Syntax-Directed Transduction and Determinism

Translated and adapted by L. Breveglieri

SYNTAX TRANSDUCTION AND SEMANTIC

SYNTAX TRANSDUCTION

Abstract definition of the correspondence between two formal languages

The translation is based on local transformations of the source text

- replaces each character with another one or with an entire string
- depends on a table that specifies how to transliterate the source alphabet into the destination one

There are two syntax transduction models, regular and syntactic (free)

Finite transduction defined with a regular expression or with a finite transducer, i.e., a finite state automaton that can output a string

Syntactic scheme or transduction grammar that uses a free grammar to define the source and destination languages

- the abstract syntactic transducer is a pushdown automaton
- uses the same algorithms as for parsing, i.e., *ELR* and *ELL*

SYNTAX-DIRECTED SEMANTIC TRANSDUCTION

Attribute grammars and semantic transductions directed by syntax

This approach is semi-formal and exploits the concepts to design compilers

Attribute grammars specify rule by rule the semantic functions to apply

They also compute the entire transduction in a compositional way

The addition of attributes and semantic functions to the finite transducers is well documented, for instance in lexical analysis, type checking, the transduction of conditional instructions and semantic-directed syntax analysis

ANALOGIES OF FORMAL LANGUAGE AND TRANSDUCTION THEORIES

Set-theoretic: the set of the language phrases corresponds to the set of the pairs (source and destination strings) that model the transduction relation

Generative: the language grammar becomes a transduction grammar, which generates pairs of phrases (source and destination)

Operative: the finite state automaton or the pushdown automaton becomes a transducer automaton or a syntax analyzer that computes the transduction

TRANSDUCTION RELATION AND TRANSDUCTION FUNCTION

Given a source and a destination alphabet Σ and Δ, a *transduction* (or *translation*) is a correspondence of strings that is modeled as a mathematical binary relation between the universal languages Σ^* and Δ^*, i.e., as a subset of the product $\Sigma^* \times \Delta^*$

The transduction relation ρ is a set of pairs (x, y) with $x \in \Sigma^*$ and $y \in \Delta^*$, and the languages L_1 and L_2 are the projections of ρ over the 1st and 2nd component

$$\rho = \{ (x, y), \ldots \} \subseteq \Sigma^* \times \Delta^*$$

$$L_1 = \{ x \in \Sigma^* \mid \text{ for some string } y \text{ such that } (x, y) \in \rho \}$$
$$L_2 = \{ y \in \Delta^* \mid \text{ for some string } x \text{ such that } (x, y) \in \rho \}$$

Equivalently a transduction τ is a total function, in general multi-valued

$$\tau : \Sigma^* \to \wp(\Delta^*)$$
$$\tau(x) = \{ y \in \Delta^* \mid (x, y) \in \rho \}$$
$$L_2 = \tau(L_1) = \bigcup_{x \in \Sigma^*} \tau(x)$$

The inverse transduction is as follows

$$\tau^{-1} : \Delta^* \to \wp(\Sigma^*)$$
$$\tau^{-1}(y) = \{ x \in \Sigma^* \mid y \in \tau(x) \}$$

There are the following seven transduction cases, depending on the function type

1) *total*: every source string has an image of one or more destination strings (possibly including the empty string)
2) *partial*: a few source strings may not have any image string, i.e., they are not translated into any destination string (not into the empty string either)
3) *one-valued*: every source string has an image of at most one destination string
4) *multi-valued*: a few source strings may have an image of two or more destination strings (even of infinitely many strings)

The remaining cases make sense only if the transduction is total and one-valued

5) *injective*: any two different source strings have different image strings, i.e., every destination string may be the image of at most one source string
6) *surjective*: every destination string is the image of one or more source strings
7) *one-to-one* or *bijective*: there is a one-to-one correspondence between source and destination strings

A transduction is one-to-one if and only if it is both injective and surjective

Such a transduction is invertible and its inverse is one-to-one as well

TRANSLITERATION

Define a transduction as the character-by-character repetition of a punctual transformation

The simplest such case is that of *transliteration*, also called *homomorphism*

Every source character is transduced into a corresponding destination character or more generally into a string, possibly even into the empty string

Transliteration is surely a one-valued transduction, but as said before in general its inverse transduction is partial and may be multi-valued

If a transliteration can cancel a source character and replace it by the empty string, then its inverse transduction is surely multi-valued and actually infinitely-valued !

The essence of transliteration is that each character is translated independently of how the sourrounding characters are translated

Such a notion of transduction is weak and obviously insufficient to compile all the real technical languages

REGULAR (or RATIONAL) TRANSDUCTION

In order to define a transduction relation one can exploit regexps: in this case the regular operators union, concatenation and star are applied in parallel to pairs of strings (source, destination) rather than to individual strings

EXAMPLE – coherent transliteration of an operator

The source text is a list of unary integers separated by the operator "/"

The transduction changes the operator "/" either into the symbol ":" or into the symbol divide, but it never mixes these alternatives

The integers are in unary notation

$$\Sigma = \{1, /\} \quad \Delta = \{1, :, \div\}$$

$$(3/5/2\ ,\ 3:5:2)\ ,\ (3/5/2\ ,\ 3 \div 5 \div 2)$$

$$(1,1)^{+}((/,:)(1,1)^{+})^{*} \cup (1,1)^{+}((/,\div)(1,1)^{+})^{*}$$

$$\left(\frac{1}{1}\right)^{+}\left(\frac{/}{:}\left(\frac{1}{1}\right)^{+}\right)^{*} \cup \left(\frac{1}{1}\right)^{+}\left(\frac{/}{\div}\left(\frac{1}{1}\right)^{+}\right)^{*}$$

$$\frac{1}{1}\frac{/}{\div}\frac{1}{1}\frac{1}{1} \quad (1/11,\ 1 \div 11)$$

DEFINITION – regular or rational transduction expression (regexprt)

A regular or rational transduction expression (regexprt) *r* is a regexp that contains union, concatenation and star (and cross) operators, the terms of which are pairs (*u*, *v*) of strings (possibly empty) over the source and destination alphabets.

$$\tau = \{(x, y), \ldots\} \subseteq \quad \text{finite subsets of} \quad \left(\Sigma^* \times \Delta^*\right)$$

where

$Z \subset \Sigma^* \times \Delta^*$ is the set of the pairs (*u*, *v*) contained in the regexprt *r*.

The pairs (*x*, *y*) of corresponding source and destination strings are such that:

- there exists a string $z \in Z^*$ that belongs to the regular set *r*
- *x* and *y* are respectively the projection of *z* on the 1st and the 2nd component

The set of the source strings defined by a regexprt is a regular language. The same holds for the set of the destination strings

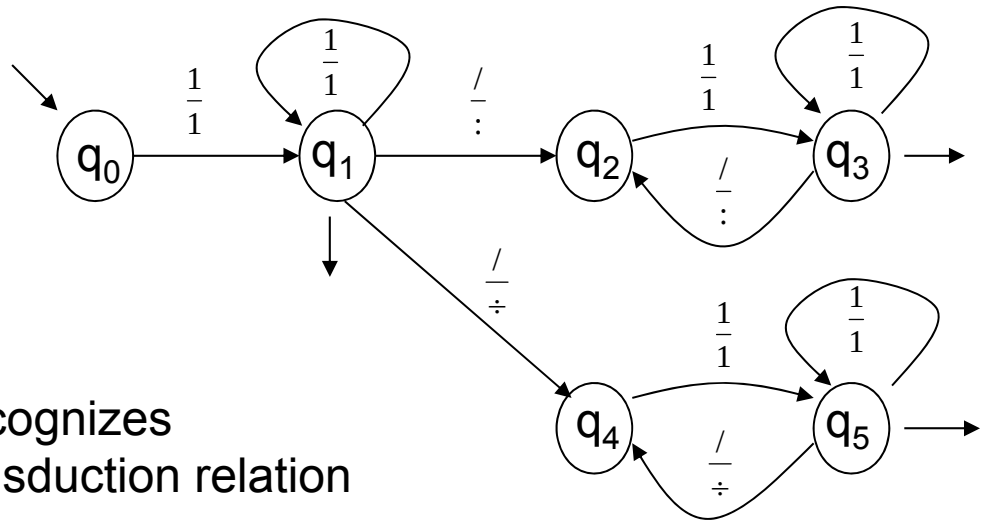
It is not granted, instead, that a relation from source language to destination language, both regular languages, may be definable by means of a regexprt

TWO-TAPE RECOGNIZER AUTOMATON

Think of the set Z of the pairs contained in the regexprt as terminal symbols; then it is simple to associate to it a recognizer automaton with two input tapes

EXAMPLE (continued):

the finite state automaton aside recognizes that (11/1, 11:1) belongs to the transduction relation and that (11/1,1:1) does not



The concept of regexprt and of two-tape recognizer is a method for defining the transduction, and also for building transduction functions in a rigorous way. It also allows to deal with both lexical and syntax analyzers in a uniform way

FORMS OF THE TWO-TAPE AUTOMATON

When the labels of the arcs of a TWO-TAPE automaton are projected over the first component (source alphabet), one obtains a pure recognizer (with only one input tape) over the source alphabet Σ; this is called *underlying input automaton*, and it is the recognizer of the pure source language

In describing the two-tape recognizer one can always suppose, without any loss of generality, that every move reads one, and only one, char in the source tape

DEFINITION – TWO-TAPE AUTOMATON

An automaton with two input tapes (two-tape automaton) has, similarly to an ordinary automaton with only one input tape, a state set Q, an initial state q_0, and a subset F of final states. The transition function is:

$$\delta : (Q \times \Sigma \times \Delta^*) \rightarrow \text{finite subsets of } Q$$

If $q' \in \delta(q, a, u)$, when the automatons is in state q and reads a and u in the first and second tape, respectively, it may go to the next state q'. The acceptance condition is the same as that holding in the case of ordinary one-input recognizers

Normal form: a move may read only one character in either tape, not in both ones. The labels of the arcs can only be of either types:

$$\frac{a}{\varepsilon}, \frac{\varepsilon}{a} \qquad a \in \Sigma \bigcup \Delta$$

More coincisely: $\frac{(a)^* b}{d} \mid \frac{(a)^* c}{e}$ equivalent to: $\frac{(a)^*}{\varepsilon}\frac{b}{d} \mid \frac{(a)^*}{\varepsilon}\frac{c}{e}$

PROPERTY – the familes of regular transductions defined by regexprts and two-tape automata (in general non-deterministic) coincide

A regexprt defines a regular language R that hasas alphabet the cartesian product of the source and destination alphabets

$$(u,v) \equiv \frac{u}{v} \text{ where } u \in \Sigma^*, v \in \Delta^*$$

By separating the source and destination components in the regexprt, one obtains an interesting formulation of the regular transduction relation.

NIVAT THEOREM the following conditions are equivalent

1. the transduction relation ρ is regular with $\rho \subseteq \Sigma^* \times \Delta^*$

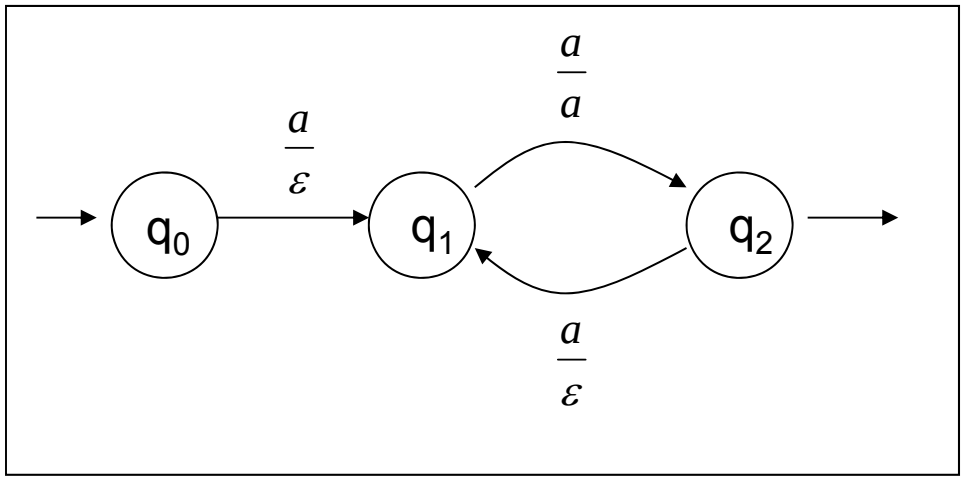
2. there exists a regular language R over the alphabet C and two alphabetic homomorphims (or translitterations) h_1: C → Σ and h_2: C → Δ such that: $\rho = \{(h_1(z), h_2(z)) \mid z \in R\}$

3. if the two alphabets Σ and Δ are disjoint, there exists a language R over the union Σ U Δ such that: $\rho = \{(h_\Sigma(z), h_\Delta(z)) \mid z \in R\}$
where h_Σ and h_Δ are the projections of Σ U Δ over the component source and destination alphabets Σ and Δ, respectively

EXAMPLE – HALVING – the image string is the halved source string

Transduction relation: $\{(a^{2n}, a^n) \mid n \geq 1\}$ regexprt $\left(\frac{aa}{a}\right)^+$

An equivalent two-tape automaton:



Nivat theorem
(2nd statement)

$\left(\frac{a}{\varepsilon}\frac{a}{a}\right)^+ \quad \frac{a}{\varepsilon} = c \quad \frac{a}{a} = d \quad C = \{c, d\}$

$R = (cd)^+$ homomorphisms h_1 h_2

c	a	ε
d	a	a

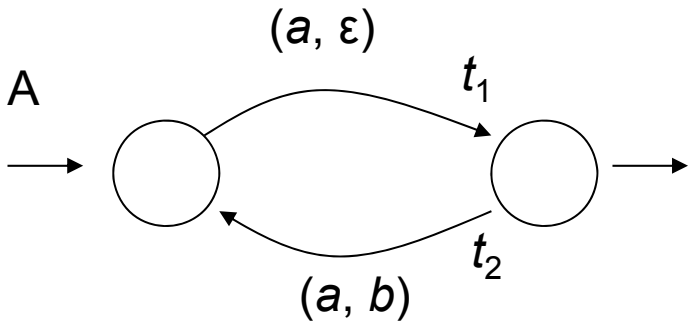
The two alphabetic hom.s generate the required transduction. For instance, if $z = cdcd \in R$ one has $h_1(z) = aaaa$ and $h_2(z) = aa$

Nivat theorem
(3rd statement)

Projecting over the source and destination alphabets defines the strings that correspond to each other in the transduction

$$\Sigma = \{a\} \quad \Delta = \{b\} \quad \{(a^{2n}, b^n) \mid n \geq 1\} \quad \left(\frac{a}{\varepsilon}\frac{a}{b}\right)^+$$

$$R \subseteq (\Sigma \cup \Delta)^* \quad R = (a\varepsilon ab)^+ = (aab)^+$$

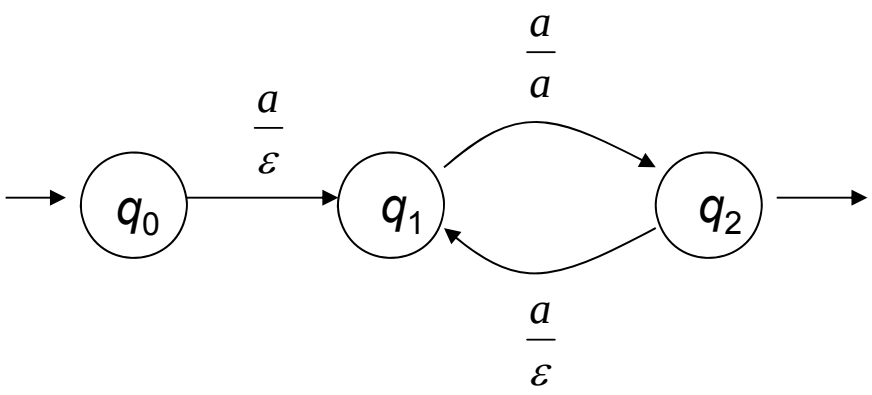


$$T = \{t_1, t_2\} \quad L(A) = (t_1 t_2)^+$$

$$\Pi_\Sigma(t_1) = a \quad \Pi_\Sigma(t_2) = a$$

$$\Pi_\Delta(t_1) = \varepsilon \quad \Pi_\Delta(t_2) = b$$

The correspondence between the finite state automaton model and the right linear grammar model allows to design a **transduction grammar**, along with the two-tape automaton and the regexprt



$$S \to \frac{a}{\varepsilon} Q_1$$

$$Q_1 \to \frac{a}{a} Q_2$$

$$Q_2 \to \frac{a}{\varepsilon} Q_1 \mid \frac{\varepsilon}{\varepsilon}$$

A grammar-based definition may be preferable in the context of syntax transductions, where the syntactic support is a free grammar

TRANSDUCTION FUNCTION AND TRANSDUCER AUTOMATON

An automaton can be used as an algorithm to compute a transduction function. Define a new model: transducer automaton or input-output automaton (IO automaton). It reads a source string in the input tape and writes a destination string on the output tape.

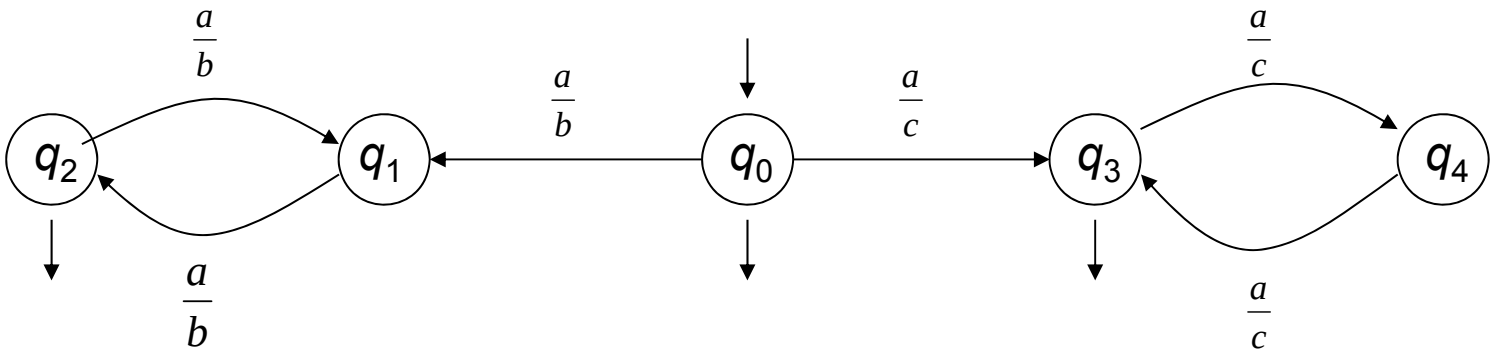
The cases deserving most interest are those of a one-valued transduction function and of a transduction function computable in a deterministic way.

EXAMPLE – a non-deterministic transduction

Is a one-valued transduction, which can not be computed in a deterministic way by means of a deterministic finite state IO-automaton

$$\rho = \left\{(a^{2n}, b^{2n}) \mid n \geq 0\right\} \bigcup \left\{(a^{2n+1}, c^{2n+1}) \mid n \geq 0\right\}$$

$$\tau(a^n) = \begin{cases} b^n, n \text{ even} \\ c^n, n \text{ odd} \end{cases} \quad \text{regexprt} \quad \left(\frac{a^2}{b^2}\right)^* \bigcup \frac{a}{c}\left(\frac{a^2}{c^2}\right)^*$$



The two-tape automaton is det, but the IO automaton is not det

$$\tau(aa) = bb$$

Given the input string *aa*, two computations are possible: $q_0 \rightarrow q_1 \rightarrow q_2$ or $q_0 \rightarrow q_3 \rightarrow q_4$, but only the former one is valid and reaches a final state

To compute the transduction function, the input tape is read and the output tape is written. This way of doing is efficient. As soon as the input string is fully scanned, the transducer should be in a final state and can write the last suffix on the output tape, depending on the specific final state it has reached

DEFINITION – deterministic model of a sequential transducer. A finite state sequential det. transducer T, or IO automaton, is a deterministic machine defined as follows.
It has a finite set Q of states, a source alphabet Σ, an initial state q_0, a subset F of final states and three one-valued functions:

1. the transition function δ, which computes the next state
2. the output function η, which computes the string to write in each move
3. the final function φ, which computes the last suffix to concatenate to the output string when the transducer reaches a final state

The domains and images are: $\delta : Q \times \Sigma \to Q, \quad \eta : Q \times \Sigma \to \Delta^*, \quad \varphi : Q \times \{\text{-}|\} \to \Delta^*$

$$\delta(q,a) = q'$$
$$\eta(q,a) = u$$

$$q \xrightarrow[u]{a} q'$$

The transduction $\tau(x)$ computed by T is the concatenation of two strings:
the string computed by the output function and that computed by the final function

$$\left\{ yz \in \Delta^* \mid \exists \text{ a computation labeled by } \frac{x}{y} \quad \text{ending in} \quad r \in F \wedge z = \varphi(r, \text{-}) \right\}$$

The IO automaton T is deterministic if the underlying input automaton
$(Q, \Sigma, \delta, q_0, F)$ is deterministic and both the output and the final functions
are one-valued

The IO automaton T is instead non-deterministic, even if the underlying input automaton is deterministic, if between two states of T there is an arc of the type:

A function computable by means of a sequential finite state transducer (IO automaton) is said to be a sequential function

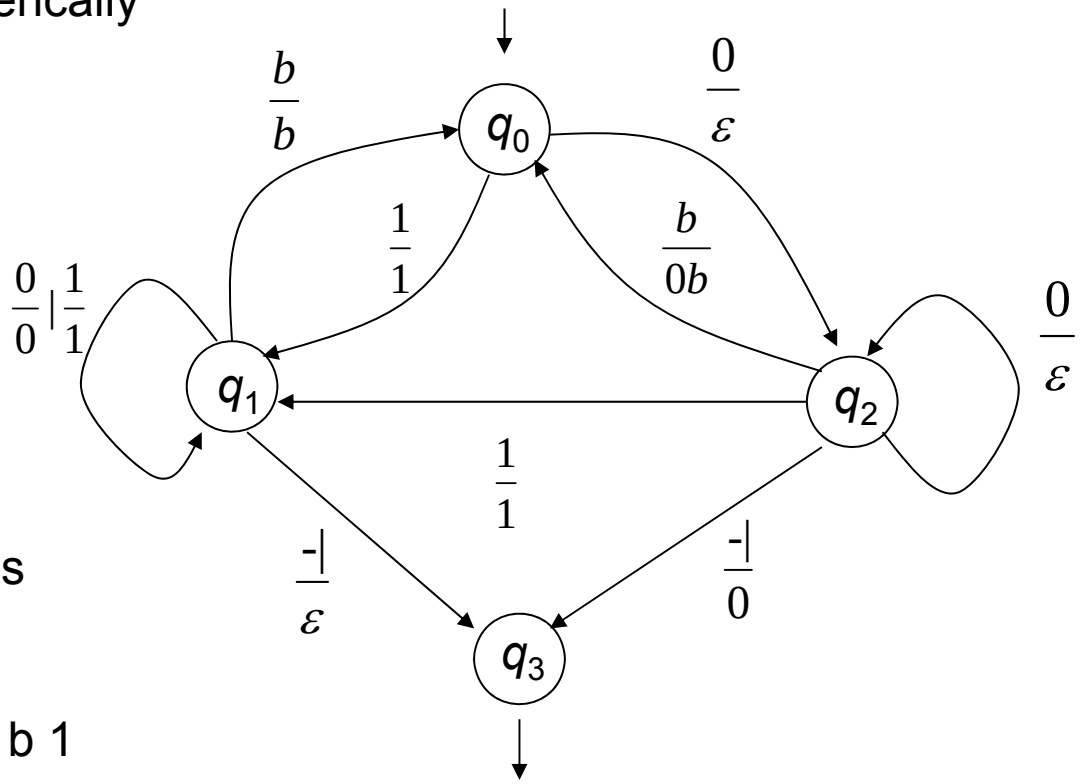
$$\frac{a}{\{b\} \bigcup \{c\}}$$

multi-valued output function

EXAMPLE – elimination of leading zeroes

Text is a list of binary integer numbers, separated by a blank (b). Transduction (one-valued) removes the leading zeroes, which do not matter numerically

$$\left(\frac{0^{+}b}{0b} \mid \left(\frac{0}{\varepsilon}\right)^{*}\frac{1}{1}\left(\frac{0}{0} \mid \frac{1}{1}\right)^{*}\frac{b}{b}\right)^{*}\left(\frac{0^{+}\text{-}\mid}{0} \mid \left(\frac{0}{\varepsilon}\right)^{*}\frac{1}{1}\left(\frac{0}{0} \mid \frac{1}{1}\right)^{*}\frac{\text{-}\mid}{\varepsilon}\right)$$



Transducing the text

0 0 b 0 1 -|

passes through the states

$q_0\, q_2\, q_0\, q_2\, q_1\, q_3$

and writes

ε ε 0 b ε 1 ε = 0 b 1

EXAMPLE

$$\tau(a^n) = \begin{cases} p, & n \text{ even} \\ d, & n \text{ odd} \end{cases}$$

The example exploits the concept of final function, which is included in the sequential transducer model

The sequential transducer has two states, both final, corresponding to the parity class of the source string. The transducer does not write anything in the output tape while it is executing the moves, until it reaches a final state; then, depending on the reached final state, it outputs the symbol *p* rather than *d*

The composition of two sequential functions is still a sequential function

TWO-PASS SEQUENTIAL TRANSDUCTION WITH MIRRORING – given a generic transduction, specified by means of a regexprt or a two-tape automaton, there does not always exist a deterministic sequential transducer (a det. IO automaton) that can compute it directly. However, it is always possible to compute such a transduction in two passes: the former det. seq. transducer scans the input string from left to right and outputs an intermediate string, which is soon after given as input to the latter det. seq. transducer (different from the former one), which scans it from right to left (opposite direction) and outputs the final string. In this way it is possible, for instance, to compute deterministically the transduction of a^n into b^n if *n* is even, or in c^n if *n* is odd

So far only regular (or rational) transductions have been considered, computed by a deterministic algorithm with a finite memory that examines the source string in one or two passes. But memory finiteness is a limit for many situations in computer science.

EXAMPLE – The transduction relation below is not regular, as it is necessary to store the entire source string before outputting its mirrored image.

$$\left\{ (x, x^R) \mid x \in (a \mid b)^* \right\}$$

Nivat: the transduction can be obtained by Nivat theorem, but it is necessary to construct the language R by merging the source string *x* and the image string $y = (x')^R$ (which should also be translitterated over the alphabet { *a*', *b*' }); however the language R is not regular any longer

PURE SYNTAX TRANSDUCTON

When the source language is defined by means of a grammar, it is natural to consider transductions where syntactic structures are translated individually. The image of the entire phrase can then be computed by composing the individual transductions

To this purpose it is necessary to match source and destination grammar rules

DEFINITION – CONTEXT-FREE (or ALGEBRAIC) TRANSDUCTION

A *transduction grammar* G is a free grammar over the terminal alphabet C consisting of pairs (*u*, *v*) of strings over the source and destination alphabets, respectively

$$G = (V, \Sigma, \Delta, P, S)$$

$$C \subseteq \Sigma^* \times \Delta^* \qquad \frac{u}{v}$$

The *syntactic transduction scheme* associated with grammar G is a set of pairs of source and destination rules, obtained eliminating from the grammar G the characters of the source and destination alphabets, respectively

The *transduction relation* defined by grammar G is the following:

$$\tau(G) = \{(x, y) \mid \exists z \in L(G) \wedge x = h_\Sigma(z) \wedge y = h_\Delta(z)\}$$

$$h_\Sigma : C \to \Sigma \quad h_\Delta : C \to \Delta$$

The transduction grammar and the syntactic transduction scheme are simply two notational variants defining the same transduction relation

EXAMPLE – algebraic transduction that computes mirroring

Transduction grammar:

$$S \to \frac{a}{\varepsilon} S \frac{\varepsilon}{a} \mid \frac{b}{\varepsilon} S \frac{\varepsilon}{b} \mid \frac{\varepsilon}{\varepsilon}$$

Transduction scheme:

source gramm. G_1	dest. gramm. G_2
$S \to aS$	$S \to Sa$
$S \to bS$	$S \to Sb$
$S \to \varepsilon$	$S \to \varepsilon$

To obtain a pair of strings of the transduction relation:

a) construct a derivation for string *z*

$$S \Rightarrow \frac{a}{\varepsilon} S \frac{\varepsilon}{a} \Rightarrow \frac{a}{\varepsilon}\frac{a}{\varepsilon} S \frac{\varepsilon}{a}\frac{\varepsilon}{a} \Rightarrow$$
$$\Rightarrow \frac{a}{\varepsilon}\frac{a}{\varepsilon}\frac{b}{\varepsilon} S \frac{\varepsilon}{b}\frac{\varepsilon}{a}\frac{\varepsilon}{a} \Rightarrow \frac{a}{\varepsilon}\frac{a}{\varepsilon}\frac{b}{\varepsilon}\frac{\varepsilon}{\varepsilon}\frac{\varepsilon}{b}\frac{\varepsilon}{a}\frac{\varepsilon}{a} = z$$

b) project *z* over the two alphabets

$$h_{\Sigma}(z) = aab \qquad h_{\Delta}(z) = baa$$

EXAMPLE – transduction of palindromes – changes the terminal characters of the left and right halves of the string

$$G_p = \{S \to aSa' \mid bSb' \mid \varepsilon\}$$

This transduction grammar is actually a translitterated version of that of the previous example

$$\frac{a}{\varepsilon} \text{ in } a, \quad \frac{b}{\varepsilon} \text{ in } b, \quad \frac{\varepsilon}{a} \text{ in } a', \quad \frac{\varepsilon}{b} \text{ in } b'$$

The above observation can be generalised, obtaining a version of the Nivat theorem suited for algebraic transductions

PROPERTY – free languages and syntactic transductions

The following statements are equivalent:

1. The transduction relation is defined by a transduction grammar G
2. There exists a free language L over C and two alphabetic hom.s h_1 and h_2 such that:
3. If Σ and Δ are disjoint alphabets, there exists a language L over Σ U Δ such that:

$$\tau \subseteq \Sigma^* \times \Delta^*$$

$$\tau = \{(h_1(z), h_2(z)) \mid z \in L\}$$

$$\tau = \{(h_\Sigma(z), h_\Delta(z)) \mid z \in L\}$$

EXAMPLE – transducing string x $\in$ (*a* | *b*)* into the mirror image *y* = x^R

Such a transduction can be expressed by means of the palindrome language.

$$L = \left\{ uu'^{R} \mid u \in (a \mid b)^{*} \wedge u' \in (a' \mid b')^{*} \right\} = \left\{ \varepsilon, aa', ..., abbb'b'a', ... \right\}$$

Here are the two alphabetic homomorphisms (or translitterations):

	h_1	h_2
a	a	ε
b	b	ε
a'	ε	a
b'	ε	b

The string *abb'a'* $\in$ L is projected into the string pair (*ab*, *ba*)

INFIX (or POLISH) NOTATION

An application of syntax transduction: conversion of expressions (arithmetic, logical, etc) into notations differing as for the position of the operators and the possible presence of delimiters, like pharenteses

Degree of an operator: is the number of arguments (unary, binary, …, operators)
Variadic operator: an operator with variable degree
Prefix / Postfix / Infix / Mixfix operator.

$$o_o\ arg_1\ o_2\ arg_2\ ...o_{n-1}\ arg_n\ o_n$$

$$\text{if}\ \ arg_1\ \ \text{then}\ \ arg_2\ \ [\text{else}\ arg_3]$$

$$\text{jump\_if\_false}\ \ arg_1\ arg_2$$

Polish notation: does not use pharenteses and the operators are all prefix or all postfix

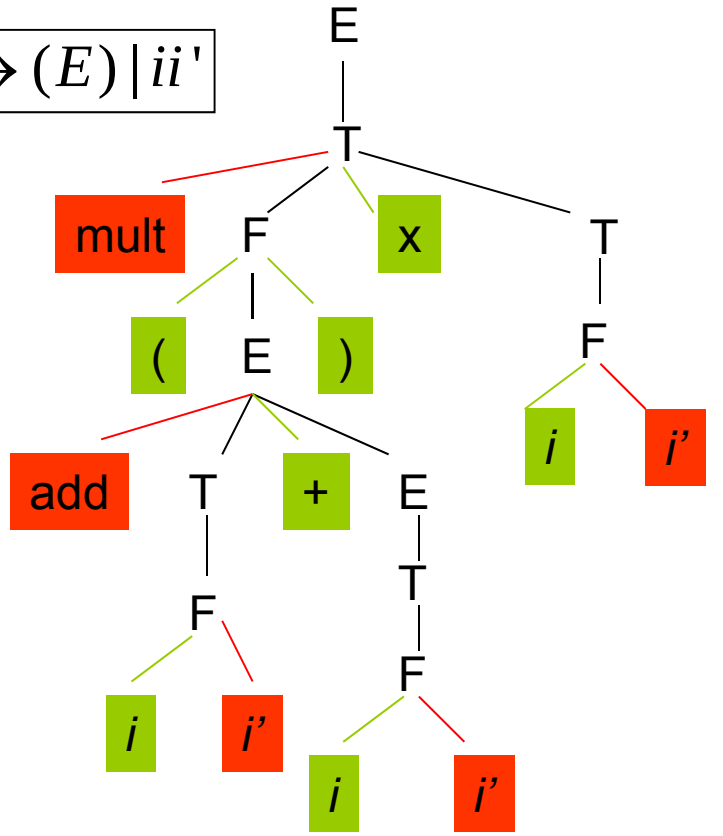
EXAMPLE – Converting an arithmetic expression from the standard notation (infix) into the prefix notation (this is frequently used in compilers to eliminate pharenteses)
Transduction grammar:

$$E \to \text{add } T + E \mid T \quad T \to \text{mult } F \times T \mid F \quad F \to (E) \mid ii'$$

Note: if the source and destination alphabets are disjoint, the transduction grammar can be denoted more simply, as it is not necessary to separate the source and dest. symbols by the fraction line

$$E \to \text{add } T + E \;\; \ldots \;\; E \to \frac{\varepsilon}{\text{add}} \; T \frac{+}{\varepsilon} E$$

source gramm. G_1	dest. gramm. G_2
$E \to T + E$	$E \to \text{add } T\ E$
$E \to T$	$E \to T$
$T \to F \times T$	$T \to \text{mult } F\ T$
$T \to F$	$T \to F$
$F \to (E)$	$F \to E$
$F \to i$	$F \to i'$



(i + i) x i

mult add i' i' i'

AMBIGUITY OF THE SOURCE GRAMMAR

The transduction grammars or schemes of practical interest are one-valued.

If the source grammar is ambiguous → two or more syntax trees
different source and destination trees
different images

source gram. G_2

$E \to \text{add}\ \ T\ \ E$

$E \to T$

$T \to \text{mult}\ \ F\ \ T$

$T \to F$

$F \to E$

$F \to i'$

dest. gram. G_1

$E \to T + E$

$E \to T$

$T \to F \times T$

$T \to F$

$F \to (E)$

$F \to i$

G_2 is ambiguous as it admits the following circular derivation

$$E \Rightarrow T \Rightarrow F \Rightarrow E$$

$$E \Rightarrow T \Rightarrow F \Rightarrow i',$$
$$E \Rightarrow T \Rightarrow F \Rightarrow E \Rightarrow T \Rightarrow F \Rightarrow i',...$$

different transductions:

$$E \Rightarrow T \Rightarrow F \Rightarrow i',$$
$$E \Rightarrow T \Rightarrow F \Rightarrow (E) \Rightarrow (T) \Rightarrow (F) \Rightarrow (i'),...$$

Even if the source grammar is not ambiguous, the transduction may be multi-valued

EXAMPLE

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

$G_1 = \{S \to aS \mid a\}$ is not ambiguous

$\tau(aa) = \{bd, cd\}$

PROPERTY

Let T be a transduction grammar (or scheme) such that:

1) the underlying source grammar G_1 is not ambiguous
2) every rule of G_1 corresponds to one rule of T

Then the transduction defined by T is one-valued

If the transduction grammar T is ambiguous, then also the underlying source grammar G_1 is so. The opposite implication does not hold, in general

EXAMPLE: unambiguos transduction grammar, ambiguous underlying source grammar

$$S \to \frac{\text{if}}{\text{if}}\frac{c}{c}\frac{\text{then}}{\text{then}} S \frac{\varepsilon}{\text{end\_if}} \mid \frac{\text{if}}{\text{if}}\frac{c}{c}\frac{\text{then}}{\text{then}} S \frac{\text{else}}{\text{else}} S \frac{\varepsilon}{\text{end\_if}} \mid a$$

To design a compiler, multi-valued transduction grammars should be avoided, as they may cause multiple transductions

TRANSDUCTION GRAMMAR AND PUSHDOWN TRANSDUCER AUTOMATON

To compute a transduction defined by means of a transduction grammar, the transducer automaton need have a unbounded pushdown stack memory

A pushdown transducer (or pushdown IO automaton) is a recognizer pushdown automaton empowered with a one-way write head and an output tape. It can write a character at each transition

DEFINITION – pushdown transducer

Eight entities:

Q	state set
Σ	source (inpout) alphabet
Γ	stack (memory) alphabet
Δ	destination (output) alphabet
δ	transition and output function
q_0	initial state
Z_0	initial stack symbol
F	subset of final states

Domain of δ: $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$

Image of δ: $Q \times \Gamma^* \times \Delta^*$

Meaning: if $(q'', \gamma, y) = \delta(q', a, Z)$ the transducer, from the current state q', when reads a from the input tape and pops Z from the stack, moves to state q'', pushes γ onto tha stack and writes y onto the output tape

The end condition may be defined by final state or by empty stack

FROM THE TRANSDUCTION GRAMMAR TO THE AUTOMATON

Transduction schemes and automata are equivalent representations of the same transduction relation. The former is generative, the latter procedural. It is always possible to convert each into the other one

PROPERTY: a transduction relation is defined by a transduction grammar (or scheme) if and only if it is defined by a pushdown transduction automaton

CONSTRUCTION OF THE (INDETERMINISTIC) PREDICTIVE PUSHDOWN TRANSDUCER STARTING FROM THE TRANSDUCTION GRAMMAR

Formalize the correspondence between grammar rule and automaton move.
C is the set of the pairs aside
(input char, output string):

$$\frac{\varepsilon}{v}, v \in \Delta^{+} \qquad \frac{b}{w}, b \in \Sigma, w \in \Delta^{*}$$

(the automaton is not deterministic, in general; by adding finite states it may be sometimes turned into deterministic form, though not always)

	Rule	Move	Comment
1	$A \to \frac{\varepsilon}{v} BA_1 \ldots A_n$ $n \geq 0, v \in \Delta^+, B \in V,$ $A_i \in (C \cup V)$	**if** (top = A) **then** **write** (*v*) **pop** **push** ($A_n \ldots A_1 B$)	write output string *v* push symbols B $A_1 \ldots A_n$
2	$A \to \frac{b}{w} A_1 \ldots A_n$ $n \geq 0, b \in \Sigma, w \in \Delta^*,$ $A_i \in (C \cup V)$	**if** (cur_char = *b* $\wedge$ top = A) **then** **write** (*v*) **pop** **push** ($A_n \ldots A_1$) **shift input head**	read input char *b* write output string *w* push symbols $A_1 \ldots A_n$
3	$A \to BA_1 \ldots A_n$ $n \geq 0, B \in V,$ $A_i \in (C \cup V)$	**if** (top = A) **then** **pop** **push** ($A_n \ldots A_1$)	push symbols $A_1 \ldots A_n$
4	$A \to \frac{\varepsilon}{v} \quad v \in \Delta^+$	**if** (top = A) **then** **write** (*v*) **pop**	write output string *v*

	Rule	Move	Comment
5	$A \to \varepsilon$	**if** (top = A) **then** **pop**	
6	for every pair $\frac{\varepsilon}{v} \in C$	**if** (top = ε / *v*) **then** **write** (*v*) **pop**	write output string *v*
7	for every pair $\frac{b}{w} \in C$	**if** (cur_char = *b* Λ top = *b* / *w*) **then** **write** (*w*) **pop** **shift input head**	read input char *b* write output string *v*
8		**if** (cur_char = -| Λ stack empty) **then** **accept** **halt**	accept and end

EXAMPLE – extend to a transducer the recognizer of the language

L = {$a^*\ a^m\ b^m \mid m > 0$} - τ($a^k\ a^m\ b^m$) = $d^m\ c^k$

	Rule	Move
1	$S \to \frac{a}{\varepsilon} S \frac{\varepsilon}{c}$	**if** (cur_char = *a* Λ top = S) **then pop**; **push** (ε / *c*S); **shift input head**
2	$S \to A$	**if** (top = S) then **pop**; **push** (A)
3	$A \to \frac{a}{d} A \frac{b}{\varepsilon}$	**if** (cur_char = *a* Λ top = A) then **pop**; **write** (*d*); **push** (*b* / εA); **shift input head**
4	$A \to \frac{a}{d} \frac{b}{\varepsilon}$	**if** (cur_char = *a* Λ top = A) then **pop**; **write** (*d*); **push** (*b* / ε); **shift input head**
5	– – –	**if** (top = ε / *c*) then **pop**; **write** (*c*);
6	– – –	**if** (cur_char = *b* Λ top = *b* / ε) then **pop**; **shift input head**
7	– – –	**if** (cur_char = -| Λ empty stack) then **accept**; **halt**

Not every transduction defined by a transduction free grammar can be computed in a deterministic way

EXAMPLE – a inherently non-deterministic transduction
(typical example of something not to do)

No det. pushdown transducer automaton can compute the transduction shown aside

$$\tau(u\text{-}|)=u^R u, \quad u \in \{a,b\}^*$$

A det. transducer should push the string *u* and, soon after reading the terminator, it should pop *u* to output the mirror image. But in this way it looses any information about *u*, and therefore it will not be able to output the direct image of *u* itself

For practical purposes only the deterministic cases are of importance. Suitable analysis algorithms follow

SYNTAX ANALYSIS AND ON-LINE TRANSDUCTION

The transduction can be constructed directly and efficiently. A syntactic analyzer is transformed into the corresponding syntactic transducer

Given a transduction grammar, suppose the underlying source grammar allows the construction of a deterministic syntax analyzer. To compute the transduction, execute the syntax analysis and, as the syntax tree is built, output the corresponding transduction

TOP-DOWN ANALYZER (LL): can always be transformed into a transducer

BOTTOM-UP ANALYZER (LR): to be transformed into a transducer, grammar rules need satisfy a special restrictive condition - the write action can only occur at the end of the production (when the analyzer performs reduction)

TOP-DOWN DETERMINISTIC TRANSDUCTION

If the source grammar is LL(*k*), complete the parser with write actions and obtain the corresponding transducer

This construction is very simple. The transducer can also be designed as a set of recursive syntactic procedures

EXAMPLE: translate a string into the mirror image

The source grammar is LL(1), and the lookahead sets are {*a*} {*b*} and {-|}.

$$S \rightarrow \frac{a}{\varepsilon} S \frac{\varepsilon}{a} \mid \frac{b}{\varepsilon} S \frac{\varepsilon}{b} \mid \varepsilon$$

Automaton moves are:

stack	cc = *a*	cc = *b*	cc = -|	ε
S	pop; push (ε / *a*S)	pop; push(ε / *b*S)	pop	
ε / *a*				write (*a*)
ε / *b*				write (*b*)

EXAMPLE – translate infix expressions to postfix – show recursive version as well

Source language: arithmetic expressions with pharenteses.

The transduction converts the infix form to the postfix form.

Example: v x (v + v) v v v add mult

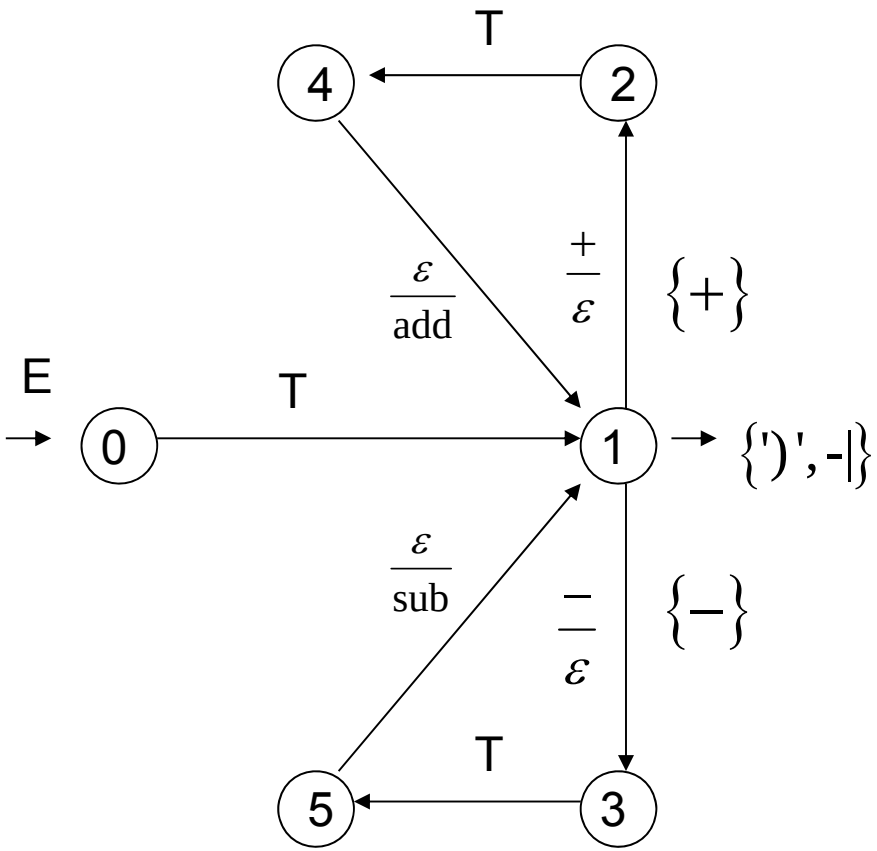
EBNF transduction grammar to postfix form:

$$E \to T\left(\frac{+}{\varepsilon}T\frac{\varepsilon}{\text{add}}\,|\,\frac{-}{\varepsilon}T\frac{\varepsilon}{\text{sub}}\right)^*$$

$$T \to F\left(\frac{\times}{\varepsilon}F\frac{\varepsilon}{\text{mult}}\,|\,\frac{\div}{\varepsilon}F\frac{\varepsilon}{\text{div}}\right)^*$$

$$F \to \frac{v}{v}\,|\,\frac{'('}{\varepsilon}E\frac{')'}{\varepsilon}$$

$$\Sigma = \{+,\times,-,\div,(,),v\},\quad \Delta = \{\text{add},\text{sub},\text{mult},\text{div},\text{v}\}$$



automaton version

```
procedure E
        call T
        while (cc ∈ {+, -}) do
                case (cc = '+') begin
                        cc := next
                        call T
                        write ('add')
                end
                case (cc = '-') begin
                        cc := next
                        call T
                        write ('sub')
                end
                otherwise error
                end case
        end do
end E
```

recursive version
(with syntatctic procedures)