Recognition of Free Languages

Translated and adapted by L. Breveglieri

RECOGNITION OF FREE LANGUAGES

AUTOMATON MODEL FOR FREE LANGUAGES

- finite state memory and pushdown stack memory
- automaton moves correspond to grammar rules
- in general the automaton is nondeterministic
- the automaton has two memories: state and stack

CONTENTS

- pushdown automaton model (non-deterministic)
- deterministic automaton (deterministic free languages)
- parsing algorithm for deterministic grammars
  - fast / linear time / for the *ELL* (1) and *ELR* (1) families
- general parsing algorithm (Earley) for non-deterministic grammars

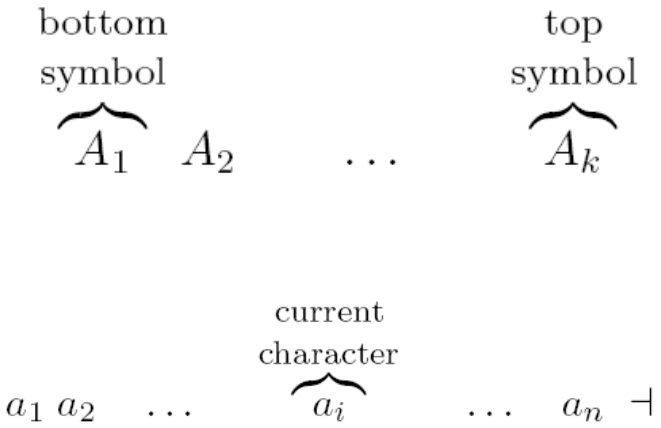
PUSDOWN STACK AUTOMATON

Auxiliary memory structured as an unbounded stack of symbols

Input string (or source string)

Available operations

- *push (B)*, *push (B_1, B_2, ..., B_n)*: places the symbol(s) onto the stack top, i.e., writes them on the right of A_k
- *pop*: removes symbol A_k from the stack top, if the stack is not empty; otherwise reads Z_0
- *stack emptyness test*: yields *true* if the stack is empty, otherwise yields *false*



The symbol Z_0 is the stack bottom and can be read, not removed

"⊣" is the end-of-text (or terminator) character of the input string

A configuration is triple: *current state*, *current char*, *stack contents*

A MOVE OF THE PUSHDOWN AUTOMATON

- reads the current character and shifts the input head, or performs a spontaneous move without shifting the input head
- reads the stack top symbol and removes it from the top if the stack is not empty, or reads the stack symbol Z_0 (without removing it) if the stack is empty
- depending on the current character, state and stack top symbol, it goes into the next state and places none, one or more symbols onto the stack top

DEFINITION OF PUSHDOWN (STACK) AUTOMATON

A pushdown automaton *M* (in general nondeterministic) is defined by

Q	*a finite set of states of the control unit*
Σ	*a finite input alphabet*
Γ	*a finite stack alphabet*
δ	*a transition function*
$q_0 \in Q$	*the initial state*
$Z_0 \in \Gamma$	*the initial stack symbol*
$F \subseteq Q$	*a set of final states (obviously finite)*

TRANSITION FUNCTION δ

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

image the set of the subsets of $Q \times \Gamma^*$ or $\wp (Q \times \Gamma^*)$

spontaneous move

non-determinism

READING MOVE – in the state q with symbol Z_0 on the stack top, the automaton reads char a and enters one of the states p_i with $1 \leq i \leq n$, after orderly executing the operations *pop* and *push* (γ_i)

$$\delta(q, a, Z) = \{ (p_1, \gamma_1), (p_2, \gamma_2), \ldots, (p_n, \gamma_n) \}$$

SPONTANEOUS MOVE – in the state q with symbol Z_0 on the stack top, the automaton does not read any input character and enters one of the states p_i with $1 \leq i \leq n$, after orderly executing the operations *pop* and *push* (γ_i)

$$\delta(q, \varepsilon, Z) = \{ (p_1, \gamma_1), (p_2, \gamma_2), \ldots, (p_n, \gamma_n) \}$$

NOTE – choosing the i-th action out of the n possible ones is not deterministic; shifting the input head is automatic; the stack top symbol is always popped; and the pushed string may be empty

NON-DETERMINISM – for a triple (*state*, *input*, *stack top*) there are two or more possible moves that consume none or one input character

INSTANTANEOUS CONFIGURATION OF A MACHINE *M*

It is a triple

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

that describes

q	the current state of the control unit
y	the part of the input string *x* that still has to be scanned
η	the current contents (a string) of the pushdown stack

INITIAL AND FINAL CONFIGURATIONS

(q_0, x, Z_0) $\qquad$ $(q, \varepsilon, \lambda)$ q is a final state and $\lambda \in \Gamma^*$

TRANSITION FROM A CONFIGURATION TO ANOTHER ONE

$$(q, y, \eta) \rightarrow (p, z, \lambda)$$

A chain of one or more transitions is denoted by $\overset{+}{\rightarrow}$

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

NOTE – every move cancels the stack top symbol; if we wish to keep it on stack, it can be immediately placed again onto the stack top by the move itself

An input string *x* is *accepted by final state* if there is the following computation

$$(q_0,\, x,\, Z_0) \overset{*}{\longmapsto} (q,\, \varepsilon,\, \lambda)$$

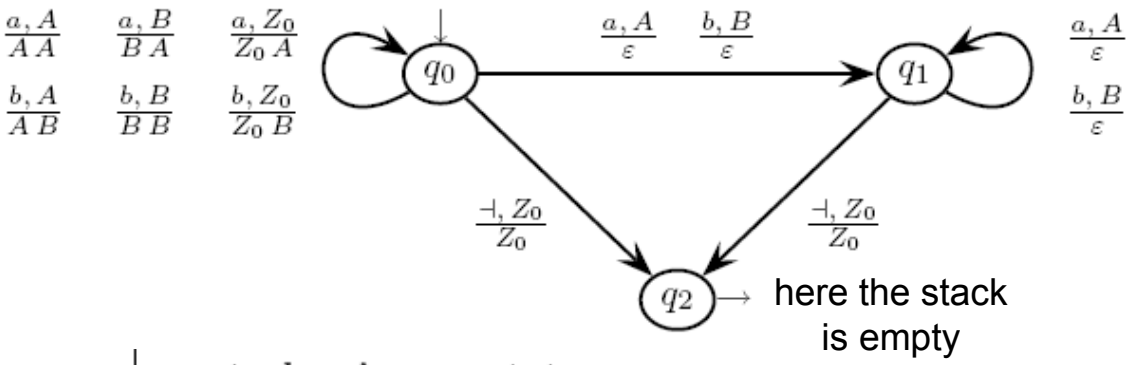
where $q \in F$ and $\lambda \in \Gamma^*$, whereas there is not any specific condition for λ; sometimes λ happens to be the empty string, but this is not necessary

STATE-TRANSITION GRAPH FOR PUSHDOWN AUTOMATA

EXAMPLE – palindromes of even length accepted by final state by a pushdown recognizer (non-deterministic)

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

String *a a* is accepted as it leads to the final state. The empty string is accepted by the move from q_0 to q_2



stack	inp. x	state	comment
			"bets" that | x | > 2
Z_0	$a\ a\ \dashv$	q_0	
$Z_0\ A$	$a\ \dashv$	q_0	
$Z_0\ A\ A$	$\dashv$	q_0	failure: no move is defined for $(q_0, \dashv, A)$

stack	inp. x	state	comment
			"bets" that | x | = 2
Z_0	$a\ a\ \dashv$	q_0	
$Z_0\ A$	$a\ \dashv$	q_0	
Z_0	$\dashv$	q_1	
Z_0	$\dashv$	q_2	recognition with final state

FROM A GRAMMAR TO A PUSHDOWN AUTOMATON

1. Grammar rules can be viewed as the instructions of a non-deterministic pushdown automaton (with only one finite state). Intuitively such an automaton works in a *goal-oriented* way and uses the stack as a notebook of the sequence of actions to undertake in the next future
2. The stack symbols can be both terminals and non-terminals of the grammar. If the stack contains the symbol sequence $A_1 \ldots A_k$, then the automaton executes first the action associated with A_k, which should recognize if in the input string from the position of the currrent character a_i there is a string w that can be derived from A_k; if it is so, then the action shifts the input head of $| w |$ positions
3. An action can be recursively divided into a series of sub-actions, if to recognize the non-terminal symbol A_k it is necessary to recognise other (non-)terminals

The initial action is the grammar axiom: the pushdown recognizer must check if the source string can be derived from the axiom. Initially the stack contains only the symbol Z_0 and the axiom S, and the input head is positioned on the initial (leftmost) character of the input string. At every step the automaton chooses (non-deterministically) one applicable grammar rule and executes the corresponding move. The input string is recognized (accepted) when, and only when, it is completely scanned and the stack is empty

given a grammar G = (V, Σ, P, S), with A, $B \in$ V, $b \in \Sigma$ and $A_i \in$ V $\cup \Sigma$

grammar rule	*automaton move*	*comment*
$A \to B\ A_1\ \ldots\ A_n$ with $n \geq 0$	if $top = A$ then pop; push $(A_n\ \ldots\ A_1 B)$	to recognize A first recognize $B\ A_1\ \ldots\ A_n$
$A \to b\ A_1\ \ldots\ A_n$ with $n \geq 0$	if $cc = b$ and $top = A$ then pop; push $(A_n\ \ldots\ A_1)$; shift reading head	character b was expected as next one and has been read so it remains to recognize $A_1\ \ldots\ A_n$
$A \to \varepsilon$	if $top = A$ then pop	the empty string deriving from A has been recognized
for every character $b \in \Sigma$	if $cc = b$ and $top = b$ then pop; shift reading head	character b was expected as next one and has been read
$- - -$	if $cc = \dashv$ and the stack is empty then accept; halt	the string has been entirely scanned and the agenda contains no goals

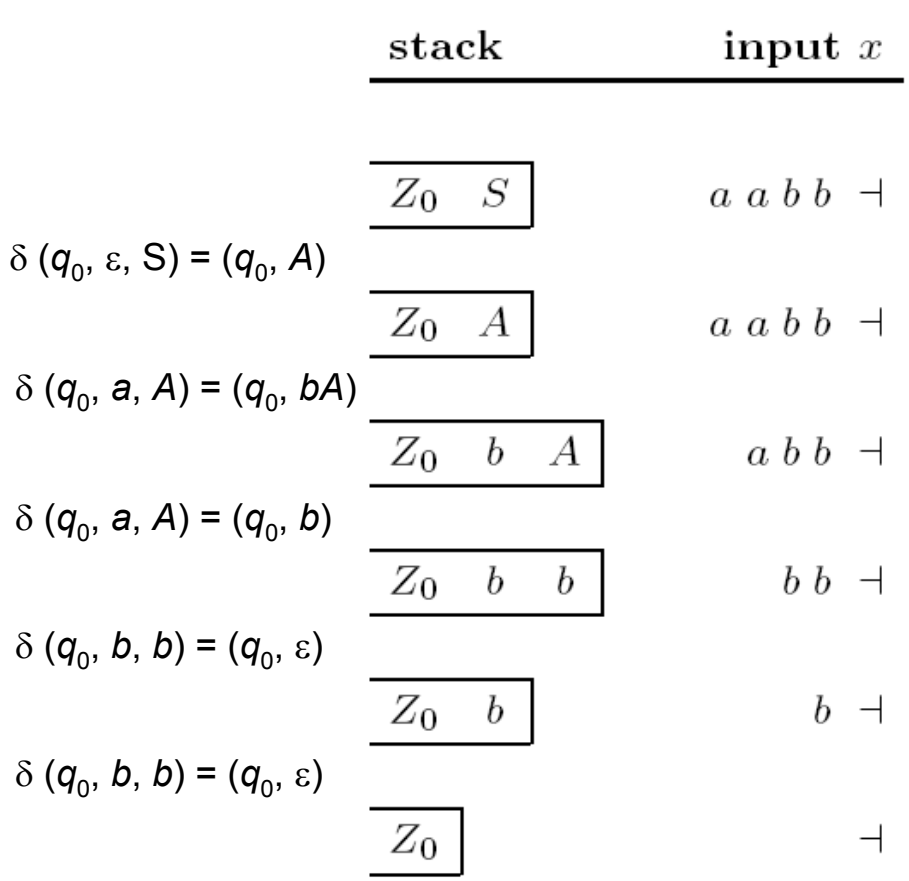
EXAMPLE – rules and moves of the predictive language recognizer

$$L = \left\{ a^n b^m \mid n \geq m \geq 1 \right\}$$

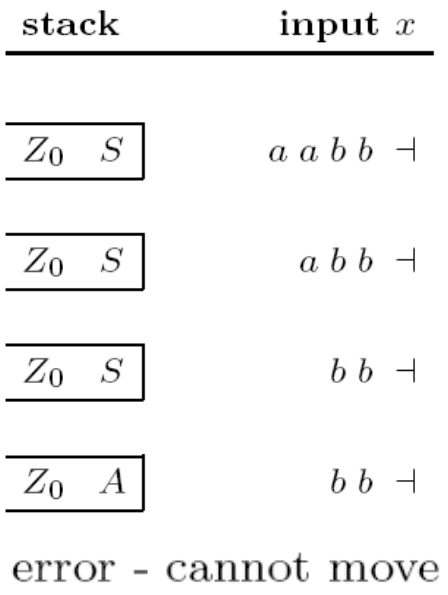
#	*grammar rule*	*automaton move*
1	$S \rightarrow a\ S$	if $cc = a$ and $top = S$ then pop; push (S); shift
2	$S \rightarrow A$	if $top = S$ then pop; push (A)
3	$A \rightarrow a\ A\ b$	if $cc = a$ and $top = A$ then pop; push $(b\ A)$; shift
4	$A \rightarrow\ a\ b$	if $cc = a$ and $top = A$ then pop; push (b); shift
5		if $cc = b$ and $top = b$ then pop; shift
6		if $cc = \dashv$ and the stack is empty then accept; halt

There is non-determinism between moves 1 and 2 (move 2 can be chosen even with an *a* in the input), and between moves 3 and 4. String $a^n b^m$ with $n \geq m \geq 1$ is analized as $a^{n-m} a^m b^m$. The machine guesses the position where the suffix $a^m b^m$ begins and it chooses move 1 or 2. It also guesses where the substring a^m ends and the suffix b^m begins, so where it chooses move 3 or 4

The atomaton recognizes a string if, and only if, the string is generated by the grammar: for every successful automaton computation there exists a grammar derivation and viceversa; thus the automaton simulates the *leftmost derivations* of the grammar



However the automaton cannot guess the correct derivation; it has to examine all the possibilities, e.g.



A string is accepted by two or more different computations if, and only if, the grammar is ambiguous

The construction from grammar to automaton can be inverted and used to obtain the grammar by starting from the pushdown automaton, if this is one-state.

PROPERTY – the family of free languages generated by free grammars coincides with the family of the languages recognized by one-state pushdown automata (in general non-deterministic)

NOTE – the construction does not work for multi-state pushdown automata; however the property holds in general, for the automata with any number of states, although the proof needs more sophisticated concepts and tools; see the bibliography

Unfortunately in general the resulting pushdown automaton is non-deterministic, as it explores all the moves applicable at any point and has an exponential time complexity with respect to the length of the source string. There are more efficient algorithms (see next)

VARIETIES OF PUSHDOWN AUTOMATA

The pusdown automaton defined in the previous example differs from that directly obtainable from the grammar for two reasons: it has two or more states and uses a different acceptance condition (final state instead of empty stack)

POSSIBLE ACCEPTANCE MODES

1. *by final state*: accepts when enters a final state independently of the stack contents
2. *by empty stack*: accepts when the stack gets empty independently of the current state
3. *combined*: by final state and empty stack

PROPERTY – for the family of (non-deterministic) pushdown automata with states, the three acceptance modes listed above are equivalent,i.e., the three families of recognized languages coincide

OPERATION WITHOUT SPONTANEOUS LOOPS AND ON-LINE

A generic pushdown automaton may execute an unlimited number of moves without reading any input character. This happens if, and only if, it enters a loop made only of spontaneous moves. Such a behaviour prevents it of completely reading the input string, or causes it to execute an unlimited number of moves before deciding whether to accept or reject the string. Both behaviours are undesirable in the practice

It is always possible to build an equivalent automaton with no spontaneous loops

A pushdown automaton operates in on-line mode if it decides whether to accept or reject the string as soon as it reads the last character of the input string, and then it does not execute any other move. Clearly from a practical perspective the on-line mode is a desirable behaviour

It is always possible to build an equivalent automaton that works in on-line mode

FREE LANGUAGES AND PUSDOWN AUTOMATA – ONE FAMILY

The language accepted by a generic pushdown automaton is free. Since every free language can be recognized by a non-deterministic one-state pushdown automaton, it follows

PROPERTY – the family CF of (context-)free languages coincides with that of the languages recognized by unrestricted pushdown automata

And more specifically

PROPERTY – the family CF of (context-) free languages coincides with that of the languages recognized by the one-state non-deterministic pushdown automata

RECOGNIZING THE INTERSECTION OF FREE AND REGULAR LANGUAGES

It is easy to justify (if not to prove rigorously) that the intersection of a free and a regular language is free as well

Given a grammar G and a finite state automaton *A*, the pushdown automaton *M* that recognizes the intersection *L* (*G*) ∩ *L* (*A*) can be obtained as follows

1) construct the one-state pushdown automaton *N* that recognizes *L* (*G*) by empty stack
2) construct the pushdown automaton *M* (with states), the state-transition graph of which is the cartesian product of those of *N* and *A*, by the cartesian product construction so that the actions of *M* on the stack are the same as those of *N*

The obtained pushdown automaton *M*

1. as its states, has pairs of states of the component machines *N* and *A*
2. accepts by final state and empty stack (combined acceptance mode)
3. the states that contain a final state of *A* are themselves final
4. is deterministic, if both component machines *N* and *A* are so
5. accepts by final state all and only the strings that belong to the intersection language

EXAMPLE – $L_{\text{Dyck}} \cap a^* a'^+ = \{ a^n a'^n \mid n \geq 1 \}$

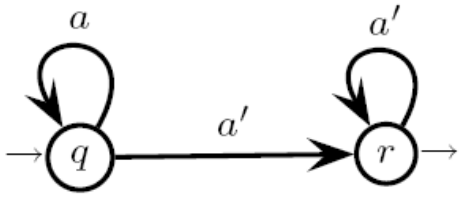
Dyck language with one nest only

$\Sigma = \{ a, a' \}$

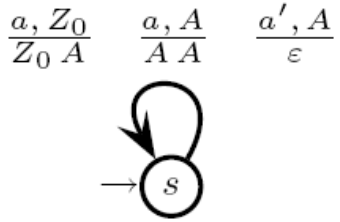
pushdown automaton that recognizes by empty stack

finite state automaton that recognizes by final state

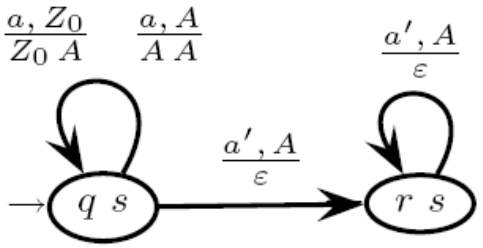
finite automaton



(Cartesian) product construction of PDA and FSA



pushdown machine N



product machine M

product pushdown automaton that recognizes by empty stack in the final state (*r*, *s*)

PUSHDOWN AUTOMATA AND DETERMINISTIC LANGUAGES (DET)

We study more in detail the deterministic recognizers and their languages, which are the most used in the compilers thanks to their efficiency

Nondeterminism is absent if the transition function δ is one-valued and

if δ (q, a, A) is defined then δ (q, ε, A) is undefined

if δ (q, ε, A) is defined then δ (q, a, A) is undefined for every $a \in \Sigma$

If the transition function does not exhibit any form of non-determinism, then the automaton is deterministic and the recognized language is deterministic, too

NOTE – a deterministic pushdown automaton may have spontaneous moves

RELATIONS BETWEEN FAMILIES CF (free lang.) AND DET (free det. lang.)

EXAMPLE – union (nondeterministic) of deterministic languages

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

The automaton should push the letters *a*. If the string belongs to the former set (e.g., *a a b b*), then it should pop a letter *a* for each letter *b*. If the string belongs to the latter set (e.g., *a a b b b b*), then it should pop a letter *a* for each two *b*'s. Yet it cannot know which way is right, and so it has to try both ways

L' ,$L'' \in DET \Rightarrow L', L'' \in CF$

$L = L' \cup L'' \Rightarrow L \notin DET$ but $L \in CF$

therefore $DET \subseteq CF$ and $DET \neq CF$

We conclude that the family *DET* of the deterministic free languages is strictly contained in the family *CF* of all the free languages

CLOSURE PROPERTIES OF THE DETERMINSTIC LANGUAGES

We denote by *L*, *D* and *R* a language that belongs to the family *CF, DET* and *REG*, respectively

operation	*property*	(already known property)
reversal	$D^R \notin DET$	$D^R \in CF$
star	$D^* \notin DET$	$D^* \in CF$
complement	$\neg D \in DET$	$\neg L \notin CF$
union	$D_1 \cup D_2 \notin DET$ $D \cup R \in DET$	$D_1 \cup D_2 \in CF$
concatenation	$D_1 \cdot D_2 \notin DET$ $D \cdot R \in DET$	$D_1 \cdot D_2 \in CF$
intersection	$D \cap R \in DET$	$D_1 \cap D_2 \notin CF$

NOTE – the typical regular operations (*R*, * , $\cup$, $\cdot$) do not preserve determinism

SYNTAX ANALYSIS

Given a grammar G, the syntax analyzer (parser)

- reads a source string
- if the string belongs to the language *L* (*G*), then
  - exhibits a derivation
  - or builds a syntax tree

  of the string
- else stops and notifies the error, possibly with a diagnostic

If the source string is ambiguous, then the result is a forest of trees or a set of derivations

BOTTOM-UP AND TOP-DOWN ANALYSIS

A syntax tree corresponds to different derivation orders: rightmost, leftmost, …

There are two analyzer classes depending on whether the derivation is rightmost or leftmost, and on the reconstruction order of the derivation

BOTTOM-UP ANALYSIS
RIGHTMOST derivation in *INVERSE* order
TREE from leaves to root through reductions

TOP-DOWN ANALYS
LEFTMOST derivation in *DIRECT* order
TREE from root to leaves through expansions

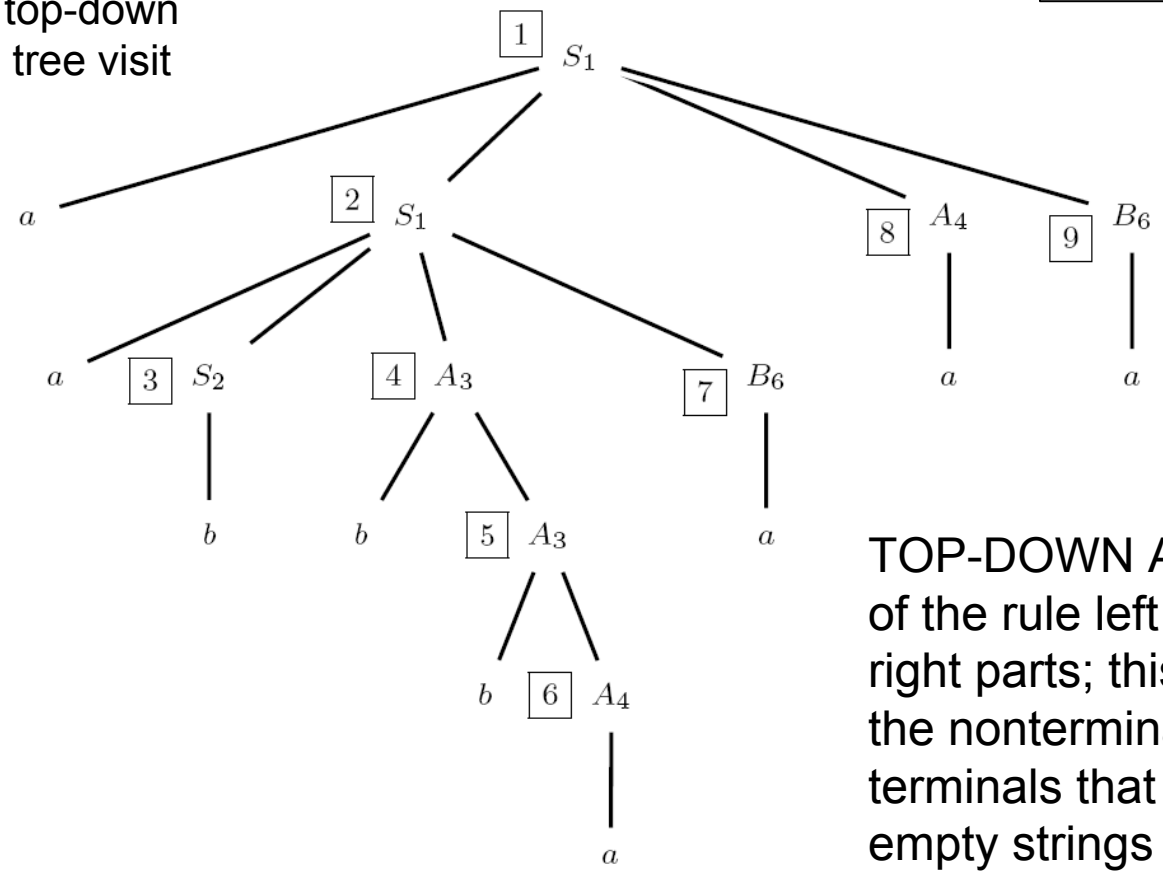
EXAMPLE – top-down analysis of the phrase

a a b b b a a a a

numbering indicates the application order of the rules

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

top-down tree visit



TOP-DOWN ANALYZER – expansion of the rule left parts into the respective right parts; this process ends when all the nonterminals are transformed into terminals that match the text or into empty strings

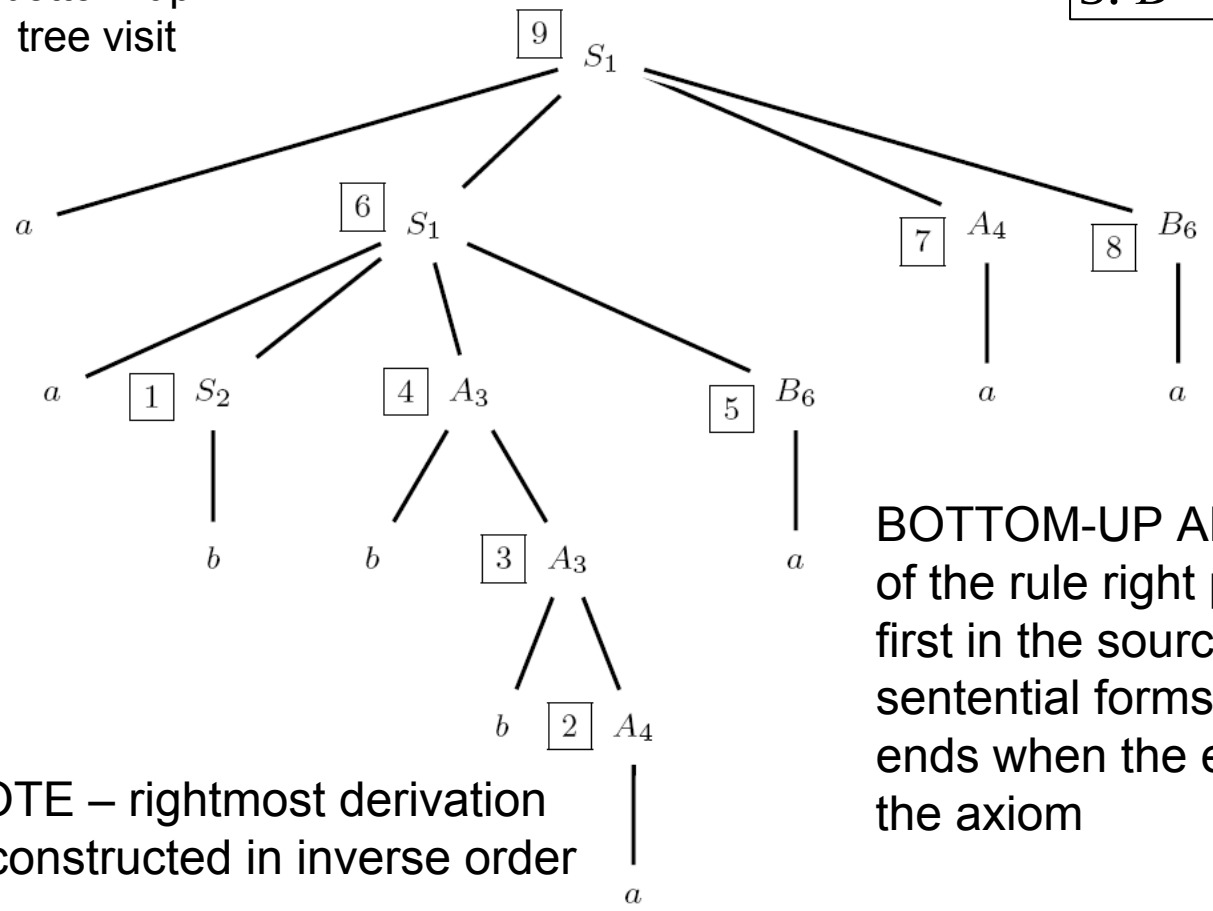
EXAMPLE – bottom-up analysis of the phrase

a a b b b a a a a

numbering indicates the application order of the rules

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

bottom-up tree visit



BOTTOM-UP ANALYSIS – reduction of the rule right parts to nonterminals, first in the source string, then in the sentential forms obtained; this process ends when the entire string reduces to the axiom

NOTE – rightmost derivation reconstructed in inverse order

EXAMPLE – derivation of the phrase *a a b b b a a a a*

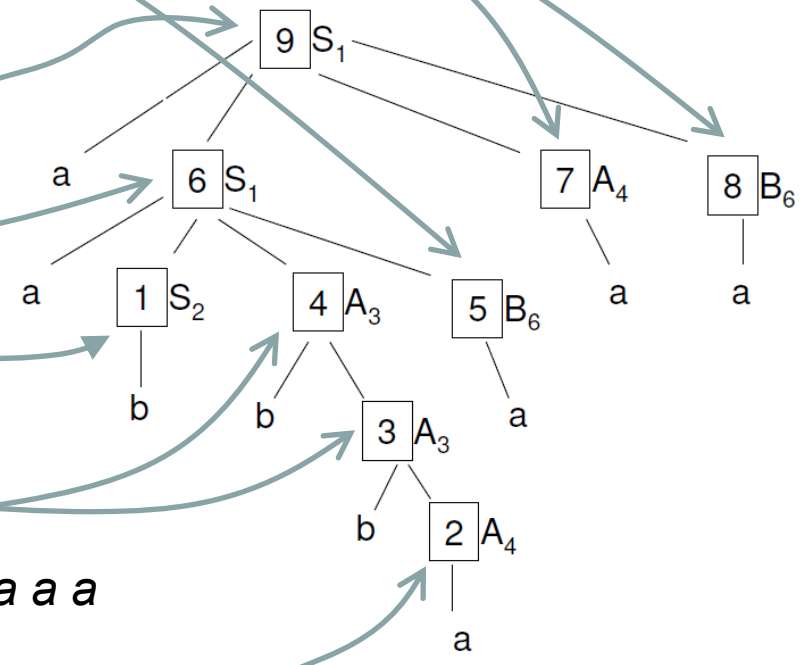
1. $S \to aSAB$	2. $S \to b$
3. $A \to bA$	4. $A \to a$
5. $B \to cB$	6. $B \to a$

$S \Rightarrow a\ S\ A\ B \Rightarrow a\ S\ A\ a \Rightarrow a\ S\ a\ a \Rightarrow a\ a\ S\ A\ B\ a\ a \Rightarrow a\ a\ S\ A\ a\ a\ a \Rightarrow$
$\Rightarrow a\ a\ S\ b\ A\ a\ a\ a \Rightarrow a\ a\ S\ b\ b\ A\ a\ a\ a \Rightarrow a\ a\ S\ b\ b\ a\ a\ a\ a \Rightarrow a\ a\ b\ b\ b\ a\ a\ a\ a$

$S \Leftarrow \mathbf{\underline{aSAB}} \Leftarrow aSA\underline{a} \Leftarrow aS\underline{a}a \Leftarrow a\mathbf{\underline{aSAB}}aa \Leftarrow$

$\Leftarrow aaSA\underline{a}aa \Leftarrow aaS\mathbf{\underline{bA}}aaa \Leftarrow$

$\Leftarrow aaSb\mathbf{\underline{bA}}aaa \Leftarrow aaSbb\underline{a}aaa \Leftarrow a\ a\ \mathbf{\underline{b}}\ b\ b\ a\ a\ a\ a$



In the bottom-up analysis the reduction operations (here >red>) transform the string part read and reduce it to a string $\alpha \in (\Sigma \cup V)^*$; this string includes all the reductions operated. Underlining indicates the string portion to reduce, called *reduction handle*

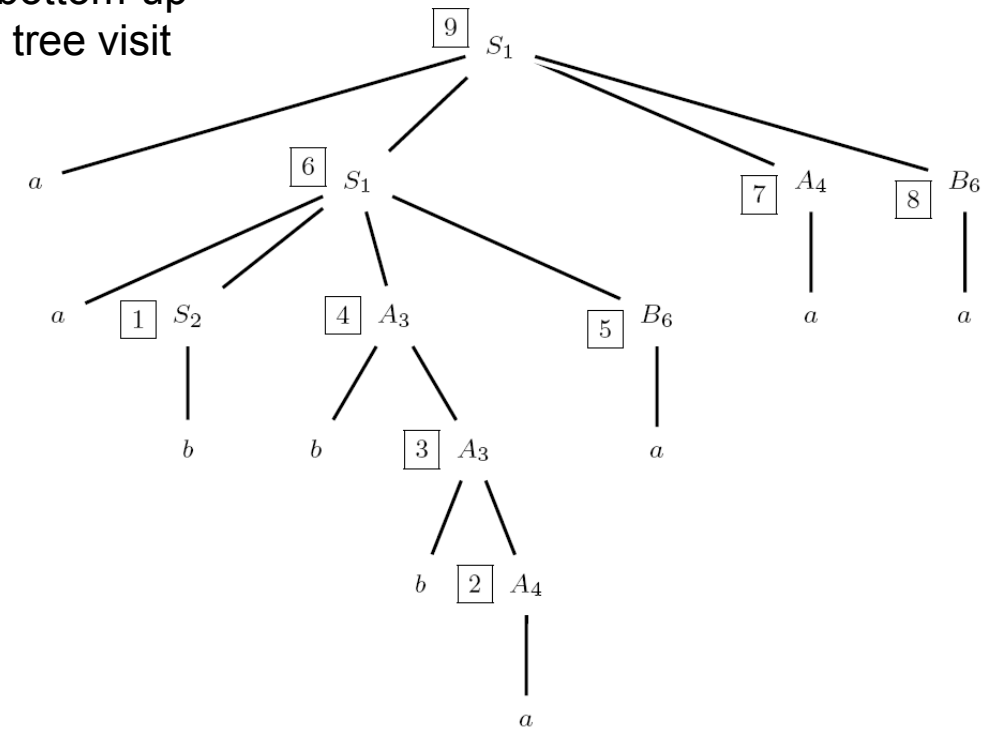
*aa**b**bbaaaa* >red> *aaSbb**a**aaa* >red> *aaSb**bA**aaa* >red> *aaS**bA**aaa* >red>

>red> *aaSA**a**aa* >red> *a**aSAB**aa* >red> *aS**a**a* >red> *aSA**a*** >red>

>red> ***aSAB*** >red> *S*

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

bottom-up tree visit



GRAMMAR AS A NETWORK OF FINITE STATE AUTOMATA

Suppose G is in extended form (*EBNF*): each nonterminal has one rule $A \to \alpha$, where α is a regular expression over terminals and nonterminals. The regexp α defines a regular language $\Rightarrow$ there is a finite automaton M_A that recognizes α

A transition of M_A labeled with a nonterminal B is interpreted as a "call" to the automaton M_B (if $B = A$ it is a recursive call)

denote - *machines* the finite automata of the nonterminals of G
- *automaton* (or *PDA*) the pushdown machine that analyzes $L(G)$
- *network* (or simply *net*) the set of all the machines of G

TERMINOLOGY

Σ terminal alphabet $V = \{ S, A, B, \ldots\}$ nonterminal alphabet

$S \to \sigma$ $A \to \alpha$ $B \to \beta$ …	grammar rules
R_S R_A R_B …	regular lang. over $\Sigma \cup V$ defined by σ α β…
M_S M_A M_B …	det. finite machines recognizing R_S R_A R_B …
$\mathcal{M} = \{ M_S, M_A, M_B, \ldots \}$	machine network

Suppose the machine states are all distinct by name

$M_A \;\; Q_A \;\; 0_A \;\; F_A$ machine A, state set, initial state, final state set

$M_B \;\; Q_B \;\; 0_B \;\; F_B$ …

…

$R\,(M_A, q)$ is the regular language $\subseteq (\Sigma \cup V)^*$ accepted by M_A starting from q, therefore $R\,(M_A, 0_A) = R_A$

For the final language, in general context free

$$L\,(M_A, q) = \{\, y \in \Sigma^* \mid \exists\, \eta \quad \eta \in R\,(M_A, q) \wedge \eta \Rightarrow^*_G y \,\}$$

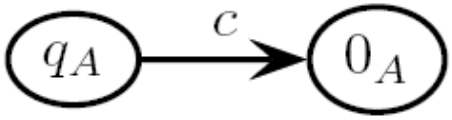
in short $L\,(q)$ if M_A is apparent from the context

therefore $L\,(M_A, 0_A) = L_A\,(\mathrm{G})$

$L\,(M_S, 0_S) = L_S\,(G) = L\,(\mathcal{M})$

Set an additional constraint to the machine of the network, so the initial state 0_A of machine *A* is never re-entered after the start of the computation

This means that no machine M_A has an arc like

M_A 

This constraint is easily fulfilled, at worst the machine needs a new initial state

$\Rightarrow$ the machine is no more minimal, but only due to one state

We say that the automata that satisfy such a condition are

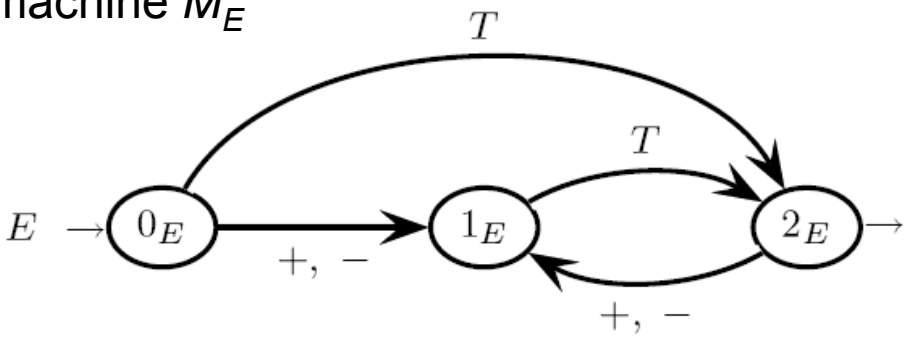
normalized or with a non-reentering initial state

NOTE – it is not forbidden that the initial state 0_A be also final and this necessarily happens whenever $\varepsilon \in L\,(M_A, 0_A)$

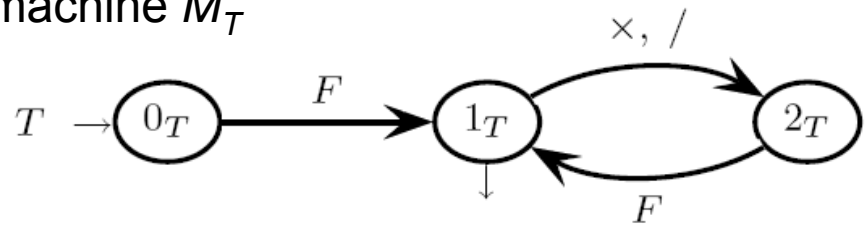
EXAMPLE – arithmetic expressions

$$\begin{cases} E \to [\,+ \mid -\,]\; T\; \big(\,(\,+ \mid -\,)\; T\,\big)^* \\ T \to F\; \big(\,(\,\times \mid /\,)\; F\,\big)^* \\ F \to a \mid \text{`(' } E \text{ `)'} \end{cases}$$

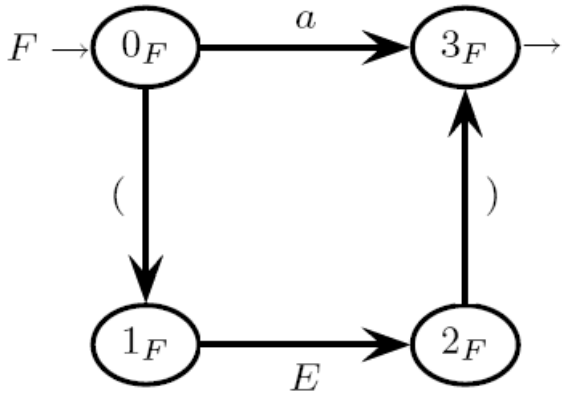
machine M_E



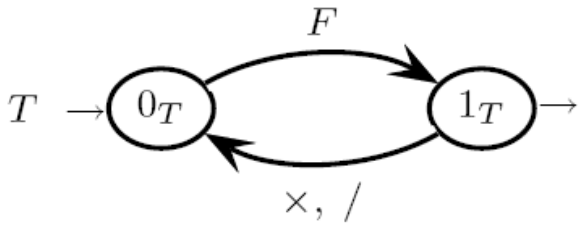
machine M_T



machine M_F

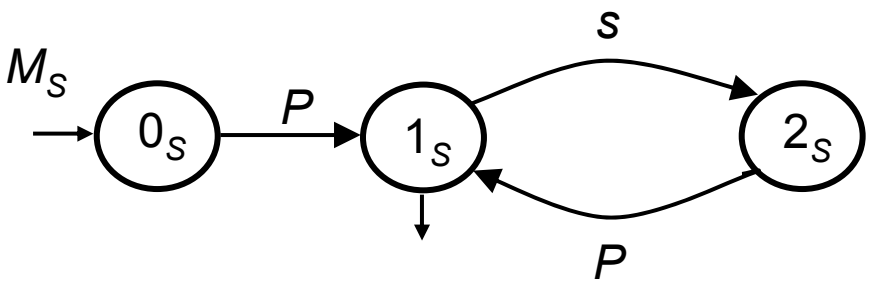


machine M_F non-normalized

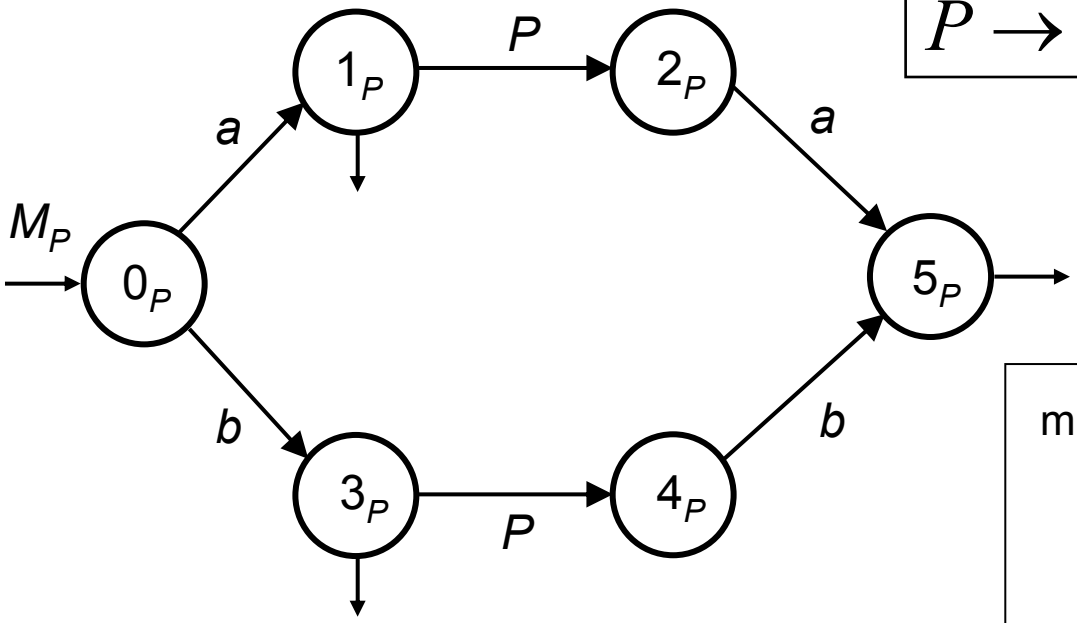


all the machines are deterministic over the grammar total alphabet (terminals and non-terminals)

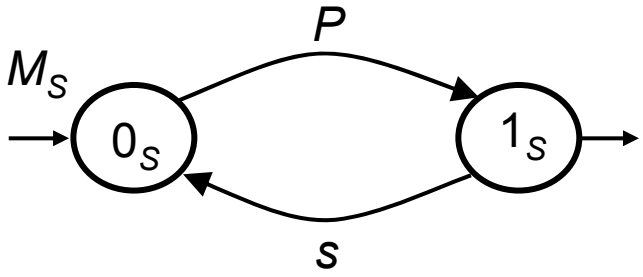
EXAMPLE – list of palindromes of odd length separated by ”*s*”



$$S \to P\,(\,s\,P\,)^*$$
$$P \to a\,P\,a \mid b\,P\,b \mid a \mid b$$



minimal M_S yet non-normalized



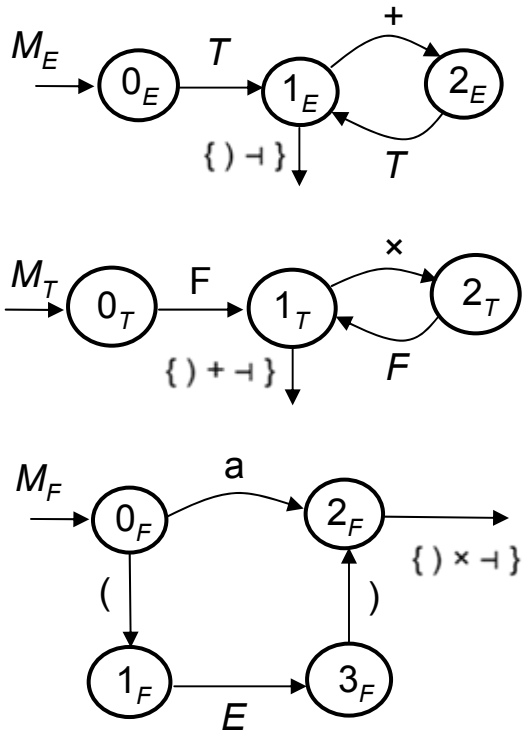
TOP-DOWN ANALYSIS BY MEANS OF RECURSIVE PROCEDURES

Simple and elegant: the procedure code reflects the machine transitions

$E \rightarrow T\,(+\,T)^*$
$T \rightarrow F\,(\times\,F)^*$
$F \rightarrow a \mid (\,E\,)$

EXAMPLE – arithmetic expressions

When the finite automaton has a bifurcation state, to decide which direction to take we must watch all the symbols that appear on the arcs that leave the state, included the final darts of the machine



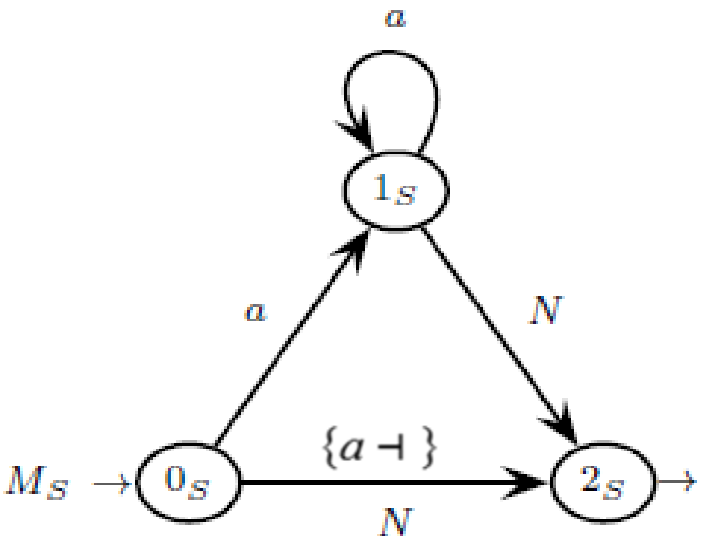

```
procedure E
 call T;
 while(cc=+)
  cc:=next;
  call T;
 end;
 if(cc∈{) ⊣})
   return:
  else error;
 endif
end E;
```

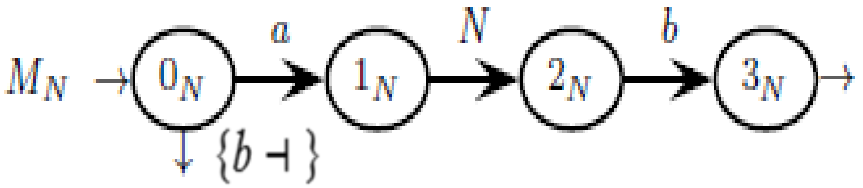
```
procedure T
 call F;
 while(cc=×)
  cc:=next;
  call F;
 end;
 if(cc∈{)+ ⊣})
   return:
  else error;
 endif
end T;
```

```
procedure F
 if(cc=a)
   cc=next;
  elsif(cc=()
   cc:=next;
   call E;
   if(cc=))
     cc:=next;
     if(cc∈{)× ⊣})
        return;
      else error;
     endif;
    else error;
   endif;
  else error;
 endif;
end F;
```

NOTE – top-down does not work when the symbol sets on the arcs that leave the same state (the so-called *guide sets*) are not disjoint

EXAMPLE $S \rightarrow a^* N$ $N \rightarrow a\ N\ b \mid \varepsilon$ $L = \{ a^n b^m \mid n \geq m \geq 0 \}$



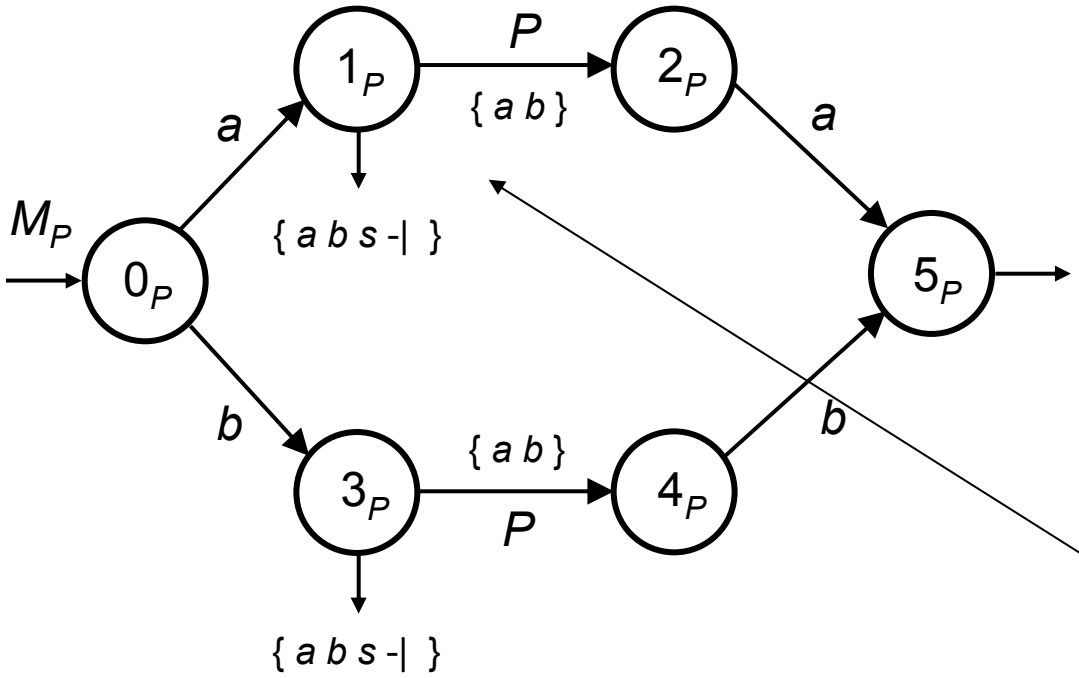



```
procedure S
   if (cc = 'a')
      call N ?
      cc := next ?
   ...
end S
```

ONE MORE EXAMPLE – list of palindromes of odd length separated by ”*s*”

NOTE – it is a non-deterministic language, bottom-up does not work either

$$S \rightarrow P\,(\,s\,P\,)^*$$
$$P \rightarrow a\,P\,a \mid b\,P\,b \mid a \mid b$$




```
procedure P
 if (cc == 'a')
    cc := next
    if (cc ∈ {a b})
       call P ?
       return ?
    ...
end P
```

this recursive procedure is unable to decide what to do by watching the next symbol expected