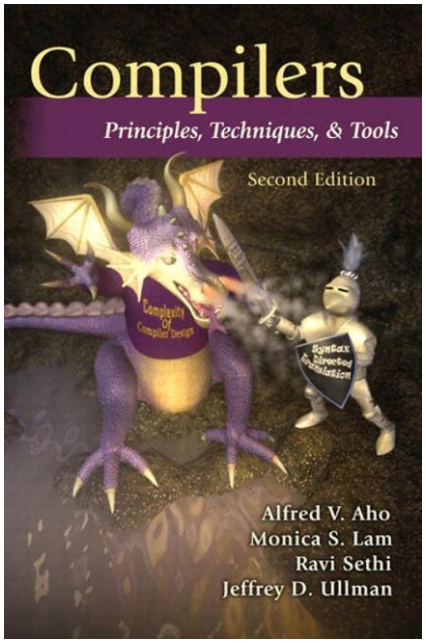
Compiler

Lec 02

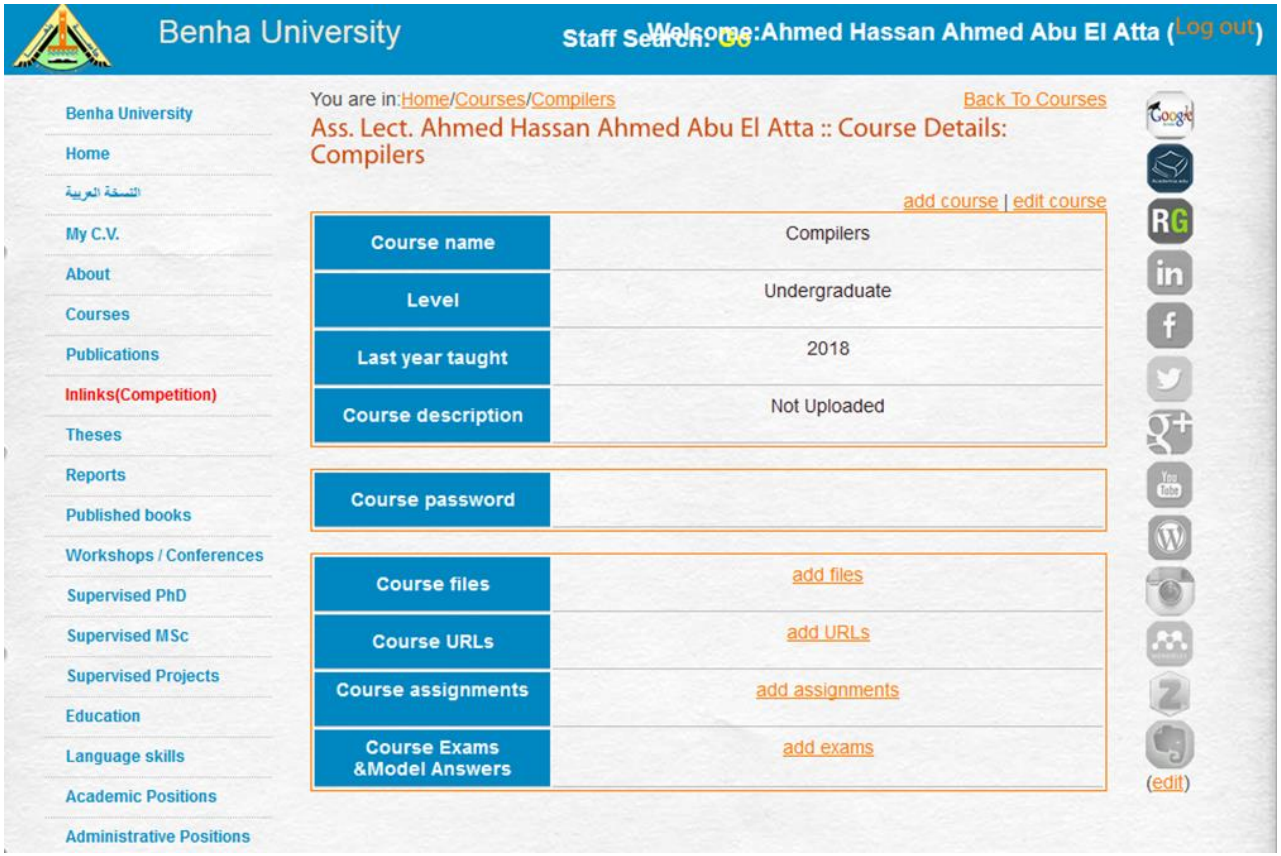
Book

Compilers: Principles, Techniques, and Tools is a computer science textbook by Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman about compiler construction.

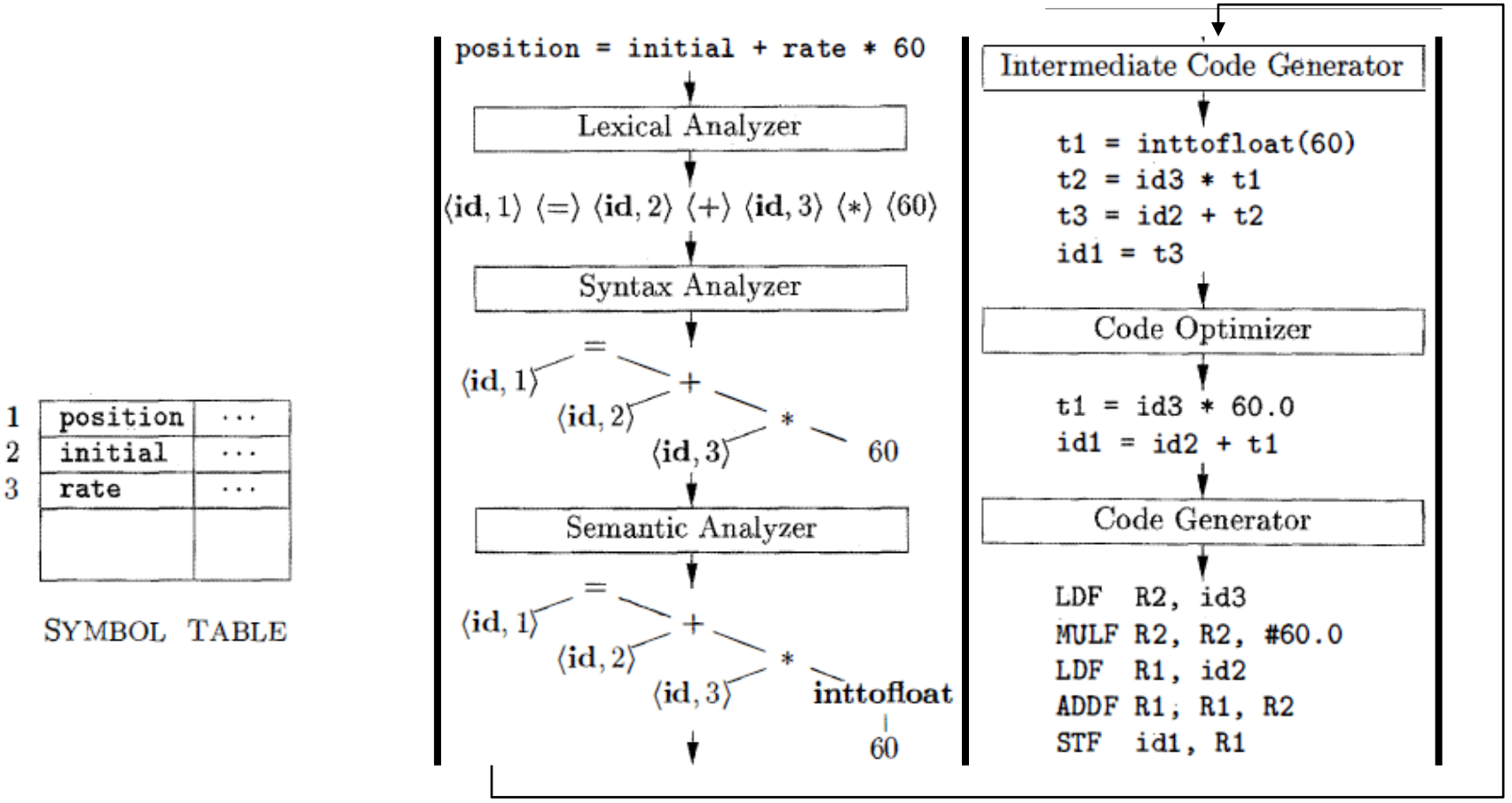


PowerPoint

http://www.bu.edu.eg/staff/ahmedaboalatah14-courses/14779



Phases of a compiler



Lexical Analysis

PART ONE

The Tasks of the Lexical Analyzer

- The main task of the lexical analyzer is to ***read the input characters*** of the source program. Group them into ***lexemes.*** Produce as output a ***sequence of tokens*** for each lexeme in the source program.
- When the lexical analyzer discovers a lexeme constituting an ***identifier***, it needs to enter that lexeme into the ***symbol table***.
- Remove comments and whitespace (blank, newline, tab, and perhaps other characters that are used to separate tokens in the input).
- keep track of the ***number of newline*** characters seen, so it can associate a line number with each ***error message***.
- If the source program uses a ***macro-preprocessor***, the ***expansion of macros*** may also be ***performed by the lexical analyzer***.

Lexical Analyzer & Parsing

- Lexical analysis and parsing are separated:
  - ***Simplicity***: For example, a parser that had to deal with comments and whitespace would be more complex than one that can assume comments and whitespace have already been removed by the lexical analyzer.
  - ***Efficiency***: A separate lexical analyzer allows us to apply specialized techniques that serve only the lexical task, not the job of parsing.
  - ***Compiler portability is enhanced***. Input-device-specific features can be restricted to the lexical analyzer.

Tokens

- A token is a pair consisting of a ***token name*** and ***an optional attribute value***.
- The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier.

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>

Patterns

- A pattern is a description of ***the form*** that the lexemes of a token may take.
- In the case of a ***keyword*** as a token, the pattern is just ***the sequence of characters that form the keyword***.
- For ***identifiers*** and some other tokens, the pattern is a more complex structure that is ***matched by many strings***.

else

[0-9]+

[a-z]+

Lexemes

- A lexeme is ***a sequence of characters*** in the source program that ***matches the pattern for a token*** and is identified by the lexical analyzer as ***an instance of that token***.

E = M * C ** 2

E

=

M

*

C

**

2

Examples

1. One token for each keyword.
2. Tokens for the operators, either individually or in classes such as the token Comparison
3. One token representing all identifiers.
4. One or more tokens representing constants, such as numbers.
5. Tokens for each punctuation symbol, such as left and right parentheses, comma, and semicolon.

Token	Patterns	Lexemes
if	if	if
else	else	else
id	[a-z]+	pi, score
integer	[0-9]+	0, 25, 685
comparison	<|>|<=	<, >, <=, ==

Attributes for Tokens

- The pattern for token ***number*** matches both ***0*** and ***1***.
- It is extremely important for the code generator to know which lexeme was found in the source program.
- Thus , in many cases the lexical analyzer returns to the parser an attribute value that describes the lexeme represented by the token.

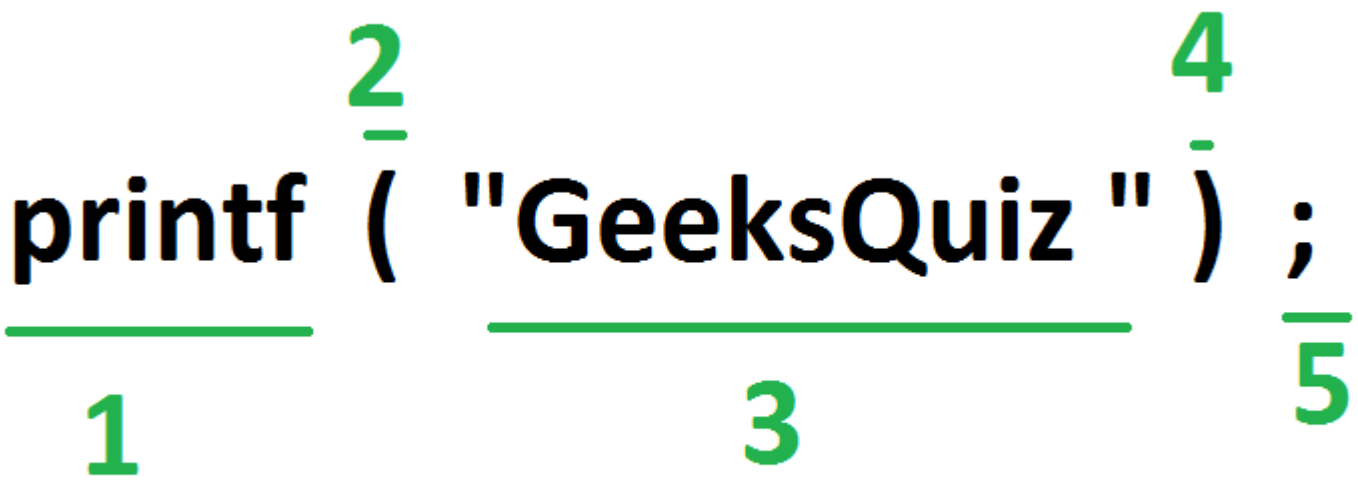
<**id**, pointer to symbol-table entry>

<**exp_op**>

<**number**, integer value>

<type, lexeme, line, column>

Example



Problems When Recognizing Tokens

- Fortran 90 ignores whitespace.
- First mean Do5i = 1.25.

DO 5 I = 1.25

- Second, mean Do statement.

DO 5 I = 1,25

- Need to lookahead.

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.

Input Buffering

- Sometimes lexical analyzer needs to lookahead some symbols to decide about the token to return
  - In C language: we need to look after -, = or < to decide what token to return (- > , ==, or <=).
  - In Fortran: DO 5 I = 1.25
- We need to introduce a two buffer scheme to handle large lookaheads safely

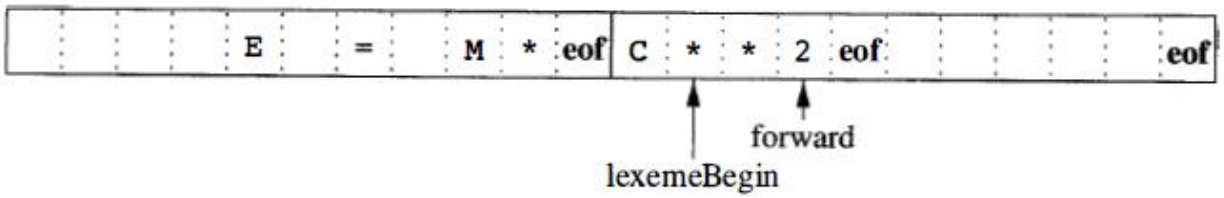


Figure 3.4: Sentinels at the end of each buffer

Specification of Tokens

- Two issues in lexical analysis.
  - How to specify tokens (patterns)?
  - How to recognize the tokens giving a token specification?
- How to specify tokens:
  - All the basic elements in a language must be tokens so that they can be recognized.
    - Token types: constant, identifier, reserved word, operator and misc. symbol.
- Tokens are specified by ***regular expressions***.

Operations on Languages

- *alphabet* : a finite set of symbols. E.g. {a, b, c}
- A *string* over an alphabet is a finite sequence of symbols drawn from that alphabet (sometimes a string is also called a sentence or a word).
- A *language* is a set of strings over an alphabet.
- Operation on languages (a set):

OPERATION	DEFINITION AND NOTATION
Union of L and M	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
Concatenation of L and M	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
Kleene closure of L	$L^* = \cup_{i=0}^{\infty} L^i$
Positive closure of L	$L^+ = \cup_{i=1}^{\infty} L^i$

- Example:
  - L={aa, bb, cc}, M = {abc}

Regular expression

- Ɛ is a regular expression, L(Ɛ) = {Ɛ}
- If a is a symbol in ∑then a is a regular expression, L(a) = {a}
- (r) | (s) is a regular expression denoting the language L(r) ∪ L(s)
- (r)(s) is a regular expression denoting the language L(r)L(s)
- (r)* is a regular expression denoting (L(r))*
- (r) is a regular expression denting L(r)

Example

Example 3.4: Let $\Sigma = \{a, b\}$.

1. The regular expression $\mathbf{a|b}$ denotes the language $\{a, b\}$.
2. $(\mathbf{a|b})(\mathbf{a|b})$ denotes $\{aa, ab, ba, bb\}$, the language of all strings of length two over the alphabet Σ. Another regular expression for the same language is $\mathbf{aa|ab|ba|bb}$.
3. $\mathbf{a}^*$ denotes the language consisting of all strings of zero or more a's, that is, $\{\epsilon, a, aa, aaa, \ldots\}$.
4. $(\mathbf{a|b})^*$ denotes the set of all strings consisting of zero or more instances of a or b, that is, all strings of a's and b's: $\{\epsilon, a, b, aa, ab, ba, bb, aaa, \ldots\}$. Another regular expression for the same language is $(\mathbf{a}^*\mathbf{b}^*)^*$.
5. $\mathbf{a|a}^*\mathbf{b}$ denotes the language $\{a, b, ab, aab, aaab, \ldots\}$, that is, the string a and all strings consisting of zero or more a's and ending in b.

Finite Automata

A finite Automata or ***FA*** is defined by the

$$M = (Q, \Sigma, \delta, q_0, F),$$

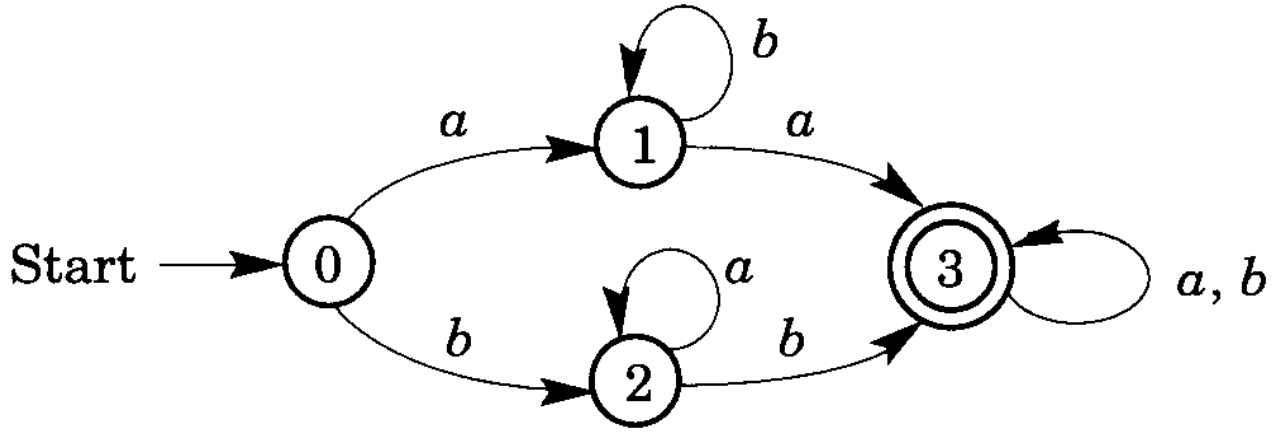
where

- Q is a finite set of states,
- Σ is a finite set of symbols called the input alphabet,
- $\delta : Q \times \Sigma \rightarrow Q$ is a total function called the transition function,
- $q_0 \in Q$ is the initial state,
- $F \subseteq Q$ is a set of final states.

Example Finite Automaton

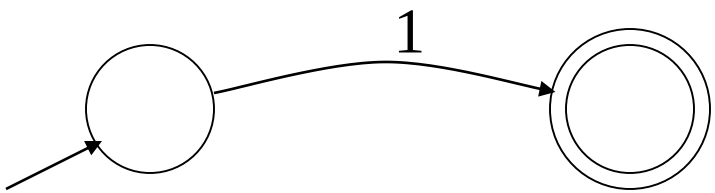
A path 0, 1, 1, 3 with edges labeled a, *b,* a. Since 0 is the start state and 3 is a final state, we conclude that the FA accepts the string aba.

The FA also accepts the string baaabab by traveling along the path 0, 2, 2, 2, 2, 3, 3, 3.



A Simple Example

A finite automaton that accepts only “1”

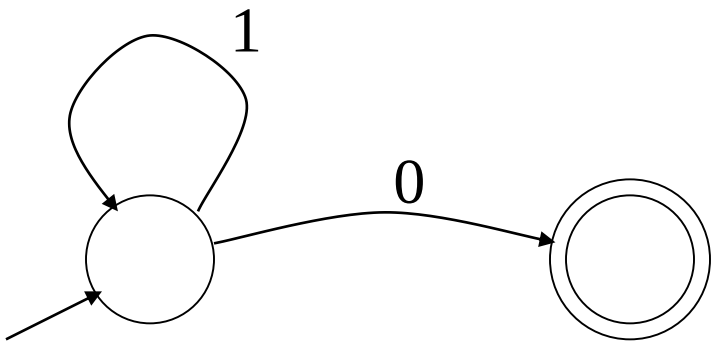


A finite automaton accepts a string if we can follow transitions labeled with the characters in the string from the start to some accepting state

Another Simple Example

A finite automaton accepting any number of 1's followed by a single 0

Alphabet: {0,1}

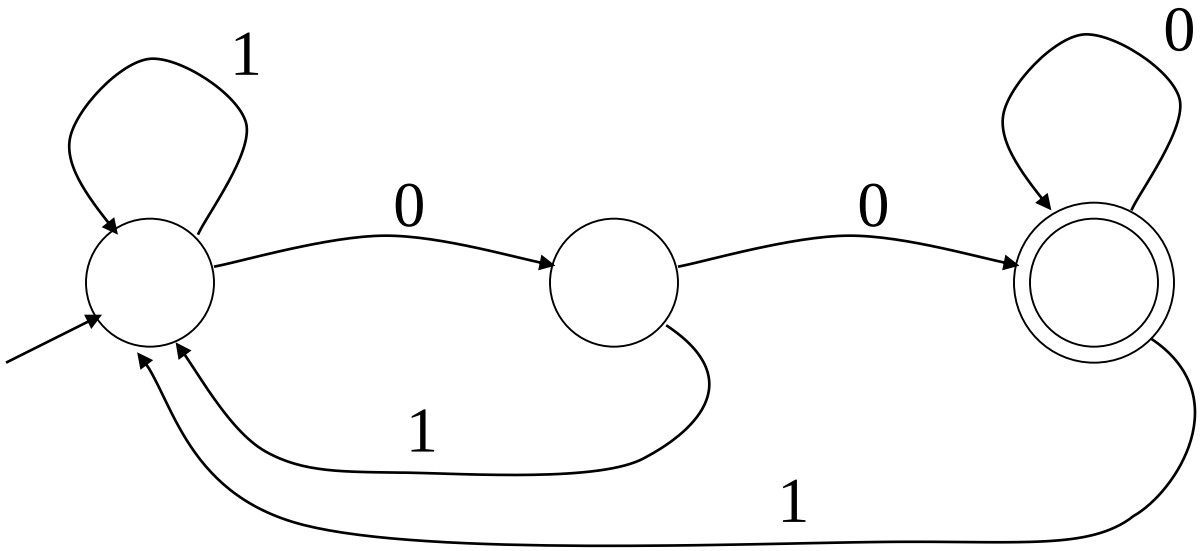


Check that "1110" is accepted but "110…" is not

And Another Example

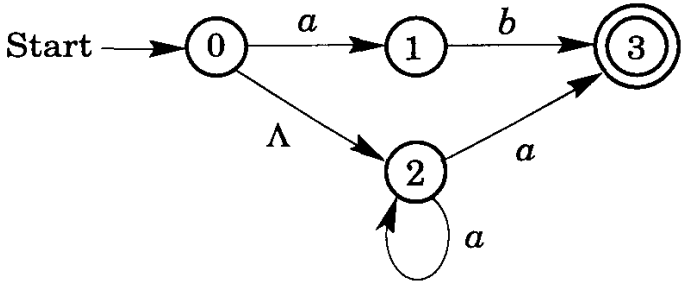
Alphabet {0,1}

What language does this recognize?



Nondeterministic Finite Automata (NFA)

1. Edge with ***ε.***
2. Missing labels
3. Multiply edges start from the same state with the same label

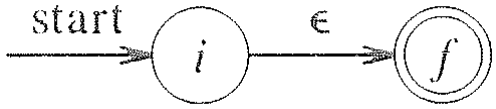


RE to NFA

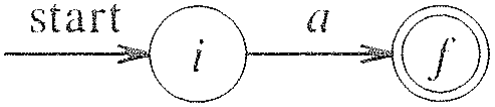
First parse *r* into its constituent sub expressions.

Construct NFA's for each of the basic symbols in *r*.

- for ε

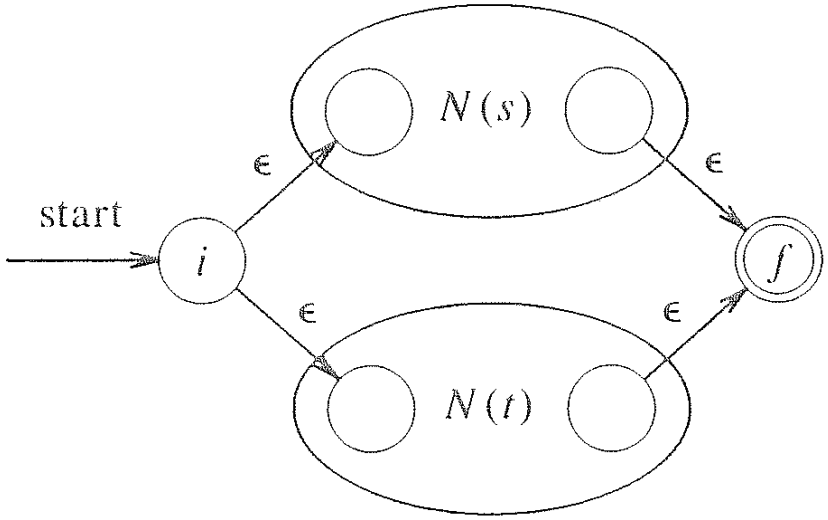


- for *a* in Σ

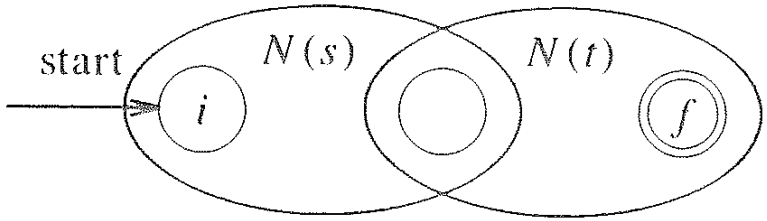


RE to NFA (cont.)

For the regular expression *s*|*t*,



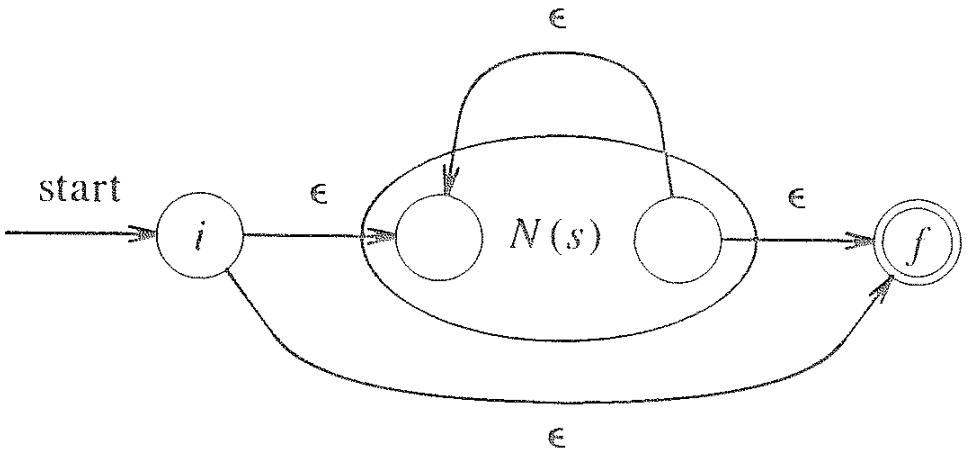
For the regular expression *st*,



RE to NFA (cont.)

For the regular expression *s**,

For the parenthesized regular expression (*s*), use *N(s)* itself as the NFA.



Every time we construct a new state, we give it a distinct name.

Example 3.24

Construct an NFA for r = (alb)*abb .

Examples

- Convert the following RE to NFA:
  - a | b
  - (a | b) (a | b)
  - a *
  - (a | b)*
  - a | a*b