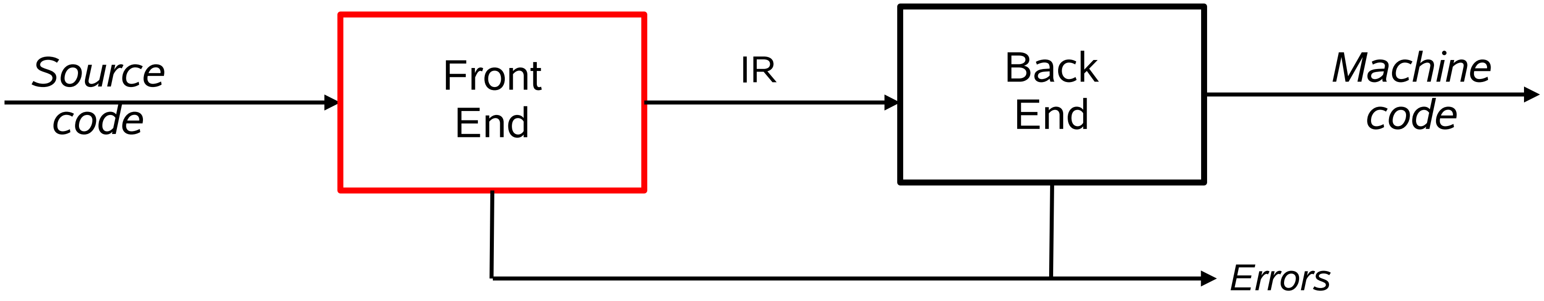
# Lexical Analysis - An Introduction

Copyright 2003, Keith D. Cooper, Ken Kennedy & Linda Torczon, all rights reserved.
Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.

# The Front End

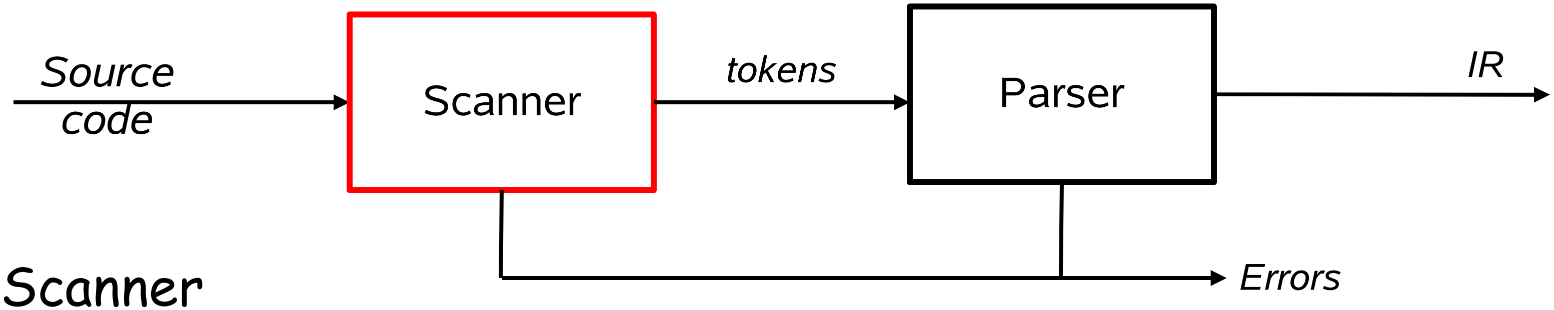



The purpose of the front end is to deal with the input language

- Perform a membership test: code ∈ source language?
- Is the program well-formed (semantically) ?
- Build an IR version of the code for the rest of the compiler

*The front end is not monolithic*

# The Front End




## Scanner

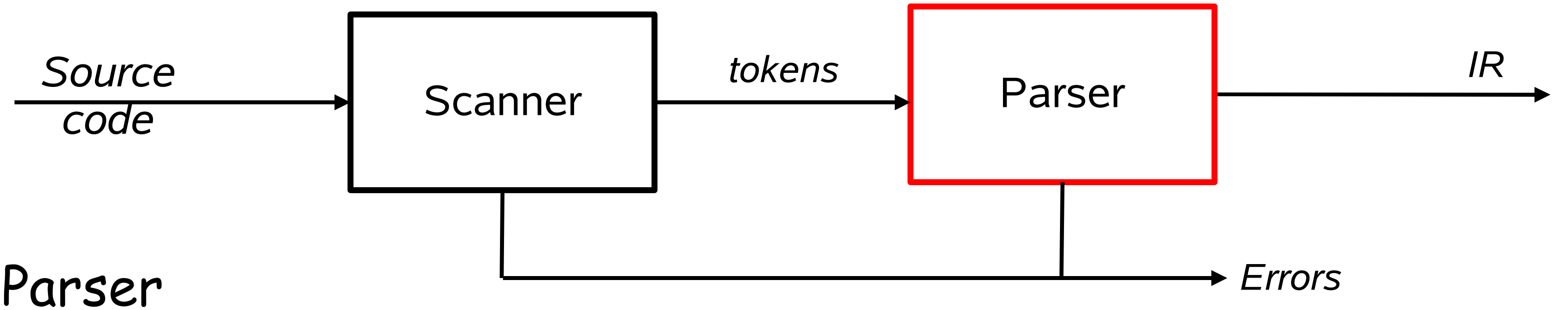
- Maps stream of characters into words
  - → Basic unit of syntax
  - → x = x + y ; *becomes*

    <id,x> <eq,=> <id,x> <pl,+> <id,y> <sc,; >
- Characters that form a word are its *lexeme*
- Its *part of speech* (or *syntactic category*) is called its *token type*
- Scanner discards white space & (often) comments

Speed is an issue in scanning

⇒ use a specialized recognizer

# The Front End




Parser

- Checks stream of classified words (*parts of speech*) for grammatical correctness
- Determines if code is syntactically well-formed
- Guides checking at deeper levels than syntax
- Builds an IR representation of the code

*We'll come back to parsing in a couple of lectures*

# The Big Picture

- Language syntax is specified with *parts of speech*, not *words*
- Syntax checking matches *parts of speech* against a grammar

| | | | |
|---|---|---|---|
| 1. | *goal* | $\rightarrow$ | *expr* |
| 2. | *expr* | $\rightarrow$ | *expr op term* |
| 3. | | \| | *term* |
| 4. | *term* | $\rightarrow$ | number |
| 5. | | \| | id |
| 6. | *op* | $\rightarrow$ | + |
| 7. | | \| | – |

*S* = *goal*

*T* = { number, id, +, - }

*N* = { *goal*, *expr*, *term*, *op* }

*P* = { 1, 2, 3, 4, 5, 6, 7}

# The Big Picture

- Language syntax is specified with *parts of speech*, not *words*
- Syntax checking matches *parts of speech* against a grammar

| | | | |
|---|---|---|---|
| 1. | *goal* | $\rightarrow$ | *expr* |
| 2. | *expr* | $\rightarrow$ | *expr op term* |
| 3. | | \| | *term* |
| 4. | *term* | $\rightarrow$ | number |
| 5. | | \| | id |
| 6. | *op* | $\rightarrow$ | + |
| 7. | | \| | – |

*S* = *goal*

*T* = { number, id, +, - }

*N* = { *goal*, *expr*, *term*, *op* }

*P* = { 1, 2, 3, 4, 5, 6, 7}

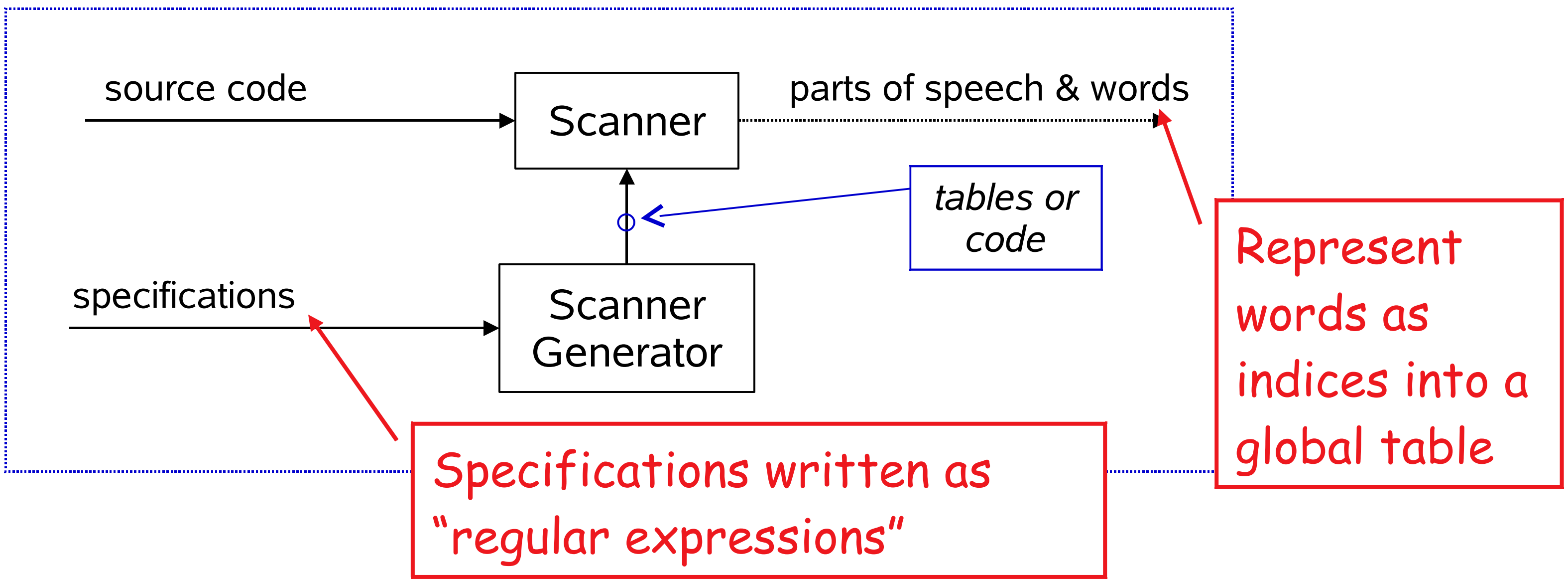
No words here!

Parts of speech, not words!

# The Big Picture

Why study lexical analysis?

- We want to avoid writing scanners by hand
- We want to harness the theory from other classes




Goals:

→ To simplify specification & implementation of scanners

→ To understand the underlying techniques and technologies

# Regular Expressions

Lexical patterns form a *regular language*

*** *any finite language is regular* ***

*Regular expressions* (REs) describe regular languages

Ever type "rm *.o a.out" ?

Regular Expression (over alphabet $\Sigma$)

- $\varepsilon$ is a RE denoting the set $\{\varepsilon\}$
- If $\underline{a}$ is in $\Sigma$, then $\underline{a}$ is a RE denoting $\{\underline{a}\}$
- If $x$ and $y$ are REs denoting $L(x)$ and $L(y)$ then
  - → $x \mid y$ is an RE denoting $L(x) \cup L(y)$
  - → $xy$ is an RE denoting $L(x)L(y)$
  - → $x^*$ is an RE denoting $L(x)^*$

Precedence is *closure*, then *concatenation*, then *alternation*

# Set Operations (review)

| Operation | Definition |
|---|---|
| *Union of L and M* <br> *Written $L \cup M$* | $L \cup M = \{s \mid s \in L \text{ or } s \in M\}$ |
| *Concatenation of L and M* <br> *Written LM* | $LM = \{st \mid s \in L \text{ and } t \in M\}$ |
| *Kleene closure of L* <br> *Written $L^*$* | $L^* = \bigcup_{0 \le i \le \infty} L^i$ |
| *Positive Closure of L* <br> *Written $L^+$* | $L^+ = \bigcup_{1 \le i \le \infty} L^i$ |

# Examples of Regular Expressions

## Identifiers:

$$\textit{Letter} \rightarrow (\underline{a}|\underline{b}|\underline{c}| \ldots |\underline{z}|\underline{A}|\underline{B}|\underline{C}| \ldots |\underline{Z})$$

$$\textit{Digit} \rightarrow (\underline{0}|\underline{1}|\underline{2}| \ldots |\underline{9})$$

$$\textit{Identifier} \rightarrow \textit{Letter} \, ( \textit{Letter} \mid \textit{Digit} )^*$$

## Numbers:

$$\textit{Integer} \rightarrow (\underline{+}|\underline{-}|\varepsilon) \, (\underline{0}| \, (\underline{1}|\underline{2}|\underline{3}| \ldots |\underline{9})(\textit{Digit}^*) \, )$$

$$\textit{Decimal} \rightarrow \textit{Integer} \, \underline{.} \, \textit{Digit}^*$$

$$\textit{Real} \rightarrow ( \textit{Integer} \mid \textit{Decimal} ) \, \underline{E} \, (\underline{+}|\underline{-}|\varepsilon) \, \textit{Digit}^*$$

$$\textit{Complex} \rightarrow \underline{(} \, \textit{Real} \, \underline{,} \, \textit{Real} \, \underline{)}$$

*Numbers can get much more complicated!*

# Regular Expressions (the point)

*Regular expressions can be used to specify the words to be translated to parts of speech by a lexical analyzer*

Using results from automata theory and theory of algorithms, we can automatically build recognizers from regular expressions

Some of you may have seen this construction for string pattern matching

⇒ We study REs and associated theory to automate scanner construction !

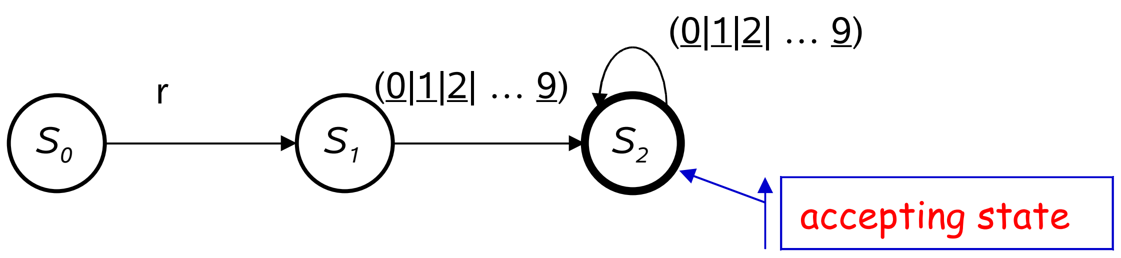
# Example

Consider the problem of recognizing ILOC register names

$$\textit{Register} \rightarrow \texttt{r}\ (\underline{0}|\underline{1}|\underline{2}|\ \ldots\ |\ \underline{9})\ (\underline{0}|\underline{1}|\underline{2}|\ \ldots\ |\ \underline{9})^*$$

- Allows registers of arbitrary number
- Requires at least one digit

RE corresponds to a recognizer (or DFA)



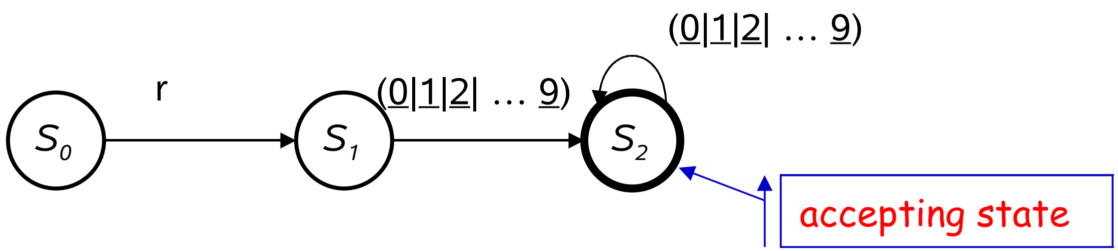

Recognizer for *Register*

*Transitions on other inputs go to an error state, $s_e$*

# Example (continued)

DFA operation

- Start in state $S_0$ & take transitions on each input character
- DFA accepts a word $\underline{x}$ *iff* $\underline{x}$ leaves it in a final state ($S_2$)




Recognizer for *Register*

So,

- $\underline{r17}$ takes it through $s_0$, $s_1$, $s_2$ and accepts
- $\underline{r}$ takes it through $s_0$, $s_1$ and fails
- $\underline{a}$ takes it straight to $s_e$

# Example (continued)

To be useful, recognizer must turn into code

```
Char ← next character
State ← s0

while (Char ≠ EOF)
    State ← δ(State,Char)
    Char ← next character

if (State is a final state )
    then report success
    else  report failure
```

*Skeleton recognizer*

| $\delta$ | r | 0,1,2,3,4,5,6,7,8,9 | *All others* |
|---|---|---|---|
| $s_0$ | $s_1$ | $s_e$ | $s_e$ |
| $s_1$ | $s_e$ | $s_2$ | $s_e$ |
| $s_2$ | $s_e$ | $s_2$ | $s_e$ |
| $s_e$ | $s_e$ | $s_e$ | $s_e$ |

*Table encoding RE*

# Example (continued)

To be useful, recognizer must turn into code

```
Char ← next character
State ← s0

while (Char ≠ EOF)
    State ← δ(State,Char)
    perform specified action
    Char ← next character

if (State is a final state )
    then report success
    else  report failure
```

*Skeleton recognizer*

| $\delta$ | r | 0,1,2,3,4, 5,6,7,8,9 | *All others* |
|---|---|---|---|
| $s_0$ | $s_1$ *start* | $s_e$ *error* | $s_e$ *error* |
| $s_1$ | $s_e$ *error* | $s_2$ add | $s_e$ *error* |
| $s_2$ | $s_e$ *error* | $s_2$ add | $s_e$ *error* |
| $s_e$ | $s_e$ *error* | $s_e$ *error* | $s_e$ *error* |

*Table encoding RE*

# What if we need a tighter specification?

r *Digit Digit** allows arbitrary numbers

- Accepts r00000
- Accepts r99999
- What if we want to limit it to r0 through r31 ?

Write a tighter regular expression

→ *Register* → r ( (0|1|2) (*Digit* | ε) | (4|5|6|7|8|9) | (3|30|31) )

→ *Register* → r0|r1|r2| … |r31|r00|r01|r02| … |r09

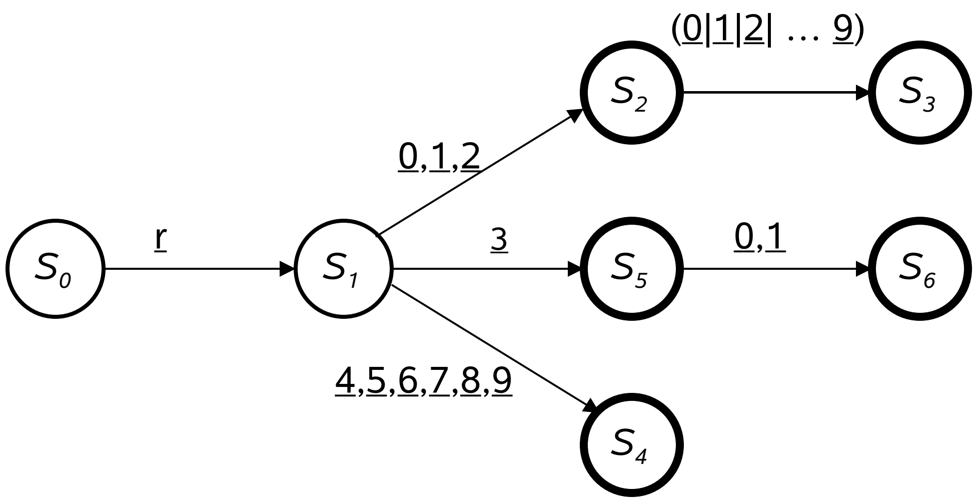
Produces a more complex DFA

- Has more states
- Same cost per transition
- Same basic implementation

# Tighter register specification (continued)

The DFA for

*Register* → r ( (0|1|2) (*Digit* | ε) | (4|5|6|7|8|9) | (3|30|31) )




- Accepts a more constrained set of registers
- Same set of actions, more states

# Tighter register specification (continued)

| $\delta$ | r | 0,1 | 2 | 3 | 4-9 | All others |
|---|---|---|---|---|---|---|
| $s_0$ | $s_1$ | $s_e$ | $s_e$ | $s_e$ | $s_e$ | $s_e$ |
| $s_1$ | $s_e$ | $s_2$ | $s_2$ | $s_5$ | $s_4$ | $s_e$ |
| $s_2$ | $s_e$ | $s_3$ | $s_3$ | $s_3$ | $s_3$ | $s_e$ |
| $s_3$ | $s_e$ | $s_e$ | $s_e$ | $s_e$ | $s_e$ | $s_e$ |
| $s_4$ | $s_e$ | $s_e$ | $s_e$ | $s_e$ | $s_e$ | $s_e$ |
| $s_5$ | $s_e$ | $s_6$ | $s_e$ | $s_e$ | $s_e$ | $s_e$ |
| $s_6$ | $s_e$ | $s_e$ | $s_e$ | $s_e$ | $s_e$ | $s_e$ |
| $s_e$ | $s_e$ | $s_e$ | $s_e$ | $s_e$ | $s_e$ | $s_e$ |

Runs in the same skeleton recognizer

*Table encoding RE for the tighter register specification*

# Goal

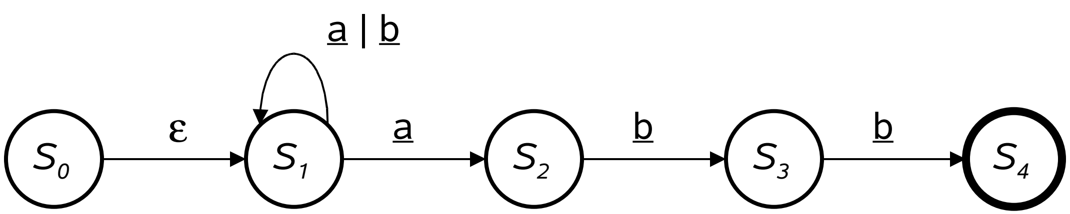
- We will show how to construct a finite state automaton to recognize any RE
- Overview:
  - → Direct construction of a nondeterministic finite automaton (NFA) to recognize a given RE
    - Requires ε-transitions to combine regular subexpressions
  - → Construct a deterministic finite automaton (DFA) to simulate the NFA
    - Use a set-of-states construction
  - → Minimize the number of states
    - Hopcroft state minimization algorithm
  - → Generate the scanner code
    - Additional specifications needed for details

# Non-deterministic Finite Automata

Each RE corresponds to a *deterministic finite automaton* (DFA)

- May be hard to directly construct the right DFA

What about an RE such as ( a | b )* abb ?




This is a little different

- $S_0$ has a transition on ε
- $S_1$ has two transitions on a

This is a *non-deterministic finite automaton* (NFA)

# Non-deterministic Finite Automata

- An NFA accepts a string *x* iff ∃ a path though the transition graph from $s_0$ to a final state such that the edge labels spell *x*
- Transitions on ε consume no input
- To "run" the NFA, start in $s_0$ and *guess* the right transition at each step
  - → Always guess correctly
  - → If some sequence of correct guesses accepts x then accept

Why study NFAs?

- They are the key to automating the RE→DFA construction
- We can paste together NFAs with ε-transitions

# Relationship between NFAs and DFAs

DFA is a special case of an NFA

- DFA has no ε transitions
- DFA's transition function is single-valued
- Same rules will work

DFA can be simulated with an NFA

→ *Obviously*

NFA can be simulated with a DFA *(less obvious)*

- Simulate sets of possible states
- Possible exponential blowup in the state space
- Still, one state per character in the input stream

# Automating Scanner Construction

To convert a specification into code:

1 Write down the RE for the input language

2 Build a big NFA

3 Build the DFA that simulates the NFA

4 Systematically shrink the DFA

5 Turn it into code

Scanner generators

- Lex and Flex work along these lines
- Algorithms are well-known and well-understood
- Key issue is interface to parser *(define all parts of speech)*
- You could build one in a weekend!

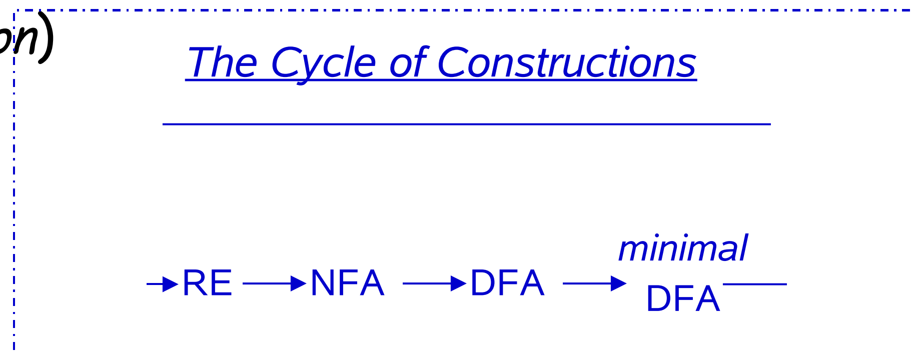
# Automating Scanner Construction

RE→ NFA *(Thompson's construction)*

- Build an NFA for each term
- Combine them with ε-moves

NFA → DFA *(subset construction)*

- Build the simulation




DFA → Minimal DFA

- Hopcroft's algorithm

DFA →RE *(Not part of the scanner construction)*

- All pairs, all paths problem