String Manipulation Languages

CS164: Introduction to Programming Languages and Compilers

Lectures 2 and 3, Fall 2009

Instructor: **Ras Bodik**

GSI: **Joel Galenson**

UC Berkeley

Administrativia

Section times poll:

Thu 2-3 and Wed 11-12 most popular, in that order.

We asked for a room. We'll see if we get one.

Office hours (Ras):

Tu 2:30-3:30, Th 5-6.

in 569 Soda (ParLab) send email if door is closer after hours

First project assigned tonight. See course wiki.

http://cs164fa09.pbworks.com/

Please turn off your cell phones and close laptops.

Outline for today

- String processing: five motivating applications
- Discovering a suitable programming model
- Regular expressions and their interpreter
- Non-deterministic finite automaton
- Regular expression compiler

1. Web scraping and rewriting

Your first project (out today). Using GreaseMonkey.

Opportunity: Web is more readable when you view the print-friendly, ad-free pages. Automate this!

Problem: Your script will rewrite links on a page to go directly to a print-friendly version of target page.

How: You will have to fetch the target of the link, look in the target HTML for a best-guess link to print-friendly alternative page, then rewrite the link.

2. Cucumber: a Ruby testing framework

Cucumber test file (format is customizable):

```
Scenario: Test the banking web service
  Given I log in as "bonnie" with password "clyde"
  When I go to "Accounts"
  Then I should see a link "Our Robbery Savings"
  When I follow this link
  Then I the value of "Interest" should be "$1,024.00"
```

We could interpret this with a GreaseMonkey script:

"Given" clause makes script go to web site and enter login/password.

"When" clause clicks on the link Accounts.

"Then" clause tests that resulting page contains a link to given account.

Programming problem:

how to customize a GM script for a new a test file format?

3. Lexical analysis in compiler/interpreter

Input: a program

```
function timedCount() { // my function
  document.getElementById('txt').value=c;
}
```

Output: a sequence of tokens

```
FUNCTION, ID("timedCount"), LPAR, RPAR, LCUR,
ID("document"), DOT, ID("getElementById"), LPAR,
STRING("txt"), RPAR, DOT, ID("value"), ASGN, ID("c"),
SEMI, RCUR
```

Our problem: how to concisely describe the tokens

Notes on lexical analysis

When needed, token is associated with its *lexeme*

- Lexer partitions the input into lexemes

Whitespace and comments are skipped

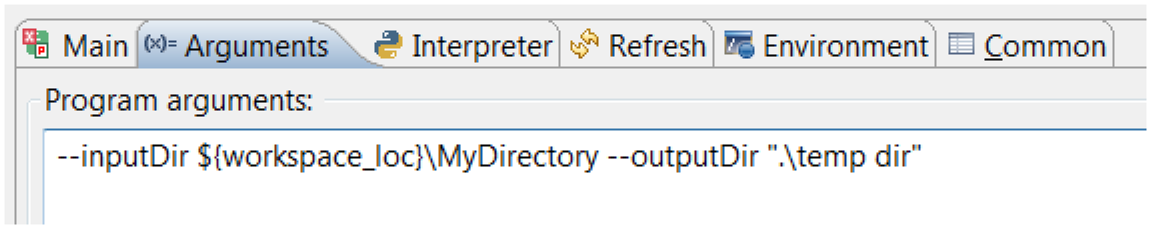
- In which language cannot whitespace be skipped?

Stream of tokens is fed to compiler

For bored students: typically, tokens flow to compiler, and no information flows back. In which language does the compiler need to communicate back to lexer? Give scenario.

4. File name processing languages

In shell scripts and IDEs:



Program arguments:

```
--inputDir ${workspace_loc}\MyDirectory --outputDir ".\temp dir"
```

Eclipse translates this line into arguments passed to the interpreter when the program is invoked

```
args = { "--inputDir", "c:\\wspace\\MyDirectory",
         "--inputDir", ".\\temp dir" }
```

Variable substitution, character escaping, quotes, …

trickier than it seems (Eclipse got this language wrong)

5. Search for strings in text editors

Imagine you want to search for names containing a ' and correct them. Examples:

D'Souza --> D’Souza

D`Souza --> D’Souza

Your replacement should avoid quote characters used as quotations:

`quoted text’

Our plan on string processing languages

Design a small language for string processing

- find a string
- extract parts from it

Develop an efficient implementation

Also, interesting applications

and fundamental algorithms

Let's write a scanner (lexical analyzer)

First we'll write it by hand, in an imperative language

- We'll examine the repetitious plumbing in the scanner
- Then We'll hide the plumbing by abstracting it away

A simple scanner will do. Only four tokens:

TOKEN	**Lexeme**
ID	a sequence of one or more letters or digits starting with a letter
EQUALS	“==“
PLUS	“+”
TIMES	“*”

Imperative scanner

```
c=nextChar();
if (c == '=') { c=nextChar(); if (c == '=') {return EQUALS;}}
if (c == '+') { return PLUS; }
if (c == '*') { return TIMES; }
if (c is a letter) {
   c=NextChar();
   while (c is a letter or digit) {  c=NextChar(); }
   undoNextChar(c);
   return ID;
}
```

Note: this scanner does not handle errors. What happens if the input is

"var1 = var2"   It should be var1 == var2

An error should be reported at around '='.

Imperative scanner

You could write your entire scanner in this style

- and for small scanners this style is appropriate

Why does this code breaks as the task gets bigger? Try to add:

- lexemes that start with the same string: "if" and "iffy"
- C-style comments: `/* anything here /* nested comments */ */`
- string literals with escape sequences: "... \t ... \" ... "
- error handling, e.g., "string literal missing a closing quote"

Real-world imperative scanners get unwieldy

- lexical structure of the scanned language not obvious
- scanner code obscured it by spreading around the string comparisons and other actions

Real scanner get unwieldy (ex: JavaScript)

```
/* Scan XML comment. */
if (MatchChar(ts, '-')) {
    if (!MatchChar(ts, '-'))
        goto bad_xml_markup;
    while ((c = GetChar(ts)) != '-' || !MatchChar(ts, '-')) {
        if (c == EOF)
            goto bad_xml_markup;
        ADD_TO_TOKENBUF(c);
    }
    tt = TOK_XMLCOMMENT;
    tp->t_op = JSOP_XMLCOMMENT;
    goto finish_xml_markup;
}

/* Scan CDATA section. */
if (MatchChar(ts, '[')) {
    jschar cp[6];
    if (PeekChars(ts, 6, cp) &&
        cp[0] == 'C' &&
        cp[1] == 'D' &&
        cp[2] == 'A' &&
        cp[3] == 'T' &&
        cp[4] == 'A' &&
        cp[5] == '[') {
        SkipChars(ts, 6);
        while ((c = GetChar(ts)) != ']' ||
               !PeekChars(ts, 2, cp) ||
               cp[0] != ']' ||
               cp[1] != '>') {
            if (c == EOF)
                goto bad_xml_markup;
            ADD_TO_TOKENBUF(c);
```

From http://mxr.mozilla.org/mozilla/source/js/src/jsscan.c

Imperative Lexer: what vs. how

```
c=nextChar();
if (c == ‘=‘) { c=nextChar(); if (c == ‘=‘) {return EQUALS;}}
if (c == ‘+’) { return PLUS; }
if (c == ‘*’) { return TIMES; }
if (c is a letter) {
   c=NextChar();
   while (c is a letter or digit) {  c=NextChar(); }
   undoNextChar(c);
   return ID;
}
```

☞ little logic, much plumbing

Identifying the plumbing (the how, part 1)

```
c=nextChar();
if (c == '=') { c=nextChar(); if (c == '=') {return EQUALS;}}
if (c == '+') { return PLUS; }
if (c == '*') { return TIMES; }
if (c is a letter) {
   c=NextChar();
   while (c is a letter or digit) {  c=NextChar(); }
   undoNextChar(c);
   return ID;
}
```

☞ characters are read always the same way

Identifying the plumbing (the how, part 2)

```
c=nextChar();
if (c == ‘=‘) { c=nextChar(); if (c == ‘=‘) {return EQUALS;}}
if (c == ‘+’) { return PLUS; }
if (c == ‘*’) { return TIMES; }
if (c is a letter) {
   c=NextChar();
   while (c is a letter or digit) {  c=NextChar(); }
   undoNextChar(c);
   return ID;
}
```

☞ tokens are always return-ed

Identifying the plumbing (the how, part3)

```
c=nextChar();
if (c == ‘=‘) { c=nextChar(); if (c == ‘=‘) {return EQUALS;}}
if (c == ‘+’) { return PLUS; }
if (c == ‘*’) { return TIMES; }
if (c is a letter) {
   c=NextChar();
   while (c is a letter or digit) {  c=NextChar(); }
   undoNextChar(c);
   return ID;
}
```

☞ the lookahead is explicit (programmer-managed)

Identifying the plumbing (the how)

```
c=nextChar();
if (c == ‘=‘) { c=nextChar(); if (c == ‘=‘) {return EQUALS;}}
if (c == ‘+’) { return PLUS; }
if (c == ‘*’) { return TIMES; }
if (c is a letter) {
   c=NextChar();
   while (c is a letter or digit) {  c=NextChar(); }
   undoNextChar(c);
   return ID;
}
```

☞ must build decision tree out of nested if’s (yuck!)

Can we hide the plumbing?

In a cleaner code, we want to avoid the following

- if's and while's to construct the decision tree
- calls to the read method
- explicit **return** statements
- explicit lookahead code

Ideally, we want code that looks like the specification:

TOKEN	Lexeme
ID	a sequence of one or more letters or digits starting with a letter
EQUALS	"=="
PLUS	"+"
TIMES	"*"

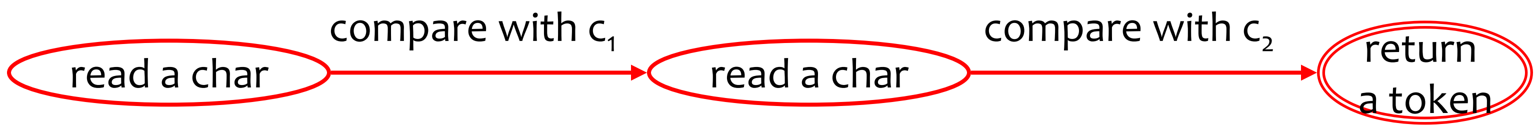
Separate out the how (plumbing)

The code actually follows a simple pattern:

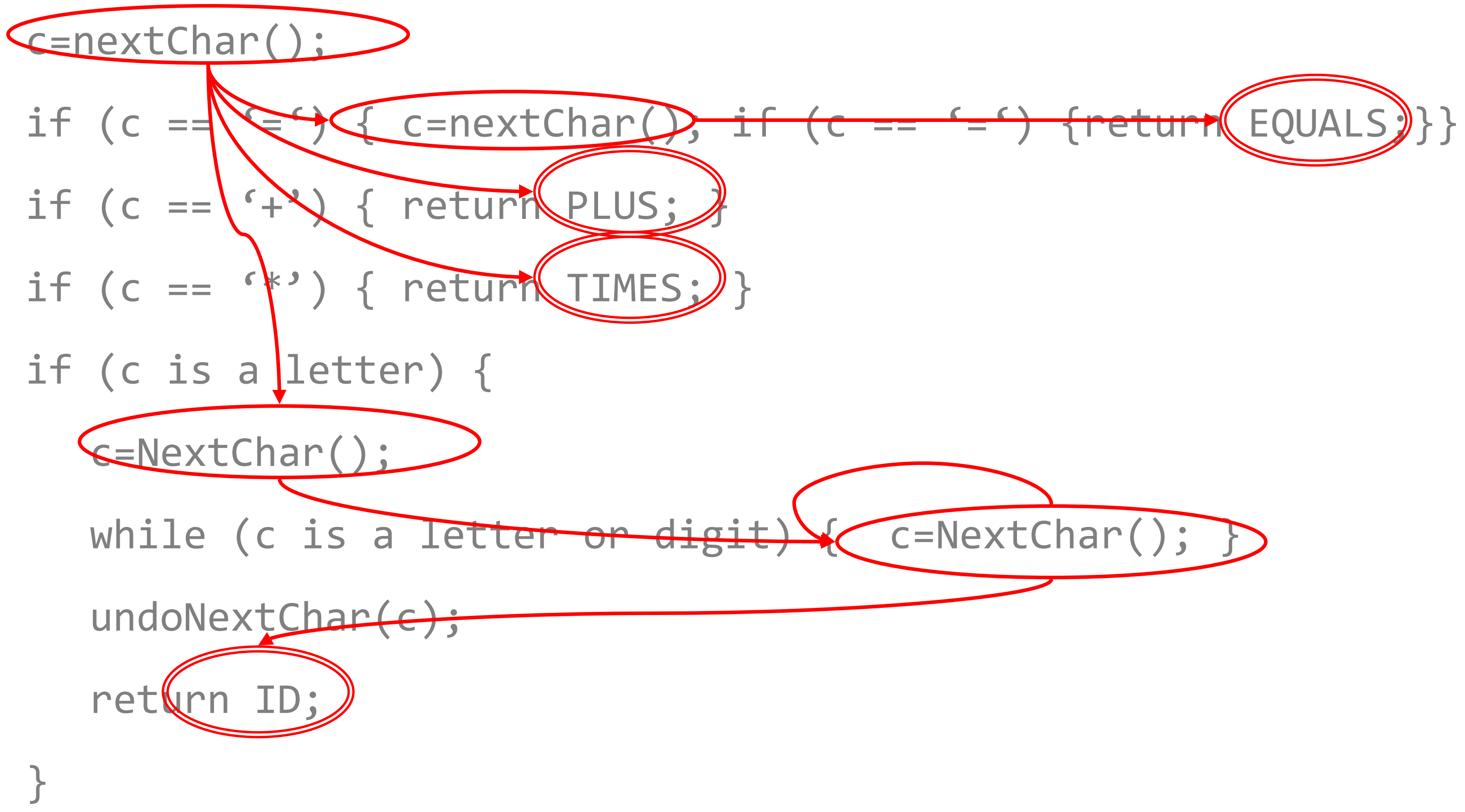
- read next char,
- compare it with some predetermined char
- if matched, jump to a different read of next char
- repeat this until a lexeme built; then return a token.

Is there already a programming language for encoding this concisely?

- yes, **finite-state automata**!
- finite: number of states is fixed, not input dependent

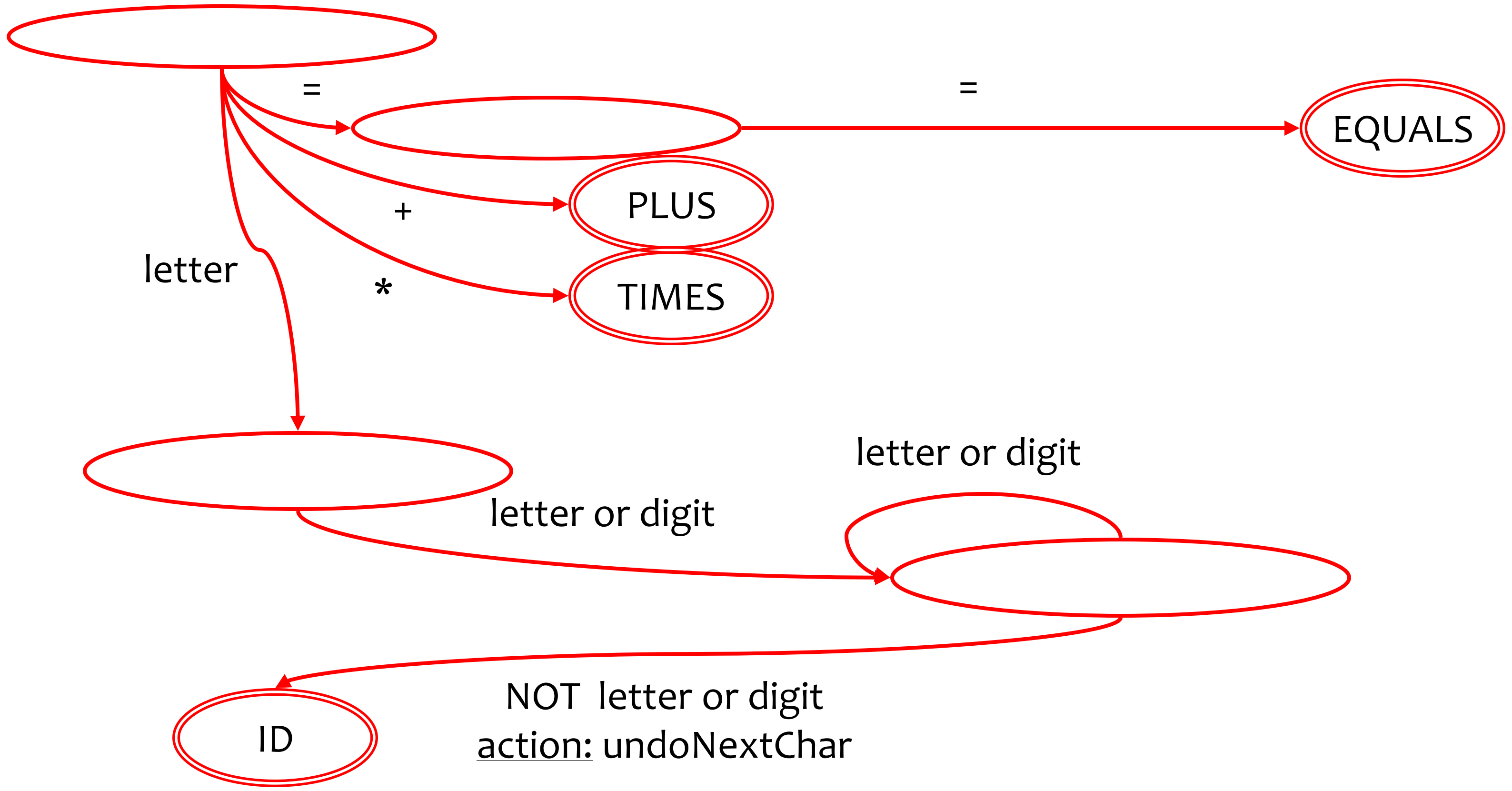


Separate out the what



```
c=nextChar();
if (c == ‘=‘) { c=nextChar(); if (c == ‘=‘) {return EQUALS;}}
if (c == ‘+’) { return PLUS; }
if (c == ‘*’) { return TIMES; }
if (c is a letter) {
   c=NextChar();
   while (c is a letter or digit) { c=NextChar(); }
   undoNextChar(c);
   return ID;
}
```

Here is the automaton; we'll refine it later



A declarative scanner

Part 1: declarative (the what)

describe each token as a finite automaton

- must be supplied for each scanner, of course (it specifies the lexical properties of the input language)

Part 2: imperative (the how)

connect these automata into a scanner automaton

- common to all scanners (like a library)
- responsible for the mechanics of scanning

Declarative programming

If we describe the scanner with finite automata, we allow the programmer to focus on:

- what the lexer should do,
- rather than how it should be done.

how: imperative programming

what: declarative programming

declarative programming is usually preferable

Blog post for today

http://james-iry.blogspot.com/2009/05/brief-incomplete-and-mostly-wrong.html

…

1957 - John Backus and IBM create FORTRAN. There's nothing funny about IBM or FORTRAN. It is a syntax error to write FORTRAN while not wearing a blue tie.

…

1964 - John Kemeny and Thomas Kurtz create BASIC, an unstructured programming language for non-computer scientists.

1965 - Kemeny and Kurtz go to 1964.

…

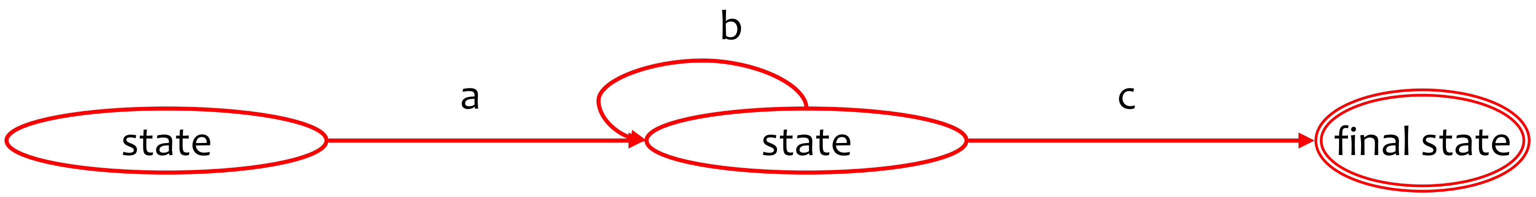
1980 - Alan Kay creates Smalltalk and invents the term "object oriented." When asked what that means he replies, "Smalltalk programs are just objects." When asked what objects are made of he replies, "objects." When asked again he says "look, it's all objects all the way down. Until you reach turtles.“

…

1987 - Larry Wall falls asleep and hits Larry Wall's forehead on the keyboard. Upon waking Larry Wall decides that the string of characters on Larry Wall's monitor isn't random but an example program in a programming language that God wants His prophet, Larry Wall, to design. Perl is born.

Now we need a notation for automata

Convenience, clarity dictates a textual language



Kleene invented regular expressions for the purpose:

a.b	a followed by b
a*	zero or more repetitions of a
a | b	a or b

Our example: $a.b^*.c$

Regular expressions

Regular expressions contain:

- characters : these must match the input string
- meta-characters: these serve as operators

exists in math

$. \quad * \quad |$

Operators can take regular expressions (recursive definition)

char	any character is a regular expression
~~re1.re2~~ $r_1 . r_2$	so is ~~re1~~ r_1 followed by ~~re2~~ r_2
re*	zero or more repetitions of re
re1 | re2	match re1 or re2
re +	one or more instances of re
[1-5]	same as (1|2|3|4|5) ; [ ] denotes a *character class*
[^"]	any character but the quote
\d	matches any digit
\w	matches any letter

Re+

$re . re^*$

Finite automata, in more detail

Deterministic, non-deterministic

DFAs

Deterministic finite automata (DFA)

We'll use DFA's as recognizers:

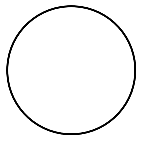
- recognizer accepts a set of strings, and rejects all others

Ex: in a lexer, DFA tells us if a string is a valid lexeme

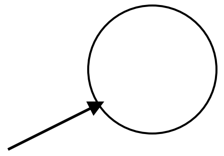
- the DFA for identifiers accepts "xyx" but rejects "3e4".

Finite-Automata State Graphs

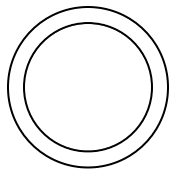
- A state



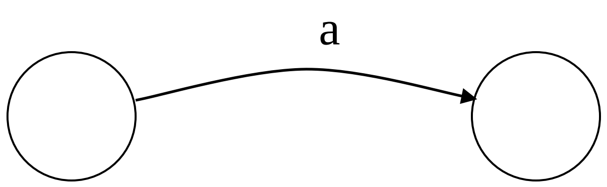
- The start state



- A final state



- A transition



Finite Automata

Transition

$$s_1 \rightarrow^{a} s_2$$

Is read

In state s_1 on input "a" go to state s_2

String <u>accepted</u> if

entire string consumed and automaton is in accepting state

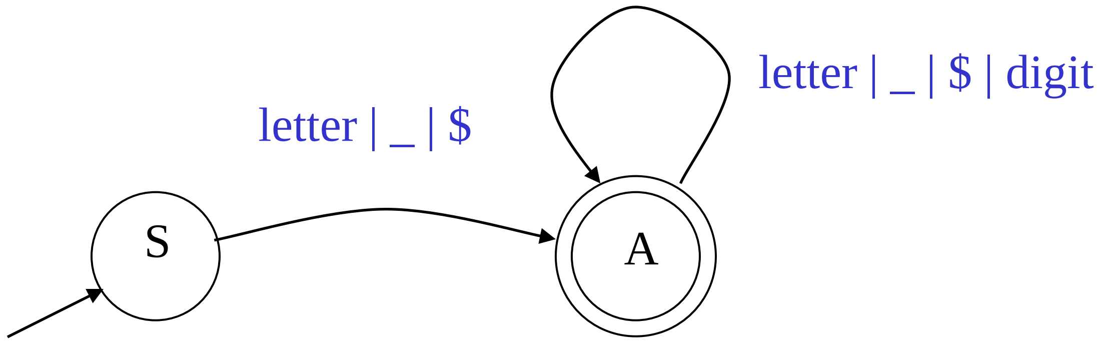
Rejected otherwise. Two possibilities for rejection:

- string consumed but not in accepting state
- next input character allows no transition (automaton is stuck)

Deterministic Finite Automata

Example: JavaScript Identifiers

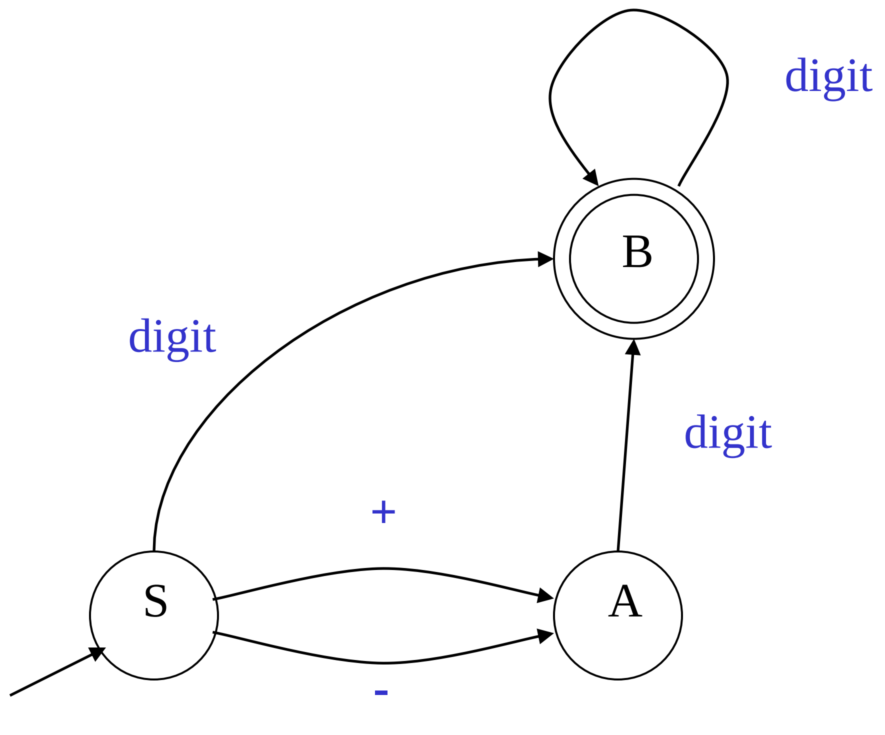
– sequences of 1+ letters or underscores or dollar signs or digits, starting with a letter or underscore or a dollar sign:



Example: Integer Literals

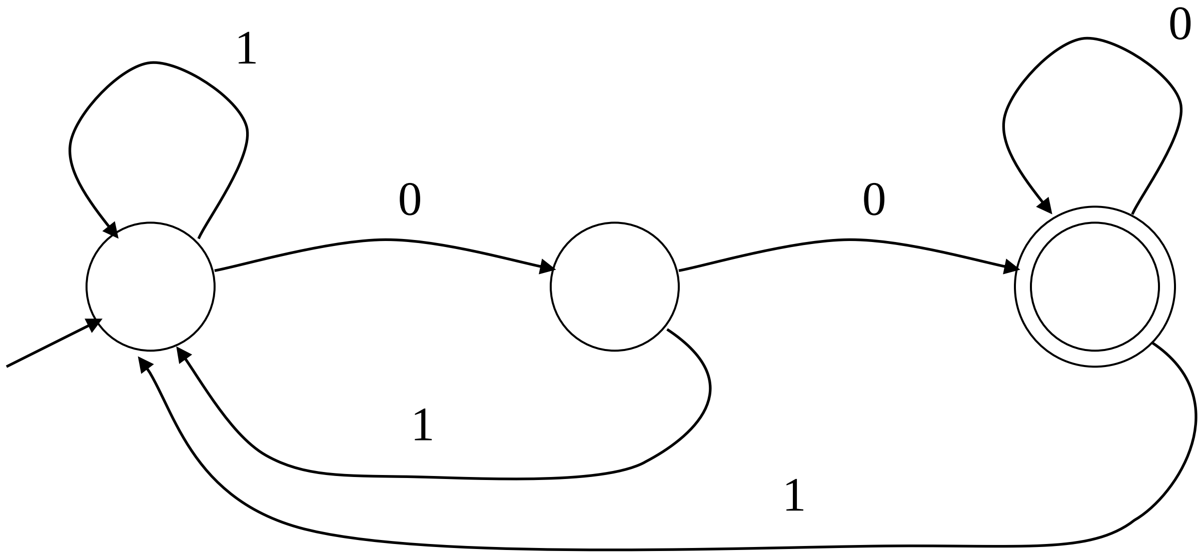
DFA that recognizes integer literals

- with an optional + or - sign:



And another (more abstract) example

- Alphabet {0,1}
- What strings does this recognize?



Formal Definition

A finite automaton is a 5-tuple $(\Sigma, Q, \Delta, q, F)$ where:

- Σ : an input alphabet
- Q: a set of states
- q: a start state q
- F: a set of final states $F \subseteq Q$
- Δ: a state transition function: $Q \times \Sigma \to Q$
  (i.e., encodes transitions state $\to^{\text{input}}$ state)

Language defined by DFA

- The language defined by a DFA is the set of strings accepted by the DFA.
  - in the language of the identifier DFA shown above:
    - x, tmp2, XyZzy, position27.
  - *not* in the language of the identifier DFA shown above:
    - 123, a?, 13apples.

NFAs

Deterministic vs. Nondeterministic Automata

Deterministic Finite Automata (DFA)

- in each state, at most one transition per input character
- no ε-moves: each transition consumes an input character

Nondeterministic Finite Automata (NFA)

- allows multiple outgoing transitions for one input
- can have $\boxed{\varepsilon}$-moves

Finite automata need finite memory

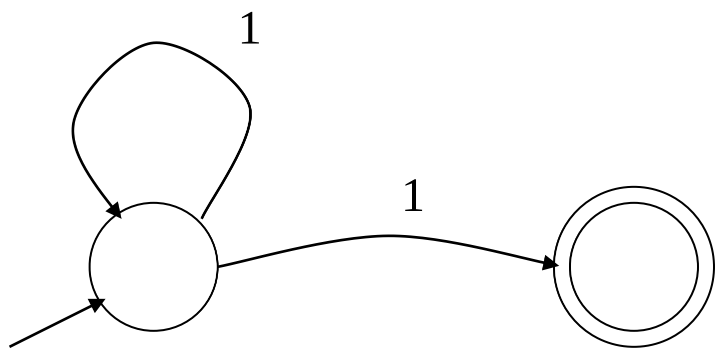
- we only need to encode the current state

NFA's can be in $\underline{\text{multiple states}}$ at once

- $\underline{\text{still a finite set}}$

A simple NFA example

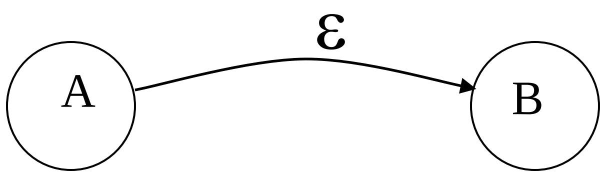
- Alphabet: { 0, 1 }



- Nondeterminism:
  - when multiple choices exist, automaton "magically" guesses which transition to take so that the string can be accepted
  - on input "11" the automaton could be in either state

Epsilon Moves

Another kind of transition: ε-moves



machine allowed to move from state A to state B without consuming an input character

Execution of Finite Automata

A DFA can take only one path through the state graph

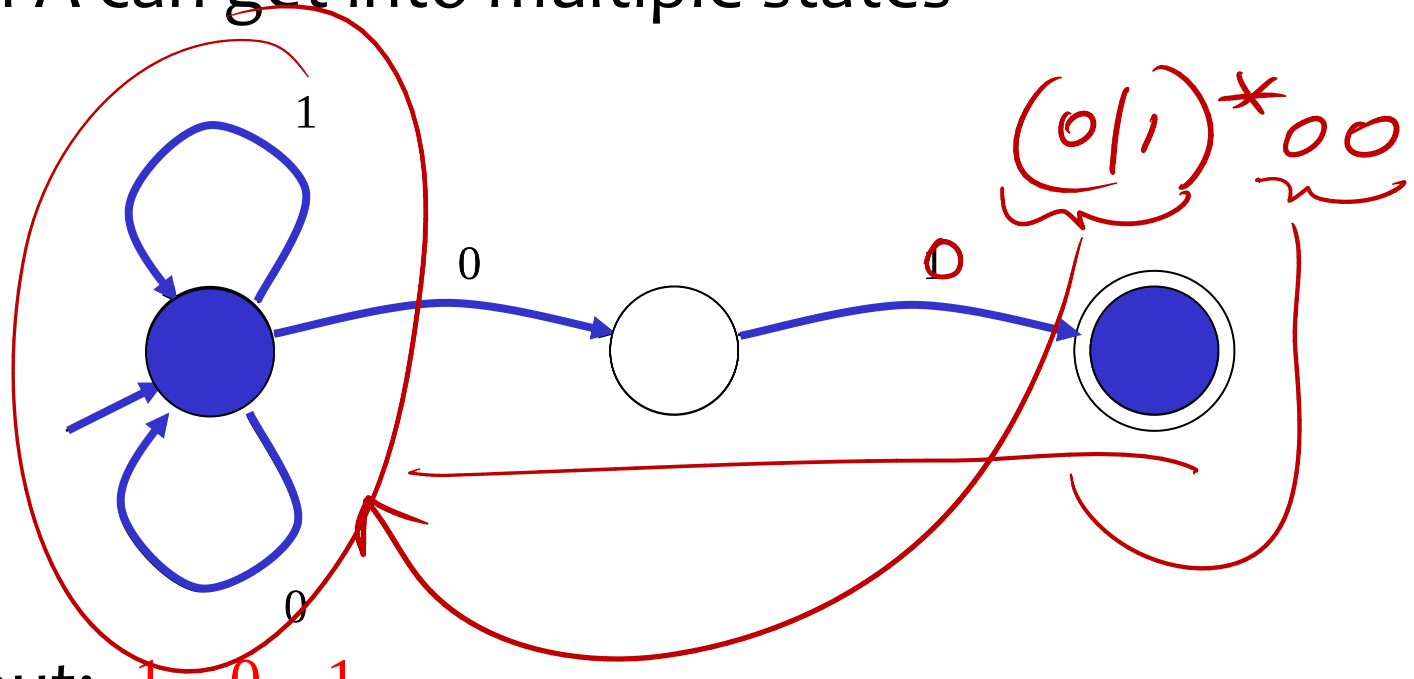
- completely determined by input

NFAs can choose

- whether to make ε-moves
- which of multiple transitions for a single input to take
- so we think of an NFA as being in one of multiple states (see next example)

Acceptance of NFAs

An NFA can get into multiple states



Input: 1 0 1

Rule: NFA accepts if it can get into a final state

NFA vs. DFA (1)

NFA's and DFA's are equally powerful

- each NFA can be translated into a corresponding DFA
  - one that recognizes same strings
- NFAs and DFAs recognize the same set of languages
  - called regular languages

NFA's are more convenient ...

- allow composition of automata

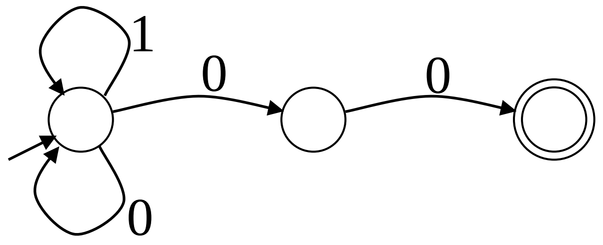
... while DFAs are easier to implement, faster

- there are no choices to consider
- hence automaton always in at most one state

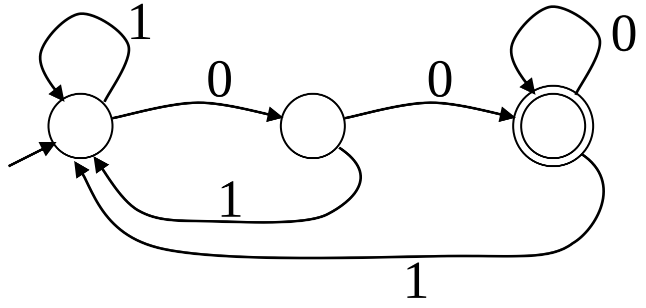
NFA vs. DFA (2)

For a given language the NFA can be simpler than a DFA

NFA



DFA



DFA can be exponentially larger than NFA

Compiling r.e. to NFA

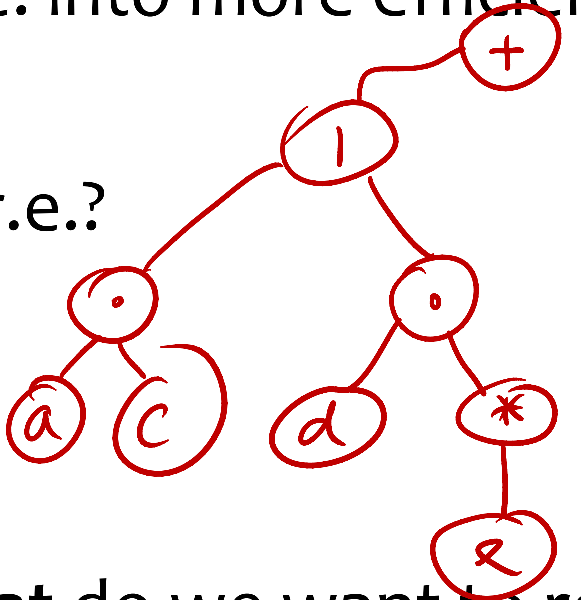
How would you proceed?

Representing regular expressions

We want to compile a r.e. into more efficient code.

How do we represent a r.e.?

```
(a.c|d.e*)+
```



Abstract Syntax tree of the r.e.

First we need to ask: **what** do we want to represent?

What is a **data structure** to represent this?

Example

`(a.c|d.e*)+`

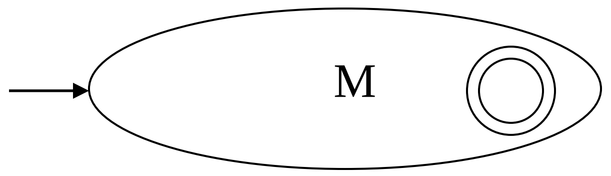
/[^ "]+/g

$$\underbrace{a/b/g}_{\text{R.e.}}$$

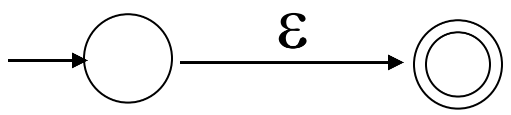
Regular Expressions to NFA (1)

For each kind of rexp, define an NFA

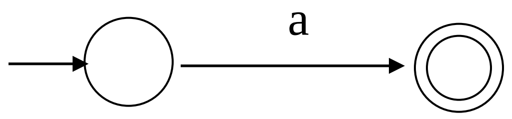
- Notation: NFA for rexp M



- For ε:

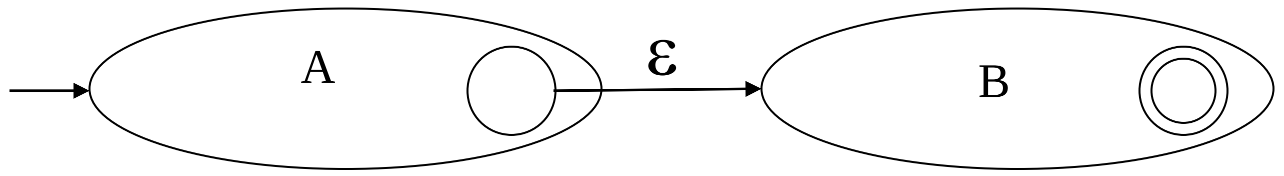


- For literal character a:

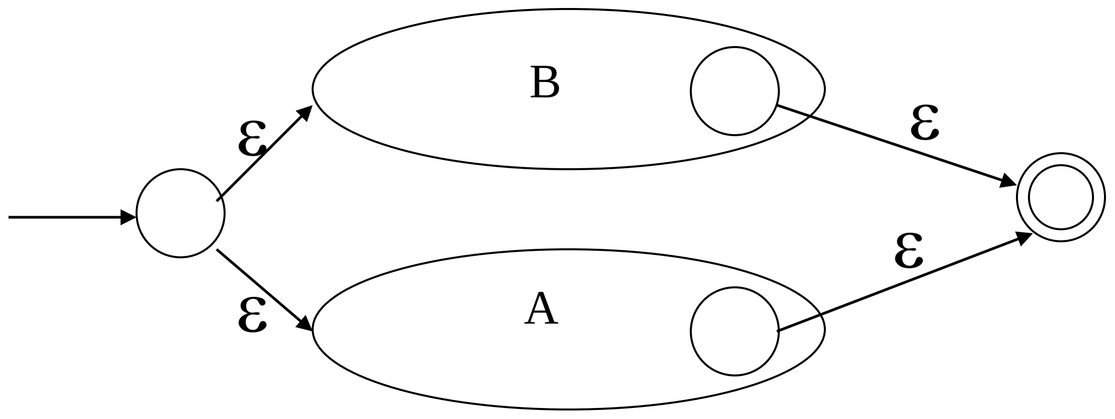


Regular Expressions to NFA (2)

For A . B

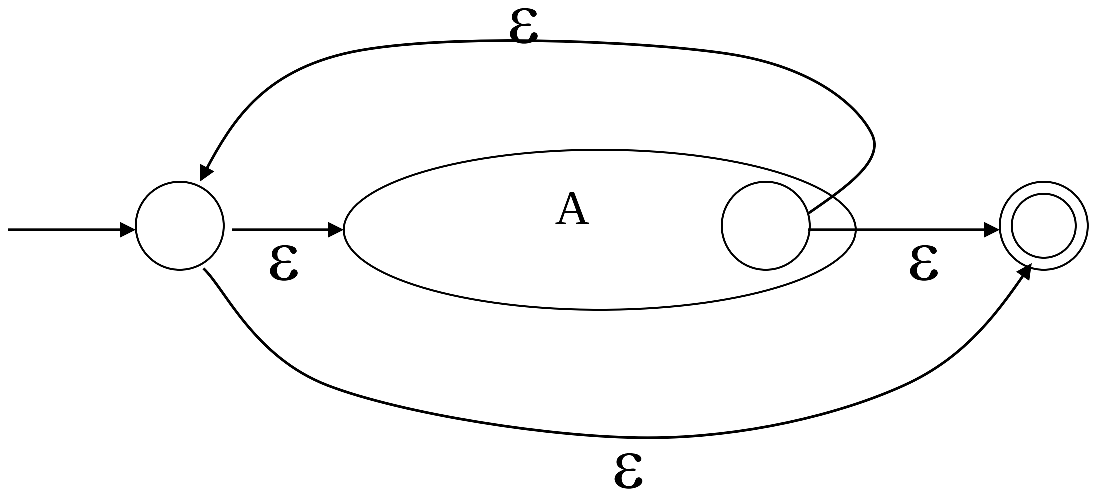


For A | B



Regular Expressions to NFA (3)

For A*

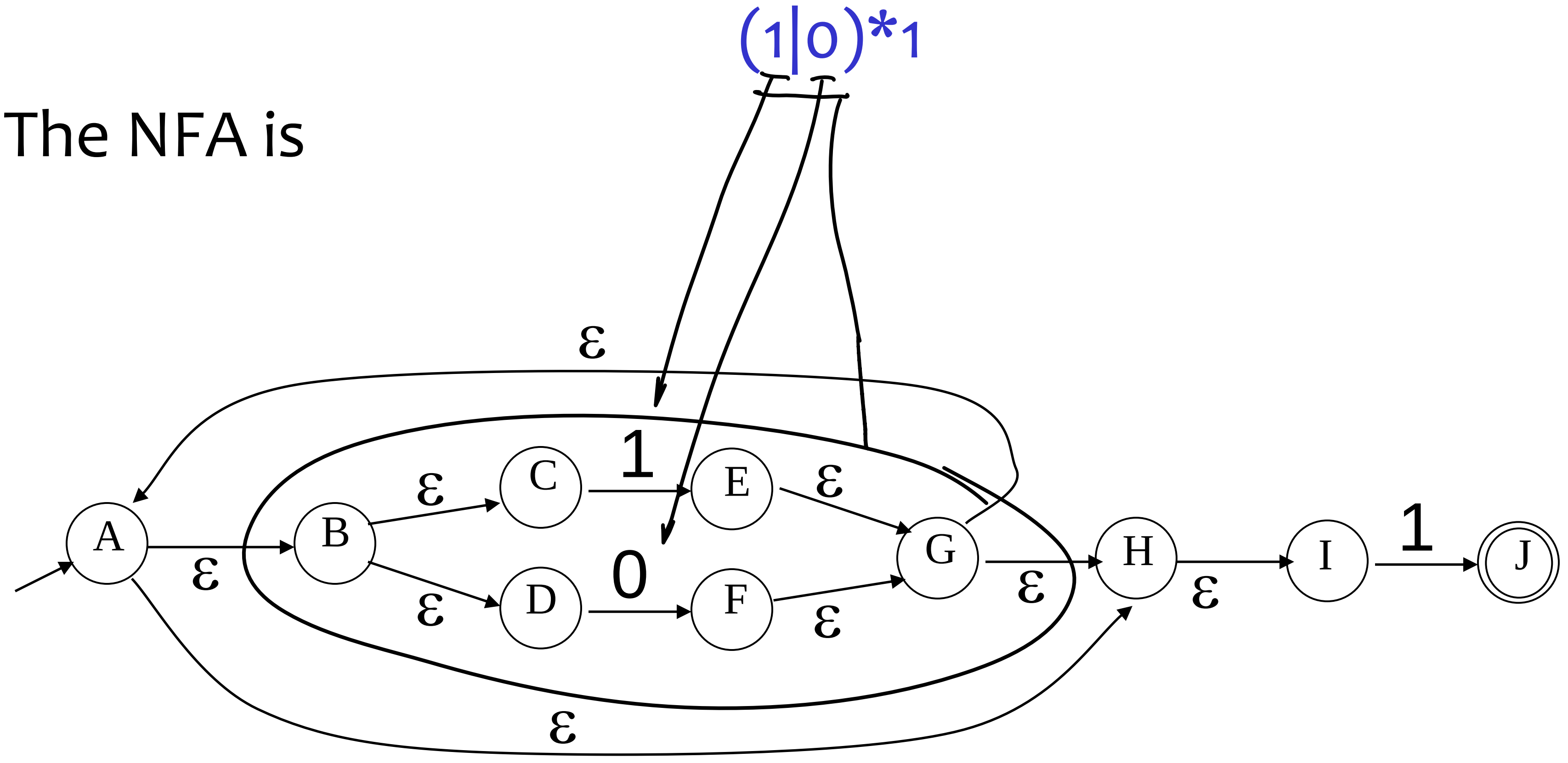


Example of RegExp -> NFA conversion

Consider the regular expression

(1|0)*1

The NFA is



Summary of DFA, NFA, Regexp

What you need to understand and remember

- what is DFA, NFA, regular expression
- the three have equal expressive power
- what is the "expressive power"
- you can convert
  - RE → NFA → DFA
  - NFA → RE
  - and hence also DFA → RE, because DFA is a special case of NFA
- NFAs are easier to use, more costly to execute
  - NFA emulation O(S^2)-times slower than DFA
  - conversion NFA→DFA incurs exponential cost in space

Some of these concepts will be covered in the section

String Manipulation Languages

CS164: Introduction to Programming Languages and Compilers

Lectures 2 and 3, Fall 2009

Instructor: **Ras Bodik**

GSI: **Joel Galenson**

UC Berkeley

Hint to *question for bored students*

Q: In lexical analysis, tokens typically flow to compiler, and no information flows back. In which language does the compiler need to communicate back to scanner? Give a scenario.

The language: JavaScript

The scenario: consider this code fragment: e/f/g

Find an explanation (coming in a few slides)

Overview for today

- Practice and theory remixed

Questions about Project 1?

Due in a week

Designed so that you learn a lot by typing very little

Remember our string processing problems

1. Web scraping and rewriting (your Project 1)

   Find and linkify mailing addresses

2. Cucumber, a Ruby testing framework

   `When I go to "Accounts" Then I should see link "My Savings"`

3. Lexical analysis

   float x=3.13 --> FLOATTYPE, ID("x"), EQ, FLOATLIT(3.14)

4. File name manipulation language

   ${dir}\foo --> "c:\\MyDir\\foo"

5. Search and replace

   \`quoted text’ D\`Souza --> \`quoted text’ D’Souza

What "language primitives " handle all these?

Accept: the whole string match

Does the entire string ***s*** match a pattern ***r***?

Match: a prefix match

Does some prefix of ***s*** match a pattern ***r***?

Search: find a substring

Does a substring of ***s*** match a pattern ***r***?

Tokenize: Lexical analysis

Partition ***s*** into lexemes, each accepted by a pattern ***r***

Extract: as match and search but extract substrings

Pattern ***r*** indicates, with ( ), which substrings to extract

Replace: replace substrings found with a new string

String search in JavaScript (for you reference)

Basic search methods for String objects:

```
"string".indexOf("rin")                        →  2
"string".indexOf(new RegExp("r*n"))            →  -1
"string".search(new RegExp("r*n"))             → 4
"string".search(new RegExp("r.*n"))            → 2
"string".search(/r.*n/)                        → 2
"string".match(/tri|str/)                      → ["str"]
"string".match(/ri|st/g)                       → ["st", "ri"]
"string".match(/tri|str/g)                     → ["str"]
```

Same for Python (for you reference)

Basic search methods for String objects:

```
re.match(r"tri|rin", "string")              → no match
re.search(r"tri|rin", "string").group(0)    → 'tri'
re.compile(r"tri|str").findall("string")    →  ['str']
re.compile(r"ri|st").findall("string")      →  ['st', 'ri']
re.search(r"(tr)|(in)", "string").groups()  →  ('tr', None)
```

note: match() expects the match to start at index 0

Problem 5: search and replace

Quite a few concepts can be demoed here.

```
text = "  `quoted text’ D`Souza  "
```

Want to convert it to

```
text = "  `quoted text’ D’Souza  "
```

Need to find ` and ' that is surrounded by cap letters

```
print(re.sub(r"([A-Z])['`]([A-Z])", "\\1\x92\\2", text))
```

and replace it. Explanations

- Sub replaces first match in text with new string
- \1 stands for text captured by the first group, ([A-Z])
- \\1 makes sure that re engine receives \1
- \x92 is ’

Lexical analysis (simplified to its essence)

Given a definition of lexical structure of the language

r = [a-zA-Z][a-zA-Z]* | [0-9]+ | \+ | - | * | / | /[^/]*/g?

partition the input

so that each partition (lexeme) matches pattern r (Cond 1)

What if multiple partitions exists?

that is, multiple ways to split the input

==> Strengthen condition (1) to ensure unique partition

So that the language is unambiguously defined

Common disambiguating rule: *maximal munch*

Scan left-to right; each lexeme consumes as much as possible

Scanning JavaScript (answer to bored Q)

Key fragment of lexical definition (lexemes)

```
r = [a-zA-Z][a-zA-Z]* | [0-9]+ | \+ | - | \* | / | /[^/]*/g?
```

Consider this input fragment

```
... e/f/g ...
```

Is it ID,DIV,ID,DIV,ID or ID,REGEX ?

Maximal munch produces the latter.

But we sometimes want the former.

When? Depends on the context.

The parser can tell the scanner whether, when it encounters '/' it is expecting DIV or REGEX.

Another puzzle

Use a regex to test whether a number is prime.

Performance depends on more than input size

Consider regular expression `X(.+)+X`

- Input1: `"X==================X"` match
- Input2: `"X===================="` no match

JavaScript: `"X====…=========X".search(/X(.+)+X/)`

- Match: fast   No match: **slow**

Python: `re.search(r'X(.+)+X','=XX=======…=====X=')`

- Match: fast   No match: **slow**

awk: `echo '=XX====…======X=' | gawk '/X(.+)+X/'`

- Match: fast   No match: **fast**

Blog post of the day

Regular Expressions: Now You Have Two Problems

I love regular expressions. No, I'm not sure you understand: I really *love* regular expressions.

You may find it a little odd that a hack who grew up using a language with the ain't keyword would fall so head over heels in love with something as obtuse and arcane as regular expressions. I'm not sure how that works. But it does. **Regular expressions rock.** They should absolutely be a key part of every modern coder's toolkit.

If you've ever talked about regular expressions with another programmer, you've invariably heard this 1997 chestnut:

Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems.

Representing regular expressions

We want to compile a r.e. into more efficient code.

How do we represent a r.e.?

```
    (a.c|d.e*)+
```

First we need to ask: **what** do we want to represent?

What is a **data structure** to represent this?

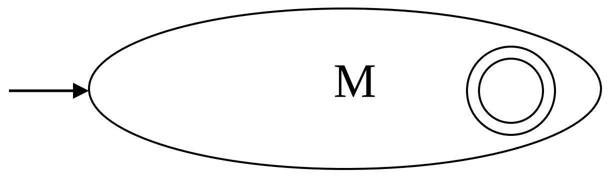
Example of abstract syntax tree (AST)

```
(a.c|d.e*)+
```

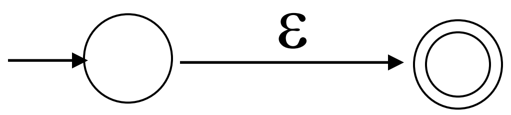
Regular Expressions to NFA (1)

For each kind of rexp, define an NFA

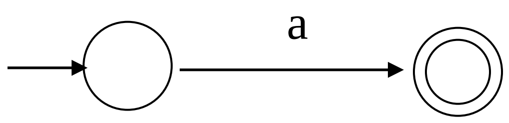
- Notation: NFA for rexp M



- For ε:

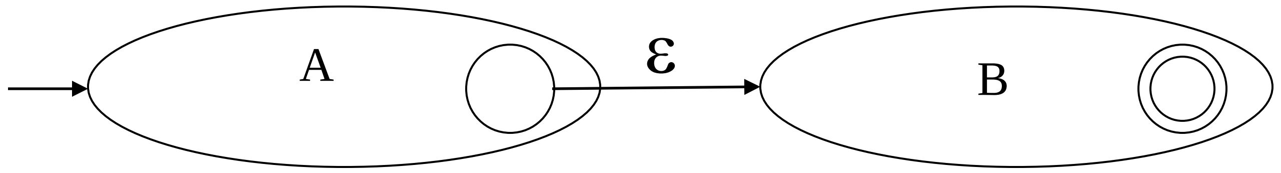


- For literal character a:

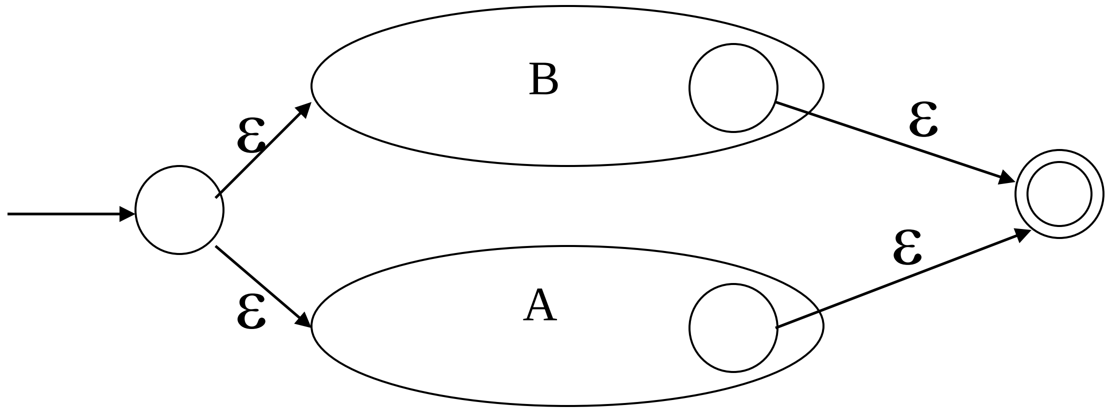


Regular Expressions to NFA (2)

For A . B

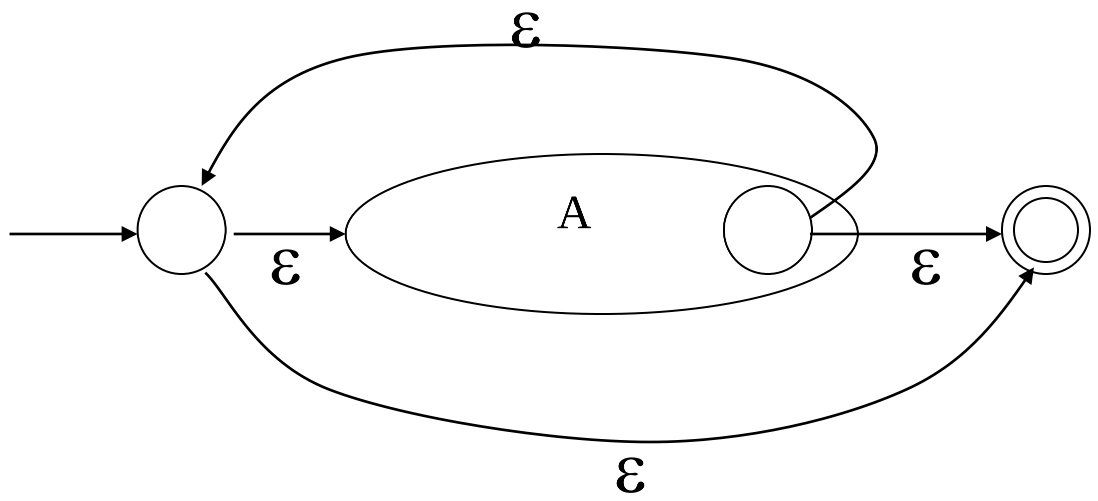


For A | B



Regular Expressions to NFA (3)

For A*

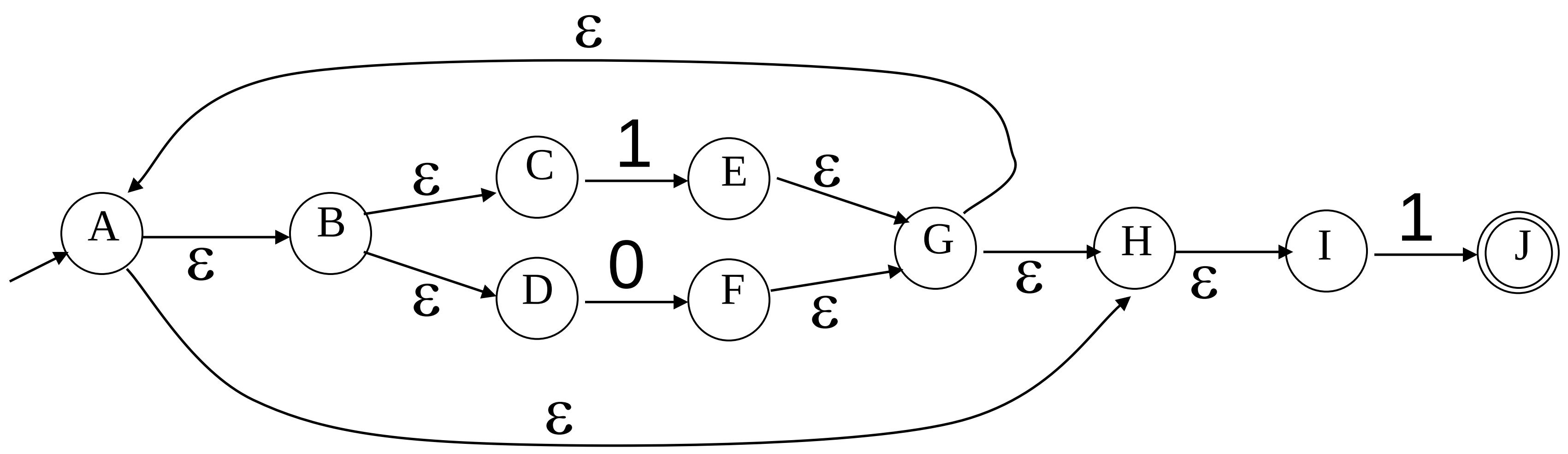


Example of RegExp -> NFA conversion

Consider the regular expression

(1|0)*1

The NFA is



Answer to puzzle: Primality Test

First, represent a number n as a unary string

    7 == '1111111'

Conveniently, we'll use Python's * operator

    str = '1'*n   # concatenates '1' n times

n not prime if str can be written as ('1'*k)*m, k>1, m>1

    (11+)\1+   # recall that \1 matches whatever (11+) matches

Special handling for n=1. Also, $ matches end of string

    re.match(r'1$|(11+)\1+$', '1'*n)

Note this is a *regex*, not a regular expression

    Regexes can tell apart strings that reg expressions can't

Expressiveness of recognizers

What does it mean to "tell strings apart"?

- Or "test a string" or "recognize a language", where language = a (potentially infinite) set of strings

It is to accept only a string with that has some property

- such as can be written as ('1'*k)*m, k>1, m>1
- or contains only balanced parentheses: ((())()(()))

Why can't a reg expression test for ('1'*k)*m, k>1,m>1 ?

- Recall reg expression: char . | *
- We can use sugar to add e+, by rewriting e+ to e.e*
- We can also add e++, which means 2+ of e: e++ --> e.e.e*

… continued

So it seems we can test for ('1'*k)*m, k>1,m>1, right?

(1++)++	rewrite 1++ using e++ --> e.e+
(11+)++	rewrite (11+)++ using e++ --> e.e+
(11+)(11+)+	

Now why isn't (11+)(11+)+ the same as (11+)\1+ ?

How do we show these test for different property?

Example from Jeff Friedl's book

Imagine you want to parse a config file:

```
   filesToCompile=a.cpp b.cpp
```

The regex for this command line format:

```
   [a-zA-Z]+=.*
```

Now let's allow an optional \n-separated 2nd line:

```
   filesToCompile=a.cpp b.cpp \<\n>
                  d.cpp e.h
```

We extend the original regex:

```
    [a-zA-Z]+=.*(\\\n.*)?
```

This regex does not match our two-line input. Why?

What compiler textbooks don’t teach you

The textbook *string matching* problem is simple:

*Does a regex r match the **entire** string s?*

- a clean statement and suitable for theoretical study
- here is where regexes and FSMs are equivalent

The matching problem in the Real World:

*Given a string s and a regex r, find a **substring** in s matching r.*

Do you see the language design issue here?

- There may be many such substrings.
- We need to decide **which** substring to find.

It is easy to agree where the substring should start:

- the matched substring should be the **leftmost** match

Two schools of regexes

They differ in where it should end:

Declarative approach: longest of all matches

- conceptually, enumerate all matches and return longest

Operational approach: define behavior of *, | operators

e* match e as many times as possible while allowing the remainder of the regex t o match

e|e select leftmost choice while allowing remainder to match

```
filesToCompile=a.cpp b.cpp \<\n> d.cpp e.h
```

```
[a-zA-Z]+  = .* ( \\ \n .* )?
```

These are important differences

We saw a non-contrived regex can behave differently

- personal story: I spent 3 hours debugging a similar regex
- despite reading the manual carefully

The (greedy) operational semantics of *

- does not guarantee longest match (in case you need it)
- forces the programmer to reason about backtracking

It seems that backtracking is nice to reason about

- because it’s local: no need to consider the entire regex
- cognitive load is actually higher, as it breaks composition

Where in history of *re* did things go wrong?

It's tempting to blame perl

- but the greedy regex semantics seems older
- there are other reasons why backtracking is used

Hypothesis 1:creators of re libs knew not that NFA can

- can be the target language for compiling regexes
- find all matches simultaneously (no backtracking)
- be implemented efficiently (convert NFA to DFA)

Hypothesis 2: their hands were tied

- Ken Thompson's algorithm for re-to-NFA was patented

With backtracking came the greedy semantics

- longest match would be expensive (must try all matches)
- so semantics was defined greedily, and non-compositionally

Concepts

- Syntax tree-directed translation (re to NFA)
- recognizers: tell strings apart
- NFA, DFA, regular expressions = equally powerful
- Syntax sugar: e+ to e.e*
- Compositionality: be weary of greedy semantics
- Metacharacters: characters with special meaning