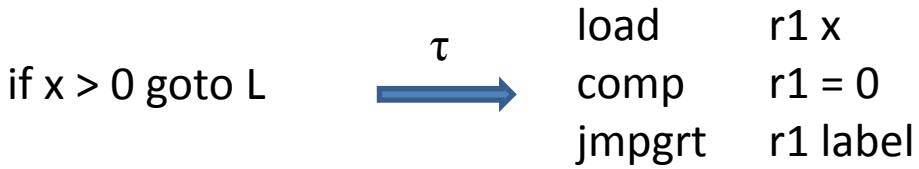
# Purely Syntactic Translations

*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

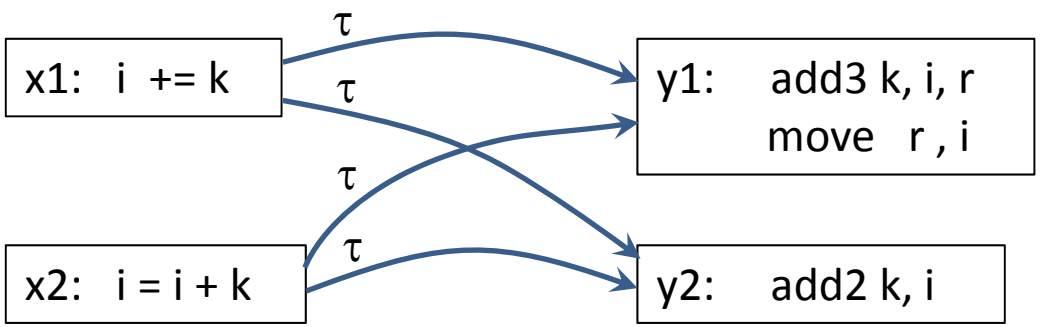
Translation in an abstract setting: a map Source Language ⇔ Target Language

Ex.: transl. from C to Assembler

it is a **relation**

$\boldsymbol{\tau} = \{(x1, y1), (x1, y2), (x2, y1), (x2, y2)\}$

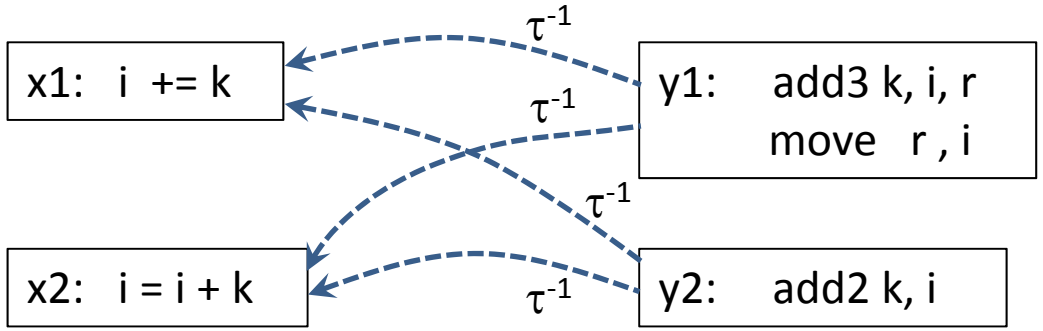
The image of $x1$ is $\tau(x1) = \{y1, y2\}$




The translation is ***many-valued***, or ***not single valued***, or ***ambiguous***

The inverse translation, $\tau^{-1}$, is not single valued, that is, $\tau$ is not injective

Example: $x1 \in \tau^{-1}(y1) = \{x1, x2\}$



# 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 $\tau$ and $\tau^{-1}$ are single valued: ex., in cryptography, encryption and decryption of a text

# Syntax translation schemes: introductory example

Image string obtained through

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)

$$L_1 = \{\, a^n b^m \mid n \geq m > 0 \,\} \qquad \tau(a^n b^m) = c^{n-m}\, d$$

Source grammar $G_1$

$S \to a\ S$

$S \to A$

$A \to a\ A\ b$

$A \to ab$

Target grammar $G_2$

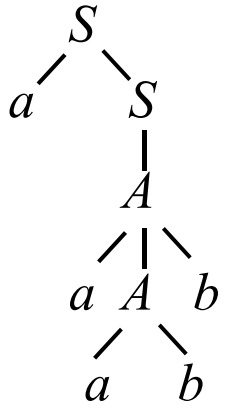
$S \to c\ S$

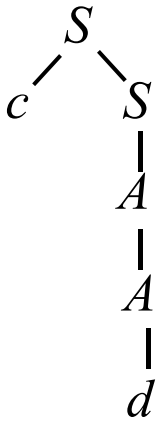
$S \to A$

$A \to\ A$

$A \to d$

Translation: source tree $\Rightarrow$ target tree




$\tau(a^3 b^2) = cd$

is $\tau^{-1}$ single valued?

# 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***

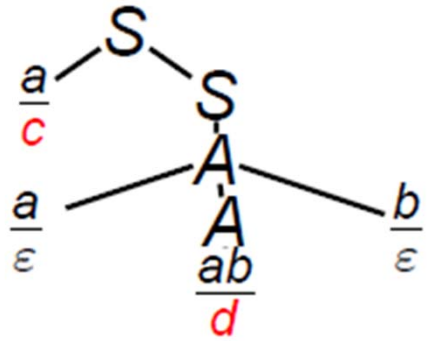
| **source gramm. $G_1$** | **target gramm. $G_2$** | **OK?** |
|---|---|---|
| $A \rightarrow aBcBD$ | $A \rightarrow xBByDy$ | *YES* |
| $A \rightarrow aBcBD$ | $A \rightarrow xBBy$ | ***No: D is missing*** |
| $A \rightarrow aBcBD$ | $A \rightarrow xBDBy$ | ***No: order of NT changed*** |

# 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. $G_t$** | **source gramm. $G_1$** | **target gramm. $G_2$** |
|---|---|---|
| $S \to \frac{a}{c} S$ | $S \to aS$ | $S \to cS$ |
| $S \to A$ | $S \to A$ | $S \to A$ |
| $A \to \frac{a}{\varepsilon} A \frac{b}{\varepsilon}$ | $A \to aAb$ | $A \to A$ |
| $A \to \frac{ab}{d}$ | $A \to ab$ | $A \to d$ |

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

**parethesized functional**: *add* (*i*1, *mult* (*i*2, *i*3) )

**infix**: $i1 + (i2 \times i3)$

**polish**: **prefix**: *add i*1,*mult i*2, *i*3

**postfix**: *i*1, *i*2, *i*3 *mult add*

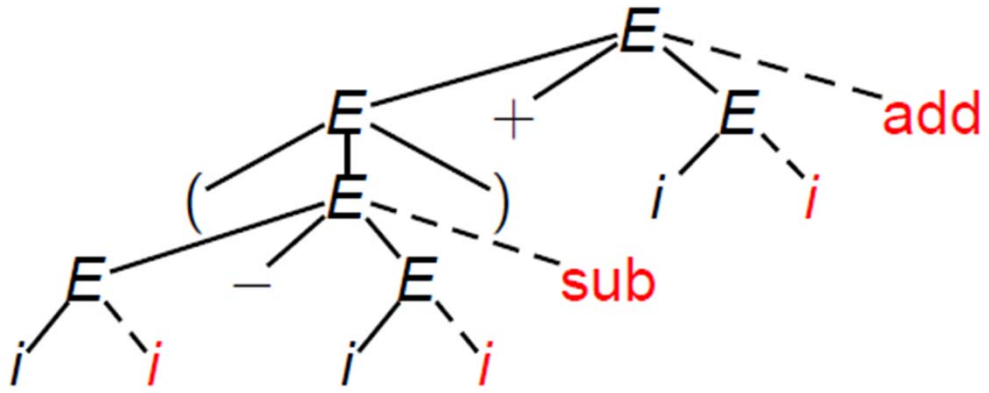
Polish expressions are concise

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. $G_t$** | **another (equivalent alternative) rappresentation** |
|---|---|
| $E \rightarrow E \ \underset{\varepsilon}{+} E \ \underset{add}{\varepsilon}$ | $E \rightarrow E + E \ \{ \ add \ \}$ |
| $E \rightarrow E \ \underset{\varepsilon}{-} E \ \underset{sub}{\varepsilon}$ | $E \rightarrow E - E \ \{ \ sub \ \}$ |
| $E \rightarrow \underset{\varepsilon}{(} E \underset{\varepsilon}{)}$ | $E \rightarrow ( \ E \ )$ |
| $E \rightarrow \underset{i}{i}$ | $E \rightarrow i \ \ \{ \ i \ \}$ |



# 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: $L = \{ a^n \mid n \geq 1 \}$

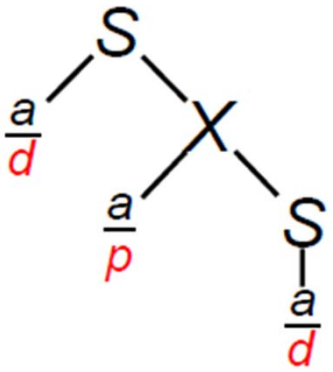
$$\tau(a^n) = \begin{cases} (dp)^{n/2} & n \ \ even \\ (dp)^{(n-1)/2}\, d & n\ odd \end{cases}$$

The natural grammar for $L$: $\boldsymbol{S \rightarrow aS \mid a}$ is unfit; no distinction of even and odd $\boldsymbol{a}$

Modify $\boldsymbol{G}$: $\boldsymbol{S \rightarrow aX \mid a,\ X \rightarrow aS \mid a}$

From this $\boldsymbol{G}$ the scheme $\boldsymbol{G_t}$ that defines $\boldsymbol{\tau}$:

$$S \rightarrow \frac{a}{d} X \mid \frac{a}{d}, \qquad X \rightarrow \frac{a}{p} S \mid \frac{a}{p}$$

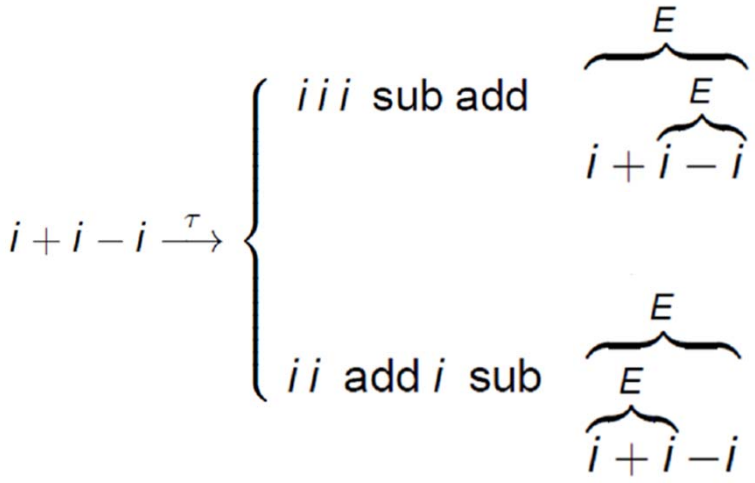


# Ambiguous (i.e., many-valued) Translation

A translation defined by a scheme $\langle G_1, G_2 \rangle$ is ambiguous ***only if*** $\boldsymbol{G_1}$ is so

Example: infix to postfix translation with $\boldsymbol{G_1}$ having bilateral recursion

| $G_1$ | $G_2$ |
|---|---|
| $E \to E + E$ | $E \to EE$ add |
| $E \to E - E$ | $E \to EE$ sub |
| $E \to i$ | $E \to i$ |



# Ambiguous Translation: another example

When $G_1$ has duplicated rules

| $G_1$ | $G_2$ |
|---|---|
| $S \to aS$ | $S \to bS$ |
| $S \to aS$ | $S \to cS$ |
| $S \to a$ | $S \to d$ |

$$aa \xrightarrow{\tau} \{bd, \quad cd\}$$

NB: the translation grammar $G_t$ is **not** ambiguous:

$$G_t: \quad S \to \frac{a}{b}S \mid \frac{a}{c}S \mid \frac{a}{d}$$

# Computing the translation

Analogy with syntax analysis

| | | |
|---|---|---|
| grammar | $\Leftrightarrow$ | push down automaton / parser |
| translation scheme/grammar | $\Leftrightarrow$ | push down automaton / parser with ***writing actions*** |

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

$$L = \{a^n b^m \mid n \geq m \geq 1\} \qquad \tau(a^n b^m) = c^{n-m}\, d$$

| Rule | Move |
|---|---|
| **1.** $S \to \frac{a}{c} S$ | **if** *cc* = *a* ∧ *top*=*S* **then** write(c); pop; push(S); cc:=next **end if** |
| **2.** $S \to A$ | **if** *top* = *S* **then** pop; push(A) **end if** |
| **3.** $A \to \frac{a}{\varepsilon} A \frac{b}{\varepsilon}$ | **if** *cc* = *a* ∧ *top* = *A* **then** pop; push($\frac{b}{\varepsilon}$ A); cc:=next **end if** |
| **4.** $A \to \frac{ab}{d}$ | **if** *cc* = *a* ∧ *top* = *A* **then** pop; push($\frac{b}{d}$); cc:=next **end if** |
| **5. --** | **if** *cc* = *b* ∧ *top* = $\frac{b}{x}$ **then** pop; write(x); cc:=next **end if** |
| **6. --** | **if** *cc* = ⊣ ∧ *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 : $\boldsymbol{L = (a \mid b)^*}$ $\qquad$ $\boldsymbol{\tau(u) = u^R}$

$G_1$: $\quad \boldsymbol{S \to a\,S}$ $\quad$ $\boldsymbol{S \to b\,S}$ $\quad$ $\boldsymbol{S \to \varepsilon}$

$G_2$: $\quad \boldsymbol{S \to S\,a}$ $\quad$ $\boldsymbol{S \to S\,b}$ $\quad$ $\boldsymbol{S \to \varepsilon}$

$$\boldsymbol{G_t}\text{: } S \to \frac{a}{\varepsilon} S \frac{\varepsilon}{a} \mid \frac{b}{\varepsilon} S \frac{\varepsilon}{b} \mid \varepsilon$$

```
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_withTranlsation
{
    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***

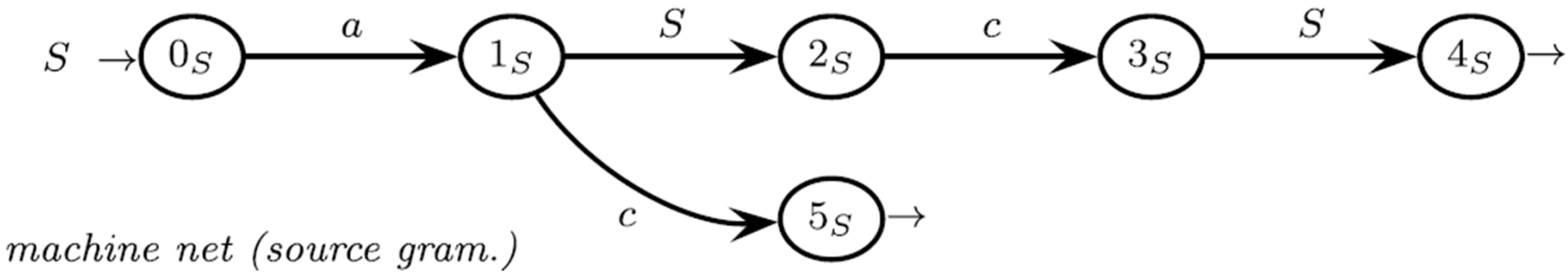
To ensure that, the transl. grammar $G_t$ must be normalized, in the ***postfix normal form***

# 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:

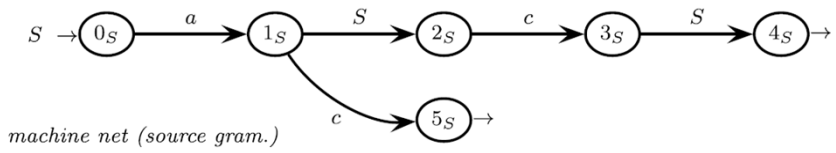
$$G_t\colon\ S \to \underset{\boldsymbol{b}}{\boldsymbol{a}}\, S \,\underset{\boldsymbol{e}}{\boldsymbol{c}}\, S \ \Big|\ \underset{\varepsilon}{a}\ \underset{\varepsilon}{c} \qquad \tau(ac) = \varepsilon \qquad \tau(\boldsymbol{a}\ ac\ \boldsymbol{c}\ ac) = \boldsymbol{b}\ \boldsymbol{e}$$

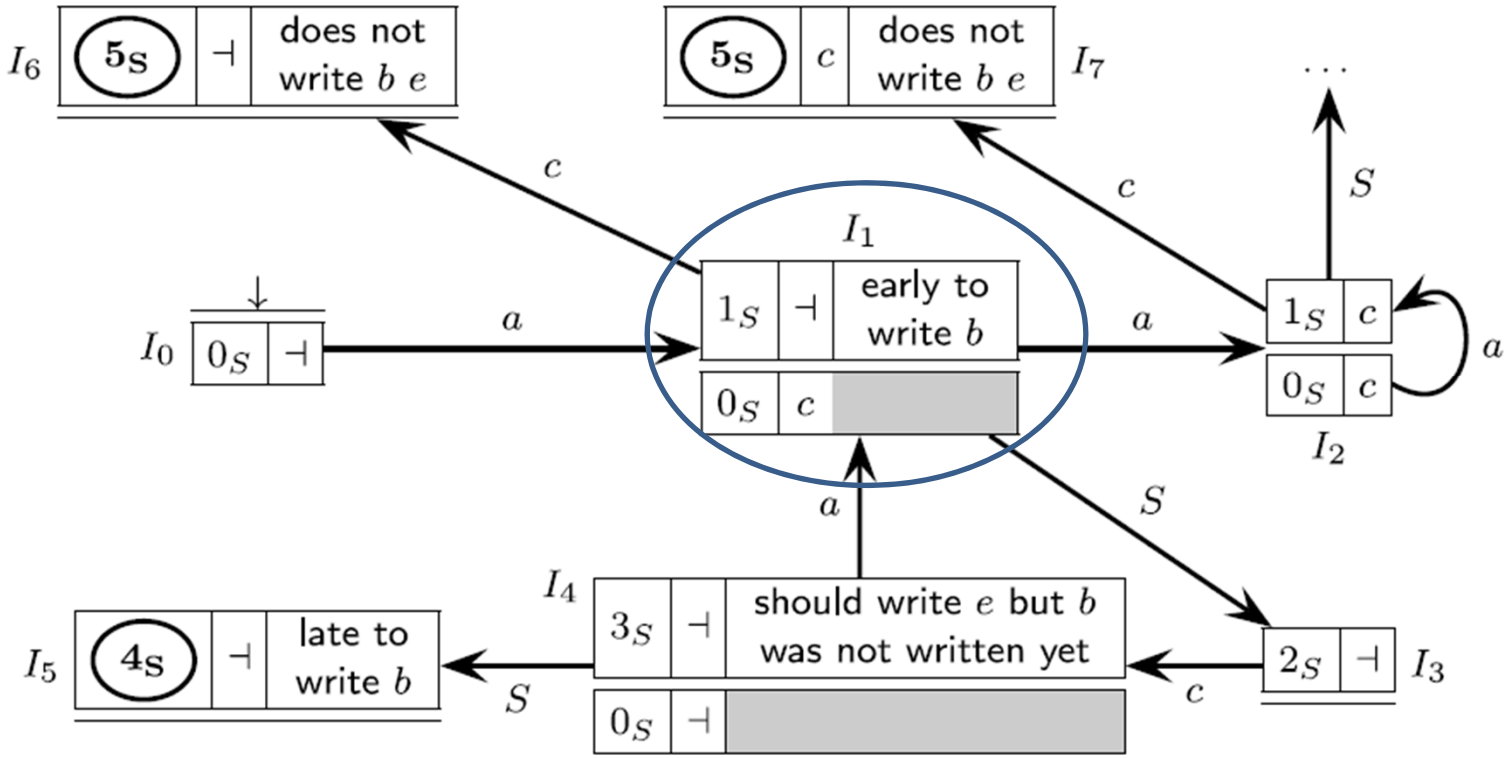
Machine for the source grammar, which is *ELR*(1)



# Pilot with writing actions: a first try that does not work

$G_t$: $S \to \frac{a}{b} S \frac{c}{e} S \mid \frac{a}{\varepsilon} \frac{c}{\varepsilon}$




$\tau(ac) = \varepsilon$ while $\tau(\mathbf{a}\ ac\ \mathbf{c}\ ac) = \mathbf{b}\ \mathbf{e}$

if in $I_1$ it does not write $b$ then $\tau(ac) = \varepsilon$ (correct), but $\tau(\mathbf{a}\ ac\ \mathbf{c}\ ac) = \mathbf{e}$ (incorrect)
if in $I_1$ it does write $b$ then $\tau(\mathbf{a}\ ac\ \mathbf{c}\ ac) = \mathbf{b}\ \mathbf{e}$ (correct), but $\tau(ac) = \mathbf{b}$ (incorrect)

# **Postfix** form of the translation grammar $G_t = (G_1, G_2)$

Every rule of the target grammar $G_2$ has the form ($\Delta$ is the target terminal alphabet):

$$A \to \underbrace{\gamma}_{\in V^*} \quad \underbrace{w}_{\in \Delta^*}$$

that is: first the nonterminals, then the terminals

The gramm. of previous example $G_t$: $S \to \frac{a}{b} S \frac{c}{e} S \mid \frac{a}{\varepsilon} \frac{c}{\varepsilon}$ **is not** in the postfix form

grammar normalization:

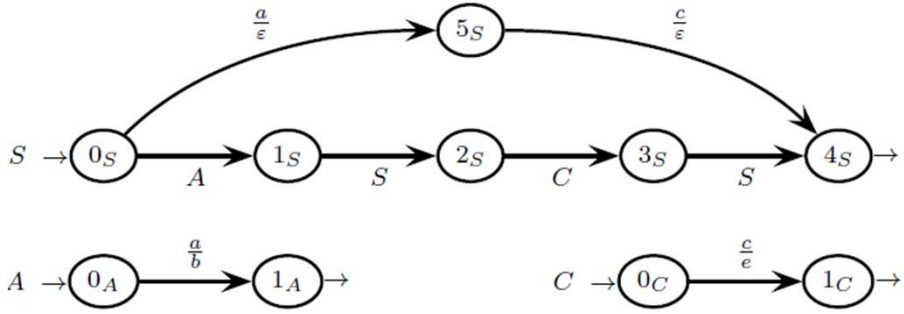
introduce additional nonterminals in place of terminal parts that are not suffix

$$G_\tau: \quad S \to ASCS \mid ac \qquad A \to \frac{a}{b} \qquad C \to \frac{c}{e}$$

$$G_{2\text{postfix}}: \quad S \to ASCS \mid \varepsilon \qquad A \to b \qquad C \to e$$

The new pilot emits the tranlation 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

$$\begin{cases} a^{2n} \xrightarrow{\tau} b^{2n} & : \quad n \geq 0 \\ a^{2n+1} \xrightarrow{\tau} c^{2n+1} & : \quad n \geq 0 \end{cases}$$

translation grammar $G_t$

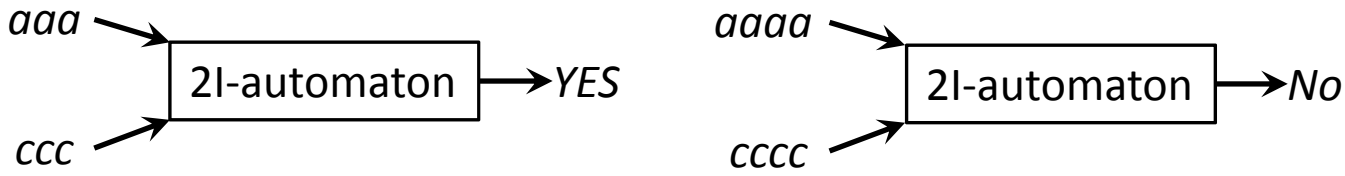
$$\begin{cases} A_0 \rightarrow \frac{a}{c}A_1 \mid \frac{a}{c} \mid \frac{a}{b}A_3 \mid \varepsilon \\ A_1 \rightarrow \frac{a}{c}A_2 \mid \varepsilon \\ A_2 \rightarrow \frac{a}{c}A_1 \\ A_3 \rightarrow \frac{a}{b}A_4 \\ A_4 \rightarrow \frac{a}{b}A_3 \mid \varepsilon \end{cases}$$

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 $\tau$ , i.e., a set of pairs of strings $\in \Sigma^* \times \Delta^*$ ($\Delta$ is the target alph.)




Transition labels are pairs **s** written as

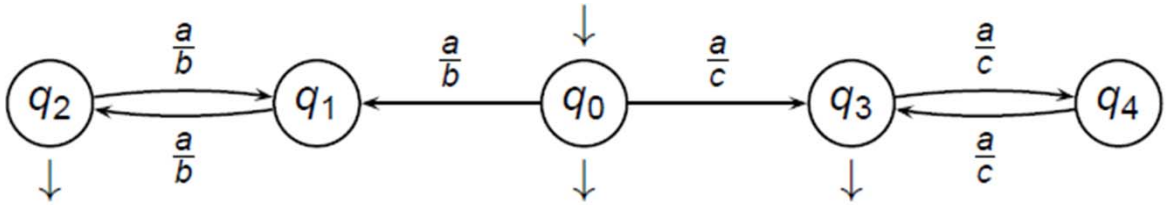
$$\frac{a}{b}, \text{ where } a \in \Sigma \cup \varepsilon, \ b \in \Delta \cup \varepsilon$$

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

$$\begin{cases} a^{2n} \xrightarrow{\tau} b^{2n} & : \quad n \geq 0 \\ a^{2n+1} \xrightarrow{\tau} c^{2n+1} & : \quad n \geq 0 \end{cases}$$




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 $\boldsymbol{\tau}$ :

1. Right- (or left-) linear translation grammar
2. 2I finite automaton
3. Regular translation expression
4. Regular lang. $\boldsymbol{R_t}$ of alphabet $\boldsymbol{\Gamma}$ and two transliterations $\boldsymbol{h_1 : \Gamma \to \Sigma \cup \{\varepsilon\}}$ (for the source), $\boldsymbol{h_2 : \Gamma \to \Delta \cup \{\varepsilon\}}$ (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

$R_t = (pp)^* \cup d(dd)^*$ $h_1(p) = h_1(d) = a$ $h_2(p) = b$ $h_2(d) = c$

$h_1{=}aaaa$ $\overbrace{pppp}$ $h_2{=}bbbb$ $\quad$ $h_1{=}aaa$ $\overbrace{ddd}$ $h_2{=}ccc$

# Non regular translation of regular languages!

Even if **both** $L_1$ and $L_2$ are regular, the translation is not necessarily regular

Ex.: $L_1 = (a \mid b)^*$ $L_2 = (a \mid b)^*$ $\tau(x) = x^R$

A 2I finite state automaton cannot check if the 2nd tape contains the reflection of the first one

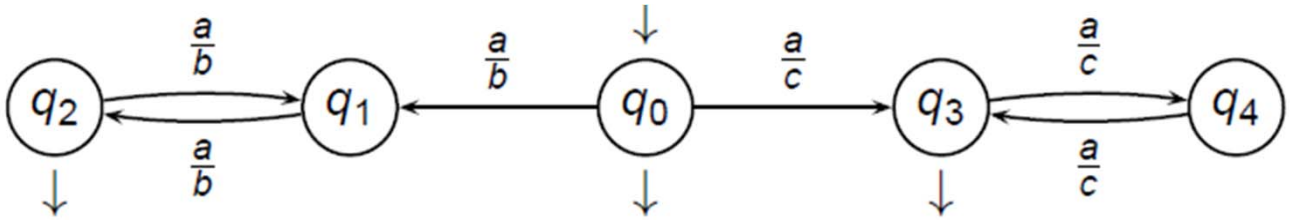
An unbounded stack memory is necessary

# Finite transducer or IO-automaton

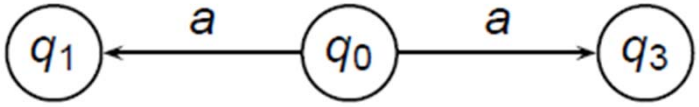
- the second tape is viewed as an output
- the machine ***computes*** the translation as a function of the source string: $\boldsymbol{y = \tau\,(x)}$
- Several applications:
  - lexeme (token) recognition in the lexical analysis (see lessons on Flex)
  - transformation of simple texts, or signal sequences (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 $\boldsymbol{\Delta}$)

# 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:

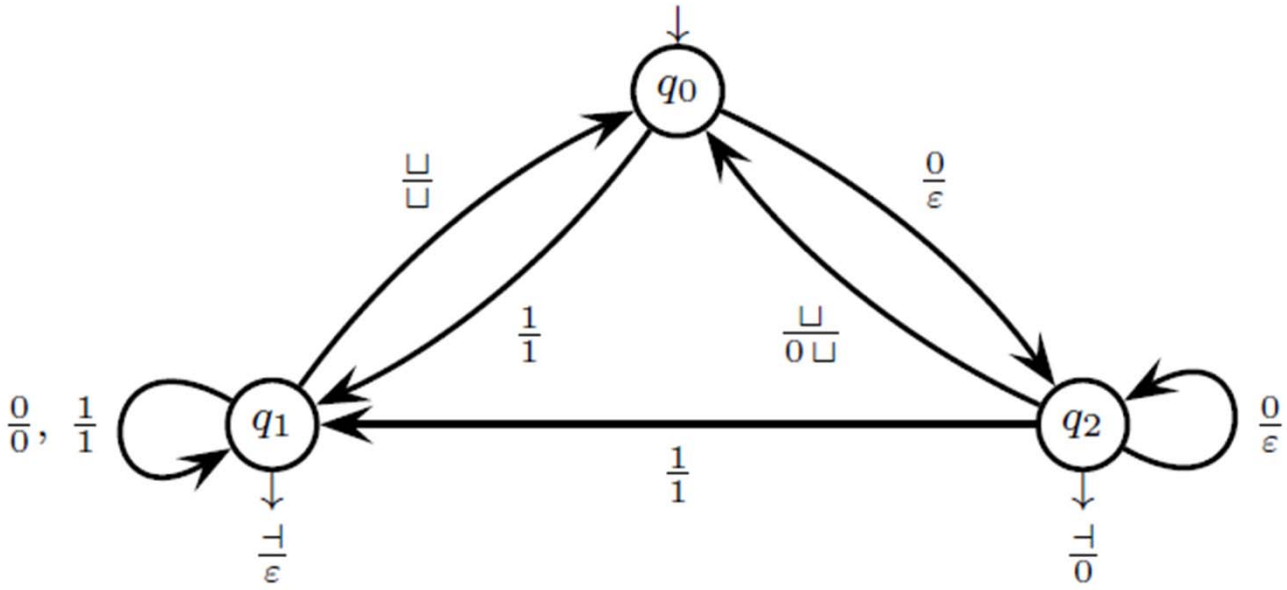
***sequential transducer***

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 « $\dashv/\boldsymbol{s}$ » 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 $q_1$