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.:

		load	r1 x
if x > 0 goto L	$\xrightarrow{\tau}$	comp	r1 = 0
		jmpgrt	r1 label

? 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. $LL(1)$
    2. $LR(1)$
4. Special cases: finite tranducers, regular translation expressions

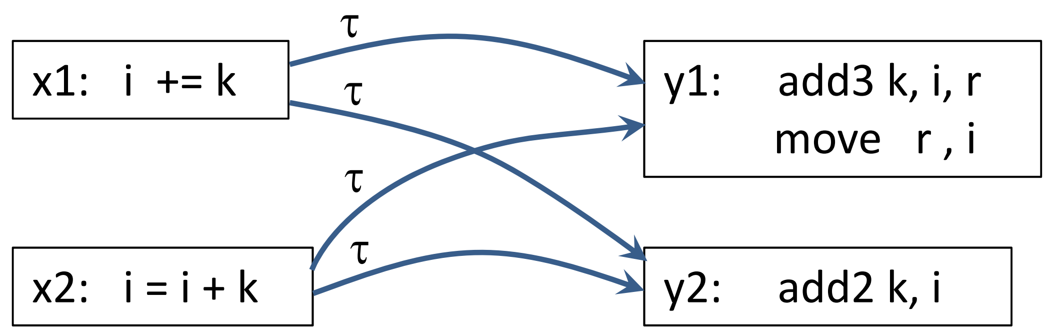
Translation in an abstract setting: a map Source Language ⇔ Target Language

Ex.: transl. from C to Assembler

it is a **relation**

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

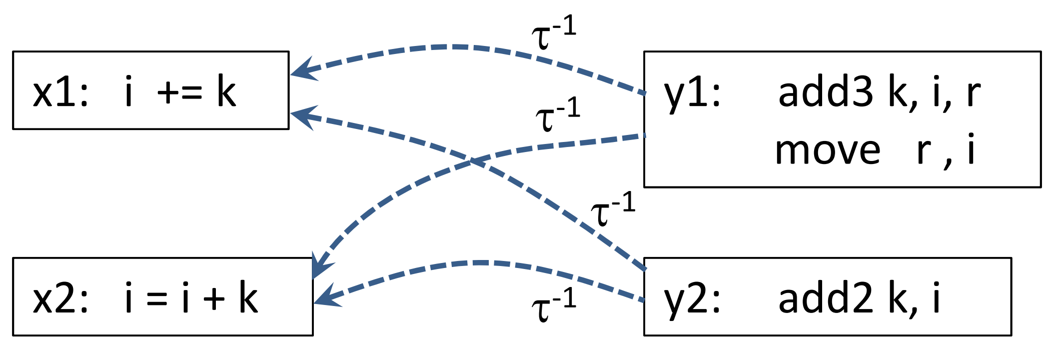
The image of $x1$ is $\tau(x1) = \{y1, y2\}$



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

The inverse translation, τ^{-1}, is not single valued, that is, τ 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

- ***functional***, or ***single-valued*** 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

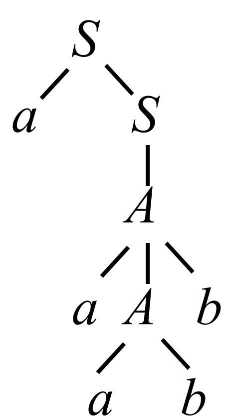
Syntax translation schemes: introductory example

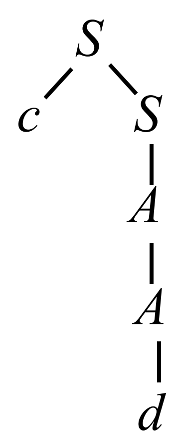
Image string obtained through
simple modifications of the syntax tree of the source string
that do not change its structure
(i.e., the nonleaf/nonterminal nodes and the arcs among them)

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

Source grammar G_1	Target grammar G_2
$S \rightarrow a\, S$	$S \rightarrow c\, S$
$S \rightarrow A$	$S \rightarrow A$
$A \rightarrow a\, A\, b$	$A \rightarrow A$
$A \rightarrow ab$	$A \rightarrow d$

Translation: source tree $\Rightarrow$ target tree





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

is τ^{-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** number 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***

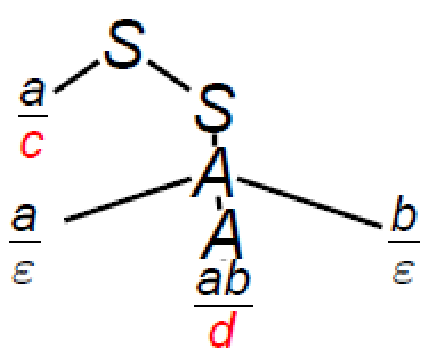
source gramm. G_1	**target gramm. G_2**	**OK?**
$A \to aBcBD$	$A \to xBByDy$	*YES*
$A \to aBcBD$	$A \to xBBy$	***No: D is missing***
$A \to aBcBD$	$A \to xBDBy$	***No: order of NT changed***

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* (i1, *mult* (i2, i3))

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

polish: **prefix**: *add* i1,*mult* i2, i3

postfix: i1, i2, i3 *mult add*

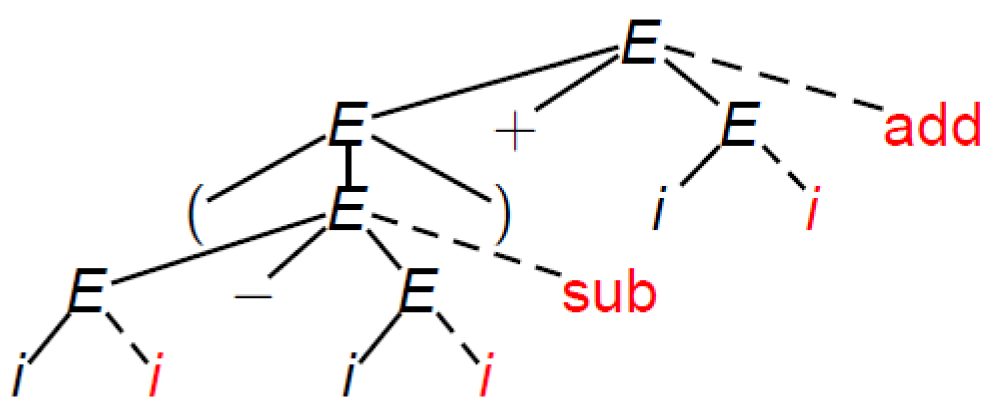
Polish expressions are concise

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 $\rightarrow$ postfix conversion

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



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

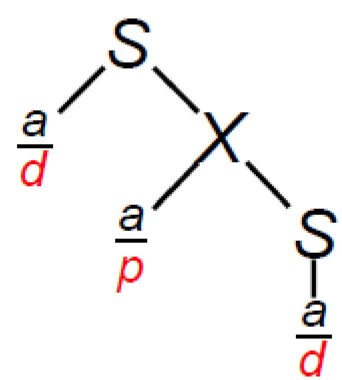
$$\tau(a^n) = \begin{cases} (dp)^{n/2} & if \quad n \ is \ even \\ (dp)^{(n-1)/2}\, d & if \quad n \ is \ odd \end{cases}$$

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

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

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

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



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 $\boldsymbol{G_1}$ having bilateral recursion

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

$$i + i - i \xrightarrow{\tau} \begin{cases} i\,i\,i \text{ sub add} & \overbrace{i + \overbrace{i - i}^{E}}^{E} \\ \\ i\,i \text{ add } i \text{ sub} & \overbrace{\overbrace{i + i}^{E} - i}^{E} \end{cases}$$

Ambiguous Translation: another example

When G_1 has duplicated rules

$$
\begin{array}{l|l}
G_1 & G_2 \\
S \rightarrow aS & S \rightarrow bS \\
S \rightarrow aS & S \rightarrow cS \\
S \rightarrow a & S \rightarrow d
\end{array}
$$

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

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

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

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. top-down ($LL(1)$) parser with writing actions
2. bottom-up ($LR(1)$) parser with writing actions
3. finite state transducer

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 : $\boldsymbol{L = (a \mid b)^*}$ $\boldsymbol{\tau(u) = u^R}$

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

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

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

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
}

Side remark: Similarly, in the Syntax Directed Translation (SDT) method (with attribute grammars) the parser is enriched with actions that compute 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 and safe to execute ***write actions only at reduction time***

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

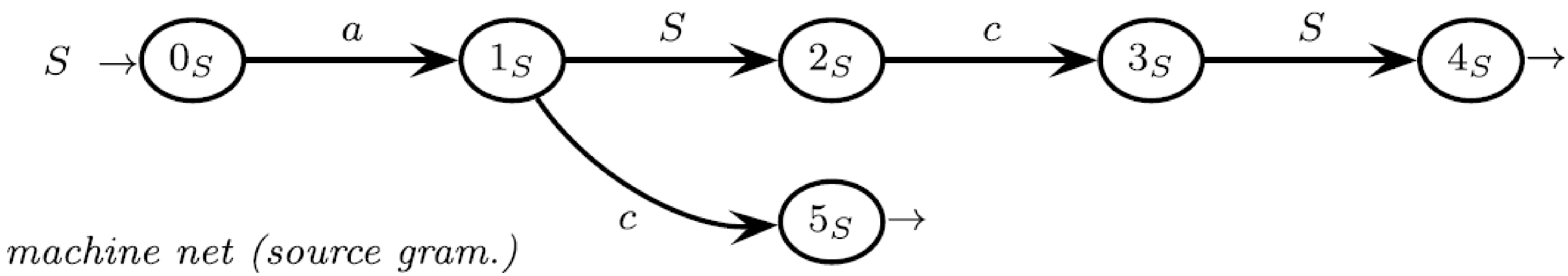
Negative example and conversion to the postfix normal form

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:

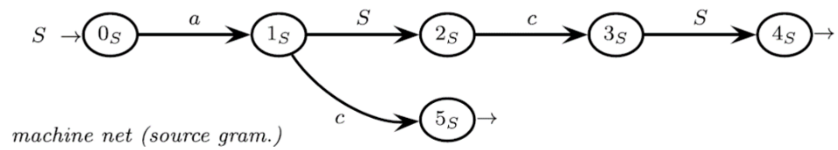
$$G_t\colon\ S \rightarrow \frac{\boldsymbol{a}}{\boldsymbol{b}}\, S\, \frac{\boldsymbol{c}}{\boldsymbol{e}}\, S \ \mid\ \frac{a}{\varepsilon}\,\frac{c}{\varepsilon} \qquad \tau(ac) = \varepsilon \qquad \tau(\boldsymbol{a}\ ac\ \boldsymbol{c}\ ac) = \boldsymbol{b}\ \boldsymbol{e}$$

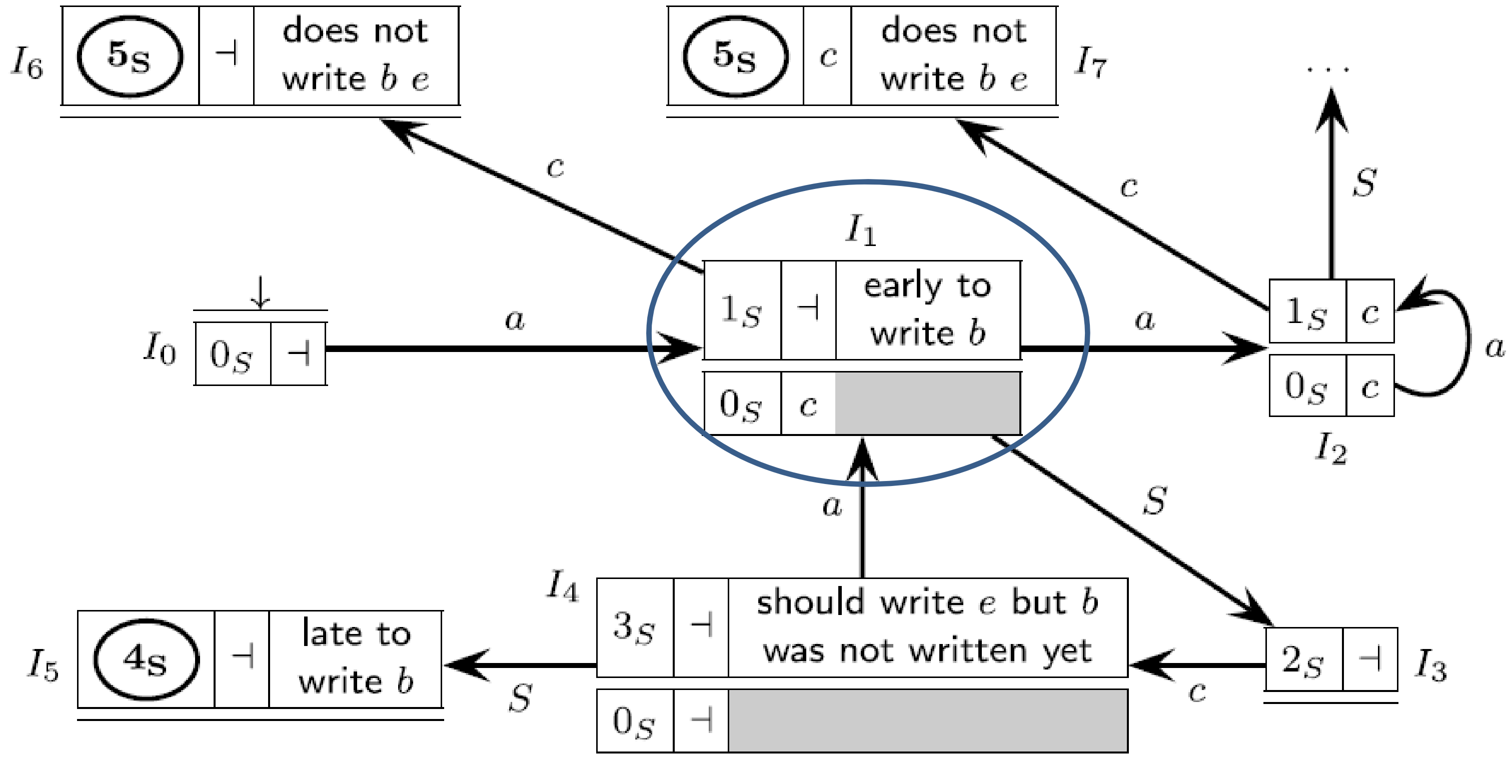
Machine for the source grammar, which is $ELR(1)$



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

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





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

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

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

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 (very easy, but … there still may be problems …):

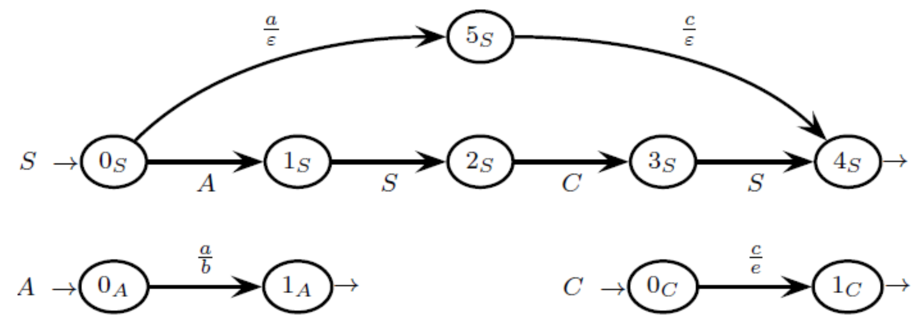
introduce additional nonterminals in place of terminal parts that are not suffix

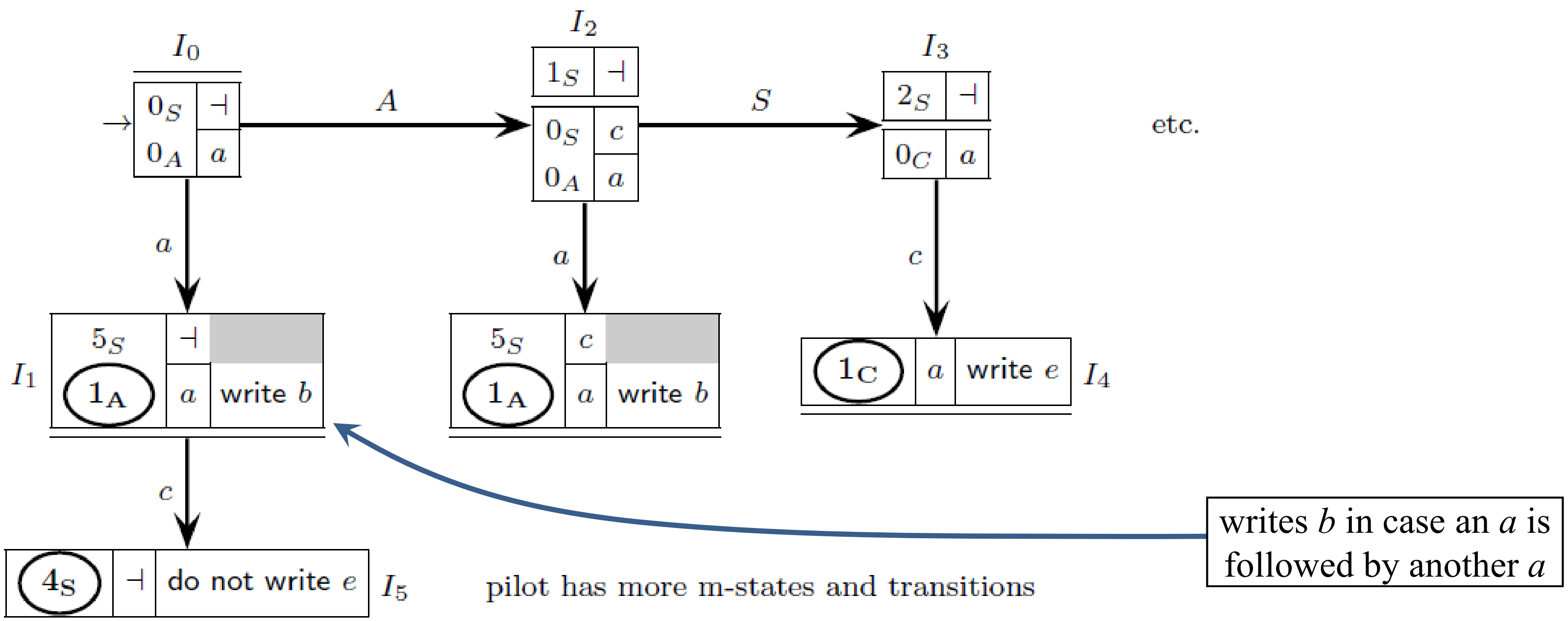
$$G_\tau: \quad S \to A S C S \mid ac \qquad A \to \frac{a}{b} \qquad C \to \frac{c}{e}$$

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

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): it's a «short blanket» …

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 (NB: right linear)

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

derivations of type $A_0 \Rightarrow \ldots A_1 \Rightarrow \ldots A_2 \Rightarrow \ldots A_1 \ \ldots$ generate ***odd*** length strings

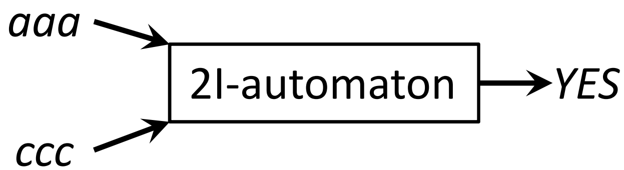
derivations of type $A_0 \Rightarrow \ldots A_3 \Rightarrow \ldots A_4 \ \ldots$ generate ***even*** length strings

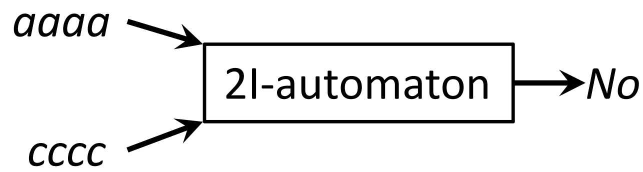
the finite state automaton A_t that accepts $L(G_t)$ can be viewed in two ways

- machine with two input tapes: 2I-automaton (AKA Rabin & Scott machine)
  - it “recognizes” or “accepts” or “defines” 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

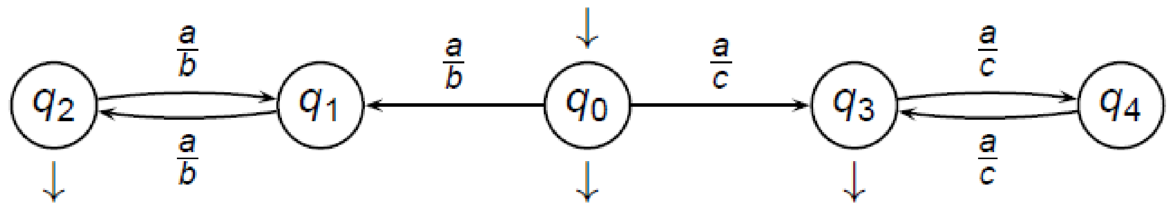
$$\frac{a}{b}, \text{ where } a \in \Sigma \cup \varepsilon, \;\; b \in \Delta \cup \varepsilon$$

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

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



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 $(a^5, c^5) \in \tau$

Non regular translation of regular languages!

Even if ***both*** languages $\boldsymbol{L_1}$ and $\boldsymbol{L_2}$ are regular, the *translation* is not necessarily regular

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

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

An unbounded stack memory is necessary

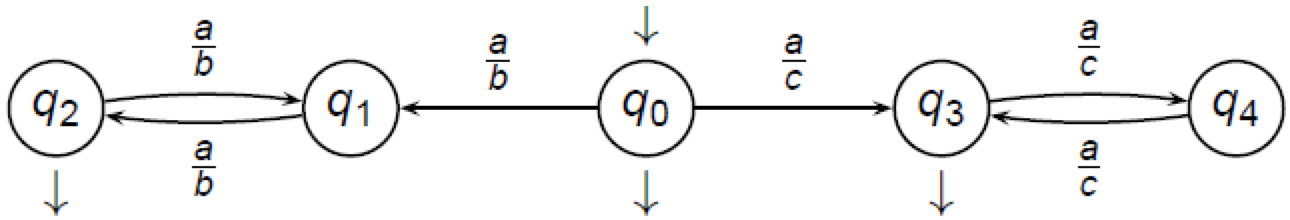
Another model for finite-state translation

Finite transducer or IO-automaton

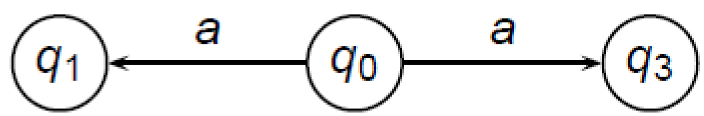
- the second tape is viewed as an output
- the machine ***computes*** the translation as a function of the source string: $\boldsymbol{y = \tau (x)}$
- Several applications:
  - lexeme (token) recognition in the lexical analysis (see lessons on Flex)
  - transformation of simple texts, or signal sequences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

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

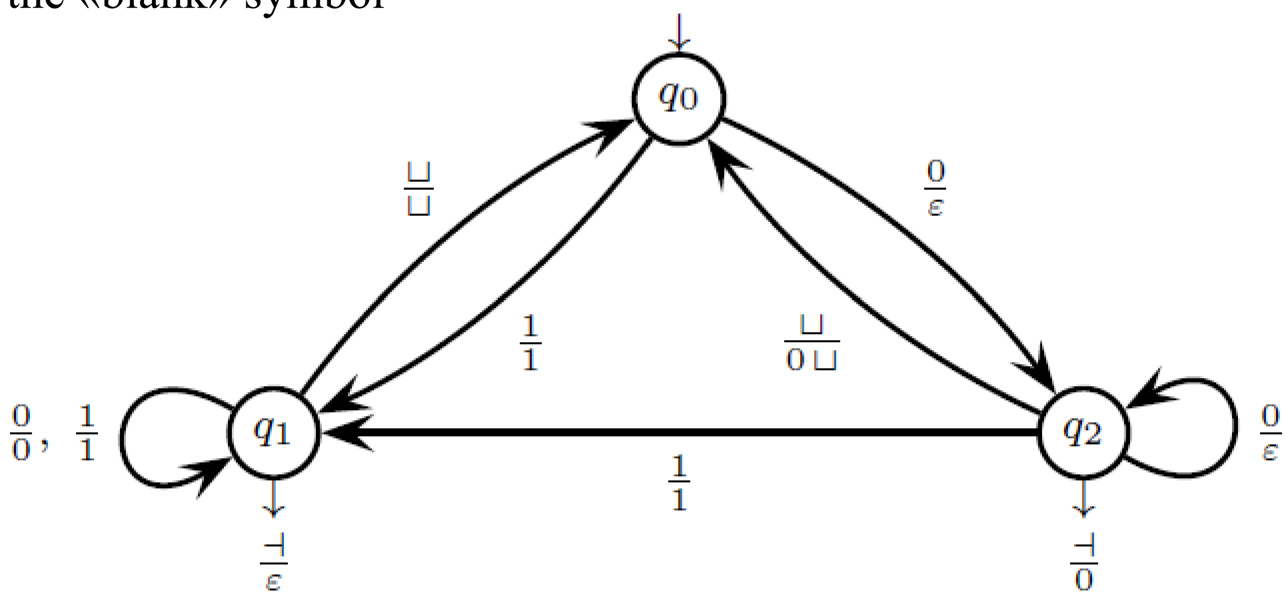
It is a variation of the deterministic IO-automaton model :

- 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 (blanks), eliminate the unsignificant leading zeroes

⊔ represents the «blank» symbol



When terminating in q_2 (a number composed only of 0's) it emits a 0,
but when it terminates in q_1 it does not emit anything
(the last number included some 1's, therefore a non empty string was just written)