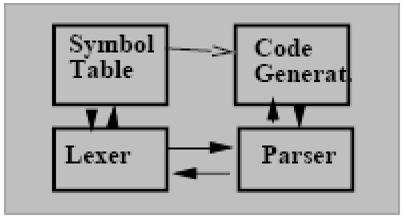
Lexical Analysis

- Dragon Book Chapter 3
- Formal Languages
- Regular Expressions
- Finite Automata Theory
- Lexical Analysis using Automata

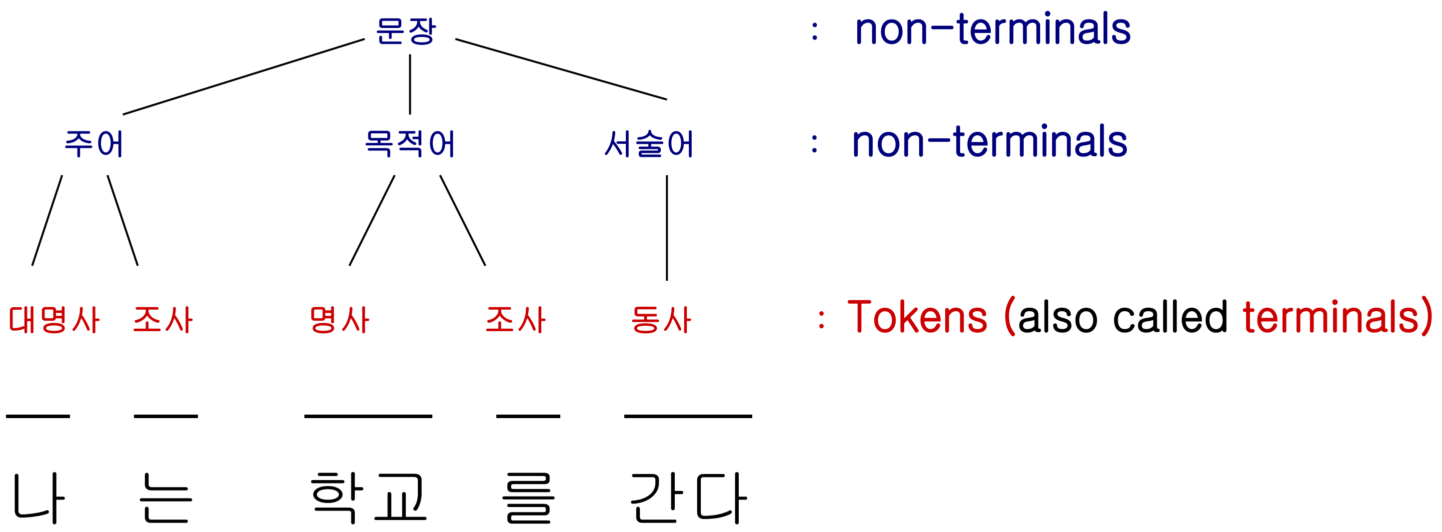
Phase Ordering of Front-Ends



- Lexical analysis (lexer)
  - Break input string into "words" called *tokens*
- Syntactic analysis (parser)
  - Recover structure from the text and put it in a parse tree
- Semantic Analysis
  - Discover "meaning" (e.g., type-checking)
  - Prepare for code generation
  - Works with a symbol table

Similarity to Natural Languages

Tokens and a Parse Tree



What is a Token?

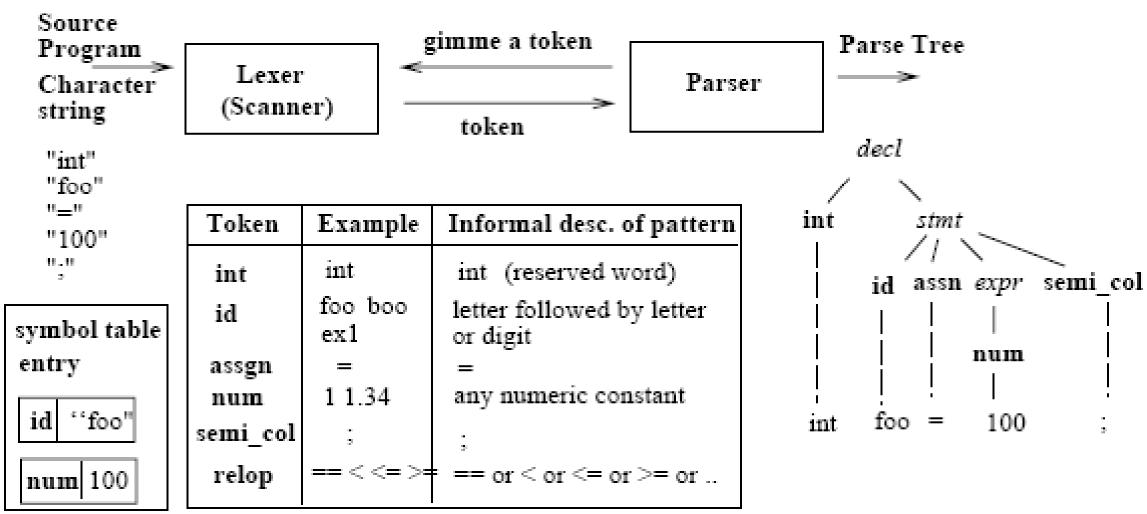
- A syntactic category
  - In English:
    - Noun, verb, adjective, …
  - In a programming language:
    - Identifier, Integer, Keyword, White-space, …
- A token corresponds to a set of strings

Terms

- *Token*
  - Syntactic “atoms” that are “terminal” symbols in the grammar from the source language
  - A data structure (or pointer to it) returned by lexer
- *Patten*
  - A “rule” that defines strings corresponding to a token
- *Lexeme*
  - A string in the source code that matches a pattern

An Example of these Terms

- `int foo = 100;`



Token	Example	Informal desc. of pattern
int	int	int (reserved word)
id	foo boo ex1	letter followed by letter or digit
assgn	=	=
num	1 1.34	any numeric constant
semi_col	;	;
relop	== < <= >=	== or < or <= or >= or ..

- The lexeme matched by the pattern for the token represents a string of characters in the source program that can be treated as a lexical unit

What are Tokens For?

- Classify substrings of a given program according to its role
- Parser relies on token classification
  - e.g., How to handle reserved keywords? As an identifier or a separate keyword for each?
- Output of the lexer is a stream of tokens which is input to the parser
- How parser and lexer co-work?
  - Parser leads the work

Lexical Analysis Problem

- Partition input string of characters into disjoint substrings which are tokens

```
if (i==j)
    z = 0;      =>   \tif (i==j)\n\t\tz = 0;\n\telse\n\t\tz = 1;
else
    z = 1;
```

- Useful tokens here: identifier, keyword, relop, integer, white space, (,), =, ;

Designing Lexical Analyzer

- First, define a set of tokens
  - Tokens should describe all items of interest
  - Choice of tokens depends on the language and the design of the parser
- Then, describe what strings belongs to each token by providing a pattern for it

Implementing Lexical Analyzer

- Implementation must do two thing:
  - Recognize substrings corresponding to tokens
  - Return the "value" or "lexeme" of the token: the substring matching the category

  Reading left-to-right, recognizing one token at a time
- The lexer usually discards "uninteresting" tokens that do not contribute to parsing
  - Examples: white space, comments
- Is it as easy as it sounds? Not actually!
  - Due to lookahead and ambiguity issues (Look at the history)

Lexical Analysis in Fortran

- Fortran rule: white space is insignificant
  - Example: "**VAR1**" is the same as "**VA R1**"
  - Left-to-right reading is not enough
    - `DO 5 I = 1,25    ==> DO  5  I  =  1 , 25`
    - `DO 5 I = 1.25    ==> DO5I = 1.25`
  - Reading left-to-right cannot tell whether **DO5I** is a variable or a **DO** statement until "**.**" or "**,**" is reached
  - "Lookahead" may be needed to decide where a token ends and the next token begins
  - Even our simple example has lookahead issues
    - e.g, "=" and "=="

Lexical Analysis in PL/I

- PL/I keywords are not reserved

```
IF THEN ELSE THEN = ELSE; ELSE ELSE = THEN
```

- PL/I Declarations

```
DECLARE (ARG1, .. ,ARGN)
```

- Cannot tell whether **DECLARE** is a keyword or an array reference until we see the charater that follows “)”, requiring an arbitrarily long lookahead

Lexical Analysis in C++

- C++ template syntax:
  - Foo<Bar>
- C++ io stream syntax:
  - Cin >> var;
- But there is a conflict with nested templates
  - Foo<Bar<int>>

Review

- The goal of lexical analysis is to
  - Partition the input string into lexemes
  - Identify the token of each lexeme
- Left-to-right scan, sometimes requiring lookahead
- We still need
  - A way to describe the lexemes of each token: **pattern**
  - A way to resolve ambiguities
    - Is "==" two equal signs "=" "=" or a single relational op?

Specifying Tokens: Regular Languages

- There are several formalisms for specifying tokens but the most popular one is "**regular languages**"
- Regular languages are not perfect but they have
  - ∃a concise (though sometimes not user-friendly) expression: **regular expression**
  - ∃a useful theory to evaluate them ➔ **finite automata**
  - ∃a well-understood, efficient implementation
  - ∃a tool to process regular expressions ➔ **`lex`**

    Lexical definitions (regular expressions) → `lex` →

    a table-driven lexer (C program)

Formal Language Theory

- **Alphabet ∑** : a finite set of symbols (characters)
  - Ex: {a,b}, an ASCII character set
- **String**: a finite sequence of symbols over ∑
  - Ex: abab, aabb, a over {a,b}; "hello" over ASCII
  - Empty string **ϵ**: zero-length string
    - $\epsilon \neq \emptyset \neq \{\epsilon\}$
- **Language**: a set of strings over ∑
  - Ex: {a, b, abab} over {a,b}
  - Ex: a set of all valid C programs over ASCII

Operations on Strings

- Concatenation (·):
  - a · b = ab, “hello” · ”there” = “hellothere”
  - Denoted by **α** · **β** = **αβ**
- Exponentiation:
  - $hello^3$ = hello · hello · hello = hellohellohello, $hello^0$ = **ϵ**
- Terms for parts of a string s
  - *prefix of* s : A string obtained by removing zero or more trailing symbols of string s: (Ex: ban is a prefix of banana)
  - *proper prefix* of s: A non-empty prefix of s that is not s

Operations on Languages

- Lex X and Y be sets of strings
  - Concatenation (·): $X \cdot Y = \{x \cdot y \mid x \in X, y \in Y\}$
    - Ex: X = {Liz, Add} Y = {Eddie, Dick}
    - X · Y = {LizEddie, LizDick, AddEddie,AddDick}
  - Exponentiation: $X^2 = X \cdot X$
    - $X^0 = \epsilon$
  - Union: $X \cup Y = \{u \mid u \in X \text{ or } u \in Y\}$
  - **Kleene's Closure**: $X^* = \bigcup_{i=0}^{\infty} X^i$
    - Ex:X = {a,b}, X^* = {ϵ, a, b, aa, ab, ba, bb, aaa, ..}

Regular Languages over ∑

- Definition of regular languages over ∑
  - Ø is regular
  - {a} is regular
  - {ϵ} is regular
  - R∪S is regular if R, S are regular
  - R·S is regular if R, S are regular
  - Nothing else

Regular Expressions (RE) over Σ

- In order to describe a regular language, we can use a regular expression (RE), which is strings over Σ representing the regular language
  - $\emptyset$ is a regular expression
  - ϵ is a regular expression
  - **a** is regular expression for **a** $\in \Sigma$
  - Let r, s be regular expressions. Then,
    - (r) | (s) is a regular expression
    - (r) · (s) is a regular expression
    - $(r)^*$ is a regular expression
  - Nothing else
  - Ex: $\Sigma = \{a, b\}$, $ab|ba^* = (a)(b)|((b)((a)^*))$

Regular Expressions & Languages

- Let s and r be REs
  - $L(\emptyset) = \emptyset$, $L(\boldsymbol{\epsilon}) = \{\boldsymbol{\epsilon}\}$, $L(a) = \{a\}$
  - $L(s \cdot r) = L(s) \cdot L(r)$, $L(s|r) = L(s) \cup L(r)$
  - $L(r^*)=(L(r))^*$
- Anything that can be constructed by a finite number of applications of the rules in the previous page is a regular expression which equally describe a regular language
  - Ex: **ab*** = {**a, ab, abb, …**}
  - Quiz: what is a RE describing at least one **a** and any number of **b**'s
    - **(a|b)*a(a|b)*** or **(a*b*)*a(a*b*)***

Non-Regular Languages

- Not all languages are regular (i.e., cannot be described by any regular expressions)
  - Ex: set of all strings of balanced parentheses
    - {(), (()), ((())), (((()))), …}
    - What about $(^*)^*$?
    - Nesting can be described by a context-free grammar
  - Ex: Set of repeating strings
    - {*w***c***w*| *w* is a string of **a**'s and **b**'s}
    - {**aca**, **abcab**, **abacaba**, …}
    - Cannot be described even by a context-free grammar
- Regular languages are not that powerful

RE Shorthands

- $r? = r|\epsilon$ (zero or one instance of r)
- $r^+ = r \cdot r^*$ (positive closure)
- Charater class: [abc] = a|b|c, [a–z] = a|b|c|⋯|z
- Ex: $([ab]c?)^+$ = {a, b, aa, ab, ac, ba, bb, bc,⋯}

Regular Definition

- For convenience, we give names to regular expressions and define other regular expressions using these names as if they are symbols
- Regular definition is a sequence of definitions of the following form,

  $d_1 \rightarrow r_1$
  $d_2 \rightarrow r_2$
  …
  $d_n \rightarrow r_n$
  - d_i is a distinct name
  - r_i is a regular expression over the symbols in $\Sigma \cup \{d_1, d_2, \cdots, d_{i-1}\}$
- For **lex** we use regular definitions to specify tokens; for example,
  - letter → [A-Za-z]
  - digit → [0-9]
  - id → letter(letter|digit)*

Examples of Regular Expressions

- Our tokens can be specified by the following
  - for → for
  - id → letter(letter|digit)*
  - relop → <|<=|==|!=|>|>=
  - num → digit^{+}(.digit^{+})?(E(+|−)?digit^{+})?
- Our lexer will strip out white spaces
  - delim → [₩t₩n]
  - ws → delim^{+}

More Regular Expression Examples

- Regular expressions are all around you!
  - Phone numbers: (02)-880-1814
    - Σ = digit $\cup$ {-,(,)}
    - exchange → digit^3
    - phone → digit^4
    - area → (digit^3)
    - phone_number = area – exchange – phone

Another Regular Expression Example

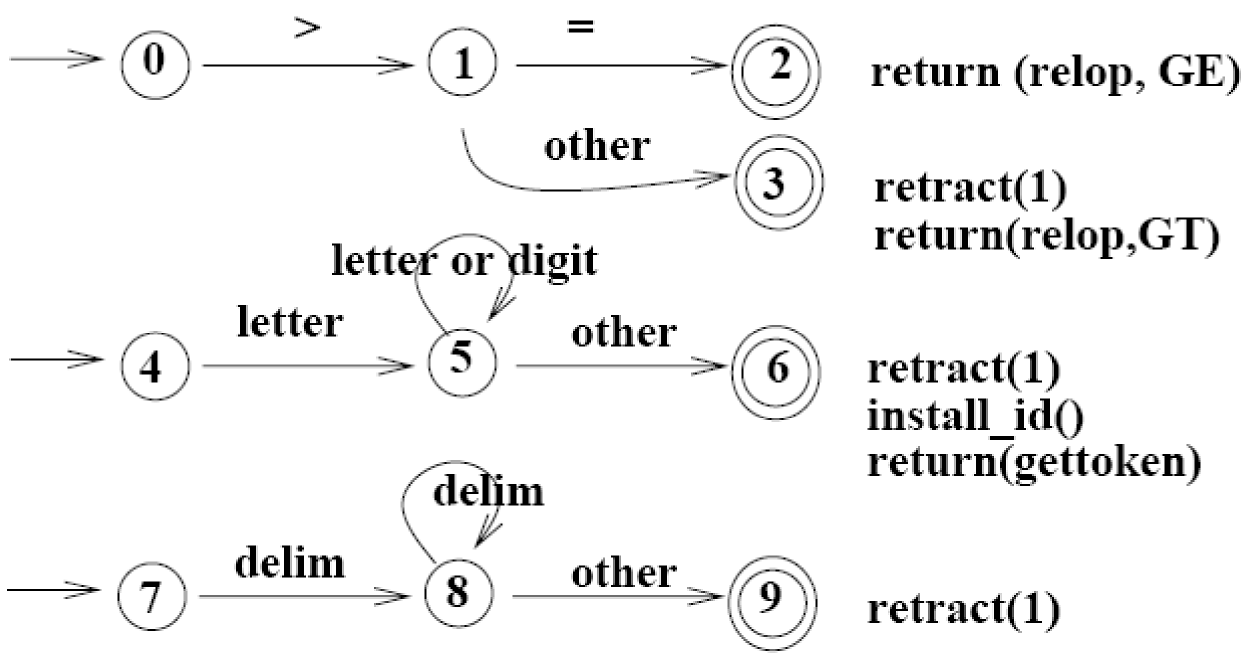
- E-mail addresses: smoon@altair.snu.ac.kr
  - Σ = letter $\cup$ {.,@}
  - Name = letter$^+$
  - Address = name'@'name'.'name'.'name'.'name
  - Real e-mail address will be more elaborate but still regular
- Other examples: file path names, etc.

Review and the Next Issue

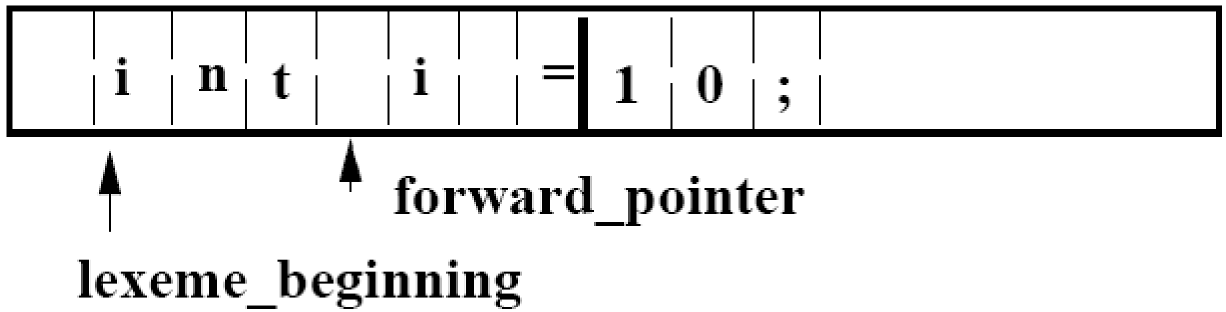
- Regular expressions are a language specification that describe many useful languages including set of tokens for programming language compilers
- We still need an implementation for them
- Our problem is
  - Given a string ***s*** and a regular expression **R**, is ***s*** $\in$ L(**R**) ?
- Solution for this problem is the base of lexical analyzer
- A naïve solution: transition diagram and input buffering
- A more elaborate solution
  - Using the theory and practice of deterministic finite automata (DFA)

Transition Diagram

- A flowchart corresponding to regular expression(s) to keep track of information as characters are scanned
  - Composed of states and edges that show transition



Input Buffering



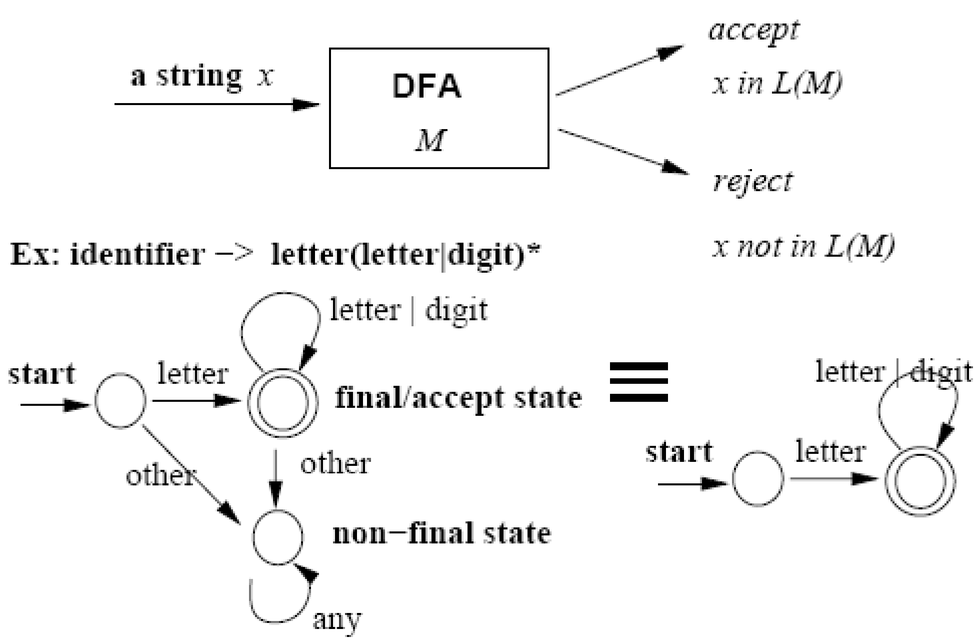
- Two pointers are maintained
  - Initially both pointers point the first character of the next lexeme
  - Forward pointer scans; if a lexeme is found, it is set to the last character of the lexeme found
  - After processing the lexeme, both pointers are set to the character immediately the lexeme

Making Lexer using Transition Diagrams

- Build a list of transition diagrams for all regular expressions
- Start from the top transition diagram and if it fails, try the next diagram until found; **fail()** is used to move the forward pointer back to the lexeme_beginning
- If a lexeme is found but requires **retract(n)**, move the forward pointer **n** charcters back
- Basically, these ideas are used when implementing deterministic finite automata (DFA) in `lex`

Deterministic Finite Automata (DFA)

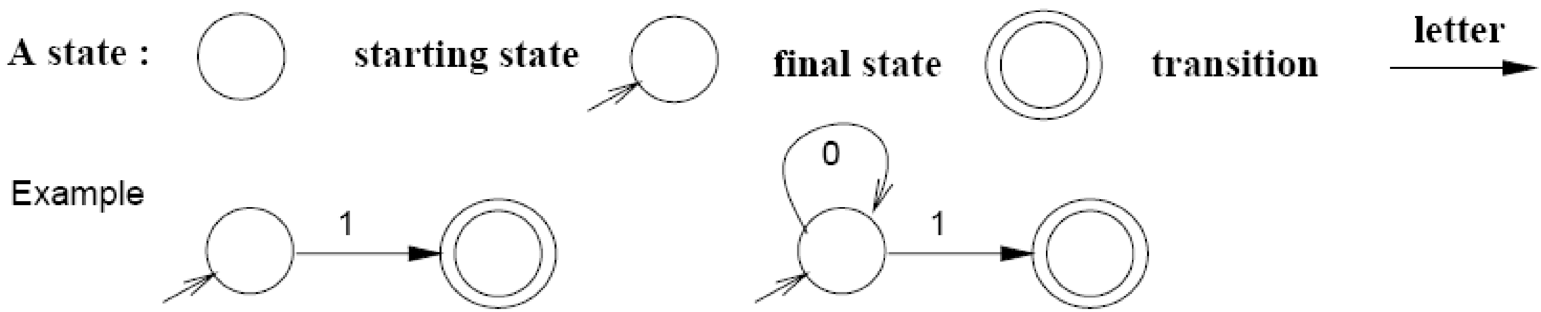
- Language recognizers with finite memory contained in states
  - A DFA accepts/rejects a given string if it is/is not a language of the DFA
- Regular languages can be recognized by DFAs



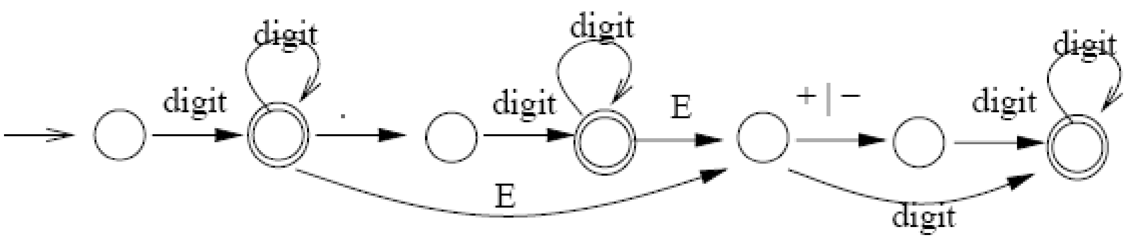
Formal Definition of a DFA

- A deterministic finite state automata M = (∑, Q, **δ**, q_0, F)
  - ∑: alphabet
  - Q: set of states
  - **δ**: Q x ∑ → Q, a transition function
  - q_0: the start state
  - F: final states
- A run on an input ***x*** is a sequence of states by "consuming" ***x***
- A string ***x*** is accepted by M if its run ends in a final state
- A language accepted by a DFA M, L(M) = {x|M *accepts* x}

Graphic Representation of DFA

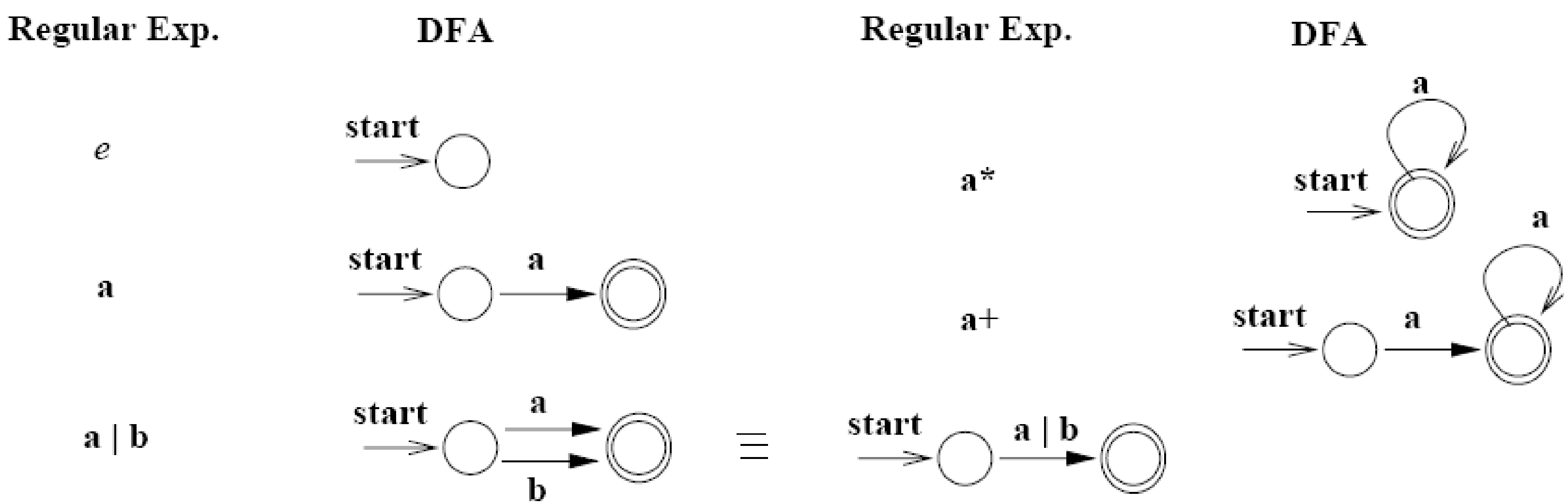


A DFA Example: A Number



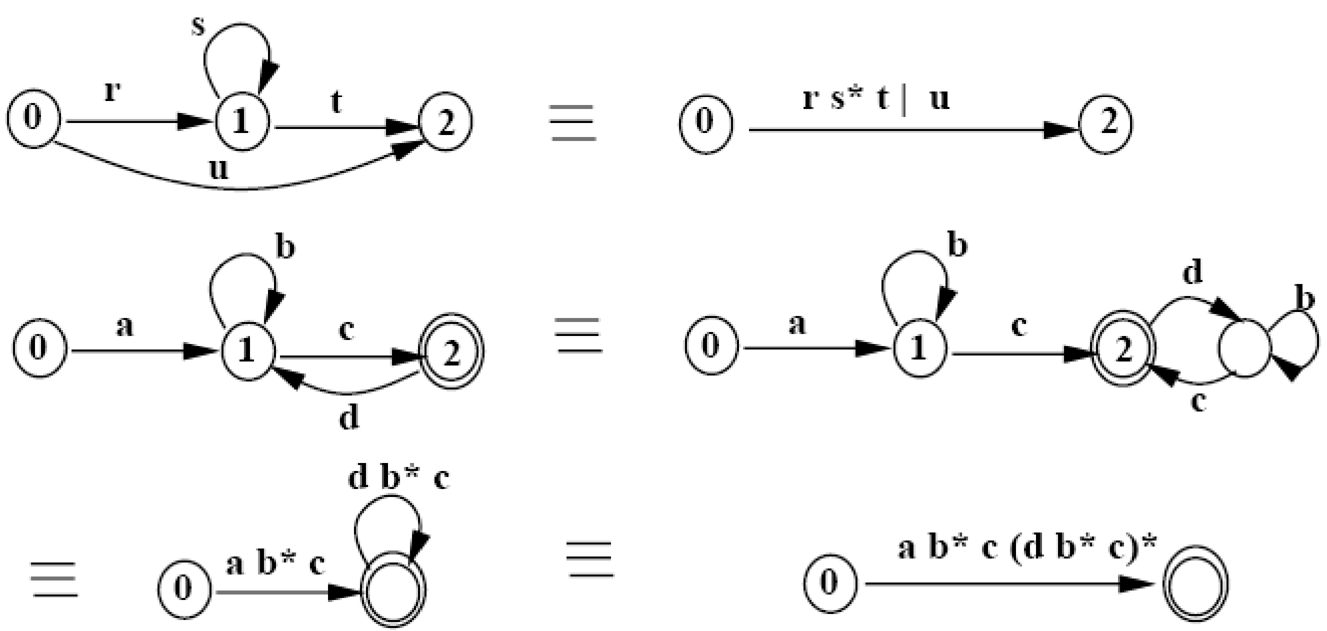
- num → digit$^+$(.digit$^+$)?(E(+|−)?digit$^+$)?

From Regular Expression to DFA



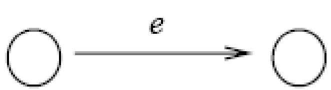
From DFA to Regular Expression

- We can determine a RE directly from a DFA either by inspection or by “removing” states from the DFA



Nondeterministic Finite Automata (NFA)

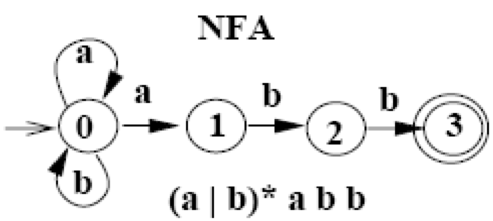
- Conversion from RE to NFA is more straightforward
  - ϵ−transition



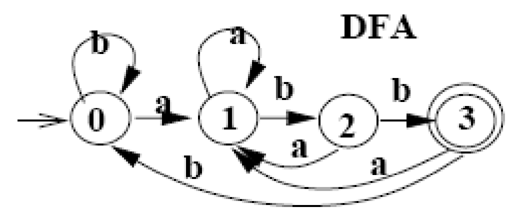
  - Multiple transitions on a single input i.e., $\delta : Q \times \Sigma \rightarrow 2^Q$
- We will not cover much of NFA stuff in this lecture
  - Conversion of NFA to DFA: subset construction Ch. 3.6
  - From RE to an NFA: Thomson's construction Ch. 3.7
  - Minimizing the number of states in DFA: Ch. 3.9
- Equivalence of RE, NFA, and DFA:
  - L(RE) = L(NFA) = L(DFA)

Subset Construction

- Basic Idea
  - Each DFA state corresponds to a set of NFA states: keep track of all possible states the NFA can be in after reading each symbol
  - The number of states in DFA is exponential in the number of states of NFA (maximum 2^n states)

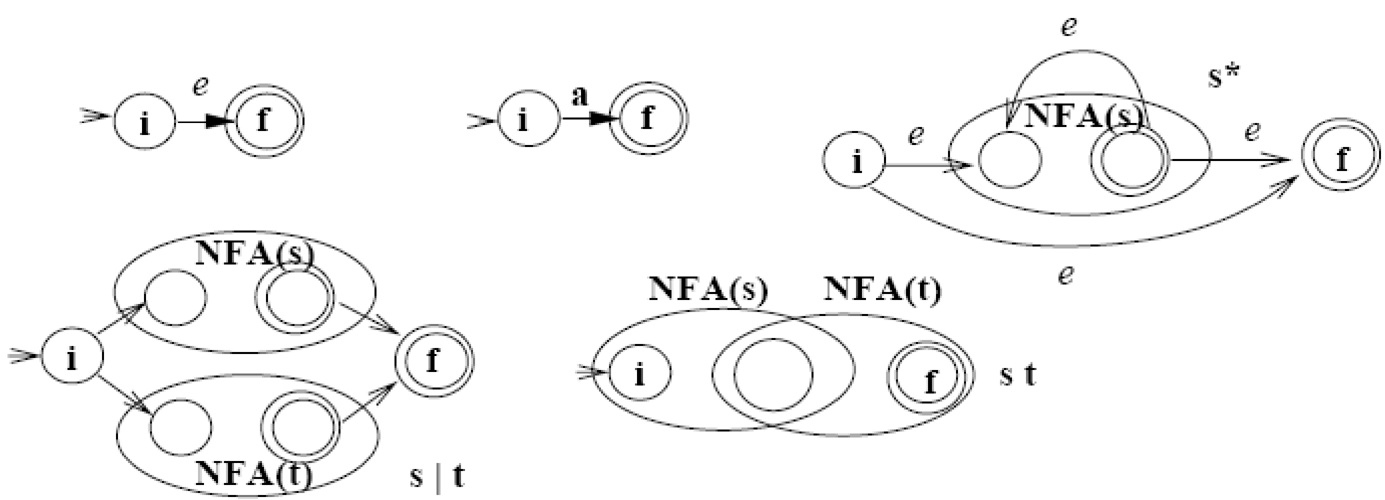


	{0}	{0,1}	{0,2}	{0,3}
a	{0,1}	{0,1}	{0,1}	{0,1}
b	{0}	{0,2}	{0,3}	{0}



Thomson's Construction

- From RE to NFA

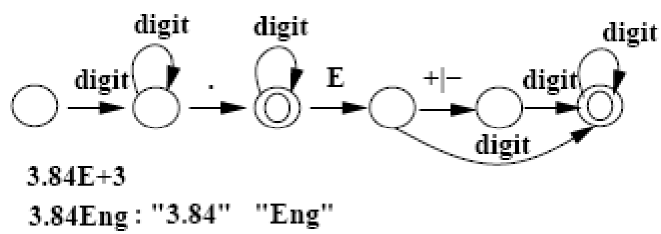


Lexical Analysis using Automata

- Automata *vs.* Lexer
  - Automata accepts/rejects *strings*
  - Lexer recognizes *tokens* (prefixes) from a longer string
  - Lookahead issues: number of characters that must be read beyond the end of a lexeme to recognize it
  - Resolving ambiguities:
    - Longest lexeme rule
    - Precedence rule

Longest Lexeme Rule

- In case of multiple matches longer ones are matched
  - Ex: floating-point numbers (digit$^+$.digit*(E(+|-)?digit$^+$)?)



  - Can be implemented with our buffer scheme: when we are in accept state, mark the input position and the pattern; keep scanning until fail when we retract the forward pointer back to the last position recorded

- Precedence rule of `lex`
  - Another rule of `lex` to resolve ambiguities: In case of ties `lex` matches the RE that is closer to the beginning of the lex input

Pitfall of Longest Lexeme Rule

The longest lexeme rule does not always work

- Ex: L = {ab, aba, baa} and input string abab

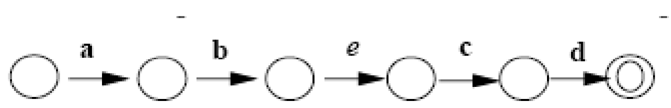
  Infinite maximum lookahead is needed for ababaaba…

  THIS IS A WRONG set of lexemes
- Unfortunately this might be a real life situation

  Ex: Floating-point numbers as defined above and resolving ".." (DOTDOT); e.g., 1..2

Lookahead Operator of lex

- Lookahead Operator
  - RE for lex input can include an operator “/” such as ab/cd, where ab is matched only if it is followed by cd



  - If matched at “d”, the forward pointer goes back to “b” position before the lexeme ab is processed

Summary of Lexical Analysis

- Token, pattern, lexeme
- Regular languages
- Regular expressions (computational model, tools)
- Finite automata (DFA, NFA)
- Lexer using automata: longest lexeme rules
- Tool: `lex`
- Programming Assignment #1
  - Writing a lexical analyzer for a subset of C, subc, using `lex` (nested comments, lookaheads, hash tables)