Outline of First Few Lectures

Quick review:

- Languages and grammars (generators)
- Regular Languages and Lexical Analysis

Next Major Topic : Parsing

- Context-Free Grammars and Languages
- Top-down Parsing
- Bottom-up Parsing
  - Shift-Reduce Parsing
  - LR Parsing
- Automatic parser construction tools

Languages and Grammars

Languages

- fixed, finite alphabet (or *symbols* or *vocabulary*)
- finite length sentences (or *strings*)
- possibly infinitely many strings
- Examples:
  - The natural numbers: { 0, 1, …, 10, 11, …}
  - Strings over {a,b} ending in a single ‘b’: { a, ab, aab, aaab, …}

Grammars

- Specify a method by which all strings of a language, L, may be generated via well-defined rules.

Recognizers

- A procedure which, given a "string", χ, answers "yes" if $\chi \in L$. (Usually also want to answer "no" if $\chi \notin L$.)

- **Scanner:**
  - Recognizer to identify the symbols or *tokens* in input
  - Uses a **Regular Language** defined by regular expressions

- **Parser:**
  - Recognizer to identify sentences (strings of tokens) in the input
  - Uses a **Context Free Grammar** defined by context-free rules

Regular Sets (Regular Languages)

Definition: <u>Regular Sets</u>

Let Σ be a finite alphabet.

1. Φ is a regular set over Σ (the empty set)
2. $\{\varepsilon\}$ is a regular set over Σ (ε is the string of length zero)
3. $\forall a \in \Sigma$, $\{a\}$ is a regular set over Σ.
4. If P and Q are regular sets over Σ,
   a. (Set Union) $P \cup Q$ is a regular set over Σ.
   b. (String Concatenation) PQ is a regular set over Σ.
   c. (Kleene Closure) P^* is a regular set over Σ

Nothing else is a regular set over Σ.

Regular Expressions

Regular Expressions: A concise notation for regular sets

(1) Φ denotes the regular set Φ.

(2) ε denotes the regular set {ε}.

(3) α denotes the regular set {α}.

(4) If p and q are regular expressions denoting the regular sets P and Q respectively, then

(a) (p | q) denotes $P \cup Q$

(b) (pq) denotes P Q

(c) (p)* denotes P*

(5) Nothing else is a regular expression.

Notation:

(p)+ = ((p)* p)

Regular vs. Non-regulars Sets

Which of these are regular sets ?

1. All strings of length zero or one character over ∑.

2. All strings of the form wcw^r where w^r is the reverse of w

3. All strings over ∑ = {0,1} with an even numbers of 0s and odd numbers of 1s

4. The set of arithmetic expressions with matched parentheses

Key Properties

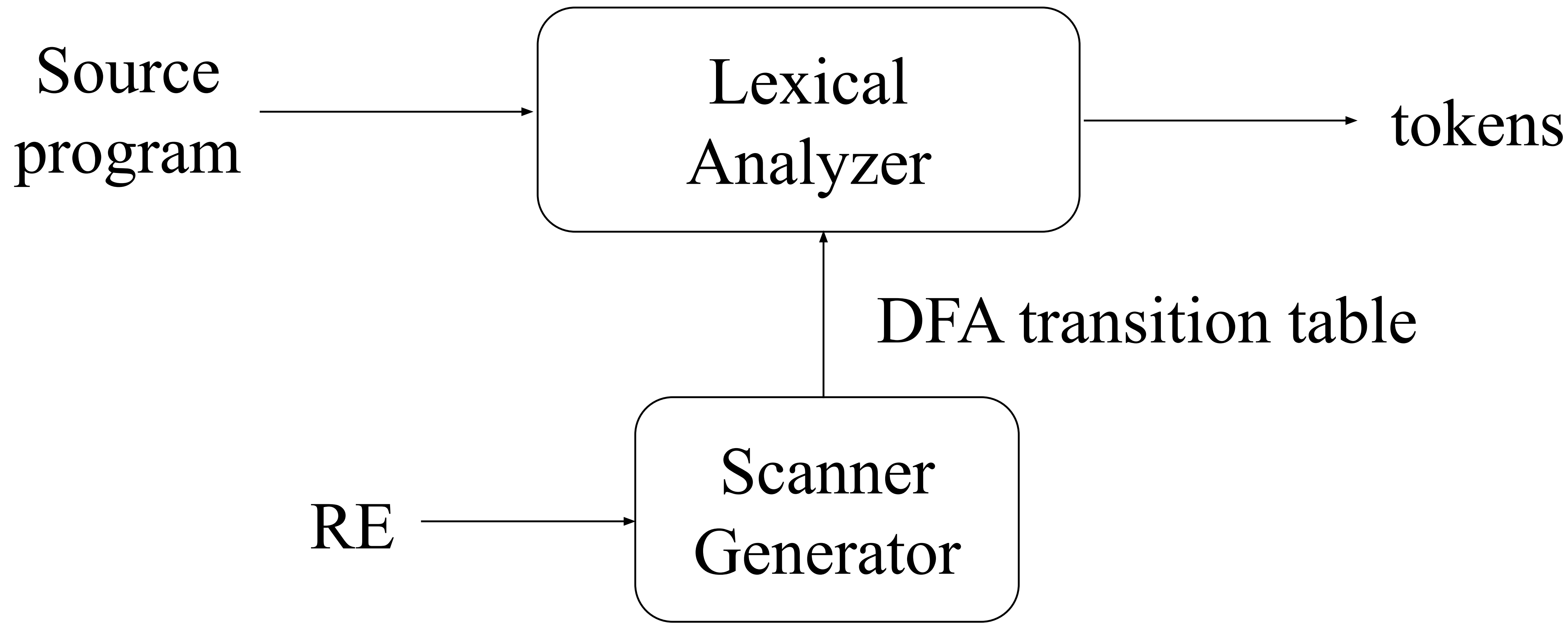
- For every RE, there is a DFSM that recognizes the language defined by that RE

- For every NFSM M_1 there is a DFSM M_2 for which $L(M_2) = L(M_1)$

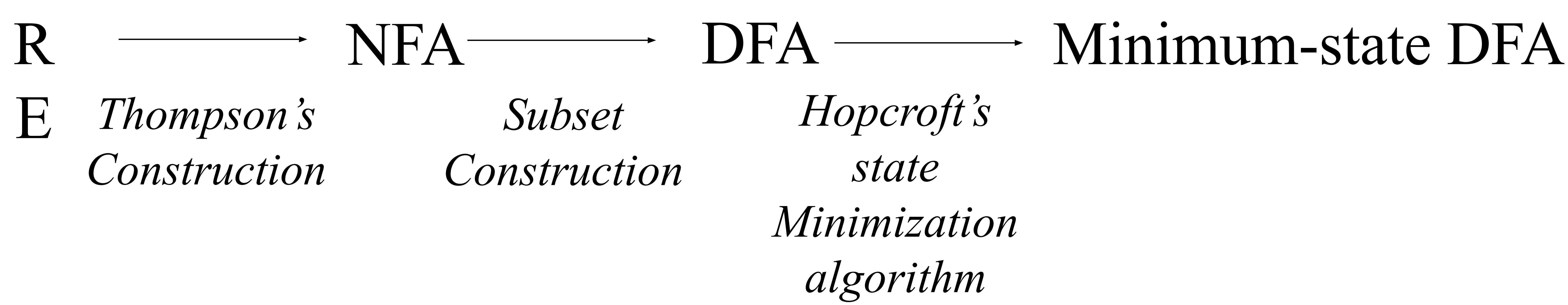
- *Thompson's Construction:*
  - Systematically generate an NFSM for a given Reg. Ex.

- *Subset construction algorithm*:
  - Converts an NFSM to an equivalent DFSM
  - *Key:* identify sets of states of NFSM that have similar behavior, and make each set a single state of the DFSM

Overview of a Scanner Generator



Scanner Generator



Implementation Issues

- Using a DFA for token recognition
- Scanner performance issues
- Handling lexical errors
- Language design issues

Token Recognition With a DFA

1. Flex input: specify tokens and actions

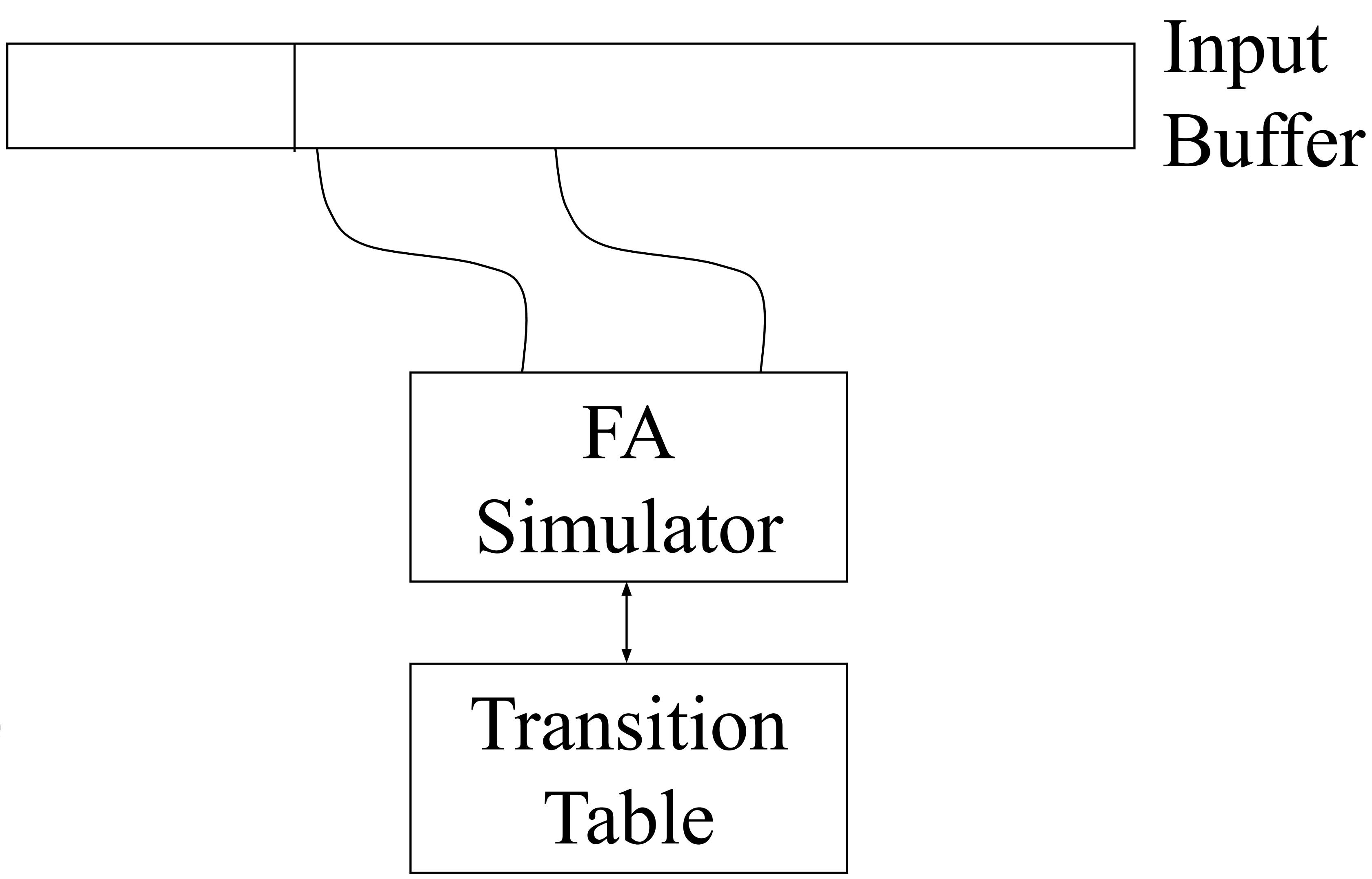
```
P1  { action }
P2  { action }
P3  { action }
```

E.g.

```
[+-]?[0-9]+ {yylval = atoi(yytext); return INTCONST;}
```

2. Flex builds DFA transition table for REs (patterns)

3. Simulate DFA:
   - Linear scan of input file
   - Two buffer pointers:
     1. Start of current lexeme
     2. Current input symbol
   - Remember symbol for last accepting state
   - Execute code for matched pattern
   - Multiple patterns may match same lexeme



Scanner Performance

- See *flex* documentation for many useful insights
  - *Options, Patterns, Actions, Performance Hints*

- Typical Practical Issues:
  - Use a single action per token (REJECT action in *flex*)
  - Avoid backtracking in the input
  - Consume as much text as possible per action
  - Trade-offs in table size vs. speed

    (see documentation of flag *-C* in *flex*)

Lexical Errors

- Scanner can catch few errors: most are syntactic

    Example: X = 900n;

- What’s a scanner to do?

  Recovery strategies:
  - Minimum-distance error correction: insert, delete, replace
  - Skip input characters until match
    - » E.g., “X”, “=”, “900”, <ERROR>, “;”

Language Design Issues

Poor language design complicates lexical analysis

- PL/I had no reserved words!

```
if then then then = else; else else = then
```

- Fortran and Algol68 ignore blanks:

```
do 10 i = 1, 25      ! DO LOOP
do 10 i = 1.25     ! ASSIGNMENT
```

- Fortran66 limited identifiers to 6 characters
  - Use states to count bounded length