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.

Abstract syntax is kept as simple as possible, compatibly with the semantic of the language.

Syntax ambiguity does not prevent the transduction of being one-valued, as the parser passes to the evaluator only one syntax tree (of many).

Simple transducers merge the two phases into one and use the effective language syntax.

Example: convert from base 2 to base 10.

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

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

To be fully explicit (if it helps), write as follows:

syntax	*semantic functions*	
$N_0 \to D_1 \bullet D_2$	$v_{0,N} := v_{1,D} + v_{2,D} \times 2^{-l_{2,D}}$	
$D_0 \to D_1 B_2$	$v_{0,D} := 2 \times v_{1,D} + v_{2,B}$	$l_{0,D} := l_{1,D} + 1$
$D_0 \to B_1$	$v_{0,D} := v_{1,B}$	$l_{0,D} := 1$
$B_0 \to 0$	$v_{0,B} := 0$	
$B_0 \to 1$	$v_{0,B} := 1$	

This way it is mostly clear, if not too teasing.

However, normally attributes need only have numerical pedices, provided the association between attributes and nonterminal symbols is tabulated aside once for all, along with the specification of the attribute domain and possibly also with a brief explanation of the interpretation (or semantic) the attribute itself is given.

Should a nonterminal symbol appear only once in a given support rule, it would be sufficient to index the attribute by the non-terminal name alone and the pedex could be omitted (though writing it may help anyway).

Somebody keeps on writing ‘v_0 of N’ instead of ‘$v_{0,N}$’ (overloaded notations) , or simply ‘v_0’ (if one knows which support rule to look at, numerical pedices alone suffice), or even on writing ‘v of N_0’ (obsolete notation); all of them are simply different notational conventions. For instance:

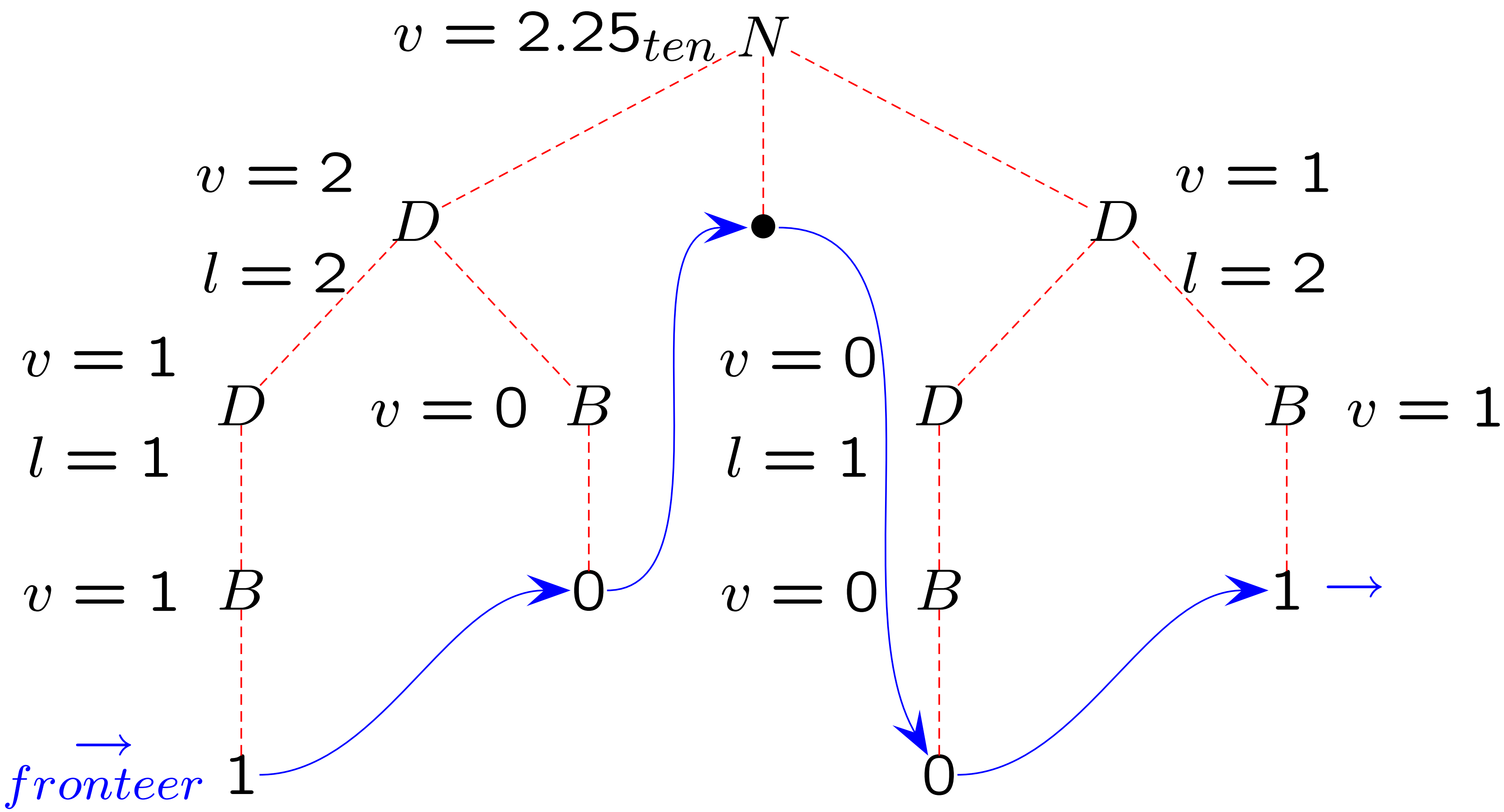
$$v_0 \text{ of N} := v_1 \text{ of D} + v_2 \text{ of D} \times 2^{-l_2 \text{ of D}}$$

$$v \text{ of } N_0 := v \text{ of } D_1 + v \text{ of } D_2 \times 2^{-l \text{ of } D_2}$$

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



Labeling the attributes with pedices is not required here; each attribute name is associated with the non-terminal name nearby (some nodes have two attributes).

Two or more attribute computation orders are possible: all of them however need not attempt to evaluate a semantic function before evaluating all those that compute the arguments thereof.

After completing the computation, the final result is that associated with the tree root.

The attributes associated with all the other (internal) nodes are themselves intermediate values and can be discarded at the end.

Most frequently the attributes associated with the tree leaves are initial, from where the computation and the subsequent propagation start.

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).

After instrumenting the text, every line must contain the maximum number of unsplit words compatible with the upper bound W to the line length; wrapping a word across two consecutive lines with hypenathion is not contemplated here (might be, but the problem gets more difficult).

Examples are left in italian (but everything applies to english as well, without any change).

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

To compute the attribute ult of the current word w_k, one need first know the column number of the last character of the word w_{k-1} that precedes immediately the current one, which by definition is the value of attribute pre of w_k, then increment to account for one blank separator, and finally add the length of the current word w_k, which by definition is the value of attribute lun of w_k.

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:

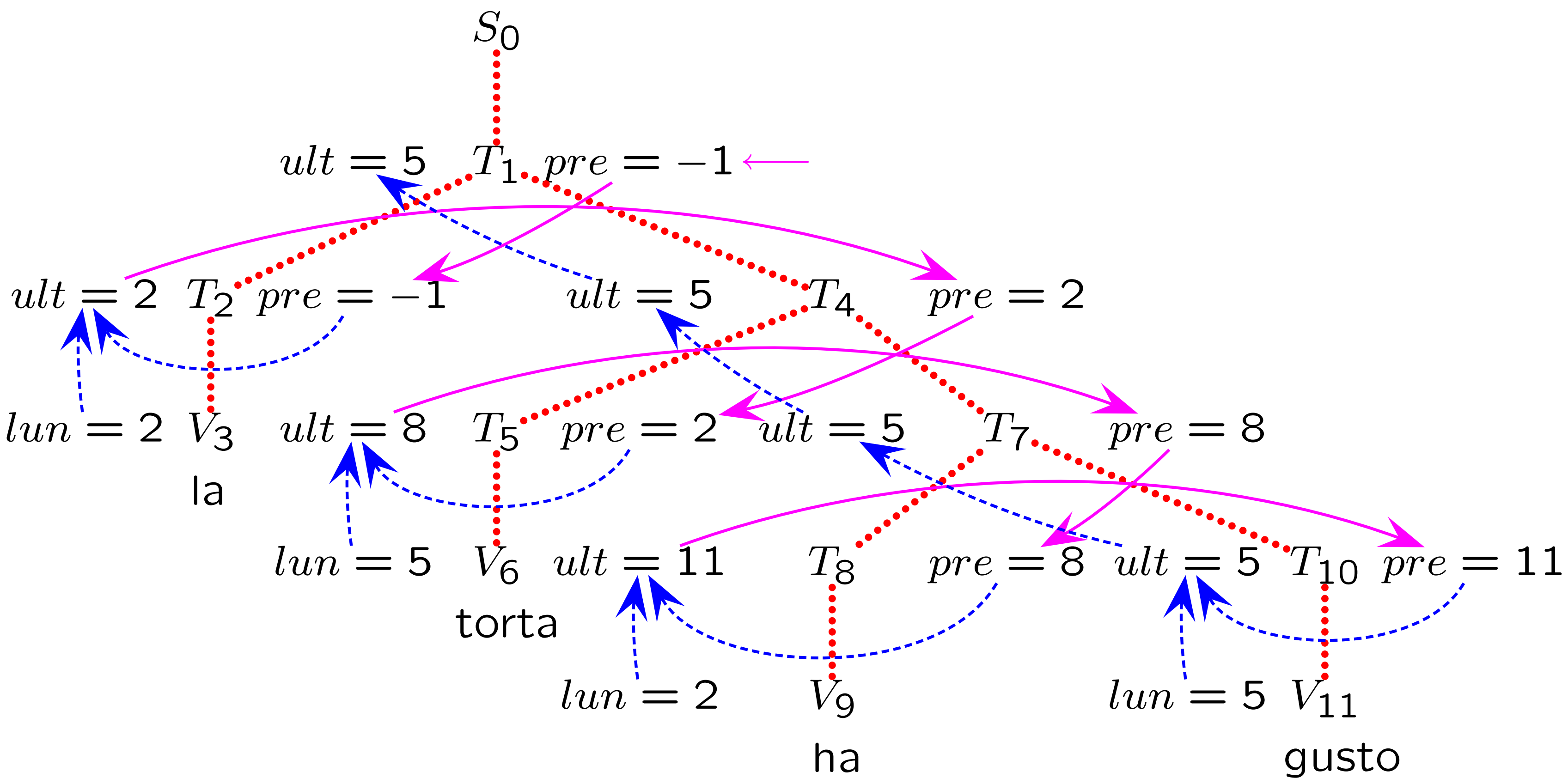
syntax	*semantic functions*	
	right attributes	*left attributes*
1: $S_0 \to T_1$	$pre_1 := -1$	
2: $T_0 \to T_1 \perp T_2$	$pre_1 := pre_0$ $pre_2 := ult_1$	$ult_0 := ult_2$
3: $T_0 \to V_1$		$ult_0 :=$ **if** $(pre_0 + 1 + lun_1 \leqslant W)$ **then** $(pre_0 + 1 + lun_1)$ **else** (lun_1) **end if**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

On the other side, an ambiguous syntactic support is preferable as it is simpler than unambiguous versions are (in general).

Therefore, in general an attribute grammar may be designed so as to have an ambiguous syntactic support, provided the designer earns some description simplicity or elegance in return.

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.

Question: the grammar shown above uses both right and left attributes, but might one model the same computation without using right attributes ? *Answer*: yes. Here is how to do:

1. compute the left attribute lun

2. compute a new left attribute $lista$ $(= list)$, the domain of which is an ordered list of integers representing the lengths of the words

In the previous tree figure, the node T that generates the following phrase:

"la torta ha gusto"

would have the attribute $lista = \langle 2, 5, 2, 5 \rangle$.

The values of $lista$ in the tree root, along with the bound W, allows to compute the column number of the last character of each word.

Drawbacks of the new proposal:

- the computation to apply at the tree root is essentially the same as the full problem

- therefore the problem is not effectively decomposed into meaningful subproblems, each decidedly simpler than or at least different from the original full problem

- all or almost all of the final information remains concentrated in the tree root and does not decorate the internal nodes of the tree
- non-scalar attributes are used, which have a complex domain (e.g. a list or a set)

Often the most elegant and efficient solution uses both left and right attributes, which interact and exchange information with one another.

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:

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

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 $\delta_0 := \ldots$.

10. Relationships of attributes and support rules:

- The left and right attributes σ_0 and δ_i ($i \neq 0$), associated with (non)terminal symbols occurring inside of the syntax support syntax rule p, are said to be *internal* to p.

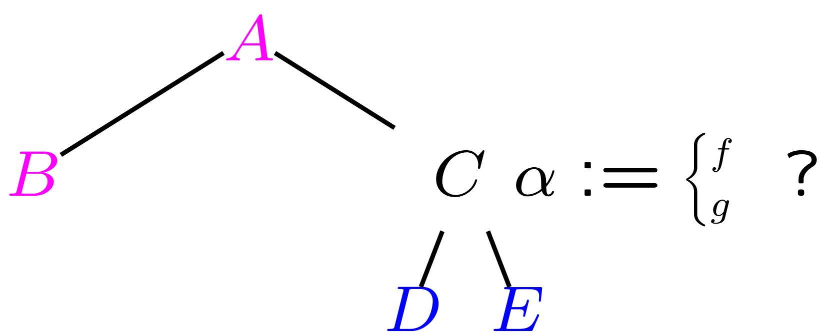
- The right and left attributes δ_0 and σ_i ($i \neq 0$), associated with (non)terminal symbols occurring inside of another syntax support syntax rule q different from rule p ($q \neq p$), are said to be *external* to p.

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 $\alpha_C := g(attr(D, E))$



Attribute α_C, internal to both rules 1 and 2, is right in 1 and left in 2: this does not work !

The value assigned to α_C would depend on the evaluation order of the two semantic functions f and g (the last to be applied would prevail).

Semantic would therefore loose the (essential) property of independence of the implementation of the semantic evaluator of the attributes.

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.

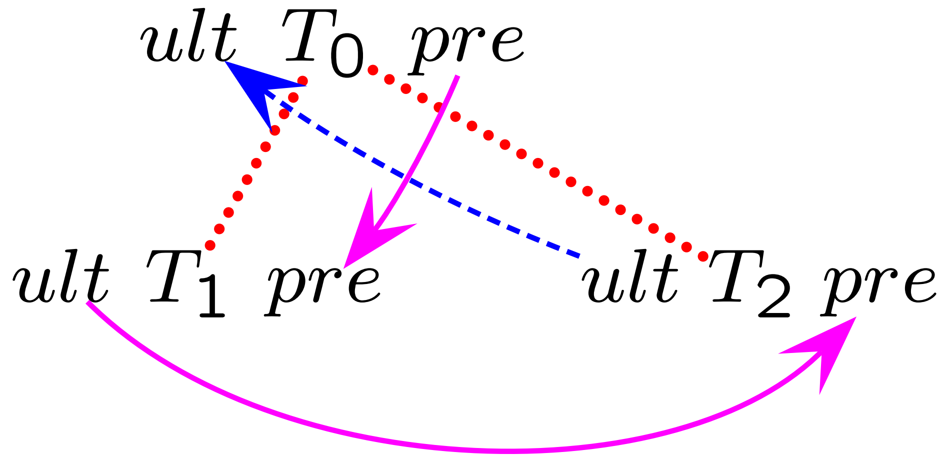
Dependence graph of a semantic function:

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

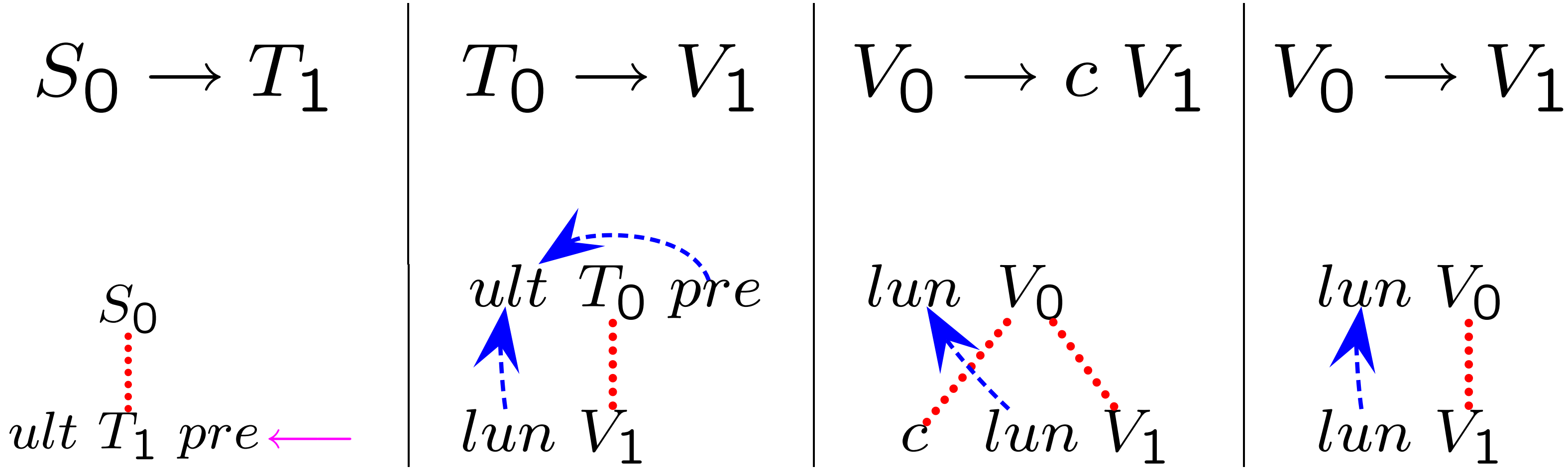
Wrap the dependence graph onto the syntax support (and possibly omit terminal symbols). Here follows an example for rule 2 before:

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



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

Other syntax rules of the same grammar:



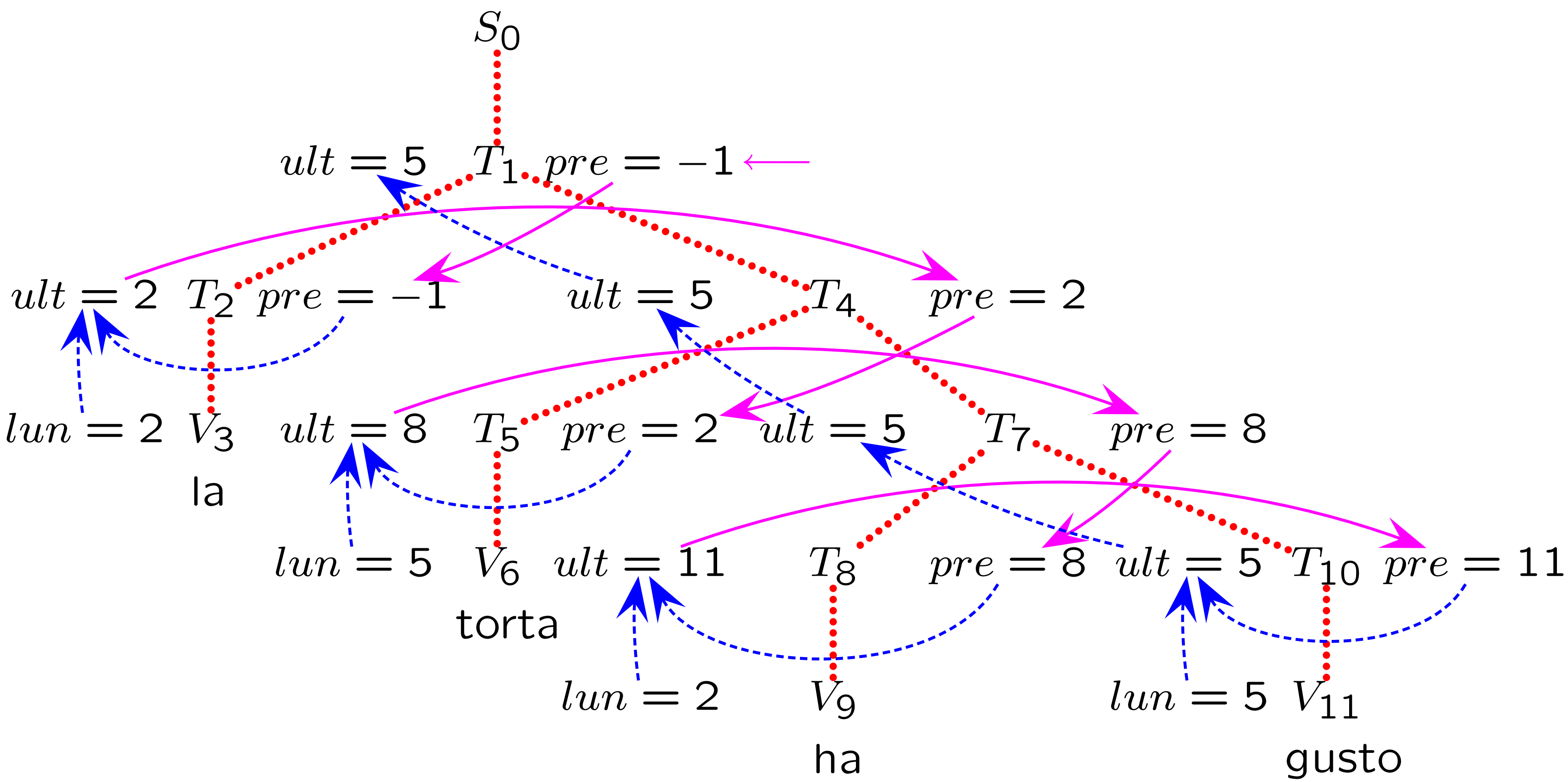
Names with and without incoming arcs indicate attributes that are internal and external to the current syntax rule, respectively.

Dependence graph of a complete syntax tree:

First complete by argument names and wrap by dependence arcs each elementary subtree corresponding to a syntax rule.

Then decorate the full syntax tree of the phrase, completing and wrapping each elementary subtree as explained before. The full tree is the support for the *thread* of function dependences.

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.

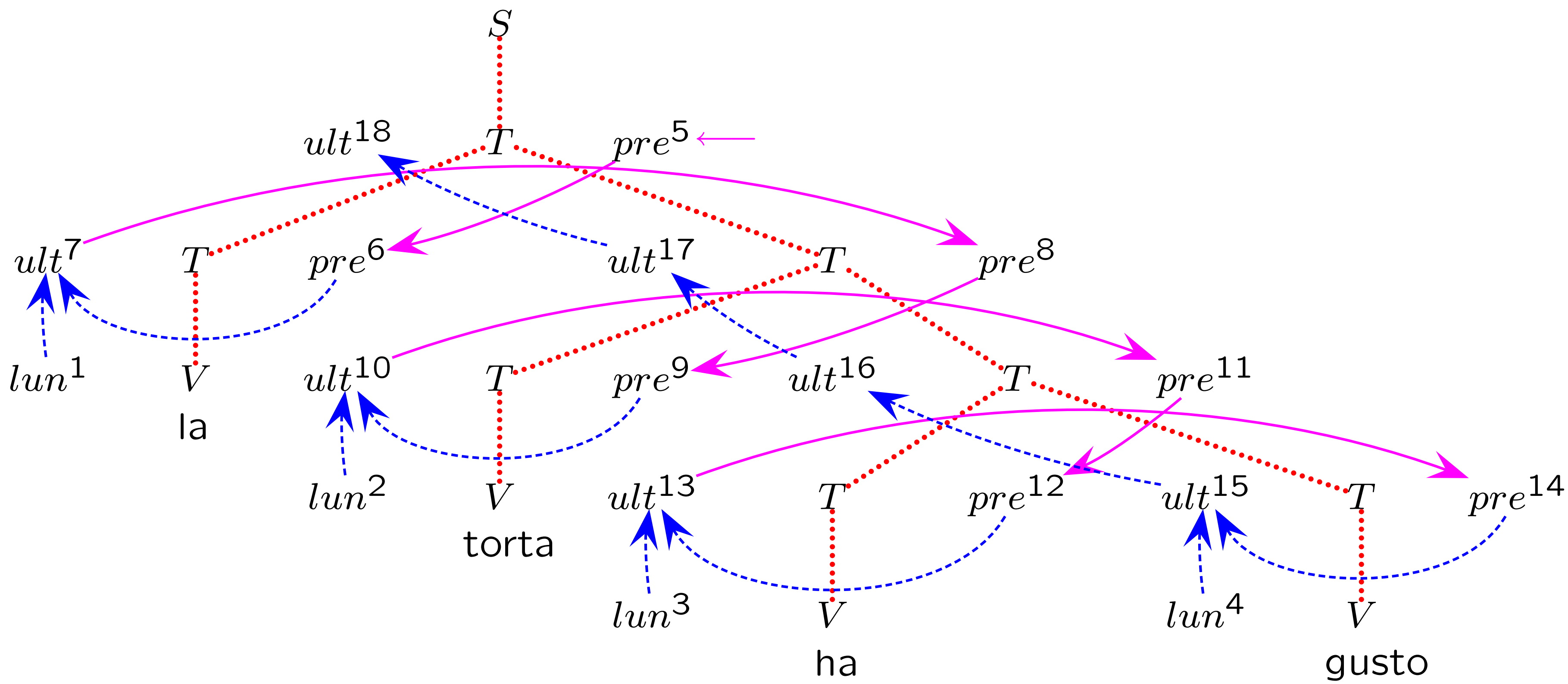
```
input G                                      -- graph to sort G = (V, E)
output ord                                   -- vector of sorted nodes
begin                                        -- start and input graph
    m := 1                                   -- set node counter
    V0 := {n ∈ V | n has no incoming arcs}   -- strip off initial nodes
    while V0 ≠ ∅ do                          -- initial sorting loop
        get node n from V0                   -- get an initial node
        ord[n] := m                          -- insert into the sorting
        m := m + 1                           -- increment node counter
    end while
    V := V \ V0                              -- remove sorted nodes
    while V ≠ ∅ do                           -- outer sorting loop
        V0 := {n ∈ V | n has no incoming arcs}   -- strip off initial nodes
        while V0 ≠ ∅ do                      -- inner sorting loop
            get node n from V0               -- get an initial node
            ord[n] := m                      -- insert into the sorting
            m := m + 1                       -- increment node counter
        end while
        V := V \ V0                          -- remove sorted nodes
        E := E \ {outgoing arcs of V0}       -- remove dangling arcs
    end while
end                                          -- return ord and stop
```

The initial loop of the algorithm sorts arbitrarily the nodes that do not have dependences (= that do not have any incoming arcs - initial nodes).

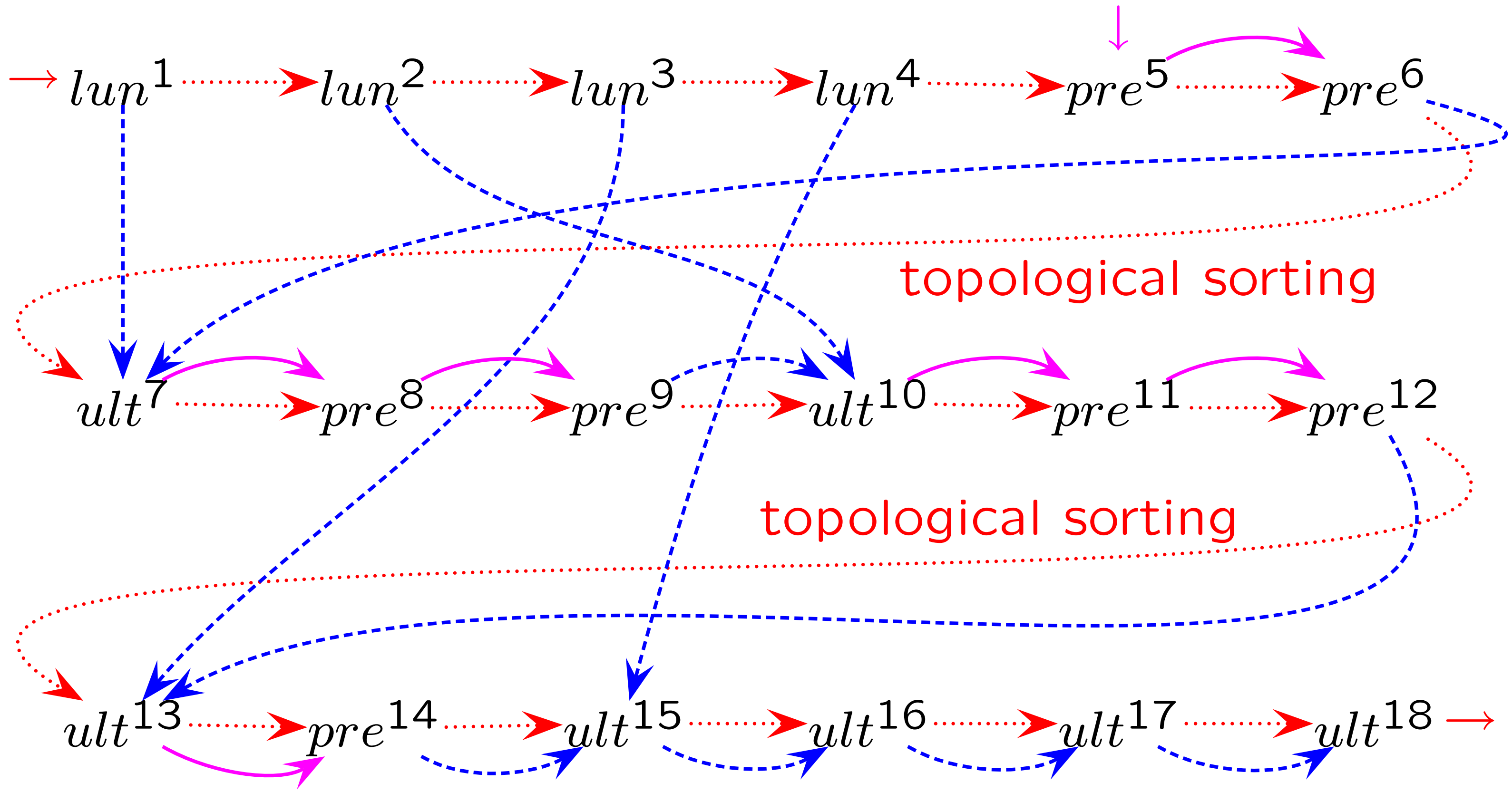
Sort the dependence graph of the previous example and obtain the following valid sorting:

$$\begin{array}{cccccc} lun^1 & lun^2 & lun^3 & lun^4 & pre^5 & pre^6 \\ ult^7 & pre^8 & pre^9 & ult^{10} & pre^{11} & pre^{12} \\ ult^{13} & pre^{14} & ult^{15} & ult^{16} & ult^{17} & ult^{18} \end{array}$$

topological sorting for attribute evaluation



display in line ...



... and all the arrows are directed forwards !

First determine the topological sorting of the attributes, using the algorithm illustrated above.

Then necessarily assign the initial node in the sorting a constant value, as such a node does not have any predecessor. Other attributes may be assigned a constant (e.g. those of terminals).

And finally compute in topological order the assignments to the remaining (non-constant) attributes, until all are done.

Fixed order scheduling

First open problem: the above mentioned procedure is universal (as it works in all cases) but inefficient: the semantic evaluator should apply the topological sorting algorithm to all the tree nodes (= attributes) before computing the assignments to the attributes themselves.

Here is a method to design a faster semantic evaluator: determine a fixed topological sorting of the tree nodes (called *node scheduling*) which is valid for every syntax tree of the grammar, that is, which is always conformant to the dependences of the attributes.

Second open problem: given an attribute grammar, how can one check whether every decorated syntax tree (that the grammar can generate) has a loop-free dependence graph ?

As the source language contains infinitely many phrases, the grammar generates as many syntax trees and therefore acyclicity can not be checked in an exhaustive way.

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, $\ldots$, 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 $D_i \rightarrow D_j$ ($i \neq j$ and $i, j \geqslant 1$), if and only if in dip_p there is arc $\alpha_i \rightarrow \beta_j$ 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

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

1. Graph dip_p is loop-free.
2. Graph dip_p does not contain any path $\sigma_i \to \ldots \to \delta_i$ $(i \geqslant 1)$ 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.

Explanation why the conditions work:

1. This is a sufficient condition (and necessary indeed) for the grammar to be loop-free.
2. If there were a path $\sigma_i \rightarrow \ldots \rightarrow \delta_i$ ($i \geqslant 1$), the right attribute δ_i could not be computed before evaluating subtree t_i, as the value of the left attribute σ_i would be known only after finishing such evaluation; but this situation would conflict with the actual scheduling.

3. Should this condition fail, the right attribute δ_i would still be unavailable when subtree t_i is scheduled, contrary to the expected behaviour of a right attribute.
4. This is a sufficient condition (and necessary indeed) to sort the brother nodes (the subtrees) in topological order and to schedule them in a way that is conformant to all the dependences in dip_p.

If the brother graph were not loop-free, there could not exist a valid scheduling for all the attributes associated with nonterminal symbols of the right part of rule p.

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:

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

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.

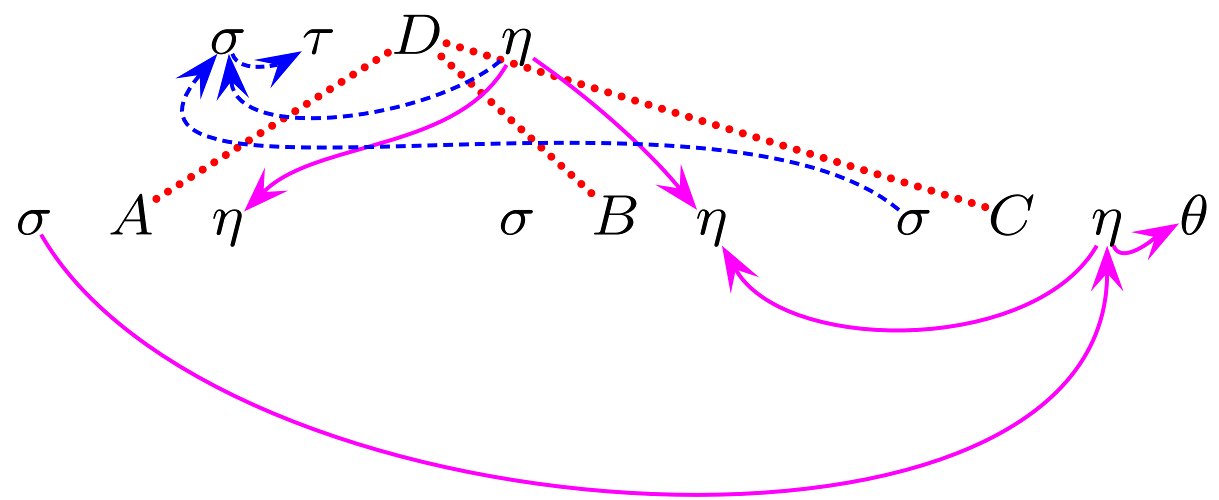
These steps must be repeated for each semantic procedure to design, that is for each nonterminal symbol of the grammar.

Example of one-sweep semantic procedure

Support rule p and dependence graph dip_p:

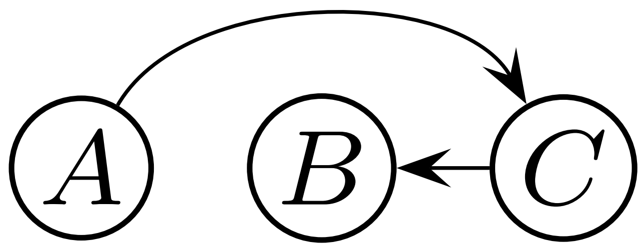
support rule

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:

$A \to C$ from dependence $\sigma_A \to \eta_C$

$C \to B$ from dependence $\eta_C \to \eta_B$

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: $D \to ABC$

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

var $\eta_A, \sigma_A, \eta_B, \sigma_B, \eta_C, \theta_C$ - - local attribute variables for parameter passing

begin - - start and input tree t_D and right attribute η of D

$\eta_A := f_1(\eta_D)$ - - 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

$\eta_C := f_2(\sigma_A)$ - - by TSD of C compute right attribute η of C

$\theta_C := f_3(\eta_C)$ - - 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

$\eta_B := f_4(\eta_D, \eta_C)$ - - 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

$\sigma_D := f_5(\eta_D; \sigma_C)$ - - by TSL compute left attribute σ of D

$\tau_D := f_6(\sigma_D)$ - - 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.

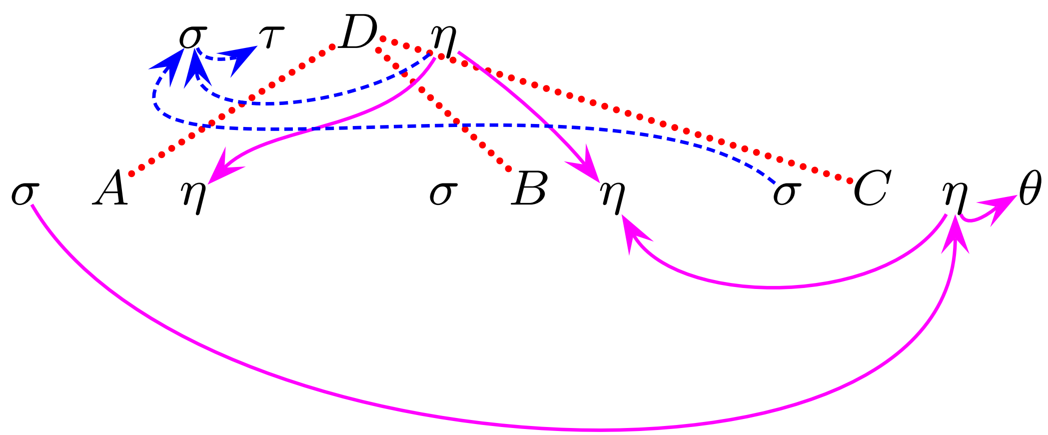
Here follows an example (the same as before):

support rule

$$D \longrightarrow ABC$$

scheduling

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



The linear prefix-postfix order is the following:

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

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).

There are three cases to consider, depending on the structure of the source language:

- *regular source language*: lexical analysis enhanced with (lexical) attributes
- *syntax of type* $LL(k)$: top-down (or recursive descent) parser enhanced with attributes
- *syntax of type* $LR(k)$: bottom-up (or shift-reduction) parser enhanced with attributes

The first case (regular) is dealt with by the well-known widespread software tools *flex* or *lex*.

The second case (top-down) is easy to design and implement manually by means of attributes only of the left type (no right attributes).

The third case (bottom-up) is dealt with by the well-known and widespread software tools *bison* or *yacc*, but to cope with its inherently higher complexity (than the others) the right attributes should be either totally excluded or permitted but only with appropriate dependence restrictions (the latter is the way bison and yacc do).

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: $D_0 \longrightarrow D_1 \dots D_r$, and follows an order that need not necessarily be that of natural integers, i.e. 1, 2, ..., $r-1$, 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

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

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

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

Part 2 of the L-condition prevents any right attribute associated with node D_i of depending on an attribute (independently of whether the latter is left or right) associated with a node D_j located on the right side of D_i.

Therefore the natural scheduling order of the subtrees, i.e. 1, 2, . . . , $r - 1$, r, is itself already conformant to the dependences that constrain how to schedule the subtrees.

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.

Attributes and interpretation:

name	*meaning*	*domain*	*type*	*nonterm.*
v	number value	real num.	left	N, D, B
l	number length	natural int.	right	D, B

Each bit value is weighted by a negative exponent equal to the distance from the fixed point.

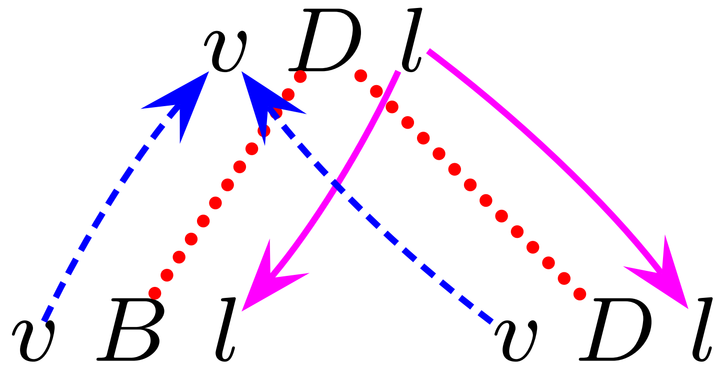
Syntax is deterministic of type LL(2) (not LL(1), check), needs a lookahead window of width 2.

How to check the L-condition for each rule

- $N \rightarrow \bullet D$ - the dependence graph dip of this rule has only arc $v_1 \rightarrow v_0$. Hence:
  - the dependence graph dip is loop-free
  - there is not any path from the left attribute v to the right attribute l, both associated with the same child node

- there is not any arc from attribute v (of any type) associated with the parent node to the right attribute l associated with the child node
- the brother graph bro does not have any arc

- $D \rightarrow BD$ - 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

- $D \to B$: idem as above ($D \to BD$)
- $B \to 0$: graph dip does not have any arc
- $B \to 1$: graph dip has only one arc, namely $l_0 \to v_0$, which is one-sweep compatible

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 l0; out v0)

begin                                - - start and input parameters
    case cc1 of                      - - check lookahead (depth 1)
        '0': begin                   - - case of alternative B → 0
            read                     - - shift lookahead window
            v0 := 0                  - - reset attribute v of B
        end
        '1': begin                   - - case of alternative B → 1
            read                     - - shift lookahead window
            v0 := 2^(-l0)            - - 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.

Semantic check

The formal language L_F defined in a purely syntactic way is only a rough approximation by excess of the technical language L_T to compile.

The following inclusion relationship holds:

$$\underbrace{L_F}_{\text{context free}} \supset \underbrace{L_T}_{\text{context dependent}}$$

It is well known that context dependent syntax (Chomsky type 1 or 0) is not practical, due to its exceeding manipulation difficulty.

Effective and practical syntax is limited to context free (type 2) if not regular (type 3).

Imagine L_T is a programming language of some kind, for instance imperative like C or Pascal.

The phrases of L_F are *syntactically correct*, but may still break several programming rules, e.g.:

- compatibility among operand types in expressions or assignments (type checking)
- type matching of formal and actual parameters in routine calls (parameter checking)
- matching between variable declaration and reference, and scope rules of local variables

How to check programming rules semantically:

- Semantic functions compute logical (boolean) attributes, also called *semantic predicates*.
- If the programming rule is violated, semantic evaluation assigns the predicate value false.
- A semantic predicate may in turn depend on other (logical or not) attributes, which represent various properties of the source text.

Example: match variable declaration and reference (for instance in an assignment) - a variable *must be declared* ahead of the point(s) where it is referenced, and the declaration and reference point(s) types *must be compatible* (in simple cases, must be identical).

Difficulty is in that the variable declaration may be far from the points where it is referenced.

The variable declaration type is stored into an attribute that is a complex data structure, called *symbol table* (ST) or also *environment*.

The ST is propagated through the syntax tree as far as it reaches the points (here the assignments) where the variable is referenced.

Pay attention to that propagating the ST would be very inefficient if it should be actually done, but this is only a conceptual model (see next how to manage and update the ST efficiently).

In the practice compilers implement and manage the ST as a global data structure (or object) in the scope of every semantic function.

The following example sketches the creation of the ST and how to use it to check the variables.

Symbol table and type checking

Here follows a simple example of type checking carried out by means of the symbol table. Two programming features (very common indeed) need be verified and matched:

- Declaring variables of scalar or vector type.
- Referencing such variables in the assignments.

Programming rules for semantic correctness:

1. one may not (re)declare a variable twice or more times, even if the repeated declaration is identical to the first one
2. one may not either reference a variable without declaring it or declare after referencing
3. only assignments between two scalar variables or two vectors of equal size are admitted

The syntactic support need distinguish variable declaration and reference in assignments:

- The name (identifier) of the variable can be used as a search key through the ST.

- The ST contains the descriptor $desc$ of the variable, which specifies the type of the variable (scalar or vector) and size (if vector).

The symbol table ST is a left attribute of all the nodes of the syntax tree t of the program.

The semantic predicate dd (left attr.) denounces the existence of a *double declaration*.

The attribute ST is propagated through the entire program to check (the entire syntax tree).

The left L and right R parts of an assignment have the same attribute $desc$, which specifies:

- a variable:
  - without index (scalar)
  - with index (array element)
- or a constant

If the searched variable name is not listed in ST, descriptor $desc$ denounces an error.

Semantic check to perform in an assignment:

- Check the existence of the two declarations.
- Check the compatibility of the two types.
- Compute predicate ea *erroneous assignment*.

In the following, for brevity syntax rules will not be labeled with numerical pedices - the reader is left the simple task of mentally associating attributes with nonterminal symbols.

syntax	*semantic functions*	- - *comment*
$S \to P$	$t_1 := \emptyset$	- - initialize symbol table
$P \to DP$	$t_2 := insert(t_0, n_1, desc_1)$	- - store desc. into sym. tab.
$P \to AP$	$t_1 := t_0$ $t_2 := t_0$	- - propagate symbol table
$P \to \varepsilon$		
$D \to id$	$dd_0 := exists(t_0, n_{id})$ **if** $(\neg dd_0)$ **then** $desc_0 :=$ 'sca' **end if** $n_0 := n_{id}$	- - declare scalar variable
$D \to id\ [const]$	$dd_0 := exists(t_0, n_{id})$ **if** $(\neg dd_0)$ **then** $desc_0 := ($'vec'$, v_{const})$ **end if** $n_0 := n_{id}$	- - declare vector variable
$A \to L := R$	$ea_0 := \neg\langle desc_1$ is compatible with $desc_2\rangle$	- - check types in assignment
$L \to id$	$desc_0 := \langle$type of n_{id} in $t_0\rangle$	- - assign sca. or vect. var.
$L \to id\ [id]$	**if** $\big(\langle$type of n_{id_1} in $t_0\rangle =$ 'vec' $\wedge \langle$type of n_{id_2} in $t_0\rangle =$ 'sca'$\big)$ **then** $desc_0 := \langle$desc. of n_{id_1} in $t_0\rangle$ **else** error **end if**	- - assign indexed variable
$R \to id$	$desc_0 := \langle$type of n_{id} in $t_0\rangle$	- - reference sca. or vect. var.
$R \to const$	$desc_0 :=$ 'sca'	- - reference constant
$R \to id\ [id]$	**if** $\big(\langle$type of n_{id_1} in $t_0\rangle =$ 'vec' $\wedge \langle$type of n_{id_2} in $t_0\rangle =$ 'sca'$\big)$ **then** $desc_0 := \langle$desc. of n_{id_1} in $t_0\rangle$ **else** error **end if**	- - reference indexed variable

Syntactically correct text (but not semantically):

$$\overbrace{a[10]}^{D_1}\ \overbrace{i}^{D_2}\ \overbrace{b}^{D_3}\ \overbrace{i := 4}^{A_4}\ \overbrace{c := a[i]}^{A_5:\ ea=true}\ \overbrace{c[30]}^{D_6}\ \overbrace{i}^{D_7:\ dd=true}\ \overbrace{a := c}^{A_8: ea=true}$$

The analyzer detects several semantic errors:

- in the assignments A_5 and A_8
- in the variable declaration D_7

Patches and add-ons for a real-world compiler:

- implement an informative and precise diagnostic messaging system
- discriminate among error types (e.g. undefined variable, incompatible type, vector index out of bound, and many others)

- inform the programmer about the location where the error has first manifested itself (line number and column number)
- every predicate computed somewhere in the syntax tree should be propagated to the tree root, along with its coordinate
- design an appropriate messaging procedure to output diagnostic warnings in a uniform and easily understandable way

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.

If the machine language is not too differing from the source language, translation can be carried out directly by the syntax parser. For instance:

- a simple and known case is the syntactic transduction to the polish from (prefix or postfix), executable by a transduction scheme

- however, translating to machine code a program written in a modern high level language like C or Java is decidedly more difficult

extremely interesting course on this topic

analisi e ottimizzazione dei programmi

analysis and optimization of programs

currently held in italian

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, ..., 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.

Semantic driven syntax analysis

In the classical approach to language processing, syntax analysis precedes semantic analysis and therefore they are independent of each other.

When syntax is ambiguous, parsing produces two or more syntax trees and semantic analysis should select later on those that are semantically valid.

However in the field of technical languages (e.g. programming languages, etc) such an event seldom occurs, and normally technical languages are designed so as to have a deterministic natural syntax, for practical purposes; hence their syntax is unambiguous as an immediate consequence of determinism.

Processing of natural language texts (english, italian or any other) represents a completely opposite situation: here syntax is often highly ambiguous and obviously can not be changed.

fascinating course on this topic

trattamento del linguaggio naturale

processing of natural language

currently held in italian

In a well designed technical language, all the phrases are semantically unambiguous, that is they have a unique interpretation or meaning (= a unique translation in some other language).

Often one can select the valid tree, out of the set of all those that are syntactically admittable, as early as in the parsing stage; to this purpose one can exploit the semantic information that is available to the compiler at parsing time.

With reference to top-down or LL(k) syntax:

- if the parser can not choose out of two rules with overlapping lookahead sets
- it can make a decision by resorting to a semantic attribute, called *driver predicate*
- and of course the driver predicate need be computed by the syntax-semantic analyzer, ahead of the decision point

The attributes are divided into two classes:

1. Driver predicates and the other attributes that depend on the former ones; this class need be fully computed at parsing time.
2. All the remaining attributes, which may be computed after building the unique valid syntax tree (with the help of driver predicates).

And the former class must satisfy the L-condition.

Therefore the driver predicate is already available when the parser need choose out of two or more alternative rules.

Consider the nonterminal symbols D_i $(1 \leqslant i \leqslant r)$ and the rule:

$$p\colon D_0 \rightarrow D_1 \ldots D_i \ldots D_r$$

and assume the syntax tree has been built from the root down to the subtrees D_1 ... D_{i-1}.

After the L-condition the driver predicate may depend of the following attributes:

- the right attributes of the parent node D_0
- the attributes (independently of whether left or right) of the child nodes that, in the above rule p, precede the subtree rooted at D_i to build

Example of a pointless[‡] language

In the programming language PLZ-SYS, designed in the '70 (by IBM) specifically for a resourceless processor, commas and the other usual punctuation symbols are missing.

The list of formal parameters of a procedure header may then be syntactically ambiguous.

[‡]May mean "without punctuation" as well as "irrelevant" or "meaningless".

For instance, there are three possible interpretations of the formal parameter types listed in the declaration header of procedure P below.

$$P \text{ proc } (X\ Y\ T1\ Z\ T2) \begin{cases} 1 & \begin{array}{l} X \text{ is of type } Y \\ T1 \text{ and } Z \text{ are of type } T2 \end{array} \\[2ex] 2 & \begin{array}{l} X \text{ and } Y \text{ are of type } T1 \\ Z \text{ is of type } T2 \end{array} \\[2ex] 3 & X, Y, T1 \text{ and } Z \text{ are of type } T2 \end{cases}$$

The following notational conventions are used:

- Each parameter is declared by type and two or more parameters of the same type can be included in the same type declaration.

- Type declarations must be placed ahead of procedure declaration headers.

Suppose the previous type declarations for procedure P are the following:

$$\text{type } T1 = \text{record} \ldots \text{end}$$

$$\text{type } T2 = \text{record} \ldots \text{end}$$

Case 1: is impossible as Y is not a type identifier, while $T1$ is not a variabile.

Case 3: is impossible for a similar reason.

Case 2: is possible and actually is the valid one.

Therefore if one resorts to semantic information, syntactic ambiguity can be eliminated.

Next see how to exploit type declarations (which precede procedure declarations and therefore are already known) to drive the syntax-semantic analyzer to choose out of cases 1, 2 and 3.

The relevant parts of the syntax structure of the declarative section D are the following:

1. T, that is type declarations

2. I, that is procedure declarations headers (here procedure execution sections do not matter)

For each type declaration, it is necessary to model a type descriptor as a left attribute and store it into the symbol table t.

Symbol n is the name or search key (in the symbol table) of an identifier (type or variable).

At the end of the analysis of the type declaration section, the symbol table is propagated to the other parts of the program (in this example the procedure declaration section).

Attribute t (symbol table) is totally copied into the right attribute rt, to be propagated downwards to the lower levels of the syntax tree.

As every type identifier occurring in the procedure declarations should have a corresponding type descriptor stored in rt, the parser can choose the correct syntax rule alternative deterministically, even if the alternative is not deterministic from a purely syntactic point of view.

A few simplifications are adopted, for brevity:

- the scope of the declared objects is global
- only the “record” (or “struct”) data structure is modeled (the keyword is “record”), and is not further expanded to list internal fields
- duplicated declarations are not checked
- the declarations of the procedure name and of the related formal parameter names are not stored into the symbol table

syntax	*semantic functions*	- - *comment*
$D_0 \to T_1\ I_2$	$rt_2 := t_1$	- - copy and propagate sym. tab.
$T_0 \to type\_id_1 =$ **record** ... **end** T_2	$t_0 := insert(t_2, n_1, \text{‘record’})$	- - store desc. into sym. tab.
$T_0 \to \varepsilon$	$t_0 := \emptyset$	- - initialize sym. tab.
$I_0 \to proc\_id_1\ (L_2)\ I_3$	$rt_2 := rt_0$ $rt_3 := rt_0$	- - pass sym. tab. to L_2 (param. list) - - and to I_3 (next procedure)
$I_0 \to \varepsilon$		
$L_0 \to V_1\ type\_id_2\ L_3$	$rt_1 := rt_0$ $rt_3 := rt_0$	- - pass sym. tab. to V_1 (param.) - - and to L_3 (rest of param. list)
$L_0 \to \varepsilon$		
$V_0 \to var\_id_1\ V_2$	$rt_1 := rt_0$ $rt_2 := rt_0$	- - pass sym. tab. to var_id_1 (param.) - - for semantic check - - and to V_2 (other same type param.s)
$V_0 \to var\_id_1$	$rt_1 := rt_0$	- - pass sym. tab. to var_id_1 (param.) - - for semantic check
$type\_id_0 \to \ldots$		- - generic id
$proc\_id_0 \to \ldots$		- - generic id
$var\_id_0 \to \ldots$		- - generic id

The pair of alternative syntax rules that expand V violates condition LL(2) (check yourself).

Type and variable identifiers are lexically indistinguishable from each other (syntagms $type\_id$ and var_id are both generic identifiers).

Therefore the parser resorts to a semantic check to solve the syntactic alternative to expand V.

As before, symbols cc_1 and cc_2 orderly represent the contents of a lookahead window of width 2.

#	*syntax*	#	*driver predicate and other semantic checks*
1:	$V_0 \rightarrow \underbrace{var\_id_1}_{cc_1} \underbrace{V_2}_{cc_2}$	1	$\langle$desc. of cc_2 in tab. $rt_0\rangle =$ 'var. id' $\wedge$
		3	$\langle$desc. of cc_1 in tab. $rt_0\rangle =$ 'var. id'
2:	$V_0 \rightarrow \underbrace{var\_id_1}_{cc_1}$	2	$\langle$desc. of cc_2 in tab. $rt_0\rangle =$ 'type id' $\wedge$
		4	$\langle$desc. of cc_1 in tab. $rt_0\rangle =$ 'var. id'

Driver predicates 1 and 2 solve the non-LL(k) syntactic alternative "$V \rightarrow var\_id\, V \;\mid\; var\_id$". Semantic predicates 3 and 4 check whether the identifier currently in cc_1 refers to a variable.

Similarly, one should associate with nonterminal "$type\_id$", in rule "$L_0 \rightarrow V_1\ type\_id_2\ L_3$", a predicate to check whether the identifier currently in cc_2 (corresponding to "$type\_id$") refers to a type.

Such a check has a purely semantic purpose, as the syntax rules expanding L are of type LL(1).

Concluding, resorting to semantic the parser can build the decorated syntax tree deterministically.