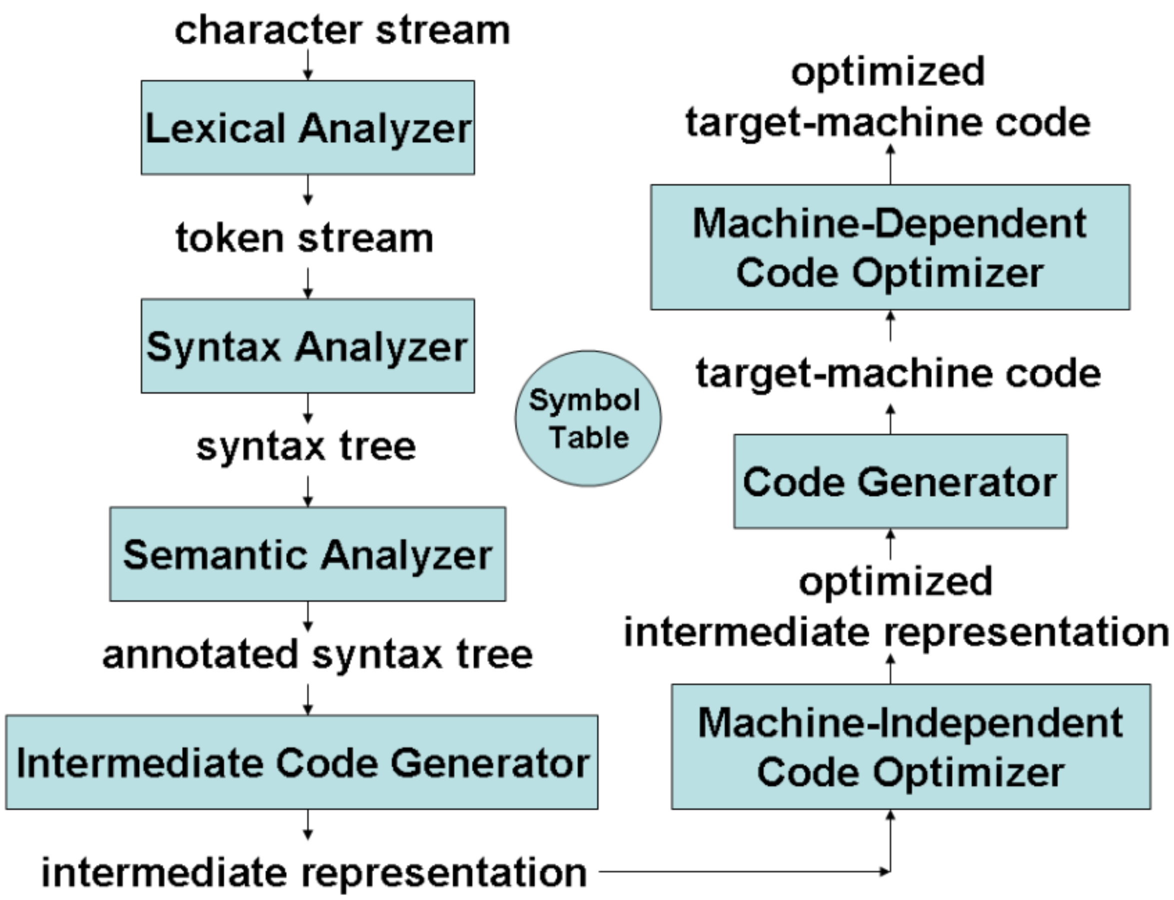
Introduction to Lexical Analysis

Outline

- What is lexical analysis?
- Tokens, patterns, and lexemes
- Difficulties in lexical analysis
- **Specification of tokens** - regular expressions and regular definitions
- **Recognition of tokens** - finite automata and transition diagrams
- **LEX** - A Lexical Analyzer Generator

Compiler Overview



What is Lexical Analysis?

- **Input**: high level language program, such as a ‘C’ program in the form of a sequence of characters
- **Output**: sequence of **tokens** that is sent to the parser for syntax analysis

- Strips off blanks, tabs, newlines, and comments from the source program
- Performs some preprocessor functions such as #define and #include in ‘C’
- Keeps track of line numbers and associates error messages from various parts of a compiler with line numbers

Why to separate lexical analysis from syntax analysis?

- **Simplification of design** - software engineering reason

- **More compact and faster parser**
  - Comments, blanks, etc., need not be handled by the parser
  - A parser is more complicated than a lexical analyzer (shrinking the grammar makes the parser faster)
    - No rules for comments, etc., are needed in the parser

- **LA based on finite automata are more efficient to implement** than pushdown automata used for parsing (due to stack)

Tokens, Patterns and Lexemes

- **Running example**: **float abs_t = -270;**
- **Token** (also called ***word***)
  - A sequence of characters which logically belong together
  - float, identifier, equal, minus, intnum, semicolon
  - Tokens are treated as terminal symbols of the grammar specifying the source language
- **Pattern**
  - The set of strings for which the same token is produced
  - The pattern is said to match each string in the set
  - **float**, **l(l+d+_)***, **=**, **-**, **d**$^+$, **;**
- **Lexeme**
  - The sequence of characters matched by a pattern to form the corresponding token
  - "float", "abs_t", "=", "-", "270", ";"

Tokens

- **TOKENS**: Keywords, operators, identifiers (names), constants, literal strings, punctuation symbols such as parentheses, brackets, commas, semicolons, and colons, etc.

- A ***unique integer*** representing the token is passed by **LA** to **the parser**

- **Attributes for tokens** (apart from the integer representing the token)
  - ***identifier***: the lexeme of the token, or a pointer into the symbol table where the lexeme is stored by the LA
  - ***intnum***: the value of the integer (similarly for floatnum, etc.)
  - ***string***: the string itself
  - The exact set of attributes are dependent on the compiler designer

Lexical analysis: Difficulties

- Certain languages do not have any reserved words, e.g., **while**, **do**, **if**, **else**, etc., are reserved in **’C’**, but not in **PL/1**
- In FORTRAN, some keywords are context-dependent
  - In the statement, **DO 10 I = 10.86**, **DO10I** is an identifier, and **DO** is not a keyword
  - But in the statement, **DO 10 I = 10, 86**, **DO** is a keyword
- Such features require substantial **look ahead** for resolution
- **Blanks** are not significant in FORTRAN and can appear in the midst of identifiers, but not so in ’C’

Specifying and recognizing tokens

- **Regular definitions**, a mechanism based on *regular expressions* are very popular for **specification of tokens**
  - Has been implemented in the lexical analyzer generator tool, **LEX**
  - We study regular expressions first, and then, token specification using **LEX**

- **Transition diagrams, a variant of finite state automata, are used to implement regular definitions** and to **recognize tokens**
  - Transition diagrams are usually used to model LA before translating them to programs by hand
  - **LEX:** automatically generates optimized FSA from regular definitions
  - We study FSA and their generation from regular expressions in order to understand transition diagrams and LEX

Languages (Some definitions)

- **Symbol**: An abstract entity
  - Examples: letters and digits
- **String**: A finite sequence of juxtaposed symbols
  - ***abcb***, ***caba*** are strings over the symbols **a,b,** and **c**
  - **|w|** is the length of the string **w**, and is the #symbols in it
- ε is the empty string and is of length 0
- **Alphabet**: A finite set of symbols
- **Language**: A set of strings of symbols from some alphabet

  - Φ and $\{\varepsilon\}$ are languages
- The **set of palindromes** over {0,1} is an **infinite** language
- The set of strings, {01, 10, 111} over {0,1} is a **finite** language
- If Σ is an alphabet, Σ* is the set of all strings over Σ

Examples of Languages

Let $\Sigma = \{a, b, c\}$

- $L_1 = \{a^m b^n | m, n \geq 0\}$ is regular
- $L_2 = \{a^n b^n | n \geq 0\}$ is context-free but not regular
- $L_3 = \{a^n b^n c^n | n \geq 0\}$ is context-sensitive but neither regular nor context-free

Automata

- **Automata are machines that accept languages**
  - **Finite State Automata** accept **RLs** (corresponding to **REs**)
  - **Pushdown Automata** accept **CFLs** (corresponding to **CFGs**)
  - Linear Bounded Automata accept CSLs (corresponding to CSGs)
  - Turing Machines accept type-0 languages (corresponding to type-0 grammars)

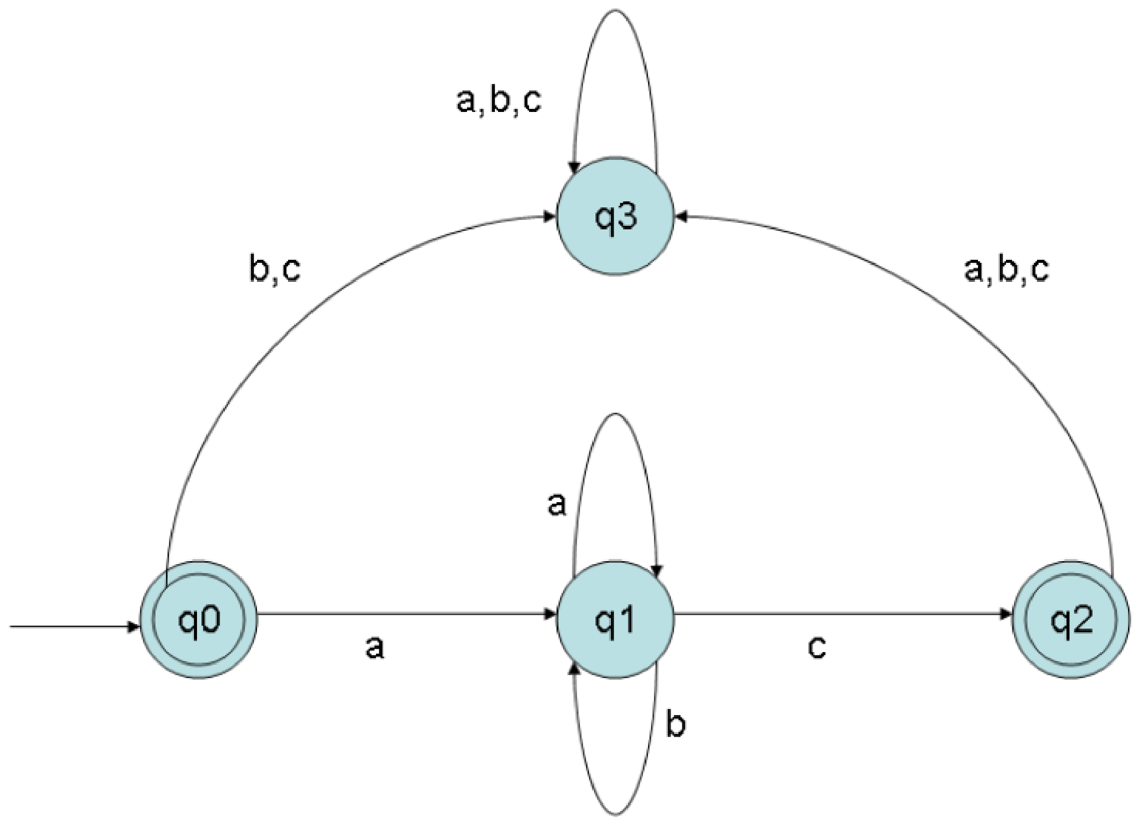
Finite State Automaton

- An FSA is an acceptor or recognizer of regular languages

- An FSA is a 5-tuple, **(Q, Σ, δ, q0, F)**, where
  - Q is a finite set of states
  - Σ is the input alphabet
  - δ is the transition function, $\boldsymbol{\delta : Q \times \Sigma \rightarrow Q}$

    That is, **δ(q, a)** is a state for each state q and input symbol a
  - q0 is the start state
  - F is the set of final or accepting states
- In one **move from some state q**, an FSA reads an input symbol, changes the state based on δ, and gets ready to read the next input symbol
- An FSA **accepts its input string**, if starting from **q0**, it consumes the entire input string, and **reaches a final state**
- If the last state reached is not a final state, then the input string is rejected

Example 1: FSA

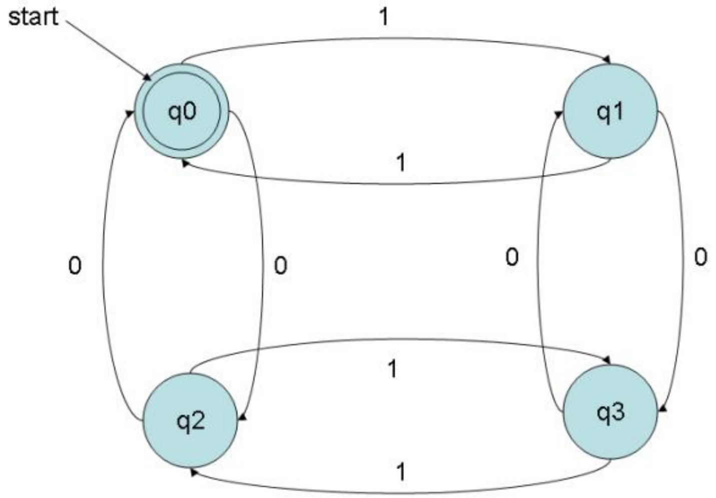


Example 1: FSA

- $Q = \{q0, q1, q2, q3\}$
- $\Sigma = \{a, b, c\}$
- q0 is the start state and $F = \{q0, q2\}$
- The transition function δ is defined by the table below
- Language accepted by the automaton?

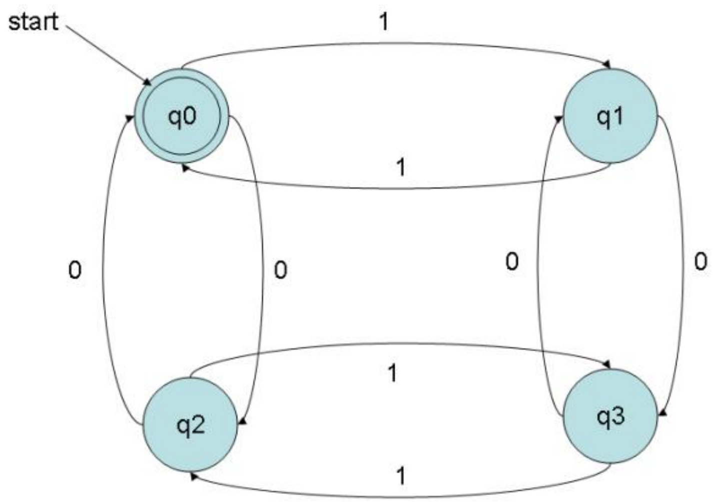
state	symbol		
	a	b	c
q_0	q_1	q_3	q_3
q_1	q_1	q_1	q_2
q_2	q_3	q_3	q_3
q_3	q_3	q_3	q_3

Example 2: FSA



- Q = {q0, q1, q2, q3}, q0 is the start state
- F = {q0}, δ is as in the figure
- Language accepted?

Example 2: FSA

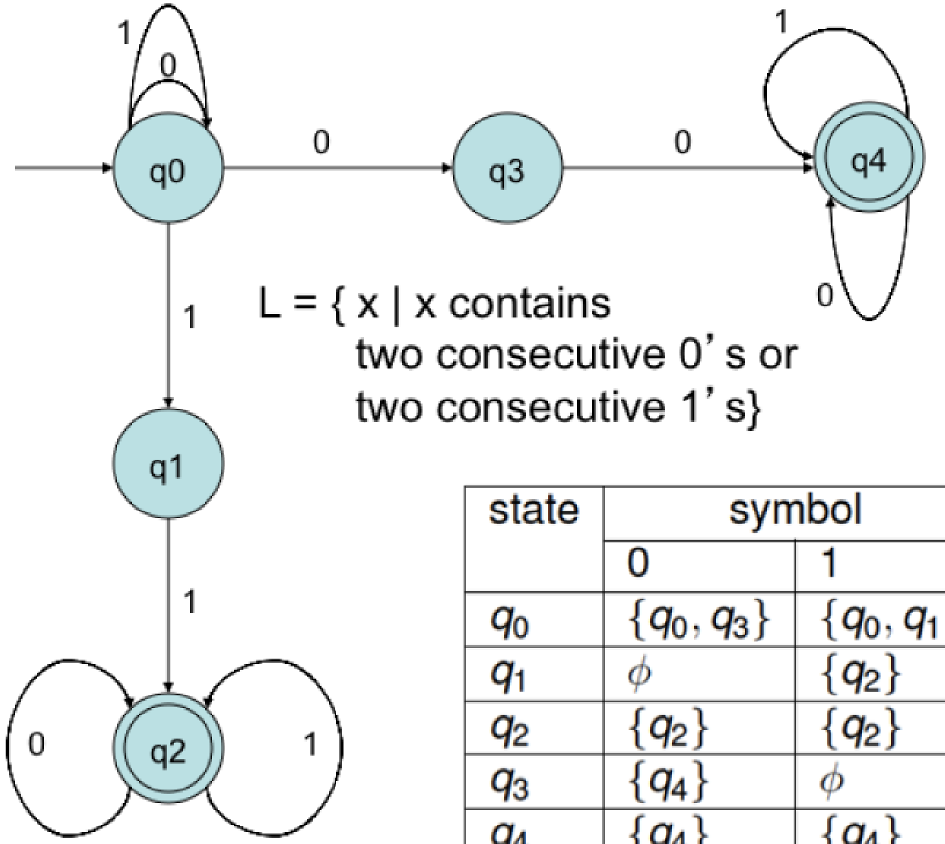


- Q = {q0, q1, q2, q3}, q0 is the start state
- F = {q0}, δ is as in the figure
- Language accepted is the set of all strings of 0's and 1's, in which the no. of 0's and the no. of 1's are even numbers

Non-deterministic finite state automata

- NFAs are FSA which allow 0, 1, or more transitions from a state on a given input symbol
- An NFA is a 5-tuple as before, but the transition function δ is different
  - $\delta(q, a)$ = the set of all states p, such that there is a transition labelled a from q to p
- A string is accepted by an NFA if there exists a sequence of transitions corresponding to the string, that leads from the start state to some final state
- Every NFA can be converted to an equivalent deterministic FA (DFA), that accepts the same language as the NFA

Example: NFA



L = { x | x contains
two consecutive 0's or
two consecutive 1's}

state	symbol	
	0	1
q_0	$\{q_0, q_3\}$	$\{q_0, q_1\}$
q_1	ϕ	$\{q_2\}$
q_2	$\{q_2\}$	$\{q_2\}$
q_3	$\{q_4\}$	ϕ
q_4	$\{q_4\}$	$\{q_4\}$

Regular Expressions

- Let Σ be an alphabet. The REs over Σ and the languages they denote (or generate) are defined
  1. ϕ is an RE. L(ϕ) = ϕ
  2. ε is an RE. L(ε) = {ε}
  3. For each a ∈ Σ, a is an RE. L(a) = {a}

  4. If **r** and **s** are REs denoting the languages **R** and **S**, respectively
     1. (**rs**) is an RE (*denotes **concatenation***)
     2. (**r + s**) is an RE, (denotes either *r* **or** *s*)
     3. (**r∗**) is an RE (denotes zero or more occurrences of *r*)

Regular Languages

- The language accepted by an FSA is the set of all strings accepted by it (regular language).

- It can be shown that for every regular expression, an FSA can be constructed and vice-versa

Examples of Regular Expressions

- Give RE for the following:
  - Set of all strings of 0’s and 1’s
  - Set of all strings of 0’s and 1’s with at least two consecutive 0’s
  - Set of all strings of 0’s and 1’s beginning with 1 and not having two consecutive 0’s

Examples of Regular Expressions

1. L = set of all strings of 0's and 1's
   $r = (0+1)^*$
   - How to generate the string 101 ?
   - $(0+1)^* \Rightarrow^4 (0+1)(0+1)(0+1)\epsilon \Rightarrow^4 101$
2. L = set of all strings of 0's and 1's, with at least two consecutive 0's
   $r = (0+1)^*00(0+1)^*$
3. $L = \{w \in \{0,1\}^* \mid w$ has two or three occurrences of 1, the first and second of which are not consecutive$\}$
   $r = 0^*10^*010^*(10^* + \epsilon)$
4. $r = (1+10)^*$
   L = set of all strings of 0's and 1's, beginning with 1 and not having two consecutive 0's
5. $r = (0+1)^*011$
   L = set of all strings of 0's and 1's ending in 011

Regular Definitions

A regular definition is a sequence of "*equations*" of the form

d1 = r1; **d2 = r2**; ... ; **dn = rn**, where each **di** is a distinct name, and each **ri** is a regular expression over the symbols **Σ ∪ {d1, d2, ..., di−1}**

Example (Identifiers and Integers)

Let Σ = {a, b, c, d, e, 0, 1, 2, 3, 4}

letter = a + b + c + d + e;

digit = 0 + 1 + 2 + 3 + 4;

identifier = letter(letter + digit)*;

number = digit digit*