

Formal Languages & Compilers

Giovanni Agosta

September 12, 2022

Learning Goals

- Understand the theoretical underpinnings of formal languages
- Acquire confidence with the theoretical tools to manipulate formal languages
- Understand how the front-end of a compiler is built
- Acquire confidence with some of the tools employed for compiler construction

The team

Instructor Prof. Giovanni Agosta (giovanni.agosta)

Teaching Assistant ing. Gabriele Magnani (gabriele.magnani)

The virtual classrooms

- <https://politecnicomilano.webex.com/meet/name.surname>
- Under standard conditions, used only for providing recordings

Artificial vs Natural Languages

- Target: machines vs humans
- Mode: non-verbal vs verbal
- Structure: formal vs non-formal

Formal Language

- Definitions:
 - a language is *formal* if its *syntax* (structure) and *semantic* (interpretation) are defined in a precise *algorithmic* way
 - it is possible to define effective procedures to
 - analyze the syntactic structure and check the grammatical correctness of the phrases
 - compute its meaning (for FLC semantics \equiv translation into a different language)
- The theory of formal languages:
 - studies the form, or syntax of the phrases
 - defines analysis methods and algorithms
 - using both *grammars* (axiomatic rules) and *automata* (mathematical tools)

Compilers

- applying theory of formal languages to the compilation (and, at large, to the automatic processing)
 - of programming languages
 - in general, to all languages of informatics and to "technical languages" es: JSON

Historical notes

- 1950 IBM FORmula TRANslator; Noam Chomsky:
mathematical model of grammars in connection with
the study of natural languages
- 1960 ALGOL, COBOL, Lisp
- syntax-directed compilation, meta-compilers
 - deep link between formal language theory and
automata theory
 - grammars: regular, context free,
context-sensitive
- 1970 Pascal, Intel PL/M, C; front-end technology maturity
- 1980 ADA; automated code generation and parallelisation
- 1990 Object oriented languages, portable/retargetable
compilers

Historical notes

- 1995 Java; dynamic compilation
- 2000 Compilers for embedded parallel architectures
- 2005 DotNet, LLVM; first compilers for FPGAs
- 2010 PGAS, OpenCL; attempts to auto-parallelise and auto-optimise programs
- 2015+ Languages for heterogeneous parallelism; use of formal languages and static analysis techniques for performance and other extra-functional concerns

Course Program

Syntax

- Formal Language Theory
- Regular Expressions & Languages
- Context-Free Grammars
- Grammars for Regular Languages

Finite Automata & Language Recognition

- Deterministic Finite Automata
- Nondeterministic Finite Automata
- Relations between Languages and Finite Automata

Phrase Recognition & Parsing

- Recognizing Context-Free Languages with Pushdown Automata
- Deterministic Syntax Analysis (bottom-up/top-down)

Semantic Translation & Static Analysis

- Translation relation and function
- Transliteration
- Regular translations
- Purely syntactic translations
- Semantic translations (attribute grammars)
- Static analysis of programs (based on FSA and dataflow equations)

Laboratory

Some practice for the theory, plus laboratory/design lessons

Compiler Design Tools

- Free & Open Source (GPL) tools for compiler generation
- Lexical analysis: scanner generation with **flex**
- Syntax analysis: parser generation with **bison**

Logistics

- In class
- Instructors:
 - Gabriele Magnani

Books

- S. Crespi Reghizzi, L. Breveglieri, A. Morzenti, Formal Languages and Compilation, Springer Verlag, 2nd (2013) or 3rd (2019) edition
- S. Crespi Reghizzi, L. Breveglieri, A. Morzenti, Linguaggi Formali e Compilazione, Esculapio, 2nd (2015) edition (*Italian*)

On Webeep

- Slides
- Exams with solutions

- Written exam only
- Open-book (hint → mere memorization of content is not sufficient...)
- Two parts that can be taken separately:
 - Practice: 1h, 20% of the score
 - Theory: 2h:15m, 80% of the score
- *Both scores need to be $\geq 15!$*
- The individual scores remain valid for 5 calls (1 solar year)

Why Compilers?

Common wisdom

“Compilers are a commodity”

Not really, compiler are continuously developed!

- *Full employment theorem* for compiler writers --> cannot build a compiler which is optimal for every use
- More importantly, any technology has a life cycle
 - Infancy: focus on key aspect, leanness, single purpose
 - Maturity: broaden focus, completeness, target as many goals as possible
 - Senescence: loss of focus, bloat, cost of maintainance
 - Death: developing a new tool becomes more cost-effective (loop back...)
- In the 2000s, GCC reached sensescence, and LLVM was in infancy
- Now, LLVM enters maturity, GCC lingers but is not anymore the industry standard!

Compiler Technologists are a scarce resource!

- ~1.5 compiler engineers for every 1000 software engineers
- ~2 compiler jobs for every 100 software engineering jobs

Data from my network on LinkedIn, 2012

Why? Compiler construction is highly specialised, few universities have dedicated courses!

Typical Employers for Compiler Technologists

Semiconductor Industry all major semiconductor companies have their own compiler teams; PoliMi Alumni at (e.g.) STMicroelectronics (IT/FR), ARM (UK), Sony (UK)

Big Tech compiler development is driven by tech giants; PoliMi Alumni at Apple, Google

Compiler Development SMEs EU industry is SME-driven; PoliMi Alumni at Codeplay (UK), rev.ng (IT, startup from former PoliMi students)

Other Areas Electronic Design Automation; Security; Software Intelligence

Today

We start with the key concepts in formal languages and regular expressions

Formal Language Theory

an Introduction

Prof. A. Morzenti

ALPHABET Σ : any ***finite*** set of symbols $\Sigma = \{a_1, a_2, \dots, a_k\}$

cardinality of the alphabet $|\Sigma| = k$

String: a sequence (\Rightarrow ordered) of alphabet elements (possibly repeated)

Language: any set of strings

$$\Sigma = \{a, b, c\} \quad L_1 = \{ab, ac\} \quad L_2 = \{bc, bbc\} \quad L_3 = \{abc, aabbcc, aaabbbccc, \dots\}$$

The strings of a language are called its ***sentences*** or ***phrases***

Language ***cardinality***: the number of its sentences

$$|L_2| = |\{bc, bbc\}| = 2 \quad |\emptyset| = 0$$

Number of occurrences of a symbol in a string $|bbc|_b = 2$, $|bbc|_a = 0$

With a slight *abuse of notation* sometimes we denote with Σ
both the alphabet and
the language of all strings of length 1

length of a string x : $|x|$
number of its elements

$$\begin{aligned} |bbc| &= 3 \\ |abbc| &= 4 \end{aligned}$$

string equality : two strings are equal if and only if (*iff*, for short)

- have the same length
- their elements, from left to right, coincide

$$x = a_1 a_2 \dots a_h \quad y = b_1 b_2 \dots b_k$$

$$x = y \text{ iff } h = k \text{ and } a_i = b_i \text{ for all } i = 1 \dots h$$

$$bbc \neq bcb \neq bc$$

OPERATIONS ON STRINGS /1

CONCATENATION (product): $x \cdot y$ or xy for short

$$x = a_1 a_2 \dots a_h \quad y = b_1 b_2 \dots b_k \quad xy = xy = a_1 a_2 \dots a_h b_1 b_2 \dots b_k$$

- associative $(xy)z = x(yz)$
- length $|xy| = |x| + |y|$

EMPTY STRING (or *null string*) ε is the **neutral element for concatenation**

for any x , $x\varepsilon = \varepsilon x = x$

length of ε : $|\varepsilon| = 0$

NOTICE: ε is **NOT** the empty set: $\varepsilon \neq \emptyset$

SUBSTRINGS: if $x=uyv$ (NB: both u and v can be ε) then

- y is a substring of x
- y is a ***proper substring*** iff $u \neq \varepsilon$ or $v \neq \varepsilon$
- u is a ***prefix*** of x
- v is a ***suffix*** of x

EXAMPLES

if $x = abccbc$ then

prefixes: $a, ab, abc, abcc, abccb, abccbc$

suffixes: $c, bc, cbc, ccbc, bccbc, abccbc$

substrings: $\dots, bc, cc, cb, abc, bcc, \dots$

OPERATIONS ON STRINGS /2

REFLECTION x^R

$$\begin{aligned}x &= a_1a_2\dots a_h \\x^R &= a_ha_{h-1}\dots a_2a_1 \\(x^R)^R &= x \\(xy)^R &= y^Rx^R \\\varepsilon^R &= \varepsilon\end{aligned}$$

$$\begin{aligned}x &= atri & x^R &= irta \\x &= bon & y &= ton \\xy &= bonton \\(xy)^R &= y^Rx^R = notnob\end{aligned}$$

REPETITION: m -th power ($m \geq 1$) of string x : concatenation of x with itself $m-1$ times

$$\begin{aligned}x^m &= \underset{1 2 3 \dots m}{xxx\dots x} \\(\text{inductive}) \quad &\left\{ \begin{array}{l} x^m = x^{m-1}x, \quad m > 0 \\ (\text{definition}) \quad \left\{ \begin{array}{l} x^0 = \varepsilon \end{array} \right. \end{array} \right.\end{aligned}$$

$$\begin{aligned}x &= ab & x^0 &= \varepsilon & x^1 &= x = ab & x^2 &= (ab)^2 = abab \\y &= a^3 = aaa & y^3 &= a^3a^3a^3 = a^9 \\x^0 &= \varepsilon & x^2 &= \varepsilon\end{aligned}$$

OPERATOR PRECEDENCE: repetition and reflection take precedence over concatenation

$$\begin{aligned}ab^2 &= abb & (ab)^2 &= abab \\ab^R &= ab & (ab)^R &= ba\end{aligned}$$

OPERATIONS ON LANGUAGES /1

OPERATIONS ARE TYPICALLY DEFINED ON A LANGUAGE
BY EXTENDING THE STRING OPERATION TO ALL ITS PHRASES

REFLECTION L^R : $L^R = \{ x \mid \exists y (y \in L \wedge x = y^R) \}$ def. by **characteristic predicate**

$\text{Prefixes}(L) = \{ y \mid y \neq \epsilon \wedge \exists x \exists z (x \in L \wedge x = yz \wedge z \neq \epsilon) \}$ NB: *proper* prefixes

set of all possible prefixes of the language L

Prefix-free language L : no proper prefix of its phrases $\in L$: $\text{Prefixes}(L) \cap L = \emptyset$

EXAMPLE: $L_1 = \{ x \mid x = a^n b^n \wedge n \geq 1 \}$ is prefix-free: $a^2 b^2 \in L_1$ $a^2 b \notin L_1$

EXAMPLE: $L_2 = \{ x \mid x = a^m b^n \wedge m > n \geq 1 \}$ is not prefix-free: $a^4 b^3 \in L_2$ $a^4 b^2 \in L_2$

OPERATIONS ON LANGUAGES / 2

Operations defined over two arguments

CONCATENATION

$$L' L'' = \{xy \mid x \in L' \wedge y \in L''\}$$

Concatenation of every string of the two languages

m -th POWER
(inductive definition)

$$L^m = L^{m-1}L, m > 0$$

$$L^0 = \{\varepsilon\}$$

NB: $\{\varepsilon\} \neq \emptyset$

NB: consequences

$$\emptyset^0 = \{\varepsilon\} \quad L.\emptyset = \emptyset.L = \emptyset \quad L.\{\varepsilon\} = \{\varepsilon\}.L = L$$

OPERATIONS ON LANGUAGES / 3

EXAMPLES

$$L_1 = \{ a^i \mid i \geq 0, i \text{ even} \} = \{ \varepsilon, a^2, a^4, \dots \}$$

$$L_2 = \{ b^j a \mid j \geq 1, j \text{ odd} \} = \{ ba, b^3 a, b^5 a, \dots \}$$

$$L_1 L_2 = \{ a^i b^j a \mid (i \geq 0, i \text{ even}) \wedge (j \geq 1, j \text{ odd}) \} =$$

$$= \{ \varepsilon ba, a^2 ba, a^4 ba, \dots, \varepsilon b^3 a, a^2 b^3 a, \dots \}$$

$$\begin{aligned} (L_1)^2 &= \{ \varepsilon, a^2, a^4, a^6, \dots \} \{ \varepsilon, a^2, a^4, a^6, \dots \} = \\ &= \{ \varepsilon, \varepsilon a^2, \varepsilon a^4, \dots, a^2 \varepsilon, a^4, \dots, a^4 \varepsilon, a^6 \dots \} = L_1 \end{aligned}$$

for each pair of even numbers
 h and k , $h+k$ is even, hence
 $a^{h+k} \in L_1$

PAY ATTENTION: the language L^m
 in general does **not** contain **only**
 phrases of L repeated m times

$$\{ x \mid x = y^m \wedge y \in L \} \subset L^m$$

$$m = 2 \quad L_1 = \{ a, b \}$$

$$\{ a^2, b^2 \} \subset L_1^2 = \{ a^2, ab, ba, b^2 \}$$

OPERATIONS ON LANGUAGES / 4

Finite length strings:

The power operator allows one to define concisely the language of strings whose length is not greater than a given integer K

Alfabeto = {a,b}

$$L = \{\varepsilon, a, b\}^3 \quad K = 3$$

$$L = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, ...bbb\}$$

Notice the role of ε
It allows one to obtain
all strings of length $< K$ (0, 1, 2)

N.B. Le potenze m-esime dei linguaggi senza stringa vuota non comprendono le stringhe delle potenze dei linguaggi di potenza $n < m$, invece se c'è la epsilon le contiene (Infatti nell'esempio sopra, la potenza 3-esima contiene anche "a", "b", proprio perchè il linguaggio ha la stringa vuota)

To rule out the empty string:

$$L = \{a, b\} \{\varepsilon, a, b\}^2$$

OPERATIONS ON LANGUAGES / 5

SET THEORETIC OPERATIONS: the customary ones are defined:
union, intersection, difference, inclusion, strict inclusion, equality

$$\cup \quad \cap \quad \setminus \quad \subseteq \quad \subset \quad =$$

UNIVERSAL LANGUAGE: the set of all
strings over the alphabet Σ ,
of any length, including 0 (i.e., string ε)

$$L_{universal} = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \dots$$

COMPLEMENT of a language L over alphabet Σ
is the set difference with respect to (w.r.t.) the
universal language (i.e., the set of strings over Σ that $\notin L$)

$$\neg L = L_{universal} \setminus L$$

hence $L_{universal} = \neg \emptyset$

OPERATIONS ON LANGUAGE / 6

EXAMPLES

The complement of a *finite* language
is *always infinite*

$$\neg(\{a,b\}^2) = \varepsilon \cup \{a,b\} \cup \{a,b\}^3 \cup \dots$$

The complement of an *infinite* one
is *not necessarily finite*

$$L = \{a^{2n} \mid n \geq 0\} \quad \neg L = \{a^{2n+1} \mid n \geq 0\}$$

Examples of the difference operation among languages

$$\Sigma = \{a, b, c\}$$

$$L_1 = \{x \mid |x|_a = |x|_b = |x|_c \geq 0\}$$

$$L_2 = \{x \mid |x|_a = |x|_b \wedge |x|_c = 1\}$$

$$L_1 \setminus L_2 = \varepsilon \cup \{x \mid |x|_a = |x|_b = |x|_c \geq 2\}$$

(same number of a, b, c , but not $=1$)

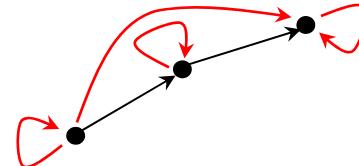
$$L_2 \setminus L_1 = \{x \mid |x|_a = |x|_b \neq |x|_c = 1\}$$

(same number of a and b ; one c ;
but $\neg(|x|_a = |x|_b = |x|_c)$, hence $|x|_a = |x|_b \neq 1$)

A frequently used algebraic operation: reflexive and transitive closure R^* of a relation R

Given a set A and a relation $R \subseteq A \times A$, $(a_1, a_2) \in R$ is also denoted as $a_1 R a_2$

R^* is a *relation* defined by:



- $x R^* x \quad \forall x \in A$, (reflexive) and
- $x_1 R x_2 \wedge x_2 R x_3 \wedge \dots x_{n-1} R x_n \Rightarrow x_1 R^* x_n$ (transitive)

If we see $a R b$ as *a step* in relation R , $x R^* y$ seen as

a chain of $n \geq 0$ steps

Se ho un numero $n \geq 0$ di passi, ottengo la chiusura riflessiva (con $n=0$) e transitiva (con $n>0$)

Example: if $R = \{(a, b), (b, c)\}$ then

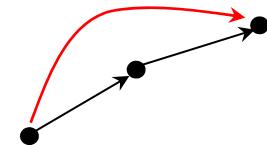
$$R^* = \{\underline{(a, a)}, \underline{(b, b)}, \underline{(c, c)}, (a, b), (b, c), \underline{(a, c)}\}$$

A variant: transitive closure R^+ of a relation R

Similarly, ***transitive*** (non reflexive) ***closure*** R^+ :
k-th power R^k with $n \geq 1$

Se ho solo un numero $n \geq 1$ di passi, ottengo solo la chiusura transitiva

$$x_1 R x_2 \wedge x_2 R x_3 \wedge \dots x_{n-1} R x_n \Rightarrow x_1 R^* x_n$$



Example: if relation R is the ***adjacency*** relation on a graph
 R^+ is the ***reachability in one or more steps***

Example: if $R = \{(a, b), (b, c)\}$ then

$$R^{*(?)} = \{ (a, b), (b, c), \underline{(a, c)} \}$$

Similarly, ***closure*** A of a ***set*** A under an ***operation*** (function)
is obtained from A by adding all elements obtained
by applying the operation any number of times

OPERATIONS ON LANGUAGES / 7

STAR OPERATOR: reflexive transitive closure under the concatenation operation

(also called **Kleene star**)

$$L^* = \bigcup_{h=0...∞} L^h = L^0 \cup L^1 \cup L^2 \dots = \varepsilon \cup L^1 \cup L^2 \dots$$

$$L = \{ab, ba\} \quad L^* = \{\varepsilon, ab, ba, abab, abba, baab, baba, \dots\}$$

(L is finite L^* is infinite)

It is the union of all the powers of the language

Every string of the star language L^* can be chopped into substrings $\in L$

The star language L^* can be equal to the base language L

$$L = \{a^{2n} \mid n \geq 0\} \quad L^* = \{a^{2n} \mid n \geq 0\} \equiv L$$

OPERATIONS ON LANGUAGES / 8

If we take Σ as the base language, then Σ^* contains all the strings built on that alphabet
(it is the *universal language* of alphabet Σ)

We often say that L is a language on alphabet Σ by writing $L \subseteq \Sigma^*$

PROPERTIES OF THE STAR OPERATOR

- monotonicity (with * the set increases): $L \subseteq L^*$
- closure under concatenation: if $x \in L^*$ and $y \in L^*$ then $xy \in L^*$
- idempotence: $(L^*)^* = L^*$
- commutativity of star and reflection $(L^*)^R = (L^R)^*$

Furthermore: $\emptyset^* = \{ \epsilon \}$ $\{ \epsilon \}^* = \{ \epsilon \}$ NB: these are cases where L^* is finite

Example of idempotence: We already noticed that, for $L = \{ a^{2n} \mid n \geq 0 \}$, it holds $L^* = L$

This derives from idempotence, because we have $L = L_0^*$ for $L_0 = \{ aa \} = \{ a^2 \}$

OPERATIONS ON LANGUAGES / 9

Example on the STAR OPERATOR

language of identifiers I as character strings that start with a letter and include any number of letters and digits

$$\Sigma_A = \{ a, b, \dots, z, A, B, \dots, Z \} \quad \Sigma_N = \{ 0, 1, 2, \dots, 9 \}$$

$$I = \Sigma_A (\Sigma_A \cup \Sigma_N)^*$$

if we stipulate $\Sigma = \Sigma_A \cup \Sigma_N$

language I_5 of identifiers of maximal length 5

$$I_5 = \Sigma_A (\Sigma \cup \{ \varepsilon \})^4$$

OPERATIONS ON LANGUAGES / 10

CROSS OPERATOR \mathbf{L}^+ : transitive closure (non reflexive) under concatenation

The union does *not* include the first power \mathbf{L}^0

Useful but not indispensable, it can be derived from the star operator *:

$$\mathbf{L}^+ = \mathbf{L} \cdot \mathbf{L}^*$$

$$\mathbf{L}^+ = \bigcup_{h=1 \dots \infty} \mathbf{L}^h = \mathbf{L}^1 \cup \mathbf{L}^2 \cup \dots$$

$$\{ab, bb\}^+ = \{ab, bb, ab^3, b^2ab, abab, b^4, \dots\}$$

$$\{\varepsilon, aa\}^+ = \{\varepsilon, a^2, a^4, \dots\} = \{a^{2n} \mid n \geq 0\}$$

if $\varepsilon \in \mathbf{L}$ then $\mathbf{L}^+ = \mathbf{L}^*$

Typically, a given language can be defined in different ways using different operators

Example: language L of strings of length ≥ 4 : $\mathbf{L} = \Sigma^4 \Sigma^*$ and also $\mathbf{L} = (\Sigma^+)^4$

OPERATIONS ON LANGUAGES / 11

QUOTIENT OPERATOR L_1 / L_2 : it shortens the phrases of L_1 by cutting off a suffix that belongs to L_2 . NB: **forward** slash (backward slash denotes set difference)

$$L = L_1 / L_2 = \{ y \mid \exists x \in L_1 \exists z \in L_2 (x = yz) \}$$

Example: $L_1 = \{a^{2n} b^{2n} \mid n > 0\}$ $L_2 = \{b^{2n+1} \mid n \geq 0\}$

$$\begin{aligned} L_1 / L_2 &= \{a^r b^s \mid (r \geq 2, \quad r \text{ even}) \wedge (1 \leq s < r, \quad s \text{ odd})\} \\ &= \{a^2 b, a^4 b, a^4 b^3, \dots\} \end{aligned}$$

$$L_2 / L_1 = \emptyset \quad \text{because no string in } L_2 \text{ has a string in } L_1 \text{ as a suffix}$$

Regular Expressions and Languages

Prof. A. Morzenti

The family of REGULAR LANGUAGES is our simplest formal language family
It can be defined in three ways:

- Algebraically (we start from this)
- By means of generative grammars
- By means of recognizer automata

a ***regular expression*** (r.e.): an expression on languages that composes languages operations

It is a string r

over the alphabet $\Sigma = \{a_1, a_2, \dots, a_k\}$ and the metasymbols

\emptyset (empty language), \cup (union), \cdot (concatenation), $*$ (star)

according to the following rules (where s and t are regular expressions):

1. $r = \emptyset$
2. $r = a, a \in \Sigma$
3. $r = (s \cup t)$ or $r = s | t$ (alternative notation)
4. $r = (s \cdot t)$ or $r = (s t)$
5. $r = (s)^*$

OPERATOR PRECEDENCE : star ‘*’, concatenation ‘.’, union ‘ \cup ’

Frequently used derived operators:

$$\begin{array}{ll} \varepsilon \text{ defined by} & \varepsilon = \emptyset^* \\ e^+ \text{ defined by} & e \cdot e^* \end{array}$$

The ***meaning*** of a r.e. r is a ***language*** L_r of alphabet Σ according to the table

expression r	language L_r
\emptyset	\emptyset
ε	$\{ \varepsilon \}$
$a \in \Sigma$	$\{ a \}$
$s \cup t$ or $s t$	$L_s \cup L_t$
$s \cdot t$ or $s\ t$	$L_s \cdot L_t$
s^*	L_s^*

a ***regular language*** is a language denoted by a regular expression

Example: language that consists of sequences of ‘1’ of length multiple of three

$$e = (111)^*$$

$$L_e = \{ \varepsilon, 111, 111111, \dots \} = \{ 1^n \mid n \bmod 3 = 0 \}$$

$$e_1 = 11(1)^* \quad \text{NB: } L_{e_1} \neq L_e$$

$$L_{e_1} = \{ 11, 111, 1111, 11111, \dots \} = \{ 111^n \mid n \geq 0 \}$$

Example: let $\Sigma = \{ +, -, \epsilon \}$ with d denoting the decimal digits $0, 1, \dots, 9$

Let us define the r.e. defining the language of integer numbers with or without sign

$$e = (+ \cup - \cup \epsilon) dd^*$$

$$L_e = \{+, -, \epsilon\} \{d\} \{d\}^*$$

Example: The language of alphabet $\{a, b\}$ such that
in any phrase the number of characters a is odd and there is at least one b

let us use two auxiliary r.e. : A_E strings with $\#a$ even, A_O , $\#a$ odd

$$e = A_E b A_O \mid A_O b A_E, \text{ where}$$

$$A_E = b^* (ab^* ab^*)^* \quad A_O = b^* ab^* (ab^* ab^*)^*$$

THE FAMILY OF REGULAR LANGUAGES (***REG***)

It is the collection of all regular languages

THE FAMILY OF FINITE LANGUAGES (***FIN***)

It is the collection of all languages having a finite cardinality

EVERY FINITE LANGUAGE IS REGULAR (hence $FIN \subseteq REG$)
because it is the union of a finite number of strings
each one being the concatenation of a finite number of alphabet symbols

$$(x_1 \cup x_2 \cup \dots \cup x_k) = (a_{1_1} a_{1_2} \dots a_{1_n} \cup \dots \cup a_{k_1} a_{k_2} \dots a_{k_m})$$

The family of regular languages also includes languages having infinite cardinality

hence inclusion is strict: $FIN \subset REG$

HOW CAN ONE DERIVE FROM A R.E. THE SENTENCES OF ITS LANGUAGE?

LET US INTRODUCE THE NOTION OF **CHOICE**:

The union and repetition operators correspond to possible choices

One obtains a subexpression by making a choice that identifies a sublanguage

expression r	choice of r
$e_1 \cup \dots \cup e_n$ or $e_1 \dots e_n$	e_k for every $1 \leq k \leq n$
e^*	ϵ or e^n for every $n \geq 1$
e^+	e^n for every $n \geq 1$

Given a r.e. one can *derive* another one

by replacing any «outermost» (or «top level») subexpression with another that is a choice of it

DERIVATION RELATION among two r.e. e' and e'' : $e' \Rightarrow e''$

$e' \Rightarrow e''$ if the two r.e. can be factorized as

$$e' = a\beta\gamma \quad e'' = a\delta\gamma$$

where δ is a choice of β

NB: the definition implies that the operator (either $|$, $*$, or $^+$) of which a choice is made is «outermost»
(see remark on next slide)

the derivation relation can be applied repeatedly, yielding relation $\stackrel{n}{\Rightarrow}, \stackrel{+}{\Rightarrow}, \stackrel{*}{\Rightarrow}$
(resp. *power*, *transitive closure* and *reflexive transitive closure* of the relation)

$e_0 \stackrel{n}{\Rightarrow} e_n$ iff $e_0 \Rightarrow e_1, e_1 \Rightarrow e_2, \dots, e_{n-1} \Rightarrow e_n$
 e_0 derives e_n (or e_n is derived from e_0) in n steps

$e_0 \stackrel{+}{\Rightarrow} e_n$ e_0 derives e_n in $n \geq 1$ steps

$e_0 \stackrel{*}{\Rightarrow} e_n$ e_0 derives e_n in $n \geq 0$ steps

Examples

Some immediate and multi-step derivations:

$$a^* \cup b^+ \Rightarrow a^*, \quad a^* \cup b^+ \Rightarrow b^+$$

$$a^* \cup b^+ \Rightarrow a^* \Rightarrow \varepsilon \quad \text{that is, } a^* \cup b^+ \xrightarrow{2} \varepsilon \quad \text{or} \quad a^* \cup b^+ \xrightarrow{+} \varepsilon$$

$$a^* \cup b^+ \Rightarrow b^+ \Rightarrow bbb \quad \text{that is, } a^* \cup b^+ \xrightarrow{2} bbb \quad \text{or} \quad a^* \cup b^+ \xrightarrow{+} bbb$$

Some of the derived r.e. include metasymbols (operators and parentheses)

other ones only symbols of Σ (also known as *terminal symbols* or *terminals*) and ε

These constitute the *language defined by the r.e.*

An alternative definition of the language of a r.e.
similar to the one we will use for grammars

$$L(r) = \left\{ x \in \Sigma^* \mid r \xrightarrow{*} x \right\}$$

NB : in derivations, operators must be chosen from external to internal

otherwise a *premature choice* would rule out valid sentences

e.g., $(a^* \mid bb)^* \Rightarrow (a^2 \mid bb)^*$ prevents subsequent derivation of sentence a^2bba^3
(see remark on previous slide)

Further examples:

$$1.(ab)^* \Rightarrow abab$$

$$2.(ab \cup c) \Rightarrow ab$$

$$3.a(ba \cup c)^*d \Rightarrow ad$$

$$4.a(ba \cup c)^*d \Rightarrow a(ba \cup c)(ba \cup c)d$$

$$5.a^*(b \cup c \cup d)f^+ \Rightarrow aaa(b \cup c \cup d)f^+$$

$$6.a^*(b \cup c \cup d)f^+ \Rightarrow a^*cf^+$$

$$7.a^*(b \cup c \cup d)f^+ \xrightarrow{+} aaacf^+ \text{ in 2 steps}$$

$$8.a^*(b \cup c \cup d)f^+ \xrightarrow{+} aaacff \text{ in 3 steps}$$

Two r.e. are ***equivalent*** if they define the same language

a phrase of a regular language can be obtained through distinct equivalent derivations

these can ***differ in the order*** of the choices used in the derivation

$$a(ba \cup c)^*d \Rightarrow a(ba \cup c)(ba \cup c)d \Rightarrow ac(ba \cup c)d \Rightarrow acbad$$

$$a(ba \cup c)^*d \Rightarrow a(ba \cup c)(ba \cup c)d \Rightarrow a(ba \cup c)bad \Rightarrow acbad$$

in order to define

AMBIGUITY OF REGULAR EXPRESSIONS

LET US DEFINE THE (numbered) **SUBEXPRESSION** OF A R.E.

1. Consider a r.e. and add all possible parentheses
2. Derive a (*numbered*) version e_N of the r.e. e
3. Identify all the (numbered) subexpressions

$$e = (a \cup (bb))^*(c^+ \cup (a \cup (bb)))$$

$$e_N = (a_1 \cup (b_2 b_3))^*(c_4^+ \cup (a_5 \cup (b_6 b_7)))$$

$$(a_1 \cup (b_2 b_3))^* \quad c_4^+ \cup (a_5 \cup (b_6 b_7))$$

$$a_1 \cup (b_2 b_3) \quad c_4^+ \quad a_5 \quad (b_6 b_7)$$

$$a_1 \quad b_2 b_3 \quad c_4 \quad a_5 \quad b_6 b_7$$

$$b_2 \quad b_3 \quad b_6 \quad b_7$$

We derive a number version of the expression. In this way we can distinguish different instances of the same character.

AMBIGUITY OF REGULAR EXPRESSIONS

a phrase may be obtained through distinct derivations, which differ not only in the order

$$(a \cup b)^* a (a \cup b)^*$$

two different ways to obtain the same string:

a's can be generated by the left part or the right part of the regular expression

$$(a \cup b)^* a (a \cup b)^* \Rightarrow (a \cup b) a (a \cup b)^* \Rightarrow aa (a \cup b)^* \Rightarrow aa \epsilon \Rightarrow aa$$

$$(a \cup b)^* a (a \cup b)^* \Rightarrow \epsilon a (a \cup b)^* \Rightarrow \epsilon a (a \cup b) \Rightarrow \epsilon aa \Rightarrow aa$$

sufficient condition for ambiguity :

a r.e. f is ambiguous if the language of the numbered version f'

includes two distinct strings x and y that coincide when numbers are erased

Example

$$f = (a \cup b)^* a (a \cup b)^*$$

$$f' = (a_1 \cup b_2)^* a_3 (a_4 \cup b_5)^* \text{ is a r.e. of alphabet } \Sigma' = \{a_1, b_2, a_3, a_4, b_5\}$$

$a_1 a_3$ and $a_3 a_4$ prove (witness) the ambiguity of the r.e. $f = (a \cup b)^* a (a \cup b)^*$

Another example (ambiguity)

$(aa \mid ba)^* a \mid b(aa|b)^*$ is ambiguous

numbered version: $(a_1a_2 \mid b_3a_4)^* a_5 \mid b_6(a_7a_8|b_9)^*$

from which one can derive $b_3a_4a_5$ and $b_6a_7a_8$

both mapped to the string baa by erasing the subscript

PAY ATTENTION: ambiguity is often a source of problems

Practical example

APPLICATION of r.e. and ambiguity: specify floating point numbers with or without sign and exponent

$$\Sigma = \{+, -, \bullet, E, d\}$$

$$r = s.c.e$$

$s = (+ \cup - \cup \varepsilon)$ provides the optional \pm sign

$c = (d^+ \bullet d^* \cup d^* \bullet d^+)$ generates integer or fractional constants with no sign

$e = (\varepsilon \cup E (+ \cup - \cup \varepsilon) d^+)$ generates the optional exponent preceded by E

$$(+ \cup - \cup \varepsilon)(d^+ \bullet d^* \cup d^* \bullet d^+)(\varepsilon \cup E (+ \cup - \cup \varepsilon) d^+)$$

optional exponent

There's an ambiguity in the central part, in fact in both following ways we can generate for ex the number 1.1:
 $d^* . d^+$
 $d+ . d^*$

$+dd \bullet E - ddd + 12 \bullet E - 341$ represents the number $12.0 \cdot 10^{-341}$

NB: r.e. for numbers with integer and fractional part is ambiguous: why?
because the two r.e. $d^+ \bullet d^*$ and $d^* \bullet d^+$ define non-disjointed languages
REMEDY?

Typical remedy: divide the language in three disjointed parts, each one modeled by a distinct e.r.

Do this as an exercise ;-)

EXTENDED REGULAR EXPRESSIONS

Extended with other operators

These are shortcut that can be proved starting from what we saw in the last lesson:

POWER: $a^h = aa\dots a$ (h times): a^n

REPETITION: from k to $n > k$: $[a]_k^n = a^k \cup a^{k+1} \cup \dots a^n$

OPTIONALITY: $(\varepsilon \cup a)$ or $[a]$

ORDERED INTERVAL: $(0 \dots 9)$ $(a \dots z)$ $(A \dots Z)$

Set theoretic operators: INTERSECTION, DIFFERENCE, COMPLEMENT

It can be shown (by studying the relation with finite automata) that set theoretic operations do **not** increase the expressive power of r.e.

(they are only useful abbreviations)

INTERSECTION: useful to define languages through **conjunction** of conditions

EXAMPLE: the language $L \subset \{a, b\}^*$ of even-length strings which contain bb

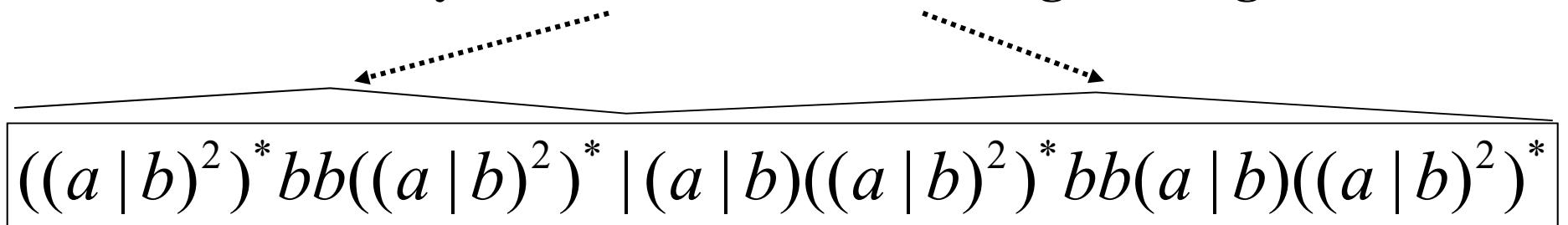
Easy to define using a r.e. with intersection :

$$e = ((a|b)^* bb (a|b)^*) \cap ((a|b)^2)^*$$

phrases including bb even-length phrases

Without intersection:

bb surrounded by two even- or two odd-length strings



Example of extended r.e. with complement operator

Language $L \subset \{a,b\}^*$ of strings **not** containing substring aa

Easy to define its complement: $\neg L = \{ x \in (a \mid b)^* \mid x \text{ contains substring } aa \}$

$$\neg L = ((a \mid b)^* aa (a \mid b)^*)$$

Therefore L can be defined by a r.e. extended with complement

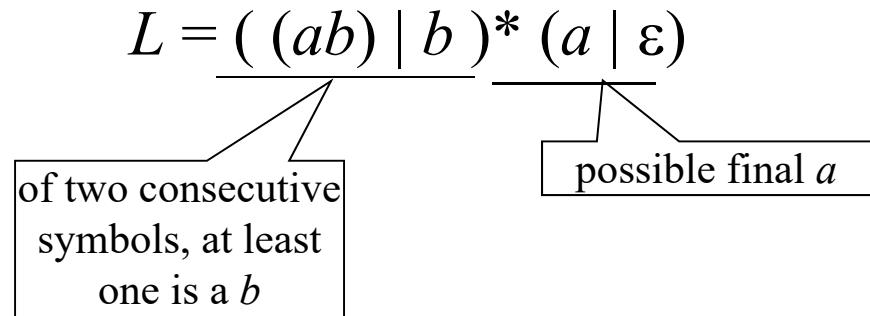
$$L = \neg((a \mid b)^* aa (a \mid b)^*)$$

Definition by a r.e. non-extended (*subjectively* less readable)

$$L = \overline{(ab) \mid b}^* (a \mid \varepsilon)$$

of two consecutive symbols, at least one is a b

possible final a



CLOSURE PROPERTIES OF THE *REG* FAMILY (family of regular languages)

Let op be a unary or binary language operator (e.g., complement, concatenation, etc.)

a family of languages is closed under op iff ...

every language obtained by applying op to languages of the family is also in the family

property: the *REG* family is closed under

concatenation, union, star

(and hence also w.r.t. the derived operators of cross ‘+’ and power)

it is an obvious consequence of the very definition of regular expression

Therefore regular languages can be combined by these operators without exiting *REG*
(i.e., obtaining languages that are still regular)

REG is also closed under INTERSECTION and COMPLEMENT

(this is not so obvious; we will use finite automata to show that)

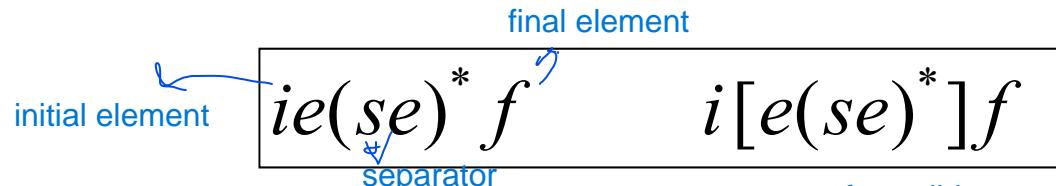
APPLICATION: REPRESENTATION OF LISTS BY MEANS OF R.E.

a list contains an unspecified number of elements e of the same type

generated by the r.e. e^+ , or e^* if it can be empty

e can be a terminal symbol or any regular subexpression

LISTS WITH SEPARATORS AND OPENING AND CLOSING MARKS



Examples from programming languages

begin istr₁; istr₂; ...; istr_n end

procedure PRINT(par₁, par₂, ..., par_n)

array MATRIX '['int₁, int₂, ..., int_n']'

case of possible empty list
(remember: square parenthesis mean optional)

LISTS WITH PRECEDENCE OR LEVELS

An element in a list can be a list of a lower level

NB: the list can be represented by a r.e. only if the **number of levels is limited**

otherwise **more powerful notations are needed (grammars)**

illimited nests (lower level lists) are not representable by regular expressions

$$list_1 = i_1 \ list_2 \ (s_1 \ list_2)^* f_1$$

$$list_2 = i_2 \ list_3 \ (s_2 \ list_3)^* f_2$$

...

$$list_k = i_k \ e_k \ (s_k \ e_k)^* f_k$$

Examples from progr. lang.

level 1: *begin instr₁; instr₂; ... instr_n end*

level 2: *WRITE (var₁, var₂, ... var_n)*

some arithmetic expressions can be viewed as lists (e.g., sums of terms)

$$3 + 5 \times 7 \times 4 - 8 \times 2 \div 5 + 8 + 3$$

Finite Automata

Prof. A. Morzenti

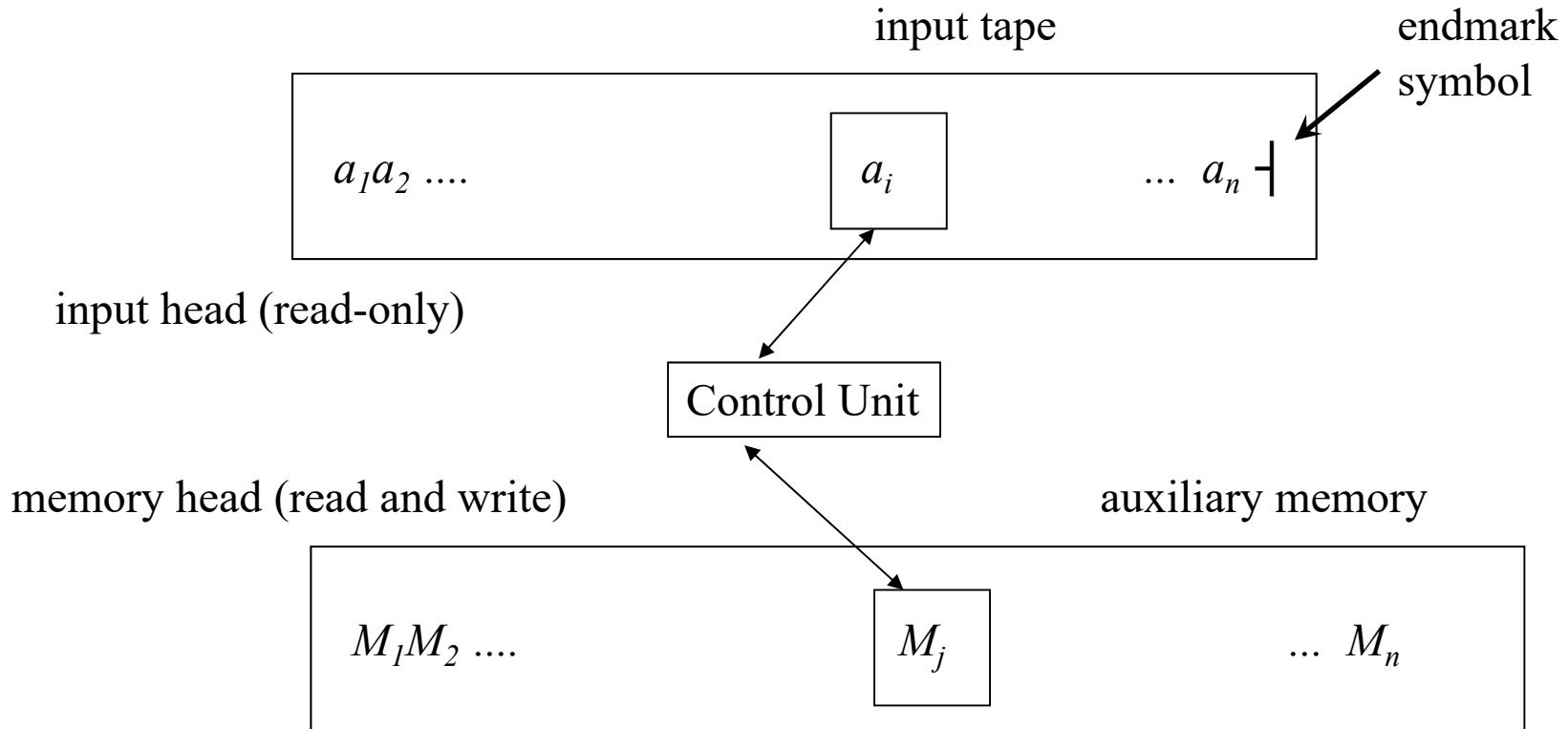
NB: up to slide 13 notions assumed to be well-known from other previous courses

Algorithms for language recognition are seen as automata, in order to:

- highlight relations between language families and grammars
- avoid early reference to software implementation details

It is then easy to provide indications to transform automata into executable code

SCHEME OF A RECOGNIZING AUTOMATON in its most general form:



AUXILIARY MEMORY

- **MISSING** (only that of the control unit): finite state automaton – it recognizes regular languages
- **STACK MEMORY**: pushdown automaton (PDA) – it recognizes CF languages

the automaton examines the source string, through a series of ***moves***:
every move depends on the symbols under the heads and on the state of the control unit

Effects of each move:

- moving the input head one position left or right
- write a symbol in memory and shift one position left or right
- change the state of the Control Unit

UNIDIRECTIONAL MACHINE: input head moves in one direction only
models a *single scan analysis*

instantaneous configuration determines future evolution

defined by three components:

- part of the input tape still unread
- content of the auxiliary memory and position of the memory head
- state of the control unit

initial configuration:

- input head on the first symbol
- control unit in the initial state
- memory storing the initial information (usually a special symbol)

computation: sequence of moves

deterministic – in every configuration at most one move (hence one next config.) is possible

nondeterministic otherwise

final configuration

- control unit in a final state
- input head on the string endmark (terminator) symbol ‘⊣’
 - sometimes alternative condition: memory contains a special symbol or empty

a string x is **accepted** (recognized) iff the automaton,
starting from initial configuration with x as input
performs a computation and reaches a final configuration
(if it is nondeterministic it may do that in many ways)

A computation ends when the automaton

- reaches a final configuration (\Rightarrow string is accepted), or
- no move can be executed (\Rightarrow string not accepted)

language accepted or *recognized* by the automaton: the set of accepted strings

two automata accepting the same language are called *equivalent*

STATE-TRANSITION DIAGRAMS

it is a labeled oriented graph

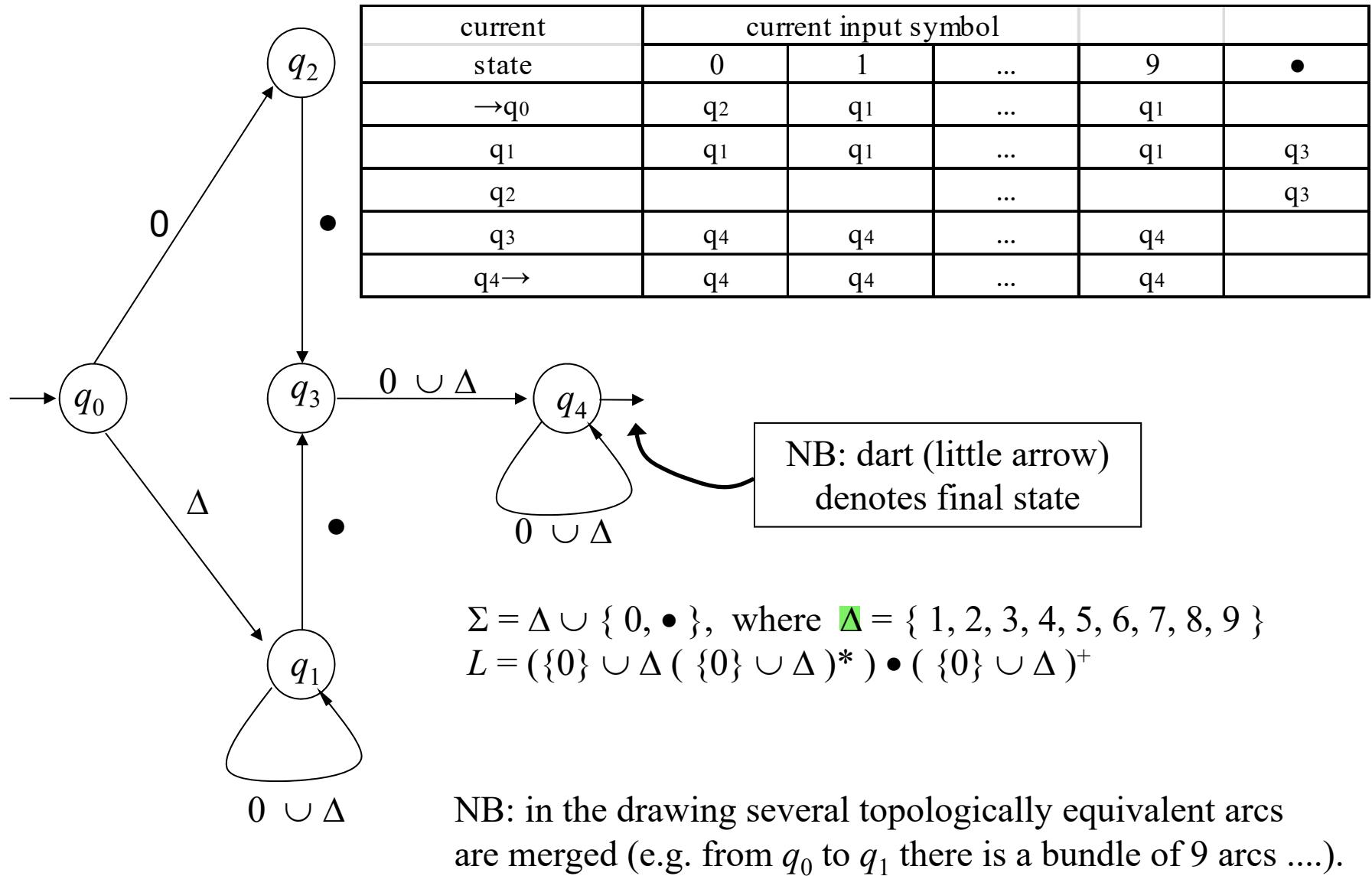
nodes: states of the control unit

arcs: denote transitions, and are labeled with the input symbols

the state-transition diagram has (in the deterministic version)

- a unique initial state
- possibly many final states

Example – decimal numeric constants



FORMAL DEFINITION OF A DETERMINISTIC FINITE AUTOMATON

Five elements:

1. Q , the set of *states* (non-empty, finite)
2. Σ , the *input alphabet* (or terminal alphabet)
3. the *transition function* (possibly partial) $\delta: (Q \times \Sigma) \rightarrow Q$
4. $q_0 \in Q$ the *initial state*
5. $F \subseteq Q$, the set of *final states*

the transition function encodes the moves of the automaton M :

if $\delta(q, a) = r$ then M , in the state q , when reading a goes into state r

alternative notation to $\delta(q, a) = r$: $q \xrightarrow{a} r$

if $\delta(q, a)$ is undefined, the automaton stops (\equiv it enters a non-final error state and rejects)

extension δ^* of δ for the strings of any length: $\delta^*: (Q \times \Sigma^*) \rightarrow Q$

defined inductively as $\delta^*(q, \varepsilon) = q$ and $\delta^*(q, xa) = \delta(\delta^*(q, x), a)$, $x \in \Sigma^*, a \in \Sigma$

for brevity we use δ also to denote its extension δ^*

COMPUTATION CARRIED OUT BY AN AUTOMATON = PATH ON THE GRAPH

STRING RECOGNITION :

string x recognized (or accepted) by automaton M iff
when scanning x , M goes from the initial to a final state:

$$\delta(q_0, x) \in F$$

Hence the empty string ε accepted iff the initial state is also final

language recognized (accepted) by atomaton M : the set of all recognized strings

$$L(M) = \{x \in \Sigma^* \mid \underline{\delta(q_0, x) \in F}\}$$

means "is a final state"

the family of languages accepted by finite state automata is called ***finite-state recognizable***

complexity of acceptance is called «real-time»: number of steps is equal to $|x|$

Example – Decimal numeric constants (follows) – The automaton M (see next) defined as:

$$Q = \{q_0, q_1, q_2, q_3, q_4\}$$

$$\Sigma = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 0, \bullet\}$$

$$q_0 = q_0$$

$$F = \{q_4\}$$

examples of transitions:

$$\delta(q_0, 3 \bullet 1) = \delta(\delta(q_0, 3 \bullet), 1) = \delta(\delta(\delta(q_0, 3), \bullet), 1) =$$

$$= \delta(\delta(q_1, \bullet), 1) = \delta(q_3, 1) = q_4$$

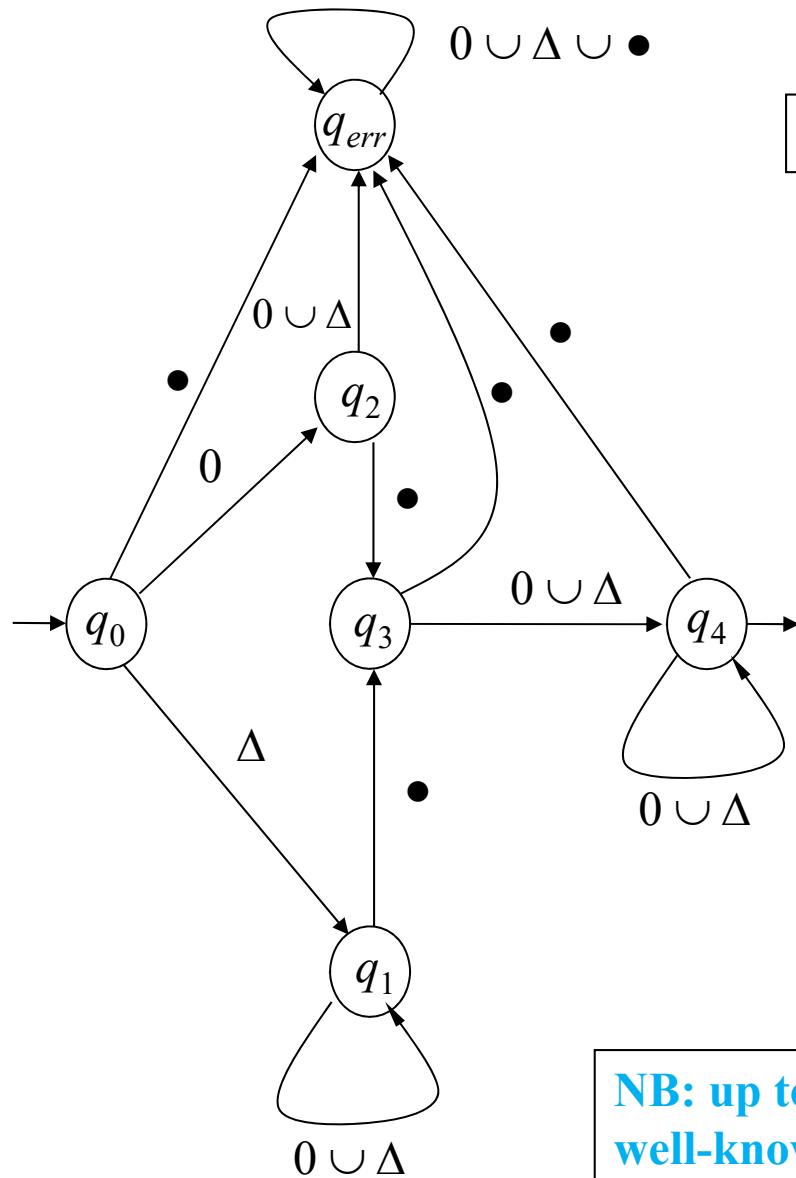
$q_4 \in F$ the string $3 \bullet 1$ is accepted

strings that are not accepted: $3 \bullet$ and 02

$$\delta(q_0, 3 \bullet) = q_3 \quad - q_3 \text{ not final} - \quad 3 \bullet \notin L$$

$$\delta(q_0, 02) = \delta(\delta(q_0, 0), 2) = \delta(q_2, 2) \quad - \text{undefined} - \quad 02 \notin L$$

COMPLETING THE AUTOMATON WITH THE ERROR STATE



q_{err} = error «sink» state

the transition function can always be made complete
by means of an **error state**
without changing the accepted language

\forall state $q \in Q$ and \forall symbol $a \in \Sigma$
if $\delta(q, a)$ is undefined
let $\delta(q, a) = q_{err}$
and \forall symbol $a \in \Sigma$ let $\delta(q_{err}, a) = q_{err}$

NB: up to here notions assumed to be
well-known from other previous courses

CLEAN AUTOMATA

An automaton can include useless parts that do not contribute to string recognition
they must be eliminated

A state q is **reachable** from a state p if there exists a computation going from p to q

A state is **accessible** if it is reachable from the initial state

A state is **postaccessible** if a final state can be reached from it

(\Rightarrow NB: the error state q_{err} is not postaccessible)

A state is **useful** if it is accessible and postaccessible
(it lays on some path from initial to final state)

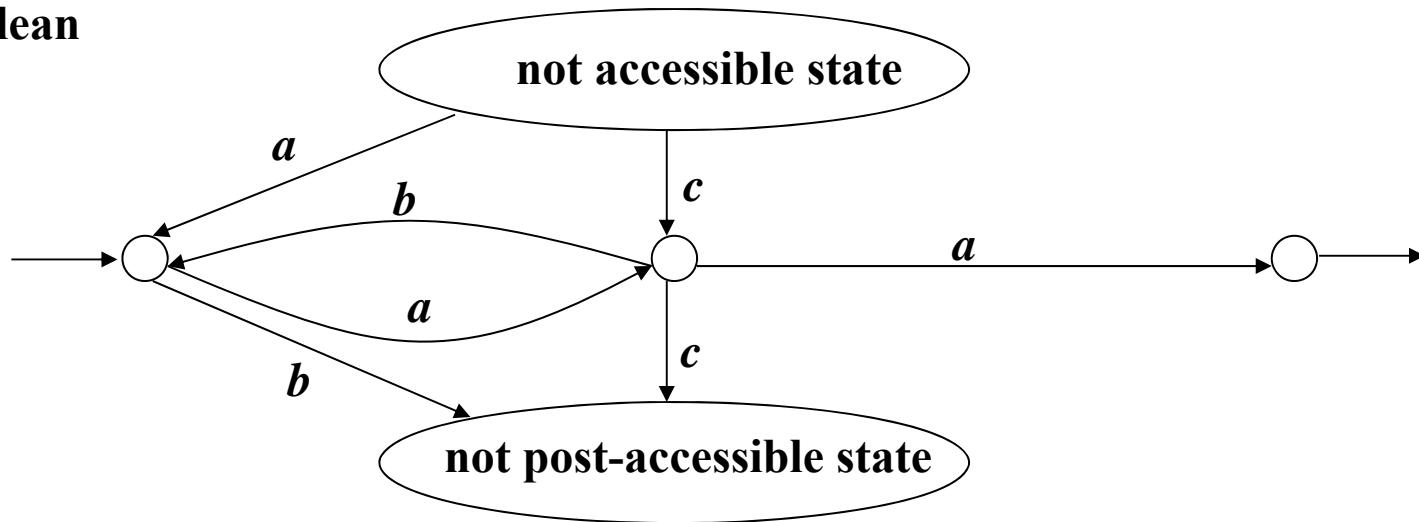
An automaton is **clean** if every state is useful

PROPERTY – Every finite automaton admits an equivalent clean automaton.

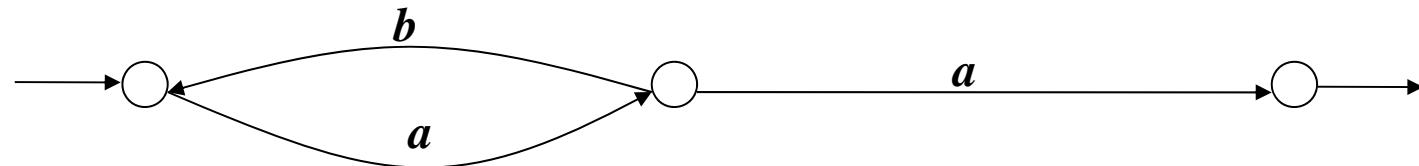
Cleaning an automaton: identify useless states, delete them and all the incident arcs

Example – useless state elimination

non-clean



clean



MINIMAL AUTOMATON

PROPERTY – For every finite-state language, the finite recognizer is minimal w.r.t. the number of states *exists and is unique* (apart from a renaming of states)

We provide a procedure for minimizing the number of states

assuming the automaton is clean except for the possible presence of error state q_{err}

INDISTINGUISHABLE STATES – state p is **indistinguishable** from state q ,

iff, \forall string x , $\delta(p, x)$ and $\delta(q, x)$ are both final, or both nonfinal

(i.e., scanning x from p and from q , one *cannot* reach two states, one final and the other not)

indistinguishability is a *binary relation*; it is *reflexive*, *symmetric*, and *transitive*

hence it is an **equivalence relation**

two indistinguishable states **can be merged**, thus reducing the number of states,
with no change in the language recognized by the automaton

it is a typical construction: the new set of states is the *quotient set* w.r.t. the equivalence class

Impossible to compute the indistinguishability relation *directly* from its definition
one should consider the whole accepted language, which may be infinite

we compute the indistinguishability relation through its complement:

the *distinguishability* relation

it can be computed through its *inductive definition*

p is *distinguishable* from q iff

1. p is final and q is not, or viceversa, or
2. $\exists a: \delta(p, a)$ is distinguishable from $\delta(q, a)$

$\Rightarrow q_{err}$ is distinguishable from every postaccessible state p , because

\exists string $x: \delta(p, x) \in F$ (just because p is postaccessible)

whereas \forall string $x: \delta(q_{err}, x) = q_{err}$

$\Rightarrow p$ is distinguishable from q (both assumed postaccessible)

if the set of labels on arcs outgoing from p and q are different

(NB: not necessarily disjointed)

In fact, if $\exists a$ such that $\delta(p, a) = p'$, with p' postaccessible, whereas $\delta(q, a) = q_{err}$,
then p is distinguishable from q
because q_{err} is distinguishable from all postaccessible states

Example

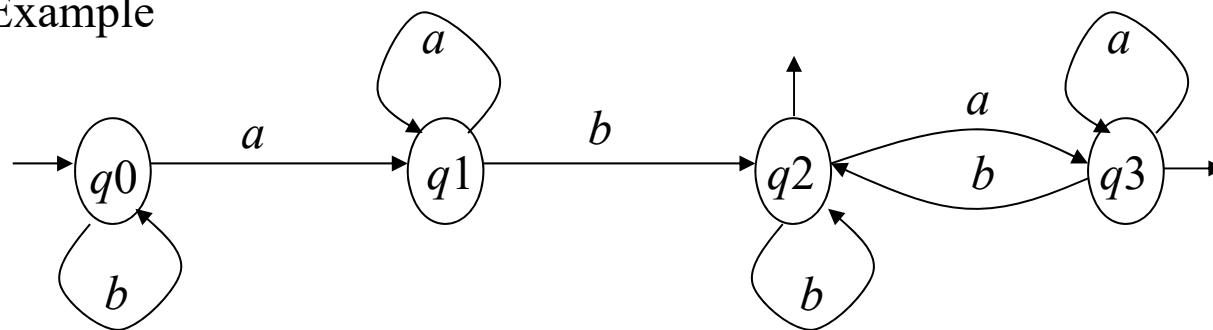


Table of indistinguishable states: initially final and nonfinal states are distinguishable



q1			
q2	X	X	
q3	X	X	
	q0	q1	q2

- $(\delta(q0,a), \delta(q1,a)) = (q1,q1)$
- $(\delta(q0,b), \delta(q1,b)) = (q0,q2)$
- $q0 \text{ dist. } q2 \rightarrow q0 \text{ dist. } q1$
- $(\delta(q2,a), \delta(q3,a)) = (q3,q3)$
- $(\delta(q2,b), \delta(q3,b)) = (q2,q2)$
- $q3=q3, q2=q2 \rightarrow q2 \text{ indist. } q3$



q1	(1,1)(0,2)		
q2	X	X	
q3	X	X	(3,3)(2,2)
	q0	q1	q2

indistinguishable pairs: q_2 and q_3
 equivalence classes of indistinguishable states:
 $[q0], [q1], [q2, q3]$.

q1	X		
q2	X	X	
q3	X	X	
	q0	q1	q2

MINIMIZATION

states of the minimal automaton M' : the equivalence classes of the indistinguishability relation
transition function: arcs among equivalence classes:

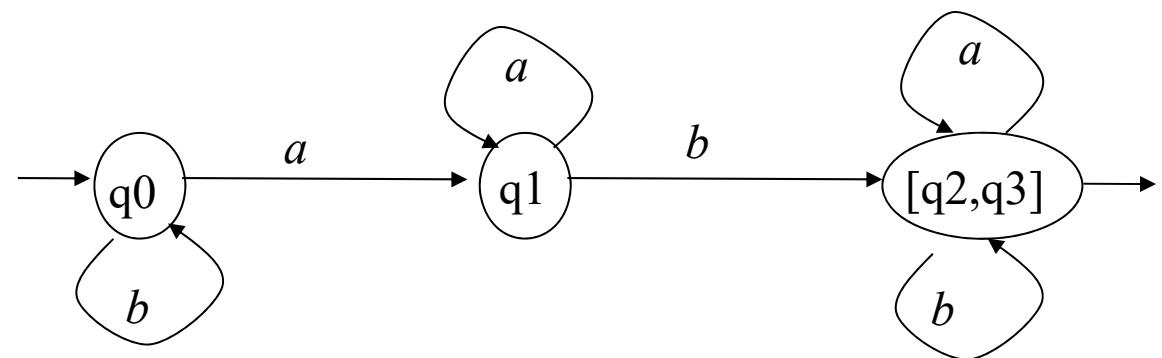
$$[\dots, p_r, \dots] \xrightarrow{b} [\dots, q_s, \dots]$$

iff in M there is an arc:

$$p_r \xrightarrow{b} q_s$$

Example (follows)

Result of minimization:



Example with non-total transition function

Suppose to modify the previous automaton M by removing the move $\delta(q_3, a) = q_3$.

We redefine as $\delta(q_3, a) = q_{err}$

q_2 and q_3 are now distinguishable:

$\delta(q_2, a) = q_3$ and $\delta(q_3, a) = q_{err}$ and q_3 is distinguishable from q_{err}

M is therefore already minimal

The minimization procedure provides a proof of the existence and unicity of a minimum automaton equivalent to any given one.

NB: This property does not hold, in general, for **nondeterministic automata** (coming next)

State minimization provides a method for **checking (deterministic) automata equivalence**:

- clean the automata,
- minimize them,
- check if they are identical (apart from a renaming of states)

Context free grammars - I

Prof. A. Morzenti

NB: many parts are well known from previous courses
I will run quickly through them and discuss more deeply the really new ones

LIMITS OF REGULAR LANGUAGES

Simple languages such as $L = \{ a^n b^n \mid n > 0 \}$

Parenthetical languages are not regular

representing basic syntactic structures like

begin begin ... begin ... end ... end end

are **not** regular

e.g., $b^+ e^+$ does not satisfy the constraint $\#b = \#e$

$(be)^+$ does not ensure nesting

GRAMMARS – a **more powerful** means to define languages

through **rewriting rules**

language phrases generated through repeated application of rules

The grammar is characterized by its set of rules

Example – Language of *palindromes*

$$L = \{ uu^R \mid u \in \{a, b\}^* \} = \{\epsilon, aa, bb, abba, baab, \dots, abbbba, \dots\}$$

a palindrome is...

$P \rightarrow \epsilon$ an empty palindrome

$P \rightarrow a P a$ a palindrome surrounded by two a 's

$P \rightarrow b P b$ a palindrome surrounded by two b 's

A chain of *derivation steps*:

$$P \Rightarrow a P a \Rightarrow ab P ba \Rightarrow abb P bba \Rightarrow abb \epsilon bba = abbbba$$

look out: distinguish the two *metasymbols*

→ separates the left and right part of a rule

⇒ derivation *relation* (rewriting)

Example: a non-empty *list of* palindromes, ex: *abba bbaabb aa*

$$L \rightarrow P L$$

$$L \rightarrow P$$

$$P \rightarrow \varepsilon \quad P \rightarrow a P a \quad P \rightarrow b P b$$

non terminal symbols:

- **L** (axiom, or **start symbol**)
- **P** (defines the component palindrome substrings)

derivation of the string *abba bbaabb aa*

(spaces added for readability, nonterm to be expanded underlined)

$$\begin{aligned} L &\Rightarrow P \underline{L} \Rightarrow P P \underline{L} \Rightarrow \underline{P} P P \Rightarrow a \underline{P} a P P \\ &\Rightarrow ab \underline{P} ba P P \Rightarrow abba \underline{P} P \Rightarrow abba b \underline{P} b P \\ &\Rightarrow abba bb \underline{P} bb P \Rightarrow abba bba \underline{P} abb P \\ &\Rightarrow abba bbaabb \underline{P} \\ &\Rightarrow abba bbaabb a \underline{P} a \Rightarrow abba bbaabb aa \end{aligned}$$

CONTEXT-FREE GRAMMAR (BNF - Backus Normal Form – TYPE 2 – FREE GRAMMAR)

defined by four entities

1. V , *non terminal alphabet*, is the set of nonterminal symbols
2. Σ , *terminal alphabet*, is the set of the symbols of which phrases/sentences are made
3. P , is the set of *rules* or *productions*
4. $S \in V$ is the specific nonterminal, called the *axiom (Start)*, from which derivations start

a rule is written as $X \rightarrow \alpha$, with $X \in V$ and $\alpha \in (V \cup \Sigma)^*$

rules with the same nonterminal X : $X \rightarrow \alpha_1, X \rightarrow \alpha_2, \dots, X \rightarrow \alpha_n$

can be written in brief as $X \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$ or $X \rightarrow \alpha_1 \cup \alpha_2 \cup \dots \cup \alpha_n$

$\alpha_1, \alpha_2, \dots, \alpha_n$ are called the *alternatives* of X

To avoid confusion,

the metasymbols ' \rightarrow ', ' $|$ ', ' \cup ', ' ε ' cannot be terminal symbols,
terminal and nonterminal alphabets must be disjointed

NOTATIONS to distinguish terminal and nonterminal symbols

- angle brackets:

$$\langle \text{if-phrase} \rangle \rightarrow \text{if } \langle \text{cond} \rangle \text{ then } \langle \text{if-phrase} \rangle \text{ else } \langle \text{if-phrase} \rangle$$

- ***bold italic***:

$$\textit{if-phrase} \rightarrow \textbf{if } \textit{cond} \textbf{ then } \textit{if-phrase} \textbf{ else } \textit{if-phrase}$$

- quotation marks “ ”

$$\textit{if-phrase} \rightarrow \text{‘if’ } \textit{cond} \text{ ‘then’ } \textit{if-phrase} \text{ ‘else’ } \textit{if-phrase}$$

- upper- versus lower-case:

$$F \rightarrow \textit{if } C \text{ then } D \text{ else } D$$

WE USUALLY ADOPT THESE **CONVENTIONS**:

- terminal characters - $\{a, b, \dots\}$
- nonterminal characters - $\{A, B, \dots\}$
- strings $\in \Sigma^*$ (only terminals) - $\{r, s, \dots, z\}$
- strings $\in (V \cup \Sigma)^*$ (terminals and nonterminals) - $\{\alpha, \beta, \dots\}$
- strings $\in V^*$ (only **nonterminals**) - σ

TYPES OF RULES (RP = right part, LP = left part)

<u>Terminal</u> : RP contains only terminals, or the empty string	$\rightarrow u \mid \epsilon$
<u>Empty (or null)</u> : RP is empty	$\rightarrow \epsilon$
<u>Initial / Axiomatic</u> : LP is the axiom	$S \rightarrow$
<u>Recursive</u> : LP occurs in RP	$A \rightarrow \alpha A \beta$
<u>Left-recursive</u> : LP is prefix of RP	$A \rightarrow A \beta$
<u>Right-recursive</u> : LP is suffix of RP	$A \rightarrow \alpha A$
<u>Left- and right-recursive</u> : conjunction of two previous cases	$A \rightarrow A \beta A$
<u>Copy or categorization</u> : RP is a single nonterminal	$A \rightarrow B$
<u>Linear</u> : at most one nonterminal in RP	$\rightarrow u B v \mid w$
<u>Right-linear</u> (type 3): linear + nonterminal is suffix	$\rightarrow u B \mid w$
<u>Left-linear</u> (type 3): linear + nonterminal is prefix	$\rightarrow B v \mid w$
<u>Homogeneous normal</u> : n nonterminals or just one terminal	$\rightarrow A_1 \dots A_n \mid a$
<u>Chomsky normal</u> (or homogeneous of degree 2): two nonterminals or just one terminal	$\rightarrow BC \mid a$
<u>Greibach normal</u> : one terminal possibly followed by nonterminals	$\rightarrow a \sigma \mid b$
<u>Operator normal</u> : two nonterminals separated by a terminal (operator); more generally, strings devoid of adjacent nonterminals	$\rightarrow A a B$

DERIVATIONS AND GENERATED LANGUAGE

Def. of *derivation relation* ‘ \Rightarrow ’

for $\beta, \gamma \in (V \cup \Sigma)^*$ β derives γ for grammar G , $\beta \stackrel{G}{\Rightarrow} \gamma$, or $\beta \Rightarrow \gamma$, iff

$$\beta = \delta A \eta, \quad A \rightarrow \alpha \quad \text{is a rule of } G, \text{ and} \quad \gamma = \delta \alpha \eta$$

the rule $A \rightarrow \alpha$ is applied in that *derivation step*, and α **reduces to** A

power, reflexive and transitive closure of ‘ \Rightarrow ’

$$\boxed{\beta_0 \stackrel{n}{\Rightarrow} \beta_n \quad \beta_0 \stackrel{*}{\Rightarrow} \beta_n \quad \beta_0 \stackrel{+}{\Rightarrow} \beta_n}$$

If $A \stackrel{*}{\Rightarrow} \alpha$ $\alpha \in (V \cup \Sigma)$ called **string form generated by G**

If $S \stackrel{*}{\Rightarrow} \alpha$ α called **sentential** or **phrase form**

If $A \stackrel{*}{\Rightarrow} s$ $s \in \Sigma^*$, s is called **phrase** or **sentence**

LANGUAGE GENERATED FROM
NONTERMINAL A OR FROM AXIOM S

$$\boxed{L_A(G) = \left\{ x \in \Sigma^* \mid A \stackrel{+}{\Rightarrow} x \right\}}$$

$$L(G) = L_S(G) = \left\{ x \in \Sigma^* \mid S \stackrel{+}{\Rightarrow} x \right\}$$

Example: Grammar G_l generates the structure of a book: it contains

- a front page (f)
- a series (denoted by the nonterm. A) of one or more chapters
- every chapter starts with the title (t) and contains a sequence (B) of one or more lines (l)

$$\begin{array}{l} S \rightarrow fA \\ A \rightarrow AtB \mid tB \\ B \rightarrow lB \mid l \end{array}$$

--> recursive representation (what in finite automata were loops)

from A one generates the string form $tBtB$ and the phrase $tlltl \in L_A(G_l)$

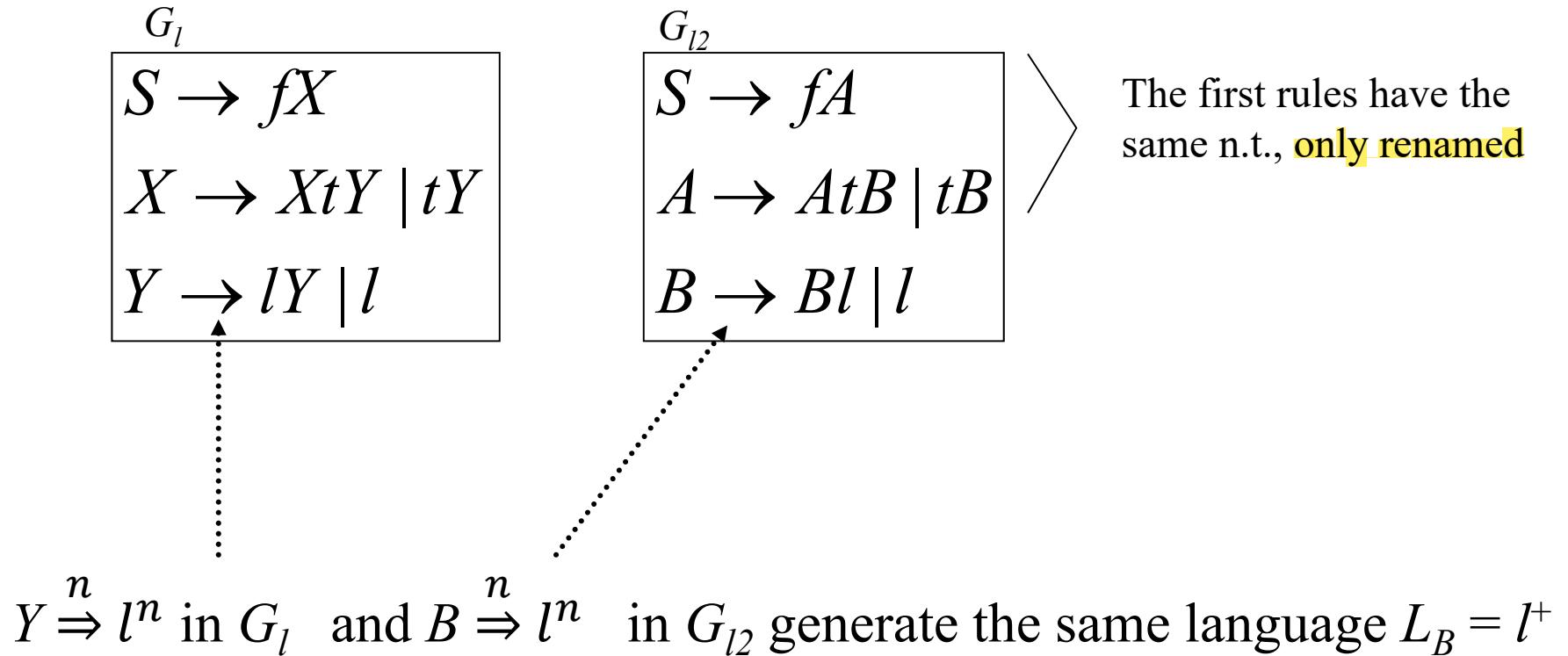
from S one generates the phrase forms $fAtlB, ftBtB$

The language generates from B is $L_B(G_l) = l^+$

$L(G_l)$, being generated by the context free grammar G_l , is **context free** or **free**

Notice: it is also regular, because it is defined also by the regular expression $f(l^+)^+$ 9 / 20

Two grammars G and G' are equivalent if they generate the same language,
that is, $L(G) = L(G')$



ERRONEOUS GRAMMARS AND USELESS RULES

A grammar G is **clean** (or **reduced**) iff for every nonterminal A :

1. A is **reachable** from the axiom S , and hence contribute to the generation of the language; that is, there exists a derivation

If it's unreachable then it's useless and therefore can be removed

$$S \xrightarrow{*} aA\beta$$

2. A is **defined**, that is, it generates a non-empty language (we are not interested in the language \emptyset)

$$L_A(G) \neq \emptyset$$

NB: $L_A(G) = \emptyset$ includes also the case when no derivation from A terminates with a terminal string s (i.e. $s \in \Sigma^*$), e.g.: $P = \{ S \rightarrow aA, A \rightarrow bS \}$

Infinite derivations that don't generate anything

GRAMMAR CLEANING: two steps algorithm:

The FIRST PHASE builds the set ***UNDEF*** of undefined nonterminals

The SECOND PHASE builds the set of unreachable nonterminals

PHASE 1- We first build the ***complement*** set $DEF = V \setminus UNDEF$

DEF is ***initialized*** from the ***terminal rules*** (the n.t. that **immediately generate a terminal string**)

$$DEF := \{ A \mid (A \rightarrow u) \in P, \text{ with } u \in \Sigma^* \}$$

The following ***update*** is repeated until a ***fixed point*** is reached :

$$DEF := DEF \cup \{ B \mid (B \rightarrow D_1D_2\dots D_n) \in P \wedge \forall i (D_i \in DEF \cup \Sigma) \}$$

Every D_i is already in DEF or it is a terminal

In algebra, the ***fixed point*** of a transformation

is an object that is transformed into itself

At each iteration, two cases can occur:

1. New nonterm. are found having the RP all with defined nonterm. or term., or
2. No new nonterm. is found, algorithm terminates (a *fixed point* has been reached)

nonterminals $\in UNDEF$ are eliminated

PHASE 2 – Consider the *produce* relation, defined as

$A \text{ produce } B \quad \text{iff} \quad (A \rightarrow \alpha B \beta) \in P, \text{ with } A \neq B \quad \alpha, \beta \text{ any string}$

C is *reachable* from S iff there exists, in the graph of the produce relation, a path from S to C

nonterminals that are not reachable can be eliminated

often another requirement is added for cleanliness condition of a grammar G :

3. G must not allow for *circular derivations*: they are not essential and introduce *ambiguity*

if $A \xrightarrow{+} A$ then

if the derivation $A \xrightarrow{+} x$ is possible

then also $A \xrightarrow{+} A \xrightarrow{+} x$ and many other similar ones exist

NB: *circular derivations* must not be confused with *recursive rules* and *derivations* !!

I can always remove circular derivation because they only lead from A to itself, ex of circular derivation:

1) A to itself:

$A \rightarrow A$

2) rules that lead to empty or A itself

$A \rightarrow BC$

$B \rightarrow \text{empty}$

$C \rightarrow A$

EXAMPLES OF GRAMMARS THAT ARE NOT CLEAN

- 1) $\{ S \rightarrow aASb, A \rightarrow b \}$ (S does not generate any phrase, i.e., $L(S)=\emptyset$)
- 2) $\{ S \rightarrow a, A \rightarrow b \}$ (A not reachable) ($\{ S \rightarrow a \}$ equiv. clean version)
- 3) $\{ S \rightarrow aASb \mid A, A \rightarrow S \mid b \}$ (circular on S and A) ($\{ S \rightarrow aSSb \mid b \}$ equiv. clean)

circularity can also derive from an empty rule (case 2 of the previous slide note)

$$X \rightarrow XY \mid \dots \quad Y \rightarrow \varepsilon \mid \dots$$

NB: even if clean, a grammar can have **redundant rules** (leading to ambiguity)

(1,4) and (2,5) generate
the same phrases

1. $S \rightarrow aASb$	4. $A \rightarrow c$
2. $S \rightarrow aBSb$	5. $B \rightarrow c$
3. $S \rightarrow \varepsilon$	

RECURSION AND LANGUAGE **INFINITY**

most interesting languages are infinite

but what determines the ability of a grammar to generate an infinite language?

infinity of the language implies unbounded phrase length

therefore the grammar must be recursive

a **derivation** $A \xrightarrow{n} xAy \quad n \geq 1$ is **recursive**

if $n = 1$ it is **immediately recursive**

A is a **recursive nonterminal**

if $x = \epsilon$ then it is **left recursive** (l.r. derivation, l.r. nonterminal)

if $y = \epsilon$ then it is **right recursive** (r.r. derivation, r.r. nonterminal)

NB: circularity and recursiveness are (very) different notions

a grammar may be recursive (admit recursive derivations) but not circular

circular \Rightarrow recursive but it is **not** the case that recursive \Rightarrow circular

necessary and sufficient condition for language $L(G)$ to be infinite,

assuming G clean and devoid of circular derivations,

is that G allows for recursive derivations

necessary condition: if no recursive derivation was possible,
then every derivation would have limited length hence $L(G)$ would be finite

sufficient condition:

$$A \xrightarrow{n} xAy \text{ implies } A \xrightarrow{+} x^m A y^m$$

for any $m \geq 1$ with $x, y \in \Sigma^*$ not both empty (because grammar is not circular)

Furthermore G clean implies

$$S \xrightarrow{*} uAv \text{ (} A \text{ reachable from } S\text{)}$$

and $A \xrightarrow{+} w$ (derivation from A terminates successfully)

therefore there exist nonterminals that generate an infinite language

$$S \xrightarrow{*} uAv \xrightarrow{+} ux^m A y^m v \xrightarrow{+} ux^m w y^m v, (\forall m \geq 1)$$

a grammar does not have recursive derivations

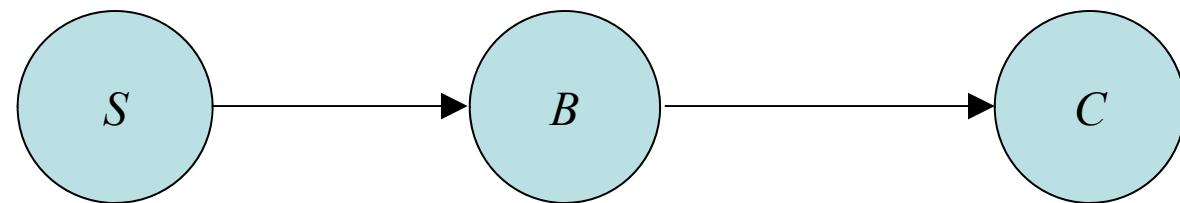
\Leftrightarrow (if and only if)

the graph of the *produce* relation has no circuits

Example

$$\begin{array}{l} S \rightarrow aBc \\ B \rightarrow ab \mid Ca \\ C \rightarrow c \end{array}$$

finite language: { $aabc, acac$ }



Example (arithmetic expressions)

$$G = \left(\underbrace{\{E, T, F\}}_{non\ term.}, \underbrace{\{i, +, *, (), ()\}}_{term.}, \underbrace{P}_{productions}, \underbrace{E}_{axiom} \right)$$

$$P = \{E \rightarrow E + T \mid T, \quad T \rightarrow T * F \mid F, \quad F \rightarrow (E) \mid i\}$$

$$L(G) = \{i, i + i + i, \quad i * i, \quad (i + i)^* i, \dots\}$$

F (factor) has indirect recursion (non immediate)

E (expression) has immediate left recursion and non immediate recursion

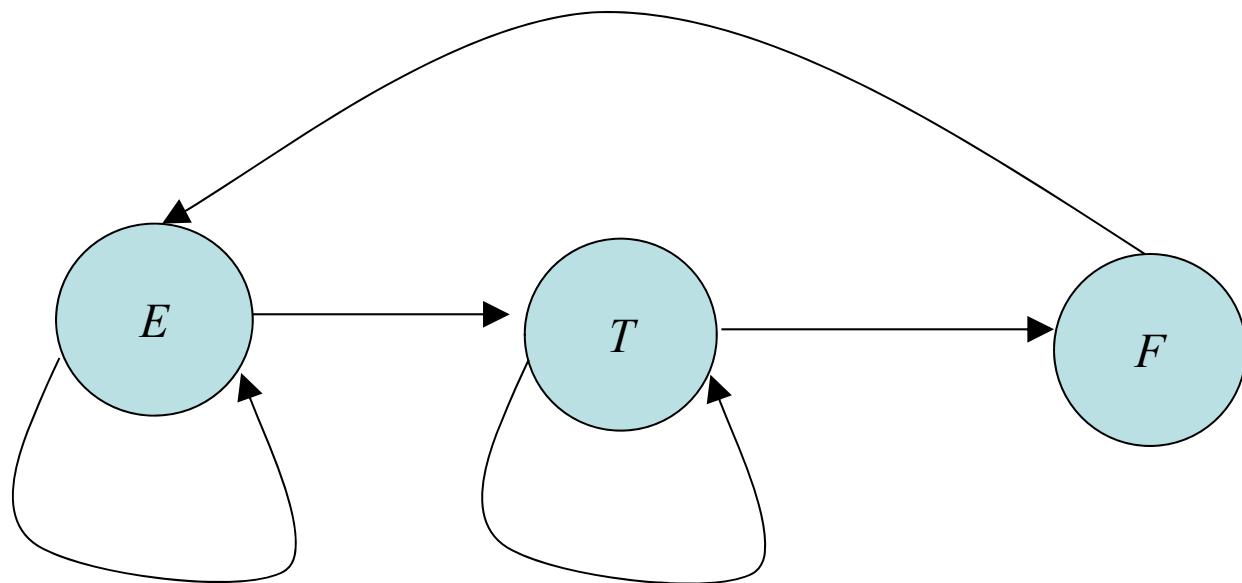
T (term) has immediate left recursion and non immediate recursion

G is clean, recursive, noncircular, hence the generated language is infinite

grammar has recursions

\Leftrightarrow

the graph of the produce relation has circuits



$$G = (\{E, T, F\}, \{i, +, *, (), ()\}, P, E)$$

$$P = \{E \rightarrow E + T | T, \quad T \rightarrow T * F | F, \quad F \rightarrow (E) | i\}$$

Context free grammars - II

Prof. A. Morzenti

SYNTAX TREES AND CANONICAL DERIVATIONS

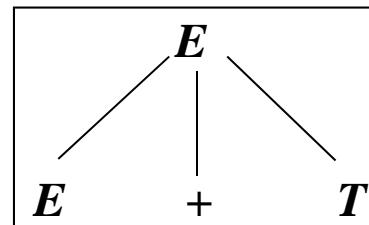
SYNTAX TREE: An oriented, sorted graph (children sorted from left to right) with no cycles, such that, for each pair of nodes, there is only one path connecting them

- it represents graphically the derivation process
- father-child relation / descendants / root node / leaf (or terminal) nodes
- degree of a node: number of its children
- root contains the axiom S
- frontier of the tree (leaf sequence from left to right) contains the generated phrase

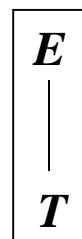
SUBTREE with root N : the tree having N as its root; it includes N and all its descendants

Expressions with sums and products: E expression, T term, F factor

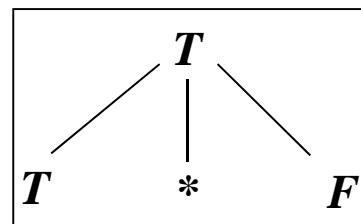
$$1. E \rightarrow E + T$$



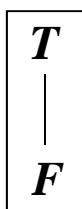
$$2. E \rightarrow T$$



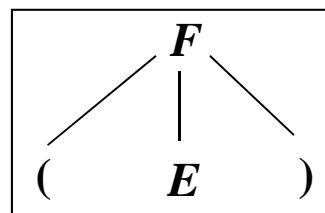
$$3. T \rightarrow T * F$$



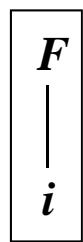
$$4. T \rightarrow F$$



$$5. F \rightarrow (E)$$



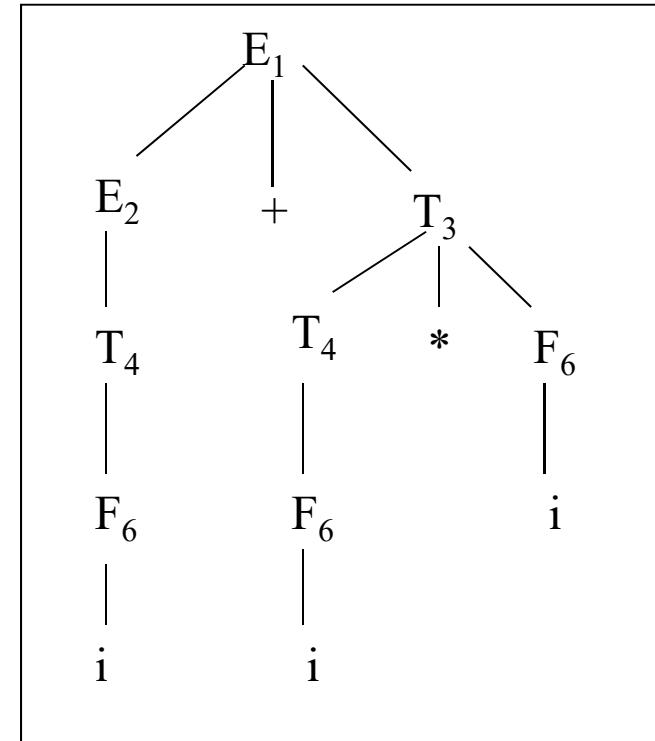
$$6. F \rightarrow i$$



Syntax tree for sentence $i + i * i$
 (n.t. labelled with the applied rule)

Linear representation of the tree (subscripts indicate the n.t. in the subtree root)

$$[[[[i]_F]_T]_E + [[[i]_F]_T * [i]_F]_T]_E$$



LEFT DERIVATION

(numbers denote the rule, expanded nonterm. is underlined)

$$\begin{aligned}
 E &\xrightarrow[1]{} \underline{E} + T \xrightarrow[2]{} \underline{T} + T \xrightarrow[4]{} \underline{F} + T \xrightarrow[6]{} i + \underline{T} \xrightarrow[3]{} i + \underline{T} * F \xrightarrow[4]{} \\
 &\quad \xrightarrow[4]{} i + \underline{F} * F \xrightarrow[6]{} i + i * \underline{F} \xrightarrow[6]{} i + i * i
 \end{aligned}$$

RIGHT DERIVATION

(numbers denote the rule, expanded nonterm. is underlined)

$$\begin{aligned}
 E &\xrightarrow[1]{} E + \underline{T} \xrightarrow[3]{} E + T * \underline{F} \xrightarrow[6]{} E + \underline{T} * i \xrightarrow[4]{} E + \underline{F} * i \xrightarrow[6]{} \underline{E} + i * i \xrightarrow[2]{} \\
 &\quad \xrightarrow[2]{} \underline{T} + i * i \xrightarrow[4]{} \underline{F} + i * i \xrightarrow[6]{} i + i * i
 \end{aligned}$$

Derivations may be neither right nor left

$$\begin{aligned} E &\Rightarrow E + \underline{T} \xrightarrow{l,r} E + T * F \xrightarrow{l} T + \underline{T} * F \xrightarrow{l} T + F * \underline{F} \xrightarrow{r} T + F * i \xrightarrow{l} \\ &\Rightarrow F + \underline{F} * i \xrightarrow{l} F + i * i \xrightarrow{l,r} i + i * i \end{aligned}$$

However, for a **fixed** syntax tree of a sentence, there exist

- a unique right derivation, and
- a unique left derivation

matching that tree

Right and left derivation are useful to define **parsing** (i.e., syntax analysis) algorithms

A different question: does a given sentence have a unique syntax tree?

Yes if thee grammar is not ambiguous,
meaning we need only one way to
produce the string.

This determines the **ambiguity** of the grammar

$$G = \left(\underbrace{\{E, T, F\}}_{non\ term.}, \underbrace{\{i, +, *, (), ()\}}_{term.}, \underbrace{P}_{productions}, \underbrace{E}_{axiom} \right)$$

$$P = \{E \rightarrow E + T \mid T, \quad T \rightarrow T * F \mid F, \quad F \rightarrow (E) \mid i\}$$

Notice that adding $E \rightarrow E|E+T$ would make the grammar ambiguous, because we would have to way to produce the same string.

$$L(G) = \{i, i+i+i, \quad i*i, \quad (i+i)*i, \dots\}$$

(apparently strange structure but necessary to model operator precedence, and avoid ambiguity)

Notice that this grammar can model precedence and associative rule

Exercise: specify the lang. of arithmetic expr. using regular expressions
 obviously not possible ;-)
 recursion makes expr. similar to lists with unbounded nesting level

IMPORTANT: as an exercise, generate many strings, to understand how the grammar works, and check that operator precedence is necessarily respected

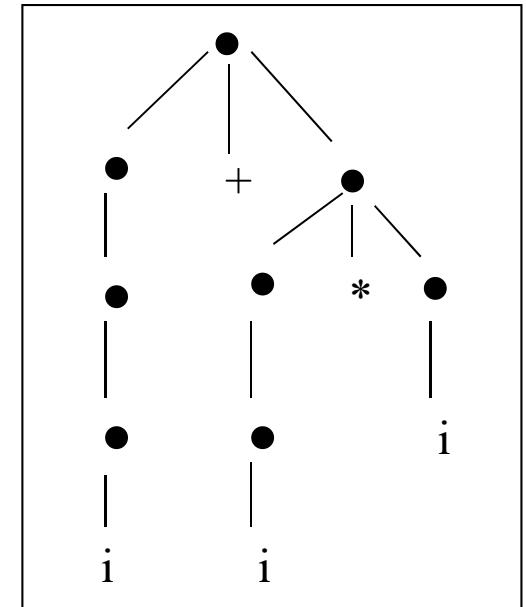
VERY IMPORTANT «Rule of thumb» for modeling precedence:

- nonterminals for ***low-precedence*** operators derived ***first***
- nonterminals for ***high-precedence*** operators derived ***later***

i.e., they are respectively closer to or farther from the axiom in the *produce* relation

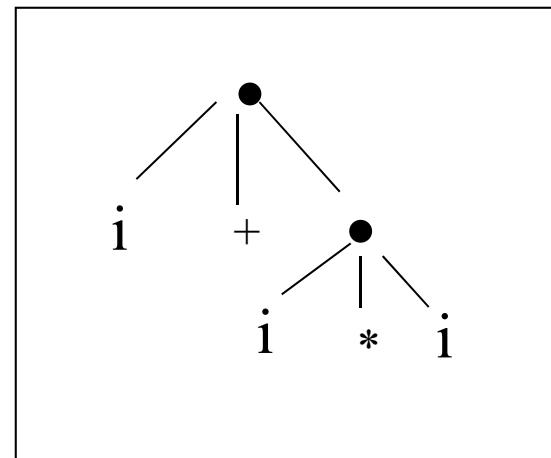
SKELETON TREE (only the frontier and the structure)

$$[[[[i]]] + [[[i]] * [i]]]$$



CONDENSED SKELETON TREE (internal nodes on a non-branching paths are merged)

$$[[i] + [[i] * [i]]]$$



PARENTHESIS LANGUAGES

structures with pairs of opening / closing marks

nested: inside a pair there can be other parenthesized structures (recursion)

(nested) structures can also be placed in sequences at the same level of nesting

Pascal:

begin ...end

C:

{...}

XML:

<title> ... </title>

LaTeX:

\begin{equation} \dots \end{equation}

Abstracting away from the type of parentheses, the paradigmatic language is

The Dyck language

Ex. alphabet:

$$\Sigma = \{ ')', '(', ']', '[' \}$$

Sentence example:

$$()[]()$$

DYCK LANGUAGE with opening parenthesis a, \dots , closing parenthesis c, \dots
Grammar is surprisingly simple

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

$$S \rightarrow aScS \mid \epsilon$$

$a a \underbrace{a c} c a a \underbrace{a c} c c c$



The language is not linear (>1 nonterm. in the right part)

Exercise: build the syntax tree for the sentence

$a a a c c a a a c c c c$

LINEAR NON REGULAR LANGUAGE:

$$L_1 = \{a^n c^n \mid n \geq 1\} = \{ac, aacc, \dots\}$$
$$S \rightarrow aSc \mid ac$$

L_1 is a proper subset of the Dyck Language:
it does not admit many nested structures at the same level
(that is, strings of type $acacac$ are ruled out)

REGULAR COMPOSITION OF FREE LANGUAGES

Applying the union, concatenation, Kleen star operations to free languages ...
one obtains free languages, that is ...

The family of free languages is closed under union, concatenation, and star

$$G_1 = (\Sigma_1, V_{N_1}, P_1, S_1) \text{ and } G_2 = (\Sigma_2, V_{N_2}, P_2, S_2)$$

$$V_{N_1} \cap V_{N_2} = \emptyset \quad S \notin (V_{N_1} \cup V_{N_2})$$

NB: one needs disjointed nonterm. sets and a new axiom

UNION:

$$G = (\Sigma_1 \cup \Sigma_2, \{S\} \cup V_{N_1} \cup V_{N_2}, \{S \rightarrow S_1 | S_2\} \cup P_1 \cup P_2, S)$$

this is ambiguous in case of not disjointed languages, meaning there are the same names of nonterminal symbols, one easy way to fix this is to rename non-terminal symbols

CONCATENATION:

$$G = (\Sigma_1 \cup \Sigma_2, \{S\} \cup V_{N_1} \cup V_{N_2}, \{S \rightarrow S_1 S_2\} \cup P_1 \cup P_2, S)$$

first generate a string of the first language and then concatenate a string of the second one

STAR: G for $(L_1)^*$ is obtained by adding to G_1 the rules $S \rightarrow SS_1 | \varepsilon$

CROSS ‘+’ (not necessary because ‘+’ is derived; this is added for simplicity):

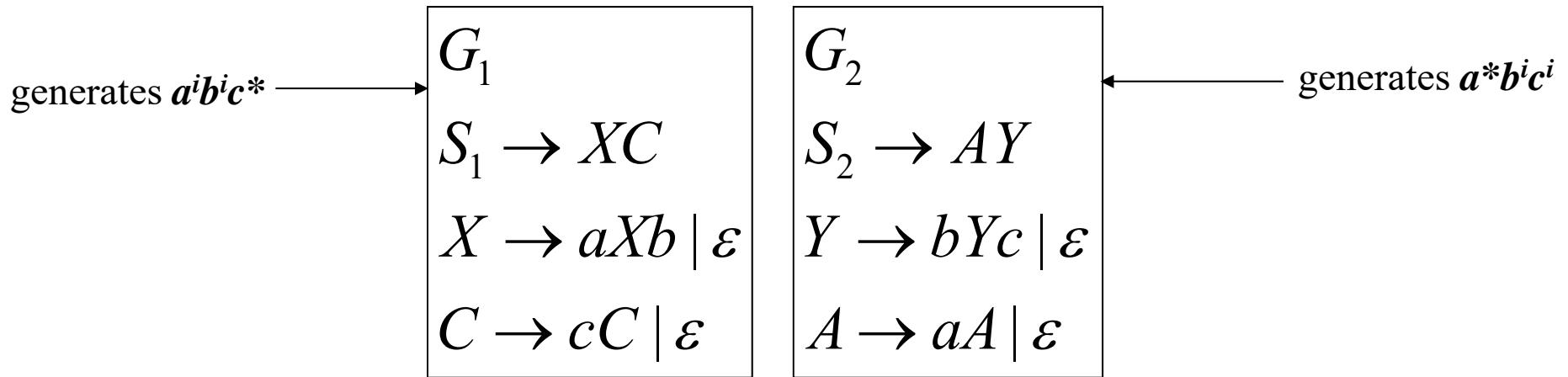
G for $(L_1)^+$ is obtained by adding to G_1 the rules $S \rightarrow SS_1 | S_1$)

BTW: The *mirror language* of $L(G)$, $(L(G))^R$ is generated by the *mirror grammar*, obtained by reversing the right part of the rules

EXAMPLE: Union of free languages

$$L = \{a^i b^j c^k \mid i = j \vee j = k\} = \{a^i b^i c^* \mid i \geq 0\} \cup \{a^* b^i c^i \mid i \geq 0\} = L_1 \cup L_2$$

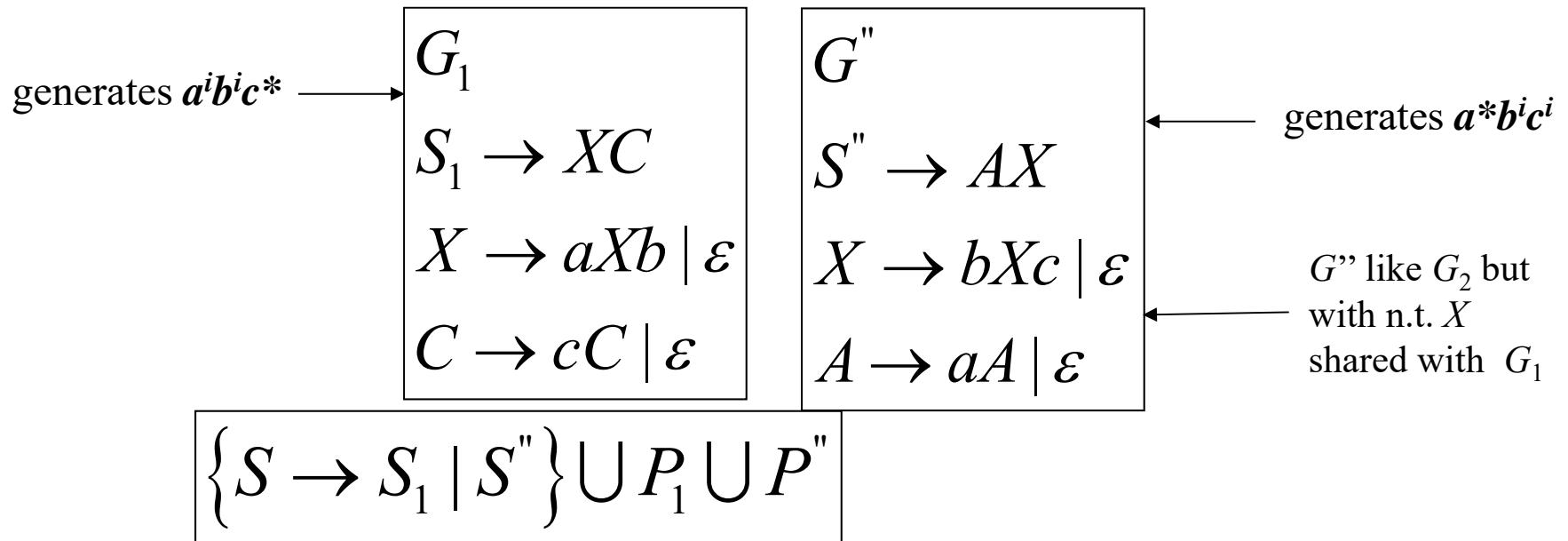
L_1 and L_2 are generated by the two grammars G_1 and G_2



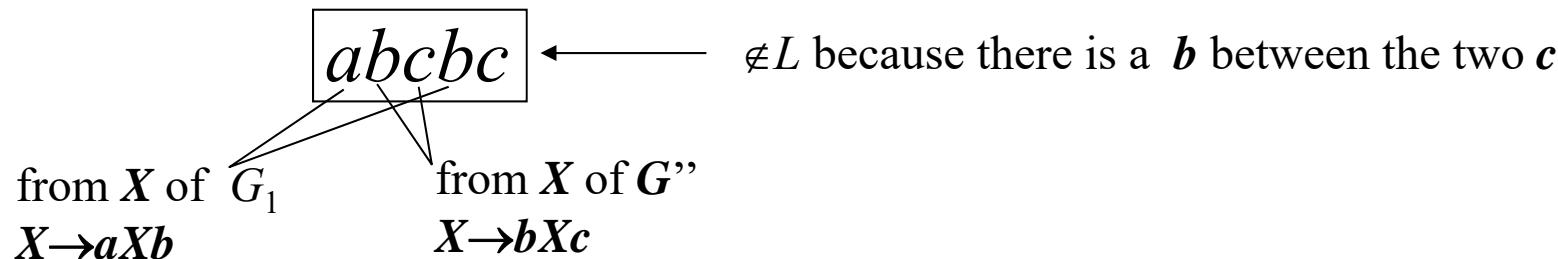
Notice that the nonterminal sets of grammars G_1 and G_2 are **DISJOINTED**

EXAMPLE: Union of free languages (follows)

What happens if the nonterm. sets are not disjoint? Let us use G'' instead of G_2



If nonterm. are not disjoint, the grammar generates a **superset** of the union language:
spurious additional sentences are generated



AMBIGUITY

Examples from natural language

“I saw the man with the binoculars.”

“La pesca è bella”

“half baked chicken”

“Questa è una rapina, dateci i soldi altrimenti *spariamo*.” “OK, allora *sparite*”...

In artificial, technical languages ambiguity must be ruled out (in the natural ones...).

We only consider SYNTACTIC AMBIGUITY:

A sentence x of a grammar G is ambiguous if it admits several distinct syntax trees

In such a case we say that the grammar G is ambiguous

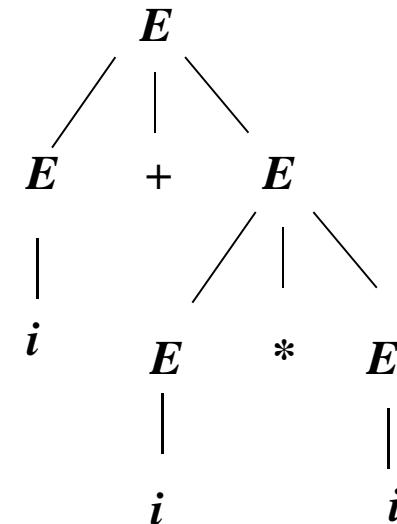
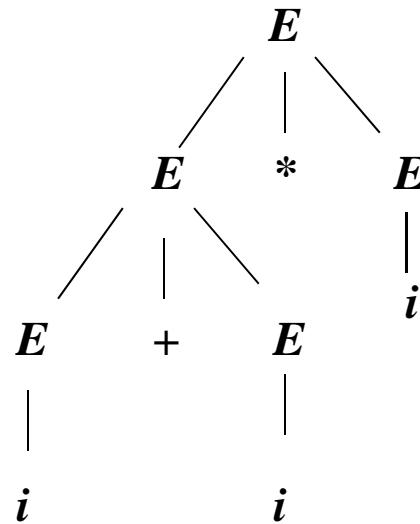
EXAMPLE: a “simple” grammar G' for arithmetic expressions (with bilateral recursion)

$$E \rightarrow E + E \mid E * E \mid (E) \mid i$$

$$E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow i + E * E \Rightarrow i + i * E \Rightarrow i + i * i$$

$$E \Rightarrow E + E \Rightarrow i + E \Rightarrow i + E * E \Rightarrow i + i * E \Rightarrow i + i * i$$

Two trees that generate the same string



the sentence $i+i+i$ is also ambiguous

G' is ambiguous and **does not enforce the usual precedence of product over sum**

No precedence of operators and associativity is not defined

NB: G' is smaller than the previous G (p.3): it has only 1 n.t. and 4 rules

in grammar design there is often a **trade-off between size and ambiguity**

The **degree of ambiguity** of a sentence x of a language $L(G)$
is the number of distinct trees of x compatible with G
for a **grammar** the degree of ambiguity
is the maximum among the degree of ambiguity of its sentences
Notice that the degree of ambiguity of sentences of a grammar can be unlimited
IMPORTANT PROBLEM: determine if a grammar is ambiguous

The **problem** is **undecidable**:
there does not exist a general algorithm that, given any free grammar,
terminates (after a finite number of steps) with the correct answer

The **absence of ambiguity** in a specific grammar can be shown
on a case by case basis, by hand, through **inductive reasonings**,
hence by analyzing a finite number of cases

Instead, to show ambiguity one can exhibit a **witness**: an ambiguous sentence

BEST APPROACH: AVOID AMBIGUITY IN THE GRAMMAR DESIGN PHASE

Example: arithmetic expressions

The degree of ambiguity is 2 for $i + i + i$ and 5 for $i + i * i + i$

$$i + \underbrace{i * i}_{\text{two ways}} + i, \quad i + i * \underbrace{i + i}_{\text{two ways}} + i, \quad i + i * i + \underbrace{i + i}_{\text{two ways}}, \quad i + i * i + i, \quad \underbrace{i + i * i + i}_{\text{five ways}}$$

For longer sentences the degree of ambiguity increases with no limit

CATALOG OF AMBIGUOUS FORMS AND REMEDIES

- 1) AMBIGUITY FROM BILATERAL RECURSION: $A \rightarrow A \dots A$

Design a grammar which is left recursive or right recursive but not both

Example 1: grammar G_1 generates $i + i + i$ in two different ways (check!).

$$G_1 : E \rightarrow E + E \mid i$$

But $L(G_1) = i (+i)^*$ is a regular language: other, simpler grammars are possible

Nonambiguous right-recursive grammar : $E \rightarrow i + E \mid i$

Nonambiguous left-recursive grammar : $E \rightarrow E + i \mid i$

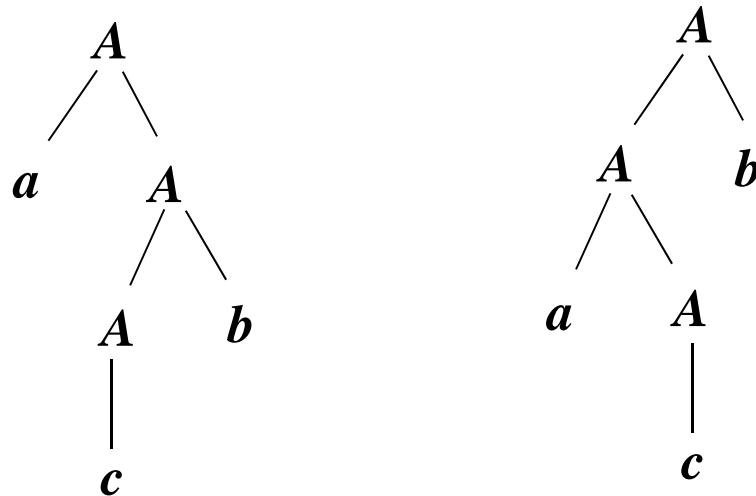
Example 2: Left and right recursion in different rules

$$L(G_2) = a^*cb^*$$

G_2 admits derivations where the a and b in a given sentence are obtained in any order

$$G_2 : A \rightarrow aA \mid Ab \mid c$$

Simplest example: two syntax trees for sentence acb



Example 2 (follows): Left and right recursion in different rules

First method to eliminate ambiguity:
Generate the two lists by distinct rules

$$\begin{aligned}L(G_2) &= a^* cb^* \\S &\rightarrow AcB \\A &\rightarrow aA \mid \epsilon \\B &\rightarrow bB \mid \epsilon\end{aligned}$$

Second method: force an ***order in the derivations***:
ex., first generate the a 's then the b 's

$$\begin{aligned}L(G_2) &= a^* cb^* \\S &\rightarrow aS \mid X \\X &\rightarrow Xb \mid c\end{aligned}$$

2) AMBIGUITY FROM LANGUAGE UNION

If $L_1 = L(G_1)$ and $L_2 = L(G_2)$ share some sentences (nonempty intersection)

The grammar G for the union language (slide n 10), is ambiguous (slide n.11)

A sentence $x \in L_1 \cap L_2$ admits two distinct derivations,

One with the rules of G_1 and the other with the rules of G_2

It is ambiguous for a grammar G that includes all the rules

Instead the sentences that belong to $L_1 \setminus L_2$ and to $L_2 \setminus L_1$ are nonambiguous

Remedy: provide disjointed set of rules for $L_1 \cap L_2$, $L_1 \setminus L_2$, and $L_2 \setminus L_1$

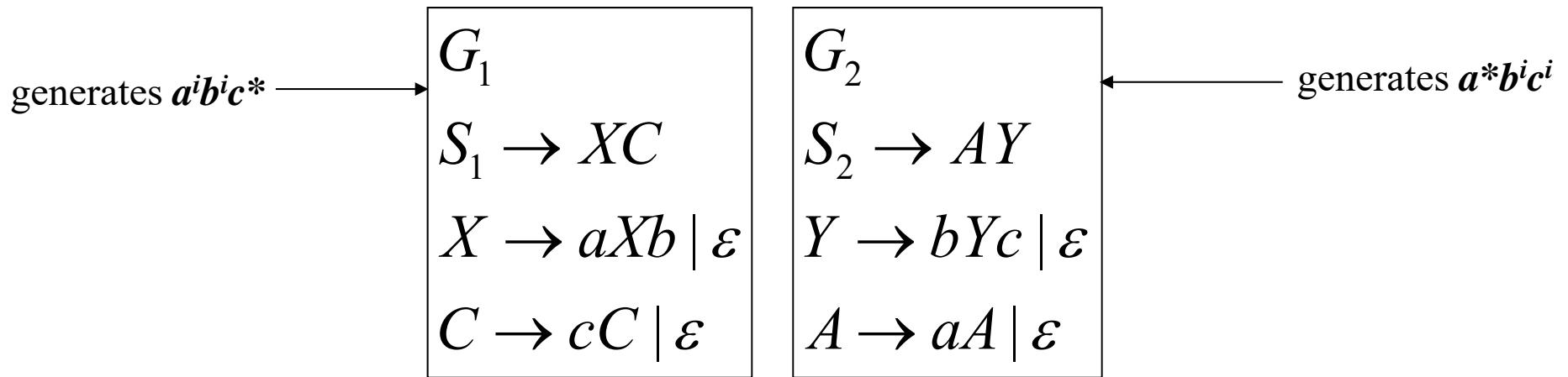
Examples in the textbook and in exam exercises

3) INHERENT AMBIGUITY

A language is INHERENTLY AMBIGUOUS if all its grammars are ambiguous
(for that language)

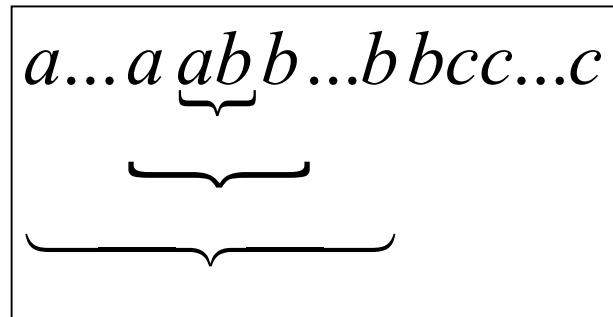
EXAMPLE:

$$L = \{a^i b^j c^k \mid i = j \vee j = k\} = \{a^i b^i c^* \mid i \geq 0\} \cup \{a^* b^i c^i \mid i \geq 0\} = L_1 \cup L_2$$

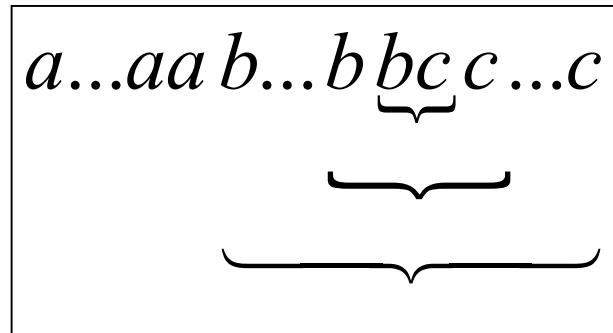


The union grammar is ambiguous for the sentences $\epsilon, abc, a^2b^2c^2, \dots$
(which constitute a non-context free language)

They are generated by G_1 which has rules ensuring that $|x|_a = |x|_b \dots$



... and also by G_2 , with rules ensuring that $|x|_b = |x|_c$



We *intuitively* argue that *any* grammar for L is ambiguous

Luckily inherent ambiguity is rare and can be avoided in technical languages

4) AMBIGUITY FROM CONCATENATION OF LANGUAGES

when the suffix of some sentence in the first language
is also a prefix of a sentence in the second one

G for the
concatenation of
 L_1 and L_2

$$G = (\Sigma_1 \cup \Sigma_2, \{S\} \cup V_{N_1} \cup V_{N_2}, \{S \rightarrow S_1 S_2\} \cup P_1 \cup P_2, S)$$

Supposing G_1 and G_2 are not ambiguous, G is ambiguous if $\exists x' \in L_1$ and $x'' \in L_2$ such that there is a non empty string v such that

$$\begin{aligned} x' &= u'v \wedge u' \in L_1 & x'' &= vz'' \wedge z'' \in L_2 \\ u'vz'' &\in L_1.L_2 \quad \text{and is ambiguous:} \\ S &\Rightarrow S_1 S_2 \xrightarrow{+} u' S_2 \xrightarrow{+} u' v z'' \\ S &\Rightarrow S_1 S_2 \xrightarrow{+} u' v S_2 \xrightarrow{+} u' v z'' \end{aligned}$$

Example – Ambiguity in the concatenation of Dyck languages

$$\Sigma_1 = \{a, a', b, b'\} \quad \Sigma_2 = \{b, b', c, c'\}$$

$$aa'bb'cc' \in L = L_1L_2$$

$$G(L): \quad S \rightarrow S_1S_2$$

$$S_1 \rightarrow aS_1a'S_1 \mid bS_1b'S_1 \mid \varepsilon$$

$$S_2 \rightarrow bS_2b'S_2 \mid cS_2c'S_2 \mid \varepsilon$$

$$\overbrace{aa'}^{\overbrace{S_1}{}^{}}\overbrace{bb'}^{\overbrace{S_2}{}^{}}\overbrace{cc'}^{\overbrace{S_2}{}^{}}$$

To prevent ambiguity one must avoid the shift of the substring from the suffix in the first language to the prefix in the second one

Easy solution: insert a ***new*** terminal symb. (e.g. '#') acting as a separator

new terminal symb.: the symbol may not belong to any of the two alphabets

$L_1 \# L_2$ is generated by the «concatenation grammar» with the axiomatic rule $S \rightarrow S_1 \# S_2$

The language is however modified

5) OTHER CASES OF AMBIGUITY

AMBIGUOUS REGULAR EXPRESSIONS

every sentence with two or more
 c is ambiguous

$$\begin{array}{l} S \rightarrow DcD \quad D \rightarrow bD \mid cD \mid \epsilon \\ \{b,c\}^* c \{b,c\}^* \end{array}$$

REMEDY: identify the leftmost c

$$S \rightarrow BcD \quad B \rightarrow bB \mid \epsilon \quad D \rightarrow bD \mid cD \mid \epsilon$$

OTHER CASE: LACK OF ORDER IN DERIVATIONS

Example: Every phrase with
 >2 b 's is ambiguous

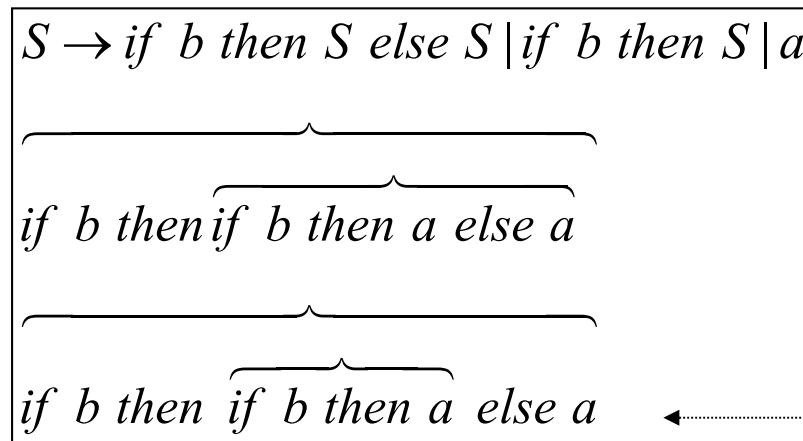
$$\begin{array}{l} S \rightarrow bSc \mid bbSc \mid \epsilon \\ S \Rightarrow bbSc \Rightarrow bbbScc \Rightarrow bbbcc \\ S \Rightarrow bSc \Rightarrow bbbScc \Rightarrow bbbcc \end{array}$$

REMEDY: Impose an order in the derivation
First the rule that generates balanced b 's and c 's
Then the one that generates excess b 's

$$S \rightarrow bSc \mid D \quad D \rightarrow bbDc \mid \epsilon$$

6) AMBIGUITY IN CONDITIONAL PHRASES:

example of «historical interest»: the (in)famous *dangling else* problem (Pascal, C, ...)


$$S \rightarrow S_E \mid S_T$$
$$S_E \rightarrow if\ b\ then\ S_E\ else\ S_E\ | a$$
$$S_T \rightarrow if\ b\ then\ S_E\ else\ S_T\ | if\ b\ then\ S$$

REMEDY 1 Rule out this interpretation (i.e., use the *closest match interpretation*);
two n.t., S_E and S_T :
 S_E always has the *else* (cannot have only the *then*),
only for S_T it is possible not to have the *else*
therefore only S_E can precede the *else*

$$S \rightarrow if\ b\ then\ S\ else\ S\ endif$$
$$| if\ b\ then\ S\ endif\ | a$$

REMEDY 2 Modify the language,
introduce a closing mark *endif*

Context Free Grammars - III

Prof. A. Morzenti

STRONG (STRUCTURAL) AND WEAK EQUIVALENCE

WEAK EQUIVALENCE: two grammars are weakly equivalent if they generate the same language: $L(G) = L(G')$.

G and G' might assign different structures (syntax trees) to the same sentence

the structure assigned to a sentence is important: it is used by translators and interpreters

STRONG or STRUCTURAL EQUIVALENCE of two grammars G and G'

$L(G) = L(G')$ (weak eq.) **and**

G and G' have the same **condensed skeleton trees**

strong eq. \rightarrow weak eq. **but** strong eq. \neq weak eq.

(hence \neg (weak eq. \rightarrow strong eq.))

strong eq. is **decidable**

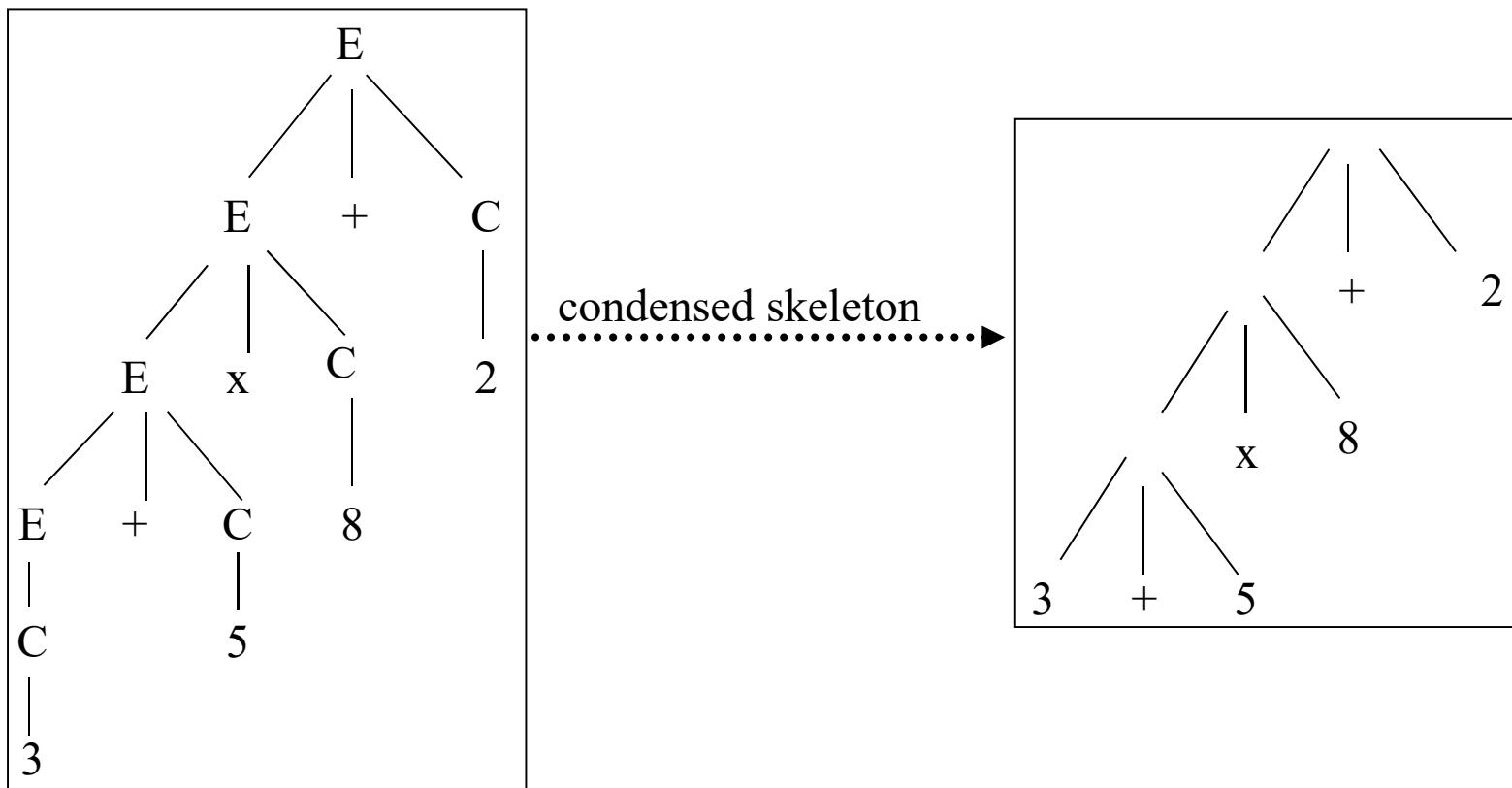
weak eq. is **not decidable**

(it can happen that one can establish that G_1 and G_2 **are not** strongly equivalent

but is unable to establish whether they are weakly equivalent or not)

Example: Structural equivalence of arithmetic expressions: $3 + 5 \times 8 + 2$

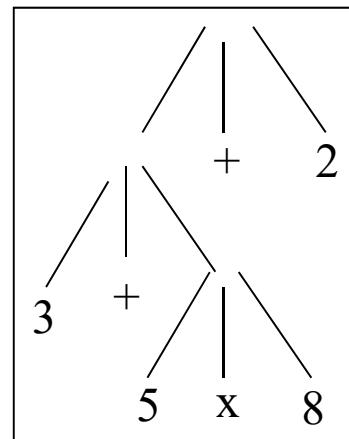
$$G_1 : \begin{array}{l} E \rightarrow E + C \quad E \rightarrow E \times C \quad E \rightarrow C \\ C \rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 \end{array}$$



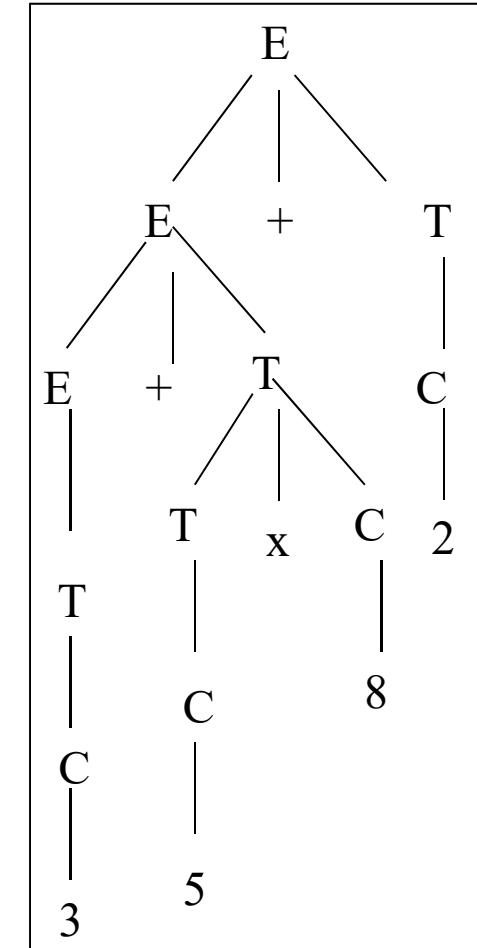
ANOTHER GRAMMAR FOR THE SAME LANGUAGE

NB: copy (or categorization) rules

: those are the rules that just lead one non-terminal to another non-terminal

$$G_2 : \begin{aligned} E &\rightarrow E + T \quad (E \rightarrow T) \quad T \rightarrow T \times C \quad (T \rightarrow C) \\ C &\rightarrow 0|1|2|3|4|5|6|7|8|9 \end{aligned}$$


condensed skeleton



G_1 and G_2 are not structurally equivalent

Semantic interpretations: $G_1: (((3 + 5) \times 8) + 2)$

$G_2: ((3 + (5 \times 8)) + 2)$

Only G_2 is structurally adequate w.r.t. the operator precedence rules (NB: G_2 is more complex): it forces the generation of a product from n.t. T only after E therefore it assigns a higher precedence to the product than to the sum
 (with respect to)

NB: this is how operator priority can be enforced through grammars

STRUCTURAL ADEQUACY must be carefully considered

it supports syntax-directed interpretation and translation

must represent correctly the operator precedence that we want to impose

another grammar, G_3 , structurally equivalent to the previous one

(NB structural equivalence concerns *condensed trees*):

$$\begin{array}{l} E \rightarrow E + T \mid T + T \mid C + T \mid E + C \mid T + C \mid C + C \mid T \times C \mid C \times C \mid C \\ T \rightarrow T \times C \mid C \times C \mid C \\ C \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{array}$$

G_3 has more rules: it does not exploit categorization rules as G_2 does

categorization can reduce the complexity of grammars

GRAMMAR NORMAL FORMS AND TRANSFORMATION

writing grammars in restricted way

Normal forms constrain the rules without reducing the family of generated languages

They are useful for both proving properties and for language design

Let us see some transformations useful both to

- obtain an equivalent normal form
- design the syntax analyzers

EXPANSION of a nonterminal (to ELIMINATE it from the rules where it appears)

In the example, we eliminate nonterm. B

Grammar $A \rightarrow \alpha B \gamma \quad B \rightarrow \beta_1 | \beta_2 | \dots | \beta_n$

Becomes $A \rightarrow \alpha \beta_1 \gamma | \alpha \beta_2 \gamma | \dots | \alpha \beta_n \gamma$

Derivation $A \Rightarrow \alpha B \gamma \Rightarrow \alpha \beta_i \gamma$

Becomes $A \Rightarrow \alpha \beta_i \gamma$

ELIMINATION OF THE AXIOM FROM RIGHT PARTS :

It is always possible to obtain right part of rules as strings $\in (\overbrace{\Sigma \cup (V \setminus \{S\})}^{\text{alphabet containing terminal + non terminal without string } S})$

Simply introduce a new axiom S_0 and the rule $S_0 \rightarrow S$

NULLABLE NONTERMINALS AND ELIMINATION OF EMPTY RULES

a nonterminal is *nullable* iff there exists a derivation: $A \xrightarrow{+} \varepsilon$

$Null \subseteq V$ is the set of nullable n.t.

computation of the set $Null$

$A \in Null$ if $A \rightarrow \varepsilon \in P$

$A \in Null$ if $(A \rightarrow A_1 A_2 \dots A_n \in P \text{ with } A_i \in V \setminus \{A\}) \wedge \forall A_i (A_i \in Null)$

NB: recursive rules cannot be used

Example – Computing nullable nonterminals

$$S \rightarrow SAB \mid AC \quad A \rightarrow aA \mid \varepsilon \quad B \rightarrow bB \mid \varepsilon \quad C \rightarrow cC \mid c$$
$$Null = \{A, B\}$$

NB: If the grammar included rule $S \rightarrow AB$ then the result would include $S \in Null$

NORMAL FORM WITHOUT NULLABLE NONTERMS

that is: no nonterminal other than the axiom is nullable

In that case the axiom is nullable only if the empty string ε is in the language

(NB: If $\varepsilon \in L$ then the axiom is necessarily nullable)

CONSTRUCTION OF THE NON-NULLABLE NORMAL FORM:

- 1) compute the *Null* set
- 2) for each rule $\in P$ add as alternatives those obtained by deleting, in the right part, **in all possible ways (NB: combinatorial effect)**, the nullable nonterm.
- 3) remove all empty rules $A \rightarrow \varepsilon$, except for $A = S$
- 4) clean the grammar and remove any circularity

Example (follows)

delete A and B in all possible ways,
remove rules of type $X \rightarrow \varepsilon$

delete circular rule $S \rightarrow S$

<i>Nullable</i>	original G	G'' to be cleaned	G'' with no empty rules
F	$S \rightarrow SAB \mid$ $\quad\quad\quad AC$	$S \rightarrow SAB \mid SA \mid SB \mid$ $\quad\quad\quad S \mid AC \mid C$	$S \rightarrow SAB \mid SA \mid SB \mid$ $\quad\quad\quad AC \mid C$
T	$A \rightarrow aA \mid \varepsilon$	$A \rightarrow aA \mid a$	$A \rightarrow aA \mid a$
T	$B \rightarrow bB \mid \varepsilon$	$B \rightarrow bB \mid b$	$B \rightarrow bB \mid b$
F	$C \rightarrow cC \mid c$	$C \rightarrow cC \mid c$	$C \rightarrow cC \mid c$

COPY (or CATEGORIZATION) RULES AND THEIR ELIMINATION

A typical example: $\text{iterative_phrase} \rightarrow \text{while_phrase} \mid \text{for_phrase} \mid \text{repeat_phrase}$

NB: copy rules factorize common parts \Rightarrow they reduce the grammar size

That's why they are often present in grammars of technical languages

However copy elimination shortens derivations and reduces the height of syntax trees

A typical *tradeoff*

Define $\text{Copy}(A) \subseteq V$

set of n.t. into which the n.t. A can be copied, possibly transitively

$$\text{Copy}(A) = \{B \in V \mid \text{there exists a derivation } A \xrightarrow{*} B\}$$

1) Computation of ***Copy*** (assume a **grammar with non empty rules**) by applying logical clauses until a fixpoint is reached

(it is simply the reflexive, transitive closure of the «copy» relation defined by copy rules)

$$A \in \text{Copy}(A)$$

$$C \in \text{Copy}(A) \text{ if } (B \in \text{Copy}(A)) \wedge (B \rightarrow C \in P)$$

(NB: since there is no empty rule, cases such as $B \rightarrow CD$ and $D \xrightarrow{*} \varepsilon$ can be ruled out)

2) Definition of the rules of a grammar G' , equivalent to G but without copy rules

$$P' := P \setminus \{A \rightarrow B \mid A, B \in V\} \quad \text{--cancel copy rules}$$

$$P' := P' \cup \{A \rightarrow \alpha \mid \exists B (B \in \text{Copy}(A) \wedge (B \rightarrow \alpha) \in P)\} \quad \text{--add new "compensating" rules}$$

(NB: for each removed copy rule, **many** other (non-copy) rules may be added
 \Rightarrow the set of rules may increase considerably in size

The derivation $A \xrightarrow{*} B \xrightarrow{*} \alpha$ shrinks to $A \xrightarrow{*} \alpha$

Example – Eliminating copy rules from a grammar of arithmetic expressions

$$\begin{array}{c}
 E \rightarrow E + T \mid T \quad T \rightarrow T \times C \mid C \\
 \text{---} \\
 C \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\
 \text{---} \\
 \text{Copy}(E) = \{E, T, C\} \quad \text{Copy}(T) = \{T, C\} \\
 \text{Copy}(C) = \{C\}
 \end{array}$$

Equivalent grammar without copy rules: remove rules $E \rightarrow T$ and $T \rightarrow C$, then ...

$$\begin{array}{c}
 T \in \text{Copy}(E) \qquad \qquad \qquad C \in \text{Copy}(E) \\
 \text{---} \\
 E \rightarrow E + T \mid T \times C \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\
 \text{---} \\
 T \rightarrow T \times C \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\
 \text{---} \\
 C \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
 \end{array}$$

Conversion of Left recursions to Right recursions

Grammars with no left recursion (l-recursion) are necessary for designing **top down parsers**

Case 1 (simple): Conversion of **IMMEDIATE L-RICURSIONS**

Introduce a new n.t. A'
new

$$\left\{ \begin{array}{l} A \rightarrow A\beta_1 | A\beta_2 | \dots | A\beta_h \\ \text{where no } \beta_i \text{ is empty} \\ A \rightarrow \gamma_1 | \gamma_2 | \dots | \gamma_k \end{array} \right.$$
$$\left\{ \begin{array}{l} A \rightarrow \gamma_1 A' | \gamma_2 A' | \dots | \gamma_k A' | \gamma_1 | \gamma_2 | \dots | \gamma_k \\ A' \rightarrow \beta_1 A' | \beta_2 A' | \dots | \beta_h A' | \beta_1 | \beta_2 | \dots | \beta_h \end{array} \right.$$

in the “new” derivation the prefix γ is generated **first**, the succeeding parts of type β **afterwards**
left recursion: string generated from the **right**; **right** recursion: string generated from the **left**;

derivation in the grammar with l-recursion $\rightarrow A \Rightarrow A\beta_2 \Rightarrow A\beta_3\beta_2 \Rightarrow \gamma_1\beta_3\beta_2$

deriv. in the gramm. without l-recursion $\rightarrow A \Rightarrow \gamma_1 A' \Rightarrow \gamma_1\beta_3 A' \Rightarrow \gamma_1\beta_3\beta_2$

Example – Conversion from l-recursion to r-recursion for arithmetic expressions

$$E \rightarrow E + T \mid T \quad T \rightarrow T * F \mid F \quad F \rightarrow (E) \mid i$$

E and T are left immediately recursive

$$E \rightarrow TE' \mid T \quad E' \rightarrow +TE' \mid +T$$

$$T \rightarrow FT' \mid F \quad T' \rightarrow *FT' \mid *F \quad F \rightarrow (E) \mid i$$

Case 2 : non immediate left recursion

it is more complex, not treated here,
see textbook, §2.5.13.8

da qui in poi ha skippato poichè sono solo di interesse per la teoria dei linguaggi formali ma non hanno applicazioni pratiche

CHOMSKY NORMAL FORM: two types of rules

1. *homogeneous binary rules:* $A \rightarrow BC$ with $B,C \in V$
2. *Terminal rules with singleton right part* $A \rightarrow a$, $a \in \Sigma$

NB: Syntax trees have internal nodes of degree 2 and leaf parent nodes of degree 1

Procedure to obtain from G (assumed without nullable n.t.) its Chomsky normal form

If the empty string is in the language, add rule: $S \rightarrow \epsilon$

Then apply iteratively the following process

for each rule of type $\longrightarrow A_0 \rightarrow A_1 A_2 \dots A_n$

add a rule of type ... $\longrightarrow A_0 \rightarrow < A_1 > < A_2 \dots A_n >$ $< A_2 \dots A_n >$ a new ancillary n.t.
introduced for this purpose

... and also another rule ... $\longrightarrow < A_2 \dots A_n > \rightarrow A_2 \dots A_n$

finally, soon or late ... \longrightarrow if A_1 is terminal $< A_1 > \rightarrow A_1$

Example: Conversion to Chomsky normal form

$$S \rightarrow dA \mid cB \quad A \rightarrow dAA \mid cS \mid c \quad B \rightarrow cBB \mid dS \mid d$$
$$S \rightarrow < d > A | < c > B$$
$$A \rightarrow < d > < AA > | < c > S | c$$
$$B \rightarrow < c > < BB > | < d > S | d$$
$$< d > \rightarrow d \quad < c > \rightarrow c$$
$$< AA > \rightarrow AA \quad < BB > \rightarrow BB$$

Real-time and Greibach normal forms

Real-time normal form: the right part of any rule has a terminal symbol as a prefix

$$A \rightarrow a\alpha \text{ with } a \in \Sigma, \alpha \in \{\Sigma \cup V\}^*$$

Greibach normal form is a special case:

Every right part consists of a terminal followed by zero or more nonterminals

$$A \rightarrow a\alpha \text{ with } a \in \Sigma, \alpha \in V^*$$

“REAL - TIME” - the term refers to a property of syntax analysis:

Every step reads and consumes one terminal symbol

Number of steps of the analysis = length of the string

Context Free Grammars - IV

Prof. A. Morzenti

FREE GRAMMARS EXTENDED WITH REGULAR EXPRESSIONS

(Extended BNF or EBNF or Regular Right Part Grammars-RPPG)

backus normal form

MORE READABLE thanks to the star and cross (iteration) and choice (union) operators

EBNF's allow for the definition of SYNTAX DIAGRAMS which can be viewed as a blueprint of the syntax analyzer flowchart

NB: The **CF** family is closed under all regular operations, therefore the generative power of EBNF is the same as that of BNF

EBNF GRAMMAR $G = \{ V, \Sigma, P, S \}$

Exactly $|V|$ rules, each in the form $A \rightarrow \eta$, with η r.e. over alphabet $V \cup \Sigma$

Terminal \cup Non-Terminal


Example: Algol-like Language

B: block; D: declaration; I=Imperative part;
 F=phrase; a=assignment; c=char; i=int;
 r=real; v=variable; b=begin; e=end

$$\begin{aligned} B &\rightarrow b[D]Ie \\ D &\rightarrow ((c|i|r)v(,v)^*)^+ \\ I &\rightarrow F(;F)^* \\ F &\rightarrow a | B \end{aligned}$$

Example of a Declaration section

char text1, text2; real temp, result;
int alpha, beta2, gamma;

Alternative BNF grammar for D :

$$D \rightarrow DE \mid E \quad E \rightarrow AF; \quad A \rightarrow c \mid i \mid r \quad F \rightarrow v, F \mid v$$

The presence of the hierarchical nested lists is hidden in the recursion

.

The BNF grammar is longer and obviously less readable

it conceals the existence of two hierarchically nested lists

Furthermore, the choice of nonterminal symbol names (A, E, F) can be arbitrary

DERIVATIONS AND TREES IN EXTENDED FREE GRAMMARS

Derivation in EBNF G defined by considering an equivalent BNF G' with infinite rules

G includes $A \rightarrow (aB)^+$

G' includes $A \rightarrow aB \mid aBaB \mid aBaBaB \mid \dots$

DERIVATION RELATION FOR EBNF G :

Given strings η_1 and $\eta_2 \in (\Sigma \cup V)^*$

η_2 is said to be *derived* immediately in G from η_1

$\eta_1 \Rightarrow \eta_2$, if the two strings can be factorized as:

$\eta_1 = \alpha A \gamma, \quad \eta_2 = \alpha \vartheta \gamma$ and there exists a rule

$A \rightarrow e^*$ such that the r.e. e admits the derivation $e \Rightarrow \vartheta$

Notice that η_1 and η_2 do not contain
r.e. operators nor parenthesis.

Only string e is a r.e. but it does not appear in the derivation if it is not terminal

Example: Extended derivation for arithmetic expressions

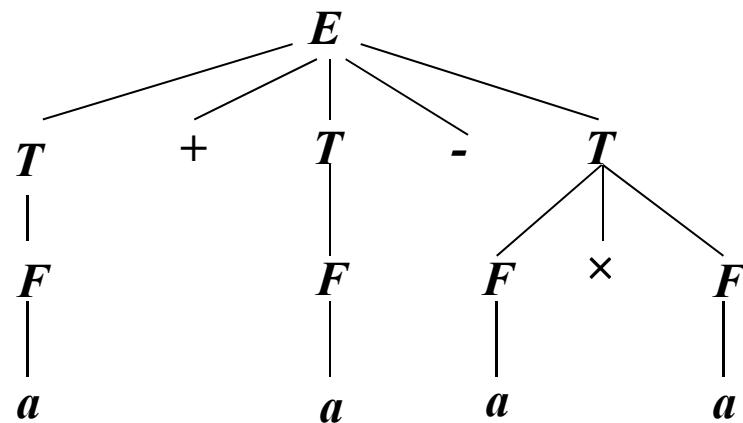
Extended grammar for arithmetic expressions with four infix operators, parentheses and variable a .

$$E \rightarrow [+ | -] T ((+ | -) T)^* \quad \text{expression}$$
$$T \rightarrow F ((\times | /) F)^* \quad \text{term}$$
$$F \rightarrow a \mid ' (E ') \quad \text{factor}$$

We still have expression but just a indirect one, which represent that we can have an expression into another expression

left derivation:

$$\begin{aligned} E &\Rightarrow T + T - T \Rightarrow F + T - T \Rightarrow a + T - T \Rightarrow a + F - T \Rightarrow \\ &\Rightarrow a + a - T \Rightarrow a + a - F \times F \Rightarrow a + a - a \times F \Rightarrow a + a - a \times a \end{aligned}$$



unbounded node degree

the **tree** is in general **wider** ...

... and **reduced in depth**

GRAMMARS OF REGULAR LANGUAGES

Regular languages are a special case of free languages

(reminder: we use the term «free» as an abbreviation for «context-free»)

They are generated by grammars with strong constraints on the form of rules

Due to these constraints the sentences of regular languages present «inevitable» repetitions

FROM REGULAR EXPRESSIONS TO CONTEXT-FREE GRAMMARS

Define the form of the rules depending on that of the regular expression

Notice that recursive rules match iterative operators (star '*' and cross '+')

REGULAR EXPRESSION

1. $r = r_1.r_2....r_k$
2. $r = r_1 \cup r_2 \cup \cup r_k$
3. $r = (r_1)^*$
4. $r = (r_1)^+$
5. $r = b \in \Sigma$
6. $r = \epsilon$

FORM OF E IN RULE $R \rightarrow E$

1. $E = E_1E_2...E_k$
2. $E = E_1 \cup E_2 \cup ... \cup E_k$
3. $E = EE_1 \mid \epsilon$ or $E = E_1E \mid \epsilon$
4. $E = EE_1 \mid E_1$ or $E = E_1E \mid E_1$
5. $E = b$
6. $E = \epsilon$

we said that regular expression are derived by others regular expressions by using the operators

Example

$$E = (abc)^* \cup (ff)^+$$

$E = E_1 \cup E_2$	hence	$E \rightarrow E_1 E_2$
$E_1 = (E_3)^*$	hence	$E_1 \rightarrow E_1 E_3 \epsilon$
$E_3 = abc$	hence	$E_3 \rightarrow abc$
$E_2 = (E_4)^+$	hence	$E_2 \rightarrow E_2 E_4 E_4$
$E_4 = ff$	hence	$E_4 \rightarrow ff$

We can therefore conclude that every regular language is free

But there are free languages that are not regular (e.g. palindromes)

$$\text{REG} \subset \text{CF} \quad (\text{REG} \subseteq \text{CF} \text{ and } \text{REG} \neq \text{CF})$$

Let us look for the subclass of free grammars that are equivalent to regular expressions



a first candidate: **LINEAR GRAMMARS**

A linear grammar has **at most one nonterm.** in its right part

$$A \rightarrow uBv \quad \text{with} \quad u, v \in \Sigma^*, B \in (V \cup \epsilon)$$

↑
non-terminal symbol

The family of linear grammars is however **still more powerful than regular languages.**

Example: non regular linear language

$$L_1 = \{a^n c^n \mid n \geq 1\} = \{ac, aacc, \dots\} \rightarrow \text{non regular (requires counting)}$$

$$S \rightarrow aSc \mid ac \rightarrow \text{same as above, but expressed}$$

as LINEAR GRAMMAR

UNILINEAR GRAMMARS (called «of type 3» in the Chomsky hierarchy)

RIGHT-LINEAR RULE:

$$A \rightarrow uB \quad \text{with} \quad u \in \Sigma^*, \quad B \in (V \cup \epsilon)$$

LEFT-LINEAR RULE:

$$A \rightarrow Bv \quad \text{with} \quad v \in \Sigma^*, \quad B \in (V \cup \epsilon)$$

A grammar is **unilinear** iff its rules are either **all right-linear** or **all left-linear**

Syntax trees grow totally unbalanced (resp. toward the right or left)

Without loss of generality one can require that a unilinear grammar has

- **STRICTLY unilinear rules**: with at most one terminal $A \rightarrow aB, \quad a \in (\Sigma \cup \epsilon) \quad B \in (V \cup \epsilon)$
- all **terminal rules are empty**: e.g., rule $B \rightarrow b$ replaced by rules $B \rightarrow bB'$ and $B' \rightarrow \epsilon$

Therefore we can assume (for the right case) just rules of type $A \rightarrow aB \mid \epsilon$ with $a \in \Sigma, B \in V$

(for the left case rules of type $A \rightarrow Ba \mid \epsilon$ with $a \in \Sigma, B \in V$)

Regular expressions can be translated into (strictly) unilinear grammars
(not proven here: it can be shown using finite state automata, see Chap.3)

Therefore

REG \subseteq UNILIN

Example: strings with substring aa and ending with b defined by (ambiguous) r.e.

$$(a \mid b)^* aa (a \mid b)^* b$$

1. right-linear grammar G_r

$$S \rightarrow aS \mid bS \mid aaA \quad A \rightarrow aA \mid bA \mid b$$

2. left-linear gramm. G_l

$$\begin{aligned} S &\rightarrow Ab \quad A \rightarrow Aa \mid Ab \mid Baa \\ &\qquad B \rightarrow Ba \mid Bb \mid \varepsilon \end{aligned}$$

3. non-unilinear equiv. gramm.

$$E_1 \rightarrow E_2 aa E_2 b \quad E_2 \rightarrow E_2 a \mid E_2 b \mid \varepsilon$$

FROM UNILINEAR GRAMMAR TO THE REGULAR EXPRESSION: LINEAR LANGUAGE EQUATIONS

We show that from any unilinear grammar one can obtain an equivalent regular expression

Therefore **UNILIN** \subseteq **REG**

hence (by the previous -still unproved- property **REG** \subseteq **UNILIN**, p.10) **UNILIN** = **REG**

Rules of the unilinear right grammar can be seen as equations . . .

. . . where unknowns: the languages generated by every nonterminal : ex. L_S, L_A, L_B, \dots

Let \mathbf{G} be

- strictly unilinear right (e.g.) and
- (with no loss of generality) with all terminal rules empty ($A \rightarrow \varepsilon$)

$x \in \Sigma^*$ is in L_A in the following cases:

1. x is the empty string (rule in $P: A \rightarrow \varepsilon$);
2. $x = ay$ (rule in $P: A \rightarrow aB$ and $y \in L_B$)

for every n.t. A_0 defined by $A_0 \rightarrow a_1A_1 \mid a_2A_2 \mid \dots \mid a_kA_k \mid \varepsilon$

$$L_{A_0} = a_1L_{A_1} \cup a_2L_{A_2} \cup \dots \cup a_kL_{A_k} \cup \varepsilon$$

therefore we obtain a system of $n = |V|$ equations in n unknowns

to be solved with the method of substitution + applying the *Arden identity*

Arden Identity

1

Equation $X = KX \cup L$ (with K nonempty lang. and L any language)

has exactly one solution

$$X = K^*L$$

$$(K^*L = KK^*L \cup L)$$

NB: we do not prove unicity

2

2 is a solution: this expr. is obtained by substituting 2 in 1

Example: Language Set Equations

Grammar for a list of (possibly missing) elements e divided by separator s

$$S \rightarrow sS \mid eA \quad A \rightarrow sS \mid \epsilon$$

is transformed into a system of equations:

subst. second eq. in the first one

distrib. prop. of concat. over \cup then
factorize L_s as a common suffix

$$\begin{cases} L_S = sL_S \cup eL_A \\ L_A = sL_S \cup \epsilon \end{cases} \xrightarrow{\text{subst. second eq. in the first one}} \begin{cases} L_S = sL_S \cup e(sL_S \cup \epsilon) \\ L_A = sL_S \cup \epsilon \end{cases} \xrightarrow{\text{distrib. prop. of concat. over } \cup \text{ then factorize } L_s \text{ as a common suffix}} \begin{cases} L_S = (s \cup es)L_S \cup e \\ L_A = sL_S \cup \epsilon \end{cases}$$

applying Arden identity

$$\begin{cases} L_S = (s \cup es)^* e \\ L_A = sL_S \cup \epsilon \quad L_A = s(s \cup es)^* e \cup \epsilon \end{cases}$$

COMPARISON OF REGULAR AND CONTEXT-FREE LANGUAGES

We introduce some properties useful to show that some languages are (not) regular
regular languages (and therefore unilinear grammars) exhibit *inevitable repetitions*

PROPERTY: Let G be a unilinear grammar

Every sufficiently long sentence x (i.e., longer than a grammar-dependent constant k)

Can be factorized as $x = t u v$ - with nonempty u

so that, $\forall n \geq 1$, the string $t u^n v \in L(G)$

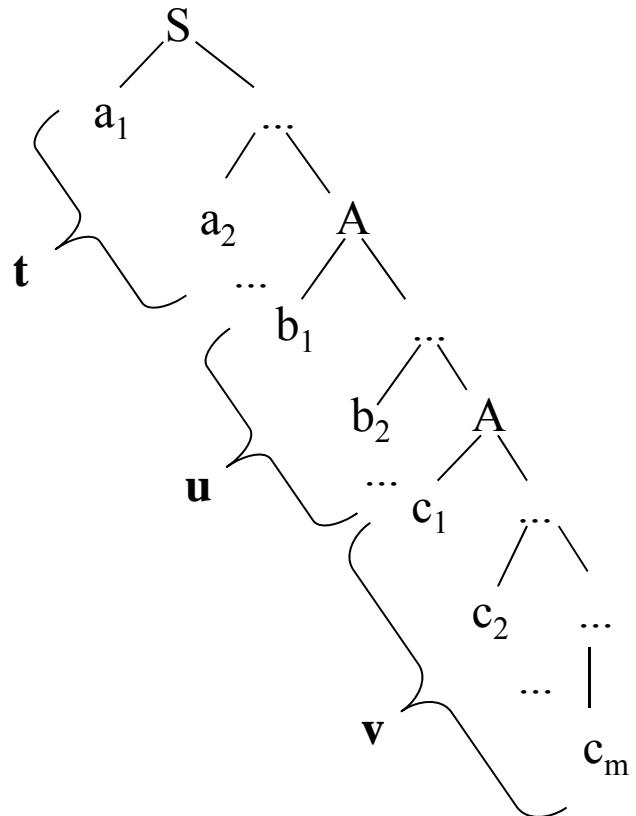
(the sentence can be «pumped» by injecting string u an arbitrary number of times)

NOTE the analogy with the *pumping lemma* of finite state automata...

Proof:

Consider a strictly right-linear G with k n.t. symbols

In the derivation of a sentence x whose length is k or more,
there is necessarily a n.t. A that appears at least two times



$$\boxed{t = a_1 a_2 \dots, \quad u = b_1 b_2 \dots \quad v = c_1 c_2 \dots c_m}$$
$$S \xrightarrow{+} t A \xrightarrow{+} t u A \xrightarrow{+} t u v$$

then it is also possible to derive tv , $tuuv$ etc.

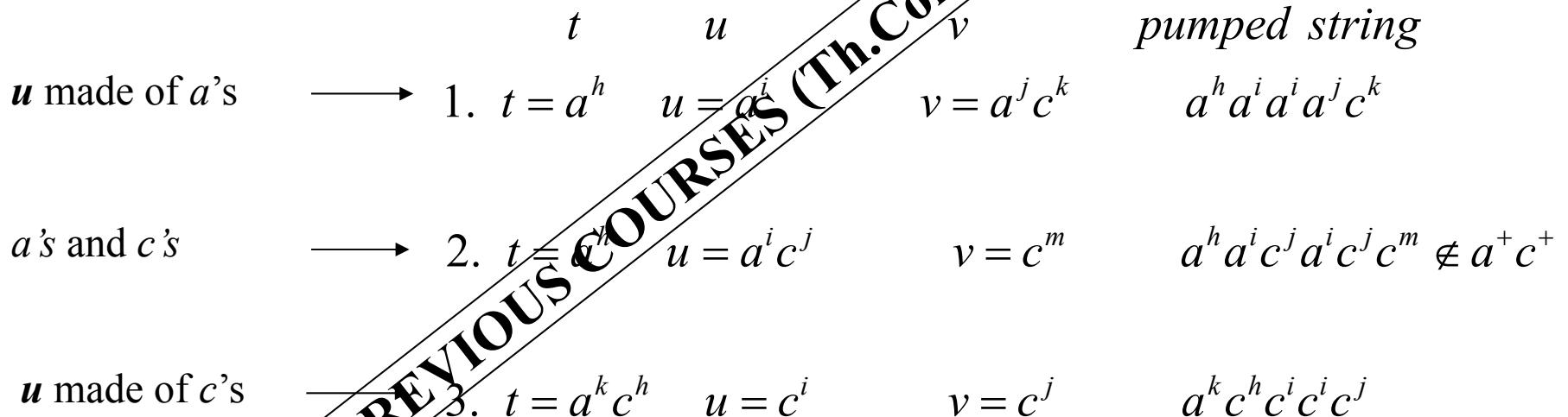
We use this property to show that the language with two equal exponents

$L_1 = \{ a^n c^n \mid n \geq 1 \}$ is **not** regular

Let us assume by contradiction that it is regular: then

NB: this k is the one of the previous proof (previous slide.)

$x = a^k c^k = tuv$ with u non empty, and there are three ways to take u



In all cases the pumped string (which would be in the language if this was regular), does not have the required form $a^n c^n$

THE ROLE OF SELF-NESTED DERIVATIONS

$$A \xrightarrow{+} uAv \quad u \neq \epsilon \wedge v \neq \epsilon$$

Self-nested derivations are

absent from unilinear grammars of regular languages

present in grammars of free nonregular languages

(palindromes, Dyck languages, language with equal exponents, ...)

lack of self-nested derivations allows one to solve lang. equations of unilinear grams.

Hence ...

GRAMMAR WITHOUT SELF-NESTED DERIVATIONS



REGULAR LANGUAGE

NB: the converse does not necessarily hold:

a grammar with self-nested derivations may generate a regular language

Example:

Self-nesting grammar generating a regular language

$$G = \{S \rightarrow aSa \mid \epsilon\}$$

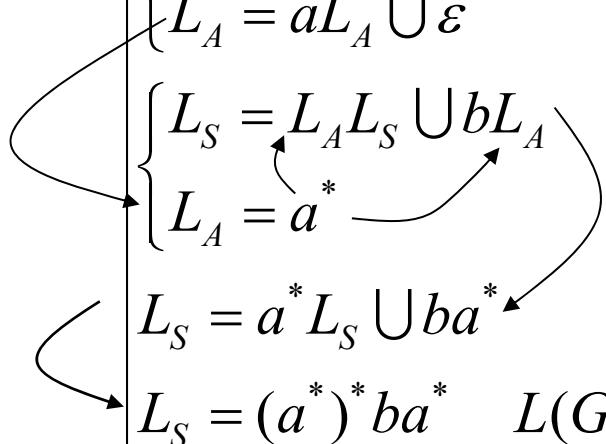
$$S \Rightarrow aSa \quad L(G) = (aa)^*$$

NB: the language is regular
it is also defined by
 $G_d = \{S \rightarrow aaS \mid \epsilon\}$

Example:

grammar **without self-nesting**

although **not linear**: unilinear gramm.
is a sufficient, not necessary condition
for generating a regular language

$$\begin{aligned} G: \quad & S \rightarrow AS \mid bA \quad A \rightarrow aA \mid \epsilon \\ & \left\{ \begin{array}{l} L_S = L_A L_S \cup bL_A \\ L_A = aL_A \cup \epsilon \end{array} \right. \\ & \left\{ \begin{array}{l} L_S = L_A L_S \cup bL_A \\ L_A = a^* \end{array} \right. \\ & L_S = a^* L_S \cup ba^* \\ & L_S = (a^*)^* ba^* \quad L(G) = a^* ba^* \end{aligned}$$


LIMITS OF CONTEXT-FREE LANGUAGES

In context-free languages all sufficiently long sentences necessarily contain

two substrings that can be repeated arbitrarily many times,

thus originating self-nested structures

This hinders the derivation of strings with *three or more* parts that are repeated the same number of times

NB: we do not provide a proof: see the textbook, §2.7.1

LIMITS OF CONTEXT-FREE LANGUAGES

THE LANGUAGE OF THREE EQUAL POWERS IS NOT FREE

$$L = \{a^n b^n c^n \mid n \geq 1\}$$

LANGUAGE OF COPIES OR REPLICA IS NOT FREE

(NB: this property is not proved here)

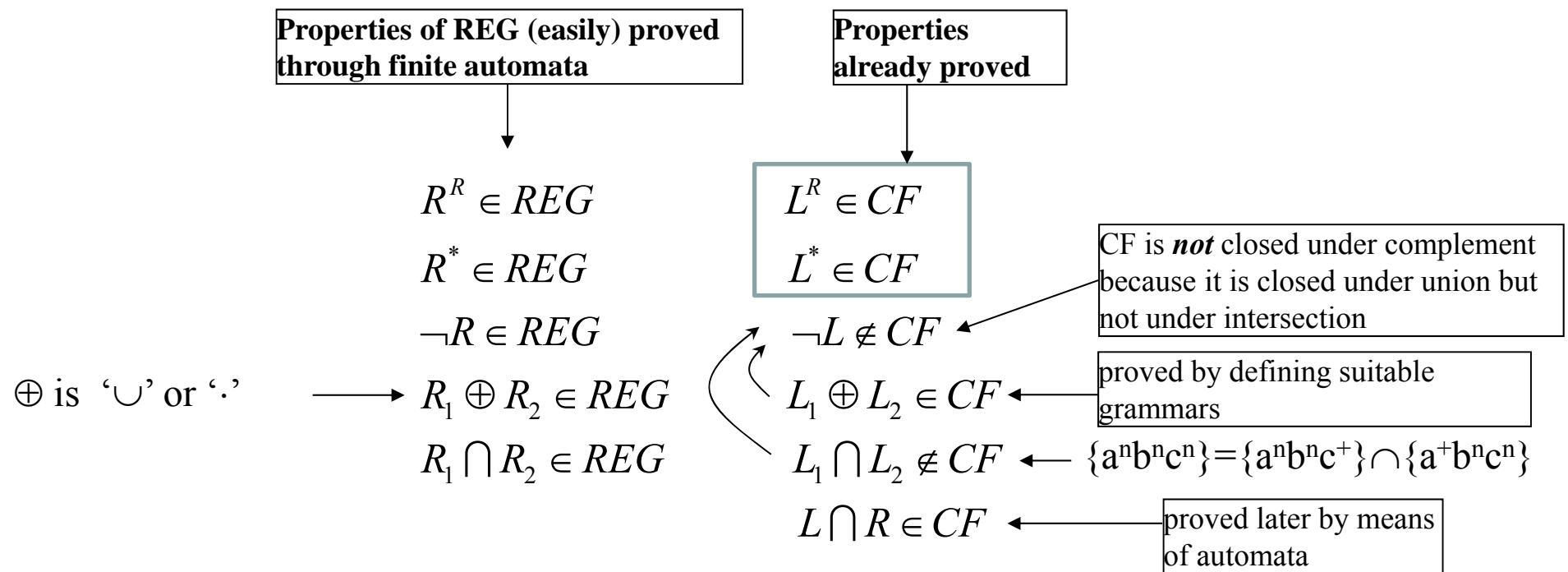
Very frequent in technical languages, when elements in two or more lists must be identical or be in some correspondence (ex. Formal and actual parameters)

$$L_{replica} = \{uu \mid u \in \Sigma^+\}$$

$$\Sigma = \{a, b\} \quad x = abbbabbb$$

CLOSURE PROPERTIES OF **REG** (regular) AND **CF** (context free) FAMILIES

Let languages $L \in CF$ and $R \in REG$:



NB: The «non-inclusion» symbol \notin means: «there exist some language not in the family»
 (not: «there is no language in the family»)

INTERSECTION OF FREE LANGUAGES WITH REGULAR LINGUAGES

To make a grammar more selective (to constrain the denoted language) ...

... one can *filter* it through (i.e., intersect it with) a regular language

(the result is always free – see slides on push-down automata and textbook, §4.3.1)

Example: Regular filters on the Dyck language L_D

sentences without substring cc , i.e.,
without nested parentheses

$$L_1 = L_D \cap \neg(\Sigma^* cc \Sigma^*) = (ac)^*$$

sentences with no more than one nest

$$L_2 = L_D \cap \neg(\Sigma^* ca \Sigma^*) = \{a^n c^n \mid n \geq 0\}$$

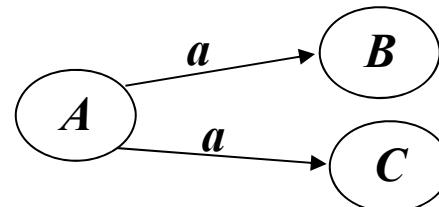
Both languages are free, the first one is also regular

Nondeterministic Finite Automata

Prof. A. Morzenti

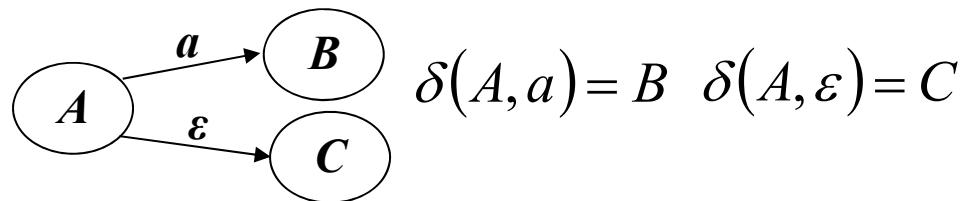
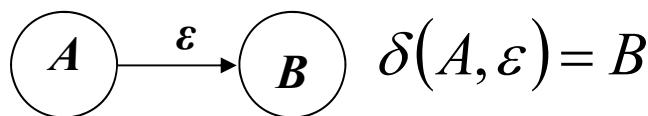
FORMS OF NONDETERMINISM

- 1) alternative moves for a unique input



$$\delta(A, a) = \{B, C\}$$

- 2) spontaneous move (or ε -move): automaton changes its state without “consuming” input



- 3) distinct initial states (useful, e.g., when merging various automata ...)

N.B. This third kind of automata can be transformed in an automata with a single initial state which is linked to the states that previously were the initial states. Notice that this new automata will still be nondeterministic.

ANALOGY WITH RIGHT-LINEAR GRAMMARS

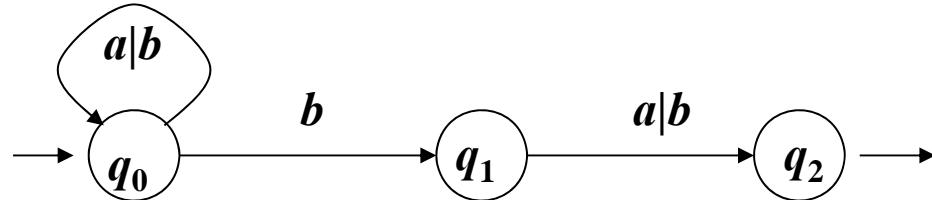
- 1) grammar with two alternatives $A \rightarrow aB \mid aC$ with $a \in \Sigma$
- 2) Grammar with copy rule $A \rightarrow B$ with $B \in V$
- 3) grammar(s) with «many axioms» (useful when composing grammars ...)

FOUR MOTIVATIONS FOR NONDETERMINISM IN FINITE STATE AUTOMATA

- 1) matching/mapping between grammars and automata
already illustrated
- 2) conciseness: language definitions through ND automata are more compact and readable

Example – language with penultimate symbol = b

$$L_2 = (a \mid b)^* b (a \mid b)$$



recognition of $baba$

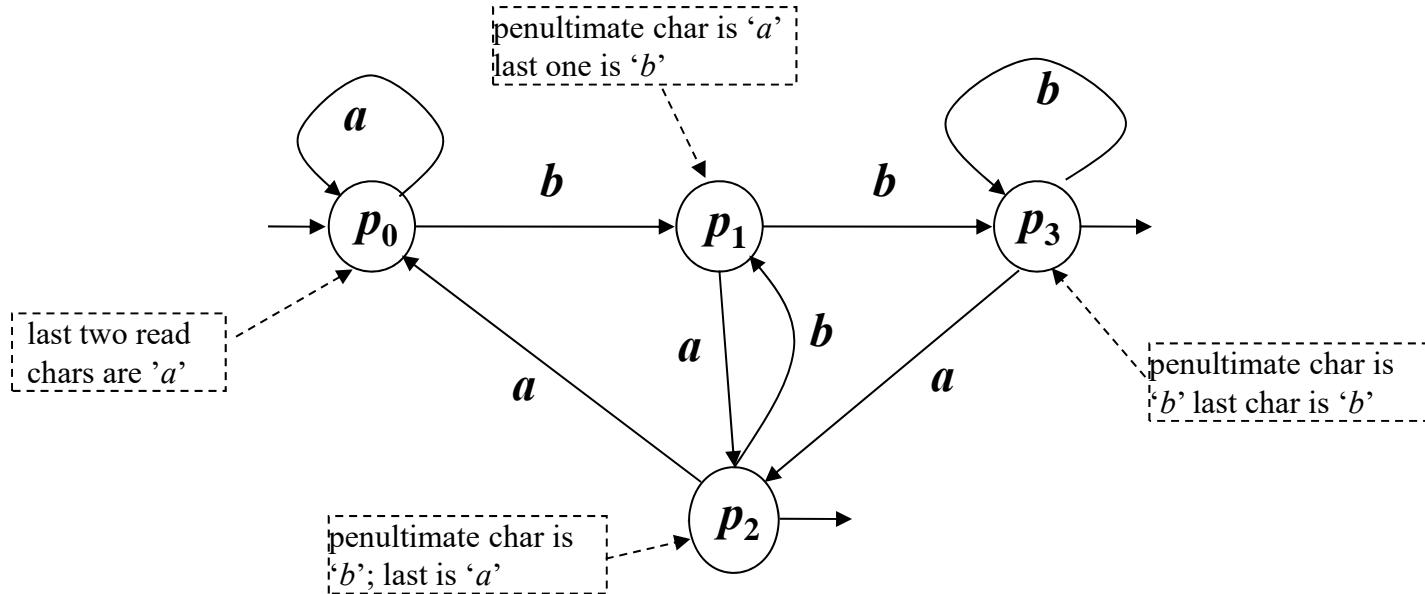
two computations, only one accepting

$$\begin{array}{ccccccccc} & b & & a & & b & & a & \\ q_0 & \rightarrow & q_0 & \rightarrow & q_0 & \rightarrow & q_1 & \rightarrow & q_2 \\ & b & & a & & b & & a & \\ q_0 & \rightarrow & q_0 & \rightarrow & q_0 & \rightarrow & q_0 & \rightarrow & q_0 \end{array}$$

Same as the previous but deterministic, note that is less concise. Notice that here I need a terminal state for each possible combination of the last 2 characters, so I need at least 4 terminal states.

the same language accepted by a **deterministic** automaton $M2$

in $M2$ the condition that the symbol before the last one is a b is not so clear



Exercise: show/prove that

Generalizing the example, from language L_2 to language L_k where the k -th last element ($k \geq 2$) is b , the **nondeterministic** automaton has $k + 1$ states,
the number of states of the **deterministic** one grows **exponentially** with k ($\approx 2^k$)

nondeterminism can make certain definitions much more concise

3) left – right duality

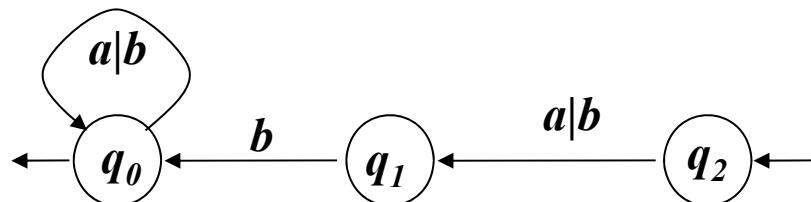
going from a (deterministic) automaton for lang. L to that for L^R requires to:

1. switch initial and final states
2. reverse the arrows direction

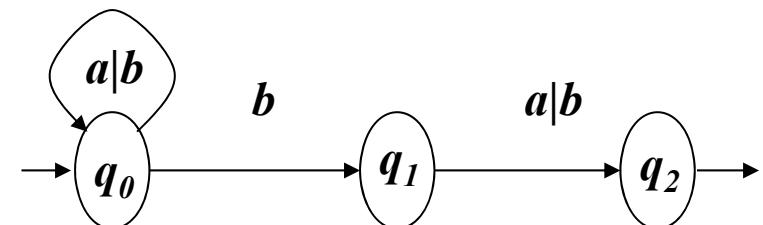
Both operations may introduce nondeterminism

Example - The language of strings having b as the penultimate symbol is the reverse image of the one having b as the second symbol

$$L' = \{x \mid b \text{ is the second symbol of } x\} \quad L' = (L_2)^R$$



b second char: deterministic

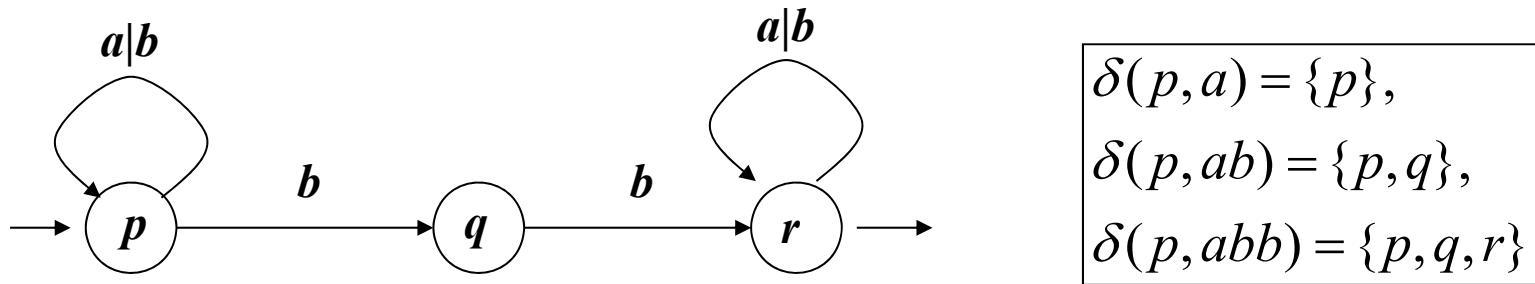


b penultimate char: nondeterministic

4) the transition through nondeterministic automata is useful in the construction of the finite recognizer of a language defined by a regular expression (see coming lessons)

Example – Search of a string in a text

Given a word $y \in \{a,b\}^*$, to check its inclusion in a text, we scan the text with the automaton accepting the language $(a \mid b)^* y (a \mid b)^*$. Example with $y = bb$



$p \xrightarrow{a} p \xrightarrow{b} p \xrightarrow{b} p \xrightarrow{b} p$	$p \xrightarrow{a} p \xrightarrow{b} p \xrightarrow{b} p \xrightarrow{b} q$
$p \xrightarrow{a} p \xrightarrow{b} p \xrightarrow{b} q \xrightarrow{b} r$	$p \xrightarrow{a} p \xrightarrow{b} q \xrightarrow{b} r \xrightarrow{b} r$

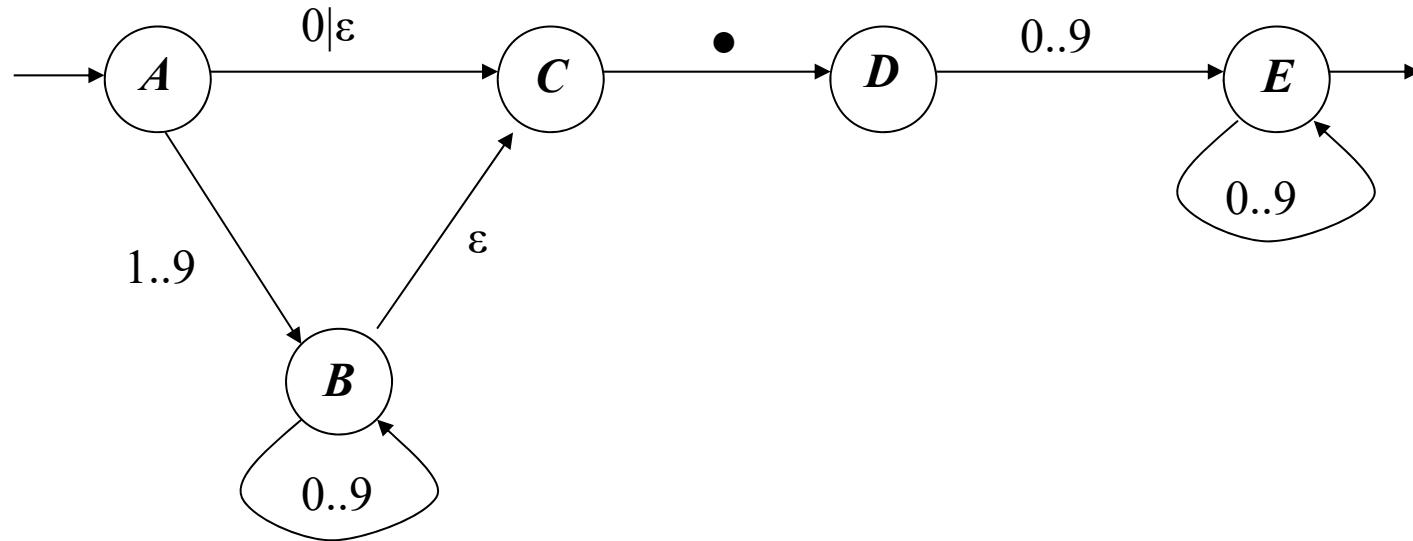
The first two computations do not «find» the word. The two last ones do

$ab \underbrace{bb}_{\text{and}} a \underbrace{bb}_{\text{b}}$

AUTOMATA WITH SPONTANEOUS MOVES

Example – decimal constants (with or without 0 before the dot, with no leading 0's in the integer part)

$$L = \left(0 \mid \varepsilon \mid \left((1..9)(0..9)^* \right) \right) \bullet (0..9)^+$$



When the automaton has spontaneous moves, the computation can be longer than the string
 (NB: the property of real-time analysis does not hold)

Computation accepting string 34•5: $A \xrightarrow{3} B \xrightarrow{4} B \xrightarrow{\varepsilon} C \xrightarrow{\bullet} D \xrightarrow{5} E$

Computation accepting string •02: $A \xrightarrow{\varepsilon} C \xrightarrow{\bullet} D \xrightarrow{0} E \xrightarrow{2} E$

UNIQUENESS OF THE INITIAL STATE

A nondeterministic automaton can have many initial states

But it is easy to obtain an equivalent one with a unique initial state ...

... by adding a «new» initial state q_0 and the ε -moves from q_0 to the former initial states

The added moves can then be eliminated (we will see how...)

EQUIVALENCE OF FINITE STATE AUTOMATA AND UNILINEAR GRAMMARS

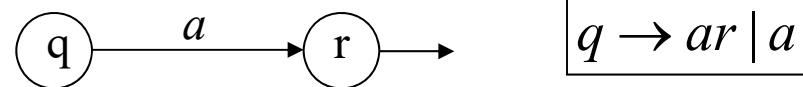
they define exactly the same languages

First we show how to derive an equivalent grammar from an automaton

The grammar: nonterminal symbols are the states Q of the automaton
axiom: the initial state

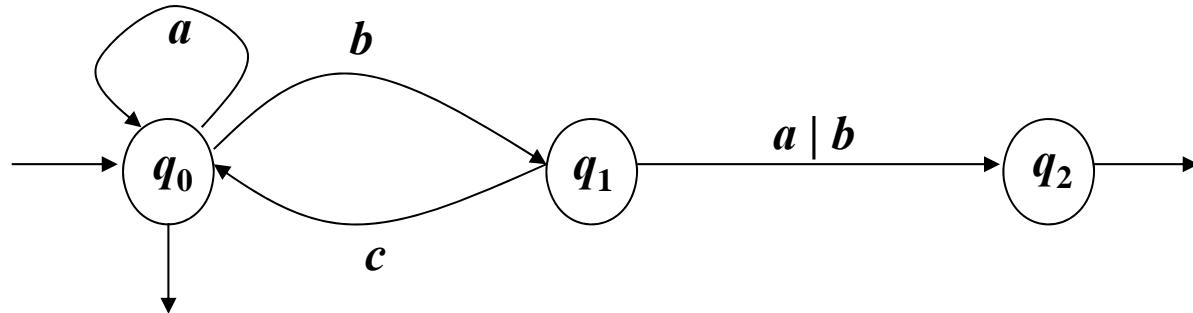


If a non-nullifiable grammar is preferred:



there is a one-to-one map: computations of the automaton \leftrightarrow derivations of the grammar
string x is accepted by the automaton iff \exists a derivation $q_0 \xrightarrow{*} x$

Example



$$q_0 \rightarrow aq_0 \mid bq_1 \mid \varepsilon$$

$$q_1 \rightarrow cq_0 \mid aq_2 \mid bq_2$$

$$q_2 \rightarrow \varepsilon$$

derivation of string bca

$$q_0 \Rightarrow bq_1 \Rightarrow bcq_0 \Rightarrow bcaq_0 \Rightarrow bca\varepsilon = bca$$

REVERSE TRANSFORMATION: GRAMMAR \Rightarrow AUTOMATON

right-linear grammar

1. Nonterm. alphabet V

2. axiom S

3. $p \rightarrow aq, a \in \Sigma$ and $p, q \in V$

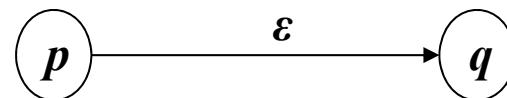
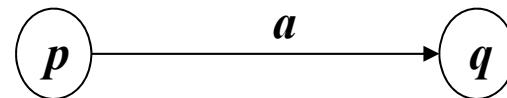
4. $p \rightarrow q$ where $p, q \in V$

5. $p \rightarrow \epsilon$

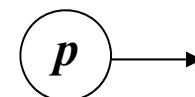
finite automaton

set of states $Q = V$

initial state $q_0 = S$



final state

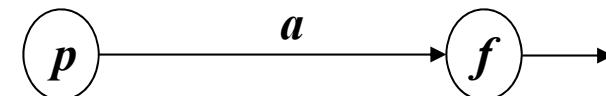


notice that in general the automaton will be nondeterministic

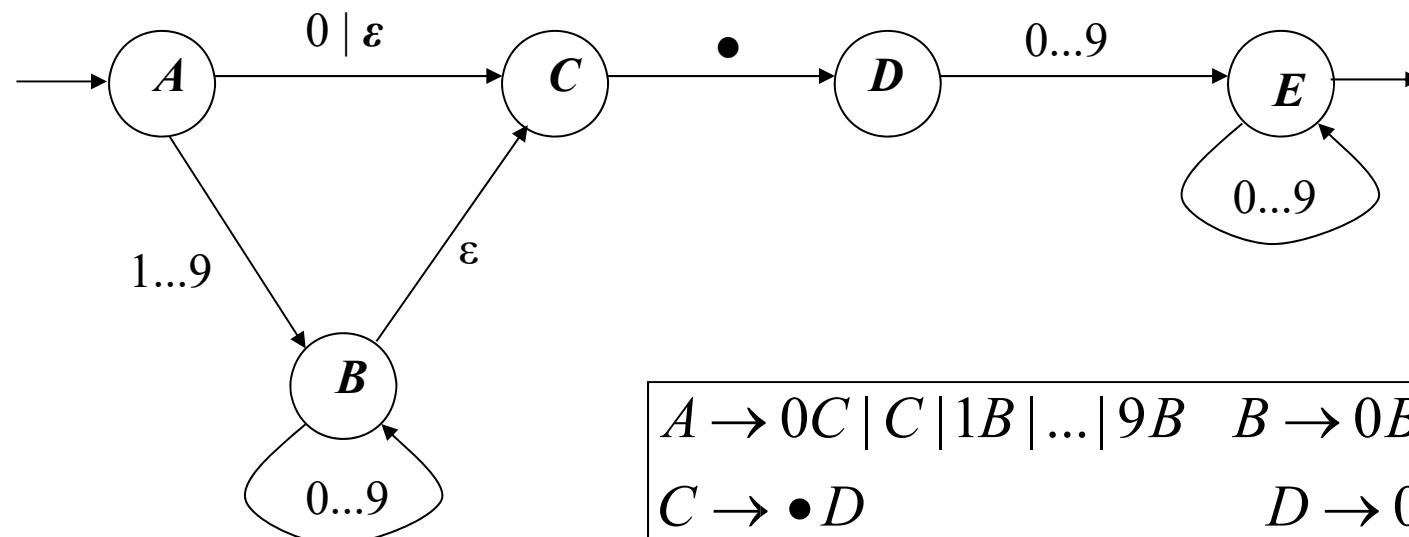
DERIVATION OF THE GRAMMAR \Leftrightarrow COMPUTATION OF THE AUTOMATON
therefore the two models define the same language

\Rightarrow A LANGUAGE IS ACCEPTED BY A FINITE AUTOMATON IF AND ONLY IF IT IS GENERATED BY A UNILINEAR GRAMMAR

If the grammar includes terminal rules of type $p \rightarrow a$, $a \in \Sigma$,
 The automaton will contain an additional state f , with f final, and the transition:



Example – Equivalence right-linear grammars – finite automata



$A \rightarrow 0C \mid C \mid 1B \mid \dots \mid 9B$	$B \rightarrow 0B \mid \dots \mid 9B \mid C$
$C \rightarrow \bullet D$	$D \rightarrow 0E \mid \dots \mid 9E$
$E \rightarrow 0E \mid \dots \mid 9E \mid \varepsilon$	axiom A

DEFINITION: an automaton is **ambiguous** if it accepts a sentence with distinct computations

one-to-one match between computations (automata) and derivations (grammars) \Rightarrow
 an automaton is ambiguous \Leftrightarrow corresponding right-linear grammar is ambiguous

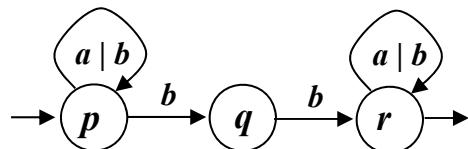
NB: for (right) linear grammars there is a 1-to-1 match between derivations and syntax trees
 (because in the derivation nonterminal phrase forms include one only nonterminal)

Example (continued) – Search of string bb in the text $abbb$

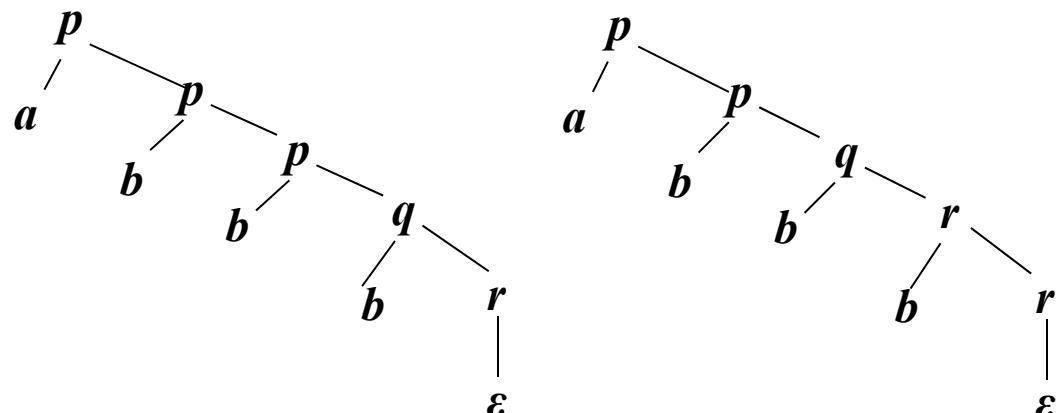
$$p \rightarrow ap \mid bp \mid bq$$

$$q \rightarrow br$$

$$r \rightarrow ar \mid br \mid \epsilon$$



$$(a \mid b)^* bb (a \mid b)^*$$



GRAMMAR \Rightarrow AUTOMATA IN CASE OF LEFT-LINEAR GRAMMARS

a variation of the case discussed above – see textbook §3.5.6

left-lin.gr. $G \Rightarrow$ (reverse) $\Rightarrow G_R$ (right-lin) \Rightarrow automaton for $L_R \Rightarrow$ (reverse) \Rightarrow automaton for $L(G)$

FROM AUTOMATA TO REGULAR EXPRESSIONS DIRECTLY THE BMC (Brzozowski & McCluskey) METHOD

Goal: reconstruct regular expression out of an automata

Assumptions

- unique initial state i without incoming arcs, and
- unique final state t without outgoing arcs

(we can manipulate an automata that doesn't respect this assumption adding an epsilon-arc to the initial state)
(almost the same as above)



(otherwise: add initial and final states connected through suitable spontaneous moves)

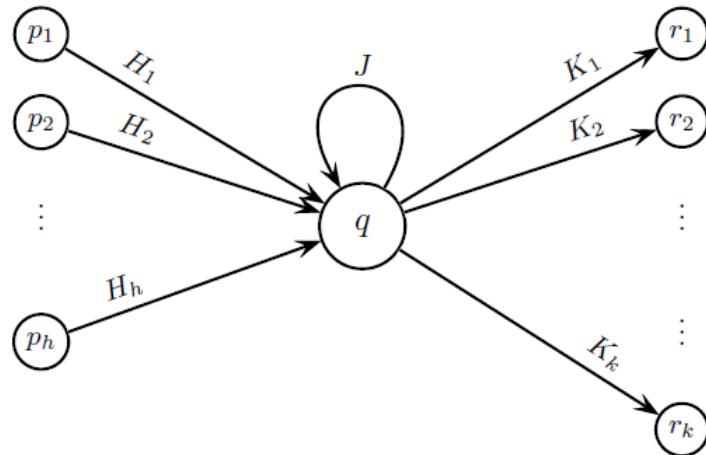
states different from i and t are called **internal**

a new **generalized automaton** is built :
equivalent to the initial one, but with arcs labeled by regular expressions

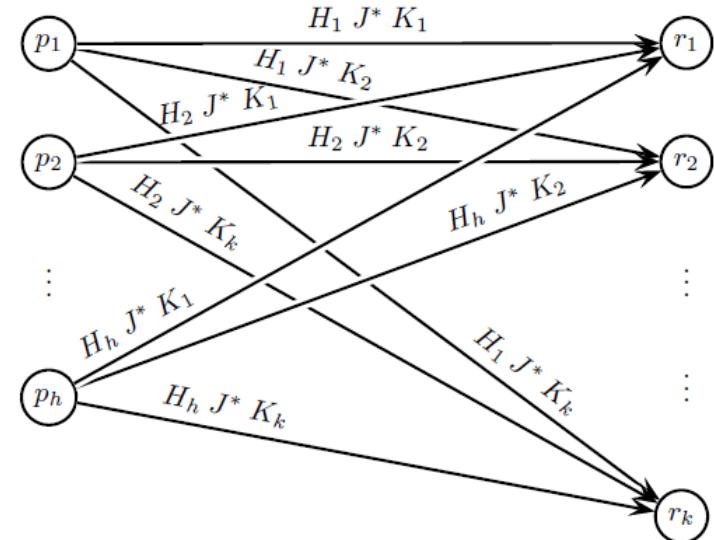
internal states are progressively eliminated
compensating moves are added, labeled by r.e. that preserve the accepted language
until only states i and t are left

then the r.e. e labeling the transition $i \xrightarrow{e} t$ is the e.r. of the language

before eliminating node q



after eliminating node q

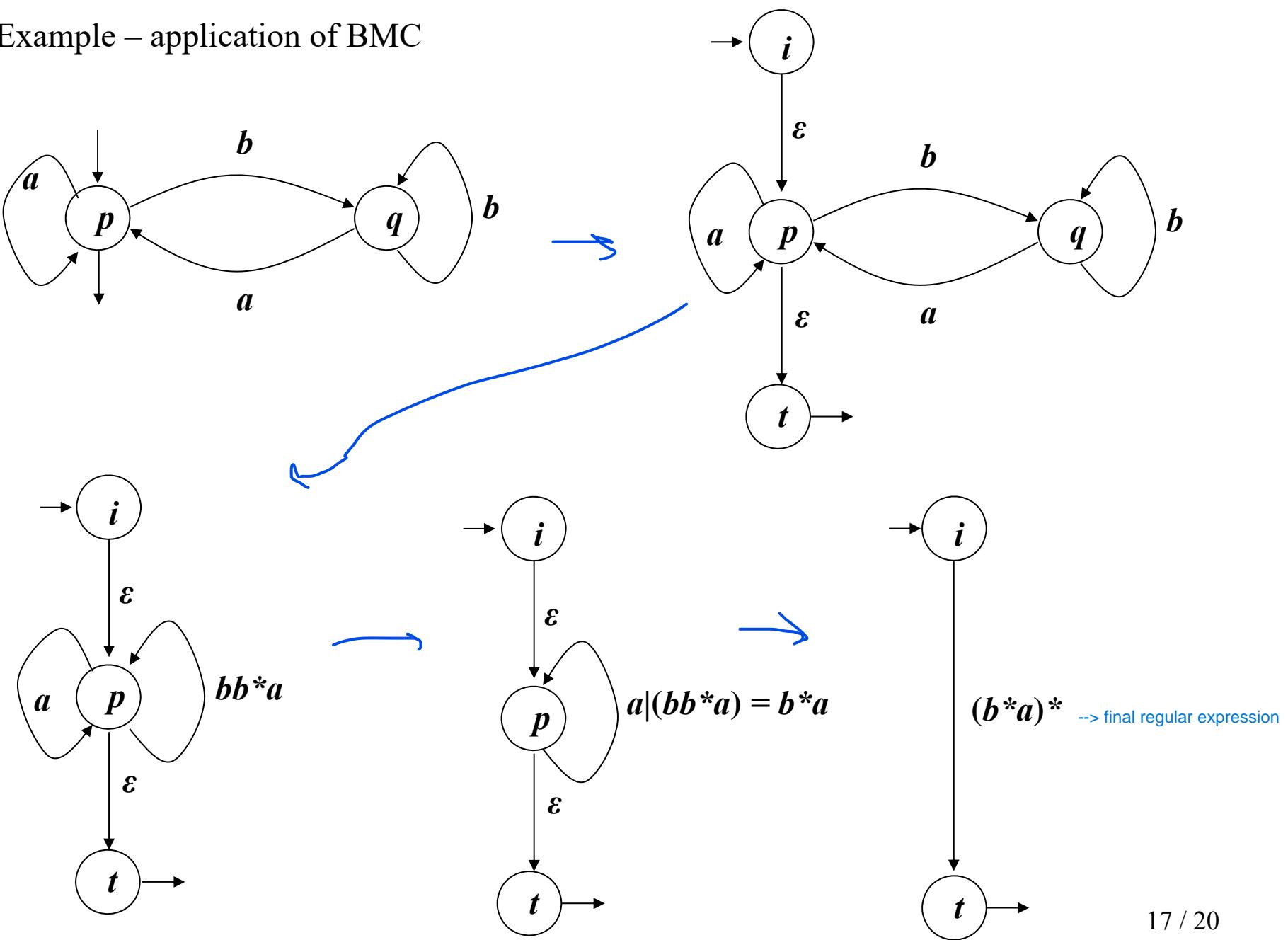


for each pair of states p_i, r_j a compensating transition: $p_i \xrightarrow{H_i J^* K_j} r_j$ (NB: possibly, $p_i = r_j$)

any resulting «parallel» transitions $p \xrightarrow{e_1} r$ and $p \xrightarrow{e_2} r$ «merged» into $p \xrightarrow{e_1 \mid e_2} r$

changing the order in the internal state elimination moves
leads to obtain formally distinct, but equivalent regular expressions
pay attention when solving exercises: simplify r.e.'s whenever possible

Example – application of BMC



ELIMINATION OF NONDETERMINISM

for efficiency, the final, implemented version of a finite recognizer must be deterministic

PROPERTY:

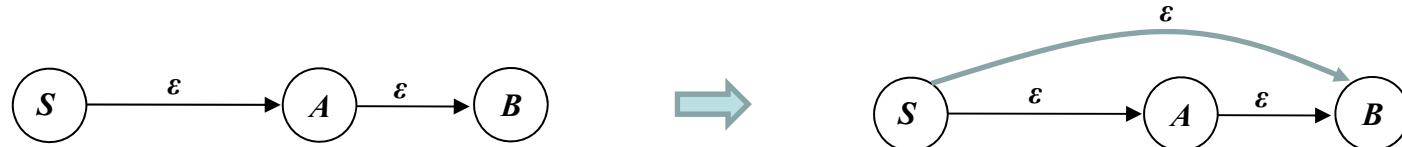
every nondeterministic automaton can be transformed into a deterministic one, and
(corollay) every unilinear grammar admits an equivalent nonambiguous grammar

determinization of a finite automaton conceptually separated in two parts
(both steps can be replaced by the Berry-Sethi construction, to be presented later)

1. elimination of spontaneous moves: move sequences that include spontaneous moves are replaced by **scanning moves** (non- ϵ arcs)
2. replacement of several nondeterministic (scanning) transitions by a single one
(reachable/accessible subset construction – the new states are subsets of the initial state set): we do not cover that; see textbook §3.7.1 (also covered by previous courses)

First phase: elimination of ε -moves – a 4-steps procedure

1: transitive closure of ε -moves



2: backward propagation of scanning moves over ε -moves



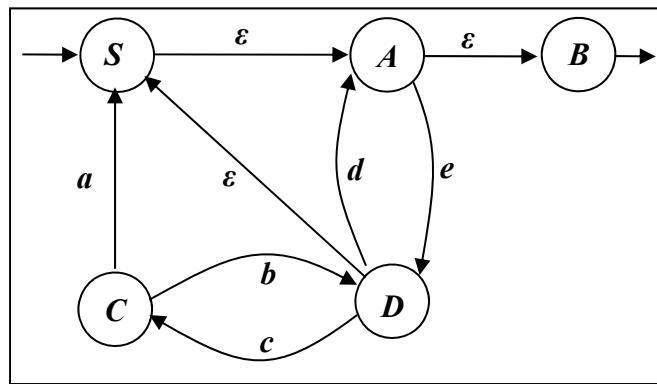
3: new final states: backward propagation of the «finality condition» for final states reached by ε -moves (antecedent states become also final)



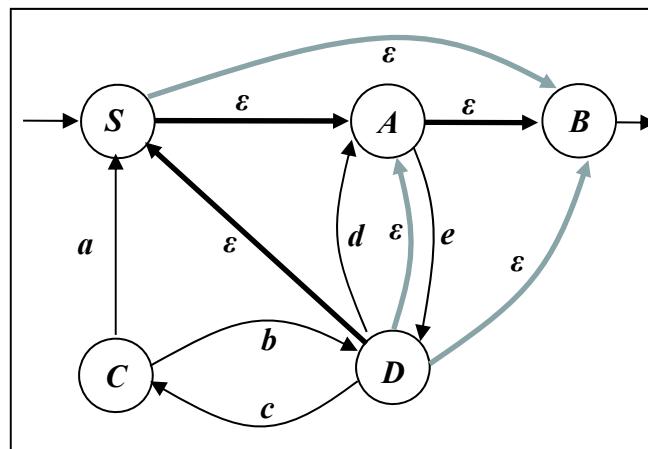
4: clean-up of ε -moves and of useless states

Comment: the purpose of steps 1-3 is to obtain an equivalent automaton where all ε -moves are redundant, so that they can be deleted without any consequence

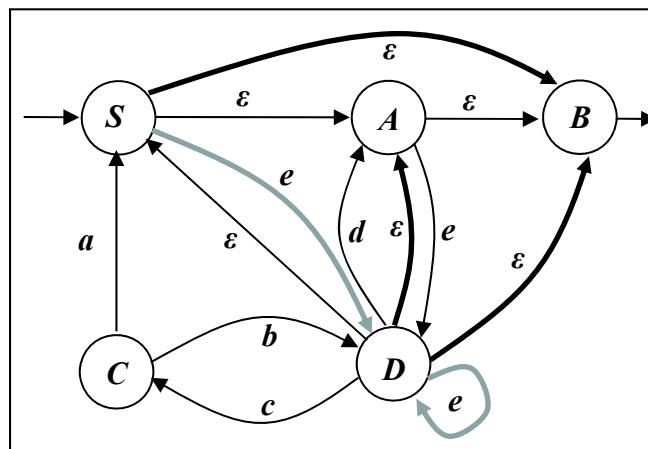
Example



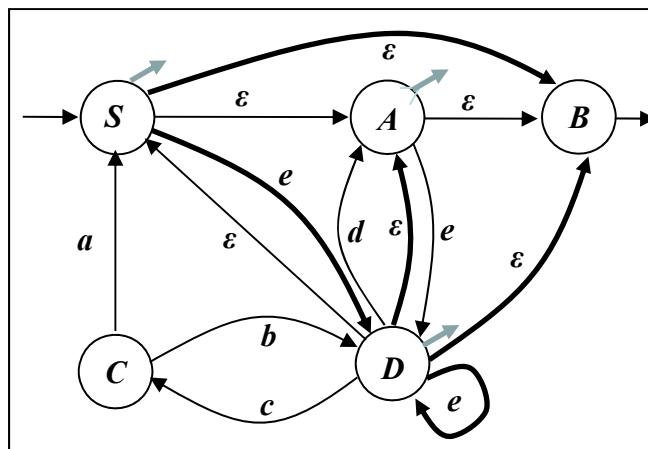
1

 →

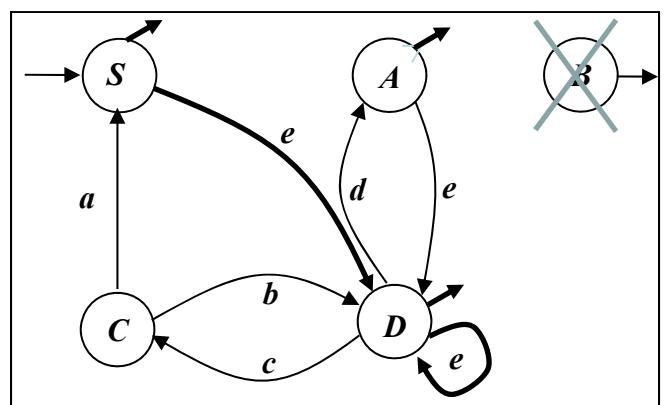
2



3



20 / 20



4

From Regular Expressions to Recognizing automata

Prof. A. Morzenti

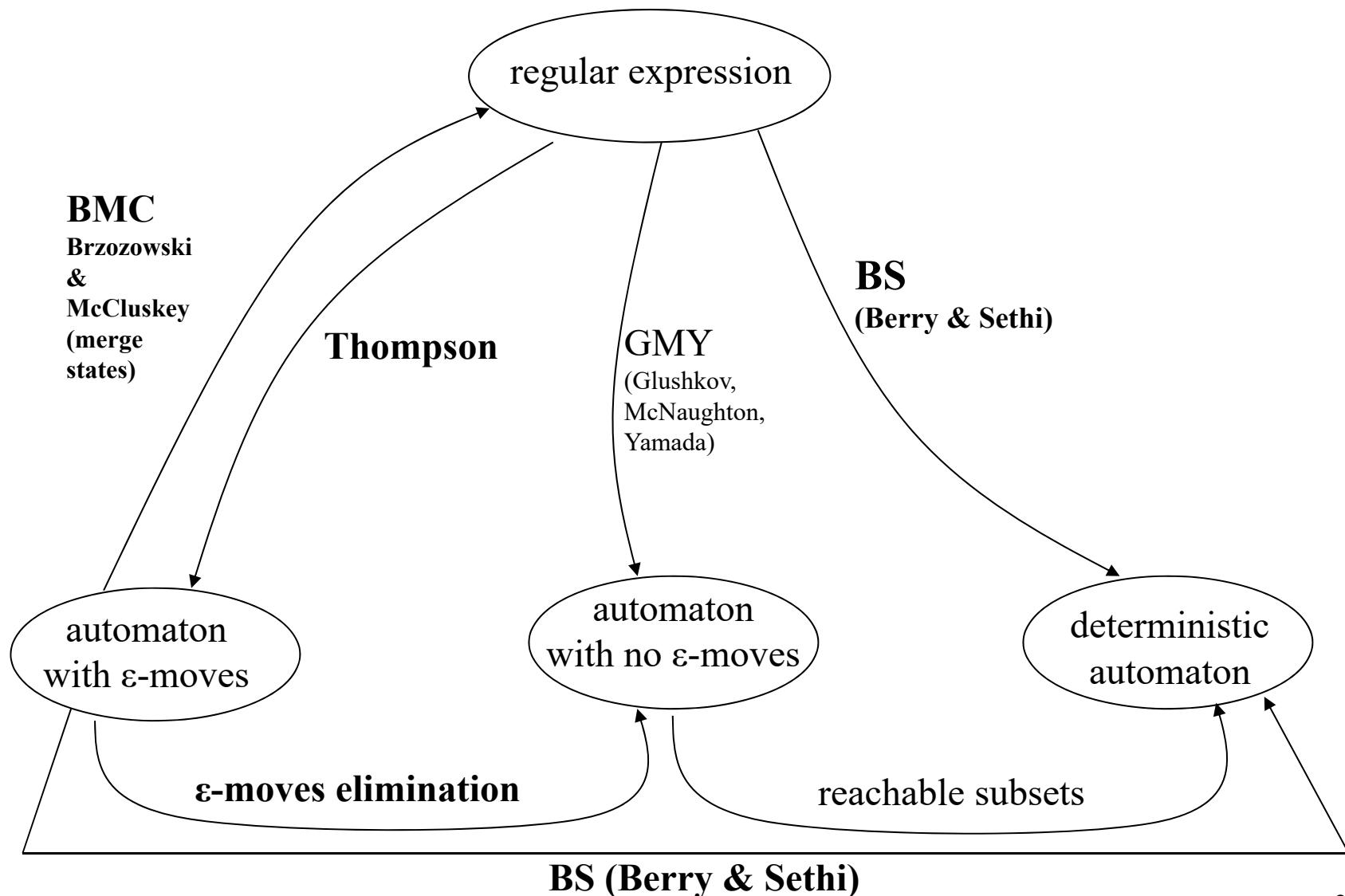
From Regular Expressions to Recognizing automata

Various algorithms, they differ in the kind of automaton and in the size of the result

the textbook reports three methods (we will discuss (1) and (3)):

- 1) THOMPSON (or structural) method
 - builds the recognizers of subexpressions
 - combines them through spontaneous moves
 - resulting automata have (several) **ϵ -moves** and are in general **nondeterministic**
- 2) GLUSHKOV, MC NAUGHTON and YAMADA (GMY) method
 - builds a **nondeterministic automaton having no spontaneous moves**
 - size is less than Thompson's
- 3) BERRY & SETHI (BS) method
 - builds a **deterministic** automaton
 - **not necessarily minimal**

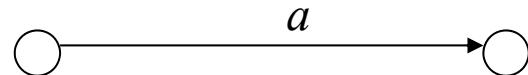
(1) and (2) can be combined with determinization algorithms,
(3) with minimization algorithms



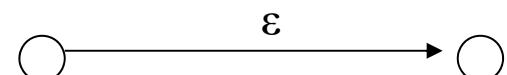
THOMPSON's STRUCTURAL METHOD

- 1) based on a systematic mapping between r.e. and recognizing automata (structural induction...)
- 2) every portion of automaton must have a unique initial and final state

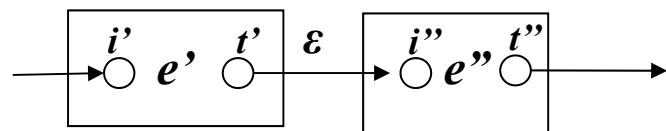
r.e. of type : a with $a \in \Sigma$



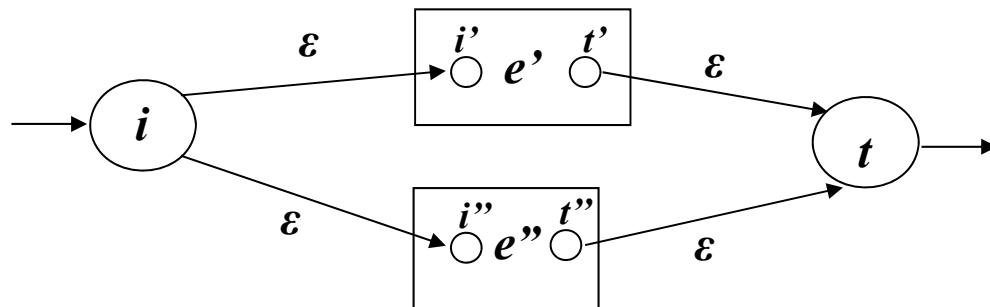
r.e. of type: ϵ



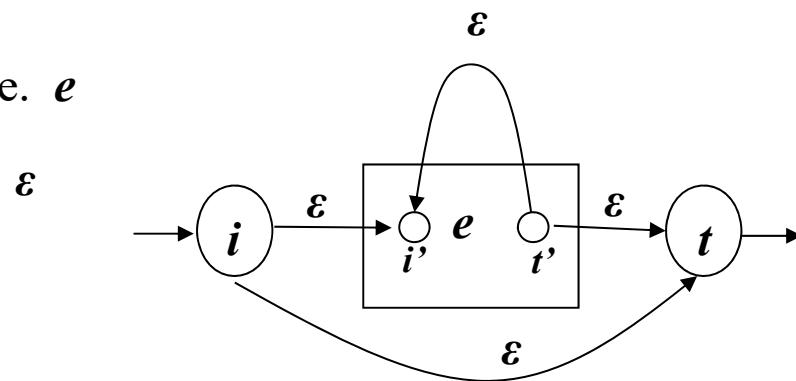
concatenation $e' \cdot e''$ of two r.e. e' and e''



Union of two r.e. e' and e''

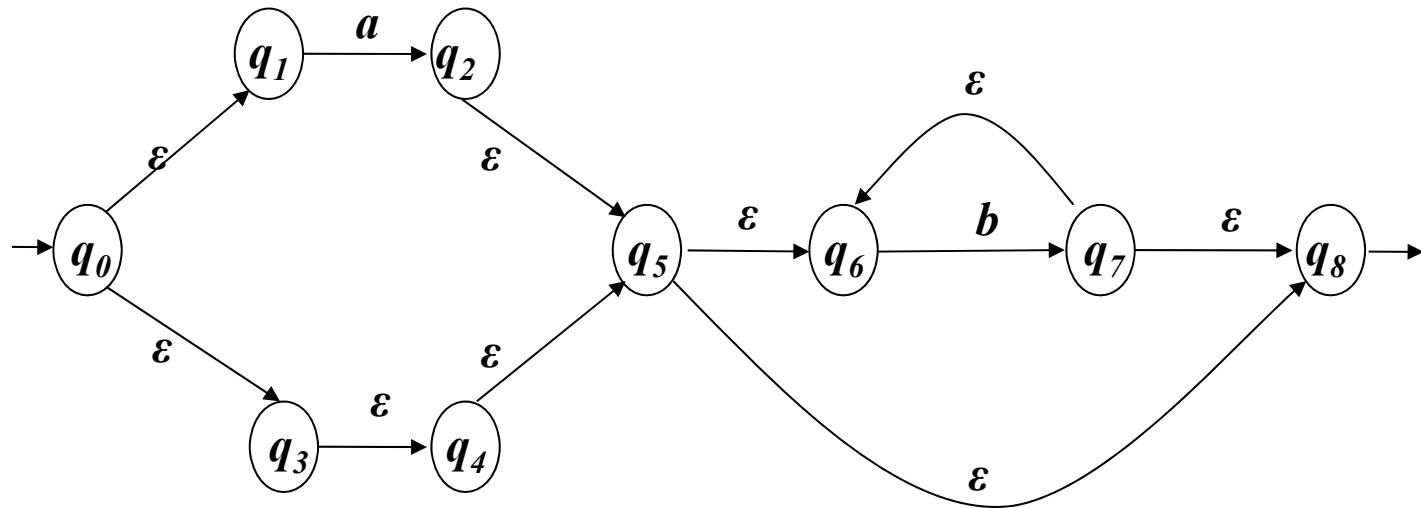


Star e^* of a r.e. e



Example

$$(a \cup \varepsilon).b^*$$



Before discussing the Berry – Sethi method, we introduce the ...

... LOCALLY TESTABLE languages, also called LOCAL (**LOC**)

LOC is a *proper subfamily* of regular languages (**LOC** \subset **REG**, **LOC** \neq **REG**)

DEFINITIONS: given a language L of alphabet Σ , define (assuming $a, b \in \Sigma$ and $x, y \in \Sigma^*$)

$$\text{set of Initial chars} \quad \text{Ini}(L) = \{a \mid ax \in L\}$$

$$\text{set of Finishing chars} \quad \text{Fin}(L) = \{b \mid xb \in L\}$$

$$\text{set of Digrams} \quad \text{Dig}(L) = \{ab \mid xaby \in L\}$$

$$\text{complement of Digrams} \quad \overline{\text{Dig}}(L) = \Sigma^2 \setminus \text{Dig}(L)$$

Example: $L_1 = (abc)^*$ has local sets $\text{Ini}(L_1) = \{a\}$, $\text{Fin}(L_1) = \{c\}$, $\text{Dig}(L_1) = \{ab, bc, ca\}$

Definitions of *Ini*, *Dig*, *Fin*, are provided similarly for individual strings

Ex.: for string $x = abc$ we have $\text{Ini}(x) = \{a\}$, $\text{Dig}(x) = \{ab, bc\}$, $\text{Fin}(x) = \{c\}$

$L_1 = (abc)^*$ includes all strings obtainable from its *Ini*, *Dig* e *Fin* (NB they are $\neq \epsilon$)

Ex.: string $abcabc$ obtained by composing a, ab, bc, ca, ab, bc, c

like in a sort of «domino game» where equal symbols are superimposed

NB: for every language L (even for non-regular languages) it «obviously» holds

$$L \setminus \{\varepsilon\} \subseteq \{ x \mid \text{Ini}(x) \in \text{Ini}(L) \wedge \text{Dig}(x) \subseteq \text{Dig}(L) \wedge \text{Fin}(x) \in \text{Fin}(L) \}$$

Because, trivially, every sentence of L

- starts (resp. ends) with a char $c \in \text{Ini}(L)$ (resp. $c \in \text{Fin}(L)$), and
- its digrams are included in those of the language

A language $L \in LOC$ includes ***all*** the strings generated by the three local sets,
i.e., the language contains ***all and only*** the strings that can be built from Ini , Fin , and Dig
(plus, possibly, ε)

$$L \in LOC \text{ iff } L \setminus \{\varepsilon\} = \{ x \mid \text{Ini}(x) \in \text{Ini}(L) \wedge \text{Dig}(x) \subseteq \text{Dig}(L) \wedge \text{Fin}(x) \in \text{Fin}(L) \}$$

Instead, $L \notin LOC$ if it does not include all strings generated from Ini , Dig , and Fin
i.e. \exists a string $x \notin L$ that is generated from the local sets of L

NB: the definition provides a ***necessary condition*** for a language to be local

and therefore a method for proving that a language ***is NOT*** local

(to prove that language L ***is not*** local, exhibit a *witness*:

a string $x \notin L$ s.t. $\text{Ini}(x) \in \text{Ini}(L)$, $\text{Fin}(x) \in \text{Fin}(L)$ and $\text{Dig}(x) \subseteq \text{Dig}(L)$)

Ex.: $L_1 = (abc)^*$ is local: $Ini(L_1) = \{a\}$, $Fin(L_1) = \{c\}$, $Dig(L_1) = \{ab, bc, ca\}$

All strings obtained from Ini , Fin , Dig are included in L_1

Example of nonlocal regular language

L_2 is **strictly included** in the set of strings generated from its local sets Ini , Dig , Fin

Indeed L_2 does not include strings of odd length nor strings with a b surrounded by a 's

$$L_2 = b(aa)^+b$$

$$Ini(L_2) = Fin(L_2) = \{b\}$$

$$Dig(L_2) = \{aa, ab, ba\}$$

$$\overline{Dig}(L_2) = \{bb\}$$

$$baab, baaaab \in L_2$$

$$baaab \notin L_2 \quad baabab \notin L_2 \quad \dots$$

L_1 and L_2 are regular, L_1 is local, L_2 is not local

Therefore $LOC \subset REG$, $LOC \neq REG$, inclusion is **strict**

For every non-local regular language L there exists a superset of it, L_{LOC} , which is local:

It contains **all** strings obtainable through $Ini(L)$, $Dig(L)$, $Fin(L)$ (a sort of trans. closure ...)

NB: language locality is determined by (presence or absence of) certain non-empty strings:

⇒ presence of ϵ in the language is immaterial

Ex.: $(abc)^*$ and $(abc)^+$ are **both** local

HOW TO BUILD A RECOGNIZER OF A LOCAL LANGUAGE FROM ITS LOCAL SETS
 it scans the string, and checks that:

- * the first char $\in \text{Ini}$
- * every pair of consecutive chars $\in \text{Dig}$
- * the last char $\in \text{Fin}$

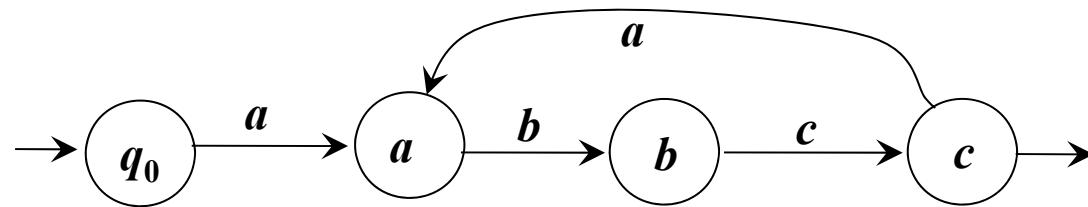
string analysis from left to right using a *shifting window* of length two,
 implemented by a very simple deterministic automaton: what must be «remembered»?

only the last read char (\Rightarrow a one-to-one mapping between Q and Σ)

construction of the recognizer of language $L \in \text{LOC}$ starting from $\text{Ini}, \text{Fin}, \text{Dig}$

1. a unique initial state q_0
2. set of states $Q = \Sigma \cup \{q_0\}$ (all states but the initial one are labeled by an element of Σ)
3. final states $F = \text{Fin}$; if $\varepsilon \in L$ then F also includes q_0
4. transition function δ : $\forall a \in \text{Ini} \ \delta(q_0, a) = a; \quad \forall xy \in \text{Dig} \ \delta(x, y) = y$

Example: $L_1 = (abc)^+$ $\text{Ini}(L_1) = \{a\}$ $\text{Fin}(L_1) = \{c\}$ $\text{Dig}(L_1) = \{ab, bc, ca\}$

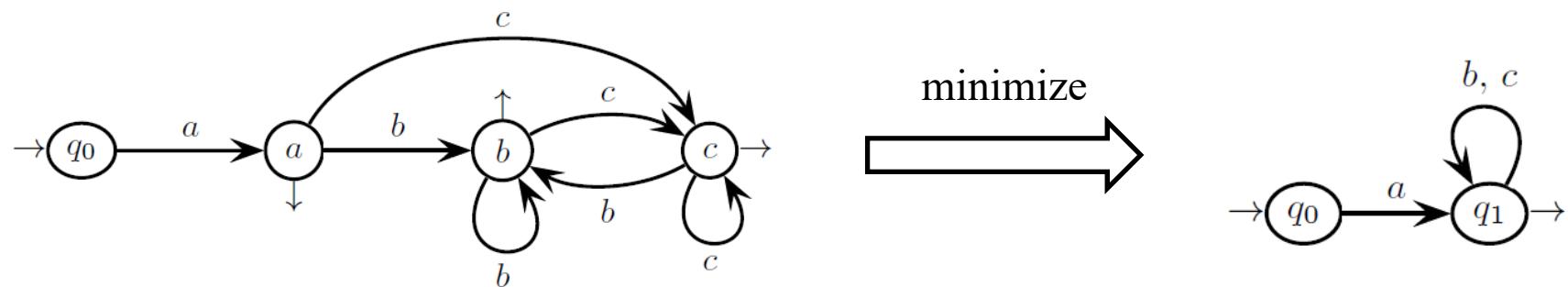


(from previous construction)

SOME CRITERIA FOR PROVING THAT A LANGUAGE L IS LOCAL

1. L is accepted by an automaton satisfying conditions 1 – 4 in previous slide (it is called ***normalized local automaton***)

however such an automaton might be NOT minimal



hence a second, less restrictive, sufficient condition:

2. L is accepted by a (possibly minimal) automaton obtained from the normalized local automaton by merging indistinguishable states

BERRY & SETHI DETERMINISTIC RECOGNIZER

(we only illustrate it: see textbook, §3.8.2.4 for a complete explanation/justification)

Let e the starting r.e. (of alphabet Σ): e.g. $e = (a \mid bb)^* (ac)^+$
 e' its **numbered version** (of alphabet Σ_N): e.g. $e' = (a_1 \mid b_2 b_3)^* (a_4 c_5)^+$

We consider expression $e' \dashv$ which includes the end-of-text mark \dashv

We define, for each symbol a of e' , the set of **Followers** of a , $Fol(a)$

It is the set of symbols that, in the strings $\in L(e' \dashv)$, can follow a

Essentially, the same information as $Dig(e' \dashv)$

$$Fol(a) = \{ b \mid ab \in Dig(e' \dashv) \}$$

Hence $\dashv \in Fol(a)$ for every $a \in Fin(e')$

Ex.: for $e' = (a_1 \mid b_2 b_3)^* (a_4 c_5)^+ \dashv$ we have

$$Fol(a_1) = \{a_1, b_2, a_4\} \quad Fol(b_2) = \{b_3\} \quad Fol(b_3) = \{a_1, b_2, a_4\}$$

$$Fol(a_4) = \{c_5\} \quad Fol(c_5) = \{a_4, \dashv\}$$

Construction of the deterministic recognizer of Berry-Sethi

Every state is (corresponds to) a subset of $\Sigma_N \cup \{\dashv\}$

It contains the symbols that one can *expect as next input*

Therefore final states are those that include (possibly among others) the end-mark \dashv

The *initial state* is the set $Ini(e' \dashv)$

States are generated from the initial one, adding transitions and new states

State generation iterated until a fixed point is reached (no new state can be generated)

During construction the transition function δ is viewed as a set of transitions $q \xrightarrow{a} q'$

BS ALGORITHM

$q_0 := \text{Ini}(e' \dashv)$; mark q_0 as *not visited*

$Q := \{q_0\}$

$\delta := \emptyset$

while there exists in Q a non-visited state q **do**

 mark q as visited

for each symbol $b \in \Sigma$ **do**

$$q' := \bigcup_{b_i \in q} \text{Fol}(b_i)$$

if $q' \neq \emptyset$ **then**

if $q' \notin Q$ **then**

 mark q' as *not visited*

$$Q := Q \cup \{q'\}$$

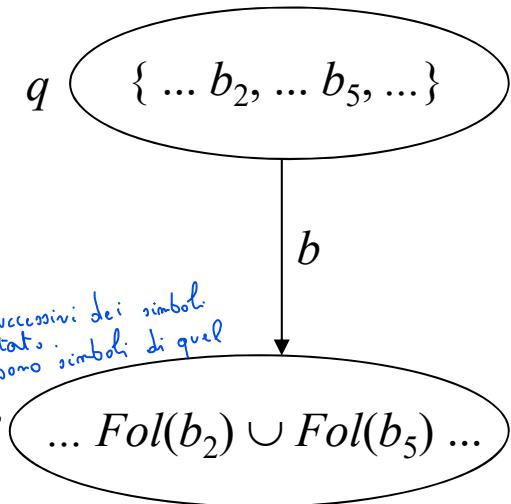
end if

$$\delta := \delta \cup \{q \xrightarrow{b} q'\}$$

end if

end do

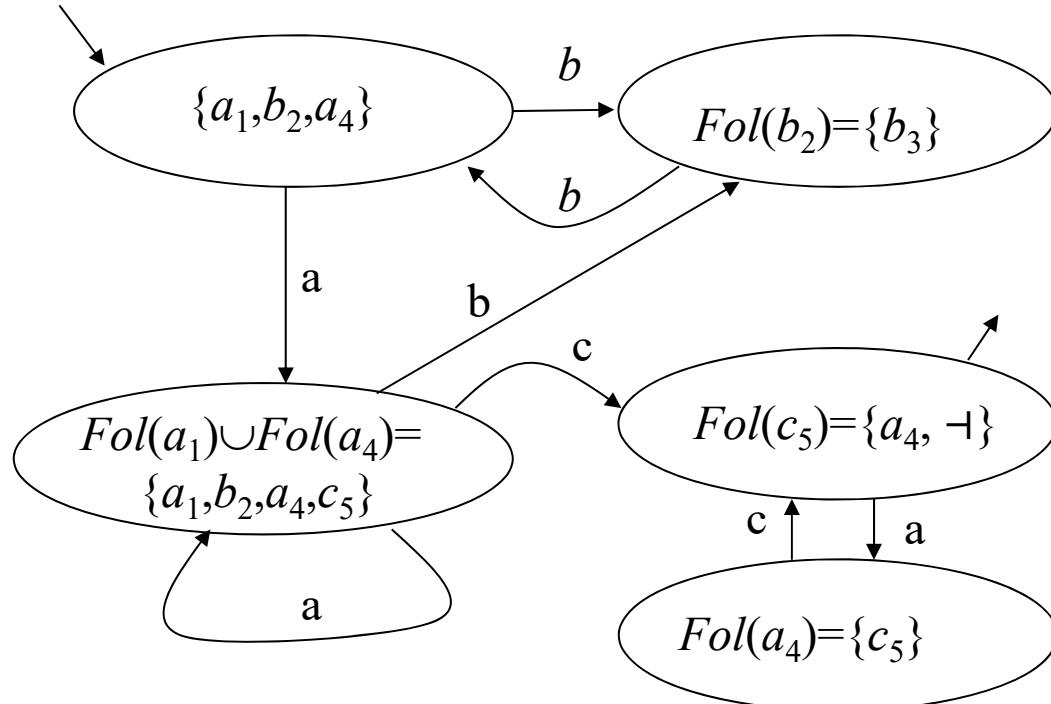
end do



Devo fare l'unione dei successivi dei simboli di tipo b presenti nello stato.
Si nota come se non ci sono simboli di quel tipo, l'unione è vuota e pertanto si ricade qui.

Passando alla lettura successiva.

Example



$e = (a \mid bb)^* (ac)^+$	
$e' \neg = (a_1 \mid b_2 b_3)^* (a_4 c_5)^+ \neg$	
$Ini(e' \neg) = \{a_1, b_2, a_4\}$	
x	$Fol(x)$
a_1	a_1, b_2, a_4
b_2	b_3
b_3	a_1, b_2, a_4
a_4	c_5
c_5	a_4, \neg

the resulting automaton is deterministic
but it can be **non-minimal**
(because of the numbering of symbols)

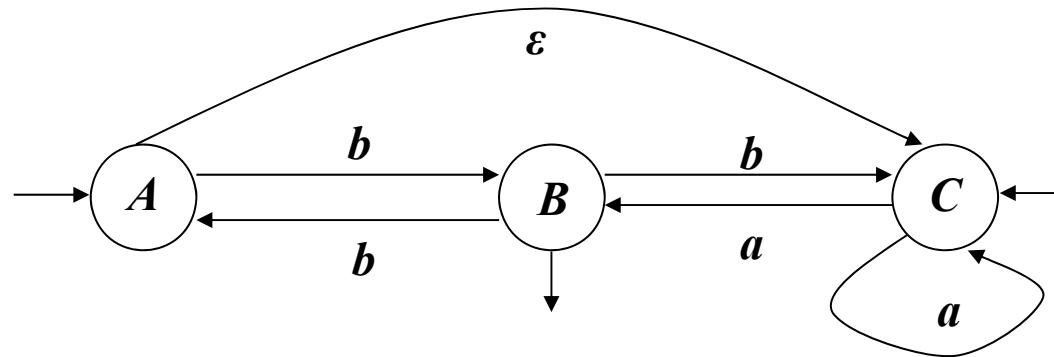
USING THE BS ALGORITHM FOR AUTOMATA DETERMINIZATION

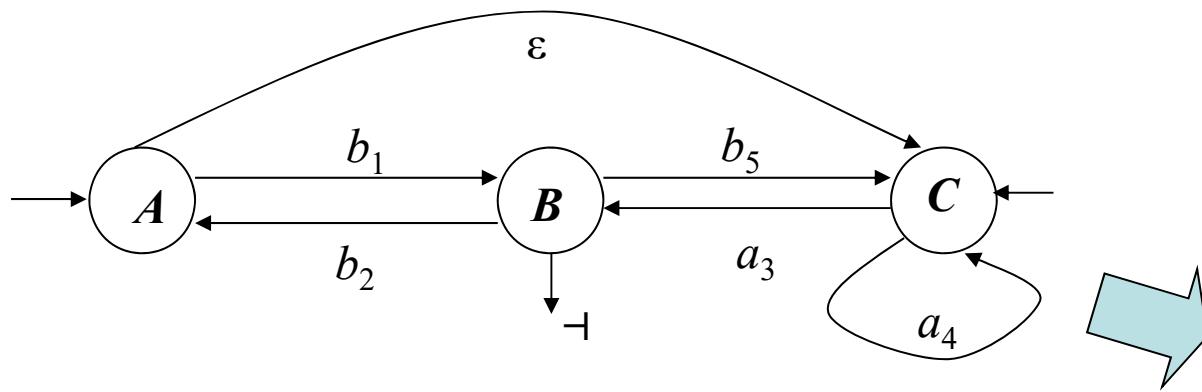
BS algorithm used for determinizing a nondeterministic automaton N with ε -arcs

1. number the non- ε arcs of N , obtaining a numbered version N' ; add an endmark ‘ \dashv ’ on darts exiting the final states
2. compute for N' the local sets Ini and Fol (using rules similar to those for a r.e.)
3. apply the BS construction, thus obtaining an automaton M

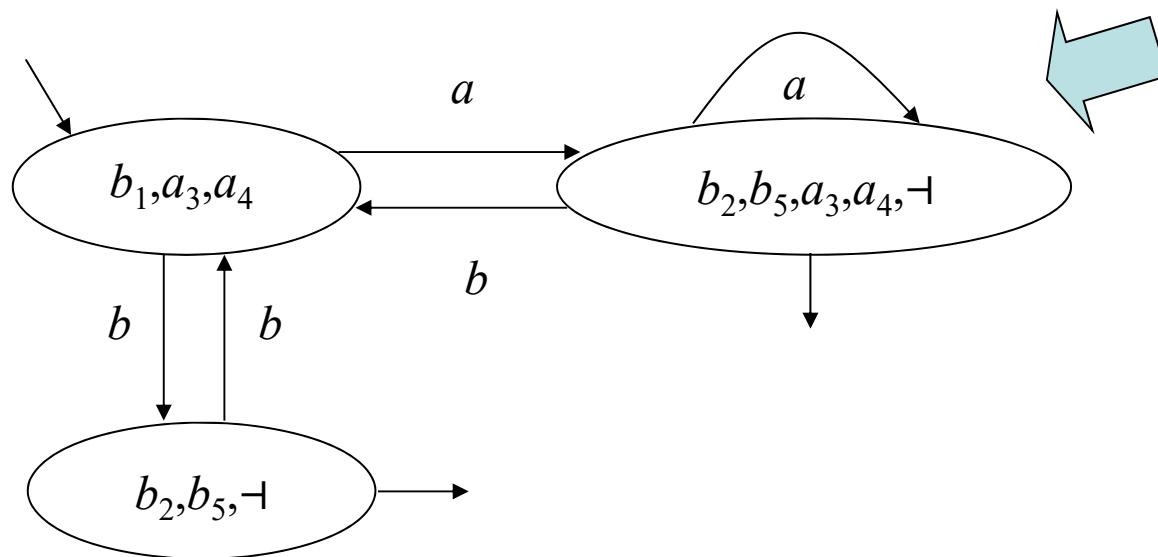
the resulting automaton is deterministic, though possibly non-minimal

Example





$Init(L(N'))-\mid$	$\{b_1, a_3, a_4\}$
$\epsilon a_3 = a_3, \epsilon a_4 = a_4$	
x	$Fol(x)$
b_1	b_2, b_5, \neg
b_2	b_1, a_3, a_4
a_3	b_2, b_5, \neg
a_4	a_3, a_4
b_5	a_3, a_4



REGULAR EXPRESSIONS WITH (1) COMPLEMENT AND (2) INTERSECTION

both topics covered by previous courses so we go through them quickly

(1) CLOSURE OF REG UNDER COMPLEMENT AND INTERSECTION

If $L, L', L'' \in REG$ then $\neg L \in REG$ $L' \cap L'' \in REG$

$$L \in REG \Rightarrow \neg L \in REG$$

proved by constructing the recognizer for the complement language $\neg L = \Sigma^* \setminus L$
starting from a *deterministic* recognizer M for L

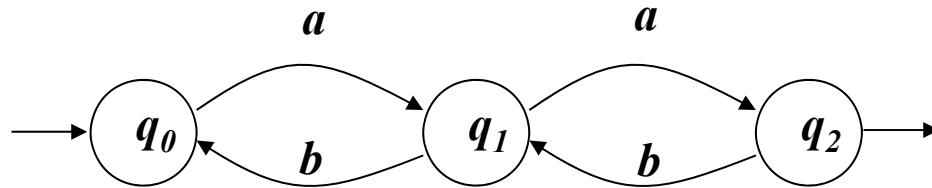
ALGORITHM: construction of the deterministic recognizer M' for the complement

We extend $M = \langle Q, \Sigma, \delta, q_0, F \rangle$ with the *error* or *sink* state $p \notin Q$ and the arcs to and from it

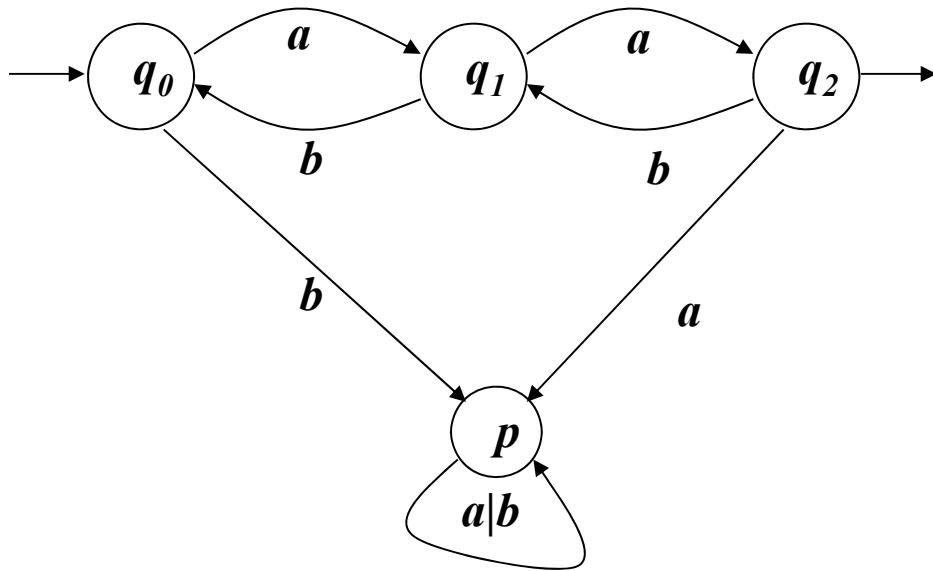
1. $Q' = Q \cup \{p\}$
2. $\delta'(q, a) = \delta(q, a)$ if $\delta(q, a)$ is defined, otherwise $\delta'(q, a) = p$; $\delta'(p, a) = p \quad \forall a \in \Sigma$
3. Switch initial and final states: $F' = (Q \setminus F) \cup \{p\}$

Example: automaton for the complement language

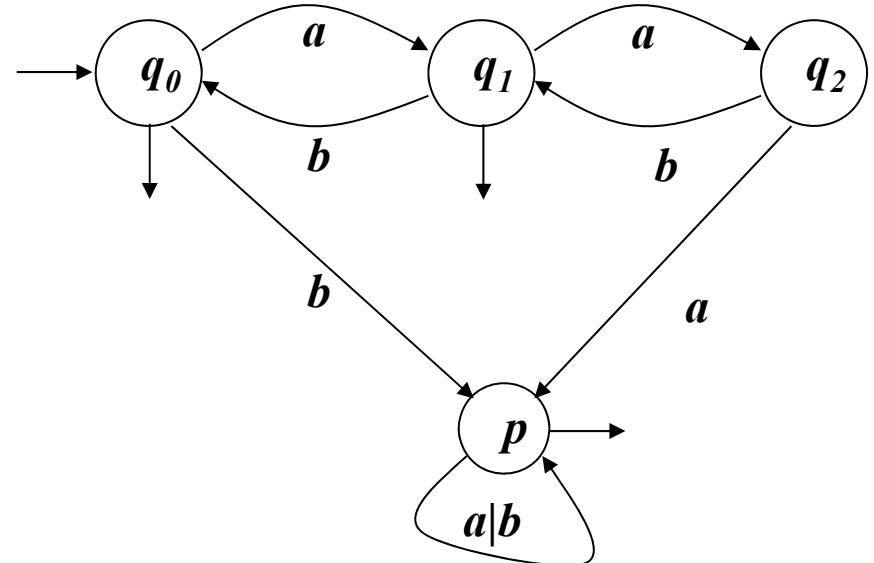
original automaton:



automaton with the sink state added:



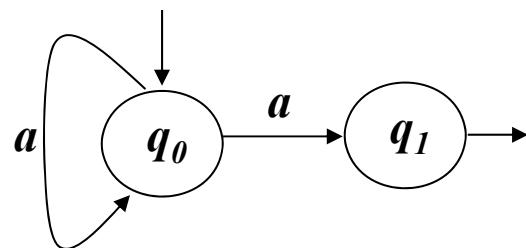
complement automaton
(final states switched):



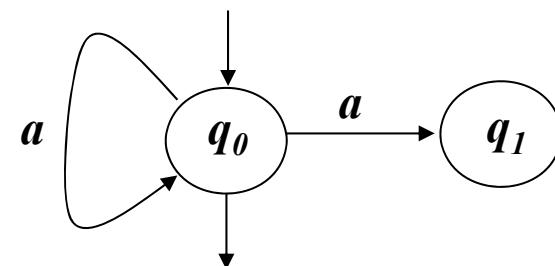
NB: the starting automaton M **must** be deterministic
 otherwise the language accepted by the complement automaton M' might not be disjointed
 and the obvious property $L \cap \neg L = \emptyset$ would be violated

A nondeterministic automaton can have, for a string $x \in L$,
 one accepting computation and a non-accepting one
 in the complement automaton M' this non-accepting computation of M would be accepting

Example:



original automaton M (*nondeterministic*)



(pseudo) complement automaton M'

$a \in L(M)$, but M has two computations for a , one accepting and one rejecting
 The pseudo complement automaton accepts string a , that also belongs to the original language

2) RECOGNIZER FOR THE INTERSECTION OF TWO REGULAR LANGUAGES

one could use the property of closure of REG under complement and union
to exploit the De Morgan identity $L_1 \cap L_2 = \neg(\neg L_1 \cup \neg L_2)$ and therefore:

- build the deterministic recognizers of L_1 and L_2
- derive those of the complement languages $\neg L_1$ and $\neg L_2$
- build the recognizer for the union (using the Thomson method)
- make the automaton deterministic
- derive the complement automaton

There is a more direct method

(CARTESIAN) PRODUCT AUTOMATON

Quite a common method:

Allows one to simulate the simultaneous execution of two automata

We assume the two automata without ε -moves but not necessarily deterministic

The state set of the product automaton M is the cartesian product of the state sets of M' and M''

A state is a pair $<q', q''>$, with $q' \in Q'$ and $q'' \in Q''$

definition of transition function:

$$<q', q''> \xrightarrow{a} <r', r''> \quad \text{if and only if} \quad q' \xrightarrow{a} r' \wedge q'' \xrightarrow{a} r''$$

Initial states I of M are also the cartesian product $I = I' \times I''$

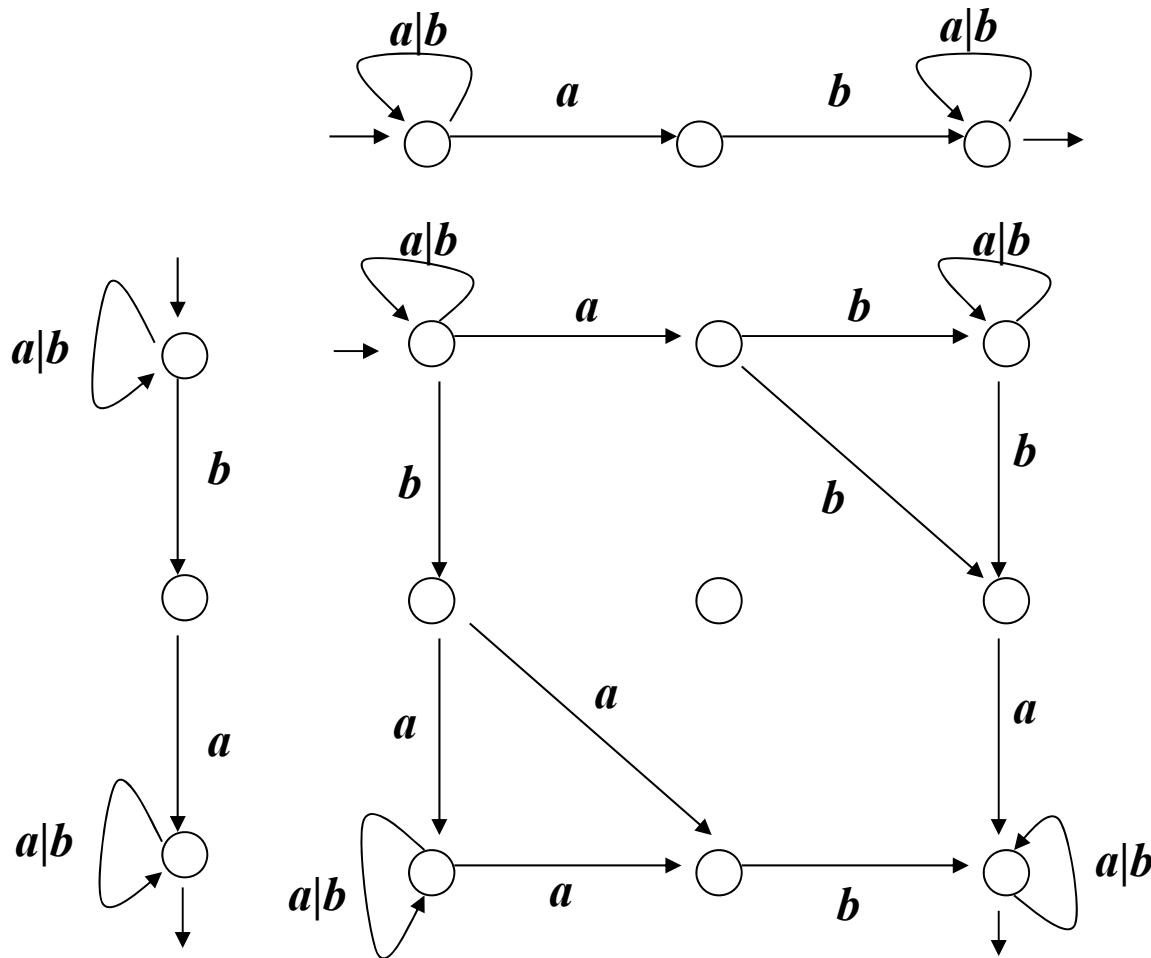
Final states are also the product $F = F' \times F''$

NOTE: The method can be applied to other set-theoretical operations (e.g. for union: final states of M are those state pairs where at least one of the two states is final)

Example – Intersection and product machine for the two languages of strings containing, respectively, substring ***ab*** and ***ba***

$$L' = (a \mid b)^* ab(a \mid b)^*$$

$$L'' = (a \mid b)^* ba(a \mid b)^*$$



Pushdown Automata and Context Free Language Parsing

Prof. A. Morzenti

NB: up to slide 13 notions assumed to be known from other previous courses

PUSHDOWN AUTOMATA

- 1) stack auxiliary memory +
input string with terminator □

- 3) operations:

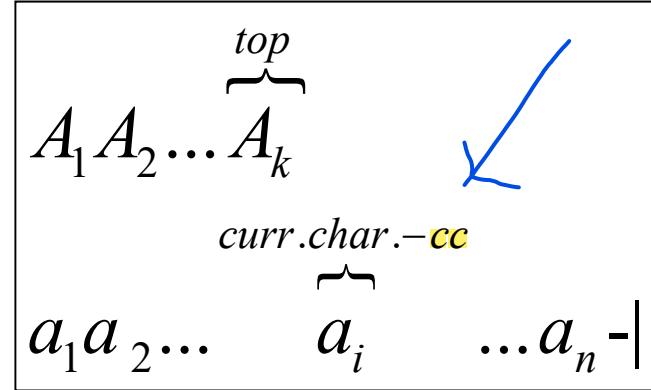
$push(B)$, $push(B_1, B_2, \dots, B_n)$: push symbol(s) on top of the stack

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)

DEFINITION OF PUSHDOWN AUTOMATON

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. $Z_0 \in \Gamma$ *initial stack symbol*
7. $F \subseteq Q$ *set of final states*

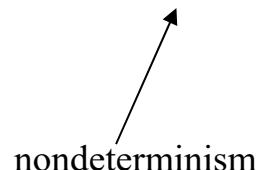
TRANSITION FUNCTION:

domain:

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

range:

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



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

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

with $n \geq 1$, $a \in \Sigma$, $Z \in \Gamma$, $p_i \in Q$, $\gamma_i \in \Gamma^*$

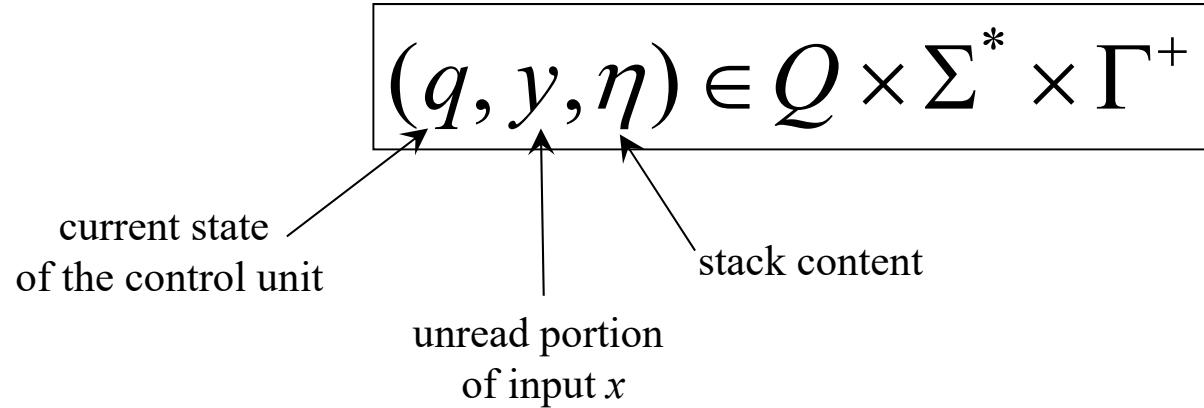
SPONTANEOUS MOVE:
(possibly nondeterministic)

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

with $n \geq 1$, $Z \in \Gamma$, $p_i \in Q$, $\gamma_i \in \Gamma^*$

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



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:

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

TRANSITION SEQUENCE: $\xrightarrow{*}, \xrightarrow{+}$

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

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:

$$(q_0, x, Z_0) \xrightarrow{+} (q, \varepsilon, \lambda) \quad q \in F \text{ and } \lambda \in \Gamma^*$$

(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

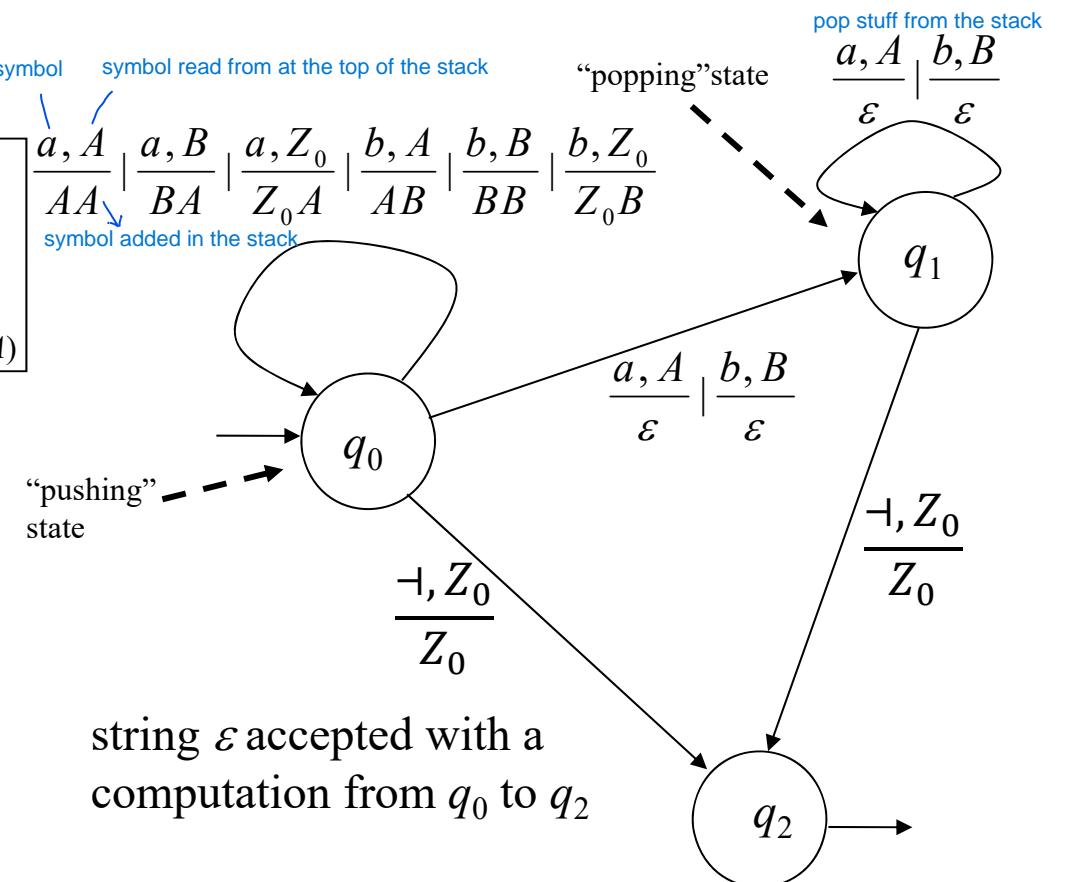
$$L = \left\{ uu^R \mid u \in \{a,b\}^* \right\}$$

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

“guesses” that $|x| > 2$

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

“guesses” that $|x| = 2$



FROM THE GRAMMAR TO THE PUSHDOWN AUTOMATON

- 1) Grammar rules seen as instructions of a ***non deterministic PDA*** with a single state; ***predictive*** (goal oriented) analysis: stack used as a list of future actions.

Only one state \Rightarrow called “***daisy automaton***”

- 2) Stack includes terminal and nonterminal symbols.

Stack = $A_1, \dots A_k$ goal is: read from input a string derived from A_k

- 3) The goal can be restated recursively in subgoals if A_k is a nonterminal from which other symbols are derived

Initial goal is the axiom: goal is to read a sentence, i.e., a string derived from axiom S

Initially stack is $Z_0 S$ and the input head on the first char of the input string

At each step the automaton chooses (nondeterministically) one of the applicable moves/rules

The string is recognized if the terminal \dashv is read with an empty stack

FROM GRAMMAR RULES TO TRANSITIONS $A, B \in V, b \in \Sigma, A_i \in V \cup \Sigma$

Rule	Move	Comment
$A \rightarrow BA_1\dots A_n \quad n \geq 0$	if $top = A$ then pop; push($A_n \dots A_1 B$) end if	to recognize A one must recognize $B A_1 \dots A_n$
$A \rightarrow bA_1\dots A_n \quad n \geq 0$	if $cc = b \wedge top = A$ then pop; push($A_n \dots A_1$); shift end if	b is the first expected char and is read; $A_1\dots A_n$ still to be accepted
$A \rightarrow \epsilon$	if $top = A$ then pop end if	ϵ deriving from A is accepted
for each char. $b \in \Sigma$	if $cc = b \wedge top = b$ then pop; shift end if	b is the first expected char and is read;
---	if $cc = \dashv \wedge$ empty stack then accept end if halt	string completely scanned, agenda completed

Example – Rules and moves of the predictive recognizer

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

<u>Rule</u>	<u>Moves</u>
1. $S \rightarrow aS$	$[\delta(q_0, a, S) = (q_0, S)]$ if $cc = a \wedge top = S$ then pop; push(S); shift end if
2. $S \rightarrow A$	$[\delta(q_0, \epsilon, S) = (q_0, A)]$ if $top = S$ then pop; push(A) end if
3. $A \rightarrow aAb$	$[\delta(q_0, a, A) = (q_0, bA)]$ if $cc = a \wedge top = A$ then pop; push(bA); shift end if
4. $A \rightarrow ab$	$[\delta(q_0, a, A) = (q_0, b)]$ if $cc = a \wedge top = A$ then pop; push(b); shift end if
5. --	$[\delta(q_0, b, b) = (q_0, \epsilon)]$ if $cc = b \wedge top = b$ then pop; shift end if
6. --	[halt] if $cc = \dashv \wedge$ empty stack then accept end if

Nondeterminism: between 1 and 2 (2 can also be chosen with input a); between 3 and 4

String $a^n b^m$, $n \geq m \geq 1$ analyzed as $a^{n-m} a^m b^m$,

“guessing nondeterministically” position where $a^m b^m$ starts, by choice between moves 1 and 2

“guess” of position where a^m ends and b^m starts by choice between moves 3 and 4

this automaton accepts a string iff the grammar generates it
for every accepting computation there is a derivation and viceversa
the automaton ***simulates the leftmost derivations***

the automaton must explore all derivations including nonaccepting ones
a **string is accepted by several computations iff it is ambiguous**

$S \Rightarrow A \Rightarrow aAb \Rightarrow aabb$
accepting computation:

Stack x

$\delta(q_0, \epsilon, S) = (q_0, A)$	$Z_0 S$	$aabb-$
$\delta(q_0, a, A) = (q_0, bA)$	$Z_0 A$	$aabb-$
$\delta(q_0, a, A) = (q_0, b)$	$Z_0 bA$	$abb-$
$\delta(q_0, b, b) = (q_0, \epsilon)$	$Z_0 bb$	$bb-$
$\delta(q_0, b, b) = (q_0, \epsilon)$	$Z_0 b$	$b-$
	Z_0	-

conversion grammar rules – PDA transitions is bidirectional:

One can also obtain in a systematic way a grammar from any PDA

Therefore:

LANGUAGES DEFINED BY **NONDETERMINISTIC** PDA's
AND CF LANGUAGES
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

Another important PDA feature: absence of spontaneous loops and on-line functioning

PROPERTY: any PDA can be converted into an equivalent one

- with no cycles of spontaneous moves
- which can decide acceptance right after reading the last input symbol

INTERSECTION OF REGULAR AND CONTEXT FREE LANGUAGES

We can now justify the statement:

Intersection of a ***CF*** and a ***REG*** language is ***CF***

Given grammar ***G*** and automaton ***A***, obtain PDA ***M*** accepting $L(\mathbf{G}) \cap L(\mathbf{A})$ as follows:

- 1) build PDA ***N*** accepting ***L(G)*** with empty stack
- 2) build machine ***M*** product of machines ***N*** and ***A***,
applying the known construction for finite automata adapted so that
the product machine ***M*** manipulates the stack in the same way as component ***N***

IDEA: resulting PDA «incorporates» the states of ***A***: it can check that input $x \in L(\mathbf{A})$

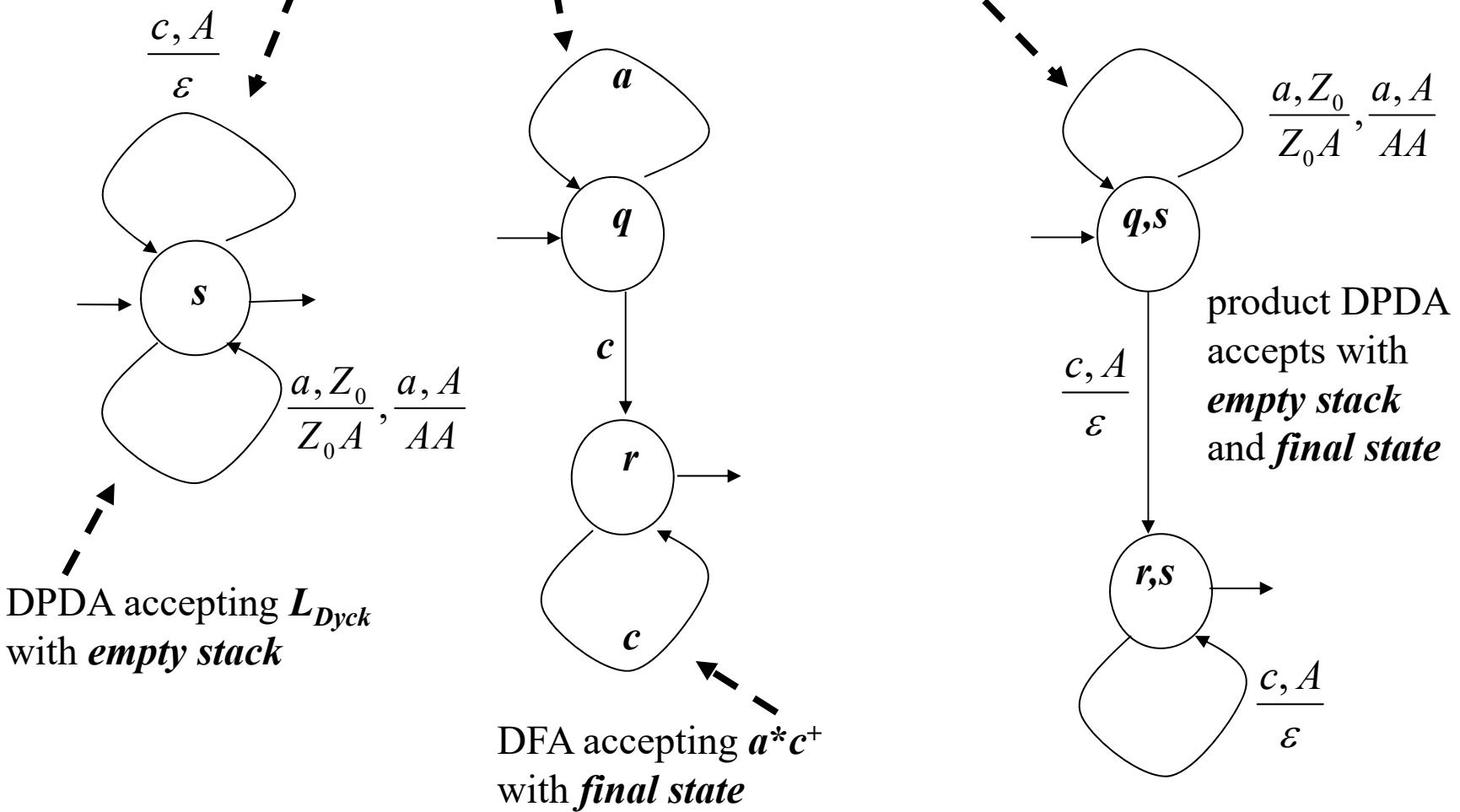
The machine thus constructed:

- has internal states that are the product of the state sets of the component machines
- accepts with **final state and empty stack**
- final states are those including a final state of the finite automaton ***A***
- is deterministic if so are both machines ***N*** and ***A***
- accepts exactly the strings of the intersection of the two languages

Example

$$L_{Dyck} \cap (a^*c^+) = \{a^n c^n \mid n \geq 1\}$$

Dyck language with one nest



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

if $\delta(q, a, A)$ is defined for some $a \in \Sigma$, then $\delta(q, \epsilon, A)$ is not defined
if $\delta(q, \epsilon, A)$ is defined then $\delta(q, a, A)$ is not defined for any $a \in \Sigma$

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

$$L = \{a^n b^n \mid n \geq 1\} \cup \{a^n b^{2n} \mid n \geq 1\} = L' \cup L''$$

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

$L', L'' \in \text{DET}$, $L', L'' \in \text{CF}$, $L = L' \cup L''$, $L \in \text{CF}$ but $L \notin \text{DET}$,

hence **$\text{DET} \subseteq \text{CF}$ and $\text{DET} \neq \text{CF}$**

CLOSURE PROPERTIES OF DETERMINISTIC CF LANGUAGES

We denote as L , D , and R a language in the family CF , DET and REG

Operation	Property	(Property already known)
Reflection	$D^R \notin DET$	$D^R \in CF$
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	$D_1.D_2 \notin DET, D.R \in DET$	$D_1.D_2 \in CF$
Intersection	$D \cap R \in DET$	$D_1 \cap D_2 \notin CF$

NB: the typical operations on languages (R , $*$, \cup , \cdot) **DO NOT** preserve determinism

Do not worry: we will find a procedure to determine if a given free *grammar* $\in DET$

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

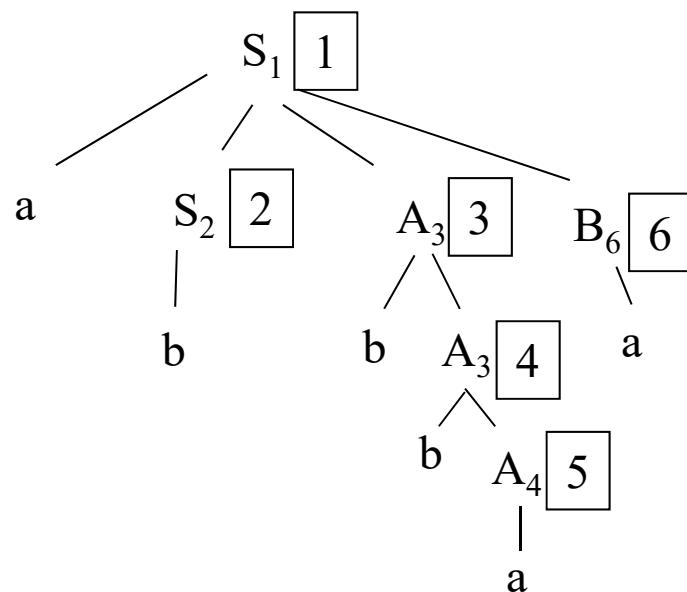
framed numbers = order in rule application

subscripts of nonterminals in the tree = applied rule

Leftmost: always expanded first nonterm. from the left

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

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



Example – bottom-up analysis of sentence:

a b b b a a

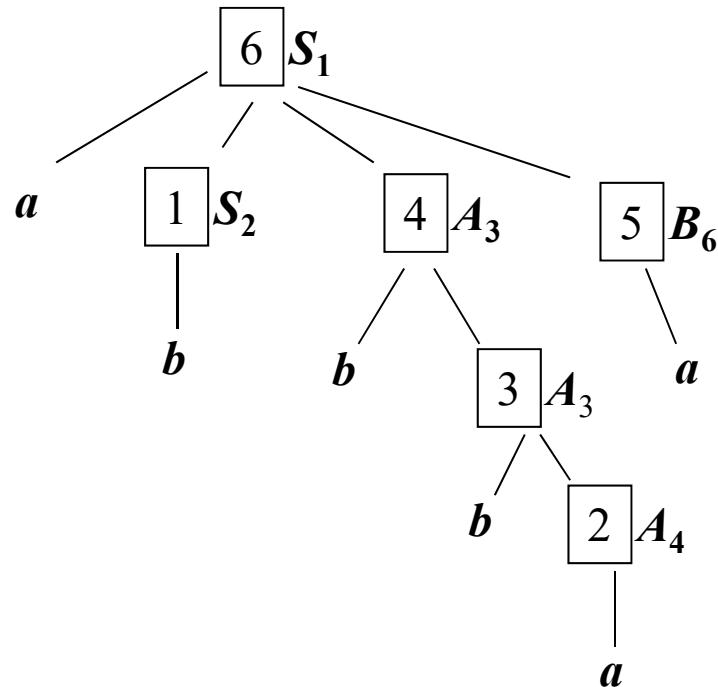
Derivation (rightmost n.t. always expanded):

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

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

framed numbers = order in reduction sequence

subscripts of nonterminals in the tree = applied rule

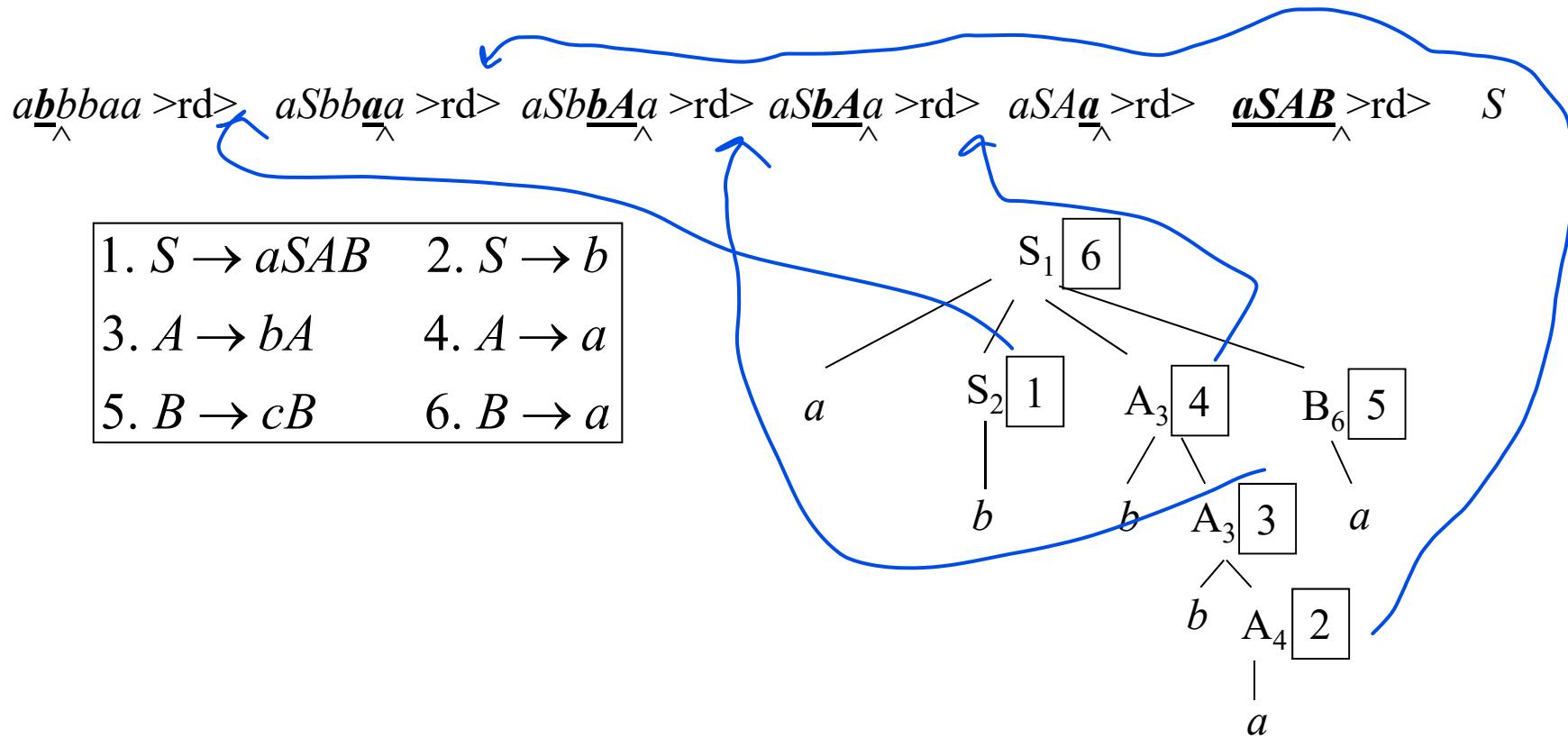


right parts of rules are *reduced* as they are scanned, first in the sentence, then in the phrase forms obtained after the *reductions*. The process terminates when the entire string is reduced to the axiom

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 ($>\text{rd}>$ below) transform 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 part to be reduced, called «*handle*»



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

GRAMMARS AS NETWORKS OF FINITE AUTOMATA

Suppose G in extended form: every nonterminal has a unique rule

$A \rightarrow \alpha$ with α regular expression on terminals and nonterminals

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

any transition of M_A labeled by a nonterminal B

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

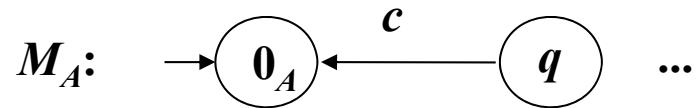
let us call

- “**machines**” the finite automata of the various nonterminals,
 - “**automaton**” the PDA that accepts and parses the language $L(G)$
 - “**net**” the set of all machines
-
- $L(q)$ the set of terminal strings generated along a path of a machine, starting from state q (possibly including calls of other machines) and reaching a final state (examples follow)

We set a further requirement on machines corresponding to nonterminals

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

No machine M_A may include a transition like



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 ...

⇒ the machine is not minimal, but only one state has been added

Automata satisfying this condition are called

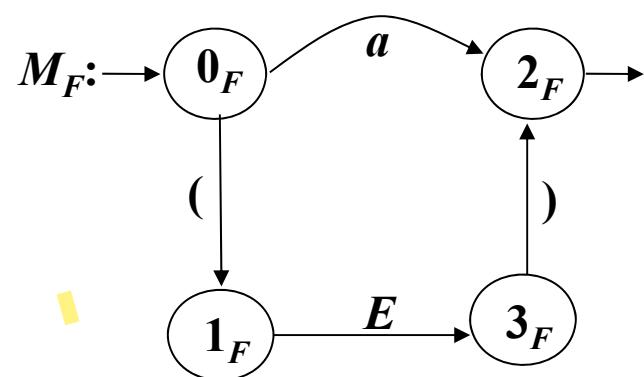
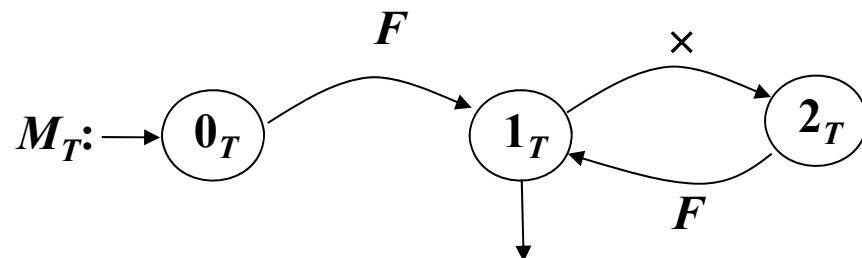
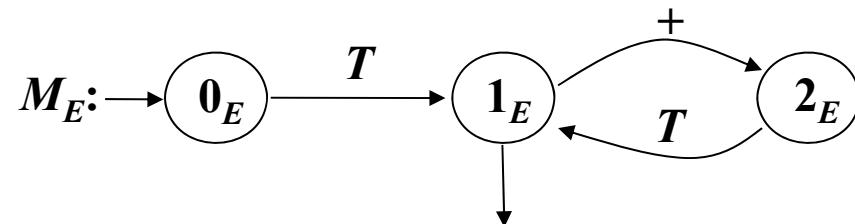
normalized or with initial state **non recirculating** or **non reentrant**

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

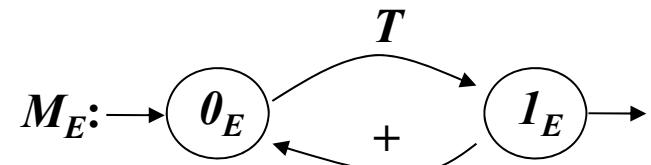
(which is necessary if $\epsilon \in L(A)$)

Example – Arithmetic expressions

$E \rightarrow T (+ T)^*$
 $T \rightarrow F (\times F)^*$
 $F \rightarrow a \mid (' E ')$



NB: M_E minimal not normalized



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

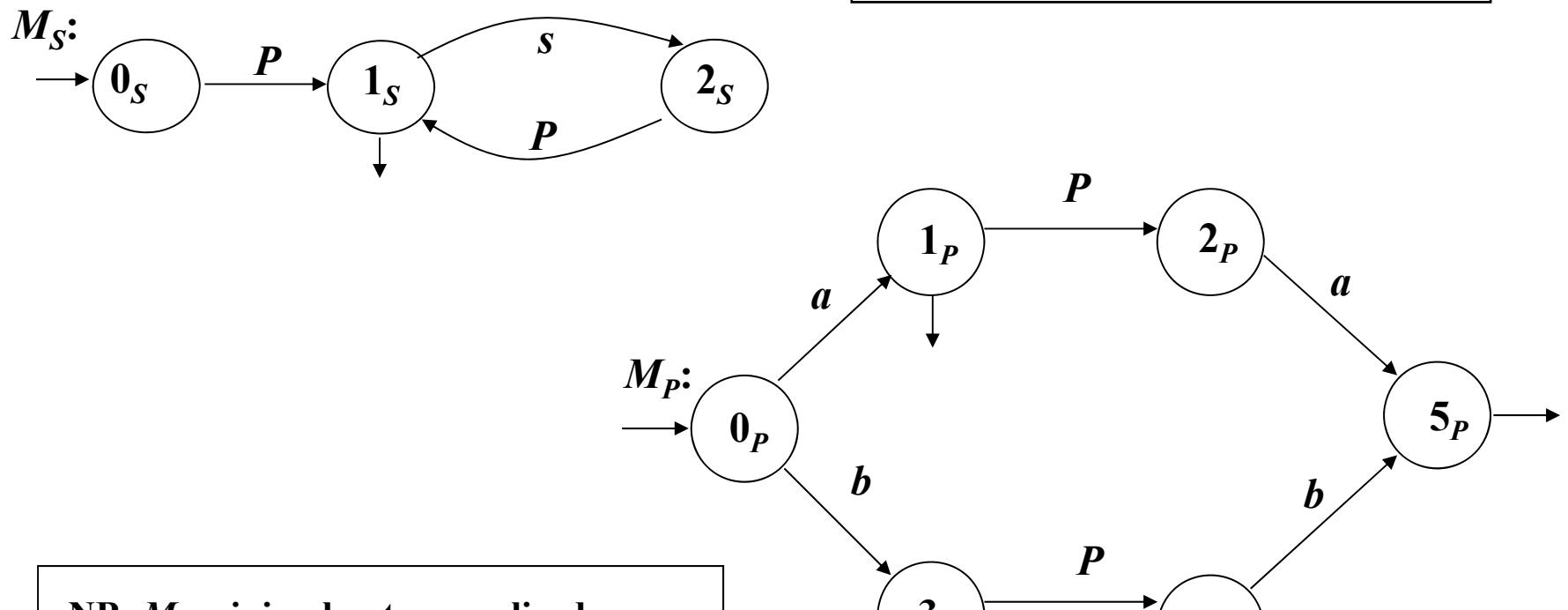
$L(3_F) = \{ \) \}$

$L(0_F) = L(F)$ the language generated from F

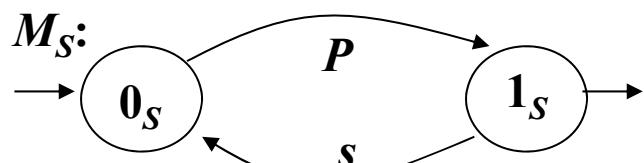
$L(1_F) = L(E) \cdot \{ \) \}$

Example – List of odd-length palindromes, separated by ‘s’

$$\boxed{S \rightarrow P (s P)^* \\ P \rightarrow aP a \mid bP b \mid a \mid b}$$



NB: M_S minimal not normalized



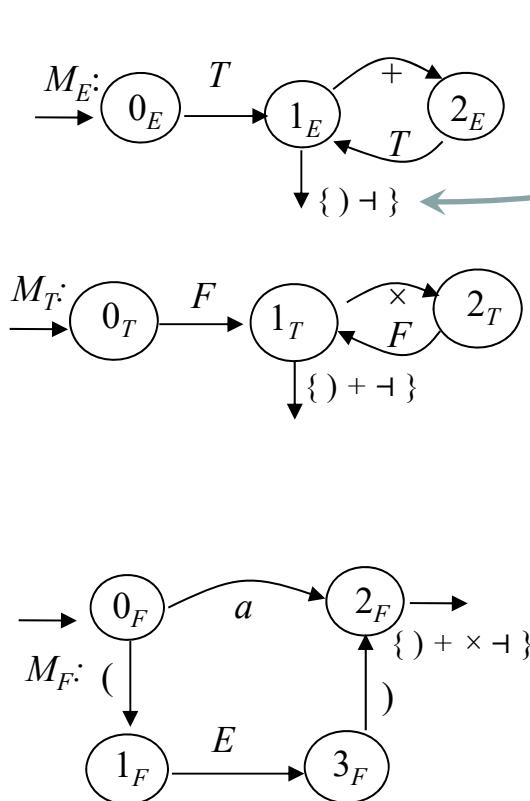
TOP-DOWN ANALYSIS WITH RECURSIVE PROCEDURES

Simple and elegant: one procedure for each nonterminal; the code reflects the transitions

$$\begin{aligned} E &\rightarrow T (+ T)^* \\ T &\rightarrow F (\times F)^* \\ F &\rightarrow a \mid (' E ') \end{aligned}$$

Example – Arithmetic expressions

When in the automaton there are **bifurcations**, to decide which one to follow one must consider the symbols encountered on every arc, including those encountered when «exiting the machine»



```
procedure E
call T;
while (cc = '+')
cc:=next;
call T;
end;
if (cc ∈ { ) -} )
return;
else error;
endif
end E;
```

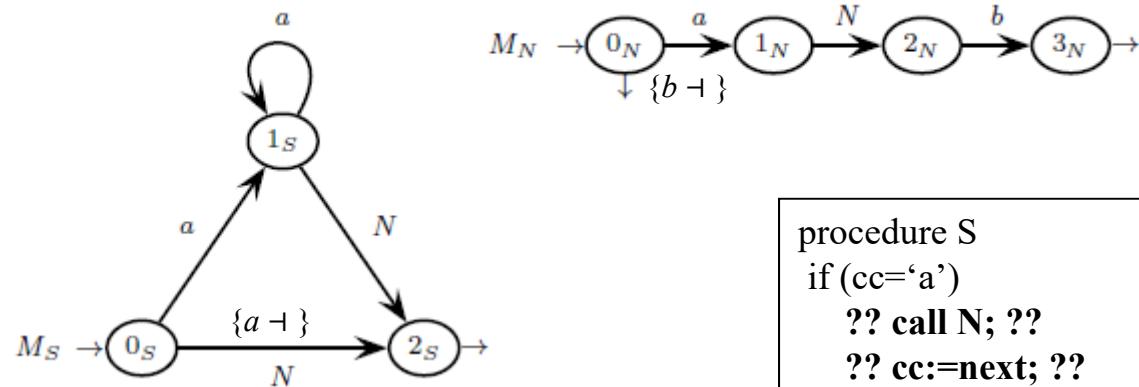
cc is current char,
symbol under input head
updated by the **next** procedure

```
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;
else error;
endif;
else error;
endif;
if (cc ∈ { ) + × -} )
return;
else error;
endif;
end F;
```

NB: this method **does not work** if
 the sets of possible symbols on arcs outgoing from a single state (the «guide sets»)
 are **not disjointed**

Example: $S \rightarrow a^* N$ $N \rightarrow aNb \mid \epsilon$ $L = \{ a^n b^m \mid n \geq m \geq 0 \}$



```

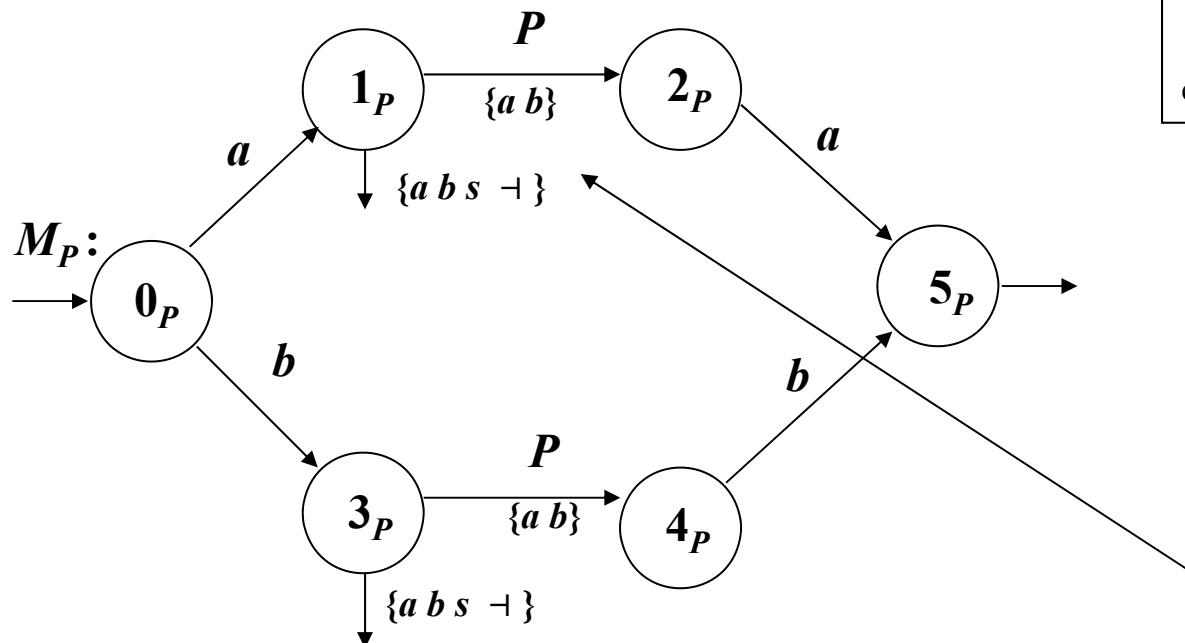
procedure S
if(cc='a')
    ?? call N; ??
    ?? cc:=next; ??
    ...
end S;
    
```

NB: this language/grammar cannot be analyzed by the top-down method, ...
 ... but it can be analyzed by the bottom-up method, which is more powerful

Another Example – List of odd-length palindromes, separated by an ‘s’

(NB: it is a nondeterministic language: even the bottom-up method does not work)

$$S \rightarrow P(sP)^*$$
$$P \rightarrow aPa \mid bPb \mid a \mid b$$



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

the recursive procedure cannot decide based on the next symbol

Bottom-up Syntax Analysis – $ELR(k)$

PART 1

Prof. A. Morzenti

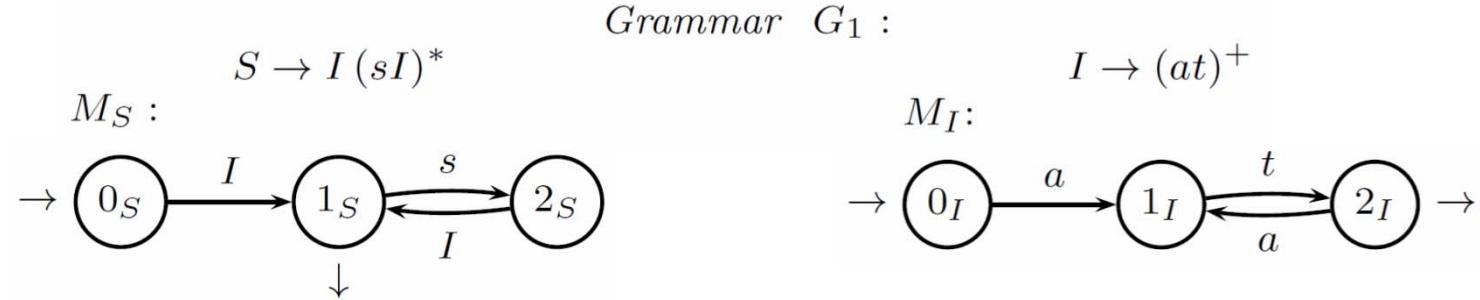
[How to build parsers using bottom up strategy](#)

Informal/intuitive example of bottom-up syntax analysis

example grammar for: sequence of lists $(at)^+$ separated by s

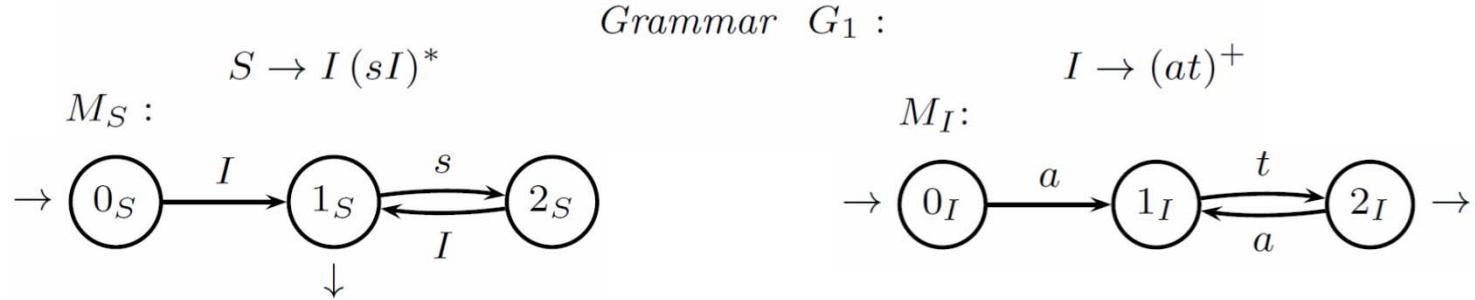
Informal/intuitive example of bottom-up syntax analysis

example grammar for: sequence of lists $(at)^+$ separated by s



Informal/intuitive example of bottom-up syntax analysis

example grammar for: sequence of lists $(at)^+$ separated by s

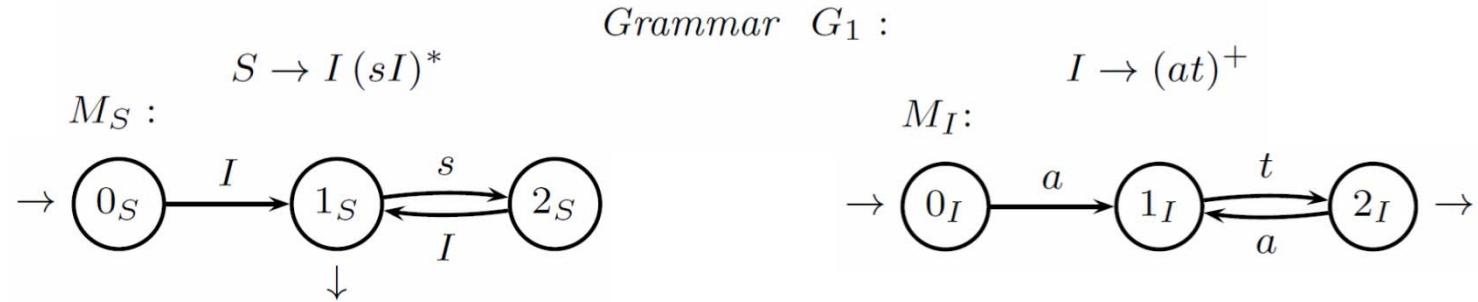


We add the sets of followers of nonterminals $\text{follow}(S)$ and $\text{follow}(I)$ useful for the analysis

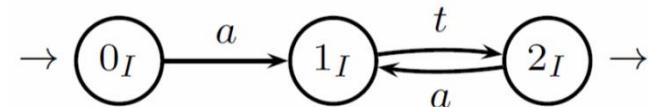
character that appear after a particular non terminal

Informal/intuitive example of bottom-up syntax analysis

example grammar for: sequence of lists $(at)^+$ separated by s

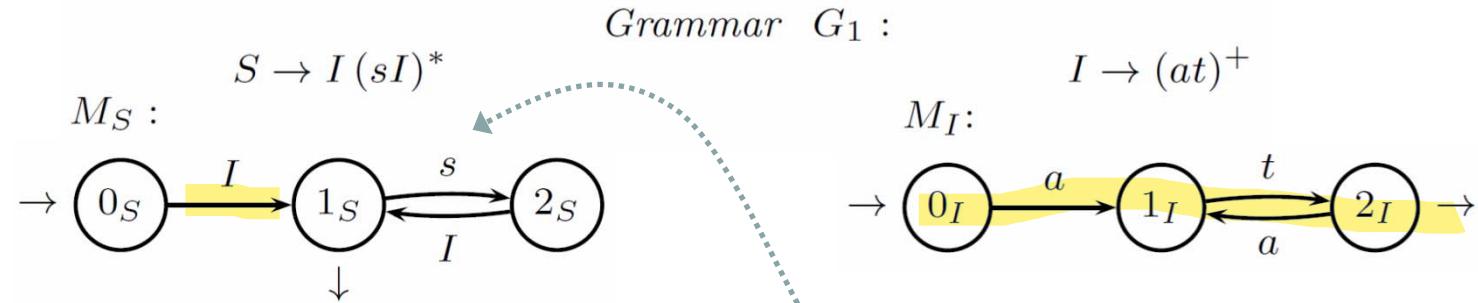


We add the *sets of followers* of nonterminals $\text{follow}(S)$ and $\text{follow}(I)$ useful for the analysis

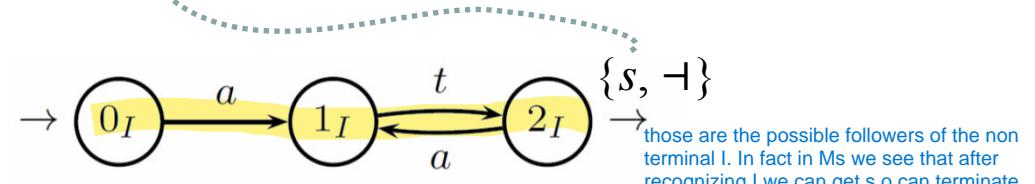


Informal/intuitive example of bottom-up syntax analysis

example grammar for: sequence of lists $(at)^+$ separated by s

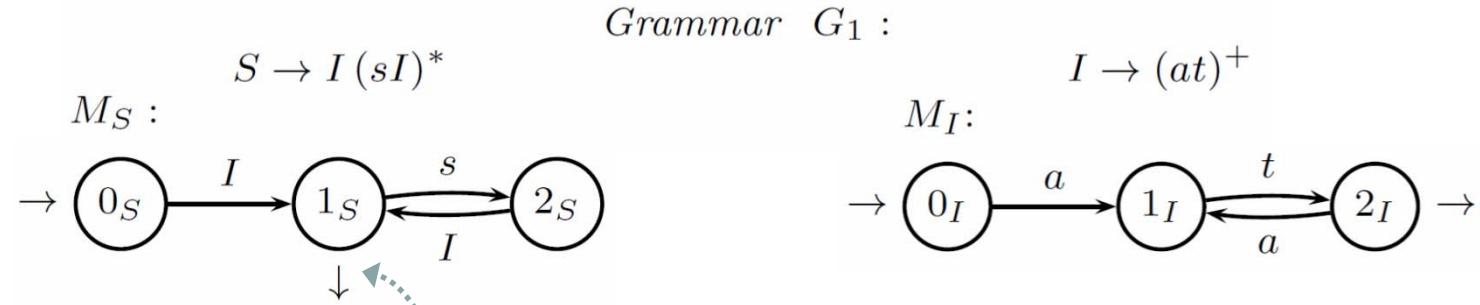


We add the *sets of followers* of nonterminals $\text{follow}(S)$ and $\text{follow}(I)$ useful for the analysis

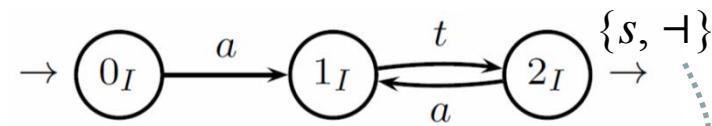


Informal/intuitive example of bottom-up syntax analysis

example grammar for: sequence of lists $(at)^+$ separated by s

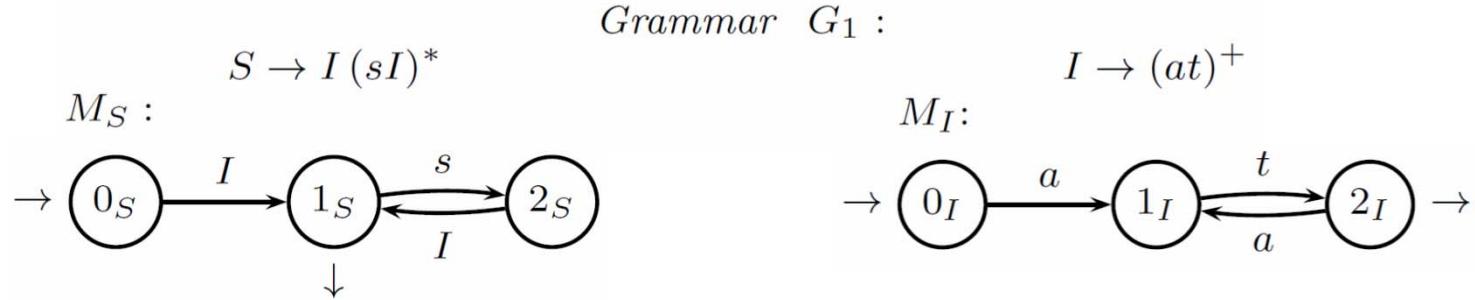


We add the *sets of followers* of nonterminals $\text{follow}(S)$ and $\text{follow}(I)$ useful for the analysis



Informal/intuitive example of bottom-up syntax analysis

example grammar for: sequence of lists $(at)^+$ separated by s

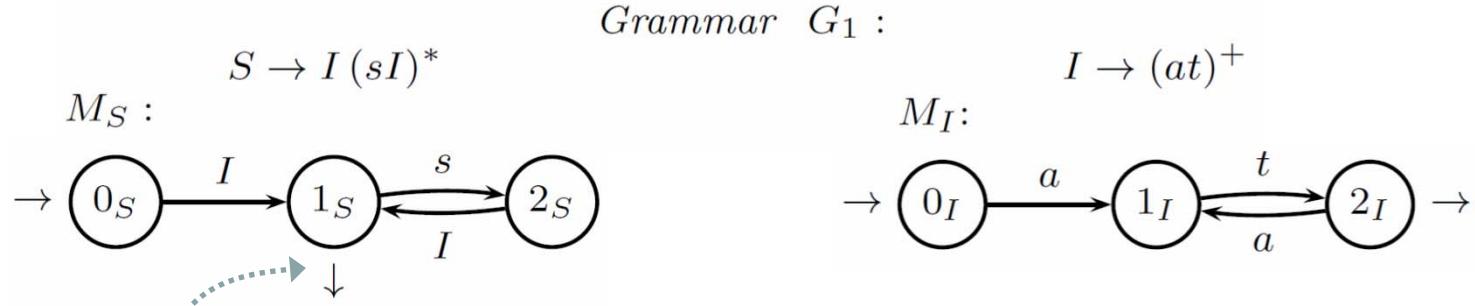


We add the *sets of followers* of nonterminals $\text{follow}(S)$ and $\text{follow}(I)$ useful for the analysis

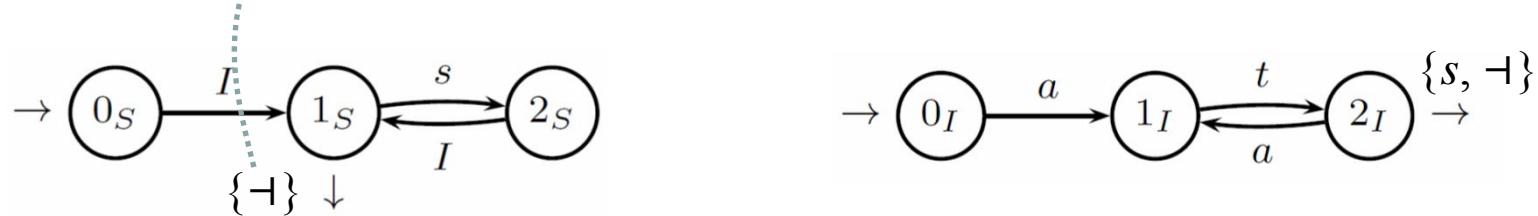


Informal/intuitive example of bottom-up syntax analysis

example grammar for: sequence of lists $(at)^+$ separated by s

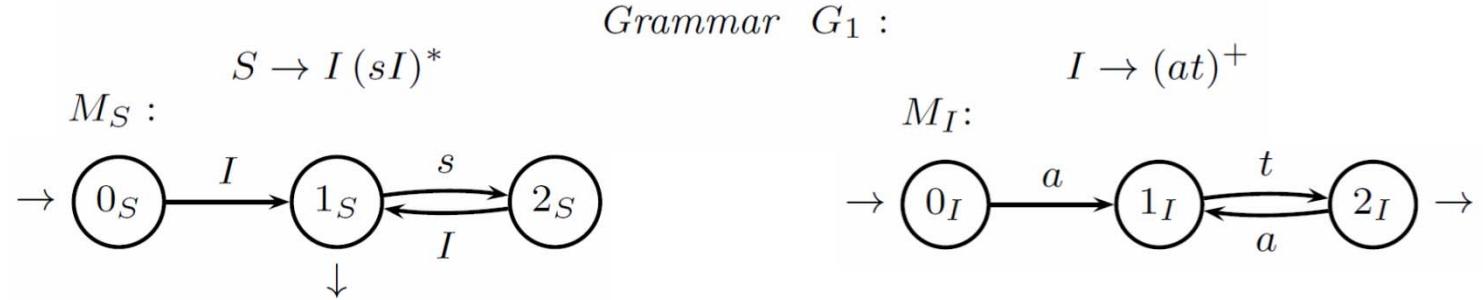


We add the *sets of followers* of nonterminals $\text{follow}(S)$ and $\text{follow}(I)$ useful for the analysis

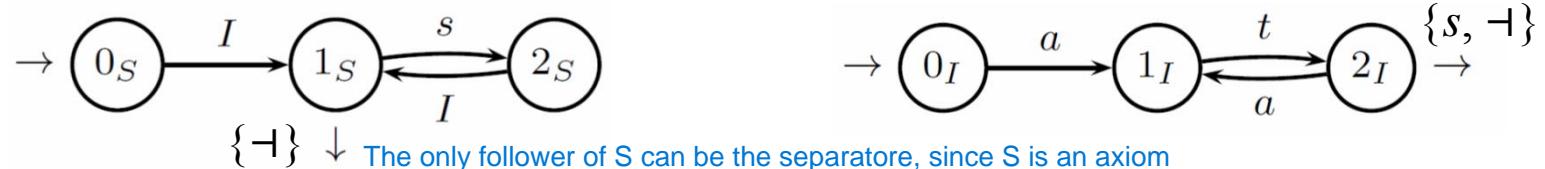


Informal/intuitive example of bottom-up syntax analysis

example grammar for: sequence of lists $(at)^+$ separated by s



We add the *sets of followers* of nonterminals $\text{follow}(S)$ and $\text{follow}(I)$ useful for the analysis

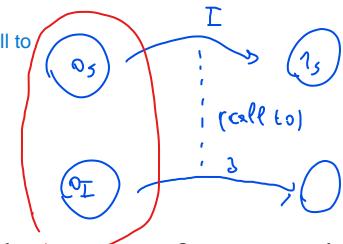


sets of followers (also called **lookahead**) allow the parser to decide when

- to continue the analysis following a machine transition (**shift move**), or
- to recognize a nonterminal and to build a subtree (**reduce move**)

Trace of the analysis for string $x = atatsat \dashv$

From 0_S the only way to exit is to recognize I . However this is a call to the I recognizer automata, which starts with recognizing a



Trace of the analysis for string $x = atatsat \dashv$

Analysis of S starts with that of I : a sort of ϵ -move (called **closure**)

It pushes on the stack the **macro-state** (m-state) $\{0_S, 0_I\}$ including the initial states of M_S and M_I

	Stack	x						Comment
1	$\{0_S, 0_I\}$	a	t	a	t	s	a	$t \dashv$ shift

Trace of the analysis for string $x = atatsat \dashv$

Analysis of S starts with that of I : a sort of ϵ -move (called **closure**)

It pushes on the stack the **macro-state** (m-state) $\{\mathbf{0}_S, \mathbf{0}_I\}$ including the initial states of M_S and M_I

	<i>Stack</i>	<i>x</i>					<i>Comment</i>
1	$\{\mathbf{0}_S, \mathbf{0}_I\}$	a	t	a	t	$s a t \dashv$	shift

read a , transition $0_I \xrightarrow{a} 1_I$, macrostate $\{1_I\}$ is pushed

2	$\{\mathbf{0}_S, \mathbf{0}_I\}$	$a \{1_I\}$	t	a	t	$s a t \dashv$	shift
---	----------------------------------	-------------	-----	-----	-----	----------------	-------

Trace of the analysis for string $x = atatsat \vdash$

Analysis of S starts with that of I : a sort of ε -move (called ***closure***)

It pushes on the stack the ***macro-state*** (m-state) $\{0_S, 0_I\}$ including the initial states of M_S and M_I

	Stack		x					Comment	
1	$\{0_S, 0_I\}$	a	t	a	t	s	a	$t \vdash$	shift

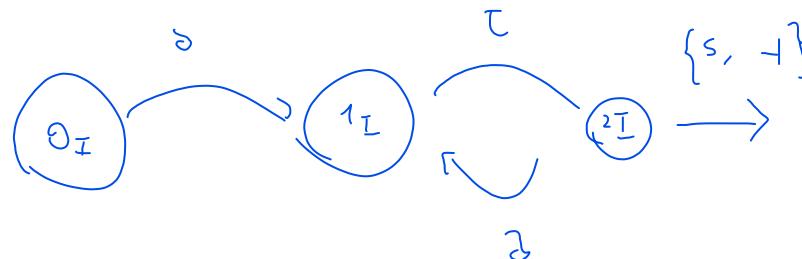
read a , transition $0_I \xrightarrow{a} 1_I$, macrostate $\{1_I\}$ is pushed

2	$\{0_S, 0_I\}$	a	$\{1_I\}$	t	a	t	s	a	$t \vdash$
									shift

after shift of t first choice: reduce or shift? it shifts because current char $cc=a \notin follow(I)$
remind that $follow(A)$ often called ***lookahead set***

3	$\{0_S, 0_I\}$	a	$\{1_I\}$	t	$\{2_I\}$	<u>a</u>	t	s	a	$t \vdash$	no reduce, do shift	
4	$\{0_S, 0_I\}$	a	$\{1_I\}$	t	$\{2_I\}$	a	$\{1_I\}$	t	s	a	$t \vdash$	shift

In step 3 the next character is a , we don't know if we should go to the "a" arc or to the s (termination) arc and do the reduction.



Trace of the analysis for string $x = atatsat \dashv$

Analysis of S starts with that of I : a sort of ϵ -move (called ***closure***)

It pushes on the stack the ***macro-state*** (m-state) $\{\mathbf{0}_S, \mathbf{0}_I\}$ including the initial states of M_S and M_I

	Stack		x				Comment
1	$\{\mathbf{0}_S, \mathbf{0}_I\}$	a	t	a	t	$s a t \dashv$	shift

read a , transition $0_I \xrightarrow{a} 1_I$, macrostate $\{1_I\}$ is pushed

2	$\{\mathbf{0}_S, \mathbf{0}_I\}$	$a \{1_I\}$	t	a	t	$s a t \dashv$	shift
---	----------------------------------	-------------	-----	-----	-----	----------------	-------

after shift of t first choice: reduce or shift? it shifts because current char $cc=a \notin follow(I)$
 remind that $follow(A)$ often called ***lookahead set***

3	$\{\mathbf{0}_S, \mathbf{0}_I\}$	$a \{1_I\}$	$t \{2_I\}$	a	t	$s a t \dashv$	no reduce, do shift
4	$\{\mathbf{0}_S, \mathbf{0}_I\}$	$a \{1_I\}$	$t \{2_I\}$	$a \{1_I\}$	t	$s a t \dashv$	shift

next (below) second choice between reduce and shift: it reduces because $cc=s \in follow(I)$
 which is the reduction handle? Go back in the stack until initial state of M_I : $\mathbf{0}_I$
 but watch out: there are cases in which things are more complex ...

	$\overbrace{\{\mathbf{0}_S, \mathbf{0}_I\} \ a \{1_I\} \ t \{2_I\} \ a \{1_I\} \ t \{2_I\}}^{\text{segment to be reduced to } I}$						
5	$\{\mathbf{0}_S, \mathbf{0}_I\}$	$a \{1_I\}$	$t \{2_I\}$	$a \{1_I\}$	$t \{2_I\}$	$s a t \dashv$	s follows I : reduce $atat \rightsquigarrow I$

--> reduction

Since we started from $\mathbf{0}_I$, the reduction must involve everything we read up until that state: pop everything from the stack and replace it with just I .

Trace of the analysis for string $x = atatsat \vdash$

Analysis of S starts with that of I : a sort of ε -move (called ***closure***)

It pushes on the stack the ***macro-state*** (m-state) $\{\mathbf{0}_S, \mathbf{0}_I\}$ including the initial states of M_S and M_I

	Stack		x				Comment
1	$\{\mathbf{0}_S, \mathbf{0}_I\}$	a	t	a	t	$s a t \vdash$	shift

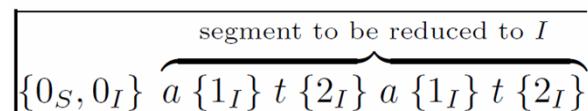
read a , transition $0_I \xrightarrow{a} 1_I$, macrostate $\{1_I\}$ is pushed

2	$\{\mathbf{0}_S, \mathbf{0}_I\}$	$a \{1_I\}$	t	a	t	$s a t \vdash$	shift
---	----------------------------------	-------------	-----	-----	-----	----------------	-------

after shift of t first choice: reduce or shift? it shifts because current char $cc=a \notin follow(I)$
remind that $follow(A)$ often called ***lookahead set***

3	$\{\mathbf{0}_S, \mathbf{0}_I\}$	$a \{1_I\}$	$t \{2_I\}$	a	t	$s a t \vdash$	no reduce, do shift
4	$\{\mathbf{0}_S, \mathbf{0}_I\}$	$a \{1_I\}$	$t \{2_I\}$	$a \{1_I\}$	t	$s a t \vdash$	shift

next (below) second choice between reduce and shift: it reduces because $cc=s \in follow(I)$
which is the reduction handle? Go back in the stack until initial state of M_I : $\mathbf{0}_I$
but watch out: there are cases in which things are more complex ...



5	$\{\mathbf{0}_S, \mathbf{0}_I\}$	$a \{1_I\}$	$t \{2_I\}$	$a \{1_I\}$	$t \{2_I\}$	$s a t \vdash$	s follows I : reduce $atat \rightsquigarrow I$
---	----------------------------------	-------------	-------------	-------------	-------------	----------------	---

the reduction builds the first fragment of the syntax tree

\overbrace{atat}^I

FIRST REDUCTION
(FIRST PART OF SYNTAX TREE)

the reduction $a \rightarrow I$ is registered in the stack by means of a **NONterminal shift**

6	$\{0_S, 0_I\}$	$s \ a \ t \dashv$	nonterminal shift $0_S \xrightarrow{I} 1_S$
7	$\{0_S, 0_I\} \ I \ \{1_S\}$	$s \ a \ t \dashv$	no reduce, do shift

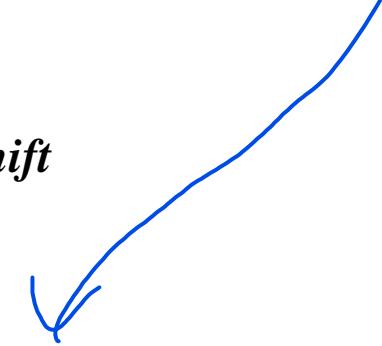
Dopo averlo shiftato
riconserviamo shiftato
di un non-terminal
--> since there's an s we can't reduce (no way to exit from automata)

Since we've recognized I (with the reduction) we now are in state 1s

The goal of this bottom up recognizer is to emulate a non deterministic recognizer

the reduction $a \rightarrow I$ is registered in the stack by means of a ***NONterminal shift***

6	$\{0_S, 0_I\}$	$s \ a \ t \dashv$	nonterminal shift $0_S \xrightarrow{I} 1_S$
7	$\{0_S, 0_I\} \ I \ \{1_S\}$	$s \ a \ t \dashv$	no reduce, do shift



then another choice between reduce and shift: it executes a **shift** because $cc=s \notin follow(S)$

the reduction $at \rightarrow I$ is registered in the stack by means of a ***NONterminal shift***

6	$\{0_S, 0_I\}$		$s \ a \ t \ \dashv$	nonterminal shift $0_S \xrightarrow{I} 1_S$
7	$\{0_S, 0_I\} \ I \ \{1_S\}$		$s \ a \ t \ \dashv$	no reduce, do shift

then another choice between reduce and shift: it executes a shift because $cc = s \notin follow(S)$

8	$\{0_S, 0_I\} \ I \ \{1_S\} \ s \ \{2_S, 0_I\}$		$a \ t \ \dashv$	shift
9	$\{0_S, 0_I\} \ I \ \{1_S\} \ s \ \{2_S, 0_I\} \ a \ \{1_I\}$		$t \ \dashv$	shift
10	$\{0_S, 0_I\} \ I \ \{1_S\} \ s \ \{2_S, 0_I\} \ a \ \{1_I\} \ t \ \{2_I\}$		\dashv	\dashv follows I ; reduce $at \rightsquigarrow I$

then a **reduction $at \rightarrow I$** because $cc = \dashv \in follow(I)$

syntax tree fragment

\overbrace{at}^I

the reduction $at \rightarrow I$ is registered in the stack by means of a ***NONterminal shift***

6	$\{0_S, 0_I\}$	$s \ a \ t \dashv$	nonterminal shift $0_S \xrightarrow{I} 1_S$
7	$\{0_S, 0_I\} \ I \{1_S\}$	$s \ a \ t \dashv$	no reduce, do shift

then another choice between reduce and shift: it executes a shift because $cc = s \notin follow(S)$

8	$\{0_S, 0_I\} \ I \{1_S\} \ s \{2_S, 0_I\}$	$a \ t \dashv$	shift
9	$\{0_S, 0_I\} \ I \{1_S\} \ s \{2_S, 0_I\} \ a \{1_I\}$	$t \dashv$	shift
10	$\{0_S, 0_I\} \ I \{1_S\} \ s \{2_S, 0_I\} \ a \{1_I\} \ t \{2_I\}$	\dashv	\dashv follows I ; reduce $at \rightsquigarrow I$

then a reduction $at \rightarrow I$ because $cc = \dashv \in follow(I)$

syntax tree fragment



nonterminal shift



11	$\{0_S, 0_I\} \ I \{1_S\} \ s \{2_S, 0_I\}$	\dashv	nonterminal shift $2_S \xrightarrow{I} 1_S$
12	$\{0_S, 0_I\} \ I \{1_S\} \ s \{2_S, 0_I\} \ I \{1_S\}$	\dashv	reduce $IsI \rightsquigarrow S$

the reduction $at \rightarrow I$ is registered in the stack by means of a ***NONterminal shift***

6	$\{0_S, 0_I\}$	$s \ a \ t \dashv$	nonterminal shift $0_S \xrightarrow{I} 1_S$
7	$\{0_S, 0_I\} \ I \{1_S\}$	$s \ a \ t \dashv$	no reduce, do shift

then another choice between reduce and shift: it executes a shift because $cc = s \notin follow(S)$

8	$\{0_S, 0_I\} \ I \{1_S\} \ s \{2_S, 0_I\}$	$a \ t \dashv$	shift
9	$\{0_S, 0_I\} \ I \{1_S\} \ s \{2_S, 0_I\} \ a \{1_I\}$	$t \dashv$	shift
10	$\{0_S, 0_I\} \ I \{1_S\} \ s \{2_S, 0_I\} \ a \{1_I\} \ t \{2_I\}$	\dashv	\dashv follows I ; reduce $at \rightsquigarrow I$

then a reduction $at \rightarrow I$ because $cc = \dashv \in follow(I)$

syntax tree fragment

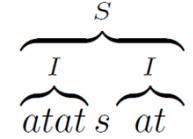


nonterminal shift

11	$\{0_S, 0_I\} \ I \{1_S\} \ s \{2_S, 0_I\}$	\dashv	nonterminal shift $2_S \xrightarrow{I} 1_S$
12	$\{0_S, 0_I\} \ I \{1_S\} \ s \{2_S, 0_I\} \ I \{1_S\}$	\dashv	reduce $IsI \rightsquigarrow S$

then a reduction $IsI \rightarrow S$ because $cc = \dashv \in follow(S)$

syntax tree fragment



the reduction $at \rightarrow I$ is registered in the stack by means of a **NONterminal shift**

6	$\{0_S, 0_I\}$	$s \ a \ t \dashv$	nonterminal shift $0_S \xrightarrow{I} 1_S$
7	$\{0_S, 0_I\} \ I \ \{1_S\}$	$s \ a \ t \dashv$	no reduce, do shift

then another choice between reduce and shift: it executes a shift because $cc = s \notin follow(S)$

8	$\{0_S, 0_I\} \ I \ \{1_S\} \ s \ \{2_S, 0_I\}$	$a \ t \dashv$	shift
9	$\{0_S, 0_I\} \ I \ \{1_S\} \ s \ \{2_S, 0_I\} \ a \ \{1_I\}$	$t \dashv$	shift
10	$\{0_S, 0_I\} \ I \ \{1_S\} \ s \ \{2_S, 0_I\} \ a \ \{1_I\} \ t \ \{2_I\}$	\dashv	\dashv follows I ; reduce $at \rightsquigarrow I$

then a reduction $at \rightarrow I$ because $cc = \dashv \in follow(I)$

syntax tree fragment

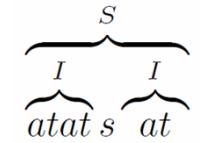


nonterminal shift

11	$\{0_S, 0_I\} \ I \ \{1_S\} \ s \ \{2_S, 0_I\}$	\dashv	nonterminal shift $2_S \xrightarrow{I} 1_S$
12	$\{0_S, 0_I\} \ I \ \{1_S\} \ s \ \{2_S, 0_I\} \ I \ \{1_S\}$	\dashv	reduce $IsI \rightsquigarrow S$

then a reduction $IsI \rightarrow S$ because $cc = \dashv \in follow(S)$

syntax tree fragment



13	$\{0_S, 0_I\}$	\dashv	recognize
----	----------------	----------	-----------

Condition for acceptance:

a **reduction to S** has just been executed and
the stack contains **initial state 0_S** of M_S
input is $cc = \dashv$

\Rightarrow the string is completely scanned and derives from axiom S

the reduction $at \rightarrow I$ is registered in the stack by means of a **NONterminal shift**

6	$\{0_S, 0_I\}$	$s \ a \ t \dashv$	nonterminal shift $0_S \xrightarrow{I} 1_S$
7	$\{0_S, 0_I\} \ I \{1_S\}$	$s \ a \ t \dashv$	no reduce, do shift

then another choice between reduce and shift: it executes a shift because $cc = s \notin follow(S)$

8	$\{0_S, 0_I\} \ I \{1_S\} \ s \{2_S, 0_I\}$	$a \ t \dashv$	shift
9	$\{0_S, 0_I\} \ I \{1_S\} \ s \{2_S, 0_I\} \ a \{1_I\}$	$t \dashv$	shift
10	$\{0_S, 0_I\} \ I \{1_S\} \ s \{2_S, 0_I\} \ a \{1_I\} \ t \{2_I\}$	\dashv	\dashv follows I ; reduce $at \rightsquigarrow I$

then a reduction $at \rightarrow I$ because $cc = \dashv \in follow(I)$

syntax tree fragment

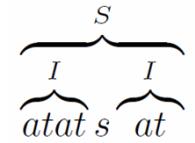


nonterminal shift

11	$\{0_S, 0_I\} \ I \{1_S\} \ s \{2_S, 0_I\}$	\dashv	nonterminal shift $2_S \xrightarrow{I} 1_S$
12	$\{0_S, 0_I\} \ I \{1_S\} \ s \{2_S, 0_I\} \ I \{1_S\}$	\dashv	reduce $IsI \rightsquigarrow S$

then a reduction $IsI \rightarrow S$ because $cc = \dashv \in follow(S)$

syntax tree fragment



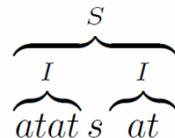
13	$\{0_S, 0_I\}$	\dashv	recognize
----	----------------	----------	-----------

Condition for acceptance:

a **reduction to S** has just been executed and
the stack contains **initial state 0_S** of M_S
input is $cc = \dashv$

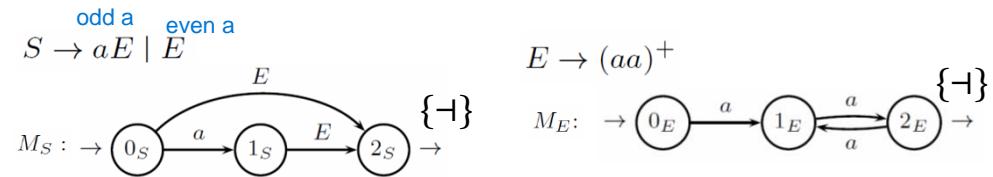
\Rightarrow the string is completely scanned and derives from axiom S

Complete syntax tree



it is not always so easy to choose the correct reduction: it may be necessary to advance further in the string...

Example: another grammar G_2



even-length strings $(aa)^+ = L(0_E)$ and odd-length $a(aa)^+ = a \cdot L(1_S)$ derived very differently

derivation becomes clear only after complete scan, when the terminator \vdash is encountered

Example: analysis of string $aaa\vdash$

computation that generates it: $0_S \xrightarrow{a} 1_S \xrightarrow{\varepsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$ NB: state 2_E is final

But there is another computation, which generates $aaaa\vdash$: $0_S \xrightarrow{\varepsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$

the parser, while scanning the string, «doesn't know», and investigates both hypotheses

$$0_S \xrightarrow{a} 1_S$$

$$0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$$

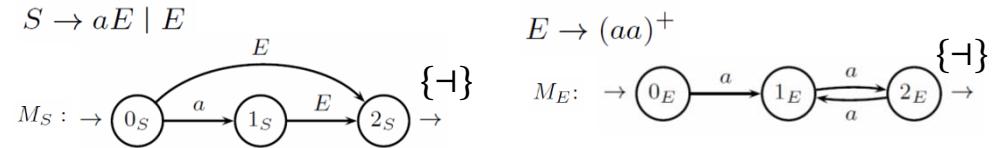
$$0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$$

Until you count the number on a's you will not be able to distinguish if you recognize an odd number or the even number.

we need pointers to let us recognize which is the starting point --> we need to add into the stack those pointers

it is not always so easy to choose the correct reduction: it may be necessary to advance further in the string...

Example: another grammar G_2



even-length strings $(aa)^+ = L(0_E)$ and odd-length $a(aa)^+ = a \cdot L(1_S)$ derived very differently

derivation becomes clear only after complete scan, when the terminator \vdash is encountered

Example: analysis of string $aaa\vdash$

computation that generates it: $0_S \xrightarrow{a} 1_S \xrightarrow{\varepsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$ NB: state 2_E is final

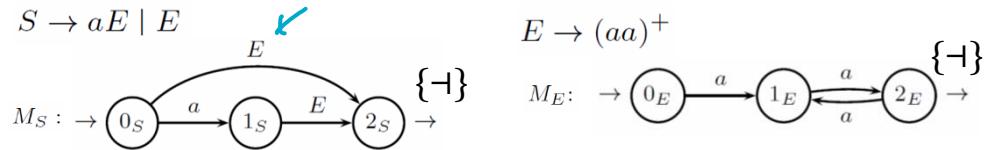
But there is another computation, which generates $aaaa\vdash$: $0_S \xrightarrow{\varepsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$

the parser, while scanning the string, «doesn't know», and investigates both hypotheses

0_S

it is not always so easy to choose the correct reduction: it may be necessary to advance further in the string...

Example: another grammar G_2



even-length strings $(aa)^+ = L(0_E)$ and odd-length $a(aa)^+ = a \cdot L(1_S)$ derived very differently

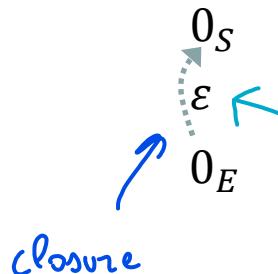
derivation becomes clear only after complete scan, when the terminator \dashv is encountered

Example: analysis of string $aaa\dashv$

computation that generates it: $0_S \xrightarrow{a} 1_S \xrightarrow{\epsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$ NB: state 2_E is final

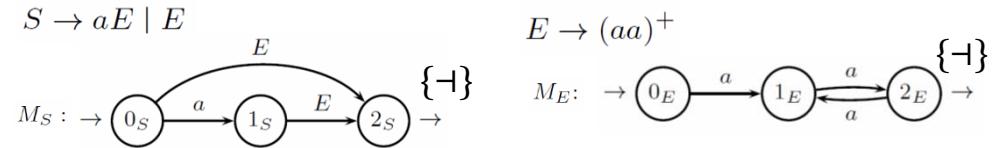
But there is another computation, which generates $aaaa\dashv$: $0_S \xrightarrow{\epsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$

the parser, while scanning the string, «doesn't know», and investigates both hypotheses



it is not always so easy to choose the correct reduction: it may be necessary to advance further in the string...

Example: another grammar G_2



even-length strings $(aa)^+ = L(0_E)$ and odd-length $a(aa)^+ = a \cdot L(1_S)$ derived very differently

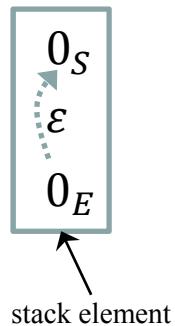
derivation becomes clear only after complete scan, when the terminator \vdash is encountered

Example: analysis of string $aaa\vdash$

computation that generates it: $0_S \xrightarrow{a} 1_S \xrightarrow{\varepsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$ NB: state 2_E is final

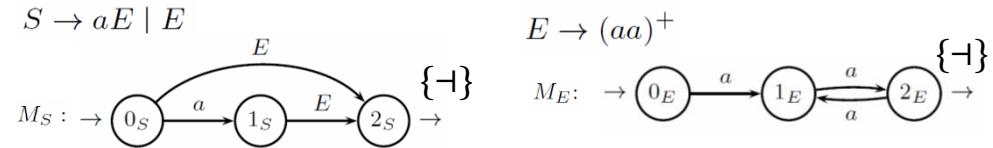
But there is another computation, which generates $aaaa\vdash$: $0_S \xrightarrow{\varepsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$

the parser, while scanning the string, «doesn't know», and investigates both hypotheses



it is not always so easy to choose the correct reduction: it may be necessary to advance further in the string...

Example: another grammar G_2



even-length strings $(aa)^+ = L(0_E)$ and odd-length $a(aa)^+ = a \cdot L(1_S)$ derived very differently

derivation becomes clear only after complete scan, when the terminator \vdash is encountered

Example: analysis of string $aaa\vdash$

computation that generates it: $0_S \xrightarrow{a} 1_S \xrightarrow{\varepsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$ NB: state 2_E is final

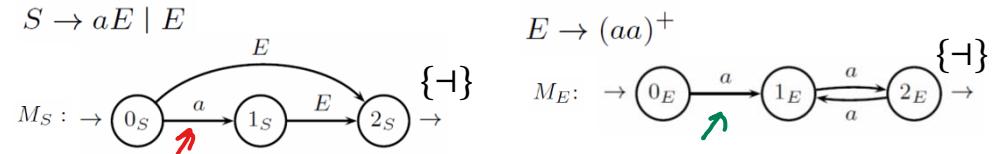
But there is another computation, which generates $aaaa\vdash$: $0_S \xrightarrow{\varepsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$

the parser, while scanning the string, «doesn't know», and investigates both hypotheses



it is not always so easy to choose the correct reduction: it may be necessary to advance further in the string...

Example: another grammar G_2



even-length strings $(aa)^+ = L(0_E)$ and odd-length $a(aa)^+ = a \cdot L(1_S)$ derived very differently

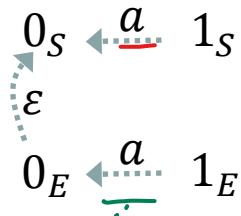
derivation becomes clear only after complete scan, when the terminator \vdash is encountered

Example: analysis of string $aaa\vdash$

computation that generates it: $0_S \xrightarrow{a} 1_S \xrightarrow{\varepsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$ NB: state 2_E is final

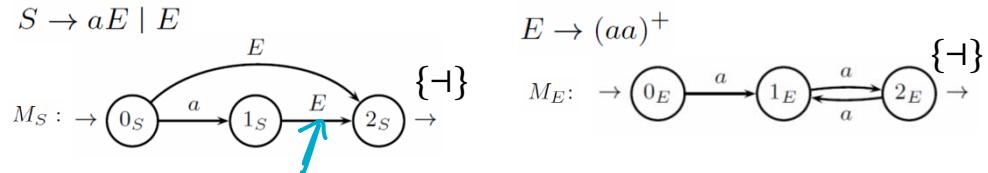
But there is another computation, which generates $aaaa\vdash$: $0_S \xrightarrow{\varepsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$

the parser, while scanning the string, «doesn't know», and investigates both hypotheses



it is not always so easy to choose the correct reduction: it may be necessary to advance further in the string...

Example: another grammar G_2



even-length strings $(aa)^+ = L(0_E)$ and odd-length $a(aa)^+ = a \cdot L(1_S)$ derived very differently

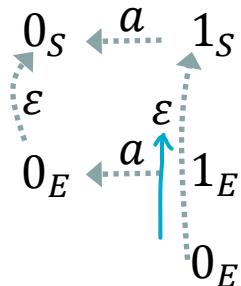
derivation becomes clear only after complete scan, when the terminator \vdash is encountered

Example: analysis of string $aaa\vdash$

computation that generates it: $0_S \xrightarrow{a} 1_S \xrightarrow{\varepsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$ NB: state 2_E is final

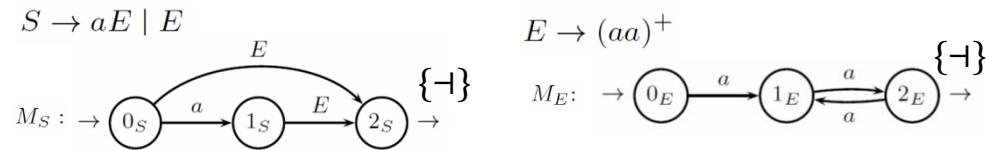
But there is another computation, which generates $aaaa\vdash$: $0_S \xrightarrow{\varepsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$

the parser, while scanning the string, «doesn't know», and investigates both hypotheses



it is not always so easy to choose the correct reduction: it may be necessary to advance further in the string...

Example: another grammar G_2



even-length strings $(aa)^+ = L(0_E)$ and odd-length $a(aa)^+ = a \cdot L(1_S)$ derived very differently

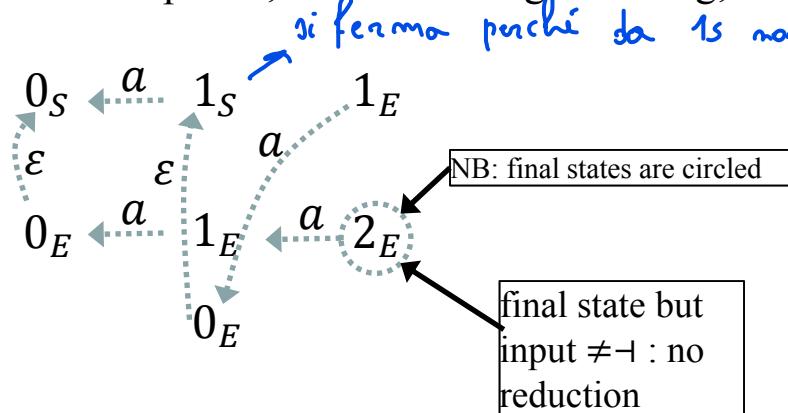
derivation becomes clear only after complete scan, when the terminator \vdash is encountered

Example: analysis of string $aaa\vdash$

computation that generates it: $0_S \xrightarrow{a} 1_S \xrightarrow{\varepsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$ NB: state 2_E is final

But there is another computation, which generates $aaaa\vdash$: $0_S \xrightarrow{\varepsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$

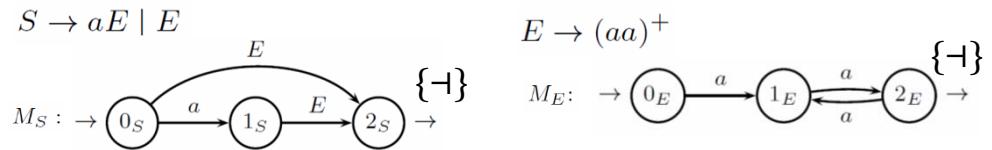
the parser, while scanning the string, «doesn't know», and investigates both hypotheses



In entrambe le ipotesi siamo quindi in M_E , ma non possiamo sapere se siamo passati per 1_S o no, possiamo scoprirlo solo alla fine
in base a quale delle due ipotesi ci porterà in uno stato finale al termine della stringa. A quel punto seguiamo i puntatori per conoscere le giuste riduzioni.

it is not always so easy to choose the correct reduction: it may be necessary to advance further in the string...

Example: another grammar G_2



even-length strings $(aa)^+ = L(0_E)$ and odd-length $a(aa)^+ = a \cdot L(1_S)$ derived very differently

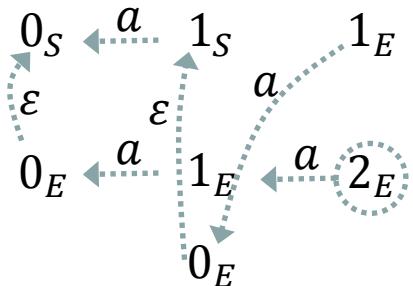
derivation becomes clear only after complete scan, when the terminator \vdash is encountered

Example: analysis of string $aaa\vdash$

computation that generates it: $0_S \xrightarrow{a} 1_S \xrightarrow{\varepsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$ NB: state 2_E is final

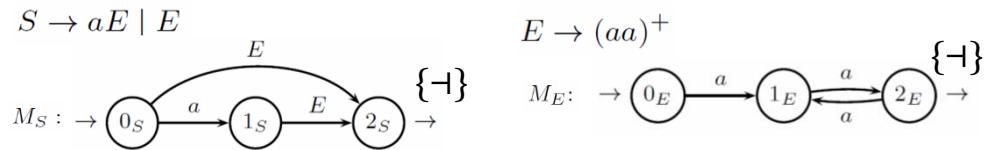
But there is another computation, which generates $aaaa\vdash$: $0_S \xrightarrow{\varepsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$

the parser, while scanning the string, «doesn't know», and investigates both hypotheses



it is not always so easy to choose the correct reduction: it may be necessary to advance further in the string...

Example: another grammar G_2



even-length strings $(aa)^+ = L(0_E)$ and odd-length $a(aa)^+ = a \cdot L(1_S)$ derived very differently

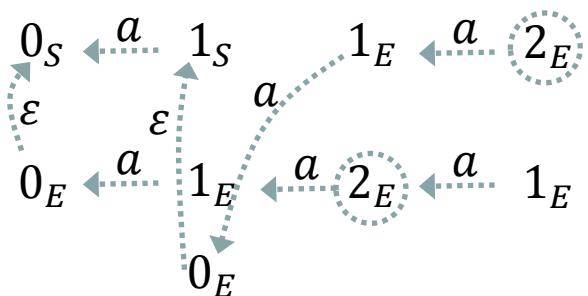
derivation becomes clear only after complete scan, when the terminator \dashv is encountered

Example: analysis of string $aaa\dashv$

computation that generates it: $0_S \xrightarrow{a} 1_S \xrightarrow{\varepsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$ NB: state 2_E is final

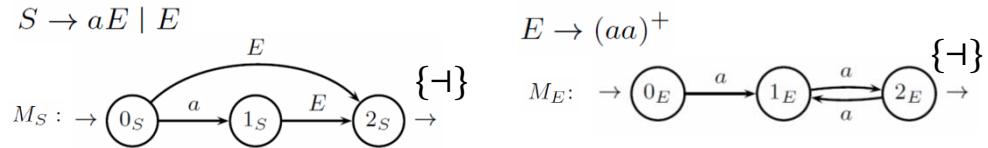
But there is another computation, which generates $aaaa\dashv$: $0_S \xrightarrow{\varepsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$

the parser, while scanning the string, «doesn't know», and investigates both hypotheses



it is not always so easy to choose the correct reduction: it may be necessary to advance further in the string...

Example: another grammar G_2



even-length strings $(aa)^+ = L(0_E)$ and odd-length $a(aa)^+ = a \cdot L(1_S)$ derived very differently

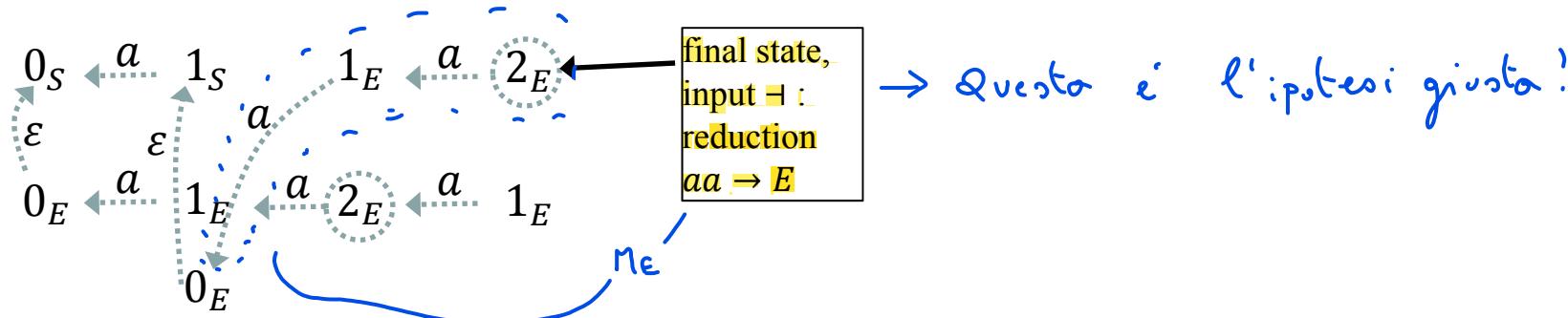
derivation becomes clear only after complete scan, when the terminator \dashv is encountered

Example: analysis of string $aaa\dashv$

computation that generates it: $0_S \xrightarrow{a} 1_S \xrightarrow{\epsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$ NB: state 2_E is final

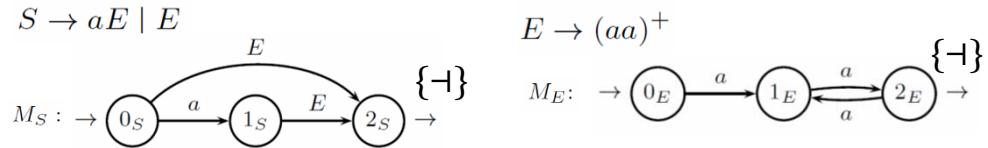
But there is another computation, which generates $aaaa\dashv$: $0_S \xrightarrow{\epsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$

the parser, while scanning the string, «doesn't know», and investigates both hypotheses



it is not always so easy to choose the correct reduction: it may be necessary to advance further in the string...

Example: another grammar G_2



even-length strings $(aa)^+ = L(0_E)$ and odd-length $a(aa)^+ = a \cdot L(1_S)$ derived very differently

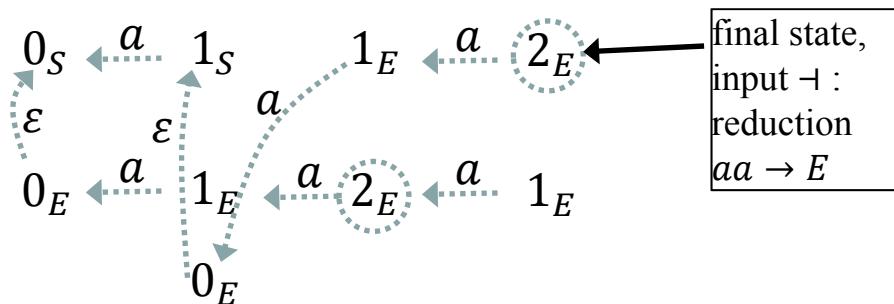
derivation becomes clear only after complete scan, when the terminator \dashv is encountered

Example: analysis of string $aaa\dashv$

computation that generates it: $0_S \xrightarrow{a} 1_S \xrightarrow{\epsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$ NB: state 2_E is final

But there is another computation, which generates $aaaa\dashv$: $0_S \xrightarrow{\epsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$

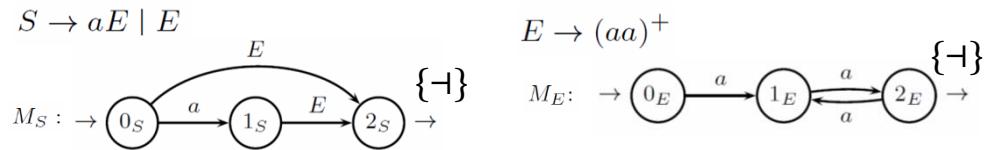
the parser, while scanning the string, «doesn't know», and investigates both hypotheses



NB: on the stack there are pointers, but they have a bounded value, hence the stack alphabet is finite, we are still using the PDA model

it is not always so easy to choose the correct reduction: it may be necessary to advance further in the string...

Example: another grammar G_2



even-length strings $(aa)^+ = L(0_E)$ and odd-length $a(aa)^+ = a \cdot L(1_S)$ derived very differently

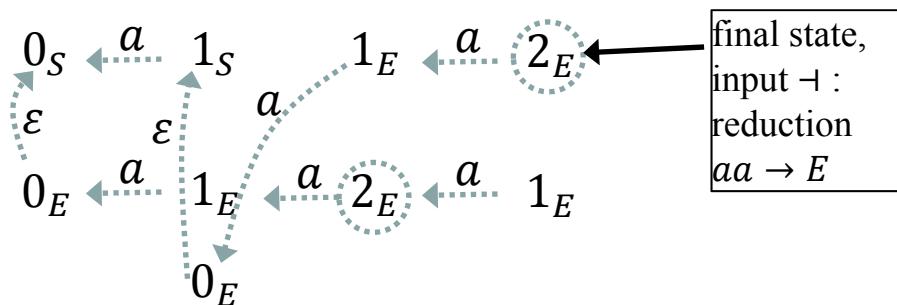
derivation becomes clear only after complete scan, when the terminator \dashv is encountered

Example: analysis of string $aaa\dashv$

computation that generates it: $0_S \xrightarrow{a} 1_S \xrightarrow{\epsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$ NB: state 2_E is final

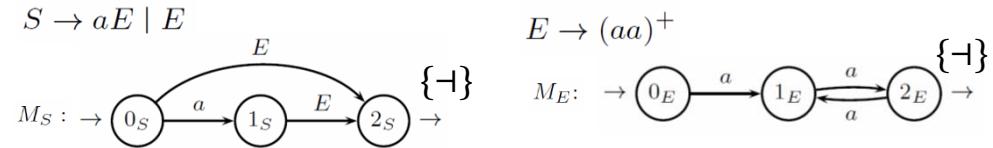
But there is another computation, which generates $aaaa\dashv$: $0_S \xrightarrow{\epsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$

the parser, while scanning the string, «doesn't know», and investigates both hypotheses



it is not always so easy to choose the correct reduction: it may be necessary to advance further in the string...

Example: another grammar G_2



even-length strings $(aa)^+ = L(0_E)$ and odd-length $a(aa)^+ = a \cdot L(1_S)$ derived very differently

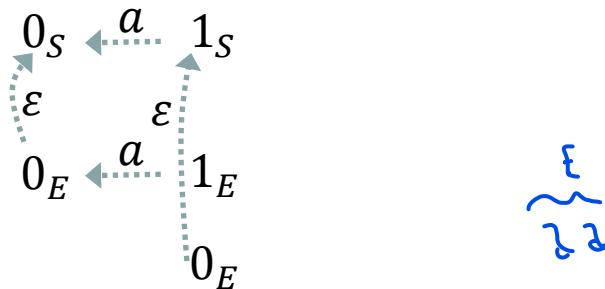
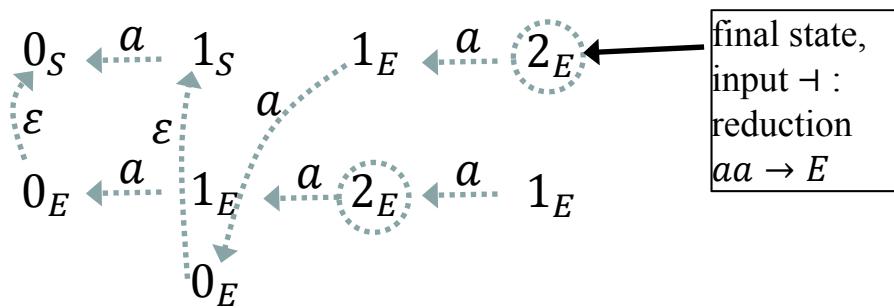
derivation becomes clear only after complete scan, when the terminator \vdash is encountered

Example: analysis of string $aaa\vdash$

computation that generates it: $0_S \xrightarrow{a} 1_S \xrightarrow{\varepsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$ NB: state 2_E is final

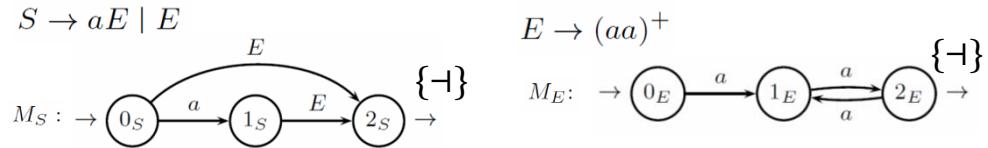
But there is another computation, which generates $aaaa\vdash$: $0_S \xrightarrow{\varepsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$

the parser, while scanning the string, «doesn't know», and investigates both hypotheses



it is not always so easy to choose the correct reduction: it may be necessary to advance further in the string...

Example: another grammar G_2



even-length strings $(aa)^+ = L(0_E)$ and odd-length $a(aa)^+ = a \cdot L(1_S)$ derived very differently

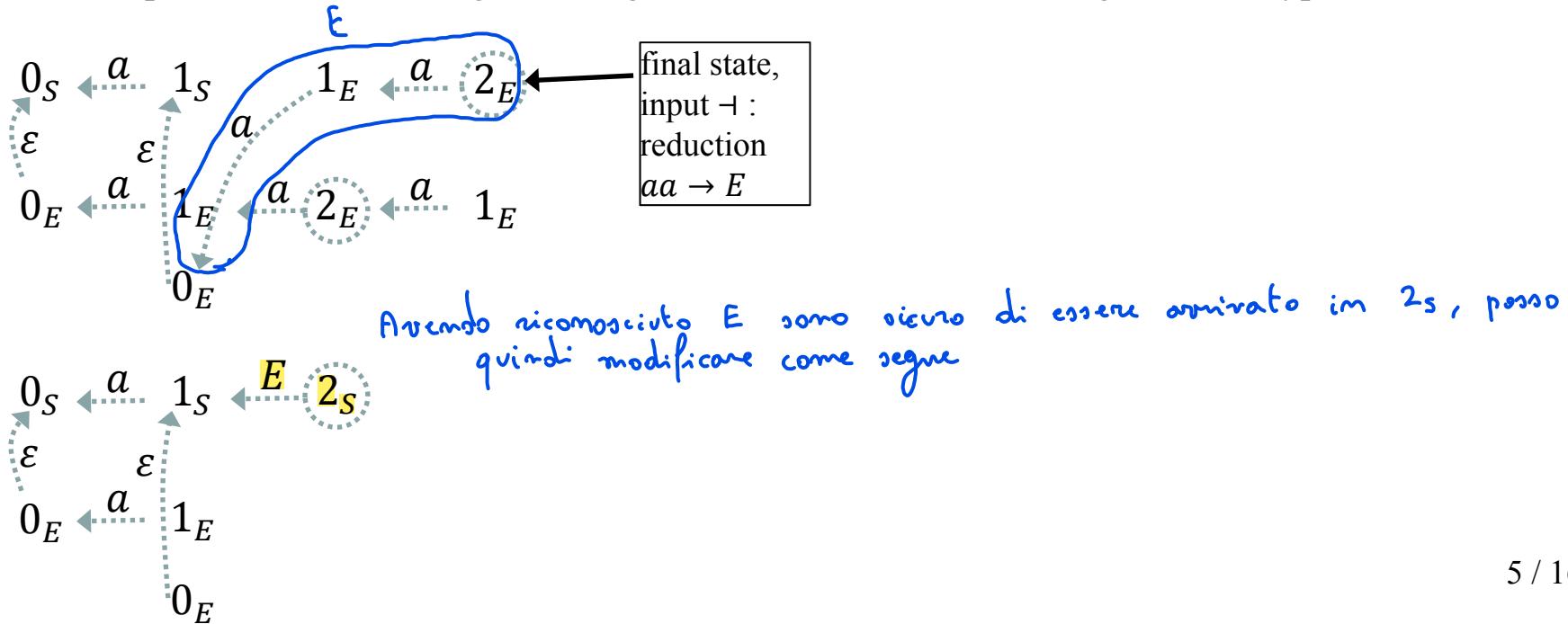
derivation becomes clear only after complete scan, when the terminator \vdash is encountered

Example: analysis of string $aaa\vdash$

computation that generates it: $0_S \xrightarrow{a} 1_S \xrightarrow{\varepsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$ NB: state 2_E is final

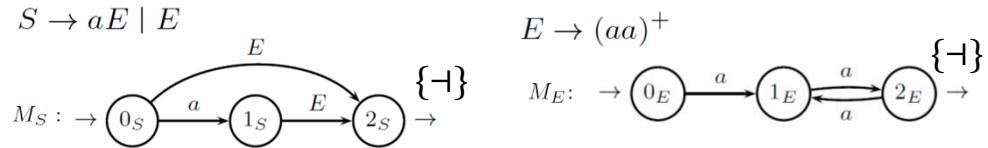
But there is another computation, which generates $aaaa\vdash$: $0_S \xrightarrow{\varepsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$

the parser, while scanning the string, «doesn't know», and investigates both hypotheses



it is not always so easy to choose the correct reduction: it may be necessary to advance further in the string...

Example: another grammar G_2



even-length strings $(aa)^+ = L(0_E)$ and odd-length $a(aa)^+ = a \cdot L(1_S)$ derived very differently

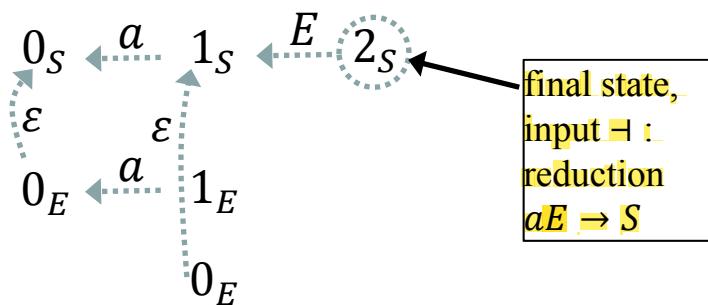
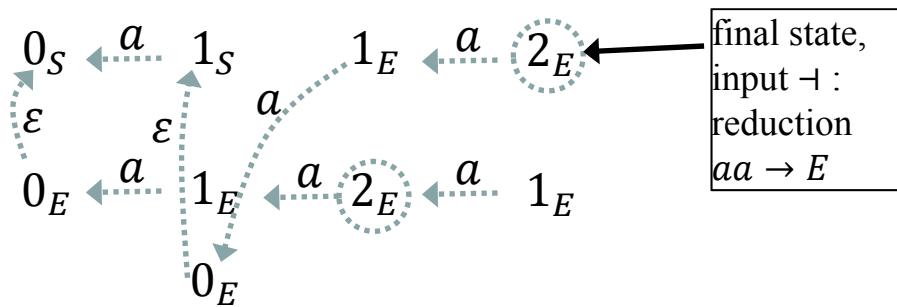
derivation becomes clear only after complete scan, when the terminator \vdash is encountered

Example: analysis of string $aaa\vdash$

computation that generates it: $0_S \xrightarrow{a} 1_S \xrightarrow{\varepsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$ NB: state 2_E is final

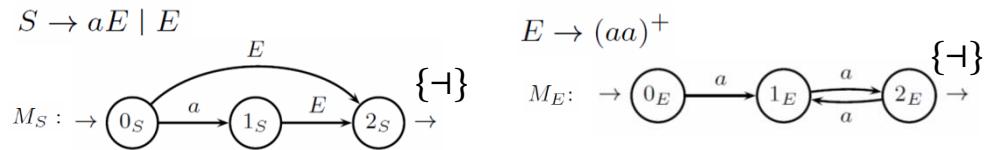
But there is another computation, which generates $aaaa\vdash$: $0_S \xrightarrow{\varepsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$

the parser, while scanning the string, «doesn't know», and investigates both hypotheses



it is not always so easy to choose the correct reduction: it may be necessary to advance further in the string...

Example: another grammar G_2



even-length strings $(aa)^+ = L(0_E)$ and odd-length $a(aa)^+ = a \cdot L(1_S)$ derived very differently

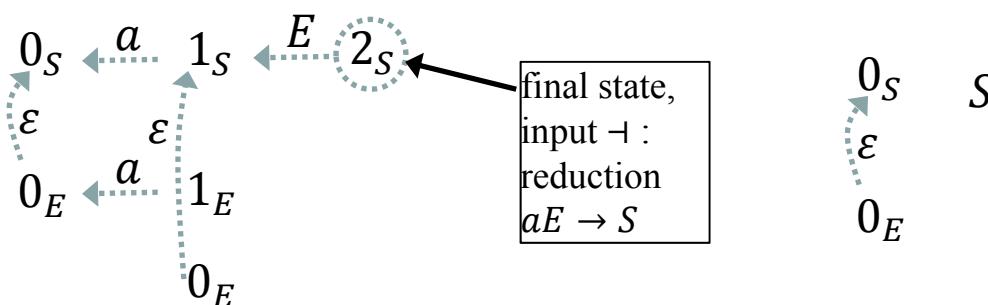
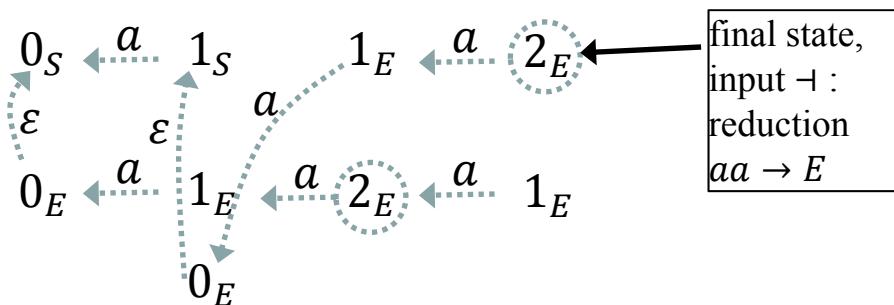
derivation becomes clear only after complete scan, when the terminator \dashv is encountered

Example: analysis of string $aaa\dashv$

computation that generates it: $0_S \xrightarrow{a} 1_S \xrightarrow{\epsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$ NB: state 2_E is final

But there is another computation, which generates $aaaa\dashv$: $0_S \xrightarrow{\epsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$

the parser, while scanning the string, «doesn't know», and investigates both hypotheses

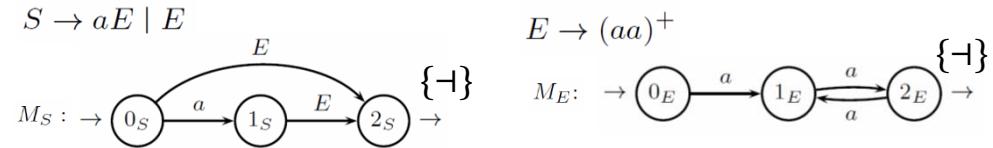


0_S
 ϵ
 0_E

S

it is not always so easy to choose the correct reduction: it may be necessary to advance further in the string...

Example: another grammar G_2



even-length strings $(aa)^+ = L(0_E)$ and odd-length $a(aa)^+ = a \cdot L(1_S)$ derived very differently

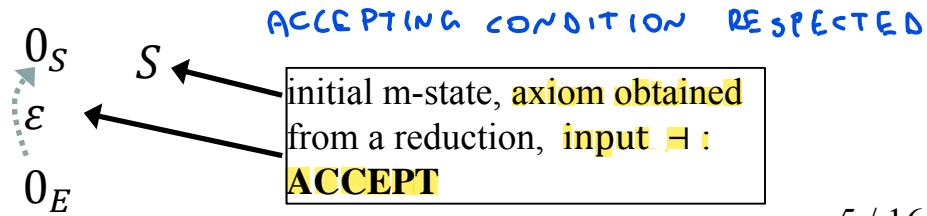
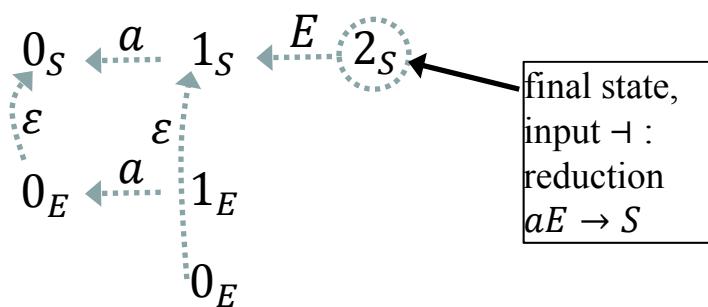
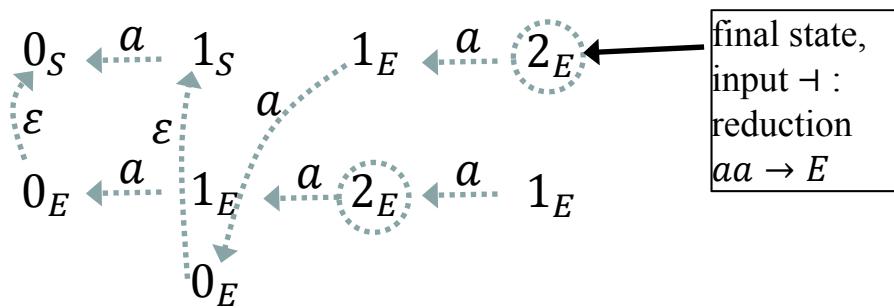
derivation becomes clear only after complete scan, when the terminator \dashv is encountered

Example: analysis of string $aaa\dashv$

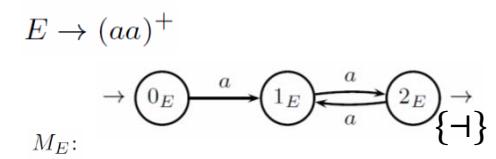
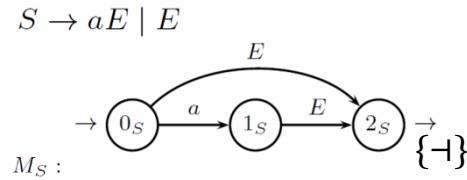
computation that generates it: $0_S \xrightarrow{a} 1_S \xrightarrow{\epsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$ NB: state 2_E is final

But there is another computation, which generates $aaaa\dashv$: $0_S \xrightarrow{\epsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$

the parser, while scanning the string, «doesn't know», and investigates both hypotheses

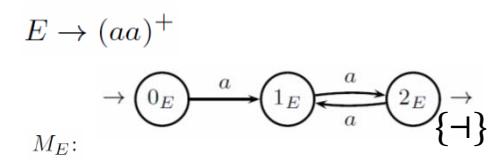
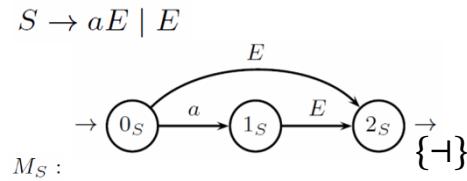


another example: analysis of string $aaaa\vdash$



Computation that generates it: $\mathbf{0}_S \xrightarrow{\varepsilon} \mathbf{0}_E \xrightarrow{a} \mathbf{1}_E \xrightarrow{a} \mathbf{2}_E \xrightarrow{a} \mathbf{1}_E \xrightarrow{a} \mathbf{2}_E$

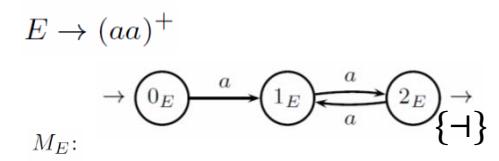
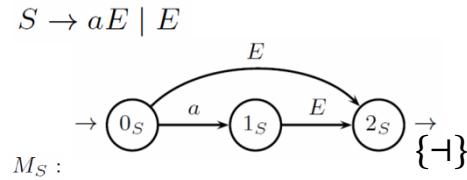
another example: analysis of string $aaaa\vdash$



Computation that generates it: $0_S \xrightarrow{\epsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$

0_S

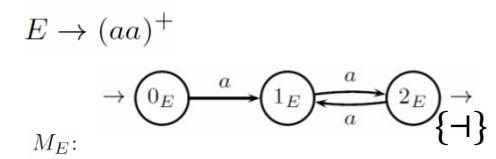
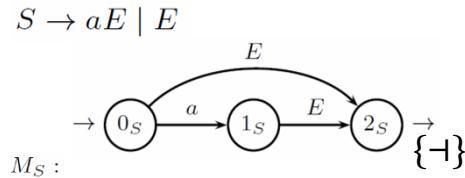
another example: analysis of string $aaaa\vdash$



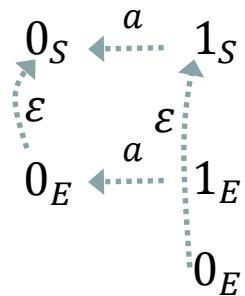
Computation that generates it: $0_S \xrightarrow{\varepsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$

0_S
 $\xrightarrow{\varepsilon}$
 0_E

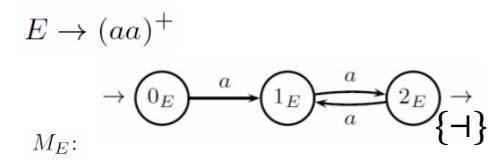
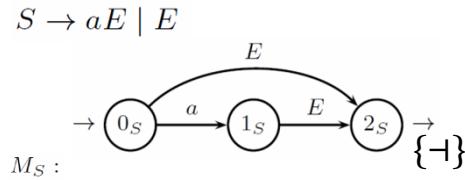
another example: analysis of string $aaaa\vdash$



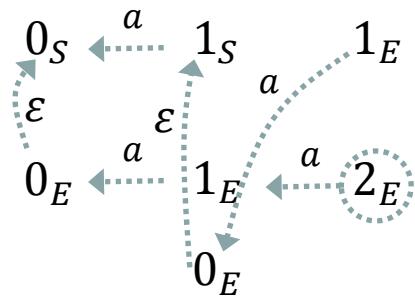
Computation that generates it: $0_S \xrightarrow{\varepsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$



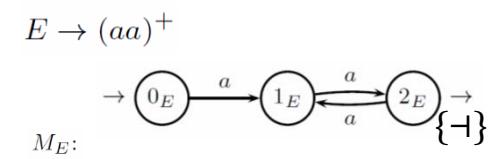
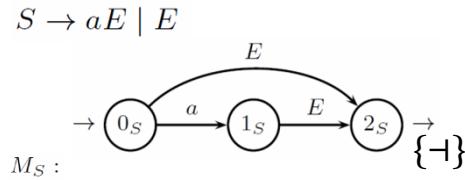
another example: analysis of string $aaaa\vdash$



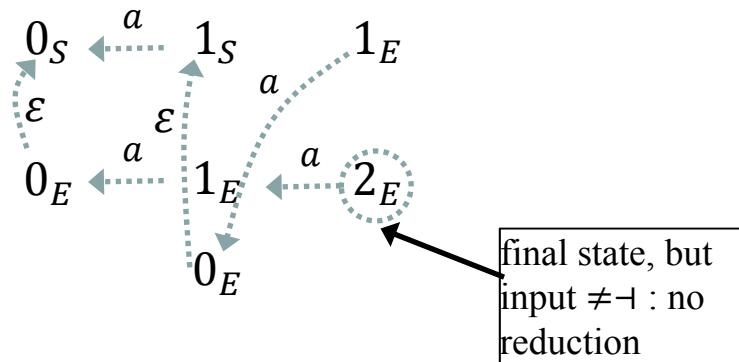
Computation that generates it: $0_S \xrightarrow{\varepsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$



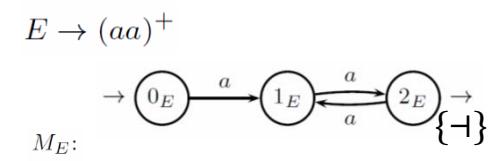
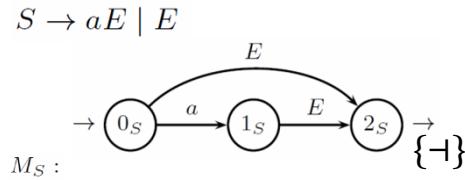
another example: analysis of string $aaaa\vdash$



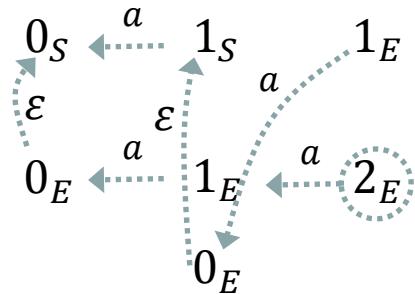
Computation that generates it: $0_S \xrightarrow{\varepsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$



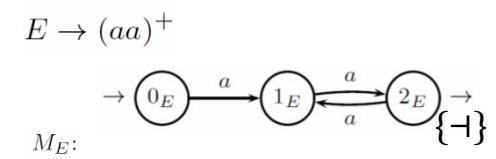
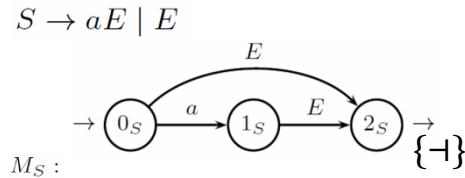
another example: analysis of string $aaaa\vdash$



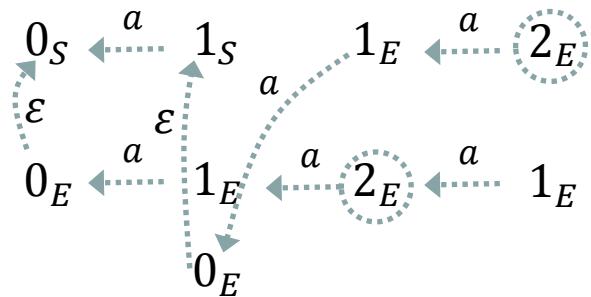
Computation that generates it: $0_S \xrightarrow{\epsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$



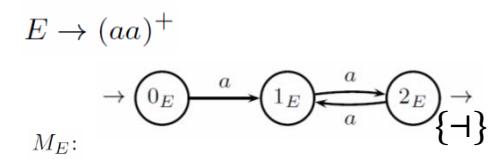
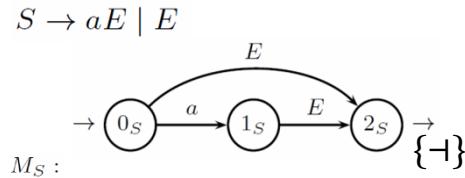
another example: analysis of string $aaaa\vdash$



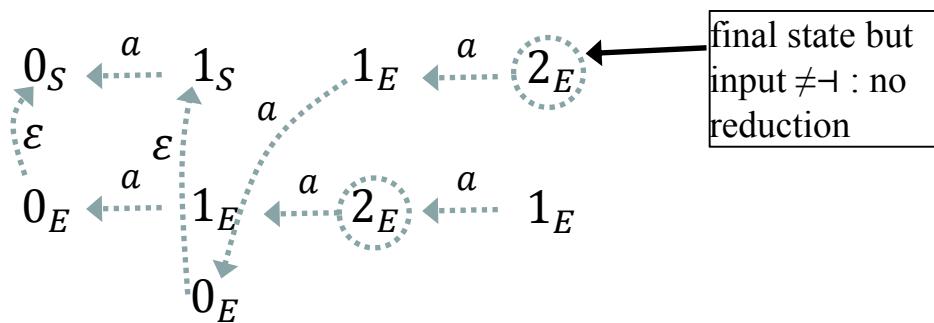
Computation that generates it: $0_S \xrightarrow{\varepsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$



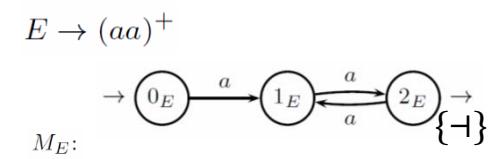
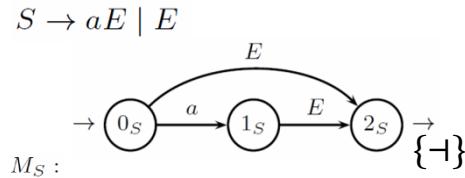
another example: analysis of string $aaaa\vdash$



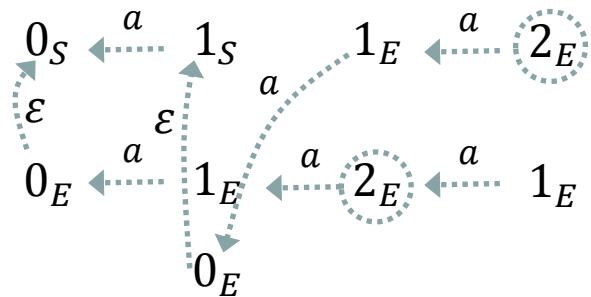
Computation that generates it: $0_S \xrightarrow{\varepsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$



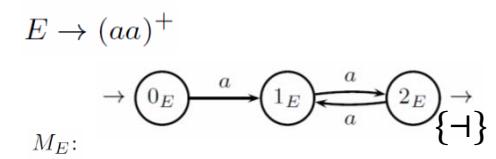
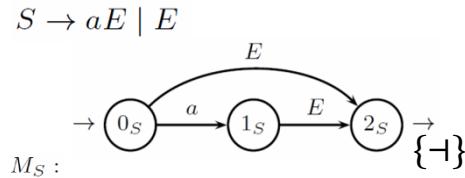
another example: analysis of string $aaaa\vdash$



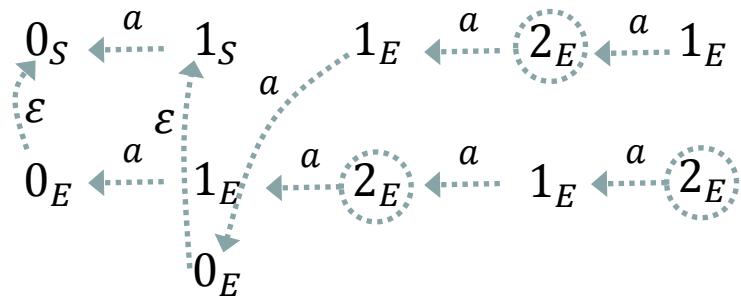
Computation that generates it: $0_S \xrightarrow{\varepsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$



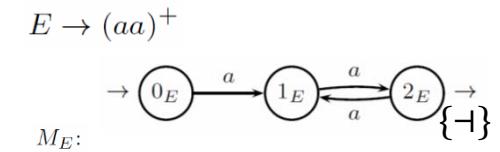
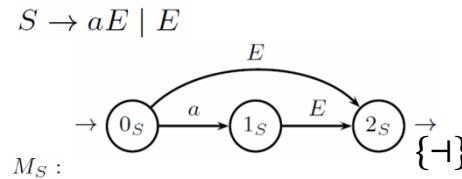
another example: analysis of string $aaaa\vdash$



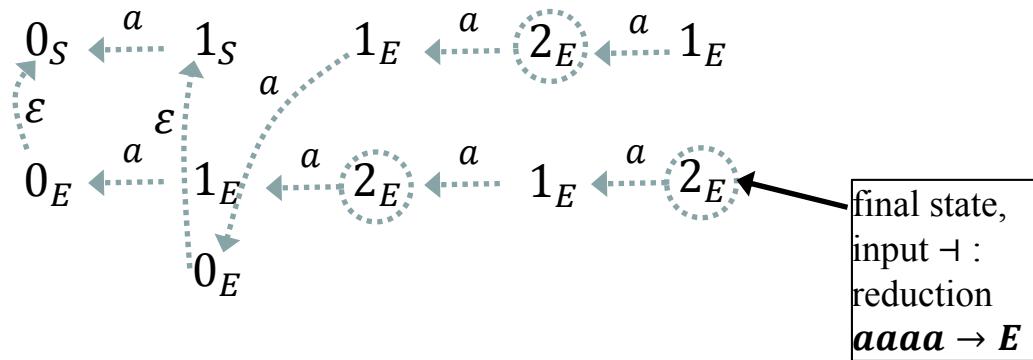
Computation that generates it: $0_S \xrightarrow{\varepsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$



another example: analysis of string $aaaa\vdash$

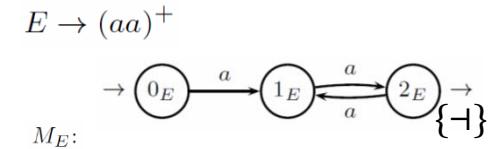
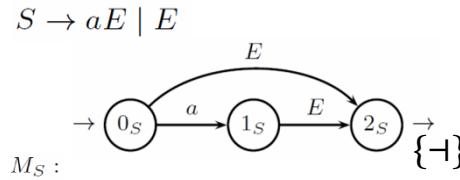


Computation that generates it: $0_S \xrightarrow{\epsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$

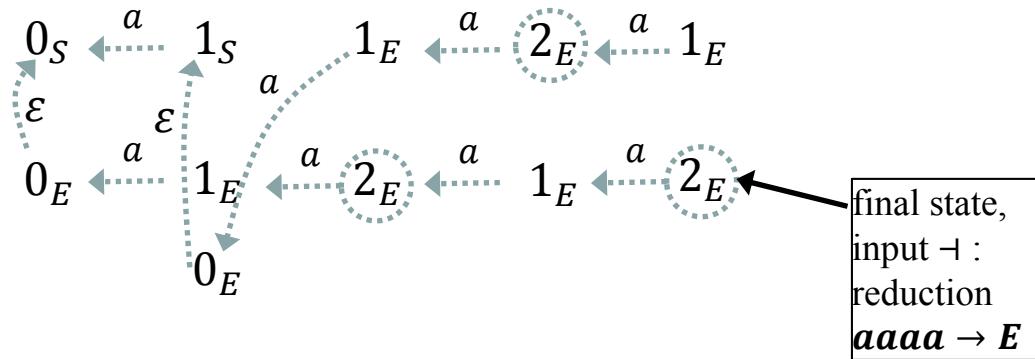


It's the final state of machine E so we need to reduce to it but we need to follow pointers backwards to understand which one among the different initial states 0_E

another example: analysis of string $aaaa\vdash$

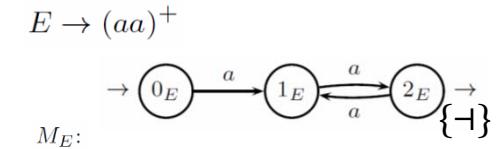
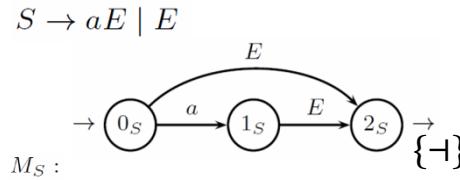


Computation that generates it: $0_S \xrightarrow{\epsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$

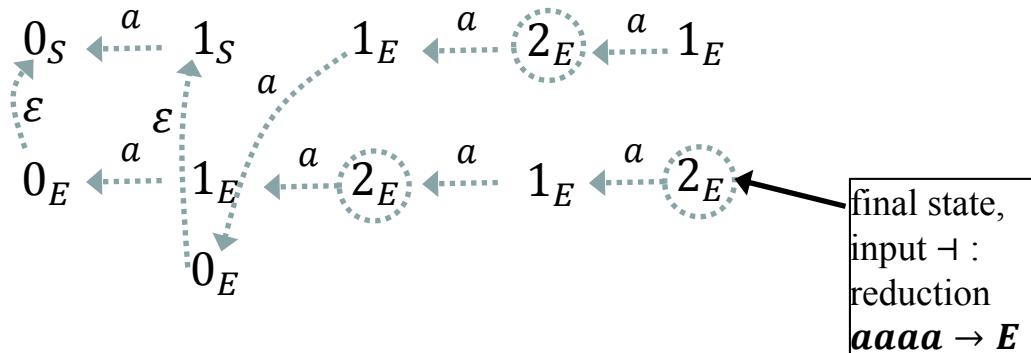


How to determine the handle length? Follow the pointers up to the initial state 0_E

another example: analysis of string $aaaa\vdash$



Computation that generates it: $0_S \xrightarrow{\epsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$

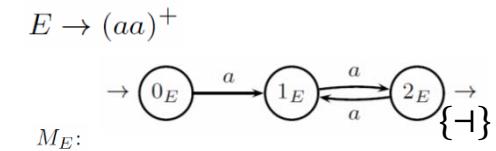
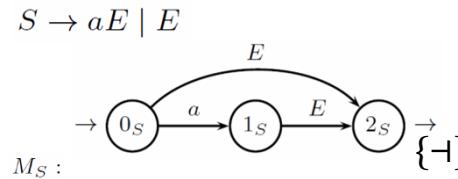


How to determine the handle length? Follow the pointers up to the initial state 0_E

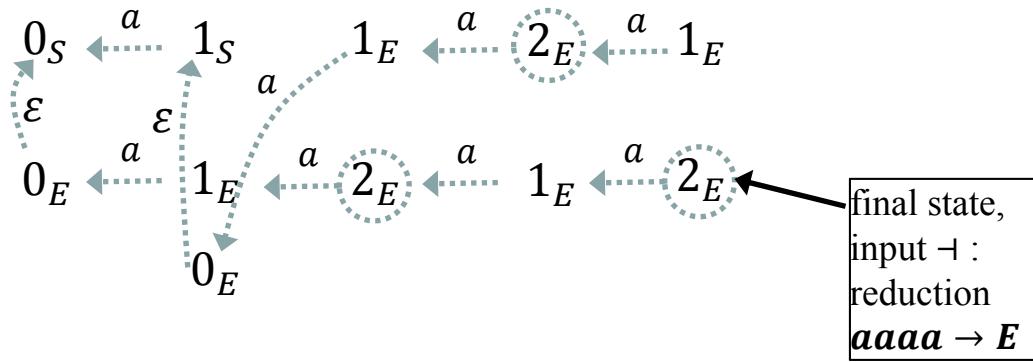
That's why the initial state 0_X of machine M_X must be non-recirculating: it is used as a mark for the handle starting point

).

another example: analysis of string $aaaa\vdash$



Computation that generates it: $0_S \xrightarrow{\varepsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$



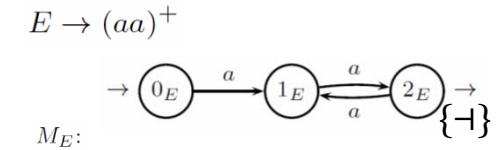
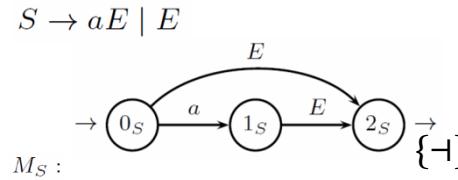
How to determine the handle length? Follow the pointers up to the initial state 0_E

That's why the **initial state 0_X** of machine M_X must be **non-recirculating**: it is used as a mark for the handle starting point

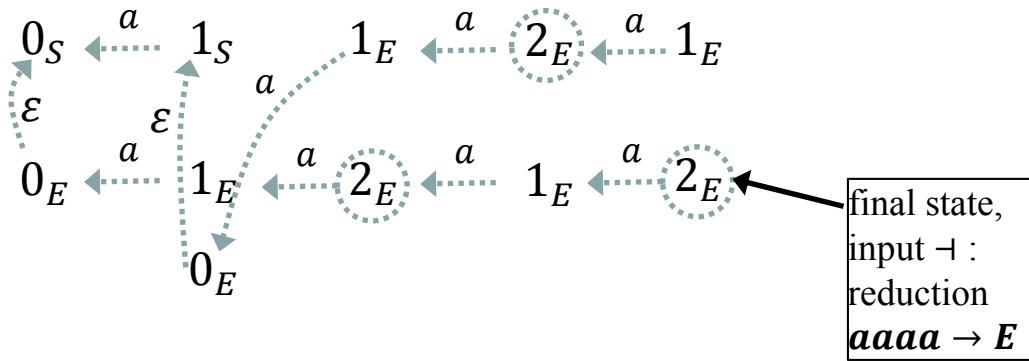


Everytime we perform a reduction
we need to perform a non terminal shift

another example: analysis of string $aaaa\vdash$

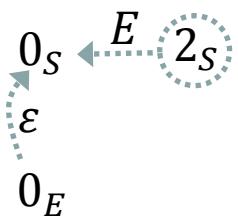


Computation that generates it: $0_S \xrightarrow{\epsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$

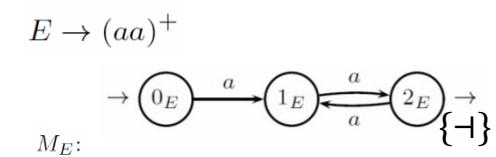
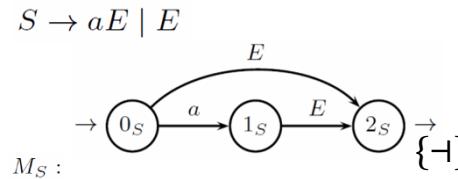


How to determine the handle length? Follow the pointers up to the initial state 0_E

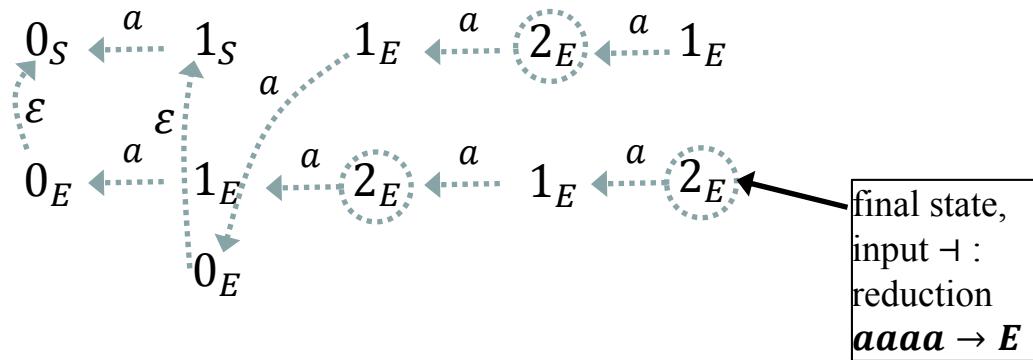
That's why the **initial state 0_X** of machine M_X must be **non-recirculating**: it is used as a mark for the handle starting point



another example: analysis of string $aaaa\vdash$

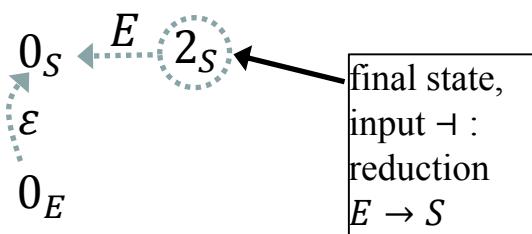


Computation that generates it: $0_S \xrightarrow{\varepsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$

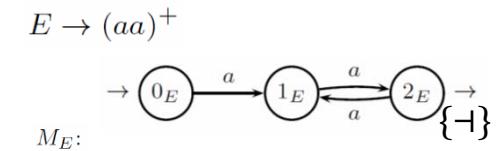
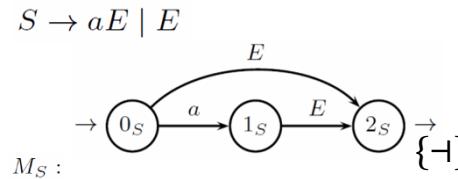


How to determine the handle length? Follow the pointers up to the initial state 0_E

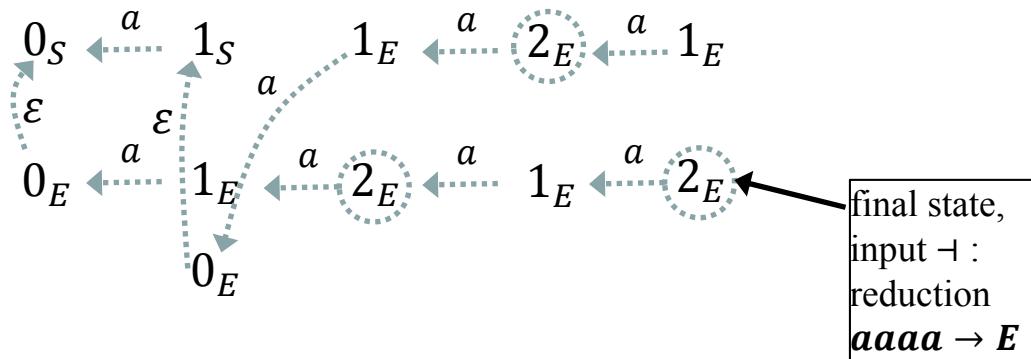
That's why the **initial state 0_X** of machine M_X must be **non-recirculating**: it is used as a mark for the handle starting point



another example: analysis of string $aaaa\vdash$

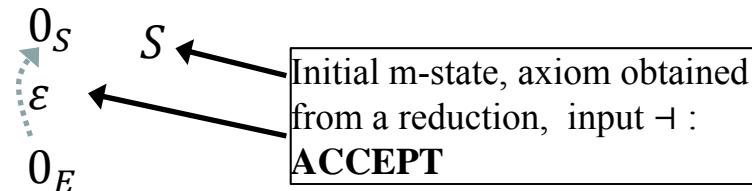
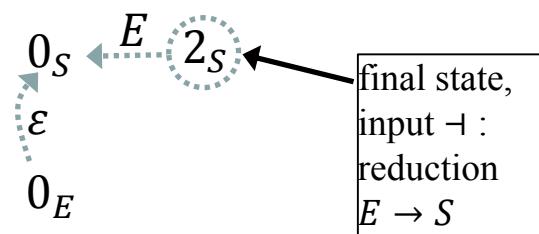


Computation that generates it: $0_S \xrightarrow{\varepsilon} 0_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E \xrightarrow{a} 1_E \xrightarrow{a} 2_E$



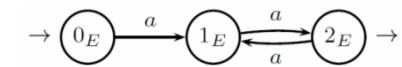
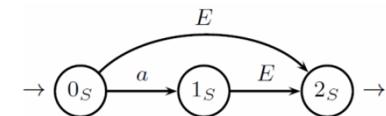
How to determine the handle length? Follow the pointers up to the initial state 0_E

That's why the **initial state 0_X** of machine M_X must be **non-recirculating**: it is used as a mark for the handle starting point



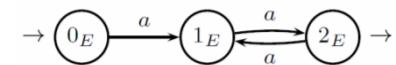
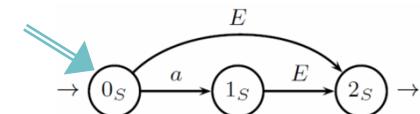
- construction of the **pilot automaton**, which
 - Guides the analysis of the stack automaton
 - Incorporates in its states (macro-states, m-states) info on all possible computations, which, scanning a sentential form, reach that m-state
- if necessary, the m-states contain many (machine)states corresponding to various conjectures on the derivation (analysis threads) investigated in parallel
- in the stack of the parser the states of the *analysis threads* are linked using pointers
- the pilot automaton is built statically (in advance *once forall*) and used to guide all possible analyses

- construction of the *pilot automaton*, which
 - Guides the analysis of the stack automaton
 - Incorporates in its states (macro-states, m-states) info on all possible computations, which, scanning a sentential form, reach that m-state
- if necessary, the m-states contain many (machine)states corresponding to various conjectures on the derivation (analysis threads) investigated in parallel
- *in the stack* of the parser the states of the *analysis threads* are linked using pointers
- the pilot automaton is built statically (in advance *once forall*) and used to guide all possible analyses



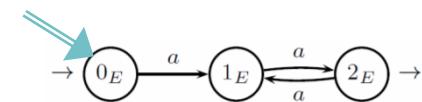
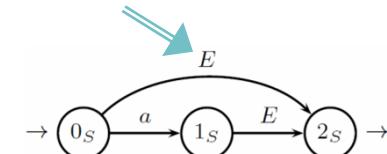
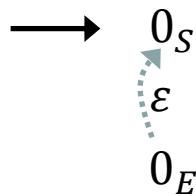
- construction of the *pilot automaton*, which
 - Guides the analysis of the stack automaton
 - Incorporates in its states (macro-states, m-states) info on all possible computations, which, scanning a sentential form, reach that m-state
- if necessary, the m-states contain many (machine)states corresponding to various conjectures on the derivation (analysis threads) investigated in parallel
- *in the stack* of the parser the states of the *analysis threads* are linked using pointers
- the pilot automaton is built statically (in advance *once forall*) and used to guide all possible analyses

→ 0_S

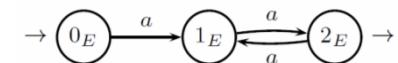
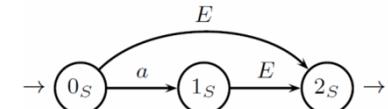
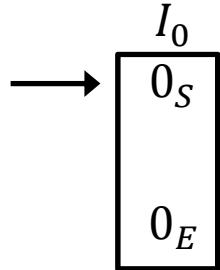


- construction of the *pilot automaton*, which
 - Guides the analysis of the stack automaton
 - Incorporates in its states (macro-states, m-states) info on all possible computations, which, scanning a sentential form, reach that m-state
- if necessary, the m-states contain many (machine)states corresponding to various conjectures on the derivation (analysis threads) investigated in parallel
- *in the stack* of the parser the states of the *analysis threads* are linked using pointers
- the pilot automaton is built statically (in advance *once forall*) and used to guide all possible analyses

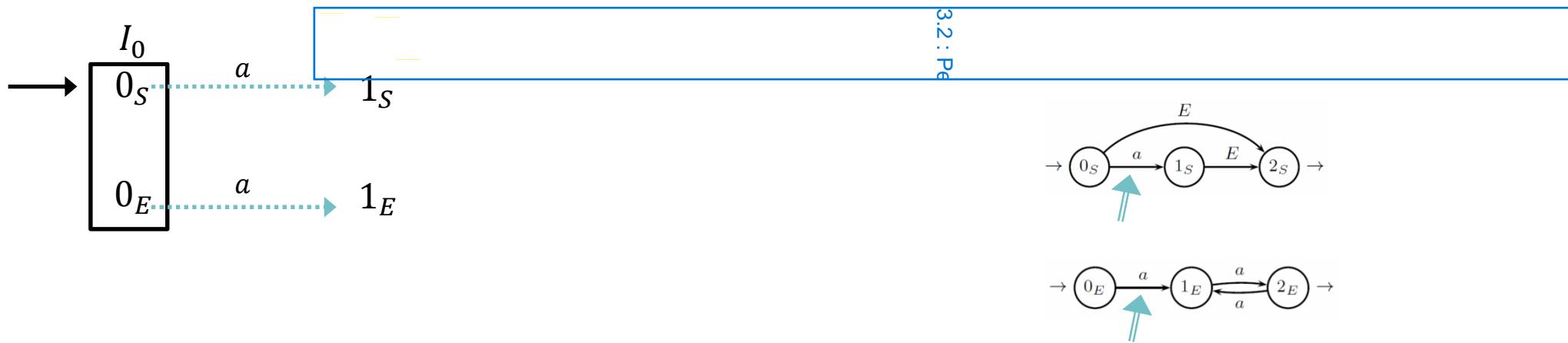
Ho fatto la chiusura di 0_S : verifico che ci siano frecce uscenti etichettate da caratteri non terminali e in caso positivo inserisco lo stato iniziale della macchina associata a tale terminale. In questo caso E



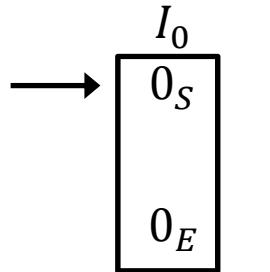
- construction of the *pilot automaton*, which
 - Guides the analysis of the stack automaton
 - Incorporates in its states (macro-states, m-states) info on all possible computations, which, scanning a sentential form, reach that m-state
- if necessary, the m-states contain many (machine)states corresponding to various conjectures on the derivation (analysis threads) investigated in parallel
- *in the stack* of the parser the states of the *analysis threads* are linked using pointers
- the pilot automaton is built statically (in advance *once forall*) and used to guide all possible analyses



- construction of the *pilot automaton*, which
 - Guides the analysis of the stack automaton
 - Incorporates in its states (macro-states, m-states) info on all possible computations, which, scanning a sentential form, reach that m-state
- if necessary, the m-states contain many (machine)states corresponding to various conjectures on the derivation (analysis threads) investigated in parallel
- *in the stack* of the parser the states of the *analysis threads* are linked using pointers
- the pilot automaton is built statically (in advance *once forall*) and used to guide all possible analyses

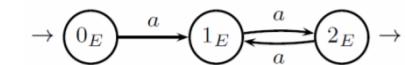
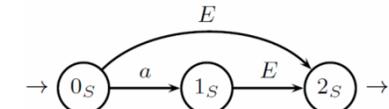


- construction of the *pilot automaton*, which
 - Guides the analysis of the stack automaton
 - Incorporates in its states (macro-states, m-states) info on all possible computations, which, scanning a sentential form, reach that m-state
- if necessary, the m-states contain many (machine)states corresponding to various conjectures on the derivation (analysis threads) investigated in parallel
- *in the stack* of the parser the states of the *analysis threads* are linked using pointers
- the pilot automaton is built statically (in advance *once forall*) and used to guide all possible analyses



1_S

1_E



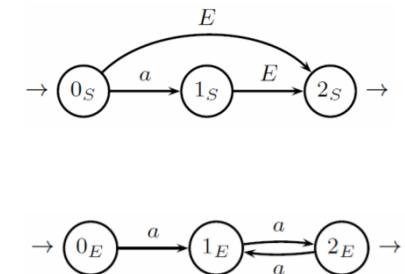
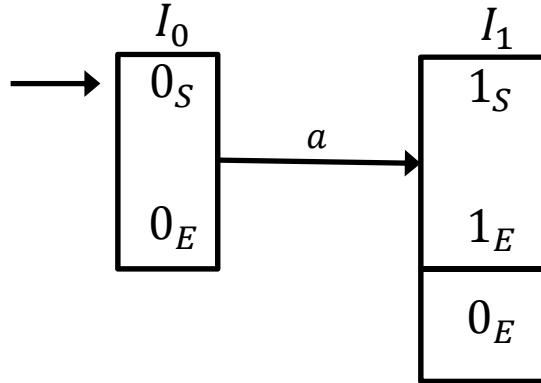
- construction of the *pilot automaton*, which
 - Guides the analysis of the stack automaton
 - Incorporates in its states (macro-states, m-states) info on all possible computations, which, scanning a sentential form, reach that m-state
- if necessary, the m-states contain many (machine)states corresponding to various conjectures on the derivation (analysis threads) investigated in parallel
- *in the stack* of the parser the states of the *analysis threads* are linked using pointers
- the pilot automaton is built statically (in advance *once forall*) and used to guide all possible analyses



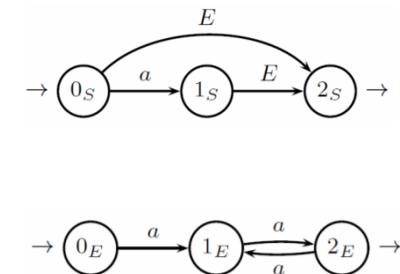
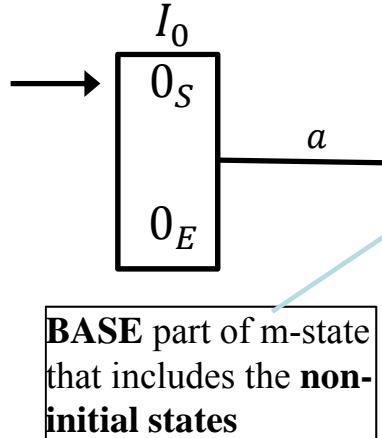
- construction of the *pilot automaton*, which
 - Guides the analysis of the stack automaton
 - Incorporates in its states (macro-states, m-states) info on all possible computations, which, scanning a sentential form, reach that m-state
- if necessary, the m-states contain many (machine)states corresponding to various conjectures on the derivation (analysis threads) investigated in parallel
- *in the stack* of the parser the states of the *analysis threads* are linked using pointers
- the pilot automaton is built statically (in advance *once forall*) and used to guide all possible analyses



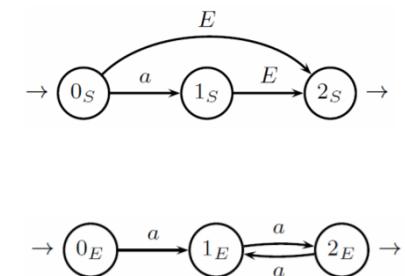
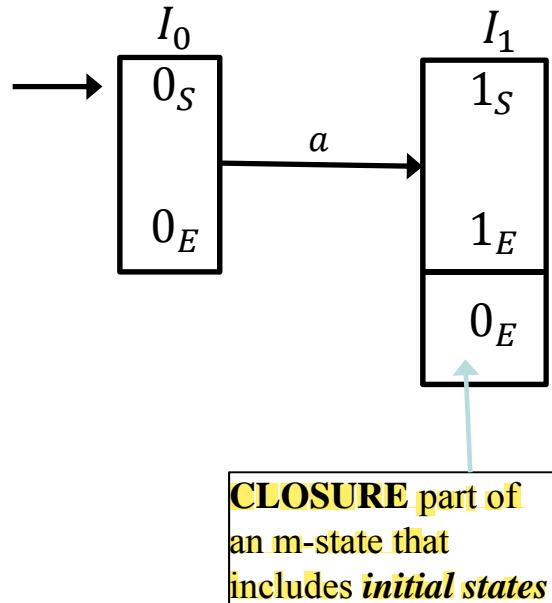
- construction of the *pilot automaton*, which
 - Guides the analysis of the stack automaton
 - Incorporates in its states (macro-states, m-states) info on all possible computations, which, scanning a sentential form, reach that m-state
- if necessary, the m-states contain many (machine)states corresponding to various conjectures on the derivation (analysis threads) investigated in parallel
- *in the stack* of the parser the states of the *analysis threads* are linked using pointers
- the pilot automaton is built statically (in advance *once forall*) and used to guide all possible analyses



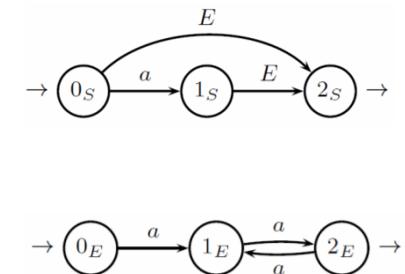
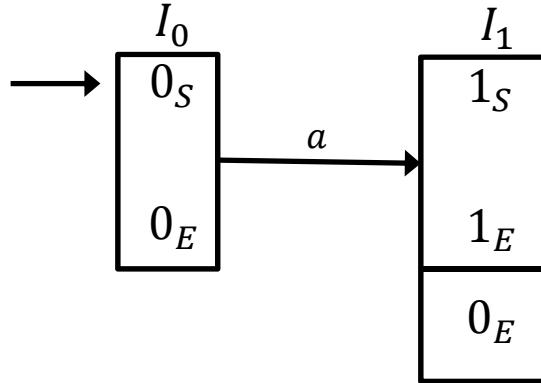
- construction of the *pilot automaton*, which
 - Guides the analysis of the stack automaton
 - Incorporates in its states (macro-states, m-states) info on all possible computations, which, scanning a sentential form, reach that m-state
- if necessary, the m-states contain many (machine)states corresponding to various conjectures on the derivation (analysis threads) investigated in parallel
- *in the stack* of the parser the states of the *analysis threads* are linked using pointers
- the pilot automaton is built statically (in advance *once forall*) and used to guide all possible analyses



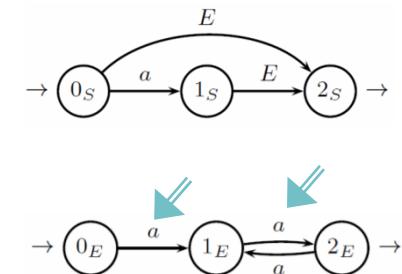
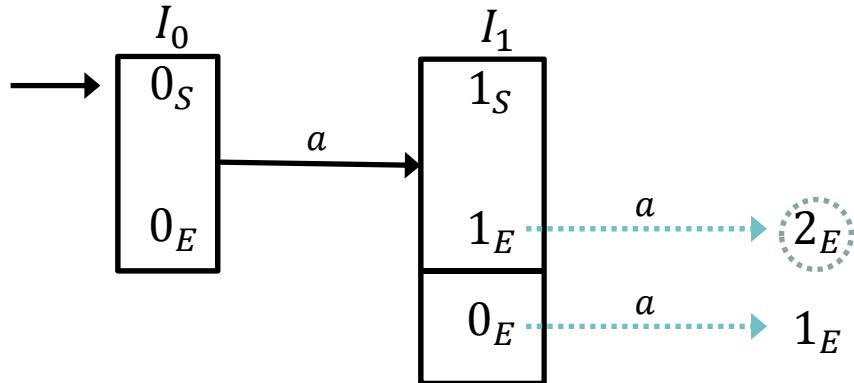
- construction of the *pilot automaton*, which
 - Guides the analysis of the stack automaton
 - Incorporates in its states (macro-states, m-states) info on all possible computations, which, scanning a sentential form, reach that m-state
- if necessary, the m-states contain many (machine)states corresponding to various conjectures on the derivation (analysis threads) investigated in parallel
- *in the stack* of the parser the states of the *analysis threads* are linked using pointers
- the pilot automaton is built statically (in advance *once forall*) and used to guide all possible analyses



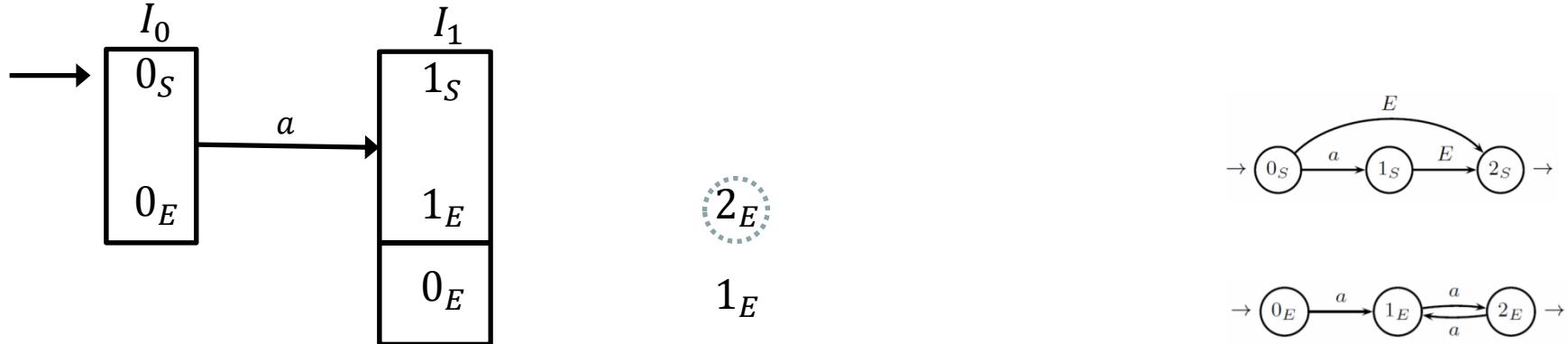
- construction of the *pilot automaton*, which
 - Guides the analysis of the stack automaton
 - Incorporates in its states (macro-states, m-states) info on all possible computations, which, scanning a sentential form, reach that m-state
- if necessary, the m-states contain many (machine)states corresponding to various conjectures on the derivation (analysis threads) investigated in parallel
- *in the stack* of the parser the states of the *analysis threads* are linked using pointers
- the pilot automaton is built statically (in advance *once forall*) and used to guide all possible analyses



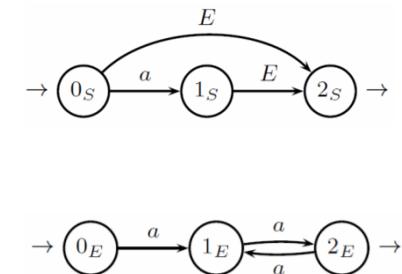
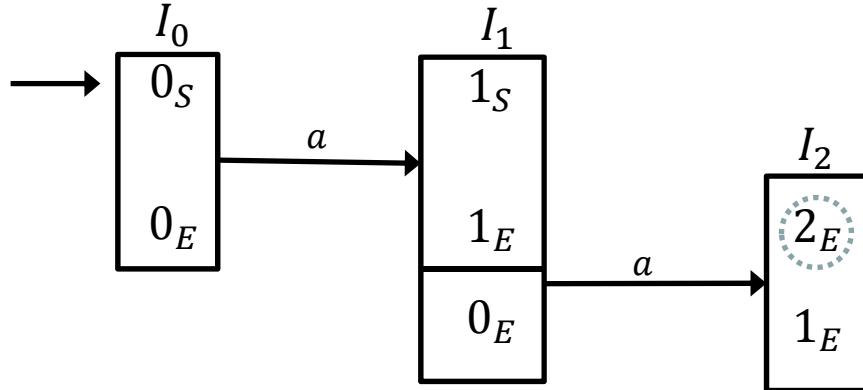
- construction of the *pilot automaton*, which
 - Guides the analysis of the stack automaton
 - Incorporates in its states (macro-states, m-states) info on all possible computations, which, scanning a sentential form, reach that m-state
- if necessary, the m-states contain many (machine)states corresponding to various conjectures on the derivation (analysis threads) investigated in parallel
- *in the stack* of the parser the states of the *analysis threads* are linked using pointers
- the pilot automaton is built statically (in advance *once forall*) and used to guide all possible analyses



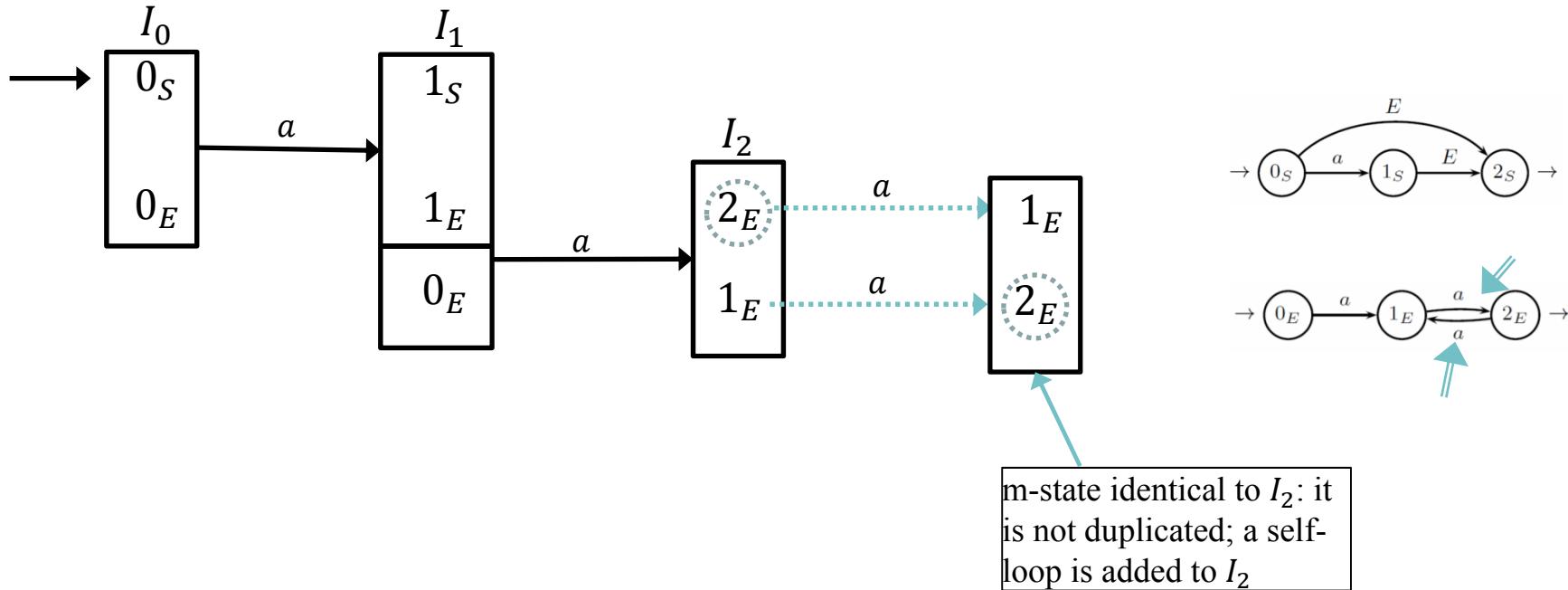
- construction of the *pilot automaton*, which
 - Guides the analysis of the stack automaton
 - Incorporates in its states (macro-states, m-states) info on all possible computations, which, scanning a sentential form, reach that m-state
- if necessary, the m-states contain many (machine)states corresponding to various conjectures on the derivation (analysis threads) investigated in parallel
- *in the stack* of the parser the states of the *analysis threads* are linked using pointers
- the pilot automaton is built statically (in advance *once forall*) and used to guide all possible analyses



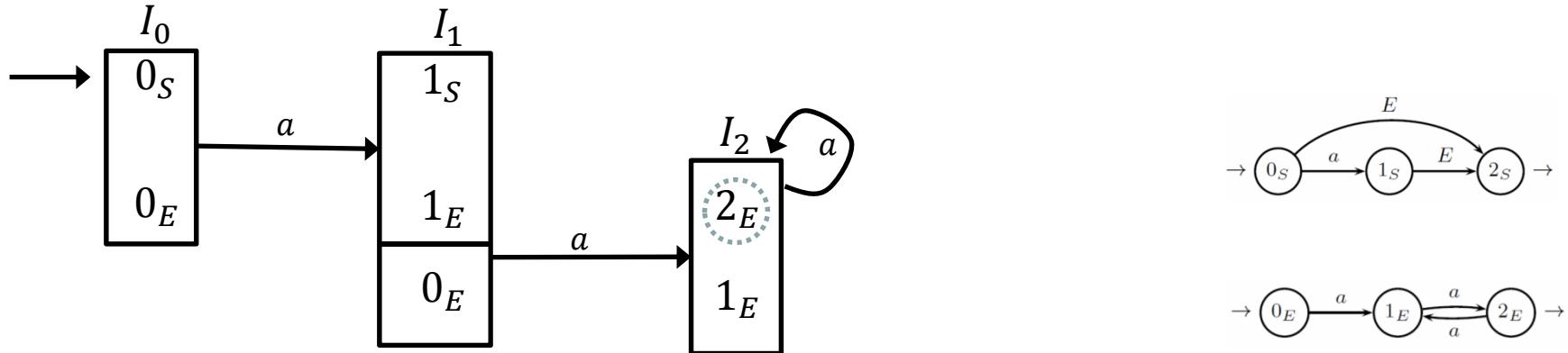
- construction of the *pilot automaton*, which
 - Guides the analysis of the stack automaton
 - Incorporates in its states (macro-states, m-states) info on all possible computations, which, scanning a sentential form, reach that m-state
- if necessary, the m-states contain many (machine)states corresponding to various conjectures on the derivation (analysis threads) investigated in parallel
- *in the stack* of the parser the states of the *analysis threads* are linked using pointers
- the pilot automaton is built statically (in advance *once forall*) and used to guide all possible analyses



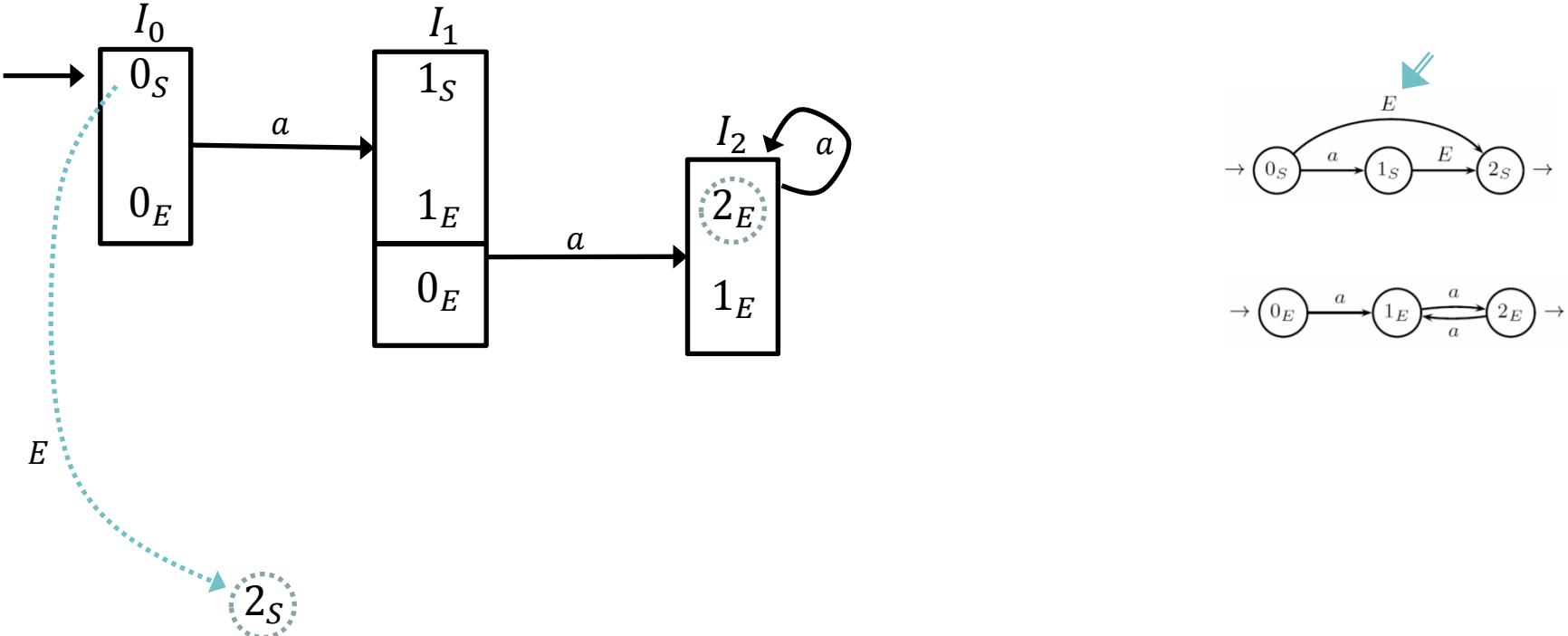
- construction of the *pilot automaton*, which
 - Guides the analysis of the stack automaton
 - Incorporates in its states (macro-states, m-states) info on all possible computations, which, scanning a sentential form, reach that m-state
- if necessary, the m-states contain many (machine)states corresponding to various conjectures on the derivation (analysis threads) investigated in parallel
- *in the stack* of the parser the states of the *analysis threads* are linked using pointers
- the pilot automaton is built statically (in advance *once forall*) and used to guide all possible analyses



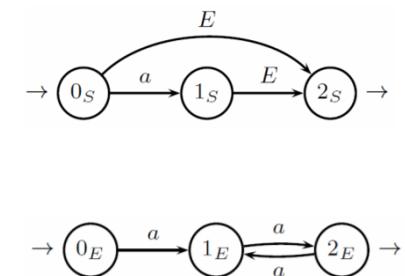
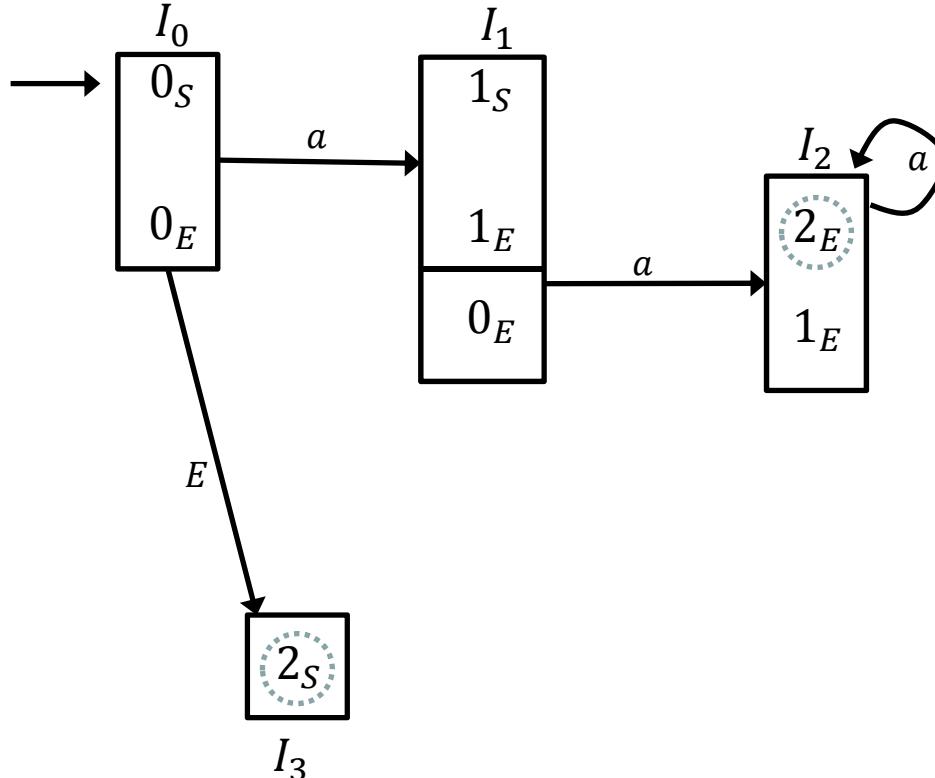
- construction of the *pilot automaton*, which
 - Guides the analysis of the stack automaton
 - Incorporates in its states (macro-states, m-states) info on all possible computations, which, scanning a sentential form, reach that m-state
- if necessary, the m-states contain many (machine)states corresponding to various conjectures on the derivation (analysis threads) investigated in parallel
- *in the stack* of the parser the states of the *analysis threads* are linked using pointers
- the pilot automaton is built statically (in advance *once forall*) and used to guide all possible analyses



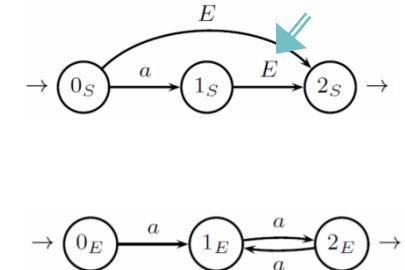
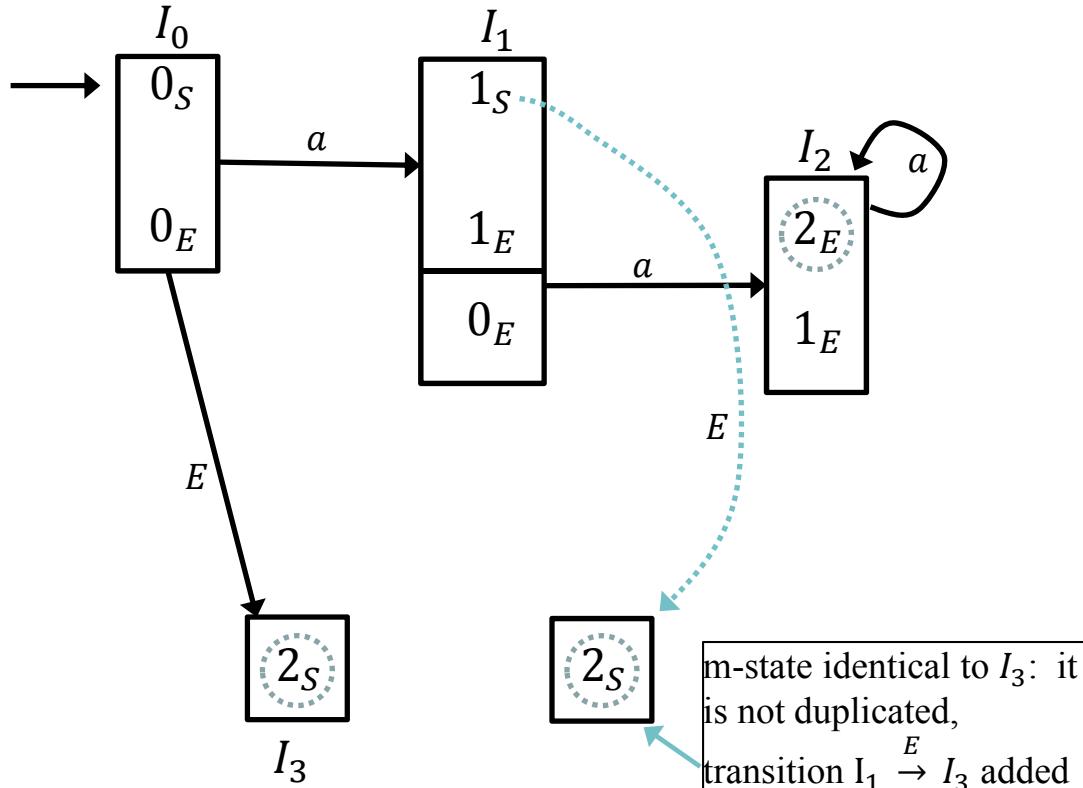
- construction of the *pilot automaton*, which
 - Guides the analysis of the stack automaton
 - Incorporates in its states (macro-states, m-states) info on all possible computations, which, scanning a sentential form, reach that m-state
- if necessary, the m-states contain many (machine)states corresponding to various conjectures on the derivation (analysis threads) investigated in parallel
- *in the stack* of the parser the states of the *analysis threads* are linked using pointers
- the pilot automaton is built statically (in advance *once forall*) and used to guide all possible analyses



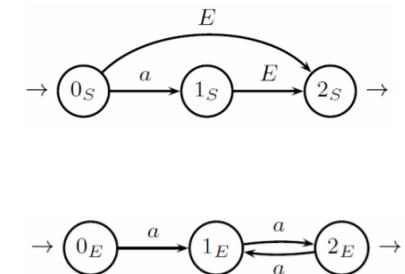
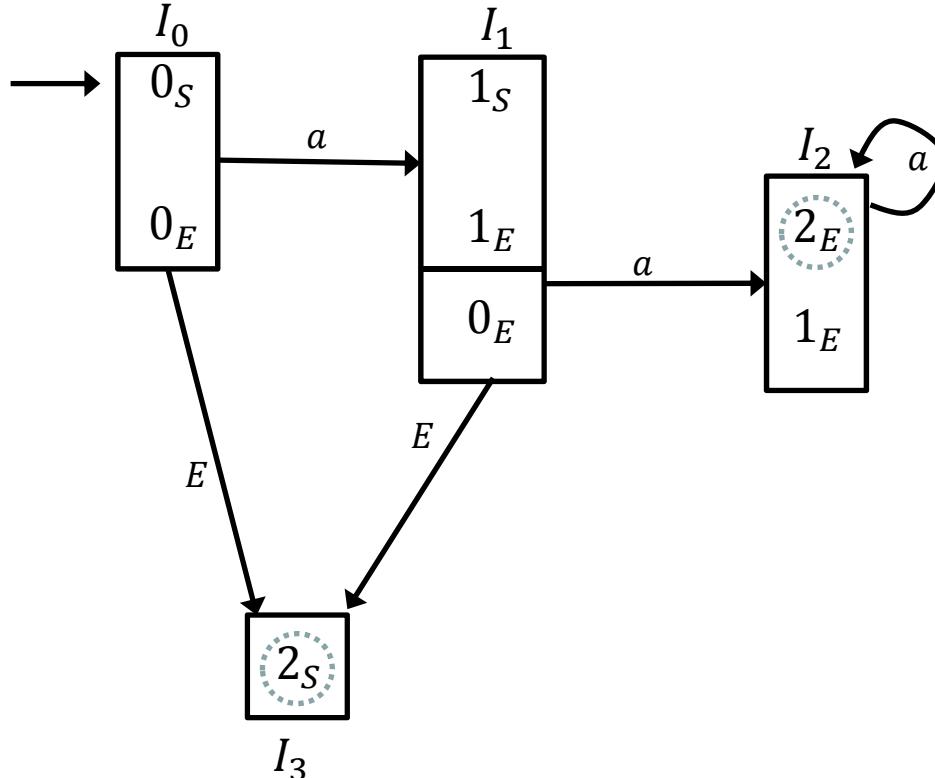
- construction of the *pilot automaton*, which
 - Guides the analysis of the stack automaton
 - Incorporates in its states (macro-states, m-states) info on all possible computations, which, scanning a sentential form, reach that m-state
- if necessary, the m-states contain many (machine)states corresponding to various conjectures on the derivation (analysis threads) investigated in parallel
- *in the stack* of the parser the states of the *analysis threads* are linked using pointers
- the pilot automaton is built statically (in advance *once forall*) and used to guide all possible analyses



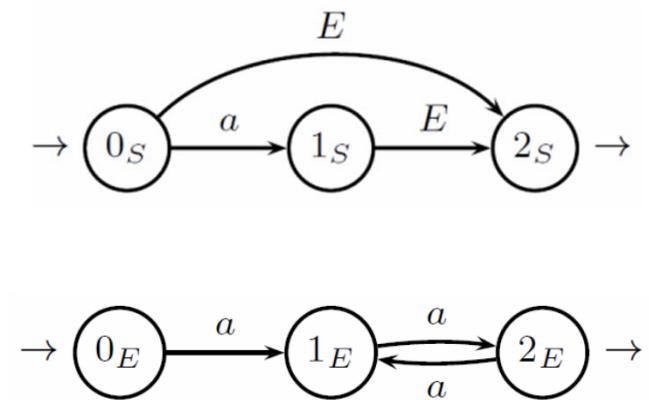
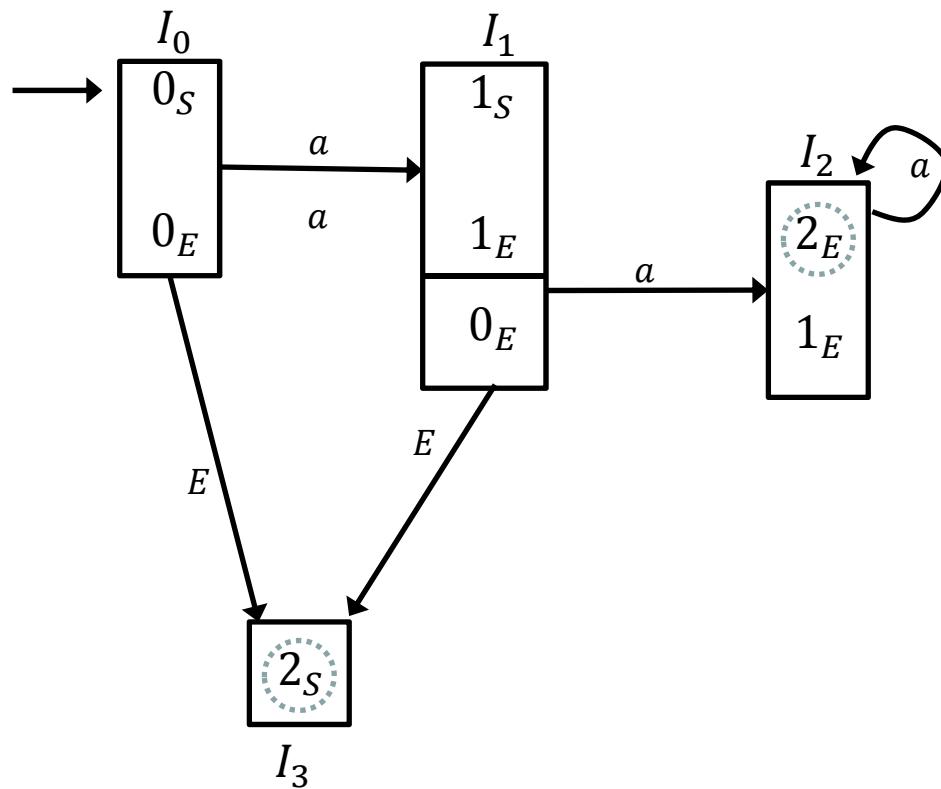
- construction of the *pilot automaton*, which
 - Guides the analysis of the stack automaton
 - Incorporates in its states (macro-states, m-states) info on all possible computations, which, scanning a sentential form, reach that m-state
- if necessary, the m-states contain many (machine)states corresponding to various conjectures on the derivation (analysis threads) investigated in parallel
- *in the stack* of the parser the states of the *analysis threads* are linked using pointers
- the pilot automaton is built statically (in advance *once forall*) and used to guide all possible analyses



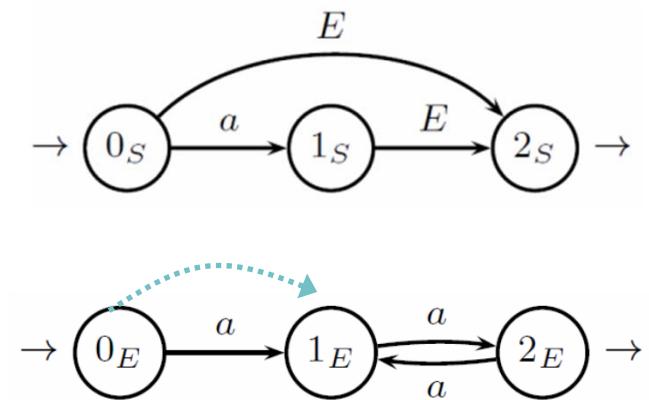
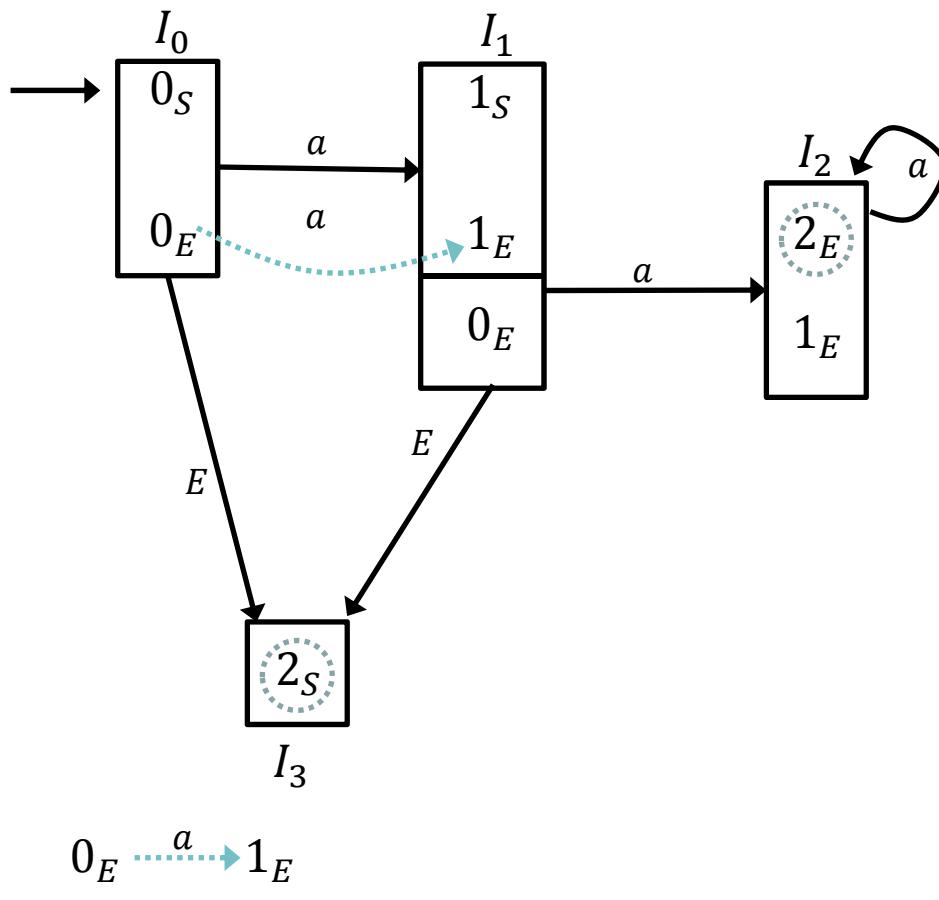
- construction of the *pilot automaton*, which
 - Guides the analysis of the stack automaton
 - Incorporates in its states (macro-states, m-states) info on all possible computations, which, scanning a sentential form, reach that m-state
- if necessary, the m-states contain many (machine)states corresponding to various conjectures on the derivation (analysis threads) investigated in parallel
- *in the stack* of the parser the states of the *analysis threads* are linked using pointers
- the pilot automaton is built statically (in advance *once forall*) and used to guide all possible analyses



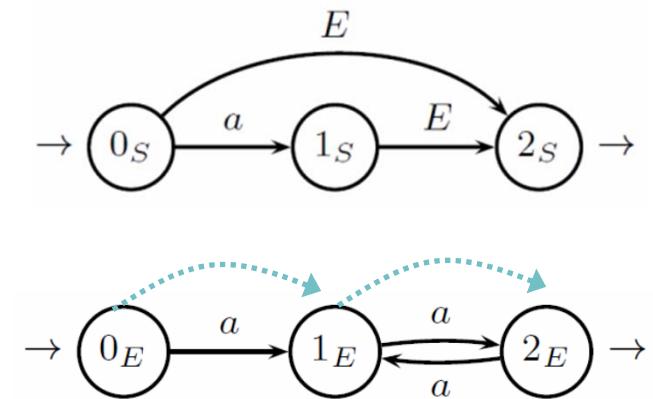
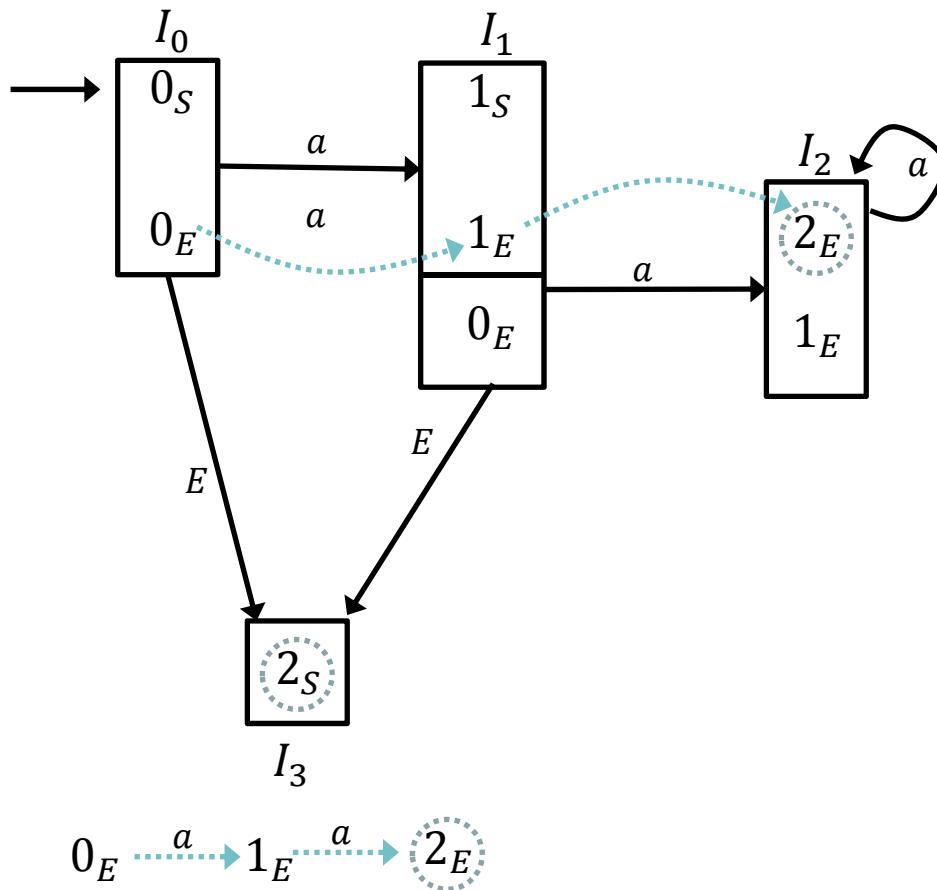
paths incorporated into the *pilot automaton*



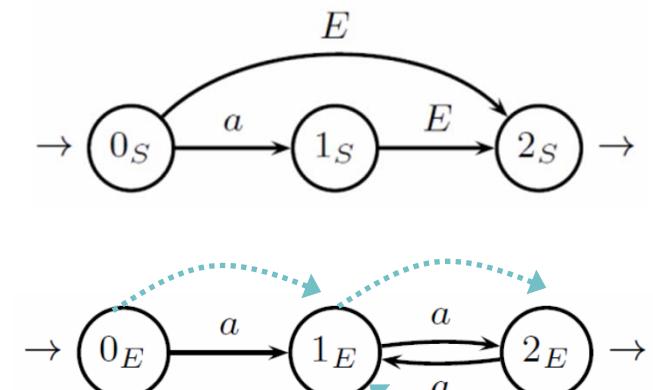
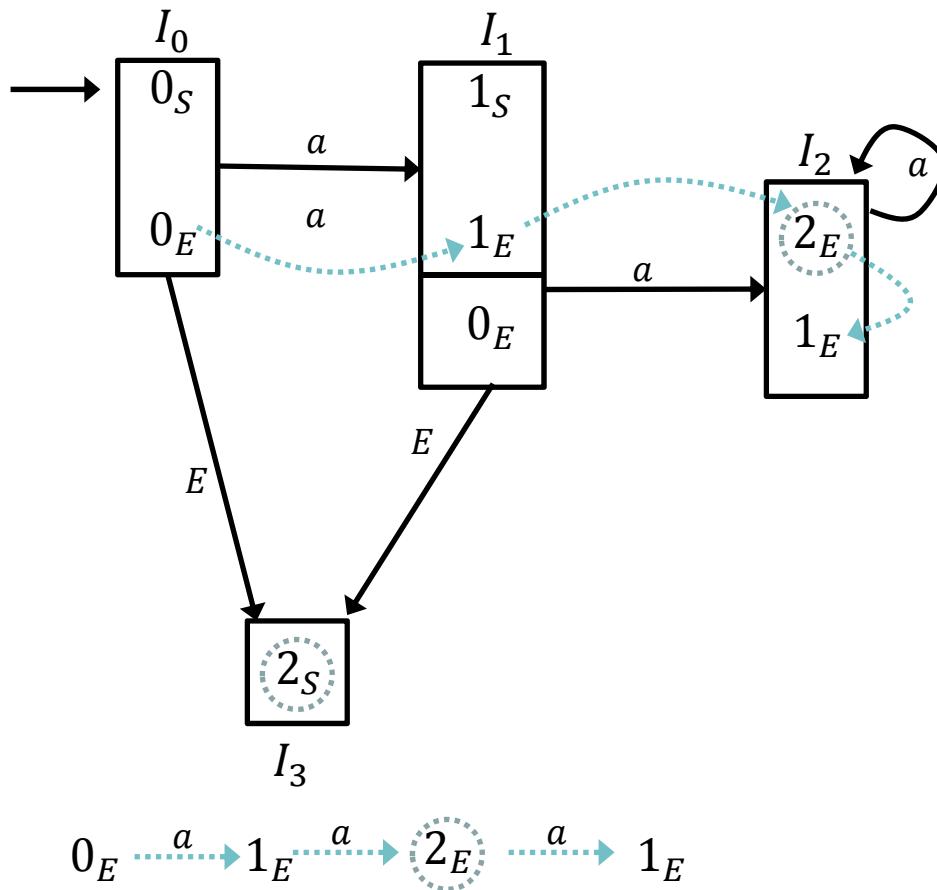
paths incorporated into the *pilot automaton*



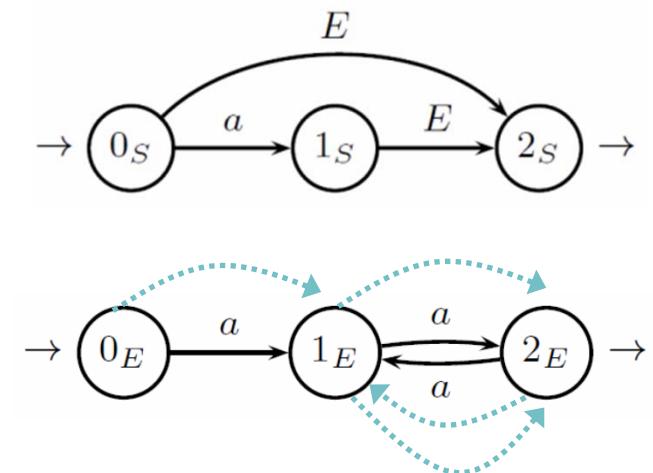
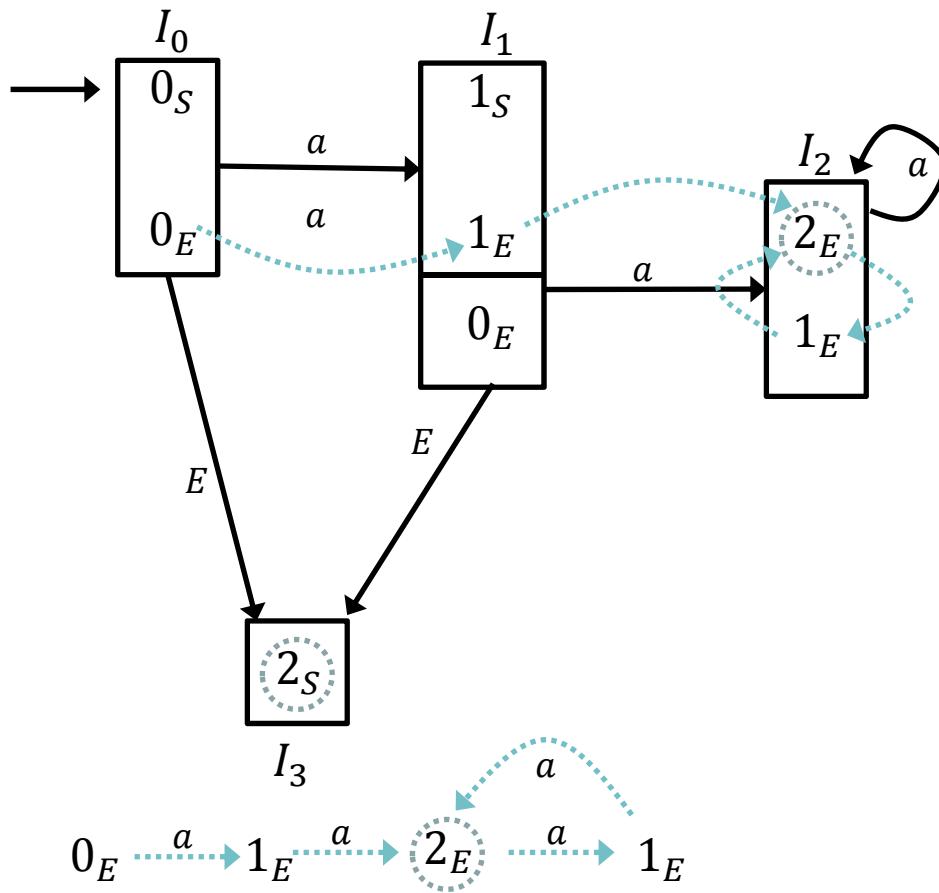
paths incorporated into the *pilot automaton*



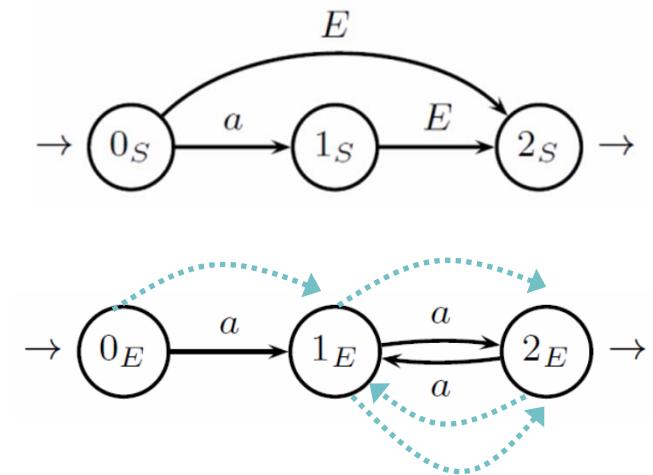
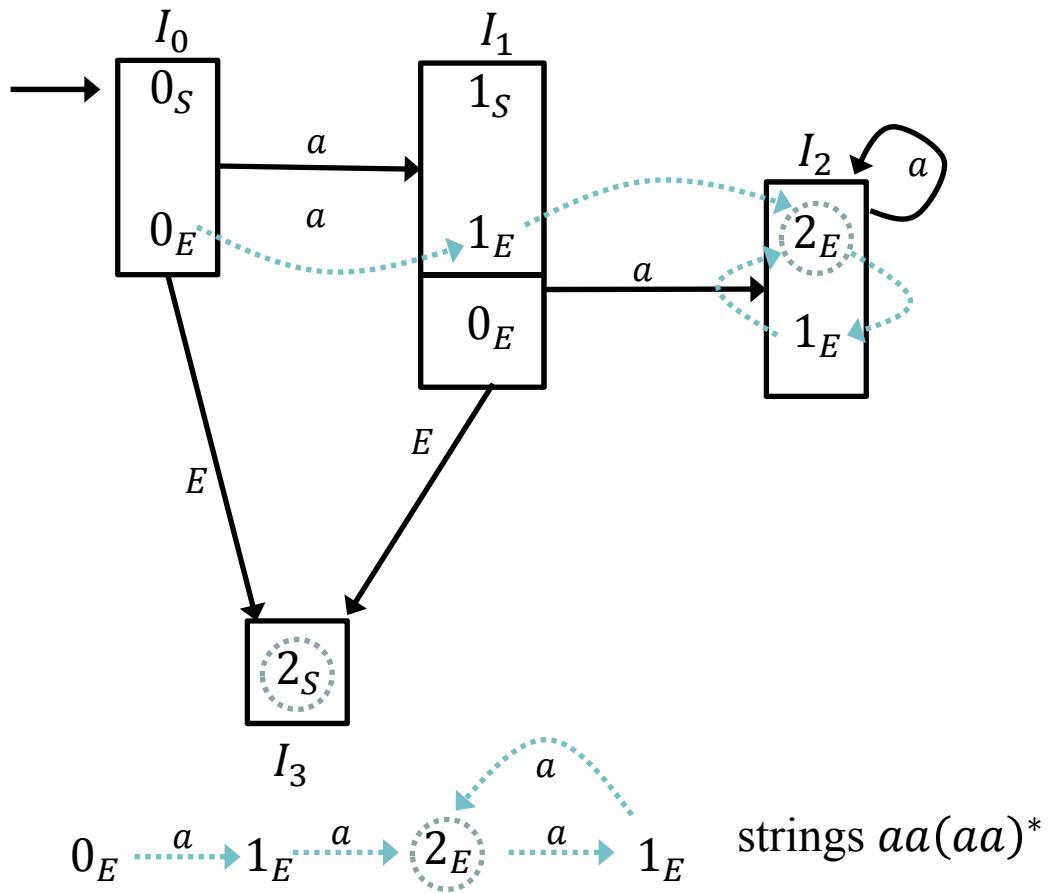
paths incorporated into the *pilot automaton*



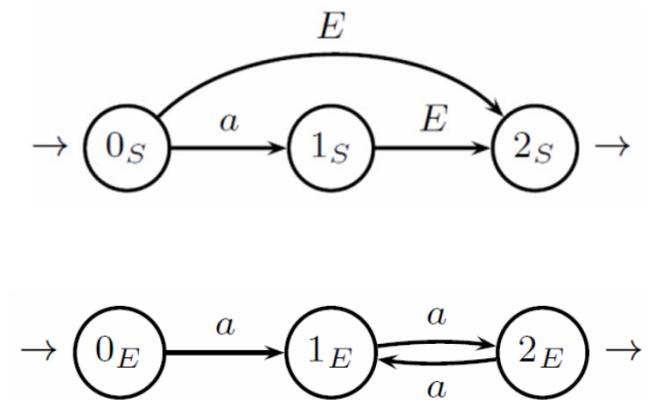
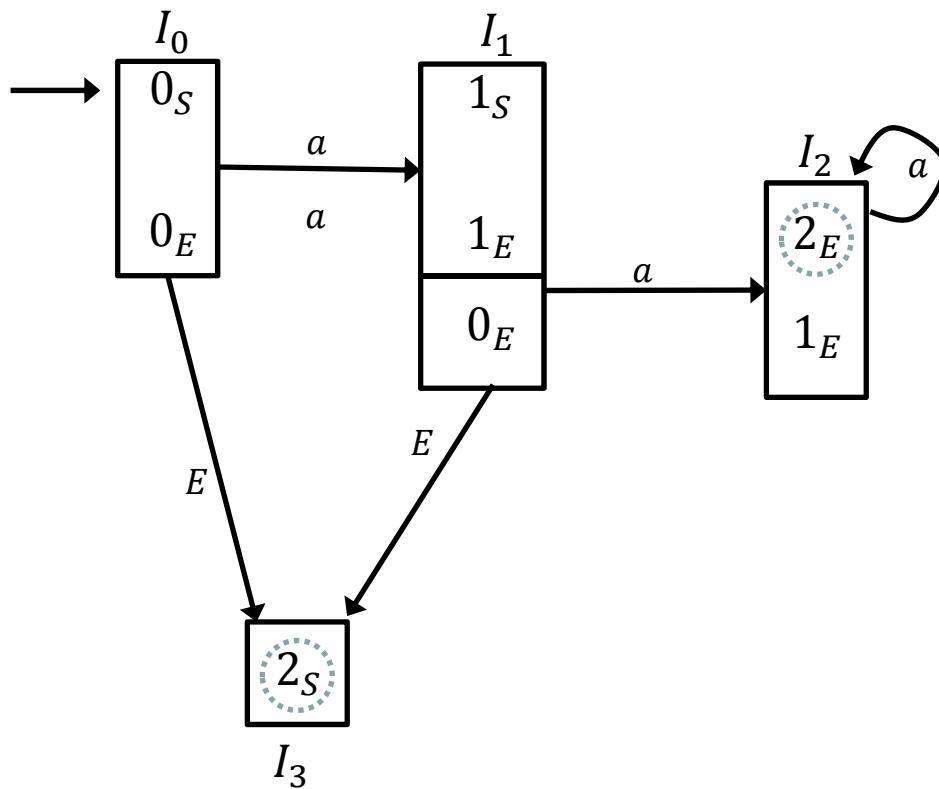
paths incorporated into the *pilot automaton*



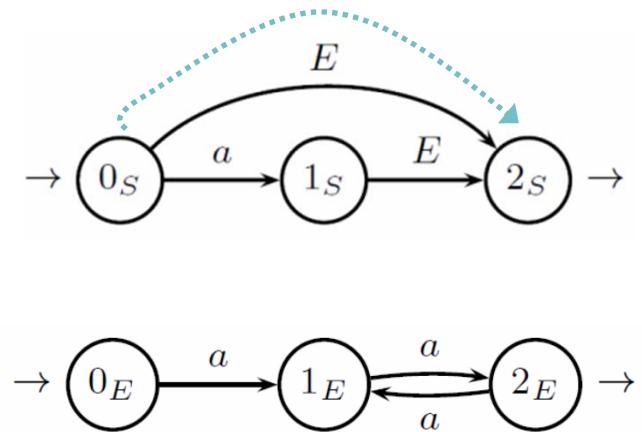
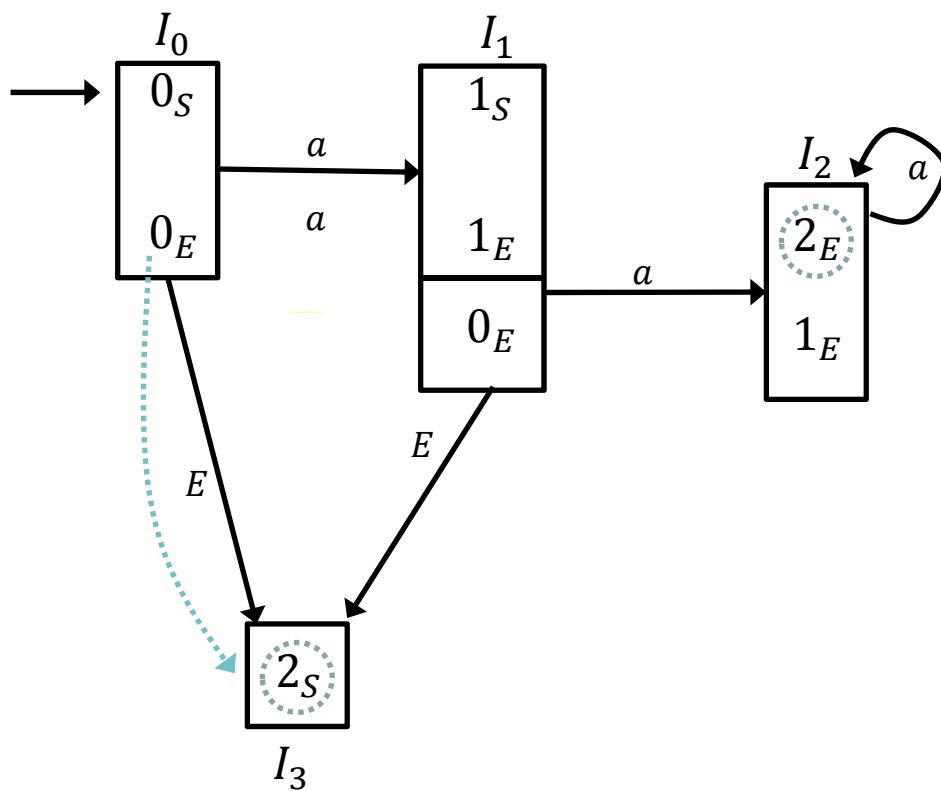
paths incorporated into the *pilot automaton*



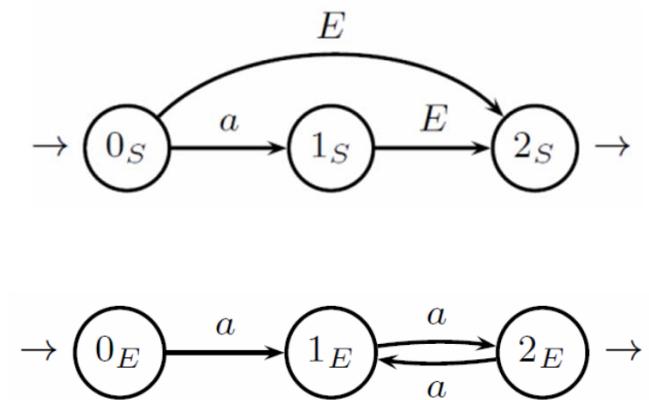
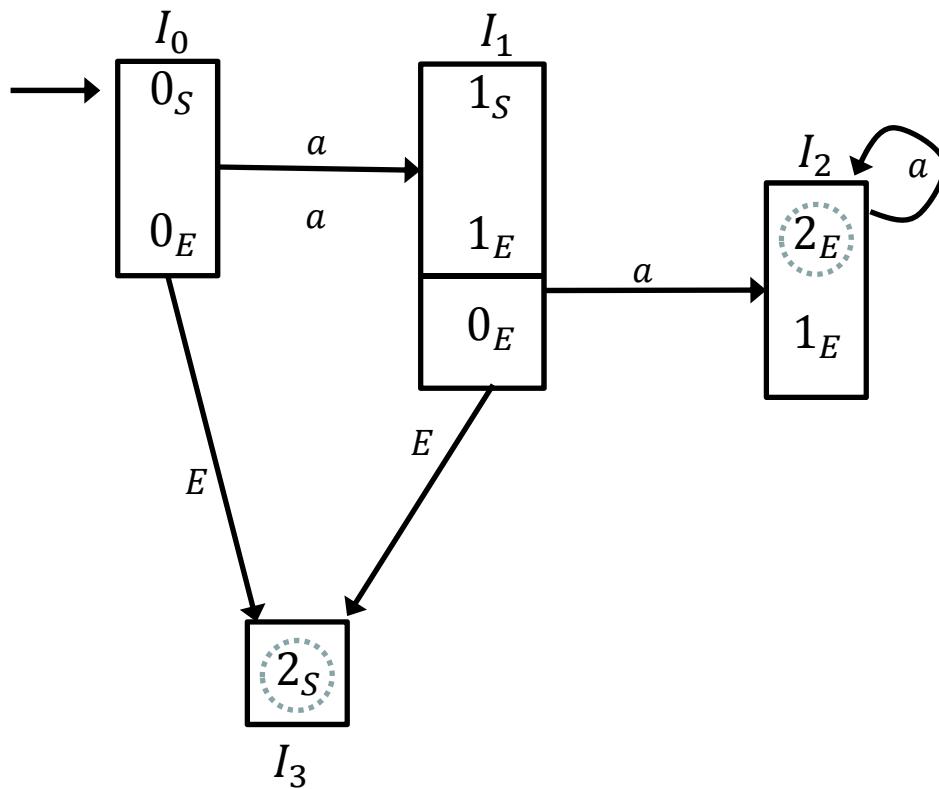
paths incorporated into the *pilot automaton*



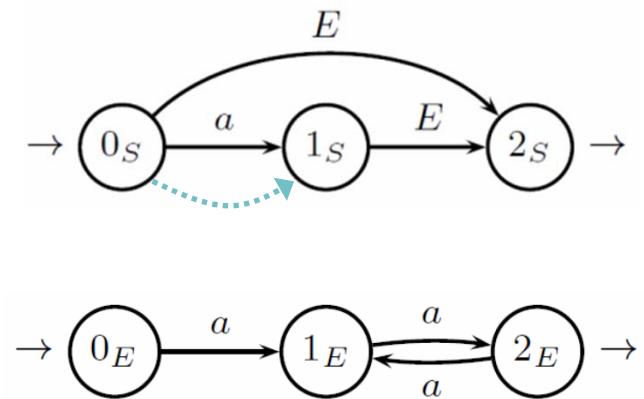
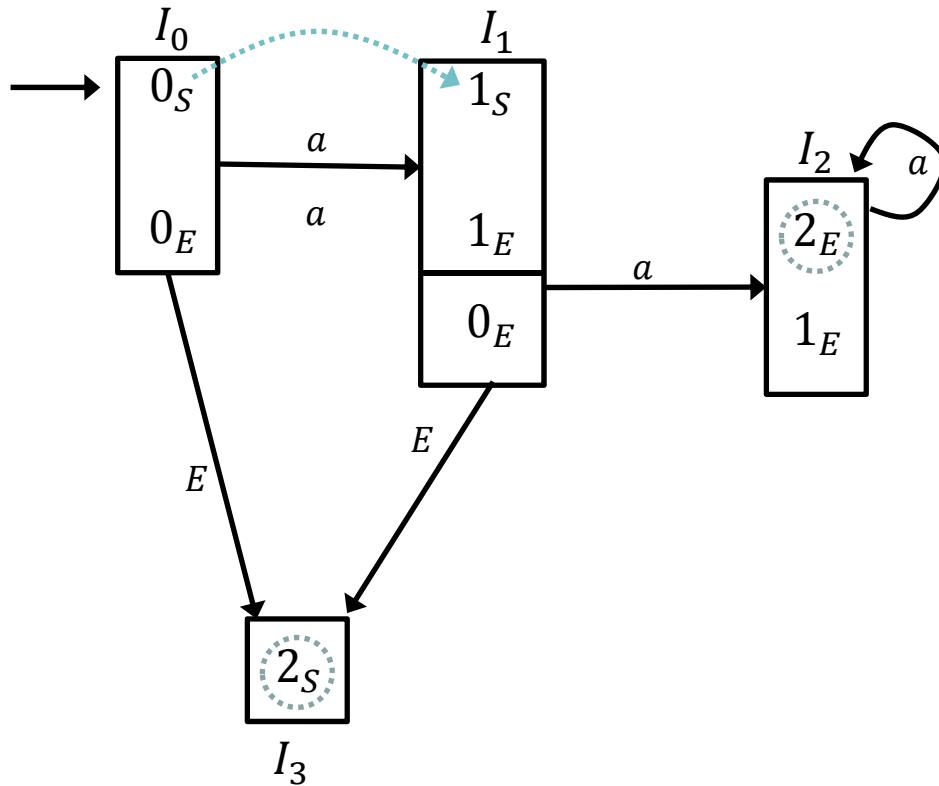
paths incorporated into the *pilot automaton*



paths incorporated into the *pilot automaton*

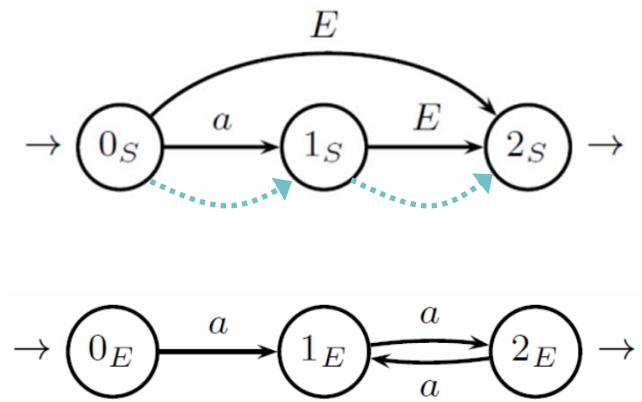
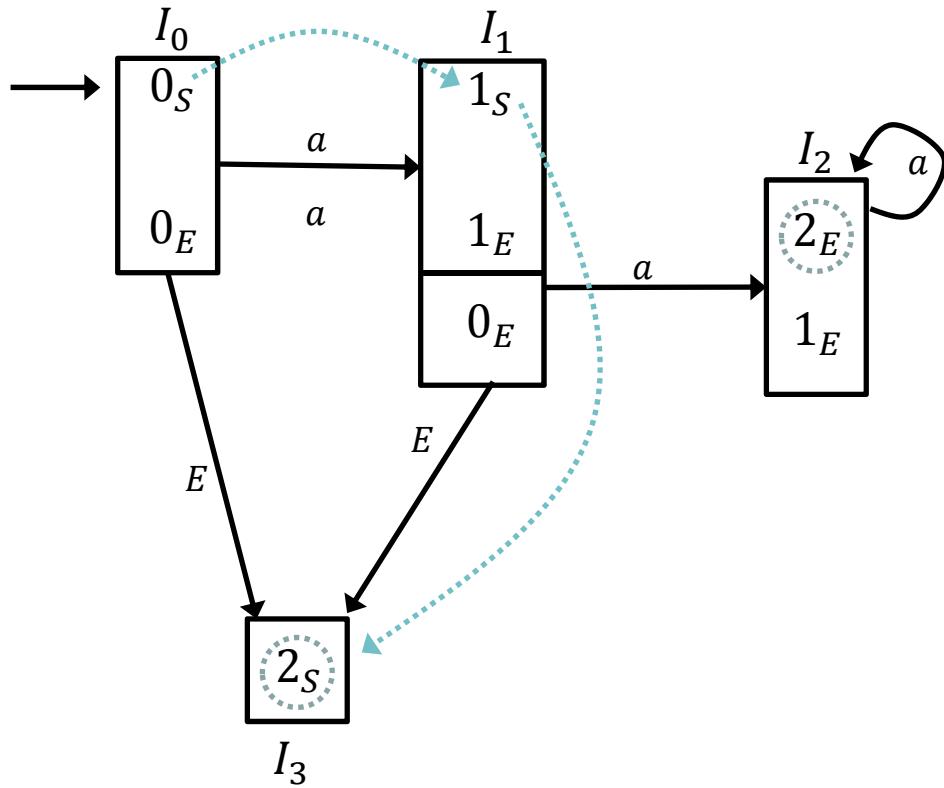


paths incorporated into the *pilot automaton*



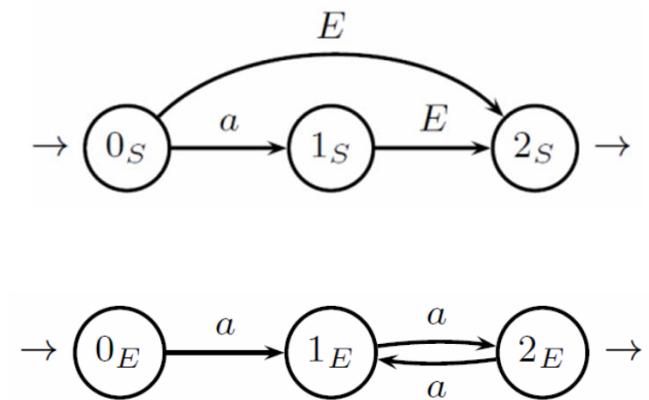
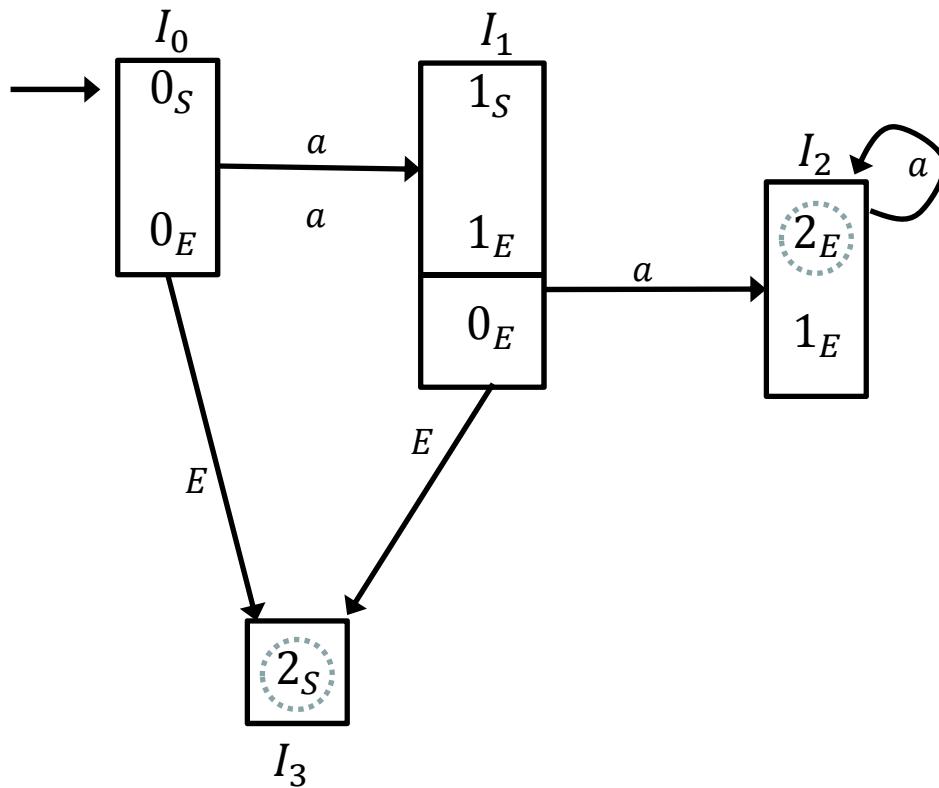
$$0_S \xrightarrow{a} 1_S$$

paths incorporated into the *pilot automaton*



$$0_S \xrightarrow{a} 1_S \xrightarrow{E} 2_S$$

paths incorporated into the *pilot automaton*

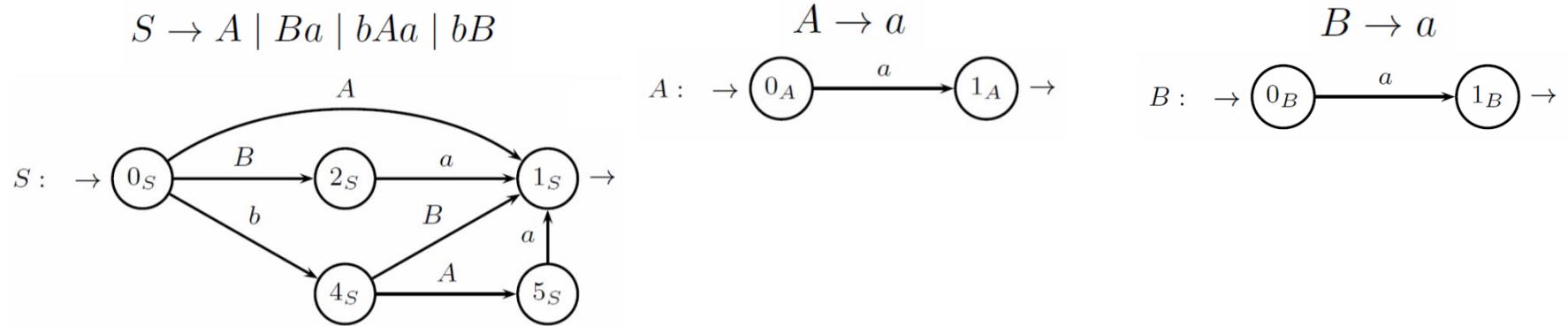


In the previous examples we didn't need lookahead since every character was the same, the only point was only to check if the string was terminated or not

LOOKAHEAD: the parser *looks ahead* to choose the next action:

Very often one char is enough : analysis **ELR(1)** (in gen.: **ELR(k)** looks at next k chars)

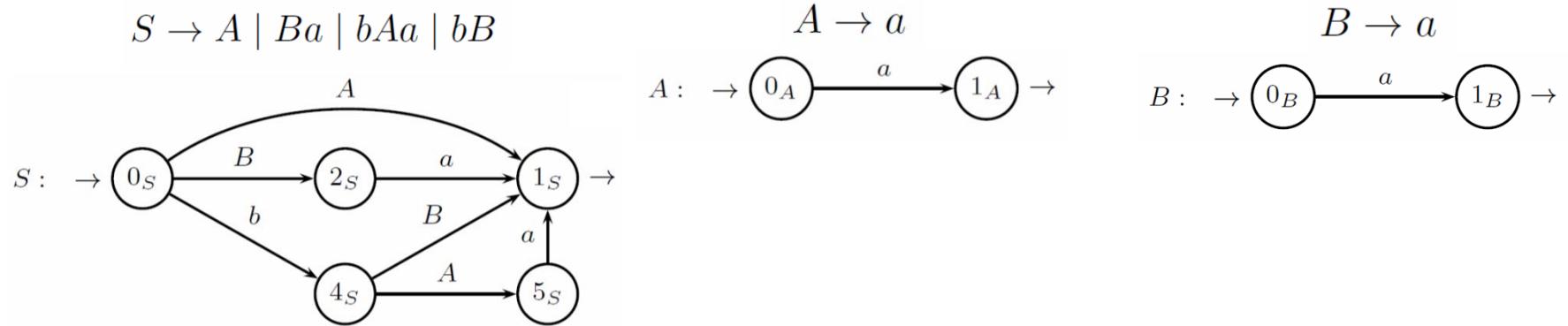
Example: language where lookahead is needed (NB a finite language (G not recursive): $L=\{a, aa, ba, baa\}$)



LOOKAHEAD: the parser *looks ahead* to choose the next action:

Very often one char is enough : analysis **ELR(1)** (in gen.: $ELR(k)$ looks at next k chars)

Example: language where lookahead is needed (NB a finite language (G not recursive): $L=\{a, aa, ba, baa\}$)

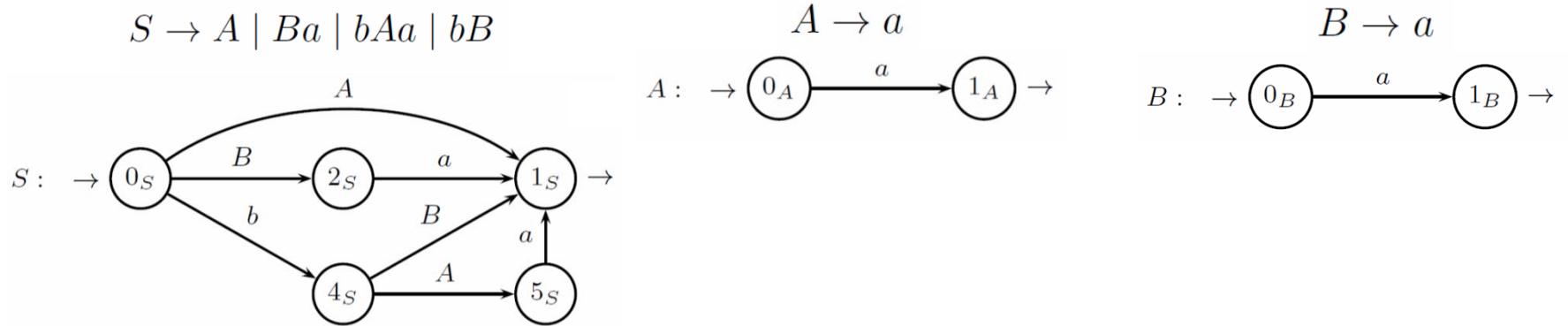


Consider analysis of strings **baa** \dashv and **ba** \dashv

LOOKAHEAD: the parser *looks ahead* to choose the next action:

Very often one char is enough : analysis **ELR(1)** (in gen.: $ELR(k)$ looks at next k chars)

Example: language where lookahead is needed (NB a finite language (G not recursive): $L=\{a, aa, ba, baa\}$)



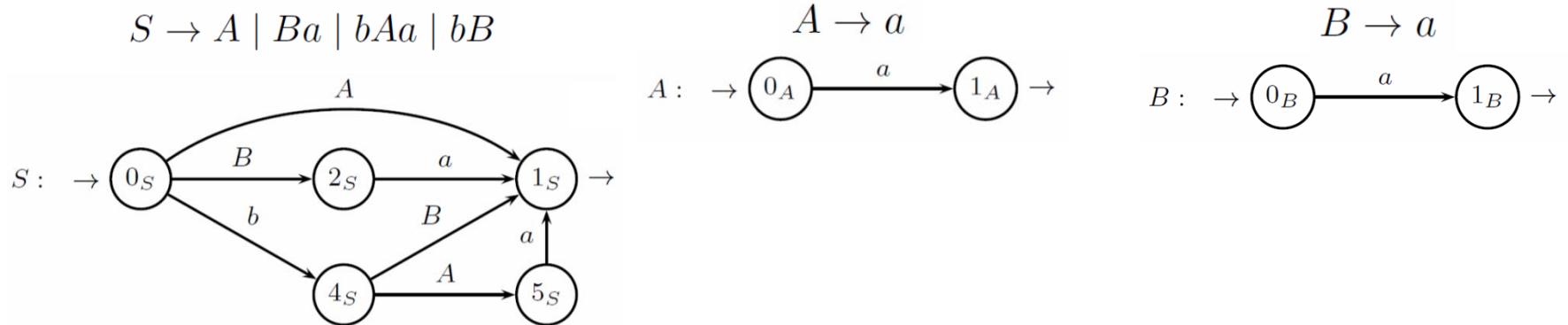
Consider analysis of strings **baa** \dashv and **ba** \dashv

0_S

LOOKAHEAD: the parser *looks ahead* to choose the next action:

Very often one char is enough : analysis **ELR(1)** (in gen.: $ELR(k)$ looks at next k chars)

Example: language where lookahead is needed (NB a finite language (G not recursive): $L=\{a, aa, ba, baa\}$)



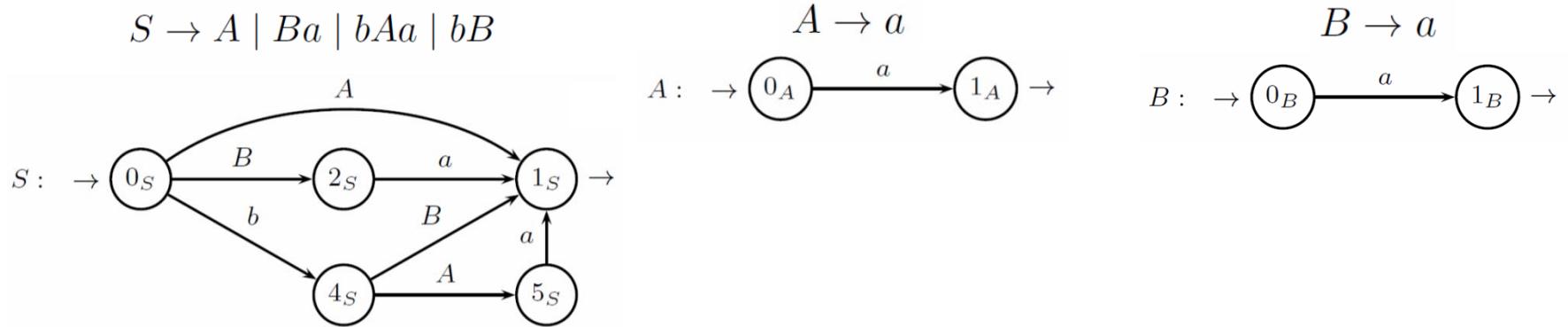
Consider analysis of strings **baa** \dashv and **ba** \dashv



LOOKAHEAD: the parser *looks ahead* to choose the next action:

Very often one char is enough : analysis **ELR(1)** (in gen.: $ELR(k)$ looks at next k chars)

Example: language where lookahead is needed (NB a finite language (G not recursive): $L=\{a, aa, ba, baa\}$)



Consider analysis of strings **baa** ⊢ and **ba** ⊢

$$0_S \xleftarrow{b} 4_S$$

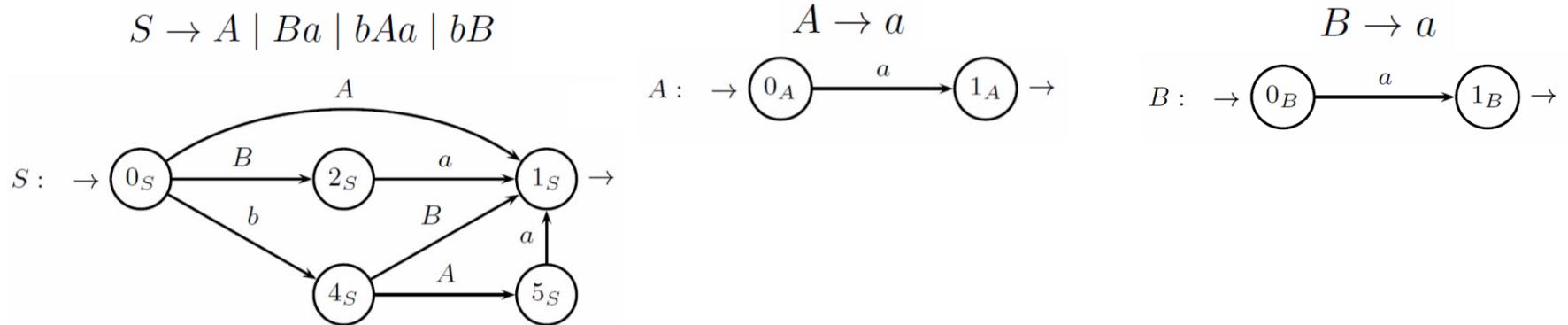
$$0_A$$

$$0_B$$

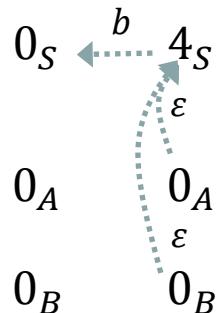
LOOKAHEAD: the parser *looks ahead* to choose the next action:

Very often one char is enough : analysis **ELR(1)** (in gen.: $ELR(k)$ looks at next k chars)

Example: language where lookahead is needed (NB a finite language (G not recursive): $L=\{a, aa, ba, baa\}$)



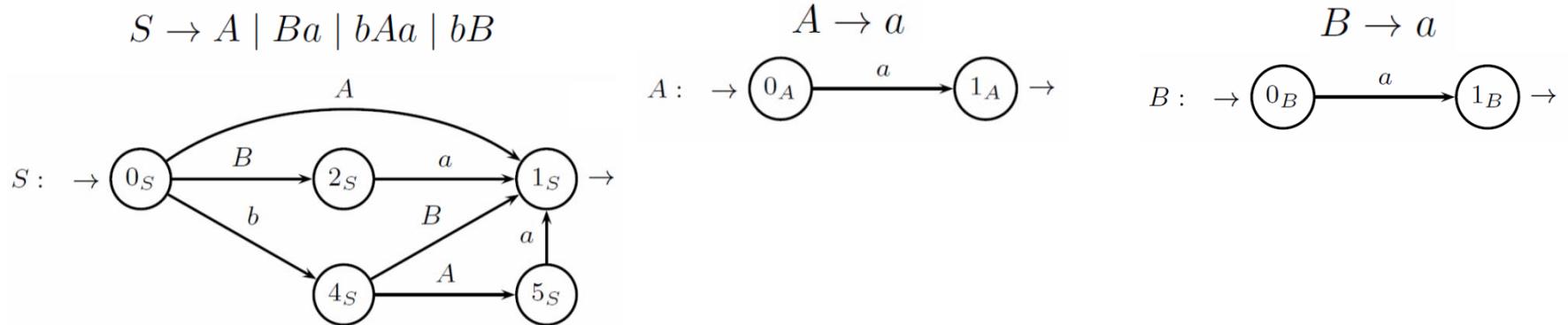
Consider analysis of strings **baa** \dashv and **ba** \dashv



LOOKAHEAD: the parser *looks ahead* to choose the next action:

Very often one char is enough : analysis **ELR(1)** (in gen.: $ELR(k)$ looks at next k chars)

Example: language where lookahead is needed (NB a finite language (G not recursive): $L=\{a, aa, ba, baa\}$)



Consider analysis of strings **baa** \dashv and **ba** \dashv

$$0_S \xleftarrow{b} 4_S$$

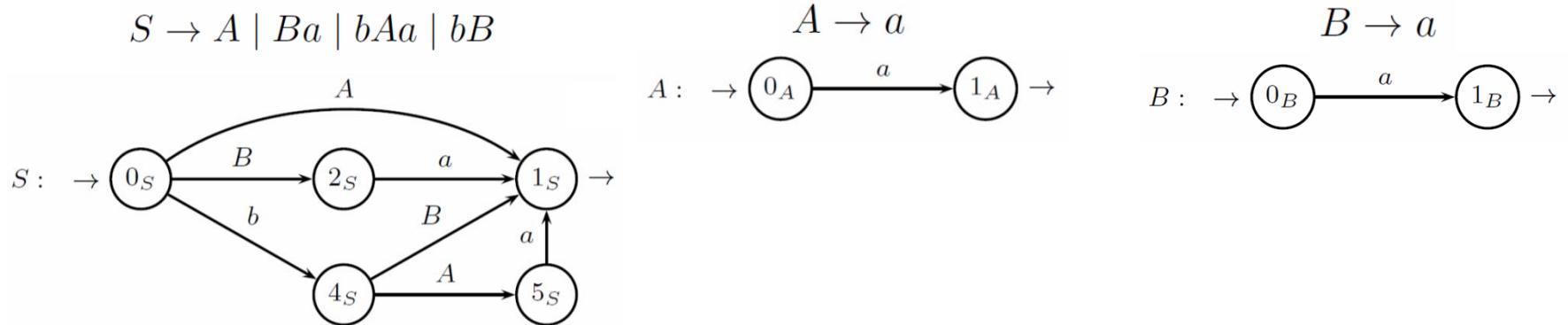
$$0_A \quad 0_A \xleftarrow{a} 1_A$$

$$0_B \quad 0_B \xleftarrow{a} 1_B$$

LOOKAHEAD: the parser *looks ahead* to choose the next action:

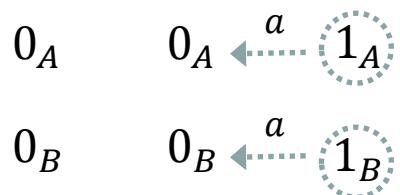
Very often one char is enough : analysis **ELR(1)** (in gen.: $ELR(k)$ looks at next k chars)

Example: language where lookahead is needed (NB a finite language (G not recursive): $L=\{a, aa, ba, baa\}$)



Consider analysis of strings $baa \dashv$ and $ba \dashv$

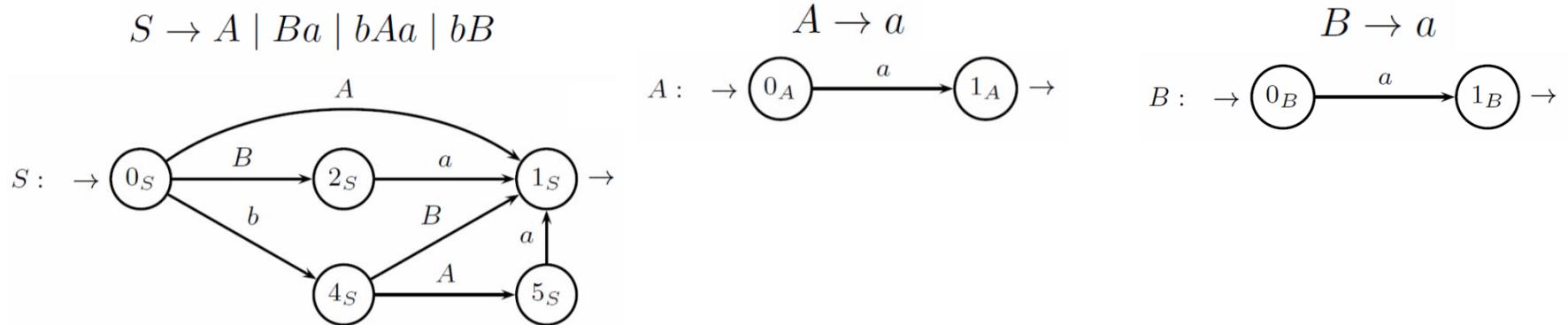
$0_S \xleftarrow{b} 4_S$ Both states 1_A and 1_B are final: a reduction is in order; $a \rightarrow A$ or $a \rightarrow B$?



LOOKAHEAD: the parser *looks ahead* to choose the next action:

Very often one char is enough : analysis **ELR(1)** (in gen.: $ELR(k)$ looks at next k chars)

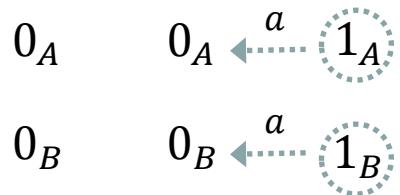
Example: language where lookahead is needed (NB a finite language (G not recursive): $L=\{a, aa, ba, baa\}$)



Consider analysis of strings **baa** ⊢ and **ba** ⊢

$0_S \xleftarrow{b} 4_S$ Both states 1_A and 1_B are final: a reduction is in order; $a \rightarrow A$ or $a \rightarrow B$?

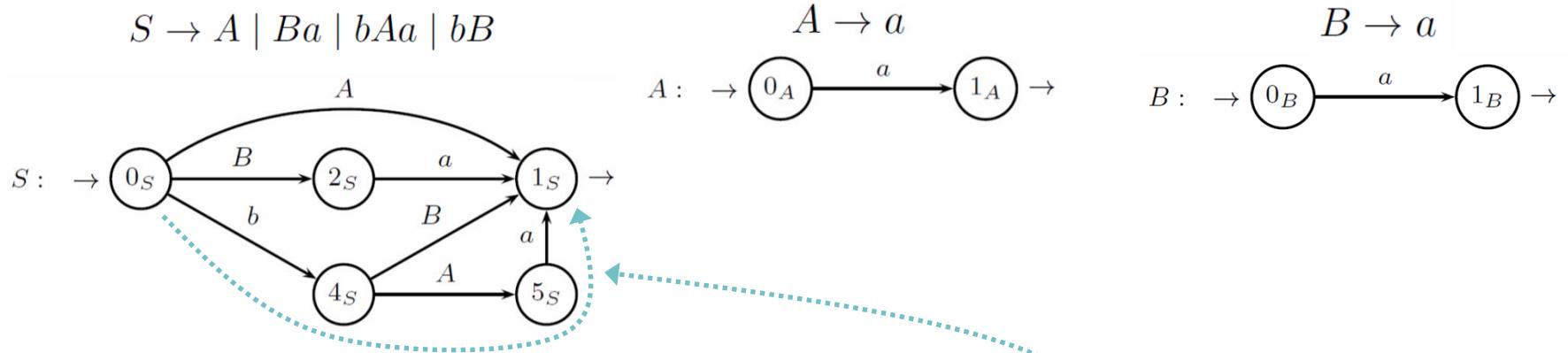
It depends on what comes next in the input



LOOKAHEAD: the parser *looks ahead* to choose the next action:

Very often one char is enough : analysis **ELR(1)** (in gen.: $ELR(k)$ looks at next k chars)

Example: language where lookahead is needed (NB a finite language (G not recursive): $L=\{a, aa, ba, baa\}$)



Consider analysis of strings $\mathbf{baa} \dashv$ and $\mathbf{ba} \dashv$

$$0_S \xleftarrow{b} 4_S$$

Both states 1_A and 1_B are final: a reduction is in order; $a \rightarrow A$ or $a \rightarrow B$?

It depends on what comes next in the input

$$0_A$$

$$0_A \xleftarrow{a} 1_A$$

if there is an a then the derivation is $S \Rightarrow bAa \Rightarrow baa$ hence reduction is $a \rightarrow A$

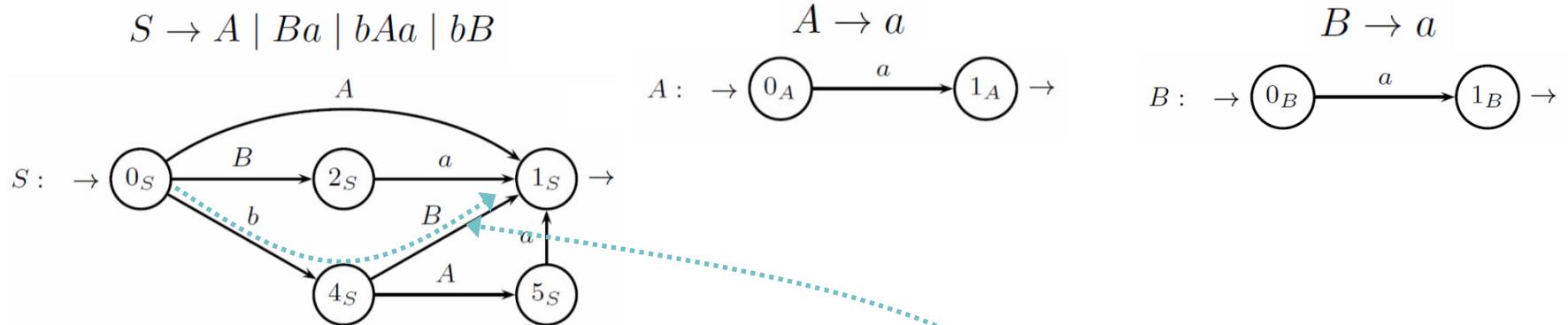
$$0_B$$

$$0_B \xleftarrow{a} 1_B$$

LOOKAHEAD: the parser *looks ahead* to choose the next action:

Very often one char is enough : analysis **ELR(1)** (in gen.: $ELR(k)$ looks at next k chars)

Example: language where lookahead is needed (NB a finite language (G not recursive): $L=\{a, aa, ba, baa\}$)



Consider analysis of strings $\mathbf{baa} \dashv$ and $\mathbf{ba} \dashv$

$$0_S \xleftarrow{b} 4_S$$

Both states 1_A and 1_B are final: a reduction is in order; $a \rightarrow A$ or $a \rightarrow B$?

It depends on what comes next in the input

$$0_A$$

$$0_A \xleftarrow{a} 1_A$$

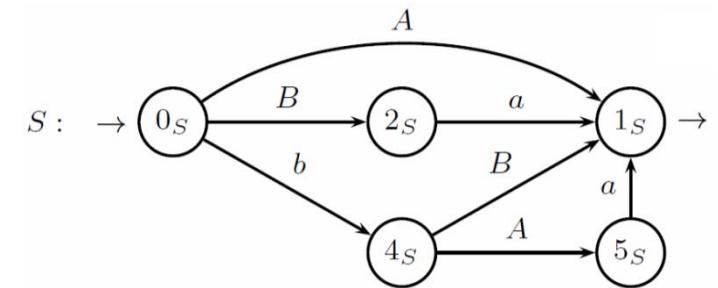
if there is an a then the derivation is $S \Rightarrow bAa \Rightarrow baa$ hence reduction is $a \rightarrow A$

$$0_B$$

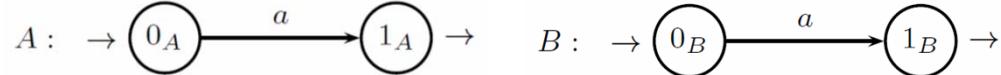
$$0_B \xleftarrow{a} 1_B$$

if there is a \dashv then the derivation is $S \Rightarrow bB \Rightarrow ba$ hence reduction is $a \rightarrow B$

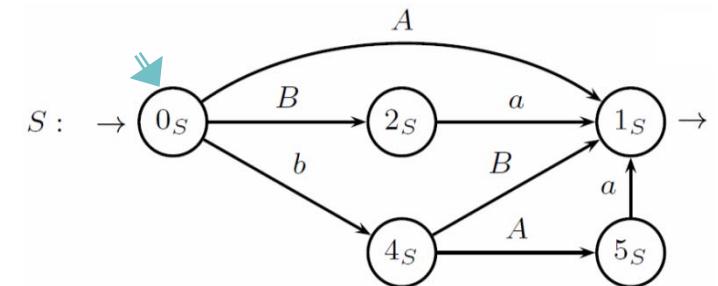
Let us build the relevant part of the pilot automaton



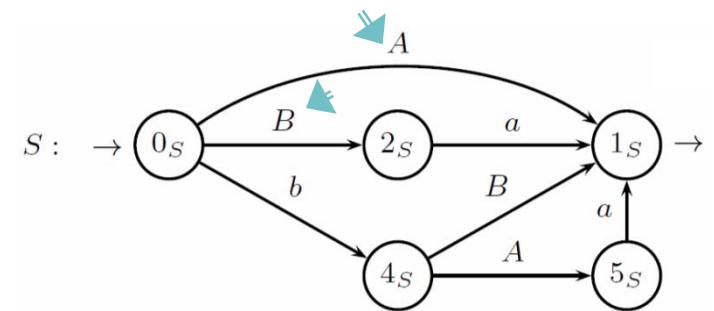
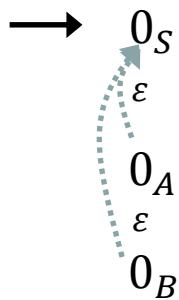
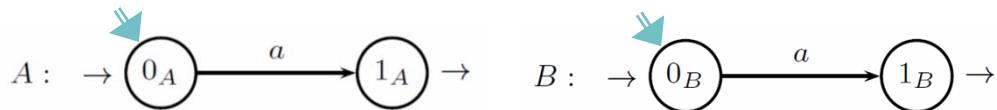
Let us build the relevant part of the pilot automaton



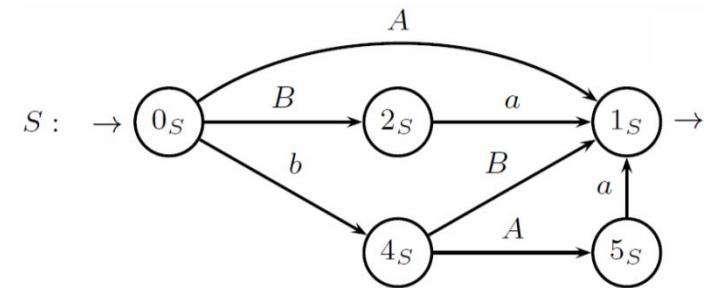
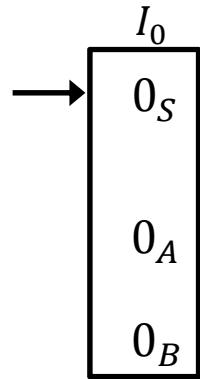
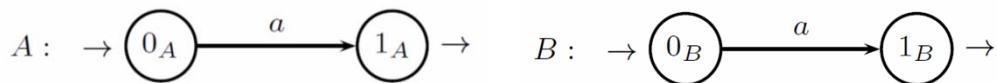
$\rightarrow 0_S$



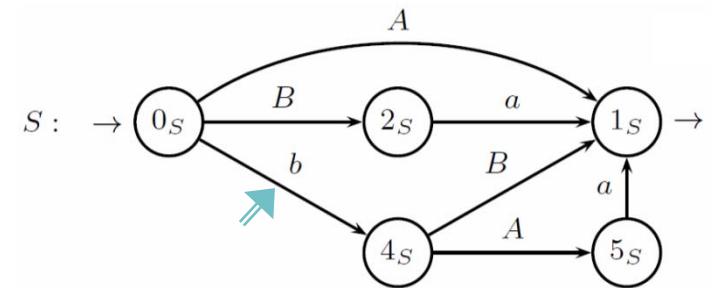
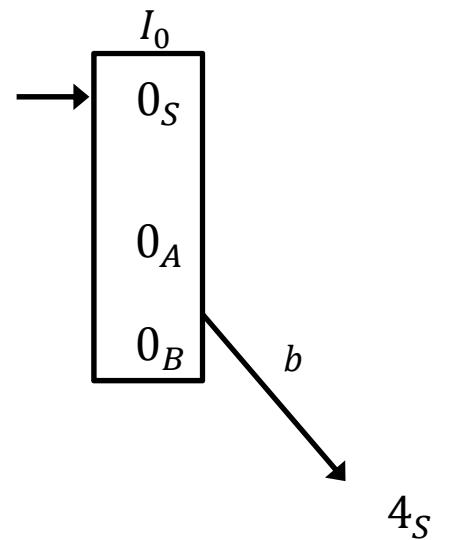
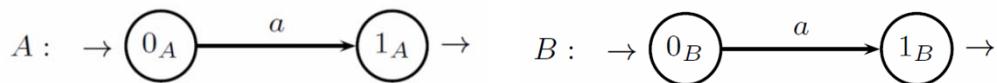
Let us build the relevant part of the pilot automaton



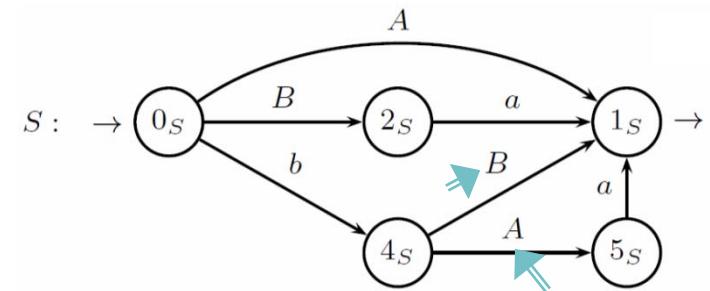
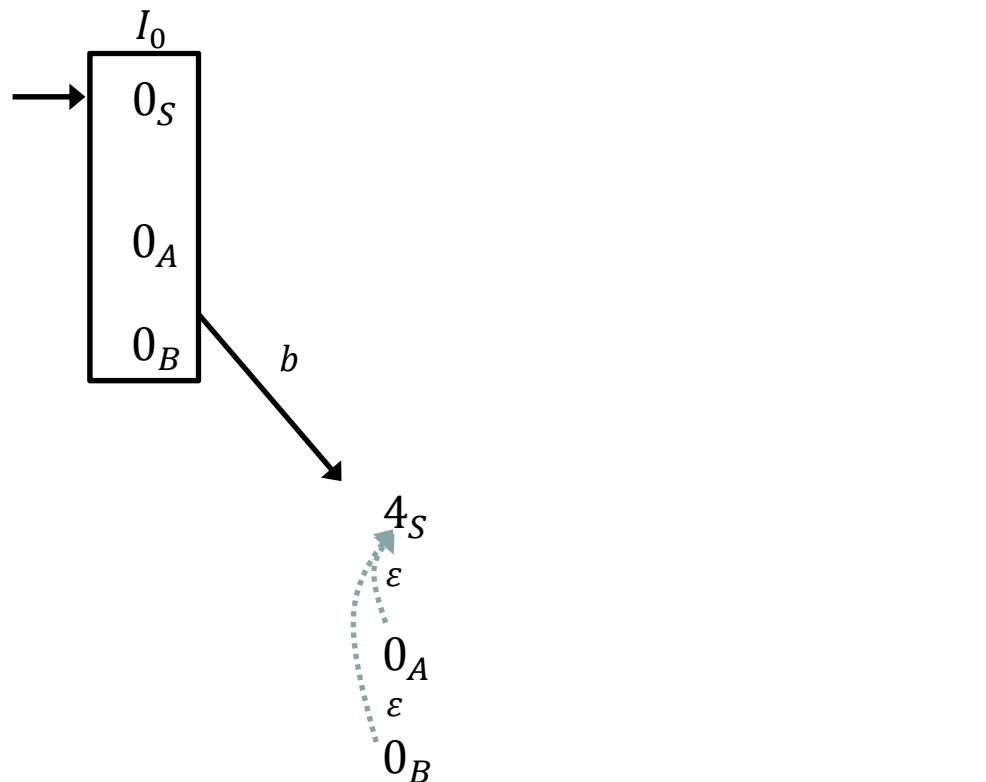
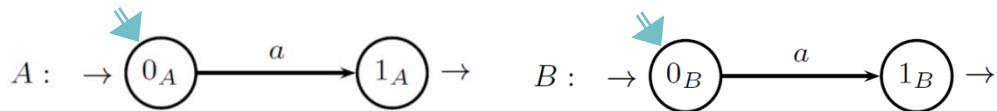
Let us build the relevant part of the pilot automaton



Let us build the relevant part of the pilot automaton

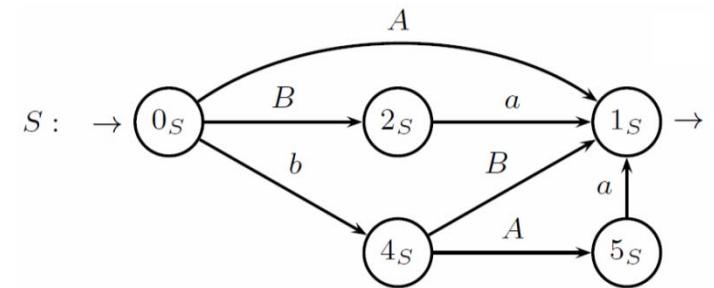
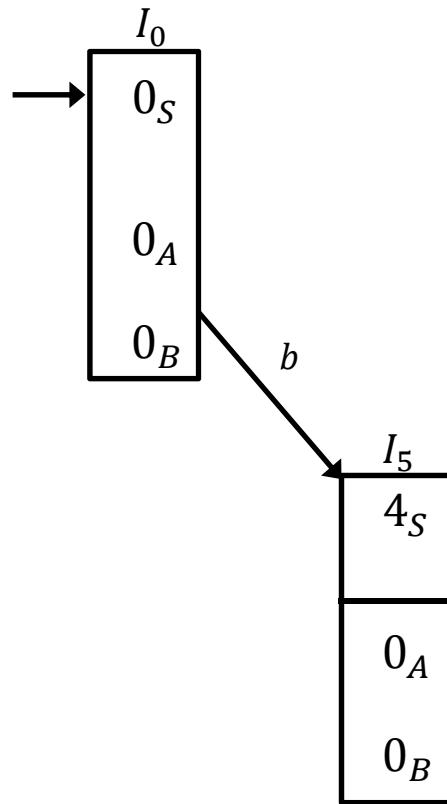


Let us build the relevant part of the pilot automaton

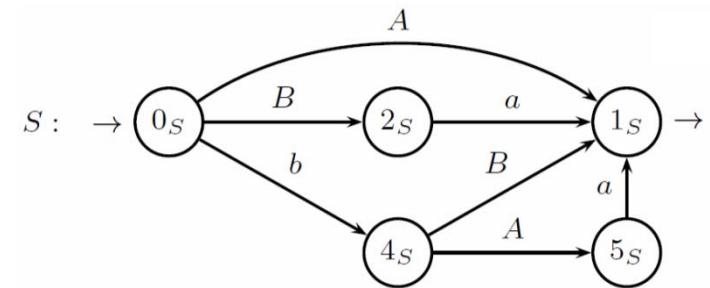
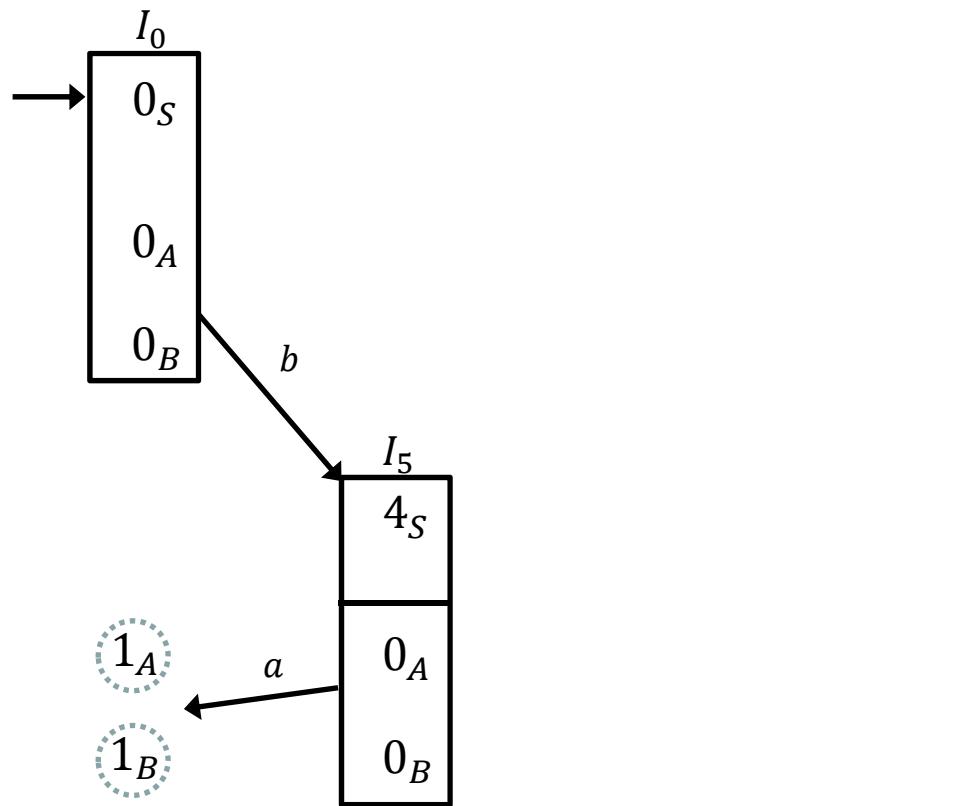
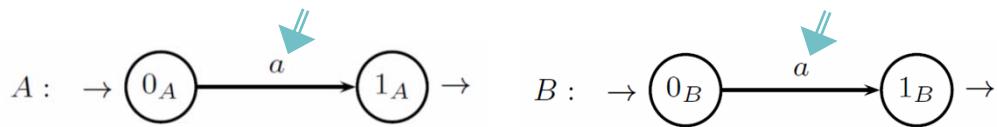


Let us build the relevant part of the pilot automaton

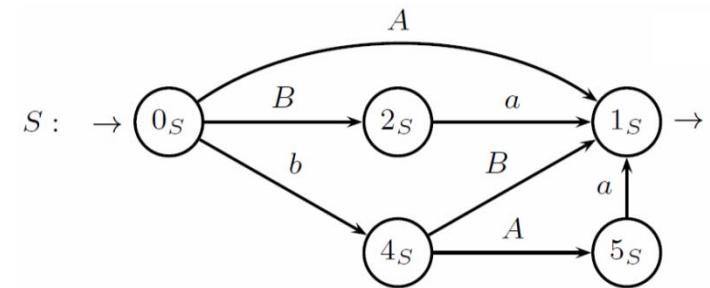
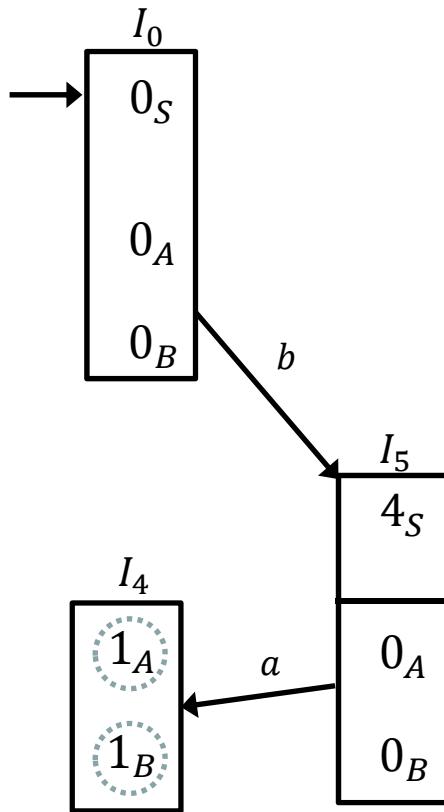
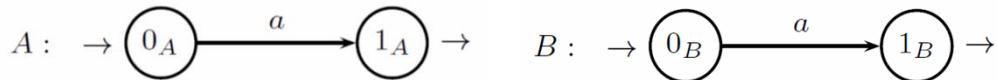
$$A : \rightarrow 0_A \xrightarrow{a} 1_A \rightarrow \quad B : \rightarrow 0_B \xrightarrow{a} 1_B \rightarrow$$



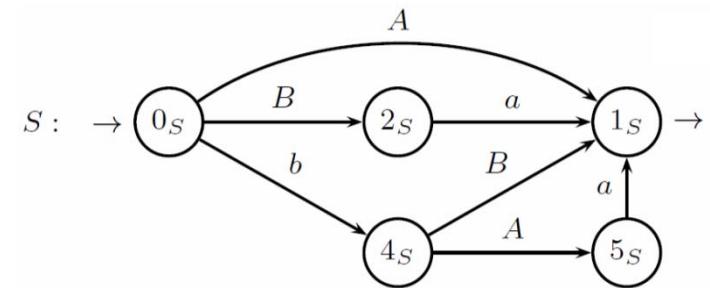
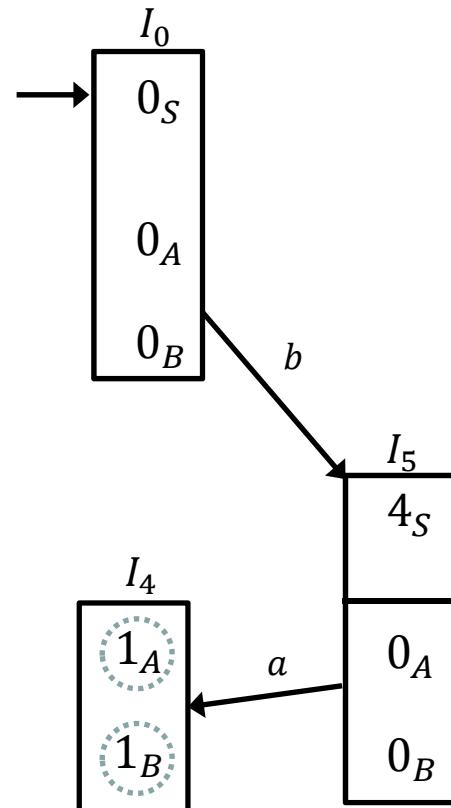
Let us build the relevant part of the pilot automaton



Let us build the relevant part of the pilot automaton

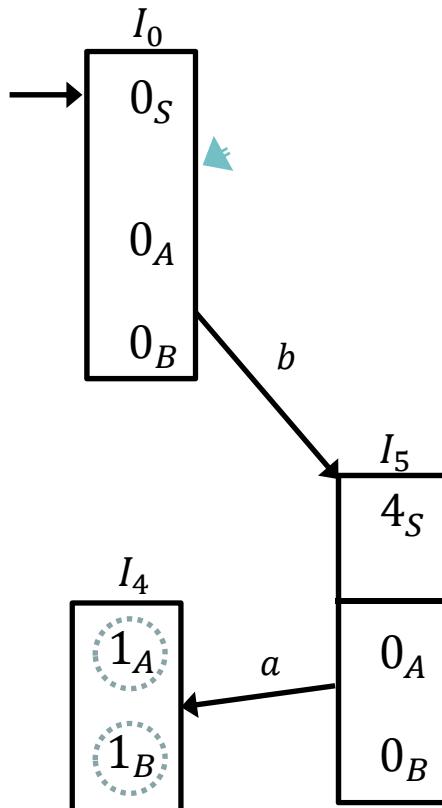
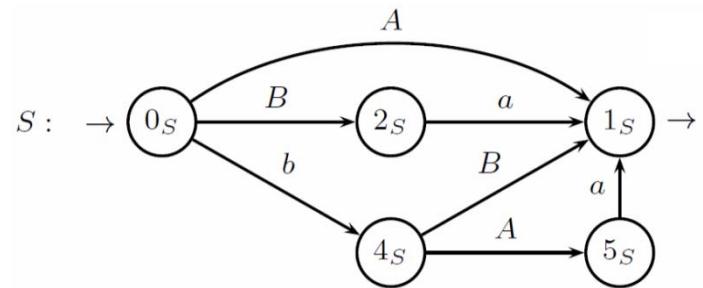


Let us build the relevant part of the pilot automaton

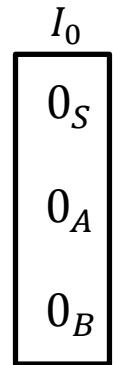


We use the pilot automaton for analyzing string $ba \dots$

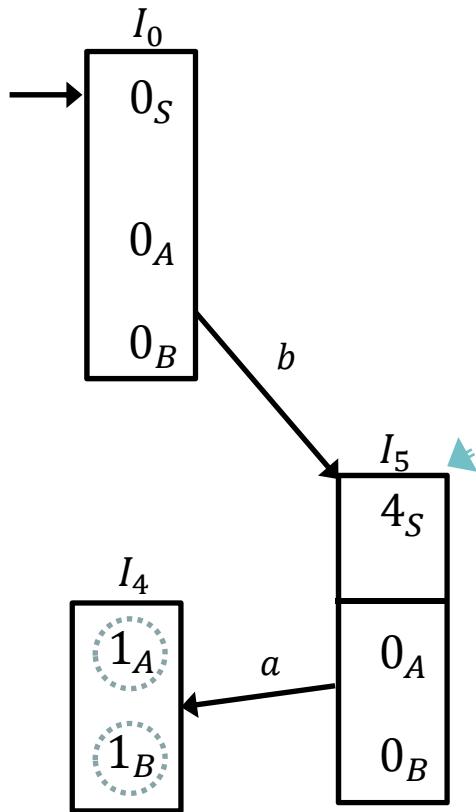
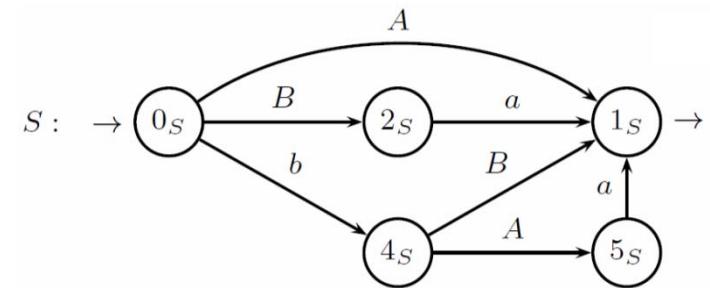
Let us build the relevant part of the pilot automaton



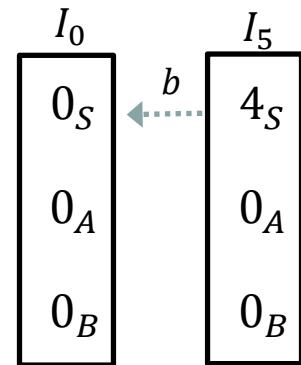
We use the pilot automaton for analyzing string $ba \dots$



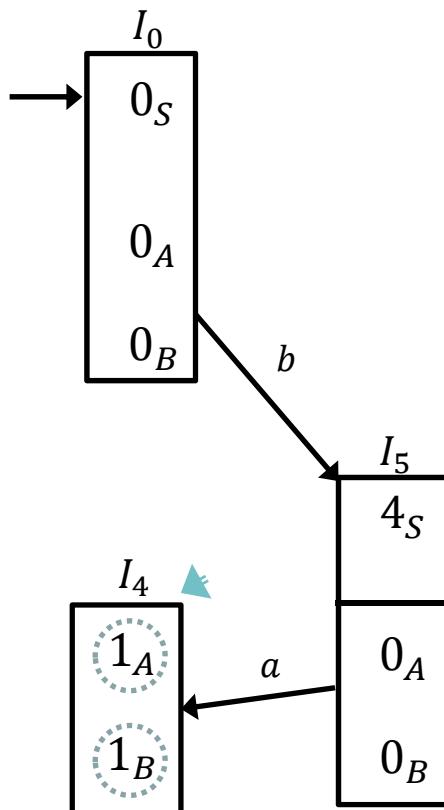
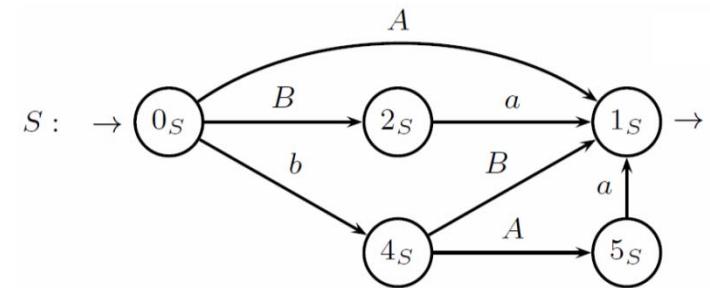
Let us build the relevant part of the pilot automaton



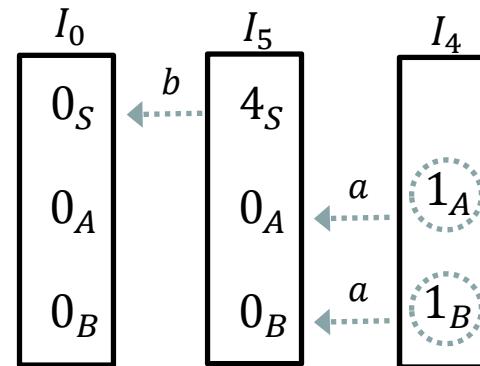
We use the pilot automaton for analyzing string $ba \dots$



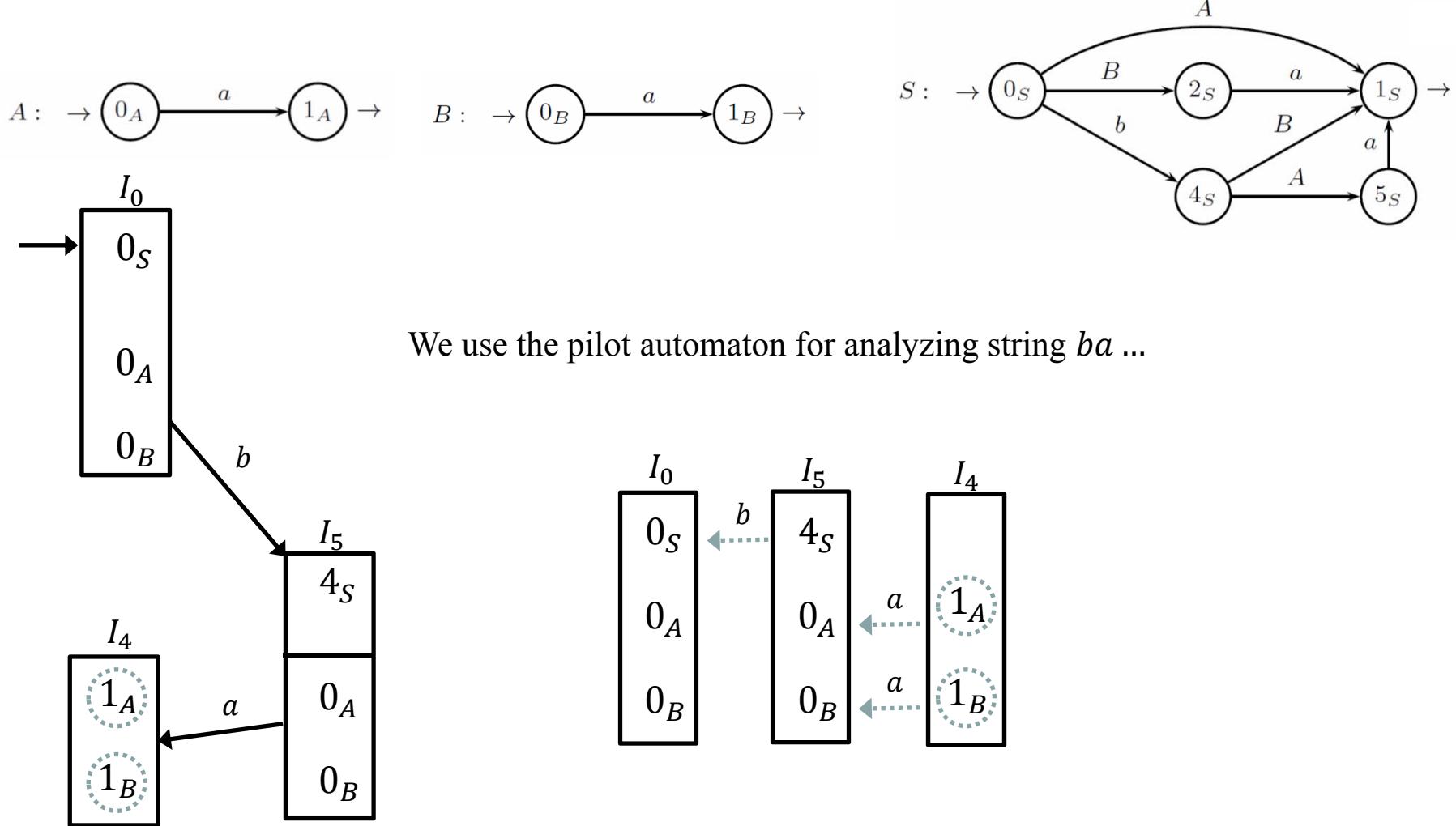
Let us build the relevant part of the pilot automaton



We use the pilot automaton for analyzing string $ba \dots$

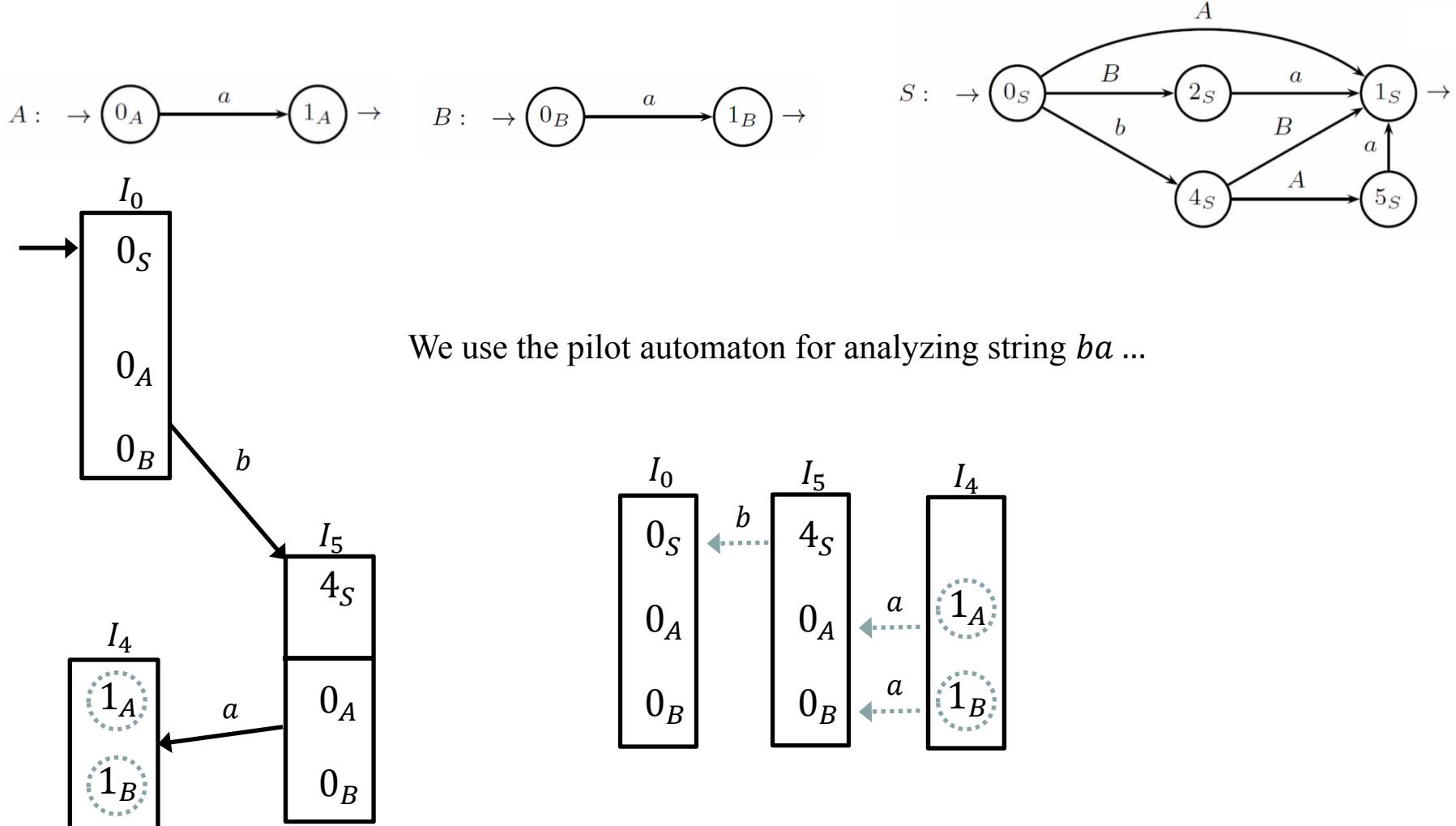


Let us build the relevant part of the pilot automaton



the pilot's m-state I_4 **does not help** in choosing between the two possible reductions ($a \rightarrow A$ and $a \rightarrow B$)

Let us build the relevant part of the pilot automaton



the pilot's m-state I_4 **does not help** in choosing between the two possible reductions ($a \rightarrow A$ and $a \rightarrow B$)

This situation is called **reduction-reduction conflict**

we need to use the lookahead in the pilot automaton to solve this conflict

Solution: we incorporate the *lookahead* in the pilot automaton m-state
as a consequence the number of m-states increases: the automaton executes a “subtler” analysis

Solution: we incorporate the *lookahead* in the pilot automaton m-state
as a consequence the number of m-states increases: the automaton executes a “subtler” analysis

lookahead = set of chars that can follow the string derived from the current nonterminal

a char in the lookahead will be the input when the final state of the current machine will be reached
its presence in the input indicates that the reduction is in order

Solution: we incorporate the *lookahead* in the pilot automaton m-state
as a consequence the number of m-states increases: the automaton executes a “subtler” analysis

lookahead = set of chars that can follow the string derived from the current nonterminal

a char in the lookahead will be the input when the final state of the current machine will be reached
its presence in the input indicates that the reduction is in order

Now every pilot m-state contains a set of *items* (also called *candidates* in the textbook)

item = pair: $\langle \text{state}, \text{lookahead} \rangle$

Solution: we incorporate the ***lookahead*** in the pilot automaton m-state

as a consequence the number of m-states increases: the automaton executes a “subtler” analysis

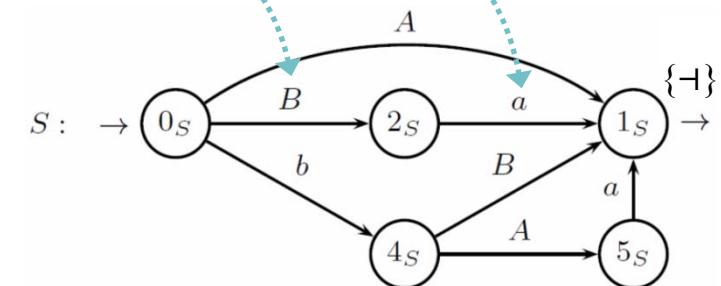
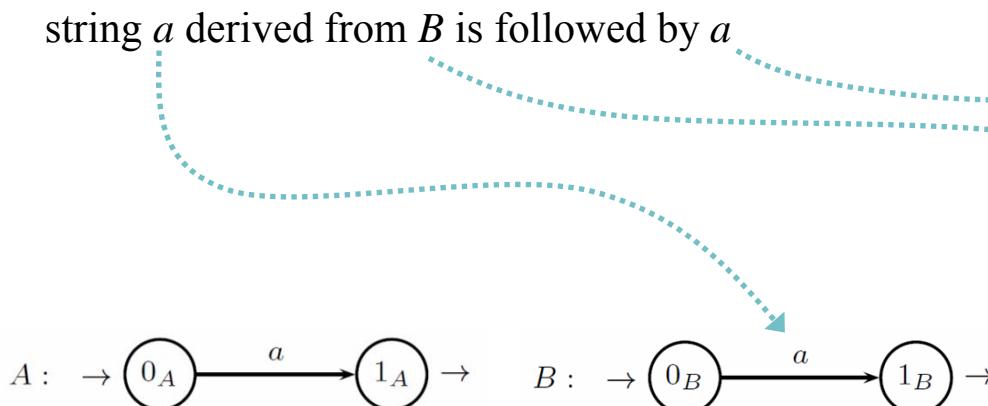
lookahead = set of chars that can follow the string derived from the current nonterminal

a char in the lookahead will be the input when the final state of the current machine will be reached
its presence in the input indicates that the reduction is in order

Now every pilot m-state contains a set of ***items*** (also called ***candidates*** in the textbook)

item = pair: $\langle \text{state}, \text{lookahead} \rangle$

Example: derivation $S \Rightarrow^* Ba \Rightarrow aa$



Solution: we incorporate the **lookahead** in the pilot automaton m-state

as a consequence the number of m-states increases: the automaton executes a “subtler” analysis

lookahead = set of chars that can follow the string derived from the current nonterminal

a char in the lookahead will be the input when the final state of the current machine will be reached
its presence in the input indicates that the reduction is in order

Now every pilot m-state contains a set of **items** (also called **candidates** in the textbook)

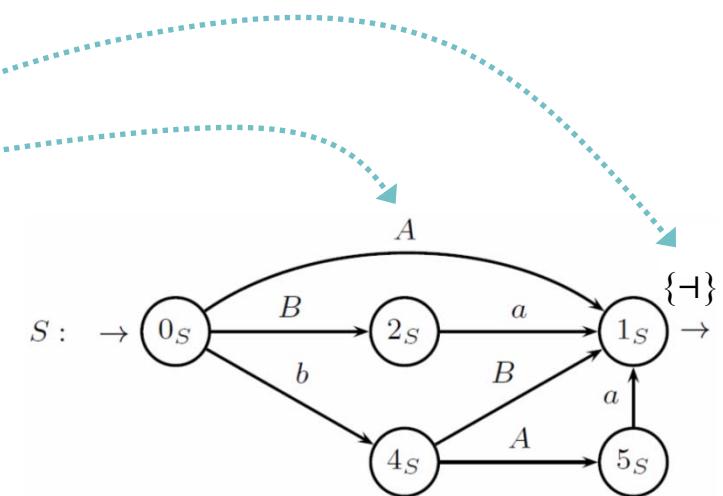
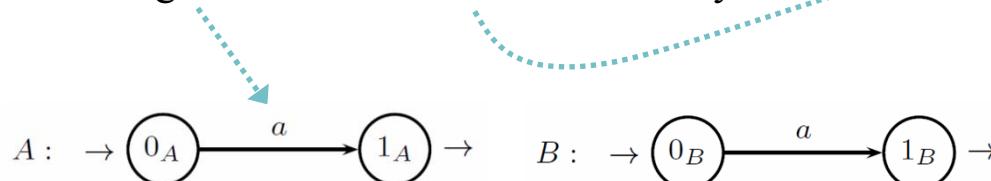
item = pair: $\langle \text{state}, \text{lookahead} \rangle$

Example: derivation $S \Rightarrow B a \Rightarrow aa$

string a derived from B is followed by a

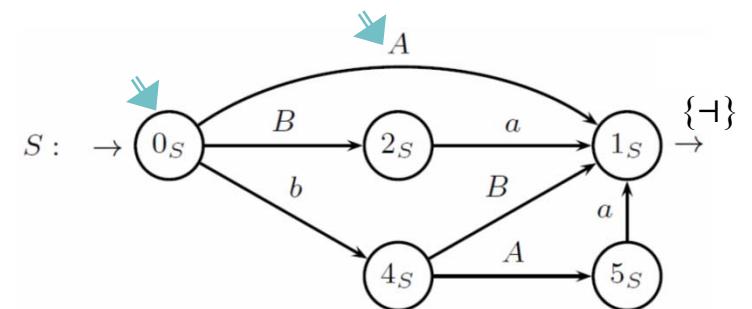
Example: derivation $S \Rightarrow A \Rightarrow a$

string a derived from A is followed by \vdash



Construction of the pilot automaton

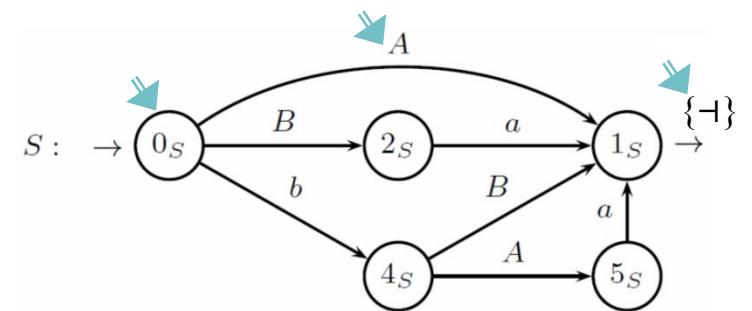
$$A : \rightarrow (0_A) \xrightarrow{a} (1_A) \rightarrow \quad B : \rightarrow (0_B) \xrightarrow{a} (1_B) \rightarrow$$



Construction of the pilot automaton

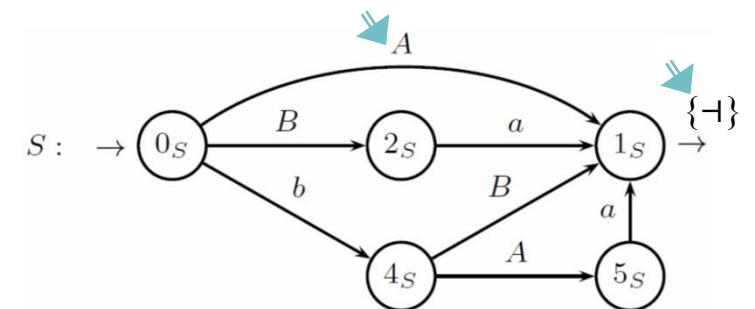
$$A : \rightarrow 0_A \xrightarrow{a} 1_A \rightarrow \quad B : \rightarrow 0_B \xrightarrow{a} 1_B \rightarrow$$

$\rightarrow 0_S \dashv$



Construction of the pilot automaton

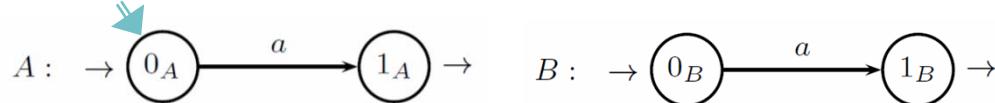
$$A : \rightarrow 0_A \xrightarrow{a} 1_A \rightarrow \quad B : \rightarrow 0_B \xrightarrow{a} 1_B \rightarrow$$



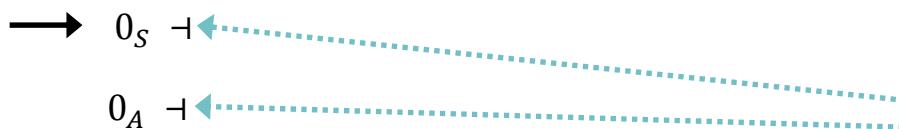
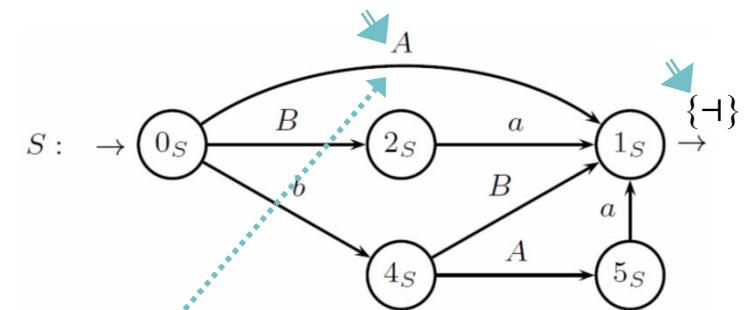
$\rightarrow 0_S \dashv$

$0_A \dashv$

Construction of the pilot automaton



Usual construction but we take into account what follows (lookahead)



The lookahead is the same because in M_S nothing follows the nonterm. A

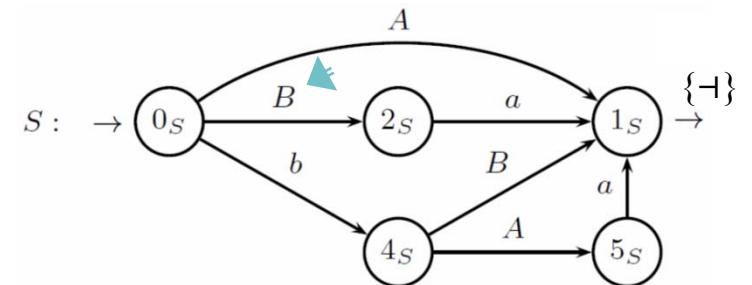
Construction of the pilot automaton

$$A : \rightarrow 0_A \xrightarrow{a} 1_A \rightarrow \quad B : \rightarrow 0_B \xrightarrow{a} 1_B \rightarrow$$

$\rightarrow 0_S \dashv$

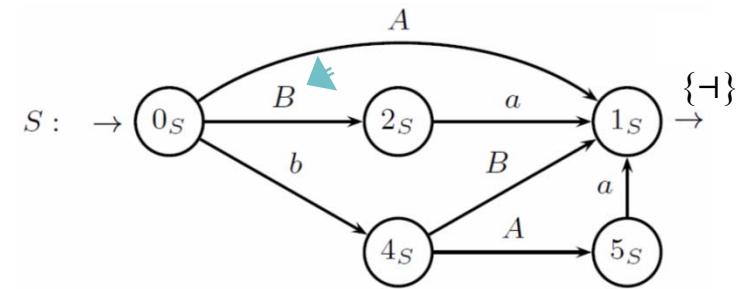
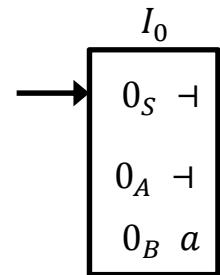
$0_A \dashv$

$0_B \ a$



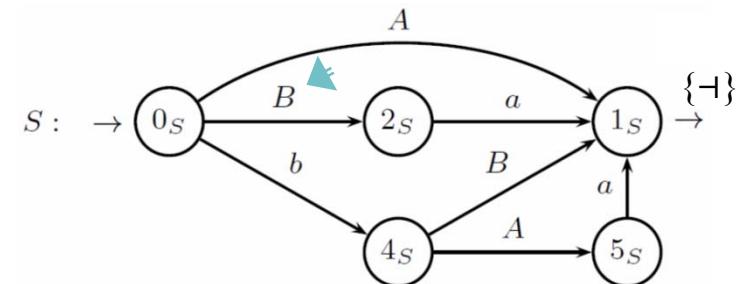
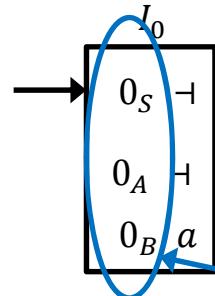
Construction of the pilot automaton

$$A : \rightarrow 0_A \xrightarrow{a} 1_A \rightarrow \quad B : \rightarrow 0_B \xrightarrow{a} 1_B \rightarrow$$



Construction of the pilot automaton

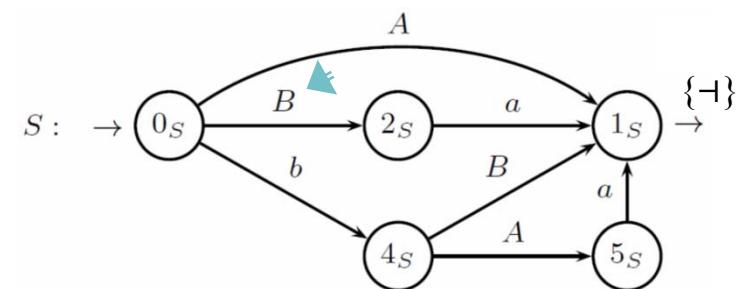
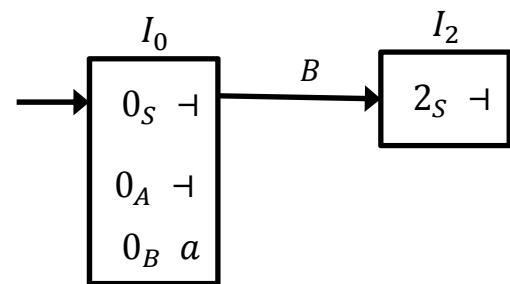
$$A : \rightarrow 0_A \xrightarrow{a} 1_A \rightarrow \quad B : \rightarrow 0_B \xrightarrow{a} 1_B \rightarrow$$



The set of (machine) states inside an m-state is called **kernel**

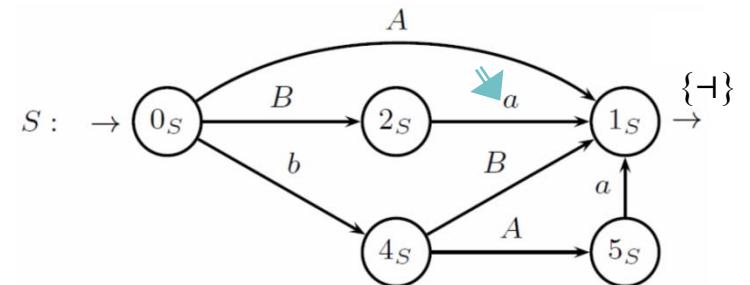
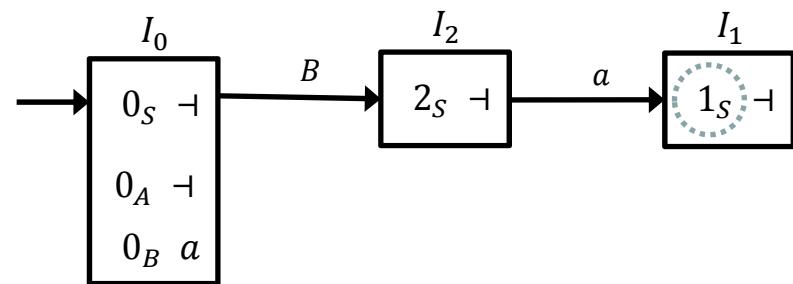
Construction of the pilot automaton

$$A : \rightarrow 0_A \xrightarrow{a} 1_A \rightarrow \quad B : \rightarrow 0_B \xrightarrow{a} 1_B \rightarrow$$

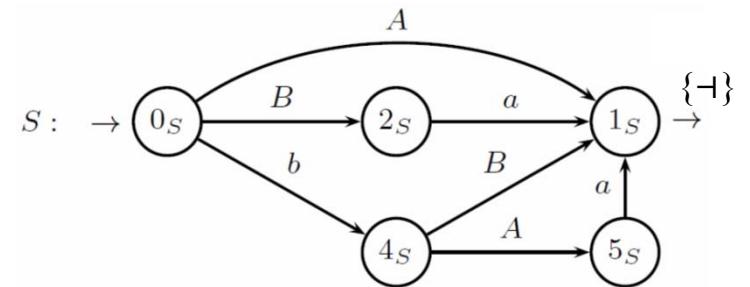
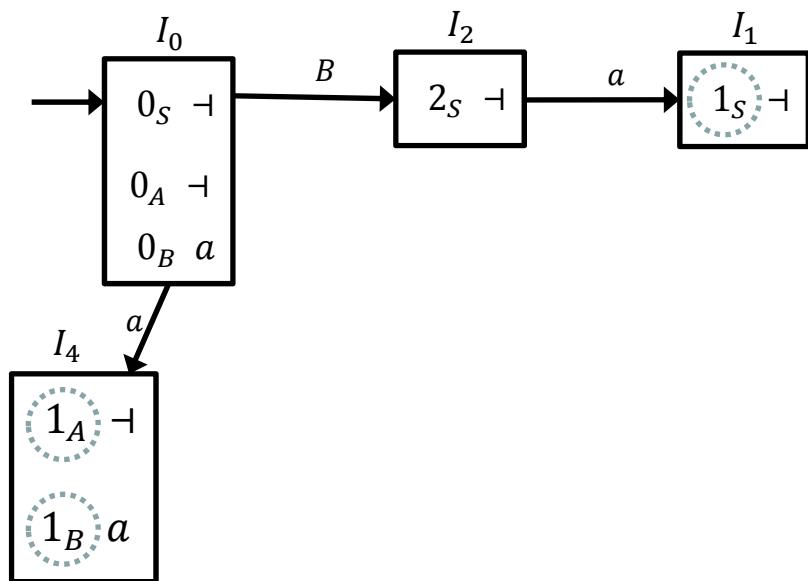
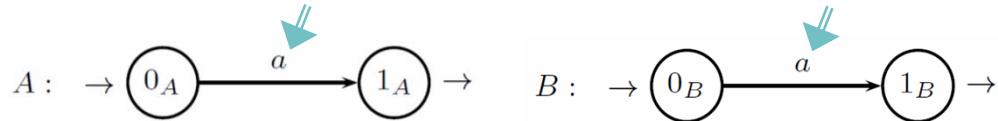


Construction of the pilot automaton

$$A : \rightarrow 0_A \xrightarrow{a} 1_A \rightarrow \quad B : \rightarrow 0_B \xrightarrow{a} 1_B \rightarrow$$



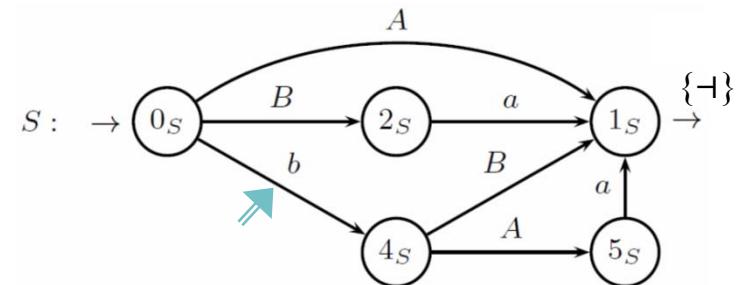
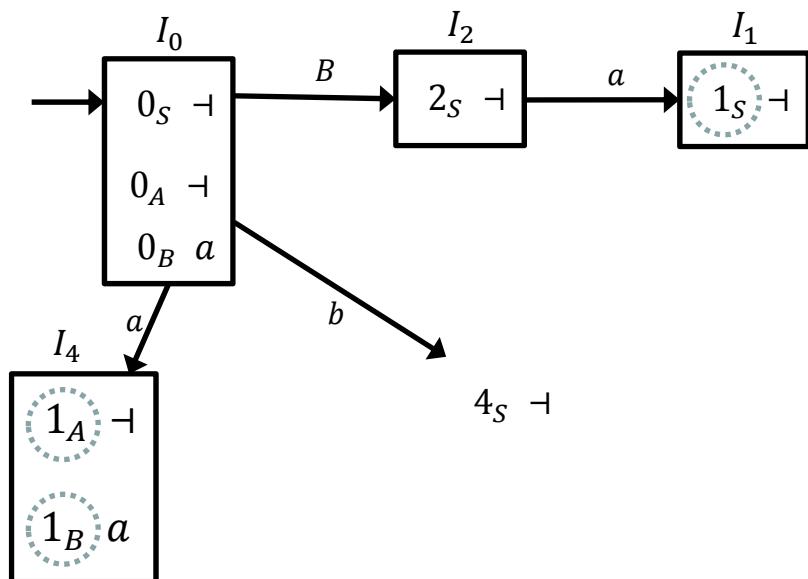
Construction of the pilot automaton



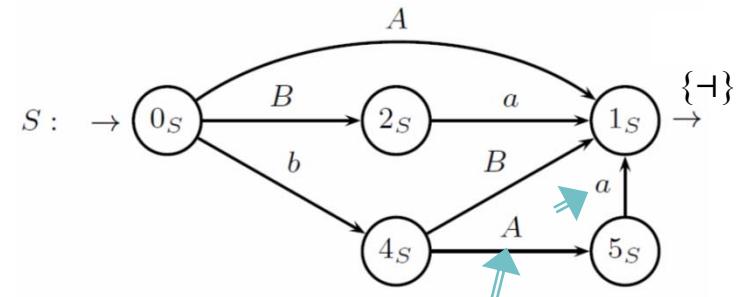
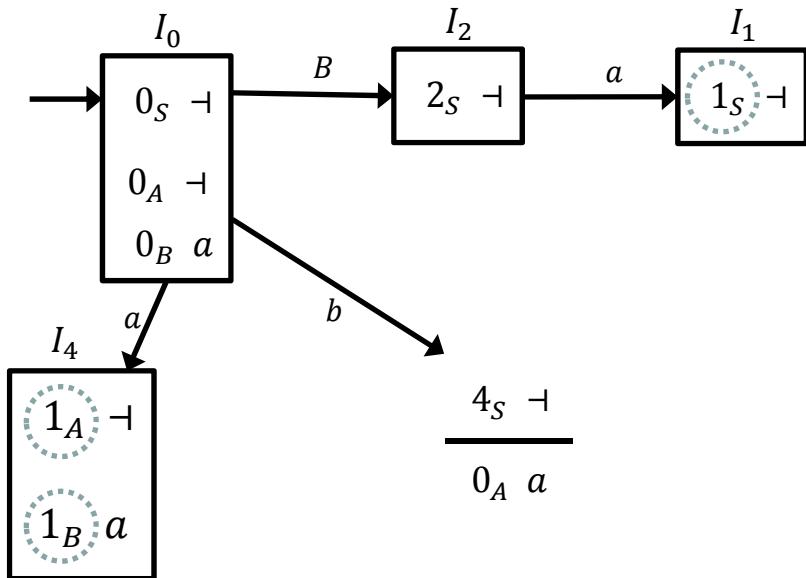
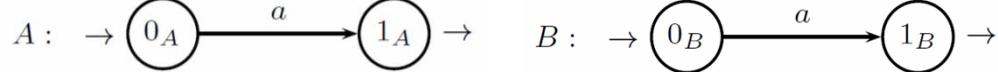
lookahead is what we read after the entire machine so its inherited

Construction of the pilot automaton

$$A : \rightarrow 0_A \xrightarrow{a} 1_A \rightarrow \quad B : \rightarrow 0_B \xrightarrow{a} 1_B \rightarrow$$



Construction of the pilot automaton

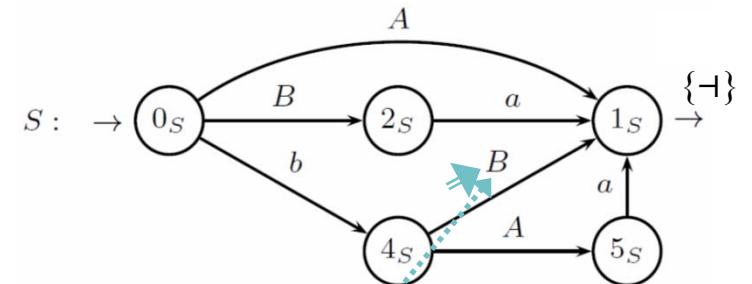
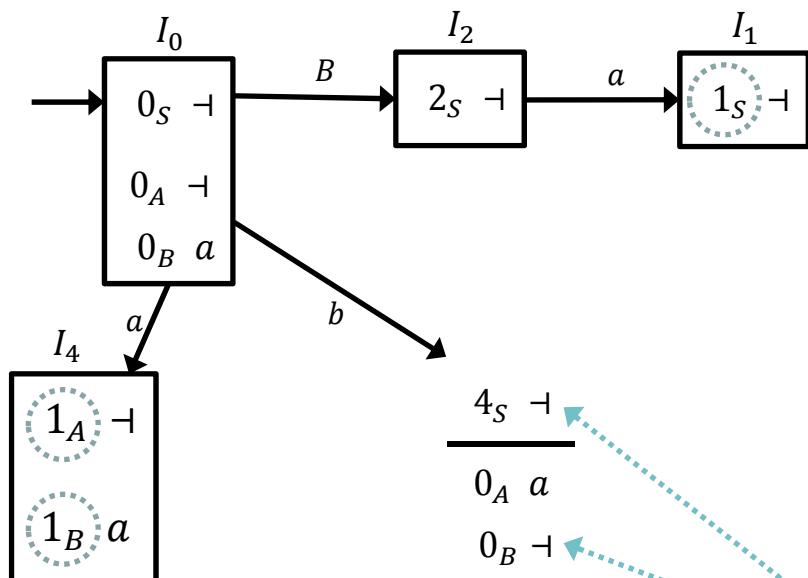


Computing lookahead:

- same for predecessor if we're doing shift move (or nonterminal shift)
- if we're doing a closure we compute the lookahead based on the machine net (what can come after the terminal move)

Construction of the pilot automaton

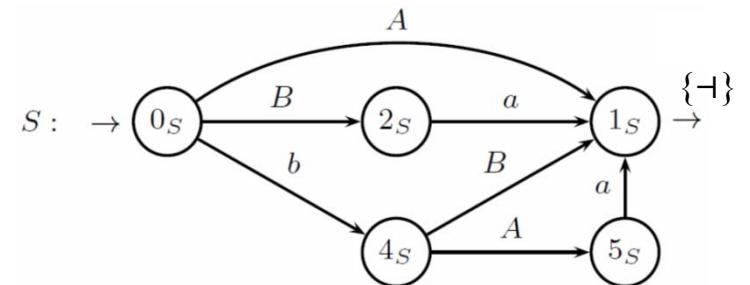
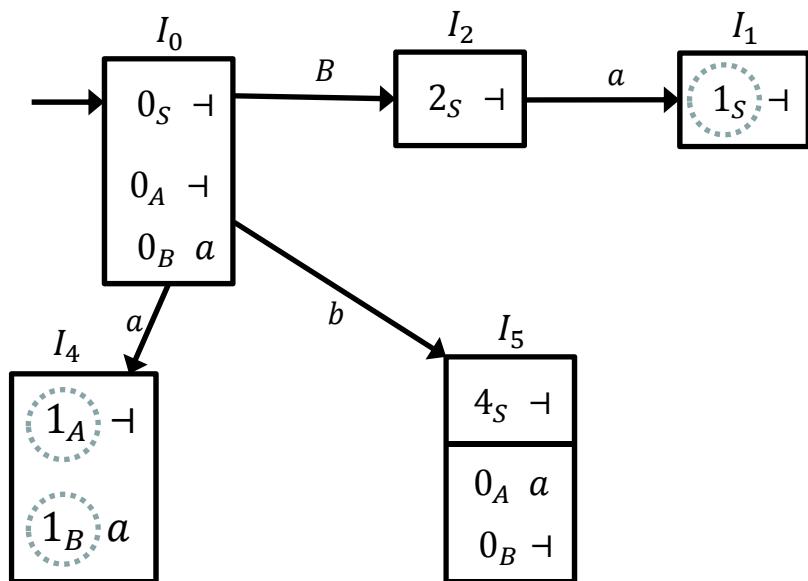
$$A : \rightarrow 0_A \xrightarrow{a} 1_A \rightarrow \quad B : \rightarrow 0_B \xrightarrow{a} 1_B \rightarrow$$



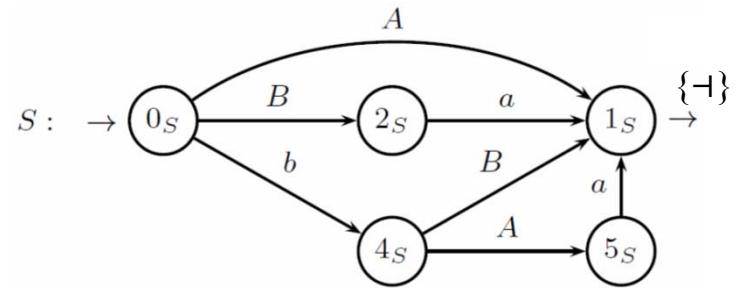
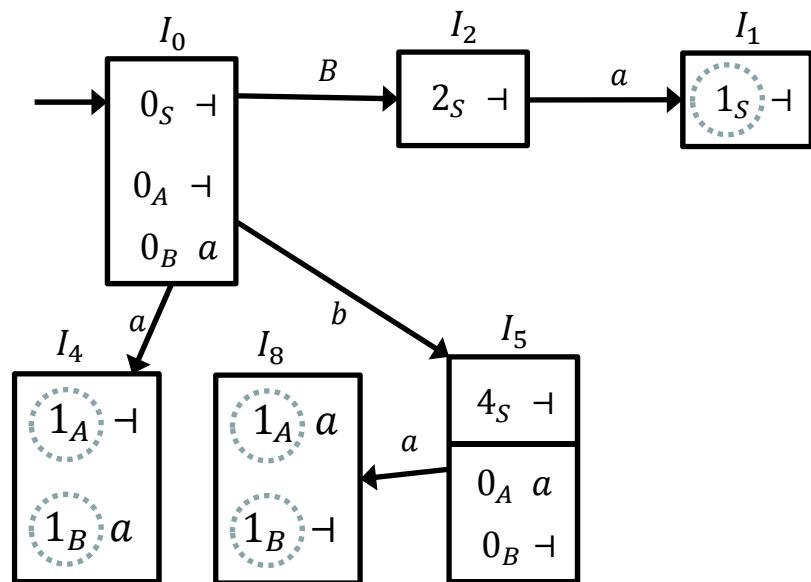
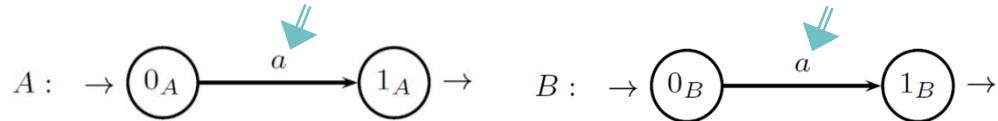
The lookahead is the same because in M_S nothing follows the nonterm. B

Construction of the pilot automaton

$$A : \rightarrow 0_A \xrightarrow{a} 1_A \rightarrow \quad B : \rightarrow 0_B \xrightarrow{a} 1_B \rightarrow$$

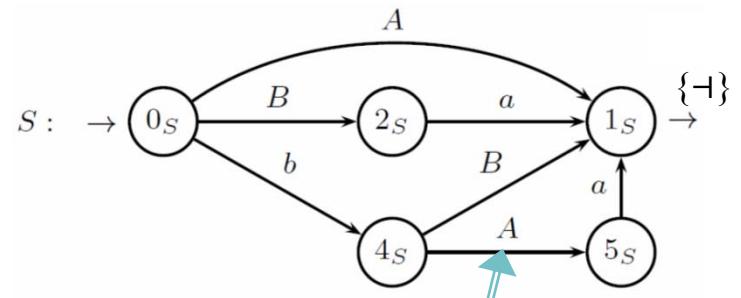
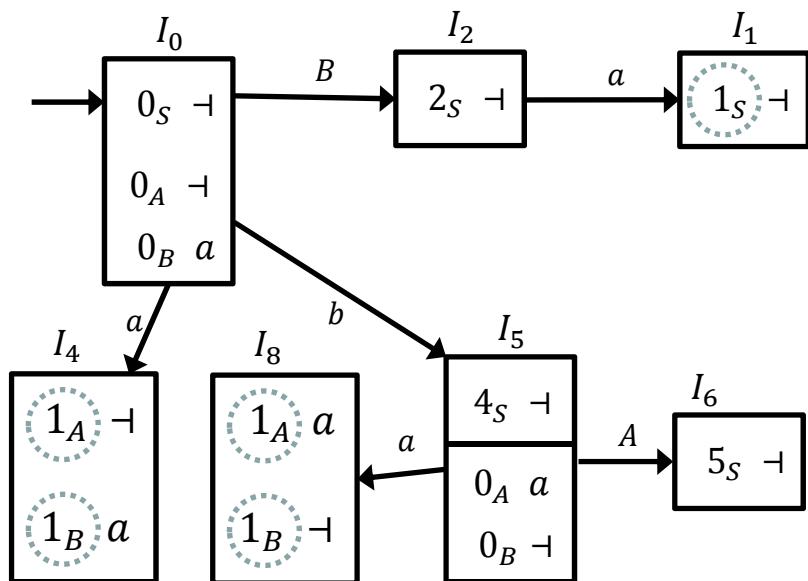


Construction of the pilot automaton



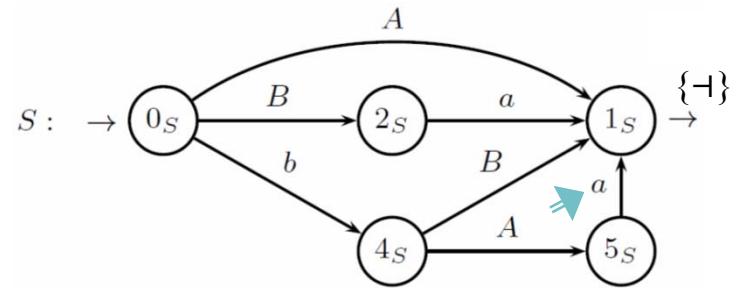
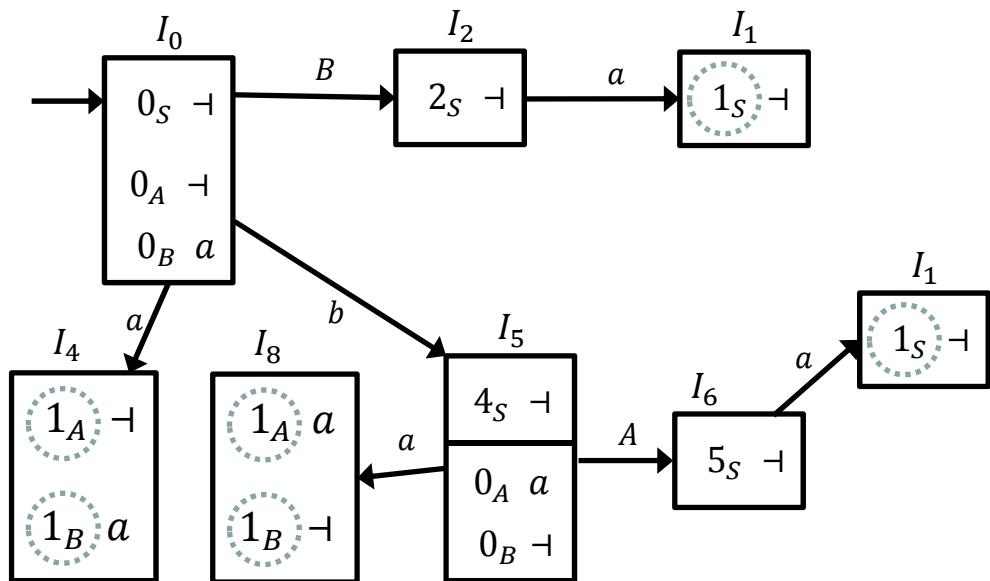
Construction of the pilot automaton

$$A : \rightarrow 0_A \xrightarrow{a} 1_A \rightarrow \quad B : \rightarrow 0_B \xrightarrow{a} 1_B \rightarrow$$

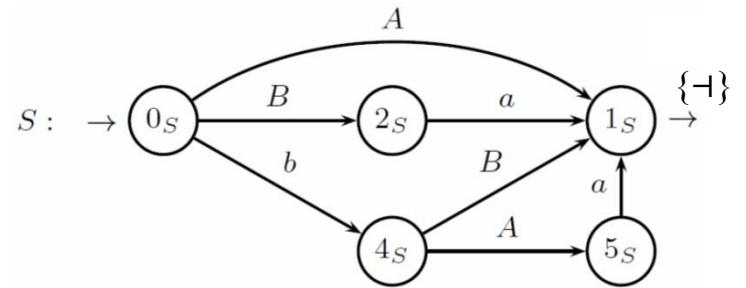
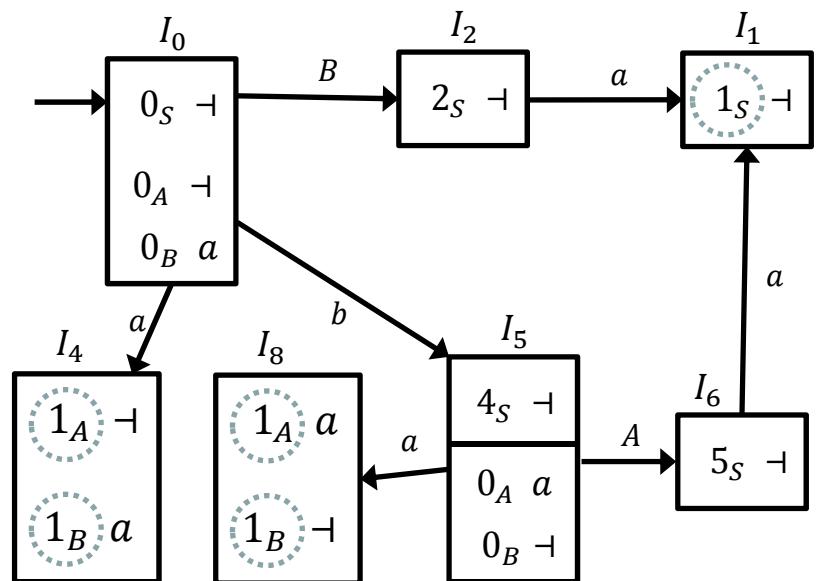


Construction of the pilot automaton

$$A : \rightarrow 0_A \xrightarrow{a} 1_A \rightarrow \quad B : \rightarrow 0_B \xrightarrow{a} 1_B \rightarrow$$

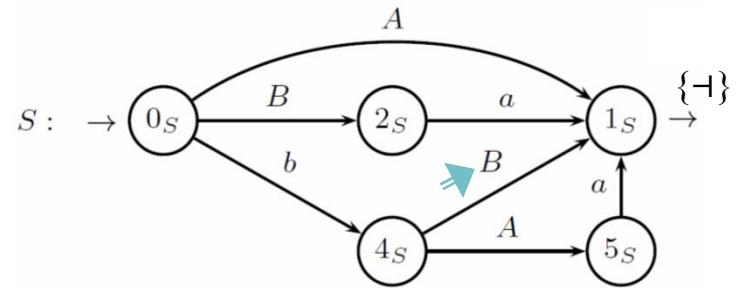
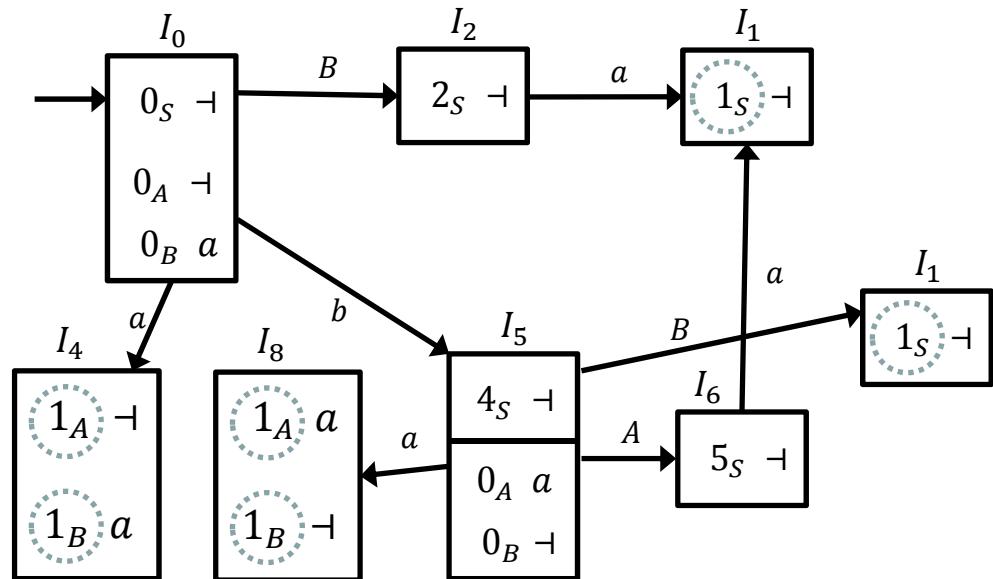


Construction of the pilot automaton



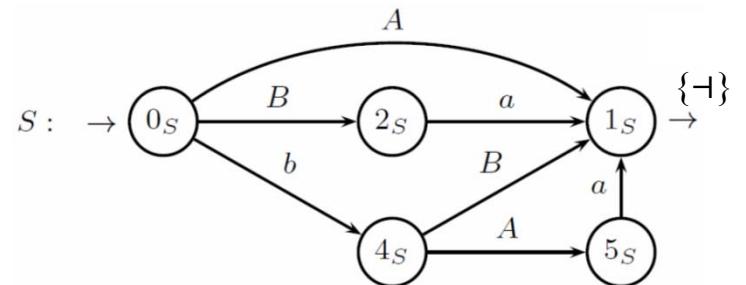
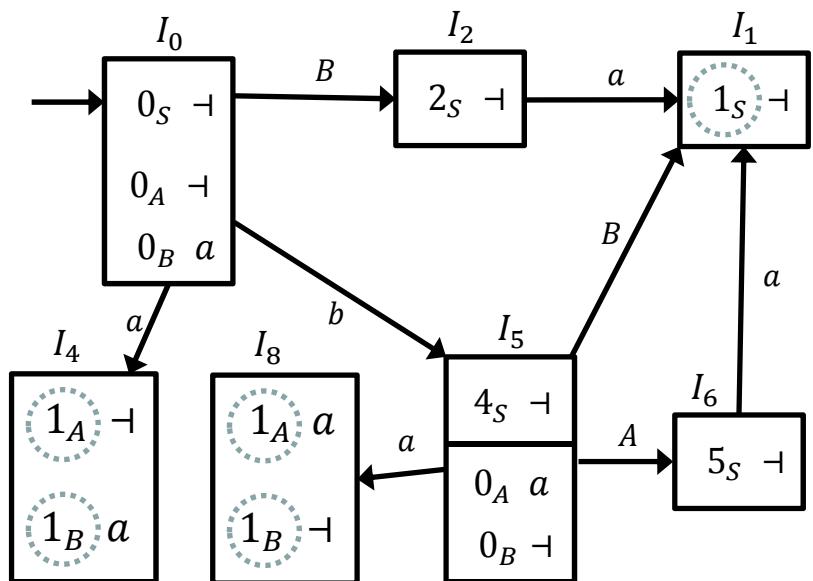
Construction of the pilot automaton

$$A : \rightarrow 0_A \xrightarrow{a} 1_A \rightarrow \quad B : \rightarrow 0_B \xrightarrow{a} 1_B \rightarrow$$



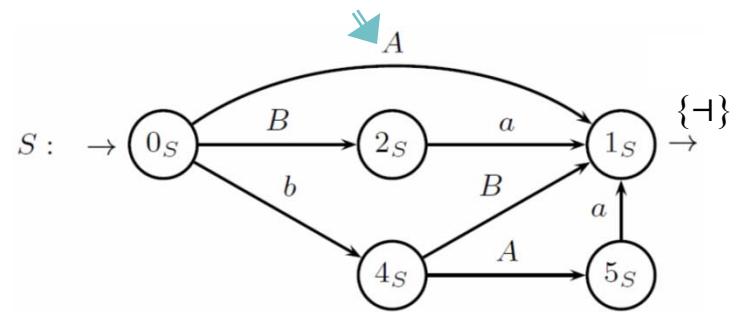
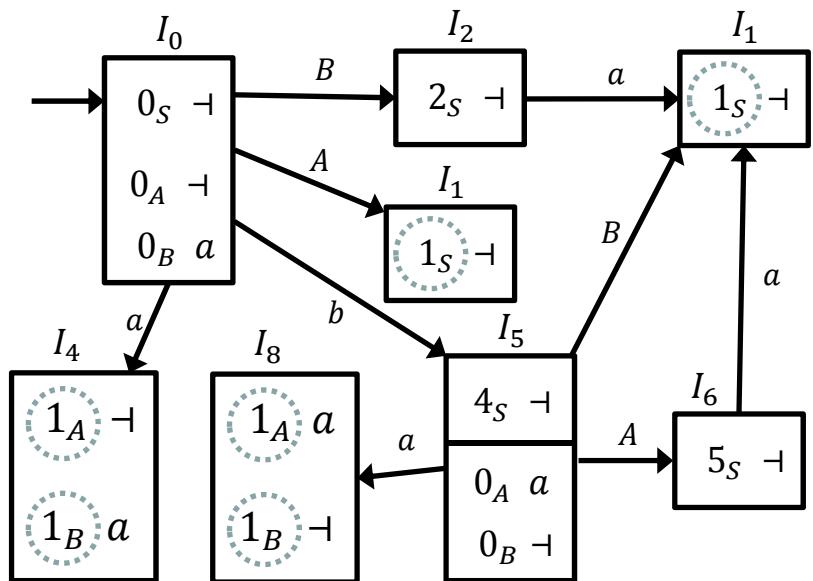
Construction of the pilot automaton

$$A : \rightarrow 0_A \xrightarrow{a} 1_A \rightarrow \quad B : \rightarrow 0_B \xrightarrow{a} 1_B \rightarrow$$



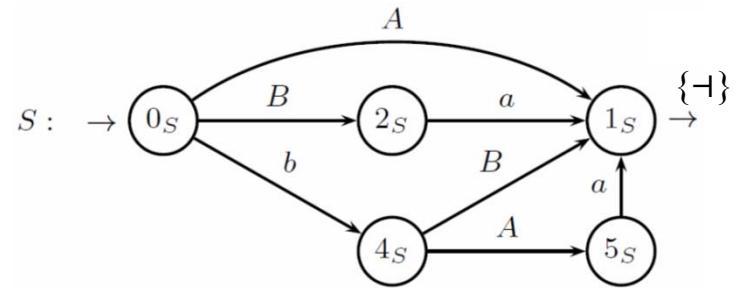
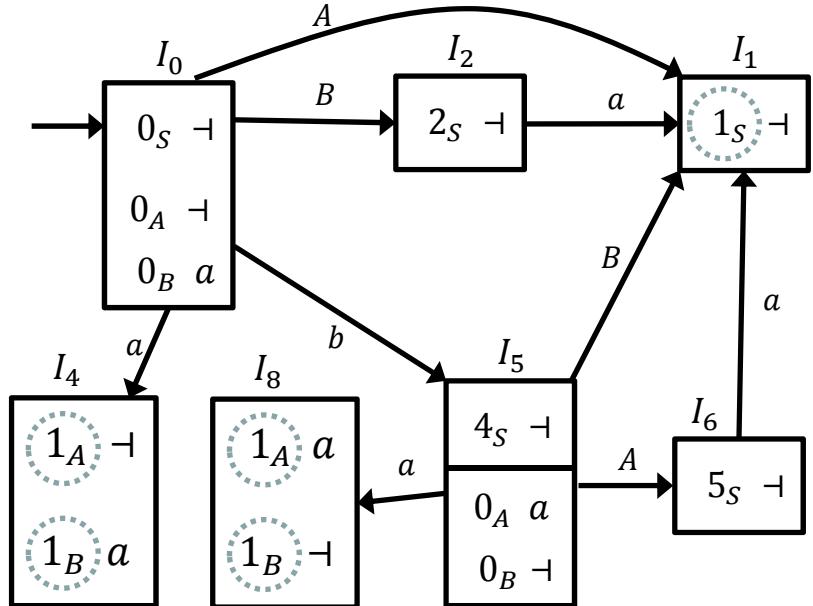
Construction of the pilot automaton

$$A : \rightarrow 0_A \xrightarrow{a} 1_A \rightarrow \quad B : \rightarrow 0_B \xrightarrow{a} 1_B \rightarrow$$



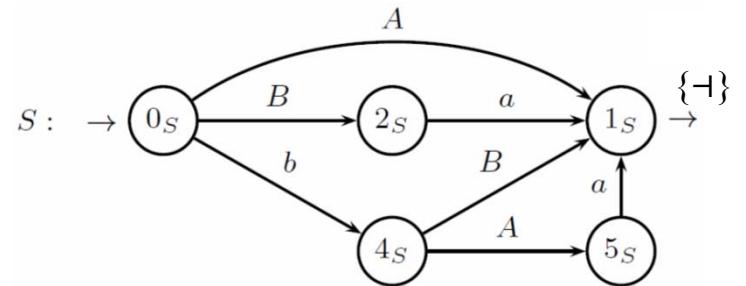
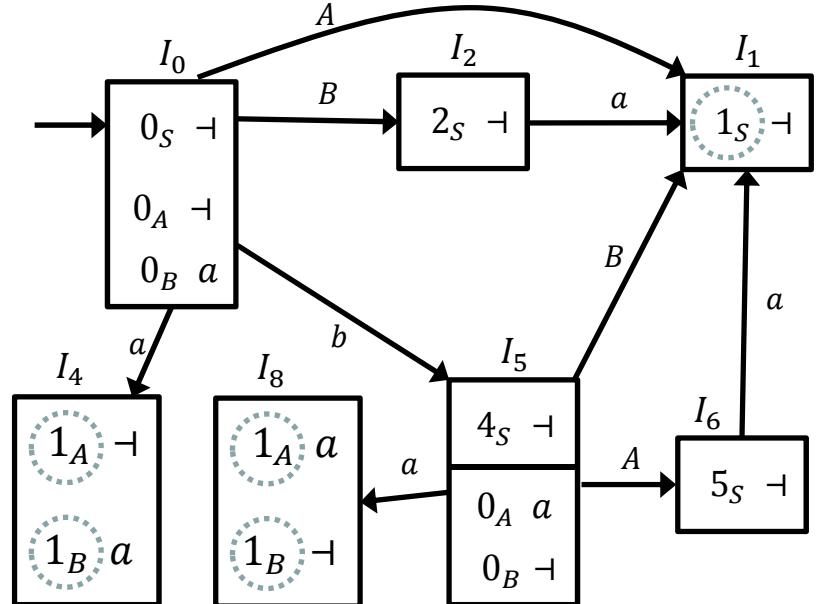
Construction of the pilot automaton

$$A : \rightarrow 0_A \xrightarrow{a} 1_A \rightarrow \quad B : \rightarrow 0_B \xrightarrow{a} 1_B \rightarrow$$



Construction of the pilot automaton

$$A : \rightarrow 0_A \xrightarrow{a} 1_A \rightarrow \quad B : \rightarrow 0_B \xrightarrow{a} 1_B \rightarrow$$

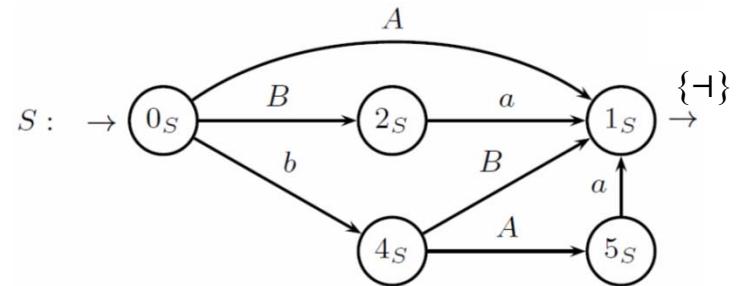
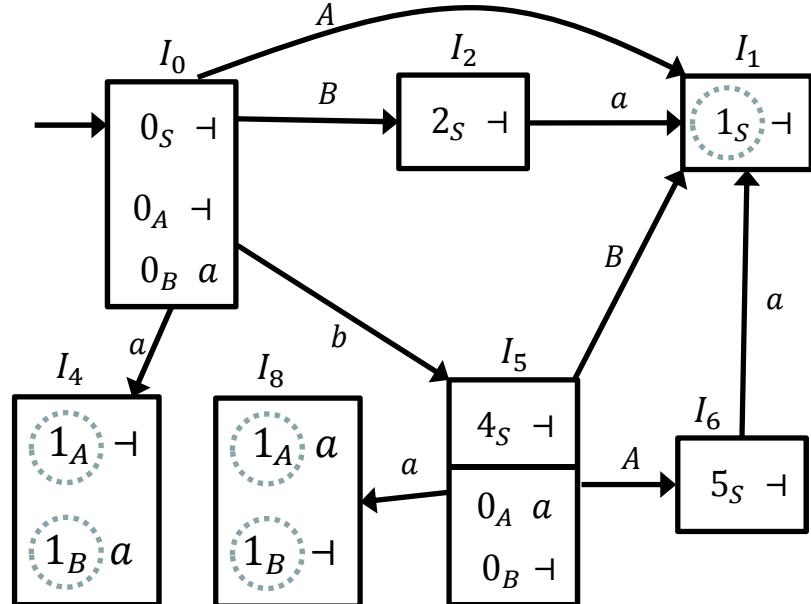


NB: the lookahead changes (i.e., it *may change*) only when a **closure** part is computed

because the closure indicates the start of the analysis of a string derived from a new nonterminal and one must determine what can be expected (in the parsed string) after the string derived from the nonterminal whose initial state is placed in the closure

Construction of the pilot automaton

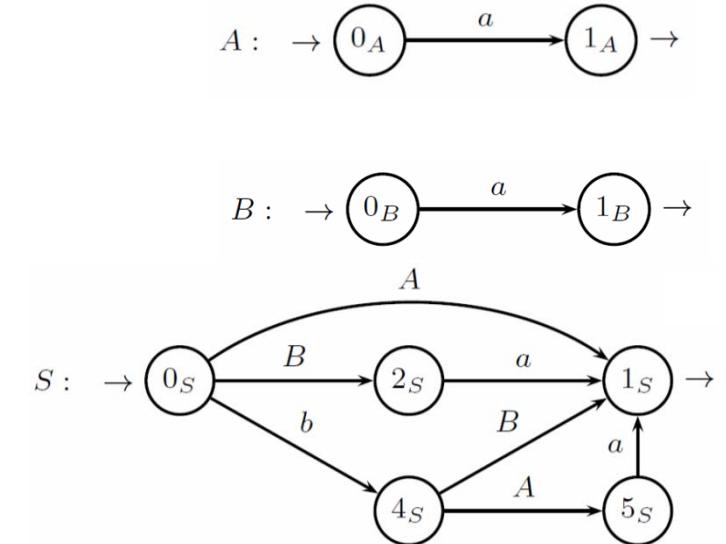
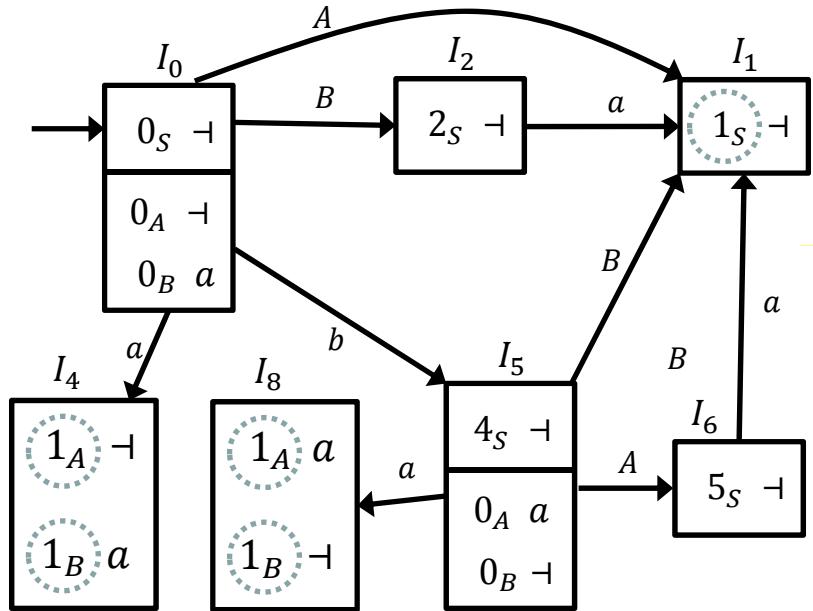
$$A : \rightarrow 0_A \xrightarrow{a} 1_A \rightarrow \quad B : \rightarrow 0_B \xrightarrow{a} 1_B \rightarrow$$

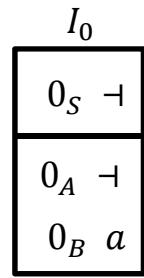
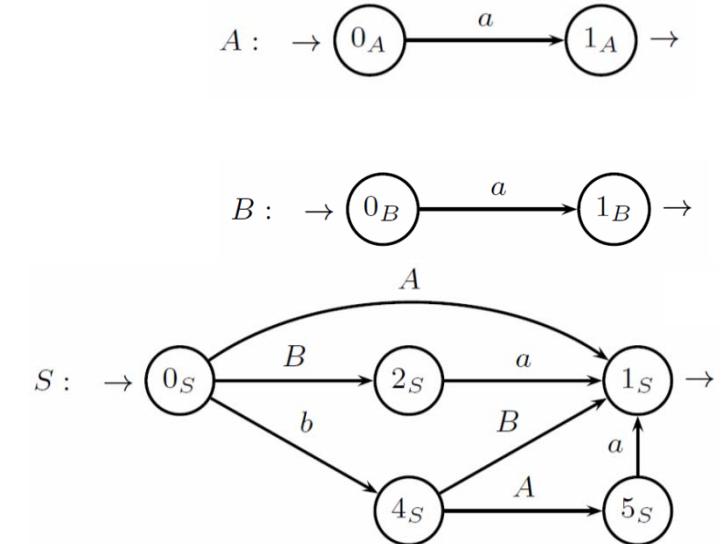
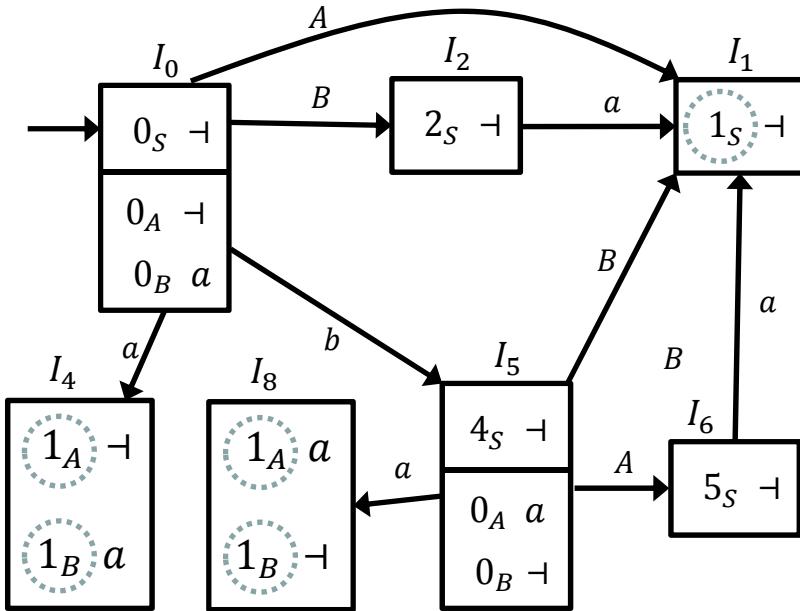


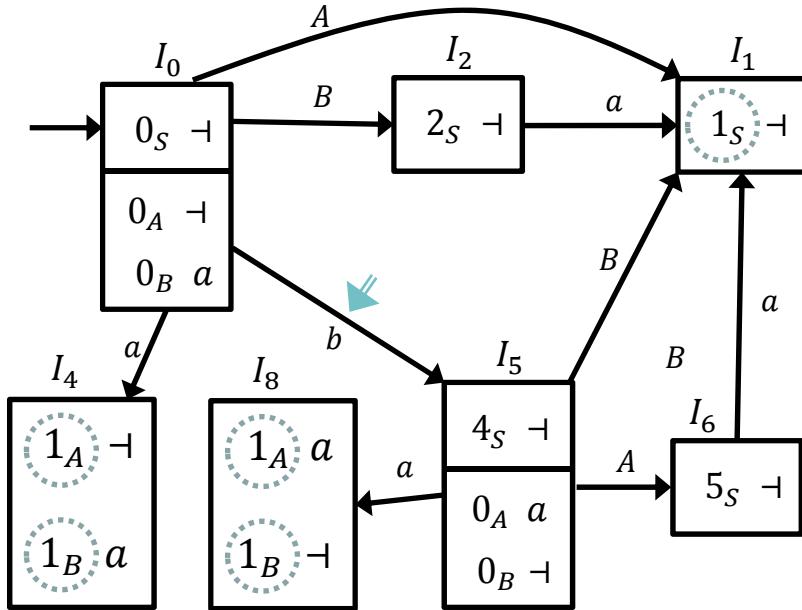
NB: the lookahead changes (i.e., it *may change*) only when a ***closure*** part is computed

because the closure indicates the start of the analysis of a string derived from a new nonterminal and one must determine what can be expected (in the parsed string) after the string derived from the nonterminal whose initial state is placed in the closure

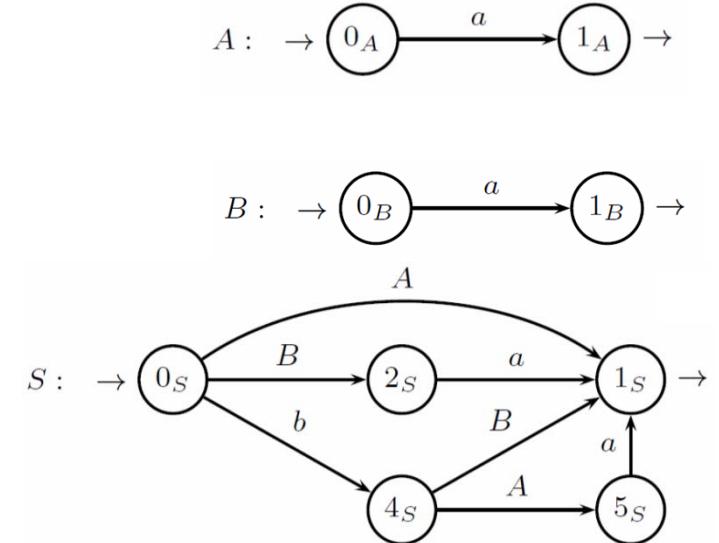
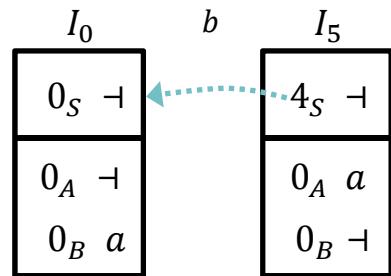
instead, the lookahead does not change when a ***shift*** is considered
(the next state is reached inside the same machine)

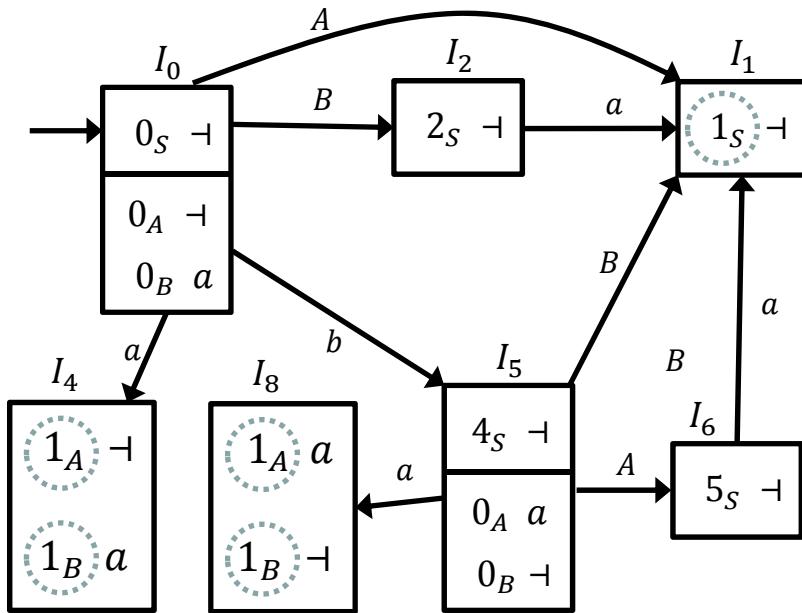




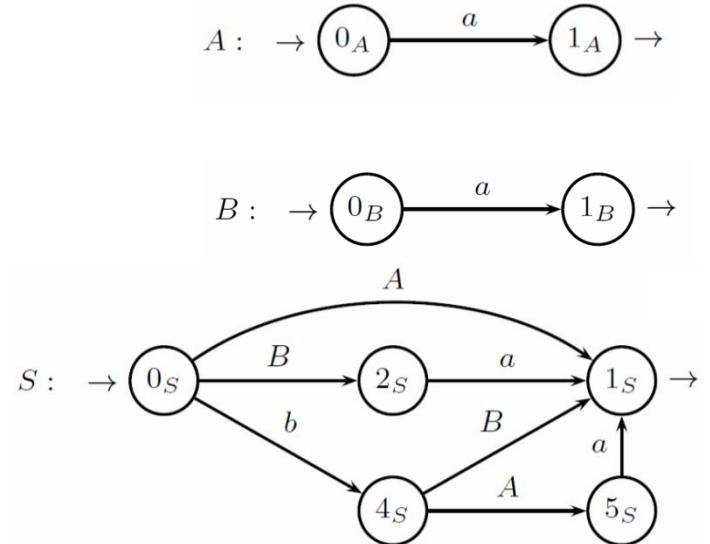
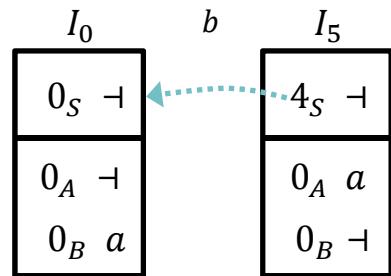


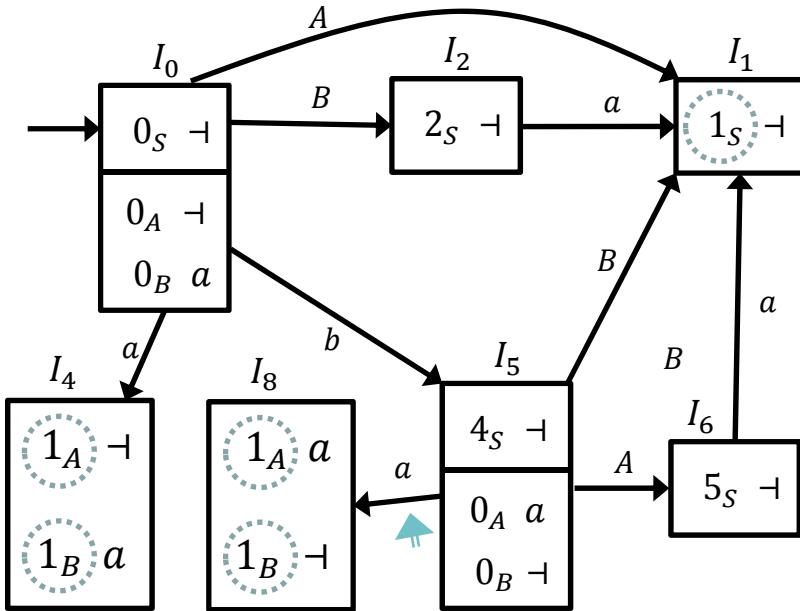
Analysis of string $baa \dashv$



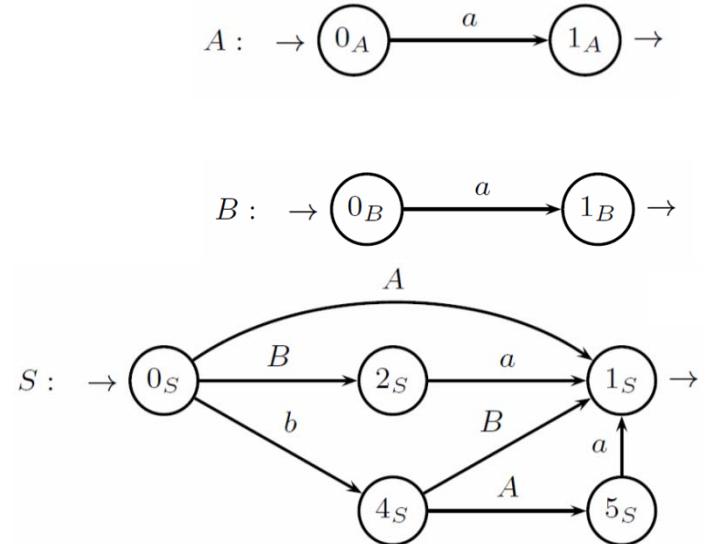
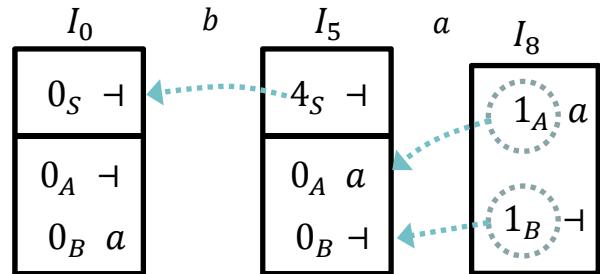


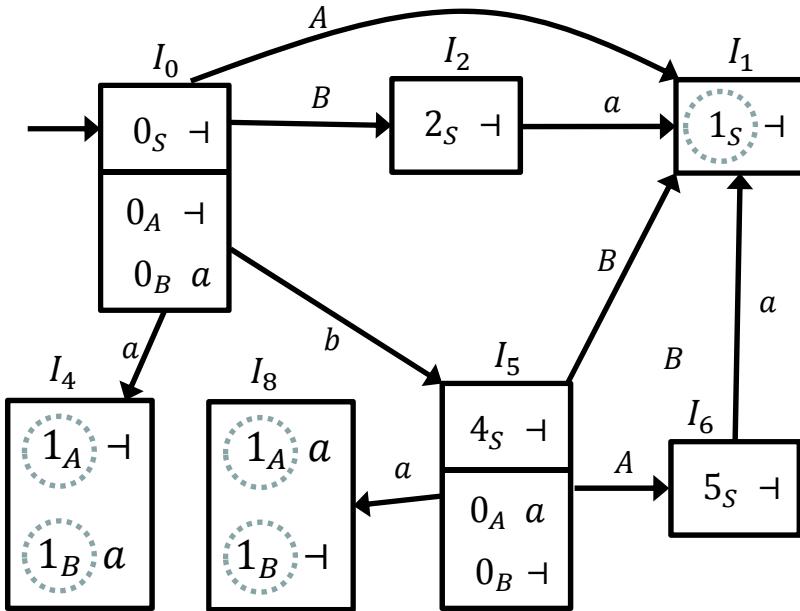
Analysis of string $baa \dashv$



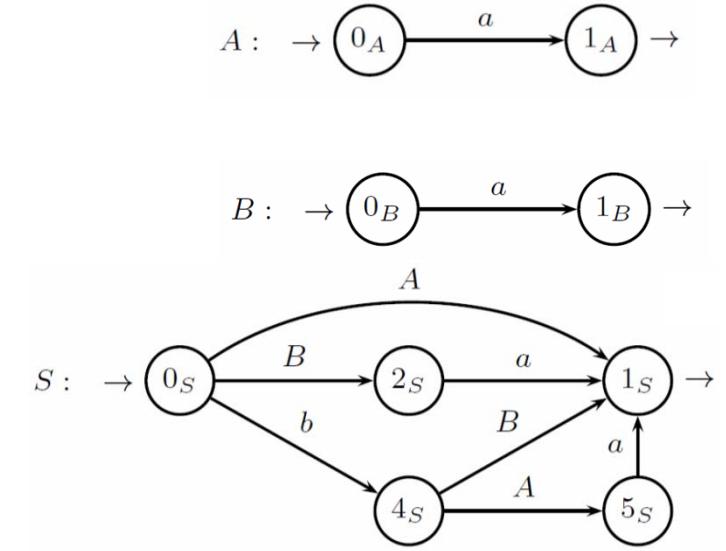
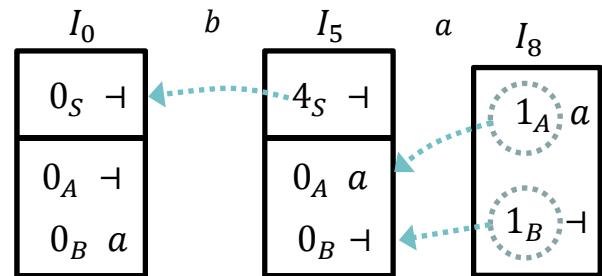


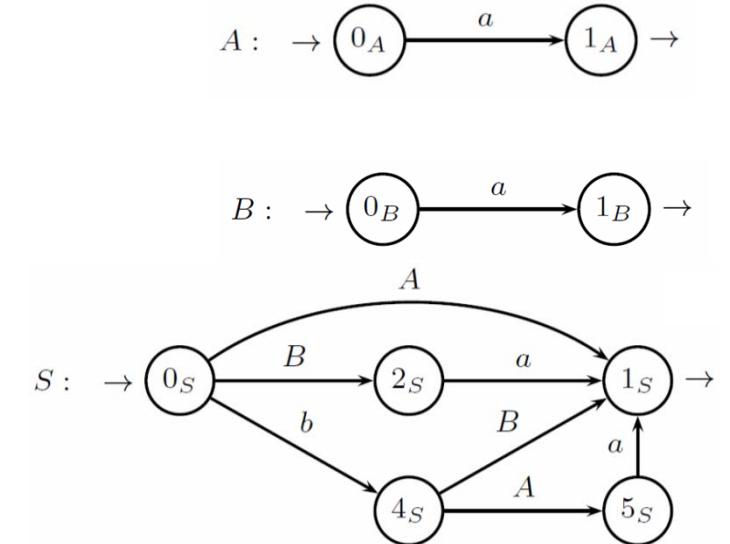
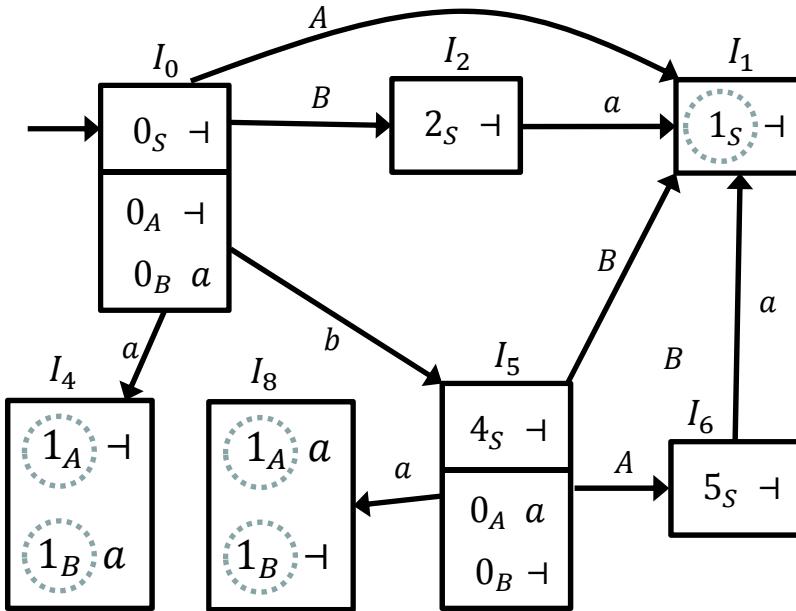
Analysis of string $baa \dashv$





Analysis of string $baa \dashv$

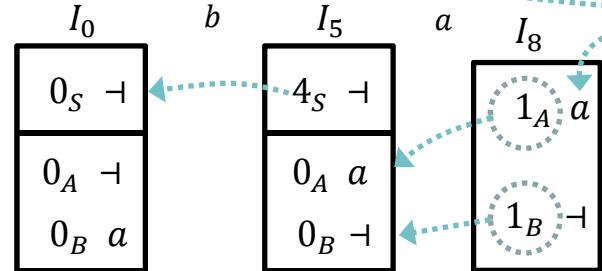


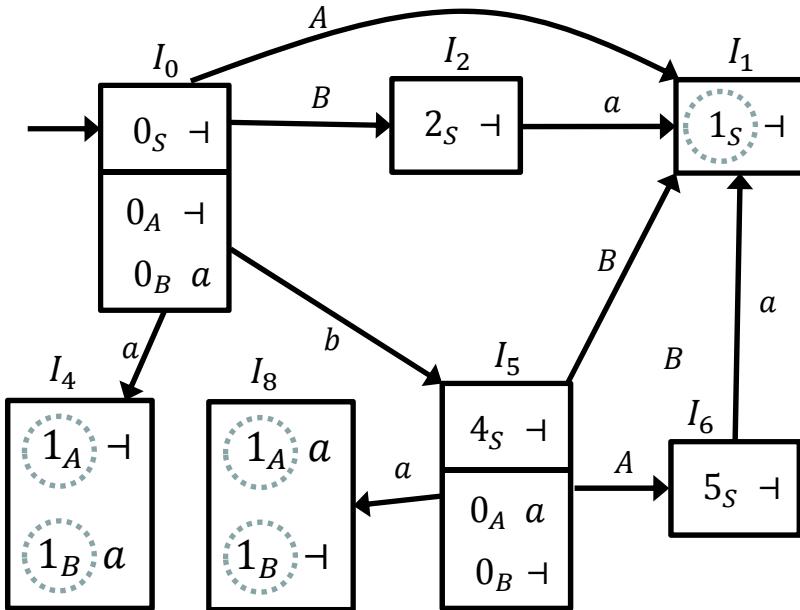


Analysis of string

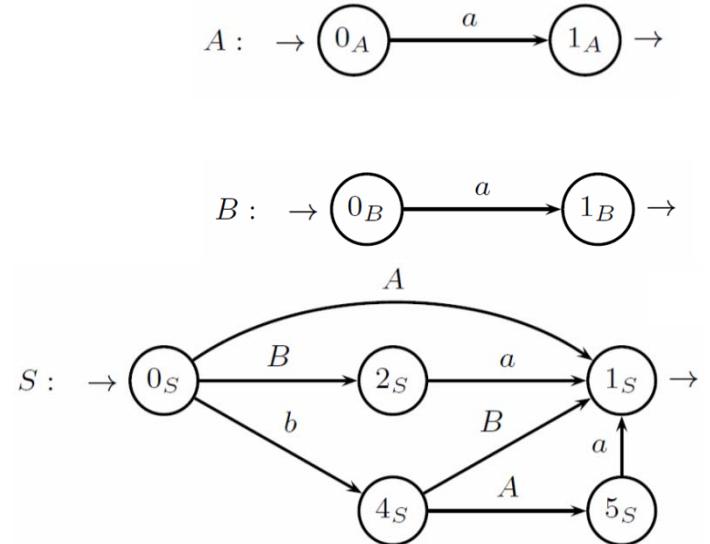
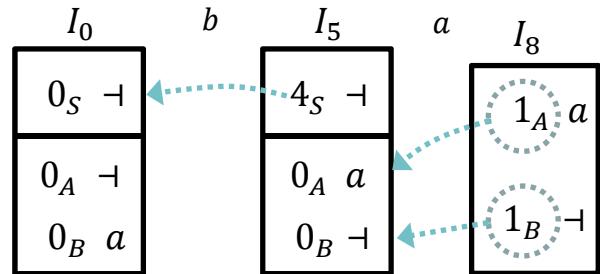
$baa \dashv$

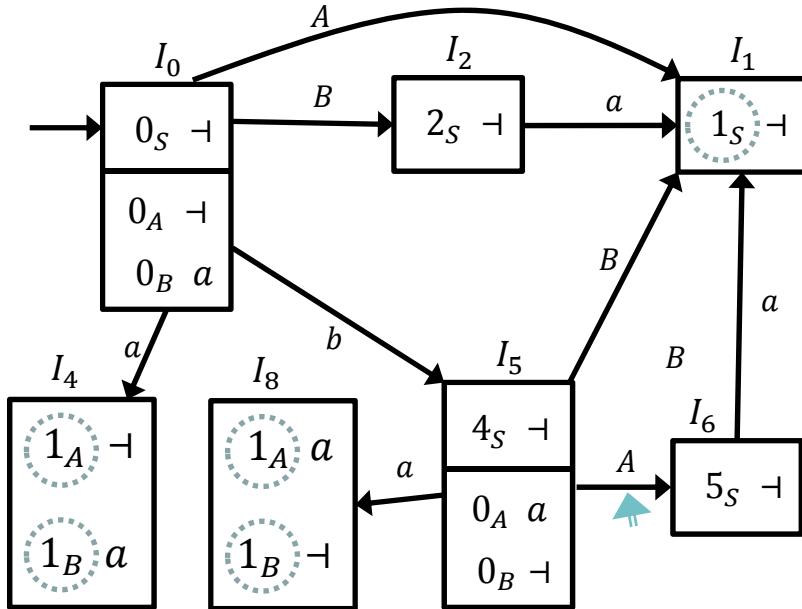
lookahead = input: reduce $a \rightarrow A$



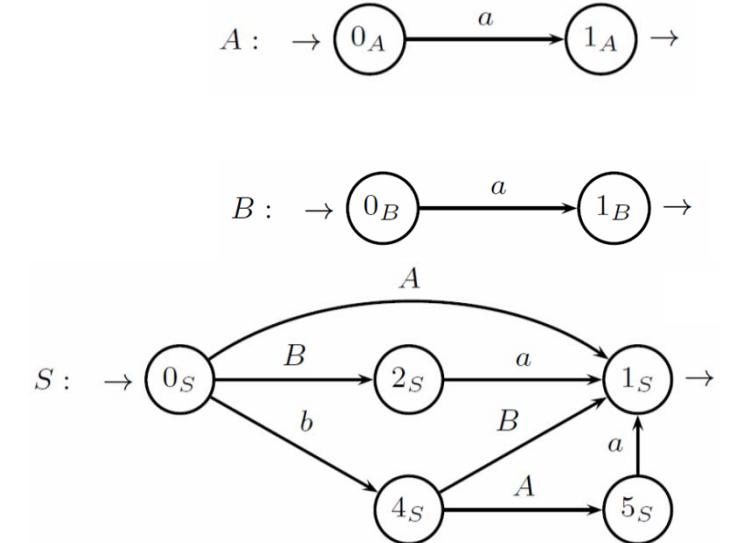


Analysis of string $baa \dashv$

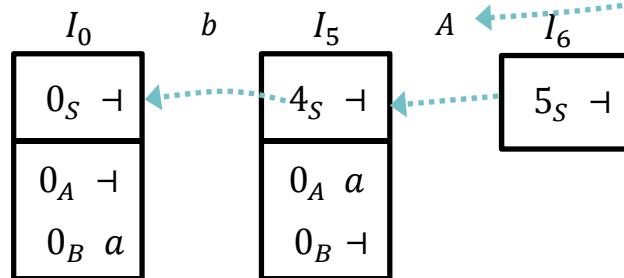


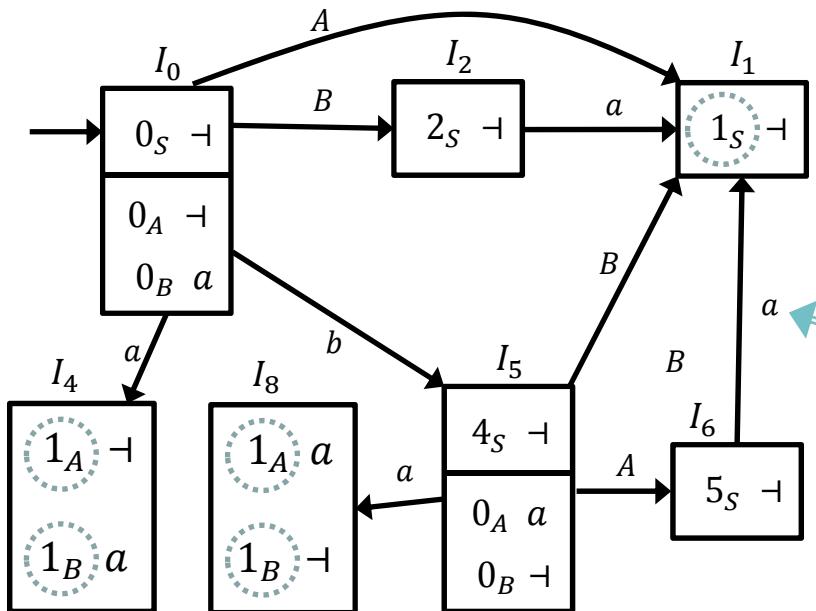


Analysis of string $baa \dashv$

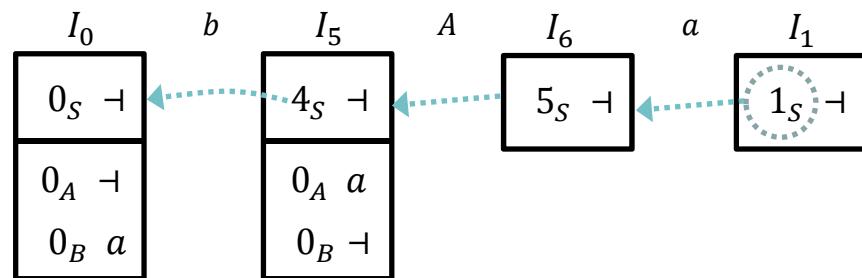
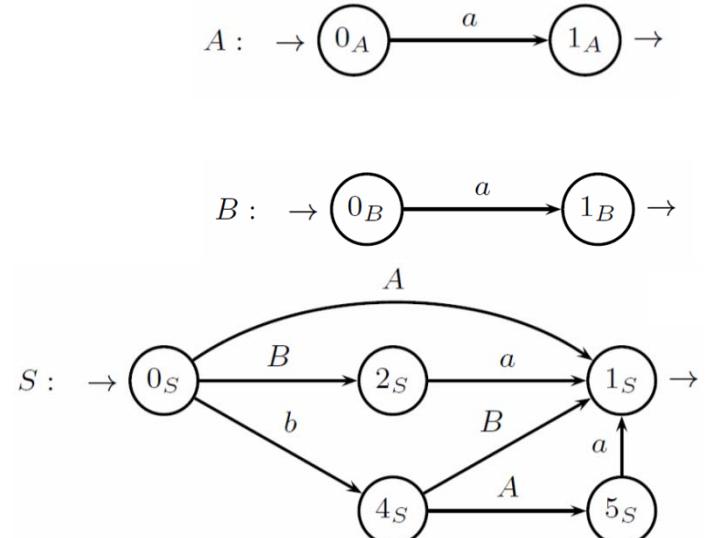


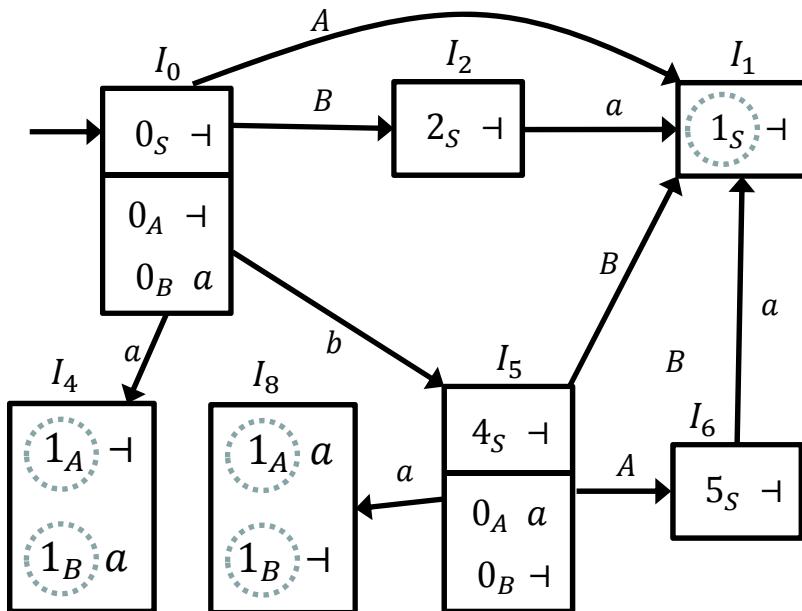
NB: nonterminal shift of A



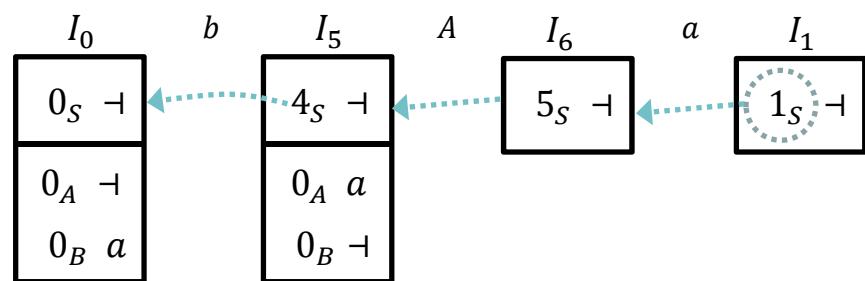
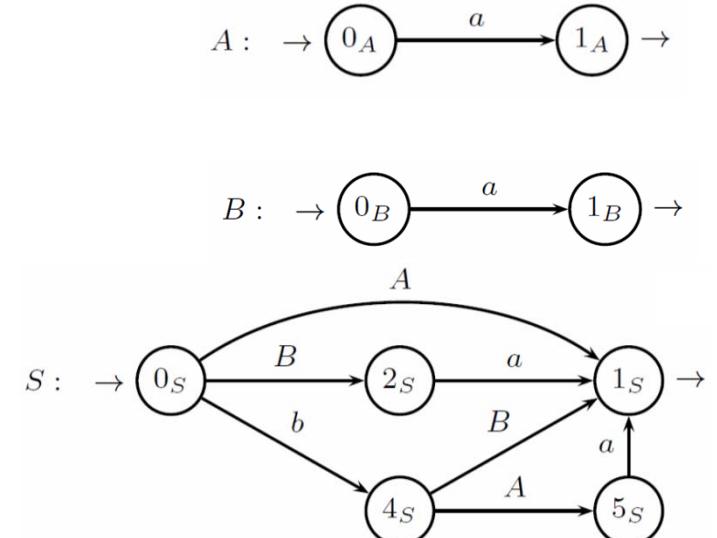


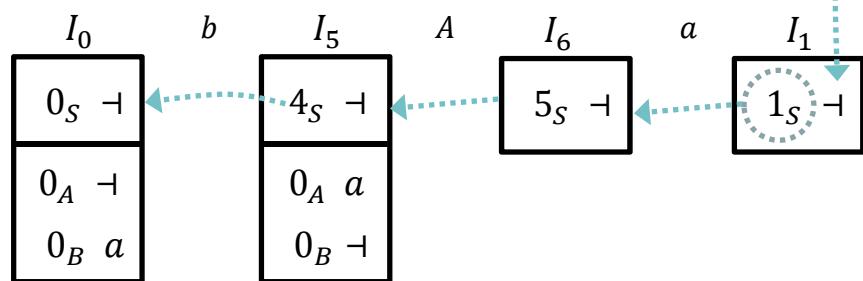
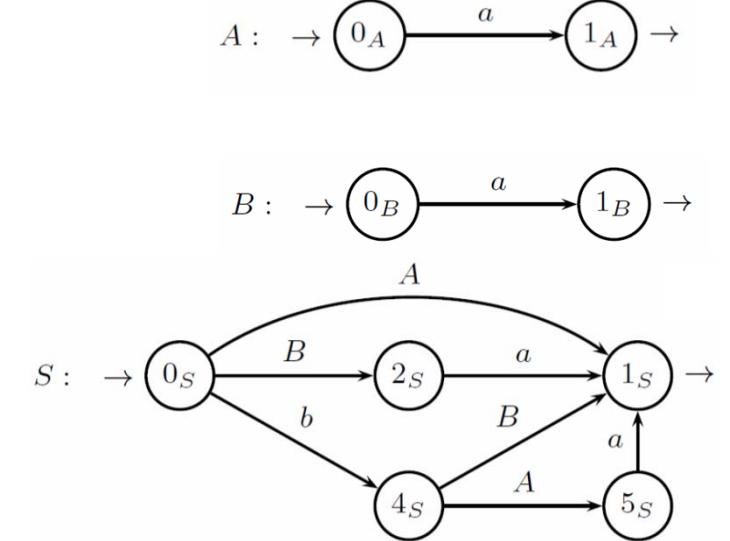
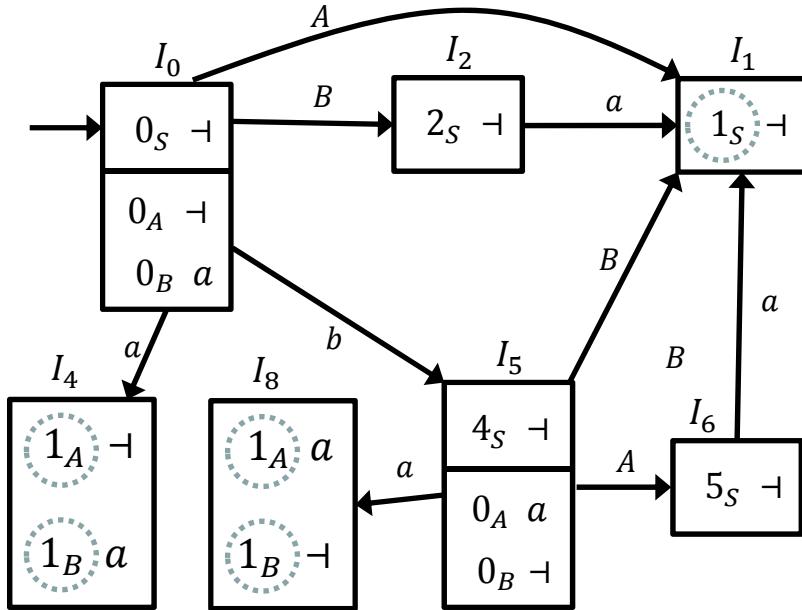
Analysis of string $baa \dashv$



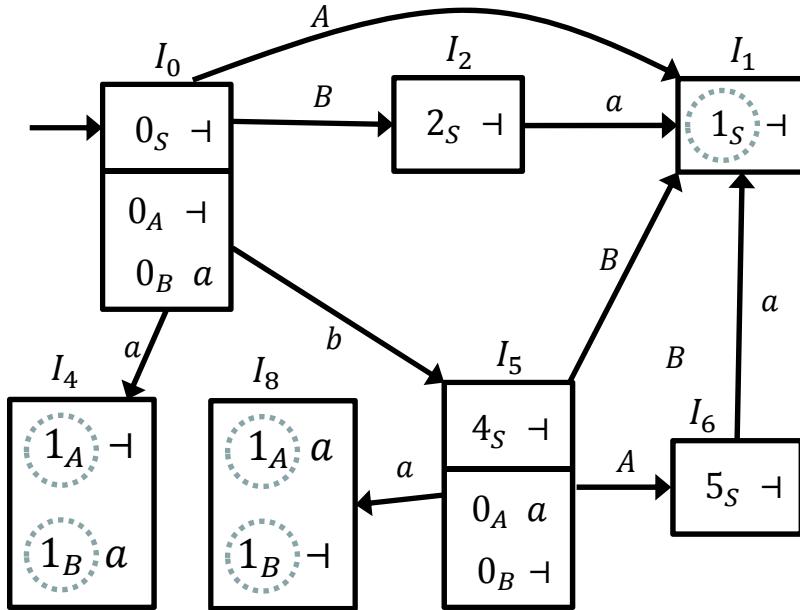


Analysis of string $baa \dashv$

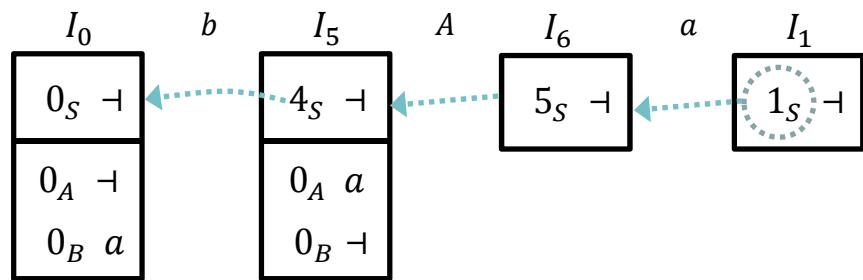
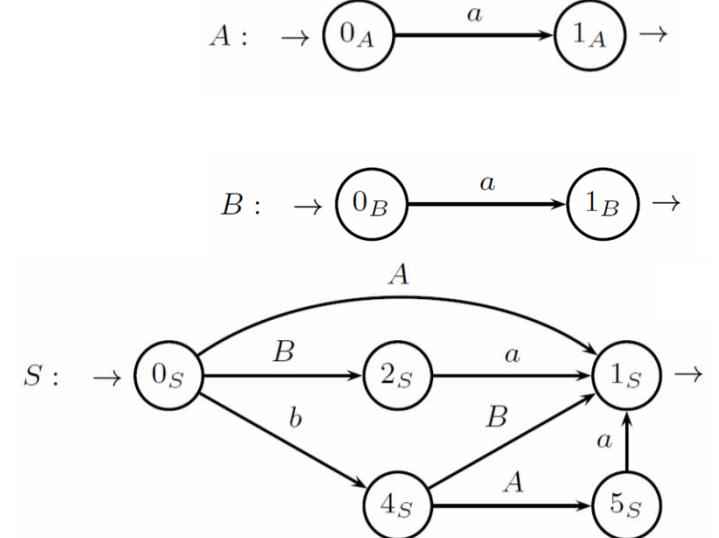


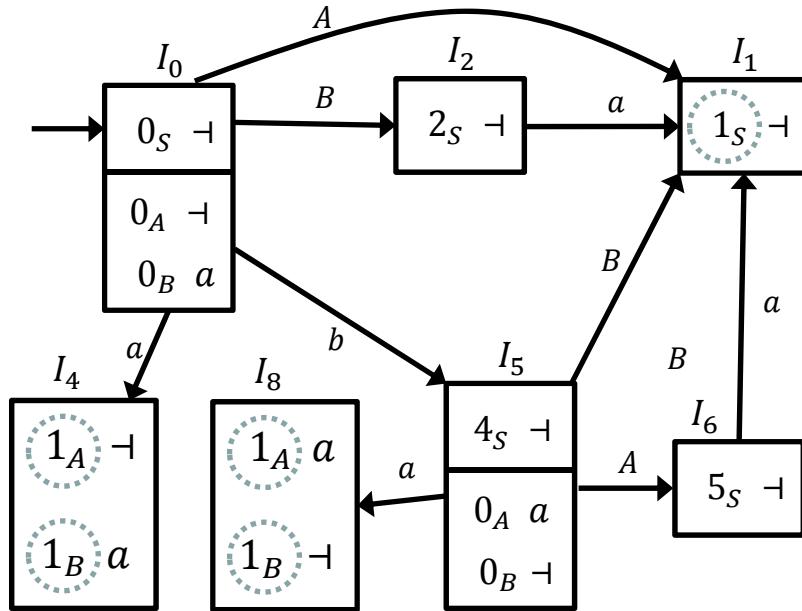


lookahead = input: reduce $bAa \rightarrow S$

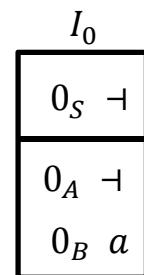
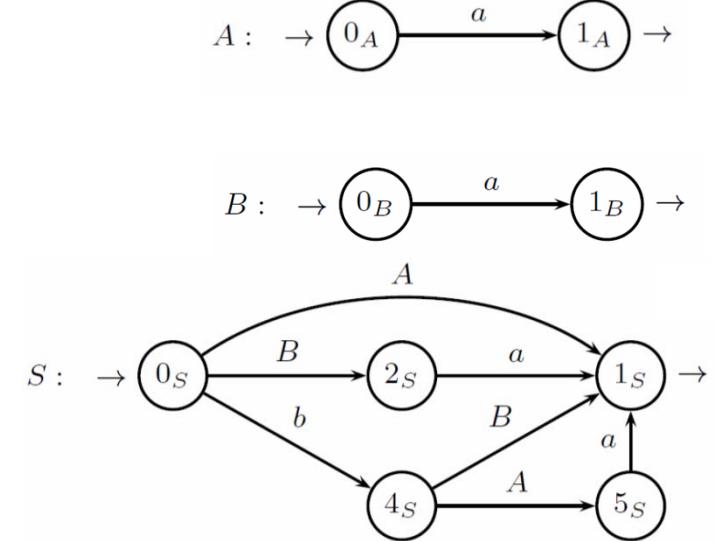


Analysis of string $bba \dashv$



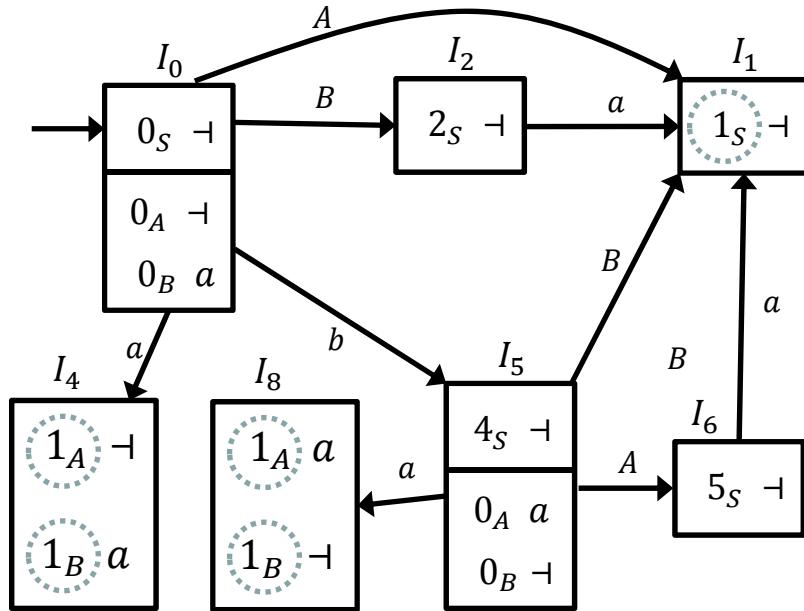


Analysis of string $baa \dashv$



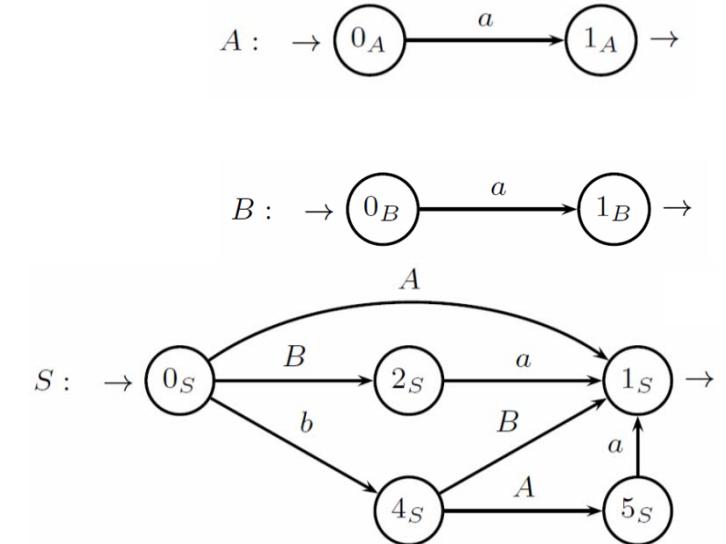
S

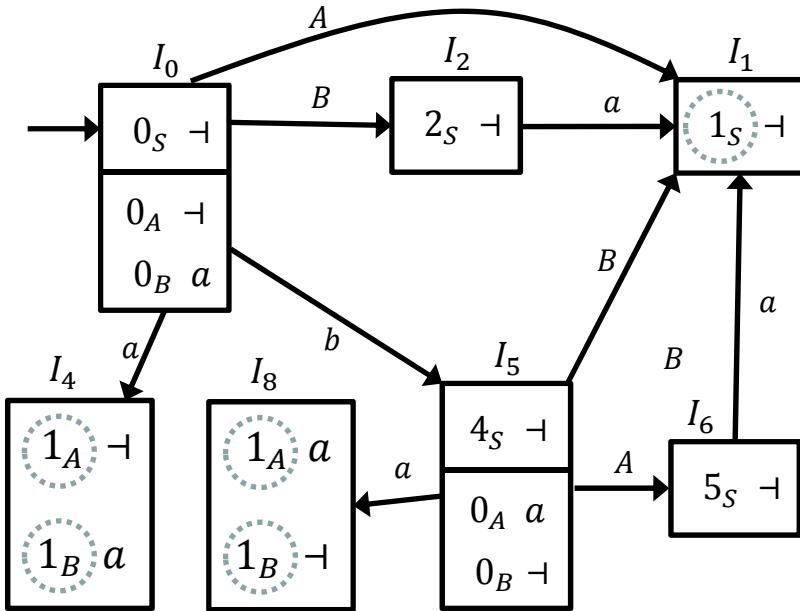
Initial m-state, axiom obtained
from a reduction, input \dashv :
ACCEPT



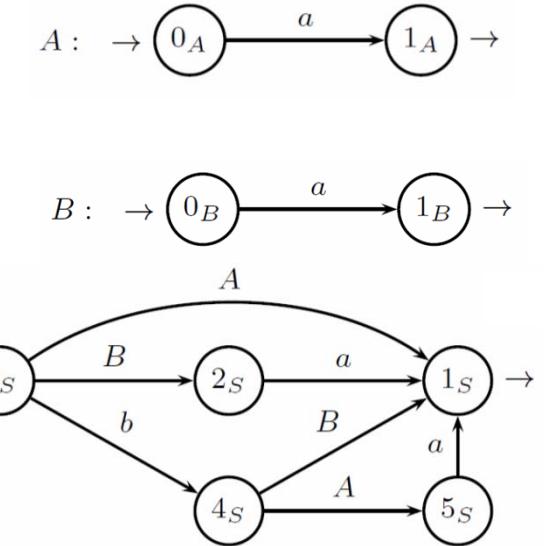
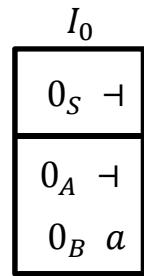
Analysis of string

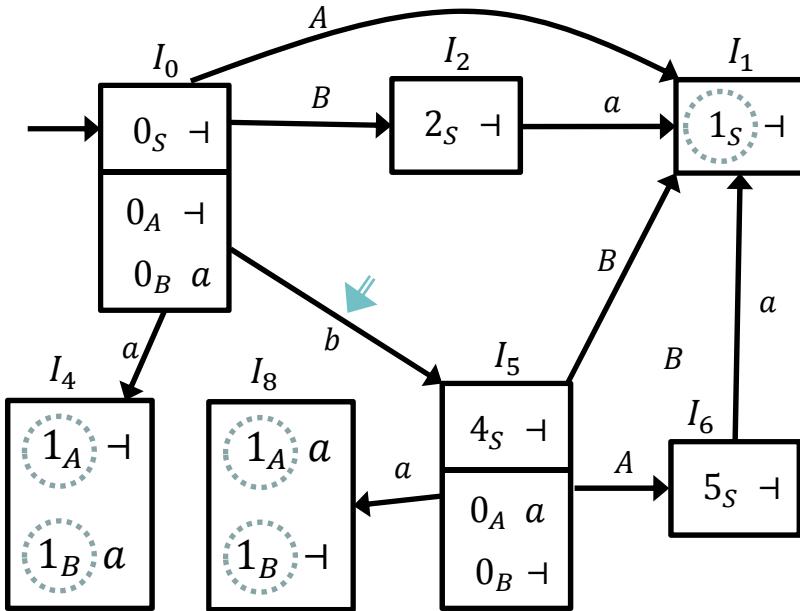
$ba \dashv$



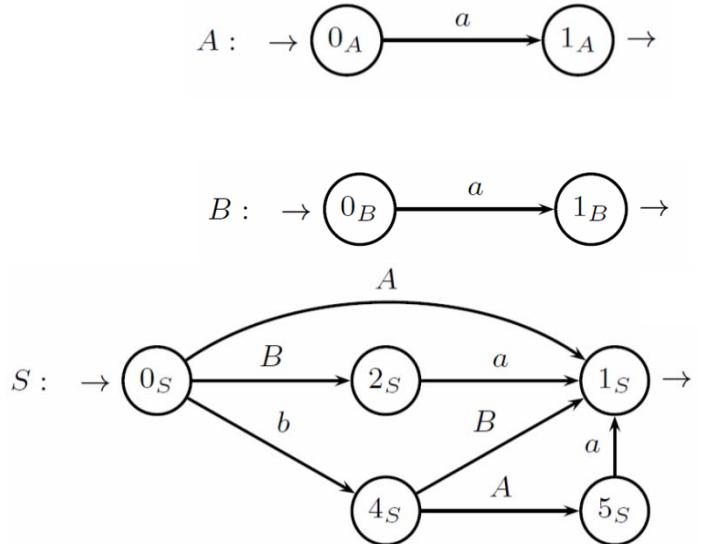
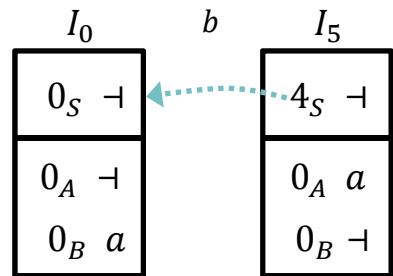


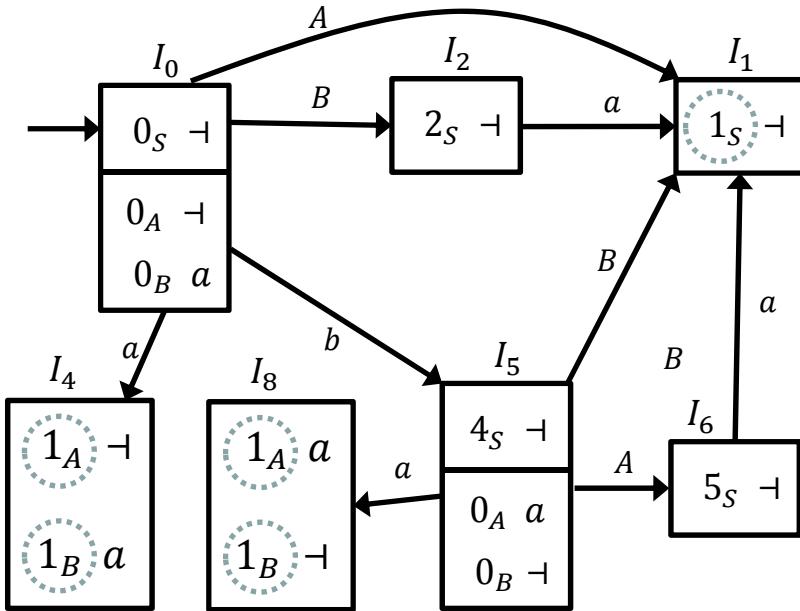
Analysis of string $ba \dashv$



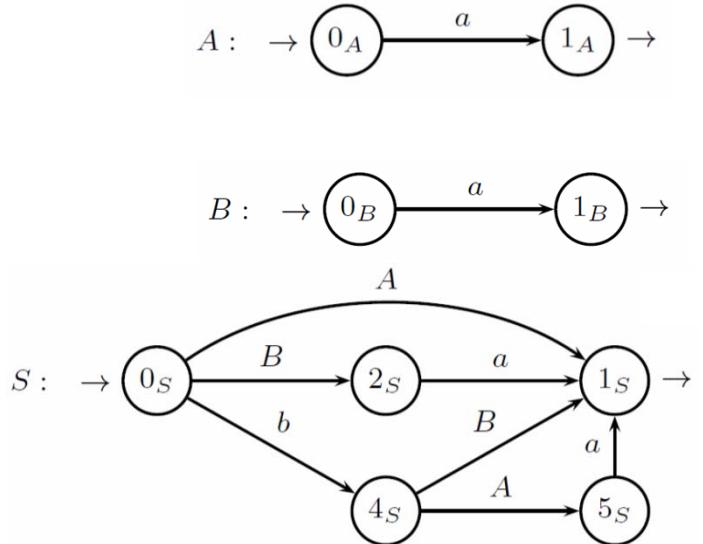
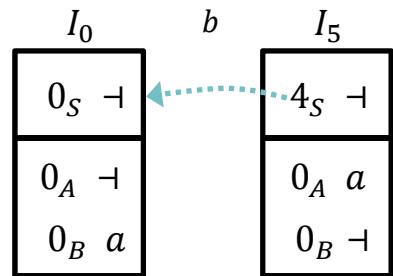


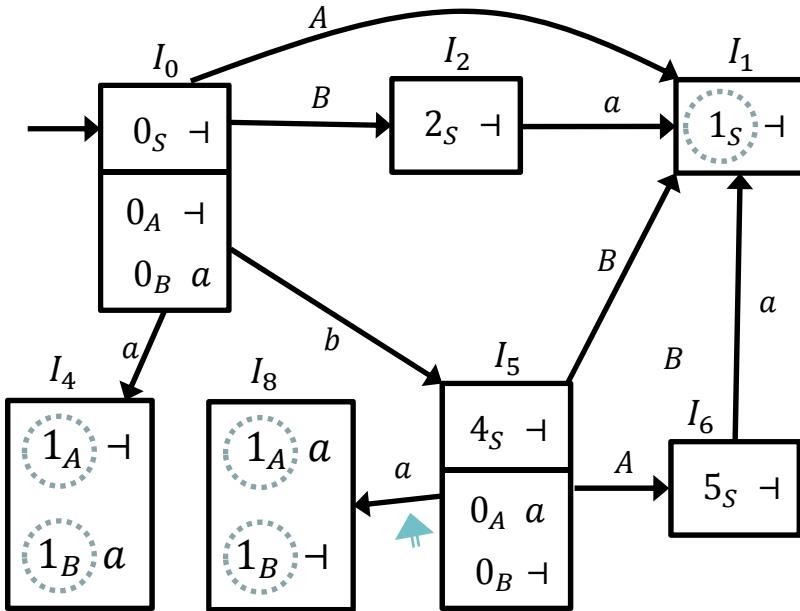
Analysis of string $ba \dashv$



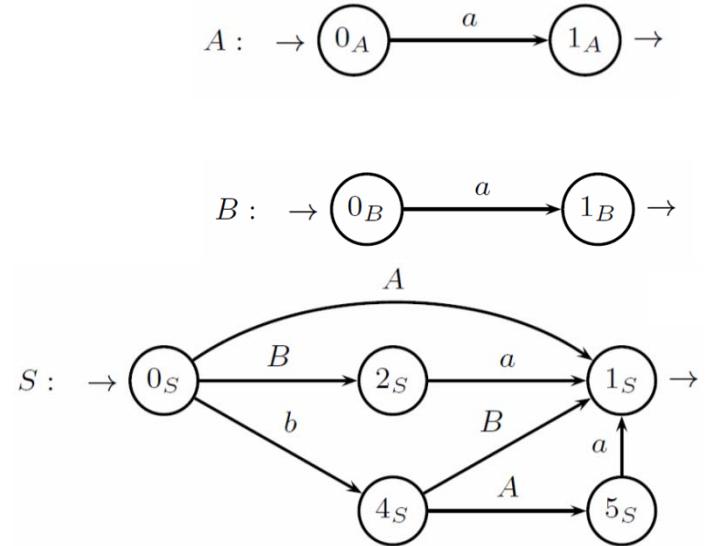
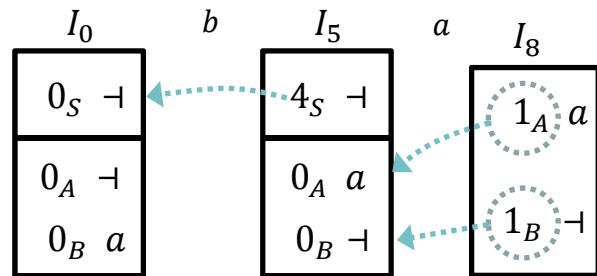


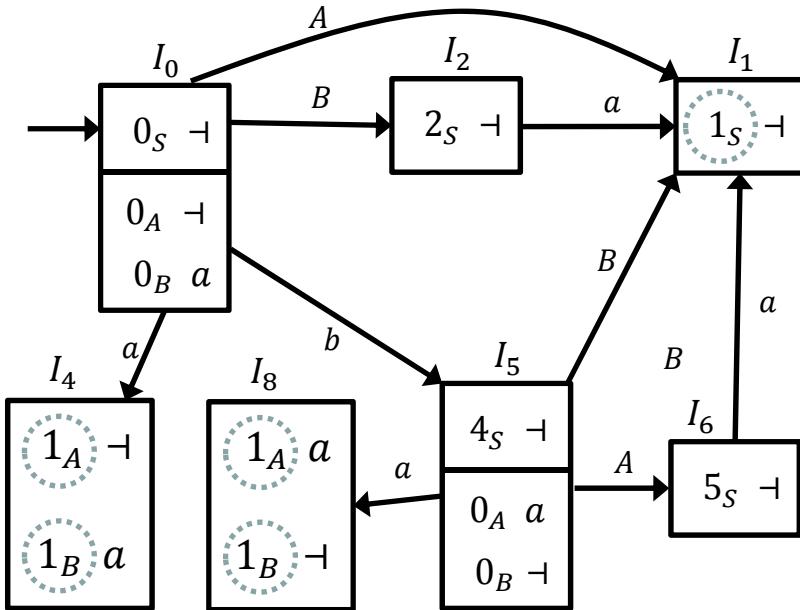
Analysis of string $ba \dashv$



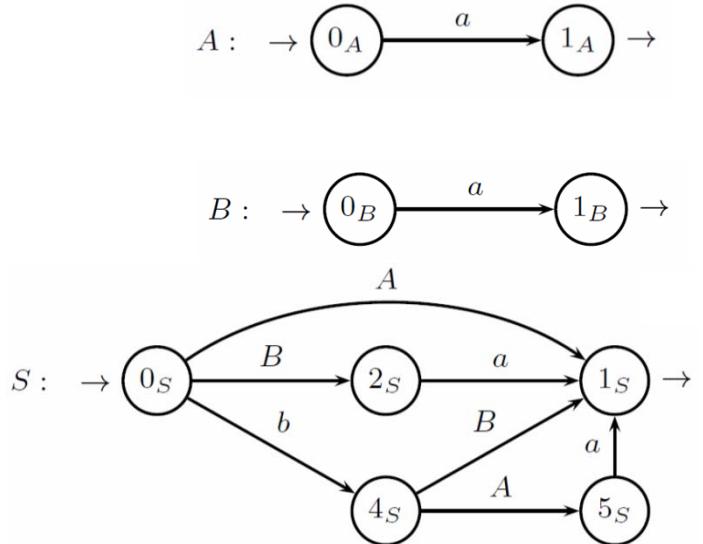
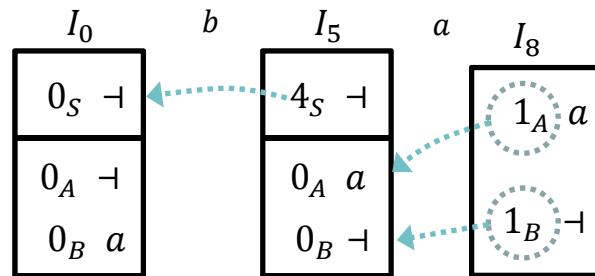


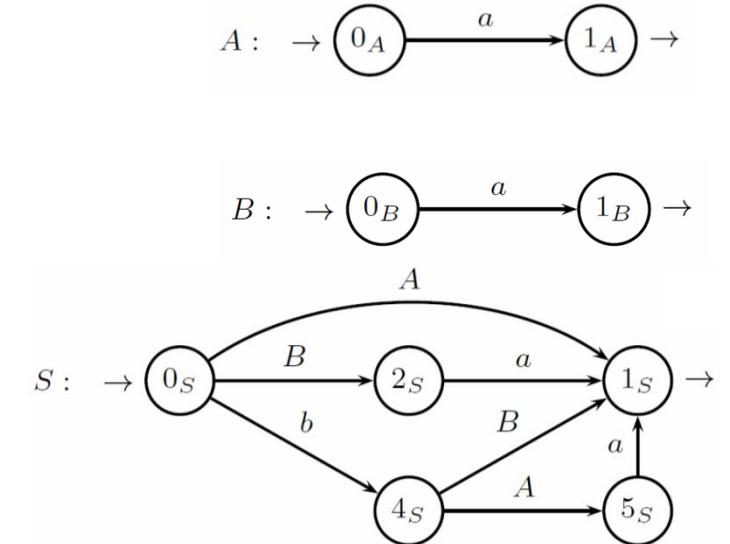
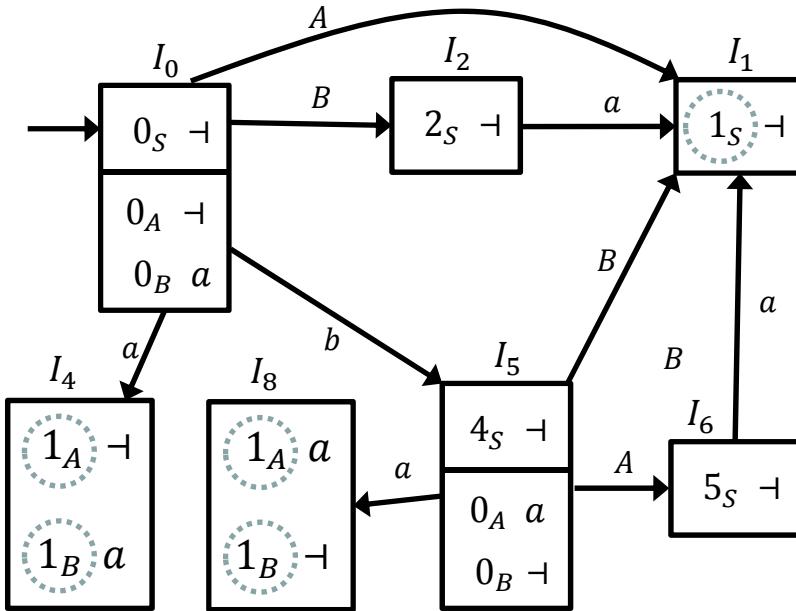
Analysis of string $ba \dashv$





Analysis of string $ba \dashv$

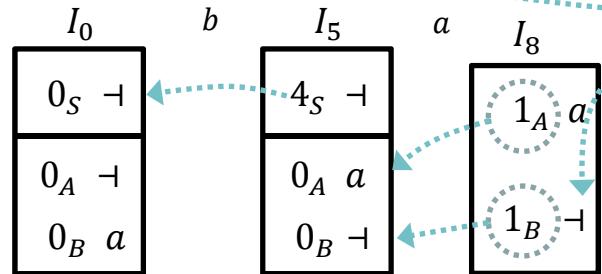


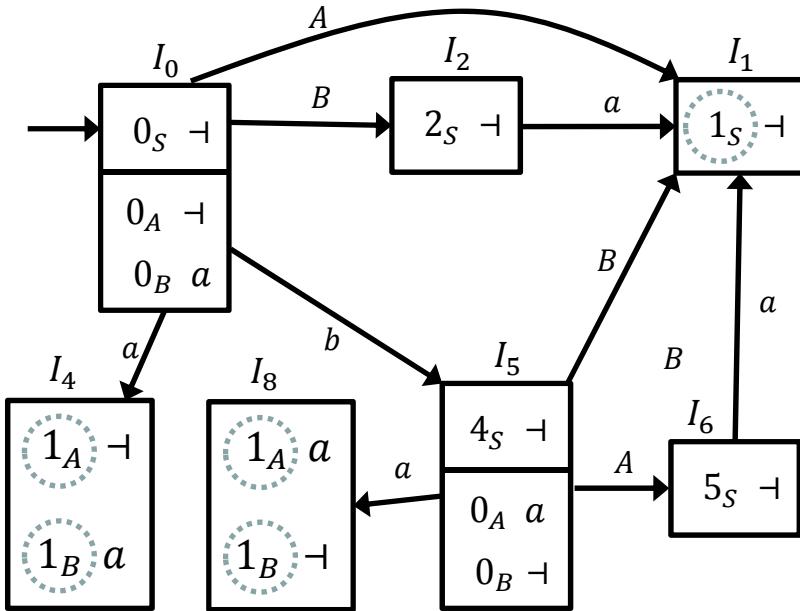


Analysis of string

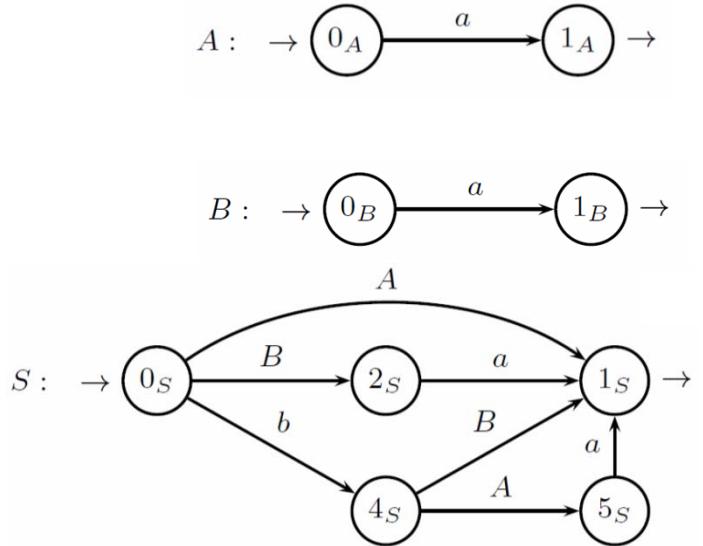
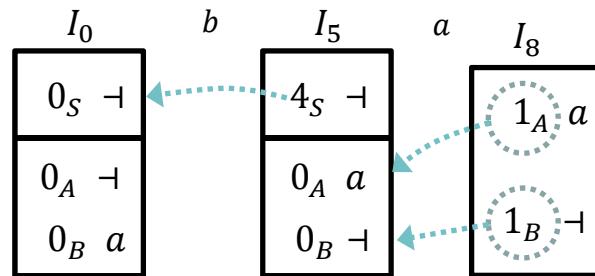
$ba \dashv$

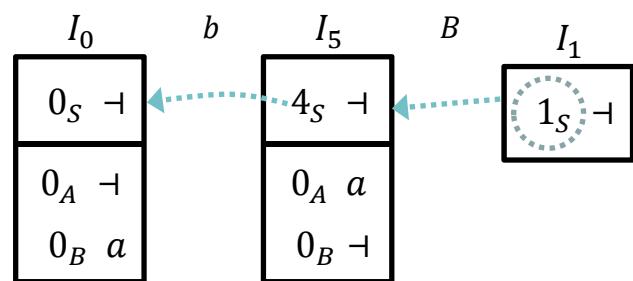
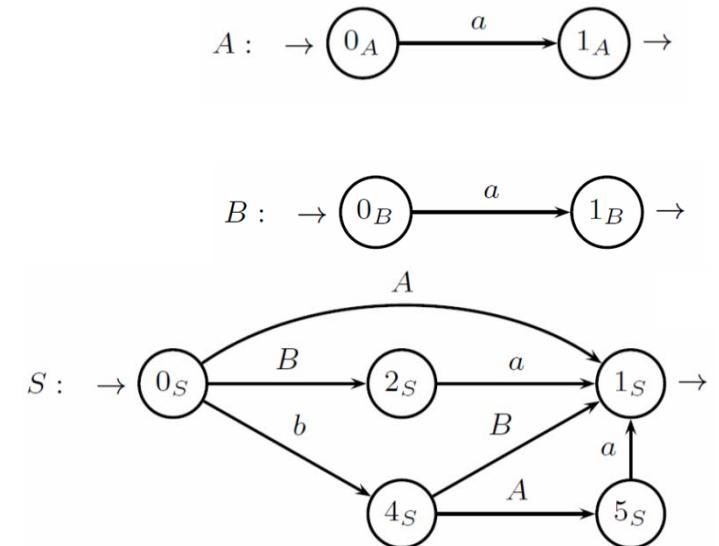
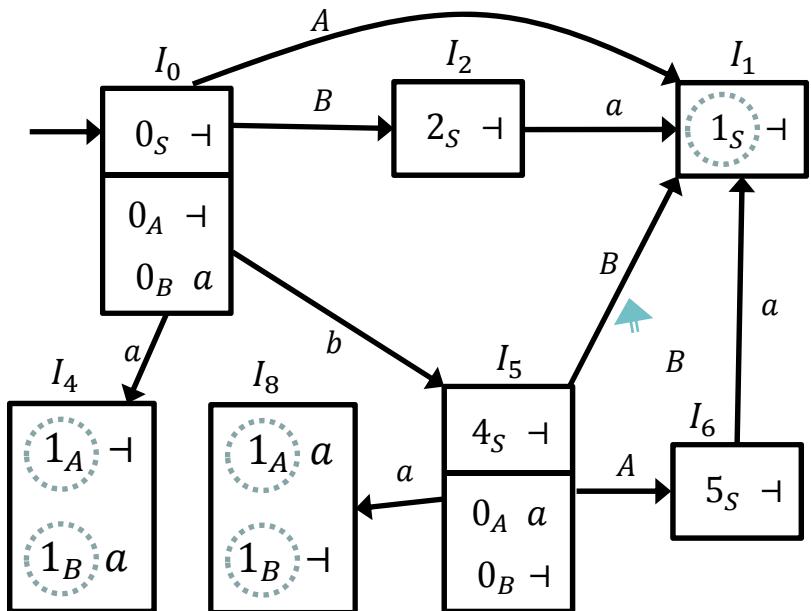
lookahead = input: reduce $a \rightarrow B$

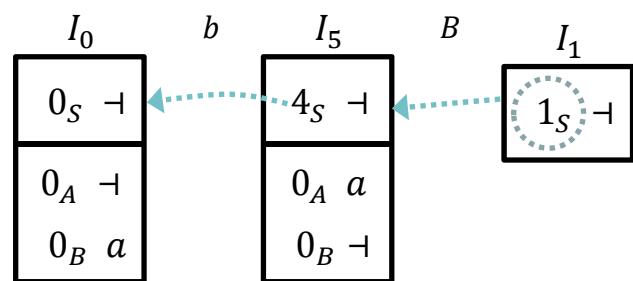
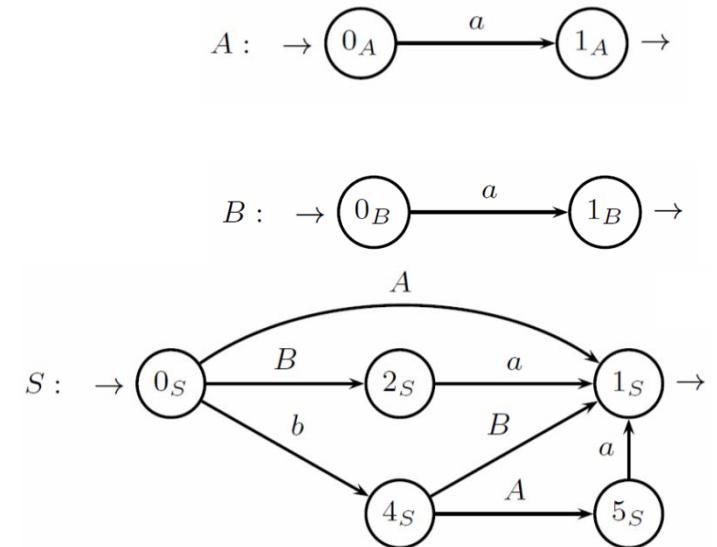
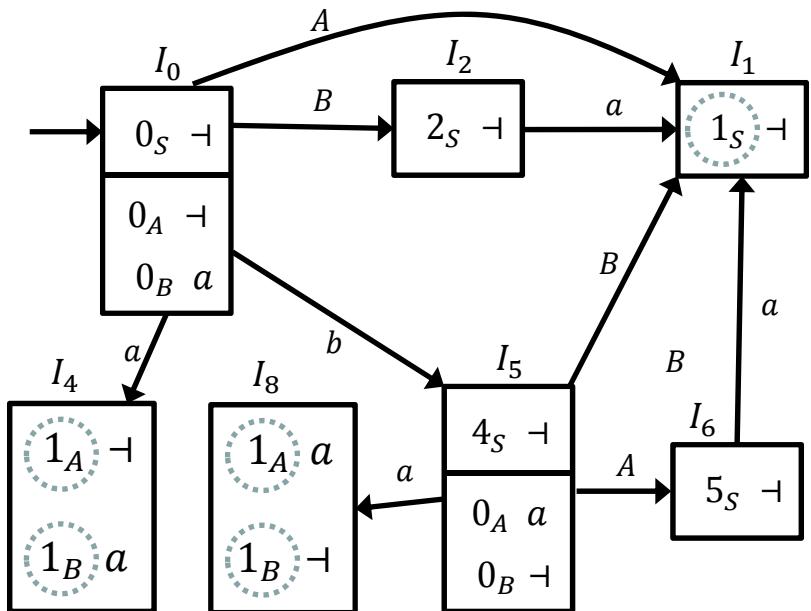


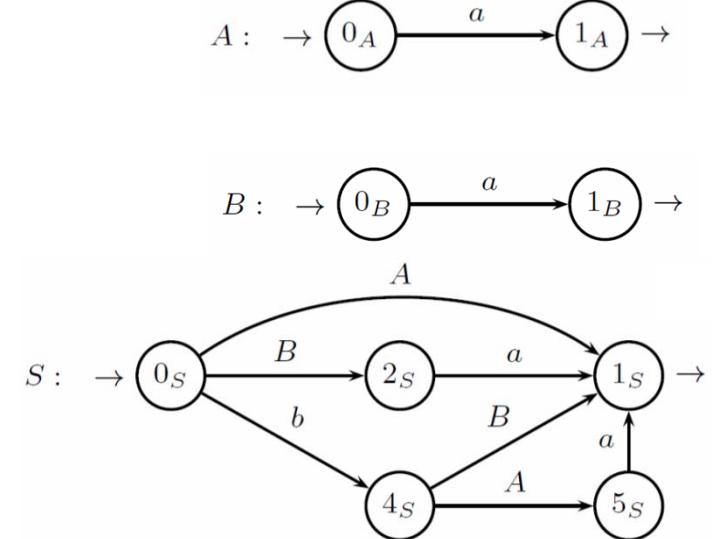
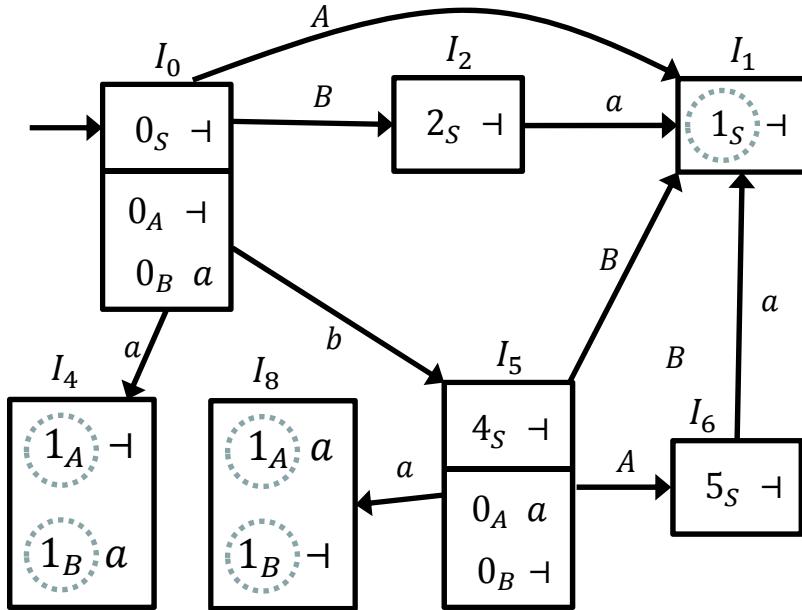


Analysis of string $ba \dashv$

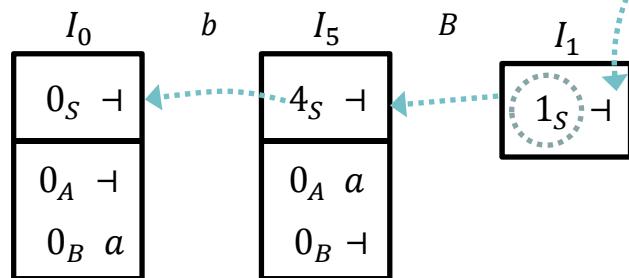


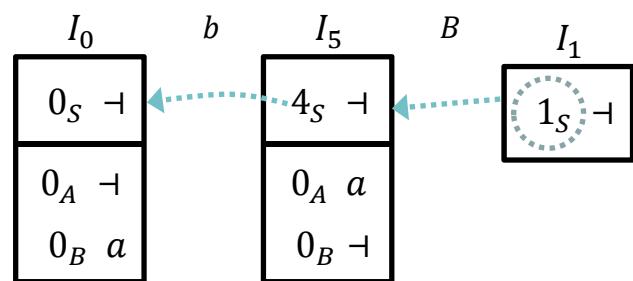
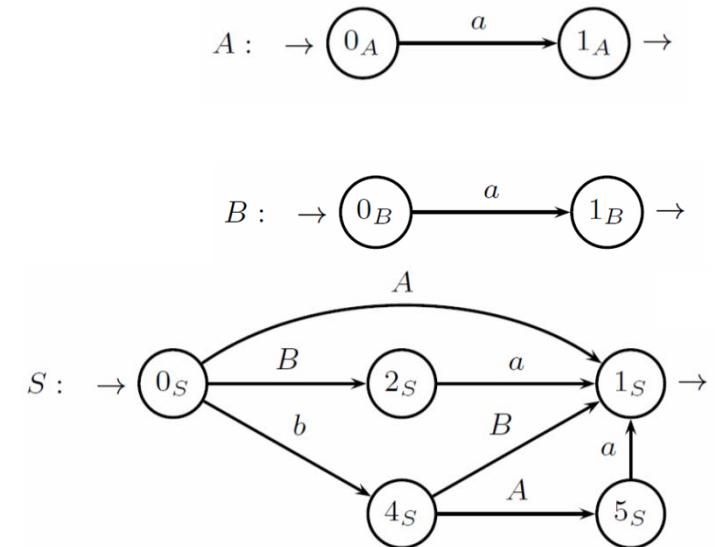
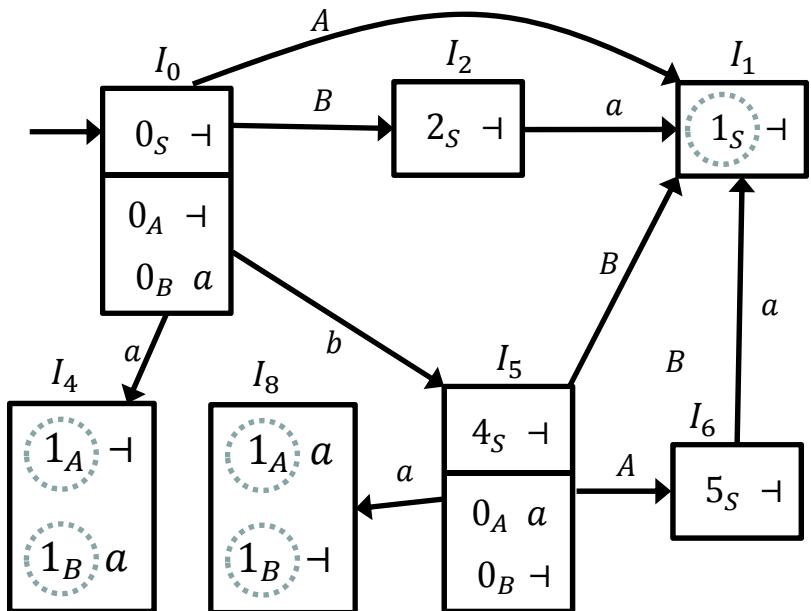


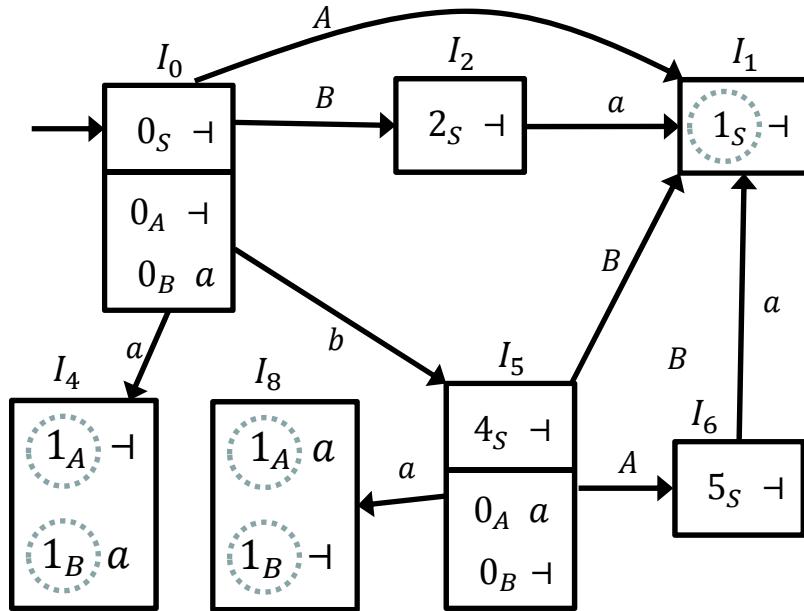




lookahead = input: reduce $bB \rightarrow S$

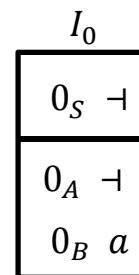
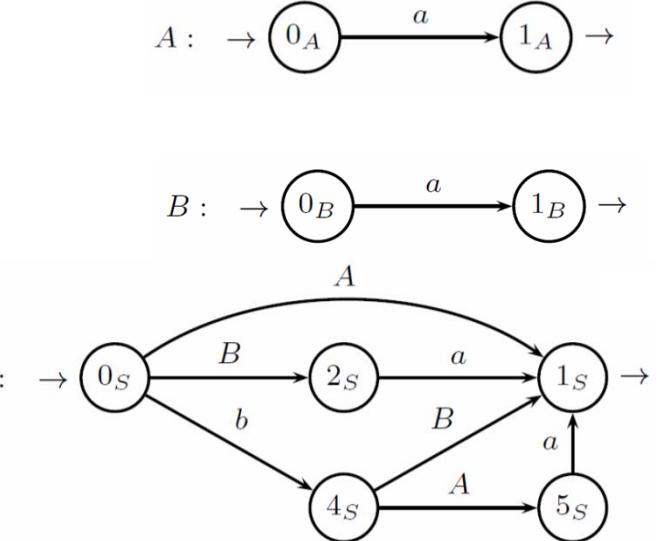






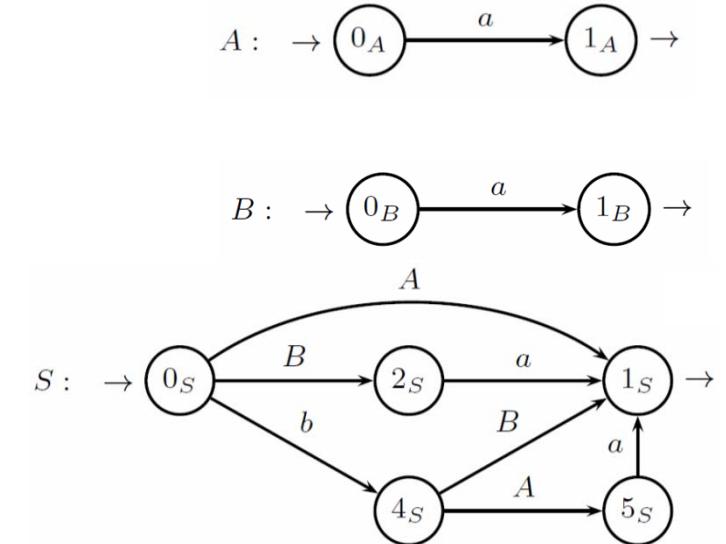
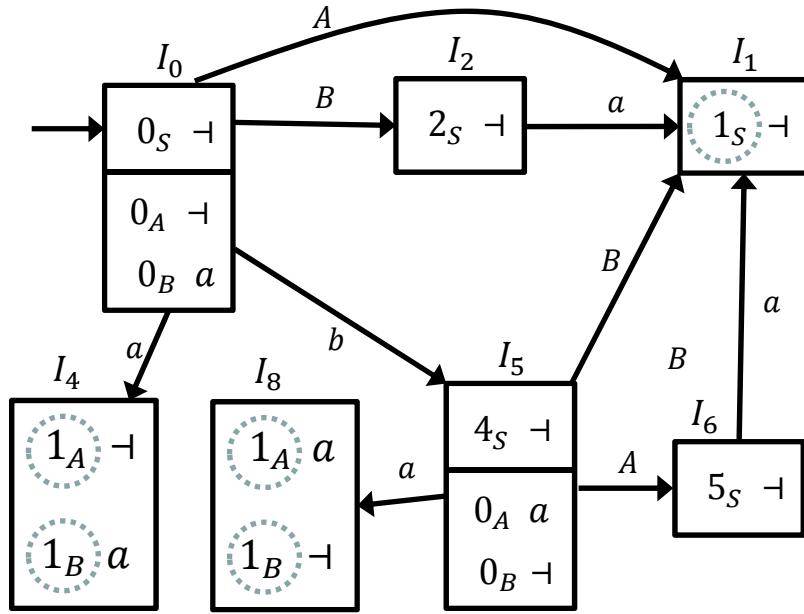
Analysis of string

$ba \dashv$



S

Initial m-state, axiom obtained
from a reduction, input \dashv :
ACCEPT

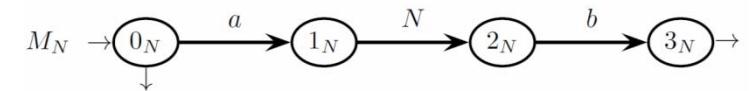
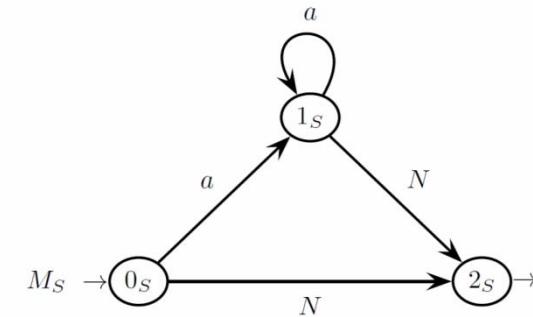


Exercise: simulate the analysis of strings $a \dashv$ and $aa \dashv$

Example (more interesting but still simple) with an infinite language

Example (more interesting but still simple) with an infinite language

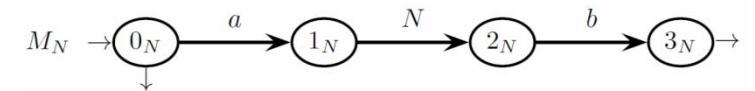
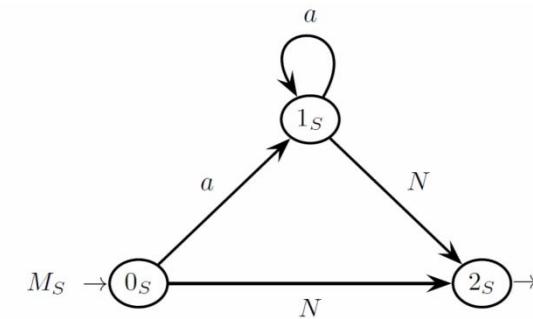
$$\begin{cases} S \rightarrow a^* N \\ N \rightarrow a N b \mid \varepsilon \end{cases} \quad L = \{a^n b^m \mid n \geq m \geq 0\}$$



Example (more interesting but still simple) with an infinite language

$$\begin{cases} S \rightarrow a^* N \\ N \rightarrow a N b \mid \varepsilon \end{cases} \quad L = \{a^n b^m \mid n \geq m \geq 0\}$$

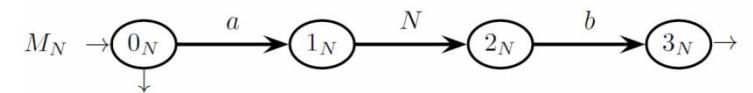
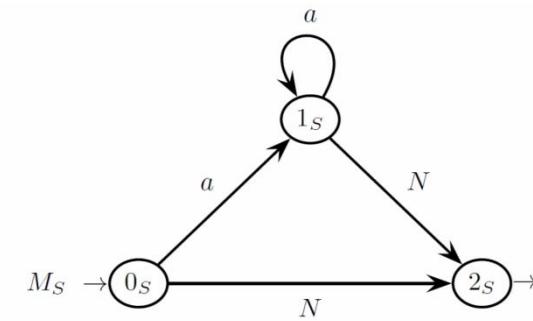
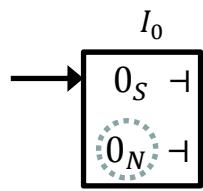
pilot has m-states with several items in the base



Example (more interesting but still simple) with an infinite language

$$\begin{cases} S \rightarrow a^* N \\ N \rightarrow a N b \mid \varepsilon \end{cases} \quad L = \{a^n b^m \mid n \geq m \geq 0\}$$

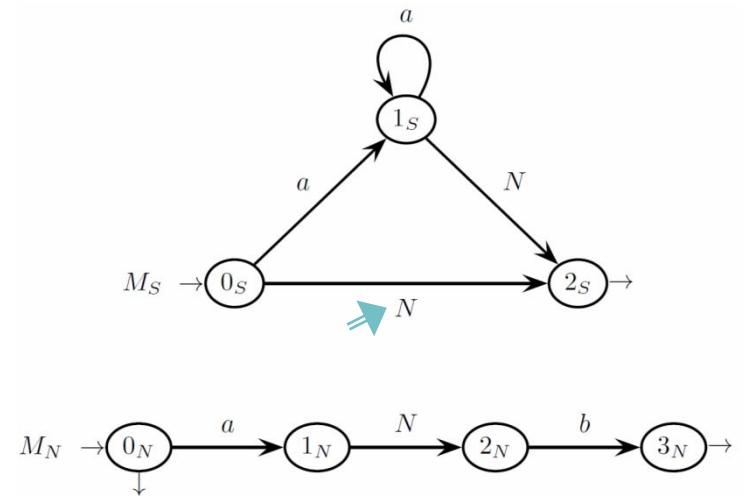
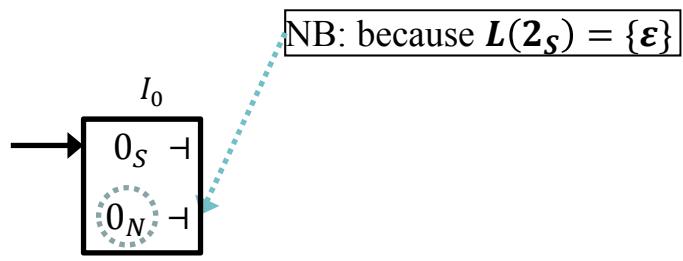
pilot has m-states with several items in the base



Example (more interesting but still simple) with an infinite language

$$\begin{cases} S \rightarrow a^* N \\ N \rightarrow a N b \mid \varepsilon \end{cases} \quad L = \{a^n b^m \mid n \geq m \geq 0\}$$

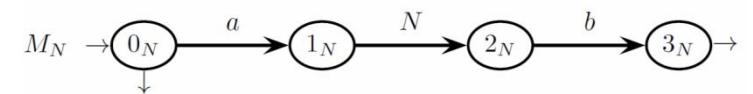
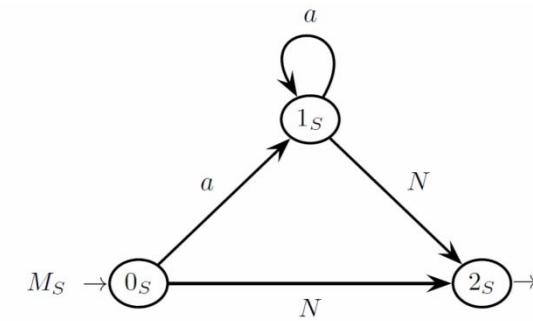
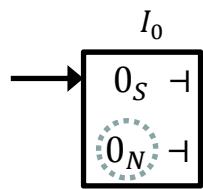
pilot has m-states with several items in the base



Example (more interesting but still simple) with an infinite language

$$\begin{cases} S \rightarrow a^* N \\ N \rightarrow a N b \mid \varepsilon \end{cases} \quad L = \{a^n b^m \mid n \geq m \geq 0\}$$

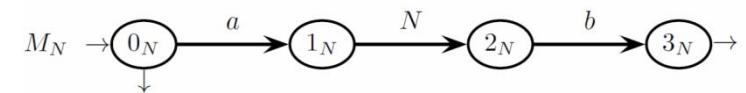
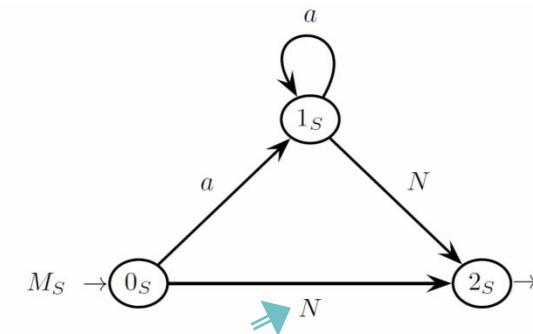
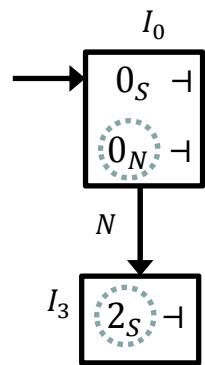
pilot has m-states with several items in the base



Example (more interesting but still simple) with an infinite language

$$\begin{cases} S \rightarrow a^* N \\ N \rightarrow a N b \mid \varepsilon \end{cases} \quad L = \{a^n b^m \mid n \geq m \geq 0\}$$

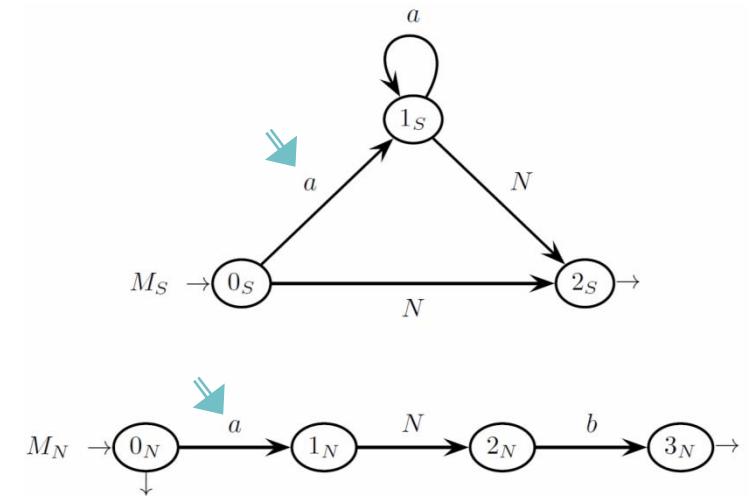
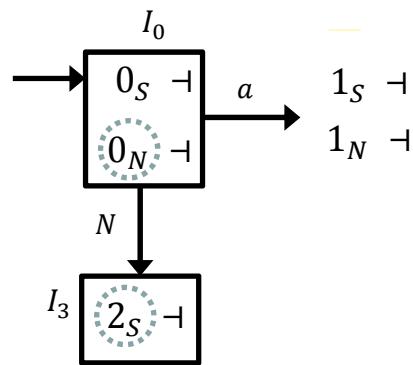
pilot has m-states with several items in the base



Example (more interesting but still simple) with an infinite language

$$\begin{cases} S \rightarrow a^* N \\ N \rightarrow a N b \mid \varepsilon \end{cases} \quad L = \{a^n b^m \mid n \geq m \geq 0\}$$

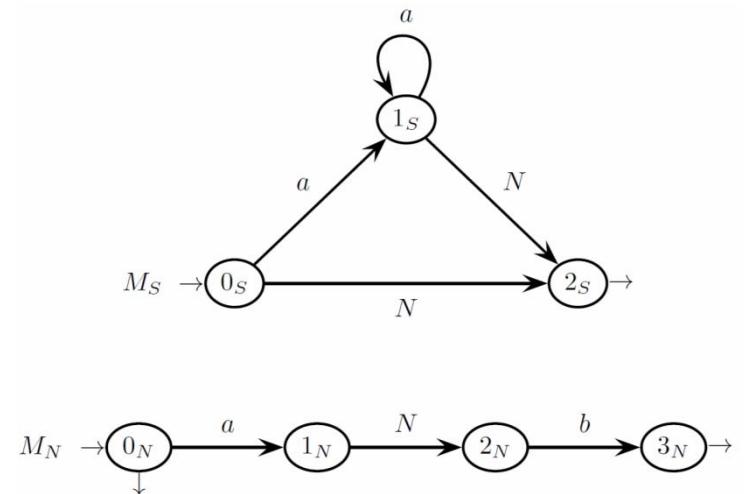
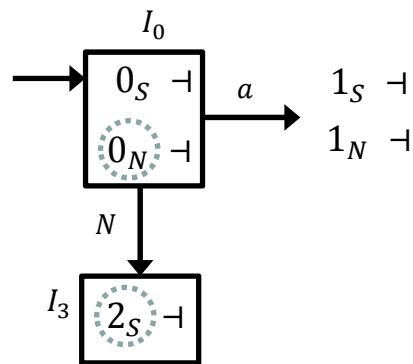
pilot has m-states with several items in the base



Example (more interesting but still simple) with an infinite language

$$\begin{cases} S \rightarrow a^* N \\ N \rightarrow a N b \mid \varepsilon \end{cases} \quad L = \{a^n b^m \mid n \geq m \geq 0\}$$

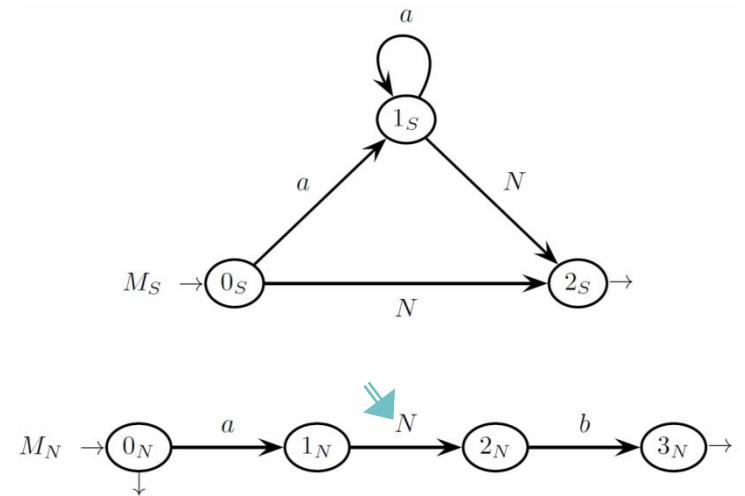
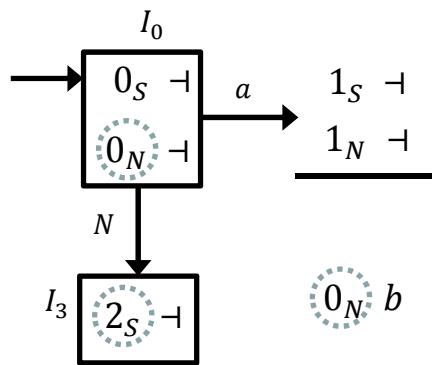
pilot has m-states with several items in the base



Example (more interesting but still simple) with an infinite language

$$\begin{cases} S \rightarrow a^* N \\ N \rightarrow a N b \mid \varepsilon \end{cases} \quad L = \{a^n b^m \mid n \geq m \geq 0\}$$

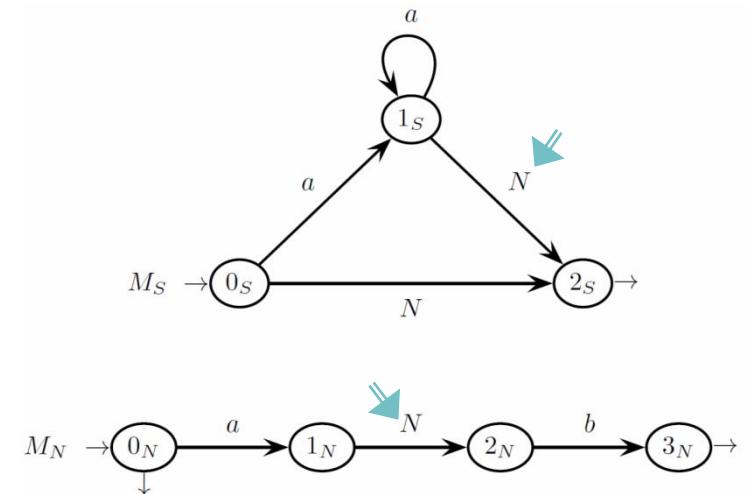
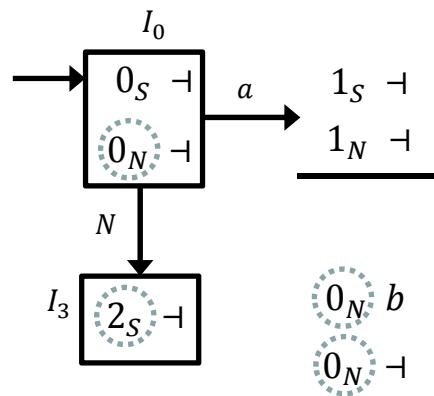
pilot has m-states with several items in the base



Example (more interesting but still simple) with an infinite language

$$\begin{cases} S \rightarrow a^* N \\ N \rightarrow a N b \mid \varepsilon \end{cases} \quad L = \{a^n b^m \mid n \geq m \geq 0\}$$

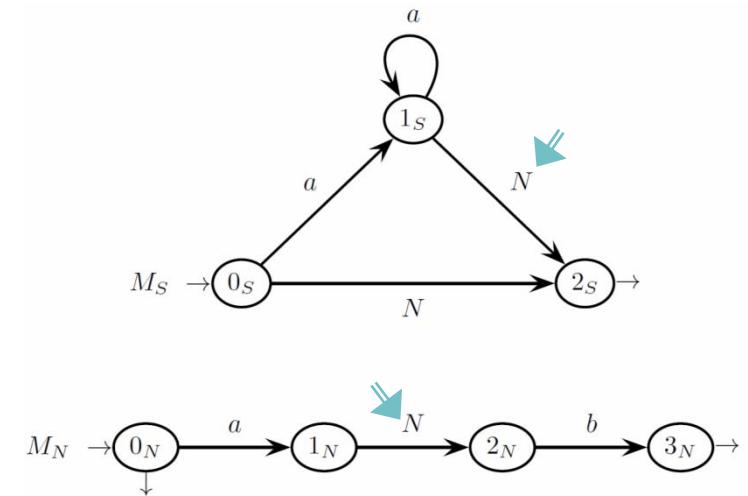
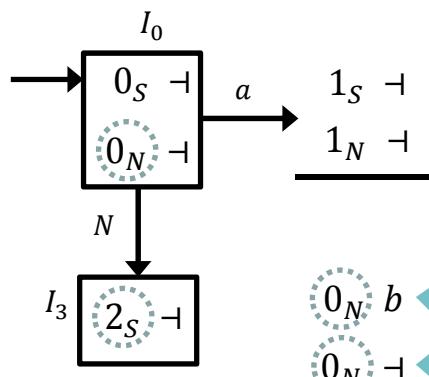
pilot has m-states with several items in the base



Example (more interesting but still simple) with an infinite language

$$\begin{cases} S \rightarrow a^* N \\ N \rightarrow a N b \mid \varepsilon \end{cases} \quad L = \{a^n b^m \mid n \geq m \geq 0\}$$

pilot has m-states with several items in the base

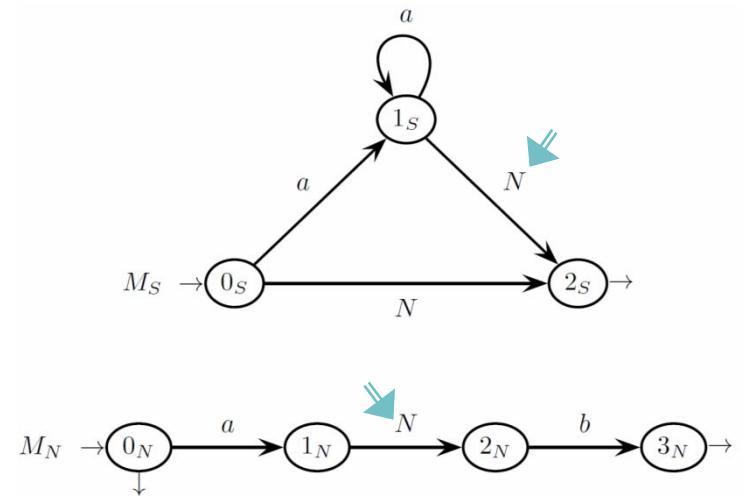
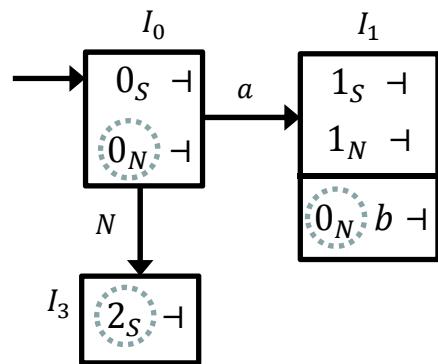


NB: items with the same state are grouped merging the lookahead

Example (more interesting but still simple) with an infinite language

$$\begin{cases} S \rightarrow a^* N \\ N \rightarrow a N b \mid \varepsilon \end{cases} \quad L = \{a^n b^m \mid n \geq m \geq 0\}$$

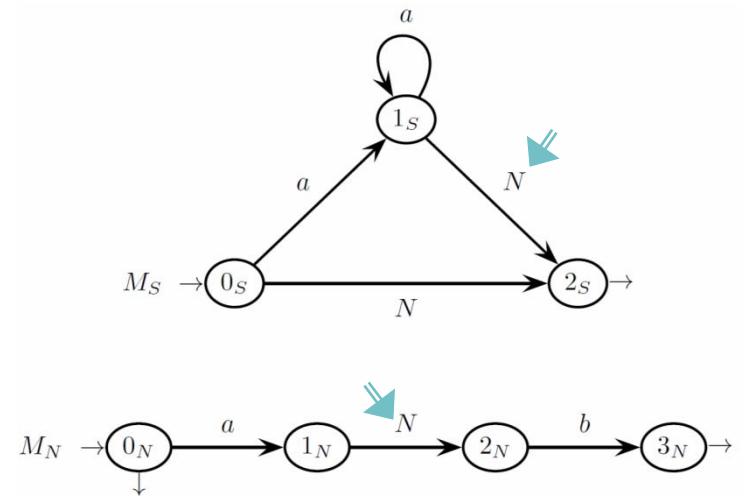
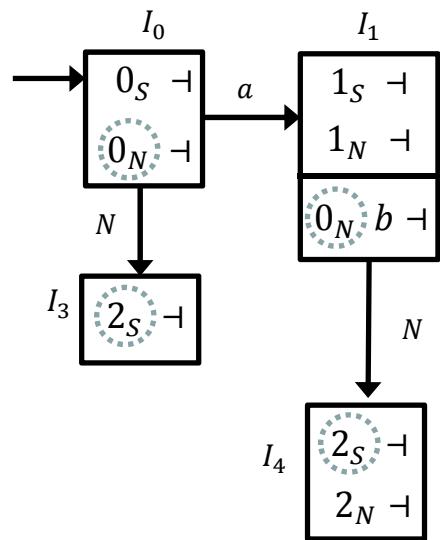
pilot has m-states with several items in the base



Example (more interesting but still simple) with an infinite language

$$\begin{cases} S \rightarrow a^* N \\ N \rightarrow a N b \mid \varepsilon \end{cases} \quad L = \{a^n b^m \mid n \geq m \geq 0\}$$

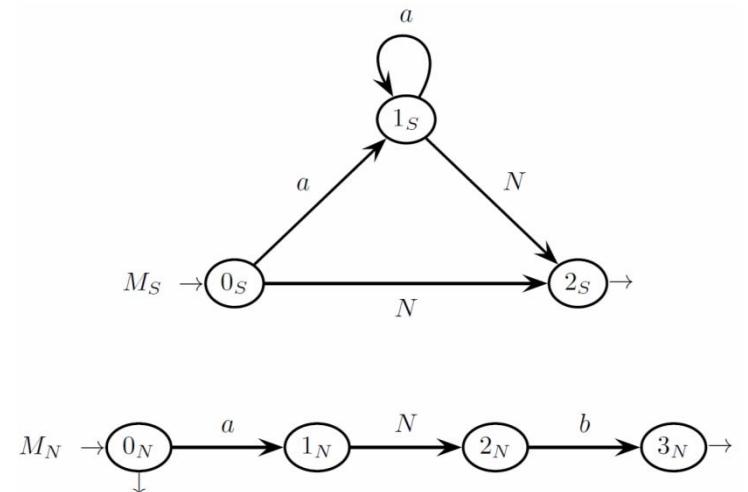
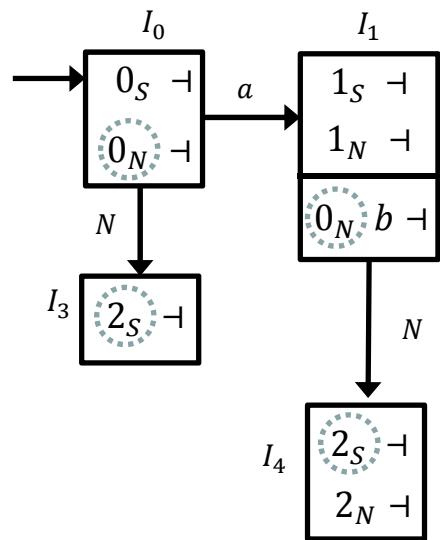
pilot has m-states with several items in the base



Example (more interesting but still simple) with an infinite language

$$\begin{cases} S \rightarrow a^* N \\ N \rightarrow a \ N \ b \mid \varepsilon \end{cases} \quad L = \{a^n \ b^m \mid n \geq m \geq 0\}$$

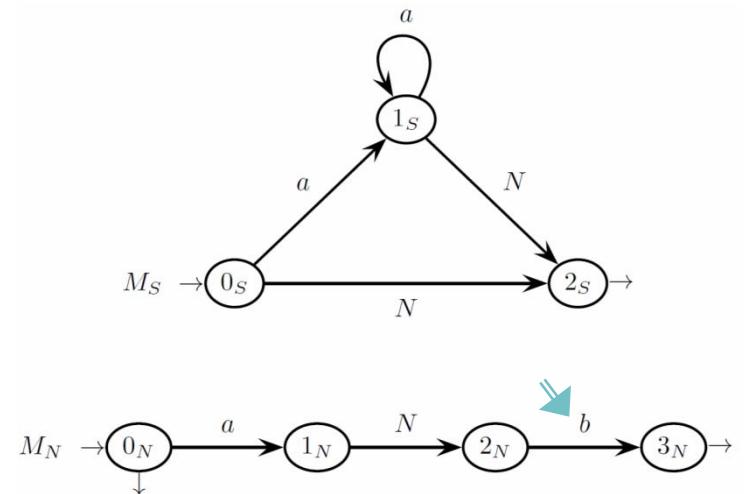
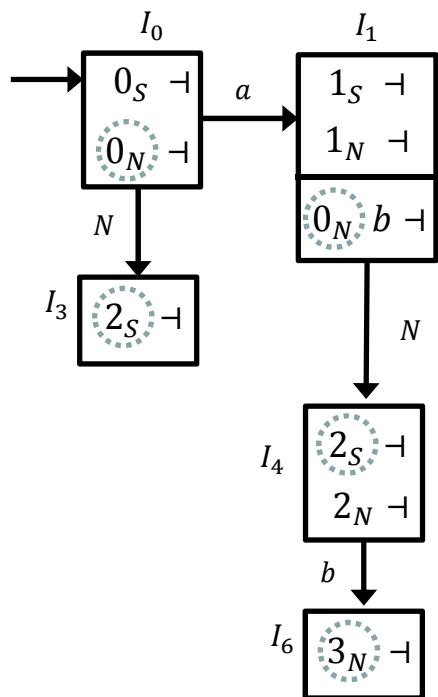
pilot has m-states with several items in the base



Example (more interesting but still simple) with an infinite language

$$\begin{cases} S \rightarrow a^* N \\ N \rightarrow a \ N \ b \mid \varepsilon \end{cases} \quad L = \{a^n \ b^m \mid n \geq m \geq 0\}$$

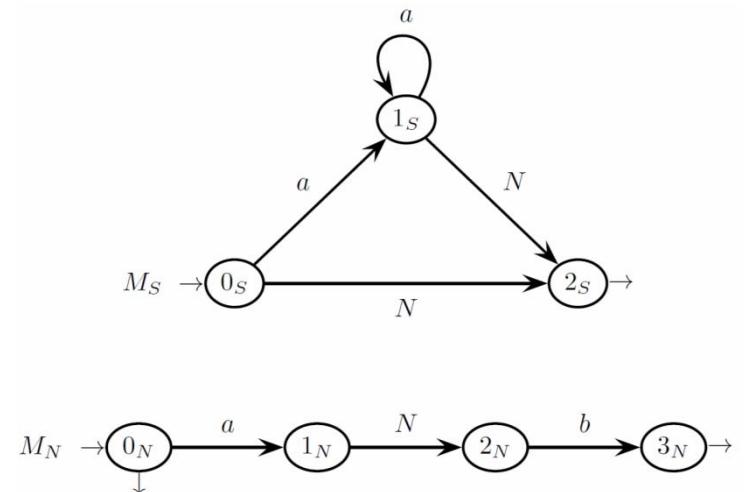
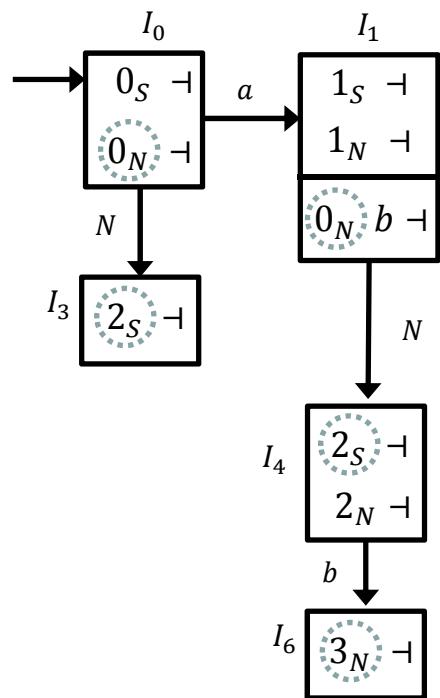
pilot has m-states with several items in the base



Example (more interesting but still simple) with an infinite language

$$\begin{cases} S \rightarrow a^* N \\ N \rightarrow a \ N \ b \mid \varepsilon \end{cases} \quad L = \{a^n \ b^m \mid n \geq m \geq 0\}$$

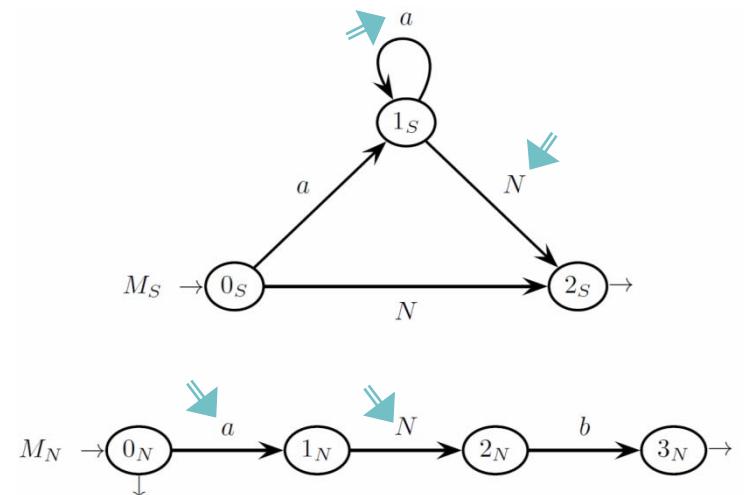
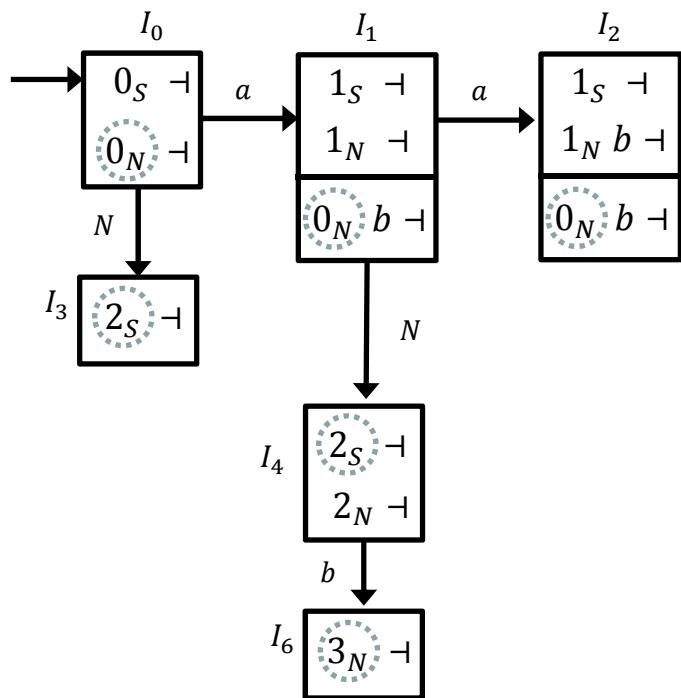
pilot has m-states with several items in the base



Example (more interesting but still simple) with an infinite language

$$\begin{cases} S \rightarrow a^* N \\ N \rightarrow a \ N \ b \mid \varepsilon \end{cases} \quad L = \{a^n \ b^m \mid n \geq m \geq 0\}$$

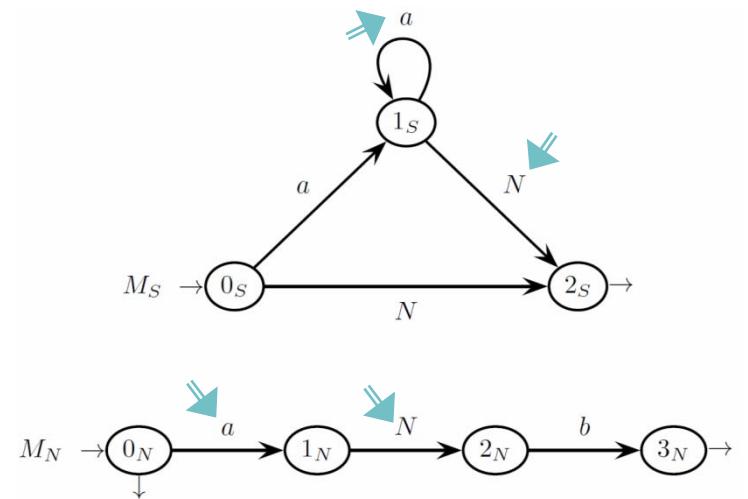
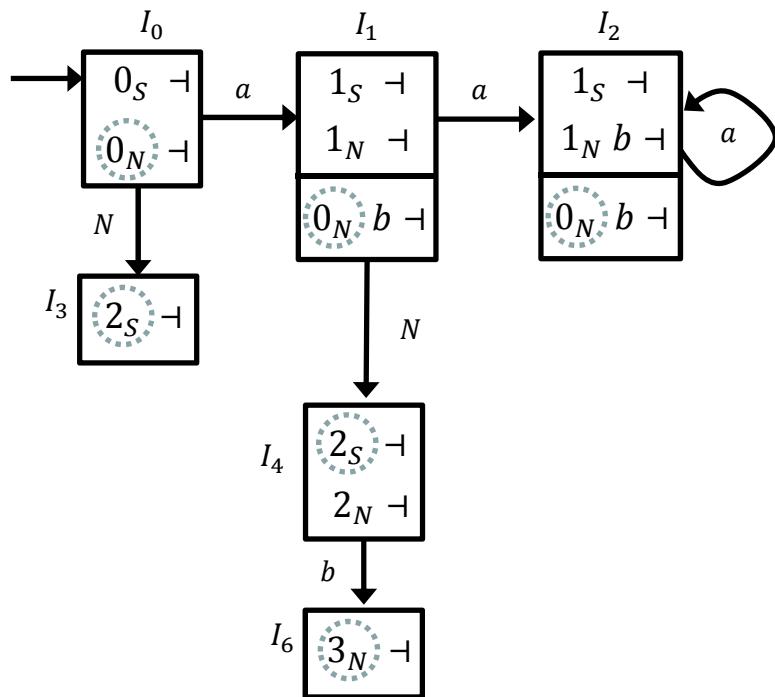
pilot has m-states with several items in the base



Example (more interesting but still simple) with an infinite language

$$\begin{cases} S \rightarrow a^* N \\ N \rightarrow a \ N \ b \mid \varepsilon \end{cases} \quad L = \{a^n \ b^m \mid n \geq m \geq 0\}$$

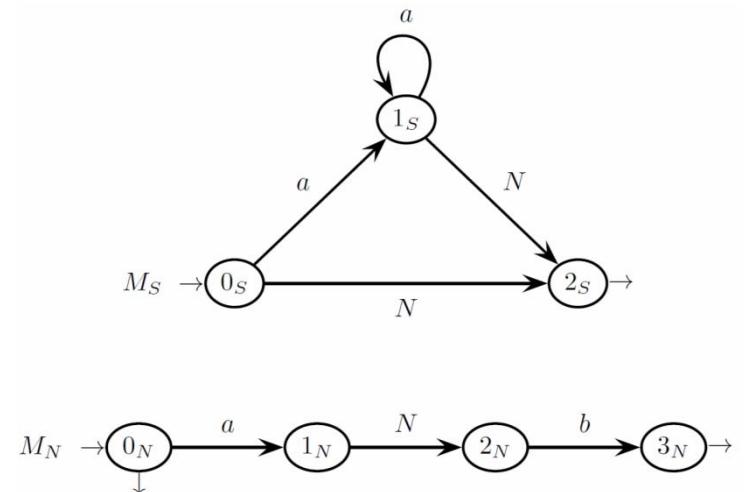
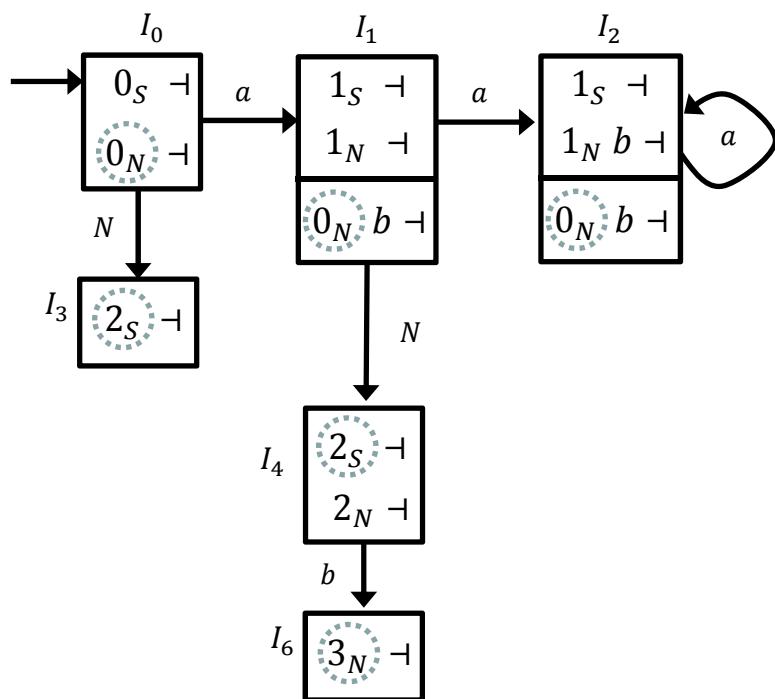
pilot has m-states with several items in the base



Example (more interesting but still simple) with an infinite language

$$\begin{cases} S \rightarrow a^* N \\ N \rightarrow a \ N \ b \mid \varepsilon \end{cases} \quad L = \{a^n \ b^m \mid n \geq m \geq 0\}$$

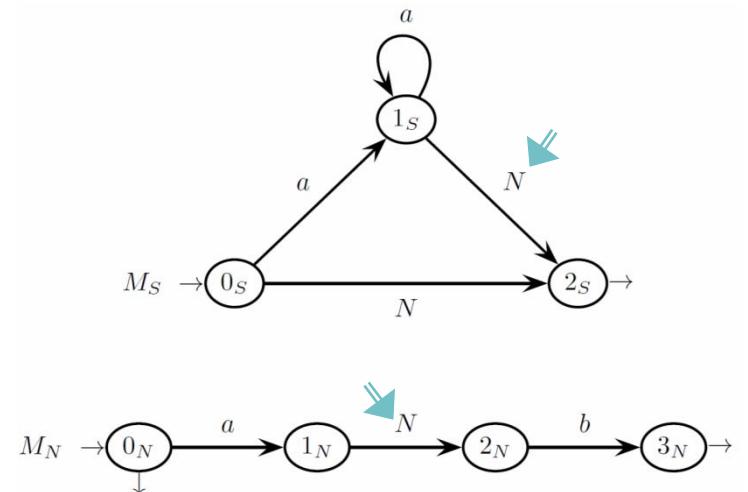
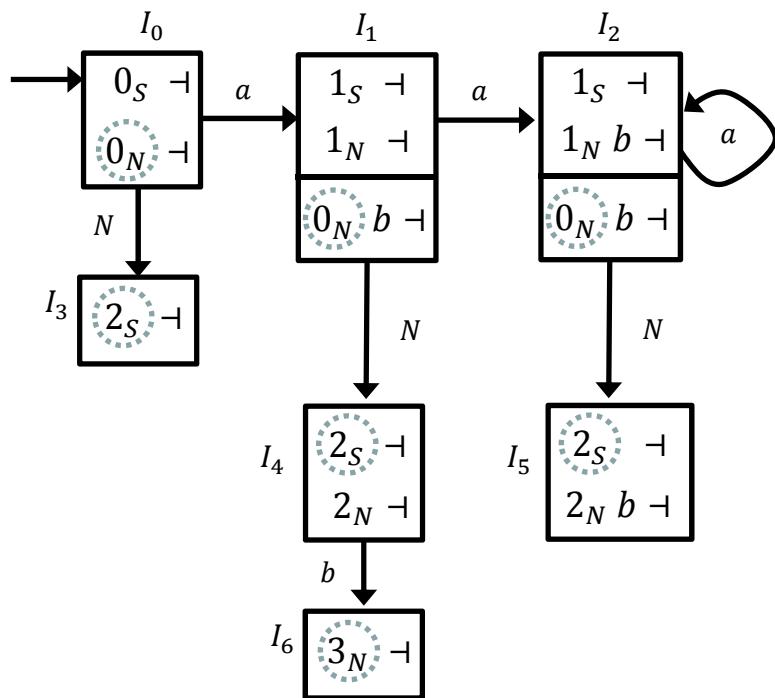
pilot has m-states with several items in the base



Example (more interesting but still simple) with an infinite language

$$\begin{cases} S \rightarrow a^* N \\ N \rightarrow a \ N \ b \mid \varepsilon \end{cases} \quad L = \{a^n \ b^m \mid n \geq m \geq 0\}$$

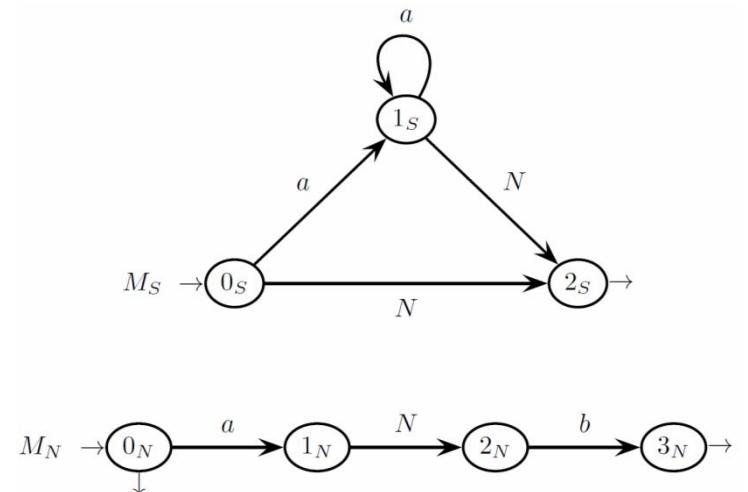
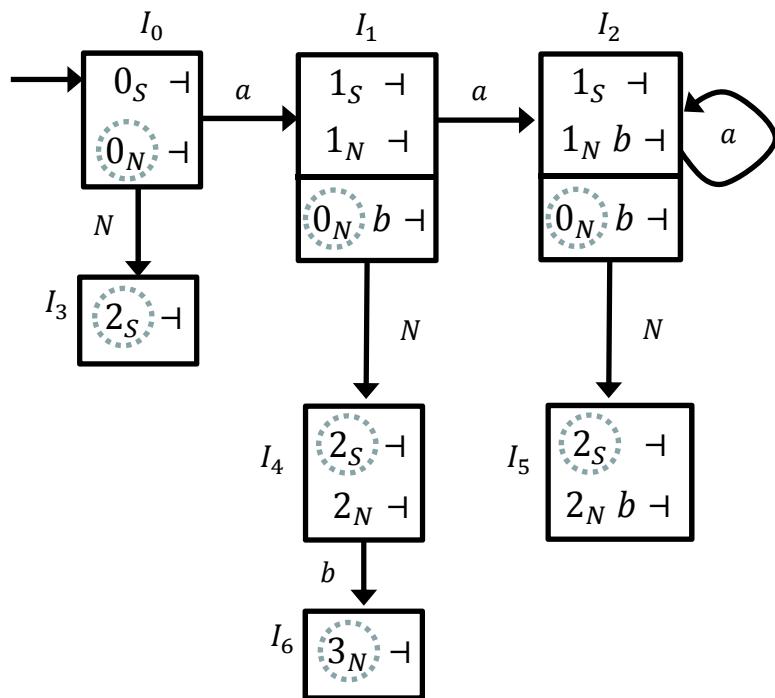
pilot has m-states with several items in the base



Example (more interesting but still simple) with an infinite language

$$\begin{cases} S \rightarrow a^* N \\ N \rightarrow a \ N \ b \mid \varepsilon \end{cases} \quad L = \{a^n \ b^m \mid n \geq m \geq 0\}$$

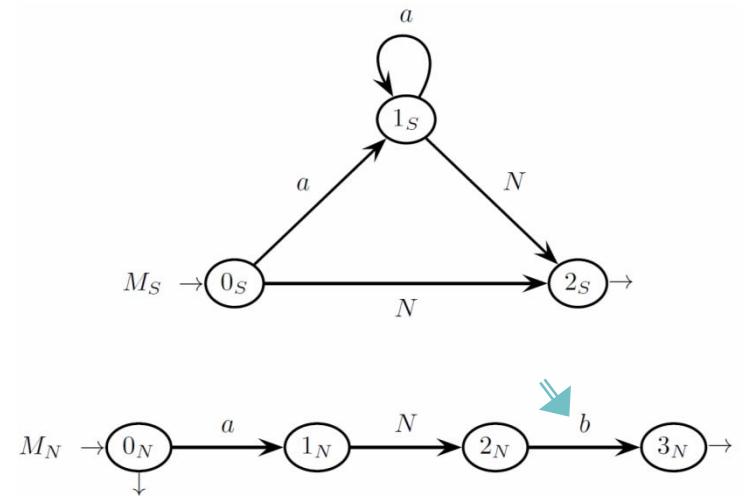
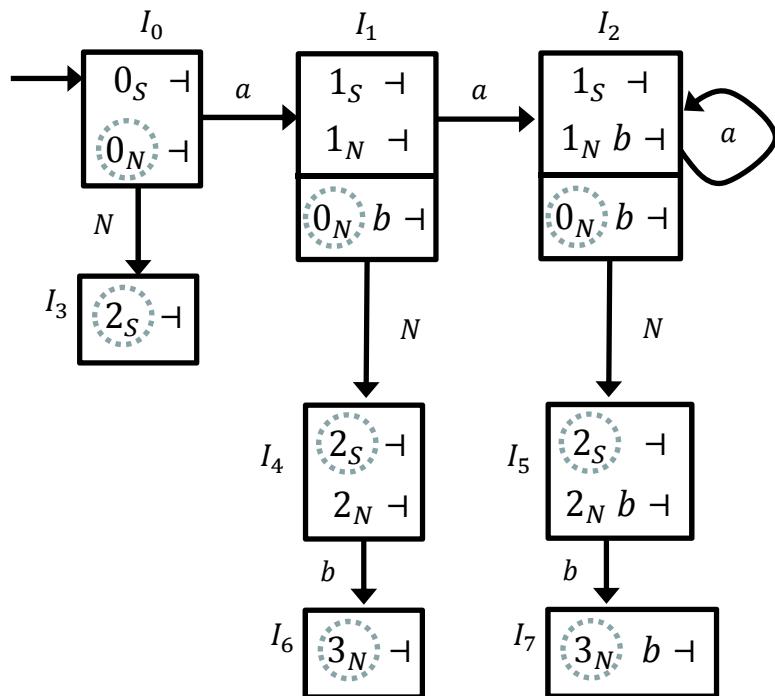
pilot has m-states with several items in the base



Example (more interesting but still simple) with an infinite language

$$\left\{ \begin{array}{l} S \rightarrow a^* N \\ N \rightarrow a \ N \ b \mid \varepsilon \end{array} \right. \quad L = \{a^n \ b^m \mid n \geq m \geq 0\}$$

pilot has m-states with several items in the base

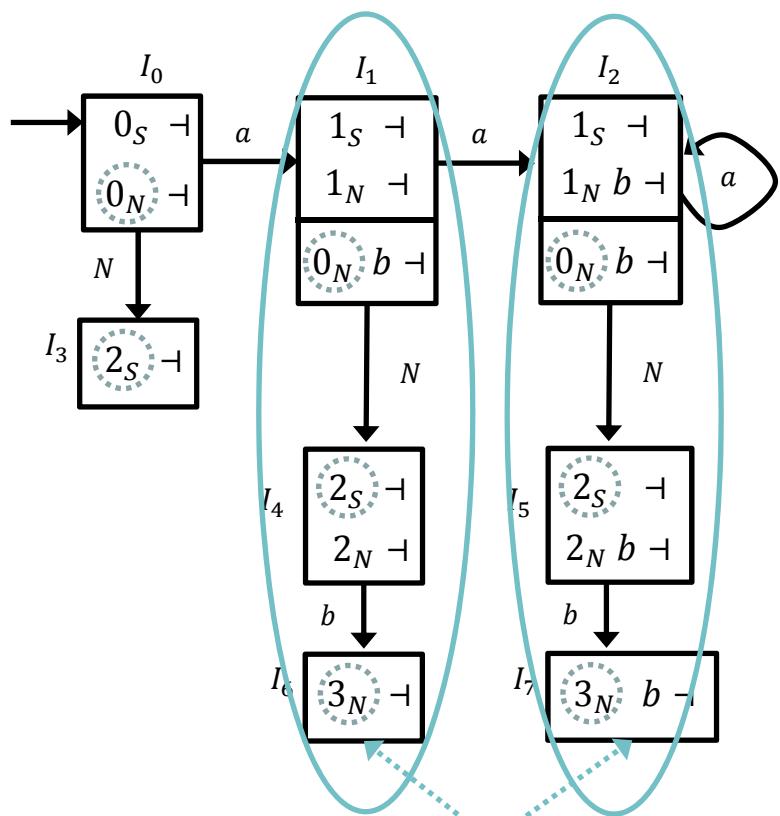


Example (more interesting but still simple) with an infinite language

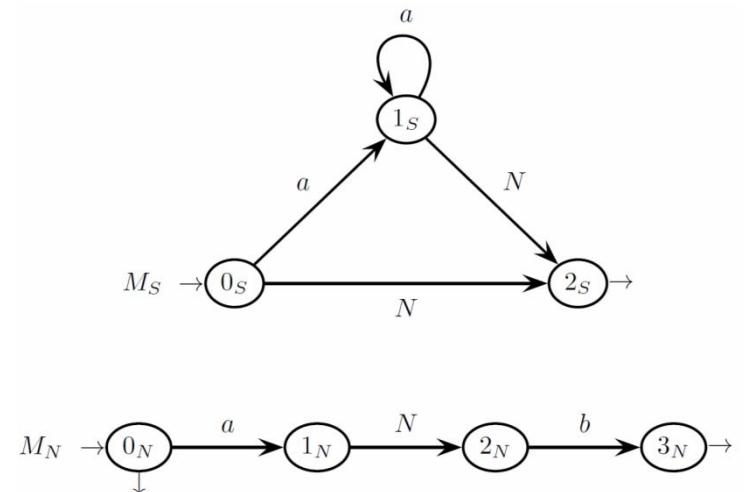
$$\left\{ \begin{array}{l} S \rightarrow a^* N \\ N \rightarrow a \ N \ b \mid \varepsilon \end{array} \right.$$

$L = \{a^n b^m \mid n \geq m \geq 0\}$

pilot has m-states with several items in the base



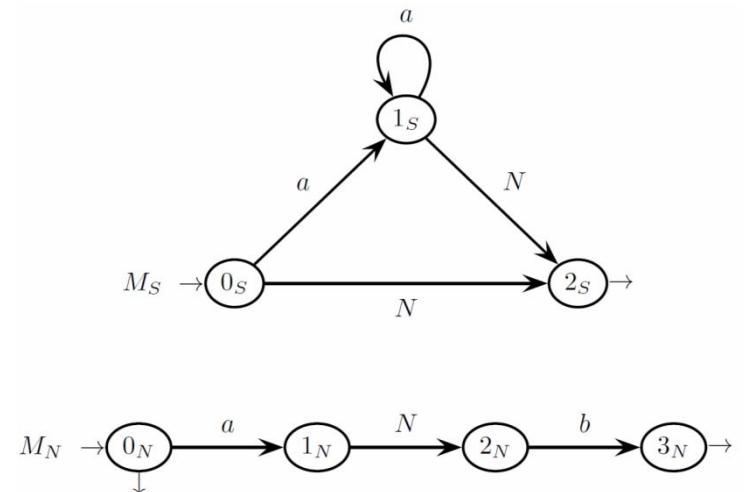
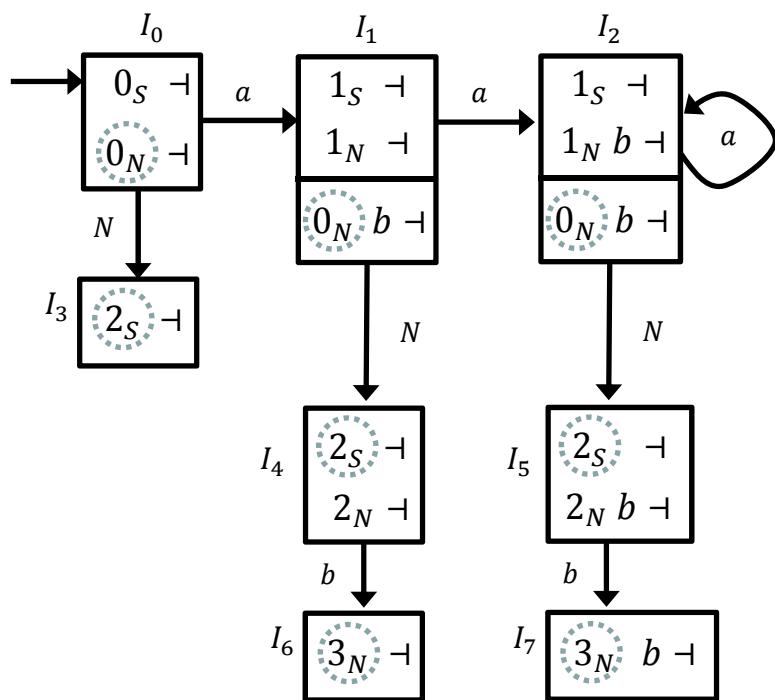
NB: often the pilot automaton has various groups of m-states with the same **kernel** (set of states in the base and closure) but with distinct lookahead sets



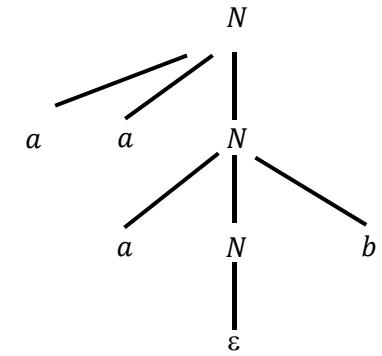
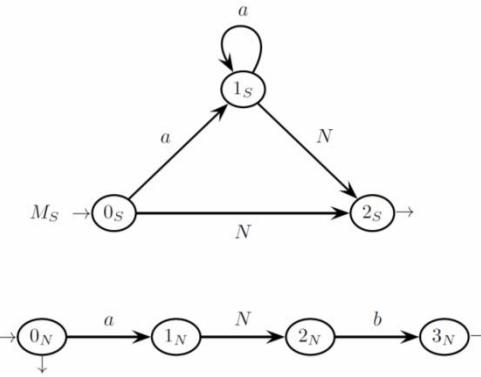
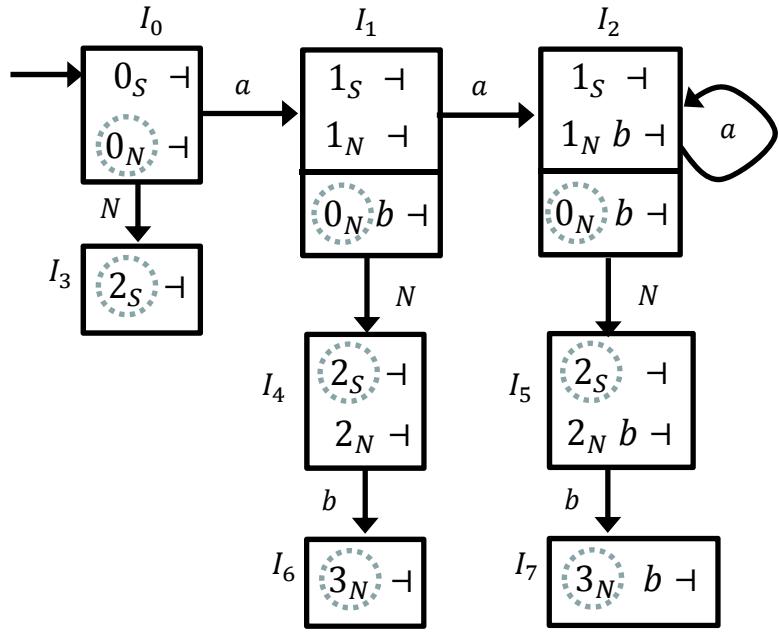
Example (more interesting but still simple) with an infinite language

$$\left\{ \begin{array}{l} S \rightarrow a^* N \\ N \rightarrow a \ N \ b \mid \varepsilon \end{array} \right. \quad L = \{a^n \ b^m \mid n \geq m \geq 0\}$$

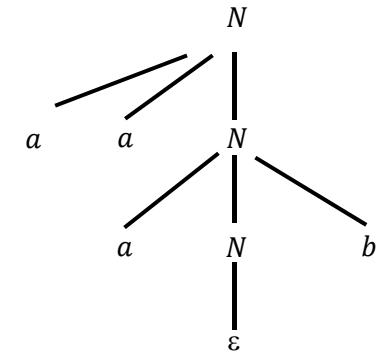
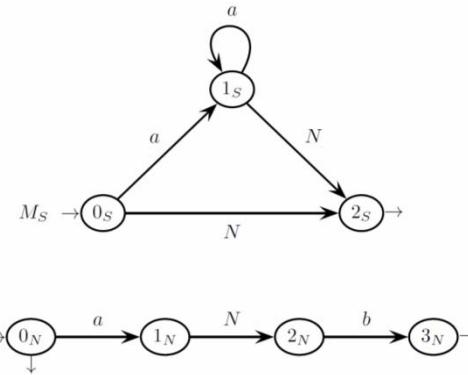
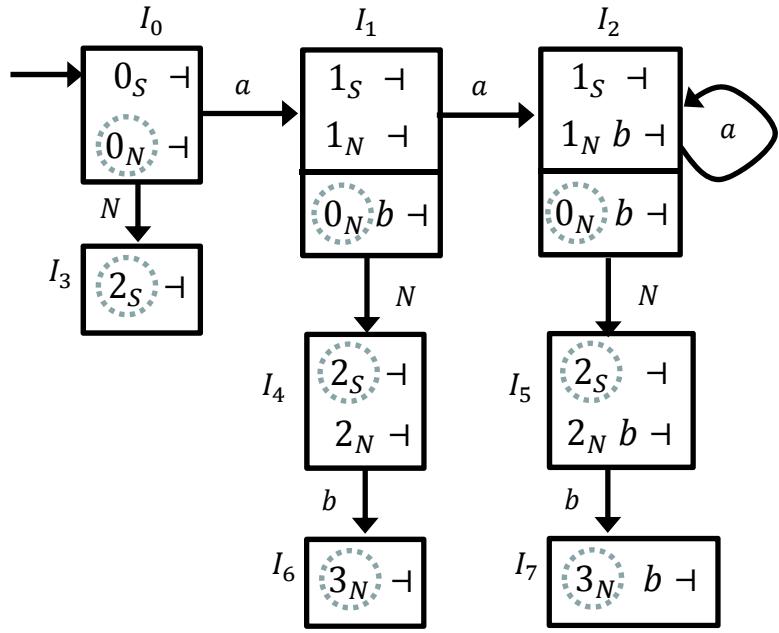
pilot has m-states with several items in the base



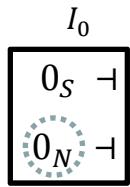
multiple candidates in the base of an m-state \Rightarrow PDA must perform several analysis attempts in parallel

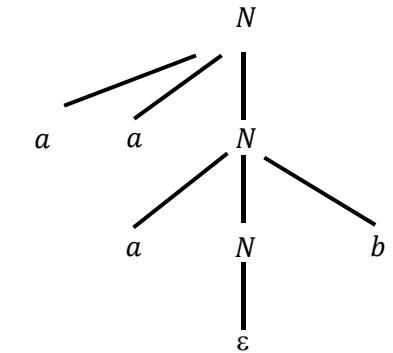
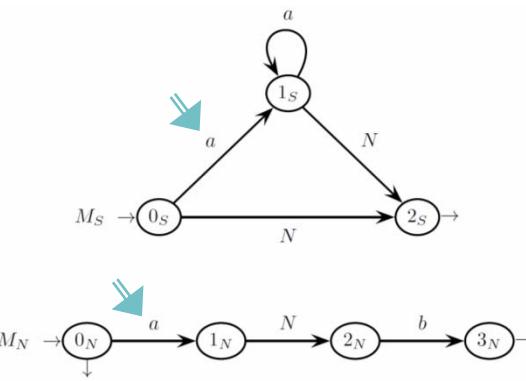
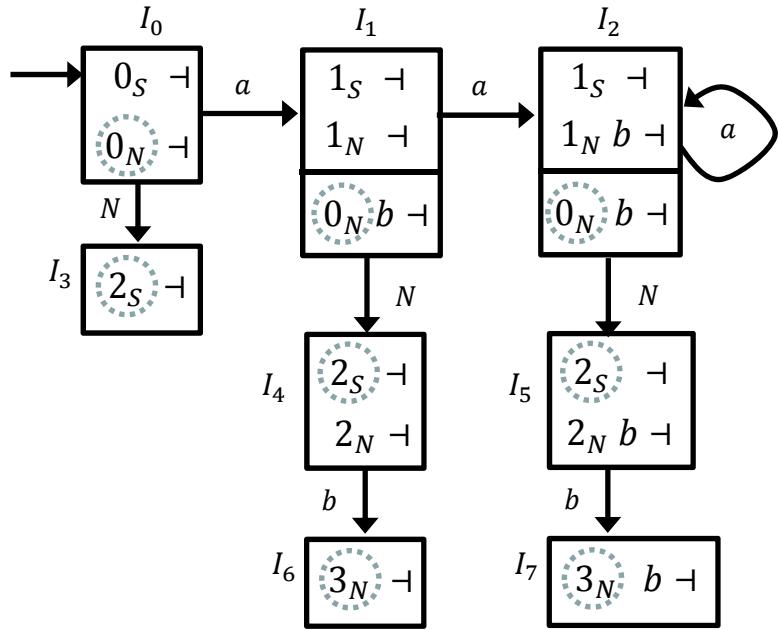


Example: analysis of string ***aaab*** \dashv

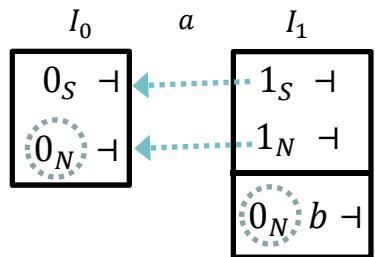


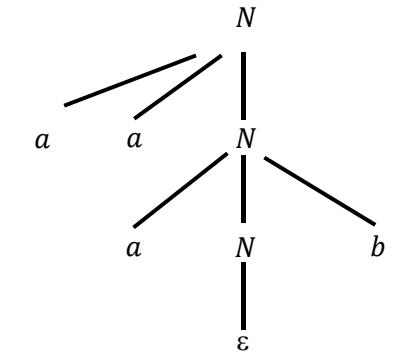
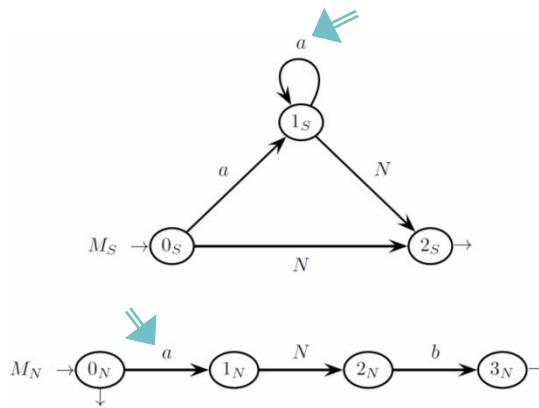
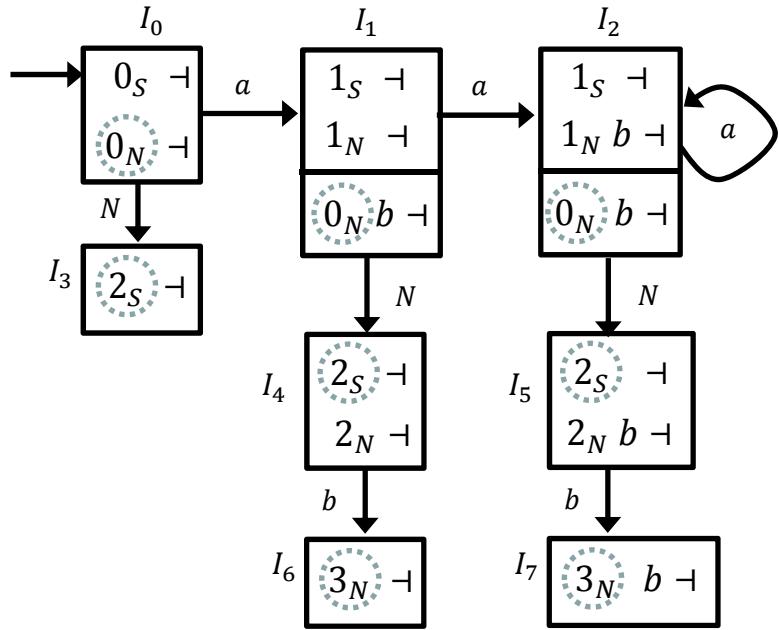
Example: analysis of string $aaab \dashv$



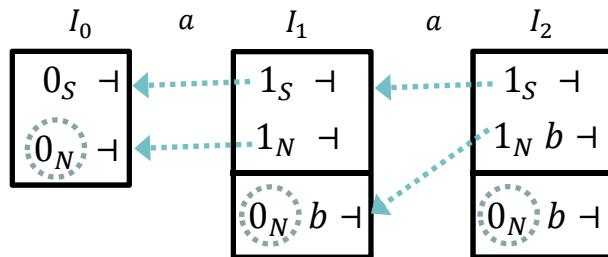


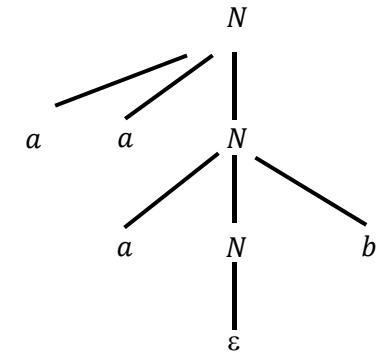
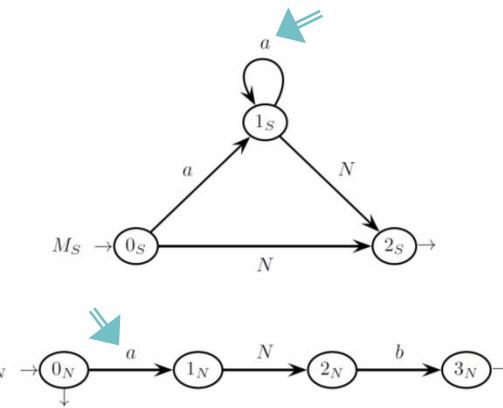
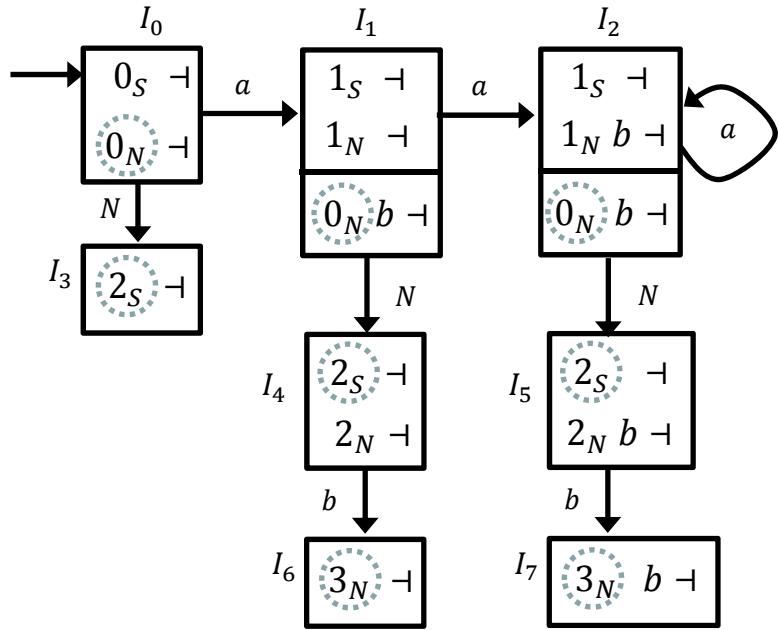
Example: analysis of string $aaab \dashv$



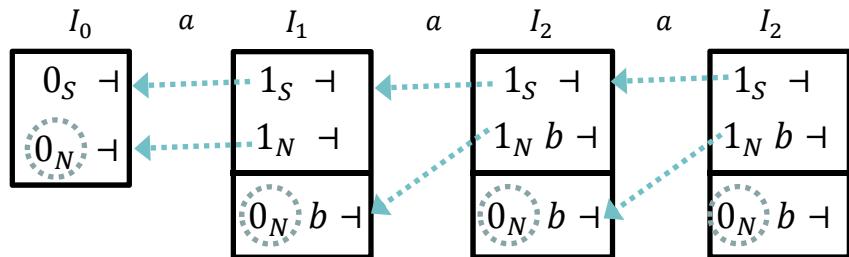


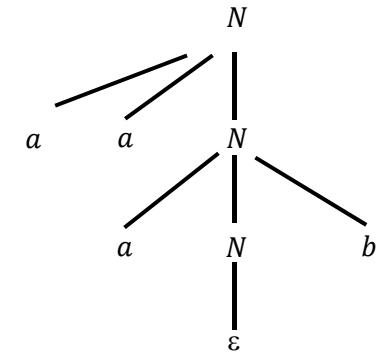
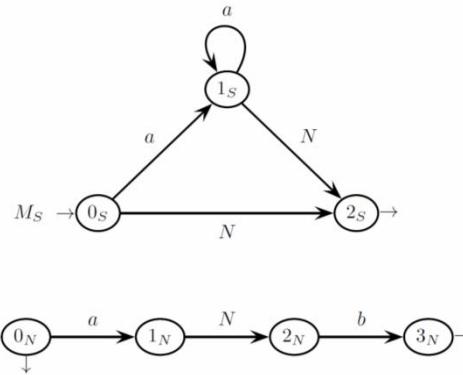
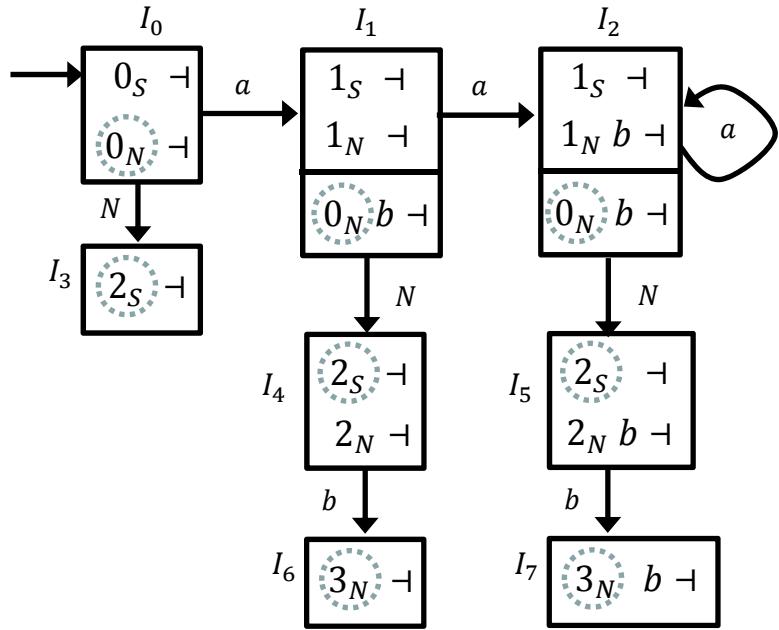
Example: analysis of string $aaab \dashv$



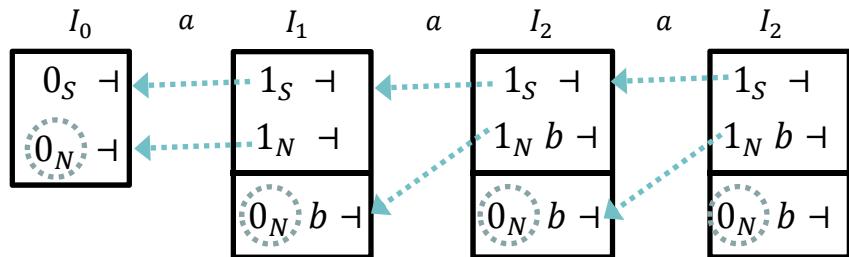


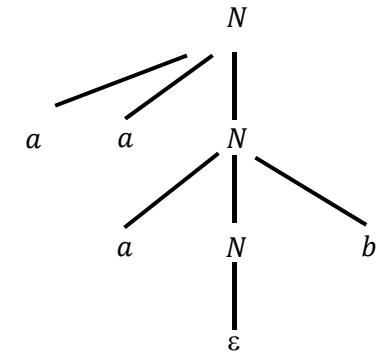
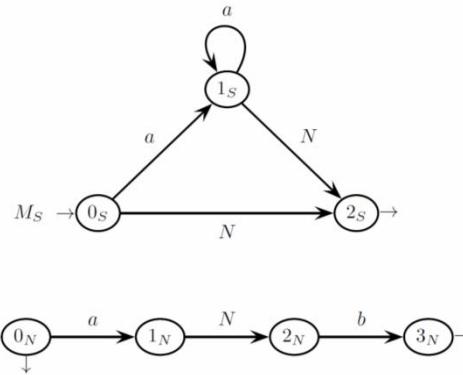
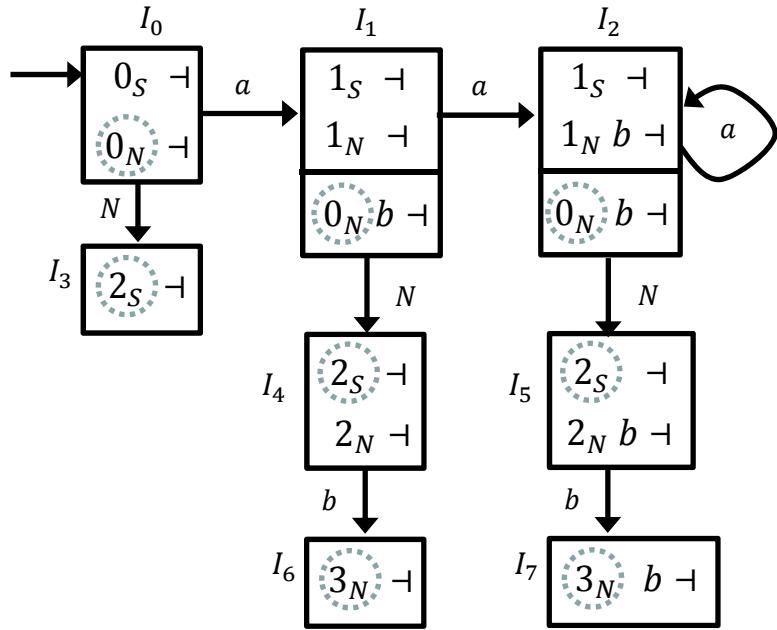
Example: analysis of string ***aaab*** ⊢





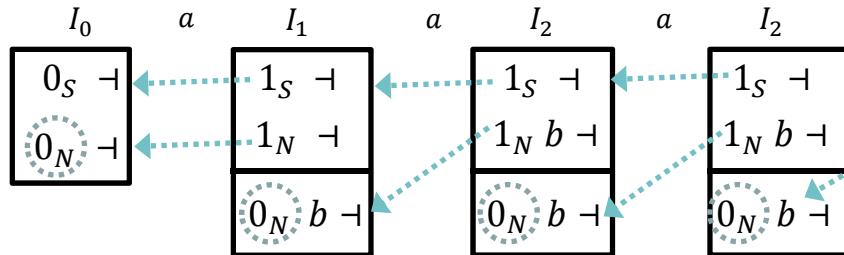
Example: analysis of string ***aaab*** ⊢

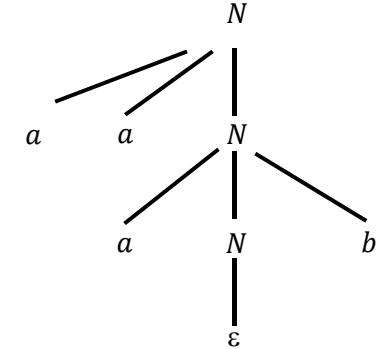
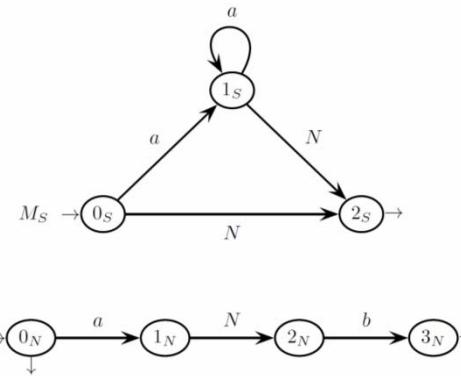
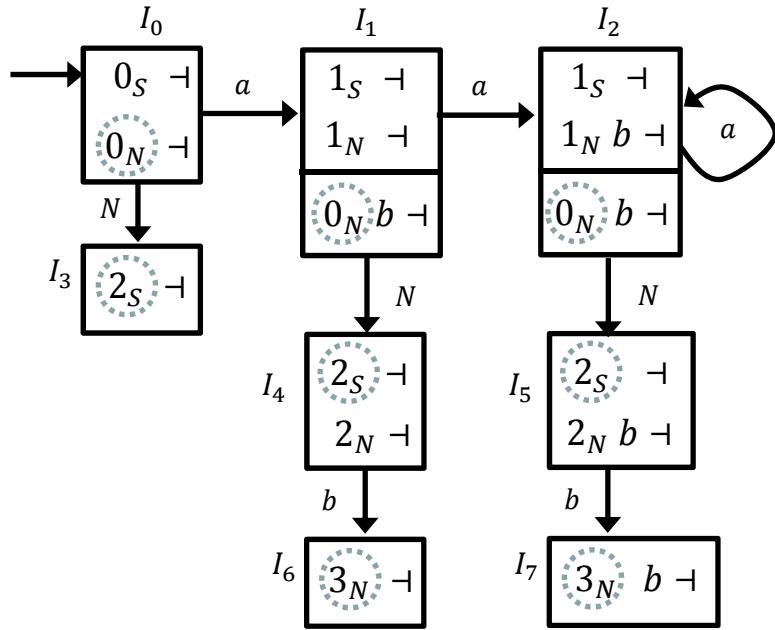




Example: analysis of string $aabb \dashv$

lookahead = input: reduce $\epsilon \rightarrow N$

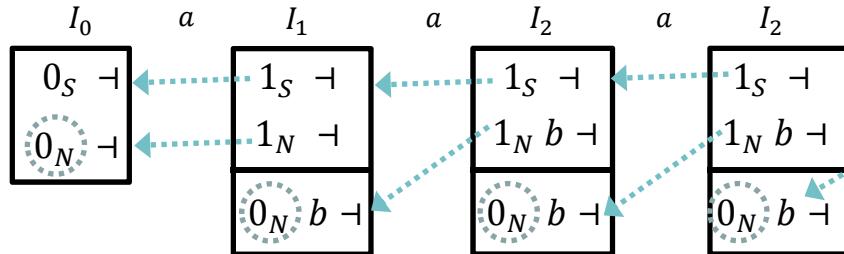


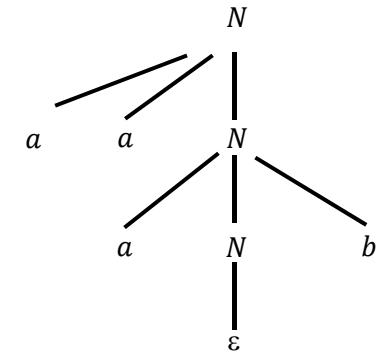
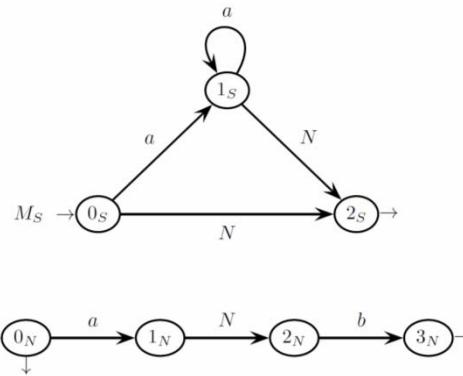
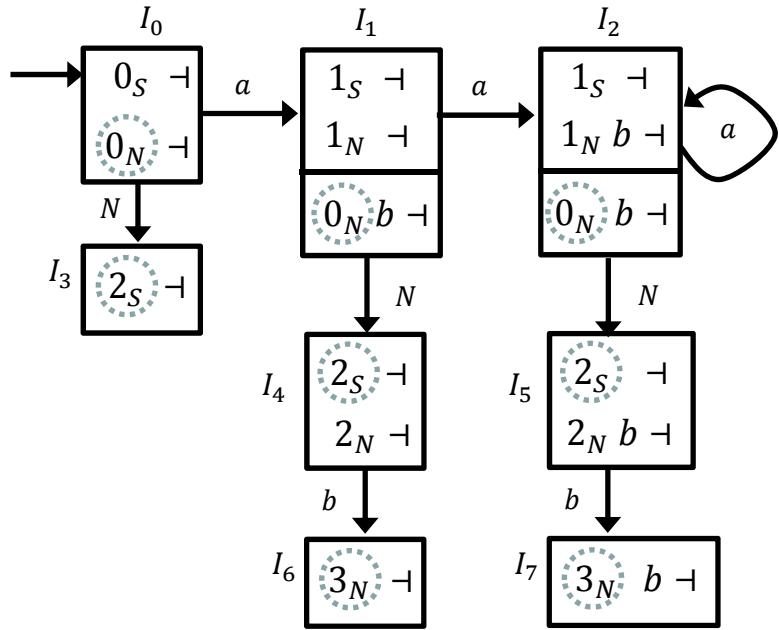


NB: length of the reduction handle (ε) is 0:
 ε (i.e., nothing) is popped from the stack

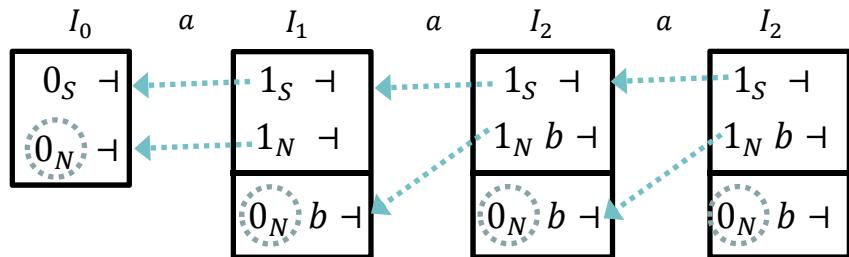
Example: analysis of string $aaab \dashv$

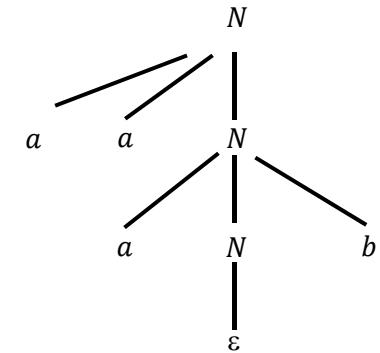
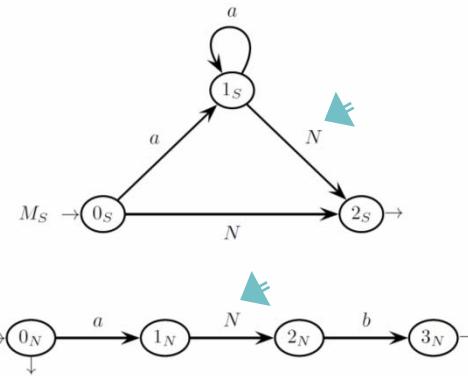
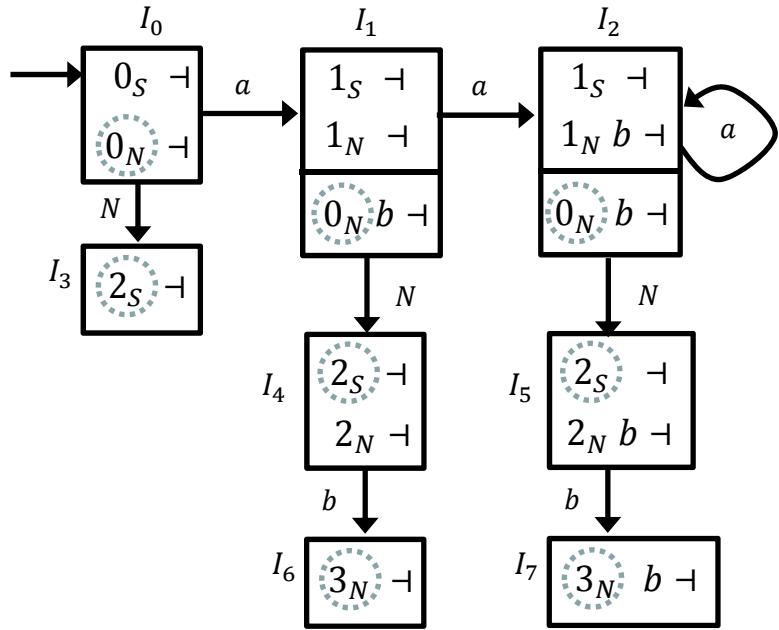
lookahead = input: reduce $\varepsilon \rightarrow N$



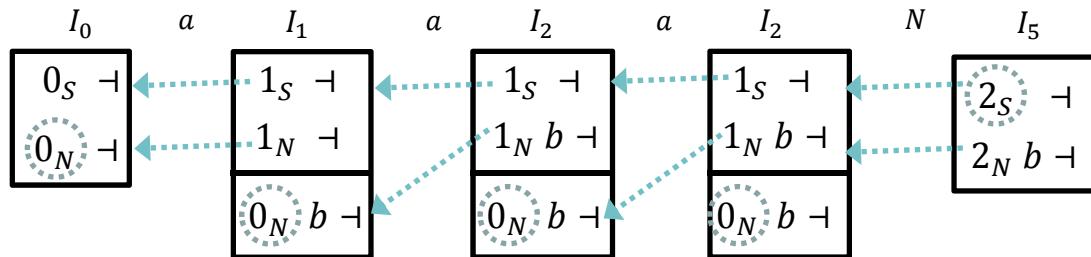


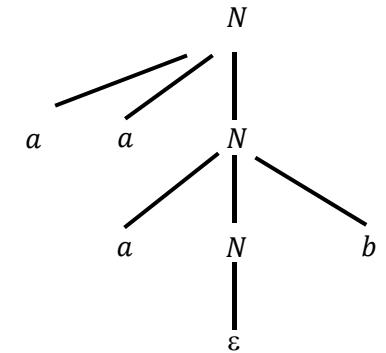
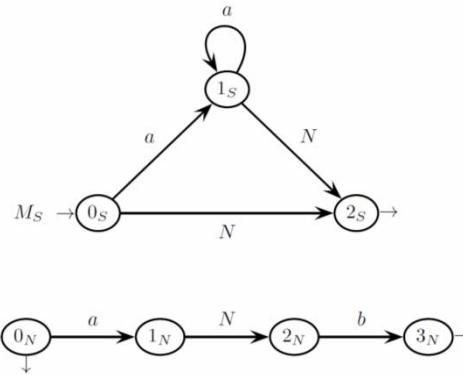
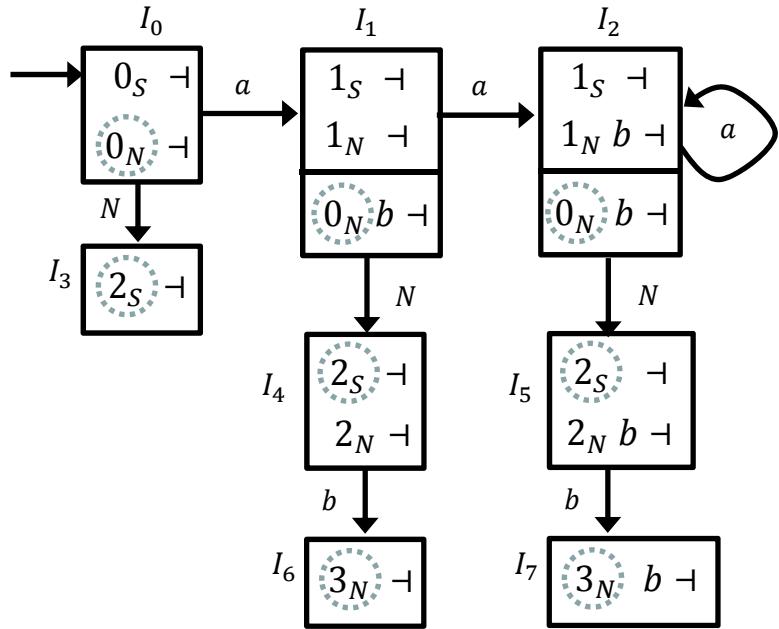
Example: analysis of string ***aaab*** ⊢



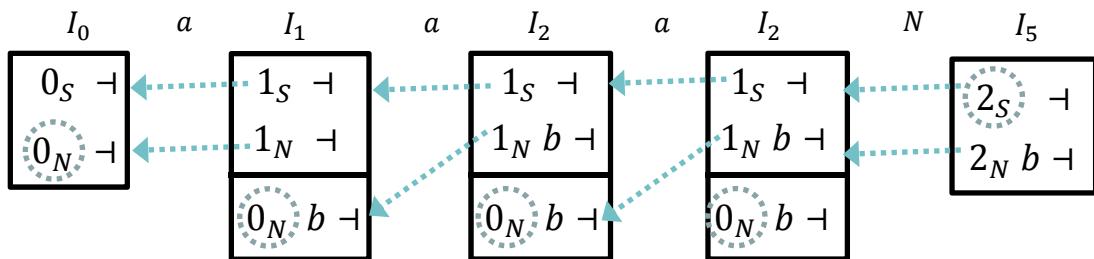


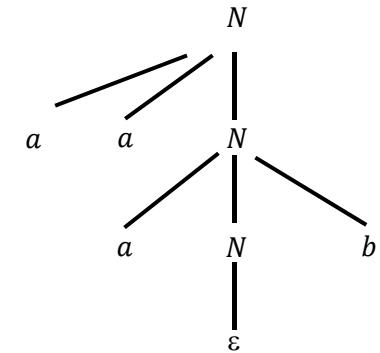
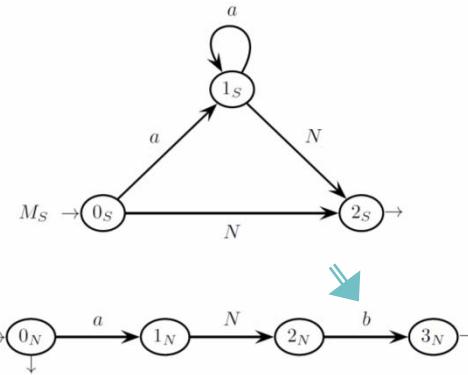
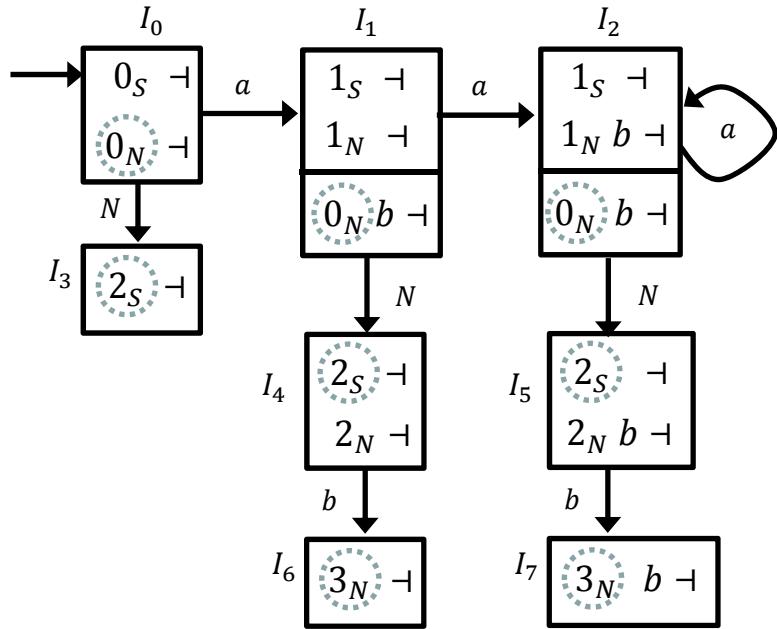
Example: analysis of string $aaab \dashv$



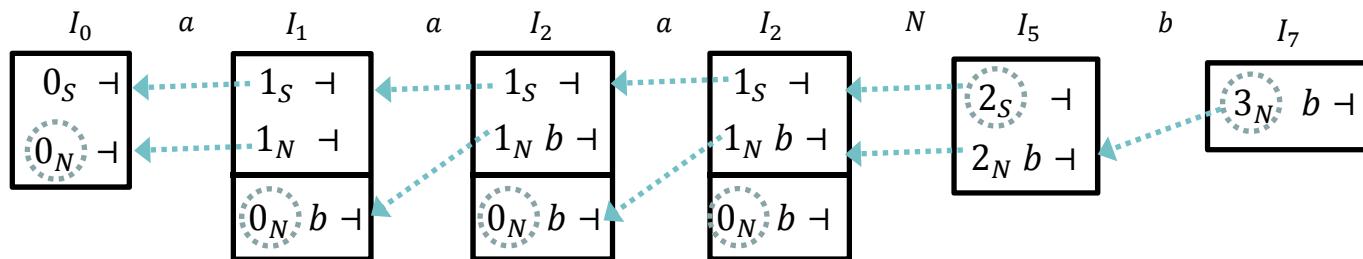


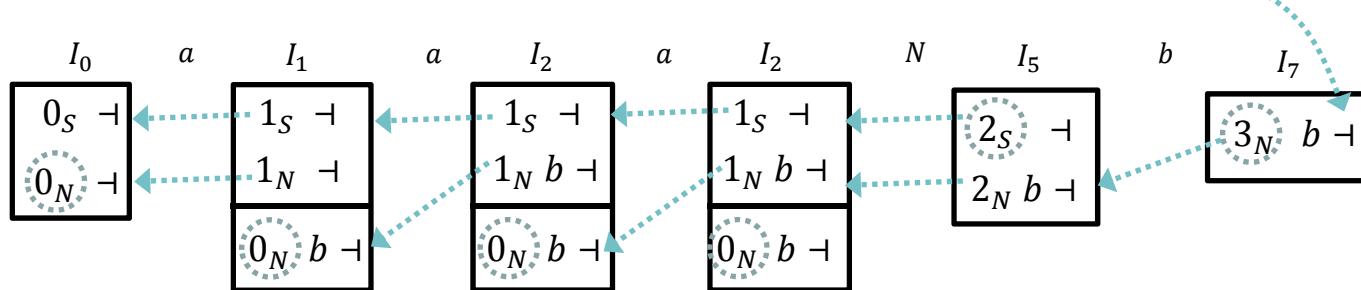
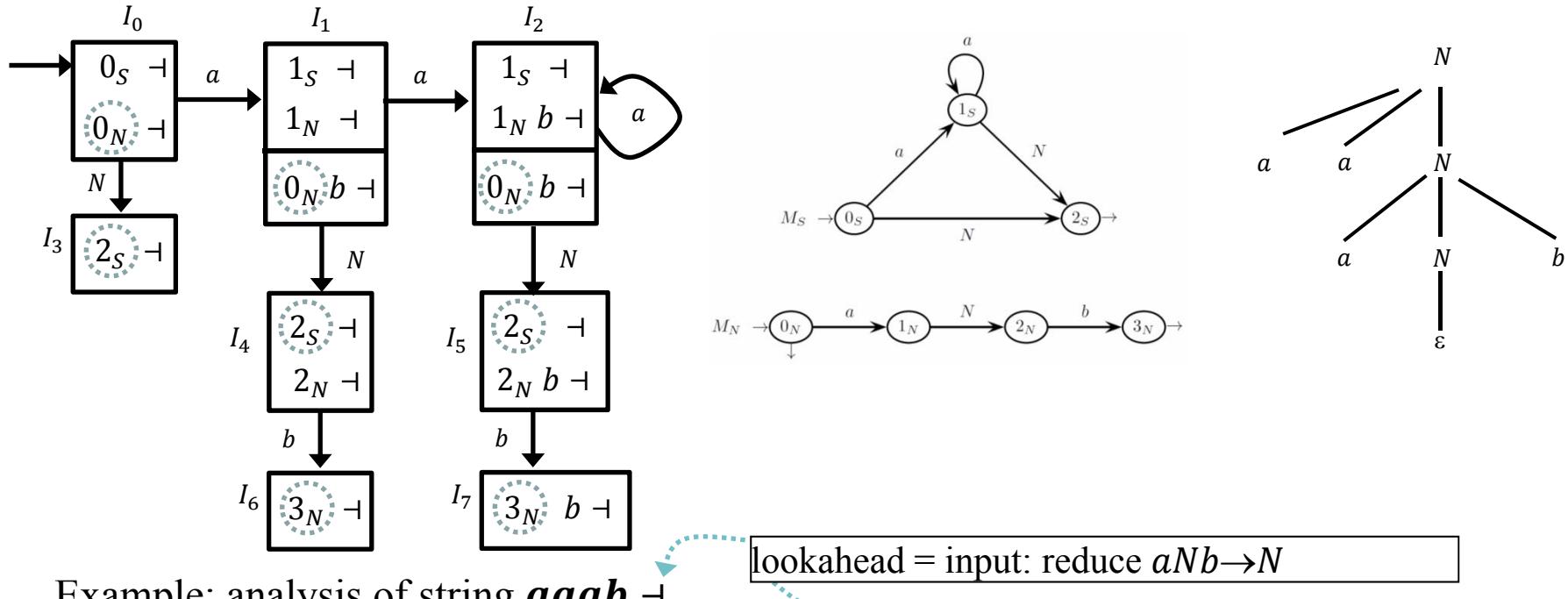
Example: analysis of string $aaab \dashv$

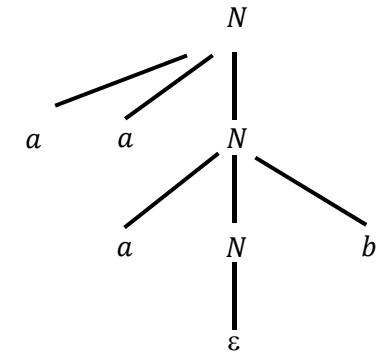
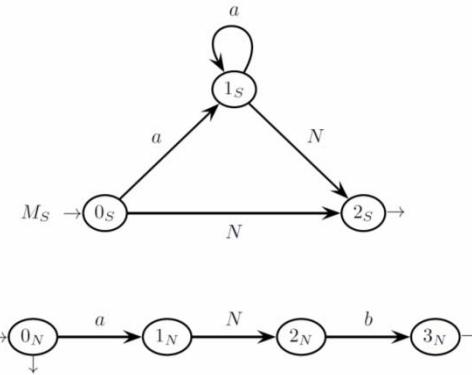
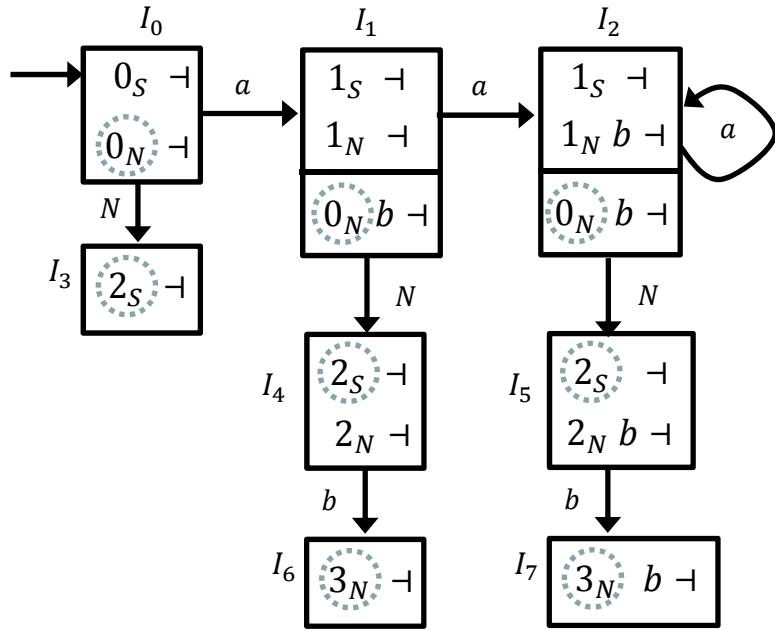




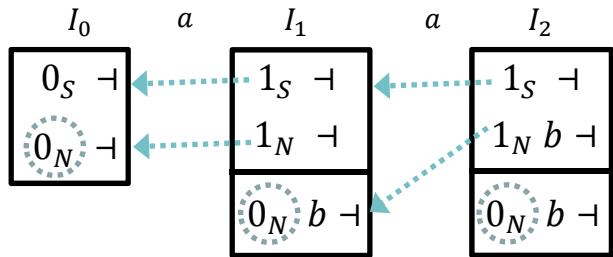
Example: analysis of string $aaab \dashv$

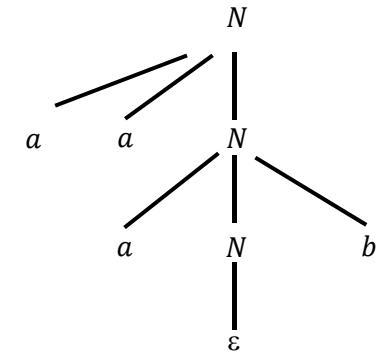
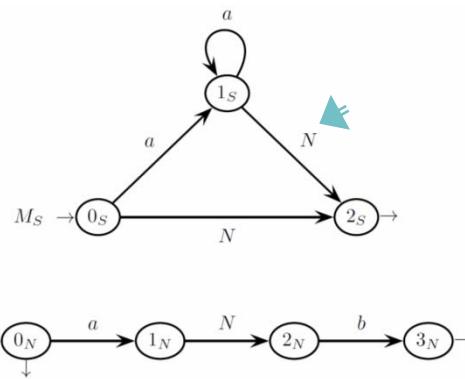
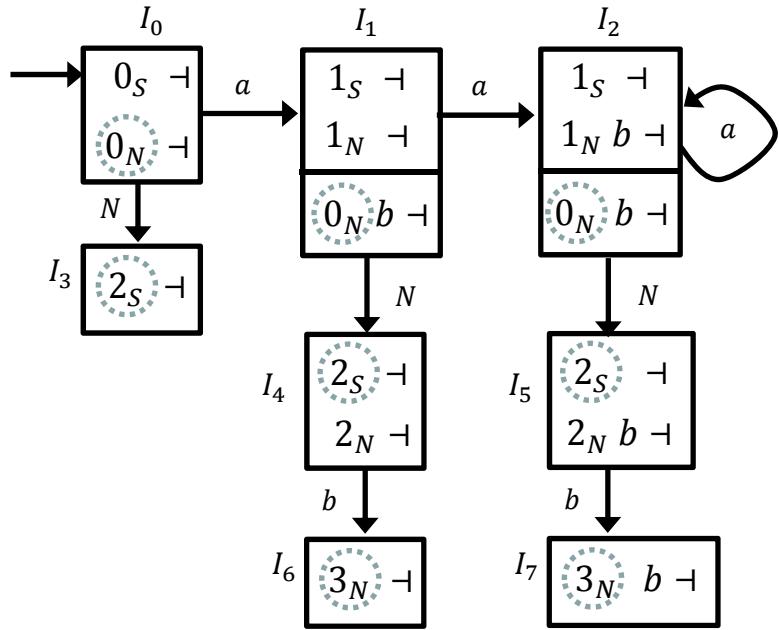




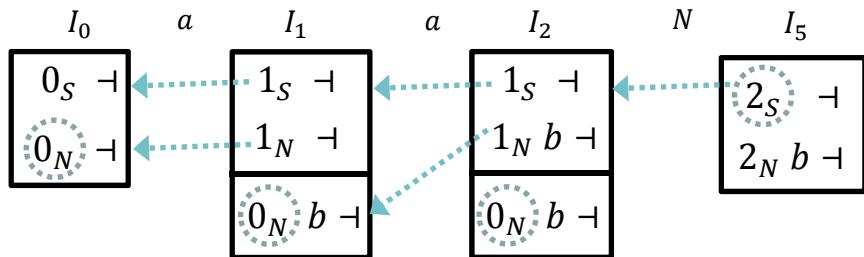


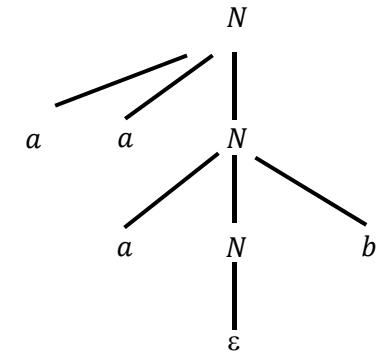
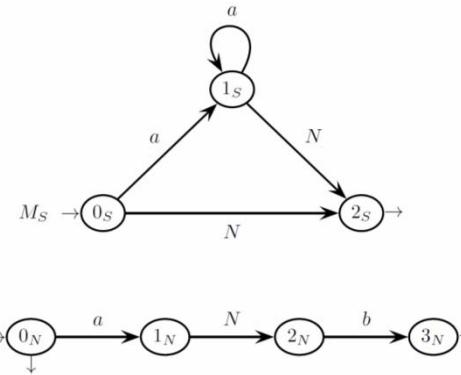
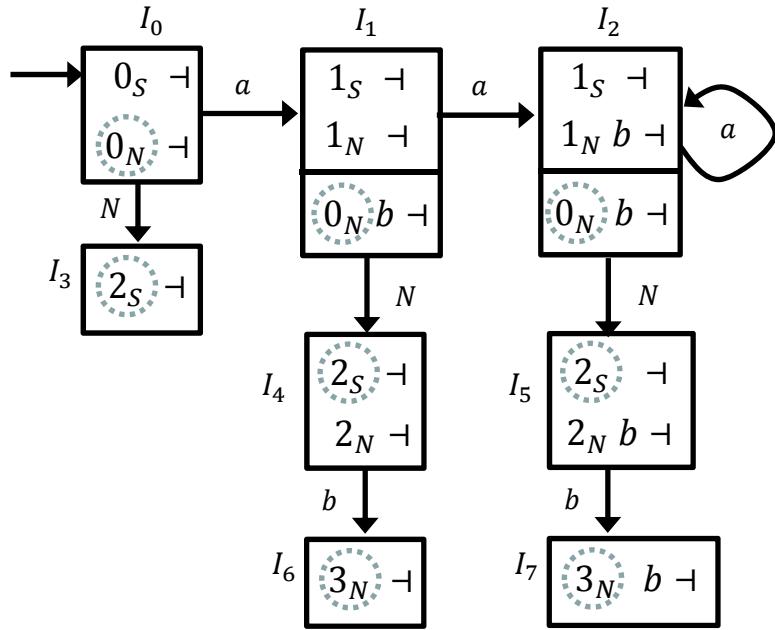
Example: analysis of string $aaab \dashv$



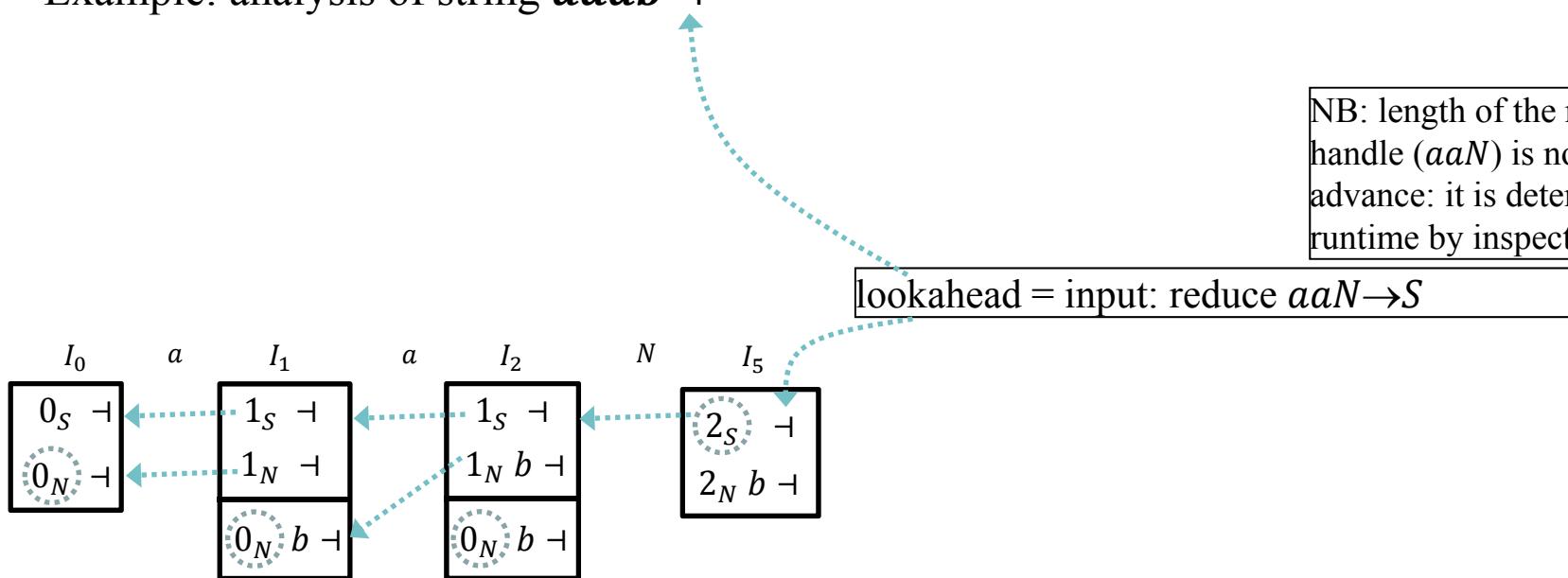


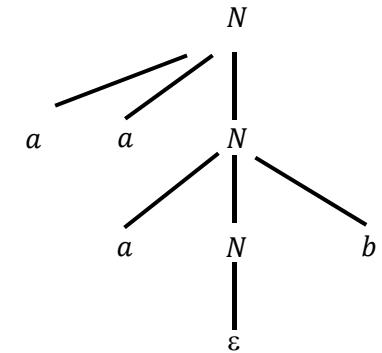
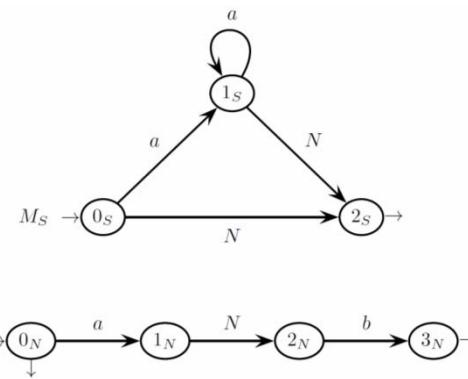
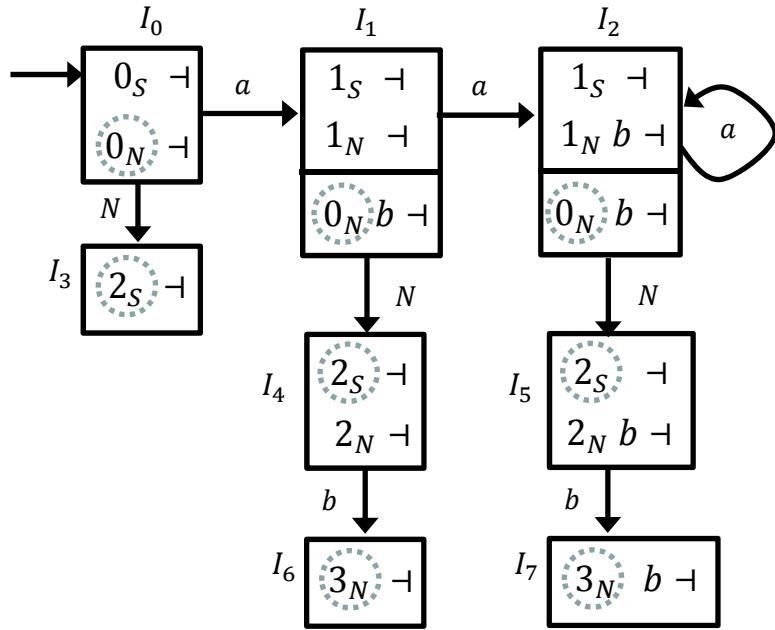
Example: analysis of string $aaab \dashv$



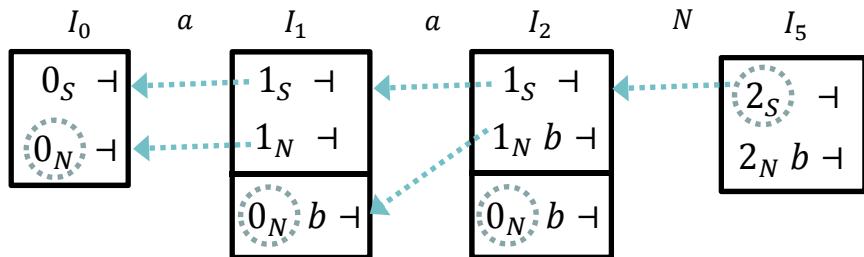


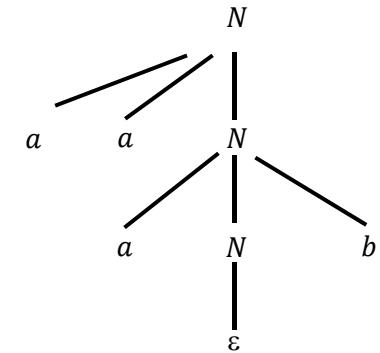
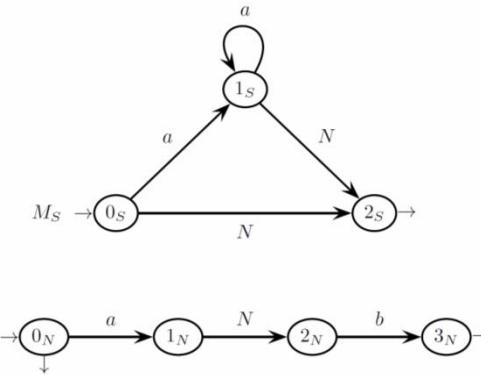
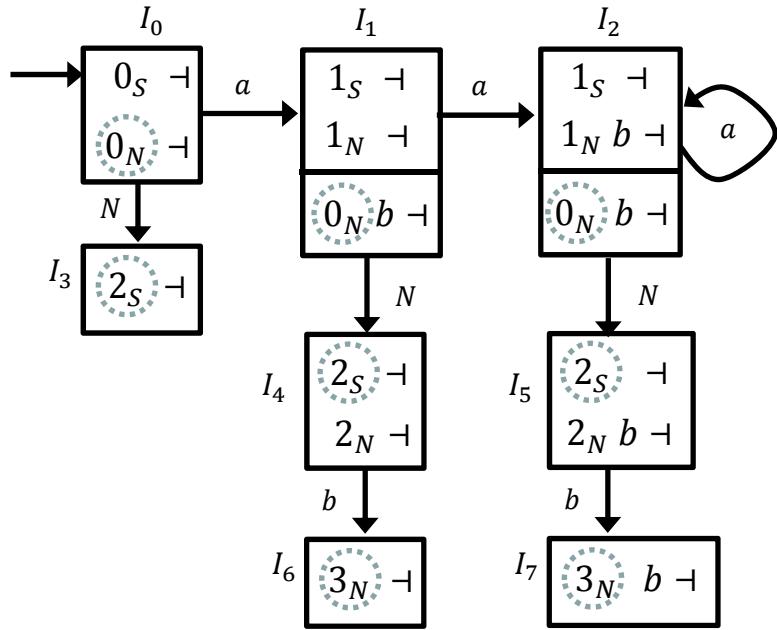
Example: analysis of string $aaab \dashv$





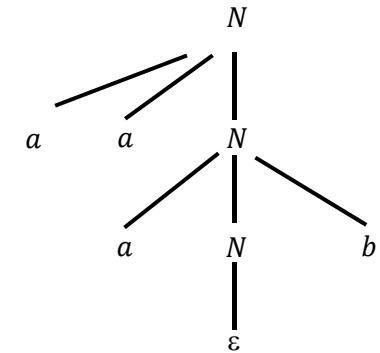
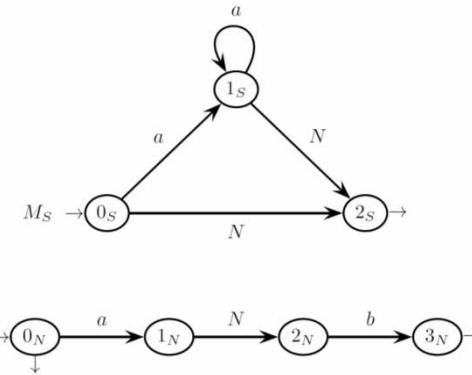
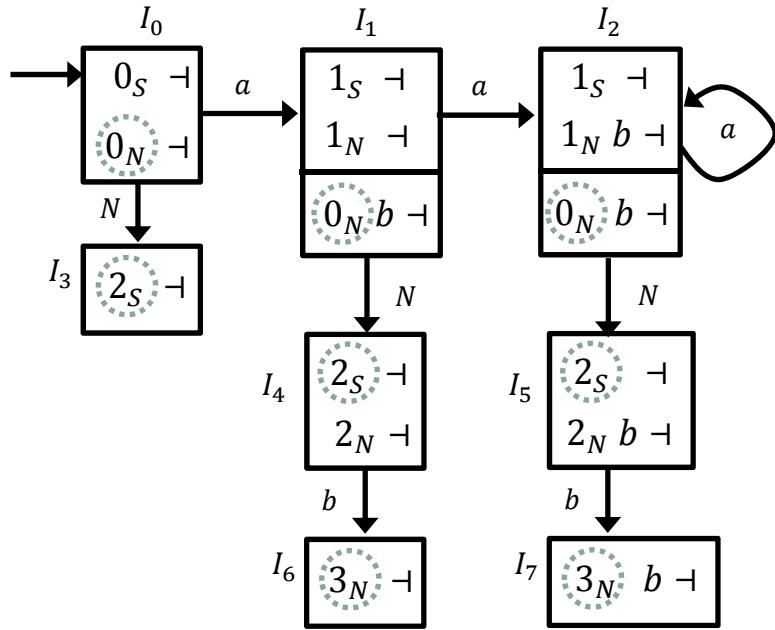
Example: analysis of string $aaab \dashv$





Example: analysis of string $aaab \dashv$





Example: analysis of string ***aaab*** ⊢

Bottom-up Syntax Analysis– *ELR(1)**

PART 2

* *ELR(1)* = Left scan Rightmost derivation with lookahead 1 for Extended grammars

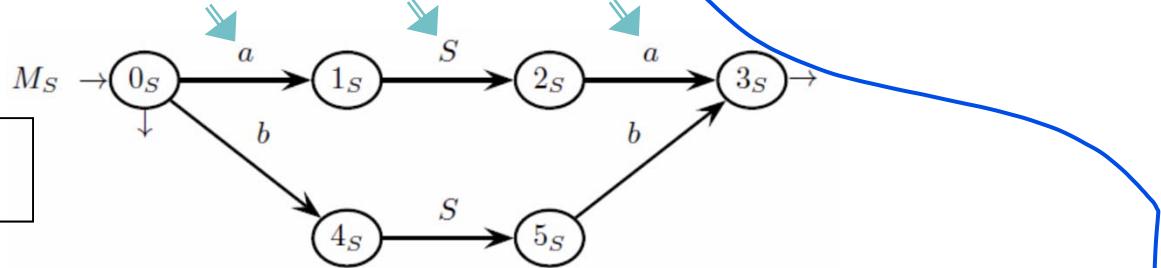
Prof. A. Morzenti

What can go wrong in the $ELR(1)$ analysis: (1) shift-reduction conflict

$$S \rightarrow aSa \mid bSb \mid \varepsilon$$

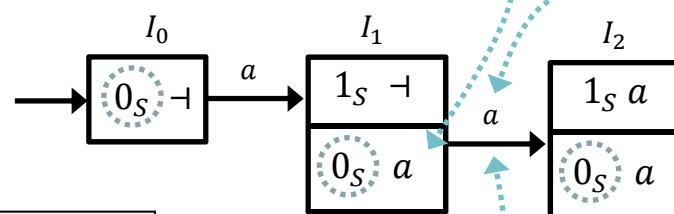
the grammar is **nondeterministic**:

the center of the string cannot be identified

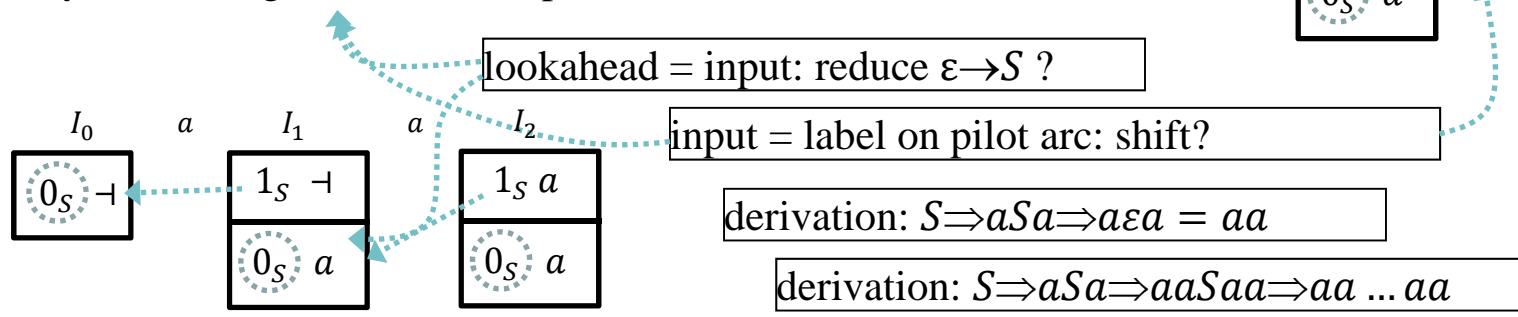


let us build only one part of the pilot (the one for the a 's; the other part for the b 's is symmetric)

shift-reduction conflict in I_1



analysis of string $aa \dots \dashv$ (ex. input $aa \dashv$ or $aa \dots aa \dashv$)



cannot distinguish the case in which the S is empty so we are reading the a between $2s$ and $3s$ or the S is producing so the a is the one between $0s$ and $1s$

Intuitively: PDA, after shifting an ' a ', cannot choose between

- reduce because it has reached the center of the palindrome (as if the string was $aa \dashv$)
- shift because it has not yet reached the center (e.g. if the string was $aa \dots aa \dashv$)

but there is **no way to know it: the language is (intrinsically) nondeterministic**

it does not admit a $ELR(1)$ grammar

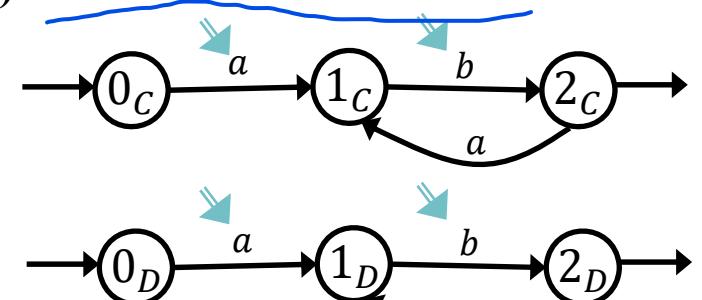
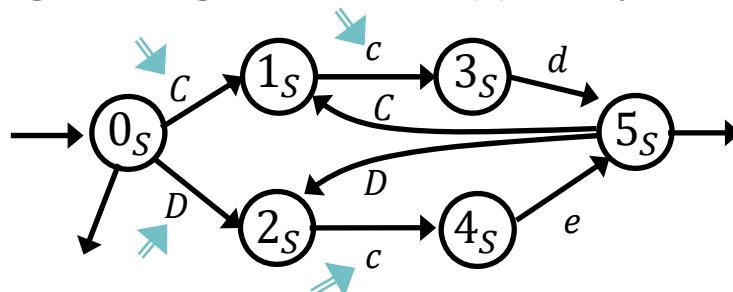
not that in this case the conflict is solvable with longer lookahead

What can go wrong in the $ELR(1)$ analysis: (2) reduce-reduce conflict

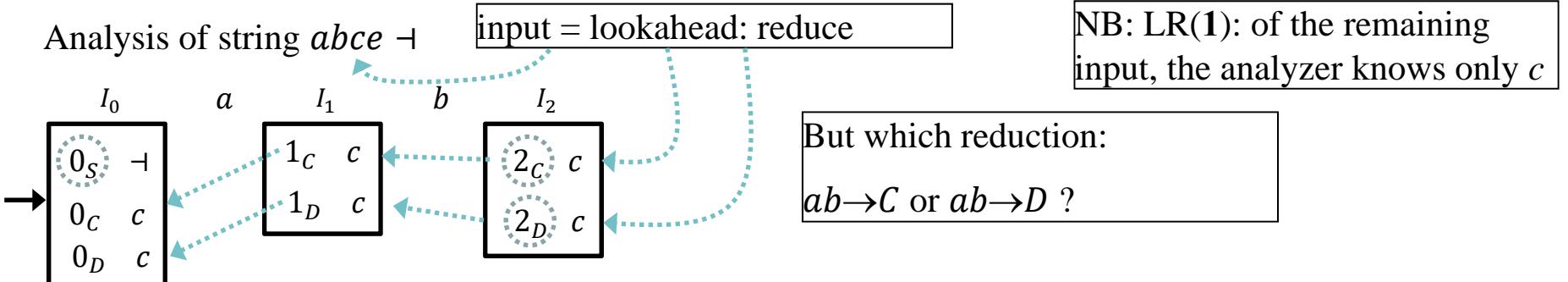
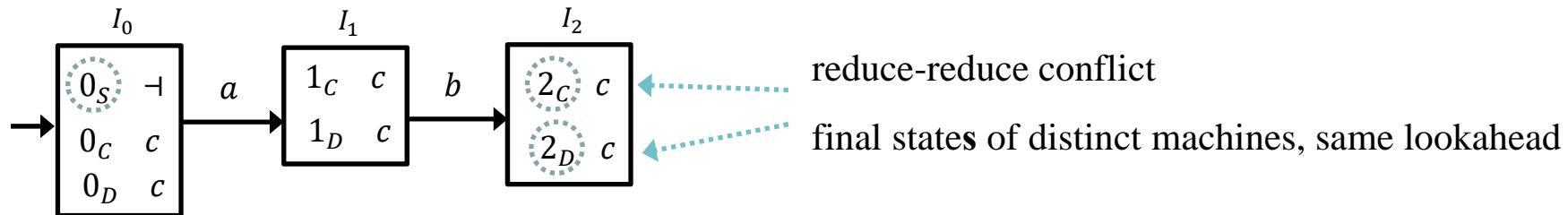
$$S \rightarrow (Ccd \mid Dce)^*$$

$$C \rightarrow (ab)^+$$

$$D \rightarrow (ab)^+$$

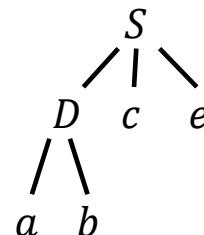


let us build only the pilot portion that shows the conflict



NB: the string $abce \dashv$ is not ambiguous

Derivation: $S \Rightarrow Dce \Rightarrow abce$, syntax tree



One can remedy (in this case, but not always) by modifying the grammar

The c part, common to the two alternatives, can be incorporated into the two rules of C and D

Analyzer can distinguish the two alternatives, lookahead now is d (for C) and e (for D)

$$\begin{array}{l} S \rightarrow (Ccd \mid Dce)^* \\ C \rightarrow (ab)^+ \\ D \rightarrow (ab)^+ \end{array} \quad \xrightarrow{\hspace{1cm}} \quad \begin{array}{l} S \rightarrow (Cd \mid De)^* \\ C \rightarrow (ab)^+c \\ D \rightarrow (ab)^+c \end{array}$$

BTW: In fact, by substituting C and D into S one can notice that the language is regular

$$S \rightarrow ((ab)^+cd \mid (ab)^+ce)^* \quad \text{This one is a } ELR(1) \text{ grammar. Exercise: build the pilot:}$$

$$S \rightarrow ((ab)^+c(d \mid e))^*$$

One can see that it is isomorphic to the machine of S

In general: when a grammar is not $ELR(1)$, then its language either

- **may** admit a different grammar which is $ELR(1)$, or
- **may not** admit any, e.g. if it is inherently ambiguous or nondeterministic

Let us provide a formalization/encoding of the pilot automaton construction
 (see the textbook for further details)

DEF: an **item** is a pair $\langle q, a \rangle \in Q \times (\Sigma \cup \{\dashv\})$

for ease of writing, items with the same state are grouped

$$\langle q, \{a_1, a_2, \dots, a_k\} \rangle \Leftrightarrow \{\langle q, a_1 \rangle, \langle q, a_2 \rangle, \dots, \langle q, a_k \rangle\}$$

DEF: **shift** operation

NE: with the shift operation the lookahead does not change: it changes only when the **closure** operation is performed

$$\begin{cases} \vartheta(\langle p_A, \rho \rangle, X) = \langle q_A, \rho \rangle & \text{if in } M_A \text{ there is a trans. } p_A \xrightarrow{X} q_A \\ \emptyset & \text{otherwise} \end{cases}$$

The shift operation is defined also for m-states (seen as sets I of items) in the obvious way

$$\vartheta(I, X) = \{\vartheta(c, X) \mid c \in I\}$$

We assume the function $\text{closure}(C)$ is defined

it computes the **closure** of a set C of items with lookahead

(we do not report yet the detailed definition of **closure**)

CONSTRUCTION OF THE PILOT AUTOMATON \mathcal{P}

$\mathcal{P} = \langle R, \Sigma \cup V, \vartheta, I_0 \rangle$ where

- R set of states (m-states: every m-state $\in R$ is a set of items)
- alphabet $\Sigma \cup V$ is that of the grammar and of the machine net
- initial state $I_0 = \text{closure}(\langle 0_S, \{\dashv\} \rangle)$ (0_S initial state of the machine for axiom S)
- set R and transition function ϑ are computed by the following procedure

```

 $R' := \{ I_0 \}$                                 -- prepare the initial m-state  $I_0$ 
-- loop that updates the m-state set and the state-transition function
do
     $R := R'$                                 -- update the m-state set  $R$ 
    -- loop that computes possibly new m-states and arcs
    for (each m-state  $I \in R$  and symbol  $X \in \Sigma \cup V$ ) do
        -- compute the base of a m-state and its closure
         $I' := \text{closure}(\vartheta(I, X))$ 
        -- check if the m-state is not empty and add the arc
        if ( $I' \neq \emptyset$ ) then
            add arc  $I \xrightarrow{X} I'$  to the graph of  $\vartheta$ 
            -- check if the m-state is a new one and add it
            if ( $I' \notin R$ ) then
                add m-state  $I'$  to the set  $R'$ 
            end if
        end if
    end for
    while ( $R \neq R'$ )                                -- repeat if the graph has grown

```

a typical incremental construction: it terminates when nothing new is found

no state is final; the pilot is used to drive the PDA, not to accept strings

We now provide a formalization/encoding of the ***closure*** function
 (for details see the textbook, 2ndEd §4.5.2, or 3rdEd §4.5.3)

closure(C) is the closure of a set C of items with lookahead
 A typical fixpoint computation

$K := C$

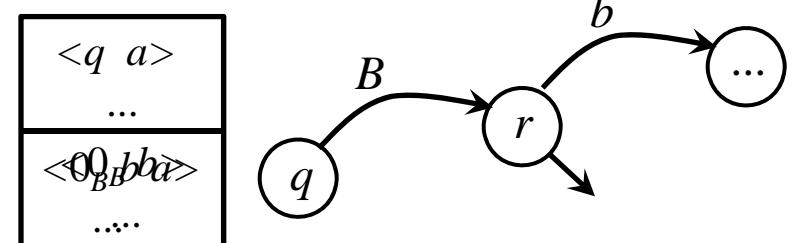
repeat

$K := K \cup \{ \langle 0_B, b \rangle \text{ such that } \exists \langle q, a \rangle \in K \text{ and }$
 $\exists \text{arc } q \xrightarrow{B} r \text{ in } \mathcal{M} \text{ and } b \in \text{Ini}(L(r) \cdot a) \}$

until nothing changes

***closure(C)* := K**

we need the set of characters we can find the move from q to r .



computes the set of the machines reached from q through one or more invocations, with no state transition

for machine M_B (represented by the initial state 0_B) lookahead includes character b
 that can occur when M_B terminates

Effect of term $\text{Ini}(L(r) \cdot a)$ when $\varepsilon \in L(r)$ (e.g., when state r is final): char a is included in the lookahead and the item in the closure above is $\langle 0_B b a \rangle$

IMPORTANT REMARK

So far in the analysis examples we push in the stack the pilot m-states
with the items including the lookahead

In the coming grammar examples (with multiple transitions and convergent arcs)
during the analysis one must separate in the stack the items with same state and different lookahead

Grammar with convergent arcs are rare

usually, in visualizations of analysis, one can keep together items with the same lookahead
if one knows that the grammar does not have any convergent arc

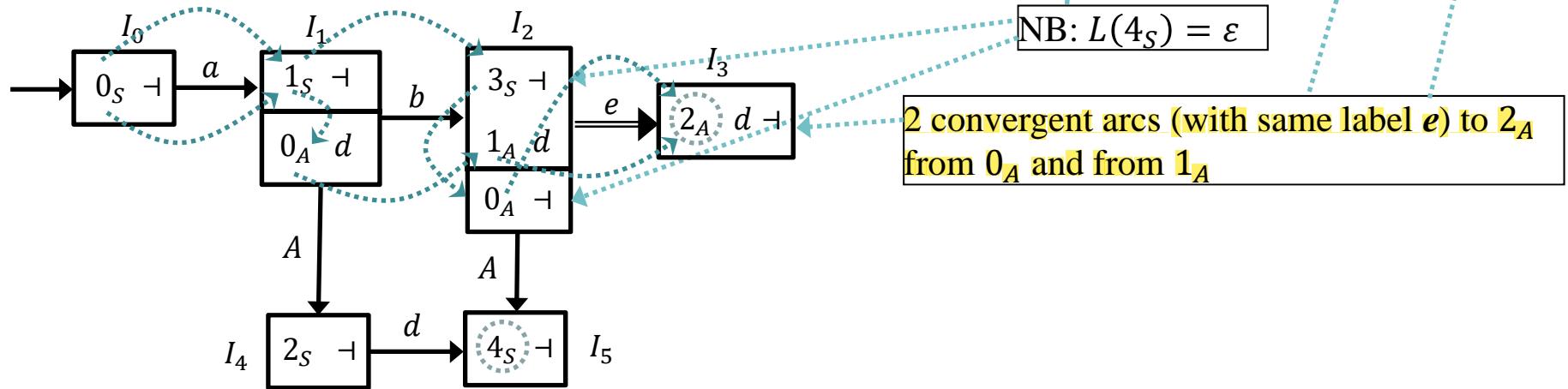
Case where we have multiple transitions with conflictual convergent arcs.

What can go wrong with ELR(1) analysis: (3) **multiple trans. + conflictual convergent arcs**
 (NB: these are contrived cases: rarely met in practice)

Let us first consider a case with no conflicts

$$\begin{aligned} S &\rightarrow a(Ad \mid bA) \\ A &\rightarrow e \mid (be) \end{aligned}$$

construction of the pilot:

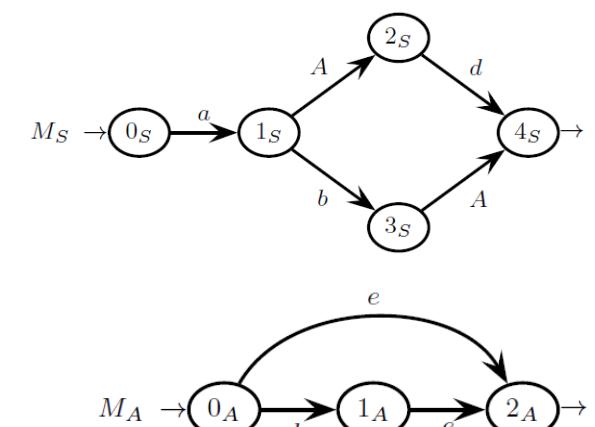
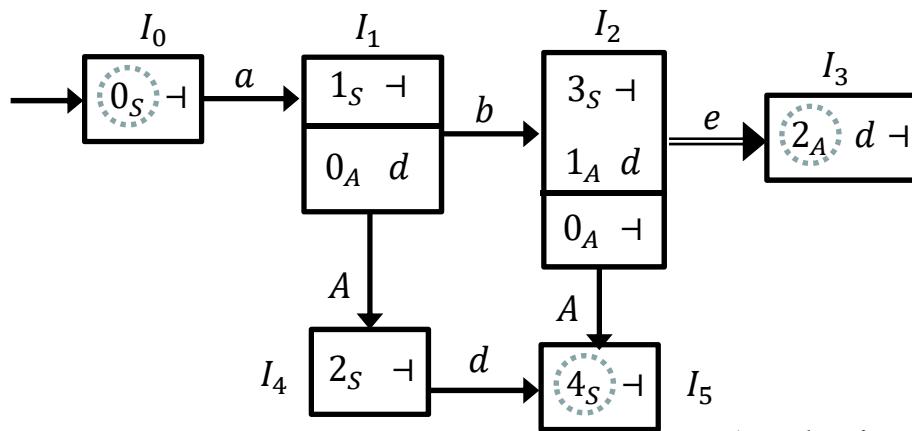


In the pilot there are two paths from 0_S to 2_A :

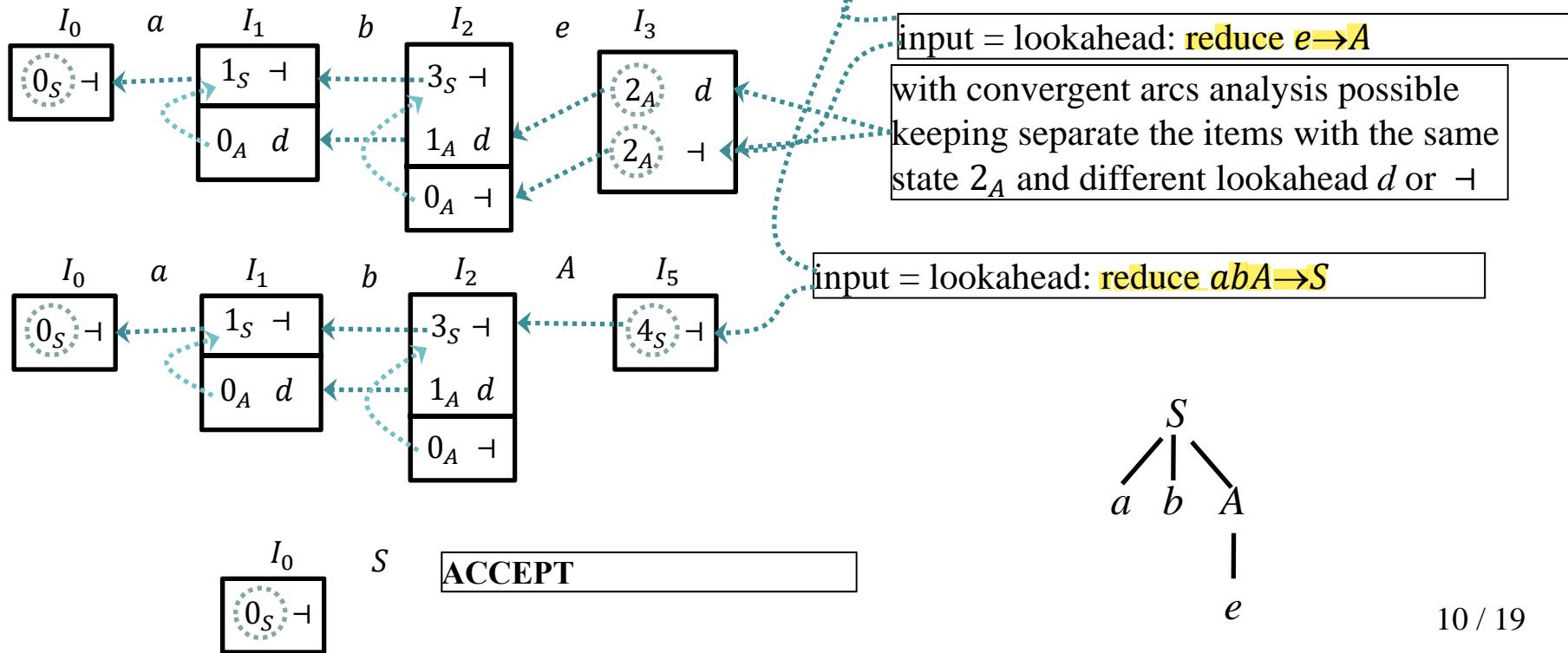
- $0_S \xrightarrow{a} 1_S \xrightarrow{\epsilon} 0_A \xrightarrow{b} 1_A \xrightarrow{e} 2_A$ corresponds to prefix abe of string $abed$, derivation $S \Rightarrow aAd \Rightarrow abed$
- $0_S \xrightarrow{a} 1_S \xrightarrow{b} 3_S \xrightarrow{\epsilon} 0_A \xrightarrow{e} 2_A$ corresponds to an entire string abe , derivation $S \Rightarrow abA \Rightarrow abe$

The two cases are distinguished by the lookahead (d or ϵ)

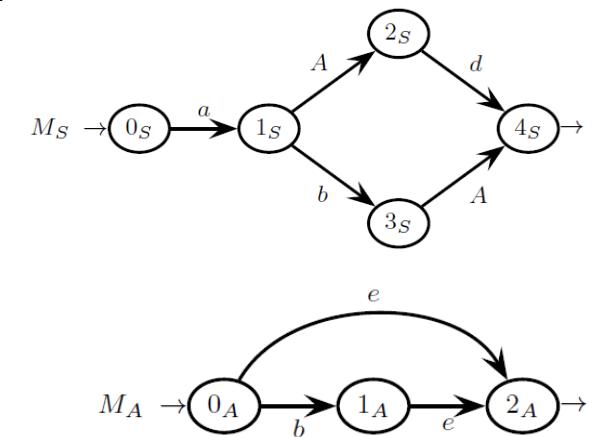
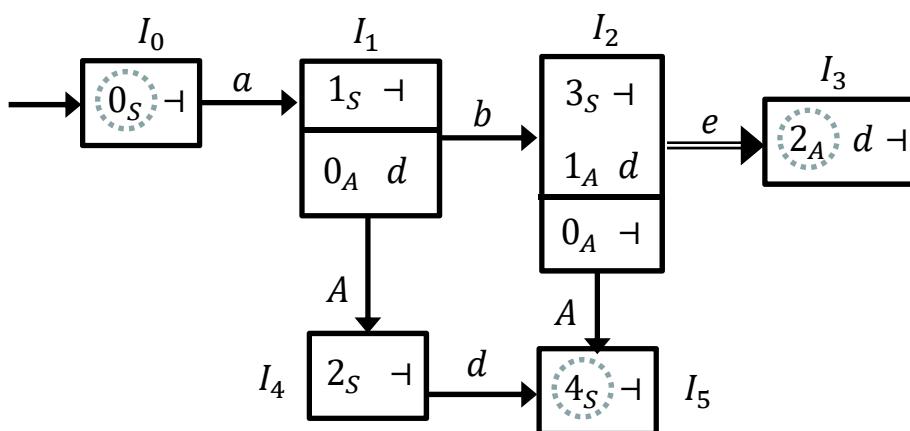
Multiple transitions + convergent arcs analysis of the two strings $abe \vdash e$ $abed \vdash$



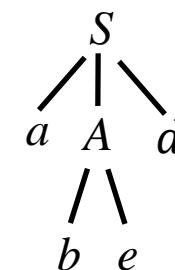
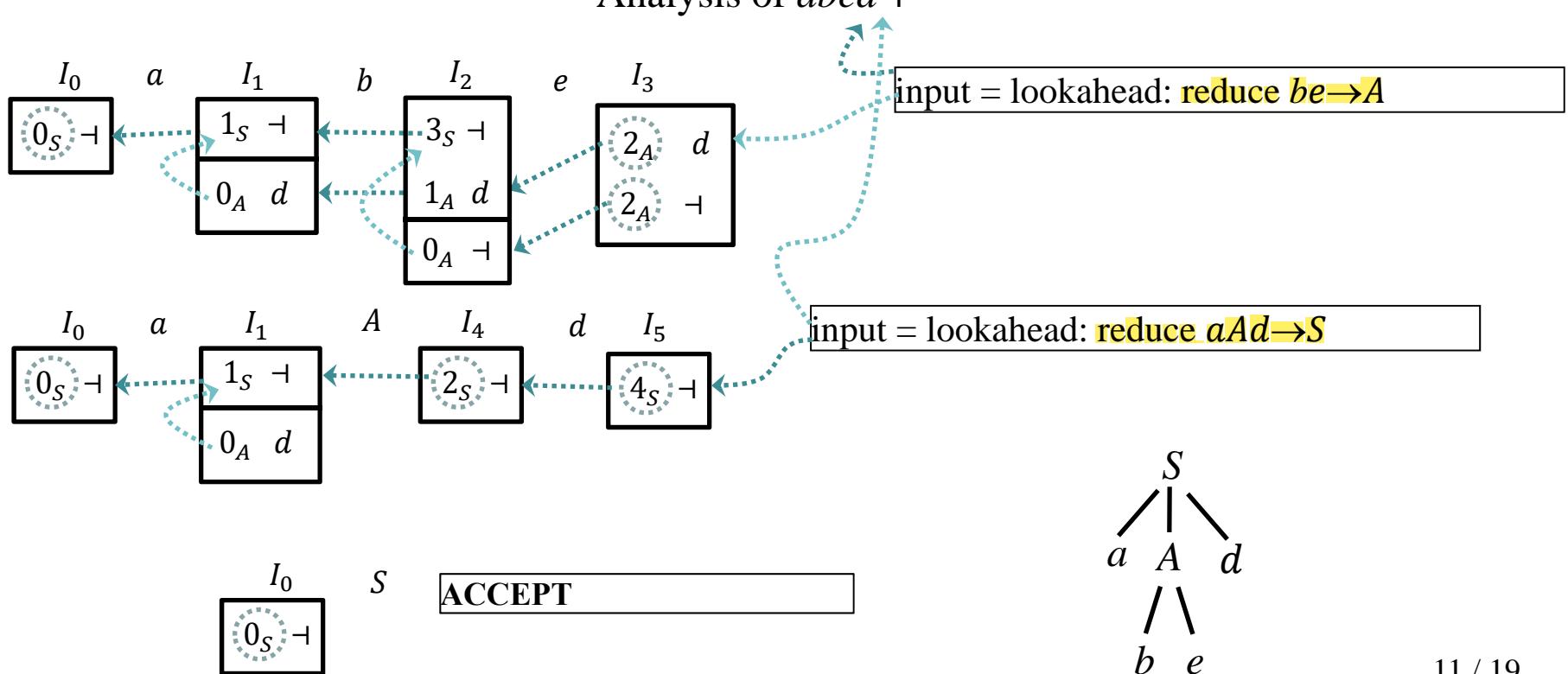
Analysis of $abe \vdash$



Multiple transitions + convergent arcs analysis of string $abed\vdash$



Analysis of $abed\vdash$



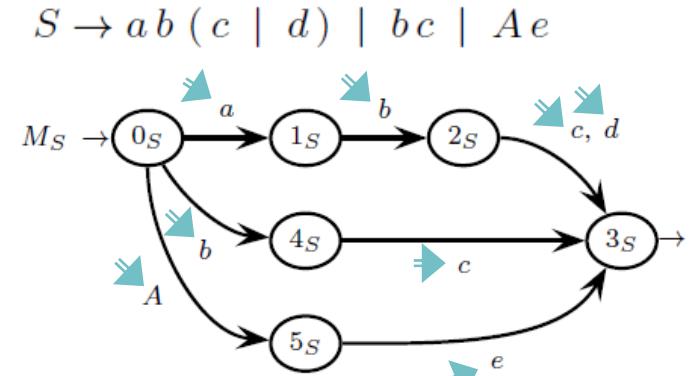
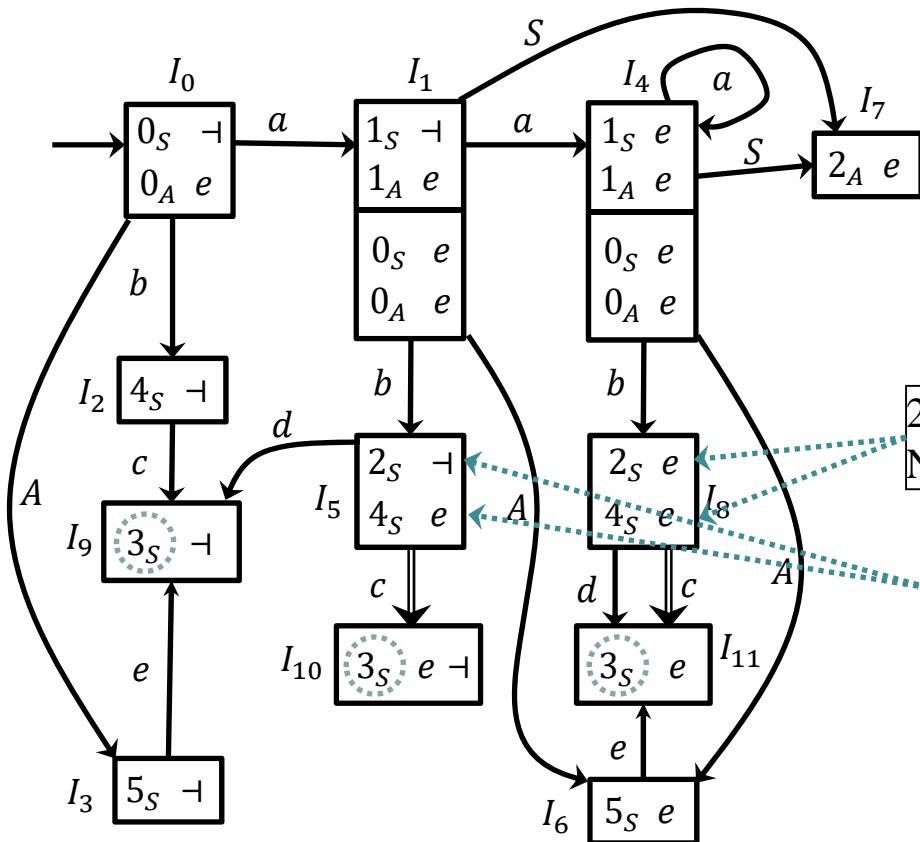
11 / 19

ELR() analysis can delay the recognition and so doing it is able to distinguish in these cases (multiple transitions+convergent arcs)

An unfortunate case: multiple transitions + convergent arcs with conflict

Example:

pilot:



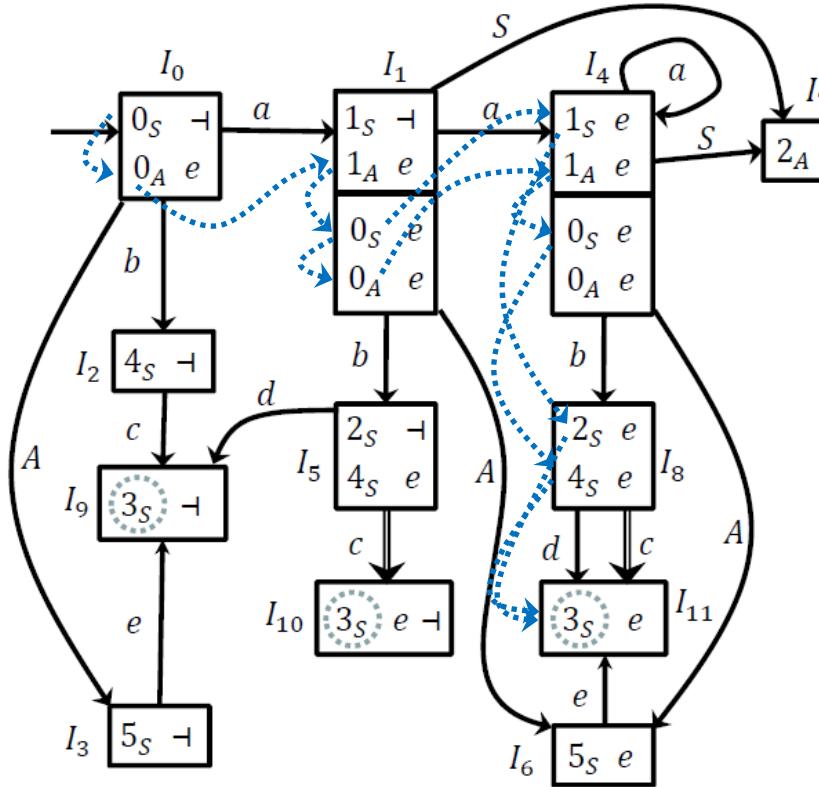
Strange machines, mutual recursion

2 convergent arcs to 3_S from 2_S and from 4_S
Now **lookaheads are not disjointed**: conflict

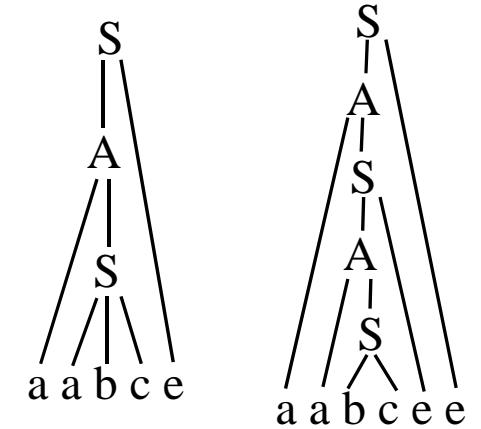
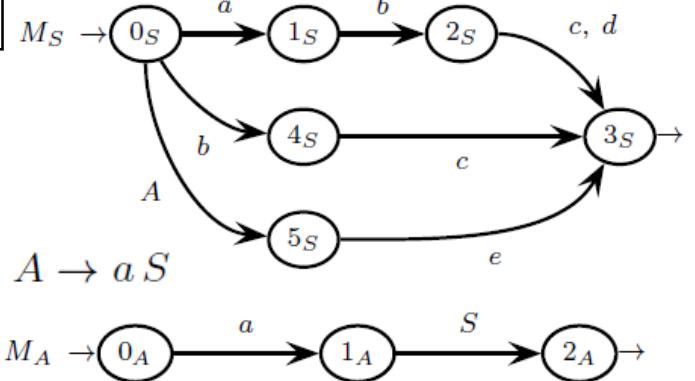
2 convergent arcs to 3_S from 2_S and from 4_S
disjointed lookahead, no conflict

analysis of string $aabce$

the PDA, after reading c , uncertain between $bc \rightarrow S$ and $abc \rightarrow S$
the input string could also be $aabceee$



$S \rightarrow a b (c \mid d) \mid bc \mid Ae$



two computations of the machine net generate string $aabc$

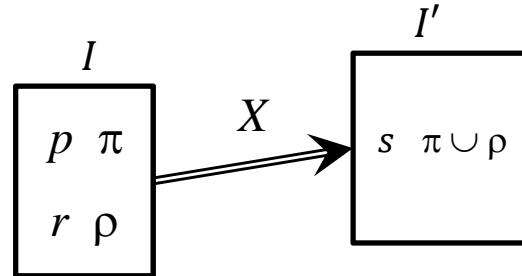
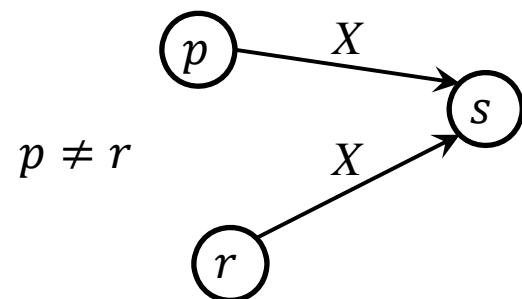
and lead, on the pilot, to the same m-state, same state and same lookahead --> what reduction do we do?

$$0_S \xrightarrow{\epsilon} 0_A \xrightarrow{a} 1_A \xrightarrow{\epsilon} 0_S \xrightarrow{a} 1_S \xrightarrow{b} 2_S \xrightarrow{c} 3_S$$

$$0_S \xrightarrow{\epsilon} 0_A \xrightarrow{a} 1_A \xrightarrow{\epsilon} 0_S \xrightarrow{\epsilon} 0_A \xrightarrow{a} 1_A \xrightarrow{\epsilon} 0_S \xrightarrow{b} 4_S \xrightarrow{c} 3_S$$

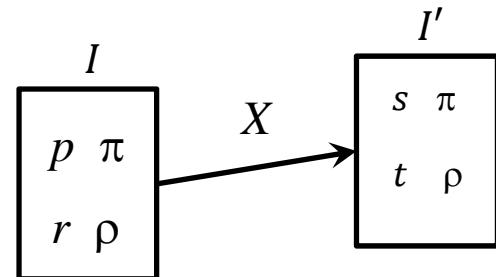
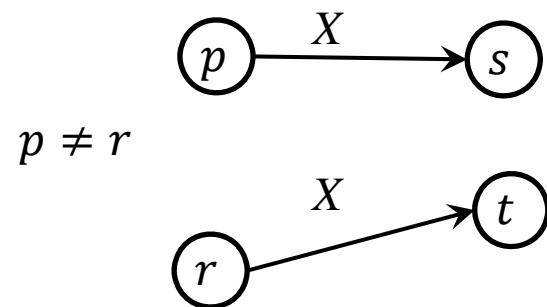
A reduction-reduction conflict
occurs in m-state I_{11}

visual representation of a convergence conflict



NB: conflict occurs if $\pi \cap \rho \neq \emptyset$ --> in our example it was just the same character

NB: not to be confused with the presence of multiple transitions (which is **harmless**)!



they lead to different states, so multiple transitions are **harmless**, the problem is with convergence conflict!

In conclusion, to check applicability of bottom-up parsing method (*LR*)

called *ELR(1)* condition (Left scan Rightmost derivation with lookahead 1 for Extended grammars)

- Build the pilot automaton
- Check the following conditions
 1. Absence of **shift - reduce conflicts**
 2. Absence of **reduce – reduce conflicts**
 3. Absence of **convergence conflicts**

COMPLEXITY OF PARSING

when analyzing string x , with length $n = |x|$, we argue that the number of elements in the stack is $O(n)$
(i.e., $\leq k \cdot n + c$ for some $k, c \in N$)
to determine the number of moves of the PDA, we add up

$n_T = \# \text{ terminal shift}$, $n_N = \# \text{ nonterminal shift}$, $n_R = \# \text{ reductions}$,

clearly, $n_T = n$ and $n_N = n_R$ (\forall reduction also a nonterminal shift is executed)

\Rightarrow total number of moves is

$$n_T + n_R + n_N = n + 2 \cdot n_R$$

furthermore

- number of reductions involving a terminal (e.g. $A \rightarrow a$, $A \rightarrow aB$, $A \rightarrow B a C$) is $\leq n$
- number of reductions not involving terminals, i.e., of type $A \rightarrow \epsilon$ or $A \rightarrow B$ (copy rules) or $A \rightarrow B C$ etc..., is $O(n)$, because the grammar has no circular derivations

Therefore the complexity of the parsing algorithm is $O(n)$

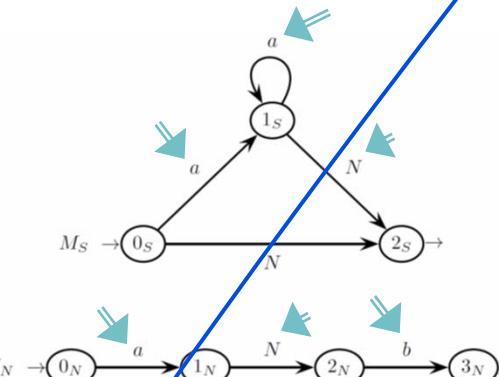
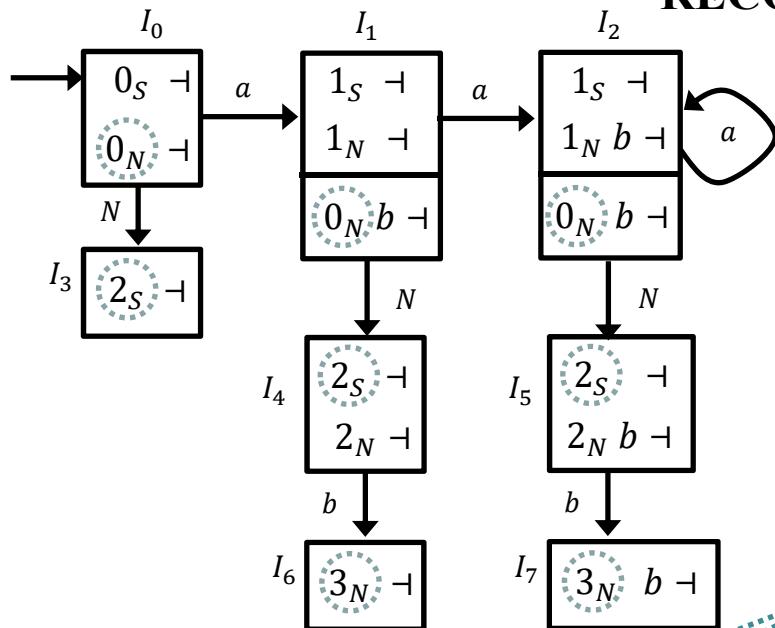
PARSER IMPLEMENTATION USING A VECTOR-STACK

in practice, by coding with common programming languages, one can access the stack elements below the top
⇒ the pointer component of the tuples in the stack can be an integer (with no value restriction)
that points directly to the position in the stack where the handle elements start
from a theoretical viewpoint, the parser model is not a PDA anymore (the stack alphabet is infinite)
in practice, this implementation is possible and easy in all common programming platforms

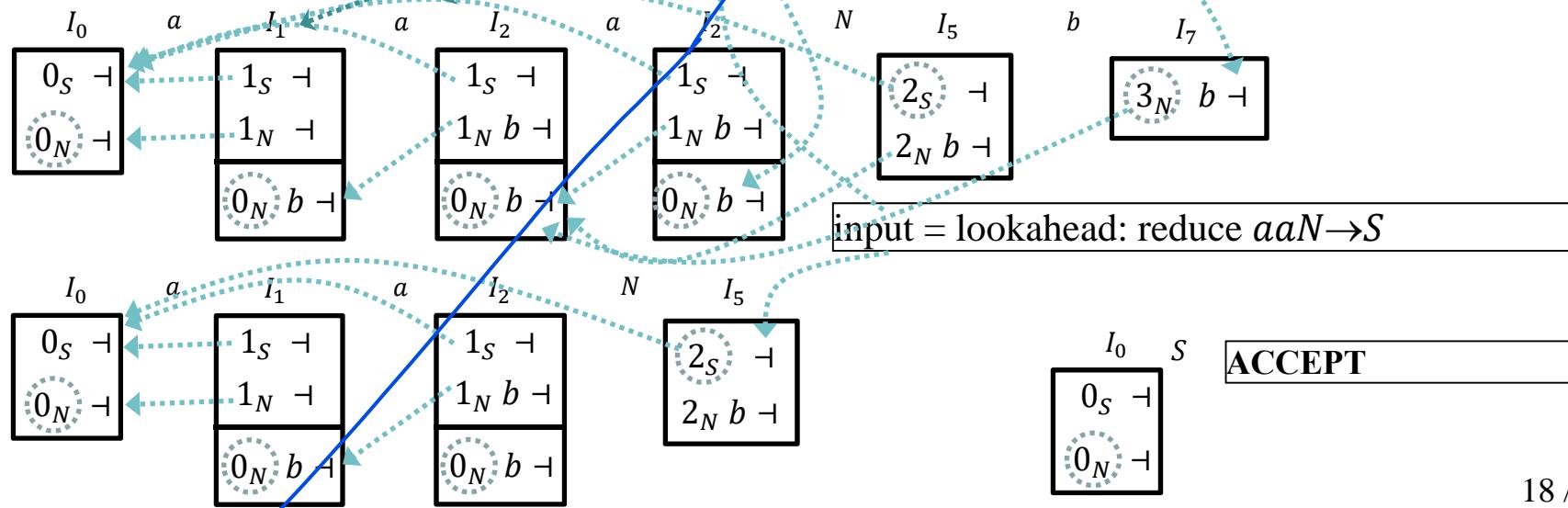
In the tuples corresponding to items of the m-state base, instead of a pointer to the previous element
one can *copy the same value* (which is a stack position index) included in the previous stack element
at reduction time the stack is popped until the position denoted by the index (excluded)

This method of using integer indexes is also exploited in the Earley Algorithm
that can be applied to a much wider class of grammars than the ELR method seen so far

RECONSIDER AN EXAMPLE already treated



Example analysis of string $aabb \dashv$

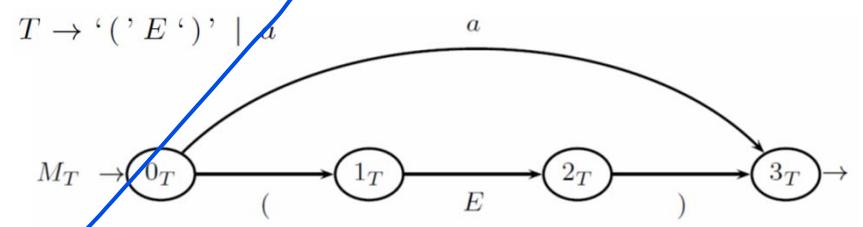
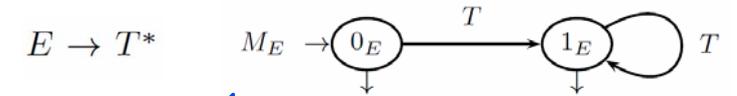
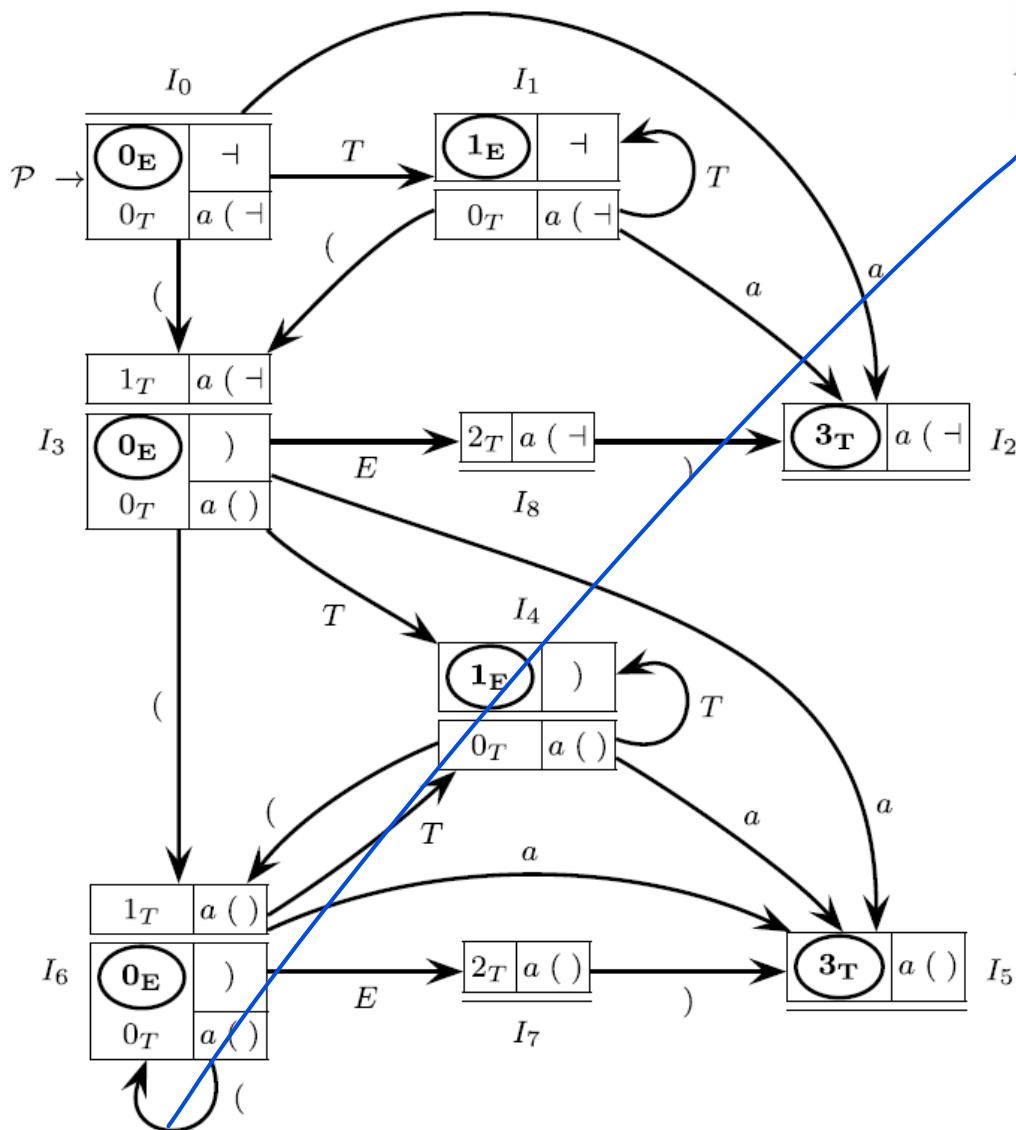


Final Example of an $ELR(1)$ grammar

Exercise: build the pilot automaton

The grammar (axiom E)

can be analyzed using the top-down method (simpler)



Notice: in all m-states the base includes only one item, hence during parsing there is only one thread of analysis

Top-Down Syntax Analysis – $ELL(k)$

Prof. A. Morzenti

REVIEW

Example – top-down analysis of sentence: $a\ a\ b\ b\ b\ a\ a\ a\ a$

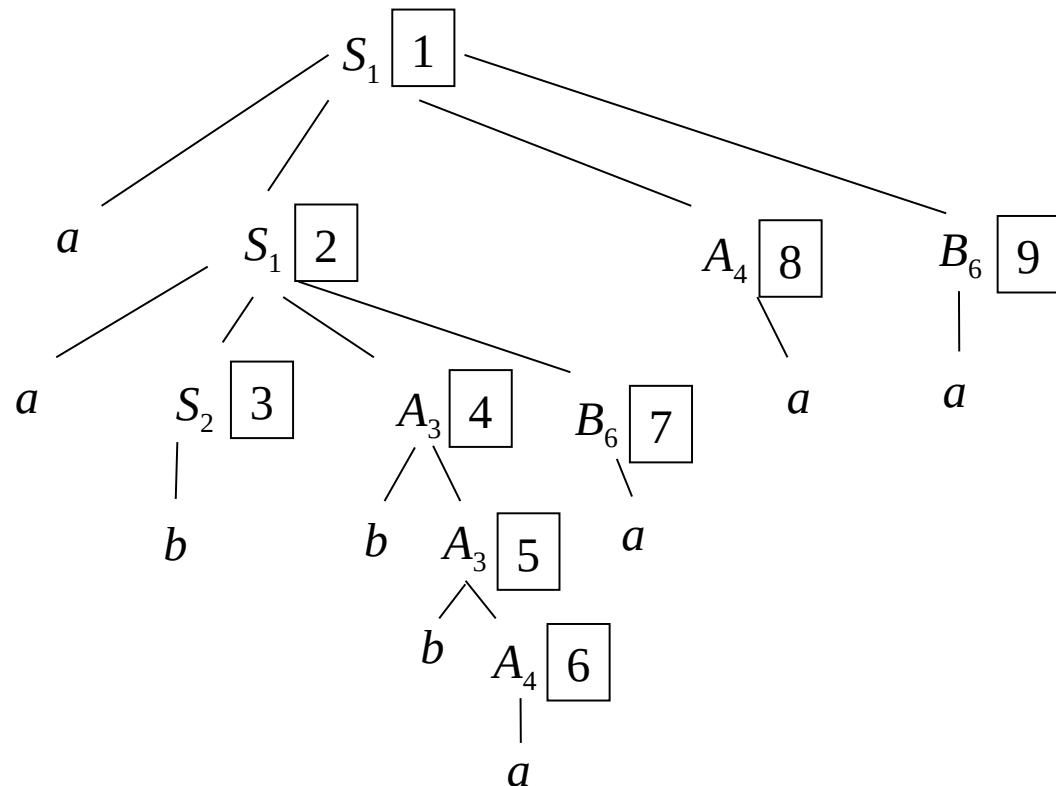
Framed numbers = order in rule application

subscripts of nonterminals in the tree = applied rule

Leftmost: always expanded 1st nonterm. from left

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

$S \Rightarrow aSAB \Rightarrow aaSABAB \Rightarrow aabABAB \Rightarrow aabbABAB \Rightarrow aabbbABAB \Rightarrow aabbbaBAB \Rightarrow$
 $\Rightarrow aabbbaaAB \Rightarrow aabbbaaaB \Rightarrow aabbbaaaa$

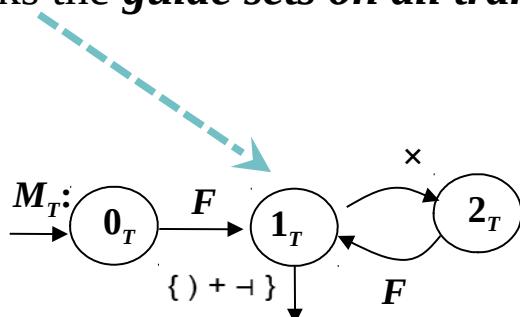


TOP DOWN ANALYSIS

Top down parsing procedures are guided directly by the transition diagrams of the machines they determine the nonterminal from which a string is derived
very soon, when the first character of that string is read

In the textbook top-down parsing is derived from bottom-up by imposing further restrictions
the pilot automaton is transformed into the grammar machine net
annotated with information to allow for a unique choice in presence of forks
a **guide set** added to each arc: it tells which chars are encountered following the arc
in case of forks the **guide sets on all transitions** must be **disjointed**

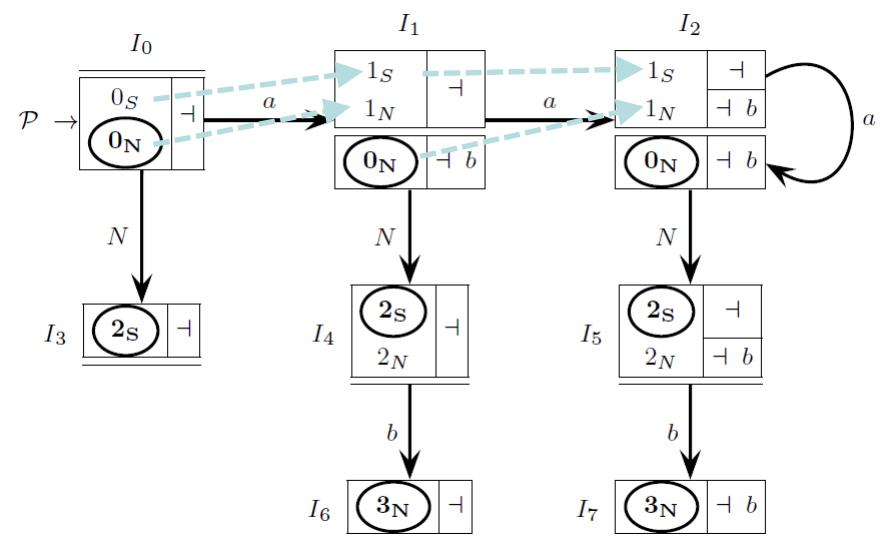
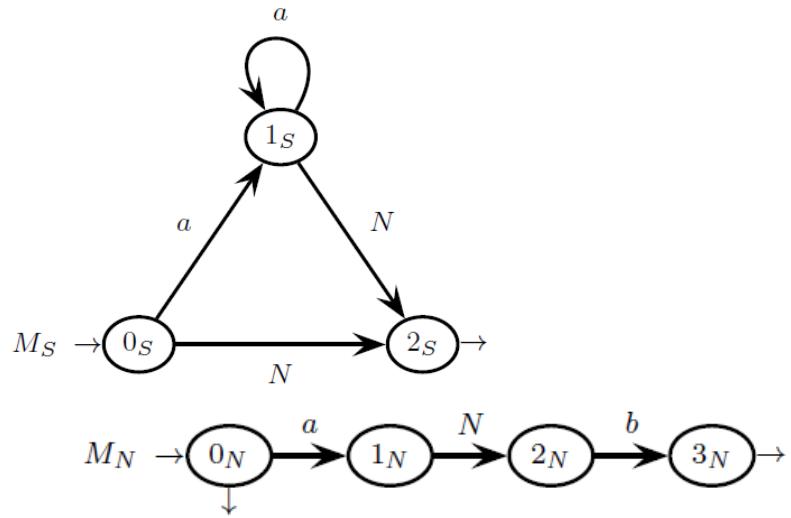
$$\begin{aligned} E &\rightarrow T (+ T)^* \\ T &\rightarrow F (\times F)^* \\ F &\rightarrow a \mid (' E ') \end{aligned}$$



```
procedure T
  call F;
  while(cc=='x')
    cc:=next;
    call F;
  end;
  if(cc in {} != y)
    return;
  else error;
  endif
end T;
```

WHAT CAN PREVENT TOP-DOWN PARSING - 1 - MULTIPLE TRANSITIONS

An m-state I has *multiple transitions* if $\langle p, \pi \rangle, \langle r, \rho \rangle \in I$ and both $\delta(p, X)$ and $\delta(r, X)$ are defined
 Example: grammar $S \rightarrow a^* N \quad N \rightarrow aNb \mid \epsilon$: multiple transitions between I_0 and I_1 , I_1 and I_2



multiple transitions originate several analysis threads, typical of **bottom up (ELR)** analysis
 several hypotheses are simultaneously investigated

this prevents top-down parsing, where only one analysis hypothesis is considered

Absence of multiple transitions is called ***Single Transition Property* (STP)**

(NB: obviously STP ensures absence of convergent arcs in the pilot
 there are no pairs of distinct paths that might possibly rejoin)

WHAT CAN PREVENT TOP-DOWN PARSING - 2 - LEFT RECURSIVE DERIVATIONS

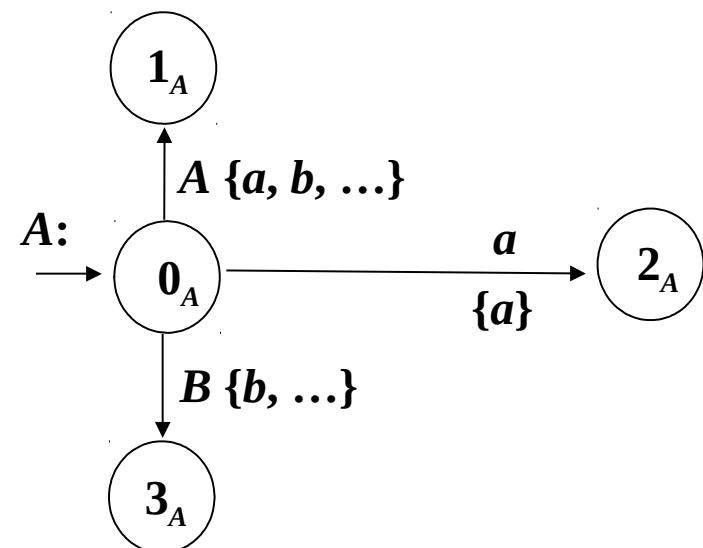
A derivation $A \rightarrow A^+$, with $M \neq \emptyset$, is called **left recursive**

It derives from rules with left recursion, either direct or indirect
possibly in presence of **nullable nonterminals**

Example: $E \rightarrow X E + a \mid a \quad X \rightarrow \epsilon \mid b$ - derivation: $E \Rightarrow X E + a \Rightarrow E + a$

Presence of left recursive derivations prevents top-down deterministic analysis

Example: $A \rightarrow A \dots \mid a \dots \mid B$
 $B \rightarrow b \dots$



ELL(1) CONDITION FOR A GRAMMAR

A grammar G , represented as a machine net, is **ELL(1)** iff:

1. The pilot automaton satisfies the **ELR(1)** condition
2. There are no left recursive derivations
3. The pilot automaton has no multiple transitions (STP satisfied)

NB: therefore the family of **ELL(1)** languages is a subset of the **ELR(1)** languages

ELR(1) is a more general/powerful method than **ELL(1)**

apparently, to check the **ELL(1)** condition it is necessary to build the pilot automaton

actually, it is not necessary, we can avoid pilot construction

THE ELL(1) CONDITION CAN BE CHECKED BY BUILDING THE PARSER CONTROL FLOW GRAPH (PCFG)

It constitutes the control flow graph of the code of the **ELL(1)** parser

From the PCFG one can easily derive (also manually) the parser program code
it consists of a set of recursive procedures, one for each nonterminal

the PCFG is obtained from the machine net by adding

- call arcs
- for each $a \text{ cal } B \text{ arc } (NB: \text{ if } A = B \text{ then direct recursion})$
- for each $q_A \xrightarrow{B} r_A$ a call arc $q_A \xrightarrow{} 0_B$ ($NB: \text{ if } A = B \text{ then direct recursion}$)
- prospect sets of the states, and then from these...
...the guide sets for the following arcs of the machine net (almost all) (almost all) :
- terminal shift arcs like: (trivial...)
- terminal shift arcs like: $q \xrightarrow{} r$ (trivial...)
- call arcs like:
- call arcs like: $q_A \xrightarrow{} 0_B$
- exiting darts $\xrightarrow{} \text{tagging the final states } f_{\mathcal{F}_A}$ of machine M_A (with $f_{\mathcal{F}_A} \in F_A$)
- (NB: no guide set on nonterminal shift arcs) $\xrightarrow{B} q_A \xrightarrow{} r_A$)
• (NB: no guide set on nonterminal shift arcs $q_A \xrightarrow{} r_A$)

What is the **prospect set** of a state q_X of machine M_X ?

it is the set of symbols that can be encountered when exiting machine M_X (in all cases)

- akin to the lookahead sets of LR analysis

- one can show that $\text{Pros}(q_X) = \text{union of the lookahead sets of all pilot items with state } q_X$

Meaning of the guide set on an arc Gui(arc) :

the set of symbols that can be met while going through the arc

There are three cases:

1. terminal shift arc: $\text{Gui}(q_i \xrightarrow{a} q_f) = \{a\}$

2. don't exiting from a final state f : $\text{Gui}(q_i \xrightarrow{\cdot} \pi_f) = \pi_f$

3. null arc:

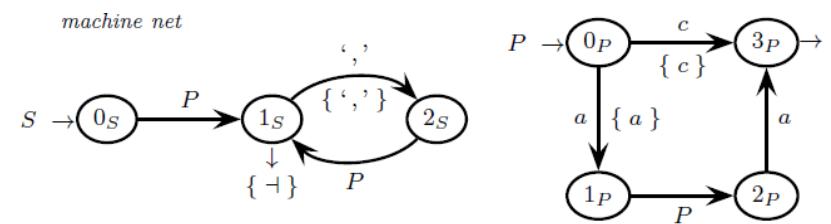
$\text{Gui}(q_i \xrightarrow{\cdot} 0_B) = \{\text{initials of strings derived from } B\} \cup \{\text{symbols following } B \text{ if } B \text{ is nullable}\}$

Guide sets can be computed (also) manually by simply “inspecting” the machine net

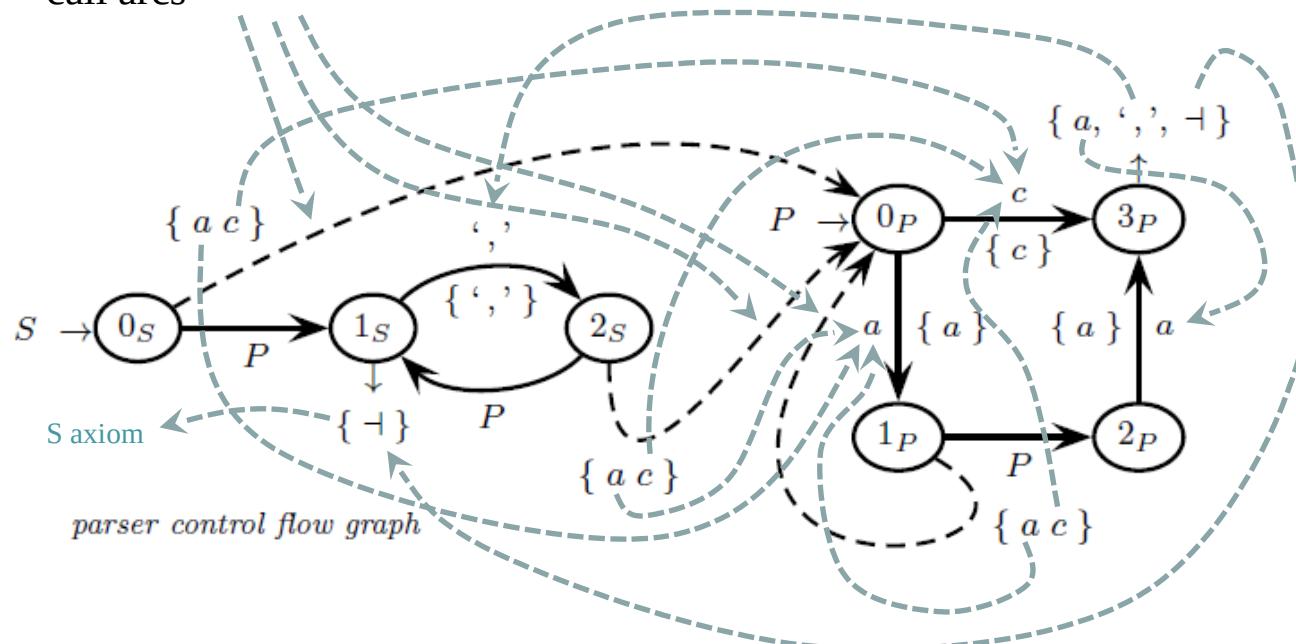
NB: in the slides sometimes prospect sets placed inside the circle denoting the final state

Guide sets for the grammar / machine net

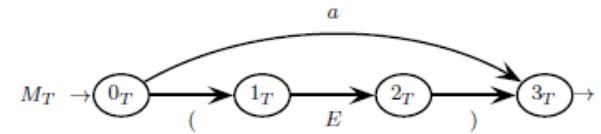
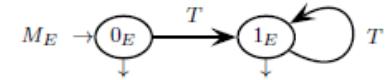
very simple because P is not nullable



call arcs

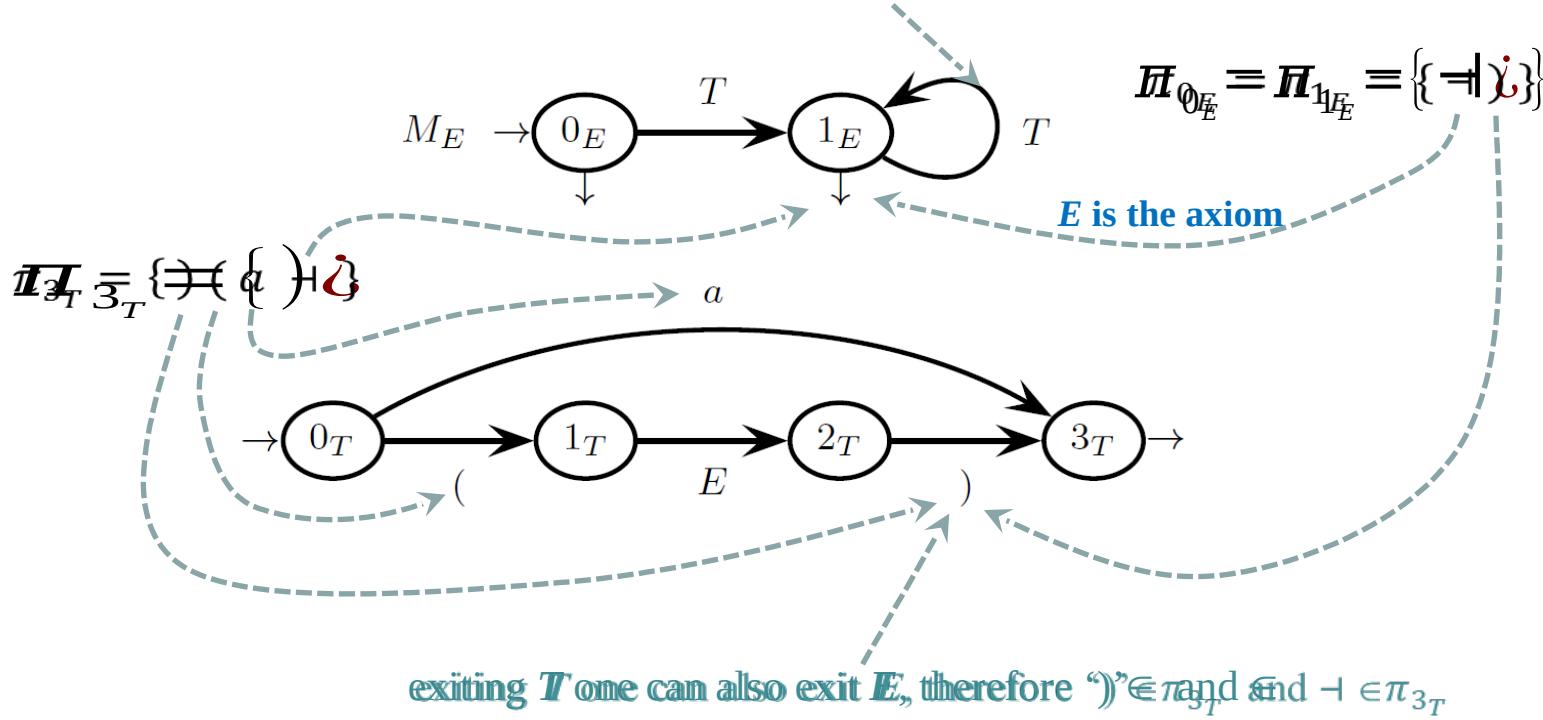


Prospect sets for a grammar / machine net

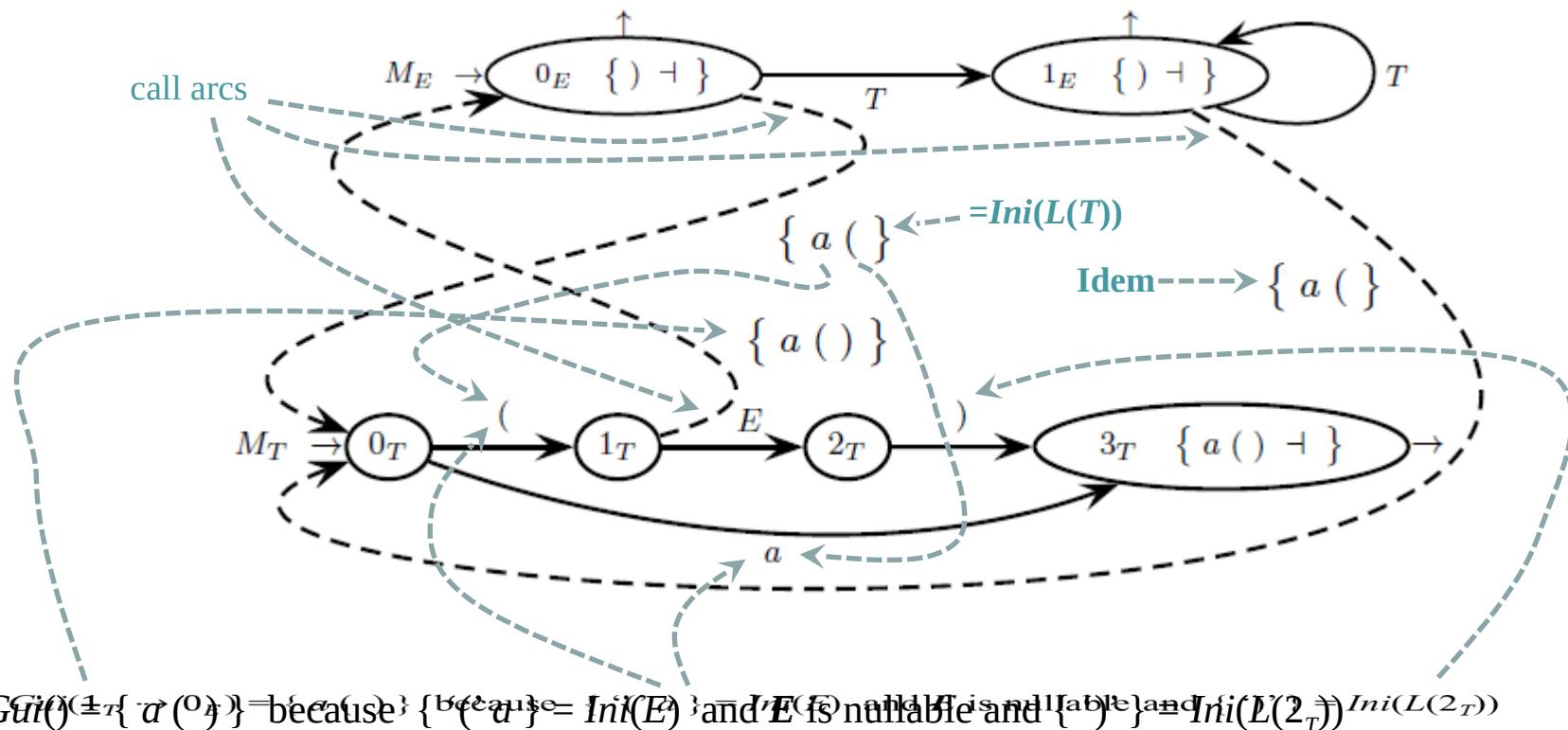
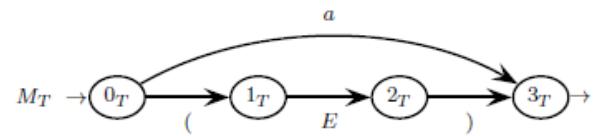
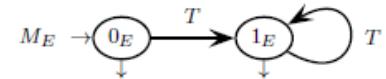


(we only show those on the final states because they coincide with the guide sets on exit darts)

after T there can be another T therefore $\text{Init}(L(T)) \subseteq \alpha_{3_T} \}$



Guide sets for a grammar / machine net



In the PCFG almost all arcs (except for the **NON**terminal shift) are interpreted as conditional instructions

terminal shift arcs $p \xrightarrow{a} q$ are gone through if current char $cc = a$

call arcs $q_A \xrightarrow{O_B} q_B$ are gone through if current char $cc \in Gui(q_A \xrightarrow{O_B} q_B)$

exit arcs $f_A \xrightarrow{\cdot} f_B$ from a node with prospect set \cdot are gone through if current char $cc \in Gui(f_A \xrightarrow{\cdot} f_B)$

instead **NON**terminal shift arcs $p \xrightarrow{a} q$ are (inconditional) «return from machine call»

It can be proved that

disjointed guide sets on alternative arcs \Leftrightarrow ELL(1) condition satisfied

Therefore

- ELL(1) condition can be checked through the guide sets, without building **the pilot automaton** :-)
- if guide sets are disjointed then top-down syntax analysis is possible

PARSER IMPLEMENTATION BY MEANS OF RECURSIVE PROCEDURES

the PCFG is translated into a set of parameterless recursive procedures
(\Rightarrow one procedure for each nonterminal)

call arc \Rightarrow procedure call

terminal shift arc \Rightarrow call of **next** procedure (interface to the scanner/lexical analyzer)

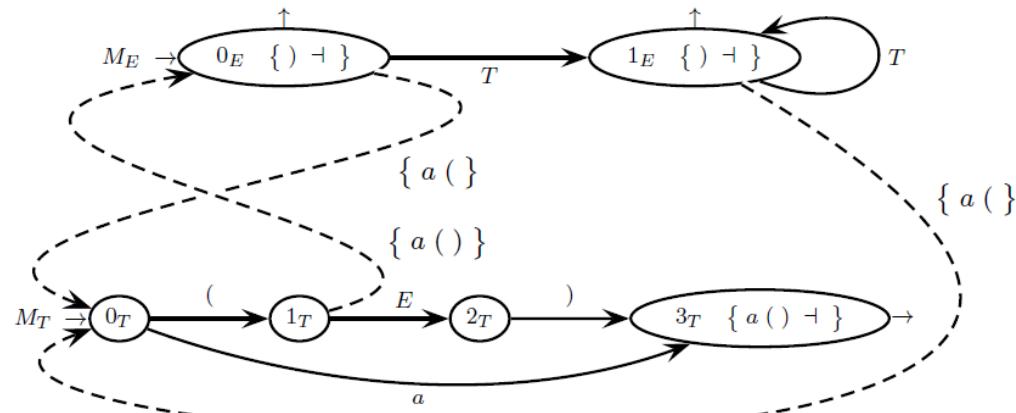
final state dart \Rightarrow procedure return

guide sets are used to choose the next move

the analysis starts by invoking the procedure for the axiom

the technique is called **recursive descent**

running example



```

procedure E
    -- optimized
    while cc  $\in \{ a( \}$  do
        call T
    end while
    if cc  $\in \{ ) + \}$  then
        return
    else
        error
    end if
end procedure

```

```

procedure T
    -- state  $0_T$ 
    if cc  $\in \{ a \}$  then
        cc = next
    else if cc  $\in \{ ( \}$  then
        cc = next
    -- state  $1_T$ 
    if cc  $\in \{ a( \}$  then
        call E
    else
        error
    end if
    -- state  $2_T$ 
    if cc  $\in \{ ) \}$  then
        cc = next
    else
        error
    end if
    -- state  $3_T$ 
    if cc  $\in \{ a() + \}$  then
        return
    else
        error
    end if
end procedure

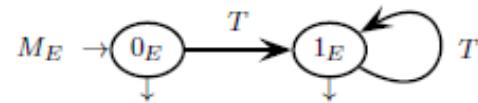
```

An alternative implementation:
TOP-DOWN PREDICTIVE ANALYZER
ALGORITHM WITH EXPLICIT STACK (NO RECURSIVE PROCEDURES)

Based on the PCFG F

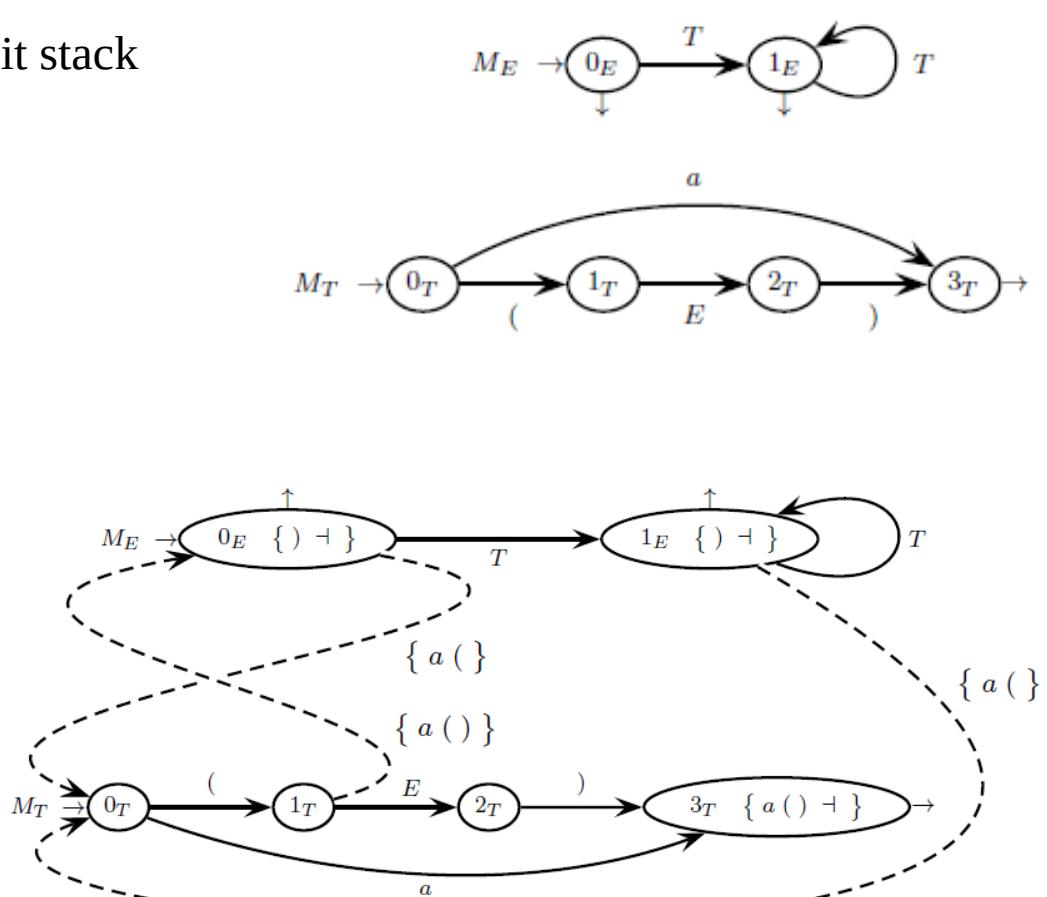
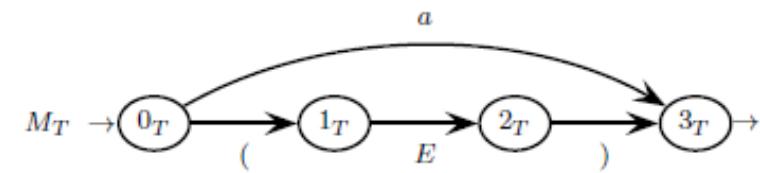
- Initially the stack contains $\langle 0_S \rangle$
- If the stack top is $\langle q_A \rangle$ (i.e., M_{A1} is the active machine and it is in state q_A) and the current char is cc
 - (scan move): if $q_A r_A^{cc} \rightarrow q_A$ read cc and replace $\langle q_A \rangle$ with $\langle q_A \rangle$; (pop; push);
 - (call move): if F includes arcs $q_A 0_B$ and r_A^B and $cc \in G_{q_A}(q_A \rightarrow 0_B)$ then pop; push $\langle r_A \rangle$; push $\langle 0_B \rangle$ r_A is the return point
 - (return move): if $q_A \in F_A$ (q_A final state for M_A), π is the prospect set of q_A and $cc \in \pi$ then pop;
 - (acceptance move): if $A \neq S$ (A is the axiom), $q_A \in F_S$ and $cc =$ then accept the string and halt;
 - in any other case: reject and halt

predictive analysis: example with explicit stack



analysis trace for string $x = (a)$

stack	x	move x	move
0_E	(a)	call T (a) \vdash	call T
1_E 0_T	(a)	scan (a) \vdash	scan
1_E 1_T	a)	call E a) \vdash	call E
1_E 2_T 0_E	a)	call T a) \vdash	call T
1_E 2_T 1_E 0_T	a)	call T a) \vdash	scan
1_E 2_T 1_E 1_T	a)	scan	return T
1_E 2_T 1_E 2_T)	return T	return E
1_E 2_T 1_E)	return E	scan
1_E 2_T 3_T)	scan \dashv	return T
1_E 3_T	return T	\dashv	accept
1_E			accept



analysis trace for string $x = ()$

0_E	()	call T	call T
1_E 0_T	()	scan	scan
1_E 1_T)	call E	call E
1_E 2_T)	return	return
1_E 2_T 0_E)	return	scan
1_E 2_T 1_E)	return	return
1_E 3_T)	scan	return
1_E 3_T	return	\dashv	accept
1_E			accept

analysis trace for string $x = aa$

0_E	a	call T	call T
1_E 0_T	a	scan	scan
1_E 3_T	\dashv	return T	return T
1_E			accept
1_E			accept

INCREASING THE LOOKAHEAD FOR **NON-ELL(1)** GRAMMARS

If a grammar is not ELL(1) it sometimes may be possible to transform it to make it ELL(1)
this transformation can be complex and make the grammar less terse

Possible alternative: increase the lookahead

the parser chooses the next move based on a number $k > 1$ of input chars

if one obtains disjointed guide sets of length k then the ELL(k) analysis is feasible

we do **not** provide systematic rules to compute the guide sets of with lookahead $k \geq 2$
we just inspect the machines to determine them, based on intuition

Example: conflict between identifiers of instruction labels and variable names

id is both a label for any instruction

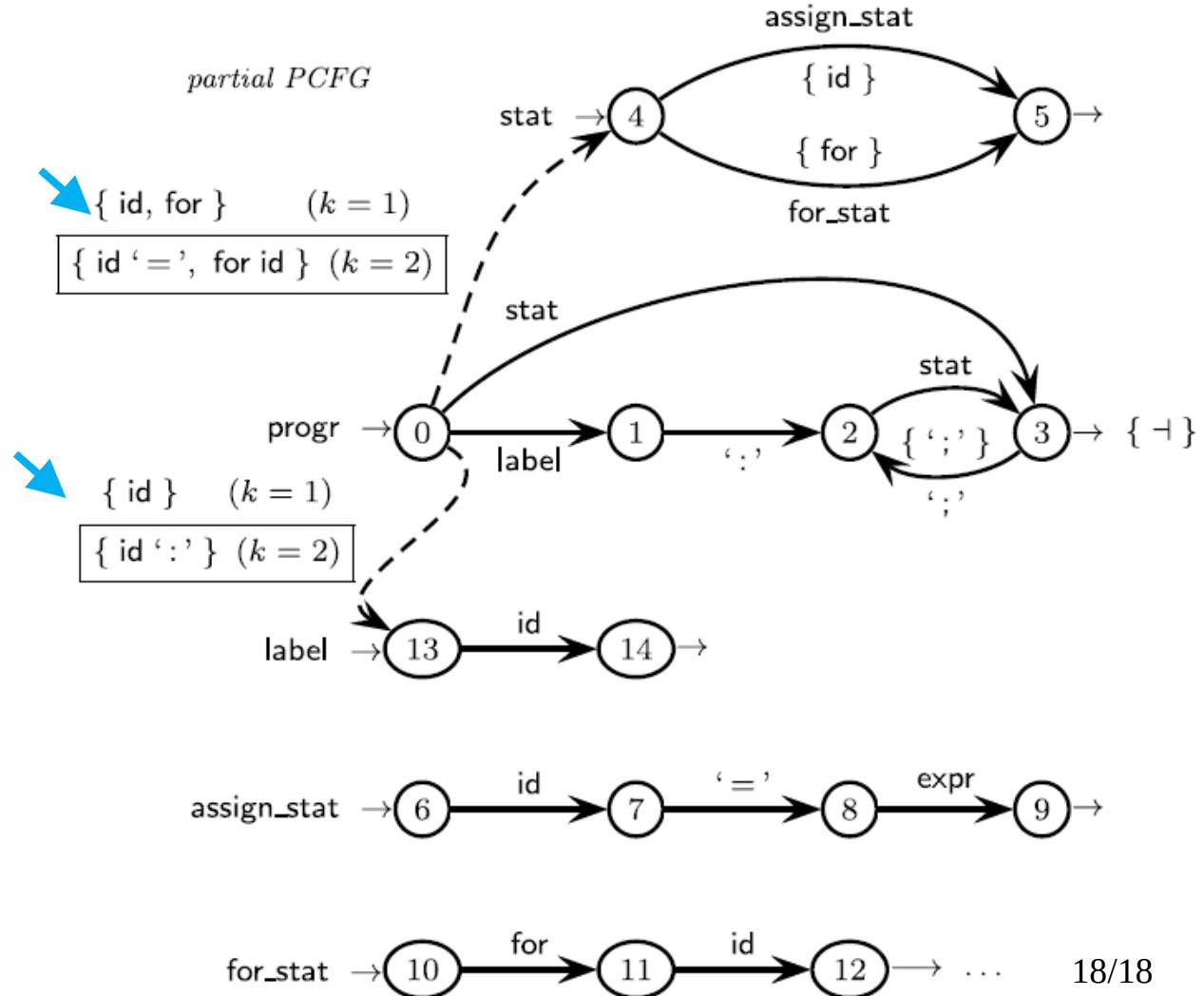
and a variable name in the left side of assignments

To discern one must look beyond the next token

if “**id** :” \Rightarrow label

if “**id** =” \Rightarrow assignment

NB: guide sets of length $k=2$
are framed



Top-down Syntax Analysis

ELL (k) Method

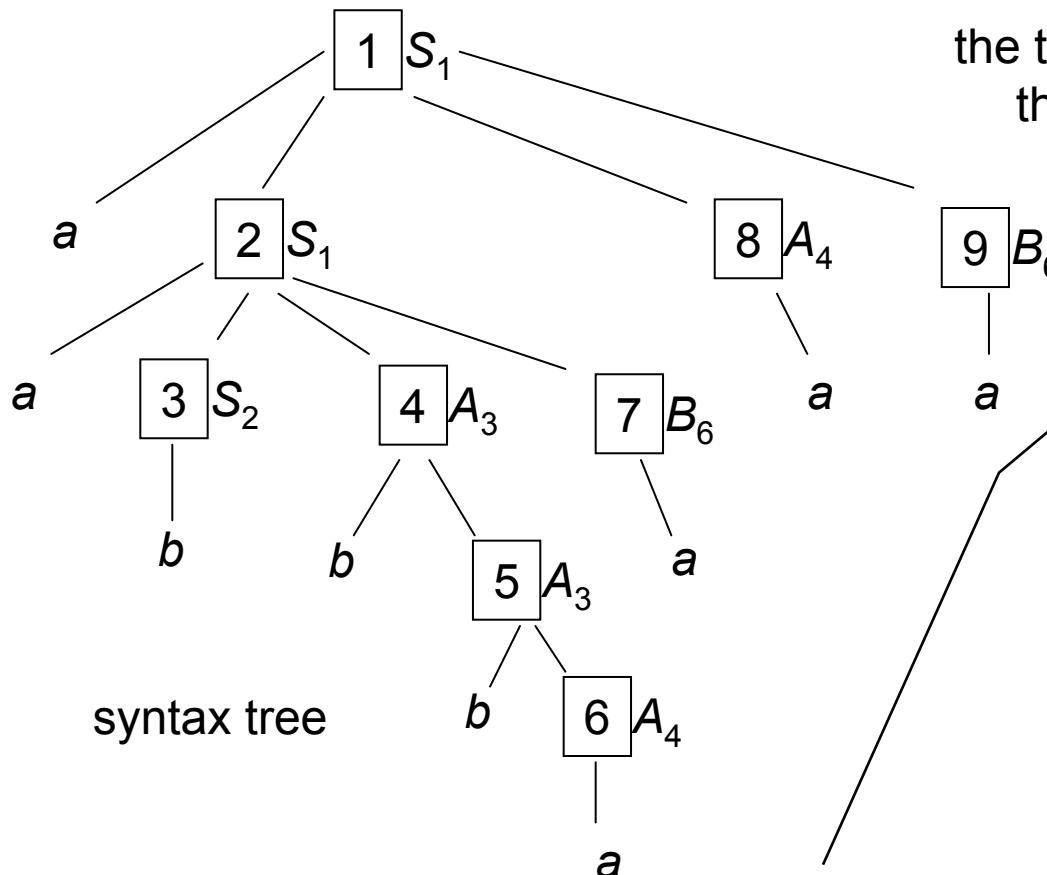
Translated and adapted by L. Breveglieri

TOP-DOWN ANALYSIS – 1

EXAMPLE – analysis of the valid phrase

a a b b b a a a a

The framed numbers indicate the application order of the grammar rules, and the non-terminal subscripts indicate the number of the applied rule



the tree is built top-down left-to-right
the derivation order is leftmost

BNF grammar

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

TOP-DOWN ANALYSIS – 2

The analysis procedure is driven by the machine network – no need of a pilot graph, i.e., the machine net itself can directly work as a pilot for the syntax analyzer *PDA*

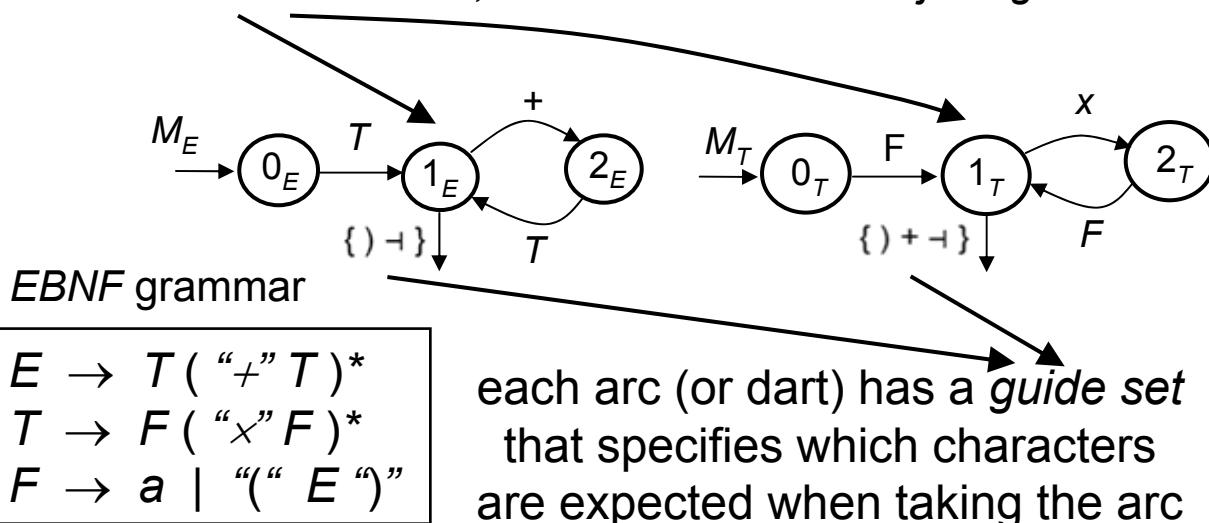
The syntax analyzer can recognize very early which rule it should apply to a phrase, in practice as soon as it finds the first (leftmost) character generated by a rule

We obtain the top-down analysis from the bottom-up one by some restrictions

Final target: the machine net itself will work as the “pilot graph” of the syntax analyzer

Yet the net has to be annotated, to choose which branch to take in the doubtful cases

in the bifurcation states, determinism wants disjoint guide sets



```

procedure T
  call F
  while (cc == 'x')
    cc := next
    call F
  end
  if (cc ∈ { ) + - | } )
    return
  else
    error
  endif
end T

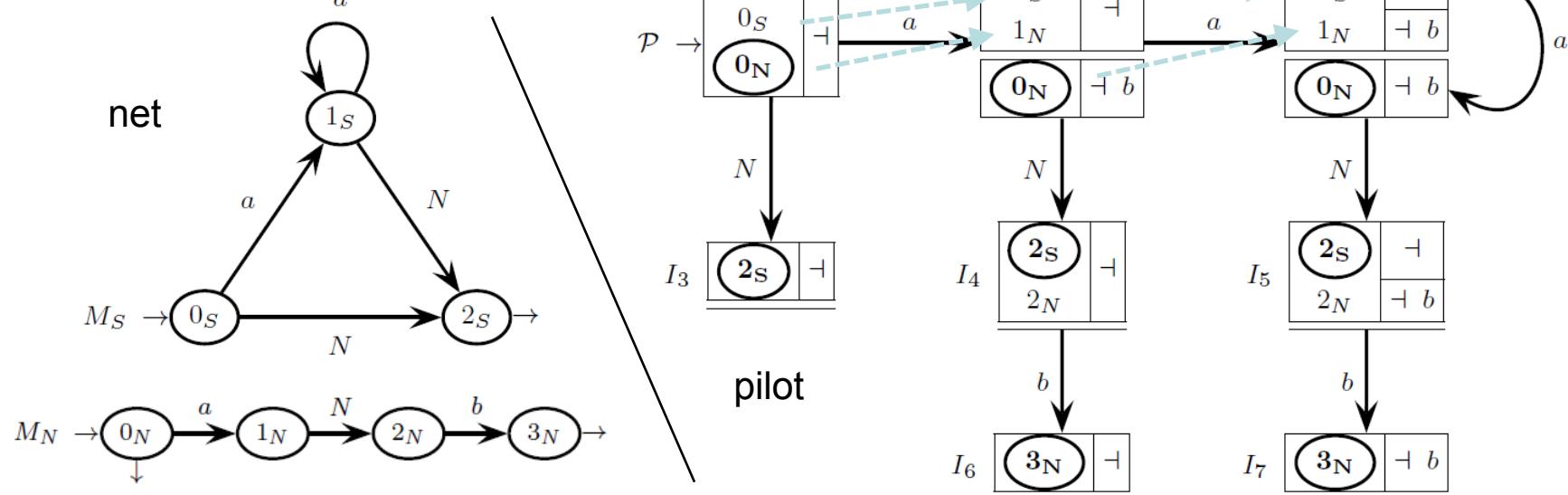
```

RECALLS OF BOTTOM-UP ANALYSIS

The problem with top-down is that decision about which path pursue must be taken immediately

A m-state I has a *multiple transition* if there are two (or more) items $\langle p, \pi \rangle, \langle r, \rho \rangle \in I$ and their two (or more) next states $\delta(p, X), \delta(r, X)$ are both (or all) defined

EXAMPLE – grammar $S \rightarrow a^* N$ and $N \rightarrow a N b \mid \epsilon$ has multiple transitions between I_0 and I_1 , between I_1 and I_2 , ...



A multiple transition originates multiple analysis threads, which are typical in the *ELR* (1) analysis. They carry on in parallel different analysis hypotheses, therefore they are incompatible with top-down analysis

No multiple transitions: *Single Transition Property (STP)*

STP ensures there are not any convergent arcs in the pilot

LEFTMOST RECURSIVE DERIVATIONS

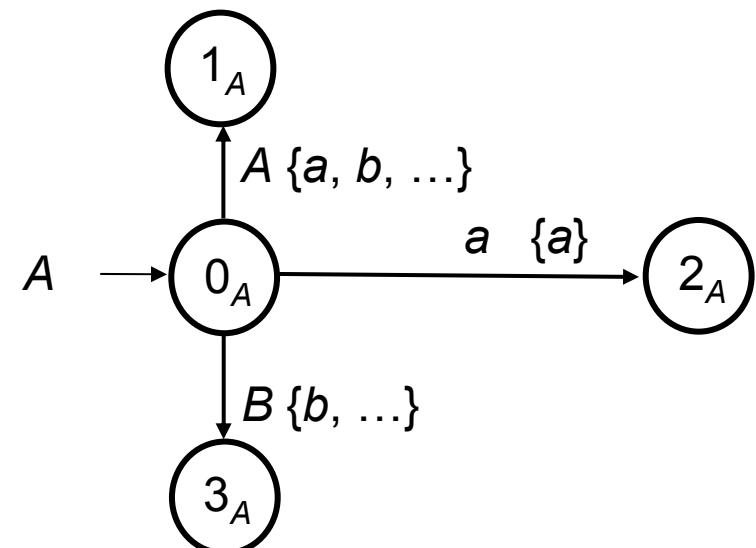
A derivation $A \Rightarrow^+ A v$ with $v \neq \varepsilon$ is said to be **leftmost recursive**

It is caused by the rules with a leftmost recursion, direct or indirect, possibly in the presence of *nullable nonterminals*

EXAMPLE $E \rightarrow XE + a \mid a$ $X \rightarrow \varepsilon \mid b$

Leftmost recursive derivations make top-down analysis impossible

EXAMPLE $A \rightarrow A.... \mid a.... \mid B$
 $B \rightarrow b....$



ELL (1) CONDITION FOR AN EBNF GRAMMAR

A grammar G (in general *EBNF*) represented as a machine net is *ELL (1)* if

- 1) it does not have any *leftmost recursive derivations*
- 2) its pilot graph satisfies the *ELR (1) condition*
- 3) its pilot graph does not have any *multiple transitions* (i.e., has the *STP*)

These three requirements are not completely independent of one another

In particular requirement (1) could be further restricted, but for simplicity it is better to keep all the three of them in the above form

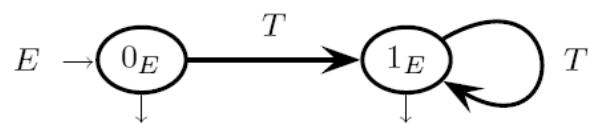
The *ELL (1)* condition above implies that the family of the *ELL (1)* grammars is contained in that of the *ELR (1)* grammars

The containment is strict; furthermore, also the family of *ELL (1)* languages is strictly contained in that of the *ELR (1)* (i.e., deterministic) languages

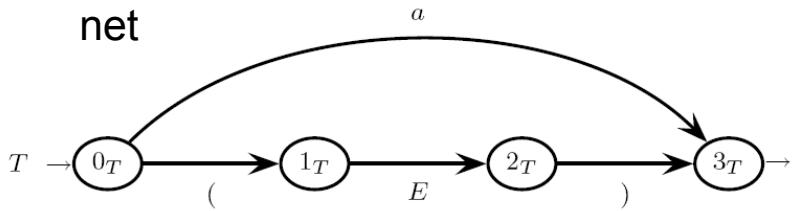
THE GRAMMAR OF THE RUNNING EXAMPLE IS ELL (1)

$E \rightarrow T^*$ EBNF grammar

$T \rightarrow ' (' E ') ' \mid a$



net

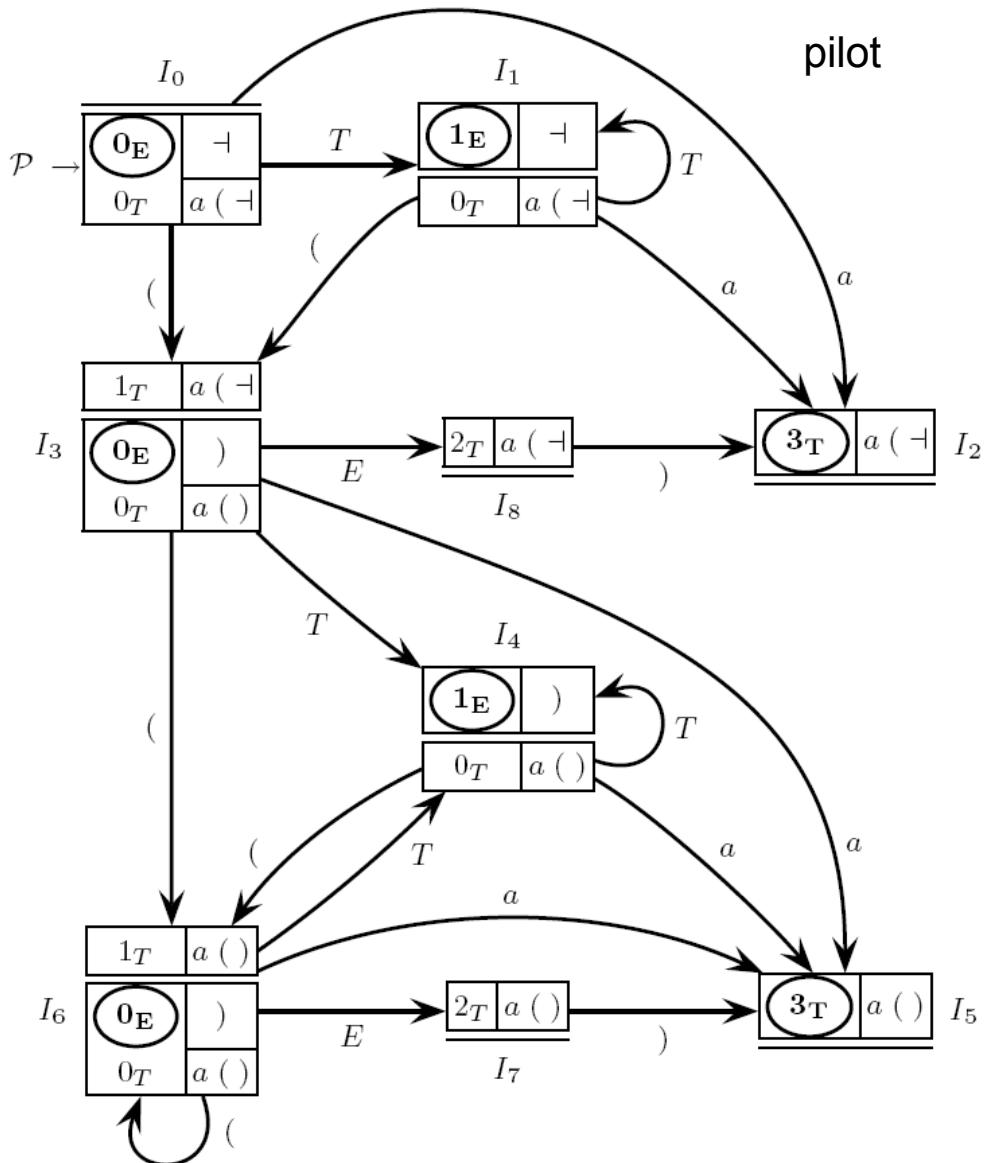


ELL (1) condition

no left recursion

pilot is ELR (1)

pilot has the STP



ELL (1) ANALYSIS TURNS OUT TO BE SUBSTANTIALLY SIMPLIFIED

ANTICIPATED (PREDICTIVE) DECISION – only one item in each m-state base
⇒ we know immediately which rule to apply (no need to wait for the reduction)

STACK POINTERS UNNECESSARY – once the rule to be applied is known
⇒ no need to carry on more than one analysis thread
⇒ unnecessary to keep the items corresponding to other analysis hypotheses

CONTRACTION OF THE STACK – only one analysis thread
⇒ no need to push on stack the state path followed
⇒ it suffices to push on stack the sequence of machines followed

FURTHER RESTRUCTURING OF THE PILOT
⇒ separate the closure states (the initial states of the machines)
⇒ the pilot graph is isomorphic to the machine net
⇒ add some new arcs (*call arcs*) to the shift arcs

Call arc from q_A to 0_B if and only if machine M_A has a transition $q_A \xrightarrow{B} r_A$

Two or more initial states in the closure of a m-state originate a *call chain*

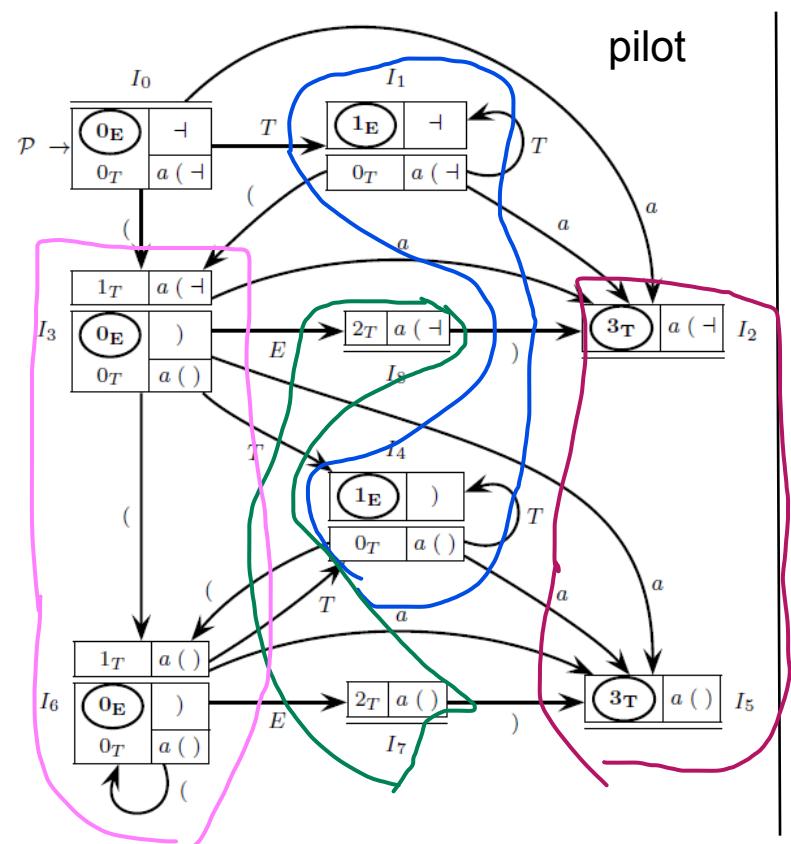
SIMPLIFICATION OF A PILOT THAT SATISFIES THE ELL (1) CONDITION

M-states with same kernel (kernel-identical) are unified (unify look-ahead) \Rightarrow pilot more compact

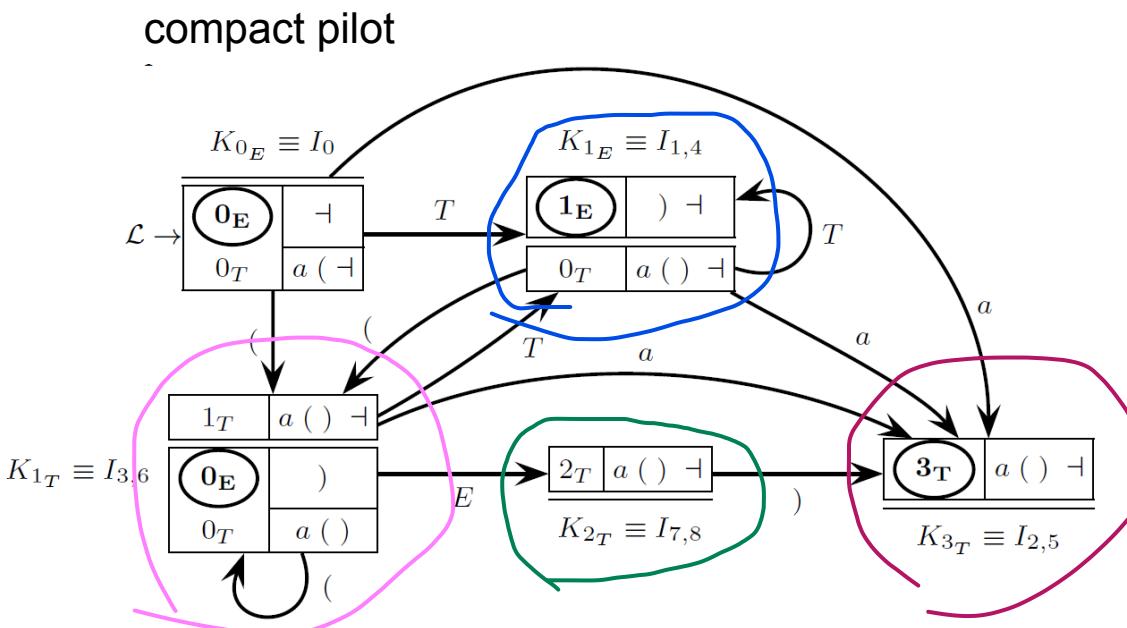
The kernel-identity relation is an equivalence relation

Transitions with same label from kernel-identical m-states go into kernel-identical m-states

The m-state bases (with non-initial states) contain only one item (consequence of STP)
 \Rightarrow they are in a one-to-one correspondence with the non-initial states of the machine net



series 13

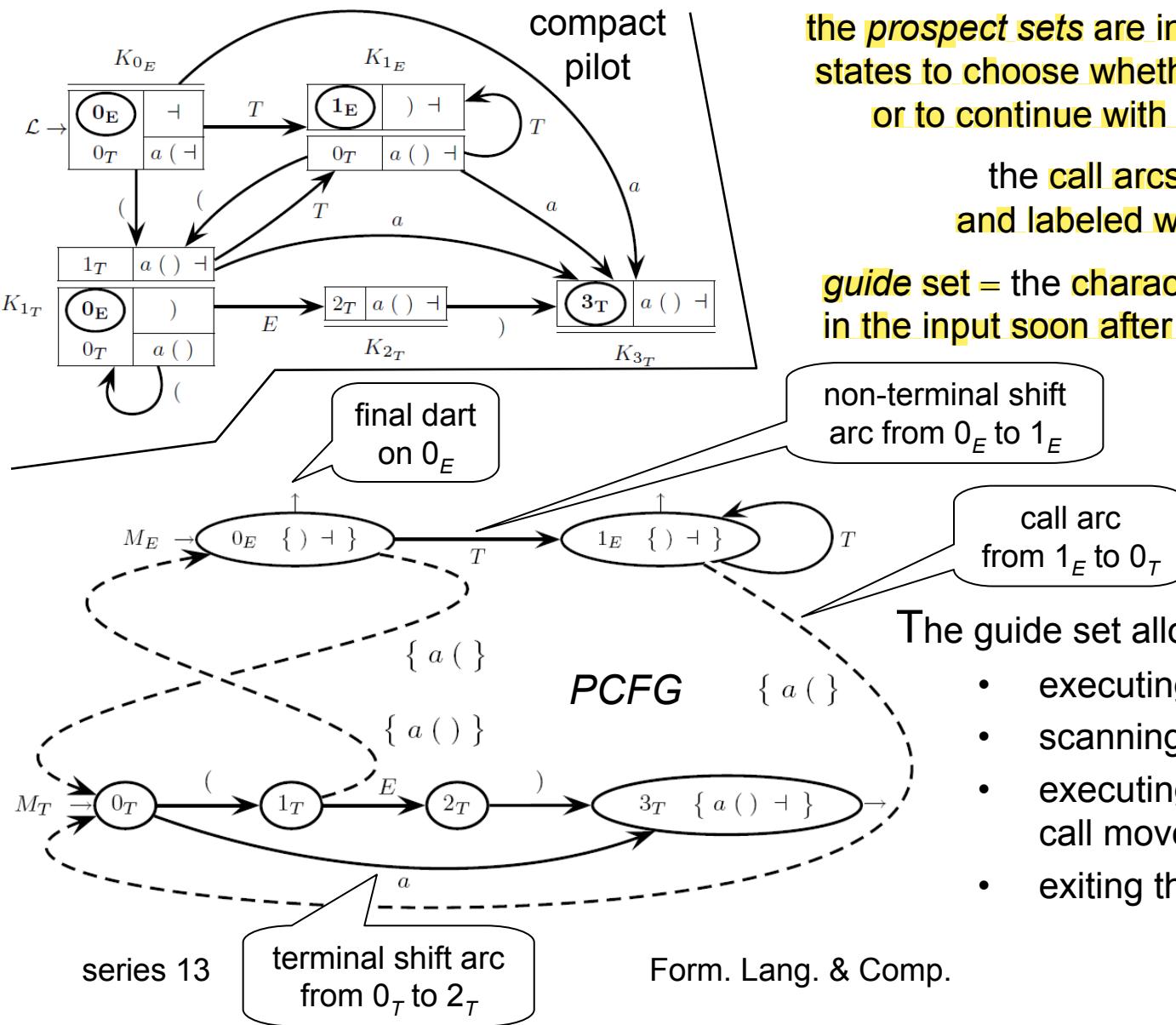


NOTE: the pilot graph “starts looking like” the machine net ...

the net is in slide 7

RESULTING PILOT AUTOMATON – PARSER CONTROL FLOW GRAPH (PCFG)

The Parser Control-Flow Graph (*PCFG*) is the control unit of the *ELL* (1) syntax analyzer *PDA*



the **prospect sets** are included only in the final states to choose whether to exit the machine or to continue with some more moves

the **call arcs** are dashed and labeled with a **guide set**

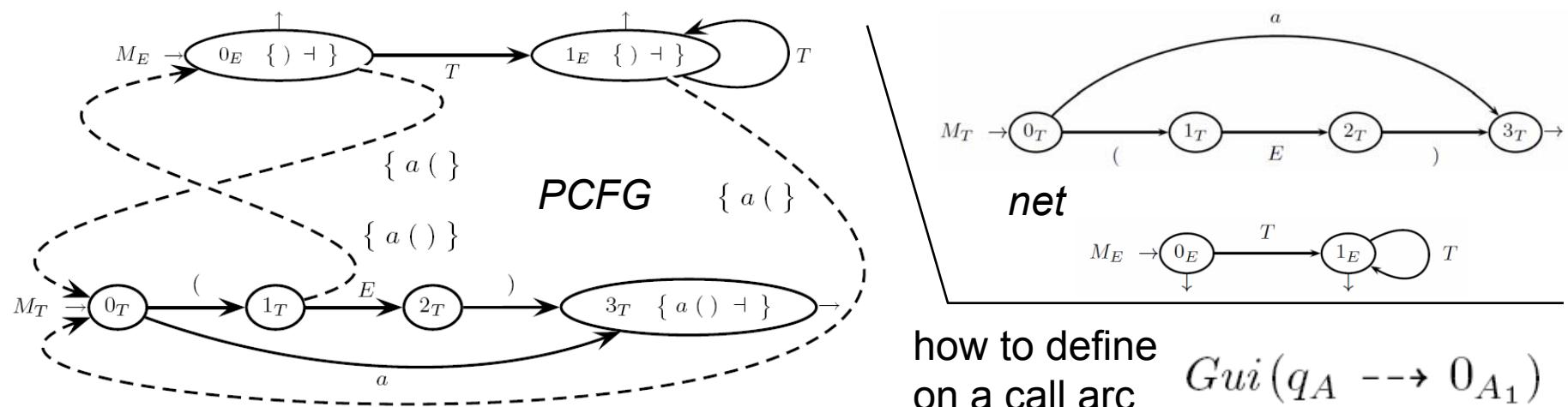
guide set = the **characters** that are **expected** in the input soon after calling the machine

non-terminal shift arc from 0_E to 1_E

call arc from 1_E to 0_T

The guide set allows to choose whether

- executing one call move
- scanning a terminal symbol
- executing one of two or more call moves
- exiting the machine (if final state)



GUIDE SET

$$b \in Gui(q_A \dashrightarrow 0_{A_1})$$

$$b \in Ini(L(0_{A_1}))$$

A_1 is nullable and $b \in Ini(L(r_A))$

A_1 and $L(r_A)$ are both nullable and $b \in \pi_{r_A}$

\exists in \mathcal{F} a call arc $0_{A_1} \xrightarrow{\gamma_2} 0_{A_2}$ and $b \in \gamma_2$

The guide sets of the call arcs that depart from the same state have to

- be disjoint from one another, and
- be disjoint from all the scan arcs from the same state

Guide set on a dart of a final state f_A that contains an item $\langle f_A, \pi \rangle$

$Gui(f_A \rightarrow) = \pi$ the look-ahead π indicates the chars expected when M_A finishes

Trivially $Gui(p \rightarrow^a q) = a$ for a terminal shift arc, where a is a terminal symbol

SUMMARY

In a *PCFG* almost all the arcs (except the non-terminal shift) are interpreted as conditional instructions

Terminal arcs $p \rightarrow^a q$ run if and only if the current character $cc = a$

Call arcs $q_A \rightarrow 0_B$ run if and only if the current character is $cc \in Gui$ ($q_A \rightarrow 0_B$)

Exit arcs (darts) $f_A \rightarrow$ from a state with an item $\langle f_A, \pi \rangle$ run if and only if the current character is $cc \in \pi$

The non-terminal arcs $p \rightarrow^A q$ are interpreted as (unconditioned) return instructions from a machine

It is possible to prove that

disjoint guide sets (in the bifurcation states) \Leftrightarrow condition *ELL* (1) satisfied

PREDICTIVE TOP-DOWN ANALYZER – ALGORITHM

The stack elements are the states of the *PCFG*, i.e., the net states

The stack is initialized with element $\langle 0_E \rangle$

Suppose $\langle q_A \rangle$ is the stack top – it means that machine M_A is active and in the state q_A

The *ELL* syntax analyzer (*PDA*) has four move types

scan move if the shift arc $q_A \xrightarrow{cc} r_A$ exists, then scan the next token and replace the stack top by $\langle r_A \rangle$ (the active machine does not change)

call move if there exists a call arc $q_A \xrightarrow{\gamma} 0_B$ such that $cc \in \gamma$, let $q_A \xrightarrow{B} r_A$ be the corresponding nonterminal shift arc; then pop, push element $\langle r_A \rangle$ and push element $\langle 0_B \rangle$

return move if q_A is a final state and token cc is in the prospect set associated with q_A , then pop

recognition move if M_A is the axiom machine, q_A is a final state and $cc = \dashv$, then accept and halt

In any other case the analyzer stops and rejects the input string

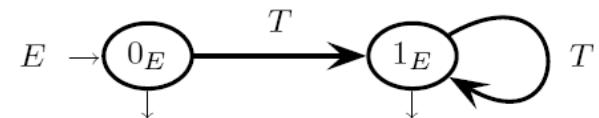
string $x = (a)$

stack	x	predicate
$\langle 0_E \rangle$	$(a) \dashv$	$(\in \gamma = \{ a () \dashv \})$
$\langle 1_E \rangle \langle 0_T \rangle$	$(a) \dashv$	scan
$\langle 1_E \rangle \langle 1_T \rangle$	$a) \dashv$	$a \in \gamma = \{ a () \dashv \}$
$\langle 1_E \rangle \langle 2_T \rangle \langle 0_E \rangle$	$a) \dashv$	$a \in \gamma = \{ a () \dashv \}$
$\langle 1_E \rangle \langle 2_T \rangle \langle 1_E \rangle \langle 0_T \rangle$	$a) \dashv$	scan
$\langle 1_E \rangle \langle 2_T \rangle \langle 1_E \rangle \langle 3_T \rangle$	$) \dashv$	$) \in \pi = \{ a () \dashv \})$
$\langle 1_E \rangle \langle 2_T \rangle \langle 1_E \rangle$	$) \dashv$	$) \in \pi = \{ \) \dashv \}$
$\langle 1_E \rangle \langle 2_T \rangle$	$) \dashv$	scan
$\langle 1_E \rangle \langle 3_T \rangle$	\dashv	$\dashv \in \pi = \{ a () \dashv \}$
$\langle 1_E \rangle$	\dashv	$\dashv \in \pi = \{ \) \dashv \}$ accept

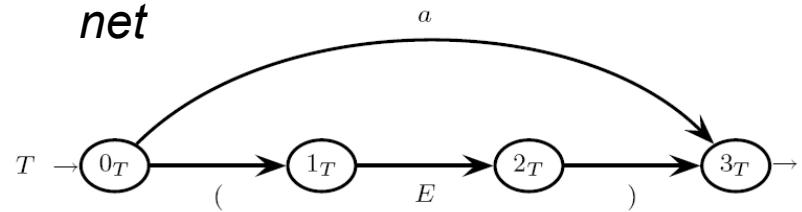
analysis trace of string $x = ()$

0_E	$0 \dashv$	call
$1_E 0_T$	$0 \dashv$	
$1_E 1_T$	$) \dashv$	call
$1_E 2_T 0_E$	$) \dashv$	return
$1_E 2_T$	$) \dashv$	
$1_E 3_T$	\dashv	return
1_E	\dashv	accept

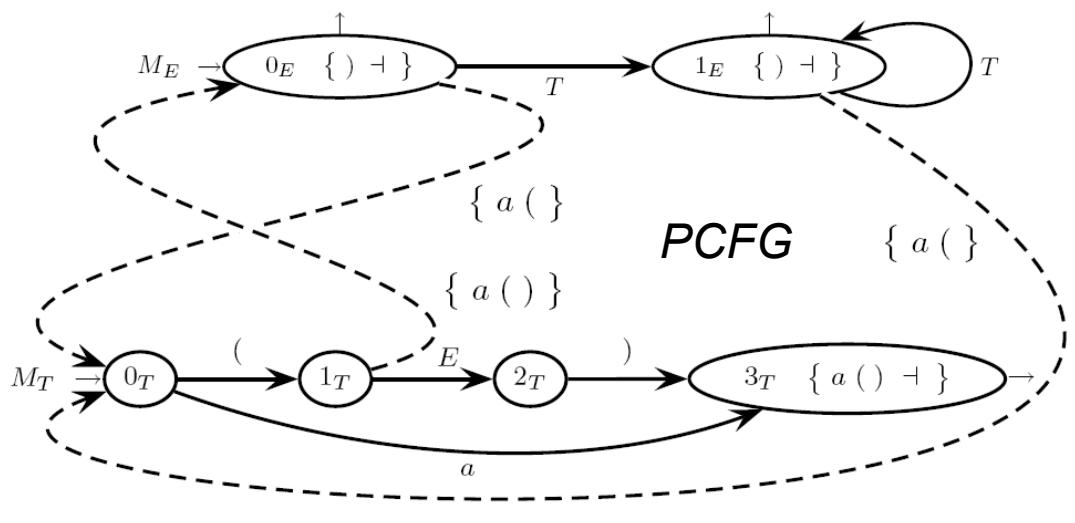
ANALYSIS EXAMPLES



net



two-step
call move



PCFG

analysis trace of string $x = a$		
0_E	$a \dashv$	call
$1_E 0_T$	$a \dashv$	
$1_E 3_T$	\dashv	return
1_E	\dashv	accept

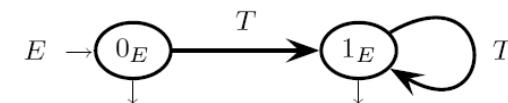
LEFTMOST DERIVATION – RIGHT LINEARIZED GRAMMAR

stack	x	<i>leftmost derivation of the right linearized grammar</i>
$\langle 0_E \rangle$	$(a) \dashv$	$0_E \Rightarrow 0_T 1_E$
$\langle 1_E \rangle \langle 0_T \rangle$	$(a) \dashv$	$0_E \xrightarrow{+} (1_T 1_E)$
$\langle 1_E \rangle \langle 1_T \rangle$	$a) \dashv$	$0_E \xrightarrow{+} (0_E 2_T 1_E)$
$\langle 1_E \rangle \langle 2_T \rangle \langle 0_E \rangle$	$a) \dashv$	$0_E \xrightarrow{+} (0_T 1_E 2_T 1_E)$
$\langle 1_E \rangle \langle 2_T \rangle \langle 1_E \rangle \langle 0_T \rangle$	$a) \dashv$	$0_E \xrightarrow{+} (a 3_T 1_E 2_T 1_E)$
$\langle 1_E \rangle \langle 2_T \rangle \langle 1_E \rangle \langle 3_T \rangle$	$) \dashv$	$0_E \xrightarrow{+} (a \varepsilon 1_E 2_T 1_E)$
$\langle 1_E \rangle \langle 2_T \rangle \langle 1_E \rangle$	$) \dashv$	$0_E \xrightarrow{+} (a \varepsilon 2_T 1_E)$
$\langle 1_E \rangle \langle 2_T \rangle$	$) \dashv$	$0_E \xrightarrow{+} (a) 3_T 1_E$
$\langle 1_E \rangle \langle 3_T \rangle$	\dashv	$0_E \xrightarrow{+} (a) \varepsilon 1_E$
$\langle 1_E \rangle$	\dashv	$0_E \xrightarrow{+} (a) \varepsilon$
$E \Rightarrow T \Rightarrow (E) \Rightarrow (T) \Rightarrow (a)$		

leftmost derivation of the EBNF grammar

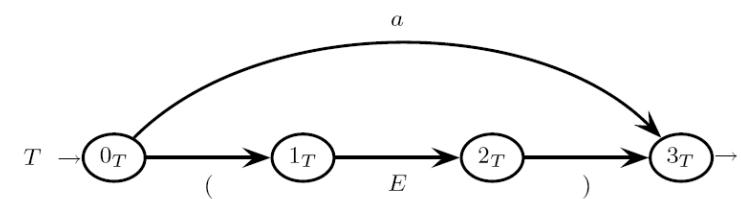
$E \rightarrow T^*$
$T \rightarrow ' (' E ') ' \mid a$

EBNF
grammar



$0_E \rightarrow 0_T 1_E \mid \varepsilon$
$1_E \rightarrow 0_T 1_E \mid \varepsilon$

right linear
grammar
of M_E



$0_T \rightarrow a 3_T \mid ' (' 1_T$
$2_T \rightarrow ') ' 3_T$
$1_T \rightarrow 0_E 2_T$
$3_T \rightarrow \varepsilon$

right linear
grammar
of M_T

IMPLEMENTING THE PARSER BY MEANS OF RECURSIVE PROCEDURES

Each machine becomes a procedure without parameters
⇒ we have one procedure per each non-terminal

Call move ⇒ procedure call

Scan move ⇒ call procedure *next* (the interface to the lexical analyzer or scanner)

Return move ⇒ return from procedure

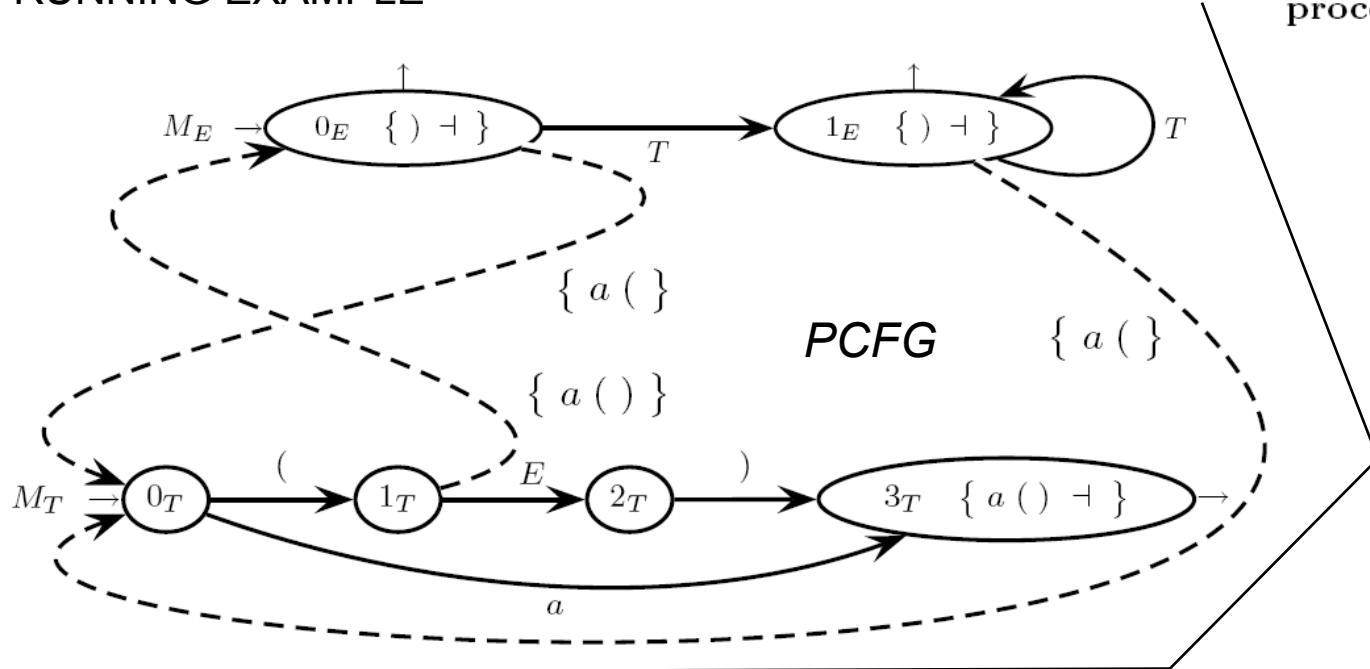
Parser Control Flow Graph becomes the control graph of the procedure

Use the guide sets and the prospect sets to choose which move to execute

The analysis starts by calling the axiomatic procedure

Technique also known as recursive descent

RUNNING EXAMPLE



RECURSIVE DESCENT PARSER

```
program ELL_PARSER
    cc = next
    call E
    if cc ∈ { ->} then accept
    else reject end if
end program
```

procedure E

```
-- optimized
while cc ∈ { a () } do
    call T
end while
if cc ∈ { ) ->} then
    return
else
    error
end if
end procedure
```

procedure T

```
-- state 0_T
if cc ∈ { a } then
    cc = next
else if cc ∈ { ( ) } then
    cc = next
-- state 1_T
if cc ∈ { a () } then
    call E
else
    error
end if
-- state 2_T
if cc ∈ { ) } then
    cc = next
else
    error
end if
-- state 3_T
if cc ∈ { a () ->} then
    return
else
    error
end if
end procedure
```

DIRECT SIMPLIFIED CONSTRUCTION OF THE PCFG

No need to build the full *ELR* pilot graph and derive the *PCFG* from it

Directly put call arcs on the machine net and annotate the net with the *prospect* and *guide* sets

Sets of recursive equations to compute the prospect sets and then the guide sets

prospect set π

- we (re)use the rules already seen for computing the look-ahead sets of *ELR*
- the recursive rules below compute a prospect set for each state of the *PCFG*, though the ELL condition uses only those in the final states (with a final dart)

For an initial state 0_A

$$\pi_{0_A} := \pi_{0_A} \cup \bigcup_{q_i \xrightarrow{A} r_i} \left(\text{Init}(L(r_i)) \cup \text{if } \text{Nullable}(L(r_i)) \text{ then } \pi_{q_i} \text{ else } \emptyset \right)$$

the same chars in π may originate
from different terms of the equation

For any other state q

$$\pi_q := \bigcup_{p_i \xrightarrow{X_i} q} \pi_{p_i}$$

initialization – set to \dashv the prospect set
of the initial state of the axiomatic machine
and to empty all the other prospect sets

GUIDE SET Gui – essentially the same clauses of the already seen definition

Equation for the guide set of a call arc (uses the prospect set)

$$Gui(q_A \dashrightarrow 0_{A_1}) := \bigcup \begin{cases} Ini(L(A_1)) \\ \hline \text{if } Nullable(A_1) \text{ then } Ini(L(r_A)) \\ \text{else } \emptyset \text{ endif} \\ \hline \text{if } Nullable(A_1) \wedge Nullable(L(r_A)) \text{ then } \pi_{r_A} \\ \text{else } \emptyset \text{ endif} \\ \hline \bigcup_{0_{A_1} \dashrightarrow 0_{B_i}} Gui(0_{A_1} \dashrightarrow 0_{B_i}) \end{cases}$$

the same chars in Gui
may come from different
terms of the equation

Furthermore (for the final darts and the terminal shift arcs)

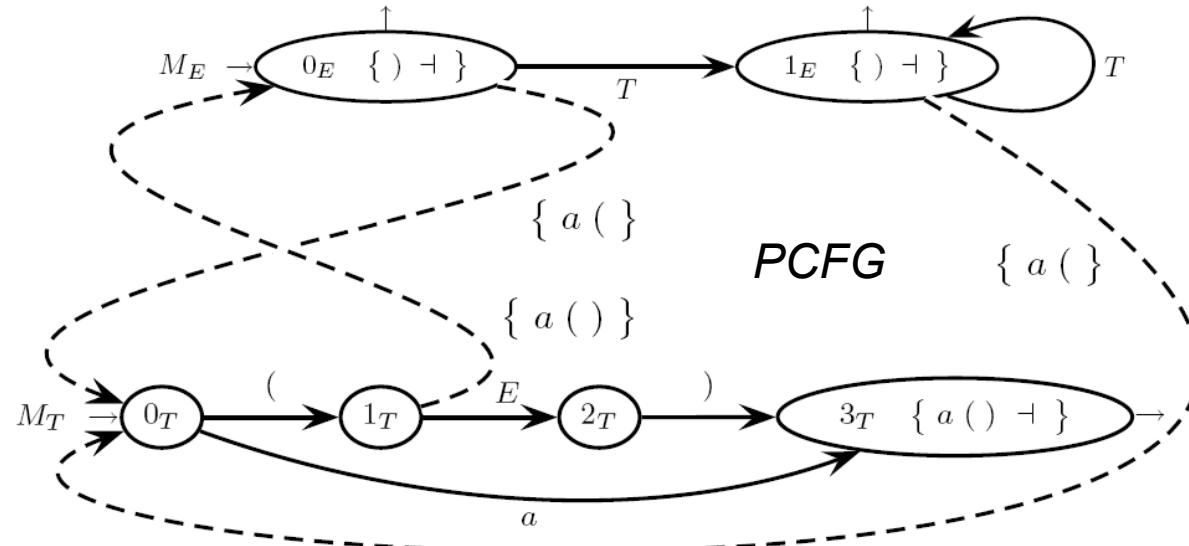
$$Gui(f_A \rightarrow) := \pi_{f_A}$$

f_a is a final state

$$Gui\left(q_A \xrightarrow{a} r_A\right) := \{ a \}$$

initialization – set to empty all the guide sets

RUNNING EXAMPLE – VERY FEW ITERATIONS SUFFICE



1. first iteratively compute the prospect sets
2. then iteratively compute the guide set

$$1_T \xrightarrow{E} 2_T \Rightarrow \pi_{0_E} := \pi_{0_E} \cup \text{Ini}(2_T) = \{\dashv\} \cup \{\} = \{\dashv\}$$

prospect sets

$$0_E \xrightarrow{T} 1_E : 1_E \xrightarrow{T} 1_E \Rightarrow \pi_{0_T} := \pi_{0_T} \cup \text{Ini}(1_E) \cup \pi_{0_E} = \emptyset \cup \{(a) \cup \{\dashv\}\} = \{(a \dashv)\}$$

guide sets

$$\text{Gui}(0_E \rightarrow 0_T) := \text{Ini}(T) = \{ (a \}$$

$$\text{Gui}(1_E \rightarrow 0_T) := \text{Ini}(T) = \{ (a \}$$

$$\text{Gui}(1_T \rightarrow 0_E) := \text{Ini}(E) \cup \text{Ini}(2_T) \cup \text{Gui}(0_E \rightarrow 0_T) = \{(a) \cup \{\}\} \cup \{(a\} = \{(a)\}$$

Prospect sets of						Guide sets of		
0_E	1_E	0_T	1_T	2_T	3_T	$0_E \dashv \rightarrow 0_T$	$1_E \dashv \rightarrow 0_T$	$1_T \dashv \rightarrow 0_E$
\dashv	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset
$\dashv \dashv$	$\dashv \dashv$	$(a \dashv)$	(a)	(a)				
$\dashv \dashv$	$\dashv \dashv$	$(a \dashv)$	(a)	(a)	(a)			

LENGTHENING THE PROSPECTION FOR GRAMMARS THAT ARE NOT *ELL* (1)

If the *ELL* (1) condition is not verified, we can try to modify the grammar and make it of type *ELL* (1)

This approach may take a long time and be a hard work

- sometimes easy, e.g., turning left into right recursion
- or when finding out that the language is regular ...

Alternative approach – use a longer look-ahead

the analyzer looks at a number $k > 1$ of consecutive characters (or tokens) in the input

if the guide sets of length k on alternative moves are disjoint, then the *ELL* (k) analysis is possible

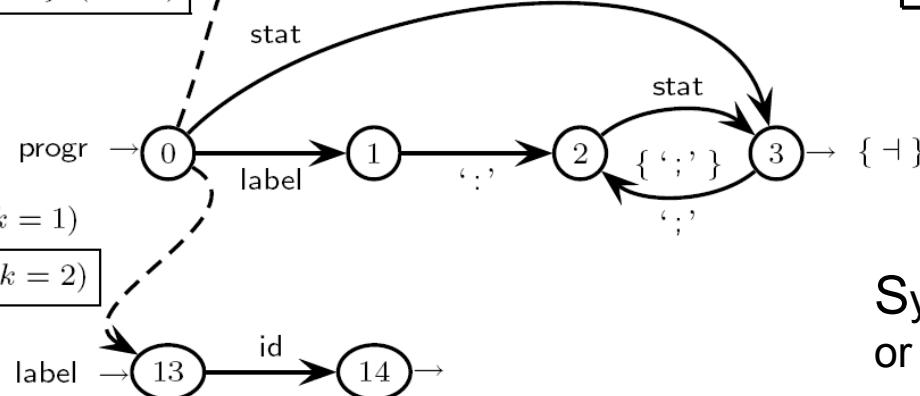
Example with look-ahead length $k = 2$

EXAMPLE – conflict between instruction labels and variable names

partial PCFG

{ id, for } ($k = 1$)

{ id '=', for id } ($k = 2$)



{ id } ($k = 1$)

{ id ':' } ($k = 2$)



$\langle \text{progr} \rangle \rightarrow [\langle \text{label} \rangle \text{ ':' }] \langle \text{stat} \rangle (\text{ ';' } \langle \text{stat} \rangle)^*$

$\langle \text{stat} \rangle \rightarrow \langle \text{assign_stat} \rangle \mid \langle \text{for_stat} \rangle \mid \dots$

$\langle \text{assign_stat} \rangle \rightarrow \text{id} \text{ '=' } \langle \text{expr} \rangle$

$\langle \text{for_stat} \rangle \rightarrow \text{for id} \dots$

$\langle \text{label} \rangle \rightarrow \text{id}$

$\langle \text{expr} \rangle \rightarrow \dots$

NOTE – the guide sets of length $k = 2$ are framed

Symbol "id" may be an instruction label or a variable to be assigned

To distinguish, look at the next token

if "id :" \Rightarrow label

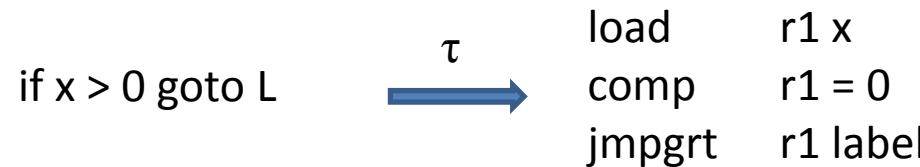
if "id =" \Rightarrow assignment

Purely Syntactic Translations

Prof. A. Morzenti

Translation from a *source* string to a *target* (image) string

The difference between the two strings may be significant, ex.:



? how to specify the translation in order to design the translator in a systematic way?

Central idea: structure-based translation guided by the source language syntax

- **Purely syntactic translation:**

exploits the notions of automata, regular expressions, and grammars

- **syntax directed translation (SDT)**

adds to the grammar certain functions “encoded” in a SW specification language

SDT computes the value of certain variables necessary for the translation (semantic attributes)

translator model known as **attribute grammars**

Outline for purely syntactic translations

1. Abstract definition of translation (as a mathematical relation or function), ambiguity
2. Syntactic translation schemes (translation grammars, i.e., pairs <source-grammar, target-grammar>)
3. Pushdown translator automaton:
 1. nondeterministic
 2. $LL(1)$
 3. $LR(1)$
4. Special cases: finite tranducers, regular translation expressions

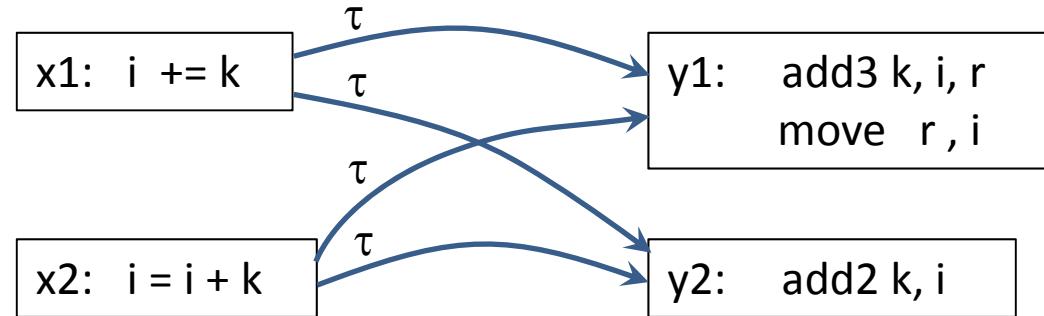
Translation in an abstract setting: a map Source Language \Leftrightarrow Target Language

Ex.: transl. from C to Assembler

it is a **relation**

$$\tau = \{(x_1, y_1), (x_1, y_2), (x_2, y_1), (x_2, y_2)\}$$

The image of x_1 is $\tau(x_1) = \{y_1, y_2\}$

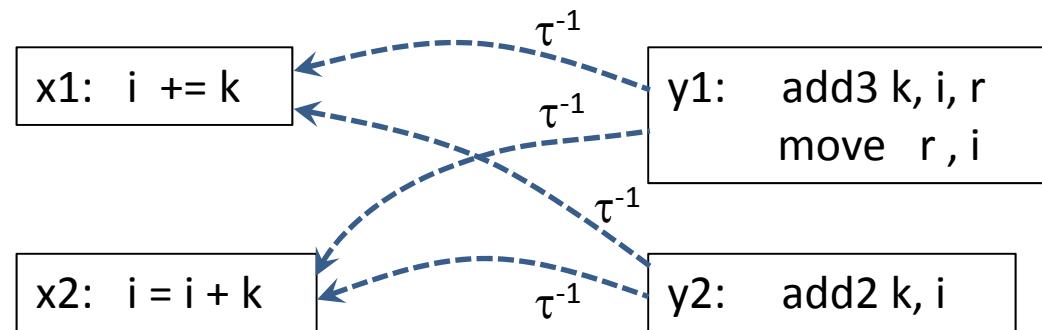


Multiple ways to express the same operation in C, also true in assembly

The translation is **many-valued**, or **not single valued**, or **ambiguous**

The **inverse translation**, τ^{-1} , is **not single valued**, that is, τ is **not injective**

Example: $x_1 \in \tau^{-1}(y_1) = \{x_1, x_2\}$



Abstract Translation: other properties

- **surjective** translation: every sentence of the target language is the image of some sentence of the source language;
 - but a specific program in a machine language might not have any corresponding program in the source language:
 - ex.: certain unstructured loops cannot be written in Pascal
- for a specific compiler (e.g. GCC for IntelX86), every source string has 1 and only 1 image, and the translation computed by the compiler is therefore unique
- **Bijective** (one-to-one) translation: if both τ and τ^{-1} are single valued: ex., in cryptography, encryption and decryption of a text

Syntax translation schemes: introductory example

Image string obtained through

with syntactic means we can make changes to the syntax tree

simple modifications of the syntax tree of the source string
that do not change its structure

(i.e., the nonleaf/nonterminal nodes and the arcs among them)

$$L_1 = \{ a^n b^m \mid n \geq m > 0 \} \quad \tau(a^n b^m) = c^{n-m} d$$

Source grammar G_1

$$S \rightarrow a S$$

$$S \rightarrow A$$

$$A \rightarrow a A b$$

$$A \rightarrow ab$$

Target grammar G_2

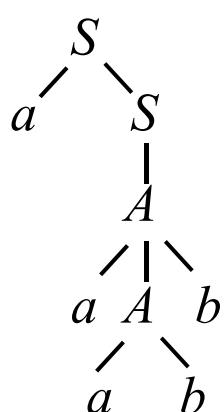
$$S \rightarrow c S$$

$$S \rightarrow A$$

$$A \rightarrow A \quad \text{--> this rule here is necessary because we need the two grammars to be identical in terms of non terminals}$$

$$A \rightarrow d$$

Translation: source tree \Rightarrow target tree



$$\tau(a^3 b^2) = cd$$

is τ^{-1} single valued?

No, in fact the obtained strings are images of all possible string obtained by the infinite possible values of m and n

Translation grammar and scheme

Syntactic translation scheme:

1. 1-1 map between rules of G_1 and G_2 (hence same number of rules)
2. matching rules differ only **in the terminal symbols**
3. in matching rules the **nonterminals** are the **same** and in the **same order**

Consequences of 1. 2. 3. : one can

- combine the scheme into a unique **translation grammar G_t**
- Compute the translation by means of a push down automaton (explained later on), possibly (NB!) ***the same automaton used for syntax analysis***

source gramm. G_1	target gramm. G_2	OK?
$A \rightarrow aBcBD$	$A \rightarrow xBByDy$	YES
$A \rightarrow aBcBD$	$A \rightarrow xBBy$	No: D is missing
$A \rightarrow aBcBD$	$A \rightarrow xBDBy$	No: order of NT changed

Translation grammar and scheme: example

The source and target grammars combined into the **translation grammar**
the terminal part uses “fractions”: num.=source, denom.=target

transl.gramm. G_t

$$S \rightarrow \frac{a}{c} S$$

$$S \rightarrow A$$

$$A \rightarrow \frac{a}{\varepsilon} A \frac{b}{\varepsilon}$$

$$A \rightarrow \frac{ab}{d}$$

source gramm. G_1

$$S \rightarrow aS$$

$$S \rightarrow A$$

$$A \rightarrow aAb$$

$$A \rightarrow ab$$

target gramm. G_2

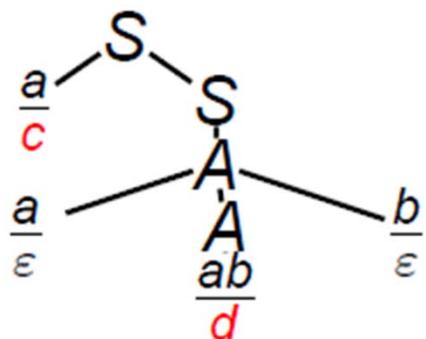
$$S \rightarrow cS$$

$$S \rightarrow A$$

$$A \rightarrow A$$

$$A \rightarrow d$$

Also the source and target syntax trees combined into a unique one
it uses the same «fractions» for the source and target terminal parts



Application: traslation of expressions

Expressions (arithm., logical, ...) with operators (add, sub, . . .)

There exist many representations of expressions

parenthesized functional: $\text{add} (\textit{i1}, \text{mult} (\textit{i2}, \textit{i3}))$

infix: $\textit{i1} + (\textit{i2} \times \textit{i3})$

polish: **prefix:** $\text{add } \textit{i1}, \text{mult } \textit{i2}, \textit{i3}$

postfix: $\textit{i1}, \textit{i2}, \textit{i3} \text{ mult add}$

Polish expressions are concise

value of postfix polish expr. can be computed immediately using a stack

for these reasons they are widely used, e.g. in the Java bytecode

Application: infix → postfix conversion

transl. gramm. G_t

$$E \rightarrow E \frac{+}{\varepsilon} E \frac{\varepsilon}{add}$$

$$E \rightarrow E \frac{-}{\varepsilon} E \frac{\varepsilon}{sub}$$

$$E \rightarrow \frac{(}{\varepsilon} E \frac{)}{\varepsilon}$$

$$E \rightarrow \frac{i}{i}$$

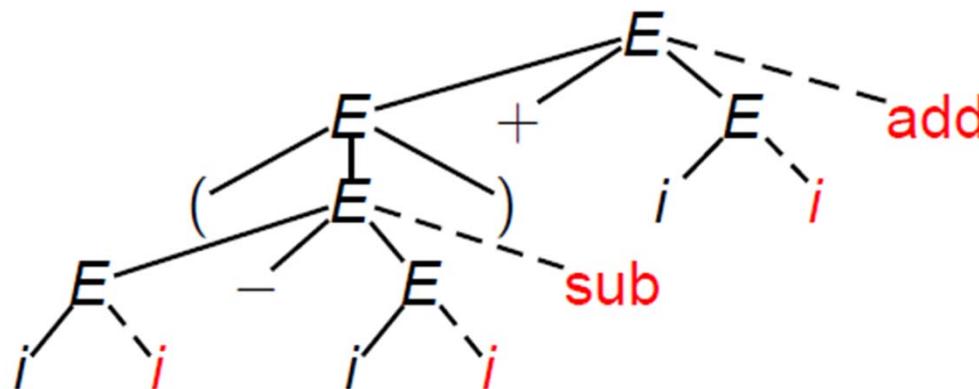
another (equivalent alternative) representation

$$E \rightarrow E + E \{ add \}$$

$$E \rightarrow E - E \{ sub \}$$

$$E \rightarrow (E)$$

$$E \rightarrow i \{ i \}$$



Adjusting the grammar to support the translation

Sometimes it is necessary to modify the source grammar

to obtain a scheme that describes the intended translation

Example: $L = \{ a^n \mid n \geq 1 \}$

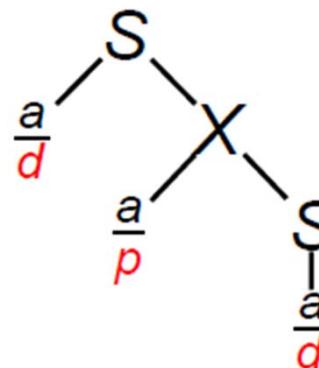
$$\tau(a^n) = \begin{cases} (dp)^{n/2} & n \text{ even} \\ (dp)^{(n-1)/2} d & n \text{ odd} \end{cases}$$

The natural grammar for L : $S \rightarrow aS \mid a$ is unfit; no distinction of even and odd a

Modify G : $S \rightarrow aX \mid a$, $X \rightarrow aS \mid a$

From this G the scheme G_t that defines τ :

$$S \rightarrow \frac{a}{d} X \mid \frac{a}{d}, \quad X \rightarrow \frac{a}{p} S \mid \frac{a}{p}$$

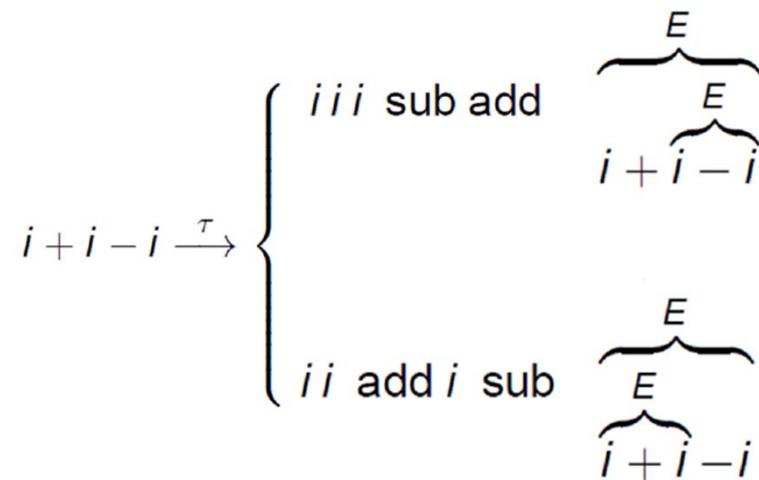


Ambiguous (i.e., many-valued) Translation

A **translation** defined by a scheme $\langle G_1, G_2 \rangle$ is **ambiguous only if G_1** is so

Example: infix to postfix translation with G_1 having bilateral recursion

G_1	G_2
$E \rightarrow E + E$	$E \rightarrow EE \text{ add}$
$E \rightarrow E - E$	$E \rightarrow EE \text{ sub}$
$E \rightarrow i$	$E \rightarrow i$



Ambiguous Translation: another example

When G_1 has duplicated rules

G_1	G_2
$S \rightarrow aS$	$S \rightarrow bS$
$S \rightarrow aS$	$S \rightarrow cS$
$S \rightarrow a$	$S \rightarrow d$

$$aa \xrightarrow{\tau} \{bd, cd\}$$

We can have a grammar which is ambiguous but admits a translation grammar which is not ambiguous

NB: the translation grammar G_t is **not** ambiguous: but is not very useful, we cannot build a translator but a recognizer of the correctness of a translator.

$$G_t : S \rightarrow \frac{a}{b}S \mid \frac{a}{c}S \mid \frac{a}{d}$$

Computing the translation

Analogy with syntax analysis

$$\begin{array}{ccc} \text{grammar} & \Leftrightarrow & \text{push down automaton / parser} \\ \\ \text{translation scheme/grammar} & \Leftrightarrow & \text{push down automaton / parser} \\ & & \text{with } \textit{writing actions} \end{array}$$

We consider the following cases :

1. IO / nondeterministic push down automaton
2. top-down ($LL(1)$) parser with writing actions
3. bottom-up ($LR(1)$) parser with writing actions
4. finite state transducer

From transl. gram. to nondet. IO/automaton with 1 state

A simple extension of the ***predictive*** method of Tab.4.1 of the textbook

$$L = \left\{ a^n b^m \mid n \geq m \geq 1 \right\} \quad \tau(a^n b^m) = c^{n-m} d$$

<u>Rule</u>	<u>Move</u>
1. $S \rightarrow \frac{a}{c} S$	if $cc = a \wedge top = S$ then write(c); pop; push(S); $cc := \text{next}$ end if
2. $S \rightarrow A$	if $top = S$ then pop; push(A) end if
3. $A \rightarrow \frac{a}{\varepsilon} A \frac{b}{\varepsilon}$	if $cc = a \wedge top = A$ then pop; push($\frac{b}{\varepsilon} A$); $cc := \text{next}$ end if
4. $A \rightarrow \frac{ab}{d}$	if $cc = a \wedge top = A$ then pop; push($\frac{b}{d}$); $cc := \text{next}$ end if
5. --	if $cc = b \wedge top = \frac{b}{x}$ then pop; write(x); $cc := \text{next}$ end if
6. --	if $cc = \dashv \wedge \text{empty stack}$ then accept end if halt

- target terminals of G_t are pushed
- they are printed when they appear on top of the stack

In general the ***translator is nondeterministic***, it is a ***purely theoretical construction***

From translation grammar to $ELL(1)$ parser with write actions

It extends the top-down recursive descent parsing technique

Necessary condition: G_1 be $ELL(1)$ (or $ELL(k)$)

Trivial example : $L = (a \mid b)^*$ $\tau(u) = u^R$

$$G_1: \quad S \rightarrow a S \\ S \rightarrow b S \\ S \rightarrow \epsilon$$

$$G_2: \quad S \rightarrow S a \\ S \rightarrow S b \\ S \rightarrow \epsilon$$

$$G_t: S \rightarrow \frac{a}{\epsilon} S \frac{\epsilon}{a} \mid \frac{b}{\epsilon} S \frac{\epsilon}{b} \mid \epsilon$$

```
procedure S
{
    if cc = a then cc := next; call S;
    elseif cc = b then cc := next; call S;
    elseif cc = - then return
    else error
}
```

```
procedure S_withTranslation
{
    if cc = a then cc := next; call S; write(a)
    elseif cc = b then cc := next; call S; write(b)
    elseif cc = - then return
    else error
}
```

BTW: in the Syntax Directed Translation (SDT) method (with attribute grammars)
the parser is enriched with actions that compute the value of semantic attributes

From translation grammars to $ELR(1)$ parser with write actions

Difference w.r.t. the $ELL(1)$ case:

the same idea (adding write actions to the parser) may not work

The writing actions after terminal shifts but before reduction might be premature ...
...because before the reduction nondeterminism has not been resolved yet

Therefore it is appropriate to execute *write actions only at reduction time*

To ensure that, the transl. grammar G_t must be normalized, in the *postfix normal form*

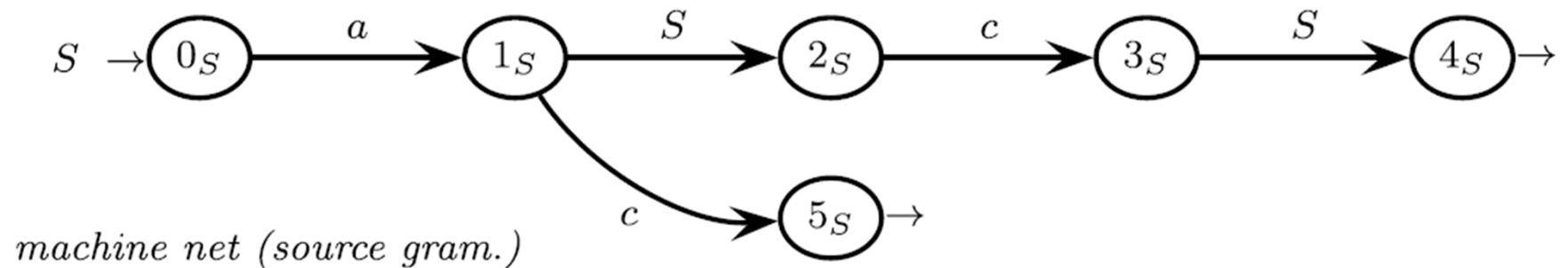
Negative example and conversion to the postfix normal form

Translation of a language similar to Dyck

a and c become b and e , but not the pairs of a, c that do not enclose other ones:

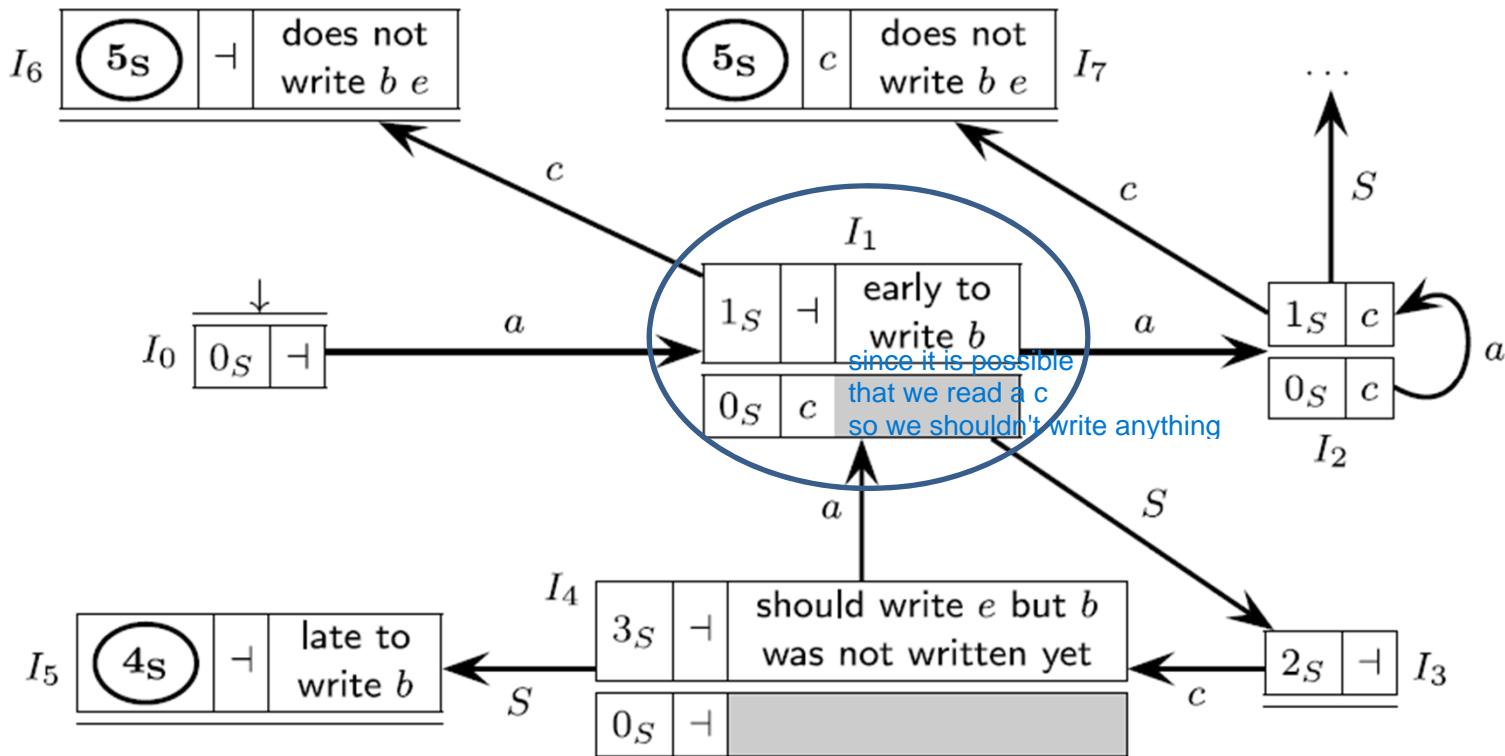
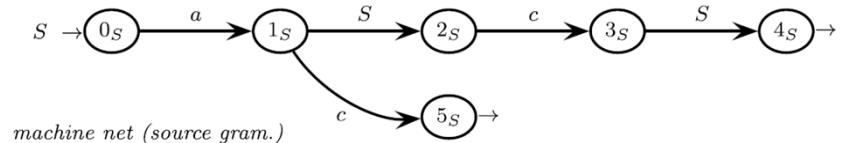
$$G_t: S \rightarrow \frac{a}{b} S \frac{c}{e} S \mid \frac{a}{\varepsilon} \frac{c}{\varepsilon} \quad \tau(ac) = \varepsilon \quad \tau(\textcolor{red}{a} ac \textcolor{blue}{c} ac) = \textcolor{red}{b} \textcolor{blue}{e}$$

Machine for the source grammar, which is $ELR(1)$



Pilot with writing actions: a first try that does not work

$$G_t: S \rightarrow \frac{a}{b} S \frac{c}{e} S \mid \frac{a}{\varepsilon} \frac{c}{\varepsilon}$$



$$\tau(ac) = \varepsilon \text{ while } \tau(\textcolor{red}{a} ac \textcolor{blue}{c} ac) = \textcolor{red}{b} \textcolor{blue}{e}$$

if in I_1 it does not write b then $\tau(ac) = \varepsilon$ (correct), but $\tau(\textcolor{red}{a} ac \textcolor{blue}{c} ac) = \textcolor{blue}{e}$ (incorrect)
 if in I_1 it does write b then $\tau(\textcolor{red}{a} ac \textcolor{blue}{c} ac) = \textcolor{red}{b} \textcolor{blue}{e}$ (correct), but $\tau(ac) = \textcolor{red}{b}$ (incorrect)

Postfix form of the translation grammar $G_t = (G_1, G_2)$

Every rule of the target grammar G_2 has the form (Δ is the target terminal alphabet):

$$A \rightarrow \underbrace{\gamma}_{\in V^*} \quad \underbrace{W}_{\in \Delta^*}$$

that is: first the nonterminals, then the terminals

The gramm. of previous example G_t : $S \rightarrow \frac{a}{b} S \frac{c}{e} S \mid \frac{a}{\varepsilon} \frac{c}{\varepsilon}$ is not in the postfix form

grammar normalization:

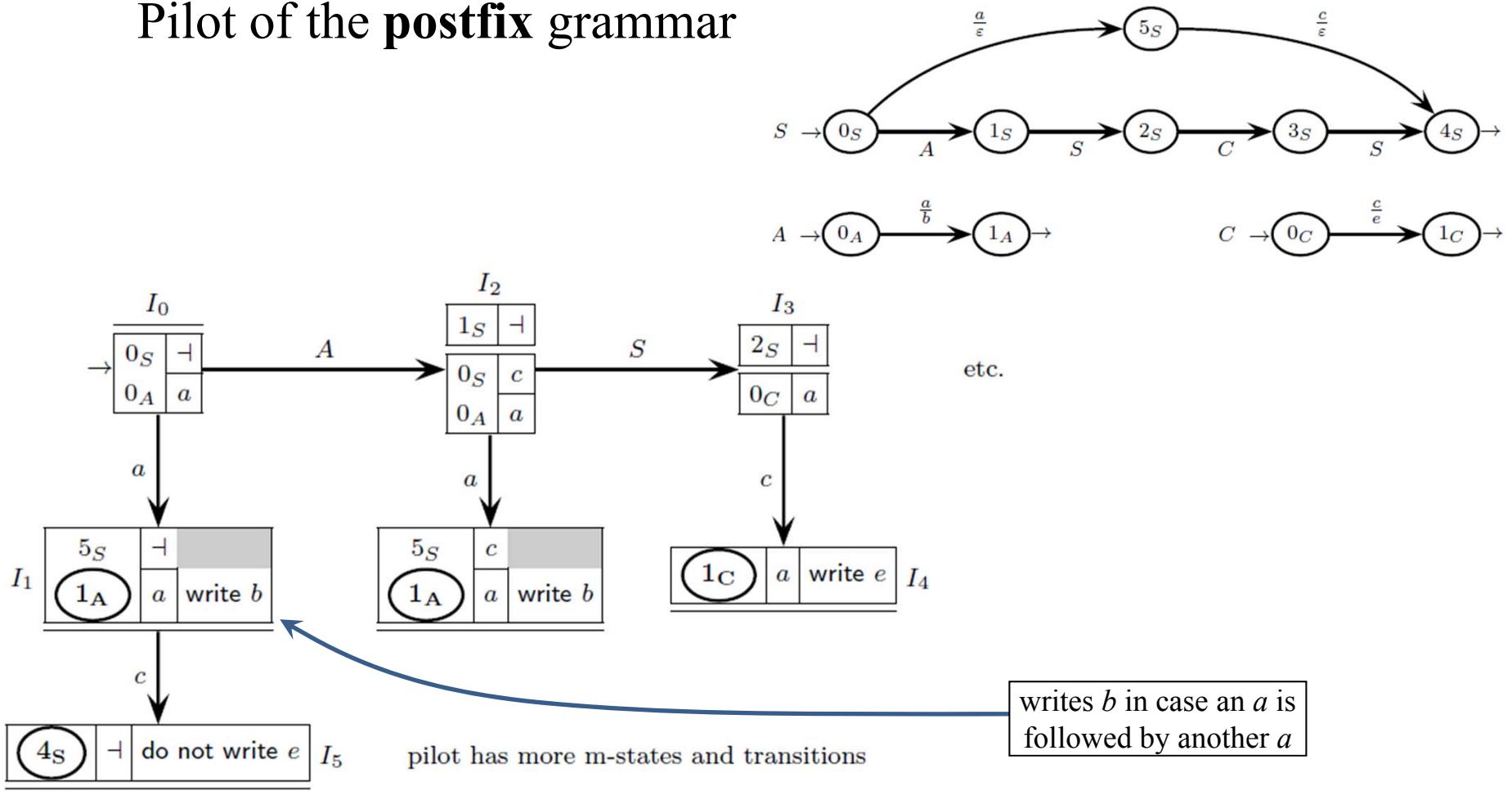
introduce additional nonterminals in place of terminal parts that are not suffix

$$G_\tau: \quad S \rightarrow A S C S \mid ac \quad \begin{array}{c} A \rightarrow \frac{a}{b} \\ C \rightarrow \frac{c}{e} \end{array}$$

$$G_{2\text{postfix}}: \quad S \rightarrow A S C S \mid \varepsilon \quad \begin{array}{c} A \rightarrow b \\ C \rightarrow e \end{array}$$

The new pilot emits the translation only at reduction times

Pilot of the postfix grammar



Drawbacks of the transformation into the postfix form

- it makes the grammar more complex and less readable
- in some cases, it can cause the loss of the $ELR(1)$ property in G_1 (see example on the textbook)

Special cases of syntactic translations : finite state and regular

- Just as free grammars include as special cases ...
 - right-linear grammars (or left-linear grammars), equivalent to
 - regular expressions
 - finite state automata
- ... similarly translation grammars include as special cases the *regular translations*, defined by:
 - regular translation expressions
 - finite transducers or 2I-automata

Right-linear translation grammar

translation grammar G_t

translation

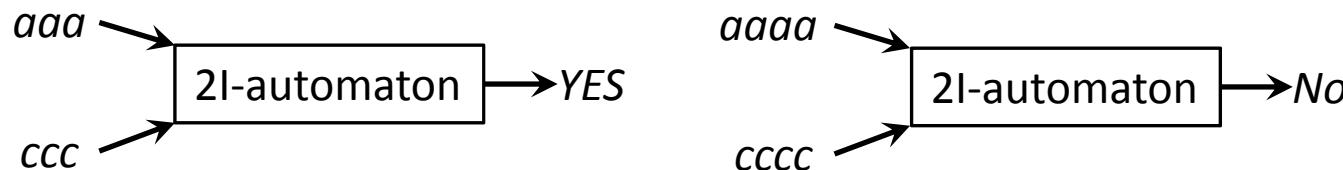
$$\left\{ \begin{array}{l} a^{2n} \xrightarrow{\tau} b^{2n} : n \geq 0 \\ a^{2n+1} \xrightarrow{\tau} c^{2n+1} : n \geq 0 \end{array} \right.$$
$$\left\{ \begin{array}{l} A_0 \rightarrow \frac{a}{c}A_1 \mid \frac{a}{c} \mid \frac{a}{b}A_3 \mid \varepsilon \\ A_1 \rightarrow \frac{a}{c}A_2 \mid \varepsilon \\ A_2 \rightarrow \frac{a}{c}A_1 \\ A_3 \rightarrow \frac{a}{b}A_4 \\ A_4 \rightarrow \frac{a}{b}A_3 \mid \varepsilon \end{array} \right.$$

the finite state automaton A_t that accepts $L(G_t)$ can be viewed in two ways

- machine with two input tapes: 2I-automaton (AKA Rabin & Scott machine)
 - it “**recognizes**” or “**accepts**” the **translation**
- machine with one input tape and one output tape: finite transducer or IO-automaton
 - it “**computes**” the **translation**

Two-input Machine

It accepts the translation relation τ , i.e., a set of pairs of strings $\in \Sigma^* \times \Delta^*$ (Δ is the target alph.)



Transition labels are pairs s written as

$$\frac{a}{b}, \text{ where } a \in \Sigma \cup \varepsilon, b \in \Delta \cup \varepsilon$$

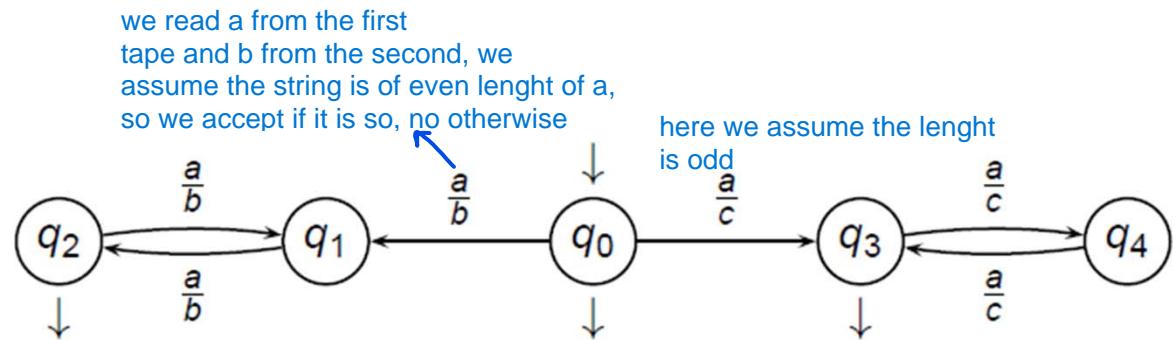
Reading $\frac{a}{b}$ advances both heads on their tape

to accept, both tapes must be completely scanned : $\frac{aaa}{cc} \notin \tau$

2I-automaton or Rabin & Scott machine

Translates to b if the number of a is even
to c if the number of a is odd

$$\left\{ \begin{array}{l} a^{2n} \xrightarrow{\tau} b^{2n} : n \geq 0 \\ a^{2n+1} \xrightarrow{\tau} c^{2n+1} : n \geq 0 \end{array} \right.$$



NB: the automaton is deterministic: two transitions exit from q_0 , but their labels are distinct

Regular Translation Expression

A regular expression containing “fractions”: the previous translation is defined by

$$E_t = \left(\frac{a^2}{b^2} \right)^* \cup \frac{a}{c} \left(\frac{a^2}{c^2} \right)^*$$

Here is the string of fractions $\frac{a}{c} \cdot \frac{a^2}{c^2} \cdot \frac{a^2}{c^2} = \frac{a^5}{c^5} \in E_t$

it corresponds to the source-target pair $(a^5, c^5) \in \tau$

Nivat's theorem

It states the equivalence of various ways to define a translation relation τ :

1. Right- (or left-) linear translation grammar
2. 2I finite automaton
3. Regular translation expression
4. Regular lang. R_t of alphabet Γ and two transliterations $h_1 : \Gamma \rightarrow \Sigma \cup \{\epsilon\}$ (for the source), $h_2 : \Gamma \rightarrow \Delta \cup \{\epsilon\}$ (for the target), such that

transliteration=char. substitution

$$\tau = \{(h_1(z), h_2(z)) \mid z \in R_t\}$$

In the running example, consider

$$R_t = (pp)^* \cup d(dd)^* \quad h_1(p) = h_1(d) = a \quad h_2(p) = b \quad h_2(d) = c$$

$$\begin{array}{ll} h_1 = \text{aaaa} & h_1 = \text{aaa} \\ \overbrace{pppp}^{\text{aaaa}} & \overbrace{ddd}^{\text{aaa}} \\ h_2 = \text{bbbb} & h_2 = \text{ccc} \end{array}$$

Non regular translation of regular languages!

Even if both L_1 and L_2 are regular, the translation is not necessarily regular

$$\text{Ex.: } L_1 = (a \mid b)^* \quad L_2 = (a \mid b)^* \quad \tau(x) = x^R$$

A 2I finite state automaton cannot check if the 2nd tape contains the reflection of the first one

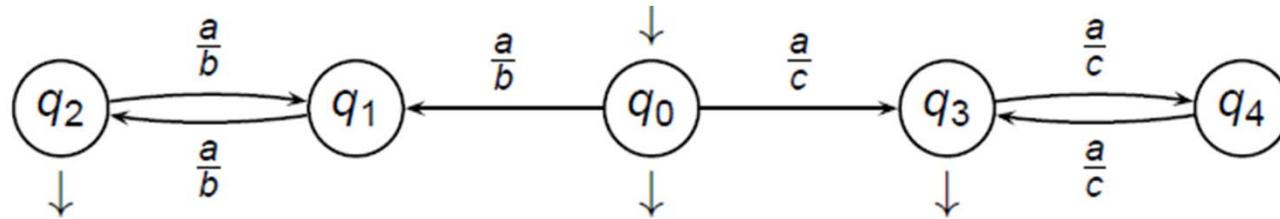
An unbounded stack memory is necessary

Finite transducer or IO-automaton

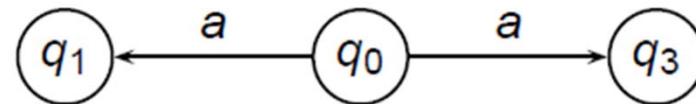
- the second tape is viewed as an output
- the machine *computes* the translation as a function of the source string: $y = \tau(x)$
- Several applications:
 - lexeme (token) recognition in the lexical analysis (see lessons on Flex)
 - transformation of simple texts, or signal sequences (ex. genome computing)
 - natural language processing: conjugation of verbs, declination of names
- Determinism: an IO-automaton is deterministic if so is the subjacent automaton (obtained by canceling the output Δ)

nondeterministic IO-automaton

A deterministic 2I-automaton, viewed as an IO-automaton, can be nondeterministic!



The subjacent automaton is nondeterministic in q_0



- There does not exist any deterministic IO-automaton for this translation
- Unlike finite state automata, IO-automata cannot always be made deterministic

Last translation model, a finite state translator used in applications:

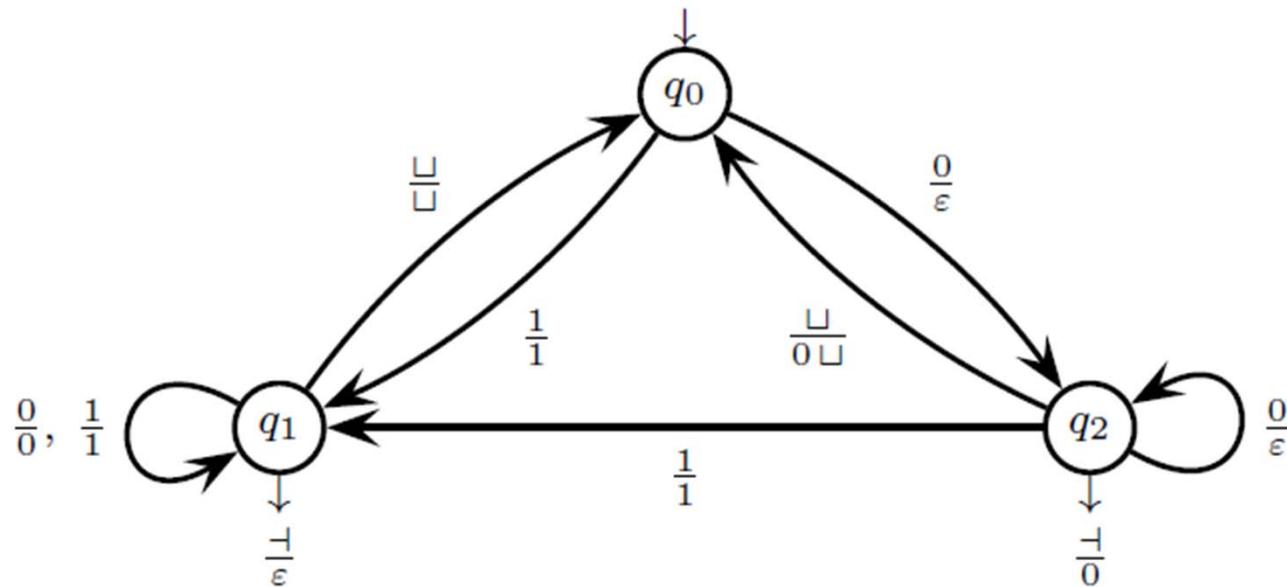
sequential transducer

It is a variation of the deterministic IO-automaton model :

- the *transition function* computes the next state
- while executing the transition, the *output function* emits a string
- when the computation terminates in a certain final state, the *final function* appends a string s to the output
 - This is represented by a label « \dashv/s » on the dart exiting the final states

Example of sequential transducer

Given a series of binary numbers separated by spaces, eliminate the unsignificant zeroes



When terminating in q_2 it emits a zero, but it does not emit anything when terminating in q_1

Static Program Analysis

Prof. A. Morzenti

COMPILATION:

FIRST STEP: translates a program into an intermediate representation that is closer to machine language

SECOND STEP: applied to the intermediate code, can have several purposes:

- VERIFICATION – further check of program correctness
- OPTIMIZATION – transform the program to improve execution efficiency (variable allocation to registers, ...)
- SCHEDULING – modify instruction execution sequence to improve exploitation of pipeline and of processor functional units

It is convenient to represent the ***program control-flow graph***
(on which the previous tasks are based) as an automaton

WARNING: each automaton defines ***one*** program ***not*** the entire source language!
The approach is quite different from that used so far

IN THIS CASE: a ***string accepted*** by the automaton is a possible ***execution trace***

STATIC ANALYSIS: the study of certain properties of the control flow graph of a program,
using the methods of automata theory, logics, or statistics
We only consider methods based on automata

THE PROGRAM AS AN AUTOMATON

- we consider only the simplest instructions (those of the intermediate representation):
 - simple variables and constants
 - variable assignments
 - simple arithmetic, relational, logic operations
 - assignment and conditional instructions not iterative ones
- we consider only INTRAPROCEDURAL ANALYSIS (not interprocedural)

PROGRAM CONTROL FLOW:

- every ***node*** is an instruction
- if instr. p is immediately followed by instr. q then graph has arc $p \rightarrow q$
an arc is also called ***program point***
- first program instruction is the initial node
- an instruction with no successor is an exit (final) node
- nonconditional instructions have at most one successor, conditional ones have one or more
- an instruction with many predecessors is a ***confluence node***

we are doing static analysis so we don't take input data, so we can't check conditions (if/else), so we follow both branches

the control-flow graph is not a completely faithful program representation:

- the TRUE/FALSE value in conditional instructions not represented
- **assignment, read, write**, instructions are substituted by the following abstractions:
 - variable *assignment* and *reading* ... *define* that variable
 - a variable occurrence in the right part of an assignment, in an expression, in a write operation ... *uses* that variable
 - hence in the graph every node (instruction) p is associated with two sets:

$$\mathbf{def}(p) \quad \text{and} \quad \mathbf{use}(p)$$

The variable in def depends on the variables in use

$$p : a := a \oplus b$$

$$\mathbf{def}(p) = \{a\}, \mathbf{use}(p) = \{a, b\}$$

The sets associated to instruction q is the same as the one of instruction w.
Since there isn't anything in the use set, the variable in the def set doesn't depend on any other variable

$$q : \mathbf{read}(a) \quad \mathbf{def}(q) = \{a\}, \mathbf{use}(q) = \emptyset$$

$$w : a := 7 \quad \mathbf{def}(w) = \{a\}, \mathbf{use}(w) = \emptyset$$

The set $\mathbf{def}(p)$ can include more than one variable in case of read instructions

such as $\mathbf{read}(a, b, c)$

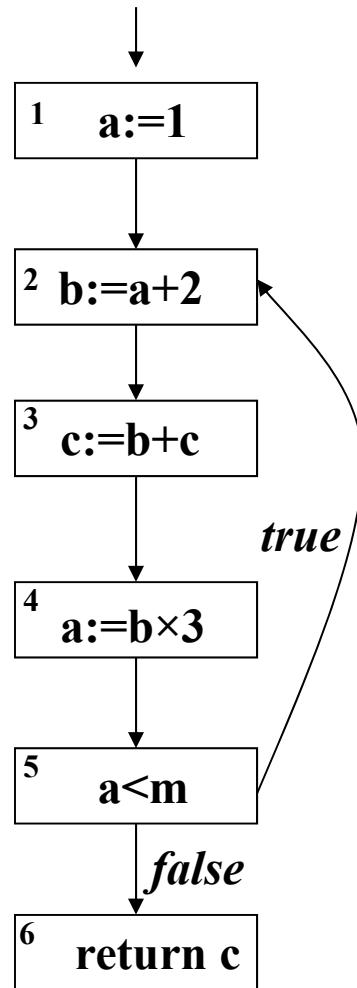
Example 1 – block diagram and control-flow graph

program

```

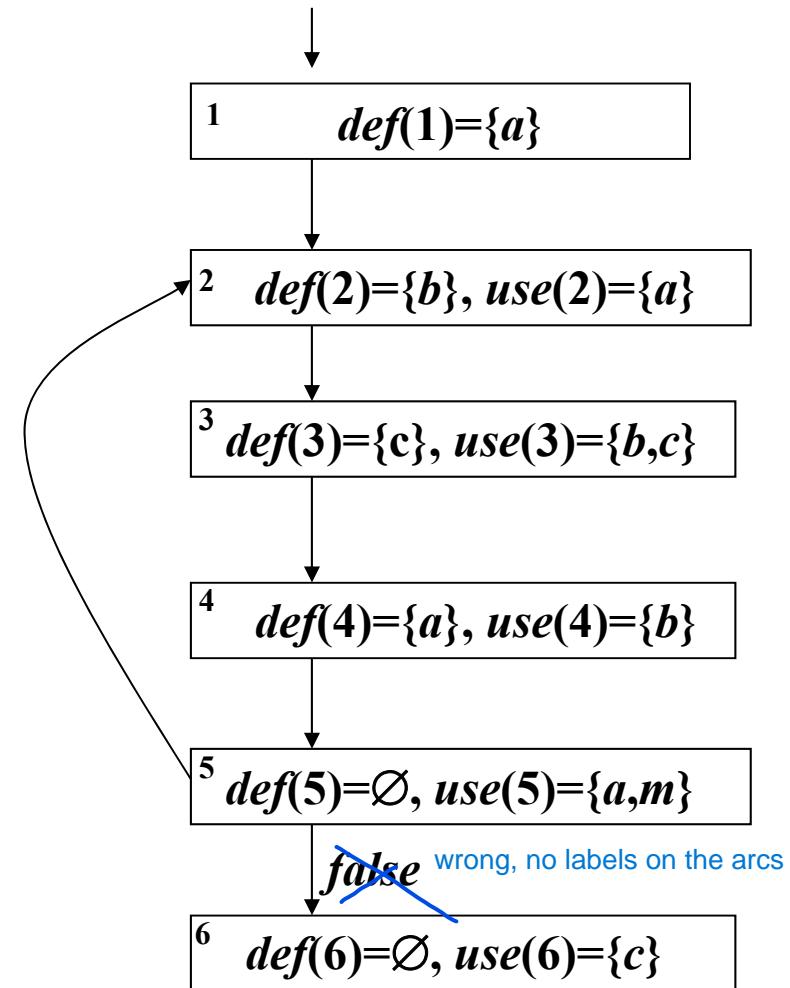
a:=1
e1: b:=a+2
      c:=b+c
      a:=b×3
      if a<m goto e1
      return c
    
```

block diagram



represents the same graph as the block diagram but abstracting the instructions by representing only their def and use sets.

control-flow graph



DEFINITION: LANGUAGE OF THE CONTROL-FLOW GRAPH

the finite state automaton A corresponding to the control-flow graph has

- **terminal alphabet**: the set of program instructions I , each represented by the triple

$$\langle n, \text{def}(n) = \{\dots\}, \text{use}(n) = \{\dots\} \rangle$$

- **initial state** : the state with no predecessor
- **final states** : the nodes with no successor

The language $L(A)$ is the set of strings over alphabet I labeling paths from initial to final states

A path represents an instruction sequence that the machine can execute when the program is launched

Previous example : $I = \{1 \dots 6\}$

An accepted path is $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 = 1234523456$

The set of paths is the language denoted by $1(2345)^+6$

CONSERVATIVE APPROXIMATION

some paths might not be executable: the control-flow graph ignores the value of the Boolean conditions which determine the execution of conditional statements

1 : if $a^{**}2 \geq 0$ then $istr_2$ else $istr_3$

The accepted language includes the two paths {12,13} but the path 13 cannot be executed

In general, it is **undecidable** if a generic path of the control-flow graph can be executed
(the halting problem of the Turing Machine can be reduced to it ...)

conservative approximation: considering all paths from input to output can lead to the diagnosis of non-existing errors, or to the allocation of unnecessary resources, but it never leads to ignoring actual, existing error conditions

HYPOTHESIS: we assume that the automaton is clean :
every instruction is on a path from the initial node to a final one

otherwise:

- the program execution might never finish
- the program might have instructions that are never executed (*unreachable code*)

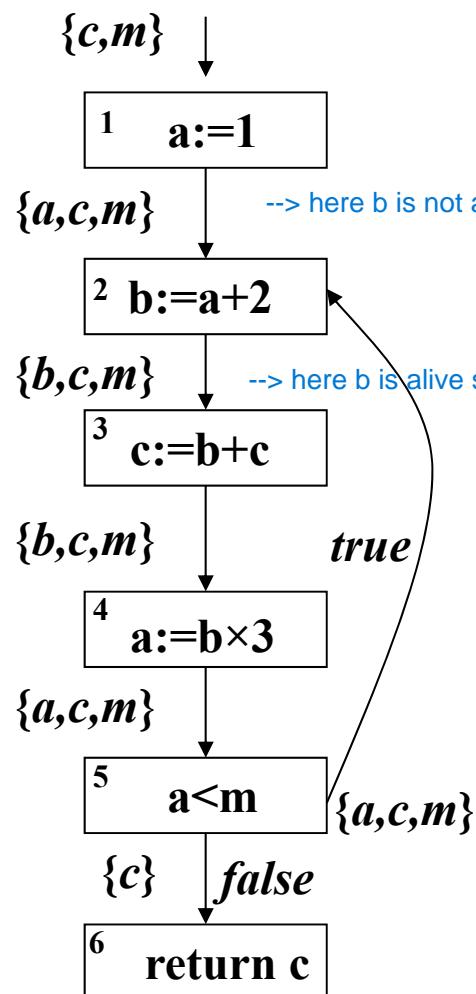
LIVENESS INTERVALS OF VARIABLES

A variable is live at some program point if some instruction that could be executed subsequently uses the value that the variable has at that point (i.e., the variable will be used before being assigned) Important to know which variables should be stored in the registers, which is important to optimization of computation

DEFINITION: a variable a is *live on an arc*, input or output of a program node p , if the graph admits a path from p to a node q such that

- instruction q *uses* a , that is, $a \in \text{use}(q)$ AND
- the path does not traverse an instruction r , $r \neq q$ that defines a , that is, such that $a \in \text{def}(r)$

live variables on arcs



A variable is
 - live-out for a node if it is live on an outgoing arc of that node
 - live-in for a node if it is live on an arc entering that node

EXAMPLE:

c is live-in for node 1 because of the path
 $123: c \in \text{use}(3)$ and neither 1 nor 2 define c

It is customary to define variable liveness on **intervals** (of paths)
 a is live in the **intervals** 12 and 452,
 a is **not** live in intervals 234 and 56

out of node 5 the live variables are $\{a, c, m\} \cup \{c\}$

Notice that a and b are never on the same set, so they can share a register

METHOD TO COMPUTE SETS OF LIVE VARIABLES: DATA-FLOW EQUATIONS

It computes simultaneously all sets of live variables at every point on the graph

It considers all paths from a given point to some instruction that uses a variable

for every final node p :

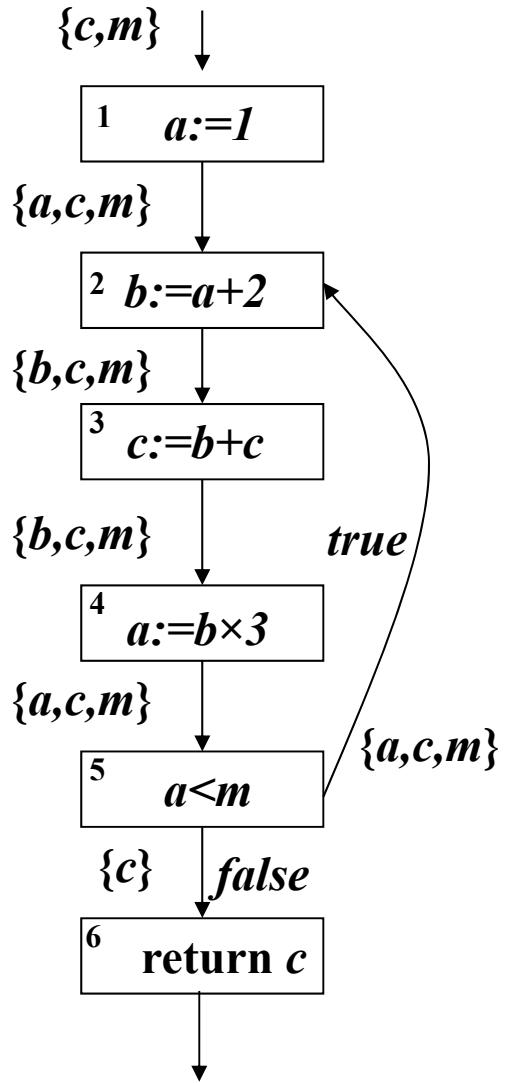
$$live_{out}(p) = \emptyset$$

--> since the program ends there are no more live variables (or are deallocated anyway)

for every other node p :

$$live_{in}(p) = use(p) \cup (live_{out}(p) \setminus def(p))$$

$$live_{out}(p) = \bigcup_{\forall q \in succ(p)} live_{in}(q)$$



for every final node p :

$$(1) \quad \text{live}_{out}(p) = \emptyset$$

for every other node p :

$$(2) \quad \text{live}_{in}(p) = \text{use}(p) \cup (\text{live}_{out}(p) \setminus \text{def}(p))$$

$$(3) \quad \text{live}_{out}(p) = \bigcup_{q \in \text{succ}(p)} \text{live}_{in}(q)$$

From (1): no variable is live at the graph exit

From (2): $\text{live}_{in}(4) = \{b, m, c\} = \{b\} \cup (\{a, c, m\} \setminus \{a\})$

From (3): $\text{succ}(5) = \{2, 6\}$

$$\begin{aligned} \text{live}_{out}(5) &= \text{live}_{in}(2) \cup \text{live}_{in}(6) = \\ &\{a, c, m\} \cup \{c\} = \{a, c, m\} \end{aligned}$$

SOLUTION OF DATA-FLOW EQUATIONS

For a graph with $|I| = n$ nodes one gets a system of $2 \cdot n$ equations in $2 \cdot n$ unknowns $live_{in}(p)$ and $live_{out}(p)$, $\forall p \in I$

The system solution is a vector of $2 \cdot n$ sets

The system is solved iteratively

by initially assigning the empty set \emptyset to every unknown (iteration $i = 0$) :

$$\forall p : live_{in}(p) = \emptyset; \quad live_{out}(p) = \emptyset$$

the values for iteration i are inserted into the system and used to compute values of iteration $i+1$

If at least one of them is different from the previous one, continue

Otherwise stop and the values of iteration $i + 1$ are the solution

(the usual fixpoint technique...)

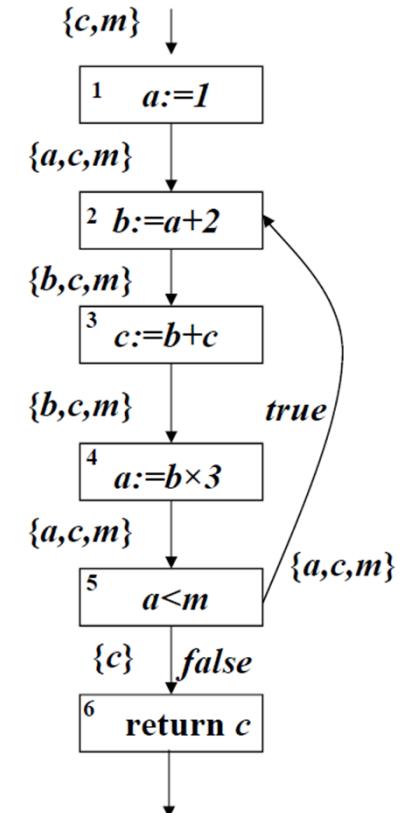
The computation converges after a bounded number of iterations:

- 1) $\text{live}_{in}(p)$ and $\text{live}_{out}(p)$ have a cardinality upperbound determined by the number of program variables
- 2) no iteration will remove elements from sets (these either increase or are unchanged)
- 3) when an iteration does not change any set, the algorithm terminates

we can prove that the algorithm terminates after a quadratic number of steps

Example – Iterative computation of the live variables

1	$in(1) = out(1) \setminus \{a\}$	$out(1) = in(2)$
2	$in(2) = \{a\} \cup (out(2) \setminus \{b\})$	$out(2) = in(3)$
3	$in(3) = \{b, c\} \cup (out(3) \setminus \{c\})$	$out(3) = in(4)$
4	$in(4) = \{b\} \cup (out(4) \setminus \{a\})$	$out(4) = in(5)$
5	$in(5) = \{a, m\} \cup out(5)$	$out(5) = in(2) \cup in(6)$
6	$in(6) = \{c\}$	$out(6) = \emptyset$



NB: at each iteration, we first compute the **in**, then the **out**

	$in = out$	\parallel	in	out	\parallel	in	out	\parallel	in	out	\parallel	in	out	\parallel
1	\emptyset	\parallel	\emptyset	a	\parallel	\emptyset	a, c	\parallel	c	a, c	\parallel	c	a, c, m	\parallel
2	\emptyset	\parallel	a	b, c	\parallel	a, c	b, c	\parallel	a, c	b, c, m	\parallel	a, c, m	b, c, m	\parallel
3	\emptyset	\parallel	b, c	b	\parallel	b, c	b, m	\parallel	b, c, m	b, c, m	\parallel	b, c, m	b, c, m	\parallel
4	\emptyset	\parallel	b	a, m	\parallel	b, m	a, c, m	\parallel	b, c, m	a, c, m	\parallel	b, c, m	a, c, m	\parallel
5	\emptyset	\parallel	a, m	a, c	\parallel	a, c, m	a, c	\parallel	a, c, m	a, c, m	\parallel	a, c, m	a, c, m	\parallel
6	\emptyset	\parallel	c	\emptyset	\parallel	c	\emptyset	\parallel	c	\emptyset	\parallel	c	\emptyset	\parallel

Complexity: $O(n^2)$ in the worst case; hardly higher than linear in practice

APPLICATION: MEMORY ALLOCATION

If two variables are never simultaneous live, they **do not interfere** and can be stored in the same memory cell or register

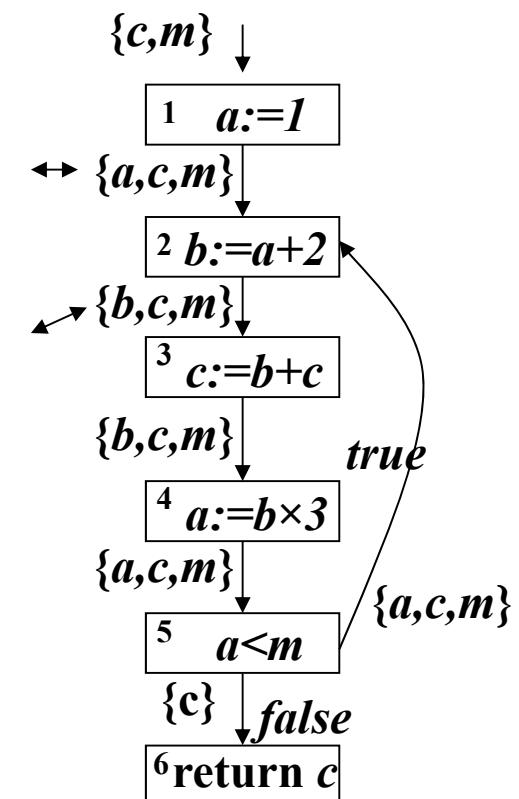
(NB: in general, given n variables there are $n*(n-1)/2$ distinct unordered pairs of variables)

Pairs (a,c) (c,m) and (a,m) interfere
(they are present in $in(2)$)

Pairs (b,c) (b,m) and (c,m) interfere
(they are present in $in(3)$)

a and b do not interfere

the four variables a, b, c, m can be stored in three ‘memory cells’



Modern compilers use such heuristics based on the interference relation to assign registers to variables

APPLICATION: USELESS DEFINITIONS

An instruction defining a variable is useless if the variable is *not live* out of the instruction

To identify useless definitions: for each instruction p defining a variable a check that $a \in out(p)$

In the previous example no definition is useless

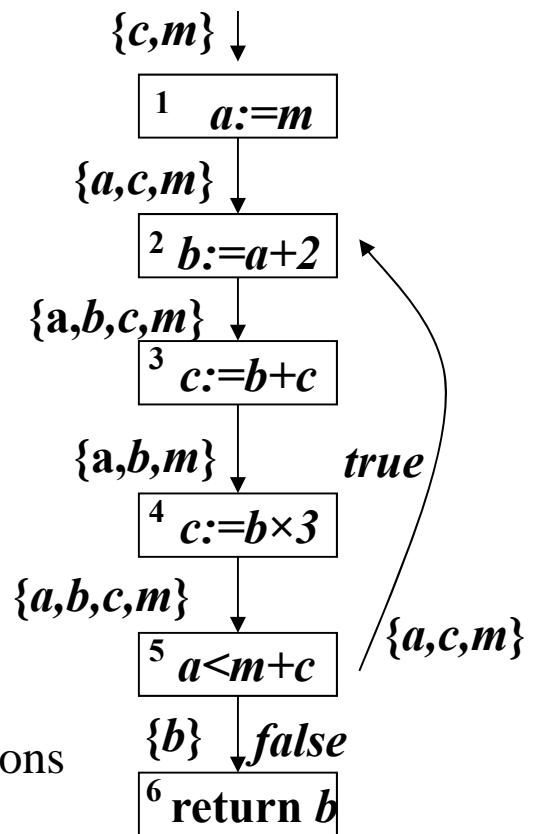
Example – Useless definitions

Variable c is not live out of 3: hence instruction 3 is useless

Removing instruction 3 the program is shortened

c is removed from $in(1)$, $in(2)$, $in(3)$ and $out(5)$

A program improvement typically allows for further optimizations



They ask liveness analysis but can also ask useless definition 17 / 29

Lecture stopped here

ANOTHER USEFUL ANALYSIS: REACHING DEFINITIONS

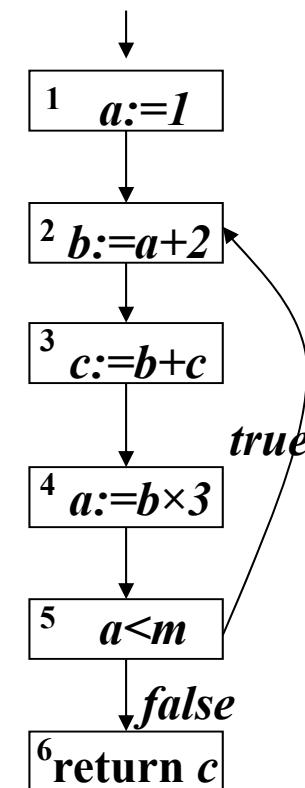
We want to identify the definitions that reach various program points

DEFINITION: A definition of a in q , a_q , reaches the entrance of instruction p (not necessarily distinct from q) if there exists a path from q to p that does not traverse any node, distinct from q , where a is defined

In such a case instruction p is using the value of a as defined by q

Previous example (p.16):

- definition a_1 reaches the entrance of 2, 3, 4 but not of 5
- definition a_4 reaches the entrance of 5, 6, 2, 3, 4



DATA-FLOW EQUATIONS FOR REACHING DEFINITIONS

Computing reaching definitions in various program points as solutions of data-flow equations

If node p defines variable a , we say that

every other definition a_q , $q \neq p$, of a is *suppressed* by p

$$\text{sup}(p) = \{ a_q \mid a \in \text{def}(p) \wedge a \in \text{def}(q) \wedge q \neq p \}$$

NB: recall that $\text{def}(p)$ can include more than one variable in case of read instructions

such as $\text{read}(a, b, c)$

DATA-FLOW EQUATIONS FOR REACHING DEFINITIONS:

Eq. (1) assumes that no varbl. is passed as input parameter

Otherwise $\mathbf{in}(1)$ contains external definitions, denoted e.g. as x ,

For the initial node 1 :

$$(1) \quad \mathbf{in}(1) = \emptyset$$

For every other node $p \in I$:

$$(2) \quad \mathbf{out}(p) = \mathbf{def}(p) \cup (\mathbf{in}(p) \setminus \mathbf{sup}(p))$$

$$(3) \quad \mathbf{in}(p) = \bigcup_{\forall q \in \mathbf{pred}(p)} \mathbf{out}(q)$$

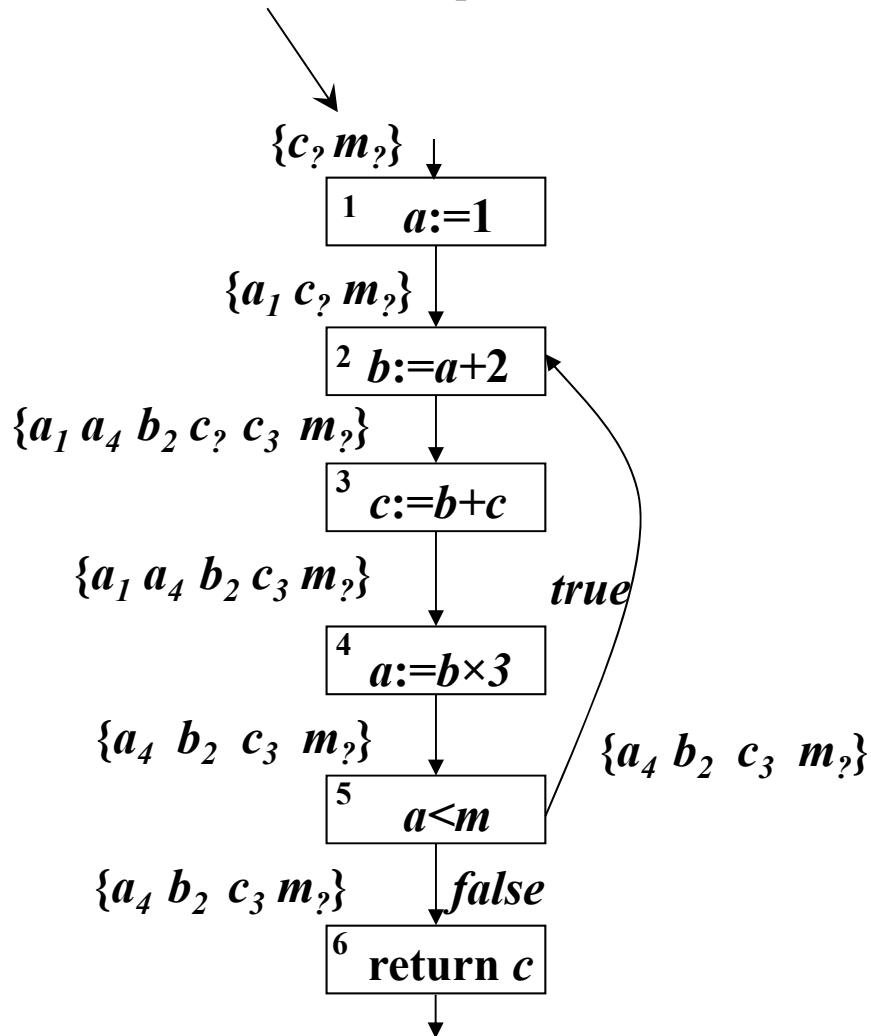
Eq. (2) includes in $\mathbf{out}(p)$ the definitions of p and those reaching the entrance of p , except for those suppressed by p

eq. (3) states that all definitions reaching the exit of some predecessor of p also reach the entrance of p

The equation system is solved by iteration, starting from empty sets, until the first fixed point
 (just as we do for the liveness equations)

Example – Reaching definitions

NB: c and m program parameters, defined in some unknown external point



node	$\ def sup$
1 $a := 1$	$\ a_1 a_4$
2 $b := a + 2$	$\ b_2 \emptyset$
3 $c := b + c$	$\ c_3 c_?$

node	$\ def sup$
4 $a := b \times 3$	$\ a_4 a_1$
5 $a < m$	$\ \emptyset \emptyset$
6 $\text{return } c$	$\ \emptyset \emptyset$

$in(1) = \{c?, m?\}$
 $out(1) = \{a_1\} \cup (in(1) \setminus \{a_4\})$
 $in(2) = out(1) \cup out(5)$
 $out(2) = \{b_2\} \cup (in(2) \setminus \emptyset) = \{b_2\} \cup in(2)$
 $in(3) = out(2)$
 $out(3) = \{c_3\} \cup (in(3) \setminus \{c_?\})$
 $in(4) = out(3)$
 $out(4) = \{a_4\} \cup (in(4) \setminus \{a_1\})$
 $in(5) = out(4)$
 $out(5) = \emptyset \cup (in(5) \setminus \emptyset) = in(5)$
 $in(6) = out(5)$
 $out(6) = \emptyset \cup (in(6) \setminus \emptyset) = in(6)$

NB: b not defined elsewhere

CONSTANT PROPAGATION a very useful and effective optimization technique

consider the possibility to replace a variable with a constant

for instance (following previous example)

it is **not** possible to replace variable a (instr. 2) with constant 1 (assigned by instr. 1: def. a_1)

because the set $in(2)$ includes also another definition of a , namely definition a_4

GENERAL CONDITION

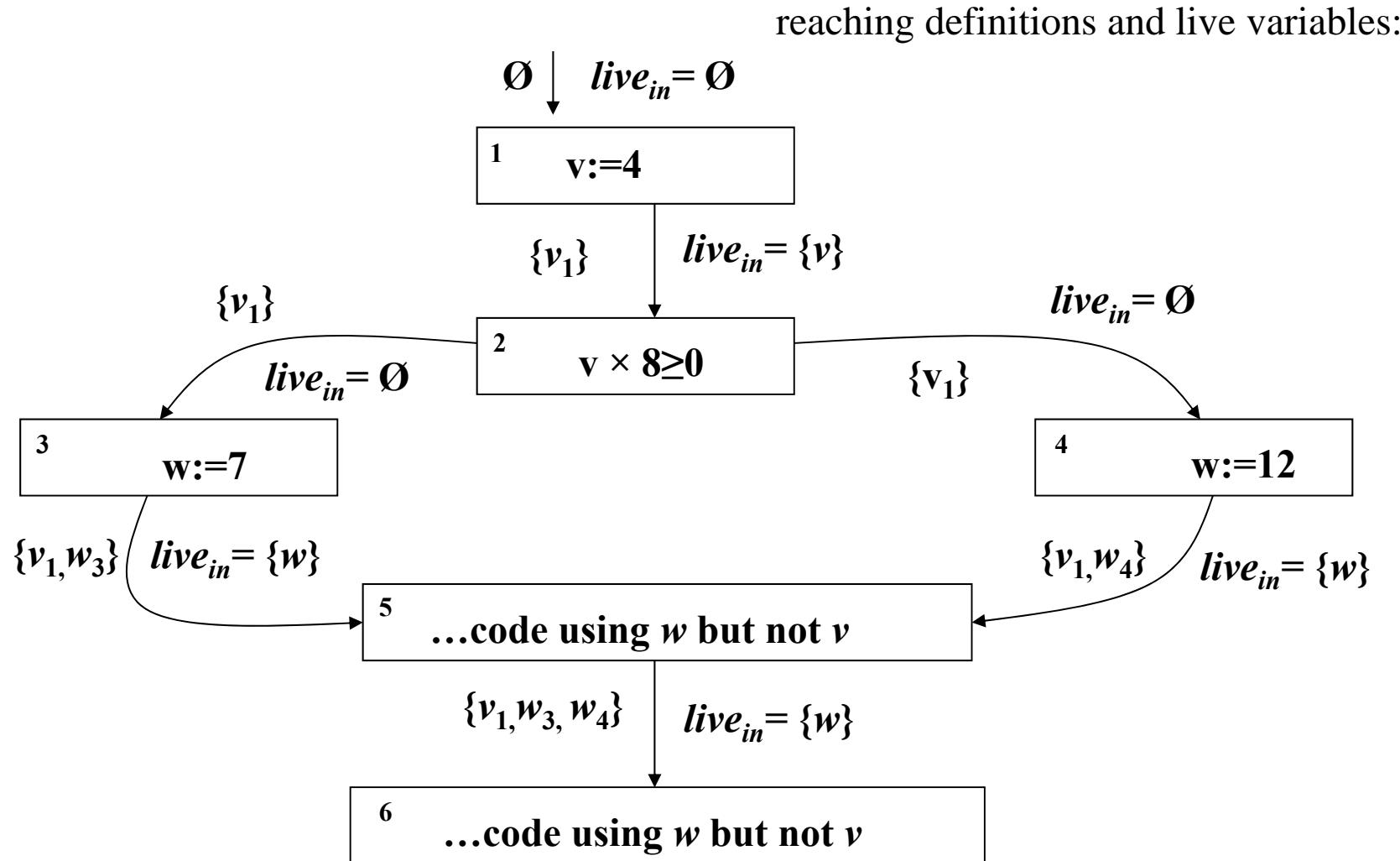
It is possible to replace a variable a , inside instruction p , with constant k if:

- 1) there exists an instruction q : $a := k$, such that definition a_q reaches the entrance of p

AND

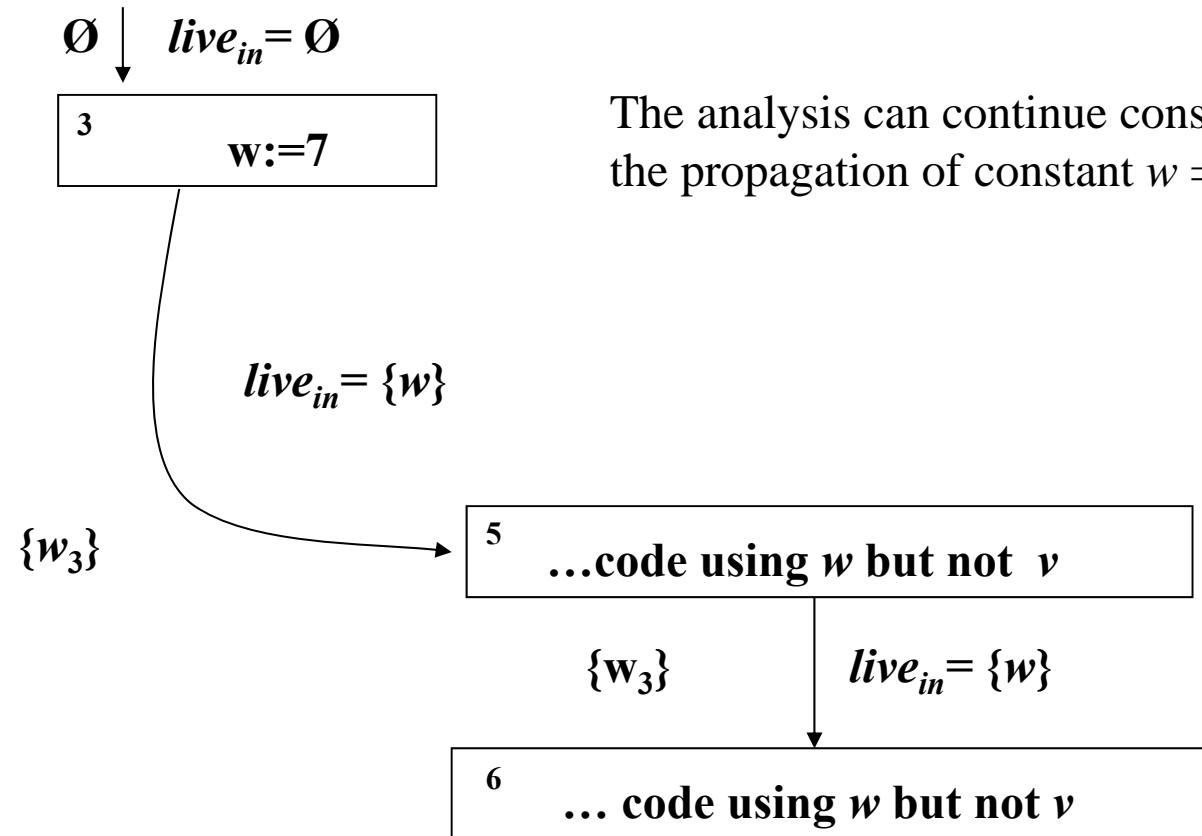
- 2) no other definition a_r , $r \neq q$ reaches the entrance of p

Example – constant propagation (improvements obtained and further induced simplifications)



propagation of constant 4 for v allows one to remove
one of the two branches of the *if* instr.

Simplified program:



The analysis can continue considering
the propagation of constant $w = 7$ to the rest of the program

AVAILABILITY OF VARIABLES AND INITIALIZATIONS

The compiler checks that at execution time every variable, when used, has a value
(it is not “undefined”)

Otherwise the variable is *unavailable* and an error occurs

DEFINITION – A variable a IS AVAILABLE on the entrance to a node p ,
if *every* path from the initial node to the entrance of p includes a definition of a
(we ignore input parameters...)

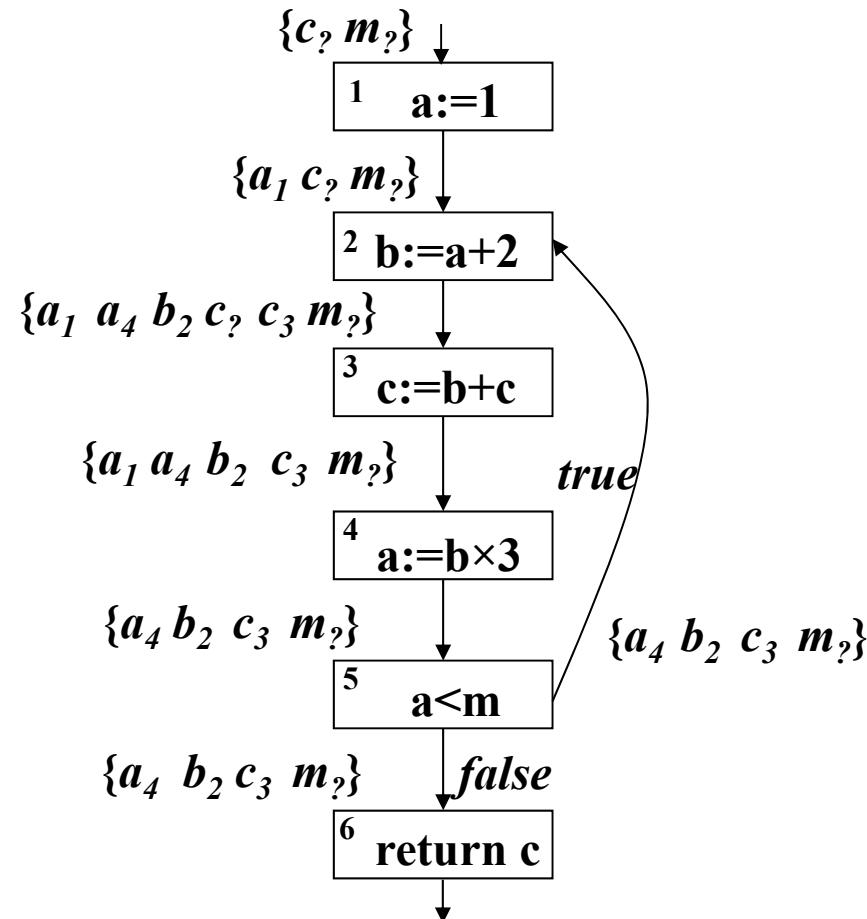
AVAILABILITY OF VARIABLES AND INITIALIZATIONS: Example

3 uses c , to which no value is assigned in path 123

c and m might be input parameters of the subprogram

a available on the entrance of 2 because it was defined in 1 and also in 4

b available on the entrance of 3 because it was defined in 2



AVAILABILITY vs REACHING DEFINITIONS

If a definition a_q reaches the entrance of p , certainly there exists a path from 1 to p that traverses the defining point q .

But there might be *another* path, from 1 to p , not traversing q nor any other definition of a : in this case it is not guaranteed that variable a is available at entrance of p

The concept of availability is stronger (i.e., more constraining) than that of reaching definition: there is a difference in *quantification over paths*

A variable a is **available** on the entrance of p if **some definition of a reaches node p in every circumstance**, i.e., **for every** node q predecessor of p , the set $out(q)$ of definitions reaching the exit of q contains a definition of a

For an instruction q we denote $out'(q)$ the set of variable definitions reaching the exit from q , with subscripts deleted. Ex.: if $out(q)=\{a_1 \ a_4 \ b_3 \ c_6\}$, then $out'(q)=\{a \ b \ c\}$

BADLY INITIALIZED VARIABLES: An instruction p is badly (or not well) initialized if there exists a predecessor q of p , such that the definitions reaching its exit do not include all variables used in p (\Rightarrow when the computation follows a path through q , some variable used in p may not have a value)

formally: **instruction p is not well initialized if**

$$\exists q \in pred(p) \text{ such that } \neg (use(p) \subseteq out'(q))$$

Example – Detecting uninitialized variables

Condition of **bad** (lack of) **initialization**

$$\exists q \in pred(p) \text{ such that } use(p) \not\subseteq out'(q)$$

This condition is false at node 2:

every predecessor (1 and 4)
includes in its set *out*

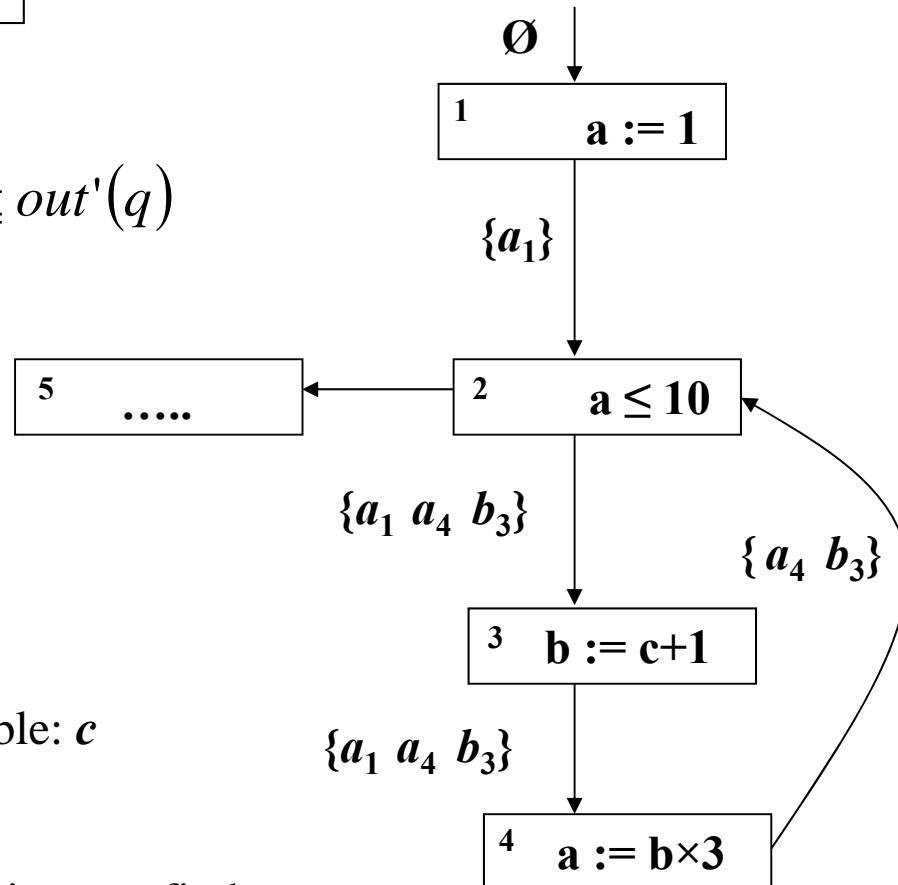
a definition of *a*, the only variable used in 2

It is true at node 3, because there is no
definition of *c* reaching the exit of 2

→Error at 3: instr. 3 uses an uninitialized variable: *c*

Deleting instruction 3, updating the sets
of reaching definitions, re-evaluating the condition one finds
that 4 is not well initialized: def. *b*₃ is no more included in *out*(3)

The analysis might proceed by deleting instr. 4 etc. ...



Attribute Grammars

Prof. A. Morzenti

DOMAIN OF APPLICATION OF ATTRIBUTE GRAMMARS

The compilation process uses tasks that cannot be defined using purely syntactic methods
Examples:

- translation of a decimal (base 10) number to binary
- translation of a record definition, *computing* the offset in central memory of every field

```
BOOK: record
    AUT: char(8); TIT: char(20); PRICE: real; QUANT: int;
end
```



Symbol	Type	Dimension	Address
BOOK	record	34	3401
AUT	string	8	3401
TIT	string	20	3409
PRICE	real	4	3429
QUANT	int	2	3433

Syntax directed translators

They use functions applied to the syntax tree to compute some *semantic attributes*
the values of the attributes constitutes the translation
(⇒ they express the *meaning* of the sentence)

attribute grammars have the same expressive power as the Turing machine
⇒ in fact, they provide a systematic compiler design method,
not a formal model that is easily analyzable, like automata or C.F.Grammars

Compilation is organized in two passes:

1. lexical+syntax analysis produces the syntax tree
2. semantic analysis or evaluation produces the decorated syntax tree
it is designed using attribute grammars

For simplicity the attribute grammar is defined w.r.t. an *abstract syntax*
a grammar that may be simpler than the real one, often ambiguous, but convenient

The ambiguity of the abstract syntax does not prevent a single-valued translation:
the parser will pass to the semantic evaluator only one syntax tree

The simpler compilers may combine the two phases in a single pass
using a unique syntax, the one of the language

Example: computing the value of a binary fractional number

Source language: $L = \{0, 1\}^+ \bullet \{0, 1\}^+$ (dot ‘•’ separates integer and fractional parts)

Translation of string $1101 \bullet 01 \in \{0, 1\}^+ \bullet \{0, 1\}^+$ is $13,25 \in \mathbb{R}$ (NB: it is a number, not a string)

Base syntax: $\{N \rightarrow D \bullet D, \quad D \rightarrow DB, \quad D \rightarrow B, \quad B \rightarrow 0, \quad B \rightarrow 1\}$

Attributes and their meaning:

Attribute	Meaning	Domain	Nonterminals that possess the attribute
v	value	decimal number	N, D, B
l	length	integer	D

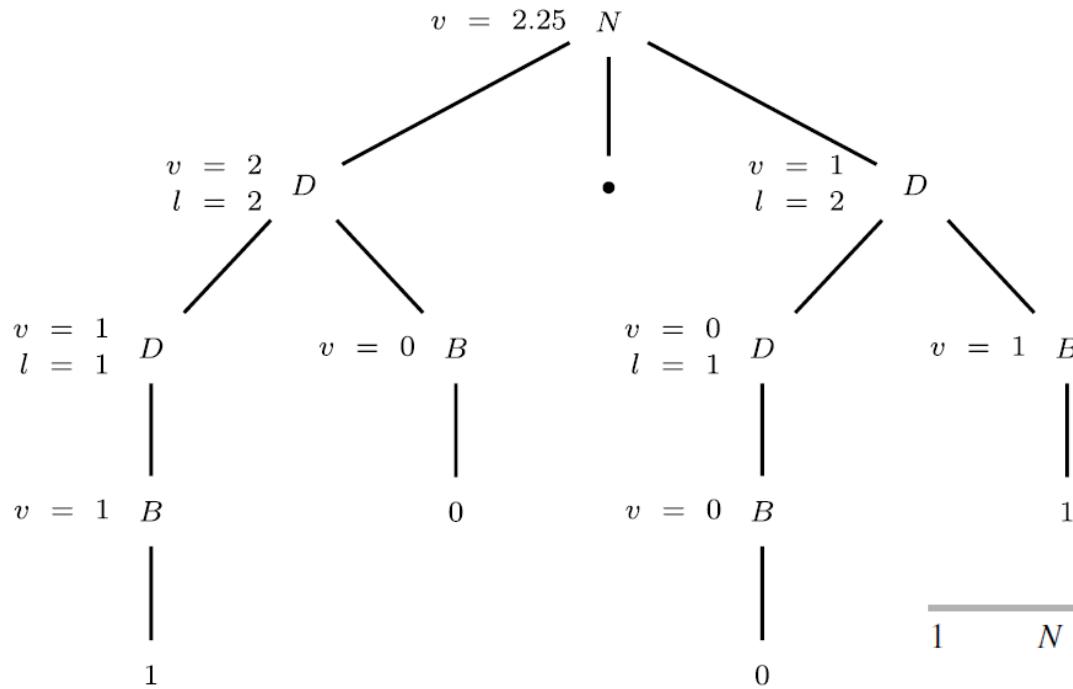
inside rules, symbol instances are numbered with subscripts ≥ 0 to be uniquely identified
semantic functions (i.e., assignments of values to attributes) are associated with syntax rules

#	Syntax	Semantic functions		Comment
1	$N \rightarrow D \bullet D$	$v_0 := v_1 + v_2 \times 2^{-l_2}$		Add integer to fractional value divide by weight 2^{l_2}
2	$D \rightarrow DB$	$v_0 := 2 \times v_1 + v_2$	$l_0 := l_1 + 1$	Compute value and length
3	$D \rightarrow B$	$v_0 := v_1$	$l_0 := 1$	
4	$B \rightarrow 0$	$v_0 := 0$		Value initialization
5	$B \rightarrow 1$	$v_0 := 1$		

we're writing functions for assignign to non terminal on the righten side values based on the leftmost side. That can be seen as assigning with some function to the left side the return value to a call to the procedure associated to the non terminal in the right side

semantic functions are applied following the dependences among attributes
 starting from attributes whose value is known
 often the initial values are in the leaves, possibly precomputed by the lexical analysis

«translation» or «meaning» of a sentence is the value of some attribute
 often these attributes are associated with the tree root



notice that attribute evaluation
 goes bottom-up

1	$N \rightarrow D \bullet D$	$v_0 := v_1 + v_2 \times 2^{-l_2}$	
2	$D \rightarrow DB$	$v_0 := 2 \times v_1 + v_2$	$l_0 := l_1 + 1$
3	$D \rightarrow B$	$v_0 := v_1$	$l_0 := 1$
4	$B \rightarrow 0$	$v_0 := 0$	
5	$B \rightarrow 1$	$v_0 := 1$	

Attributes of two types: left (or synthesized) and right (or inherited)

- left attribute** the semantic function $\sigma_0 = f(\dots)$, whereby attribute σ_0 is assigned a value, in a rule where σ_0 is an attribute of the **left nonterminal** of the rule (bottom up calculation as seen before)
- right attribute** the semantic function $\delta_i = f(\dots)$, $i \geq 1$, whereby attribute δ_i is assigned a value, in a rule where δ_i is an attribute of a symbol in the **rule right part**

Example above: all attributes are left/synthesized (typical of simplest cases)

#	Syntax	Semantic functions		Comment
1	$N \rightarrow D \bullet D$	$v_0 := v_1 + v_2 \times 2^{-l_2}$		Add integer to fractional value divide by weight 2^{l_2}
2	$D \rightarrow DB$	$v_0 := 2 \times v_1 + v_2$	$l_0 := l_1 + 1$	Compute value and length
3	$D \rightarrow B$	$v_0 := v_1$	$l_0 := 1$	
4	$B \rightarrow 0$	$v_0 := 0$		Value initialization
5	$B \rightarrow 1$	$v_0 := 1$		

A more complex example

Segmenting a free text into lines of $\leq W$ chars

The text is a list of one or more words separated by spaces

Requirement: every line must have the maximum possible number of unbroken words

The key attribute is *last*:

it indicates the column number of the last char of each word

Example: “no doubt he calls me an outlaw to catch”, $W=13$; segmented text:

1	2	3	4	5	6	7	8	9	10	11	12	13
n	o		d	o	u	b	t		h	e		
c	a	l	l	s		m	e		a	n		
o	u	t	l	a	w		t	o				
c	a	t	c	h								

attribute *last* is 2 for ‘no’ and 5 for ‘calls’

Attributes and their meaning

<i>length</i>	left	length (in chars) of the current word
<i>last</i>	left	column of the last char of current word
<i>prec</i>	right	column of the last char of previous word (-1 for first word)

fundamental relation between attributes concerning two consecutive words

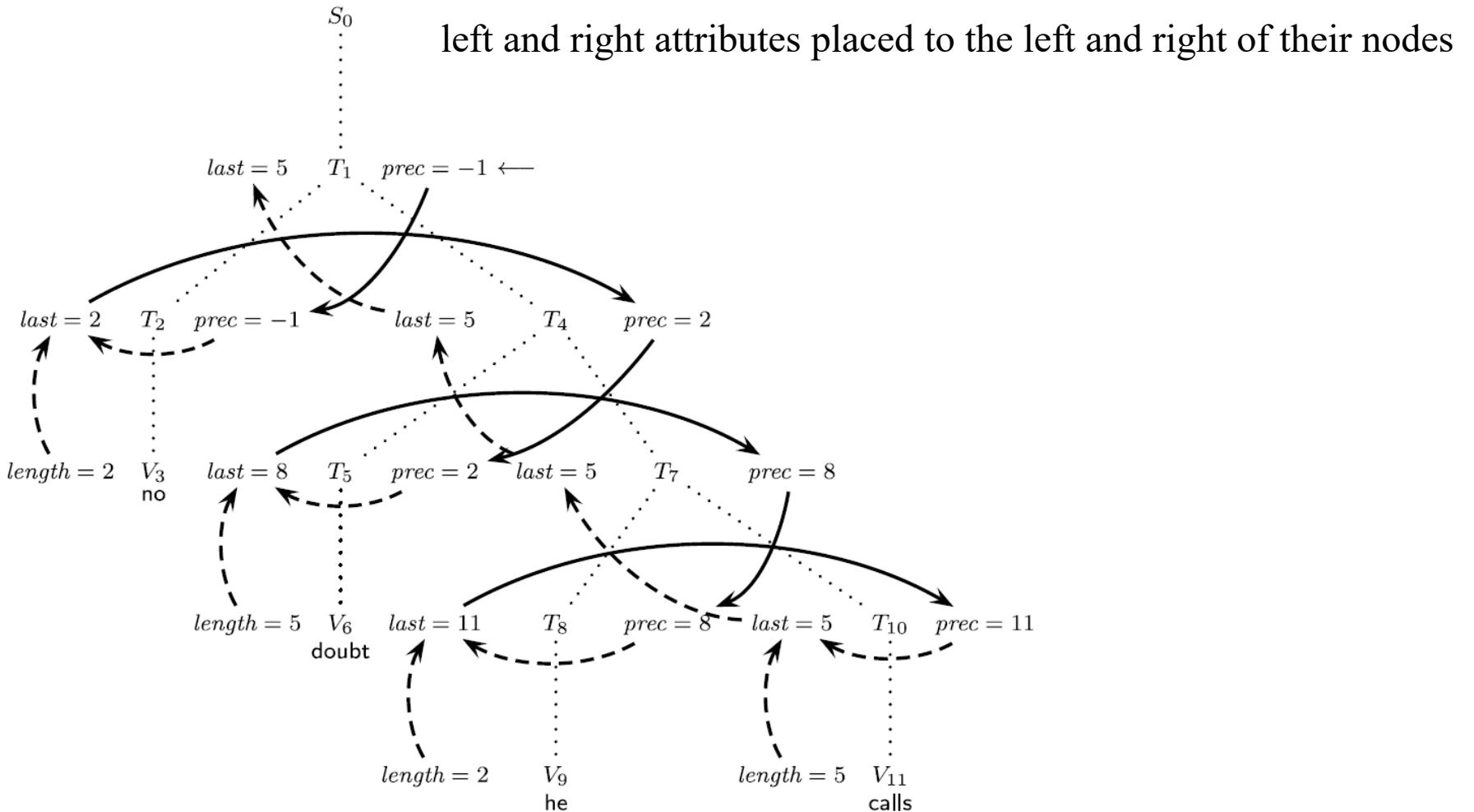
(here we adopt informally notation $\text{last}(w_k)$ for attribute ***last*** of k -th word etc.)

$$\begin{aligned}\text{last}(w_k) &:= \text{prec}(w_k) + 1 + \text{length}(w_k) \\ \text{prec}(w_0) &:= -1\end{aligned}$$

NB: \perp is
the space

#	Syntax	Right attributes	Left attributes
1	$S_0 \rightarrow T_1$	$\text{prec}_1 := -1$	
2	$T_0 \rightarrow T_1 \perp T_2$	$\text{prec}_1 := \text{prec}_0$ $\text{prec}_2 := \text{last}_1$	$\text{last}_0 := \text{last}_2$
3	$T_0 \rightarrow V_1$		$\text{last}_0 := \text{if } (\text{prec}_0 + 1 + \text{length}_1) \leq W$ $\quad \text{then } (\text{prec}_0 + 1 + \text{length}_1)$ $\quad \text{else } \text{length}_1$ $\quad \text{end if}$
4	$V_0 \rightarrow cV_1$		$\text{length}_0 := \text{length}_1 + 1$
5	$V_0 \rightarrow c$		$\text{length}_0 := 1$

Graph for the attribute dependences



the dependence graph has no circuits

Any sequence of attribute computations that complies with the dependences is suitable to evaluate the attributes

Set of *semantic functions* (or rules)

every function is associated with a syntax rule p , called its *syntax support*:

$$p: D_0 \rightarrow D_1 D_2 \dots D_r \quad r \geq 0$$

a semantic function: $\alpha_k := f(\text{attr}(\{D_0, D_1, \dots, D_r\} \setminus \{\alpha_k\}))$, $0 \leq k \leq r$

assigns a value to α_k (attribute of symbol D_k)

function f with arguments the *other* attributes of the same rule p (not α_k - no recursion)

semantic functions must be total and computable

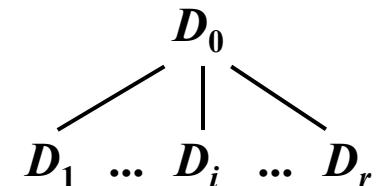
\Rightarrow they must be used as computation rules

semantic functions are written using notations taken from
software specification languages, or
algebra, or
pseudocode

$$p: D_0 \rightarrow D_1 D_2 \dots D_i \dots D_r \quad r \geq 0$$

$\sigma_0 := f(\dots)$ defines a left attribute (of the parent, in the tree portion matching the applied rule)
 $\delta_i := f(\dots)$, with $1 \leq i \leq r$, defines a right attribute (of a child, in the same tree portion)

Attributes of terminal symbols (the tree leaves), often
 are assigned their value by the lexical analysis
 or they may take as value the terminal itself

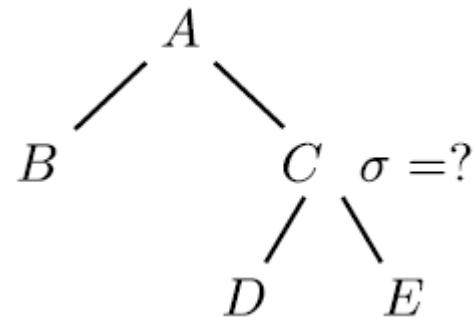


The left attributes of D_0 and the right ones of $D_i, i \geq 1$, are called **internal** for rule p
 the semantic functions for a rule p define **all** and **only** the rule's internal attributes

The right attributes D_0 and the left ones of $D_i, i \geq 1$, are called **external** for rule p
 they are defined by semantic functions applied to other parts of the tree

An attribute cannot be right for one rule and left for another one
 otherwise it would not be uniquely defined, and conflicts may arise

#	Support	Semantic functions
1	$A \rightarrow BC$	$\sigma_C := f_1(attr(A, B))$
2	$C \rightarrow DE$	$\sigma_C := f_2(attr(D, E))$



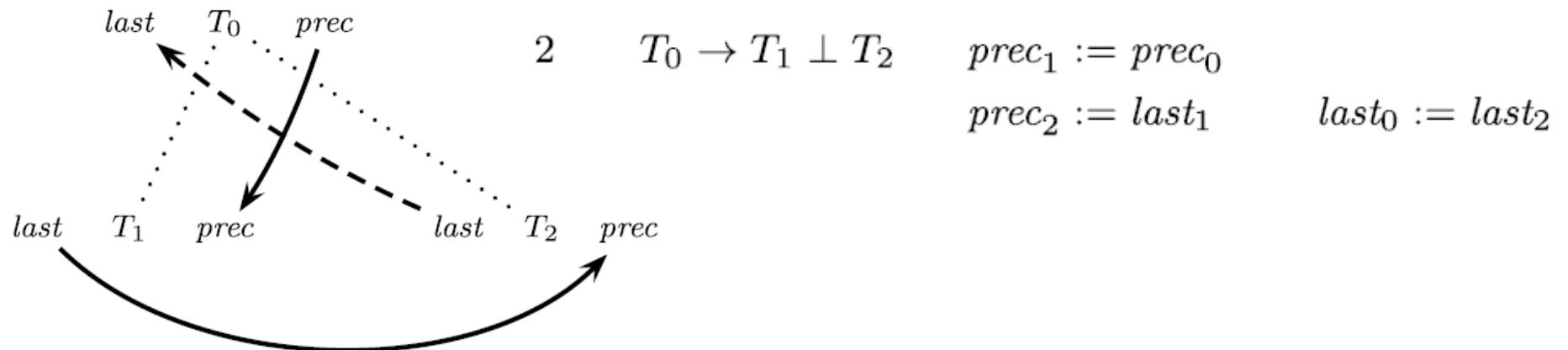
Dependence graph (relation) dep_p for the attributes associated with a syntax rule p

It is a directed graph

- the nodes are the attributes (the arguments and the results of semantic functions)
- there is an arc from every argument to the result
- left (synthesized) attributes placed to the left of tree node, right (inherited) ones to the right

The graph is superimposed to that of the syntax support

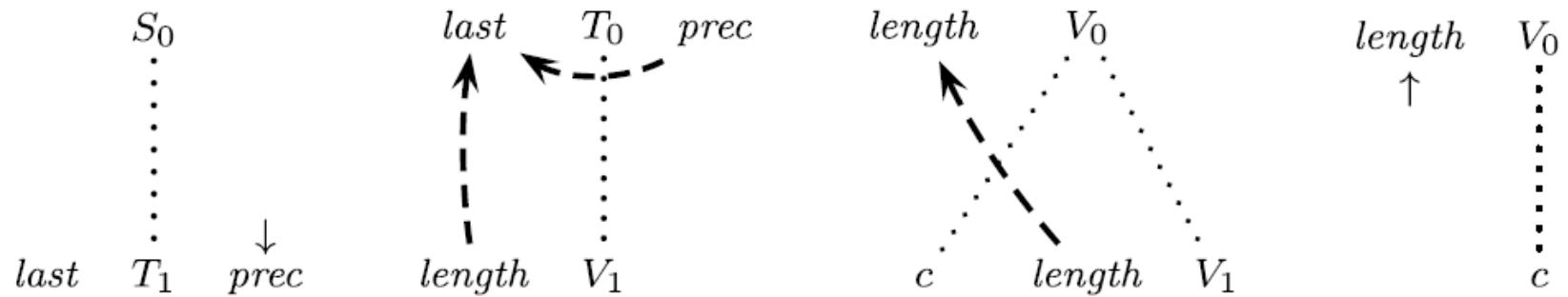
Example for rule 2 (for simplicity the terminal \perp is omitted)



left attribute: upward or leftward arrows

right attribute: down or sideways arrows

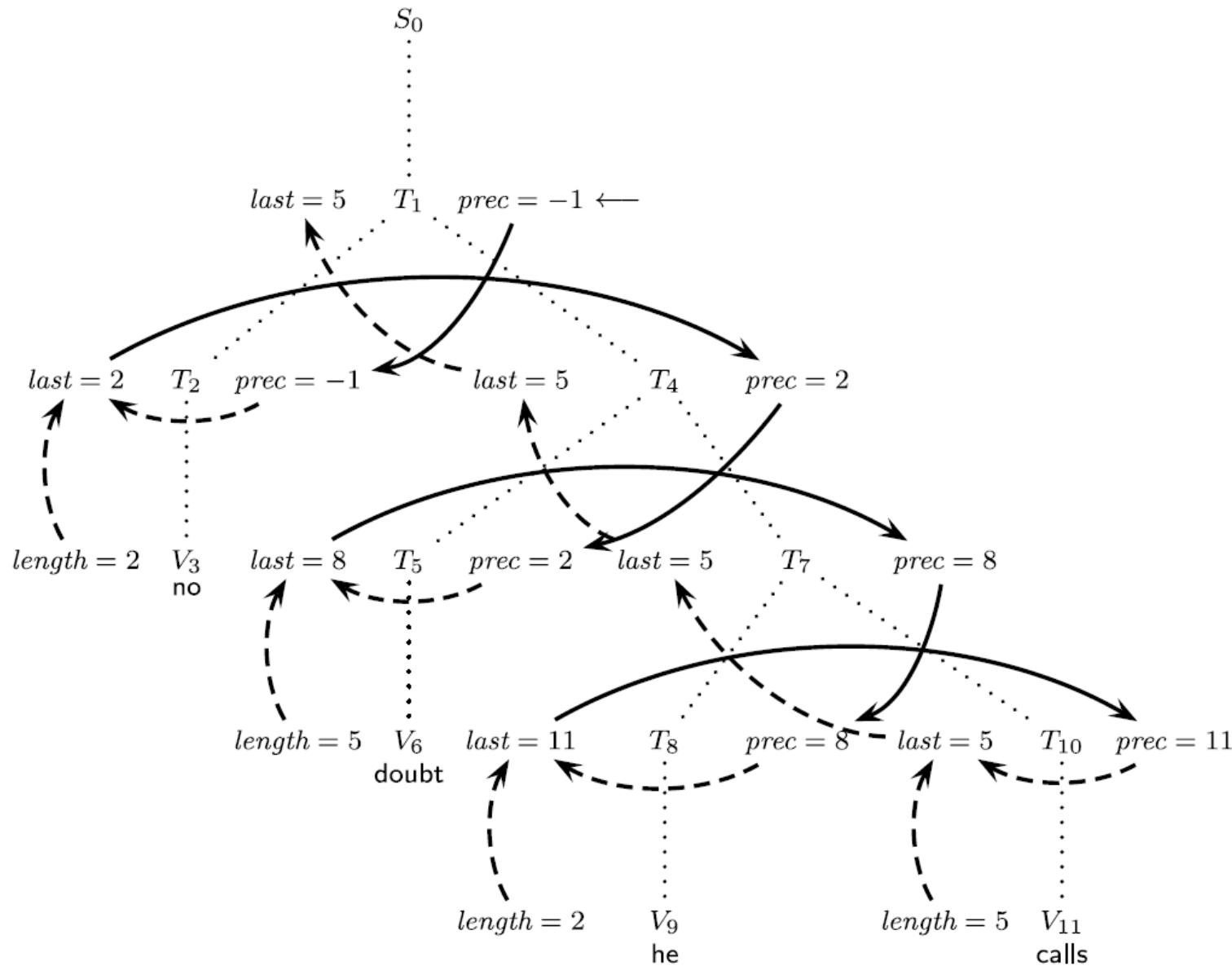
dependence graphs of the remaining productions



#	Syntax	Right attributes	Left attributes
1	$S_0 \rightarrow T_1$	$prec_1 := -1$	
2	$T_0 \rightarrow T_1 \perp T_2$	$prec_1 := prec_0$ $prec_2 := last_1$	$last_0 := last_2$
3	$T_0 \rightarrow V_1$		$last_0 := \text{if } (prec_0 + 1 + length_1) \leq W$ $\quad \text{then } (prec_0 + 1 + length_1)$ $\quad \text{else } length_1$ end if
4	$V_0 \rightarrow cV_1$		$length_0 := length_1 + 1$
5	$V_0 \rightarrow c$		$length_0 := 1$

Dependence graph for attributes of an entire syntax tree

Obtained by combining the graphs of the rules used in the various tree nodes



Existence and unicity of the solution:

If the dependence graph of the tree is acyclic

⇒ there exists a set of attribute values consistent with the dependences

(we consider this a self-evident property)

A grammar is called loop-free

if the dependence graph of every tree is acyclic

We consider only loop free grammars

(later we provide a sufficient condition to ensure that the grammar is loop-free)

For a given tree, to compute the attribute values

one must provide a total order of the attributes

so that every attribute is computed only after those that constitute its arguments

To this purpose one could use the ***Topological Sorting*** algorithm (known in the literature)

However this method is not efficient:

one must apply the sorting algorithm

before computing the attribute values

Another problem: how to determine if the grammar is loop-free
? how can one ensure that the dependence graph of **every possible string** is acyclic ?

The languages of interest are typically infinite \Rightarrow one cannot execute an exhaustive test

The property is decidable but

... the problem of deciding whether a grammar is loop-free
is NP-complete w.r.t. the grammar size

Alternative, more efficient though less general idea: fixed scheduling visit and computation

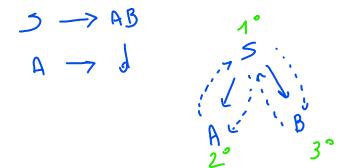
A faster evaluator based on the idea of *predetermining*
a fixed sequence of visit (scheduling)
which is valid for every tree,
according to functional dependence among attributes

in practice: one provides some (general enough) **sufficient conditions** ensuring that

- the grammar is loop-free
- all attribute values can be computed through a *depth-first visit of the tree*

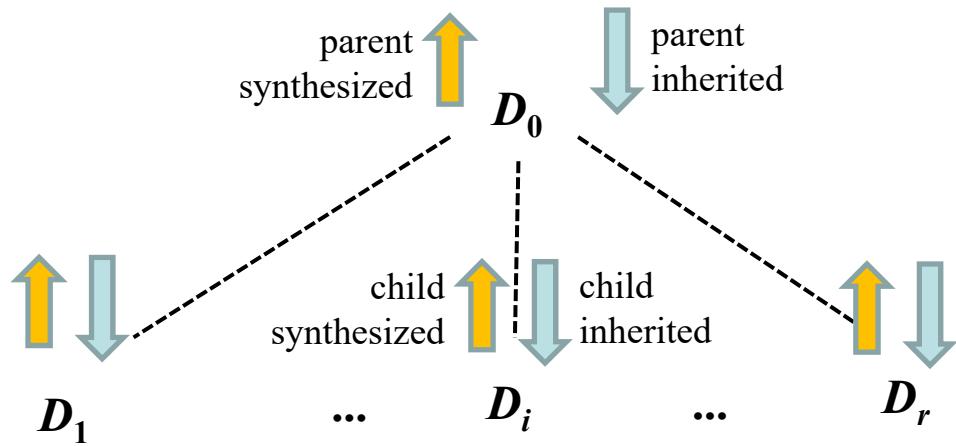
depth-first visit of the tree: implemented through recursive procedures
visit of a subtree \Leftrightarrow procedure call with the tree root as parameter

1. Start from the tree root (grammar axiom)
2. the depth-first visit of a (sub)tree includes (recursively) the depth-first visit of the subtrees rooted in its child nodes (in some specified order, e.g., left-to-right)



For each subtree t_N rooted at a node N :

3. Before visiting t_N compute the **right attributes** of node N and pass them as the **input parameters** of the procedure that implements the visit
procedure calls with input parameter passing are the «descending phase» of the visit
4. At the end of the visit of subtree t_N the **left attributes** of N become available : they are the **output parameters** of the procedure that implements the visit
procedure return and output parameter passing are the «ascending phase» of the visit



NB: order of subtree visits is specific for each rule

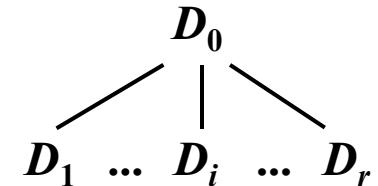
We now provide sufficient conditions on attribute dependences
that permit attribute evaluation by a depth first tree visit

Four Conditions allowing for attribute evaluation through a depth-first visit

they must be checked on the dependence graph dep_p of every syntax rule p

1. The graph dep_p has no circuit

obviously necessary for the grammar to be loop-free



2. In the graph dep_p , there exists no path $\sigma_i \rightarrow \dots \rightarrow \delta_i$, with $i \geq 1$, from a *left attribute* σ_i , to a *right attribute* δ_i , both associated with the same symbol D_i in the right part of p

because δ_i is an input parameter, and σ_i an output parameter, of the recursive call that visits subtree rooted at D_i

3. In the graph dep_p , there exists no arc $\sigma_0 \rightarrow \delta_i$ ($i \geq 1$) from a left attribute of the father D_0 to a right attribute of any child D_i

σ_0 is the output parameter of the procedure call for the parent node D_0

(to introduce the fourth condition we need an additional definition)

w.r.t. syntax rule $p: D_0 \rightarrow D_1 D_2 \dots D_r$, with $r \geq 1$, one defines

binary relation $sibl_p$ called the *sibling graph* among right part *symbols* $\{D_1, D_2, \dots, D_r\}$

In $sibl_p$ there exists an arc $D_i \rightarrow D_j$, with $i \neq j$, if and only if

in the dependence graph dep_p

there is an arc $\alpha_i \rightarrow \beta_j$, with $\alpha_i \in \text{attr}(D_i)$ and $\beta_j \in \text{attr}(D_j)$

The fourth and last condition is:

4. The graph $sibl_p$ has no circuit

necessary to define an order in the (recursive) calls on the child nodes D_1, \dots, D_r

attribute evaluation through a depth-first visit also called *one sweep evaluation*

the conditions 1 – 4 above are collectively called *one sweep (evaluation) condition*

CONSTRUCTION OF THE ONE-SWEEP EVALUATOR

One procedure for each nonterminal; its input parameters are :

- the subtree rooted at the nonterminal
- the right attributes of the subtree root

The procedure

- visits the subtree, computes its attributes and
- returns the left attributes of the root (through the output parameters)

Construction in 3 steps of the semantic evaluation procedure for rule

$$p: D_0 \rightarrow D_1 D_2 \dots D_r, \quad r \geq 1$$

1. Choose a ***Topological Order of Siblings*** D_1, D_2, \dots, D_r , ***TOS***, compatible with the sibling graph $sibl_p$
2. For each symbol D_i , with $1 \leq i \leq r$, choose a ***Topological Order of Right attributes***, ***TOR***, of symbol D_i
3. Choose a ***Topological Order of Left attributes***, ***TOL***, of symbol D_0

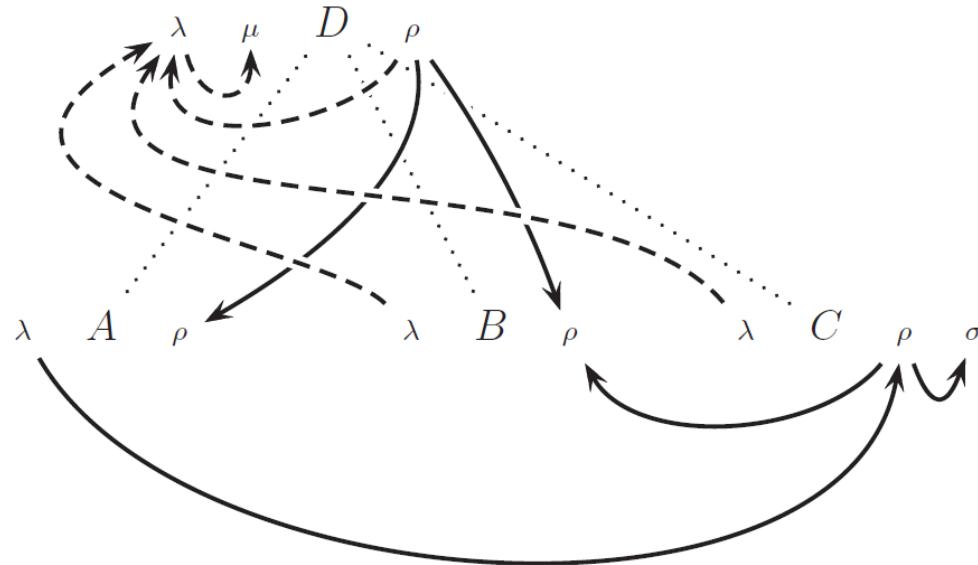
The three orders ***TOS***, ***TOR*** and ***TOL***

determine the instruction sequence in the procedure body (shown in the coming example)

Example of a one-sweep procedure

Given a syntax rule p and the dependence graph dep_p :

$$p: D \rightarrow A B C$$



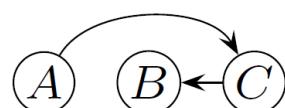
It satisfies the four conditions for attribute evaluation through a depth-first visit

1. dep_p has no circuits

2. dep_p has no path of type $\sigma_i \rightarrow \dots \rightarrow \delta_i$, $i \geq 1$

3. dep_p has no arcs of type $\sigma_0 \rightarrow \delta_i$, with $i \geq 1$

4. $sibl_p$ is acyclic



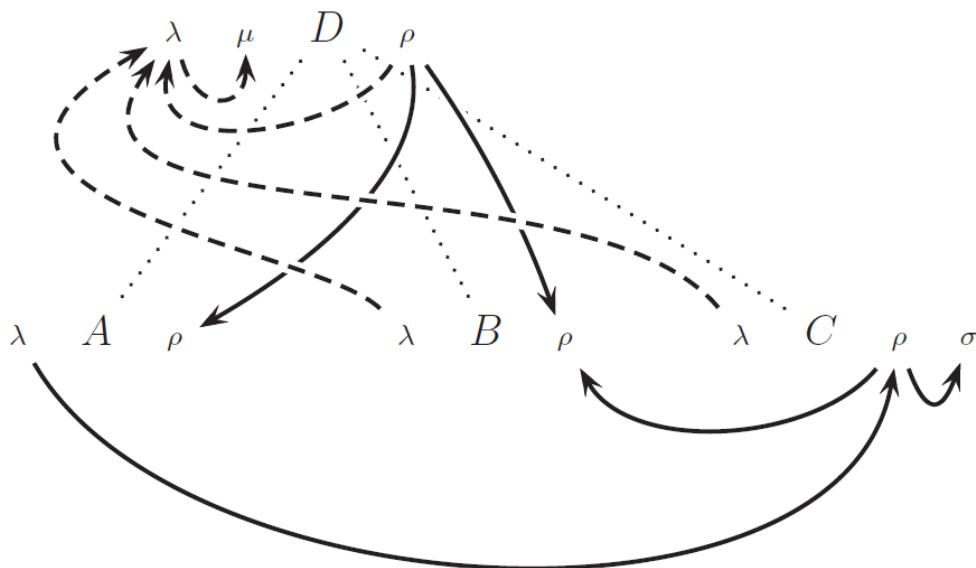
$A \rightarrow C$ derives from dependence $\lambda_A \rightarrow \rho_C$

$C \rightarrow B$ derives from dependence $\rho_C \rightarrow \rho_B$

it is acyclic so we need to find a topological order, in this case it is A,C,B

Here are the possible topological orders

- sibling graph: $TOS = A, C, B$
- right attributes of every child:
 - TOR for $A = \rho$; TOR for $B = \rho$; TOR for $C = \rho, \sigma$; --> since there are dependencies among those two attributes, we need to respect the order
 - left attributes of D : $TOL = \lambda, \mu$ --> same



```

procedure  $D$  (in  $t, \rho_D$ ; out  $\lambda_D, \mu_D$ )
  --  $t$  root of subtree to be decorated
   $\rho_A := f_1(\rho_D)$ 
  -- abstract functions are denoted  $f_1, f_2$ , etc.
   $A(t_A, \rho_A; \lambda_A)$ 
  -- invocation of  $A$  to decorate subtree  $t_A$ 
   $\rho_C := f_2(\lambda_A)$ 
   $\sigma_C := f_3(\rho_C)$ 
   $C(t_C, \rho_C, \sigma_C; \lambda_C)$ 
  -- invocation of  $C$  to decorate subtree  $t_C$ 
   $\rho_B := f_4(\rho_D, \rho_C)$ 
   $B(t_B, \rho_B; \lambda_B)$ 
  -- invocation of  $B$  to decorate subtree  $t_C$ 
   $\lambda_D := f_5(\rho_D, \lambda_B, \lambda_C)$ 
   $\mu_D := f_6(\lambda_D)$ 
end procedure

```

24 / 35

In principle this code can be generated directly out of the semantic rules

Now we can introduce and motivate an «attribute grammar design hint»:

when an initial (\Rightarrow inherited) attribute is needed in the root S of the tree (like an initialization)
then one adds a «new spurious» axiom S' and a rule $S' \rightarrow S$

... this occurs in the previous example of text segmentation
(slide 8, attribute *prec* in rule $S \rightarrow T$)

Combined Syntax and Semantic Analysis

Syntax and semantic analysis can be integrated into the parser

Simple and efficient method, suitable for simple translations

Various cases, depending on the nature of the source language

- regular source language: lexical analysis with attribute evaluation
 - can be performed with tools such as *flex* or *lex*
- ***LL(k)*** syntax: recursive top-down parser with attributes
 - can be implemented manually with left (synthesized) attributes only
- ***LR(k)*** syntax: shift-reduce parser with attributes
 - can be performed with tools such as *bison* or *yacc*
(NB: functional dependence among right attributes is strongly limited)

[Bison implements an attribute grammar](#)

Attributed recursive descent translator

Several hypotheses must be satisfied

- syntax suitable for deterministic top-down analysis (*LL*)
- attribute grammar suitable for one-sweep evaluation (depth-first visit)
- *further* conditions on functional dependence among attributes ... that we see now

Top-down analysis builds the subtrees from left to right

If combined with attribute evaluation then ...

...attribute dependences must permit a visit of subtrees
in the sequence from left to right: **1, 2, ..., r – 1, r**

Therefore: Condition **L (left-to-right)** for syntax/semantic recursive descent analysis

1. Conditions allowing for *one sweep evaluation* through depth first visit, plus
2. The sibling graph $sibl_p$ for rule $D_0 \rightarrow D_1 \dots D_r$ allows one to choose as **TOS**
the “natural” sequence D_1, D_2, \dots, D_r
i.e., $sibl_p$ does not include any arc $D_j \rightarrow D_i$ with $j > i$:
no attribute of D_i depends on an attribute of D_j with D_j placed to the right of D_i

if a grammar is **LL(k)** and satisfies the **L** condition \Rightarrow

\Rightarrow build a deterministic recursive descent parser that also evaluates the attributes

Example of a recursive descent syntax-semantic analyzer

Computes the numeric value of a binary string
encoding a value less than 1

Language: $L = \bullet(0 \mid 1)^+$

Translation (ex.): $\tau(\bullet01) = 0,25$

Grammar

Syntax	Left attributes	Right attributes
$N_0 \rightarrow \bullet D_1$	$v_0 := v_1$	$l_1 := 1$
$D_0 \rightarrow B_1 D_2$	$v_0 := v_1 + v_2$	$l_1 := l_0 \quad l_2 := l_0 + 1$
$D_0 \rightarrow B_1$	$v_0 := v_1$	$l_1 := l_0$
$B_0 \rightarrow 0$	$v_0 := 0$	
$B_0 \rightarrow 1$	$v_0 := 2^{-l_0}$	

Attributes

Attribute	Meaning	Domain	Type	Assoc. symbols
v	Value	Real	Left	N, D, B
l	Length	Integer	Right	D, B

The value of each bit is weighted by a power of 2 with negative exponent = distance from the fractional point

The syntax is deterministic $LL(2)$: lookahead=2 needed for nonterminal D

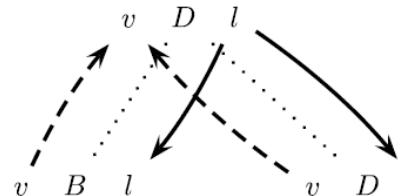
Check the L condition for every syntax rule

$$N \rightarrow \bullet D: \quad v_0 := v_1 \quad l_1 := 1$$

the dependence graph dep has the only arc $v_1 \rightarrow v_0$,
hence the L condition is satisfied, because

1. the graph has no circuit
2. there is no path from a left attribute v to a right attribute l of the same child
3. there is no arc from a left attribute v of the father to a right attribute l of a child
4. the sibling graph $sibl$ has no arc (no sibling, only a single child node)

$$D \rightarrow B D: \quad v_0 := v_1 + v_2 \quad l_1 := l_0 \quad l_2 := l_0 + 1$$



the dependence graph

- has no circuit
- there is no path from a left attribute v to a right attribute l of the same child
- no arc from a left attr. (v) of the father to a right attr. l of a child
- the sibling graph $sibl$ has no arc -> so we can choose any topological order

$$D \rightarrow B: \quad v_0 := v_1, \quad \text{same as above}$$

$$B \rightarrow 0: \quad v_0 := 0, \quad \text{dependence graph has no arc}$$

$$B \rightarrow 1: \quad v_0 := 2^{-l_0} \quad dep \text{ graph has a unique arc } l_0 \rightarrow v_0 \text{ and satisfies the } L \text{ condition}$$

Integrated syntax - semantic procedure

- in parameters: right attributes of the father
- out parameters: left attributes of the father
- variables $cc1$ and $cc2$: the current terminal symbol and the next one (syntax is $LL(2)$)
- some local variables to pass the attribute values to other internal procedures
- «*read*» function updates $cc1$ and $cc2$ (NB: syntax is $LL(2)$ but not $LL(1)$)

procedure N (**in** \emptyset ; **out** v_0)

if $cc1 = \bullet$ **then**

read

else

error

end if

$l_1 := 1$ -- initialize a local var. with right attribute of D

$D(l_1, v_0)$ -- call D to construct a subtree and compute v_0

end procedure

Syntax	Left attributes	Right attributes
$N_0 \rightarrow \bullet D_1$	$v_0 := v_1$	$l_1 := 1$

Integrated syntax - semantic procedure

procedure B (**in** l_0 ; **out** v_0)

case $cc1$ **of**

‘0’: $v_0 := 0$ - - case of rule $B \rightarrow 0$

‘1’: $v_0 := 2^{-l_0}$ - - case of rule $B \rightarrow 1$

otherwise *error*

end case ; *read*

end procedure

Grammar

Syntax	Left attributes	Right attributes
$N_0 \rightarrow \bullet D_1$	$v_0 := v_1$	$l_1 := 1$
$D_0 \rightarrow B_1 D_2$	$v_0 := v_1 + v_2$	$l_1 := l_0 \quad l_2 := l_0 + 1$
$D_0 \rightarrow B_1$	$v_0 := v_1$	$l_1 := l_0$
$B_0 \rightarrow 0$	$v_0 := 0$	
$B_0 \rightarrow 1$	$v_0 := 2^{-l_0}$	

Integrated syntax - semantic procedure

```

procedure  $D$  (in  $l_0$ ; out  $v_0$ )
  case  $cc2$  of
    ‘0’, ‘1’ :   begin           -- case of rule  $D \rightarrow BD$ 
       $B(l_0, v_1)$ 
       $l_2 := l_0 + 1$ 
       $D(l_2, v_2)$ 
       $v_0 := v_1 + v_2$ 
    end
    ‘ $\vdash$ ’ :     begin           -- case of rule  $D \rightarrow B$ 
       $B(l_0, v_1)$ 
       $v_0 := v_1$ 
    end
    otherwise error
  end case
end procedure

```

Grammar

Syntax	Left attributes	Right attributes
$N_0 \rightarrow \bullet D_1$	$v_0 := v_1$	$l_1 := 1$
$D_0 \rightarrow B_1 D_2$	$v_0 := v_1 + v_2$	$l_1 := l_0 \quad l_2 := l_0 + 1$
$D_0 \rightarrow B_1$	$v_0 := v_1$	$l_1 := l_0$

Example: Code generation for conditional control structures

if-then-else construct is converted to a combination of (conditional) jump instructions

For every generated instruction the translator needs a new label for the instruction targeted by the jump; every label must differ from previous ones used for other instructions

function *fresh* returns, at each invocation, a new integer, to be assigned to variable *n*, a right attribute of the nonterminal representing the instruction

computed translation assigned to the *tr* attribute

concatenation operator (\bullet) to combine the translation of the various fragments

Labels have the form: e397, f397, i23, ...

Example translation (assuming that the current call of *fresh* returns 7)

if ($a > b$)	$tr(a > b)$
then	jump-if-false rc, e_7
$a := a - 1$	$tr(a := a - 1)$ jump f_7
else	$e_7:$
$a := b$	$tr(a := b)$
end if	$f_7:$
...	... -- rest of the program

Grammar of the *if-then-else* conditional instruction

Syntax	Semantic functions
$F \rightarrow I$	$n_1 := fresh$ \leftarrow NB: n_0 has the value of n_1 (<i>fresh</i>) above
$I \rightarrow \text{if } (cond)$ then L_1 else L_2 end if	$tr_0 := tr_{cond} \bullet$ jump-if-false $rc_{cond}, e_{n_0} \bullet$ $tr_{L_1} \bullet$ jump $f_{n_0} \bullet$ $e_{n_0} : \bullet$ $tr_{L_2} \bullet$ $f_{n_0} :$ rc_{cond} is an attribute of nonterm. <i>cond</i>

translation of *cond* (tr_{cond}), L_1 (tr_{L_1}), and L_2 (tr_{L_2}) specified by other rules (not reported)

Proposed exercise: define similarly an attribute grammar for the translation of the iterative **while** instruction so to obtain the result here below

while ($a > b$)	i_8: $tr(a > b)$
do	jump-if-false rc, f_8
$a := a - 1$	$tr(a := a - 1)$ jump i_8
end while	f_8:
...	... - - rest of the program

F->W

W-> 'while'
cond
'do'
L
'end while'

n1=fresh
Ln0:
tr(cond)
jump_if_false rcond, f-n0
trL
jump i_n0
f_no:

