Lexical Analysis

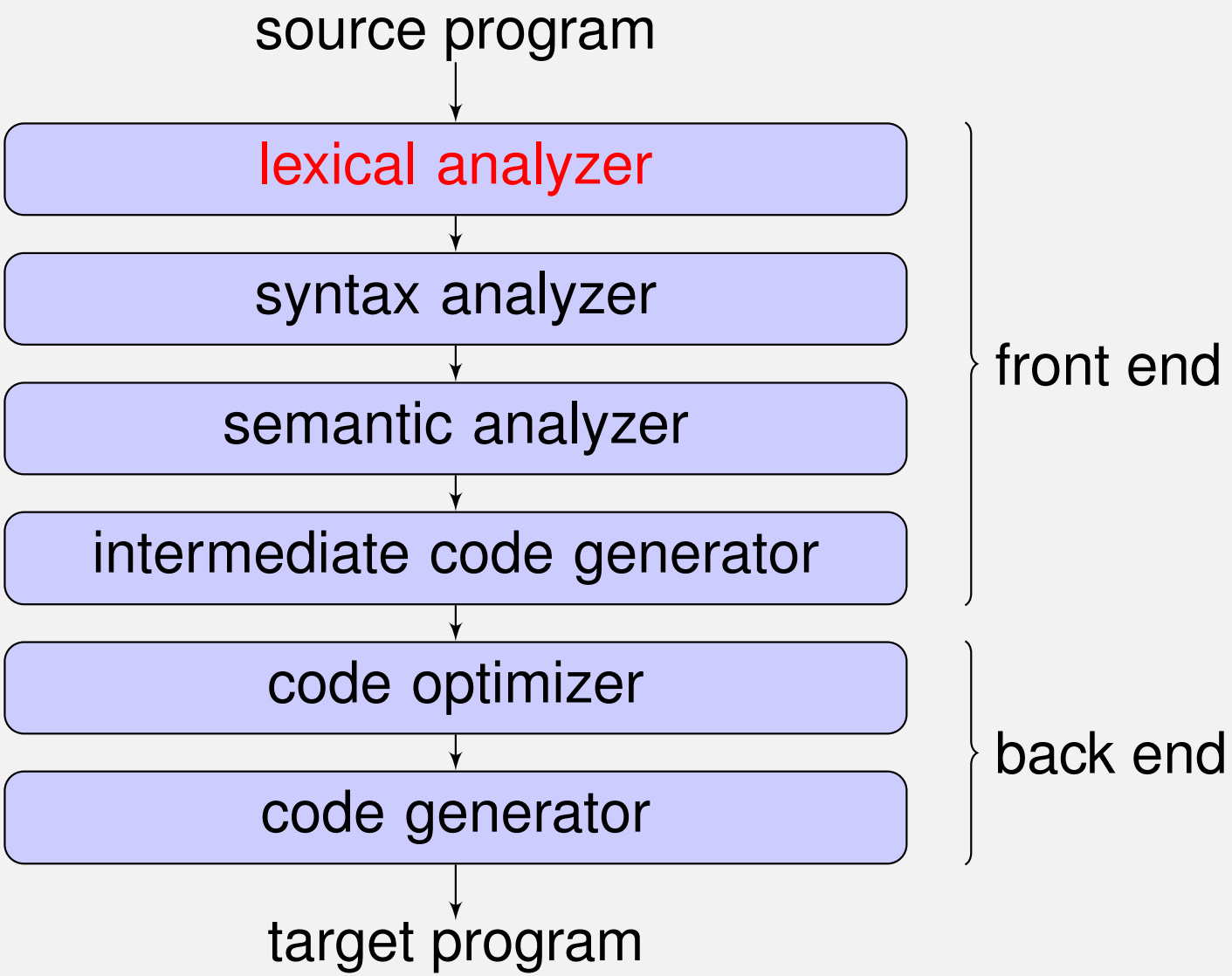
Dr. Nguyen Hua Phung

HCMC University of Technology, Viet Nam

08, 2016

Outline

Compilation Phases

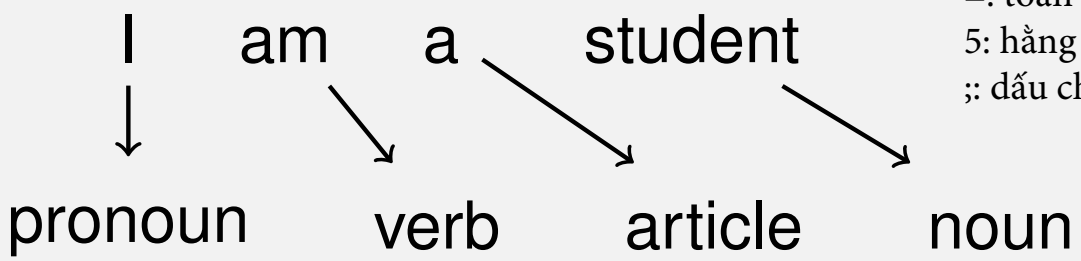


Source Program (Chương trình nguồn):
Đây là mã nguồn bạn viết bằng một ngôn ngữ lập trình cấp cao (ví dụ: C, Java, Python).
Ví dụ: int a = 5;
Lexical Analyzer (Bộ phân tích từ vựng):
Giai đoạn đầu tiên của trình biên dịch. Nó đọc mã nguồn và chia nhỏ thành các token (như từ khóa, biến, toán tử).
Ví dụ: Dòng int a = 5; sẽ được chia thành các token: int, a, =, 5, ;.
Mục tiêu: Loại bỏ các khoảng trắng, bình luận và xác định cấu trúc cơ bản.
Syntax Analyzer (Bộ phân tích cú pháp):
Kiểm tra cú pháp của chương trình dựa trên các quy tắc ngữ pháp của ngôn ngữ lập trình.
Ví dụ: Đảm bảo rằng câu lệnh int a = 5; có cú pháp đúng theo quy tắc của ngôn ngữ.
Nếu phát hiện lỗi cú pháp, chương trình không được tiếp tục.
Semantic Analyzer (Bộ phân tích ngữ nghĩa):
Kiểm tra ngữ nghĩa của chương trình, tức là ý nghĩa logic của nó.
Ví dụ: Đảm bảo biến a được khai báo trước khi sử dụng hoặc phép toán a + "hello" không hợp lệ vì kiểu dữ liệu không tương thích.
Nếu có lỗi ngữ nghĩa, quá trình biên dịch sẽ dừng lại.
Intermediate Code Generator (Bộ sinh mã trung gian):
Chuyển chương trình nguồn thành mã trung gian, một dạng mã dễ tối ưu hóa và không phụ thuộc vào máy cụ thể.
Ví dụ: int a = 5; có thể chuyển thành một mã trung gian dạng: T1 = 5; a = T1.

Code Optimizer (Bộ tối ưu mã):
Cải thiện hiệu suất của chương trình bằng cách tối ưu hóa mã trung gian, giảm thiểu tài nguyên hoặc thời gian thực thi.
Ví dụ: Xóa các biến hoặc phép toán không cần thiết, hợp nhất các lệnh thừa.
Code Generator (Bộ sinh mã đích):
Chuyển mã trung gian thành mã máy (machine code) hoặc mã phù hợp với nền tảng đích (target program).
Ví dụ: Chuyển mã trung gian thành mã hợp ngữ hoặc mã nhị phân.
Target Program (Chương trình đích):
Đây là phiên bản cuối cùng của chương trình, có thể thực thi trên máy đích.
Ví dụ: Tệp .exe hoặc .out.

Front-End: Gồm các giai đoạn từ phân tích từ vựng đến sinh mã trung gian (liên quan đến logic và ngữ pháp của ngôn ngữ).
Back-End: Gồm các giai đoạn từ tối ưu hóa mã đến sinh mã đích (liên quan đến hiệu suất và nền tảng phần cứng).

- Like a **word extractor**
  in $\Rightarrow$ i n $\Rightarrow$ in
- Like a **spell checker**
  I ogog to socholsochol
- Like a **classification**



int x = 5; sẽ được phân loại thành:
int: từ khóa (keyword).
x: tên biến (identifier).
=: toán tử gán (operator).
5: hằng số (literal).
;: dấu chấm phẩy (delimiter).

Lexical Analysis Roles

- Identify **lexemes**: substrings of the source program that belong to a grammar unit
- Return **tokens**: a lexical category of lexemes
- Ignore **spaces** such as blank, newline, tab
- Record the **position** of tokens that are used in next phases

1. Identify lexemes
Lexemes: Là các chuỗi con (substrings) trong chương trình nguồn thuộc về một đơn vị ngữ pháp (grammar unit).
Ví dụ: Trong result = oldsum - value / 100;, các từ như result, oldsum, value, và 100 đều là lexemes vì chúng có vai trò nhất định trong cú pháp của ngôn ngữ.
2. Return tokens
Tokens: Là phân loại từ vựng của các lexemes, mỗi token thuộc một danh mục từ vựng (lexical category).
Ví dụ:
result được gắn nhãn (token) là IDENT vì nó là một tên định danh (identifier).
= được gắn nhãn ASSIGN_OP vì nó là toán tử gán.
3. Ignore spaces
Lexical analyzer sẽ bỏ qua khoảng trắng (như dấu cách, dòng mới, hoặc tab) vì chúng không có ý nghĩa trong ngôn ngữ lập trình (trừ một số trường hợp đặc biệt như Python, nơi khoảng trắng ảnh hưởng cú pháp).
Ví dụ: result = oldsum sẽ được phân tích giống như result=oldsum.
4. Record the position of tokens
Ghi lại vị trí của các token (ví dụ: dòng, cột) để hỗ trợ các giai đoạn phân tích tiếp theo hoặc giúp xác định vị trí lỗi.
Ví dụ: Nếu gặp lỗi, trình biên dịch có thể báo:
"Lỗi: Ký tự '=' không hợp lệ tại dòng 1, cột 7."

Example on Lexeme and Token

r	e	s	u	l	t	’ ’	=	’ ’	o

ldsum - value / 100;

Lexemes	**Kind of Tokens**
result	IDENT
=	ASSIGN_OP
oldsum	IDENT
-	SUBSTRACT_OP
value	IDENT
/	DIV_OP
100	INT_LIT
;	SEMICOLON

How to build a lexical analyzer?

- How to build a lexical analysis for English?
  - 65000 words
  - Simply build a dictionary: {(I,pronoun);(We,pronoun);(am,verb);...}
  - Extract, search, compare
- But for a programming language?
  - How many words?
    - Identifiers: abc, cab, Abc, aBc, cAb, ...
    - Integers: 1, 10, 120, 20, 210, ...
    - ...
  - Too many words to build a dictionary, so how?
    - **Apply rules for each kind of word (token)**

1. How to build a lexical analyzer?
Một Lexical Analyzer được thiết kế để phân tích và nhận diện các đơn vị cơ bản (tokens) trong một chuỗi đầu vào (chương trình nguồn).
Cách xây dựng cụ thể sẽ phụ thuộc vào loại ngôn ngữ được phân tích (ngôn ngữ tự nhiên hoặc ngôn ngữ lập trình).
2. Lexical analysis for English
Với ngôn ngữ tự nhiên như tiếng Anh, ta có thể dễ dàng xây dựng một từ điển với tất cả các từ hợp lệ.
Ví dụ:
Tiếng Anh có khoảng 65.000 từ (từ vựng cơ bản).
Từ điển mẫu:
{
(I, pronoun),
(We, pronoun),
(am, verb),
(is, verb),
(book, noun),
(run, verb)
}
Quy trình:
Extract (Trích xuất): Tách từng từ ra từ chuỗi đầu vào.
Search (Tìm kiếm): So sánh từ với từ điển.
Compare (Đối chiếu): Xác định loại từ (pronoun, verb, noun, ...).
Lý do cách này hiệu quả với ngôn ngữ tự nhiên:
Số lượng từ vựng cố định (65.000 từ).
Dễ dàng tra cứu từ điển để phân loại từ.

3. Lexical analysis for a programming language
Với ngôn ngữ lập trình, việc xây dựng từ điển giống như tiếng Anh là không khả thi, bởi vì:

Identifiers (Tên định danh): Không giới hạn. Ví dụ: abc, cab, Abc, aBc, ... có vô hạn tổ hợp.
Integers (Số nguyên): Không thể liệt kê hết các số nguyên hợp lệ (1, 10, 120, ...).
Các ký tự khác: Ví dụ: =, +, -, *, int, float, ...
=> Quá nhiều từ, không thể lưu trữ trong một từ điển tĩnh.
4. Giải pháp: Apply rules for each kind of word (token)
Thay vì dùng từ điển, ta xây dựng các quy tắc (rules) để nhận diện từng loại token.

Ví dụ các quy tắc:
Identifiers (Tên định danh):
Quy tắc:
Bắt đầu bằng chữ cái hoặc _.
Có thể chứa chữ cái, chữ số, hoặc _.
Ví dụ hợp lệ: abc, _var, myVar123.
Ví dụ không hợp lệ: 123abc, #var.
Integers (Số nguyên):
Quy tắc:
Chỉ chứa các ký tự số (0-9).
Ví dụ hợp lệ: 1, 10, 120.
Keywords (Từ khóa):
Quy tắc:
Kiểm tra xem từ có nằm trong danh sách từ khóa của ngôn ngữ không.
Ví dụ: if, while, int, return.
Operators (Toán tử):
Quy tắc:
Xác định các ký hiệu như +, -, *, /, =.
So khớp với danh sách ký hiệu hợp lệ.
Delimiters (Ký tự phân cách):
Quy tắc:
Nhận diện các ký tự như ;, ,, {, }.
Quy trình tổng quát:
Từng bước kiểm tra chuỗi đầu vào (input string).
Áp dụng các quy tắc để phân loại thành token.
Trả về token cùng với thông tin vị trí của nó.

Ngôn ngữ lập trình: Không thể dùng từ điển vì số lượng từ quá lớn. Thay vào đó, áp dụng các quy tắc nhận diện dựa trên đặc trưng của từng loại token.

Rule Representations

1. Finite Automata (FA) – Máy tự động hữu hạn

Là mô hình toán học dùng để biểu diễn các quy tắc nhận dạng (như token trong lexical analysis).

FA gồm:

Tập trạng thái: Các điểm kiểm tra trong quá trình phân tích (states).

Bảng chuyển trạng thái: Quy định cách chuyển đổi giữa các trạng thái dựa trên đầu vào (transitions).

Trạng thái bắt đầu: Điểm xuất phát (start state).

Trạng thái kết thúc: Xác định đầu vào được chấp nhận (accepting states).

- Finite Automata
  - Deterministic Finite Automata
  - Nondeterministic Finite Automata
- Regular Expressions

Ví dụ:

Một FA nhận diện từ khóa int:

Trạng thái bắt đầu: q0.

Nhận chữ i, chuyển sang trạng thái q1.

Nhận chữ n, chuyển sang q2.

Nhận chữ t, chuyển sang trạng thái kết thúc q3.

Nếu nhận bất kỳ ký tự nào khác, FA quay lại q0 hoặc dừng.

2. Deterministic Finite Automata (DFA) – Máy tự động hữu hạn xác định

DFA là một dạng đặc biệt của FA, nơi tại mỗi trạng thái, đầu vào chỉ dẫn đến một trạng thái duy nhất.

Đặc điểm:

Không có sự lựa chọn (deterministic).

Mỗi đầu vào luôn dẫn đến một trạng thái xác định.

Ví dụ DFA nhận diện chữ số:

Tập trạng thái: {q0, q1}.

Trạng thái bắt đầu: q0.

Trạng thái kết thúc: q1.

Chuyển trạng thái:

Ở q0, nhận số (0-9), chuyển sang q1.

Ở q1, nhận tiếp số (0-9), giữ nguyên ở q1.

Nhận ký tự khác, dừng.

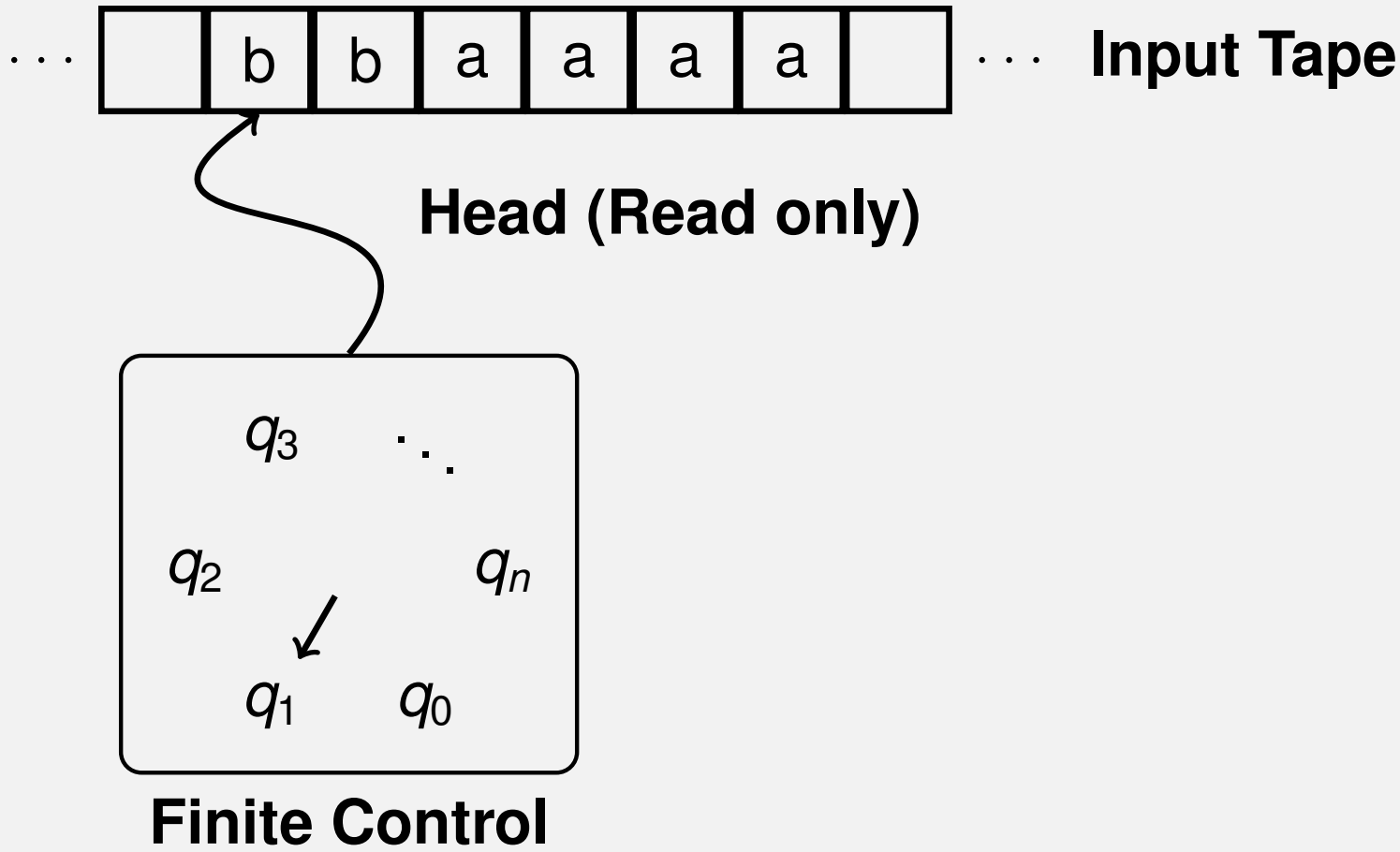
Ưu điểm của DFA: Nhanh và dễ triển khai, vì mỗi đầu vào chỉ cần kiểm tra duy nhất một lần.

3. Nondeterministic Finite Automata (NFA) – Máy tự động hữu hạn không xác định
NFA cho phép nhiều lựa chọn trạng thái cho cùng một đầu vào.
Đặc điểm:
Tại một trạng thái, đầu vào có thể dẫn đến nhiều trạng thái khác nhau.
Có thể có trạng thái chuyển tiếp ε (chuyển mà không cần đầu vào).
Ví dụ NFA nhận diện int:
Tại trạng thái q0, nhận chữ i:
Có thể chuyển sang q1 hoặc q2.
Tại trạng thái q1, nhận chữ n, chuyển sang q2.
Tại trạng thái q2, nhận chữ t, chuyển sang trạng thái kết thúc q3.
Ưu điểm của NFA:

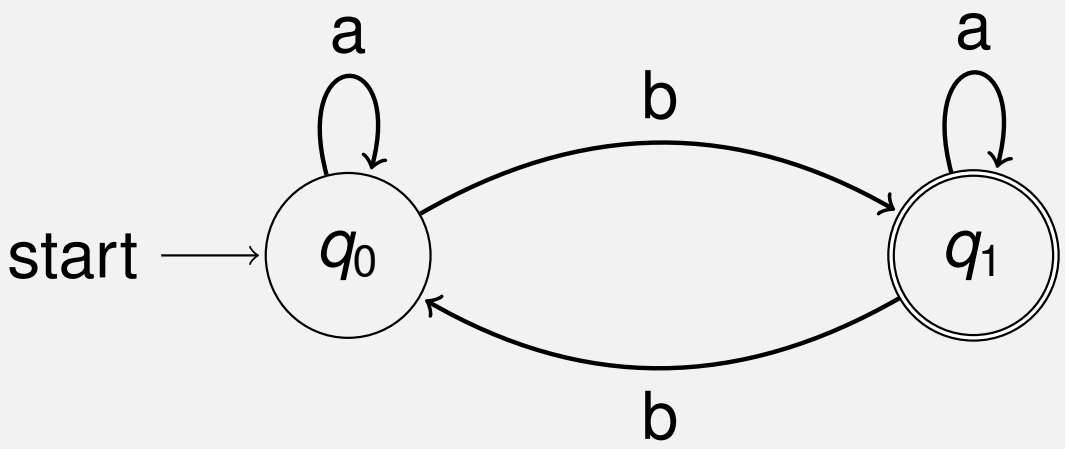
Biểu diễn ngắn gọn, dễ thiết kế.
Nhược điểm:
Phức tạp hơn khi triển khai vì cần xử lý nhiều lựa chọn.

4. Regular Expressions (Regex) – Biểu thức chính quy

Là cách viết ngắn gọn và mạnh mẽ để biểu diễn các mẫu chuỗi (patterns).
Mối quan hệ: Regex có thể được chuyển đổi thành NFA, sau đó NFA có thể được chuyển thành DFA.
Cú pháp cơ bản:
a^* → Chuỗi gồm 0 hoặc nhiều ký tự a.
ab → Chuỗi có ký tự a nối tiếp b.
a|b → a hoặc b.
[0-9] → Ký tự số từ 0 đến 9.
Ví dụ: Regex nhận diện số nguyên:
Regex: [0-9]+
[0-9]: Ký tự số từ 0 đến 9.
+: Lặp lại một hoặc nhiều lần.
Regex sẽ khớp với các chuỗi như 1, 42, 12345.


b
b
a
a
a
a
Input Tape
Head (Read only)
q_3
q_2
q_n
q_1
q_0
Finite Control

State Diagram



Input: abaabb

Current state	Read	New State
q_0	a	q_0
q_0	b	q_1
q_1	a	q_1
q_1	a	q_1
q_1	b	q_0
q_0	b	q_1

Definition

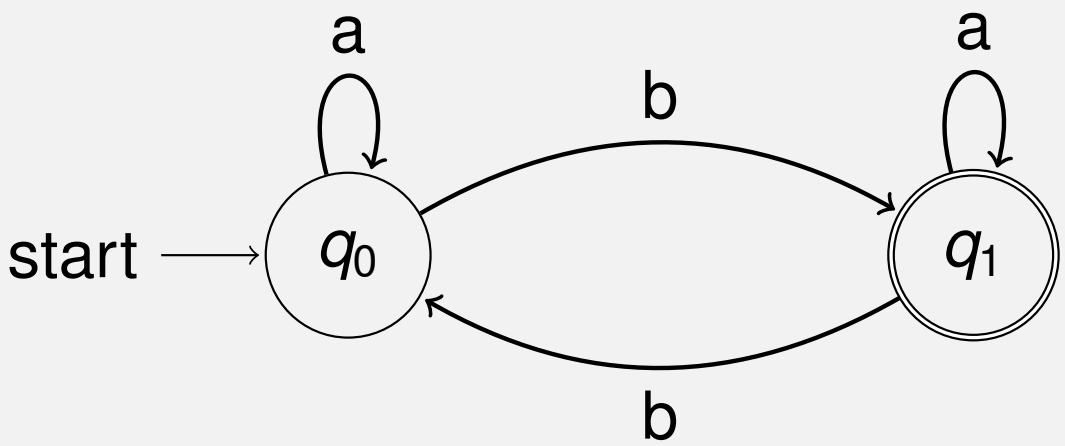
Deterministic Finite Automaton(DFA) is a 5-tuple
M =(K,Σ,δ,s,F) where

- K = a finite set of state
- Σ = alphabet
- s $\in$ K = the initial state
- F $\subseteq$ K = the set of final states
- δ = a transition function from K $\times\Sigma$ to K

Example

M =(K,Σ,δ,s,F)
where K = {q_0, q_1} Σ = {a,b} s=q_0 F={q_1}
and δ

K	Σ	$\delta(K, \Sigma)$
q_0	a	q_0
q_0	b	q_1
q_1	a	q_1
q_1	b	q_0

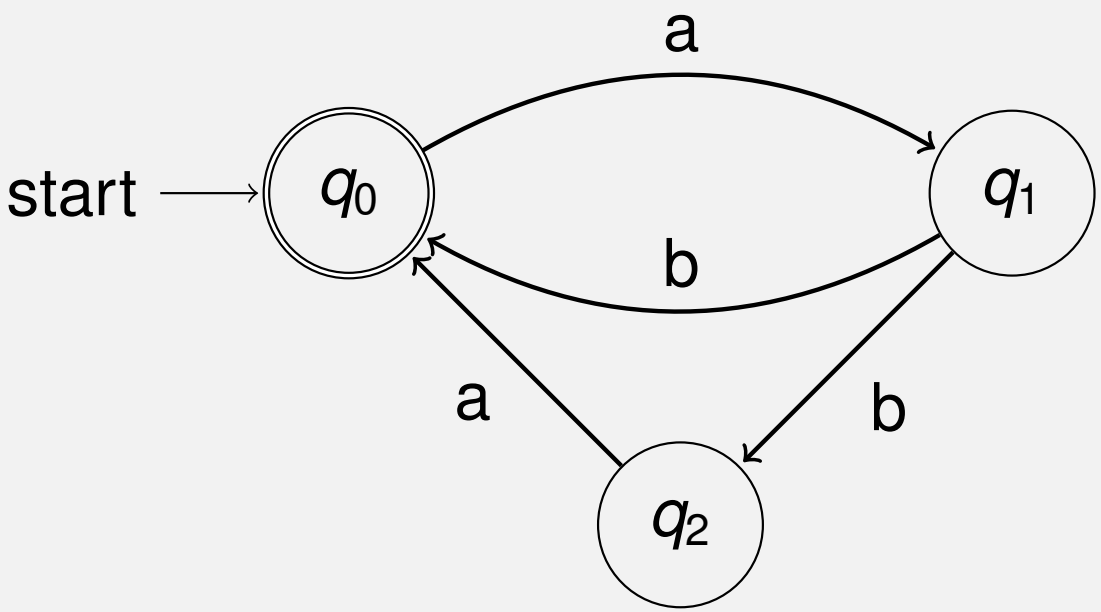


Nondeterministic Finite Automata

- Permit several possible “next states” for a given combination of current state and input symbol
- Accept the empty string $\in$ in state diagram
- Help simplifying the description of automata
- Every NFA is equivalent to a DFA

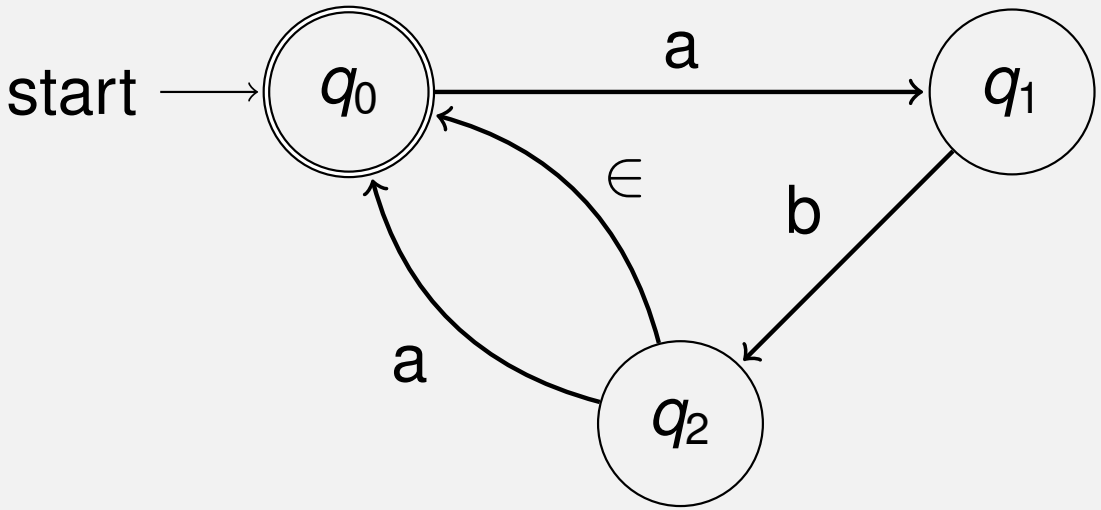
Example

Language L = ({ab} ∪ {aba})*



Example

Language L = ({ab} ∪ {aba})*



Regular Expression (regex)

- Describe regular sets of strings
- Symbols other than ( ) | * stand for themselves
- Use $\in$ for an empty string
- Concatenation α β = First part matches α, second part β
- Union α | β = Match α or β
- Kleene star α* = 0 or more matches of α
- Use ( ) for grouping

Example

RE		Language
0	=>	{ 0 }
01	=>	{ 01 }
0 | 1	=>	{0,1}
0(0|1)	=>	{00,01}
(0|1)(0|1)	=>	{00,01,10,11}
0*	=>	{∈,0,00,000,0000,...}
(0|1)*	=>	{∈,0,1,00,01,10,11,000,001,...}

Example

(i|I)(f|F)

Keyword **if** of language Pascal

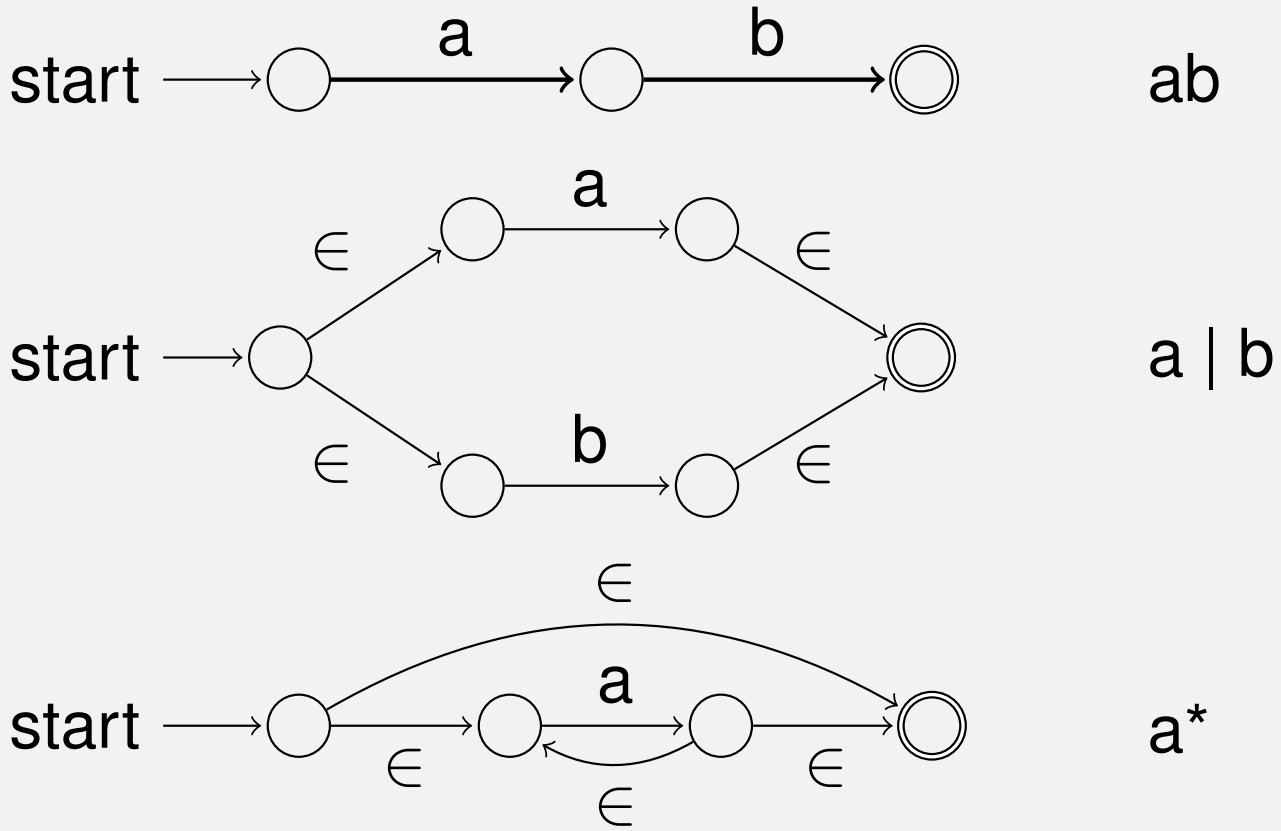
- if
- IF
- If
- iF

E(0|1|2|3|4|5|6|7|8|9)*

An E followed by a (possibly empty) sequence of digits

- E123
- E9
- E

Regular Expression and Finite Automata



Convenience Notation

- α+ = one or more (i.e. $\alpha\alpha*$)
- α? = 0 or 1 (i.e. ($\alpha| \in$))
- [xyz]= x|y|z
- [x-y]= all characters from x to y, e.g. [0-9] = all ASCII digits
- [^x-y]= all characters other than [x-y]
- . matches any character

Example

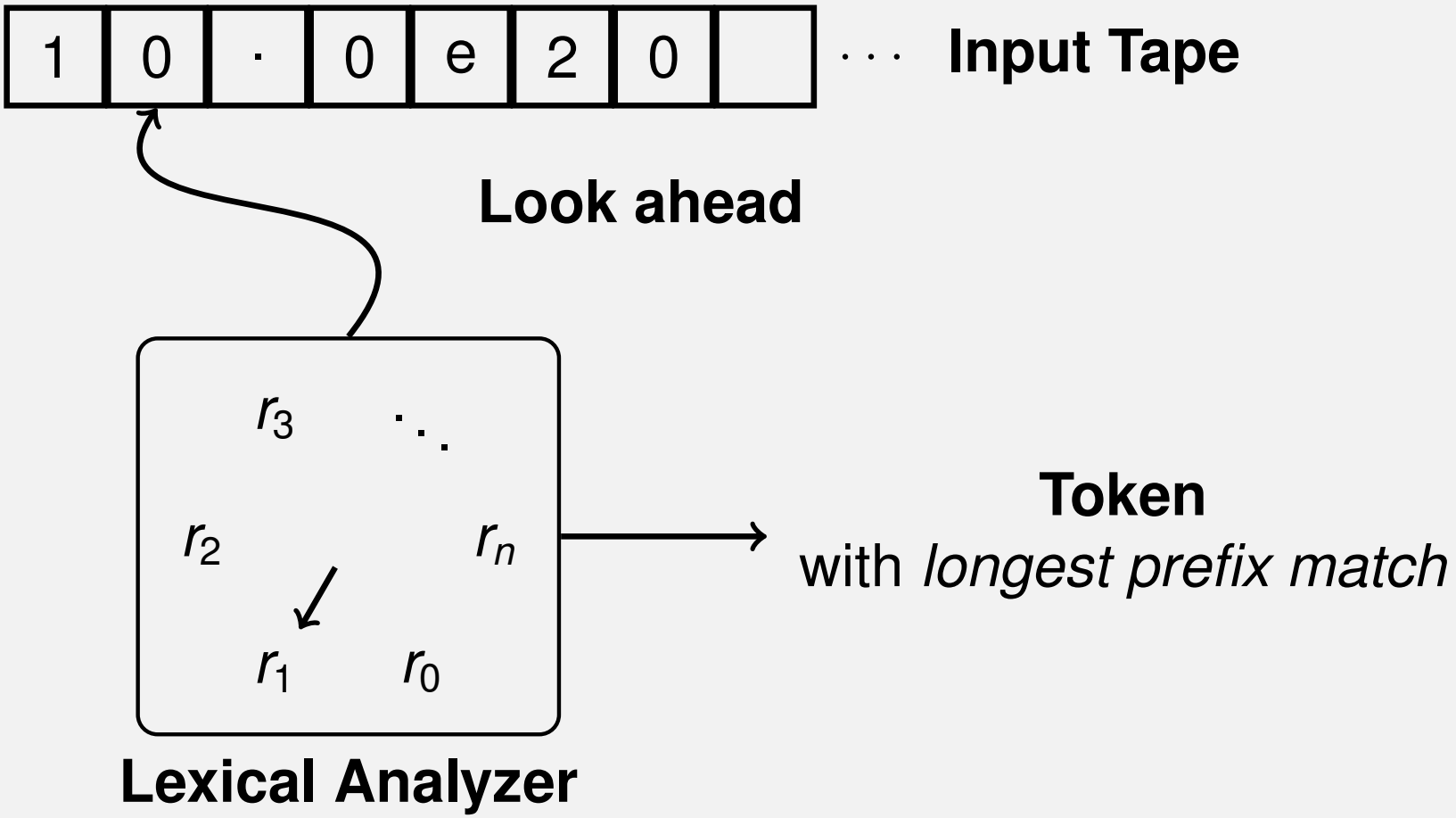
(0|1|2|3|4)	=>	[0-4]
(a|g|h|m)	=>	[aghm]
(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*	=>	[0-9]+
(E|e)(+|-|∈)(0|1|2|3|4|5|6|7|8|9)+	=>	[Ee][+-]?[0-9]+

ANTLR [1]

- ANother Tool for Language Recognition
- Terence Parr, Professor of CS at the Uni. San Francisco
- powerful parser/lexer generator

```
/**
 * Filename: Hello.g4
 */
lexer grammar Hello;

// match any digits
INT: [0-9]+;

// Hexadecimal number
HEX: 0[Xx][0-9A-Fa-f]+;

// match lower-case identifiers
ID : [a-z]+ ;

// skip spaces, tabs, newlines
WS : [ \t\r\n]+ -> skip ;
```

Lexical Analyzer



Summary

- A lexical analyzer is a pattern matcher that isolates small-scale parts of a program
- Lexical rules are represented by Regular expressions or Finite Automata.
- How to write a lexical analyzer (lexer) in ANTLR

References I

[1] ANTLR, http:antlr.org, 19 08 2016.