Backus-Naur Form (BNF)

Backus-Naur Form (**BNF**) is a notation technique used to describe **recursively** the syntax of

- programming languages
- document formats
- communication protocols
- etc.

$$
\begin{array}{lcl}
\langle\text{digit}\rangle & ::= & \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid \mathbf{3} \mid \mathbf{4} \mid \mathbf{5} \mid \mathbf{6} \mid \mathbf{7} \mid \mathbf{8} \mid \mathbf{9} \\
\langle\text{unsigned integer}\rangle & ::= & \langle\text{digit}\rangle \mid \langle\text{unsigned integer}\rangle\langle\text{digit}\rangle \\
\langle\text{integer}\rangle & ::= & \langle\text{unsigned integer}\rangle \mid +\langle\text{unsigned integer}\rangle \mid \\
 & & \qquad -\langle\text{unsigned integer}\rangle \\
\langle\text{letter}\rangle & ::= & \boldsymbol{a} \mid \boldsymbol{b} \mid \boldsymbol{c} \mid \dots \\
\langle\text{identifier}\rangle & ::= & \langle\text{letter}\rangle \mid \langle\text{identifier}\rangle\langle\text{letter}\rangle \mid \langle\text{identifier}\rangle\langle\text{digit}\rangle
\end{array}
$$

designed in the 1950–60s to define the syntax of the programming language ALGOL

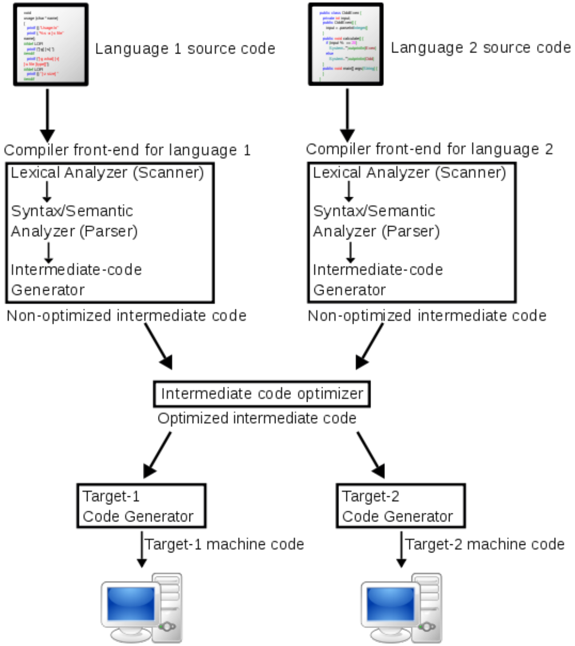
in fact, this is an example of a **context-free grammar**, Chomsky (1956)

Compilers

convert a high-level language into a **machine-executable** language

For example, $((3+4)*(6+7))$ →

LOAD 3 in register 1
LOAD 4 in register 2
ADD contents of register 2 into register 1
LOAD 6 in register 3
LOAD 7 in register 4
ADD contents of register 3 into register 4
MULTIPLY register 1 by register 4



Language 1 source code

Language 2 source code

Compiler front-end for language 1

Lexical Analyzer (Scanner)

Syntax/Semantic Analyzer (Parser)

Intermediate-code Generator

Non-optimized intermediate code

Compiler front-end for language 2

Lexical Analyzer (Scanner)

Syntax/Semantic Analyzer (Parser)

Intermediate-code Generator

Non-optimized intermediate code

Intermediate code optimizer

Optimized intermediate code

Target-1 Code Generator

Target-1 machine code

Target-2 Code Generator

Target-2 machine code

Defining languages recursively

Example 1. $L = \{a^n b^n \mid n \geq 0\}$

L = any word in the language

Basis: $\varepsilon \in L$ (the empty word is in L) $L \to \varepsilon$ **(r1)**

Induction: if w is a word in L, then so is awb $L \to aLb$ **(r2)**

BNF notation: $L ::= \varepsilon \mid aLb$

(r1), **(r2)** are understood as (substitution) **rules** (or **productions**) that **generate** all words in L

For example, the word $aabb$ is generated (or derived) as follows:

$L \Rightarrow aLb$	replace L with aLb by rule (r2)
$aLb \Rightarrow aaLbb$	replace L with aLb by rule (r2)
$aaLbb \Rightarrow aa\varepsilon bb$	replace L with ε by rule (r1)

Thus we obtain the **derivation** $L \Rightarrow aLb \Rightarrow aaLbb \Rightarrow aa\varepsilon bb = aabb$

all words are derived from r1 and r2

a word w can be derived using (r1) and (r2) if, and only if, $w \in L$

Palindromes

Example 2. Define the language P of **palindromes** over $\{0, 1\}$

(a palindrome is a string that reads the same forward and backward, e.g., `madamimadam'
or `Damn. I, Agassi, miss again. Mad')

Basis: $\varepsilon \in P$, $0 \in P$, $1 \in P$

Induction: if w is a word in P, then so is $0w0$ and $1w1$

$P \to \varepsilon$	(r1)
$P \to 0$	(r2)
$P \to 1$	(r3)
$P \to 0P0$	(r4)
$P \to 1P1$	(r5)

BNF notation: $P \ ::= \ \varepsilon \mid 0 \mid 1 \mid 0P0 \mid 1P1$

Construct a derivation of 01010

Exercise. Use the Pumping Lemma to show that P is **not regular**

you can construct a more complex palindromes by adding 0 or 1 on the sides to an existing palindromes

Context-free grammars

A **context-free grammar** (**CFG**) consists of 4 components $G = (V, \Sigma, R, S)$

- V is a finite set of symbols called **variables** (or nonterminals)
  each variable represents a language (such as L and P in Examples 1, 2)
- $S \in V$ is a **start variable**
  other variables in V represent auxiliary languages we need to define S
- Σ is a finite set of symbols called **terminals** $(V \cap \Sigma = \emptyset)$
  terminals give alphabets of languages (such as $\{a, b\}$ and $\{0, 1\}$ in Examples 1, 2)
- R is a finite set of **rules** (or **productions**) of the form $A \to w$
  where A is a variable and w is a string of variables and terminals
  rules give a recursive definition of the language

Informally: to generate a string of terminal symbols from G, we:

- Begin with the start variable.
- Apply one of the productions with the start symbol on the left-hand side,
  replacing the start symbol with the right-hand side of the production
- Repeat selecting variables and replacing them with the right-hand side of some
  corresponding production, until all variables have been replaced by terminal symbols

CFGs: derivations and languages

Let $G = (V, \Sigma, R, S)$ be a CFG

For strings u and v of variables and terminals, we say that:

v is **derivable** from u in one step in G and write $u \Rightarrow_G^1 v$ if

v can be obtained from u by replacing some occurrence of A in u with w where $A \to w$ is a rule in R

v is **derivable** from u in G and write $u \Rightarrow_G v$ if there are $u_1, u_2, \dots, u_k$ such that

$$u \Rightarrow_G^1 u_1 \Rightarrow_G^1 u_2 \Rightarrow_G^1 \cdots \Rightarrow_G^1 u_k \Rightarrow_G^1 v$$

(**derivation** of v from u in G)

The **language of the grammar** G consists of all words over Σ that are derivable from the start variable S

$$L(G) = \{w \in \Sigma^* \mid S \Rightarrow_G w\}$$

$L(G)$ is a **context-free language**

Nonpalindromes

Example 3. Define the language N of **nonpalindromes** over $\{0, 1\}$

Basis: $0w1 \in N$ and $1w0 \in N$, for any $w \in \{0, 1\}^*$

have to define the language $A = \{0, 1\}^*$ (of all binary words) as well

Induction: if w is in N, then so is $0w0$ and $1w1$

This language can be defined by the following grammar G:

$$\boxed{\begin{array}{llll} N \to 0A1 & & & \\ N \to 1A0 & & & A \to \varepsilon \\ N \to 0N0 & & & A \to 0A \\ N \to 1N1 & & & A \to 1A \end{array}}$$

BNF: $N ::= 0A1 \mid 1A0 \mid 0N0 \mid 1N1 \qquad A ::= \varepsilon \mid 0A \mid 1A$

Test: is 0010 derivable in G from N?

$$N \Rightarrow^1_G 0N0 \Rightarrow^1_G 00A10 \Rightarrow^1_G 00\varepsilon 10 = 0010$$

More tests: $N \Rightarrow_G 1011$? $\quad 0NA0 \Rightarrow_G 001A0$? $\quad N \Rightarrow_G A$?

no

Regular languages are context-free

Example 4: show that the language of the regular expression $0^*1(0 \cup 1)^*$ is context-free

This language can be defined by the following grammar:

$$
\begin{aligned}
S &\to A1B \\
A &\to \varepsilon \\
A &\to 0A \\
B &\to \varepsilon \\
B &\to 0B \\
B &\to 1B
\end{aligned}
$$

BNF:

$$
\begin{aligned}
S &::= A1B \\
A &::= \varepsilon \mid 0A \\
B &::= \varepsilon \mid 0B \mid 1B
\end{aligned}
$$

Every **regular** language is also a **context-free** language

it is also easy to encode DFAs as CFGs

(states as variables, transitions as rules)

Applications of CFGs

Consider the language of the CFG $S \ ::= \ \varepsilon \ \mid \ (S) \ \mid \ SS$

can you describe it in English?

The language of this CFG consists of all strings of '(' and ')'

with **balanced** parentheses

CFGs are used to

- describe fragments of natural languages in linguistics (N. Chomsky)
- describe programming languages and markup languages (HTML)

  (and other recursive concepts in Computer Science)
- syntactic analysis in compilers

  before a compiler can do anything, it scans the input program (a string of ASCII characters) and determines the syntactic structure of the program. This process is called **parsing**.
- give document type definitions in XML

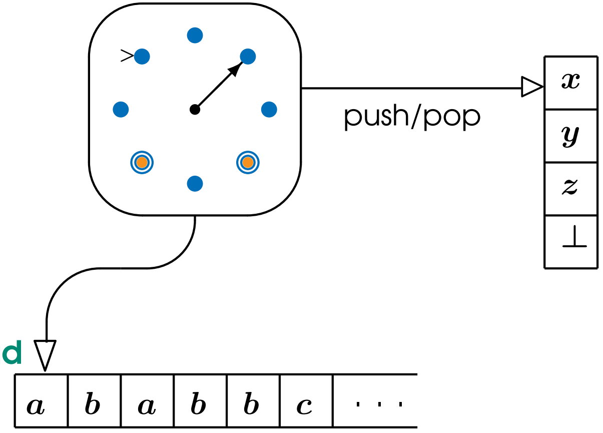
Problem

How to modify NFAs so that they could recognise context-free languages?

Pushdown automata

A (nondeterministic) **pushdown automaton (PDA)** is like an NFA, except that it has a **stack** that can be used to record a potentially **unbounded** amount of information (in some special way)

Finite control



Pushdown stack

- can **push** symbols onto the top of the stack
- or **pop** them off the top of the stack

(last–in–first–out)

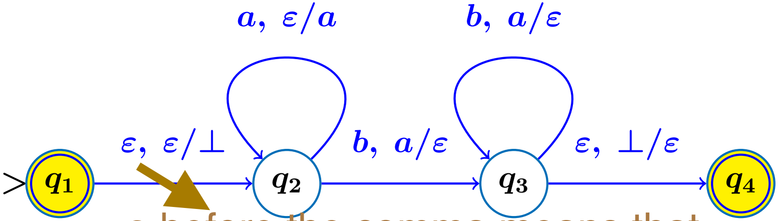
Reading head
(left to right,
read only)

A stack is a last in, first out abstract data type and data structure

PDA for $\{a^n b^n \mid n \geq 0\}$

- Read symbols from the input; as each a is read, **push** it onto the stack
- As soon as b's are seen, **pop** an a off the stack for each b read
- If reading the input is finished exactly when the stack becomes empty, accept the input
- Otherwise reject the input
- How to test for an empty stack?

Push initially some special symbol, say $\perp$ (bottom), on the stack



ε before the comma means that q1 can change without any input

$q \xrightarrow{a,\ x/\alpha} r$ (α a string) means:
if PDA is in state q,
reads a from input and
symbol x is on top of stack,
then PDA replaces x with α
and moves to state r

as before, a and x can be ε

what is the language of this automaton if we ignore the stack?

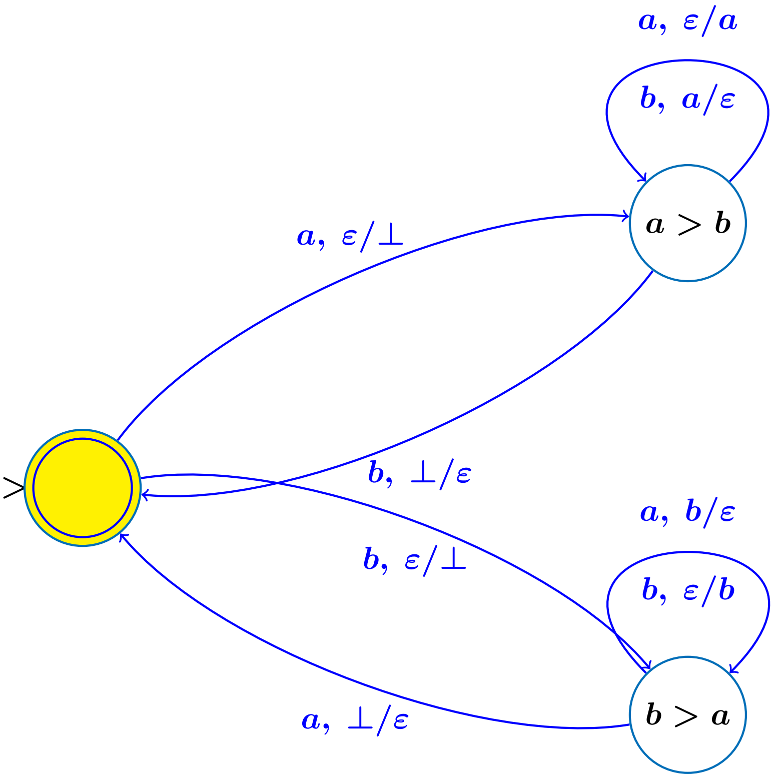
Exercise

For $\Sigma = \{a, b\}$, design a PDA and a CFG for the language

$$L \;=\; \{w \in \Sigma^* \mid w \text{ contains an equal number of } a\text{'s and } b\text{'s}\}$$

- The strategy will be to keep the excess symbols, either a's or b's, on the stack
- One state will represent an excess of a's
- Another state will represent an excess of b's
- We can tell when the excess switches from one symbol to the other because at that point the stack will be empty ($\perp$ on top)
- In fact, when the stack is empty, we may return to the start state

Exercise (cont.)



$$S \ ::= \ \varepsilon \mid aSb \mid bSa \mid SS$$

A formal definition of PDAs

A PDA is a 6-tuple $A = (Q, \Sigma, \Gamma, \delta, s, F)$ where (cf. the definition of NFAs)

- Q is a finite set of **states**
- Σ is a finite set, the **input alphabet**
- Γ is a finite set, the **stack alphabet**
- $s \in Q$ is the **initial state**
- $F \subseteq Q$ is the set of **accepting states**
- δ is a **transition relation** consisting of 'instructions' of the form $\big((q, a, x), (r, \alpha)\big)$ where q, r are states, a a symbol from Σ (input), x a symbol from Γ (stack), and α a word over Γ (stack), meaning intuitively that

if (1) A is in state q reading input symbol a on the input tape and
(2) symbol x is on the top of the stack,
then the PDA can (nondeterminism!)
(a) pop x off stack and push α onto stack (the first symbol in α is on the top),
(b) move its head right one cell past the a and enter state r

Computations of PDAs

Configuration of PDA A: $(state, word\_on\_tape, stack)$

Computation of PDA A on input w: (can be **many** computations!)

(s, au, ε) s is the initial state, $w = au$ and the stack is empty

$\downarrow$ if A contains an instruction $\big((s, a, \varepsilon), (r, xy)\big)$ then

(r, u, xy) r is the next state, head scans first symbol in u, stack is xy

$\downarrow$ if A contains an instruction $\big((r, \varepsilon, x), (q, \varepsilon)\big)$ then

(q, u, y) q is the next state, head scans first symbol in u, stack is y

$\downarrow$

$\dots$

(t, ε, α) if t is accepting ($t \in F$), then the computation is accepting

(similar to computations of NFAs)

Computations can also get stuck, end with non-accepting states, or even **loop**

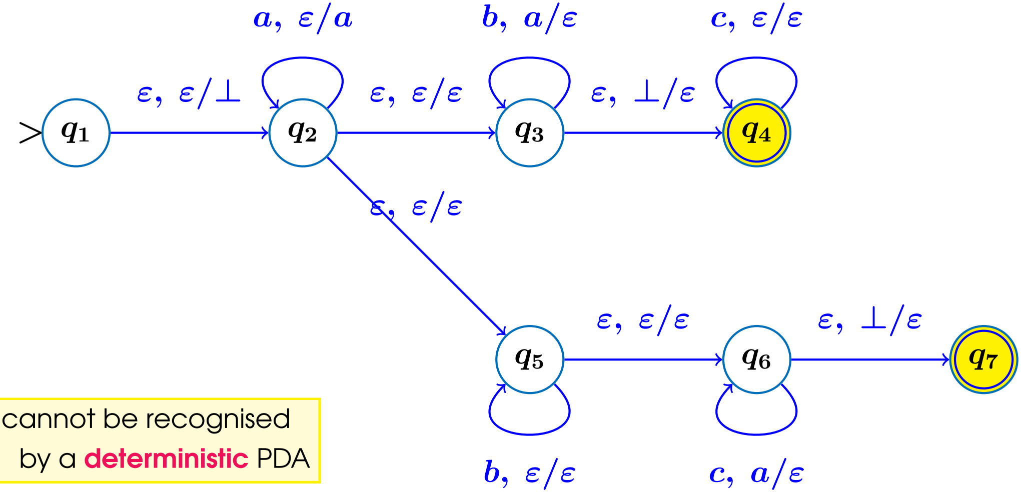
Exercise: design PDA recognising the language over $\{(,)\}$ with **balanced** parentheses

Using nondeterminism

Design a PDA recognising the language $L = \{a^i b^j c^k \mid i = j \text{ or } i = k\}$

L contains strings such as $aabbc$, $aabcc$, but not $abbcc$

Idea: start by reading and pushing the a's. When the a's are done, the PDA can match them with either the b's or the c's. Here we use **nondeterminism !**


$a,\ \varepsilon/a$
$b,\ a/\varepsilon$
$c,\ \varepsilon/\varepsilon$
$\varepsilon,\ \varepsilon/\bot$
$\varepsilon,\ \varepsilon/\varepsilon$
$\varepsilon,\ \bot/\varepsilon$
q_1
q_2
q_3
q_4
$\varepsilon,\ \varepsilon/\varepsilon$
$\varepsilon,\ \varepsilon/\varepsilon$
$\varepsilon,\ \bot/\varepsilon$
q_5
q_6
q_7
$b,\ \varepsilon/\varepsilon$
$c,\ a/\varepsilon$

this language cannot be recognised by a **deterministic** PDA

CFGs and PDAs

Context-free languages are precisely the languages recognised by pushdown automata

- There is an algorithm that, given any CFG G, constructs a PDA A such that $L(A) = L(G)$
- There is an algorithm that, given any PDA A, constructs a CFG G such that $L(G) = L(A)$

The following languages are **not** context free:

- $\{ww \mid w \in \{0,1\}^*\}$
- $\{a^n b^n c^n \mid n \geq 0\}$
- $\{a^{2^n} \mid n \geq 0\}$

can be shown using an analogue of the pumping lemma for PDAs

Unrestricted grammars (not examinable)

An **unrestricted grammar** consists of 4 components $G = (V, \Sigma, R, S)$

- V is a finite set of **variables**
- $S \in V$ is a **start variable**
- Σ is a finite set of **terminals** $(V \cap \Sigma = \emptyset)$
- R is a finite set of **rules** (or **productions**) of the form $\alpha \to \beta$

in CFGs, α is a variable!

where α and β are strings of variables and terminals

For strings u and v of variables and terminals, we say that

v is **derivable** from u in one step in G and write $u \Rightarrow^1_G v$ if

v can be obtained from u by replacing some substring α in u with β where $\alpha \to \beta$ is a rule in R

Example. The grammar G: $S \to aBSc,\ S \to abc,\ Ba \to aB,\ Bb \to bb$

generates (non-context-free) $\{a^n b^n c^n \mid n \geq 0\}$

$S \Rightarrow^1_G aBSc \Rightarrow^1_G aBabcc \Rightarrow^1_G aaBbcc \Rightarrow^1_G aabbcc$

Testing membership in languages

Problem: given a string w and a language L, decide whether w is in L

- for L given by a DFA: simulate the DFA processing of w.

  test takes time proportional to $|w|$

- for L given by a NFA with k states:

  test can be done in time proportional to $|w| \times k^2$

  each input symbol can be processed by taking the previous set of (at most k) states and looking at the successors of each of these states

- for L given by a CFG of size k: test can be done in time proportional to $|w|^3 \times k^2$

- for L given by an unrestricted grammar:

  cannot be solved by **any** mechanical procedures (such as computer programs)

Is it possible to design a formal model of computation that would capture capabilities of **any computer program ?**