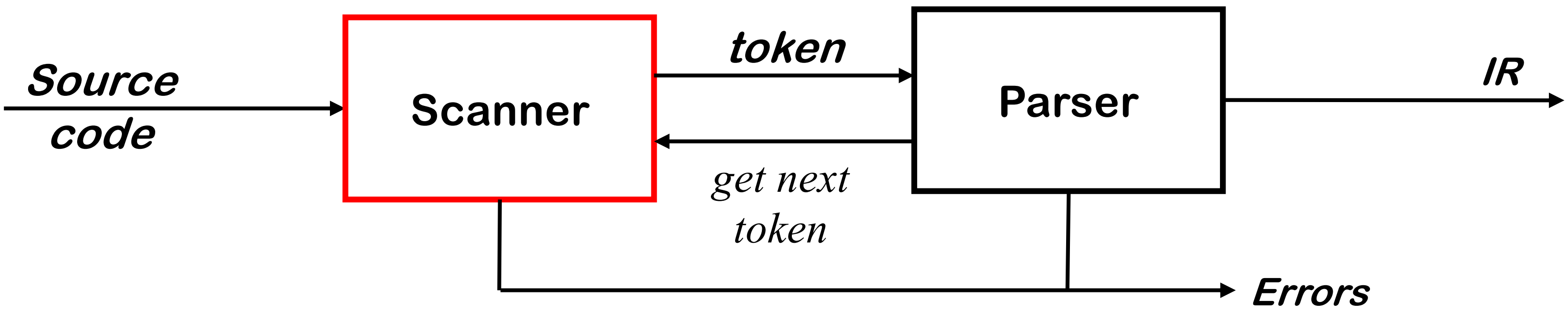
Computer Science 160
Translation of Programming Languages

Instructor: Christopher Kruegel

Lexical Analysis (Scanning)

First Phase: Lexical Analysis (Scanning)

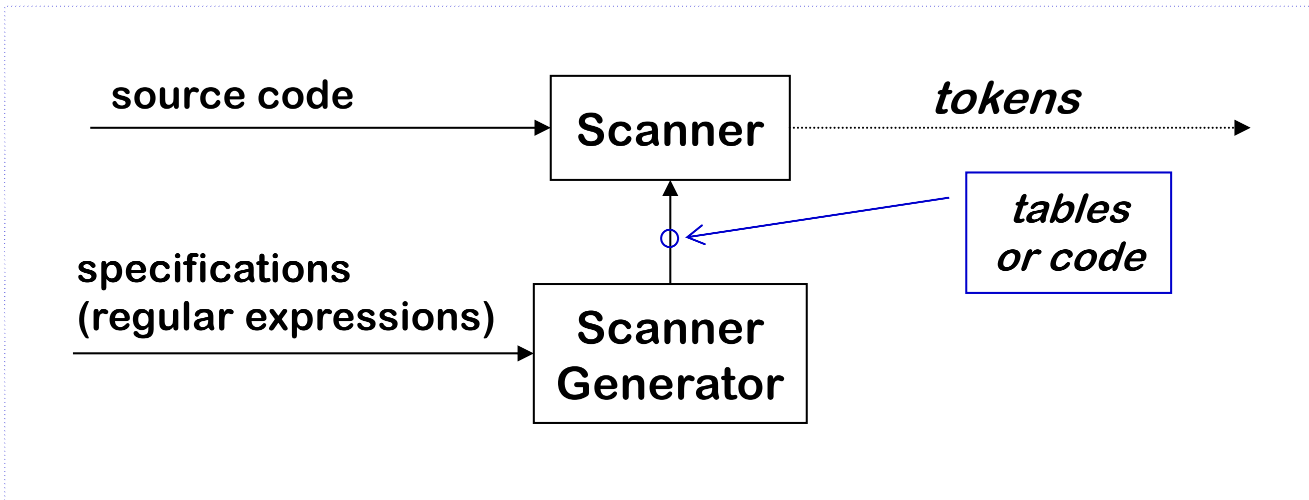


<u>Scanner</u>

- Maps stream of characters into words
  - Basic unit of syntax
- Characters that form a word are its *lexeme*
- Its syntactic category is called its *token*
- Scanner discards white space and comments

Why Lexical Analysis?

- By separating context free syntax from lexical analysis
  - We can develop efficient scanners
  - We can automate efficient scanner construction
  - We can write simple specifications for tokens



What are Tokens?

- Token: Basic unit of syntax … they are the atoms
  - Keywords

    `if, while, ...`
  - Operators

    `+, *, <=, ||, ...`
  - Identifiers (names of variables, arrays, procedures, classes)

    `i, i1, j1, count, sum, ...`
  - Numbers

    `12, 3.14, 7.2E-2, ...`

What are Tokens?

- Tokens are terminal symbols for the parser
  - Tokens are treated as indivisible units in the grammar defining the source language

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

number, **id**, +, **-** are tokens passed from scanner to parser. They form the terminal symbols of this simple grammar.

Lexical 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

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

Tokens can have Attributes

- A problem

```
if (i == j)
  z = 0;
else
  z = 1;
```

becomes

IF, LPAREN,ID,EQEQ,ID,RPAREN,
ID,EQ,NUM,SEMICOLON,ELSE,
ID,EQ,NUM,SEMICOLON

- If we send this output to the parser, is it enough? Where are the variable names, procedure, names, etc.? All identifiers look the same.
- Tokens can have **attributes** that they can pass to the parser (using the symbol table)

IF, LPAREN,<ID, i>,EQEQ,<ID, j>,RPAREN,

<ID, z>,EQ,<NUM,0>,SEMICOLON,ELSE,

<ID,z>,EQ,<NUM,1>,SEMICOLON

How do we specify lexical patterns?

Some patterns are easy

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

Specifying Lexical Patterns

Some patterns are more complex

- Identifiers
  - letter followed by letters and digits
- Numbers
  - Integer: 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)

- 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* is an RE denoting *L(x)*
  - $x \mid y$ 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***

All **left-associative**

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

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^**	$L^* = \cup_{0 \le i \le \infty} L^i$
Positive closure of L *Written L^+*	$L^+ = \cup_{1 \le i \le \infty} L^i$

Examples of Regular Expressions

- All strings of 1s and 0s
- 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

Examples of Regular Expressions

- 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
- . (dot) matches any character except the newline
- . = [^\n] \n means newline, 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

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

- **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 → (0|1|2| … |9)

Integer → (+|-)? (0| (1|2|3| … |9)(*Digit* *))

Decimal → *Integer* “.” *Digit* *

Real → (*Integer* | *Decimal*) E (+|-)?*Digit* *

Complex → “(“ *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
- When have a reasonably small number of tokens (on the order of 100?) and we are going to search through every piece of code every time we compile anything, then we have a huge amount of input to search
- 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

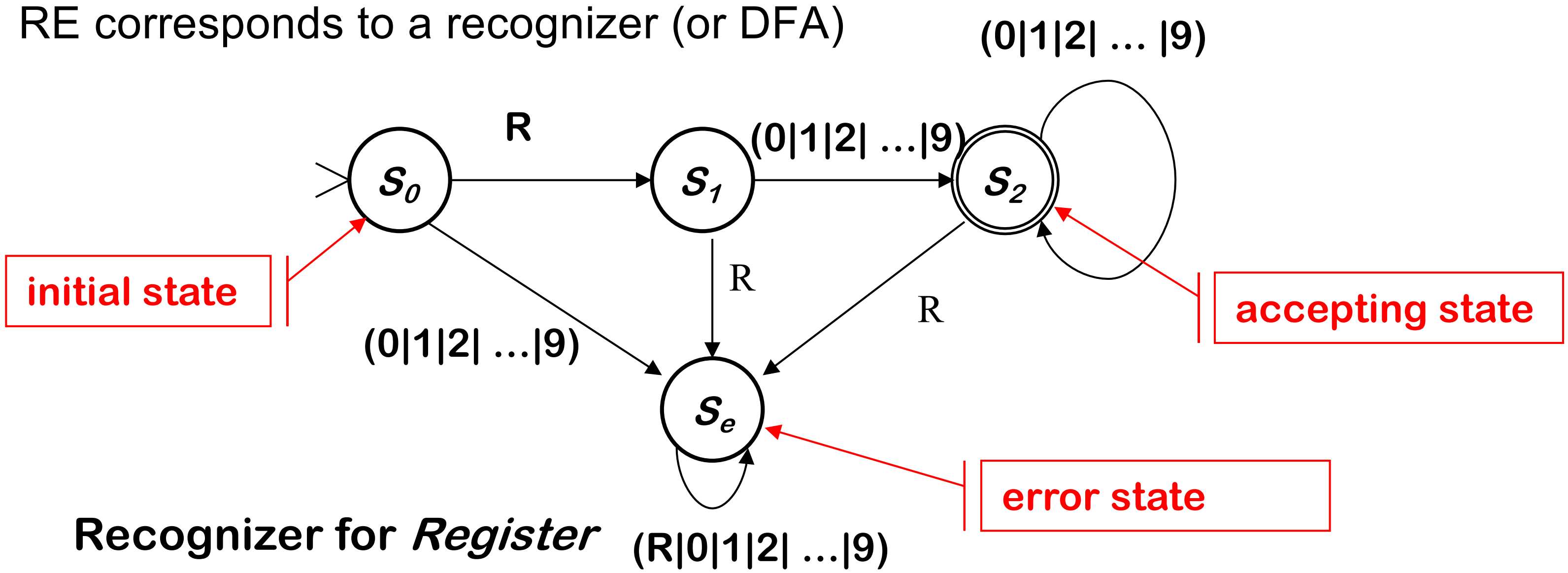
Example

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

RE corresponds to a recognizer (or DFA)



Recognizer for *Register*

Deterministic Finite Automata (DFA)

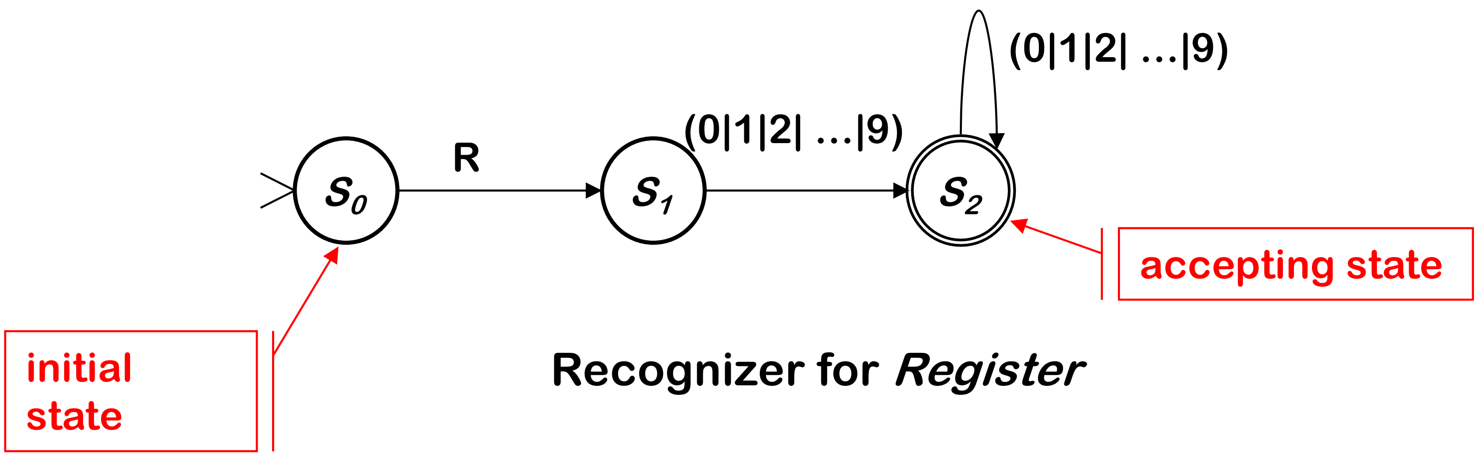
- A set of states S
  - $S = \{ s_0 , s_1 , s_2 , s_e \}$
- A set of input symbols (an alphabet) Σ
  - $\Sigma = \{ R , 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 \}$
- A transition function $\delta : S \times \Sigma \rightarrow S$
  - Maps (state, symbol) pairs to states
  - $\delta = \{ (s_0 , \mathrm{R})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

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*

Recognizing Longest Accepted Prefix


```
accepted = false;
current_string = ε; // empty string
state = s0; // initial state
if (state ∈ Final) {
  accepted_string = current_string;
  accepted = true;
}
char =get_next_char();
while (char != EOF) {
   state = δ(state,char);
   current_string = current_string + char;
   if (state ∈ Final) {
      accepted_string = current_string;
      accepted = true;
   }
   char =get_next_char();
}
if accepted
  return accepted_string;
else
   report error;
```

Given an input string, this simulation algorithm returns the **longest accepted prefix**

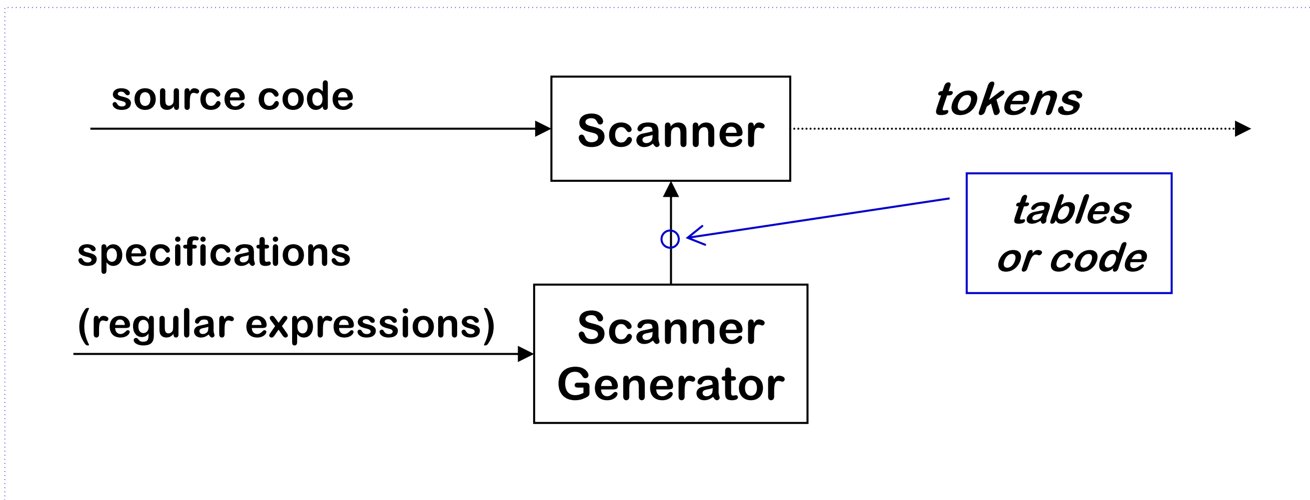
δ	R	0,1,2,3, 4,5,6, 7,8,9	*other*
s_0	s_1	s_e	s_e
s_1	s_e	s_2	s_e
s_2	s_e	s_2	s_e
s_e	s_e	s_e	s_e

$$Final = \{ s_2 \}$$

Given the input "R17R" , this simulation algorithm returns "R17"

Lexical Analysis

- Specify tokens using Regular Expressions
- Translate Regular Expressions to Finite Automata
- Use Finite Automata to generate tables or code for the scanner



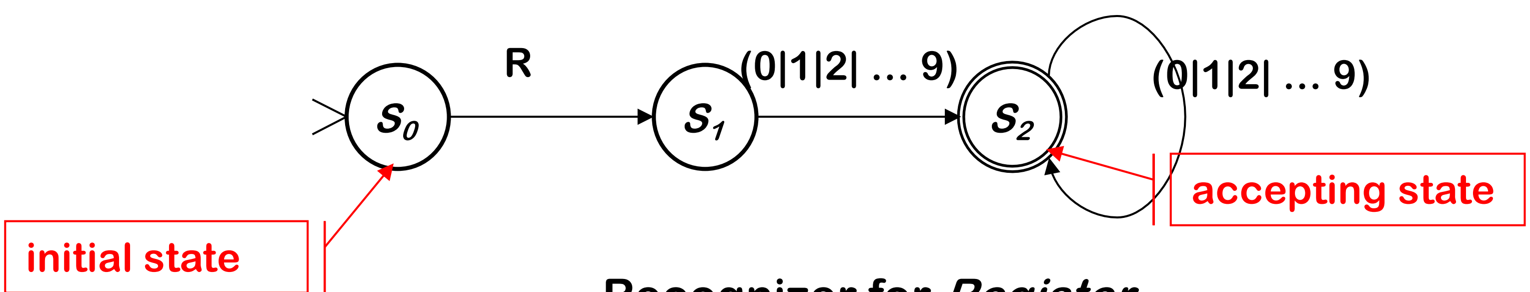
Example

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

RE corresponds to a recognizer (or DFA)



Recognizer for *Register*

Tighter Register 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* $\rightarrow$ R (
  (4|5|6|7|8|9)
  | (0|1|2) (0|1|2| … | 9 | ε)
  | (3 (0|1|ε))
)
- *Register* $\rightarrow$ R0|R1|R2| … |R31|R00|R01|R02| … |R09

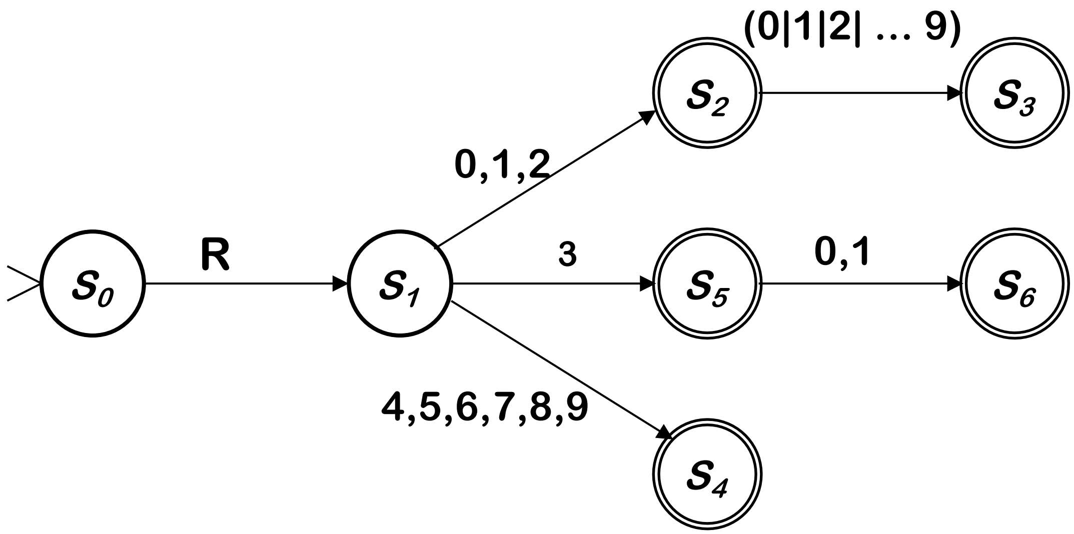
Produces a more complex DFA

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

Tighter Register Specification

The DFA for

$$Register \rightarrow \mathrm{R}\ (\ (0|1|2)\ (0|1|2|\ \ldots\ |\ 9\ |\ \varepsilon)\ |\ (4|5|6|7|8|9)\ |\ (3\ (0|1|\varepsilon))\ )$$



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

Tighter Register Specification

To implement the recognizer

- Use the same code skeleton
- Use transition table and final states for the new RE

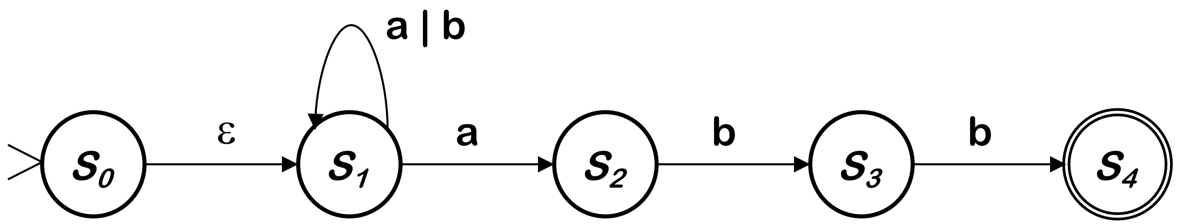
δ	R	0,1	2	3	4,5,6 7,8,9	*other*
s_0	s_1	s_e	s_e	s_e	s_e	s_e
s_1	s_e	s_2	s_2	s_5	s_4	s_e
s_2	s_e	s_3	s_3	s_3	s_3	s_e
s_3	s_e	s_e	s_e	s_e	s_e	s_e
s_4	s_e	s_e	s_e	s_e	s_e	s_e
s_5	s_e	s_6	s_e	s_e	s_e	s_e
s_6	s_e	s_e	s_e	s_e	s_e	s_e
s_e	s_e	s_e	s_e	s_e	s_e	s_e

$$Final = \{ s_2, s_3, s_4, s_5, s_6 \}$$

- Bigger tables, more space, same asymptotic costs
- Better syntax checking at the same cost

Non-deterministic Finite Automata

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

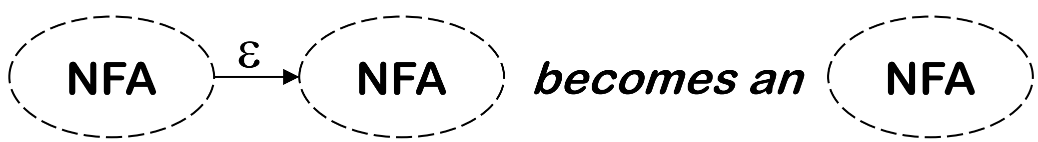
This is a *non-deterministic finite automaton* (NFA)

Non-deterministic Finite Automata

- An NFA accepts a string *x* iff there exists a path though the transition graph from s_0 to a final state and the edge labels spell *x*
- Transitions on ε consume no input
- To "run" (simulate) the NFA,
  - Start in s_0 and take *all* the transitions for each character
  - At each iteration add the states reachable by ε-transitions

Why study NFAs?

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



- They will be very important later in the class (when looking at the way that bottom-up parsing works)

Review: NFA Simulation

Two key functions

- *move(q , a)* is set of states reachable by "a" from states in *q*
- *ε-closure(q)* is set of states reachable by ε from states in *q*

```
states  = ε-closure( {s0} );
char  = get_next_char();
while (char != EOF) {
  states  = ε-closure(move(states,char));
  char = get_next_char();
}
if (states ∩ Final is not empty)
  report acceptance;
else
   report failure;
```

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*

Simulate sets of possible states that NFA can reach with states of the DFA

- Possible exponential blowup in the state space
- Still, one state per character in the input stream

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 or table

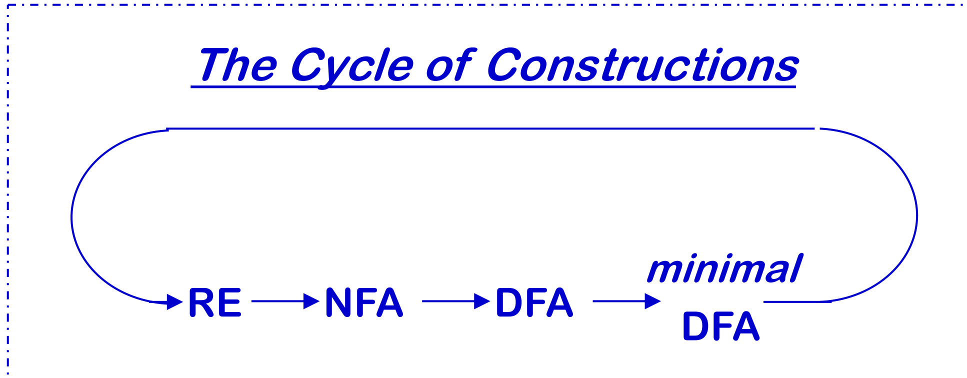
Scanner generators

- Lex , Flex, Jlex, and Jflex work along these lines
- Algorithms are well-known and well-understood
- Interface to parser is important

Relationship between RE/NFA/DFA

RE$\rightarrow$NFA *(Thompson's construction)*

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



NFA $\rightarrow$DFA *(subset construction)*

- Build the simulation

DFA $\rightarrow$ Minimal DFA

- Hopcroft's algorithm

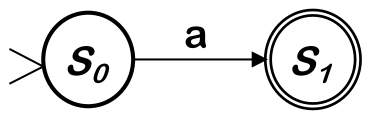
DFA $\rightarrow$RE

- All pairs, all paths problem
- Union together paths from s_0 to a final state

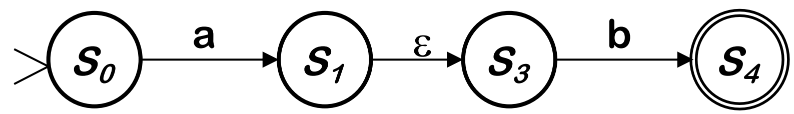
RE →NFA using Thompson's Construction

Key idea

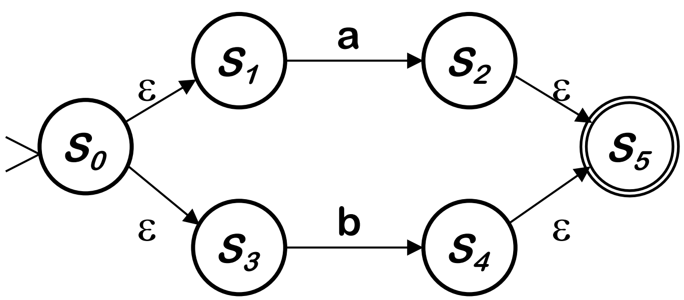
- NFA pattern for each symbol & each operator
- Join them with ε moves in precedence order



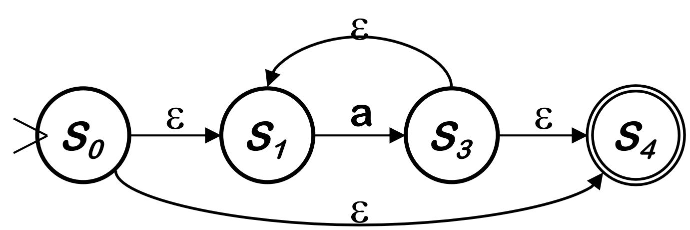
NFA for a



NFA for ab



NFA for a | b



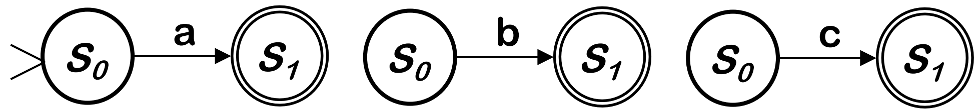
NFA for a*

Ken Thompson, CACM, 1968

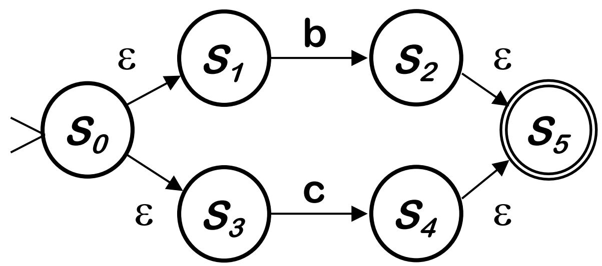
Thompson's Construction Example

Let's try a (b | c)*

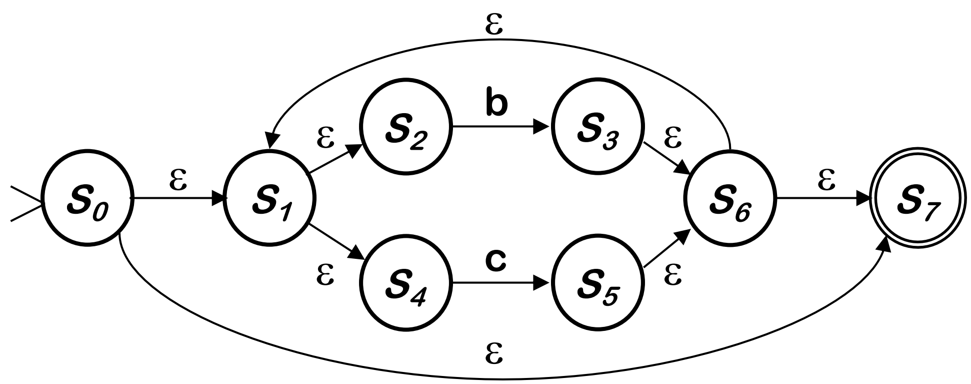
1. a, b, c



2. b | c

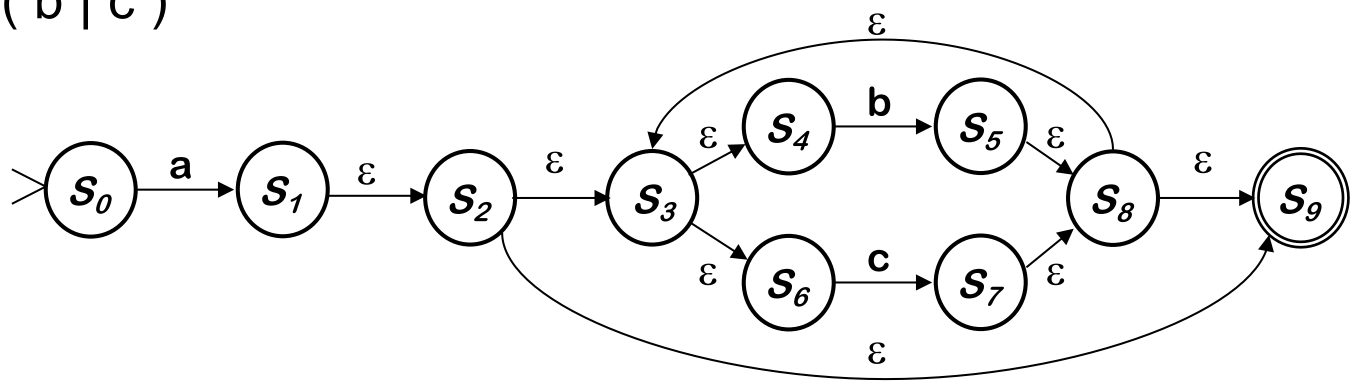


3. (b | c)*



Thompson's Construction Example

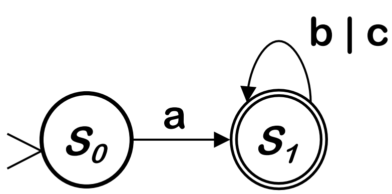
4. a (b | c)*



Given a regular expressions *r* the generated NFA is of size |N| = O(|*r*|)

- At most two new states are created at each step
- Each state has at most two incoming and two outgoing transitions
- Simulating an NFA constructed with Thompson's construction on a string *x* takes O(|N| x |*x*|)

Of course, a human would design something simpler ...



But, we can automate production of the more complex one ...

Summary of Key Points

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

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

Example: Fortran 66/77

integer
function A

```
      INTEGERFUNCTIONA
      PARAMETER(A=6,B=2)
      IMPLICIT CHARACTER*(A-B)(A-B)
      INTEGER FORMAT(10), IF(10), DO9E1
100   FORMAT(4H)=(3)
200   FORMAT(4   )=(3)
      DO9E1=1
      DO9E1=1,2
  9   IF(X)=1
      IF(X)H=1
      IF(X)300,200
300   CONTINUE
      END
C     THIS IS A "COMMENT CARD"
    $ FILE(1)
      END
```

Macro definitions

First A and B are converted to (6-2)
This statement declares that variables that begin with A and B are of data-type four character string

)=(3 is a literal constant

statement for formatting input, output

assigns value to variable DO9E1

assigns value to array element

one statement split into two lines

Example: C++


```
#include <vector>
using namespace std;
vector<vector<int>> v;
int main() {
}

error message:
coltrane% g++ a.cpp
a.cpp:3: error: '>>' should be '> >' within
         a nested template argument list
```

- The above code results in an error because the ‘>>’ scans as the shift operator. Clearly, it was intended to be a close bracket, but the scanner does not know about the *structure* of the program. The program below compiles without error.

```
#include <vector>
using namespace std;
vector<vector<int> > v;
int main() {
}
```