CS 4203 Compiler Theory

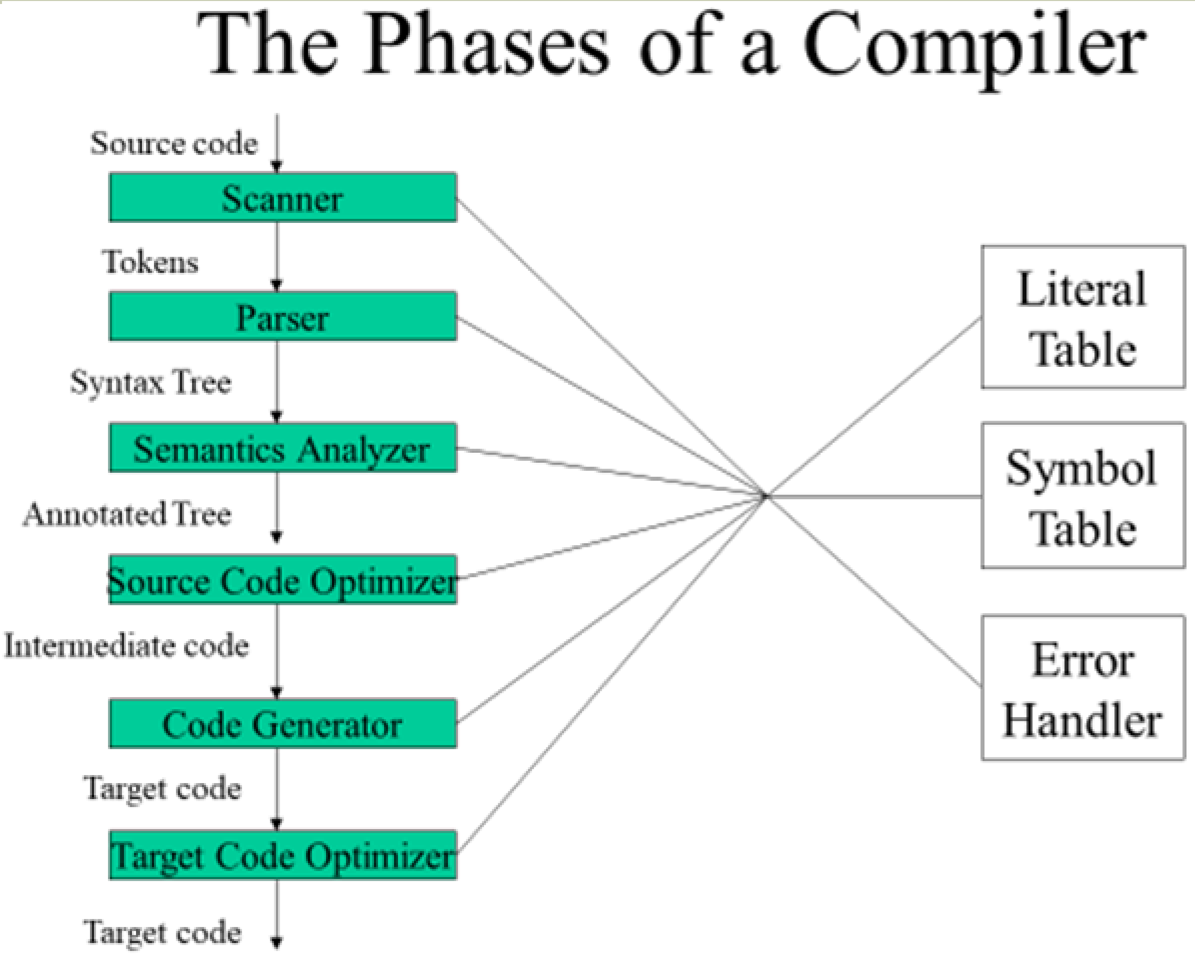
Lecture #2

Scanning (Lexical Analysis)-Part1

Outline

1. Remember From the Previous Lecture that …
2. Lexical Analysis
3. The Scanning Process
4. Regular Expression
5. Extensions to Regular Expression
6. Regular Expressions for Programming Language Tokens
7. FINITE AUTOMATA
8. Deterministic Finite Automata

1. Remember From the Previous Lecture that …



1. Remember From the Previous Lecture that …

The Role of Lexical Analyzer

- Lexical analyzer is the first phase of a compiler.
- Its main task is to read input characters and produce as output a sequence of tokens that parser uses for syntax analysis.

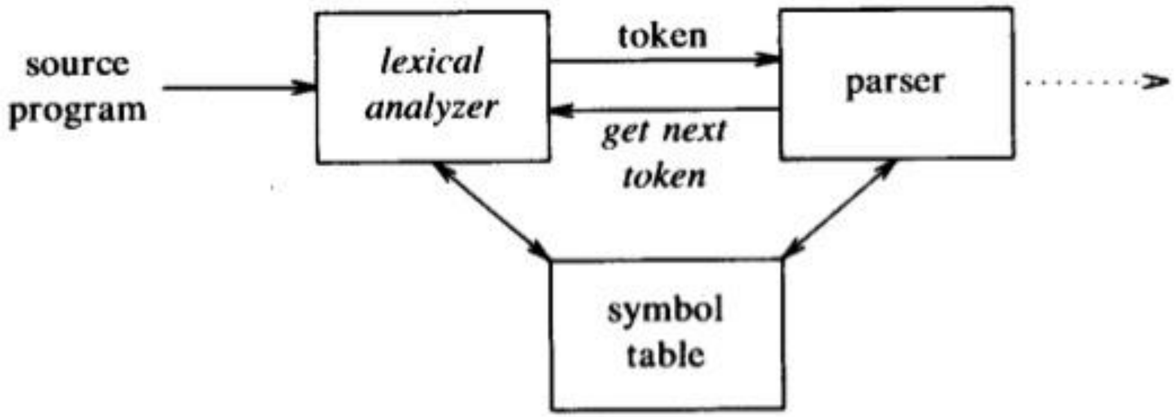
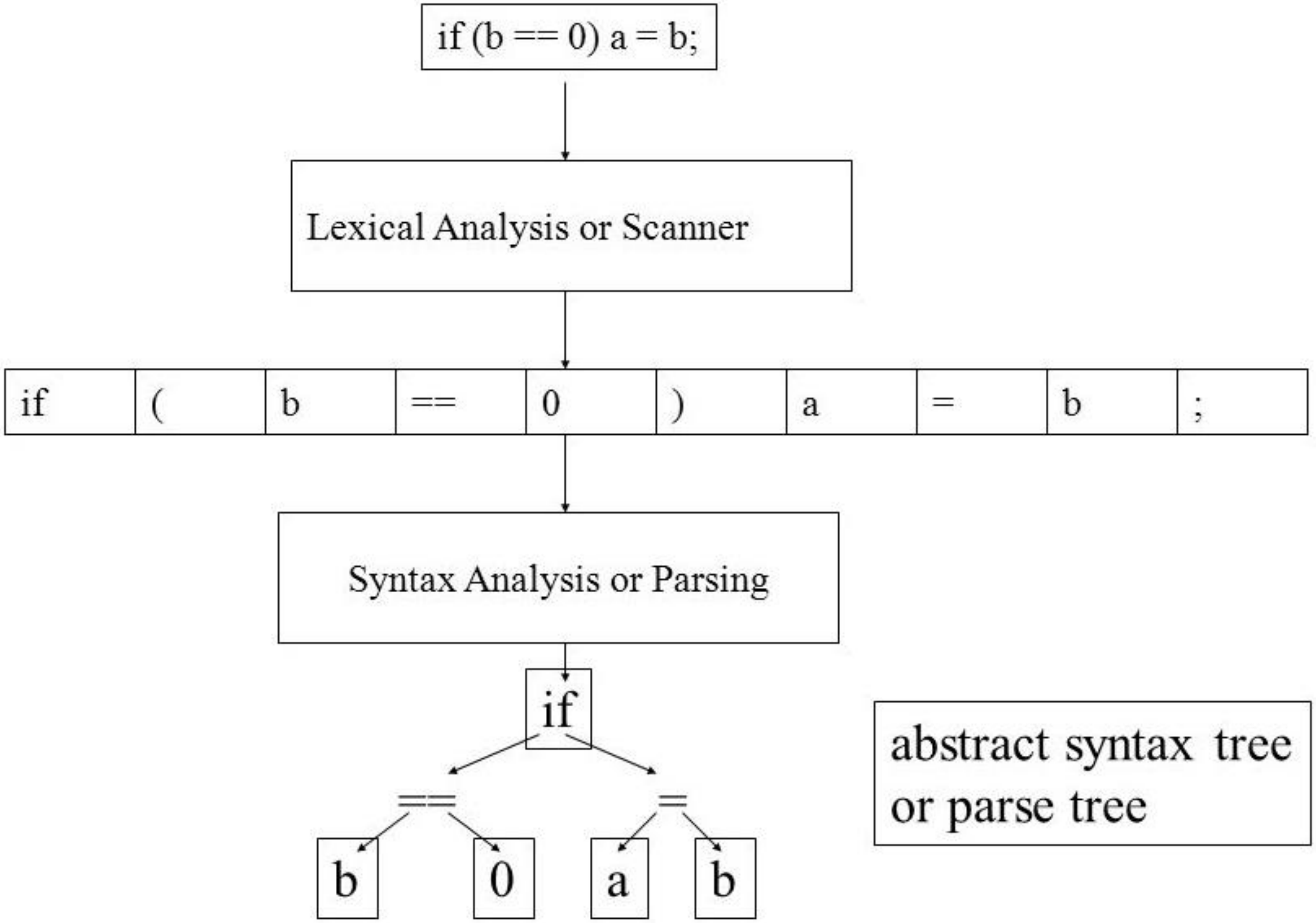


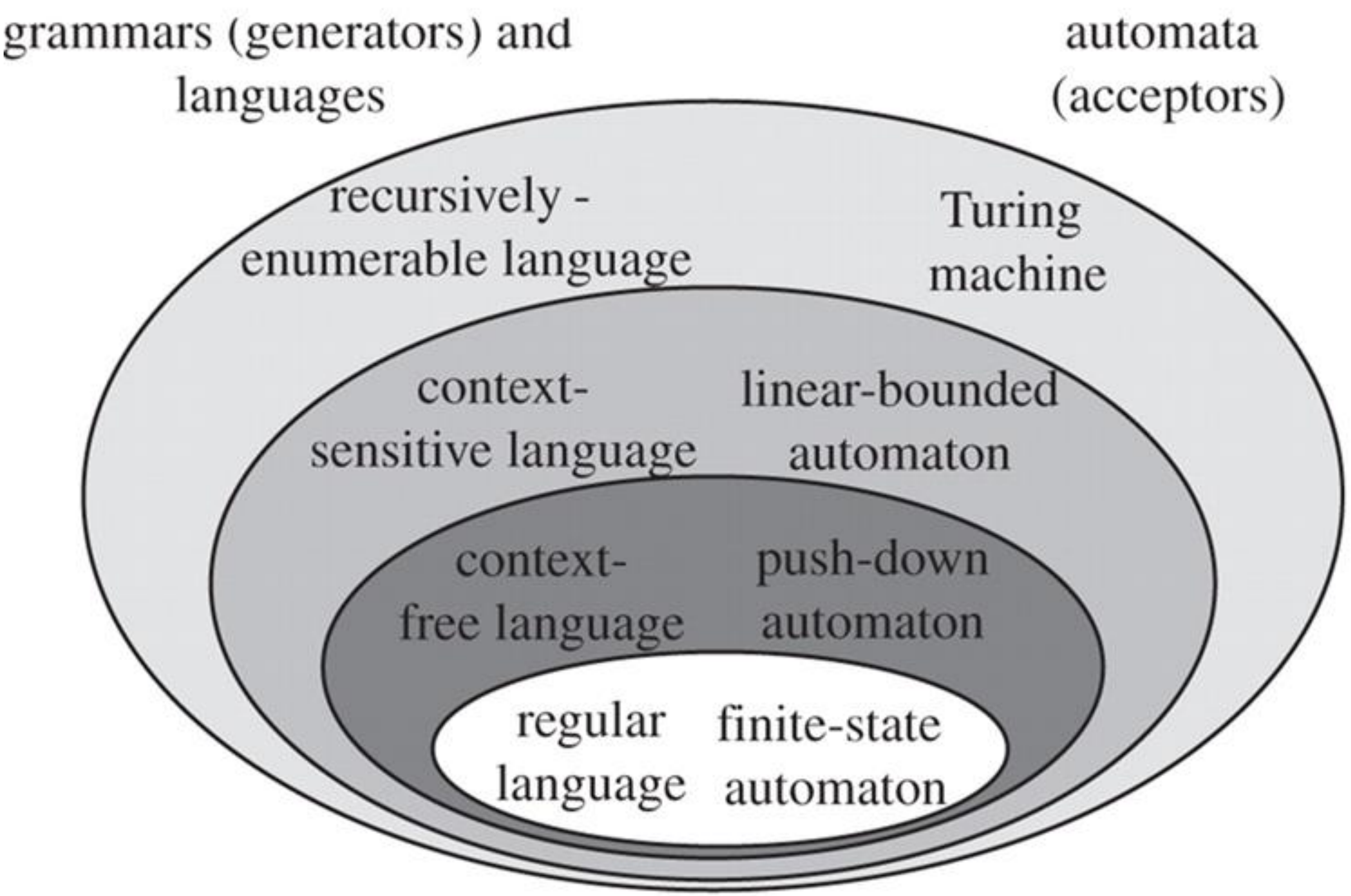
Fig. 3.1. Interaction of lexical analyzer with parser.

Winter 2007

1. Remember From the Previous Lecture that …



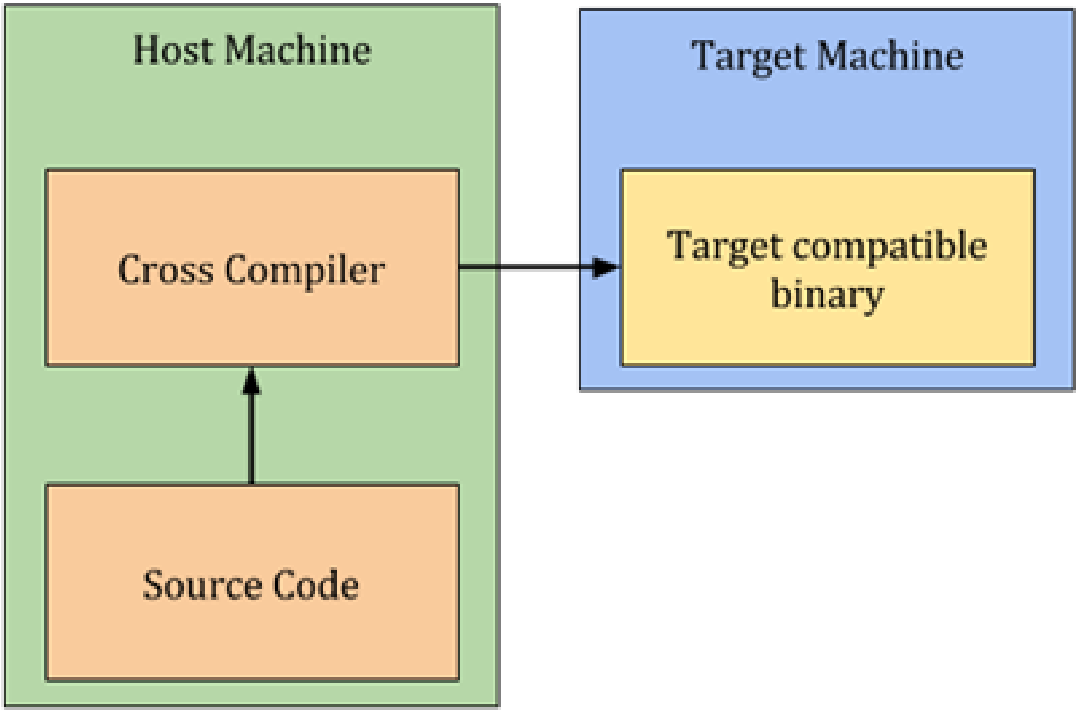
1. Remember From the Previous Lecture that …



the traditional Chomsky hierarchy

1. Remember From the Previous Lecture that …

A cross compiler is a compiler capable of creating executable code for a platform other than the one on which the compiler is running



Cross Compiler operation

1. Remember From the Previous Lecture that …

Error Handling

- Static (or compile-time) errors must be reported by a compiler
  - Generate meaningful error messages and resume compilation after each error
  - Each phase of a compiler needs different kind of error handing
- Exception handling
  - Generate extra code to perform suitable runtime tests to guarantee all such errors to cause an appropriate event during execution.

1. Remember From the Previous Lecture that …

- Lexical error is a sequence of characters that does not match the pattern of any token.
- for example
- int x=10, y=20; char * a; a= &x; x= 1xab;

In this code, 1xab is neither a number nor an identifier. So this code will show the lexical error.

- A syntax error is an error in the source code of a program. Since computer programs must follow strict syntax to compile correctly, any aspects of the code that do not conform to the syntax of the programming language will produce a syntax error.
- for example
- function testFunction()
- {echo "Just testing.";}}

1. Remember From the Previous Lecture that …

- semantic errors are a type of compile errors which are grammatically correct unlike syntax errors
- for example let us say the following declaration: int a=“hello”;
- here the statement is syntactically correct but semantically incorrect due to Type incompatibility.

2. Lexical Analysis

- The job of the lexical analyzer, or scanner, is to transform a stream of characters into a stream of tokens**.**
- The token is the smallest meaningful unit of a language**.**
- To build a scanner, we need**:**
- A language or technique for writing rules that describe tokens**.**
- A language or technique for writing programs that recognize tokens that match the rules
- For these purposes, we will use regular expressions and deterministic finite state automata, respectively

3. The Scanning Process

3.1 The Function of a Scanner

- Reading characters from the source code and form them into logical units called token
- Tokens are logical entities defined as an enumerated type
  - Typedef enum

    {IF, THEN, ELSE, PLUS, MINUS, NUM, ID,…}

    TokenType;

3. The Scanning Process

3.1 The Function of a Scanner

- Secondary tasks:
  - skip white spaces where necessary
  - skip comments
  - correlate error messages with source program (e.g. line no of errors)

3. The Scanning Process

3.2 The Categories of Tokens

- RESERVED WORDS
  - Such as IF and THEN, which represent the strings of characters “if” and “then”
- SPECIAL SYMBOLS
  - Such as PLUS and MINUS, which represent the characters “+” and “-“
- OTHER TOKENS
  - Such as NUM and ID, which represent numbers and identifiers

3. The Scanning Process

3.3 Relationship between Tokens and its String

- The string is called STRING VALUE or LEXEME of token
- Some tokens have only one lexeme, such as reserved words
- A token may have infinitely many lexemes, such as the token ID

3. The Scanning Process

3.4 The scanner operation

- The scanner will operate under the control of the parser, returning the next token from the input on demand.

4. Regular Expressions

4.1 Some Relative Basic Concepts

- Regular expressions
  - represent patterns of strings of characters.
- A regular expression *r*
  - completely defined by the set of strings it matches.
  - The set is called the language of r written as L(r)
- The set elements
  - referred to as symbols
- This set of legal symbols
  - called the alphabet and written as the Greek symbol ∑

4. Regular Expressions

4.1 Some Relative Basic Concepts

- A regular expression *r*
  - contains characters from the alphabet, indicating patterns, such *a* is the character a used as a pattern
- A regular expression *r*
  - may contain special characters called *meta-characters* or meta-symbols
- An *escape character* can be used to turn off the special meaning of a meta-character.
  - Such as backslash and quotes

4. Regular Expressions

4.2 Basic Regular Expressions

- The single characters from alphabet matching themselves
  - a matches the character *a* by writing L(a)={ a }
  - **ε** denotes the empty string, by L(**ε**)={**ε**}
  - {} or **Φ** matches no string at all, by L(**Φ**)={ }

4. Regular Expressions

4.3 Regular Expression Operations

- Choice among alternatives, indicated by the meta-character |
- Concatenation, indicated by juxtaposition
- Repetition or “closure”, indicated by the meta-character *

4. Regular Expressions
4.3 Regular Expression Operations

a) Choice Among Alternatives

- If r and s are regular expressions, then r|s is a regular expression which matches any string that is matched either by r or by s.
- In terms of languages, the language r|s is the union of language r and s, or L(r|s)=L(r)UL(s)
- A simple example, L(a|b)=L(a)U (b)={a, b}
- Choice can be extended to more than one alternative.

4. Regular Expressions

4.3 Regular Expression Operations

b) Concatenation

- If r and s are regular expression, the rs is their concatenation which matches any string that is the concatenation of two strings, the first of which matches r and the second of which matches s.
- In term of generated languages, the concatenation set of strings S1S2 is the set of strings of S1 appended by all the strings of S2.
- A simple example, (a|b)c matches ac and bc
- Concatenation can also be extended to more than two regular expressions.

4. Regular Expressions

4.3 Regular Expression Operations

c) Repetition

- The repetition operation of a regular expression, called (Kleene) closure, is written r*, where r is a regular expression. The regular expression r* matches any finite concatenation of strings, each of which matches r.
- A simple example, a* matches the strings epsilon, a, aa, aaa,…
- In term of generated language, given a set of S of string, S* is a infinite set union, but each element in it is a finite concatenation of string from S

4. Regular Expressions

4.4 Precedence of Operation and Use of Parentheses

- The standard convention

  Repetition * has highest precedence

  Concatenation is given the next highest

  | is given the lowest
- A simple example

  a|bc* is interpreted as a|(b(c*))
- Parentheses is used to indicate a different precedence

4. Regular Expressions

4.5 Name for regular expression

- Give a name to a long regular expression
  - *digit* = 0|1|2|3|4……|9
  - (0|1|2|3……|9)(0|1|2|3……|9)*
  - *digit digit**

4. Regular Expressions

4.5 Definition of Regular Expression

- A regular expression is one of the following:

  (1) A basic regular expression, a single legal character a from alphabet ∑, or meta-character **ε** or **Φ.**

  (2) The form r|s, where r and s are regular expressions

  (3) The form rs, where r and s are regular expressions

  (4) The form r*, where r is a regular expression

  (5) The form (r), where r is a regular expression

- Parentheses do not change the language.

4. Regular Expressions
4.6 Examples of Regular Expressions

Example 1:

- ∑={ a,b,c}
- the set of all strings over this alphabet that contain exactly one b.
- **(a|c)*b(a|c)***

Example 2:

- ∑={ a,b,c}
- the set of all strings that contain at most one b.
- **(a|c)*|(a|c)*b(a|c)***   **(a|c)*(b|ε)(a|c)***
- the same language may be generated by many different regular expressions.

4. Regular Expressions

4.6 Examples of Regular Expressions

Example 3:

- ∑={ a,b}
- the set of strings consists of a single b surrounded by the same number of a’s.
- S = {b, aba, aabaa,aaabaaa,……} = { $a^n b a^n$ | n≠0}
- This set can not be described by a regular expression.
  - **“regular expression can’t count ”**

- ***not all sets of strings can be generated by regular expressions***.
- **a regular set** : a set of strings that is the language for a regular expression is distinguished from other sets.

4. Regular Expressions

4.6 Examples of Regular Expressions

Example 4:

- ∑={ a,b,c}
- The strings contain no two consecutive b's
- ((a|c)* | (b(a|c))*)*
- ((a | c) | (b(a | c)))* or (a | c | ba | bc)*
  - Not yet the correct answer

The correct regular expression

- (a | c | ba | bc)* (b |ε)
- ((b |ε) (a | c | ab| cb)*
- (not b |b not b)*(b|ε) not b = a|c

4. Regular Expressions

4.6 Examples of Regular Expressions

Example 5:

- $\sum$={ a,b,c}
- ((b|c)* a(b|c)*a)* (b|c)*
- Determine a concise English description of the language
- the strings contain an even number of a's

  (nota* a nota* a)* nota*

5. Extensions to Regular Expression

5.1 List of New Operations

1) one or more repetitions

    r+

2) any character

    period “. ”

3) a range of characters

    [0-9], [a-zA-Z]

5. Extensions to Regular Expression

5.1 List of New Operations

4) any character not in a given set

~(a|b|c) a character not either a or b or c

[^abc] in Lex

5) optional sub-expressions

- r? the strings matched by r are optional

6. Regular Expressions for Programming Language Tokens

6.1 Number, Reserved word and Identifiers

Numbers

- *nat* = [0-9]+
- *signedNat* = (+|-)?*nat*
- *number* = *signedNat*(“. ”*nat*)? (E *signedNat*)?

Reserved Words and Identifiers

- *reserved* = if | while | do |………
- *letter* = [a-z A-Z]
- *digit* = [0-9]
- *identifier* = *letter(letter|digit)**

6. Regular Expressions for Programming Language Tokens

6.2 Ambiguity

Ambiguity: some strings can be matched by several different regular expressions.

- either an identifier or a keyword, **keyword interpretation preferred**.
- a single token or a sequence of several tokens, the single-token preferred.(the principle of **longest sub-string.**)

6. Regular Expressions for Programming Language Tokens

6.3 White Space and Lookahead

White space:

- Delimiters: characters that are unambiguously part of other tokens are delimiters.
- *whitespace* = (n*ewline* | *blank* | *tab* | *comment*)+
- free format or fixed format

Lookahead:

- buffering of input characters , marking places for backtracking

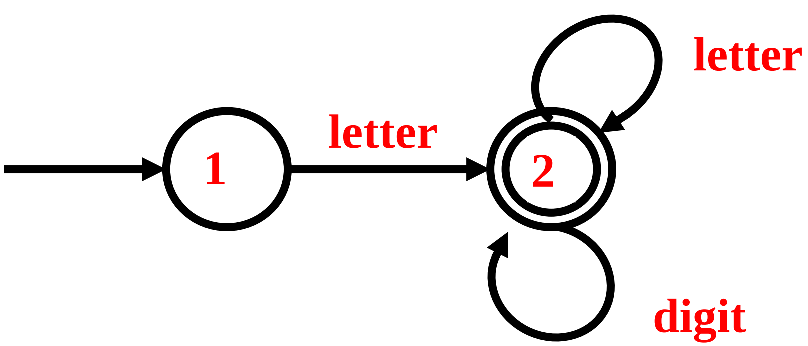
  DO99I=1,10

  DO99I=1.10

7. FINITE AUTOMATA

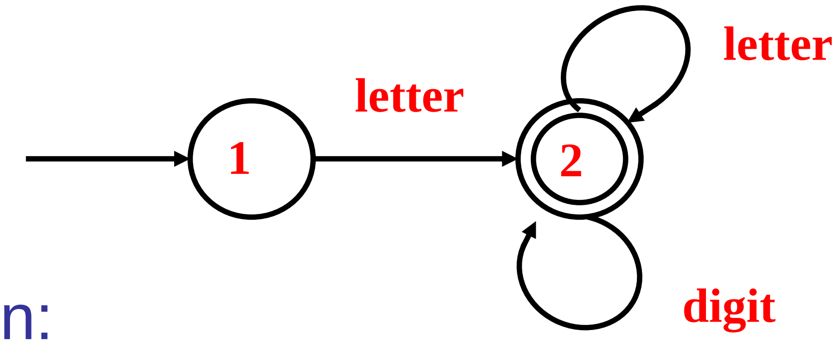
7.1 Introduction to Finite Automata

- Finite automata (finite-state machines) are a mathematical way of describing particular kinds of algorithms.
- A strong relationship between finite automata and regular expression
  - *Identifier = letter (letter | digit)**



7. FINITE AUTOMATA

7.1 Introduction to Finite Automata



- Transition:
  - Record a change from one state to another upon a match of the character or characters by which they are labeled.
- Start state:
  - The recognition process begin
  - Drawing an unlabeled arrowed line to it coming “from nowhere”
- Accepting states:
  - Represent the end of the recognition process.
  - Drawing a double-line border around the state in the diagram

8. Definition of Deterministic Finite Automata

8.1The Concept of DFA

DFA: Automata where the next state is uniquely given by the current state and the current input character.

Definition of a DFA:

A DFA (Deterministic Finite Automation) **M** consist of

(1) an alphabet $\sum$,

(2) A set of states S,

(3) a transition function $T : S \times \sum \rightarrow S$,

(4) a start state $s0 \in S$,

(5)And a set of accepting states $A \subset S$

8. Definition of Deterministic Finite Automata

8.1The Concept of DFA

The language accepted by a DFA M, written L(M), is defined to be

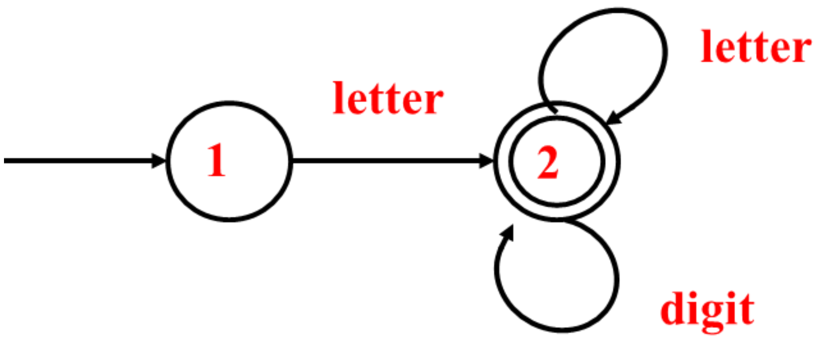
the set of strings of characters c1c2c3….cn with each ci ∈∑ such that there exist states s1 = t(s0,c1),s2 = t(s1,c2), sn = T(sn-1,cn) with sn an element of A (i.e. an accepting state).

Accepting state s_n means the same thing as the diagram:

$$\rightarrow s0 \xrightarrow{c1} s1 \xrightarrow{c2} s2 \rightarrow \ldots\ldots\ldots sn\text{-}1 \xrightarrow{cn} sn$$

8. Definition of Deterministic Finite Automata

8.2 Some differences between definition of DFA and the diagram:

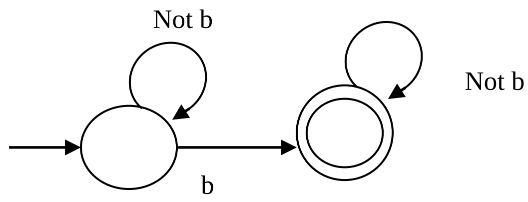


1) The definition does not restrict the set of states to **number**s
2) We have not labeled the transitions with characters but with **names** representing a set of characters
3) definitions T: $S \times \sum \rightarrow S$, T(s, c) must **have a value for every s and c**.
   - In the diagram, T (start, c) defined only if c is a letter, T(in_id, c) is defined only if c is a letter or a digit.
   - Error transitions are not drawn in the diagram but are simply assumed to always exist.

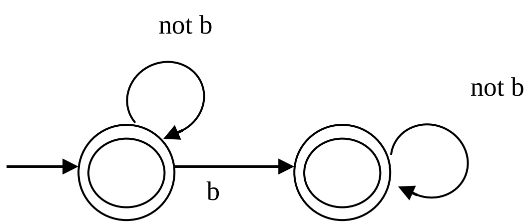
8. Definition of Deterministic Finite Automata

8.3 Examples of DFA

Example 2.6: exactly accept one b



Example 2.7: at most one b



8. Definition of Deterministic Finite Automata

8.3 Examples of DFA

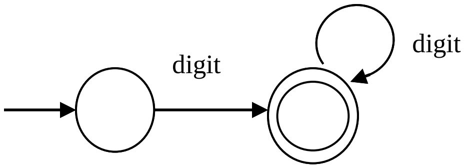
Example 2.8: digit = [0-9]

nat = digit +

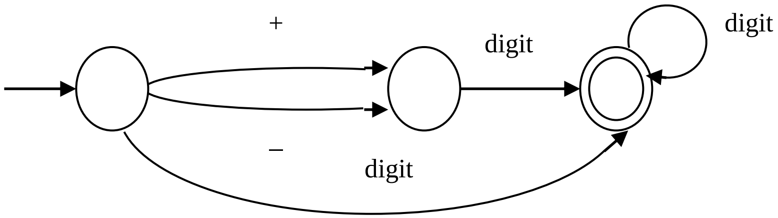
signedNat = (+|-)? nat

Number = singedNat(“.”nat)?(E signedNat)?

A DFA of nat:



A DFA of signedNat:



8. Definition of Deterministic Finite Automata

8.3 Examples of DFA

Example 2.8: digit = [0-9]

nat = digit +

signedNat = (+|-)? nat

Number = singedNat(“.”nat)?(E signedNat)?

A DFA of Number: