Pushdown Automata and Context Free Language Parsing

Prof. A. Morzenti

NB: up to slide 9 notions assumed to be known from other previous courses

PUSHDOWN AUTOMATA

$$A_1 A_2 \dots \overbrace{A_k}^{top}$$

$$a_1 a_2 \dots \overbrace{a_i}^{curr.char.-cc} \dots a_n \dashv$$

1) stack auxiliary memory +
   input string with terminator ⊣
3) operations:
   push(B), push(B_1, B_2, ... B_n): push symbol(s) on top of the stack
   empty test: a predicate that holds iff $k = 0$
   pop, if the stack is not empty, deletes A_k
4) Z_0 is the *initial* (*bottom*) stack symbol (can only be read)
5) configuration: current state, string portion from current character ***cc***, stack content

MOVE OF THE AUTOMATON:

- read *cc* and advance the head (***shift***), or (***spontaneous*** move) do not advance the head
- read the top stack symbol (possibly Z_0 if the stack is empty)
- based on the current char, state, and stack top symbol, go to a new state and replace the stack top symbol with a string (zero or more symbols)

<u>DEFINITION OF PUSHDOWN AUTOMATON</u>

A pushdown autromaton M (in general nondeterministic) is defined by:

1.	Q	*finite set of states* of the control unit
2.	Σ	*input alphabet*
3.	Γ	*stack alphabet*
4.	δ	*transition function*
5.	$q_0 \in Q$	*initial state*
6.	$Z_0 \in \Gamma$	*inital stack symbol*
7.	$F \subseteq Q$	set of *final states*

spontaneous move

TRANSITION FUNCTION:

domain: $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$

range: the powerset $\wp(Q \times \Gamma^*)$ of $Q \times \Gamma^*$

nondeterminism

READING (/ scanning / shift) MOVE:
(possibly nondeterministic)

$$\boxed{\begin{array}{l}\delta(q,a,Z)=\{(p_1,\gamma_1),(p_2,\gamma_2),\ldots(p_n,\gamma_n)\}\\ \text{with } n\geq 1,\ a\in\Sigma, Z\in\Gamma,\ \ p_i\in Q,\gamma_i\in\Gamma^*\end{array}}$$

SPONTANEOUS MOVE:
(possibly nondeterministic)

$$\boxed{\begin{array}{l}\delta(q,\varepsilon,Z)=\{(p_1,\gamma_1),(p_2,\gamma_2),\ldots(p_n,\gamma_n)\}\\ \text{with } n\geq 1,\ Z\in\Gamma,\ p_i\in Q,\ \ \gamma_i\in\Gamma^*\end{array}}$$

NONDETERMINISM: for a given triple (state, stack top, input) there are ≥2 possibilities among reading and spontaneous moves

INSTANTANEOUS CONFIGURATION OF MACHINE ***M***: a triple

$$\boxed{(q, y, \eta) \in Q \times \Sigma^* \times \Gamma^+}$$

current state
of the control unit

unread portion
of input x

stack content

INITIAL CONFIGURATION: (q_0, x, Z_0)
FINAL CONFIGURATION (q, ε, η) if $q \in F$ (NB: ε means input completely scanned)

TRANSITION FROM ONE
CONFIGURATION TO THE NEXT:

$$\boxed{(q, y, \eta) \to (p, z, \lambda)}$$

TRANSITION SEQUENCE: $\overset{*}{\to}$, $\overset{+}{\to}$

current config.	next config	applied move
$(q, az, \eta Z)$	$(p, z, \eta\gamma)$	reading move $\delta(q, a, Z) = \{(p, \gamma), ...\}$
$(q, az, \eta Z)$	$(p, az, \eta\gamma)$	spontaneous move $\delta(q, \varepsilon, Z) = \{(p, \gamma), ...\}$

NB: A ***string*** is pushed: it can be ε (just a *pop* operation)
or the same symbol previously on top (stack is unchanged)

A string *x* is recognized/accepted with final state if:

$$\boxed{(q_0, x, Z_0) \xrightarrow{+} (q, \varepsilon, \lambda)}$$

$q \in F$ and $\lambda \in \Gamma^*$
(no condition on λ, it can be ε, but not necessarily)

STATE-TRANSITION DIAGRAM FOR PUSHDOWN AUTOMATA

Example: even-length palindromes accepted with final state by a (**nondeterministic**) PDA

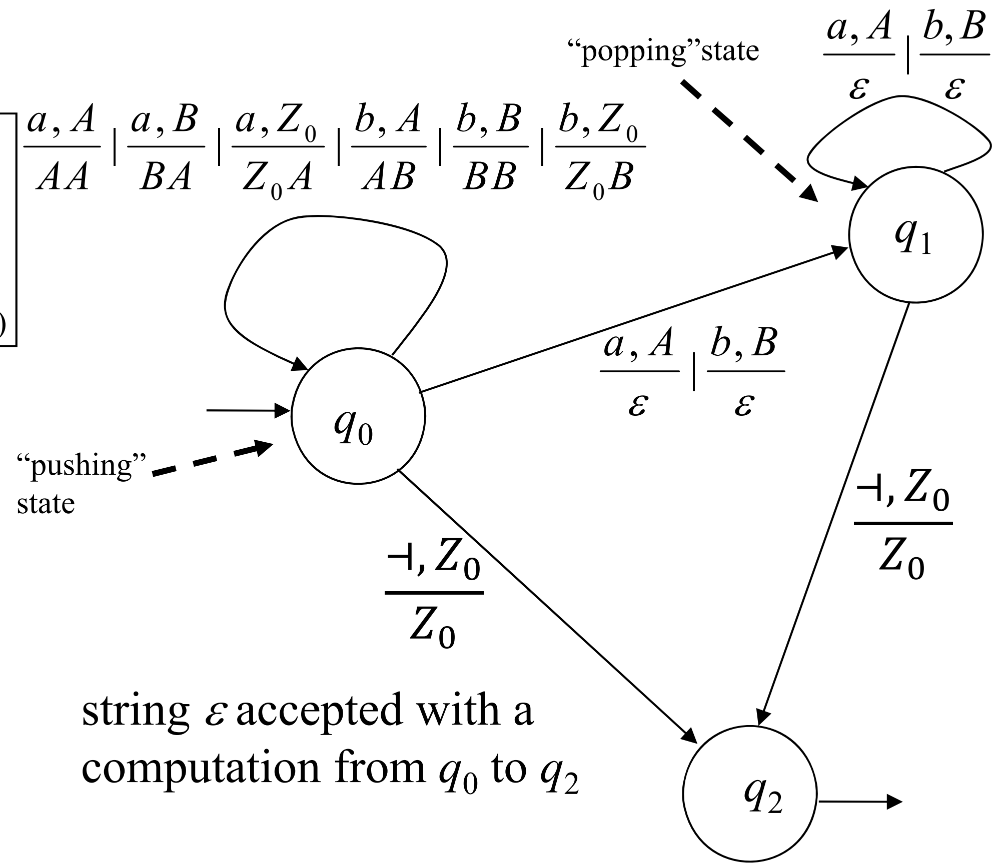
$$L = \left\{ uu^R \mid u \in \{a, b\}^* \right\}$$

Stack	x	State	Comment
Z_0	$aa\dashv$	q_0	
Z_0A	$a\dashv$	q_0	
Z_0AA	$\dashv$	q_0	reject: no move defined for (q_0, ⊣, A)

"guesses" that $|x|>2$

Stack	x	State	Comment
Z_0	$aa\dashv$	q_0	
Z_0A	$a\dashv$	q_0	
Z_0	$\dashv$	q_1	
Z_0	ε	q_2	acceptance with final state

"guesses" that $|x|=2$



string ε accepted with a computation from q_0 to q_2

There exist conversion procedures

Context-free grammar $\Leftrightarrow$ Pushdown Automaton (possibly nondeterministic)

Therefore:

LANGUAGES
ACCEPTED BY ***NONDETERMINISTIC*** PDA's
AND
GENERATED BY CF GRAMMARS
ARE A UNIQUE FAMILY

VARIETIES OF PUSHDOWN AUTOMATA

PDA can be enriched in various ways, concerning internal states and acceptance conditions

3 possible accepting modes:
- with final state (stack content immaterial)
- empty stack (current state immaterial)
- combined: (final state and empty stack)

PROPERTY – These three accepting modes are equivalent

ELEMENTS UNTIL HERE ASSUMED TO BE KNOWN FROM PREVIOUS COURSES

PUSHDOWN AUTOMATA AND DETERMINISTIC LANGUAGES (DET)

Only deterministic CF languages (those accepted by a deterministic PDA) are considered in language and compiler design due to efficiency reasons

Nondeterminism is absent if δ is one-valued and also, $\forall \boldsymbol{q} \in \boldsymbol{Q}$ and $\forall \boldsymbol{A} \in \boldsymbol{\Gamma}$

if $\boldsymbol{\delta(q, a, A)}$ is defined for some $\boldsymbol{a} \in \Sigma$, then $\boldsymbol{\delta(q, \varepsilon, A)}$ is not defined, and
if $\boldsymbol{\delta(q, \varepsilon, A)}$ is defined then $\boldsymbol{\delta(q, a, A)}$ is not defined for any $\boldsymbol{a} \in \boldsymbol{\Sigma}$

NB: therefore **a deterministic PDA *CAN* have spontaneous moves**

Relation between classe CF (Context Free Languages) and DET (deterministic ones)

Example: nondeterministic union of deterministic languages

$$\boxed{L = \left\{ a^n b^n \mid n \geq 1 \right\} \bigcup \left\{ a^n b^{2n} \mid n \geq 1 \right\} = L' \bigcup L''}$$

a PDA accepting x must push all a's and, if $x \in L'$ (e.g., *aabb*), pop one a for each b;
but if $x \in L''$ (e.g., *aabbbb*), two b's must be popped for each a in the stack
The PDA does not know which alternative holds, it must try both

L',L''∈DET, L',L''∈CF, L=L'∪L'', L∈CF but L∉DET,

hence **DET ⊆ CF and DET ≠ CF**

But do not worry: there is a procedure to determine if a given free ***grammar*** ∈ *DET*

SYNTAX ANALYSIS

Given a grammar ***G***, the syntax analyzer (***parser***)

- reads the source string x and
- if $x \in L(G)$,
    - accepts and possibly outputs a syntax tree or (equivalently) a derivation;
- otherwise it stops signalling an error (diagnosis)

TOP-DOWN AND BOTTOM-UP ANALYSIS

One given tree in general corresponds to various derivations (left, right, ...)

The two most important types of parsers characterized by the type of identified derivation: **left** or **right**, and **order** of the tree and derivation construction

TOP-DOWN ANALYSIS: builds
a **left** derivation in **direct order**
syntax tree: **from root to leaves**, through ***expansions***

BOTTOM-UP ANALYSIS: builds
right derivation in **reverse** order
syntax tree: **from leaves to root**, through ***reductions***

Example – top-down analysis of sentence:

a b b b a a

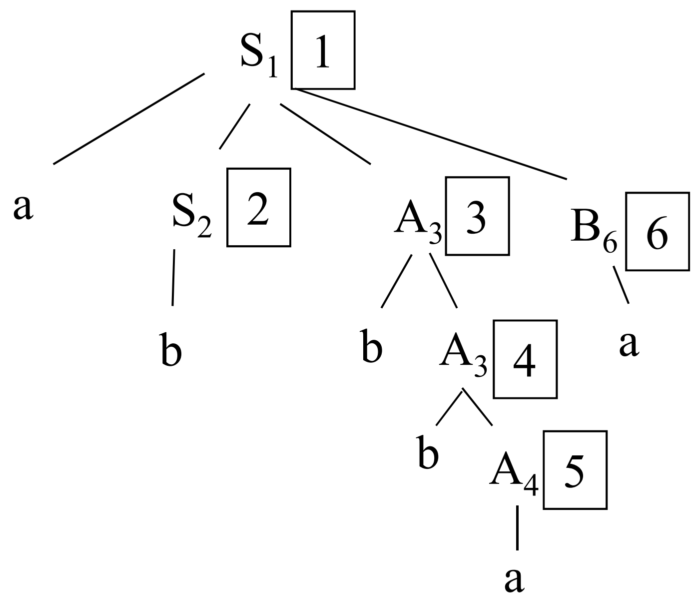
framed numbers = order in rule application

subscripts of nonterminals in the tree = applied rule

Leftmost: always expanded first nonterm. from the left

1. $S \to aSAB$	2. $S \to b$
3. $A \to bA$	4. $A \to a$
5. $B \to cB$	6. $B \to a$

$$S \Rightarrow aSAB \Rightarrow abAB \Rightarrow abbAB \Rightarrow abbbAB \Rightarrow abbbaB \Rightarrow abbbaa$$



Example – bottom-up analyis of sentence:

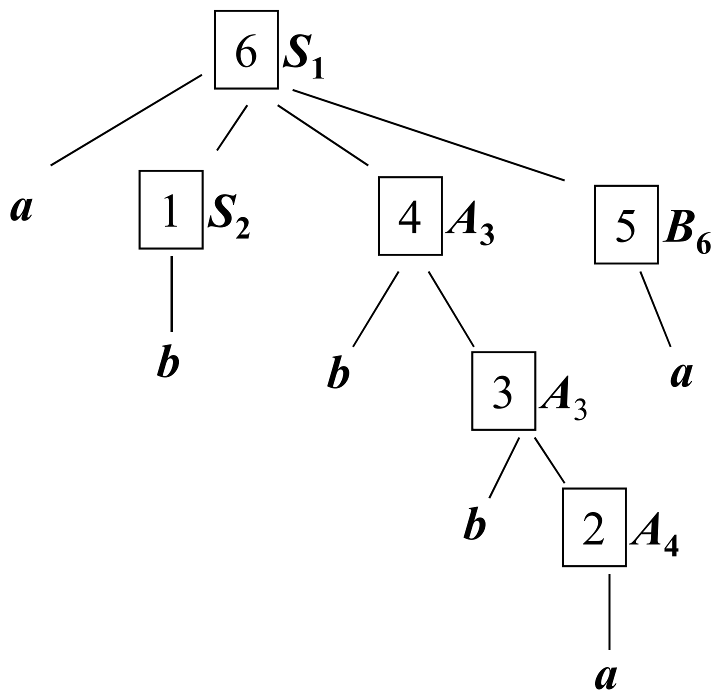
a b b b a a

1. $S \to aSAB$	2. $S \to b$
3. $A \to bA$	4. $A \to a$
5. $B \to cB$	6. $B \to a$

Derivation (rightmost n.t. always expanded):

$S \Rightarrow aSAB \Rightarrow aSAa \Rightarrow aSbAa \Rightarrow aSbbAa \Rightarrow aSbbaa \Rightarrow abbbaa$

framed numbers = order in reduction sequence
subscripts of nonterminals in the tree = applied rule



right parts of rules are ***reduced*** as they are scanned, first in the sentence, then in the phrase forms obtained after the ***reductions***. The process terminates when the entire string is reduced to the axiom

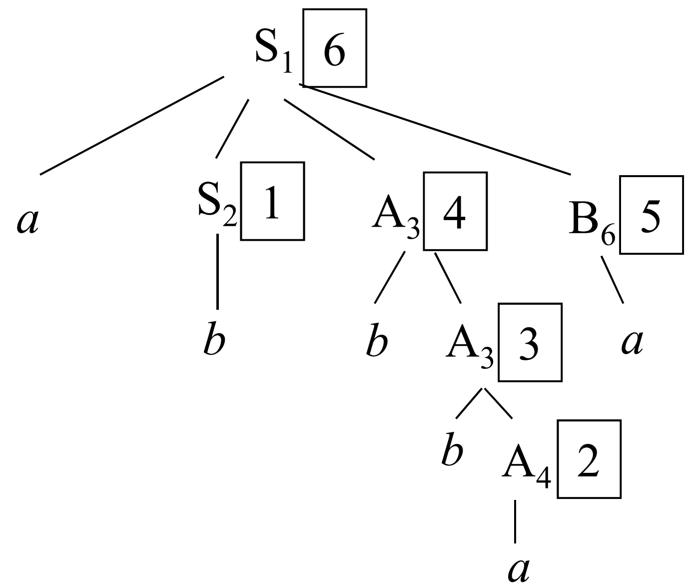
NB: reconstruction of a ***right derivation*** in ***reverse order***

Why in reverse order? Because input string is ***read from left***

Bottom-up analysis: the **reduction** operations (>rd> below) trasform prefix of a phrase form into a string $\alpha \in (\Sigma \cup V)^*$, called «***viable prefix***», that may include the result of previous reductions and is stored in the parser's stack. Here below, the '^' shows the head position (right of ^ is the unread string), **underline** shows the reduced part, called «***handle***»

$a\underline{\mathbf{b}}bbaa$ >rd> $aSbb\underline{\mathbf{a}}a$ >rd> $aSb\underline{\mathbf{bA}}a$ >rd> $aS\underline{\mathbf{bA}}a$ >rd> $aSA\underline{\mathbf{a}}$ >rd> $\underline{\mathbf{aSAB}}$ >rd> S

1. $S \rightarrow aSAB$	2. $S \rightarrow b$
3. $A \rightarrow bA$	4. $A \rightarrow a$
5. $B \rightarrow cB$	6. $B \rightarrow a$



At each step of the analysis, the parser must decide whether

- to continue and read the next symbol (shift), or
- to build a subtree from a portion of the viable prefix

it chooses based on the symbol(s) coming after the current one (***lookahead***)
NB: the reason for the choice is not explained here: it will be in coming lessons

EXTENDED GRAMMARS
REPRESENTED AS NETWORKS OF FINITE AUTOMATA

We use $\boldsymbol{G}$ in extended form: every nonterminal A has a unique rule

$\boldsymbol{A} \rightarrow \boldsymbol{\alpha}$ with $\boldsymbol{\alpha}$ regular expression on terminals and nonterminals

α defines a regular language, hence there exists a finite automaton $\boldsymbol{M_A}$ that accepts $\boldsymbol{L(\alpha)}$

any transition of $\boldsymbol{M_A}$ labeled by a nonterminal $\boldsymbol{B}$

is interpreted as a «call» of an automaton $\boldsymbol{M_B}$ (if $\boldsymbol{B=A}$ then recursive call)

let us call
- "***machines***" the finite automata of the various nonterminals,
- "***automaton***" the PDA that accepts and parses the language $\boldsymbol{L(G)}$
- "***net***" the set of all machines

- $\boldsymbol{L(q)}$ the set of terminal strings generated along a path of a machine, starting from state $\boldsymbol{q}$ (possibly including calls of other machines) and reaching a final state ($\Rightarrow$ if $\boldsymbol{q}$ is final then $\boldsymbol{L(q)}=\{\varepsilon\}$)

We set a further requirement on machines corresponding to nonterminals

The initial state $\mathbf{0}_A$ of machine $\boldsymbol{A}$ is not visited after the start of the computation

No machine $\boldsymbol{M_A}$ may include a transition like

$$\boldsymbol{M_A}: \quad \rightarrow \mathbf{0}_A \xleftarrow{\;\boldsymbol{c}\;} \boldsymbol{q} \quad \ldots$$

Requirement very easy to "implement" in case it is not satisfied …

… it suffices to add one state (to be the new initial state) and a few transitions from it …

$\Rightarrow$ the machine is not minimal, but only one state has been added

Automata satisfying this condition are called

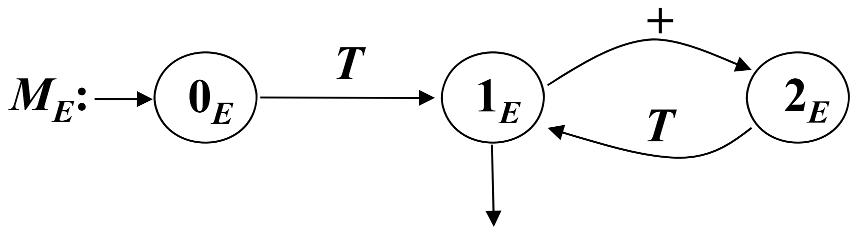
normalized or with initial state **non recirculating** or **non reentrant**

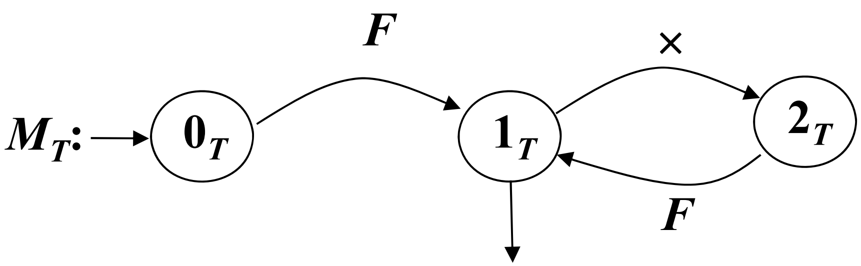
NB: it is **NOT forbidden** that the initial state be also final

(which is necessary if $\varepsilon \in \boldsymbol{L}(\boldsymbol{A})$)

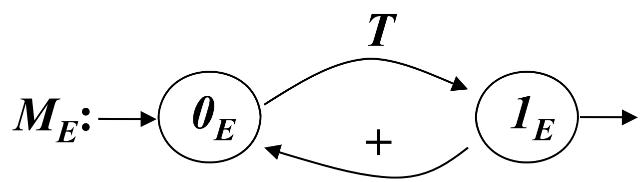
Example – Arithmetic expressions

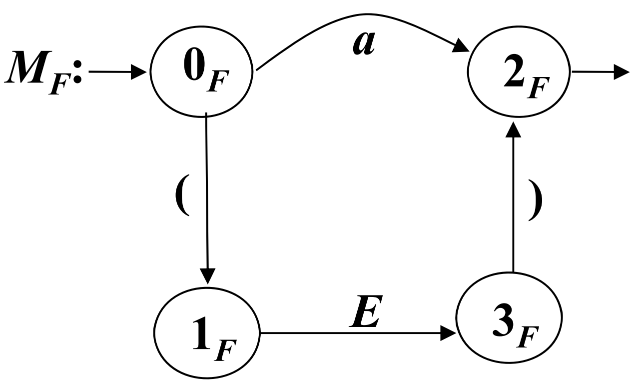
$$E \to T\,(+\,T\,)^*$$
$$T \to F\,(\,\times\,F\,)^*$$
$$F \to a \mid \text{‘(’}\ E\ \text{‘)’}$$





NB: M_E minimal not normalized





$L(2_F) = \{\ \varepsilon\ \}$ reason: state 2_F is final

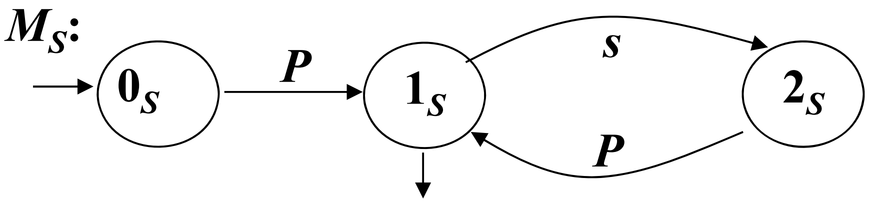
$L(3_F) = \{\ )\ \}$

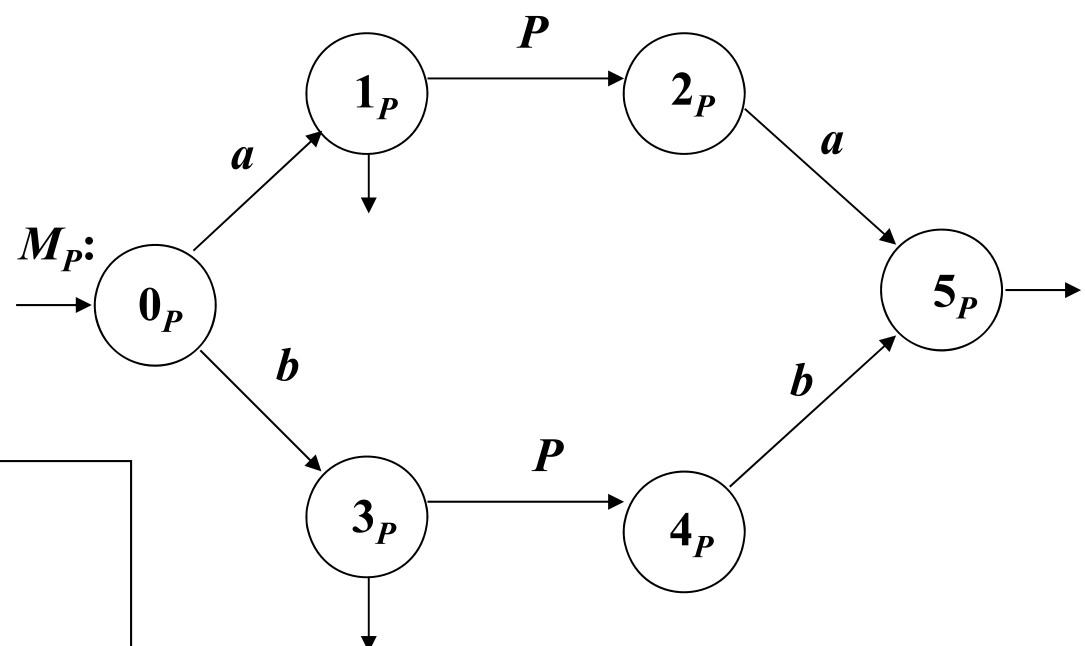
$L(0_F) = L(F)$ the language generated from $\boldsymbol{F}$

$L(1_F) = L(E) \cdot \{\ )\ \}$

Example – List of odd-length palindromes, separated by '*s*'

$$S \rightarrow P\ (\ s\ P\ )^{*}$$
$$P \rightarrow aPa \mid bPb \mid a \mid b$$





NB: M_S minimal not normalized