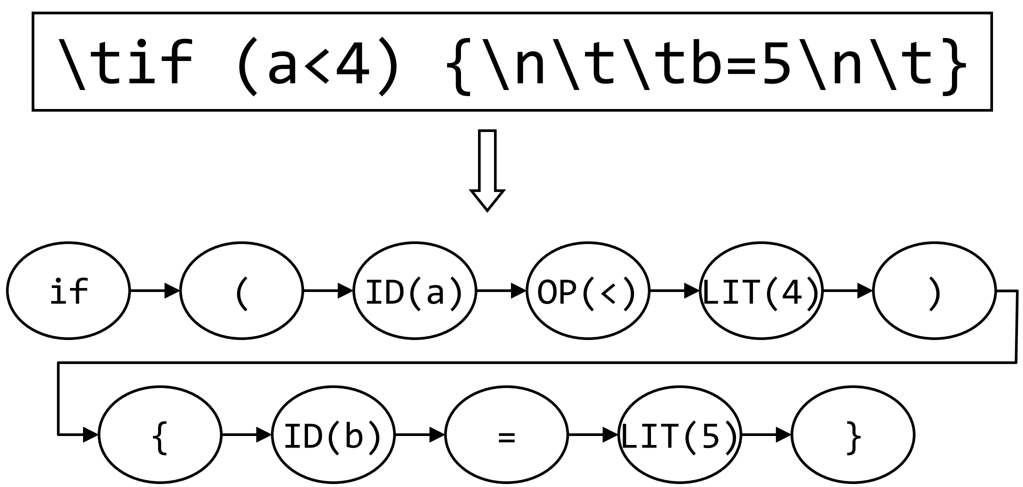
CS406: Compilers

Spring 2020

Week 3: Scanners

Scanner - Overview

- Also called lexers, lexical analyzers
- Recall: scanners break input stream up into a set of tokens
  - Identifiers, reserved words, literals, etc.



Scanner - Overview

- Divide the program text into *substrings* *or* *lexemes*
  - place dividers
- Identify the *class* of the substring identified
  - Examples: Identifiers, keywords, operators, etc.
    - Identifier – strings of letters or digits starting with a letter
    - Integer – non-empty string of digits
    - Keyword – “if”, “else”, “for” etc.
    - Blankspace - \t, \n, ‘ ‘
    - Operator – (,), <, =, etc.
- Substrings follow some pattern

Exercise

- What is the English language analogy for *class*?

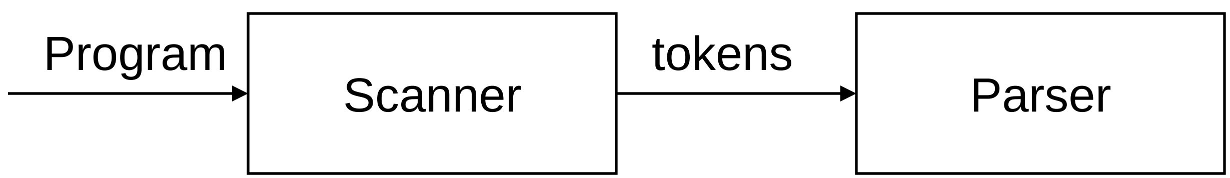
- How many tokens of class *identifier* exist in the code below?

```
for(int i=0;i<10;i++){\n\tprintf(“hello”);\n}
```

Scanner Output

- A token corresponding to each lexeme
  - Token is a pair: <class, value>

A string / lexeme / substring of program text



Scanners – interesting examples

- Fortran (white spaces are ignored)

```
DO 5 I = 1,25
DO 5 I = 1.25
```

We always need to *look ahead* to identify tokens

- PL/1

```
DECLARE (ARG1, ARG2, . . .
```

- C++

Nested template: `Quad<Square<Box>> b;`

Stream input: `std::cin >> bx;`

Scanners – what do we need to know?

1. How do we define tokens?
   – Regular expressions
2. How do we recognize tokens?
   – build code to find a lexeme that is a prefix and that belongs to one of the patterns.
3. How do we write lexers?
   – E.g. use a lexer generator tool such as Flex

Regular Expressions

- Regular sets:

  Formal: a language that can be defined by regular expressions

  Informal: a set of strings defined by regular expressions

  Strings are regular sets (with one element): pi 3.14159

    - So is the empty string: λ (ε instead)

  – Concatenations of regular sets are regular: pi3.14159
    - To avoid ambiguity, can use ( ) to group regexps together

  – A choice between two regular sets is regular, using |: (pi|3.14159)

  – 0 or more of a regular set is regular, using *: (pi)*

  – Some other notation used for convenience:
    - Use `Not` to accept all strings except those in a regular set
    - Use `?` to make a string optional: `x?` equivalent to `(x|λ)`
    - Use + to mean 1 or more strings from a set: `x+` equivalent to `xx*`
    - Use [ ] to present a range of choices: `[1-3]` equivalent to `(1|2|3)`

Examples of Regular Expressions

- Digit: D = [0-9]
- Letter: L = [A-Za-z]
- Literals (integers or floats): -?D+(.D*)?
- Identifiers: (_|L)(_|L|D)*
- Comments (as in Micro): -- Not(\n)*\n
- More complex comments (delimited by ##, can use # inside comment): ##((#|λ)Not(#))*##

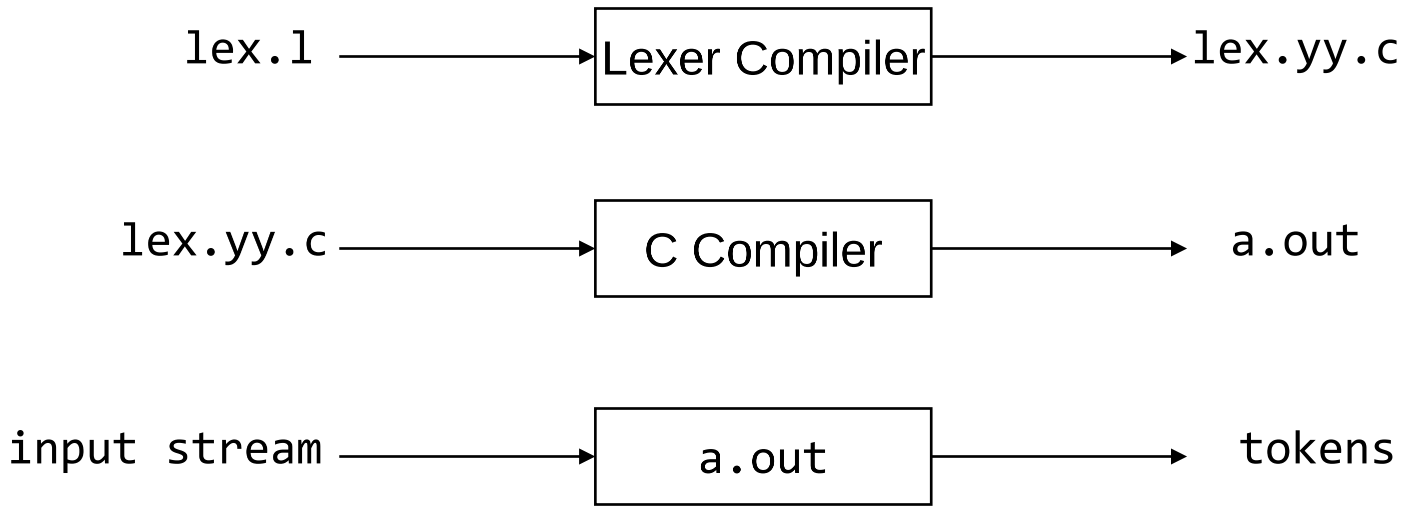
Scanner Generators

- Essentially, tools for converting regular expressions into scanners
- Lex (Flex) generates C/C++ scanners

Lex (Flex)

- Commonly used Unix scanner generator (superseded by Flex)
- Flex is a domain specific language for writing scanners
- Features:
  - Character classes : define sets of characters (e.g., digits)
  - Token definitions : `regex {action to take}`

Lex (Flex)


lex.l
Lexer Compiler
lex.yy.c
lex.yy.c
C Compiler
a.out
input stream
a.out
tokens

Lex (Flex)

- Format of lex.l

```
Declarations
%%
Translation rules
%%
Auxiliary functions
```

Lex (Flex)

```
DIGIT     [0-9]
ID        [a-z][a-z0-9]*

%%

{DIGIT}+   {
           printf( "An integer: %s (%d)\n", yytext,
           atoi( yytext ) );
        }

{DIGIT}+"."{DIGIT}* {
               printf( "A float: %s (%g)\n", yytext,
               atof( yytext ) );
           }

if|then|begin|end|procedure|function {
           printf( "A keyword: %s\n", yytext );
        }

{ID}       printf( "An identifier: %s\n", yytext );
```

Lex (Flex)

- The order in which tokens are defined matters!
- Lex will match the longest possible token
  - "ifa" becomes ID(ifa), not IF ID(a)
- If two regexes both match, Lex uses the one defined first
  - "if" becomes IF, not ID(if)
- Use action blocks to process tokens as necessary
  - Convert integer/float literals to numbers
  - Remove quotes from string literals

Recap…

- We saw what it takes to write a scanner:
  - Specify how to identify token classes (using regexps)
  - Convert the regexps to code that identifies a *prefix* of the input string as a *lexeme* matching one of the token classes
    - Using tools for automatic code generation (e.g. `Lex` / `Flex` / `ANTLR`)

How do these tools convert regexps to code?

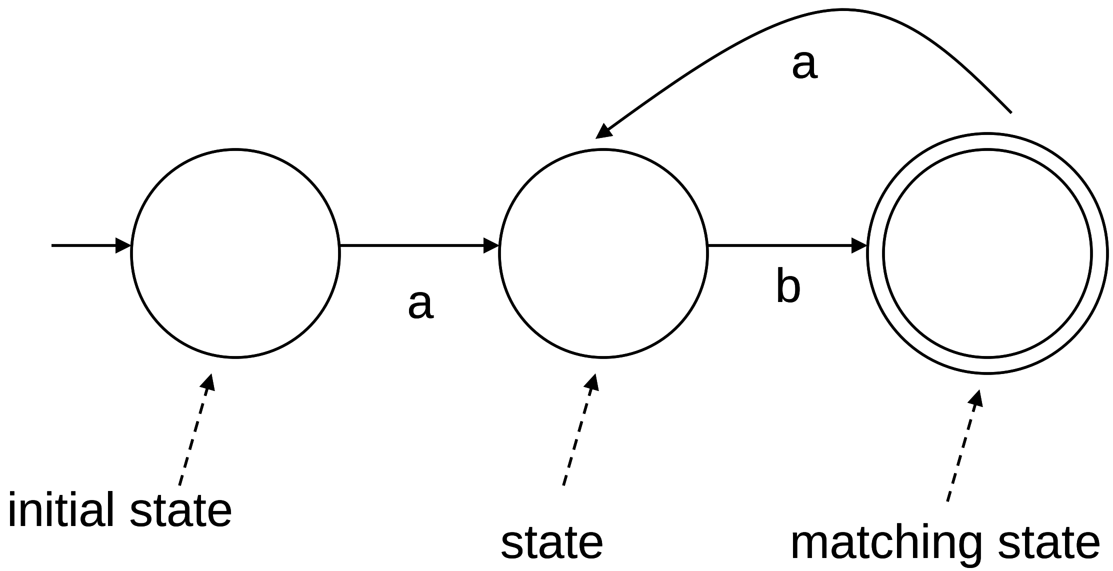
Enabling concept: Finite Automata

Finite Automata

- Another way to describe sets of strings (just like regular expressions)
- Also known as finite state machines / automata
- Reads a string, either recognizes it or not
- Features:
  - State: initial, matching / final / accepting, non-matching
  - Transition: a move from one state to another

Finite Automata

- Regular expressions and FA are equivalent*

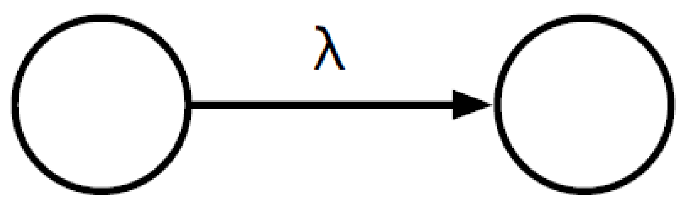


Exercise: what is the equivalent regular expression for this FA?

* Ignoring the *`empty`* regular language

λ transitions

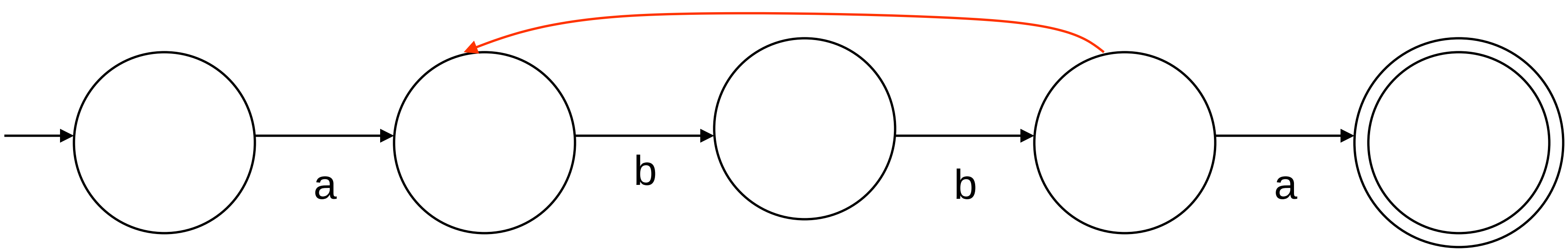
- Transitions between states that aren't triggered by seeing another character
  - Can *optionally* take the transition, but do not have to
  - Can be used to link states together



Think of this as an arrow to a state without a label

Non-deterministic Finite Automata

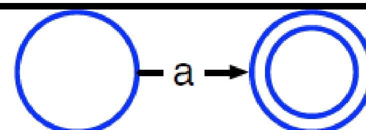
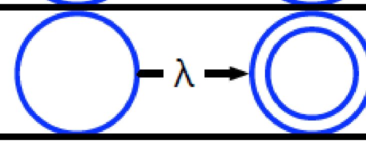
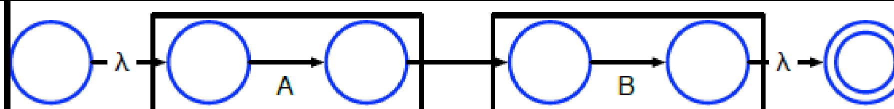
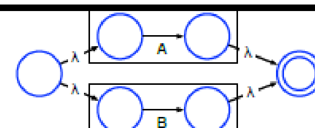
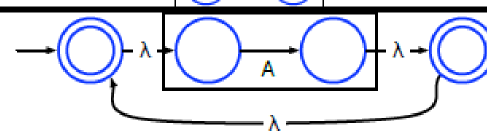
- A FA is non-deterministic if, from one state reading a single character could result in transition to multiple states (or has λ transitions)
- Sometimes regular expressions and NFAs have a close correspondence



≡

a(bb)+a

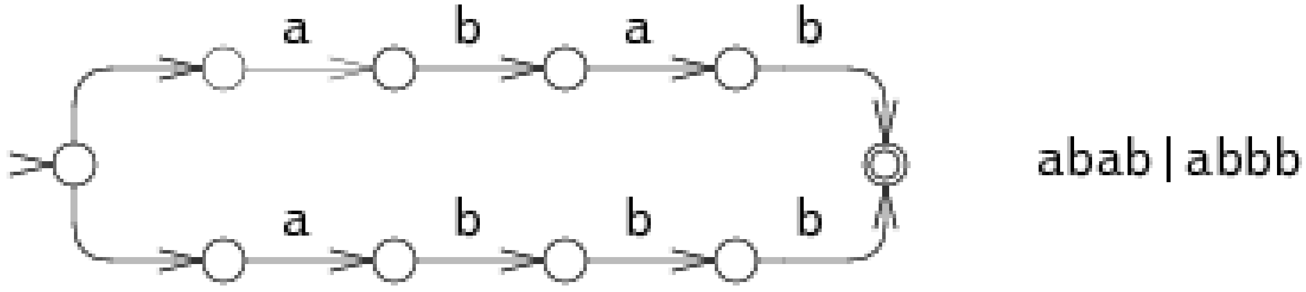
Building a FA from a regexp

Expression	FA
a	
λ	
AB	
A|B	
A*	

Mini-exercise: how do we build an FA that accepts Not(A)?

What about A? (? as in optional)

Non-deterministic Finite Automata



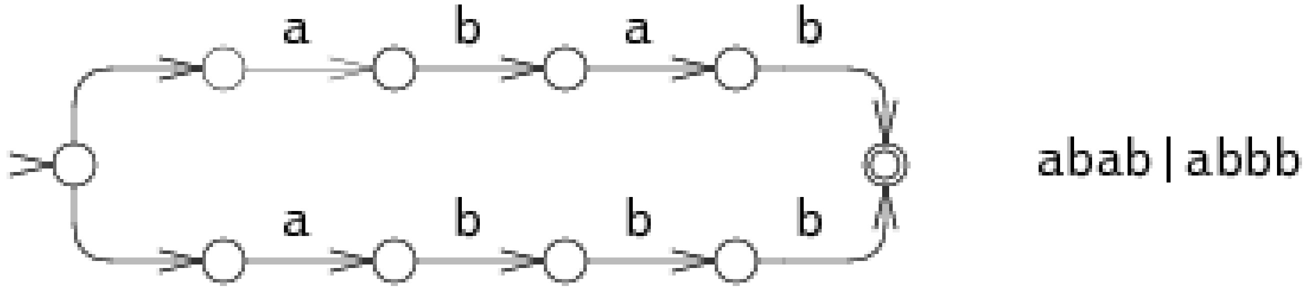
- NFAs are concise but slow
- Example:
  - Running the NFA for input string abbb requires exploring all execution paths

* picture example taken from https://swtch.com/~rsc/regexp/regexp1.html

“Running” an NFA

- Intuition: take every possible path through an NFA
  - Think: parallel execution of NFA
  - Maintain a “pointer” that tracks the current state
  - Every time there is a choice, “split” the pointer, and have one pointer follow each choice
  - Track each pointer simultaneously
    - If a pointer gets stuck, stop tracking it
    - If any pointer reaches an accept state at the end of input, accept

Non-deterministic Finite Automata



- NFAs are concise but slow
- Example:
  - Running the NFA for input string abbb requires exploring all execution paths
  - **Optimization: run through the execution paths in parallel**
    - ***Complicated. Can we do better?***

* picture example taken from https://swtch.com/~rsc/regexp/regexp1.html

NFAs to DFAs

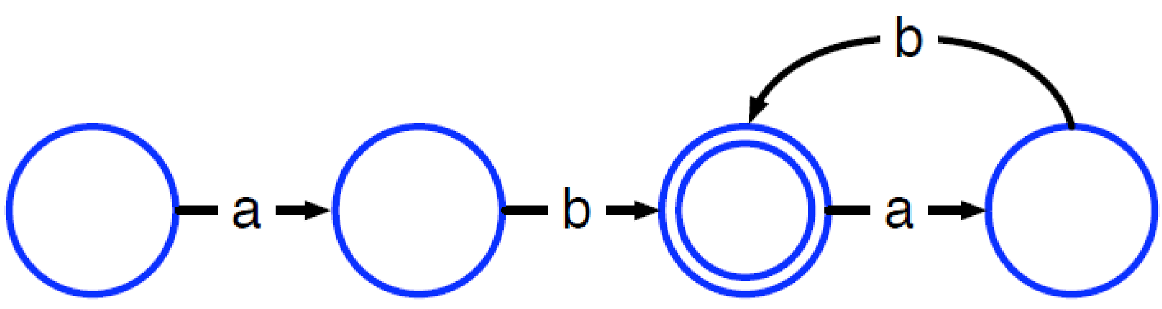
Each possible input character read leads to at most one new state

- Can convert NFAs to *deterministic* finite automata (DFAs)
  - No choices — never a need to "split" pointers
- Initial idea: simulate NFA for all possible inputs, any time there is a new configuration of pointers, create a state to capture it
  - Pointers at states 1, 3 and 4 → new state {1, 3, 4}
- Trying all possible inputs is impractical; instead, for any new state, explore all possible *next* states (that can be reached with a single character)
- Process ends when there are no new states found
- This can result in very large DFAs!

DFA reduction

- DFAs built from NFAs are not necessarily optimal
  - May contain many more states than is necessary

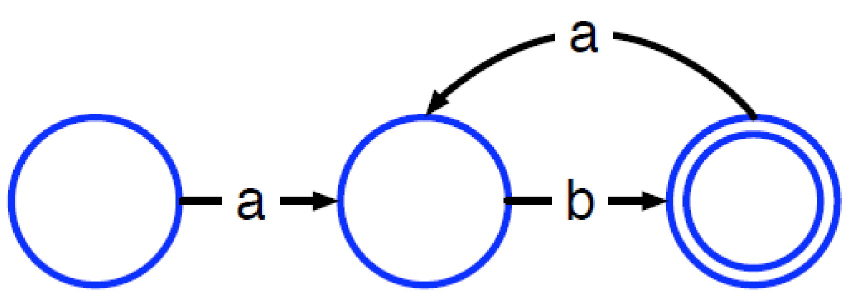
(ab)+ ≡ (ab)(ab)*



DFA reduction

- DFAs built from NFAs are not necessarily optimal
  - May contain many more states than is necessary

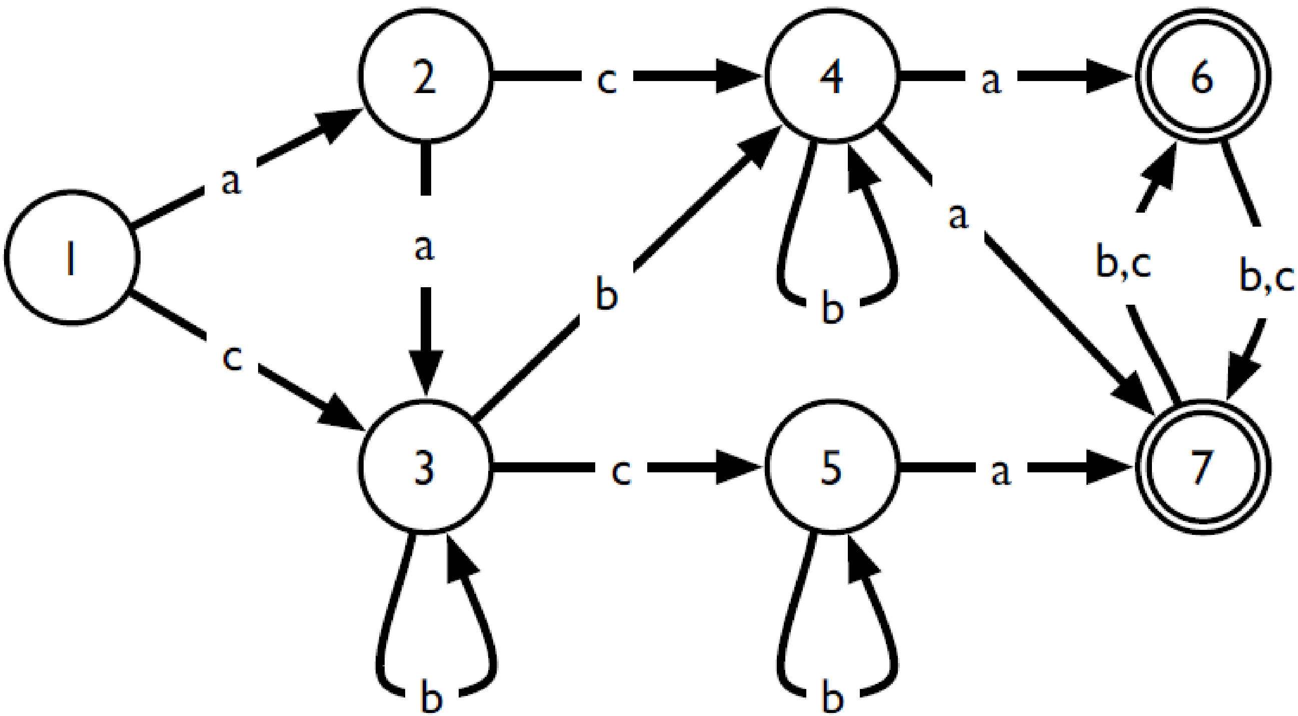
$$(ab)+ \equiv (ab)(ab)^*$$



DFA reduction

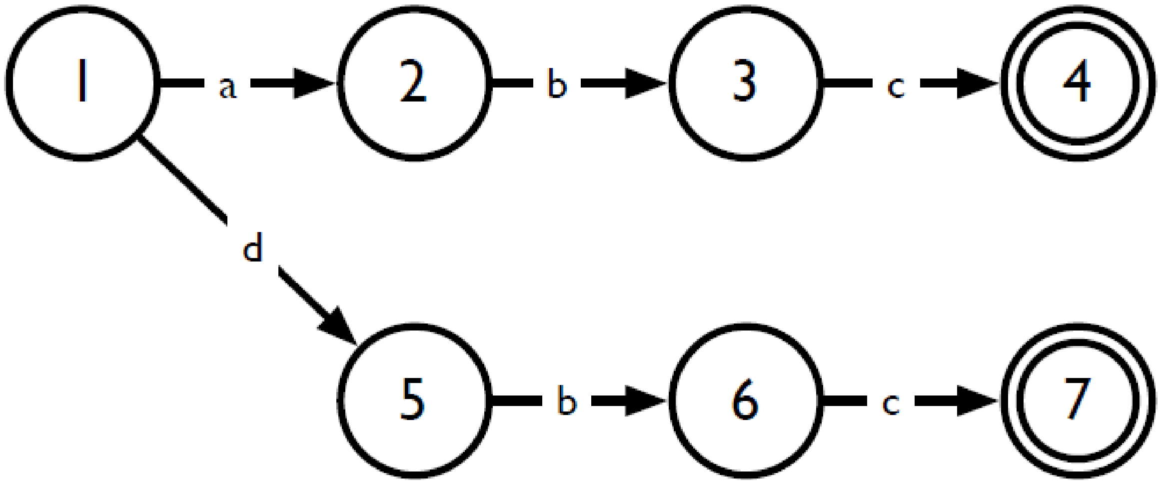
- Intuition: merge equivalent states
  - Two states are equivalent if they have the same transitions to the same states
- Basic idea of optimization algorithm
  - Start with two big nodes, one representing all the final states, the other representing all other states
  - Successively split those nodes whose transitions lead to nodes in the original DFA that are in different nodes in the optimized DFA

Example

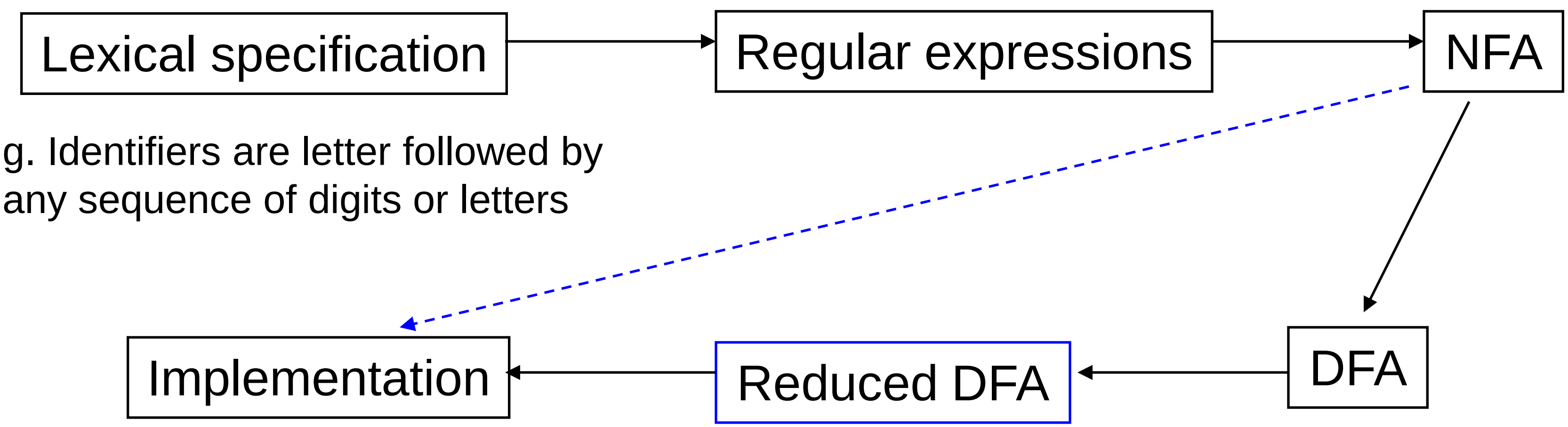


Exercise

- *Reduce the DFA*



Scanner - flowchart

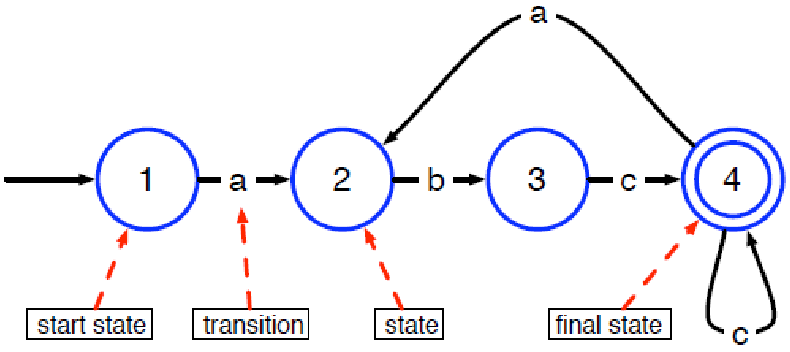


e.g. Identifiers are letter followed by any sequence of digits or letters

Implementation: Transition Tables

- Table encoding states and transitions of FA
  - I row per state, I column per possible character
  - Each entry: if automaton in a particular state sees a character, what is the next state?

State	Character		
	a	b	c
I	2		
2		3	
3			4
4	2		4



DFA Program

- Using a transition table, it is straightforward to write a program to recognize strings in a regular language

```
state = initial_state; //start state of FA
while (true) {
    next_char = getc();
    if (next_char == EOF) break;
    next_state = T[state][next_char];
    if (next_state == ERROR) break;
    state = next_state;
}
if (is_final_state(state))
    //recognized a valid string
else
    handle_error(next_char);
```

Alternate implementation

- Here's how we would implement the same program "conventionally"

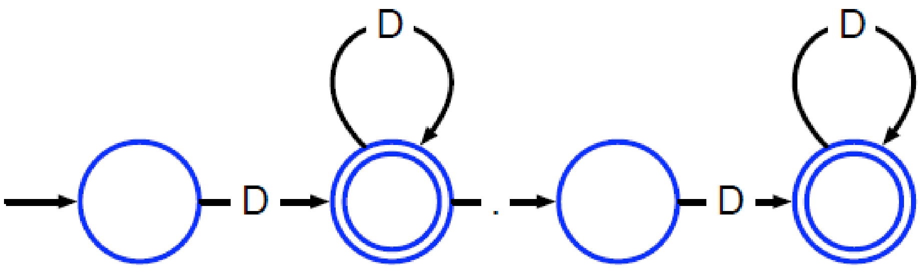
```
next_char = getc();
while (next_char == ‘a’) {
   next_char = getc();
   if (next_char != ‘b’) handle_error(next_char);
   next_char = getc();
   if (next_char != ‘c’) handle_error(next_char);
   while (next_char == ‘c’) {
      next_char = getc();
      if (next_char == EOF) return; //matched token
      if (next_char == ‘a’) break;
      if (next_char != ‘c’) handle_error(next_char);
   }
}
handle_error(next_char);
```

Lookahead

- Up until now, we have only considered matching an entire string to see if it is in a regular language
- What if we want to match multiple tokens from a file?
  - Distinguish between `int a` and `inta`
  - We need to *look ahead* to see if the next character belongs to the current token
  - If it does, we can continue
  - If it doesn't, the next character becomes part of the next token

Multi-character lookahead

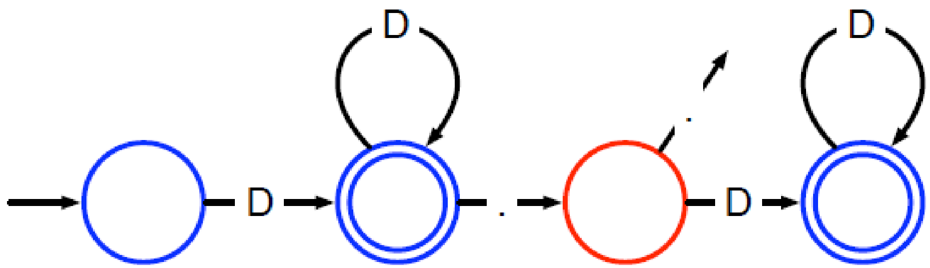
- Sometimes, a scanner will need to look ahead more than one character to distinguish tokens
- Examples
  - Fortran: DO I = 1,100 (loop) vs. DO I = 1.100 (variable assignment)
  - Pascal: 23.85 (literal) vs. 23..85 (range)



- 2 solutions: Backup or special "action" state

Multi-character lookahead

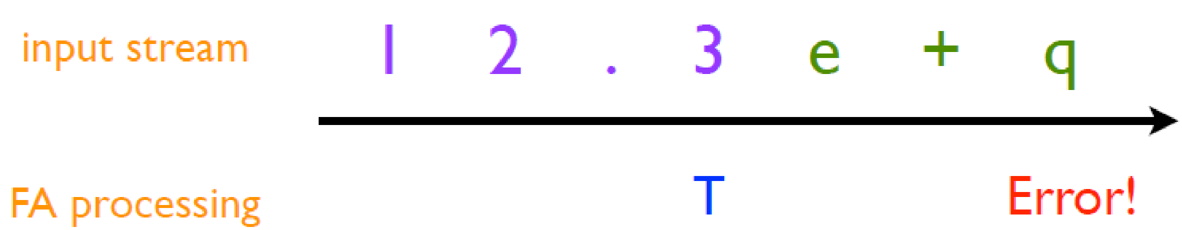
- Sometimes, a scanner will need to look ahead more than one character to distinguish tokens
- Examples
  - Fortran: DO I = 1,100 (loop) vs. DO I = 1.100 (variable assignment)
  - Pascal: 23.85 (literal) vs. 23..85 (range)



- 2 solutions: Backup or special “action” state

General approach

- Remember states (T) that can be final states
- Buffer the characters from then on
- If stuck in a non-final state, back up to T, restore buffered characters to stream
- Example: 12.3e+q



Error Recovery

- What do we do if we encounter a lexical error (a character which causes us to take an undefined transition)?
- Two options
  - Delete all currently read characters, start scanning from current location
  - Delete *first* character read, start scanning from second character
    - This presents problems with ill-formatted strings (why?)
    - One solution: create a new regexp to accept runaway strings

Next time

- We've covered how to tokenize an input program
- But how do we decide what the tokens actually say?
  - How do we recognize that

    IF ID(a) OP(<) ID(b) { ID(a) ASSIGN LIT(5) ; }

    is an if-statement?
- Next time: Parsers

Suggested Reading

- Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D.Ullman: Compilers: Principles, Techniques, and Tools, 2/E, AddisonWesley 2007
  – Chapter 3 (Sections: 3.1, 3,3, 3.6 to 3.9)
- Fisher and LeBlanc: Crafting a Compiler with C
  – Chapter 3 (Sections 3.1 to 3.4, 3.6, 3.7)