Computer Mathematics

Week 14
Computational grammars and languages

KUAS 京都先端科学大学
KYOTO UNIVERSITY of ADVANCED SCIENCE
Department of Mechanical and Electrical System Engineering

last week

non-deterministic machines

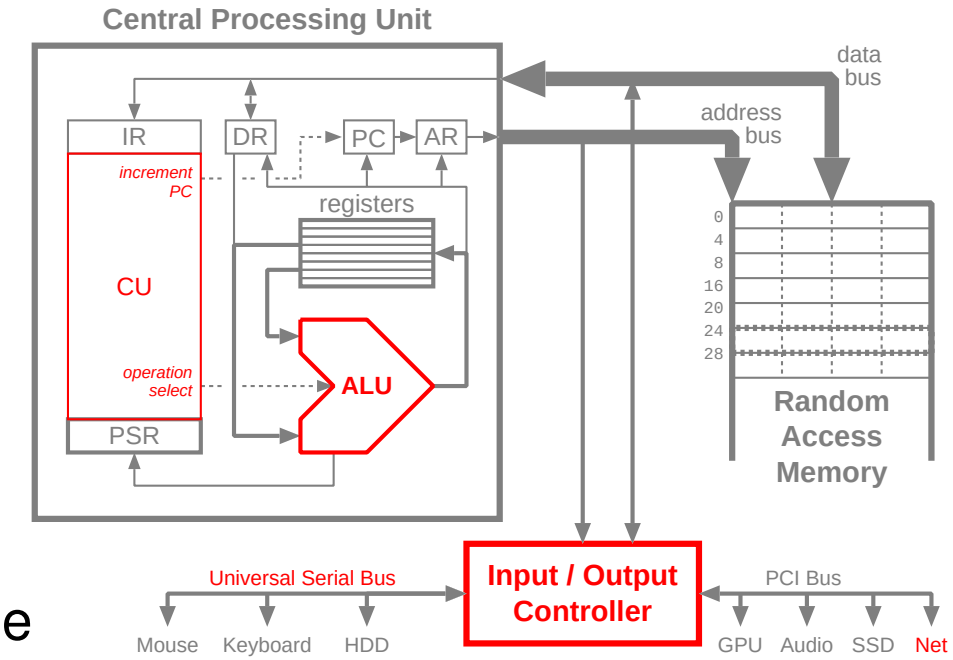
- how to simulate them

eliminating non-determinism

- make an equivalent deterministic machine
- using a clue from the simulation
  - and some real mathematics

deterministic machines

- advantages and disadvantages
- relative performance



this week

origins of computational grammars

grammars describe languages

FSMs describe languages

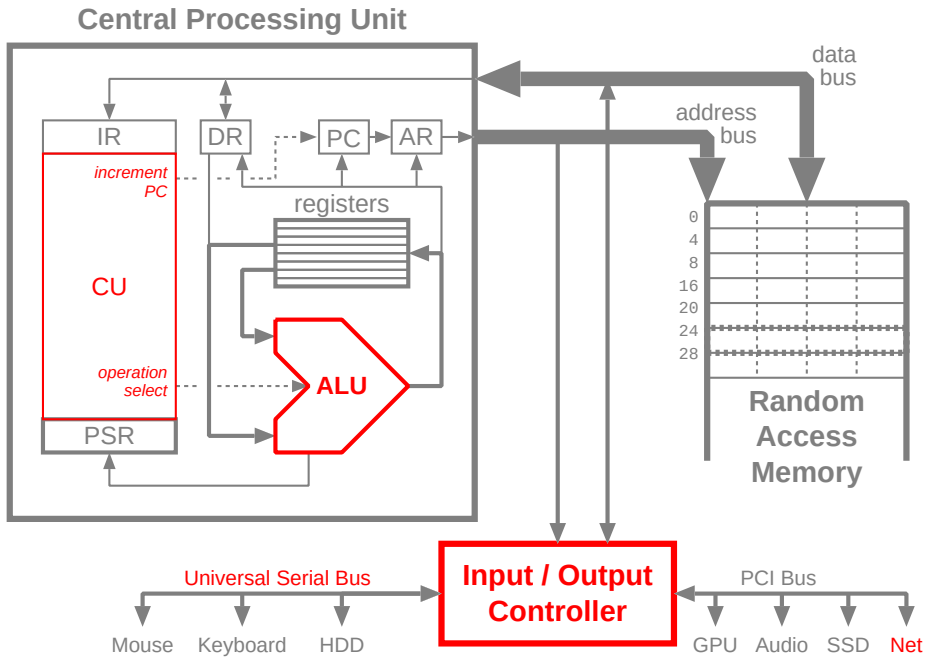
grammars corresponding to FSMs

the hierarchy of grammar types

regular grammars

context-free grammars

- and their importance to computing



origins of computational grammars

(computational) grammars invented by IBM (remember them?) in the late 1950s

- to solve the problem of translating arithmetic expressions into machine code
- for the FORTRAN (FORmula TRANslation) language

how would you describe the structure of

- a very simple (but complete) English sentence? or...
- the (mathematical) addition of two variables?

grammar rules describe patterns of symbols

English sentences (school version)...

$$<\textit{sentence}> \rightarrow <\textit{subject}><\textit{verb}><\textit{object}>$$

"a *sentence* is made from a *subject*, a *verb*, and an *object*"

- in that order

and the addition...

$$<\textit{sum}> \rightarrow <\textit{variable}> + <\textit{variable}>$$

"an *addition* is made from a *variable*, a '+' symbol, and a *variable*"

- in that order

terminal and non-terminal symbols in BNF

this notation for languages is called of Backus-Naur Form (BNF)

- John Backus was the inventor of FORTRAN at IBM
- a year later, Peter Naur adopted BNF to describe Algol-60

the items in <*angle brackets*> are *non-terminal* symbols

- they represent 'something else', defined as a BNF *rule*
- they do not appear literally in any sentence of the language
- they are still being replaced, as we work towards a final sentence

the items in `typewriter font` are *terminal* symbols

- they appear literally in written sentences
- they cannot be replaced by anything else

grammars describe languages as sets of possible sentences

so, to make a sentence according to a grammar

- replace non-terminal symbols with their right-hand side
- repeat until only terminal symbols remain

you can take this to any level of detail you want

English sentences (linguist version)...

$$\begin{aligned} \langle \textit{sentence} \rangle &\rightarrow \langle \textit{noun-phrase} \rangle \langle \textit{verb-phrase} \rangle \\ \langle \textit{noun-phrase} \rangle &\rightarrow \langle \textit{adjectives} \rangle \langle \textit{noun} \rangle \\ \langle \textit{verb-phrase} \rangle &\rightarrow \langle \textit{verb} \rangle \langle \textit{noun-phrase} \rangle \end{aligned}$$

from which we might obtain, "`hungry Jane eats tasty cheese`"

non-terminals can have more than one replacement

and the addition needs variable names...

$$\langle \textit{sum} \rangle \rightarrow \langle \textit{variable} \rangle + \langle \textit{variable} \rangle$$
$$\langle \textit{variable} \rangle \rightarrow \texttt{a} \mid \texttt{b} \mid \texttt{c} \mid \ldots \mid \texttt{x} \mid \texttt{y} \mid \texttt{z}$$

from which, "`x + y`"

the | character separates *alternatives* in BNF

- if you need a '|' literally in your BNF, write quotation marks around it
  (BNF is very pragmatic, and not too formal — just add whatever you need to it, e.g., it should be obvious what the '...' implies above)

FSMs also describe languages as sets of possible sentences

recall finite state machines:

- practical solution to many problems
- device for understanding the theory of computation
  - intimately related to grammars and formal languages

path from start state to final state generates a sequence of symbols

- corresponding to the labels on the transitions
- the same sequence is recognised (accepted) by the machine

sequences can be simple or complex

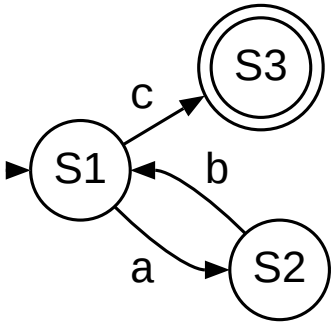
- simplicity/complexity of sequence directly relates to
- simplicity/complexity of the corresponding FSM

FSMs also describe languages as sets of possible sentences

the ability of a FSM to recognise a sequence tells us

- about the complexity of the sequence
- about the complexity of the language to which it belongs

consider the first FSM we looked at

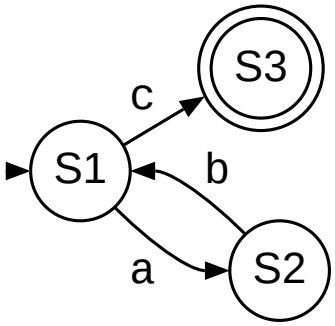


moving from state 1 to state 3

- “c”, “abc”, “ababc”, etc...

represented as a regular expression: `(ab)*c`

a grammar can describe any FSM



the same machine can be represented as a computational grammar

$<start>$	$\rightarrow$	$<S1>$	(start in $<S1>$)
$<S1>$	$\rightarrow$	a $<S2>$	(from $<S1>$, transition labelled "a" leads to $<S2>$)
	$\rightarrow$	c	(from $<S1>$, transition labelled "c" leads to final state)
$<S2>$	$\rightarrow$	b $<S1>$	(from $<S2>$, transition labelled "b" leads to $<S1>$)

the $<start>$ state is made *explicit*, and the final state is *implicit*

- you have reached the final state when there are no more non-terminals to replace

repetition is accomplished in grammars by recursion

notice how we made the repetition, corresponding to `(ab)*`

$$\langle S1 \rangle \rightarrow \texttt{a}\ \langle S2 \rangle$$
$$\langle S2 \rangle \rightarrow \texttt{b}\ \langle S1 \rangle$$

$\langle S1 \rangle$ and $\langle S2 \rangle$ are *mutually-recursive*

- $\langle S1 \rangle$ leads to $\langle S2 \rangle$ (after an "`a`"), and
- $\langle S2 \rangle$ leads straight back to $\langle S1 \rangle$ (after a "`b`")

the recursion would be *infinite*, except that the alternative route from $\langle S1 \rangle$

$$\langle S1 \rangle \rightarrow \texttt{c}$$

gives us a way to escape from it

repetition is accomplished in grammars by recursion

the recursion above is *indirect*, but it can also be *direct*

if we want to make our variable names longer than 1 letter, then this...

$$\begin{aligned} \langle \textit{sum} \rangle &\rightarrow \langle \textit{variable} \rangle + \langle \textit{variable} \rangle \\ \langle \textit{variable} \rangle &\rightarrow \langle \textit{letter} \rangle \mid \langle \textit{letter} \rangle \langle \textit{variable} \rangle \\ \langle \textit{letter} \rangle &\rightarrow \text{a} \mid \text{b} \mid \text{c} \mid \ldots \mid \text{x} \mid \text{y} \mid \text{z} \end{aligned}$$

...does the trick

for example, to make the variable name "`var`"

$$\begin{aligned} \langle \textit{variable} \rangle &\rightarrow \langle \textit{letter} \rangle \langle \textit{variable} \rangle && (\langle \textit{letter} \rangle \rightarrow \text{“v”}) \\ &\rightarrow \texttt{v} \ \langle \textit{variable} \rangle && (\langle \textit{variable} \rangle \rightarrow \langle \textit{letter} \rangle \langle \textit{variable} \rangle) \\ &\rightarrow \texttt{v} \ \langle \textit{letter} \rangle \langle \textit{variable} \rangle && (\langle \textit{letter} \rangle \rightarrow \text{“a”}) \\ &\rightarrow \texttt{v a} \ \langle \textit{variable} \rangle && (\langle \textit{variable} \rangle \rightarrow \langle \textit{letter} \rangle) \\ &\rightarrow \texttt{v a} \ \langle \textit{letter} \rangle && (\langle \textit{letter} \rangle \rightarrow \text{“r”}) \\ &\rightarrow \texttt{v a r} \end{aligned}$$

repetition is accomplished in grammars by recursion

Q: why do the rules for $<variable>$ look like this?

$$<variable> \rightarrow <letter><variable>$$
$$<variable> \rightarrow <letter> \qquad \leftarrow \text{why is this here?}$$

the relative power of FSMs and grammars in general

so, just how 'computationally powerful' are FSMs?

notice the form of the grammar rules for a FSM

$<P>$	$\rightarrow$ $<Q>$	(an ϵ-transition $P \rightarrow Q$)
$<P>$	$\rightarrow$ x $<Q>$	(a labelled transition $P \xrightarrow{\mathtt{x}} Q$)
$<P>$	$\rightarrow$ y	(a labelled transition $P \xrightarrow{\mathtt{y}}$ final state)
$<P>$	$\rightarrow$	(an ϵ-transition $P \rightarrow$ final state)

this is exactly what we need to describe any NFA

- converted to a DFA, we don't even need the ϵ-transitions

we can now start to ask profound questions about FSMs and languages:

- can we build a FSM to describe *any* kind of sequence?
- which kind(s) of grammar correspond(s) to a FSM?
- are FSMs powerful enough to describe languages generated by *any* grammar?

the relative power of FSMs and grammars in general

a FSM is clearly powerful enough to describe

- sequences, repetition, and alternation
- i.e., anything a regular expression can describe

let α and β represent non-empty sequences of two given symbols, then

- sequences of the form $\alpha\beta\alpha$ are *palindromes*

the simplest form of palindrome, with two symbols "a" and "b", has the form

$$\mathtt{a}^n\,\mathtt{b}\,\mathtt{a}^n$$

(where $\mathtt{a}^n$ means 'a repeated n times')

Q: can a FSM describe palindromes?

palindromes are pervasive in computing

why is this important?

- most of computing is made from palindromes

IBM's original problem of translating arithmetic expressions

- expression: $2 \times (3 + 4) \div 5$
- expression: $2 \times ($ *what can I put here?* $) \div 5$

many examples of 'nested' (or 'recursive') constructions

- parenthesised sub-expressions: left '(' must match right ')'
- compound statements in programming: left '{' must match right '}'
- multi-dimensional arrays: left '[' must match right ']'
- SGML, HTML, XML, etc: opening '<tag>' must match closing '</tag>'
- etc...

FSMs can only describe a restricted type of grammar

a FSM is *not* sufficient to recognise *any* of the above

- because a FSM cannot recognise palindromes
- FSMs cannot *count* beyond a fixed limit (they remember a finite, bounded past)
- you can make a FSM that recognises "b", "aba", and "aabaa", but
  - to recognise "aaabaaa" you need a *bigger* FSM
  - you cannot build a FSM that recognises *all* palindromes
    - not even when you know that the palindromes all look like: $\mathtt{a}^n\,\mathtt{b}\,\mathtt{a}^n$

many other kinds of sequence cannot be recognised by a FSM

- but they *can* be specified by a grammar

FSMs therefore correspond to a *restricted* kind of grammar

- the type of sequence they recognise is called a *regular sequence*
- corresponding exactly to those described by a *regular expression*

regular grammars have restricted rules corresponding to FSMs

describe regular sequences

correspond to regular expressions, and FSMs

contain rules that have

- a single non-terminal on the left of the $\rightarrow$
- at most one terminal and one non-terminal on the right side of the $\rightarrow$

$<P>$	$\rightarrow$	$<Q>$	(an ϵ-transition $P \rightarrow Q$)
$<P>$	$\rightarrow$	x $<Q>$	(a labelled transition $P \xrightarrow{\mathrm{x}} Q$)
$<P>$	$\rightarrow$	y	(a labelled transition $P \xrightarrow{\mathrm{y}}$ final state)
$<P>$	$\rightarrow$		(an ϵ-transition $P \rightarrow$ final state)

where the order of the terminal and non-terminal *must be consistent* in all rules

- at least within the same grammar

grammar types form a hierarchy

regular grammars are also called '*type 3*' grammars

- which implies there are other types

grammars form a *hierarchy* according to how powerful they are

- numbered from 3 down to 0
- each grammar is more powerful than (and includes) the one before it
- more powerful grammars can describe more complex languages

this hierarchy was first identified by Noam Chomsky

- look him up if you don't already know who he is
- even now, work continues to understand grammars and *parsing* better

  (parsing means finding how a given sequence can be generated by a grammar)

adding memory to a FSM makes it more powerful

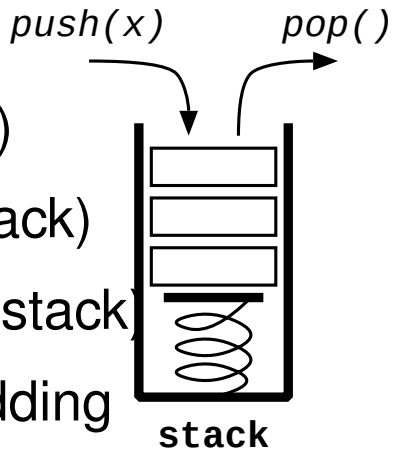
what is the machine one step more powerful than a finite state machine?

a ‘push-down machine’ (or ‘push-down automaton’)

- a FSM plus a *stack*

  (stacks used to be called ‘push-down lists’, hence the name of the machine)

a stack?



- behaves like a stack of papers in a tray (or plates in a dispenser)
- new items are added to the top of the stack (*pushed* onto the stack)
- old items are removed from the top of the stack (*popped* off the stack)
- the order of removing is therefore the opposite of the order of adding
  - the *last* thing you put *in* will be the *first* thing that comes *out*
  - stacks are an example of a last-in first-out (*LIFO*) data structure

adding memory to a FSM makes it more powerful

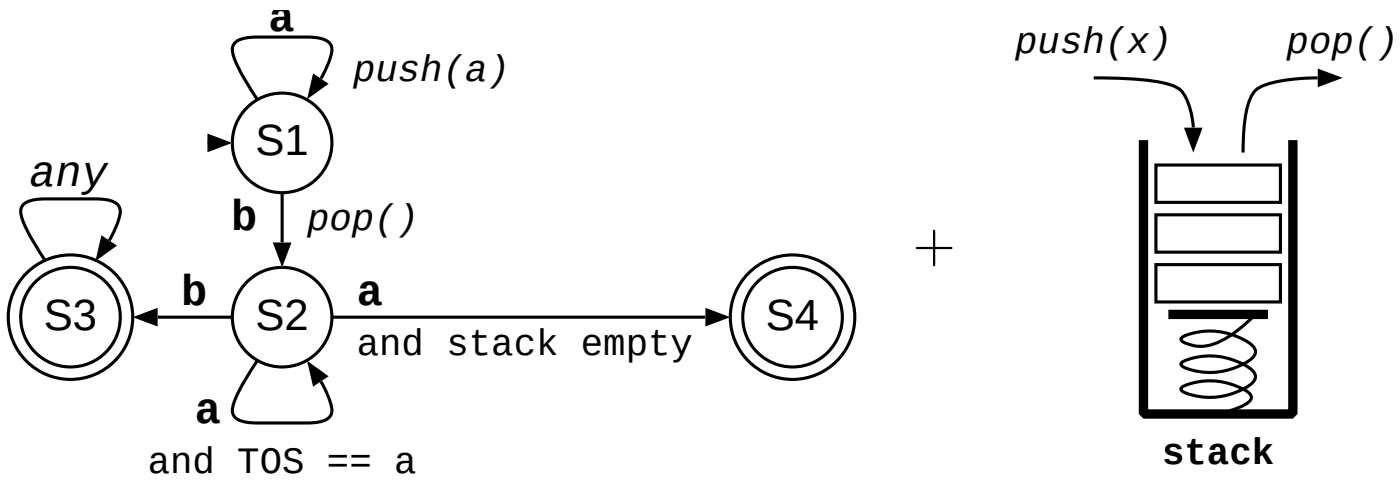
what is the machine one step more powerful than a finite state machine?

a ‘push-down machine’ (or ‘push-down automaton’)

- a FSM plus a stack

transitions can depend on an input symbol and/or the top item on the stack (TOS)

when making a transition, an item can optionally be pushed or popped



a palindrome detector that recognises: $\mathtt{a}^n\,\mathtt{b}\,\mathtt{a}^n$

reaching $<S4>$ indicates success, reaching $<S3>$ means the input was not a palindrome

FSM with memory can recognise nested structures

we have added memory to the FSM

even though it is finite memory, it is unbounded

- there is no limit on the depth of the stack
- e.g., it is not artificially limited to the length of the input sequence, etc.
- it can grow to deal with anything the input might contain

this machine can recognise all of the things the FSM could not

- parenthesised sub-expressions, compound statements
- multi-dimensional arrays, pairs of markup tags
- etc...

a push-down-machine is therefore more powerful than a FSM

it also corresponds to a more powerful type of grammar

- a *type 2*, or *context-free grammar* (CFG)

context-free grammars describe lots of import things

the rules of a CFG all have the form

$$<\textit{non-terminal}> \rightarrow \alpha$$

where α is *any* sequence of terminal and/or non-terminal symbols

CFGs are very important in computing

- almost all programming languages can be described by context-free grammars

FSM+stack is only one way to deal with CFGs

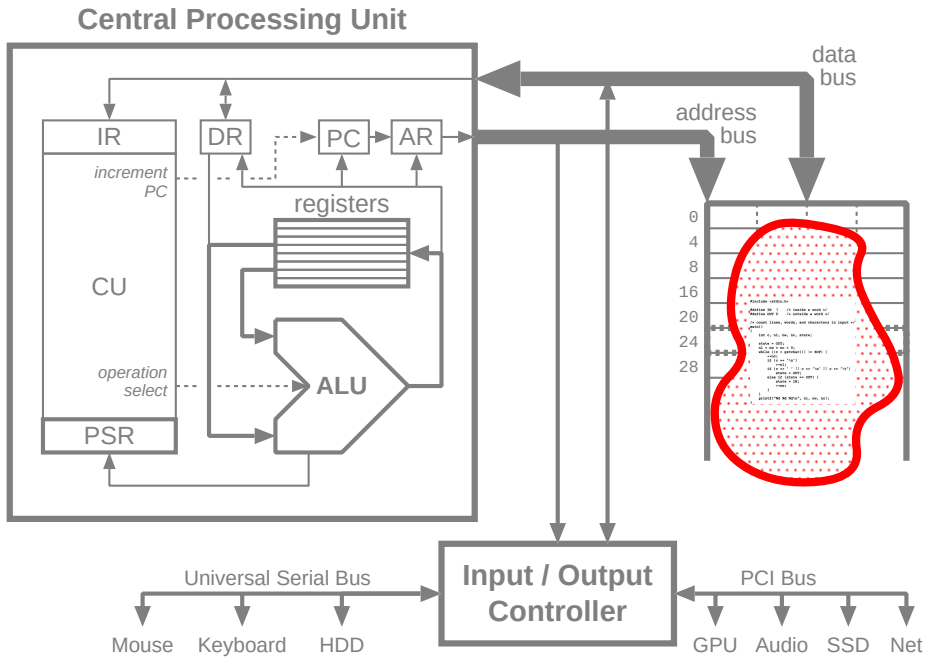
next week we will

- see some other ways of dealing with them
- discuss how to use them practically
- complete our hierarchy of machines and grammar types

next week

context-free grammars

- how they behave
- how to describe languages with them
- how to parse sequences with them



the remaining grammar types

- type 1 grammars
- type 0 grammars

and the machines that correspond to them

homework

reinforce your understanding

- implement a palindrome detector in Python
  - using a FSM plus a stack
- modify your detector to validate markup language
  - or at least whether opening/closing tags correspond
  - feed it a simple web page (HTML) and see what happens
  - how could you recover from errors, and carry on parsing?

ask about anything you do not understand

- from any of the classes so far this semester (or the lecture notes)
- it will be too late for you to try to catch up later!
- I am always happy to explain things differently and practice examples with you

glossary

context-free grammar — a grammar in which any rule for a non-terminal can always be used to replace that non-terminal. In other words, rules are always applicable regardless of the context in which they are being applied.

direct recursion — a function which calls itself, or a grammar rule in which the replacement for a non-terminal symbol includes that same non-terminal symbol.

hierarchy — a logical structure in which entities completely contain their sub-entities.

indirect recursion — recursion that occurs between two or more functions or grammar rules by the first invoking the second, the second invoking the next, and so on, until the first function or grammar rule is invoked again.

infinite recursion — recursion that never terminates.

LIFO — an acronym for last-in first-out. Any behaviour in which the order of outputs is reversed with respect to the order of inputs.

mutual recursion — two (or more) functions or grammar rules that invoke each other, leading to indirect recursion.

non-terminal symbol — a symbol in a grammar that represents another sequence of symbols (defined by one of the rules or alternatives for that non-terminal symbol). Non-terminal symbols never appear literally in a sentence or sequence of symbols permitted by their grammar.

palindrome — a symmetrical sequence of symbols that reads the same forwards and backwards.

parsing — the process of determining how a given sequence of symbols can be generated from a given grammar.

pop — remove the most recently added item from a stack.

push — add an item to a stack.

regular grammar — a grammar corresponding to a FSM, or regular expression, and which produces regular sequences.

rule — (grammar) a correspondence between a non-terminal symbol and a sequence of symbols that it stands for.

stack — a LIFO data structure.

terminal symbol — a symbol that can appear literally in a sentence or sequence of symbols permitted by its grammar.

type 2 grammar — another name for regular grammars, in Chomsky's classification of grammars.

type 3 grammar — another name for context-free grammars, in Chomsky's classification of grammars.