Attribute Grammars

Prof. A. Morzenti

DOMAIN OF APPLICATION OF ATTRIBUTE GRAMMARS

The compilation process uses tasks that cannot be defined using purely syntactic methods
Examples:

- translation of a decimal (base 10) number to binary
- translation of a record definition, ***computing*** the offset in central memory of every field

```
BOOK: record
 AUT: char(8); TIT: char(20); PRICE: real; QUANT: int;
end
```

Symbol	Type	Dimension	Address
BOOK	record	34	3401
AUT	string	8	3401
TIT	string	20	3409
PRICE	real	4	3429
QUANT	int	2	3433

Syntax directed translators

They use functions applied to the syntax tree to compute some ***semantic attributes***

the values of the attributes constitute the translation
(⇒ they express the ***meaning*** of the sentence)

attribute grammars have the same expressive power as the Turing machine
⇒ in fact, they provide a systematic compiler design method,
not a formal model that is easily analyzable, like automata or C.F.Grammars

Compilation is organized in two passes:
1. lexical+syntax analysis produces the syntax tree
2. semantic analysis or evaluation produces the decorated syntax tree
it is designed using attribute grammars

For simplicity the attribute grammar is defined w.r.t. an ***abstract syntax***
a grammar that may be simpler than the real one, often ambiguous, but convenient

The ambiguity of the abstract syntax does not prevent a single-valued translation:
the parser will pass to the semantic evaluator only one syntax tree

The simpler compilers may combine the two phases in a single pass
using a unique syntax, the one of the language

Example: computing the value of a binary fractional number

Source language: $L = \{0, 1\}^+ \bullet \{0, 1\}^+$ (dot '•' separates integer and fractional parts)

Translation of string $1101 \bullet 01 \in \{0, 1\}^+ \bullet \{0, 1\}^+$ is $13{,}25 \in \mathbb{R}$ (NB: it is a number, not a string)

Base syntax: $\{N \to D \bullet D,\ \ D \to DB,\ \ D \to B,\ \ B \to 0,\ \ B \to 1\}$

Attributes and their meaning:

Attribute	Meaning	Domain	Nonterminals that possess the attribute
v	value	decimal number	N, D, B
l	length	integer	D

inside rules, symbol instances are numbered with subscripts ≥0 to be uniquely identified
semantic functions (i.e., assignments of values to attributes) are associated with syntax rules

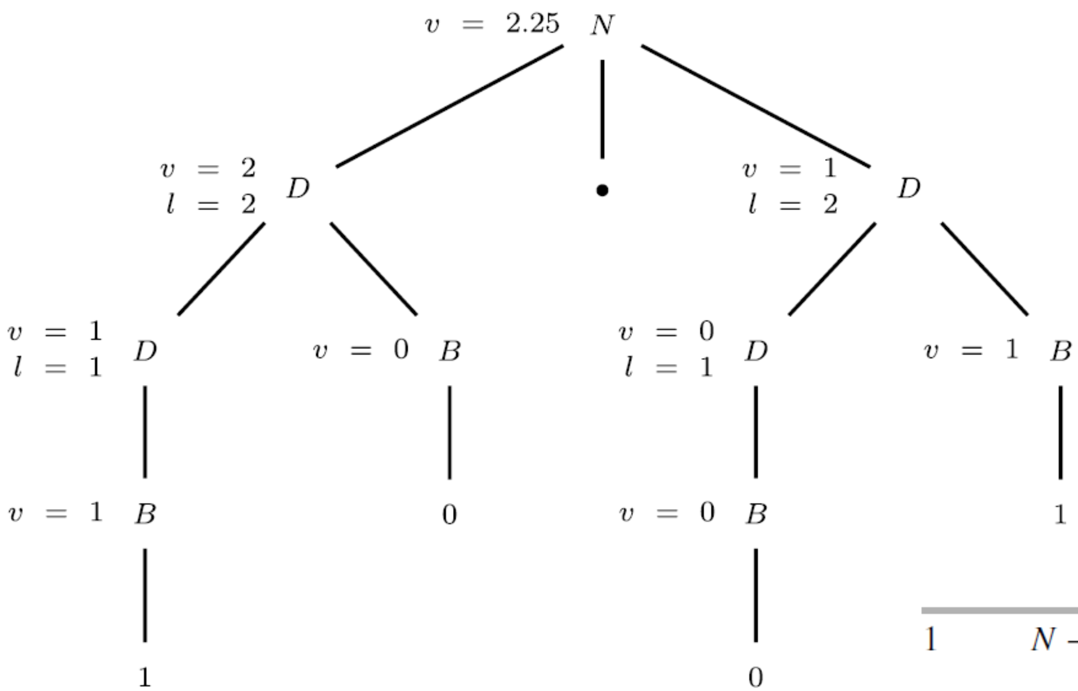
#	Syntax	Semantic functions		Comment
1	$N \to D \bullet D$	$v_0 := v_1 + v_2 \times 2^{-l_2}$		Add integer to fractional value divide by weight 2^{l_2}
2	$D \to DB$	$v_0 := 2 \times v_1 + v_2$	$l_0 := l_1 + 1$	Compute value and length
3	$D \to B$	$v_0 := v_1$	$l_0 := 1$	
4	$B \to 0$	$v_0 := 0$		Value initialization
5	$B \to 1$	$v_0 := 1$		

semantic functions are applied following the dependences among attributes

starting from attributes whose value is known

often the initial values are in the leaves, possibly precomputed by the lexical analysis

«translation» or «meaning» of a sentence is the value of some attribute typically in the root



notice that attribute evaluation goes bottom-up

1	$N \to D \bullet D$	$v_0 := v_1 + v_2 \times 2^{-l_2}$	
2	$D \to DB$	$v_0 := 2 \times v_1 + v_2$	$l_0 := l_1 + 1$
3	$D \to B$	$v_0 := v_1$	$l_0 := 1$
4	$B \to 0$	$v_0 := 0$	
5	$B \to 1$	$v_0 := 1$	

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

left attribute the semantic function $\boldsymbol{\sigma_0 = f(...)}$, whereby attribute $\boldsymbol{\sigma_0}$ is assigned a value, in a rule where $\boldsymbol{\sigma_0}$ is an attribute of the ***left nonterminal*** of the rule

right attribute the semantic function $\boldsymbol{\delta_i = f(...), i \geq 1}$, whereby attribute $\boldsymbol{\delta_i}$ is assigned a value, in a rule where $\boldsymbol{\delta_i}$ is an attribute of a symbol in the ***rule right part***

Example above: all attributes are left/synthesized (typical of simplest cases)

#	Syntax	Semantic functions		Comment
1	$N \to D \bullet D$	$v_0 := v_1 + v_2 \times 2^{-l_2}$		Add integer to fractional value divide by weight 2^{l_2}
2	$D \to D B$	$v_0 := 2 \times v_1 + v_2$	$l_0 := l_1 + 1$	Compute value and length
3	$D \to B$	$v_0 := v_1$	$l_0 := 1$	
4	$B \to 0$	$v_0 := 0$		Value initialization
5	$B \to 1$	$v_0 := 1$		

A more complex example

Segmenting a free text into lines of $\leq W$ chars

The text is a list of one or more words separated by spaces

Requirement: every line must have the maximum possible number of unbroken words

The key attribute is ***last*** :

it indicates the column number of the last char of each word

Example: “no doubt he calls me an outlaw to catch”, W=13; segmented text:

1	2	3	4	5	6	7	8	9	10	11	12	13
n	o		d	o	u	b	t		h	e		
c	a	l	l	s		m	e		a	n		
o	u	t	l	a	w		t	o				
c	a	t	c	h								

attribute ***last*** is 2 for ‘no’ and 5 for ‘calls’

Attributes and their meaning

length	left	length (in chars) of the current word
last	left	column of the last char of current word
prec	right	column of the last char of previous word (-1 for first word)

fundamental relation between attributes concerning two consecutive words
(here we adopt informally notation ***last***(w_k) for attribute ***last*** of ***k*-th** word etc.)

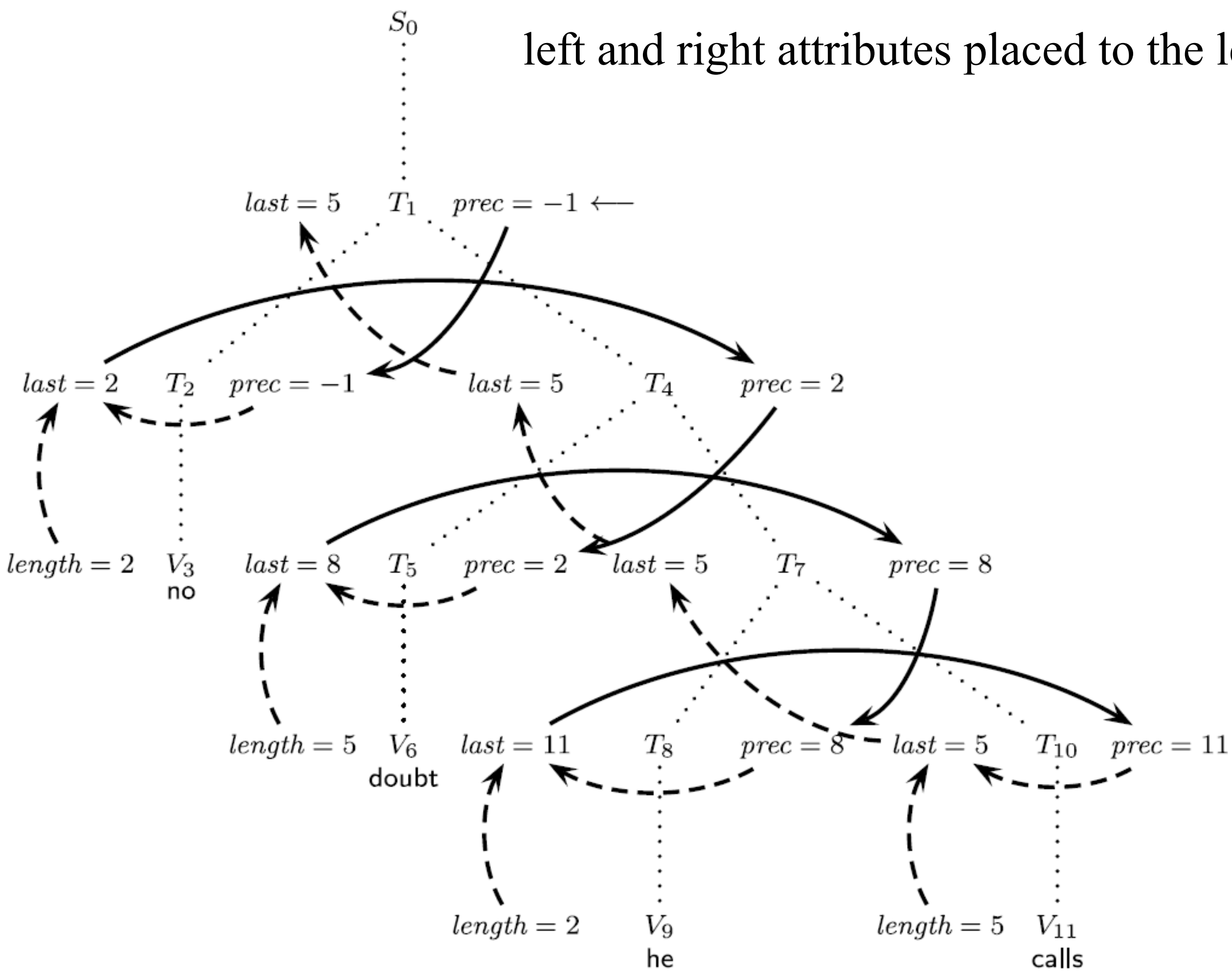
$$\boldsymbol{last(w_k) := prec(w_k)+1+length(w_k)}$$
$$\boldsymbol{prec(w_0) := -1}$$

NB: $\perp$ is the space

#	Syntax	Right attributes	Left attributes
1	$S_0 \to T_1$	$prec_1 := -1$	
2	$T_0 \to T_1 \perp T_2$	$prec_1 := prec_0$ $prec_2 := last_1$	$last_0 := last_2$
3	$T_0 \to V_1$		$last_0 :=$ **if** $(prec_0 + 1 + length_1) \leq W$ **then** $(prec_0 + 1 + length_1)$ **else** $length_1$ **end if**
4	$V_0 \to c V_1$		$length_0 := length_1 + 1$
5	$V_0 \to c$		$length_0 := 1$

Graph for the attribute dependences

left and right attributes placed to the left and right of their nodes



the dependence graph has no circuits

Any sequence of attribute computations that complies with the dependences is suitable to evaluate the attributes

Set of ***semantic functions*** (or ***rules***)

every function is associated with a syntax rule p, called its ***syntax support*** :

$$p: D_0 \rightarrow D_1 D_2 \ldots D_r \qquad r \geq 0$$

a semantic function: $\alpha_k := f(\, attr(\, \{D_0, D_1, \ldots, D_r\} \setminus \{\alpha_k\}\,)\,), \quad 0 \leq k \leq r$

assigns a value to α_k (attribute of symbol D_k)

function f with arguments the *other* attributes of the same rule p (not α_k - no recursion)

semantic functions must be total and computable

$\Rightarrow$ they must be used as computation rules

semantic functions are written using notations taken from

software specification languages, or

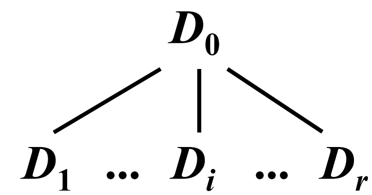
algebra, or

pseudocode

$$p: D_0 \rightarrow D_1\, D_2 \ldots D_i \ldots D_r \qquad r \geq 0$$

$\sigma_0 := f(\ldots)$ defines a left attribute (of the parent, in the tree portion matching the applied rule)
$\delta_i := f(\ldots)$, with $1 \leq i \leq r$, defines a right attribute (of a child, in the same tree portion)

Attributes of terminal symbols (the tree leaves), often
are assigned their value by the lexical analysis
or they may take as value the terminal itself

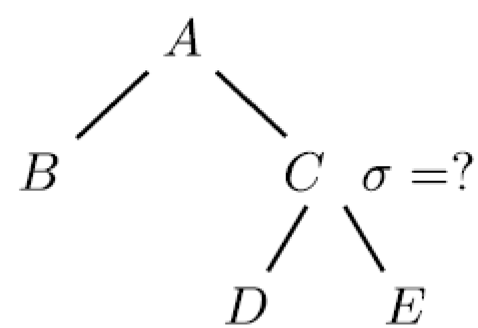


The left attributes of D_0 and the right ones of D_i, $i \geq 1$, are called ***internal*** for rule ***p***
the semantic functions for a rule ***p*** define ***all*** and ***only*** the rule's internal attributes

The right attributes D_0 and the left ones of D_i, $i \geq 1$, are called ***external*** for rule ***p***
they are defined by semantic functions applied to other parts of the tree

An attribute ***cannot*** be ***right*** for one rule ***and left*** for another one
otherwise it would not be uniquely defined, and conflicts may arise

#	Support	Semantic functions
1	$A \rightarrow BC$	$\sigma_C := f_1\,(attr(A, B))$
2	$C \rightarrow DE$	$\sigma_C := f_2\,(attr(D, E))$



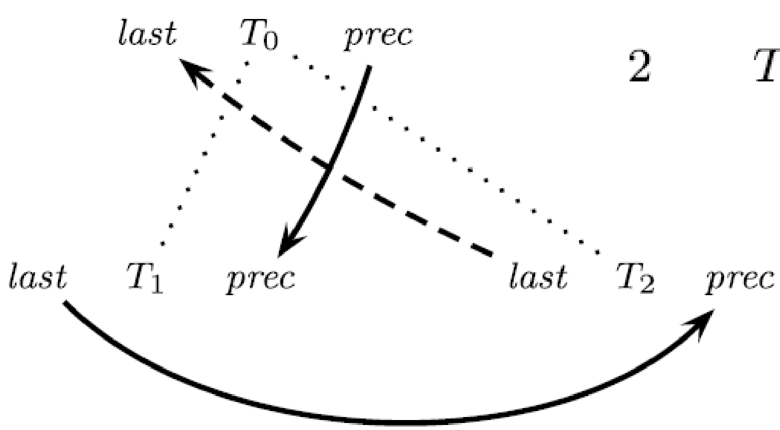
Dependence graph (relation) $\boldsymbol{dep_p}$ for the attributes associated with a syntax rule $\boldsymbol{p}$

It is a directed graph

- the nodes are the attributes (the arguments and the results of semantic functions)
- there is an arc from every argument to the result
- left (synthesized) attributes placed to the left of tree node, right (inherited) ones to the right

The graph is superimposed to that of the syntax support

Example for rule 2 (for simplicity the terminal $\perp$ is omitted)

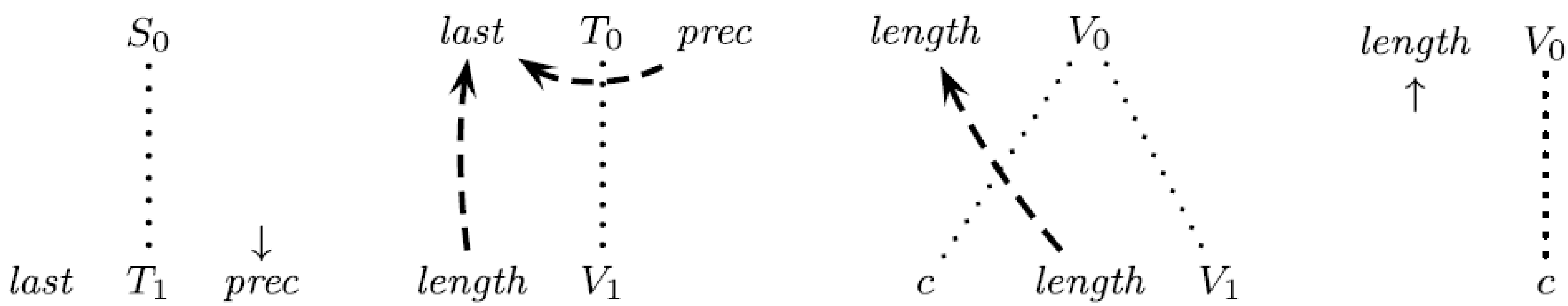


2 $\quad T_0 \rightarrow T_1 \perp T_2 \quad prec_1 := prec_0$

$prec_2 := last_1 \qquad last_0 := last_2$

left attribute: upward or leftward arrows

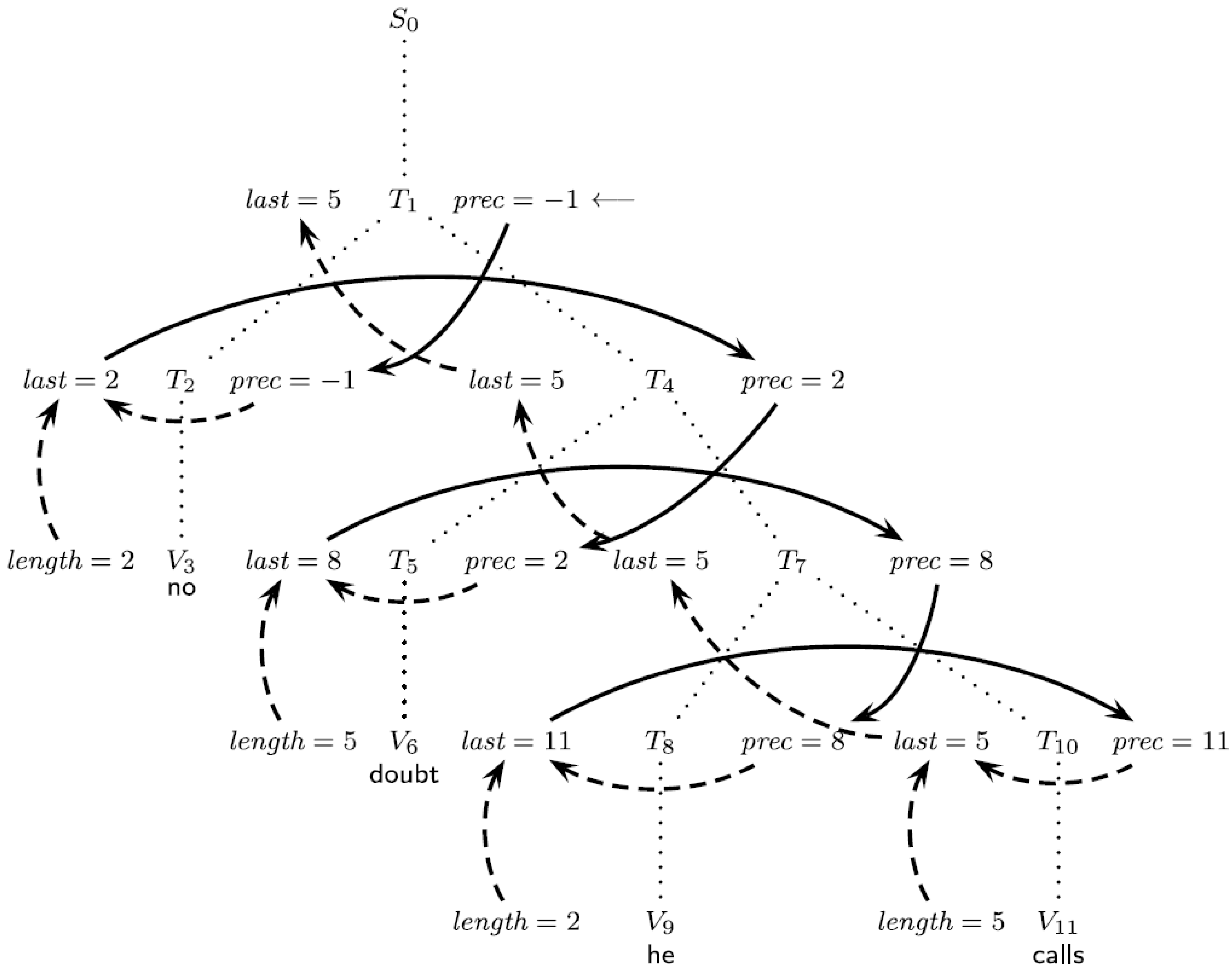
right attribute: down or sideways arrows *dependence graphs of the remaining productions*



#	Syntax	Right attributes	Left attributes
1	$S_0 \to T_1$	$prec_1 := -1$	
2	$T_0 \to T_1 \perp T_2$	$prec_1 := prec_0$ $prec_2 := last_1$	 $last_0 := last_2$
3	$T_0 \to V_1$		$last_0 :=$ **if** $(prec_0 + 1 + length_1) \leq W$ **then** $(prec_0 + 1 + length_1)$ **else** $length_1$ **end if**
4	$V_0 \to cV_1$		$length_0 := length_1 + 1$
5	$V_0 \to c$		$length_0 := 1$

Dependence graph for attributes of an entire syntax tree

Obtained by combining the graphs of the rules used in the various tree nodes



Existence and unicity of the solution:

If the dependence graph of the tree is acyclic
$\Rightarrow$ there exists a set of attribute values consistent with the dependences

(we consider this a self-evident property)

A grammar is called loop-free
if the dependence graph of every tree is acyclic

We consider only loop free grammars
(later we provide a sufficient condition to ensure that the grammar is loop-free)

For a given tree, to compute the attribute values
one must provide a total order of the attributes
so that every attribute is computed only after those preceding it in the dependence relation

To this purpose one could use the ***Topological Sorting*** algorithm (known in the literature)

However this method is not efficient: one should apply the sorting algorithm
before computing the attribute values

Another problem: how to determine if the grammar is loop-free
? how can one ensure that the dependence graph of **every *possible* string** is acyclic ?

The languages of interest are typically infinite $\Rightarrow$ one cannot execute an exhaustive test

The property is decidable but ....

... the problem of deciding whether a grammar is loop-free
is NP-complete w.r.t. the grammar size

Alternative, more efficient though less general idea: fixed scheduling visit and computation

A faster evaluator based on the idea of ***predetermining***
a fixed sequence of visit (scheduling)
which is valid for every tree,
according to functional dependence among attributes

in practice: one provides some (general enough) ***sufficient conditions*** ensuring that

- the grammar is loop-free
- all attribute values can be computed through a ***depth-first visit of the tree***

depth-first visit of the tree: implemented through recursive procedures

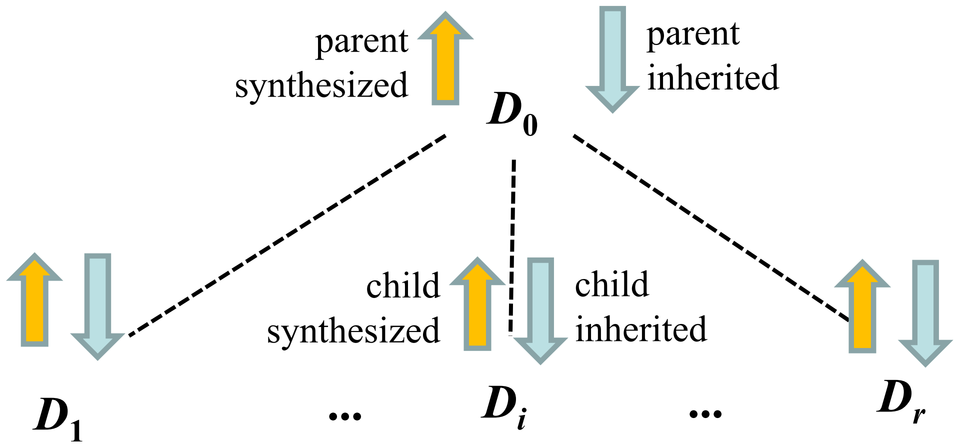
visit of a subtree ⇔ procedure call with the subtree root as parameter

1. Start from the tree root (grammar axiom)
2. the depth-first visit of a (sub)tree includes (recursively) the depth-first visit of the subtrees rooted in its child nodes (in some specified order, e.g., left-to-right)

For each subtree t_N rooted at a node N :

3. Before visiting t_N compute the ***right attributes*** of node N and pass them as the ***input parameters*** of the procedure that implements the visit;
   procedure calls with input parameter passing are the «descending phase» of the visit

4. At the end of the visit of subtree t_N the ***left attributes*** of N become available :
   they are the ***output parameters*** of the procedure that implements the visit;
   procedure return and output parameter passing are the «ascending phase» of the visit



NB: order of subtree visits of the various children is specific for each rule

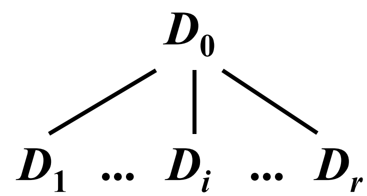
We now provide sufficient conditions on attribute dependences that permit attribute evaluation by a depth-first tree visit

Four Conditions allowing for attribute evaluation through a depth-first visit

they must be checked on the dependence graph $\boldsymbol{dep_p}$ of every syntax rule $\boldsymbol{p}$

> 1. The graph $\boldsymbol{dep_p}$ has no circuit

obviously necessary for the grammar to be loop-free



> 2. In the graph $\boldsymbol{dep_p}$ there exists no path $\sigma_i \rightarrow \ldots \rightarrow \delta_i$, with $i \geq 1$,
> from a ***left attribute*** σ_i to a ***right attribute*** δ_i
> both associated with the same symbol D_i in the right part of p

because δ_i is an input parameter, and σ_i an output parameter,
of the recursive call that visits subtree rooted at D_i

> 3. In the graph $\boldsymbol{dep_p}$ there exists no arc $\sigma_0 \rightarrow \delta_i$ ($i \geq 1$)
> from a left attribute of the father D_0 to a right attribute of any child D_i

because σ_0 is the output parameter of the procedure call for the parent node D_0

… to define an order in the recursive calls on the child nodes $\boldsymbol{D_1, \ldots, D_r}$

We introduce the fourth condition, using an additional definition

w.r.t. syntax rule $\boldsymbol{p: D_0 \rightarrow D_1 D_2 \ldots D_r}$, with $\boldsymbol{r \geq 1}$, we define

binary relation $\boldsymbol{sibl_p}$ called the ***sibling graph*** among right part ***symbols*** $\boldsymbol{\{D_1, D_2, \ldots, D_r\}}$

In $\boldsymbol{sibl_p}$ there exists an arc $\boldsymbol{D_i \rightarrow D_j}$, with $\boldsymbol{i \neq j}$, if and only if

there is a dependence for an attribute of $\boldsymbol{D_i}$ to an attribute of $\boldsymbol{D_j}$, that is,

the dependence graph $\boldsymbol{dep_p}$ has an arc $\alpha_i \rightarrow \beta_j$, with $\alpha_i \in \mathbf{attr}(\boldsymbol{D_i})$ and $\beta_j \in \mathbf{attr}(\boldsymbol{D_j})$

The fourth and last condition is:

4. The graph $\boldsymbol{sibl_p}$ has no circuit

hence on can define an order in the recursive calls on the child nodes $\boldsymbol{D_1, \ldots, D_r}$

attribute evaluation through a depth-first visit also called ***one sweep evaluation***

the conditions 1 – 4 above are collectively called ***one sweep*** **(*evaluation*)** ***condition***

CONSTRUCTION OF THE ONE-SWEEP EVALUATOR

One procedure for each nonterminal; its input parameters are :

- the subtree rooted at the nonterminal
- the right attributes of the subtree root node

The procedure

- visits the subtree, computes its attributes and
- returns the left attributes of the root (through the output parameters)

Construction in 3 steps of the semantic evaluation procedure for rule

$\boldsymbol{p: D_0 \rightarrow D_1 D_2 \ldots D_r, \quad r \geq 1}$

1. Choose a ***Topological Order of Siblings*** $\boldsymbol{D_1, D_2, \ldots, D_r}$, ***TOS***, compatible with the sibling graph $\boldsymbol{sibl_p}$

2. For each symbol $\boldsymbol{D_i}$, with $\boldsymbol{1 \leq i \leq r}$, choose a ***Topological Order of Right attributes***, ***TOR***, of symbol $\boldsymbol{D_i}$

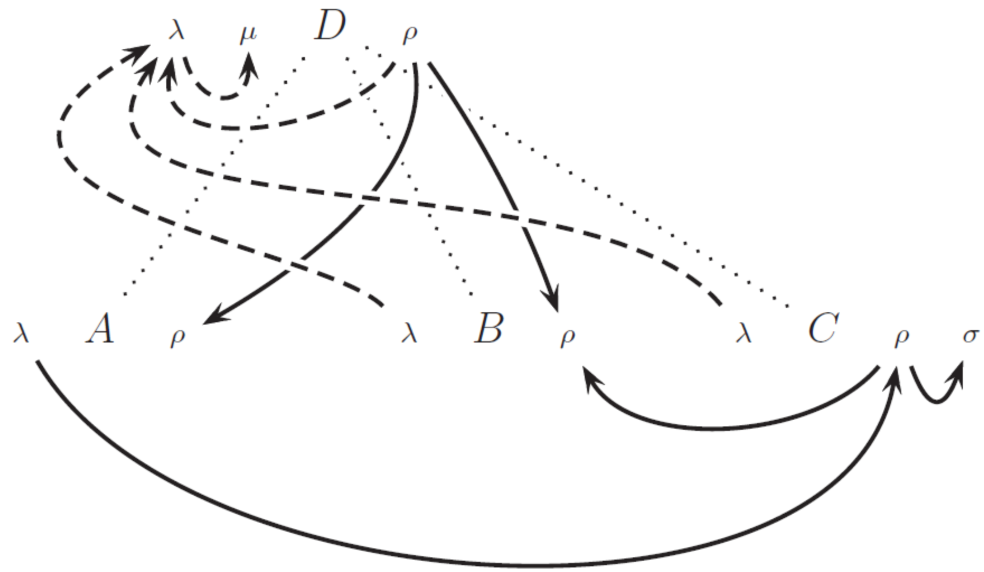
3. Choose a ***Topological Order of Left attributes, TOL***, of symbol $\boldsymbol{D_0}$

The three orders ***TOS***, ***TOR*** and ***TOL*** determine the instruction sequence in the procedure body (shown in the coming example)

Example of a one-sweep procedure

Given a syntax rule ***p*** and the dependence graph $\boldsymbol{dep_p}$:

$\boldsymbol{p}$: $\boldsymbol{D \rightarrow A\ B\ C}$



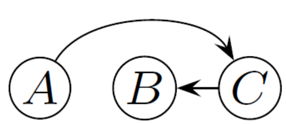
It satisfies the four conditions for attribute evaluation through a depth-first visit

1. $\boldsymbol{dep_p}$ has no circuits

2. $\boldsymbol{dep_p}$ has no path of type $\sigma_i \rightarrow \ldots \rightarrow \delta_i$, $\boldsymbol{i \geq 1}$

3. $\boldsymbol{dep_p}$ has no arcs of type $\sigma_0 \rightarrow \delta_i$, with $\boldsymbol{i \geq 1}$

4. $\boldsymbol{sibl_p}$ is acyclic

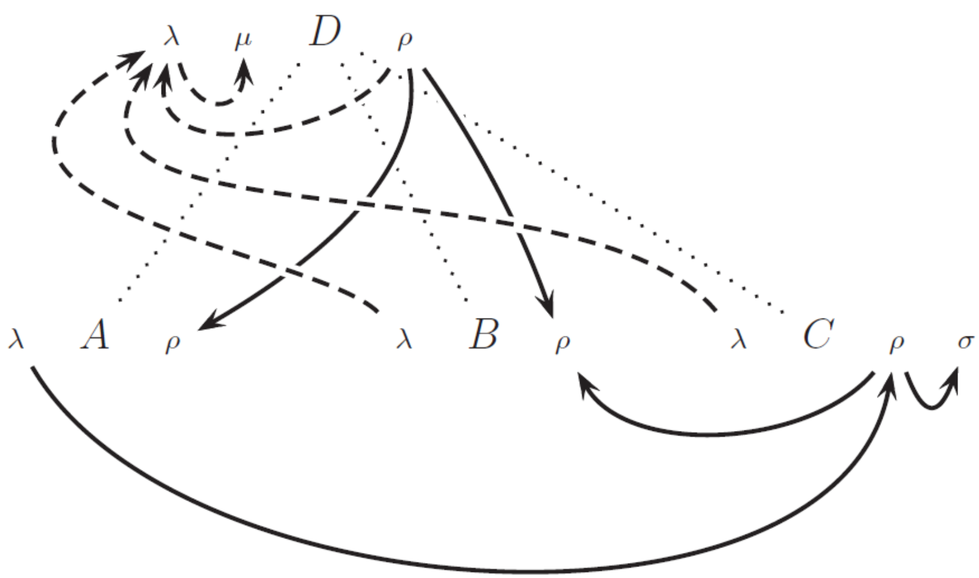


$A \rightarrow C$ derives from dependence $\lambda_A \rightarrow \rho_C$

$C \rightarrow B$ derives from dependence $\rho_C \rightarrow \rho_B$

Here are the possible topological orders

- sibling graph: ***TOS* = *A*, *C*, *B***
- right attributes of every child:
  TOR for ***A*** = ρ; ***TOR*** for ***B*** = ρ; ***TOR*** for ***C* = ρ, σ;**
- left attributes of ***D***: ***TOL* = λ, μ**



procedure D (**in** t, ρ_D; **out** λ_D, μ_D)

- - t root of subtree to be decorated

$\rho_A := f_1(\rho_D)$

- - abstract functions are denoted f_1, f_2, etc.

$A(t_A, \rho_A; \lambda_A)$

- - invocation of A to decorate subtree t_A

$\rho_C := f_2(\lambda_A)$

$\sigma_C := f_3(\rho_C)$

$C(t_C, \rho_C, \sigma_C; \lambda_C)$

- - invocation of C to decorate subtree t_C

$\rho_B := f_4(\rho_D, \rho_C)$

$B(t_B, \rho_B; \lambda_B)$

- - invocation of B to decorate subtree t_C

$\lambda_D := f_5(\rho_D, \lambda_B, \lambda_C)$

$\mu_D := f_6(\lambda_D)$

end procedure

Now we can introduce and motivate an «attribute grammar design hint»:

when an initial ($\Rightarrow$ inherited) attribute is needed in the root ***S*** of the tree (like an initialization) then one adds a «new spurious» axiom ***S'*** and a rule ***S'→S***

… this occurs in the previous example of text segmentation (slide 8, attribute ***prec*** in rule *S*→*T* ; it is the only reason for having an axiom *S* distinct from *T*)

Combined Syntax and Semantic Analysis

Syntax and semantic analysis can be integrated into the parser

Simple and efficient method, suitable for simple translations

Various cases, depending on the nature of the source language

- regular source language: lexical analysis with attribute evaluation

  can be performed with tools such as ***flex*** or ***lex***

- ***LL*(*k*)** syntax: recursive top-down parser with attributes

  can be implemented manually with left (synthesized) attributes only

- ***LR*(*k*)** syntax: shift-reduce parser with attributes

  can be performed with tools such as ***bison*** or ***yacc***
  (NB: functional dependence among right attributes is strongly limited)

Attributed recursive descent translator

Several hypotheses must be satisfied

- syntax suitable for deterministic top-down analysis (*LL*)
- attribute grammar suitable for one-sweep evaluation (depth-first visit)
- ***further*** conditions on functional dependence among attributes … that we see now

Top-down analysis builds the subtrees from left to right

If combined with attribute evaluation then …

…attribute dependences must permit a visit of subtrees in the sequence from left to right: $\mathbf{1, 2, \ldots, \boldsymbol{r} - 1, \boldsymbol{r}}$

Therefore: Condition ***L*** (***left-to-right***) for syntax/semantic recursive descent analysis

1. Conditions allowing for ***one sweep evaluation*** through depth-first visit, plus
2. The sibling graph $\boldsymbol{sibl_p}$ for rule $\boldsymbol{D_0 \rightarrow D_1 \ldots D_r}$ allows one to choose as ***TOS*** the "natural" sequence $\boldsymbol{D_1, D_2, \ldots, D_r}$
   i.e., $\boldsymbol{sibl_p}$ must not include any arc $\boldsymbol{D_j \rightarrow D_i}$ with $\boldsymbol{j > i}$:
   no attribute of $\boldsymbol{D_i}$ can depend on an attribute of $\boldsymbol{D_j}$ with $\boldsymbol{D_j}$ placed to the right of $\boldsymbol{D_i}$

if a grammar is $\boldsymbol{LL(k)}$ and satisfies the ***L*** condition $\Rightarrow$

$\Rightarrow$ build a deterministic recursive descent parser that also evaluates the attributes

Example of a recursive descent syntax-semantic analyzer

Computes the numeric value of a binary string
encoding a value less than 1

Language: $L = \bullet(0 \mid 1)^+$ Translation (ex.): $\tau(\bullet 01) = 0{,}25$

Grammar

Syntax	Left attributes	Right attributes
$N_0 \to \bullet D_1$	$v_0 := v_1$	$l_1 := 1$
$D_0 \to B_1 D_2$	$v_0 := v_1 + v_2$	$l_1 := l_0 \quad l_2 := l_0 + 1$
$D_0 \to B_1$	$v_0 := v_1$	$l_1 := l_0$
$B_0 \to 0$	$v_0 := 0$	
$B_0 \to 1$	$v_0 := 2^{-l_0}$	

Attributes

Attribute	Meaning	Domain	Type	Assoc. symbols
v	Value	Real	Left	N, D, B
l	Length	Integer	Right	D, B

The value of each bit is weighted by a power of 2 with negative exponent = distance from the fractional point

The syntax is deterministic ***LL*(2):** lookahead=2 needed for nonterminal ***D***

Check the L condition for every syntax rule

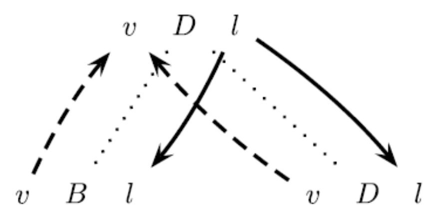
$N \rightarrow \bullet D$: $v_0 := v_1$ $l_1 := 1$

the dependence graph ***dep*** has the only arc $v_1 \rightarrow v_0$,

hence the L condition is satisfied, because

1. the graph has no circuit
2. there is no path from a left attribute v to a right attribute l of the same child
3. there is no arc from a left attribute v of the father to a right attribute l of a child
4. the sibling graph ***sibl*** has no arc

$D \rightarrow B\ D$: $v_0 := v_1 + v_2$ $l_1 := l_0$ $l_2 := l_0 + 1$



the dependence graph

– has no circuit

– there is no path from a left attribute v to a right attribute l of the same child

– no arc from a left attr. (v) of the father to a right attr. l of a child

– the sibling graph ***sibl*** has no arc

$D \rightarrow B$: $v_0 \coloneqq v_1$, same as above

$B \rightarrow 0$: $v_0 \coloneqq 0$, dependence graph has no arc

$B \rightarrow 1$: $v_0 \coloneqq 2^{-l_0}$ ***dep*** graph has a unique arc $l_0 \rightarrow v_0$ and satisfies the L condition

Integrated syntax - semantic procedure

- in parameters: right attributes of the father
- out parameters: left attributes of the father
- variables ***cc1*** and ***cc2***: the current terminal symbol and the next one (syntax is ***LL*(2)**)
- some local variables to pass the attribute values to other internal procedures
- «***read***» function updates ***cc1*** and ***cc2*** (NB: syntax is ***LL*(2)** but not ***LL*(1)**)

procedure N (**in** Ø; **out** v_0)
    if $cc1 =$ ‘ • ’ **then**
        read
    else
        error
    end if
    $l_1 := 1$ - - initialize a local var. with right attribute of D
    $D(l_1, v_0)$ - - call D to construct a subtree and compute v_0
end procedure

Syntax	Left attributes	Right attributes
$N_0 \to \bullet D_1$	$v_0 := v_1$	$l_1 := 1$

Integrated syntax - semantic procedure

```
procedure B (in l₀; out v₀)
    case cc1 of
        '0' : v₀ := 0            - - case of rule B → 0
        '1' : v₀ := 2^(−l₀)      - - case of rule B → 1
        otherwise error
    end case ;   read
end procedure
```

Grammar		
Syntax	Left attributes	Right attributes
$N_0 \to \bullet D_1$	$v_0 := v_1$	$l_1 := 1$
$D_0 \to B_1 D_2$	$v_0 := v_1 + v_2$	$l_1 := l_0 \quad l_2 := l_0 + 1$
$D_0 \to B_1$	$v_0 := v_1$	$l_1 := l_0$
$B_0 \to 0$	$v_0 := 0$	
$B_0 \to 1$	$v_0 := 2^{-l_0}$	

Integrated syntax - semantic procedure

procedure D (**in** l_0; **out** v_0)
  case $cc2$ **of**
    '0', '1' : **begin** - - case of rule $D \to B\,D$
      $B(l_0, v_1)$
      $l_2 := l_0 + 1$
      $D(l_2, v_2)$
      $v_0 := v_1 + v_2$
    end
    '$\dashv$' : **begin** - - case of rule $D \to B$
      $B(l_0, v_1)$
      $v_0 := v_1$
    end
    otherwise *error*
  end case
end procedure

Grammar		
Syntax	Left attributes	Right attributes
$N_0 \to \bullet D_1$	$v_0 := v_1$	$l_1 := 1$
$D_0 \to B_1 D_2$	$v_0 := v_1 + v_2$	$l_1 := l_0 \quad l_2 := l_0 + 1$
$D_0 \to B_1$	$v_0 := v_1$	$l_1 := l_0$

Example: Code generation for conditional control structures

if-then-else construct is converted to a combination of (conditional) jump instructions

For every generated instruction the translator needs a new label for the instruction targeted by the jump; every label must differ fom previous ones used for other instructions

function ***fresh*** returns, at each invocation, a new integer, to be assigned to variable ***n***, a right attribute of the nonterminal representing the instruction

computed translation assigned to the ***tr*** attribute

concatenation operator (•) to combine the translation of the various fragments

Labels have the form: e397, f397, i23, ...

Example translation (assuming that the current call of ***fresh*** returns 7)

if $(a > b)$	$tr(a > b)$
then	jump-if-false rc, e_7
$\quad a := a - 1$	$tr(a := a - 1)$ jump f_7
else	e_7 :
$\quad a := b$	$tr(a := b)$
end if	f_7 :
...	... - - rest of the program

Grammar of the ***if-then-else*** conditional instruction

Syntax	Semantic functions
$F \to I$	$n_1 := fresh$

$I \to$ **if** ($cond$)
 then
 L_1
 else
 L_2
 end if
	$tr_0 := tr_{cond}$ •
	 jump-if-false rc_{cond}, e_n_0 •
	 tr_{L_1} • jump f_n_0 •
	 e_n_0 : •
	 tr_{L_2} •
	 f_n_0 :

NB: n_0 has the value of n_1 (*fresh*) above

rc_{cond} is an attribute of nonterm. *cond*

translation of ***cond*** (tr_{cond}), L_1 (tr_{L_1}), and L_2 (tr_{L_2}) specified by other rules (not reported)

Proposed exercise: define similarly an attribute grammar for the translation of the iterative ***while*** instruction so to obtain the result here below

while $(a > b)$	i_8: $tr(a > b)$
do	jump-if-false rc, f_8
$\quad a := a - 1$	$tr(a := a - 1)$ jump i_8
end while	f_8:
$\dots$	$\dots$ - - rest of the program