Lecture 10

Regular Expressions

regular expressions are a small language implemented with automata or backtracking

Ras Bodik
Shaon Barman
Thibaud Hottelier

Hack Your Language!

CS164: Introduction to Programming Languages and Compilers, Spring 2012

UC Berkeley

Announcement

The CS164 Parser Contest.

Part 1: speed of your HW4 parser

- **Prize:** fun, fame, and extra credit (for speed of your HW4)
- Submit your parser for speed evaluation even before deadline. You will receive your current standing in class.

Part 2: Speed and code readability of your PA4-6

- **Prize:** fun and fame and a gift certificate.
- Measured at the end of the semester.
- Fa10 winner 5x faster than staff parser. Can you beat that?

Today is a no-laptop Thursday

If you are bored, here's a puzzle:

Write a *regex* that tests whether a number is prime.

Hint:

\n matches the string matched by the n^{th} regex group

Ex: regex `c(aa|bb)\1` matches strings `caaaa` and `cbbbb`

the prime tester must be a regex, not a regular expression!

the latter do not support \n

Where are we in CS164?

We have now built enough tools (coroutines, parser) to go after our mission: build small languages so that programmers can become more productive.

When building small languages, we program under the abstraction: we implement abstractions for others to use.

Today, we'll learn what's under the abstraction of regular expressions.

Next three lectures: small languages

What small languages have we already seen?

unit calculator, regexes, grammars, queues, jQuery, Prolog

well, Prolog is not so small; it's a rather general-purpose langauge

Today: regular expressions

the best-studied small language

We will look at three questions. How to

- **integrate** regular expressions with a host language
- **implement** regular expressions efficiently
- design composable, clear **semantics** (REs vs. regexes)

Sample Uses of Regular Expressions
and of string processing in general

Sample string processing problems

Web scraping and rewriting (HW1)

also, find and “linkify” mailing addresses

Cucumber, a Ruby testing framework with “NLP”

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

Lexical analysis in interpreters and compilers

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

Config tools include “file name manipulation” languages

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

Editors, search and replace

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

1. Web scraping and rewriting

Rewrite a web page using GreaseMonkey scripting.

The idea: web pages are more readable when you view the print-friendly, ad-free pages. Automate this!

Approach: Your script will rewrite links on a page to point to the print-friendly version of target page.

How: When user clicks on a link, fetch (but do not display) the target page; use a regex to find in the target HTML text the (best-guess) link to the print-friendly page; rewrite the link to point to that page; follow the rewritten link to display the friendly page.

2. Cucumber: a Ruby testing framework

A sample Cucumber test file:

```
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"
```

Meaning of this test:

"Given" makes the script go to login to the web site.

"When" clause clicks on the link Accounts.

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

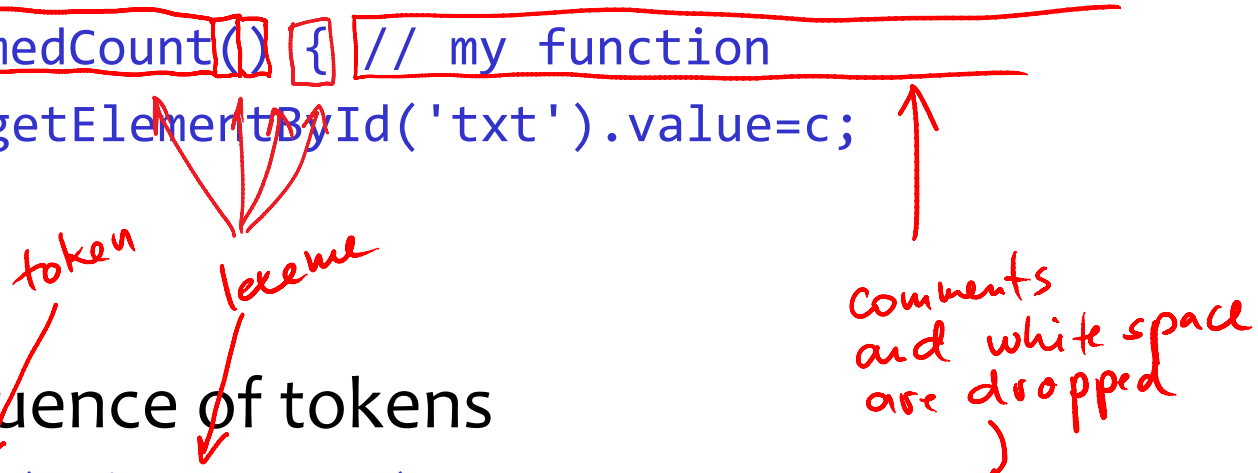
We could implement a similar tool with GreaseMonkey:

regexes can be used to parse the command.

3. Lexical analysis in a 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
```

REs facilitate concise description of tokens

Notes on lexical analysis

The lexer partitions the input into lexemes

Lexemes are mapped to tokens

The stream of tokens is fed to the parser

Some tokens are associated with their *lexemes*

Whitespace and comments are typically skipped

Another challenge question

The lexical analyzer produces a list of tokens without consulting with the parser

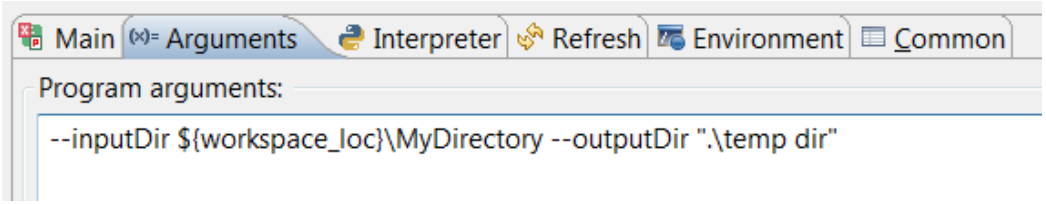
i.e., tokenizing is syntax insensitive

Q: Give a JavaScript scenario where the tokenizing depends on the context of the parser.

i.e., lexer cannot tokenize input w/out feedback from parser

4. File name processing languages

In shell scripts and IDEs, command line args can refer to variables such as workspace_loc:



This is translated into program arguments:

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

Must escape \ and quotes during translation

Tricky design. Eclipse designed this substitution language wrong: their escaping rules prevent you from expressing some values that you may want to pass into the program.

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

The challenge: your replacement should not change quotes used in quotations:

`quoted text’

Again, this can be solved conveniently with REs

Useful string processing operations

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

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

Replace: replace substrings found with a new string

Discovering automata and REs

a revisionist history

Programming model of a small language

Design of a small (domain-specific) language starts with the choice of a suitable programming model

- Why is a new model may be needed? Because procedural abstraction (a procedure-based library) cannot always hide the plumbing (implementation details)

For string-based processing, automata and regular expressions are usually the right programming model

- regexes are hard to hide under a procedure library, although HW2 showed how to do it with coroutines.

Let's use string processing as a case study on how to discover the programming model for small languages

Let's write a lexer (a.k.a. tokenizer, scanner)

First, we'll write the scanner by hand

- We'll examine the scanner's repetitious plumbing
- Then we'll hide the plumbing in a programming model

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 break 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., "a string literal missing a closing quote

Real-world imperative scanners can get unwieldy

- the lexical structure of the language may be hard to read out
- the scanner code obscures it by spreading the string comparisons and other actions across the scanner code (rather than keeping it in a single specification table)

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

Where is the logic?

```
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;
}
```

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	"*"

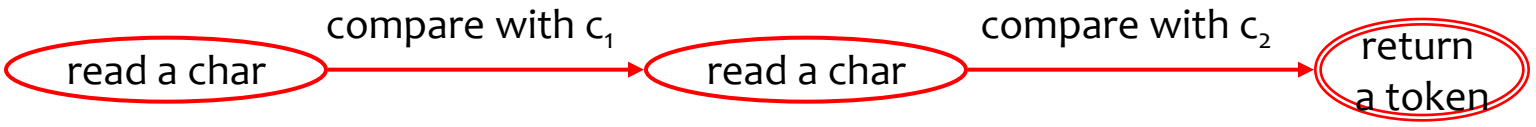
Separate out the how (plumbing)

The code actually follows a simple pattern:

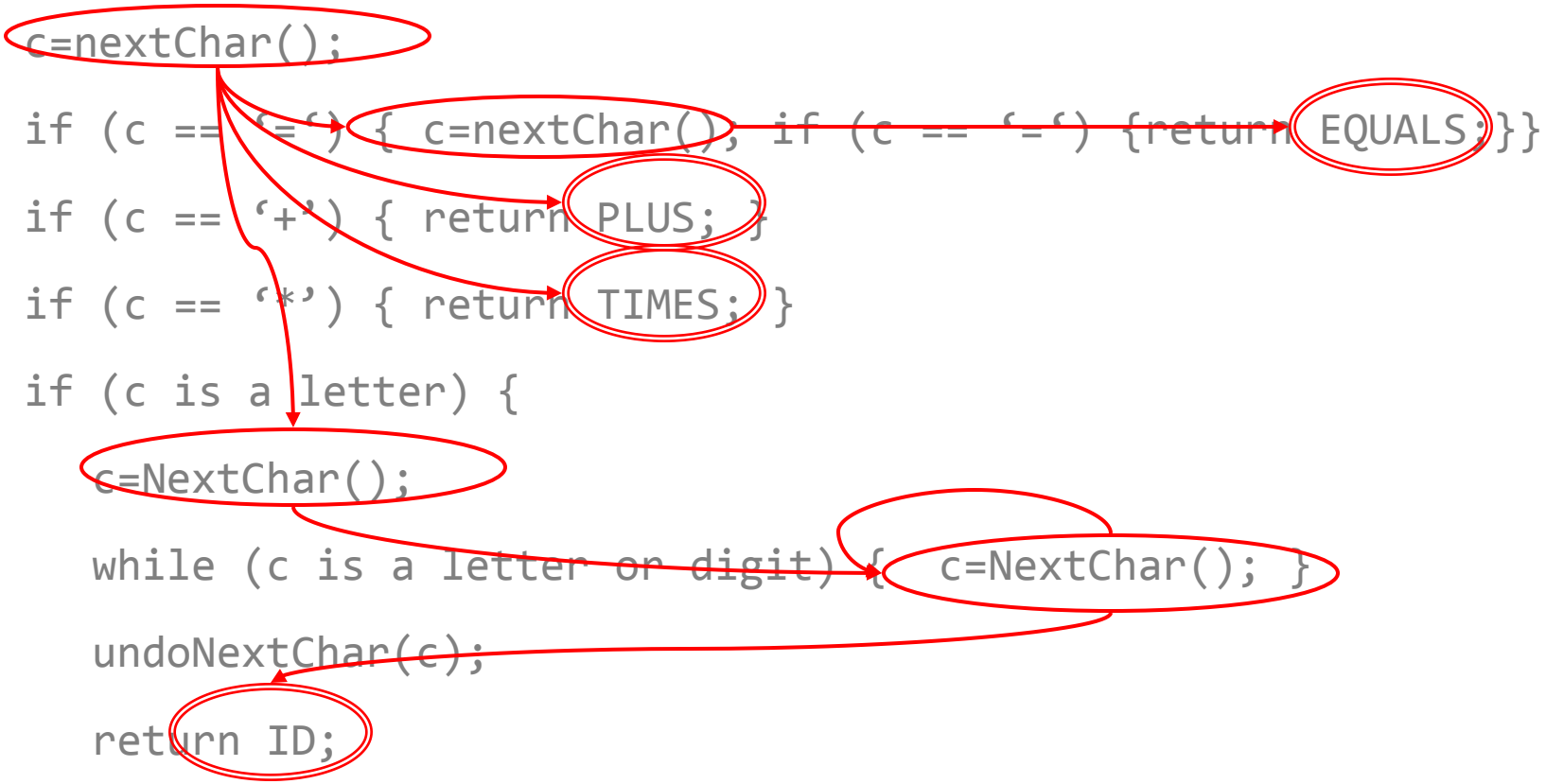
- read next char,
- compare it with some predetermined char
- if matched, jump to the next read of next char
- repeat this until a lexeme is built; then return a token.

What's a programming model for encoding this?

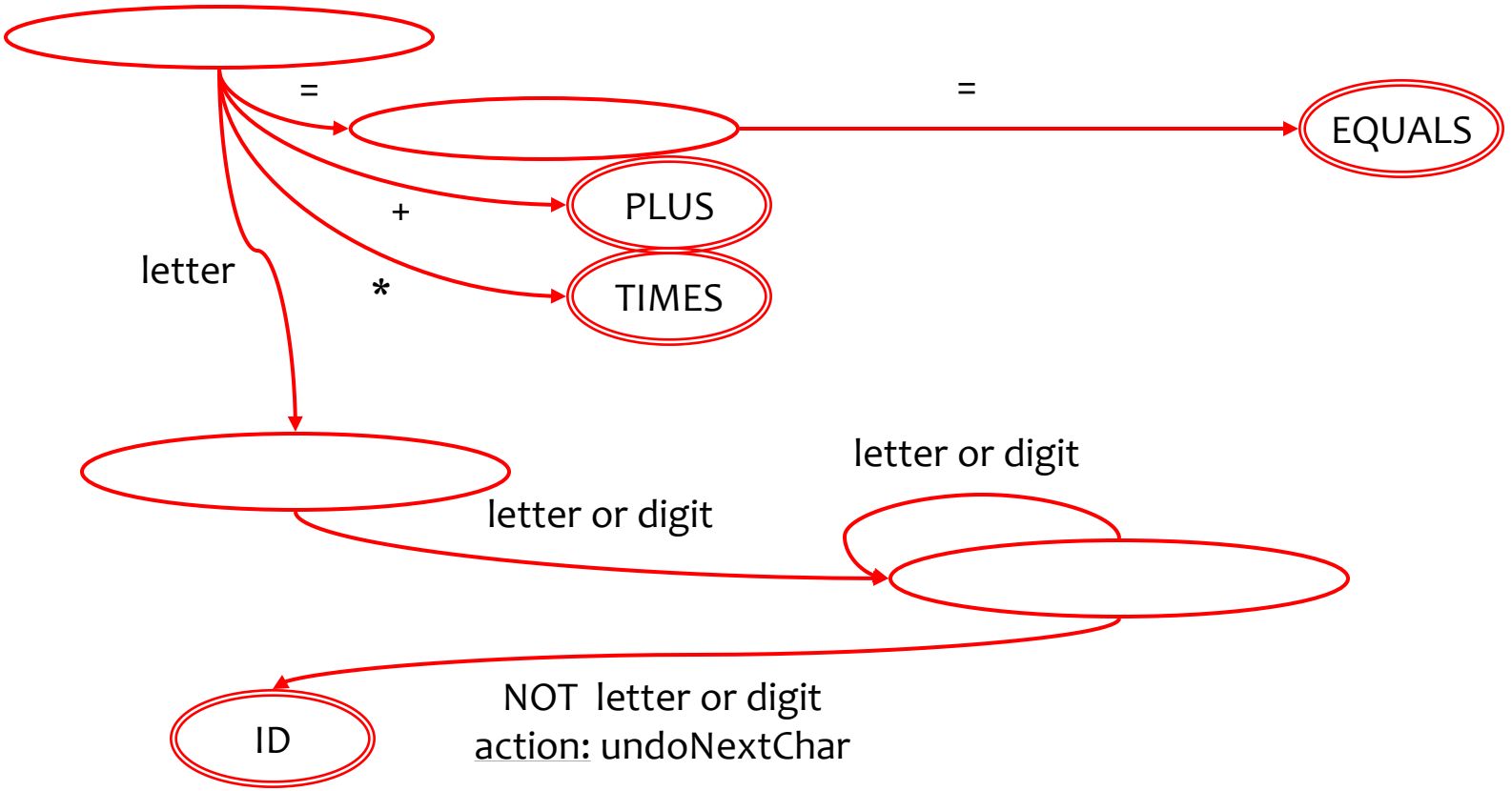
- **finite-state automata**
- finite: number of states is fixed, i.e., input independent



Separate out the what



Here is the automaton; we'll refine it later



A declarative scanner

Part 1: declarative (the what)

describe each token as a finite automaton

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

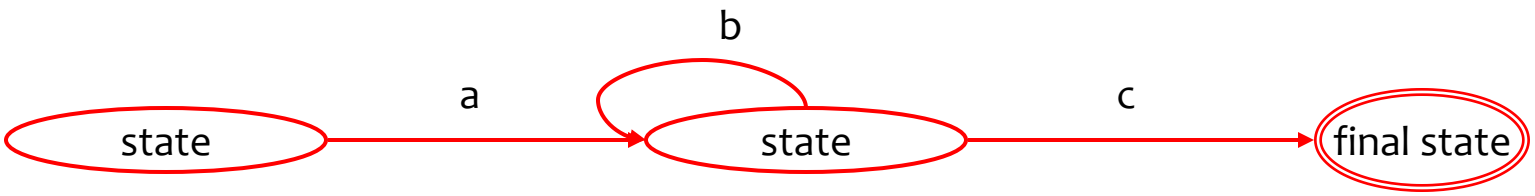
Part 2: operational (the how)

connect these automata into a scanner automaton

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

Automata are hard to draw. Need a notation.

For convenience and clarity, we want text notation



Kleene invented regular expressions for the purpose:

a b	a followed by b (sometimes written a.b)
a*	zero or more repetitions of a
a | b	a or b

don't confuse with the dot that means any character

Our example: ab*c

Regular expressions

Regular expressions contain:

- characters : these must match the input string
- meta-characters: these serve as operators (*, |, [,], etc)

Operators operate on REs (it's a recursive definition)

char	any character is a regular expression
r_1 r_2	so is r_1 followed by r_2
r*	zero or more repetitions of r
r_1 | r_2	match r_1 or r_2
r+	one or more instances of r, desugars to rr*
[1-5]	same as (1|2|3|4|5) ; [ ] denotes a *character class*
[^a]	any character but a
\d	matches any digit
\w	matches any letter

Blog post du jour

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.

...

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.

Implementing Regular Expressions

Compare performance of RE in three languages

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**

Sidenote: compare how the three languages integrate regular expressions.

This problem occurs in practice: HW1 fall 2010

Problem: Find mailing addresses in HTML and wrap them in links to google maps.

From the course newsgroup: "I had been experiencing the same problem -- some of my regex would take several minutes to finish on long pages.

`/((\w*\s*)*\d*)*Hi There/` times out on my Firefox.

`/Hi There((\w*\s*)*\d*)*/` takes a negligible amount of time.

It is not too hard to see why this is.

To fix this, if you have some part of the regex which you know must occur and does not depend on the context it is in (in this example, the string "Hi There"), then you can grep for that in the text of the entire page very quickly. Then gather the some number of characters (a hundred or so) before and after it, and search on that for the given regex. …

I got my detection to go from several minutes to a second by doing just the first."

Why do implementations differ?

Some are based on backtracking (can be slow)

- to conclude that X========== does not match X(.*)*X, backtracking needs to try all ways to match the string, including: (==)(===)(=)… and (=)(=)(==)… and …

Some are based on automata

- automata can keep track of all possible matches at once

There are semantic differences between the two

- see Lecture 11

Implementation via backtracking

Implementation with backtracking via Prolog

Turn RE into a context-free grammar.

Build the parser for the grammar with Prolog (see L8).

RE: a*

Grammar: A ::= aA | ε

Backtracking parser:

```
a([], []).
a([a|R], Out) :- a(R,Out).
parse(S) :- a(S,[]).
```

RE: (ba*)*

Grammar: S ::= bA | ε

A ::= aA | ε

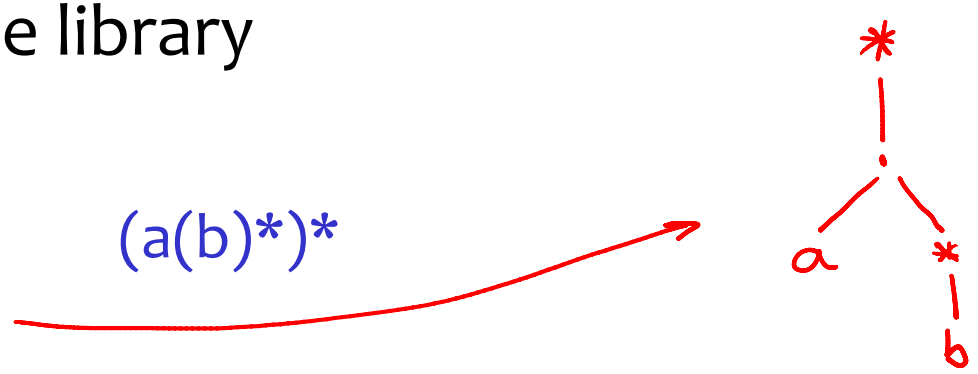
Implementation with btracking via coroutines

Step 1: Use SDT to compile RE to AST

Step 2: Walk over AST and generate code to HW2 RE coroutine library

RE: (a(b)*)*

AST:



Generated code (approximate):

```
match(s,star(seq(char("a", star(prim("b"))))))
```

SDT to obtain AST of an RE

```
%ignore /\n+/

%%

// A regular expression grammar

R ->  'a'          %{ return ('prim', n1.val) %}
  | R R            %{ return ('seq', n1.val, n2.val) %}
  | R '*'          %{ return ('star', n1.val)        %}
  | R '|' R        %{ return ('alt', n1.val, n3.val) %}
  | '(' R ')'      %{ return n2.val %}
  ;
```

Generate code in SDT, w/out intermediate AST

```
%ignore /\n+/

%%

// A regular expression grammar

R ->  'a'                 %{ return 'prim("%s")' % n1.val           %}
  | R R         %dprec 2  %{ return 'seq(%s,%s)' % (n1.val, n2.val) %}
  | R '*'       %dprec 1  %{ return 'star(%s)' % n1.val             %}
  | R '|' R     %dprec 3  %{ return 'alt(%s,%s)' % (n1.val,n3.val)  %}
  | '(' R ')'             %{ return n2.val %}
  ;
```

Implementation via automata

Finite automata, in more detail

Deterministic (DFA):

- state transition unambiguously determined by the input
- more efficient implementation

Non-deterministic (NFA):

- state transition determined by the input and an oracle
- less efficient implementation
- lend themselves to "composability", which we'll use to compile REs to automata

DFAs

Deterministic finite automata (DFA)

We'll use DFA's as recognizers:

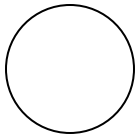
- recognizer accepts a set of strings, and rejects all others

For example, a DFA tells us if a string is a valid lexeme

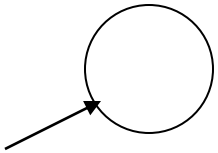
- DFA defining identifiers accepts "xyx" but rejects "+3e4"

Finite-Automata State Graphs

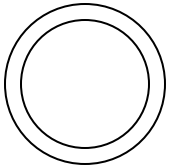
- A state



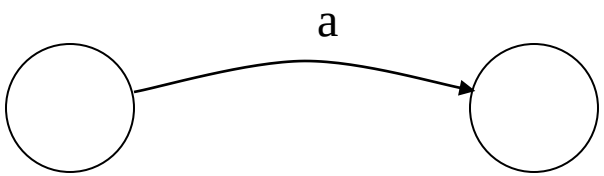
- The start state



- 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 accepted if

entire string consumed and automaton is in accepting state

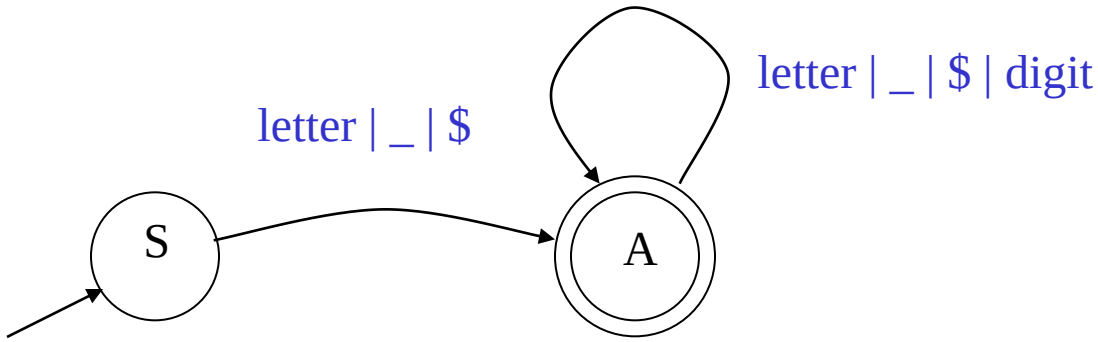
Rejected otherwise. Two possibilities for rejection:

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

Deterministic Finite Automata

Example: JavaScript Identifiers

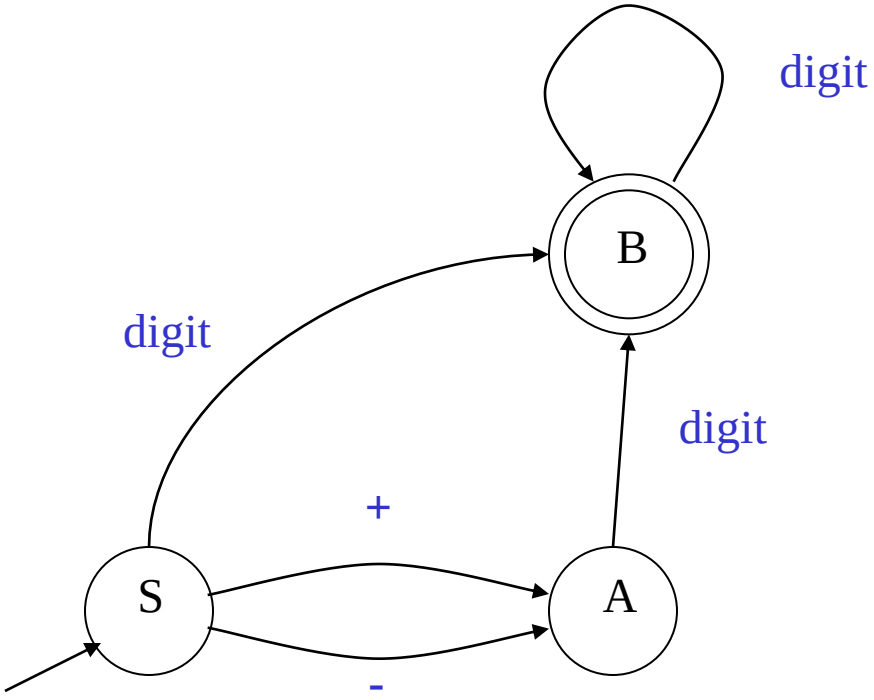
sequences of 1+ letters or underscores or dollar signs or digits, starting with a letter or underscore or a dollar sign:



Example: Integer Literals

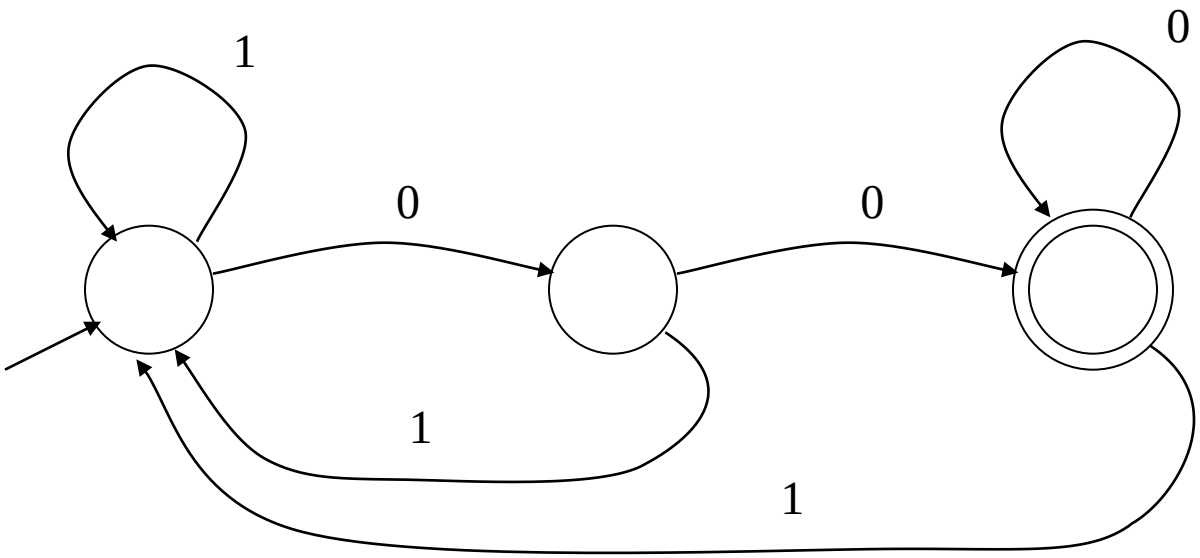
DFA that recognizes integer literals

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: $Q \times \Sigma \rightarrow Q$
  (i.e., encodes transitions state $\rightarrow^{\text{input}}$ 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 ε-moves

Finite automata need finite memory

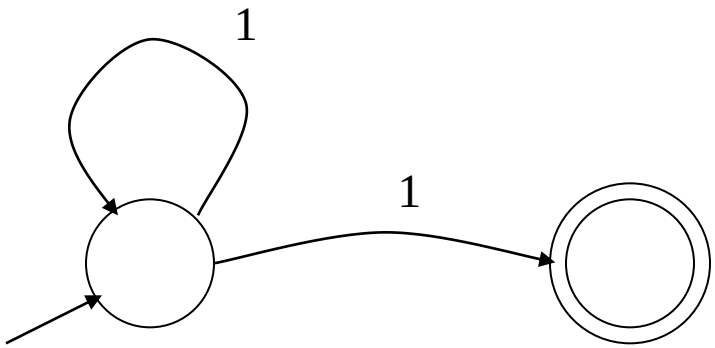
- we only need to encode the current state

NFA's can be in multiple states at once

- still a finite set

A simple NFA example

Alphabet: { 0, 1 }



Nondeterminism:

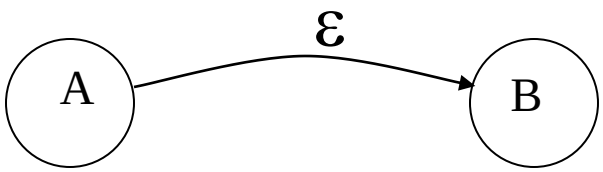
when multiple choices exist, automaton "magically" guesses which transition to take so that the string can be accepted (if it is possible to accept the string)

Example:

on input "11" the automaton could be in either state

Epsilon Moves

Another kind of transition: ε-moves



The automaton is allowed to move from state A to state B without consuming an input character

Execution of Finite Automata (DFA)

A DFA can take only one path through the state graph

- completely determined by input

Implementation: table-based

nextState := transitions[currentState, nextChar]

Execution of Finite Automata (NFA)

NFAs can choose

- whether to make ε-moves
- which of multiple transitions for a single input to take

We can think of NFAs in two alternative ways:

1) the choice is determined by an oracle

   the oracle makes a clairvoyant choice (looks ahead into the input)

2) NFAs are in several states at once (see next slide)

   these states correspond to all possible past oracle choices

We can emulate NFA

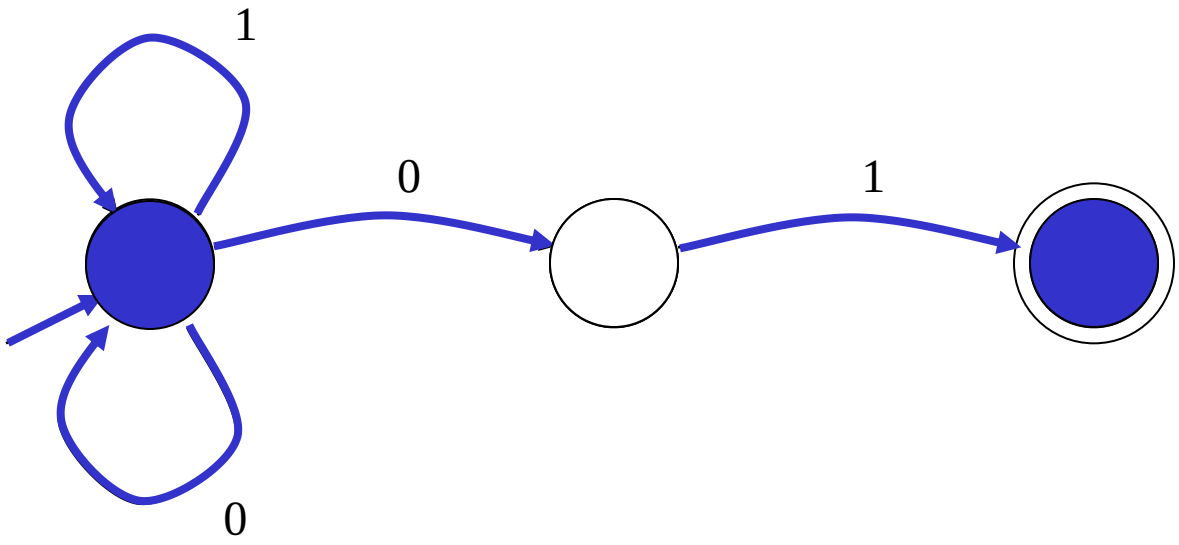
Keep track of current states. O(NS) time. S=#states

Or we can convert NFA to DFA.

O(N) matching time. But the DFA can be 2^S in size.

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

ie, there is a path from start to end labeled with the input

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

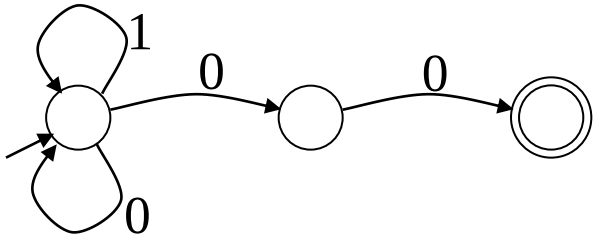
... while DFAs are easier to implement, faster

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

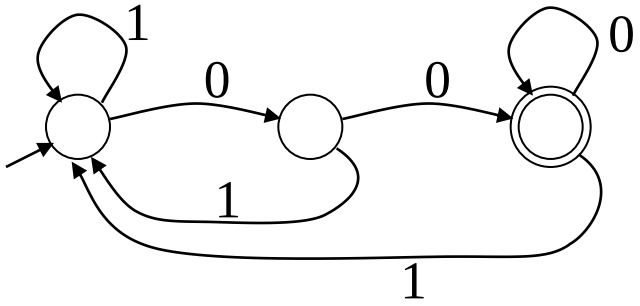
NFA vs. DFA (2)

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

NFA



DFA



DFA can be exponentially larger than NFA

eg when the NFA is translated to DFA

Translating an NFA to DFA

Key idea: NFA can be in multiple states at once.

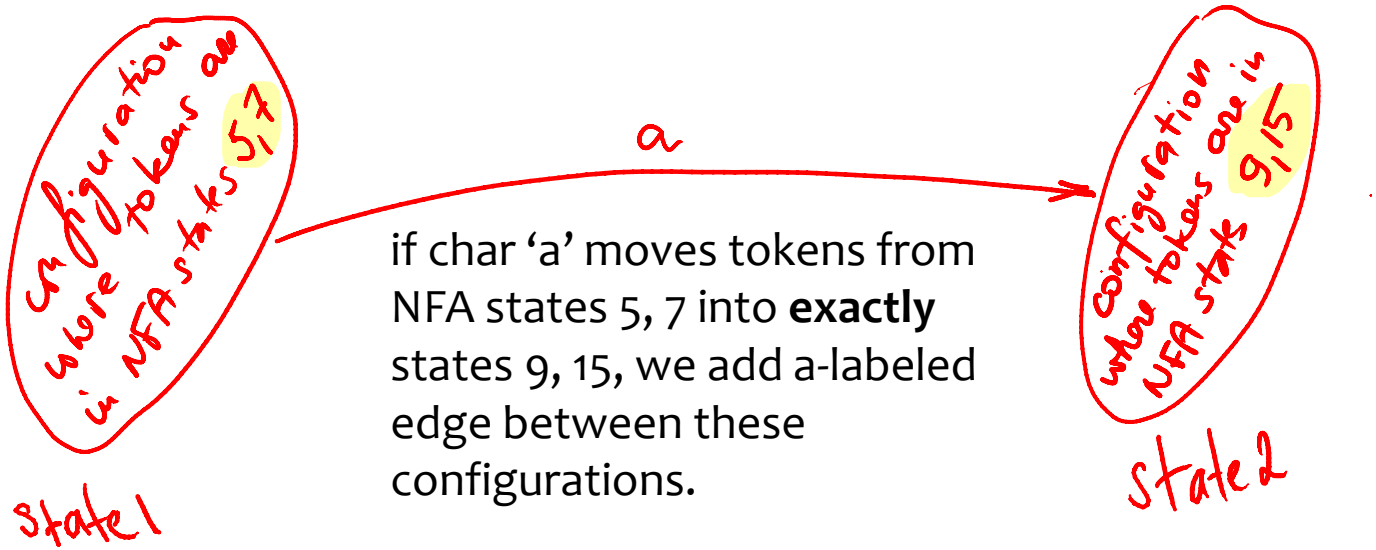
"The blue tokens can be in any subset of NFA states."

Each such subset is called a configuration.

Let's create a DFA with a state for each configuration

there are 2^N such states

How do we place transitions in this DFA?



if char 'a' moves tokens from NFA states 5, 7 into **exactly** states 9, 15, we add a-labeled edge between these configurations.

Intermission: Answer to Primality Test puzzle

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 = m*k ⇒ not prime

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

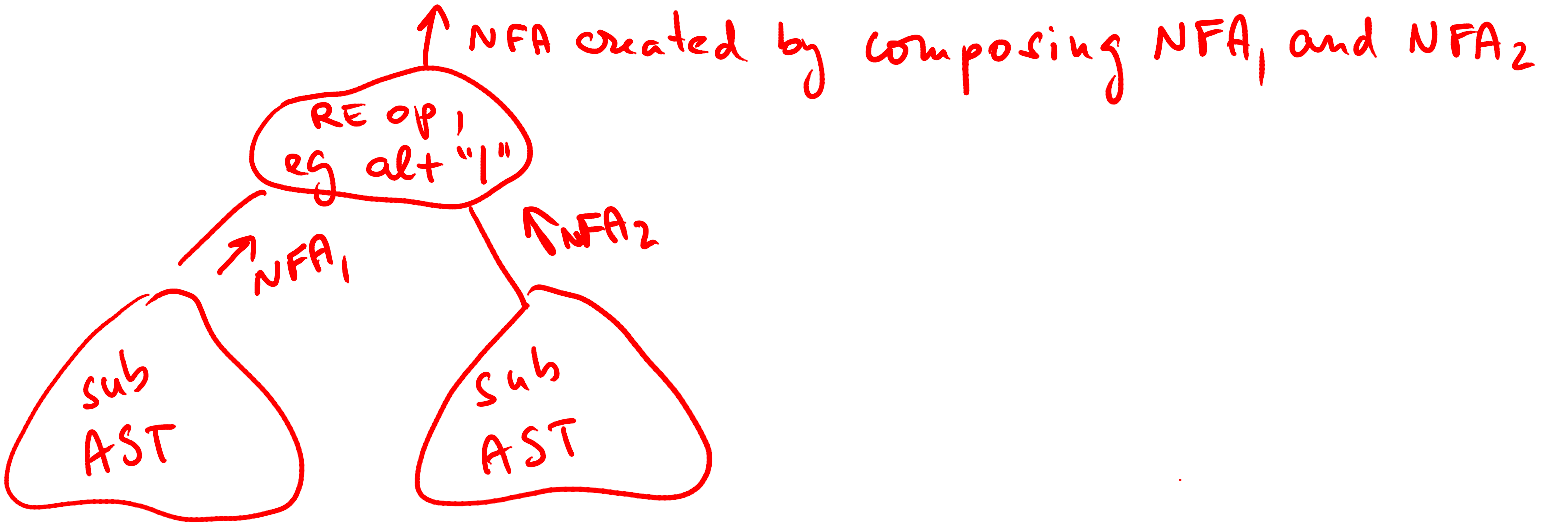
re.match(r'1$|(11+)\1+$', '1'*n) .group(1) → outputs k in unary

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

Regexes can tell apart strings that reg expressions can't

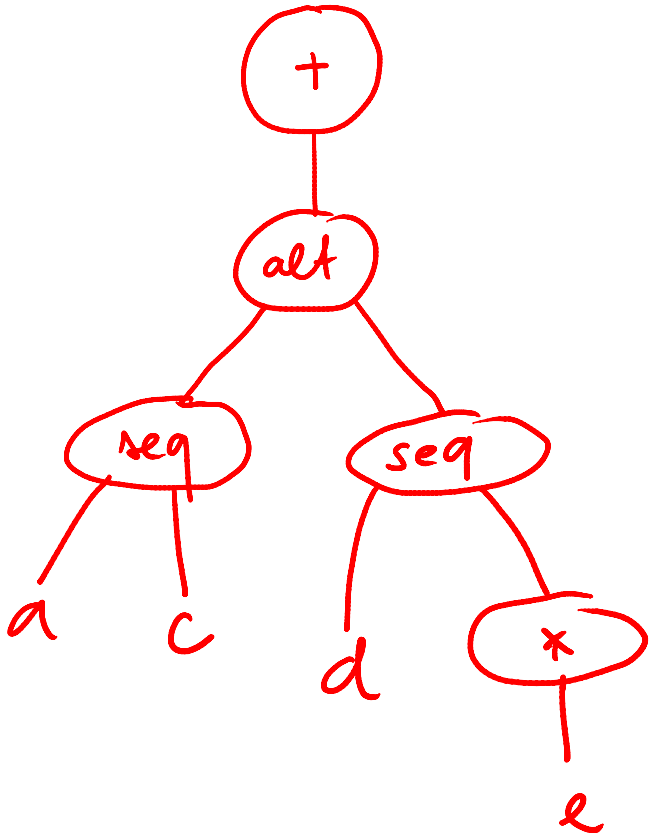
Compiling r.e. to NFA

How would you proceed?

- build AST of RE
- recursively rewrite AST into an automaton
- 

Example of abstract syntax tree (AST)

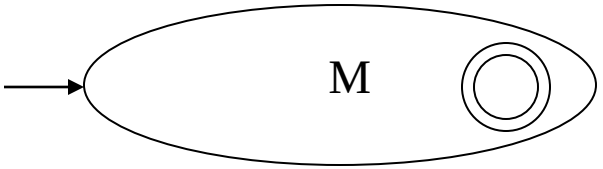
(ac|de*)+



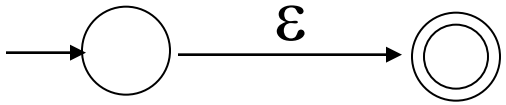
Regular Expressions to NFA (1)

For each kind of rexp, define an NFA

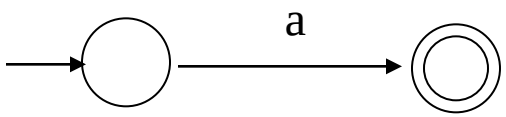
- Notation: NFA for rexp M



- For ε:

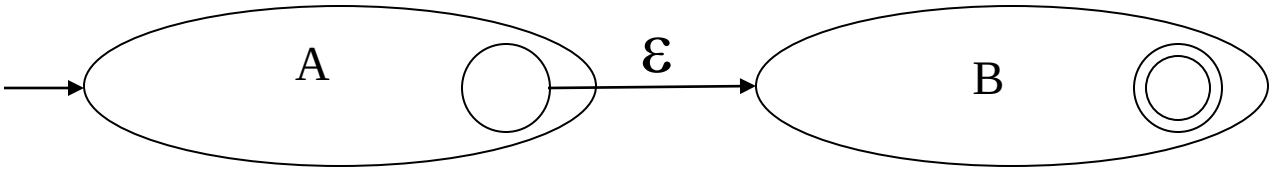


- For literal character a:

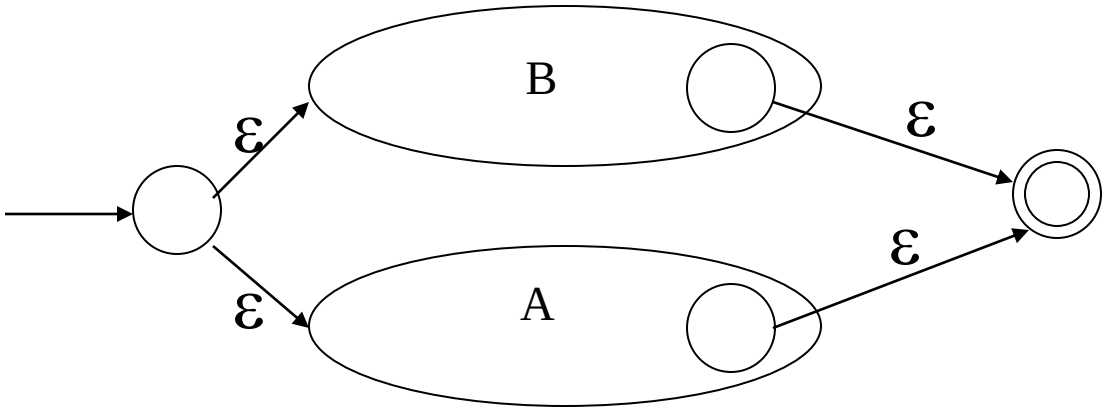


Regular Expressions to NFA (2)

For A B

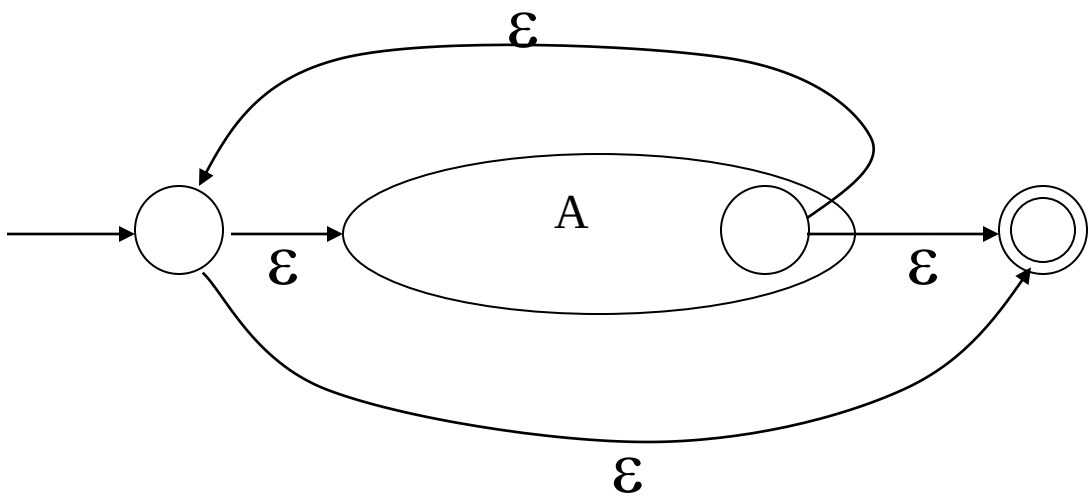


For A | B



Regular Expressions to NFA (3)

For A*

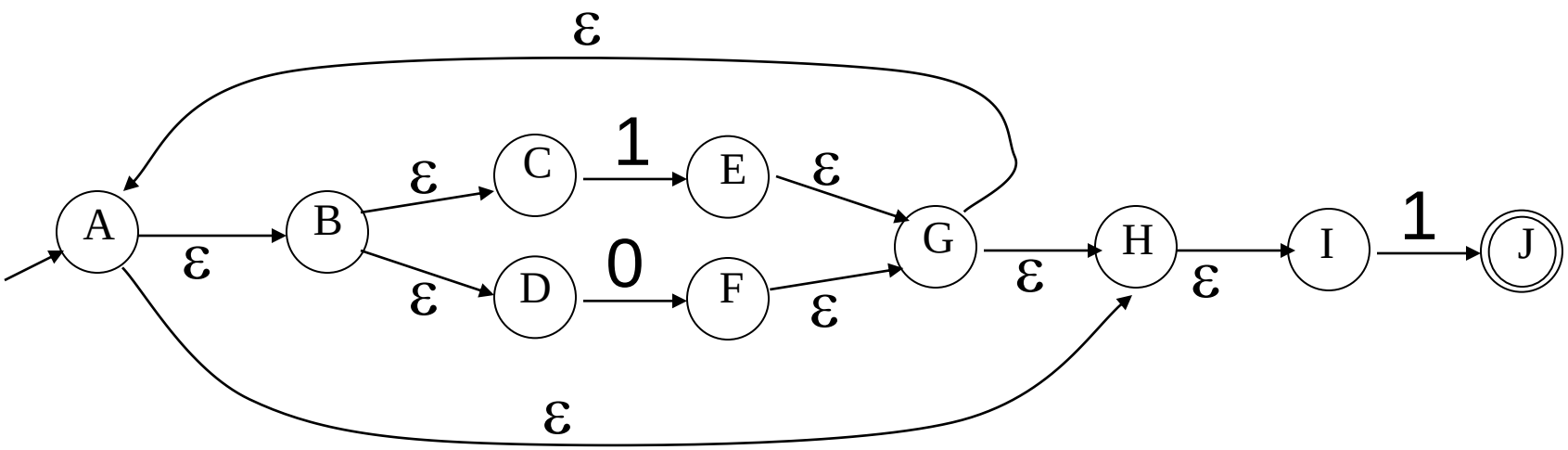


Example of RegExp -> NFA conversion

Consider the regular expression

(1|0)*1

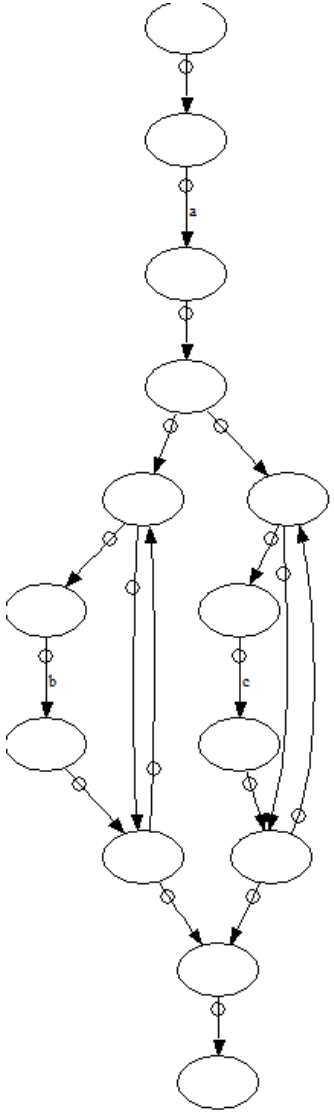
The NFA is



SDT that visualizes RE-to-NFA translation

SDT can translate (1|0)*1 not only to RE but also to a dotty file that visualizes this RE.
Dotty file:

```
digraph G {f7 -> f8 [label="a"]
f7 [label=""]
f8 [label=""]f9 -> f10 [label="b"]
f9 [label=""]
f10 [label=""]
f11 -> f9 [label=""]
f10 -> f12 [label=""]
f11 -> f12 [label=""]
f12 -> f11 [label=""]
f11 [label=""]
f12 [label=""]f13 -> f14 [label="c"]
f13 [label=""]
f14 [label=""]
f15 -> f13 [label=""]
f14 -> f16 [label=""]
f15 -> f16 [label=""]
f16 -> f15 [label=""]
f15 [label=""]
f16 [label=""]
f17 -> f11 [label=""]
f17 -> f15 [label=""]
f12 -> f18 [label=""]
f16 -> f18 [label=""]
f17 [label=""]
f18 [label=""]
f19 -> f7 [label=""]
f8 -> f17 [label=""]
f18 -> f20 [label=""]
f19 [label=""]
f20 [label=""]}
```



What strings can we tell apart with RE?

Exercise

Write a RE or automaton that accepts a string over the alphabet { (,) } iff the string has balanced parentheses:

(()(())) balanced

(()((())) not balanced

Can't be done. We need to count open left parens.

Since the input can be arbitrarily large, we need a counter or a set of states that is unbounded in size.

Sadly, finite automata have only finite number of states.

Expressiveness of RE 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 strings that have some property

- e.g., can be written as ('1'*k)*m, for some k>1, m>1
- or contain only balanced parentheses: (((())(())))

Primality testing revisited

Why can't RE test if a string matches ('1'*k)*m, k>1,m>1?

It may seems that

('1'*k)*m, k>1,m>1

is equivalent to

(11+)(11+)+

Exercise: Find a string tha matches the latter but does not match the former.

To be continued in Lecture 11

We will uncover surprising semantic differences between regexes and REs.

Concepts

- SDT converts RE to NFA: an interesting compiler
- recognizers: tell strings apart
- NFA, DFA, regular expressions = equally powerful
- but \1 (backreference) makes regexes more pwrful
- Syntax sugar: e+ desugars to e.e*
- Compositionality: be weary of greedy semantics
  - this will be covered in L11
- Metacharacters: characters with special meaning

Summary of DFA, NFA, REs

What you need to understand and remember

- what is DFA, NFA, regular expression
- the three have equal expressive power
- what is meant by 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