Formal Languages and Compilers

Attribute Grammars

Prof. Stefano Crespi Reghizzi

(translated and adapted by L. Breveglieri)

30th September 2007

Introduction and Preliminaries

Compilation needs functions uncomputable in a purely syntactic way, that is uncomputable by means of devices like the following ones:

- finite state or pushdown automaton, with two input tapes or one input and one output tape
- free transduction grammar (or equivalently syntactic transduction scheme)

Example: translate (one usually says convert) a fractional number in fixed point notation from base 2 to base 10.

Example: translate a complex data structure (e.g. record or struct) and compute the memory address to locate each data structure element at (e.g. record or struct internal fields).

Example: construct the symbol table of a data structure (e.g. record in Pascal syntax):

```
BOOK: record
  AUTHOR:   array [1..8] of char;
  TITLE:    array [1..20] of char;
  PRICE:    real;
  QUANTITY: integer;
end;
```

symbol	*type*	*size* (in bytes)	*address* (in bytes)
`BOOK`	record	34	3401
`AUTHOR`	string	8	3401
`TITLE`	string	20	3409
`PRICE`	real	4	3428
`QUANTITY`	integer	2	3432

Obviously the translation need involve arithmetic functions, to compute memory addresses.

Syntax-driven transducer:

- It is a device that uses functions working on the syntax tree and computes variables or *semantic attributes*.

- The values of the attributes constitute the translation of the source phrase or equivalently they express the *meaning* (*semantic*).

- Attribute grammars are not *formal models* (or are only partially formal models), as the procedures that compute the attributes are programs that are not entirely formalized.

- Rather, attribute grammars are a *viable* and *practical engineering methodology* to design compilers in an ordered and coherent way, and to avoid bad or unhappy choices.

Two-pass compilation:

1. *Parsing* or *syntax analysis*

    $\Rightarrow$ *abstract* syntax tree.

2. *Evaluation* or *semantic analysis*

    $\Rightarrow$ *decorated* syntax tree.

Example: convert from base 2 to base 10.

$$\text{Source lang.:} \qquad L = \{0,1\}^* \bullet \{0,1\}^*$$

The bullet '$\bullet$' separates the integer and fractional parts of the source number (conceived as a string over the alphabet $\Sigma = \{0, 1, \bullet\}$).

The meaning (or semantic, or translation) of the source string $1101 \bullet 01_{two}$ (in base two) is $13, 25_{ten}$ (in base ten).

Attribute grammar:

syntax	*semantic functions*	
$N \rightarrow D \bullet D$	$v_0 := v_1 + v_2 \times 2^{-l_2}$	
$D \rightarrow DB$	$v_0 := 2 \times v_1 + v_2$	$l_0 := l_1 + 1$
$D \rightarrow B$	$v_0 := v_1$	$l_0 := 1$
$B \rightarrow 0$	$v_0 := 0$	
$B \rightarrow 1$	$v_0 := 1$	

Consists of syntax (production) rules, paired to auxiliary semantic rules (semantic functions).

Attributes and interpretation:

name	*interpretation*	*domain*	*assoc. nonterm.*
v	value	frac. num.	N, D, B
l	length	int. num.	D

Each semantic function is associated with a production rule (which is said to be the *syntactic support* of the function). A rule may support none, one, two or more semantic functions.

Pedices v_0, v_1, v_2, l_0 and l_2 specify each production symbol (i.e. nonterminal) every attribute should be associated with:

$$\underbrace{N}_{0} \rightarrow \underbrace{D}_{1} \bullet \underbrace{D}_{2}$$

Function $v_0 := \ldots$ assignes v_0 the value of expr. $\ldots$ containing the arguments v_1, v_2 and l_2.

For instance: $v_0 := f\,(v_1, v_2, l_2) = v_1 + v_2 \times 2^{-l_2}$.

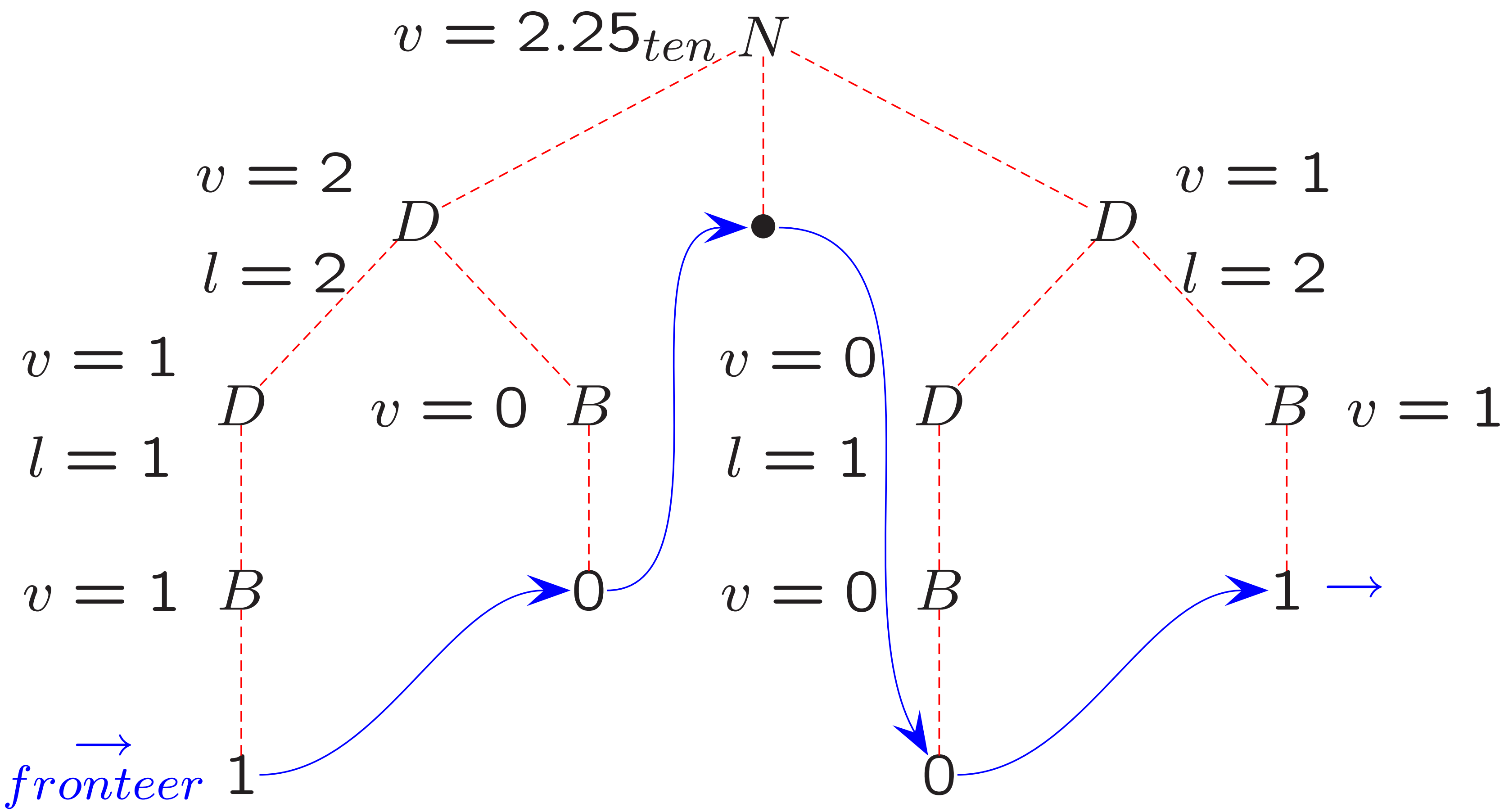
Here follows the same attribute grammar as before, denoted in a complete way:

syntax	*semantic functions*	
$N_0 \rightarrow D_1 \bullet D_2$	$v_0 := v_1 + v_2 \times 2^{-l_2}$	
$D_0 \rightarrow D_1 B_2$	$v_0 := 2 \times v_1 + v_2$	$l_0 := l_1 + 1$
$D_0 \rightarrow B_1$	$v_0 := v_1$	$l_0 := 1$
$B_0 \rightarrow 0$	$v_0 := 0$	
$B_0 \rightarrow 1$	$v_0 := 1$	

Given a syntax tree, apply a semantic function to each node and start from the nodes the arguments of which are known; usually these are terminal nodes, that is the leaves of the tree.

In this way one obtains a *decorated* syntax tree, which represents the translation of the given source string (one can still read the string on the tree fronteer). Here follows an example:

syntax tree decorated with attributes



Attributes are of two types: *left* (or *synthesized*) and *right* (or *inherited*):

- left $\Rightarrow$ associated with *parent* node D_0

- right $\Rightarrow$ associated with *child* node D_i $(i \geqslant 1)$

In the above base conversion example, all the attributes are of the left type (is a simple case).

A more structured and complex example

Problem: how to instrument ($=$ justify) a unformatted text piece (in natural language) to have lines of length $\leqslant W$ characters (W is a constant).

The text is a list of words, separated by one blank (represented by $\perp$); the terminal symbol c represents a generic character (non-blank).

The most meaningful attribute is $ultimo$ ($= last$) (short form ult): it indicates the column number (starting from 1, leftmost column) where the last (rightmost) letter of a word is located.

Consider the following phrase*:

“la	torta	ha	gusto	ma	la	grappa	ha	forza”
“the	pie	has	taste	but	the	eau-de-vie	has	strength”

and set the bound $W = 13$ (max line length).

*Pie is tasteful but “eau de vie” gives energy.

Correctly instrumented text:

1	2	3	4	5	6	7	8	9	10	11	12	13
l	a		t	o	r	t	a		h	a		
g	u	s	t	o		m	a		l	a		
g	r	a	p	p	a		h	a				
f	o	r	z	a								

Attribute $ultimo$ has value 2 and 5 for the two words 'la' and forza', respectively.

Attributes and interpretation:

- lun ($= length$) is left and expresses the *length* of the current word, measured in characters
- pre ($= previous$) is right and expresses the *column number* of the *last* (rightmost) character of the word that *precedes* immediately the current one
- ult is left and expresses the *column number* of the *last* character of the current word

This yields the following semantic function:

$$ult(w_k) := pre(w_{k-1}) + 1 + lun(w_k)$$

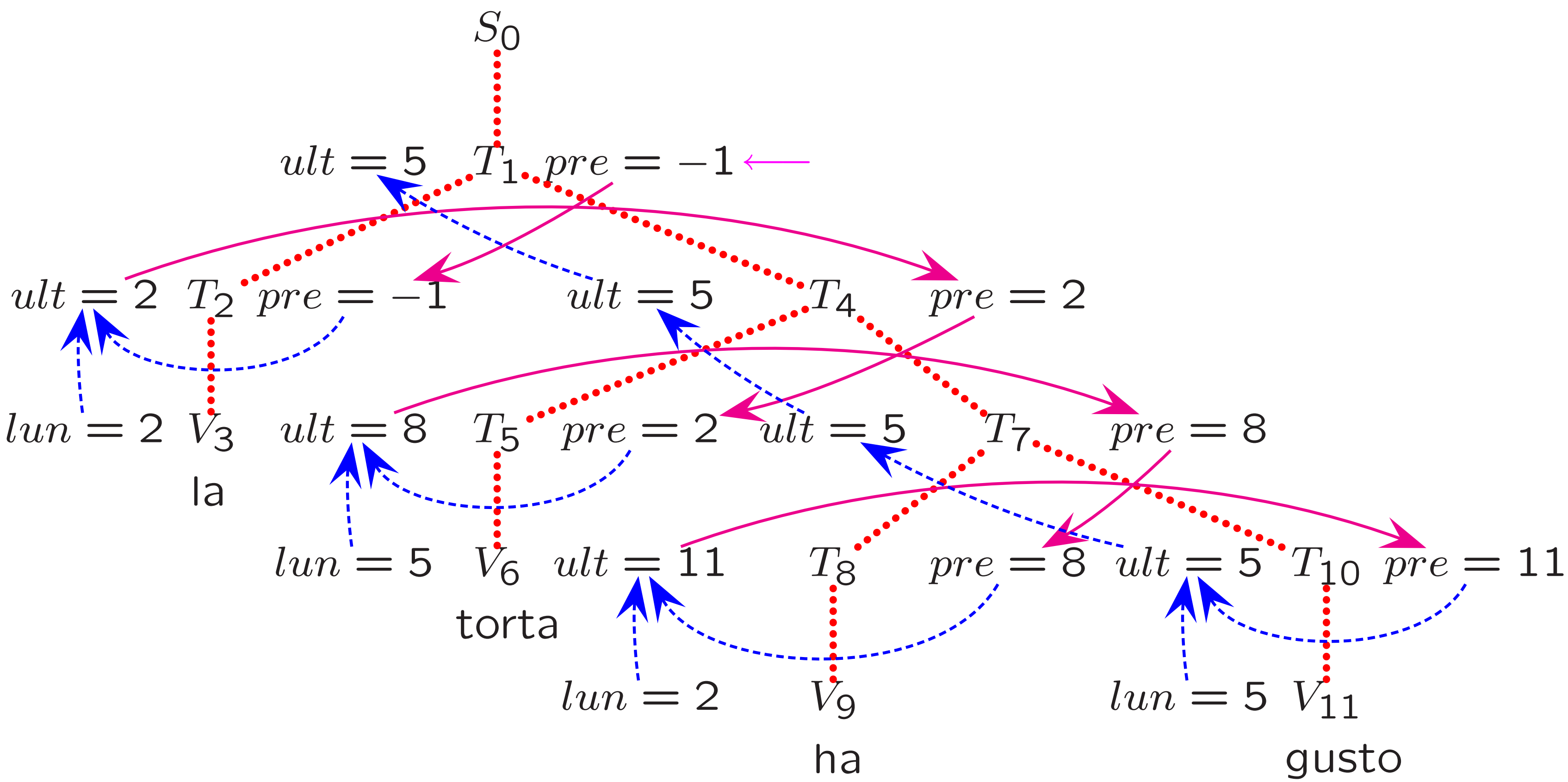
where $k \geqslant 1$. Start from the initial word w_1 and evaluate the function by sliding over the text, word by word. Set $pre(w_0) = -1$ to compensate for the constant term $+1$, as there must not be any leading blank separator on the left of w_1.

Grammar and semantic functions:

syntax	*semantic functions*	
	right attributes	*left attributes*
1: $S_0 \to T_1$	$pre_1 := -1$	
2: $T_0 \to T_1 \perp T_2$	$pre_1 := pre_0$ $pre_2 := ult_1$	$ult_0 := ult_2$
3: $T_0 \to V_1$		$ult_0 :=$ **if** $(pre_0 + 1 + lun_1 \leqslant W)$ **then** $(pre_0 + 1 + lun_1)$ **else** (lun_1) **end if**
4: $V_0 \to c\, V_1$		$lun_0 := lun_1 + 1$
5: $V_0 \to c$		$lun_0 := 1$

The syntactic support is *ambiguous* (due to rule $T \to T \bot T$, which contains two-sided recursion), but this is not a drawback; the semantic evaluator (which slides over the text and computes the semantic function above) will work on *one* syntax tree, of the many possible trees generating the same text piece, chosen in some arbitrary way which is not necessary to specify here.

Dependence graph of the semantic functions:



Notational conventions to draw the graph:

- Dashed edges: syntactic relations.
- Solid arcs: computation of pre.
- Dashed arcs: computation of ult.

Moreover, place right (pre) and left (lun, ult) attributes to the right and left side of the corresponding tree node, respectively.

The dependence graph is *loop-free* (acyclic).

Any computation order that satisfies all the dependences, allows to determine the values of the attributes.

If the grammar satisfies certain conditions (to be explained later on), results do not depend on the computation order of the semantic functions.

Definition of attribute grammar

1. Let $G = (V, \Sigma, P, S)$ be a syntax, where V and Σ are the nonterminal and terminal sets, respectively, P is the rule set and S is the axiom. Suppose that the axiom is not referenced anywhere in the right parts of the rules and that the axiomatic production is unique (both assumptions can be always made effective).

2. Define a set of *semantic attributes*, associated with nonterminal and terminal symbols. The attributes associated with a nonterminal symbol D are denoted by α, β, ... (Greek letters), and are grouped in the subset $attr(D) = \{\alpha, \beta, \ldots\}$. The set of all the attributes is divided into two disjoint subsets: *left* (or *synthesized*) *attributes*, e.g. σ, and *right* (or *inherited*) *attributes*, e.g. δ or η.

3. For every attribute (be it left or right) specify a domain, i.e. a finite or infinite set of possible attribute values. An attribute can be associated with one, two or more (non)terminal symbols. Suppose attribute $\alpha \in attr(D_i)$ is associated with symbol D_i, then write α_i with a pedex i. If there is not any danger of confusion, freely write “α of D”, “α_D” or in a similar way (see the conventions before).

4. Define a set of *semantic functions* (or *semantic rules*). Each semantic function is associated with a support syntax rule p:

$$p\colon\ D_0 \rightarrow D_1 D_2 \ldots D_r \qquad\qquad r \geqslant 1$$

Two or more semantic functions may share the same syntactic support rule. The set of all the functions associated with a given support rule p, is denoted as $fun\,(p)$ (may be empty).

5. A generic semantic function, as follows:

$$\alpha_k := f\,(attr(\{D_0, D_1, \ldots, D_r\} \setminus \{\alpha_k\}))$$

where $0 \leqslant k \leqslant r$, assigns attribute α_k (α of D_k) a value by means of an expression f, the operands of which are the attributes associated with the *same* support syntax rule p (not another one), excluding the expression value itself (α_k). Semantic functions must be *total*.

6. Semantic functions are denoted by means of a suited *semantic metalanguage*:

   - often a common use programming language (C or Pascal)
   - sometimes a pseudocode (informal)
   - or it may be even a standardized software specification language (XML and others)

7. Models of semantic functions for computing left and right attributes of rule p, respectively:

- $\sigma_0 := f(\ldots)$ defines a *left* attribute, associated with the parent node D_0
- $\delta_i := f(\ldots)$ $(1 \leqslant i \leqslant r)$ defines a *right* attribute, associated with a child node D_i

8. Attribute associated with a terminal symbol:

- is always of the right (inherited) type
- is often directly assigned a constant value during lexical analysis (before semantic analysis), a semantic function is seldom used
- and commonly is directly assigned the terminal symbol itself it is associated with

9. The elements of the set $fun(p)$ of the semantic functions that share the same support rule p, must satisfy the following conditions:

(a) for every left attribute σ_i it holds:

- if $i = 0$ *there exists one*, and only one, *defining semantic function*: $\exists!\ (\sigma_0 := f\,(\ldots)) \in fun(p)$
- if $1 \leqslant i \leqslant r$ *there does not exist any defining semantic function*: $\nexists\ (\sigma_i := f\,(\ldots)) \in fun(p)$

(b) for every right attribute δ_i it holds:

- if $1 \leqslant i \leqslant r$ *there exists one*, and only one, *defining semantic function*: $\exists!\ (\delta_i := f(\ldots)) \in fun(p)$
- if $i = 0$ *there does not exist any defining semantic function*: $\nexists\ (\delta_0 := f(\ldots)) \in fun(p)$

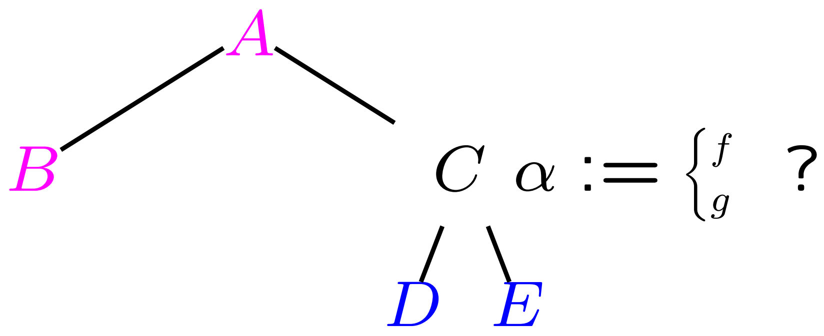
Conclusion: if σ is left never have $\sigma_i := \ldots$ $(i \neq 0)$ and if δ is right never have $\delta_0 := \ldots$.

11. It is permitted to initialize some attributes with constant values or with values computed initially by means of external functions.

This is indeed the case for the attributes (always of the right type) that are associated with the terminal symbols of the grammar.

Uniqueness of definition: an attribute α *may not* be defined as both left and right, lest on the syntax there may be two conflicting assignments to the same attribute α. For instance:

	support	*semantic function*
1 :	$A \rightarrow BC$	- - right attribute $\alpha_C := f(attr(A, B))$
2 :	$C \rightarrow DE$	- - left attribute $\alpha_C := g(attr(D, E))$



Locality principle of semantic functions:

Error: set as operand or result of a semantic function, with support rule p, an attribute that is *external* to the rule p itself.

Example: change rule 2 of the previous example (text instrumentation) and obtain what follows:

	syntax	*semantic functions*
1:	$S_0 \rightarrow T_1$	$\ldots$
2:	$T_0 \rightarrow T_1 \perp T_2$	$pre_1 := pre_0 \quad + \quad \underbrace{lun_0}_{\text{non-local attr.}}$
3:	$\ldots$	

By definition attribute lun is associated only with nonterminal V; but V does not occur in rule 2 and hence the locality condition is broken. Violating locality causes the association of attributes with symbols to get confused.

Construction of the semantic evaluator

The semantic evaluator is an attribute grammar specifying the translation but not the appropriate computation order of the attributes, which can be inferred by the evaluator itself.
The procedure to compute the attributes will be designed (automatically or manually by the designer himself), according to the function dependences among the attributes.

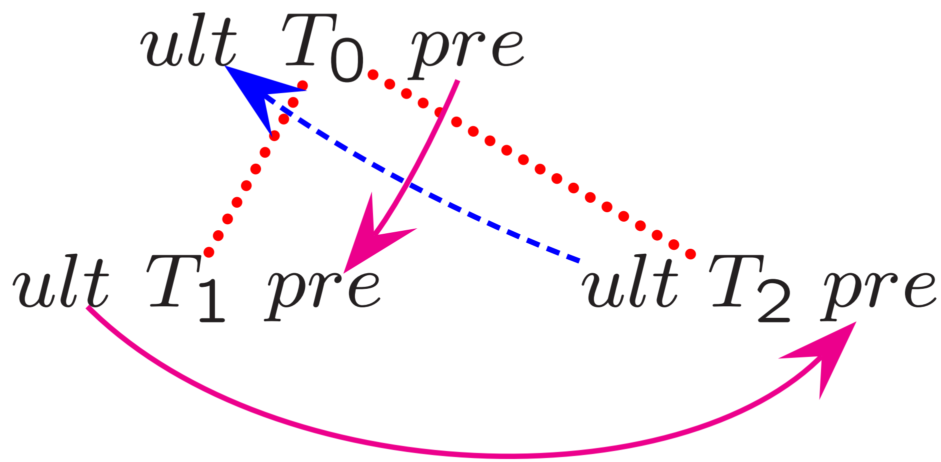
Dependence graph of a semantic function:

The dependence graph of a semantic function is directed (nodes, arcs) and is wrapped on the (elementary) syntax tree of the support rule:

- write the *left* (*synthesized*) and *right* (*inherited*) attributes on the *left* and *right* side of the (non)terminal node, respectively
- place an arc from each argument to the result

Wrap the dependence graph onto the syntax support (and possibly omit terminal symbols). Here follows an example for rule 2 before:

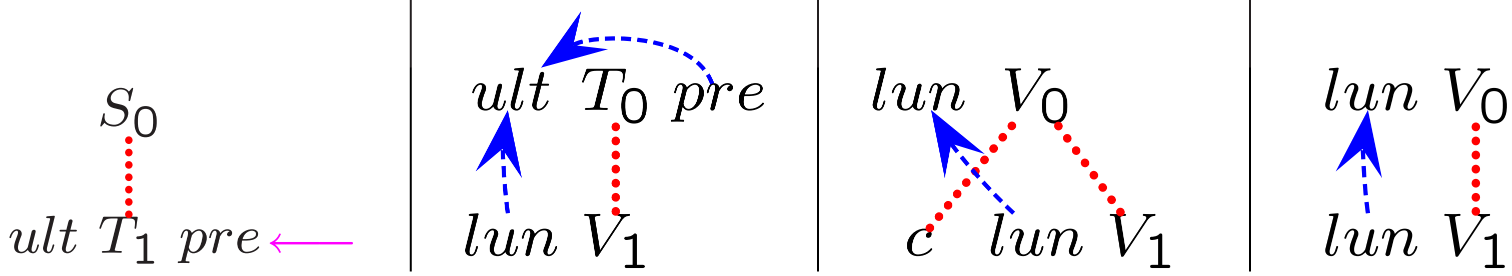
$$\begin{aligned} 2\colon T_0 &\;\rightarrow\; T_1 \perp T_2 \\ ult_0 &:= f_1\,(ult_2) \\ pre_1 &:= f_2\,(pre_0) \\ pre_2 &:= f_3\,(ult_1) \end{aligned}$$



left attr. upward arrows - right attr. downwards or sidewards - for brevity terminal $\perp$ is omitted

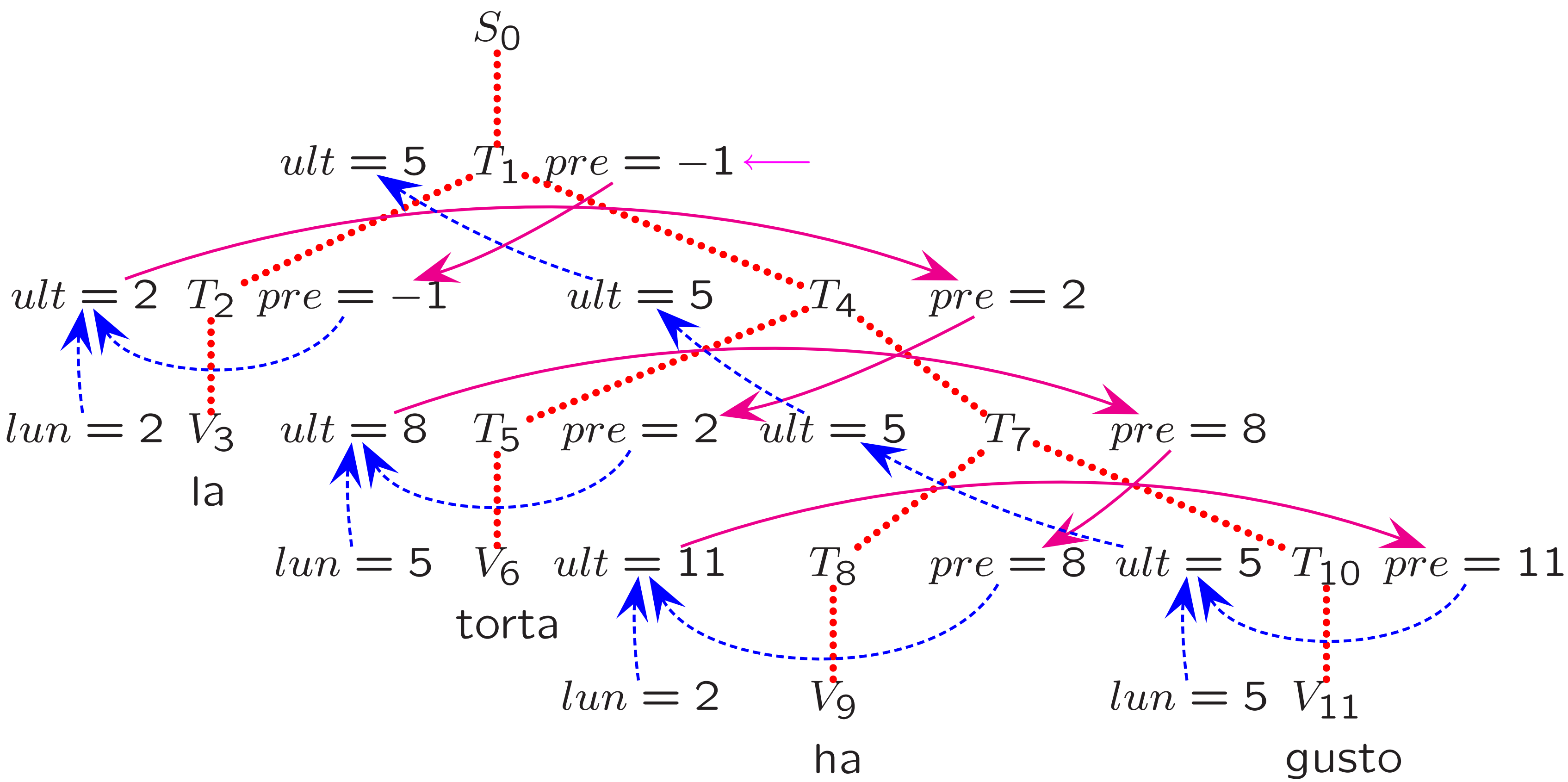
Other syntax rules of the same grammar:

$S_0 \to T_1$	$T_0 \to V_1$	$V_0 \to c\, V_1$	$V_0 \to V_1$



Names with and without incoming arcs indicate attributes that are internal and external to the current syntax rule, respectively.

full tree decorated with dependence graph



Solution existence and uniqueness

If the dependence graph of an attribute grammar is loop-free (acyclic), there exists a unique assignment of values to the attributes which is conformant to the tree dependence thread.

An attribute grammar is said to be itself *loop-free* (*acyclic*) if every syntax tree has a loop-free dependence graph.

Hypothesis: suppose the attribute grammar is always loop-free (see next how to ensure so).

First examine how to sort linearly the assignments to attributes, so that each assignment statement is executed after those computing the arguments needed to evaluate the semantic function contained in the statement itself.

Algorithm of graph topological sorting:

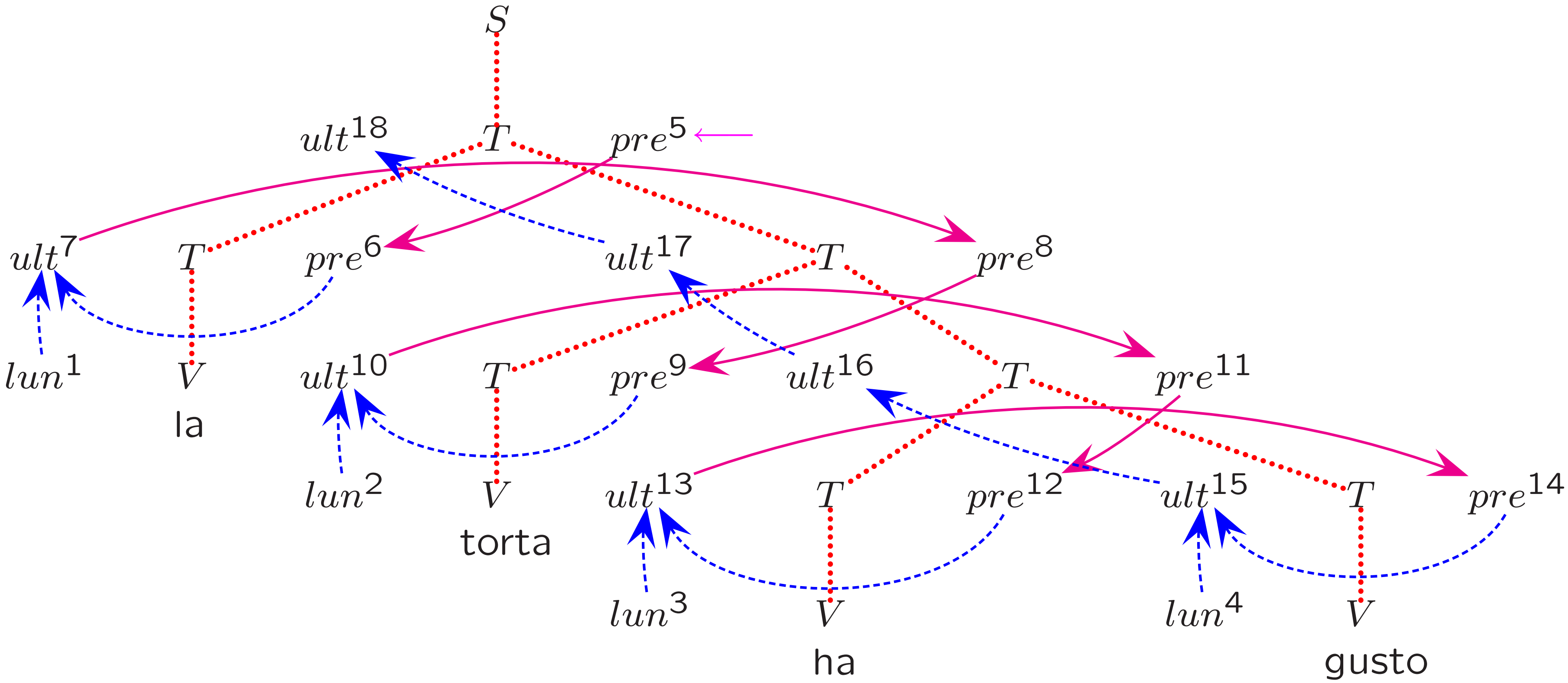
Let $G = (V, E)$ be a loop-free graph, with nodes labeled numerically, i.e. $V = \{1, 2, \ldots, |V|\}$.

The algorithm computes a linear ordering of all the nodes in the graph: *topological sorting*.

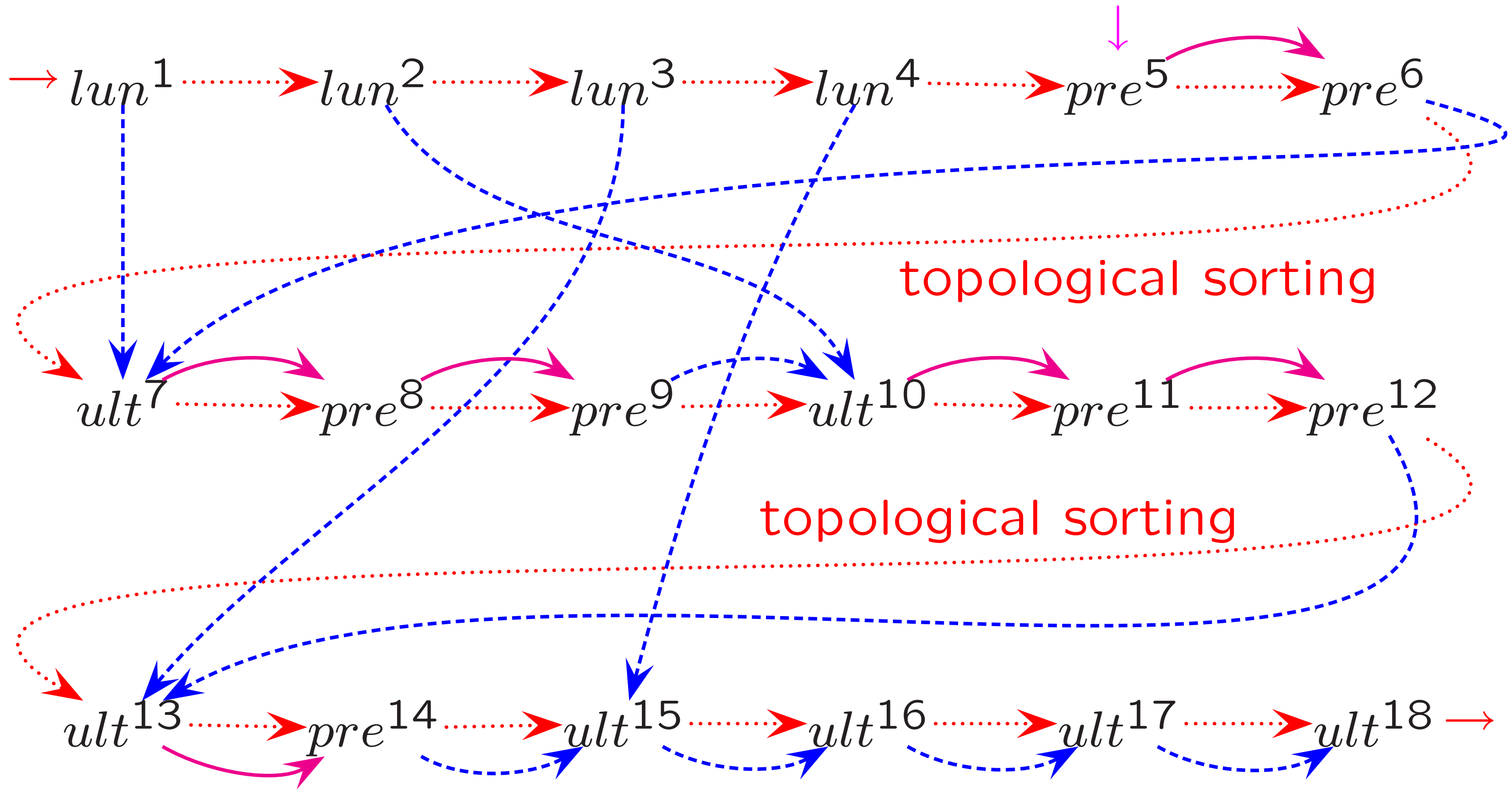
Data structure $ord\,[|V|]$ is the vector of topologically sorted nodes.

Element $ord\,[i]$ is a number that indicates the sorting position of the node labeled by i.

topological sorting for attribute evaluation



display in line ...



... and all the arrows are directed forwards !

There exists a general decision algorithm to check whether an attribute grammar is loop-free, but is computationally complex; as a matter of fact it is NP-complete, and so far to be computed it takes exponential time in the grammar size.

However the acyclicity property for the attribute grammar is a practical necessity: to circumvent this difficult decision problem, some suited sufficient conditions are given to design the node scheduling in such a way as to avoid *ipso facto* (= by definition) dependence loops in the attribute assignments of the grammar.

One-sweep semantic evaluation

A fast semantic evaluator should schedule the tree nodes in such a way as to access each node *only once* and simultaneously should compute and assign values to the associated attributes.

A semantic evaluator of the above type is said to be *one-sweep* (= *access each node only once*); the concept is similar to *real-time* computing.

In-depth tree sorting:

1. Start from the tree root (grammar axiom).

2. Let N be an internal tree node and let N_1, ..., N_r $(r \geqslant 1)$ be the child nodes of N. To schedule the subtree t_N rooted at node N, proceed recursively as follows:

(a) schedule all the subtrees t_1, t_2, ..., t_r in the in-depth way, not necessarily following the natural numerical order 1, 2, ..., r, but possibly a permutation thereof

(b) before scheduling and evaluating subtree t_N, compute the right (inherited) attributes associated with node N

(c) after scheduling and evaluating subtree t_N, compute the left (synthesized) attributes associated with node N

Caution: not every grammar is one-sweep and allows to evaluate all the attributes by accessing each of them only once.

There exist dependence threads requiring a sorting different from the in-depth one.

Compatibility conditions between in-depth sorting and one-sweep evaluation

Such (sufficient) conditions should be verifiable in a *fast* and *local* way on the elementary dependence graph dip_p of each support rule p.

If the conditions are implemented when the grammar is designed, much effort is avoided later.

In this way, designing a one-sweep semantic evaluator is an affordable and effective task.

Define the *brother graph* bro_p: it is a binary relation over the nonterminal symbols D_i ($i \geqslant 1$) of the grammar.

Given the support rule p: $D_0 \rightarrow D_1 D_2 \ldots D_r$ ($r \geqslant 1$), the nodes of bro_p are the nonterminal symbols occurring in the right part of rule p, that is symbols $\{D_1, D_2, \ldots, D_r\}$.

In bro_p there is arc $D_i \to D_j$ ($i \neq j$ and $i, j \geqslant 1$), if and only if in dip_p there is arc $\alpha_i \to \beta_j$ from any attribute α of D_i to any attribute β of D_j.

Caution: the nodes of bro_p are nonterminal symbols of the support grammar, not semantic attributes.

Therefore all the attributes of dip_p that have the same pedex j merge into node D_j of bro_p.

Graph bro_p is a homomorphic image[†] of graph dip_p, as the former is obtained by merging nodes of the latter.

[†]The image of a function (called *morphism*) with the property of mapping connected nodes of dip_p to connected nodes of bro_p.

Existence conditions of one-sweep grammar

$$\forall\, p\colon\ \ D_0 \rightarrow D_1 D_2 \ldots D_r \qquad\qquad r \geqslant 1$$

1. Graph dip_p is loop-free.
2. Graph dip_p does not contain any path $\sigma_i \rightarrow \ldots \rightarrow \delta_i$ $(i \geqslant 1)$ from a left attribute σ_i to a right attribute δ_i, both associated with the same node (nonterminal symbol) D_i of the right part of support rule p.

3. Graph dip_p does not contain any arc $\sigma_0 \rightarrow \delta_i$ ($i \geqslant 1$) from a left attribute associated with the parent node D_0 of p to any right attribute associated with a child node D_i of p.

4. And finally graph bro_p is loop-free as well.

Design algorithm of one-sweep evaluator

For each nonterminal symbol, design a *semantic procedure* with the following input parameters:

- the subtree rooted at the symbol
- the right attributes of the subtree root

The semantic procedure schedules the subtree, computes the attributes and returns the left attributes associated with the subtree root.

Here follows the list of the construction steps to design the semantic evaluation procedure:

$$\forall\, p\colon \quad D_0 \to D_1 D_2 \ldots D_r \qquad\qquad r \geqslant 1$$

1. Find a Topological Sorting of the nonterminal symbols D_1, D_2, ..., D_r with respect to the Brother graph bro_p, and name it TSB (Topological Sorting of Brothers).

2. For every symbol D_i $(1 \leqslant i \leqslant r)$, find a Topological Sorting of the right attributes of the child nonterminal symbol D_i itself, and name it TSD (Topological Sorting of D_i).

3. Find a Topological Sorting of the Left attributes of the parent nonterminal symbol D_0, and name it TSL (Topological Sorting Left).

The three topological sortings TSB, TSD and TSL will determine the sequence of statements constituting the execution body of the semantic procedure.

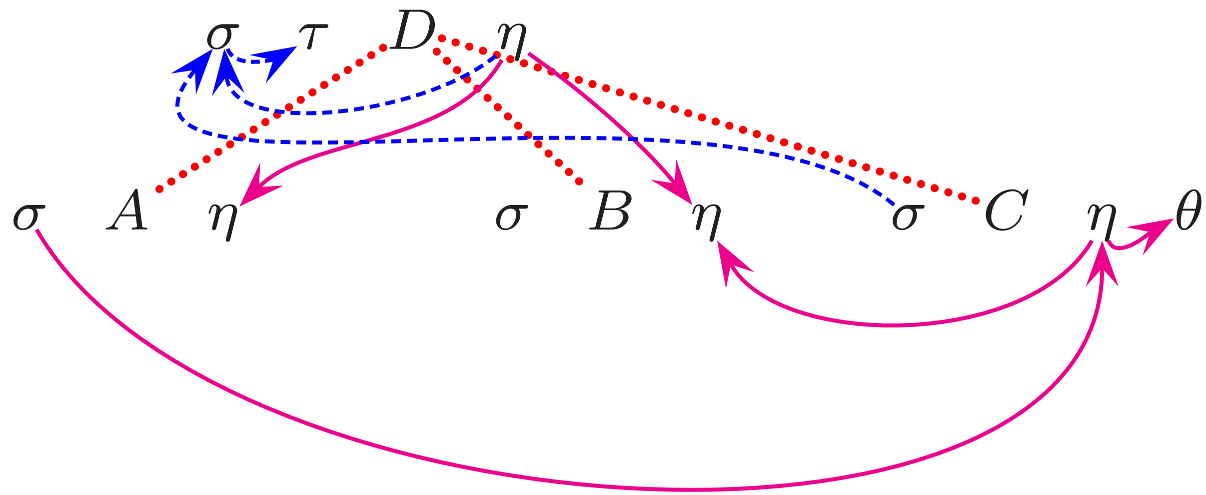
These steps must be repeated for each semantic procedure to design, that is for each nonterminal symbol of the grammar.

Example of one-sweep semantic procedure

Support rule p *and dependence graph* dip_p:

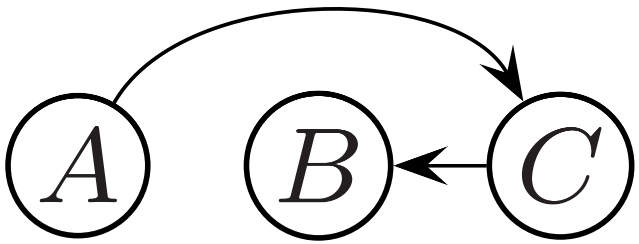
support rule

p: $D \rightarrow ABC$



Graph dip_p satisfies points 1, 2 and 3 of the one-sweep compound condition stated before.

The brother graph bro_p is loop-free:



The arcs of the graph are obtained as follows:

$$A \rightarrow C \quad \text{from dependence} \quad \sigma_A \rightarrow \eta_C$$

$$C \rightarrow B \quad \text{from dependence} \quad \eta_C \rightarrow \eta_B$$

Therefore the last point 4 of the one-sweep compound condition stated before is satisfied as well.

Possible topological sortings:

- brother graph: $TSB = A$, C, B
- right attributes of each child node:
  - TSD of $A = \eta$ (there is only one attr.)
  - TSD of $B = \eta$ (there is only one attr.)
  - TSD of $C = \eta$, θ
- left attributes: TSL of $D = \sigma$, τ

Semantic procedure of support rule p: $D \rightarrow ABC$

```
procedure D (in t_D, η_D; out σ_D, τ_D)

var η_A, σ_A, η_B, σ_B, η_C, θ_C      -- local attribute variables for parameter passing

begin                                  -- start and input tree t_D and right attribute η of D
    η_A := f_1(η_D)                    -- by TSD of A compute right attribute η of A
    A (t_A, η_A; σ_A)                  -- by TSB call A and decorate subtree t_A rooted at A

    η_C := f_2(σ_A)                    -- by TSD of C compute right attribute η of C
    θ_C := f_3(η_C)                    -- by TSD of C compute right attribute θ of C
    C (t_C, η_C, θ_C; σ_C)             -- by TSB call C and decorate subtree t_C rooted at C

    η_B := f_4(η_D, η_C)               -- by TSD of B compute right attribute η of B
    B (t_B, η_B; σ_B)                  -- by TSB call B and decorate subtree t_B rooted at B

    σ_D := f_5(η_D; σ_C)               -- by TSL compute left attribute σ of D
    τ_D := f_6(σ_D)                    -- by TSL compute left attribute τ of D
end                                    -- output left attributes σ and τ of D and stop
```

Justification with prefix-postfix arrangement

Sort linearly the nodes from left to right according to the brother graph of the current rule and interleave the attribute names as follows:

- inherited, on the left side of the reference node name
- synthesized, on the right side of the child nodes, grandchild nodes, etc (i.e. all the subtree nodes), of the reference parent node name

This is to say that the *inherited* and *synthesized* attributes are conceived as *prefix* and *postfix* operators applied to the nodes, respectively.

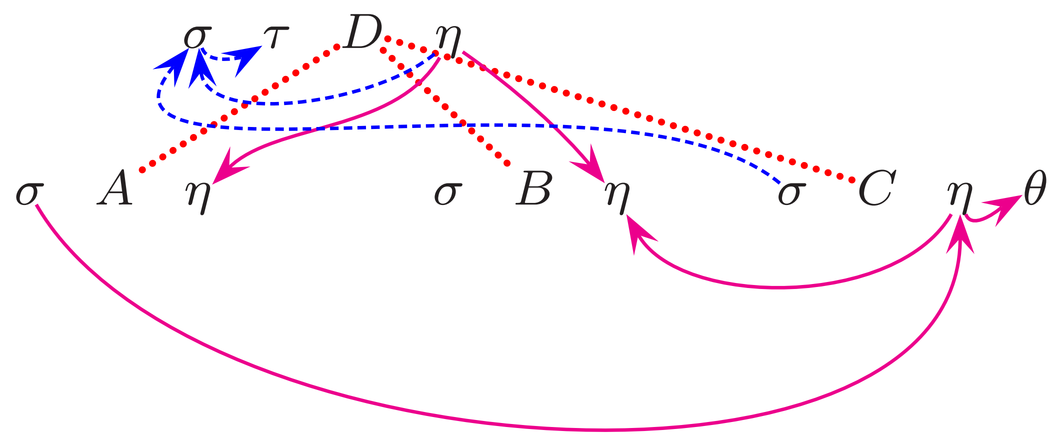
Here follows an example (the same as before):

support rule

$$D \rightarrow ABC$$

scheduling

$$D,\ A,\ C,\ B$$



The linear prefix-postfix order is the following:

$$\eta_D \quad D \quad \eta_A \quad A \quad \underbrace{\sigma_A \quad \eta_C \quad \theta_C \quad C \quad \sigma_C \quad \eta_B \quad B \quad \sigma_B}_{\text{child nodes, etc, of } D} \quad \sigma_D \quad \tau_D$$

and expresses the attribute computation order.

Next place all the dependence arrows between the linearly sorted attributes. As the inherited and synthesized attributes are computed soon after arriving at the node and soon before departing from it (and hence after decorating all the appended subtrees), respectively, the one-sweep condition is satisfied if and only if:

all dependence arrows are directed rightwards rule p: $D \to ABC$ - node scheduling D, A, C, B

$$\eta_D \quad D \quad \eta_A \quad A \quad \sigma_A \quad \eta_C \quad \theta_C \quad C \quad \sigma_C \quad \eta_B \quad B \quad \sigma_B \quad \sigma_D \quad \tau_D$$

Dependence arrows are directed rightwards, hence scanning and computing the attributes from left to right is conformant to all the dependences.

How to merge syntax and semantic analysis

If semantic evaluation could be executed directly by the syntax parser, merging (or integrating) syntax and semantic analysis into one procedure would prove to be a very efficient methodology.

This methododology fits well to situations that are not too complex (which is often the case).

Recursive descent parser with attributes

Requires some hypotheses conceived "ad hoc":

- support syntax is of type LL(k) ($k \geqslant 1$)
- attribute grammar is of one-sweep type
- moreover, the dependences among attributes must satisfy some suited *supplementary restrictions* (part 2 of L-condition, see next)

A generic one-sweep evaluator schedules the subtrees t_1, ..., t_r ($r \geqslant 1$), associated with the support syntax rule p: $D_0 \longrightarrow D_1 \ldots D_r$, and follows an order that need not necessarily be that of natural integers, i.e. 1, 2, $\ldots$, $r-1$, r.

However the chosen scheduling order is topological, so as to be compatible with the function dependences of the attributes of nodes 1, ..., r.

Instead, the recursive descent parser builds the syntax tree in the natural in-depth order.

This means that subtree t_j $(1 \leqslant j \leqslant r)$ is constructed after building all the subtrees t_1, t_2, ..., t_{j-2}, t_{j-1} (which are the left brothers of t_j).

It follows that all the function dependences forcing to schedule the subtrees according to some permutation of the natural sorting 1, 2, ..., $r-1$, r, are forbidden and must be avoided.

L-condition (Left) for recursive descent

$$\forall\, p\colon\ \ D_0 \to D_1 \ldots D_r \qquad\qquad r \geqslant 1$$

1. the one-sweep compound cond. is satisfied

2. the brother graph bro_p does not contain any arc between nodes of the following type

$$D_j \to D_i \qquad\qquad \text{where } j > i$$

Property: if an attribute grammar is such that:

- syntax satisfies condition LL(k) ($k \geqslant 1$)
- semantic functions satisfy L-condition

one can obtain a deterministic recursive descent syntax parser that embeds a semantic evaluator of the attributes (i.e. one has a semantic evaluator integrated with a syntax support parser)

Example of recursive descent integrated syntax and semantic analyzer

The analyzer converts a positive or null fractional number < 1 (in fixed point notation) from base 2 (binary) to base 10 (decimal).

Source language: $L = \bullet(0 \mid 1)^*$

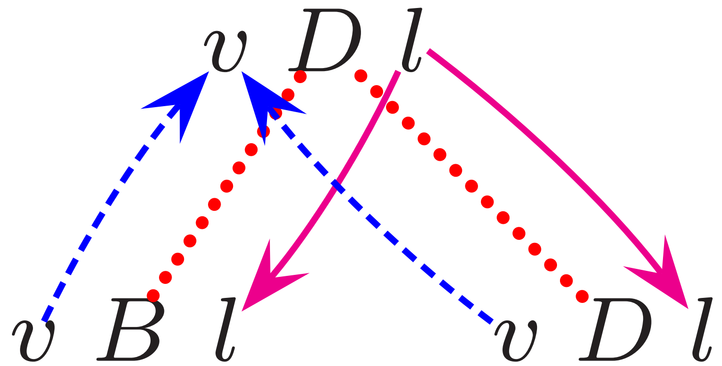
Translation sample: $\bullet 01_{two} \Rightarrow 0.25_{ten}$

Attribute grammar and semantic functions:

syntax	*semantic functions*		
$N_0 \to \bullet D_1$	$v_0 := v_1$	$l_1 := 1$	
$D_0 \to B_1 D_2$	$v_0 := v_1 + v_2$	$l_1 := l_0$	$l_2 := l_0 + 1$
$D_0 \to B_1$	$v_0 := v_1$	$l_1 := l_0$	
$B_0 \to 0$	$v_0 := 0$		
$B_0 \to 1$	$v_0 := 2^{-l_0}$		

The two grammar attributes v (left) and l (right) are associated with the two groups of nonterminal symbols $\{N, D, B\}$ and $\{D, B\}$, respectively.

- $D \to BD$ - the dependence graph dip of this rule is the following:



where the solid and dashed arrows point to the right and left attributes l and v, respectively. Hence:

- the dependence graph dip is loop-free
- there is not any path from the left attribute v to the right attribute l associated with the same child node
- there is not any arc from the left attribute v associated with the parent node to the right attribute v associated with a child node
- the brother graph bro does not have any arc

Semantic procedure:

- input parameters: parent right attributes
- output parameters: parent left attributes
- local variables: $cc1$ and $cc2$ are the *current* and *next* terminal symbols, respectively (together contain the lookahead window of width 2), and there are a few local variables to pass attributes to the inner calls of the other semantic procedures (those of the child nodes)

The system call "`read`" updates variables $cc1$ and $cc2$ (shifts lookahead window one position rightwards) - syntax is LL(2) but not LL(1):

```
procedure N (in ∅; out v0)

var l1                     - - local variables to pass attributes

begin                      - - start and input parameters
    if (cc1 = '•')         - - check lookahead (depth 1)
        then  read         - - shift lookahead window
        else  error        - - error case (warn or stop)
    end if
    l1 := 1                - - compute attribute l1 of D
    D (l1; v0)             - - pass parameters and call D
end                        - - output parameters and stop
```

```
procedure D (in l0; out v0)

var v1, v2, l2                        - - local variables to pass attributes

begin                                 - - start and input parameters
    case cc2 of                       - - check lookahead (depth 2)
        '0','1': begin                - - case of alternative D → BD
            B (l0; v1)                - - pass parameters and call B
            l2 := l0 + 1              - - compute attribute l2 of B
            D (l2; v2)                - - pass parameters and call D
            v0 := v1 + v2             - - compute attribute v0 of D
        end
        '⊣': begin                    - - case of alternative D → B
            B (l0; v1)                - - pass parameters and call B
            v0 := v1                  - - compute attribute v0 of D
        end
        otherwise  error              - - error case (warn or stop)
    end case
end                                   - - output parameters and stop
```

```
procedure B (in l_0; out v_0)

begin                                -- start and input parameters
    case cc1 of                      -- check lookahead (depth 1)
        '0': begin                   -- case of alternative B → 0
            read                     -- shift lookahead window
            v_0 := 0                 -- reset attribute v of B
        end
        '1': begin                   -- case of alternative B → 1
            read                     -- shift lookahead window
            v_0 := 2^(-l_0)          -- compute attribute v of B
        end
        otherwise  error             -- error case (warn or stop)
    end case
end                                  -- output parameters and stop
```

Call the axiomatic procedure and run the program. Various obvious optimizatios are possible.

Typical applications of attribute grammars

- Semantic check (e.g. type checking).
- Code generation (e.g. assembly code).
- Semantic driven syntax analysis.

Generation of machine code

The generation of machine code can be a more or less difficult task, depending on the semantic distance between the source (high level) and destination (low level or object) languages.

Generating correct and efficient machine code for a modern processor is a challenging problem.

Multi-pass (or stage) language translation:

- each stage translates an *intermediate* language to another one, closer to the final form
- the first stage (parser) inputs the source language (e.g. C or Java)
- the last stage (code generator) outputs the destination language (e.g. the assembly language of the processor)

Review of possible intermediate languages:

- operatorial languages in polish notation
- description languages for trees or graphs
- or instruction languages in assembly style

First stage (*front-end*): usually is a syntax driven transducer (e.g. for the Java language).
The final stages select the machine instructions to use and try to optimize various performance and cost parameters, like for instance:

- speed up the execution of the program
- reduce power consumption of the processor

Using tree pattern matching methods, the syntax tree is covered by machine code templates.

How to translate iterative and conditional high level constructs into machine code

High level constructs like the following ones:

- *if then else*
- *while do*, *repeat until*, *for do* and *loop exit*
- *case* and *switch*
- *break* and *continue*
- etc ...

should be translated using *conditional or unconditional jump and branch machine instructions*.

The transducer need generate and insert *destination labels* to tag the memory addresses where jump and branch machine instructions are directed to. These new labels must be distinguishable from those used elsewhere for different purposes (e.g. to tag variable cells, etc.).

At each invocation, the special predefined function new assigns the attribute n a new integer value, different from all those generated so far.

The format of the new destination labels is free, however in the following the lexical model $e397$, $f397$, $i23$, $\ldots$, will be adopted for these labels.

The generic attribute tr is used to store the translation of a construct of the source text.

The translation is a more or less long string of characters, containing the sequence of machine instructions; each instruction is itself a substring.

The complete translation is generated one piece at a time and the string concatenation operator $\bullet$ is used to juxtapose partial translation fragments, e.g. to concatenate machine instructions for data manipulation to jump and branch instructions and to the new destination labels.

Separators, e.g. ';' or others, are inserted between consecutive machine instructions.

Grammar of the conditional construct "if":

syntax	*semantic functions*
$F_0 \rightarrow I_1 \mid \ldots$	$n_1 := new$
$I_0 \rightarrow$ **if** $(cond)$	$tr_0 := tr_{cond} \bullet$ 'jump-if-false' $\bullet$ ' e' $\bullet\, n_0 \bullet$ ';' $\bullet$
then L_1	$tr_{L_1} \bullet$ 'jump-uncond' $\bullet$ ' f' $\bullet\, n_0 \bullet$ ';' $\bullet$
else L_2	'e' $\bullet\, n_0 \bullet$ ':' $tr_{L_2} \bullet$
end if	'f' $\bullet\, n_0 \bullet$ ':'

Symbol $\bullet$ is concatenation, while 'e' and 'f' are mnemonic for "else" and "finish", respectively.

The translation fragments of the logical condition $cond$ and of the other phrases (e.g. the instruction sequences in the "then" and "else" branches of "if") are generated by semantic functions here omitted (but all working on tr). In the following they are indicated by "$transd\_of(\ldots)$".

Suppose each partial translation fragment is automatically appended a separator ';' at the end.

Translation of a conditional (new returns 7):

source text	*machine code (assembly)*
if $(a > b)$	$transd\_of(a > b)$
	jump_if_false e7;
then $a := a - 1$	$transd\_of(a := a - 1)$
	jump_uncond f7;
else $a := b$	e7: $transd\_of(a := b)$
end if	f7: - - rest of the prog.

Grammar of the iterative construct “while”

syntax	*semantic functions*
$F_0 \to W_1 \mid \ldots$	$n_1 := new$
$W_0 \to$ **while** $(cond)$	$tr_0 :=$ ‘i’ $\bullet\ n_0 \bullet$ ‘:’ $\bullet\ tr_{cond} \bullet$
	‘jump_if_false’ $\bullet$ ‘ f’ $\bullet\ n_0 \bullet$ ‘;’$\bullet$
L_1	$tr_{L_1} \bullet$
	‘jump_uncond’ $\bullet$ ‘ i’ $\bullet\ n_0 \bullet$ ‘;’$\bullet$
end while	‘f’ $\bullet\ n_0 \bullet$ ‘:’

Symbol $\bullet$ is concatenation, while ‘i’ and ‘f’ are mnemonic for “iterate” and “finish”, respectively.

Translation of an iterative (new returns 8):

source text	*machine code (assembly)*
while $(a > b)$	i8: $transd\_of(a > b)$
	jump_if_false f8;
$a := a - 1$	$transd\_of(a := a - 1)$
	jump_uncond i8;
end while	f8: - - rest of the prog.