an accept/reject indication.
- DFAs consume the complete string and accept or reject it.

A lexer must find the end of the lexeme in the input stream and then find the next one, etc.

Lexical errors

- Some errors are out of power of lexical analyser to recognize:
  - fi (a == f(x)) . . . <ID> <OPEN> <ID> <EQL> <ID> <OPEN> <ID> <CLOSE> <CL
- However it may be able to recognize errors like:
  - d = 2r

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