4. Semantic analysis and transformation

Input: **abstract program tree**

Tasks:	**Compiler module:**
name analysis	environment module
properties of program entities	definition module
type analysis, operator identification	signature module
transformation	tree generator

Output: **target tree, intermediate code, target program in case of source-to-source**

Standard implementations and generators for compiler modules

Operations of the compiler modules are called at nodes of the abstract program tree

Model: dependent computations in trees

Specification: **attribute grammars**

generated: tree walking algorithm that calls operations
in specified contexts and in an admissable order

4.1 Attribute grammars

Attribute grammar (AG) specifies **dependent computations in the abstract program tree**
declarative: explicit dependencies only; a suitable order of execution is computed

Computations solve the tasks of semantic analysis and transformation

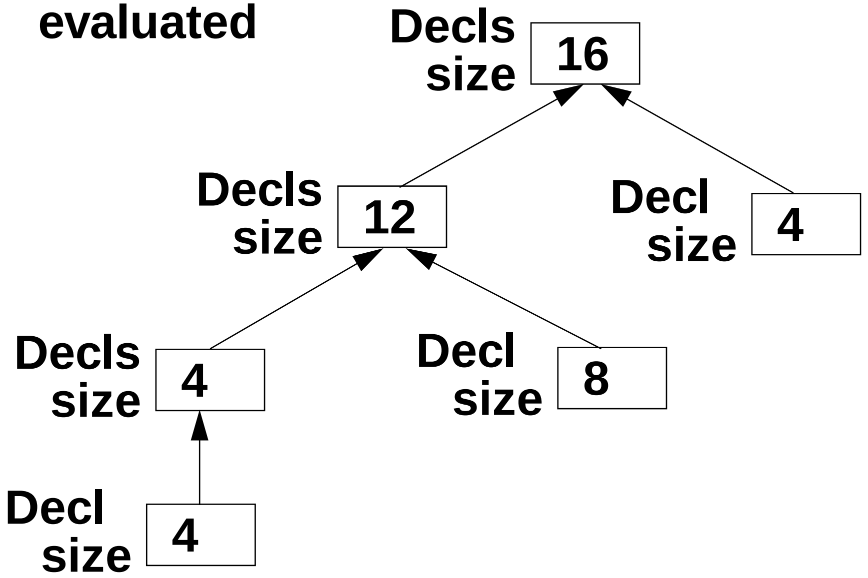
Generator produces **a plan for tree walks**
that execute calls of the computations,
such that the specified dependencies are obeyed,
computed values are propagated through the tree

Result: attribute evaluator; applicable for any tree specified by the AG

Example: attribute grammar

RULE **Decls ::= Decls Decl** COMPUTE
Decls[1].size =
Add (Decls[2].size, Decl.size);
END;
RULE **Decls ::= Decl** COMPUTE
Decls.size = Decl.size;
END;
RULE **Decl ::= Type Name** COMPUTE
Decl.size = ...;
END;

tree with dependent attributes evaluated



Basic concepts of attribute grammars

An AG specifies computations in tree:

expressed by **computations associated to productions of the abstract syntax**

```
RULE p: Y ::= u COMPUTE f(...); g(...); END;
```

computations f(...) and g(...) are executed in every tree context of type p

An AG specifies dependencies between computations:

expressed by **attributes associated to grammar symbols**

```
RULE p: X ::= u Y v COMPUTE      X.b =   f(Y.a);
                                 Y.a =   g(...);
END;
```

post-condition: `Y.a =`; pre-condition: `g(...)` / `f(Y.a)`

`f(Y.a)` uses the result of `g(...)`; hence `Y.a=g(...)` will be executed before `f(Y.a)`

dependent computations in adjacent contexts:

```
RULE r: X ::= v Y w COMPUTE X.b = f(Y.a); END;
RULE p: Y ::= u    COMPUTE Y.a = g(...); END;
```

attributes may specify dependencies without propagating any value:

```
X.GotType = ResetTypeOf(...);
Y.Type = GetTypeOf(...) <- X.GotType;
```

`ResetTypeOf` will be called before `GetTypeOf`

Definition of attribute grammars

An **attribute grammar** is defined by

a **context-free grammar G**, (abstract syntax, tree grammar)

for each **symbol X** of G a set of **attributes A(X)**, written X.a if a $\in$ A(X)

for each **production (rule) p** of G a set of **computations** of one of the forms

X.a = f (... Y.b ...)   or   g (... Y.b ...)   where X and Y occur in p

Consistency and completeness of an AG:

Each A(X) is partitioned into two disjoint subsets: AI(X) and AS(X)

AI(X): **inherited attributes** are computed in rules p where X is on the **right**-hand side of p

AS(X): **synthesized attributes** are computed in rules p where X is on the **left**-hand side of p

Each rule p: X ::= ... Y ... has exactly one computation
for all attributes of AS(X), and
for all attributes of AI(Y), for all symbol occurrences on the right-hand side of p

AG Example: Compute expression values

The AG specifies: The value of an expression is computed and printed:

```
ATTR value: int;

RULE: Root ::=  Expr  COMPUTE
  printf ("value is %d\n",
          Expr.value);
END;

TERM Number: int;

RULE: Expr ::= Number COMPUTE
  Expr.value = Number;
END;

RULE: Expr ::= Expr Opr Expr
COMPUTE
  Expr[1].value = Opr.value;
  Opr.left  = Expr[2].value;
  Opr.right = Expr[3].value;
END;
```

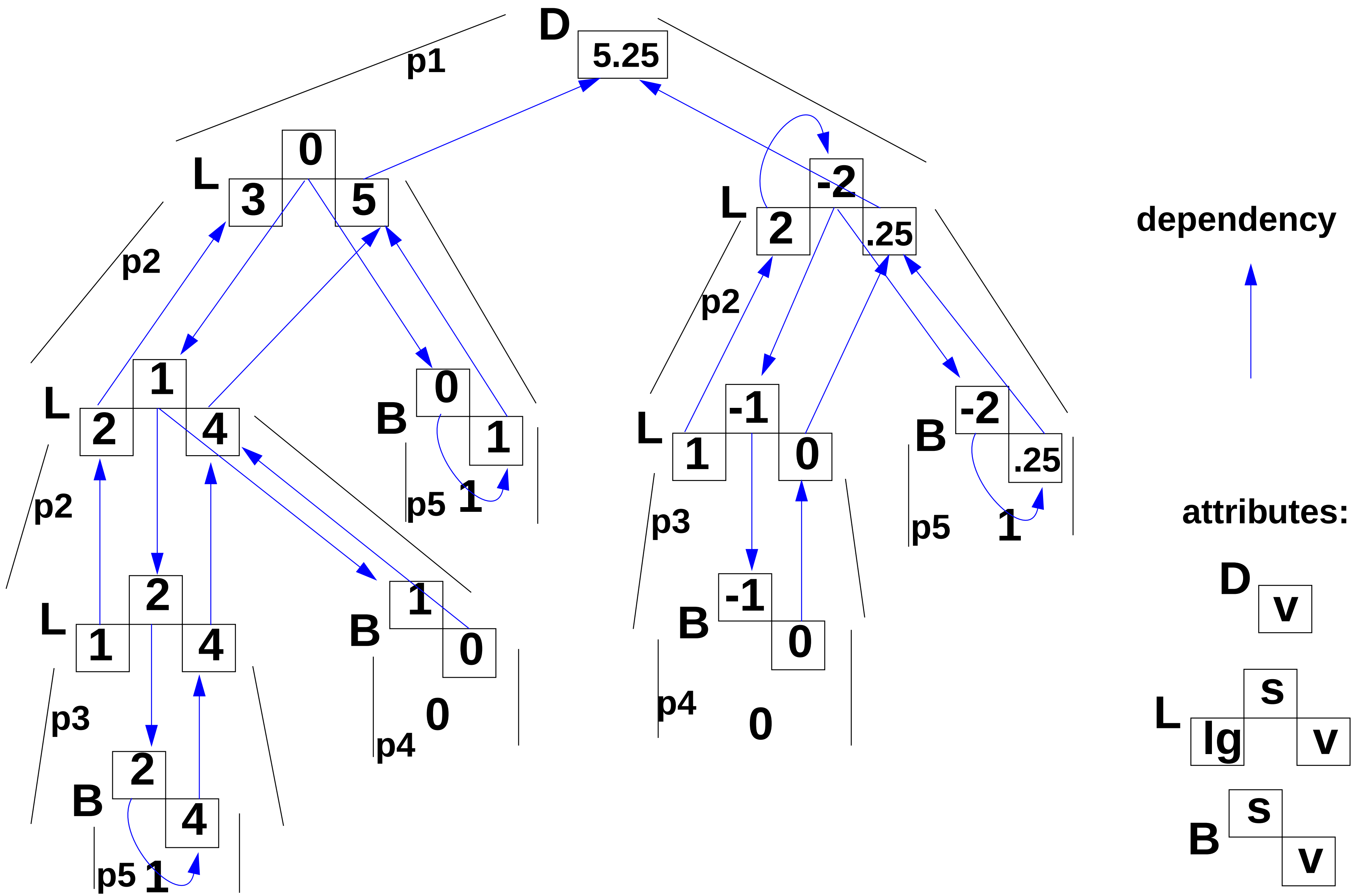
```
SYMBOL Opr: left, right: int;

RULE: Opr ::=  '+'  COMPUTE
  Opr.value  =
     ADD (Opr.left, Opr.right);
END;

RULE: Opr ::=  '*'  COMPUTE
  Opr.value =
     MUL (Opr.left, Opr.right);
END;
```

AG Binary numbers

Attributes:	**L.v, B.v**	value
	L.lg	number of digits in the sequence L
	L.s, B.s	scaling of B or the least significant digit of L

```
RULE p1:  D ::= L '.' L   COMPUTE
   D.v = ADD (L[1].v, L[2].v);
   L[1].s = 0;
   L[2].s = NEG (L[2].lg);
END;
RULE p2:  L ::= L B       COMPUTE
   L[1].v = ADD (L[2].v, B.v);
   B.s = L[1].s;
   L[2].s = ADD (L[1].s, 1);
   L[1].lg = ADD (L[2].lg, 1);
END;
RULE p3:  L ::= B         COMPUTE
   L.v = B.v;
   B.s = L.s;
   L.lg = 1;
END;
RULE p4:  B ::= '0'       COMPUTE
   B.v = 0;
END;
RULE p5:  B ::= '1'       COMPUTE
   B.v = Power2 (B.s);
END;
```

An attributed tree for AG Binary numbers

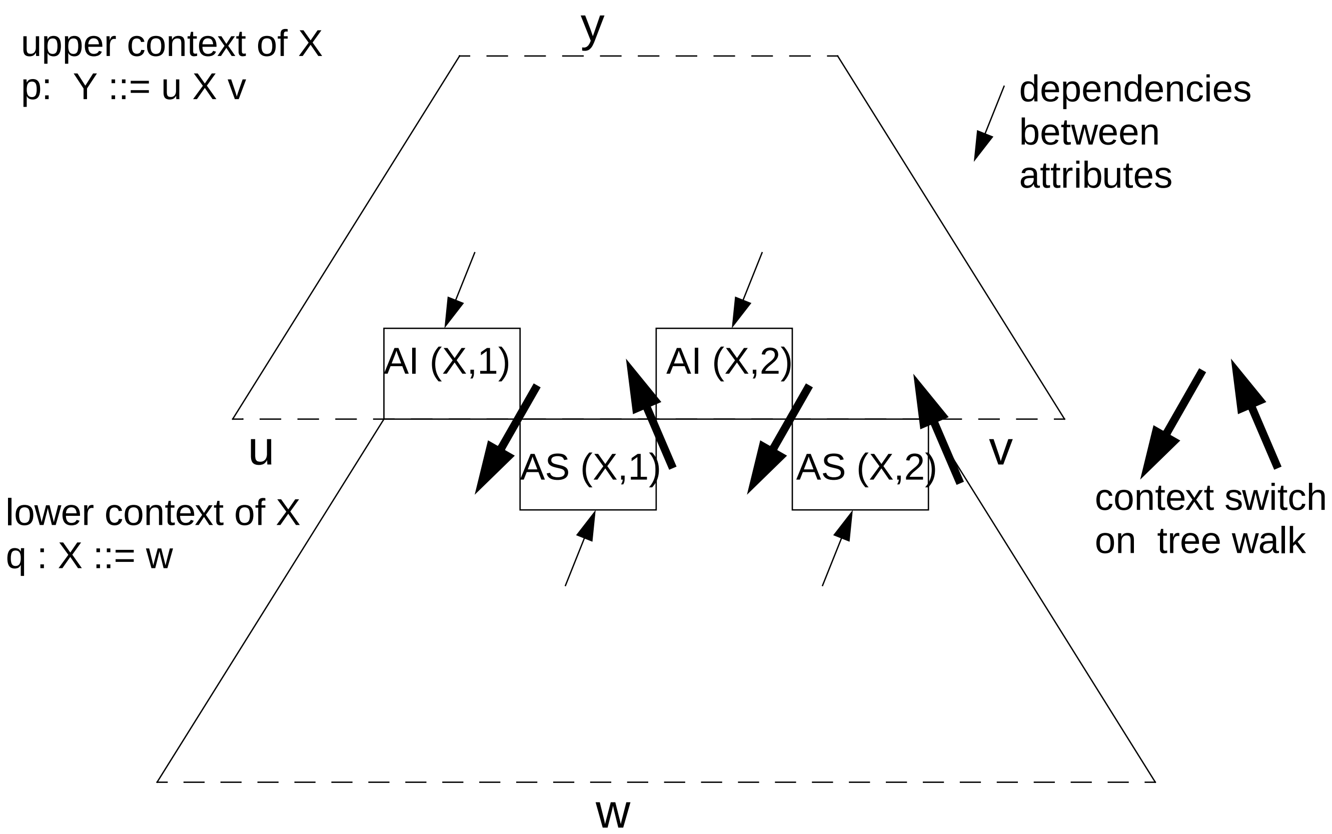


Dependency analysis for AGs

2 disjoint sets of attributes for each symbol X:

AI (X) : inherited (dt. erworben), **computed in upper contexts** of X

AS (X): synthesized (dt. abgeleitet), **computed in lower contexts** of X.



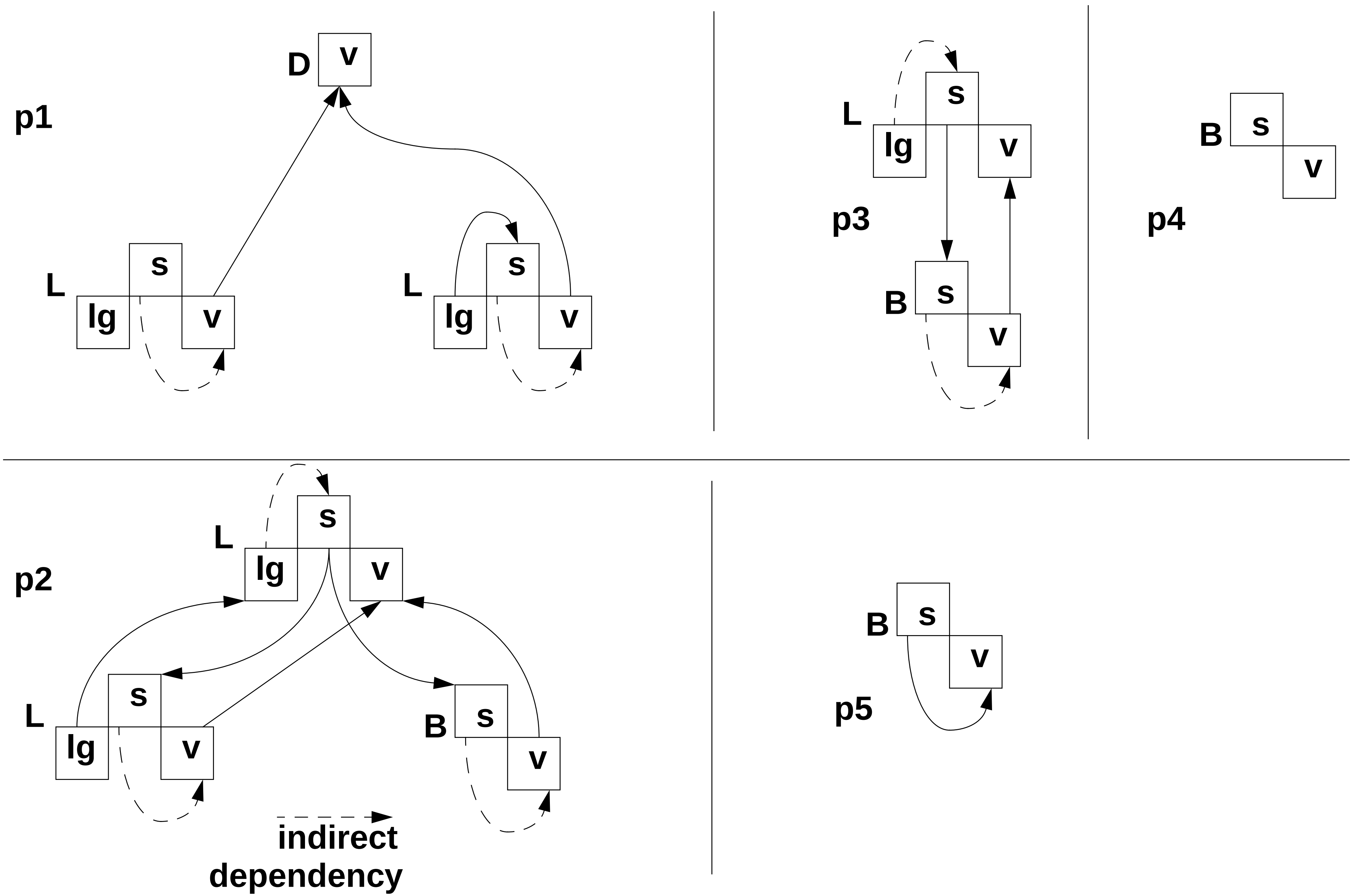
Objective: Partition of attribute sets, such that

AI (X, i) is computed **before the i-th visit** of X

AS (X, i) is computed **during the i-th visit** of X

Necessary precondition for the existence of such a partition:
No node in any tree has direct or indirect dependencies that contradict the evaluation order of the sequence of sets: AI (X, 1), AS (X, 1), ..., AI (X, k), AS (X, k)

Dependency graphs for AG Binary numbers



Construction of attribute evaluators

For a given attribute grammar an attribute evaluator is constructed:

- It is **applicable to any tree** that obeys the abstract syntax specified in the rules of the AG.
- It performs a **tree walk** and **executes computations** when visiting a context for which they are specified.
- The execution order obeys the **attribute dependencies**.

Pass-oriented strategies for the tree walk:	**AG class**
k times **depth-first left-to-right**	**LAG (k)**
k times depth-first **alternatingly left-to-right / right-to left**	**AAG (k)**
once **bottom-up**	**SAG**

The attribute dependencies of the AG are checked
whether the desired pass-oriented strategy is applicable; see LAG(k) algorithm.

non-pass-oriented strategies:
visit-sequences: **OAG**
an individual plan for each rule of the abstract syntax

Generator fits the plans to the dependencies.

Visit-sequences

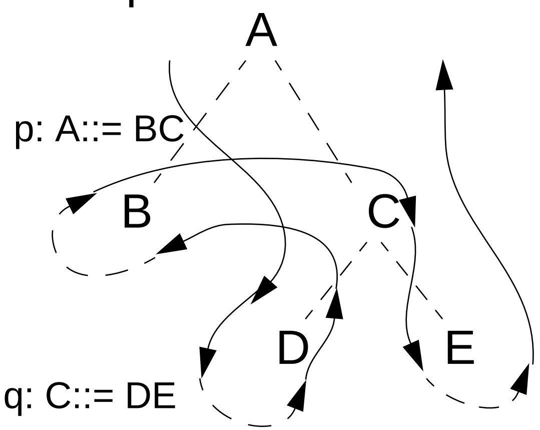
A **visit-sequence** (dt. Besuchssequenz) vs_p **for each production** of the tree grammar:

$$p: X_o ::= X_1 \;...\; X_i \;...\; X_n$$

A visit-sequence is a **sequence of operations**:

- $\downarrow i, j$ — j-th **visit of the i-th subtree**
- $\uparrow j$ — j-th r**eturn to the ancestor** node
- $eval_c$ — execution of a **computation** c associated to p

Example in the tree:

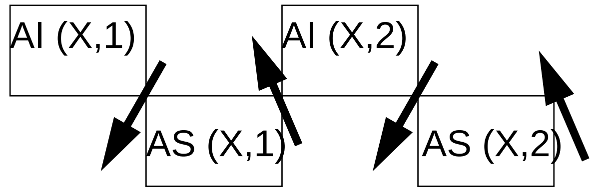


visit-sequences

vs_p: ... $\downarrow$C,1 ...$\downarrow$B,1 ...$\downarrow$C,2 ...$\uparrow$1

vs_q: ... $\downarrow$D,1 ... $\uparrow$1 ... $\downarrow$E,1 ... $\uparrow$2

attribute partitions guaranty correct interleaving:



Implementation:

one procedure for each section of a visit-sequence upto $\uparrow$

a call with a switch over applicable productions for $\downarrow$

Visit-sequences for the AG Binary numbers

vs_{p1}: D ::= L '.' L

↓L[1],1; L[1].s=0; ↓L[1],2; ↓L[2],1; L[2].s=NEG(L[2].lg);

↓L[2],2; D.v=ADD(L[1].v, L[2].v); ↑1

vs_{p2}: L ::= L B

↓L[2],1; L[1].lg=ADD(L[2].lg,1); ↑1

L[2].s=ADD(L[1].s,1); ↓L[2],2; B.s=L[1].s; ↓B,1; L[1].v=ADD(L[2].v, B.v); ↑2

vs_{p3}: L ::= B

L.lg=1; ↑1; B.s=L.s; ↓B,1; L.v=B.v; ↑2

vs_{p4}: B ::= '0'

B.v=0; ↑1

vs_{p5}: B ::= '1'

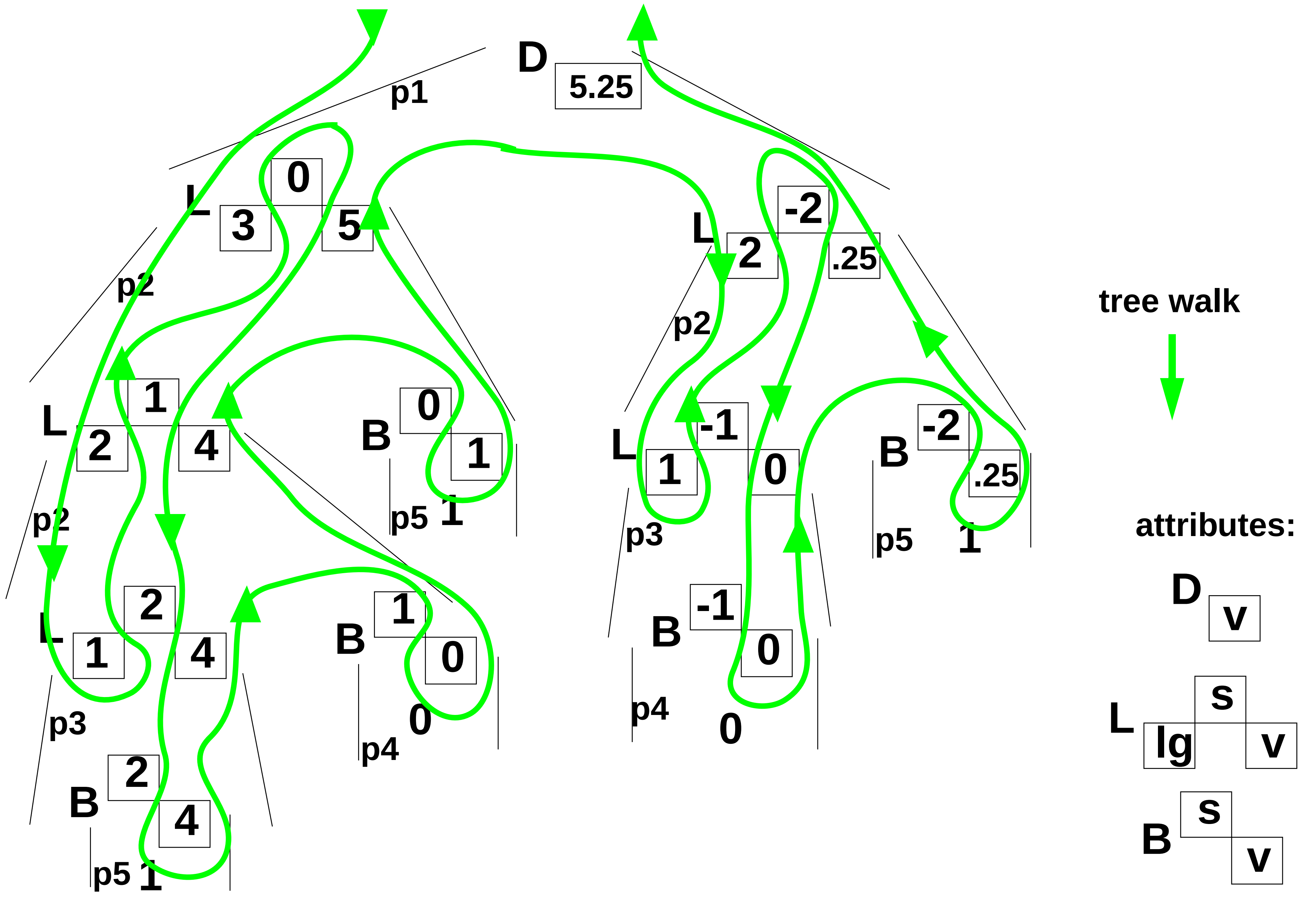
B.v=Power2(B.s); ↑1

Implementation:

Procedure vs<i><p> for each section of a vs_p to a ↑i

a call with a switch over alternative rules for ↓X,i

Tree walk for AG Binary numbers

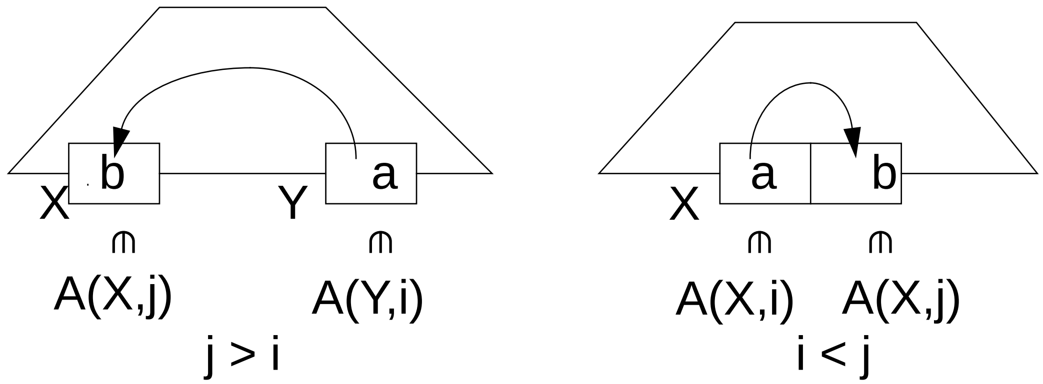


LAG (k) condition and algorithm

An AG is a LAG(k), if: For each symbol X there is an attribute partition A (X,1), ..., A (X, k), such that the attributes in A (X, i) can be computed in the i-th depth-first left-to-right pass.

Necessary and sufficient condition over dependency graphs - expressed graphically:

A dependency **from right to left**



A dependency **at one symbol** on the right-hand side

Algorithm: computes A (1), ..., A (k), if the AG is LAG(k), for i = 1, 2, ...

A (i) := all attributes that are not yet assigned

remove attributes from A(i) as long as the following rules are applicable:

- remove X.b, if there is a context where it depends on an attribute of A (i) according to the pattern given above,
- remove Z.c, if it depends on a removed attribute

Finally: all attributes are assigned to a passes i = 1, ..., k the AG **is LAG(k)**
all attributes are removed from A(i) the AG **is not LAG(k) for any k**

Generators for attribute grammars

LIGA	University of Paderborn	OAG
FNC-2	INRIA	ANCAG (Oberklasse von OAG)
Synthesizer Generator	Cornell University	OAG, inkrementell
CoCo	Universität Linz	LAG(1)

Properties of the generator LIGA

- integrated **in the Eli system**, cooperates with other Eli tools
- **high level specification language** Lido
- modular and **reusable AG components**
- object-oriented constructs usable for **abstraction of computational patterns**
- computations are **calls of functions** implemented outside the AG
- **side-effect computations** can be controlled by dependencies
- notations for **remote attribute access**
- **visit-sequence** controlled attribute evaluators, implemented in C
- **attribute storage optimization**

State attributes without values

```
RULE: Root ::= Expr COMPUTE
  Expr.print = "yes";
  printf ("\n") <- Expr.printed;
END;

RULE: Expr ::= Number COMPUTE
  Expr.printed =
    printf ("%d ", Number) <- Expr.print;
END;

RULE: Opr  ::= '+' COMPUTE
  Opr.printed = printf ("+ ") <- Opr.print;
END;

RULE: Opr  ::= '*' COMPUTE
  Opr.printed = printf ("* ") <- Opr.print;
END;

RULE: Expr  ::= Expr Opr Expr COMPUTE
  Expr[2].print = Expr[1].print;
  Expr[3].print = Expr[2].printed;
  Opr.print = Expr[3].printed;
  Expr[1].printed = Opr.printed;
END;
```

The attributes **`print`** and **`printed`** do not have a value

They just describe pre- and post-conditions of computations:

`Expr.print:`
postfix output has been done up to not including this node

`Expr.printed:`
postfix output has been done up to including this node

Dependency pattern CHAIN

```
CHAIN print: VOID;

RULE: Root ::= Expr COMPUTE
  CHAINSTART HEAD.print = "yes";
  printf ("\n ") <- TAIL.print;
END;

RULE: Expr ::= Number COMPUTE
  Expr.print =
    printf ("%d ", Number) <- Expr.print;
END;

RULE: Opr ::= '+' COMPUTE
  Opr.post = printf ("+") <- Opr.pre;
END;

RULE: Expr ::= Expr Opr Expr COMPUTE
  Opr.pre = Expr[3].print;
  Expr[1].print = Opr.post;
END;
```

A CHAIN specifies a **left-to-right depth-first** dependency through a subtree.

Trivial computations of the form X.a = Y.b in the CHAIN order can be **omitted**. They are added as needed.

Dependency pattern INCLUDING

```
ATTR depth: int;

RULE: Root ::= Block COMPUTE
  Block.depth = 0;
END;

RULE: Statement ::= Block COMPUTE
  Block.depth =
     ADD (INCLUDING Block.depth, 1);
END;

TERM Ident: int;

RULE: Definition ::= ‘define' Ident  COMPUTE
  printf ("%s defined on depth %d\n ",
           StringTable (Ident),
           INCLUDING Block.depth);
END;
```

`INCLUDING Block.depth`
accesses the **`depth`** attribut of the next upper node of type **`Block`.**

An **attribute** at the root of a subtree is **used from within the subtree**.

Propagation through the contexts in between is **omitted**.

Dependency pattern CONSTITUENTS

```
RULE: Block ::= '{' Sequence '}' COMPUTE
  Block.DefDone =
    CONSTITUENTS Definition.DefDone;
END;
RULE: Definition ::= 'Define' Ident COMPUTE
  Definition.DefDone =
    printf ("%s defined in line %d\n",
            StringTable(Ident), LINE);
END;
RULE: Usage ::= 'use' Ident COMPUTE
   printf ("%s used in line %d\n ",
           StringTable(Ident), LINE),
   <- INCLUDING BLOCK.DefDone;
END;
```

`CONSTITUENTS Definition.DefDone` accesses the **`DefDone`** attributes of all **`Definition`** nodes in the subtree below this context

A computation **accesses attributes from the subtree below** its context.

Propagation through the contexts in between is **omitted**.

The shown combination with INCLUDING is a common dependency pattern.