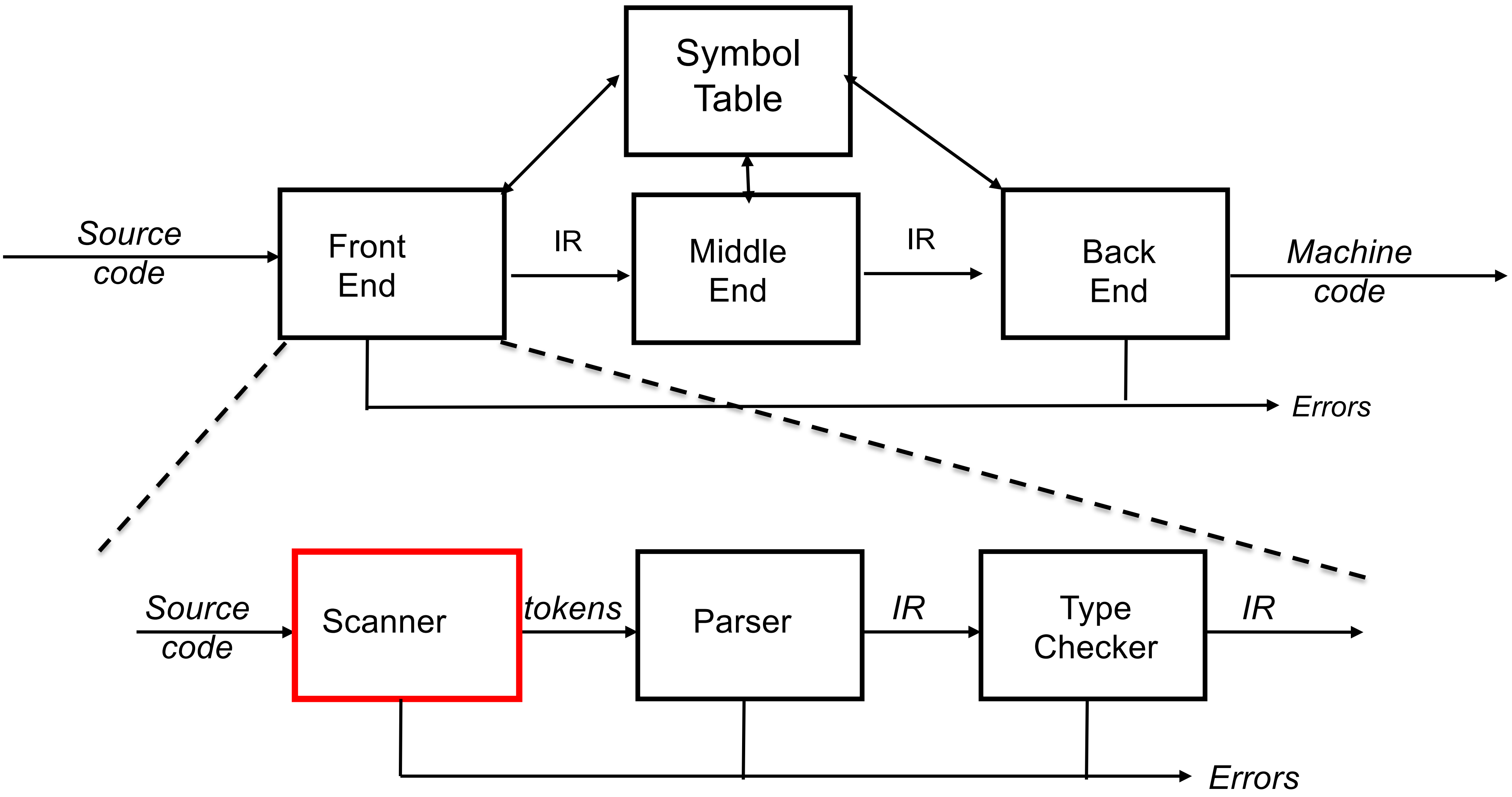
CMPSC 160
Translation of Programming Languages

Lecture 2: Lexical Analysis (Scanning)

Lexical Analysis (Scanning) in a Three-pass Compiler



Lexical Analysis (Scanning): Input and Output

- Input to the compiler

```
//simple example
while (sum < total)
{
  sum = sum + x*10;
}
```

- Or more accurately

```
//simple\bexample\nwhile\b(sum\b<\btotal)\b{\n\tsum\b=
\bsum\b+\bx*10;\n}\n
```

- The compiler scans the input file and produces a stream of ***tokens*** ***(categories of basic words in the langague)***

```
WHILE,LPAREN,<ID,sum>,LT,<ID,total>,RPAREN,LBRACE,
<ID,sum>,EQ,<ID,sum>,PLUS,<ID,x>,TIMES,<NUM,10>,
SEMICOL,RBRACE
```

What are unique here?
Natural Language VS. Programming Language

- Natural Language:
  - Word
    - Adjacent alphabetic letters are grouped together, left to right, to form a word. **Not all combinations of characters are legitimate words.**
    - A potential word, the word-building algorithm can determine its validity with a **dictionary lookup**.
  - Part of speech
    - If the word has a unique "part of speech" (noun, verb, …), dictionary lookup will also resolve that issue;
  - Non-uniqueness: a word like "stress" can be either a noun or a verb; It requires an understanding of meaning, for both the word and its context
- Programming Language
  - No dictionary lookup is needed. Instead, it is mostly **rule** based.
    - **Syntactic category**: Positive integer, Real number, Identifier, …
    - *Some* identifiers (e.g., if, while) may be reserved as keywords. These exceptions can be specified lexically: each with its own Syntactic category.
  - *Generally, **no context*** is required to determine.

Early Language like PL/I allows due parts of speech. More recent languages abandoned this idea. Think of Expressiveness VS. Efficient Scanning/parsing, …

Summary: Lexical Analysis (Scanning)

Scanner

- Transform a stream of characters into a stream of **words** in the input language.
- Each word must be classified into a syntactic category, or “part of speech”, **tokens**.
- Discards white space and comments
- Report errors and correlated information (e.g., line number)

The scanner is the **only** pass in the compiler to touch **every character** in the input program.

Compiler writers place a premium on **speed** in scanning, in part because the scanner’s input is larger.

Today and next lecture:

1) we will introduce regular expressions, a concise representation for describing the valid words in a programming language.
2) We will develop formal mechanisms -- Finite Automaton -- to generate scanners from regular expressions, either manually or automatically.

Lexical Concepts

- **Token**: Basic unit of syntax, syntactic output of the scanner
- **Lexeme**: A sequence of input characters which match to a pattern and generate the token
- **Pattern**: The rule that describes the set of strings that correspond to a token, i.e., specification of the token

Token	Lexeme	Pattern
WHILE	*while*	`while`
IF	*if*	`if`
ID	*i1*, *length*, *count*, *sqrt*	letter followed by letters and digits

How do we specify lexical patterns?

Some patterns are easy

- Keywords and operators
  - Specified as literal patterns: **if**, **then**, **else**, **while**, **=**, **+**, …

How do we specify lexical patterns?

Some patterns are more complex

- Identifiers
  - letter followed by letters and digits
- Numbers
  - Integer: An optional sign (which can be "+" or "-") followed by 0 or a digit between 1 and 9 followed by digits between 0 and 9
  - Decimal: An optional sign (which can be "+" or "-") followed by digit "0" or a nonzero digit followed by an arbitrary number of digits followed by a decimal point followed by an arbitrary number of digits

GOAL: We want to have **concise** descriptions of patterns, and we want to automatically construct the scanner from these descriptions

Regular Expressions

Regular expressions (REs) describe regular languages

Regular Expression (over alphabet Σ, alphabet could be the ASCII character set for example)

- ε (empty string) is a RE denoting the set $\{\varepsilon\}$
- If **a** is in Σ, then **a** is a RE denoting {**a**}
- If *x* and *y* are REs denoting languages *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 $L(x)^*$

<u>Precedence</u> is: *closure* first, then *concatenation*, then *alternation*

***All* <u>left-associative</u>**

$x \mid y^* z$ is equivalent to $x \mid ((y^*)\, z)$

Question Time ☺

- Examples of Regular Expressions
  - All strings of 1s and 0s
    $(0 \mid 1)^*$
  - All strings of 1s and 0s beginning with a 1
  - All strings of 0s and 1s containing at least two consecutive 1s
  - All strings of alternating 0s and 1s

Question Time ☺

- All strings of 1s and 0s

  $(0 \mid 1)^*$

- All strings of 1s and 0s beginning with a 1

  $1 (0 \mid 1)^*$

- All strings of 0s and 1s containing at least two consecutive 1s

  $(0 \mid 1)^* 1 1 (0 \mid 1)^*$

- All strings of alternating 0s and 1s

  $(\varepsilon \mid 1) (0 1)^* (\varepsilon \mid 0)$

Extensions to Regular Expressions

• $x+= x\ x^*$	denotes $L(x)^+$
• $x? = x \mid \varepsilon$	denotes $L(x) \cup \{\varepsilon\}$
• [abc] = a | b | c	matches one character in the square bracket
• a-z = a | b | c | ... | z	range
• [0-9a-z] = 0 | 1 | 2 | ... | 9 | a | b | c | ... | z	
• [^abc]	^ means negation matches any character except a, b or c
• . = [^\n]	dot matches any character except the newline \n means newline so dot is equivalent to [^\n]
• "["	matches left square bracket, meta-characters in double quotes become plain characters
• \[	matches left square bracket, meta-character after backslash becomes plain character

Regular Definitions

- We can define macros using regular expressions and use them in other regular expressions

$$
\begin{aligned}
&\mathit{Letter} &&\rightarrow (a|b|c|\ \ldots\ |z|A|B|C|\ \ldots\ |Z)\\
&\mathit{Digit} &&\rightarrow (0|1|2|\ \ldots\ |9)\\
&\mathit{Identifier} &&\rightarrow \mathit{Letter}\ (\ \mathit{Letter}\ |\ \mathit{Digit}\ )^*
\end{aligned}
$$

- **Important:** We should be able to order these definitions so that every definition uses only the definitions defined before it (i.e., no recursion)
- Regular definitions can be converted to basic regular expressions with macro expansion

Examples of Regular Expressions

Digit	$\rightarrow$ (0|1|2| … |9)
Integer	$\rightarrow$ (+|-)? (0| (1|2|3| … |9)(*Digit* *))
Decimal	$\rightarrow$ *Integer* *". "* *Digit* *
Real	$\rightarrow$ (*Integer* | *Decimal*) E (+|-)?*Digit* *
Complex	$\rightarrow$ "(" *Real* , *Real* ")"

From Regular Expressions to Scanners

- Regular expressions are useful for specifying patterns that correspond to our tokens
- We need to construct a program, our compiler for example, that recognizes these patterns and converts them into tokens
- We need it to read through the input *really fast*
- To solve this problem, **let's convert our regular expressions into state machines! –** state machines are really fast, it just requires a table lookup to process each character.

Deterministic Finite Automata (DFA)

- A set of states S
  - $S = \{ s_0 , s_1 , s_2 , s_e \}$
- A set of input symbols (an alphabet) Σ
  - $\Sigma = \{ R , 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 \}$
- A transition function $\delta : S \times \Sigma \rightarrow S$
  - Maps (state, symbol) pairs to states
  - $\delta = \{ (s_0 , R) \rightarrow s_1, (s_0 , 0\text{-}9) \rightarrow s_e ,(s_1 , 0\text{-}9) \rightarrow s_2 ,(s_1 , R) \rightarrow s_e ,$
    $(s_2 , 0\text{-}9) \rightarrow s_2 , (s_2 , R) \rightarrow s_e , (s_e , R \mid 0\text{-}9) \rightarrow s_e \}$
- A start state
  - s_0
- A set of final (or accepting) states
  - $Final = \{ s_2 \}$

A DFA accepts a word x iff there exists a path in the transition graph from start state to a final state such that the edge labels along the path spell out x

Example

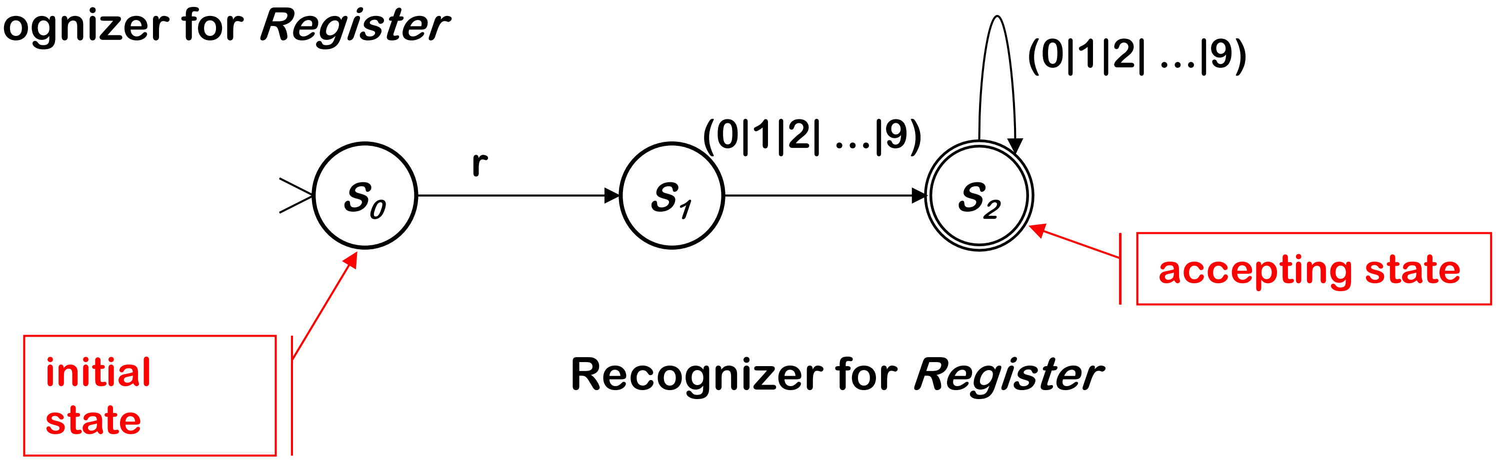
Consider the problem of recognizing register names in an assembler

$Register \rightarrow$ r (0|1|2| … |9) (0|1|2| … |9)*

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

Each RE corresponds to a recognizer (or Deterministic Finite Automata (DFA))

- **Recognizer for *Register***

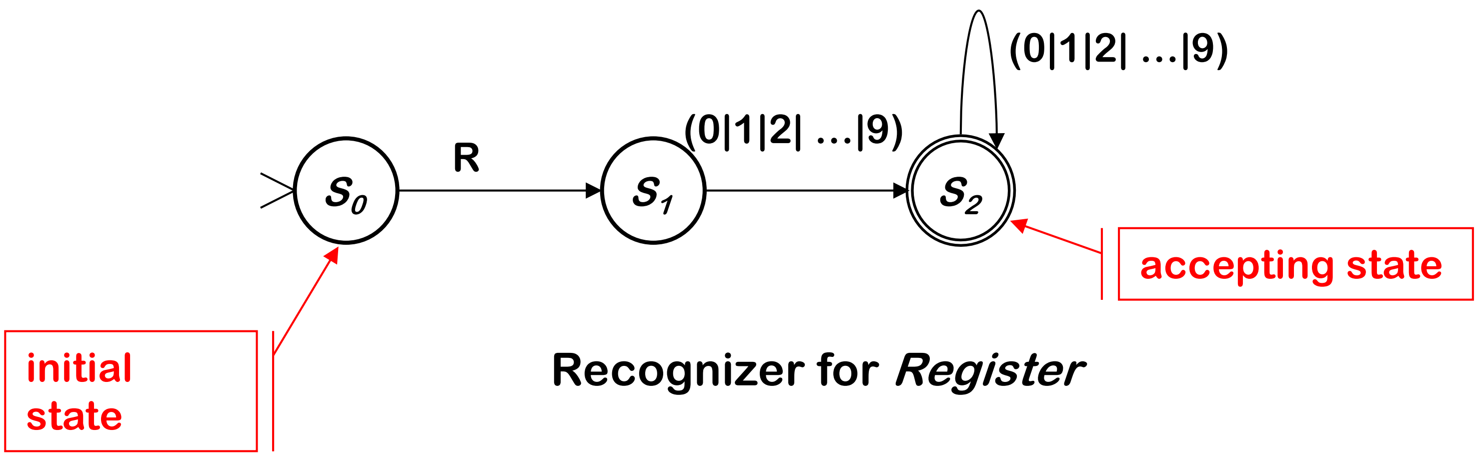


Recognizer for *Register*

Example

DFA simulation

- Start in state s_0 and follow transitions on each input character
- DFA accepts a word *x iff* *x* leaves it in a final state (s_2)



Recognizer for *Register*

- "R17" takes it through s_0 , s_1 , s_2 and accepts
- "R" takes it through s_0 , s_1 and fails
- "A" takes it straight to s_e
- "R17R" takes it through s_0 , s_1 , s_2 , s_e and rejects

Simulating a DFA

```
state = s0;
char = get_next_char();
while (char != EOF) {
  state = δ(state,char);
  char =get_next_char();
}
if (state ∈ Final)
  report acceptance;
else
   report failure;
```

We can store the transition table in a two-dimensional array:

δ	R	0,1,2,3, 4,5,6, 7,8,9	*other*
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

$Final = \{ s_2 \}$

We can also store the final states in an array

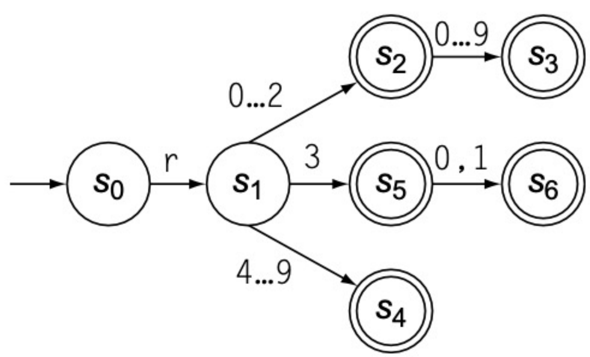
- *The recognizer translates directly into code*
- *To change DFAs, just change the arrays*
- *Takes O(|x|) time for input string x*

Example

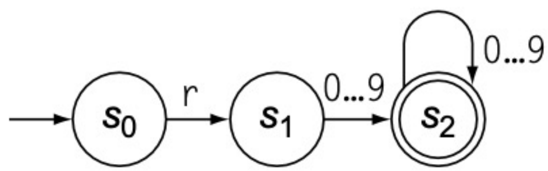
- On a real computer, however, the set of register names is severely limited— say, to 32, 64, 128, or 256 registers.
  - One way for a scanner to check validity of a register name is to convert the digits into a number and test whether or not it falls into the range of valid register numbers.
  - The alternative is to adopt a more precise re specification, such as:

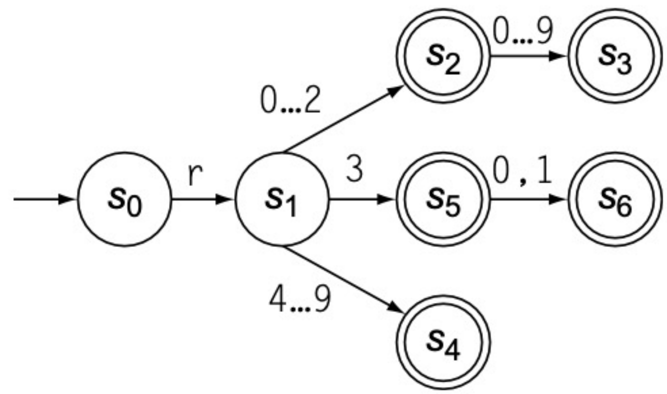
$$r\,([0\ldots 2]\,([0\ldots 9]\,|\,\epsilon)\,|\,[4\ldots 9]\,|\,(3\,(0\,|\,1\,|\,\epsilon)))$$

- The corresponding DFA looks like:



Performance Analysis





Which DFA is better in terms of performance? Why?

The cost of operating an DFA is proportional to the length of the input, not to the length or complexity of the re that generates the DFA!!!

On modern computers, the speed of memory accesses often governs the speed of computation. A smaller recognizer may fit better into the processor's cache memory.

More Complicated Regular Expression

RE: $(a \mid b)^* abb$

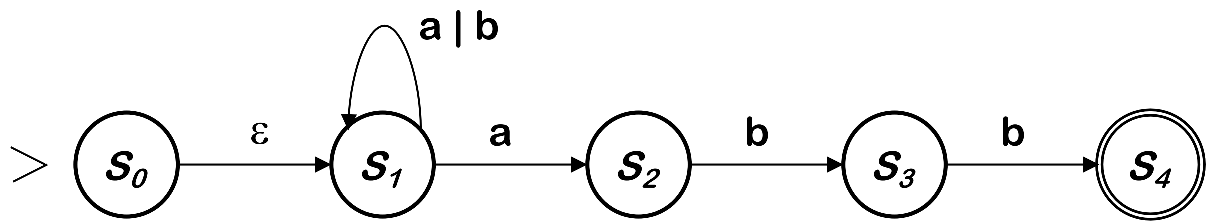
Can you automatically generate a DFA for it?

Non-deterministic Finite Automata (NFA)

Why study NFAs?

- They are the key to automating the RE→DFA construction

Non-deterministic Finite Automata (NFA) for the RE $(a \mid b)^* abb$



This is a little different

- S_0 has a transition on ε (empty string)
  - ε-transitions are allowed
- S_1 has two transitions on "a"
  - Transition function $\delta : \boldsymbol{S} \times \Sigma \rightarrow 2^S$ maps (state, symbol) pairs to sets of states

Powerset of *N*
the set of all subsets of N, denoted 2^N

NFA

- Ideally, each time the NFA must make a nondeterministic choice, it follows the transition that leads to an accepting state for the input string, if such a transition exists.

- In practice, each time the NFA must make a nondeterministic choice, the NFA clones itself to pursue each possible transition. Thus, for a given input character, the NFA is in a specific set of states, taken across **all of its clones**. In this model, the NFA pursues all paths concurrently.

- At any point, we call the specific set of states in which the NFA is active its ***configuration***. When the NFA reaches a configuration in which it has exhausted the input and one or more of the clones has reached an accepting state, the NFA accepts the string.

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

- *Obvious*

NFA can be simulated with a DFA

- *Less obvious*

NFA vs. DFA Scanners

- Given a regular expression *r* we can convert it to an NFA of size $O(|r|)$
- Given an NFA we can convert it to a DFA of size $O(2^{|r|})$
- We can simulate a DFA on string *x* in $O(|x|)$ time
- We can simulate an NFA (constructed by Thompson's construction) on a string *x* in $O(|N| \times |x|)$ time

Recognizing input string *x* for regular expression *r*

Automaton Type	Space Complexity	Time Complexity
NFA	$O(\lvert r\rvert)$	$O(\lvert r\rvert \times \lvert x\rvert)$
DFA	$O(2^{\lvert r\rvert})$	$O(\lvert x\rvert)$

Relationship between RE/NFA/DFA

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

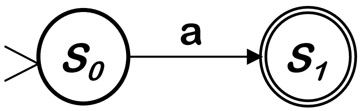
- Union together paths from s_0 to a final state

The Cycle of Constructions

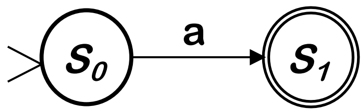
You have learned all these algorithms in CMPSC138!

RE → NFA using Thompson's Construction

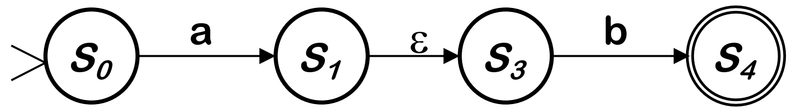
Ken Thompson, CACM, 1968



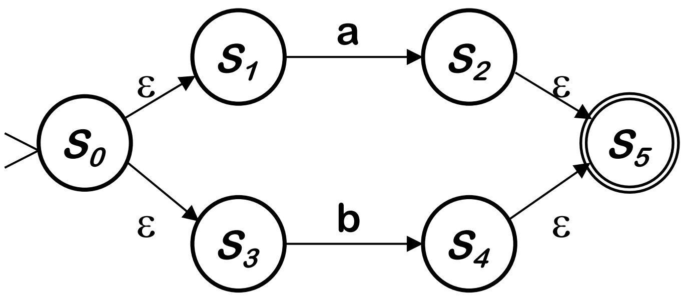
NFA for a



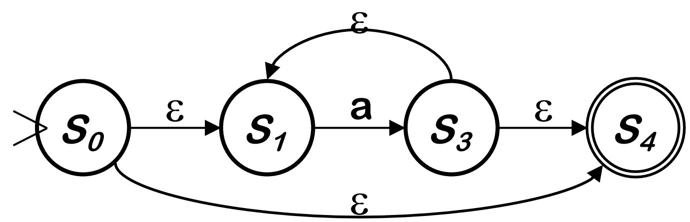
NFA for b



NFA for ab



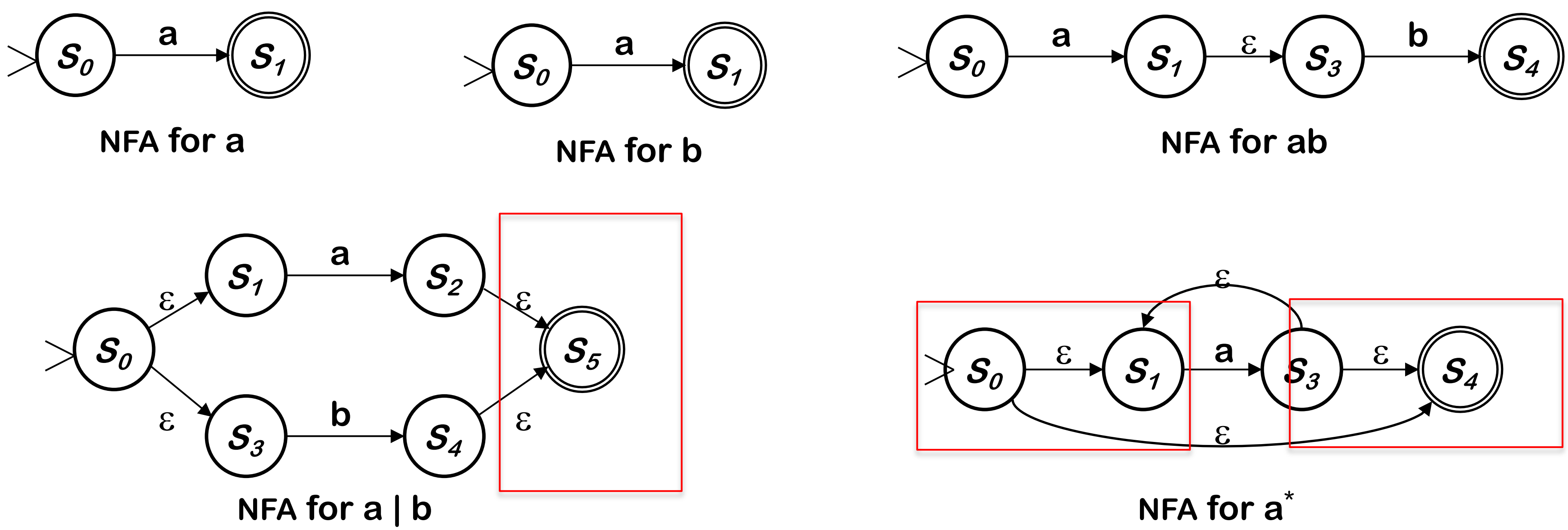
NFA for a | b



NFA for a^*

- It has **a template** for building the NFA that corresponds to a single-letter RE, and a transformation on NFAs that models the effect of each basic re operator: concatenation, alternation, and closure.

Understanding the deep insights behind these algorithm designs



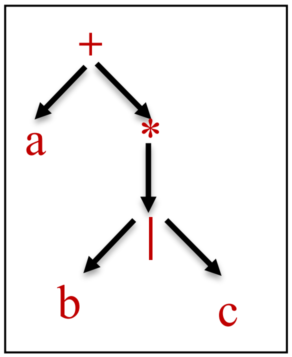
- Key insight: to simplify the combination process, we want to have a starting node with no incoming edges and a single accepting node.

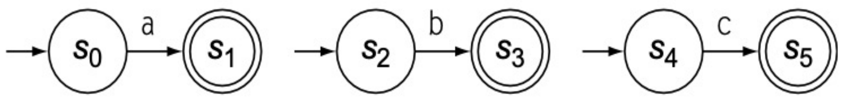
Thompson's Construction

- Two key steps:
  - The construction begins by building trivial NFAs for each character in the input RE.
  - Next, it applies the transformations for alternation, concatenation, and closure to the collection of trivial NFAs in the order dictated by precedence and parentheses.

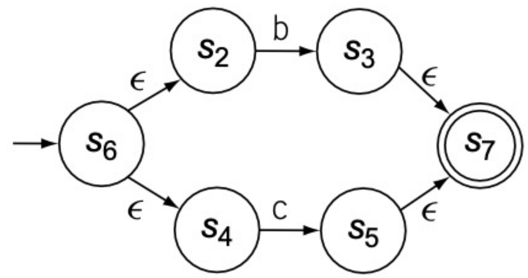
Thompson's Construction: End-to-End Example

Let's try a(b | c)*

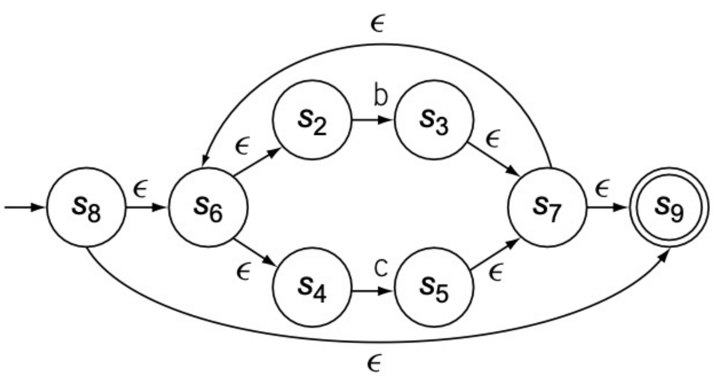




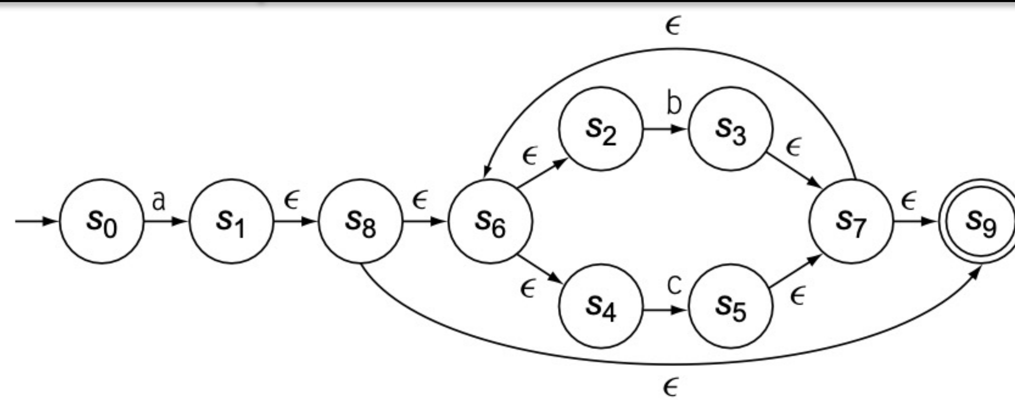
(a) NFAs for "*a*", "*b*", and "*c*"



(b) NFA for "*b* | *c*"



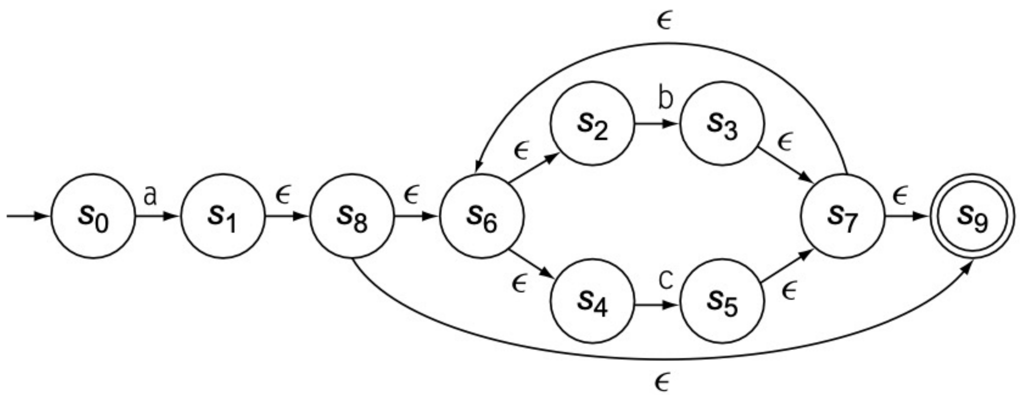
(c) NFA for "(*b* | *c*)*"



(d) NFA for "*a*(*b* | *c*)*"

NFA -> DFA

- $a(b \mid c)^*$



Question: can you build a DFA for (a | b) a* ? Can this process be done automatically for any regular expression?

NFA → DFA with Subset Construction

- The complex part of the construction is the derivation of the set of DFA states from the NFA states *N*, and the derivation of the DFA transition function.

```
q0 ← ϵ-closure({n0});
Q ← q0;
WorkList ← {q0};

while (WorkList ≠ Ø ) do
    remove q from WorkList;
    for each character c ∈ Σ do
        t ← ϵ-closure(Delta(q,c));
        T[q,c] ← t;
        if t ∉ Q then
            add t to Q and to WorkList;
    end;
end;
```

Take ε-closure of the start state of the NFA and make it the **start state** of the DFA

Termination condition

for each new state *q* of the DFA and each a $\in \Sigma$, take the ε-closure of the result and make it a state of the DFA.

Record the state transition rule.

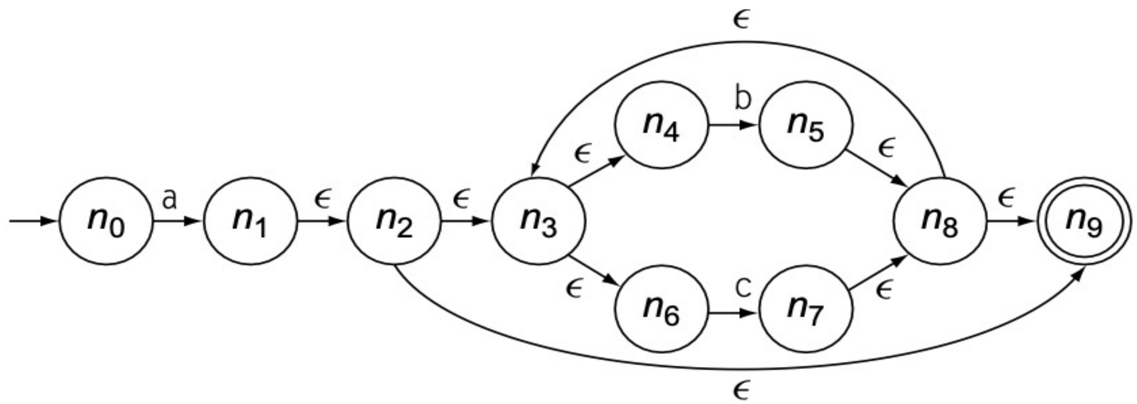
The pseudocode for subset construction

NFA $\rightarrow$ DFA with Subset Construction

- The subset construction is an example of fixed-point computation.
  - These computations terminate when they reach a state where further iteration produces the same answer—a “fixed point” in the space of successive iterates.
  - Fixed-point computations play an important and recurring role in compiler construction.

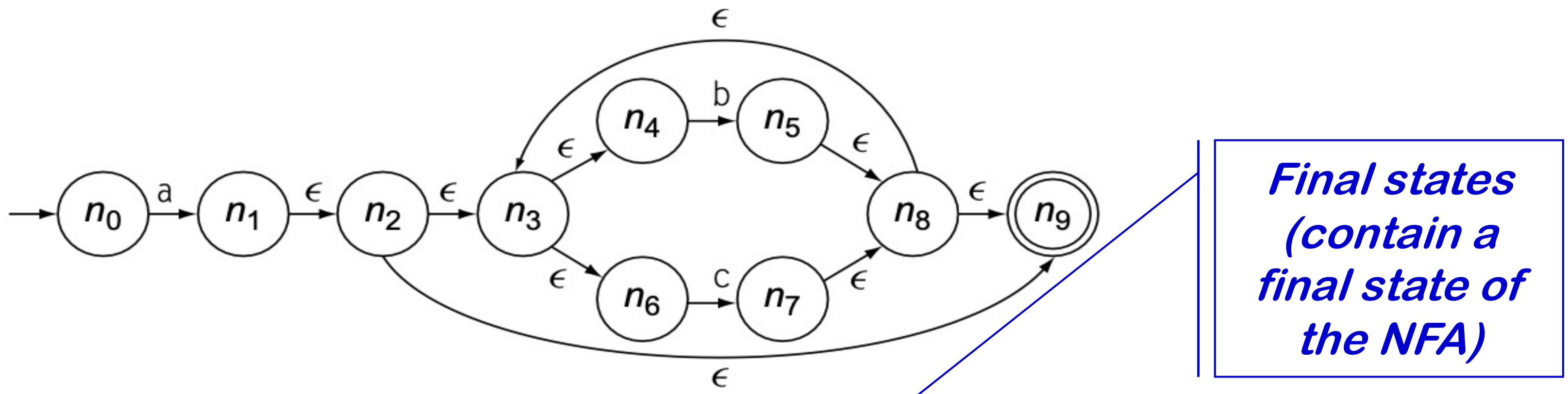
- Termination arguments for fixed-point algorithms usually depend on known properties of the domain.
  - For subset construction, why we are 100% sure the algorithm will always terminate?
  - Maximum #iteration = 2^N, where N is the number of NFA states. It may, of course, reach a fixed point and halt more quickly than that.

Example: NFA →DFA with Subset Construction



Iteration	DFA State	Contains NFA states	ε-closure(move(s,a))	ε-closure(move(s,b))	ε-closure(move(s,b))

Example: NFA →DFA with Subset Construction

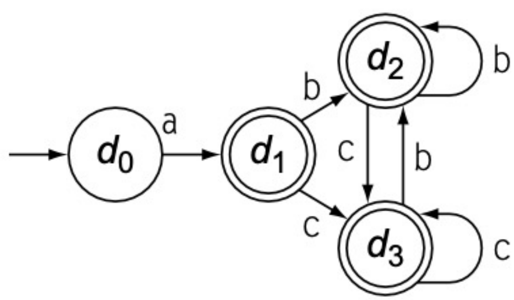


Iteration	DFA State	Contains NFA states	ε-closure(move(s,a))	ε-closure(move(s,b))	ε-closure(move(s,b))
1	d_0	{0}	{1,2,3,4,6,9}	{}	{}
2	d_1	{1,2,3,4,6,**9**}	{}	{5,8,9,3,4,6}	{7,8,9,3,4,6}
3	d_2	{5,8,**9**,3,4,6}	{}	{5,8,9,3,4,6}	{7,8,9,3,4,6}
4	d_3	{7,8,**9**,3,4,6}	{}	{5,8,9,3,4,6}	{7,8,9,3,4,6}

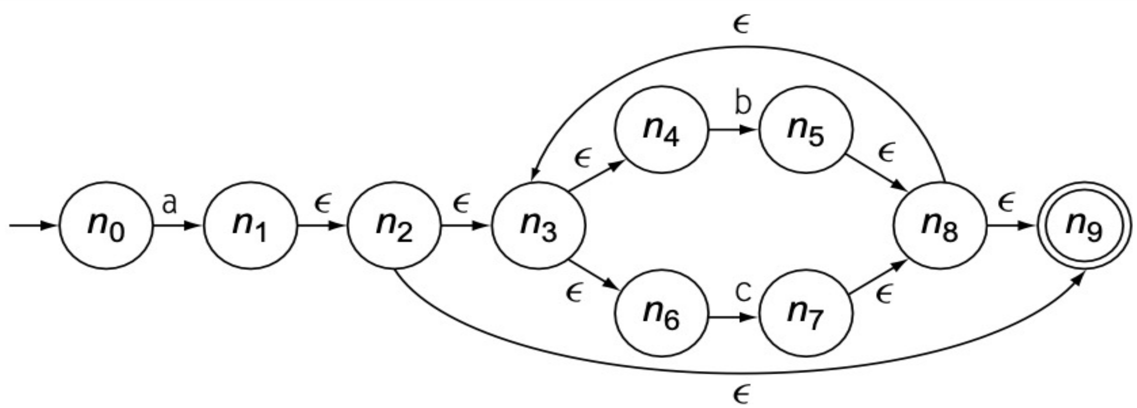
Iteration 3 does not add a new state, and all the states are processed, so the algorithm halts

Example: NFA →DFA with Subset Construction

Iteration	DFA State	Contains NFA states	ε-closure(move(s,a))	ε-closure(move(s,b))	ε-closure(move(s,b))
1	d_0	{0}	{1,2,3,4,6,9}	{}	{}
2	d_1	{1,2,3,4,6,**9**}	{}	{5,8,9,3,4,6}	{7,8,9,3,4,6}
3	d_2	{5,8,**9**,3,4,6}	{}	{5,8,9,3,4,6}	{7,8,9,3,4,6}
4	d_3	{7,8,**9**,3,4,6}	{}	{5,8,9,3,4,6}	{7,8,9,3,4,6}
5	d_4	{}	{}	{}	



Any possible improvement for subset construction?



Iteration	DFA State	Contains NFA states	ε-closure(move(s,a))	ε-closure(move(s,b))	ε-closure(move(s,b))
1	d_0	{0}	{1,2,3,4,6,9}	{}	{}
2	d_1	{1,2,3,4,6,**9**}	{}	{**5**,8,9,3,4,6}	{**7**,8,9,3,4,6}
3	d_2	{5,8,**9**,3,4,6}	{}	{**5**,8,9,3,4,6}	{**7**,8,9,3,4,6}
4	d_3	{7,8,**9**,3,4,6}	{}	{**5**,8,9,3,4,6}	{**7**,8,9,3,4,6}

We need to repeatedly compute the ε-closure of node 5 and 7.

Any possible improvement for subset construction?

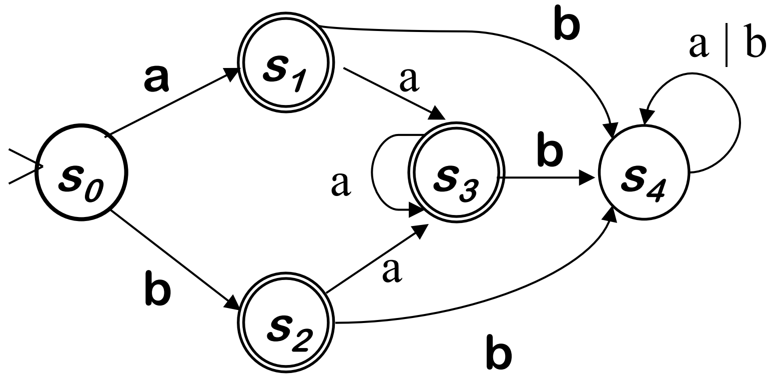
Solution: An offline algorithm that computes ε-closure($\{n\}$) for each state n in the transition graph. The algorithm is another example of a **fixed-point computation**.

```
for each state n ∈ N do
    E(n) ← {n};
end;
WorkList ← N;
while (WorkList ≠ ∅) do
  remove n from WorkList;
  t ← {n} ∪ ⋃_{n →ε p ∈ δ_N} E(p);
  if t ≠ E(n)
      then begin;
         E(n) ← t;
         WorkList ← WorkList ∪ {m | m →ε n ∈ δ_N};
      end;
end;
```

An Offline Algorithm for ε-closure.

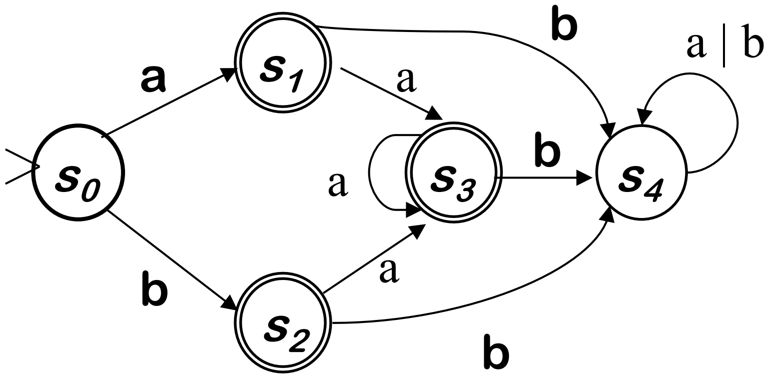
Example: NFA →DFA with Subset Construction

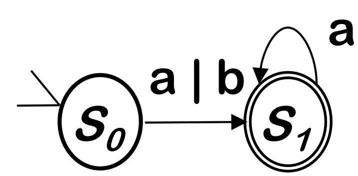
The DFA for (a | b) a*



- Not much bigger than the original but
  - In the worst case the number of states in the DFA is 2^Q (where Q is the number of states in the NFA)
- All transitions are deterministic
- Use same code skeleton as before

DFA → minimal DFA





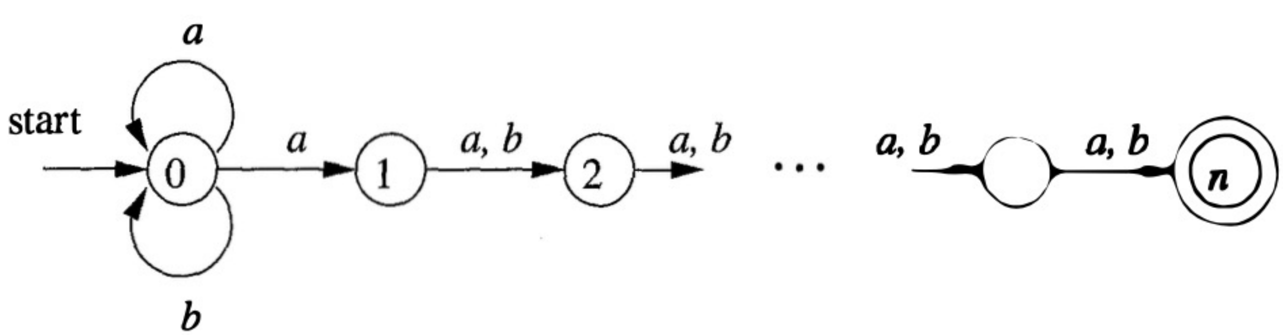
Hopcroft's algorithm: **automatically minimize a DFA to a minimal state DFA.**

The key is to find a representative set of non- distinguishable state.

Understanding the deep insights behind these algorithm designs

- Combining subset construction (NFA→DFA) and hopcroft's algorithm (DFA→ minimal DFA), at worst time, could still lead to a state exponential blowup, but when?
- Understanding the worst case would let you know the limitations of regular expressions.

$$L_n = (\mathbf{a}|\mathbf{b})^*\mathbf{a}(\mathbf{a}|\mathbf{b})^{n-1}$$



- Could you prove that any DFA for the language Ln must have at least 2n states?

Building Faster Scanners from DFAs

Table-driven recognizers (which store the transition function in an array) waste a lot of effort

- Read (& classify) the next character
- Find the next state
- Assign to the state variable
- Branch back to the top

We can do better

- Encode state & actions in the code
- Do transition tests locally
- Generate ugly, spaghetti-like code (it is OK, this is automatically generated code)
- Takes fewer operations per input character

```
state = s0;
string = ε;
char  = get_next_char();
while (char != eof) {
  state = δ(state,char);
  string = string + char;
  char = get_next_char();
}
if  (state in Final) then
  report acceptance;
else
   report failure;
```

Building Faster Scanners from the DFA

A direct-coded recognizer for Num

```
        goto s0;
    s0: string ← ε;
        char ← get_next_char();
        if (char = '+' || char= '–')
          then goto s1;
          else goto se;
    s1: string ← string+ char;
        char ← get_next_char();
        if ( '0' ≤ char ≤ '9')
          then goto s2;
          else goto se;
```

- Many fewer operations per character
- State is encoded as the location in the code

```
s2: string ← string+ char;
    char ← get_next_char();
    if ( '0' ≤ char ≤ '9')
      then goto s2;
      else if (char = '.')
        then goto s3
        else goto se;
s3 : string ← string+ char;
    char ← get_next_char();
    if ( '0' ≤ char ≤ '9')
      then goto s3;
      else if (char = eof)
         then report
acceptance;
         else goto se;
se: print error message;
    return failure;
```

Limits of Regular Languages

If REs are so useful … Why not use them for everything?

- If we add balanced parenthesis to the expressions grammar, we cannot represent it using regular expressions:

$$
\begin{aligned}
&Id \rightarrow [\text{a-zA-Z}]\ ([\text{a-zA-z}] \mid [0\text{-}9])^* \\
&Num \rightarrow [0\text{-}9]+ \\
&Term \rightarrow Id \mid Num \\
&Op \rightarrow \text{“}+\text{”} \mid \text{“}-\text{”} \mid \text{“}*\text{”} \mid \text{“}/\text{”} \\
&Expr \rightarrow Term \mid Expr\ Op\ Expr \mid \text{“(“}\ Expr\ \text{“)”}
\end{aligned}
$$

- A DFA of size *n* cannot recognize balanced parenthesis with nesting depth greater than *n*
  - Not all languages are regular

Solution: Use a more powerful formalism: *context-free grammars*

What is hard about lexical analysis?

Poor language design can complicate scanning

- Reserved words are important
  - In PL/I there are no reserved keywords, so you can right a valid statement like:

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

- Significant blanks
  - In Fortran blanks are not significant

    ```
    do 10 i = 1,25
    do 10 i = 1.25
    ```

    do loop

    assignment to variable do10i

- Closures
  - Limited identifier length adds states to the automata to count length

Summary of Lexical Analysis

The main ideas here are that:

a) When we are done with scanning, we will have a stream of tokens

b) These tokens are found by searching for a match to some regular expression in the input program. The matches can be prioritized (for example, to handle keywords)

c) To implement this efficiently, we can convert the regular expressions into state machines (which are implemented as a table lookup)

d) Luckily for us, other people have done this for us and built this functionality into a set of tools

Scanner Implementations

- An automatically generated scanner.
- A hand-coded approach.

- Course: the use of generated scanners
- Most commercial compilers and open-source compilers use hand- crafted scanners.
  - performance gain VS. convenience
  - scanners are simple and they change infrequently

fas can be viewed as specifications for a recognizer. However, they are not particularly concise specifications. To simplify scanner implementation, we need a concise notation for specifying the lexical structure of words, and a way of turning those specifications into an fa and into code that imple- ments the fa. The remaining sections of this chapter develop precisely those ideas.

a...z,

A...Z, *s* *s*0...9

$_0$ a...z, $_1$ A...Z

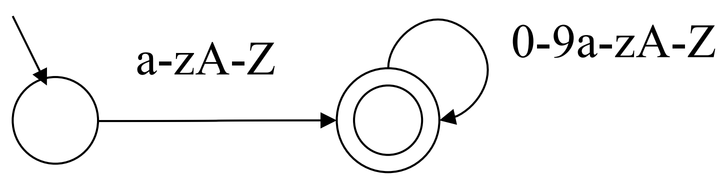
To eliminate any ambiguity, parentheses have highest precedence, followed by closure, concatenation, and alternation, in that order.

Lexical Analysis (Scanning)

- Compiler uses a set of patterns to specify valid tokens
  - tokens: LPAREN, WHILE, ID, NUM, etc.

- Each pattern is specified as a **regular expression**
  - LPAREN should match: `(`
  - WHILE should match: `while`
  - ID should match: [a-zA-Z][0-9a-zA-Z]*

- It uses **finite automata** to recognize these patterns



ID automaton

Lexical Analysis (Scanning)

- During the scan the lexical analyzer gets rid of the ***white space*** (`\b`, `\t`, `\n`, etc.) and ***comments***
- Important additional task: Error messages!
  - `Var%1` → Error! Not a token!
  - `whle` → Error? It matches the identifier token.
- Natural language analogy: Tokens correspond to words and punctuation symbols in a natural language

Operations on Languages

Operation	Definition
Union of L and M *Written* $L \cup M$	$L \cup M = \{s \mid s \in L \text{ or } s \in M\}$
Concatenation of L and M *Written LM*	$LM = \{st \mid s \in L \text{ and } t \in M\}$
Exponentiation of L *Written* L^i	$L^i = \begin{cases} \{\varepsilon\} \text{ if } i = 0 \\ L^{i-1}L \text{ if } i > 0 \end{cases}$
Kleene closure of L *Written* L^*	$L^* = \bigcup_{0 \le i \le \infty} L^i$

Is this a regular expression (language)?
$x^n\ y^n\ z$

Automating Scanner Construction

To build a scanner:

1 Write down the RE that specifies the tokens

2 Translate the RE to an NFA

3 Build the DFA that simulates the NFA

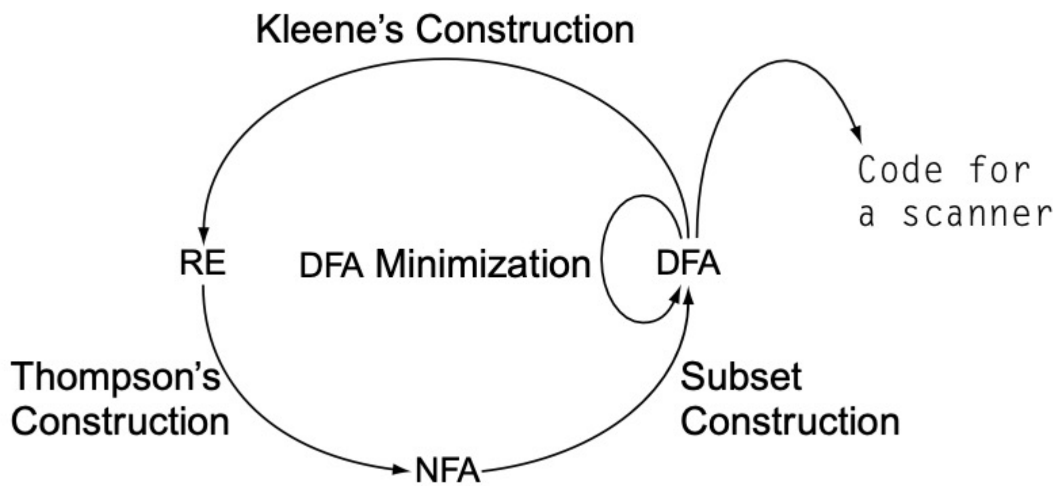
4 Systematically shrink the DFA

5 Turn it into code

Scanner generators

- Lex , Flex, Jlex, and Jflex work along these lines
- Algorithms are well-known and well-understood

From Regular Expression to Scanner



- Three key steps
  - Thompson's construction: RA → NFA
  - The subset construction: NFA → DFA
  - Hopcroft's algorithm: minimizing a DFA.
- To establish the equivalence of RE and DFA
  - Kleene's construction: DFA → RA.

Recap on lexical analysis (Scanner)

Key concepts:

- **Token**: Basic unit of syntax, syntactic output of the scanner
- **Pattern**: The rule that describes the set of strings that correspond to a token, i.e., specification of the token.
- **Lexeme**: A sequence of input characters which match to a pattern and generate the token

GOAL: We want to have concise descriptions of patterns (e.g., numbers, keywords, identifiers…), and we want to automatically construct the scanner from these descriptions

Solution: the key is regular expression (RA) and DFA.

→ **RE** → **NFA** → **DFA** → ***minimal*** **DFA**

all these algorithms in CMPSC138