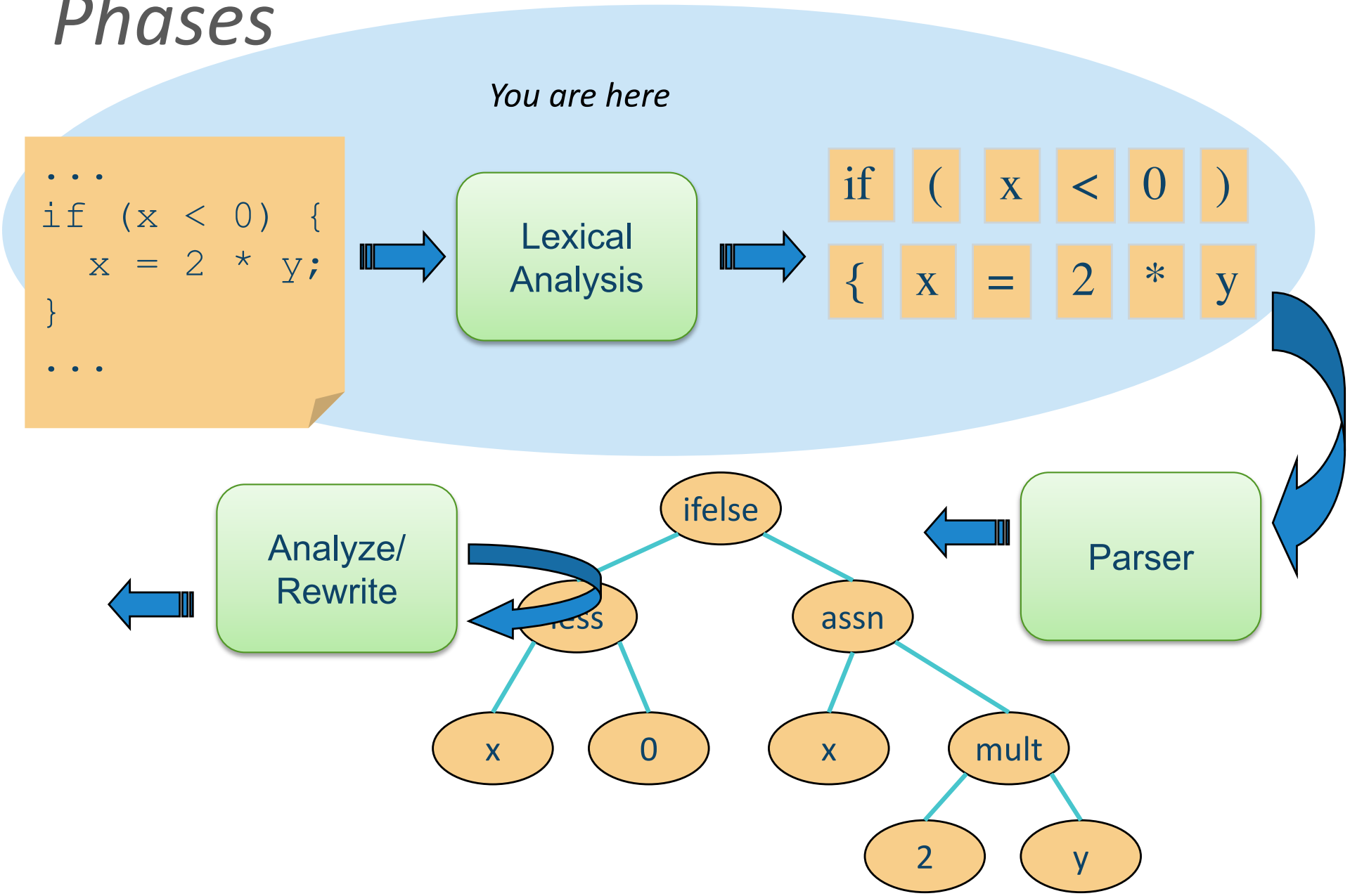
Lexical Analysis

Regular expressions & finite automata

Lexical analysis tools

Phases


You are here
...
if (x < 0) {
x = 2 * y;
}
...
Lexical Analysis
if (x < 0)
{ x = 2 * y
Parser
ifelse
assn
x
0
x
mult
2
y
Analyze/ Rewrite

Lexical Structure

Syntax customarily divided into two levels

- Lexical structure: identifiers, numbers, keywords, etc. (“tokens” or “lexemes”)
  - Defined using regular languages
- Phrase structure (“grammar”): procedures, statements, expressions, etc.
  - Defined using context-free languages

Why?

- CFGs are more powerful; why not define everything at the same level in the CFG?

Why Separate Lexical Processing?

Things you don't want to (or can't) say in a grammar

- Comments can appear anywhere spaces are allowed; they can contain anything but "*/"
- **foo** is an identifier, but **for** is a keyword
- Among matching patterns, take the longest possible

Separating lexical analysis simplifies the grammar and avoids nasty constructs

- While each individual token is described by a regular language, division into tokens is not

The Goal: Automate Lexical Analysis Generation

- We need
  - Suitable patterns for tokens
  - An algorithm for pattern matching
- Approach
  - Regular expressions are (nearly) ok
  - Regular expressions → efficient matchers in two steps
    - RE to NFA, NFA to DFA

Examples (simplified)

integer = [1-9][0-9]*

identifier = [a-z][a-z0-9_]*

WHILE = while

lexical analyzer generator (lex, flex, re-flex, jflex, etc.) turns patterns like these into a lexical analyzer

What it really looks like …

```
%{
...
%}
white = [ \t\f\n\r]
int = 0 | [1-9][0-9]*
id  = [$A-Za-z_][A-Za-z_0-9]*
%%
<YYINITIAL> {
  {id}          { return symbol(sym.SYMBOL, yytext()); }
  {white}         { /* skip */ }
  "::="         { return symbol(sym.BECOMES); }
  ^"["          { yybegin(PRODNUM); }
}
...
```

(This example is JFlex. Flex/RE-flex looks a little different.)

Regular Expressions X denote Regular Languages L(X)

Let Σ = { a, b, c, ... } be an alphabet, then

L (“x”) = { “x” }

L (ε) = { “” }

L (X|Y) = L(X) $\cup$ L(Y)

L (X Y) = { xy | x in L(X), y in L(Y) }

L (X*) = { “” } $\cup$ L(X) $\cup$ L(XX) $\cup$ L(XXX) ...

Shorthands:

X? = X|ε , X^{+}=XX*, [a-z] = a|b|...|z,

The Goal: A DFA

A state machine that consumes one character on each move and announces a pattern when it is matched

- Recognizes tokens in linear time*

  independent of number of patterns!
- With a nice interface: "Get me the next token"

But we don't get there all in one step

*almost ... actually we have to cheat a little

Two Steps

From regular expression to non-deterministic finite-state acceptor

"Thompson's construction"

Simple, but not quite what we want

From non-deterministic machine (NFA) to deterministic machine (DFA)

"Subset construction"

Ah, that's more like it

Non-deterministic finite-state acceptor

Finite set of states; one is the "start state", some are "accepting"

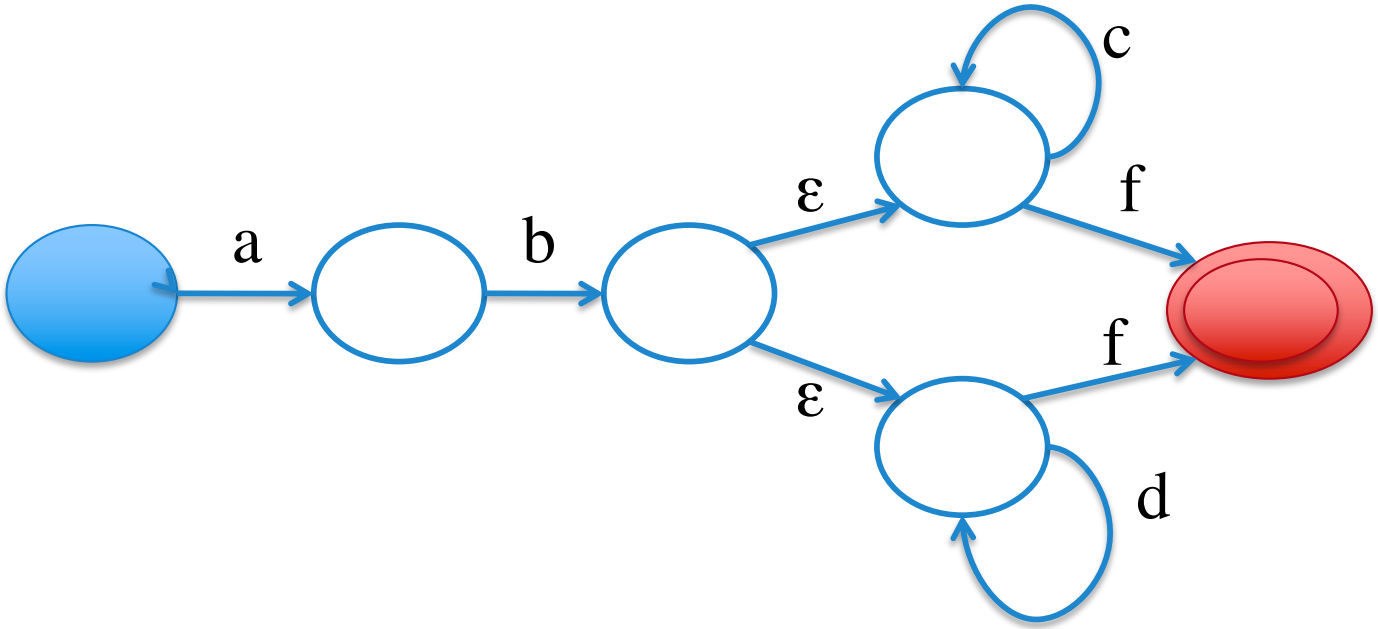
» And no other memory — that's the "finite state" part

Transition relation

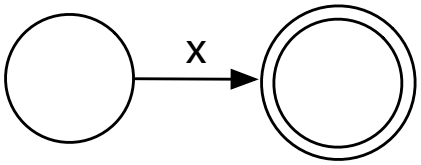
- Transitions labeled by symbols
- Also transitions labeled by ε
  - Can take without consuming a symbol

Accepts a "word" (string) if its symbols correspond to a path from start state to any accepting state.

Example: ab(c*|d*)f

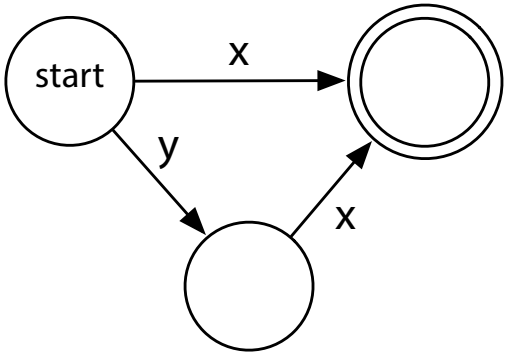


Finite state automata ("state machines")

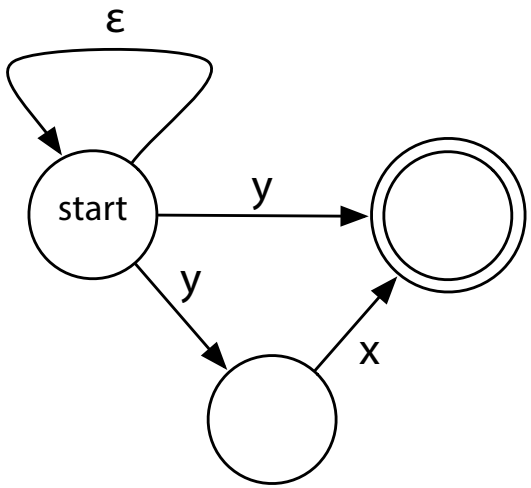


state

accepting state

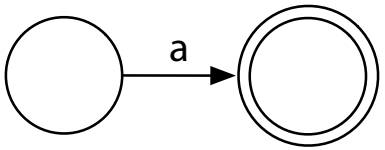
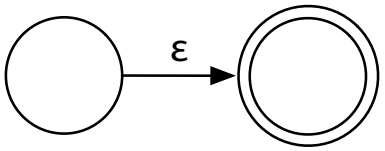


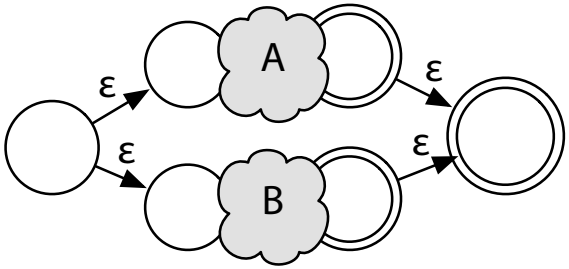
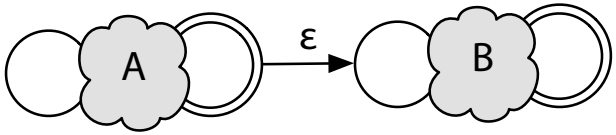
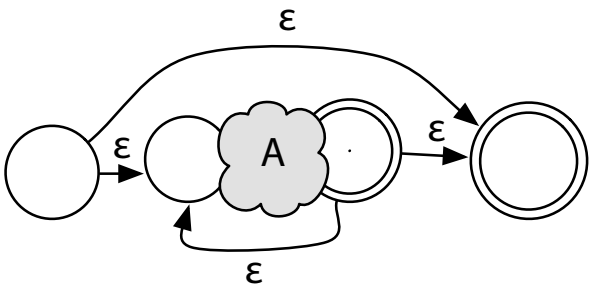
A deterministic FSA (DFA)



A non-deterministic FSA (NFA)

Thompson's construction

expression	language	automaton
a	{ a }	
ε	{''}	

A | B	L(A) ∪ L(A)	
A B	{ xy | x ∈L(A), y ∈L(B) }	
A *	{'', L(A), L(AA), L(AAA), … }	

Thompson's construction

A typical "syntax-directed" construction

Each grammar rule of regular expression corresponds to construction or composition of machines

(switch to board at this point)

Do these on the board …

Identifier (in Java, or in Quack)

Floating point constant

(ab)*(c*|d*)f

(save for subset construction)

Matching an NFA

Angelic non-determinism:

- Try all possible paths
- If any of them end in a final state, accept

Example on board: Integer constant, floating point constant, identifier

What if we want to do this in linear time (proportional to the length of input)?

Subset Construction

Three variations in compiler construction:

- Subset construction from NFA to DFA
- LR state construction in parsing
- Bottom-up tree matching for instruction selection

Fundamental space-for-time tradeoff

- If there is only a finite number of states, we can lift the transition relation between states to a function from sets of states to sets of states by pre-computing all the possible combinations

(switch to board; work examples)

Lex and its clones (Jflex, Flex, RE-flex…)

- Input: some declarations, then regular expressions (mostly one per token) and actions (C or Java code)
- Output: A program in the Java, or C, or …
  - Really it is mostly boilerplate code, plus the action parts of rules, and a big table. The automaton (state machine) is encoded in a table
- Automaton is built to match one big regular expression: $pattern_1 \mid pattern_2 \mid \ldots \mid pattern_n$
  - How can it choose the right action to execute?

Consider …

Let's build DFA for two rules:

[0-9]+ { return INT; }

[0-9]*.[0-9]+. { return FLOAT; }

We want one combined DFA, with "maximum munch" rule, and we want to use the right action.

Limitations of Regular Languages

CIS (4|5)61

October 2018

Why not regular expressions for everything?

We will switch from regular expressions to context-free grammars for the syntactic structure of Quack.

Must we? Could we use regular languages for everything?

Yes, we must.

Regular languages are *provably* inadequate for patterns that we need in a full programming language.

Languages and recognizers

In the terminology of automata theory …

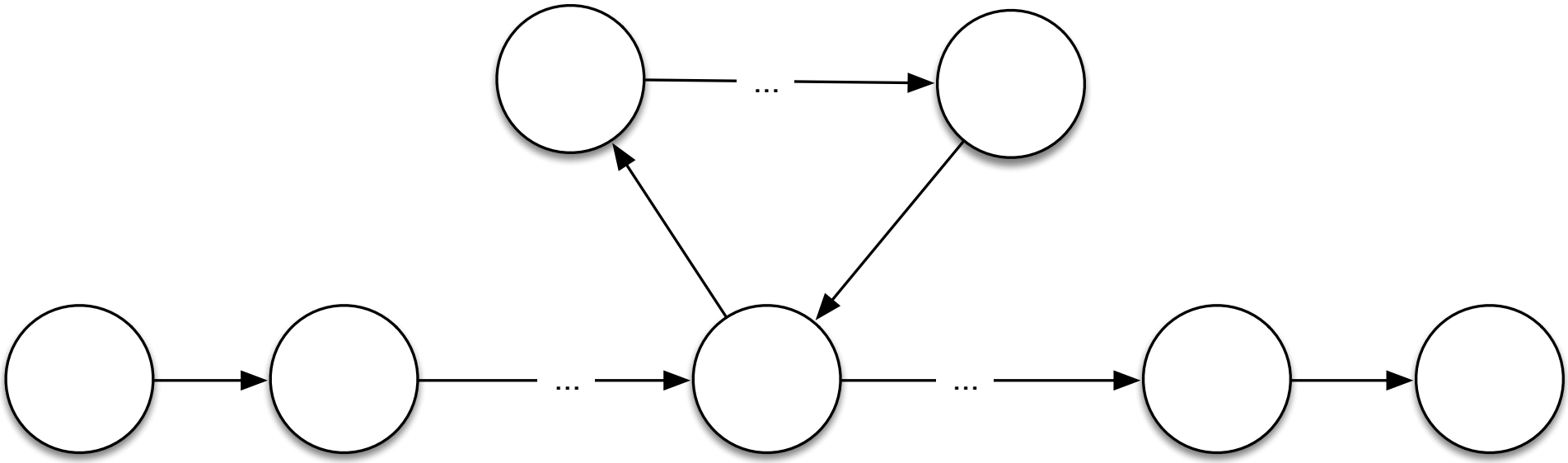
A language is a set of strings.

A regular language is denoted by a regular expression.

And we know …

The regular languages are exactly the languages that can be recognized by deterministic finite-state acceptors.

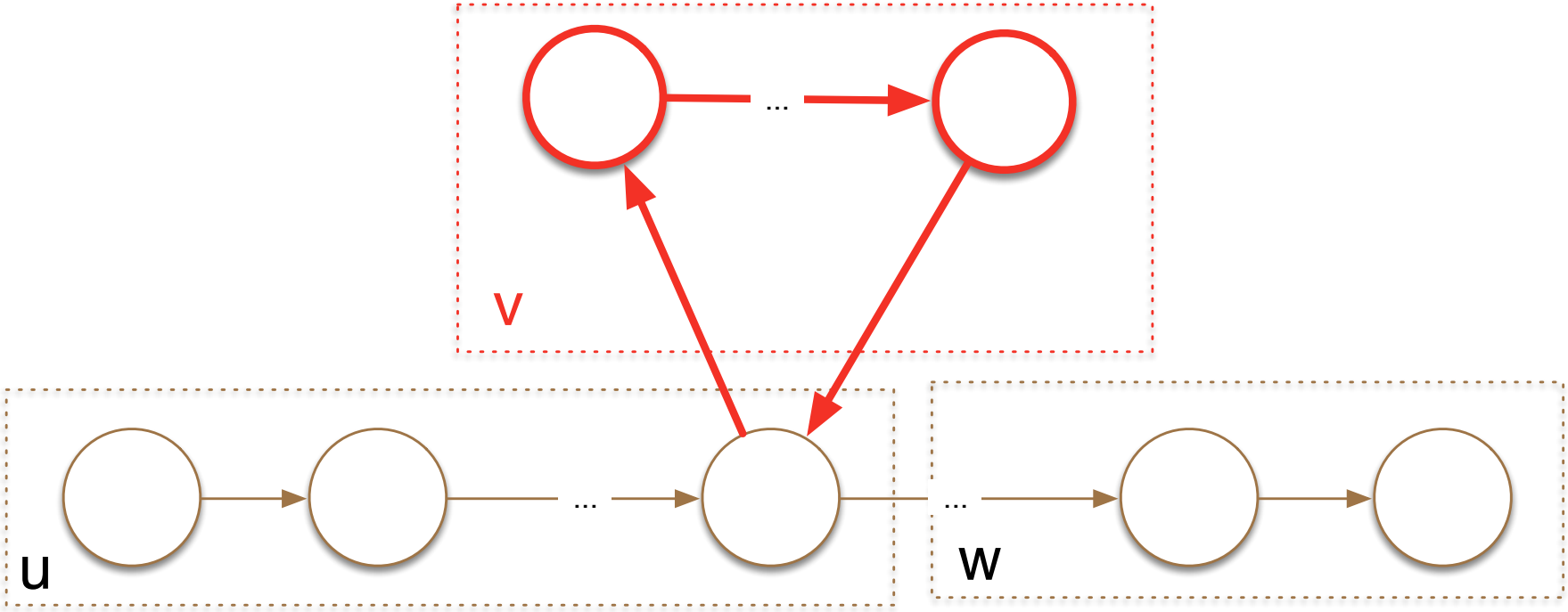
Long strings require DFA loops



The DFA has a finite number n of states. To recognize a string with more than k symbols, it must repeat a state.

Pumping lemma

A very long string z (|z| ≥ n) must repeat a state.
Divide it into u, v, w, with v the first loop



If uvw is in L, so are uw, uvvw, uvvvw, etc.

Pumping lemma: formal statement

Let L be a regular language.

There exists a fixed n such that
if $z \in L$ and $|z| \geq n$
then $z = uvw$, $|uv| \leq n$, $|v| \geq 1$,
and for all $i \geq 0$, $uv^i w \in L$

Consequences of the pumping lemma

$a^i b^i$ is not regular — no DFA can recognize it.

Because: You can form a long enough string to pump just 'a', wrecking the balance

Balanced and hierarchical structures in general are not regular

- Balanced parenthesis
- Properly nested begin/end pairs if / else pairs

…

Programming language grammars are not regular

1 2 next

Answer to: RegEx match open tags except XHTML self-contained tags

4426

You can't parse [X]HTML with regex. Because HTML can't be parsed by regex. Regex is not a tool that can be used to correctly parse HTML. As I have answered in HTML-and-regex questions here so many times before, the use of regex will not allow you to consume HTML. Regular expressions are a tool that is insufficiently sophisticated to understand the constructs employed by HTML. HTML is not a regular language and hence cannot be parsed by regular expressions. Regex queries are not equipped to break down HTML into its meaningful parts. so many times but it is not getting to me. Even enhanced irregular regular expressions as used by Perl are not up to the task of parsing HTML. You will never make me crack. HTML is a language of sufficient complexity that it cannot be parsed by regular expressions. Even Jon Skeet cannot parse HTML using regular expressions. Every time you attempt to parse HTML with regular expressions, the unholy child weeps the blood of virgins, and Russian hackers pwn your webapp. Parsing HTML with regex summons tainted souls into the realm of the living. HTML and regex go together like love, marriage, and ritual infanticide. The cannot hold it is too late. The force of regex and HTML together in the same conceptual space will destroy your mind like so much watery putty. If you parse HTML with regex you are giving in to Them and their blasphemous ways which doom us all to inhuman toil for the One whose Name cannot be expressed in the Basic Multilingual Plane, he comes. HTML-plus-regexp will liquify the nerves of the sentient whilst you observe, your psyche withering in the onslaught of horror. Regex-based HTML parsers are the cancer that is killing StackOverflow *it is too late it is too late we cannot be saved* the trangession of a child ensures regex will consume all living tissue (except for HTML which it cannot, as previously prophesied) *dear lord help us how can anyone survive this scourge* using regex to parse HTML has doomed humanity to an eternity of dread torture and security holes *using regex* as a tool to process HTML establishes a breach *between this world* and the dread realm of corrupt entities (like SGML entities, but *more corrupt) a mere glimpse* of the world of reg**ex parsers for HTML will ins**tantly transport a p*rogrammer's consciousness i*nto a wo*rl*d of ceaseless screaming, he comes~~, the pestilent slithy~~ regex-infection will **devour your HT**ML parser, application and existence for all time like Visual Basic only worse *he comes he comes do not fight* h**e comes, his** unholy radiańcé de*stroying all enlightenment, HTML tags* ***leaking from your eyes like liq****uid p*ain, the song of regular expres~~sion parsing~~ will ext*inguish the voices of mortal man from the sphere I can see it can you see it it is beautiful* the `final snuf` fing o*f the lies of Man ALL IS LOST* **ALL IS** LOST the *pony he comes* he comes~~he comes the ichor permeates all~~ MY FACE *MY FACE oh god no* **NO NOOOO** NO stop *the angles* `are` **not real ZALGO IS TONY THE PONY HE COMES**

Have you tried using an XML parser instead?

Intuition and rules of thumb

Regular expressions can't count

(although your attached actions can:

consider allowing nested comments /* /* ... */ */)

Anything hierarchical (matched braces, nested control structures, …) probably isn't regular

Context free grammars and languages are the natural way to express *nested* structures.