4. Attribute grammars and semantic analysis

Input: **abstract program tree**

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

Output: **attributed program tree**

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: a **tree walking algorithm** that calls functions of semantic modules **in specified contexts** and in an **admissible order**

4.1 Attribute grammars

Attribute grammar (AG): specifies **dependent computations in abstract program trees;**
declarative: explicitly specified dependences only; a suitable order of execution is computed

Computations solve the tasks of semantic analysis (and transformation)

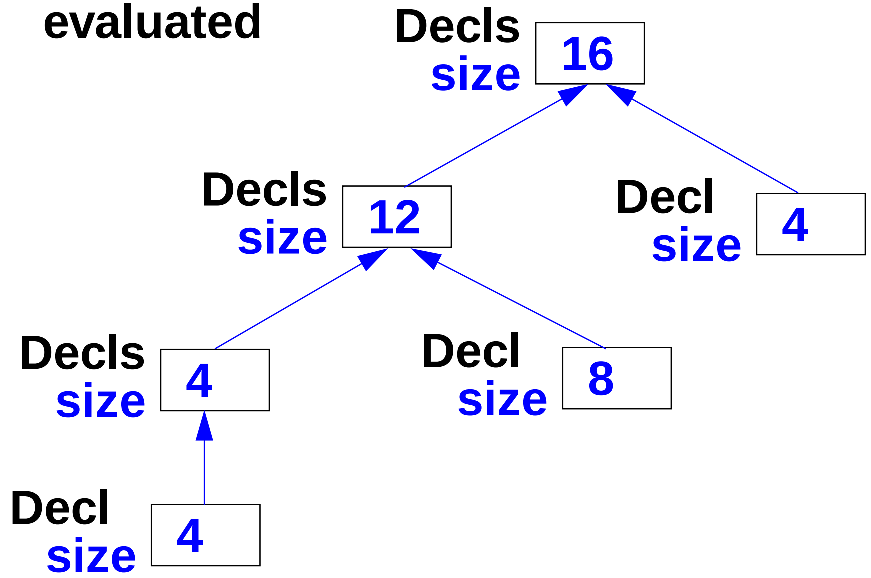
Generator produces **a plan for tree walks**
that execute calls of the computations,
such that the specified dependences are obeyed,
computed values are propagated through the tree

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

Example: AG specifies size of declarations

```
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 = Type.size;
END;
```

tree with dependent attributes evaluated

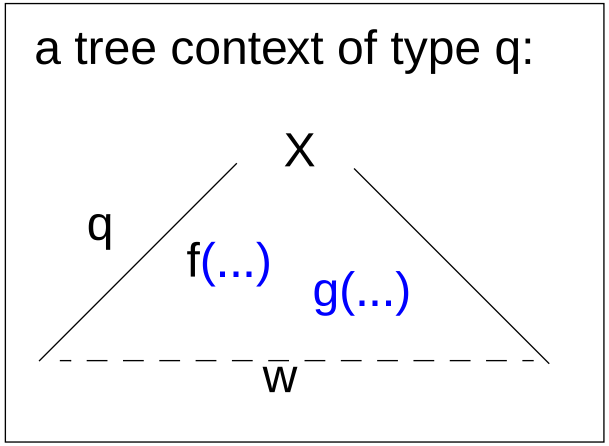


Basic concepts of attribute grammars (1)

An AG specifies **computations in trees** expressed by **computations associated to productions** of the abstract syntax

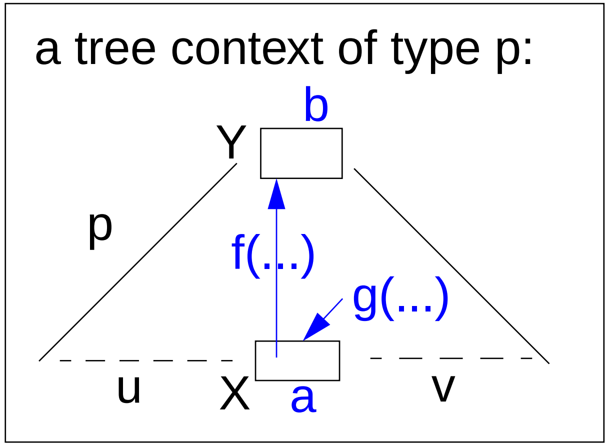
```
RULE q: X ::= w COMPUTE
   f(...); g(...);
END;
```

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



An AG specifies **dependences between computations**: expressed by **attributes associated to grammar symbols**

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



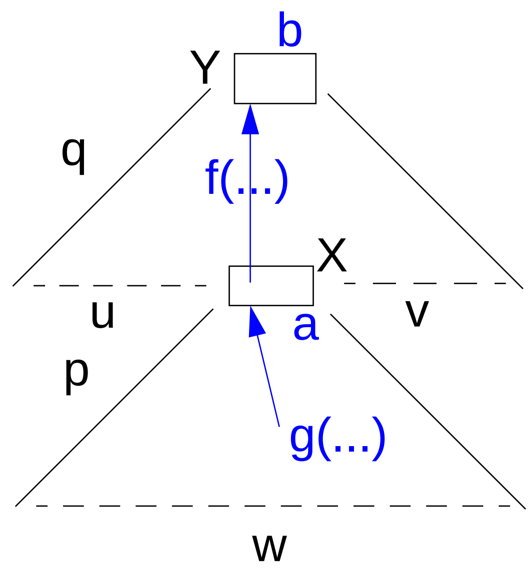
Attributes represent: **properties of symbols** and
pre- and post-conditions of computations:
post-condition = f (pre-condition)
f(X.a) uses the result of g(...); hence
X.a = g(...) is specified to be executed before f(X.a)

Basic concepts of attribute grammars (2)

dependent computations in adjacent contexts:

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

adjacent contexts
of types q and p:



attributes may specify
dependences without propagating any value;
specifies the order of effects of computations:

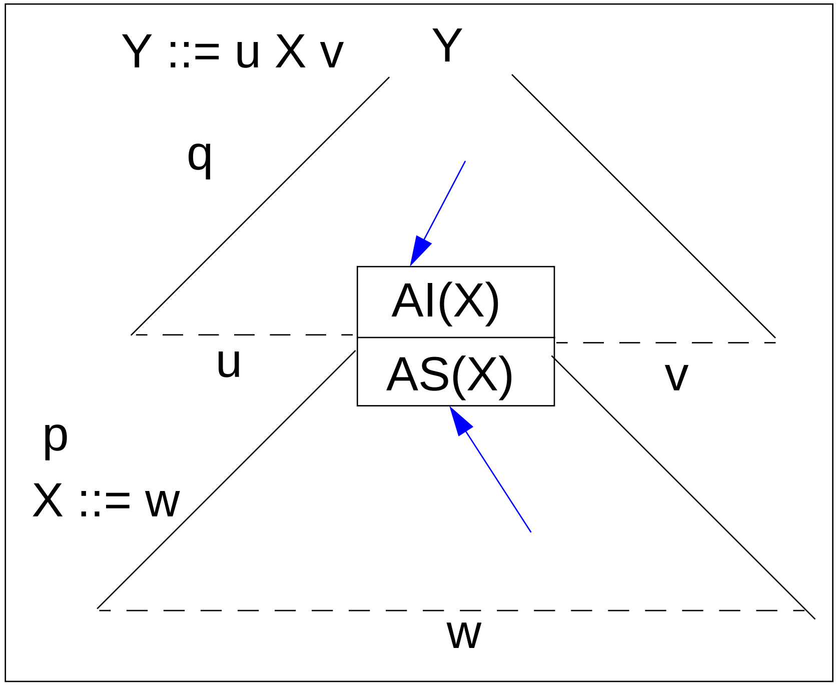
```
X.GotType = ResetTypeOf(...);
Y.Type = GetTypeOf(...) <- X.GotType;
```

`ResetTypeOf` will be called before `GetTypeOf`

Definition of attribute grammars

An **attribute grammar** AG = (G, A, C) is defined by

- a **context-free grammar G** (abstract syntax)
- for each **symbol X** of G a set of **attributes A(X)**, written X.a if a ∈ 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: Y::= ... X... has exactly one computation
for each attribute of AS(Y), for the symbol on the left-hand side of p, and
for each attribute of AI(X), for each symbol occurrence on the right-hand side of p

AG Example: Compute expression values

The AG specifies: The value of each expression is computed and printed at the root:

```
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;
```

A (Expr) = AS(Expr) = {value}

AS(Opr) = {value}
AI(Opr) = {left, right}

A(Opr) = {value, left, right}

AG Binary numbers

Attributes:

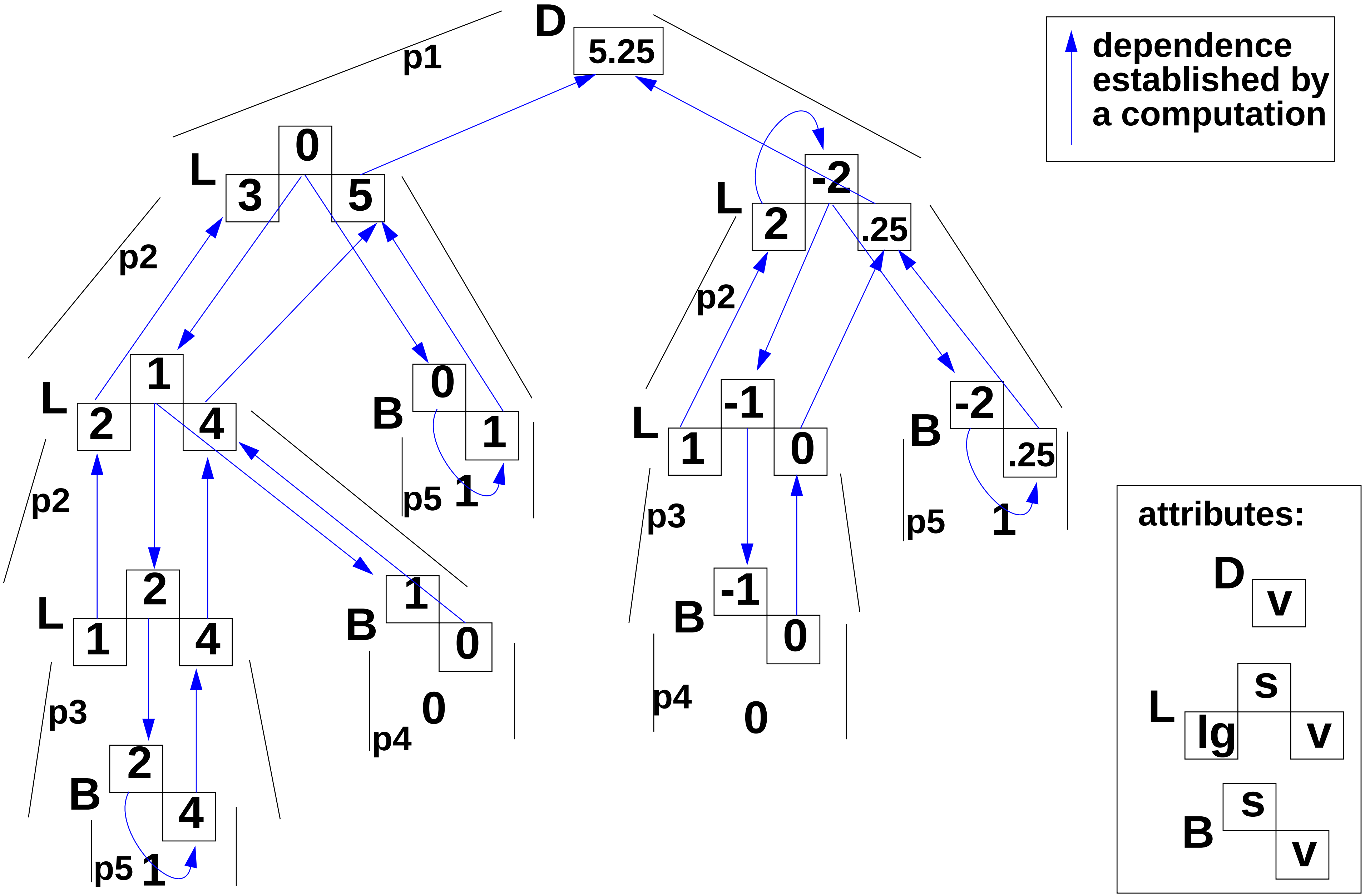
`L.v, B.v`	value
`L.lg`	number of digits in the sequence L
`L.s, B.s`	scaling of B or the least significant digit of L

```
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;
```

scaled binary value:

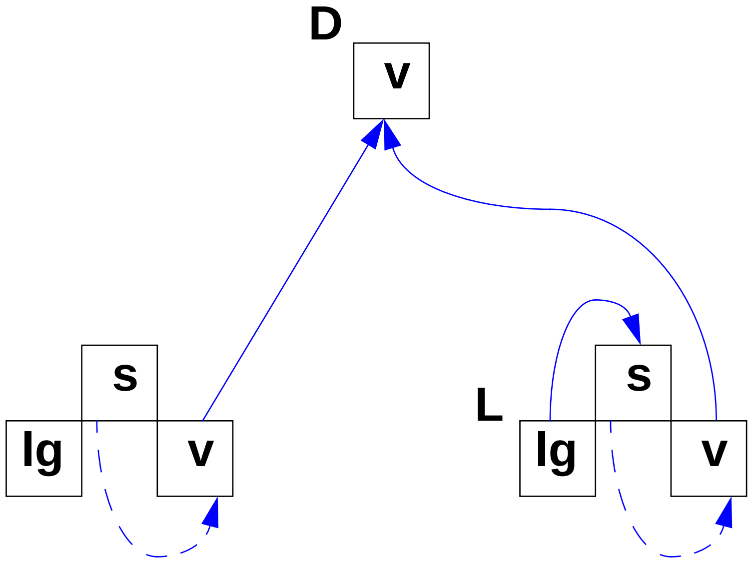
$B.v = 1 * 2^{B.s}$

An attributed tree for AG Binary numbers

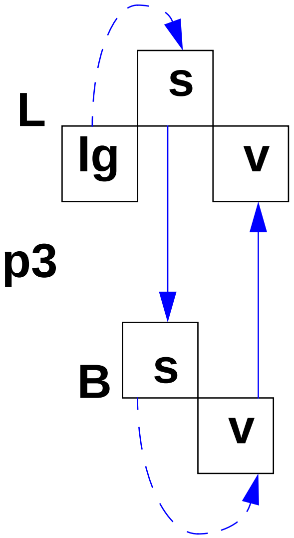


Dependence graphs for AG Binary numbers

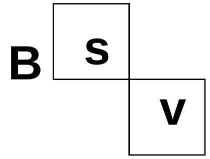
p1



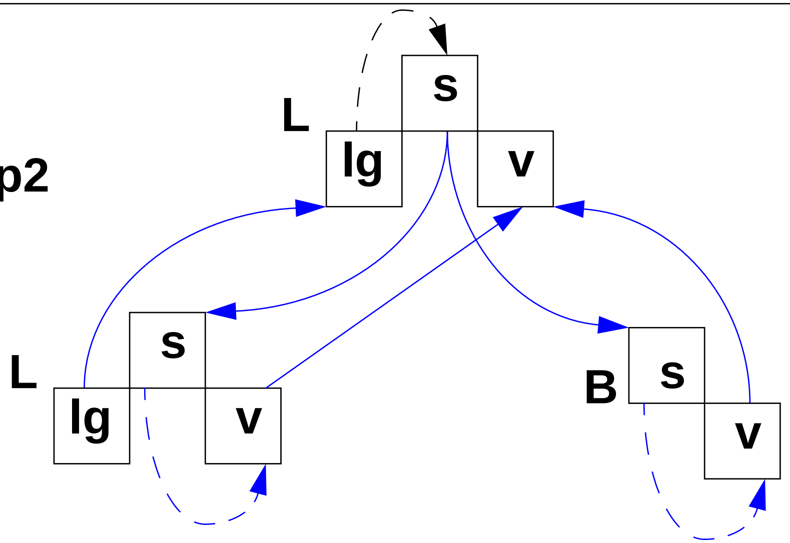
p3



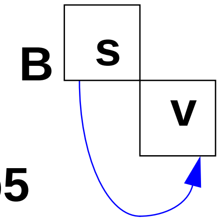
p4



p2



p5



If a tree exists, that has a path from X.a to X.b at some node of Type X, the graphs have an **indirect dependence**

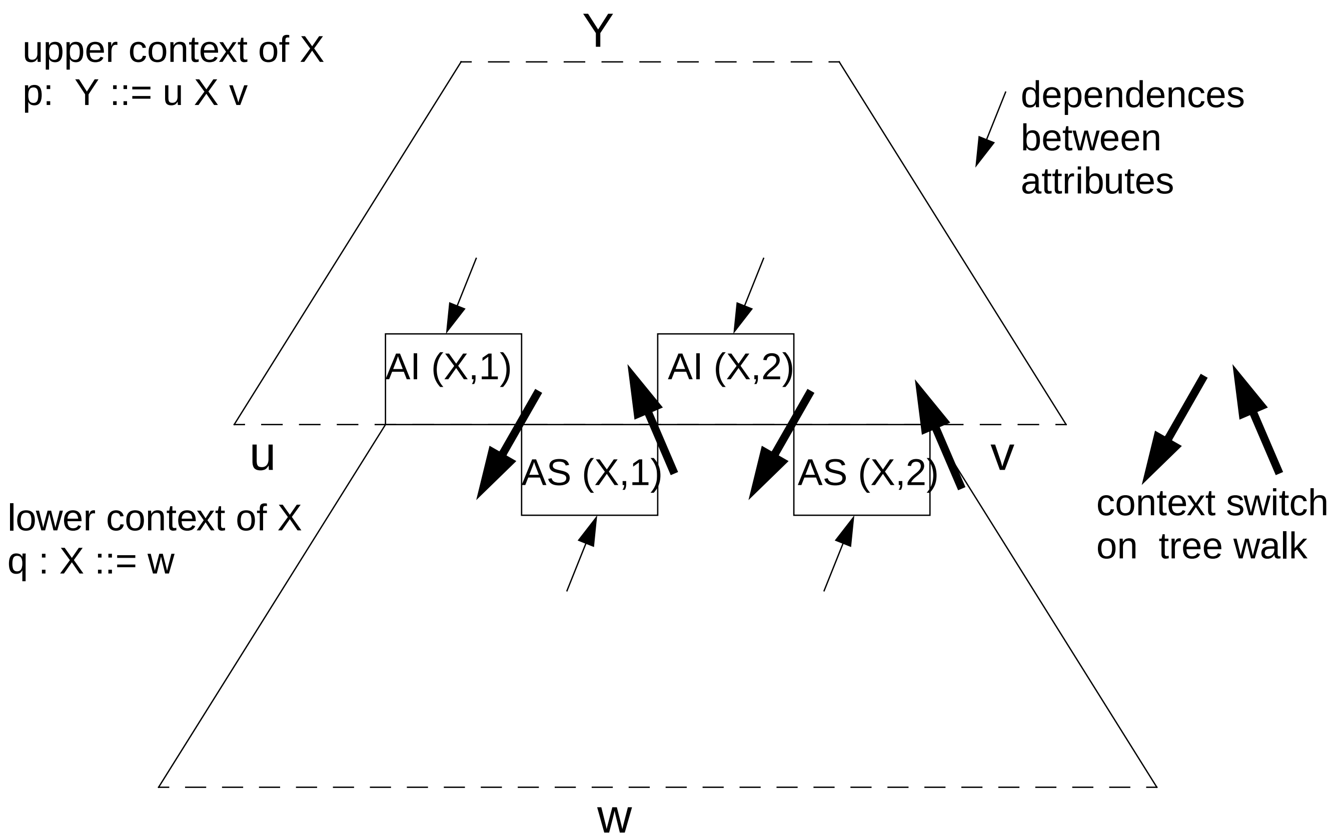
X.a - - - - → X.b

Attribute partitions

The sets AI(X) and AS(X) are **partitioned** each 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

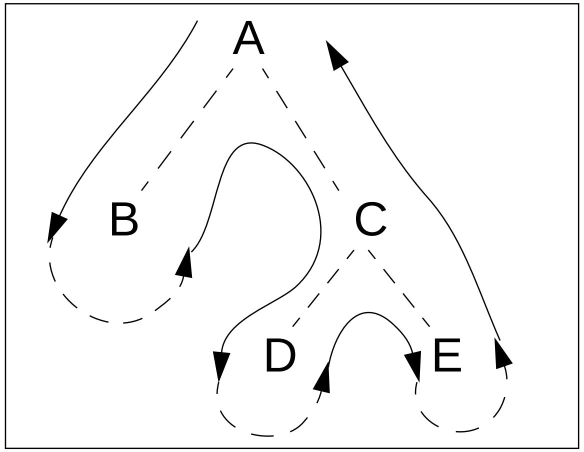


Necessary precondition for the existence of such a partition:
No node in any tree has **direct or indirect dependences that contradict the evaluation order** of the sequence of sets:AI (X, 1), AS (X, 1), ..., AI (X, k), AS (X, k)

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** in visited contexts.
- The execution order obeys the **attribute dependences**.



Pass-oriented strategies for the tree walk: **AG class:**

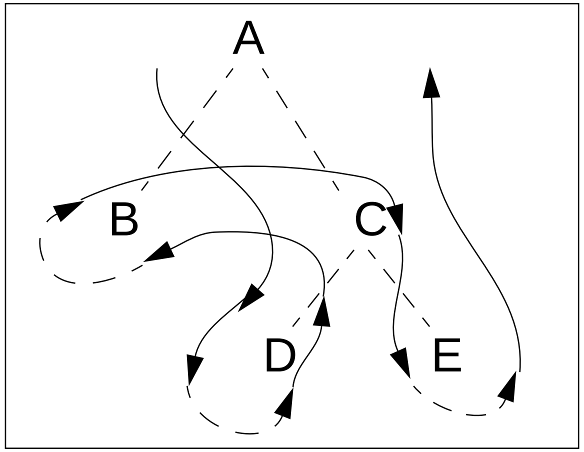
k times **depth-first left-to-right**	**LAG (k)**
k times depth-first right-to-left	RAG (k)
alternatingly left-to-right / right-to left	**AAG (k)**
once **bottom-up (synth. attributes only)**	**SAG**

AG is checked if attribute dependences fit to desired pass-oriented strategy; see LAG(k) check.

non-pass-oriented strategies:

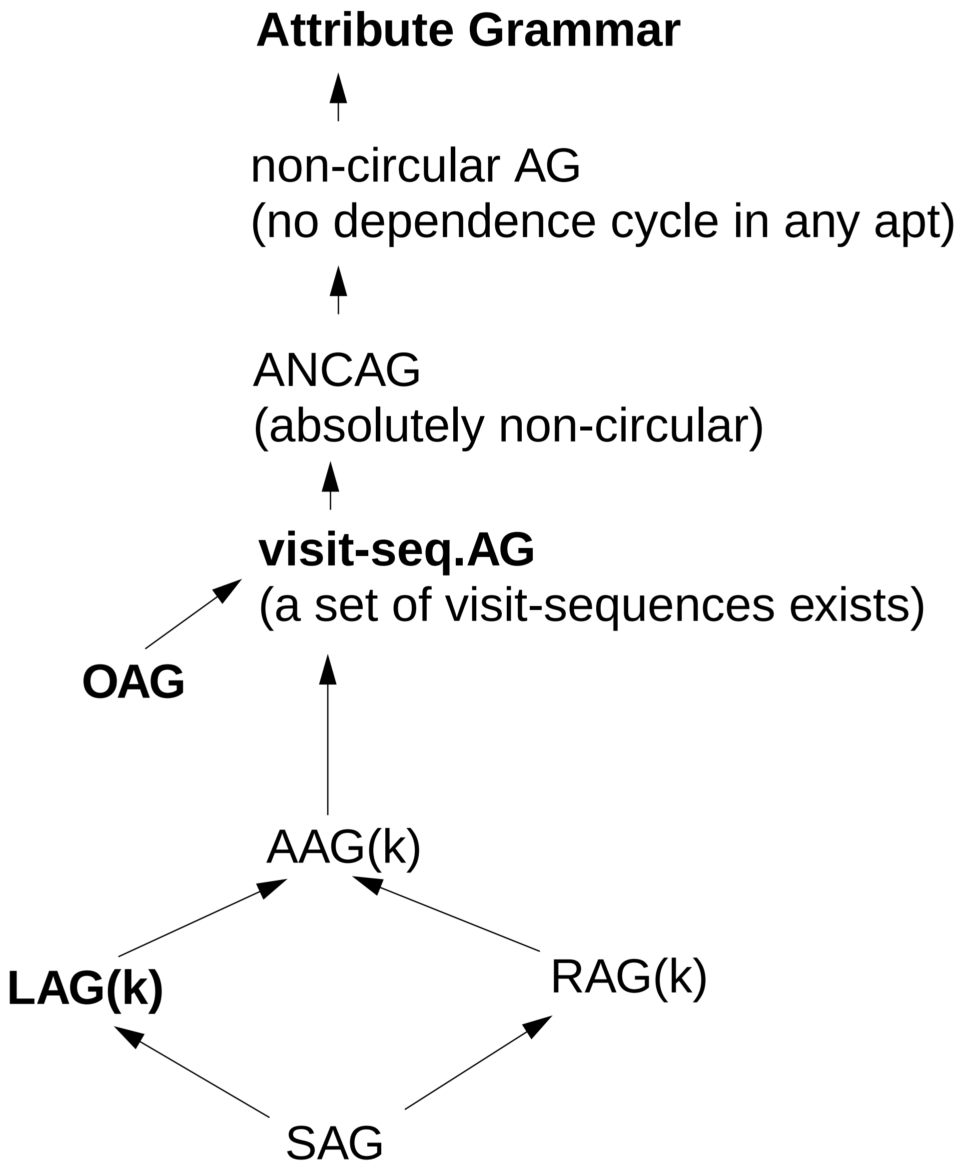
visit-sequences: **OAG**

an individual plan for each rule of the abstract syntax



A generator fits the plans to the dependences of the AG.

Hierarchy of AG classes



Visit-sequences

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

$$p: X_o ::= X_1 \ldots X_i \ldots X_n$$

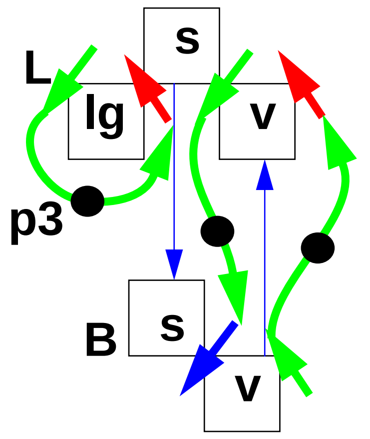
A visit-sequence is a **sequence of operations**:

↓ i, j	j-th **visit of the i-th subtree**
↑ j	j-th r**eturn to the ancestor** node
$eval_c$	execution of a **computation** c associated to p

Example out of the AG for binary numbers:

vs_{p3}: L ::= B

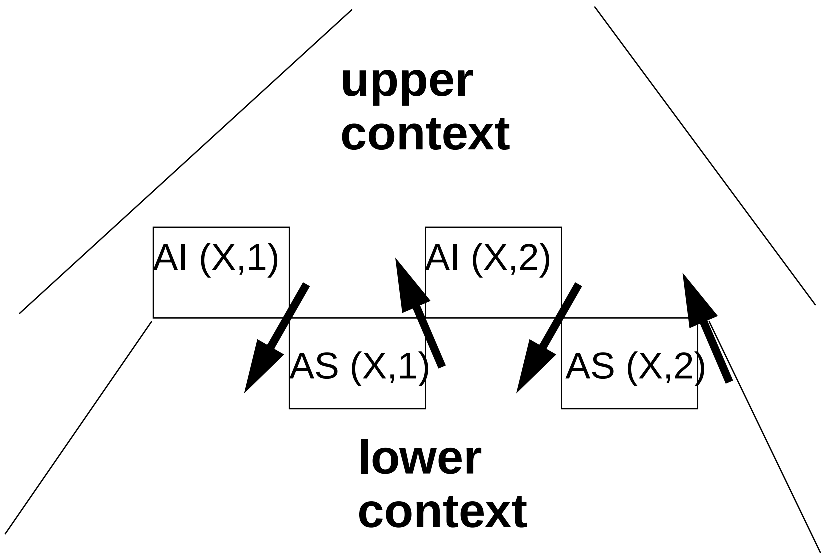
L.lg=1; ↑1; B.s=L.s; ↓B,1; L.v=B.v; ↑2



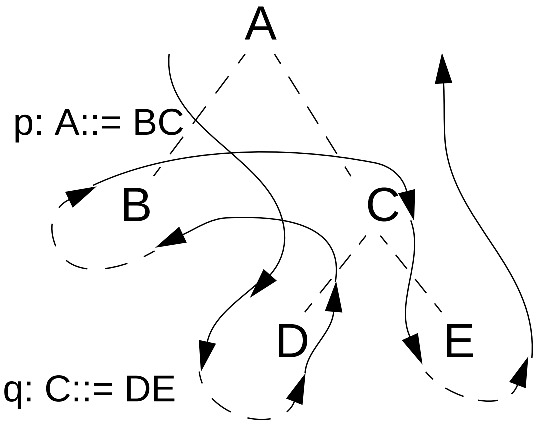
Interleaving of visit-sequences

Visit-sequences for adjacent contexts are executed interleaved.

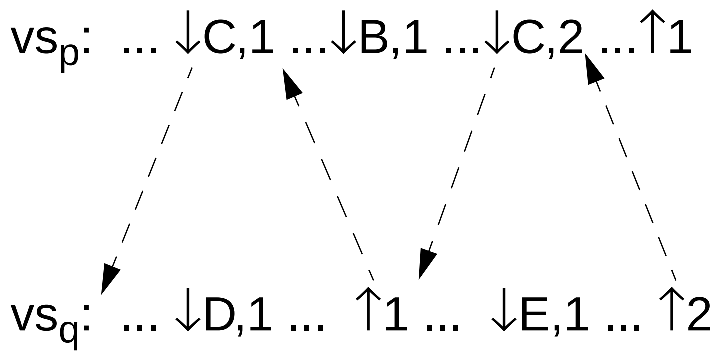
The **attribute partition** of the common nonterminal specifies the **interface** between the upper and lower visit-sequence:



Example in the tree:



interleaved visit-sequences:



Implementation:one **procedure for each section of a visit-sequence upto** ↑
a **call** with a switch over applicable productions for ↓

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

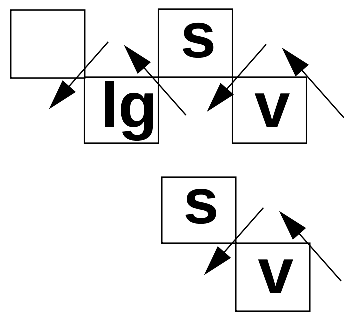
L.lg=1; ↑1; B.s=L.s; ↓B,1; L.v=B.v; ↑2

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

B.v=0; ↑1

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

B.v=Power2(B.s); ↑1

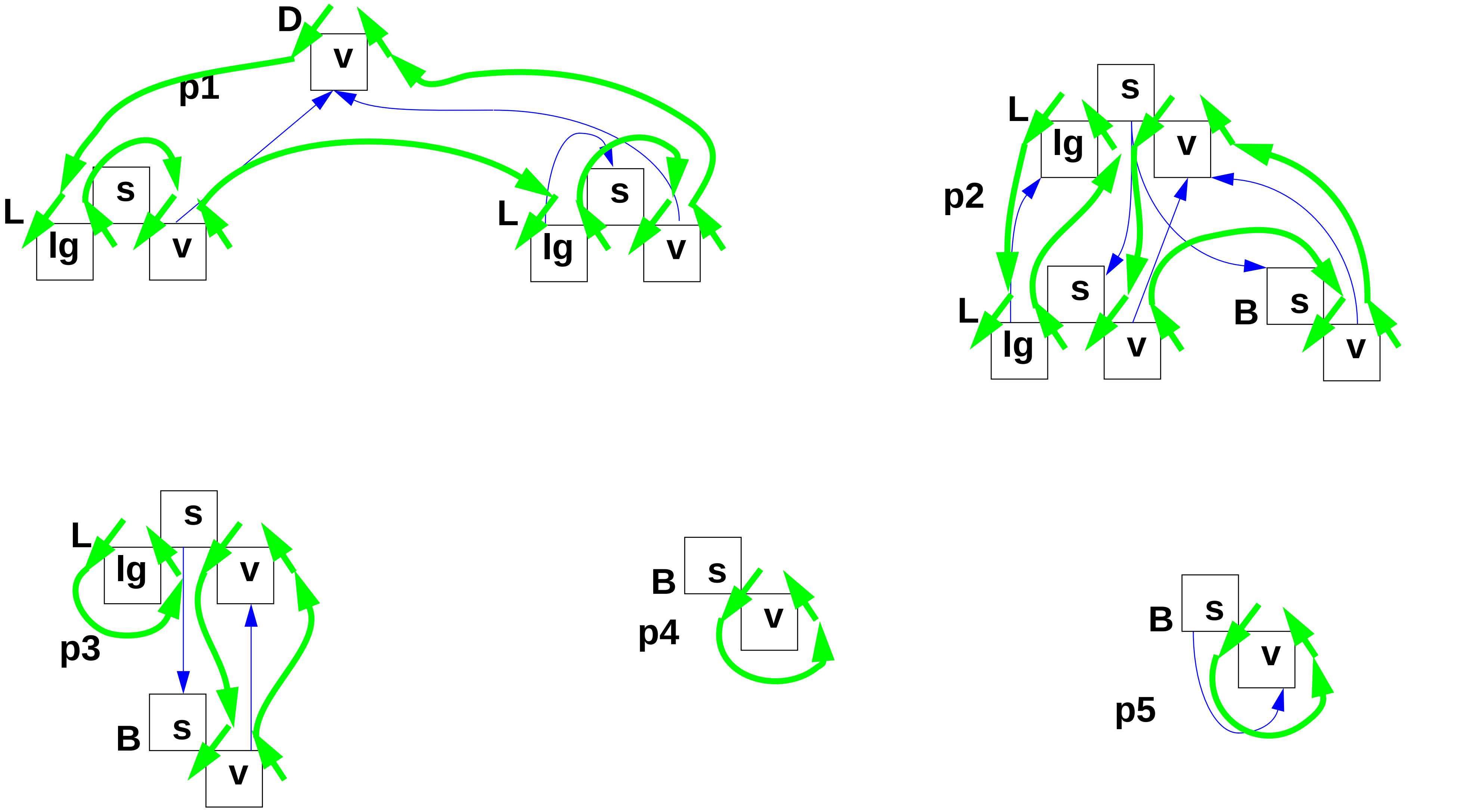


Implementation:

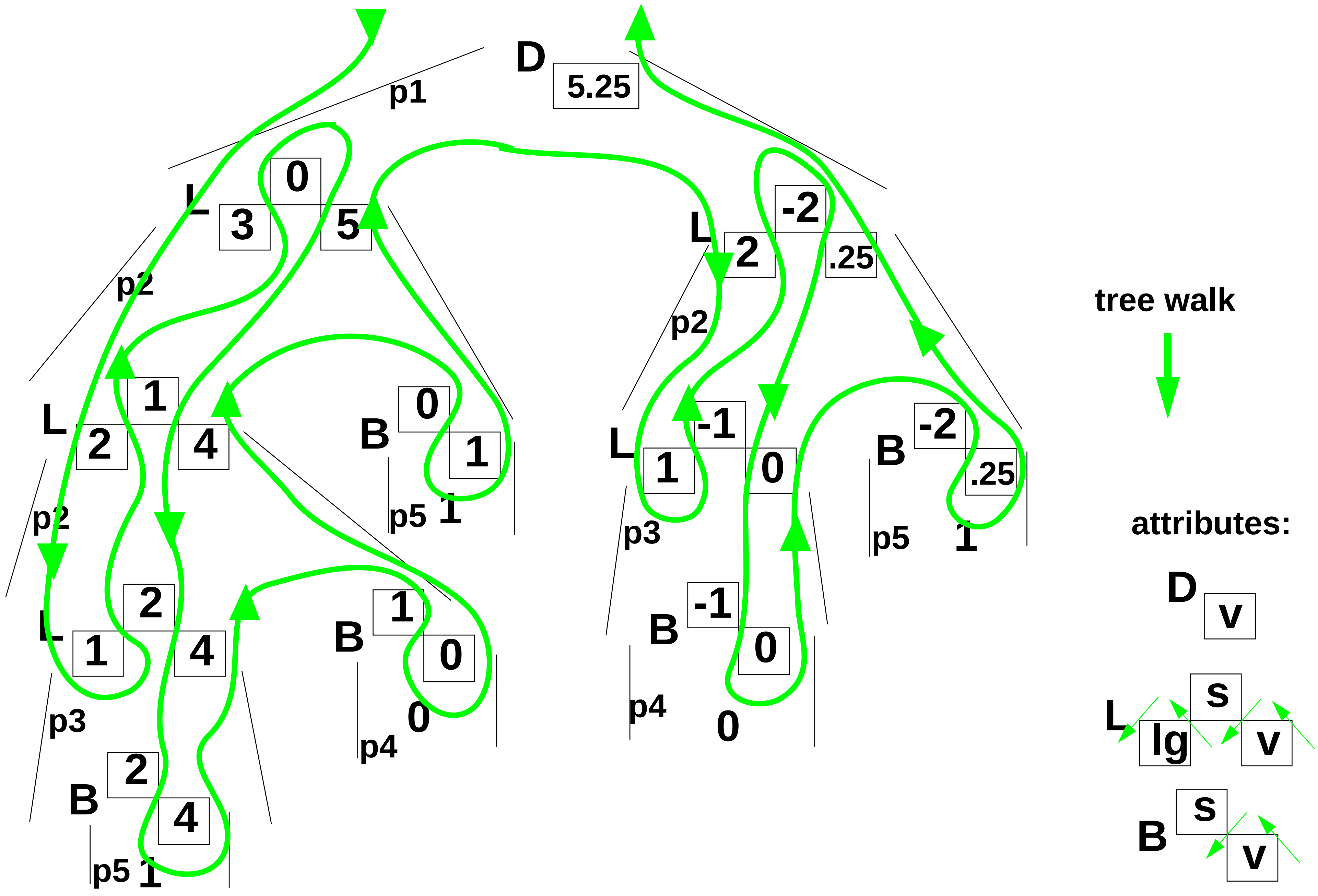
Procedure vs<i><p> for each section of a vs_p to a ↑i

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

Visit-Sequences for AG Binary numbers (tree patterns)



Tree walk for AG Binary numbers



LAG (k) condition

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.**

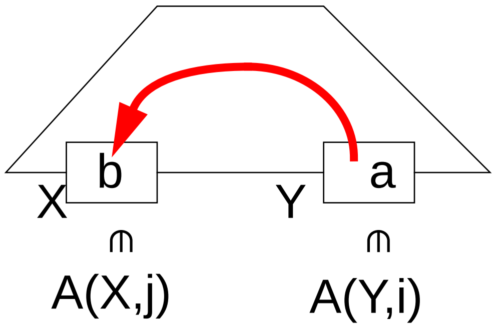
Crucial dependences:

In every dependence graph every dependence

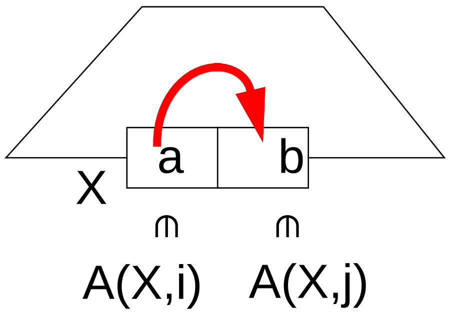
- **Y.a -> X.b** where X and Y occur on the **right-hand side** and **Y is right of X** implies that **Y.a belongs to an earlier pass than X.b**, and
- **X.a -> X.b** where X occurs on the **right-hand side** implies that **X.a belongs to an earlier pass than X.b**

Necessary and sufficient condition over dependence graphs - expressed graphically:

A dependency **from right to left**



j > i



i < j

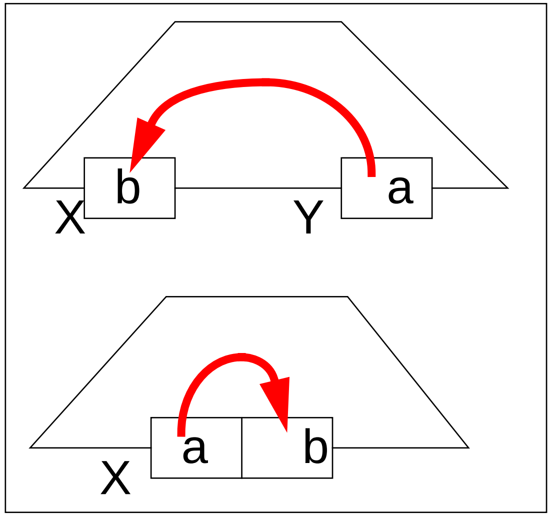
A dependence **at one symbol** on the right-hand side

LAG (k) algorithm

Algorithm checks whether there is a **k>=1** such that an **AG is LAG(k)**.

Method:

compute iteratively **`A(1)`**, ..., **`A(k)`**;
in each iteration try to allocate all remaining attributes to the current pass, i.e. **`A(i)`**;
remove those which can not be evaluated in that pass



Algorithm:

Set **`i=1`** and **`Cand=`** all attributes

repeat

set **`A(i) = Cand`**; set **`Cand`** to empty;

while still attributes can be removed from **`A(i)`** do

remove an attribute `X.b` from `A(i)` and add it to **`Cand`** if

- there is a **crucial dependence**

`Y.a` -> **`X.b`** s.t.

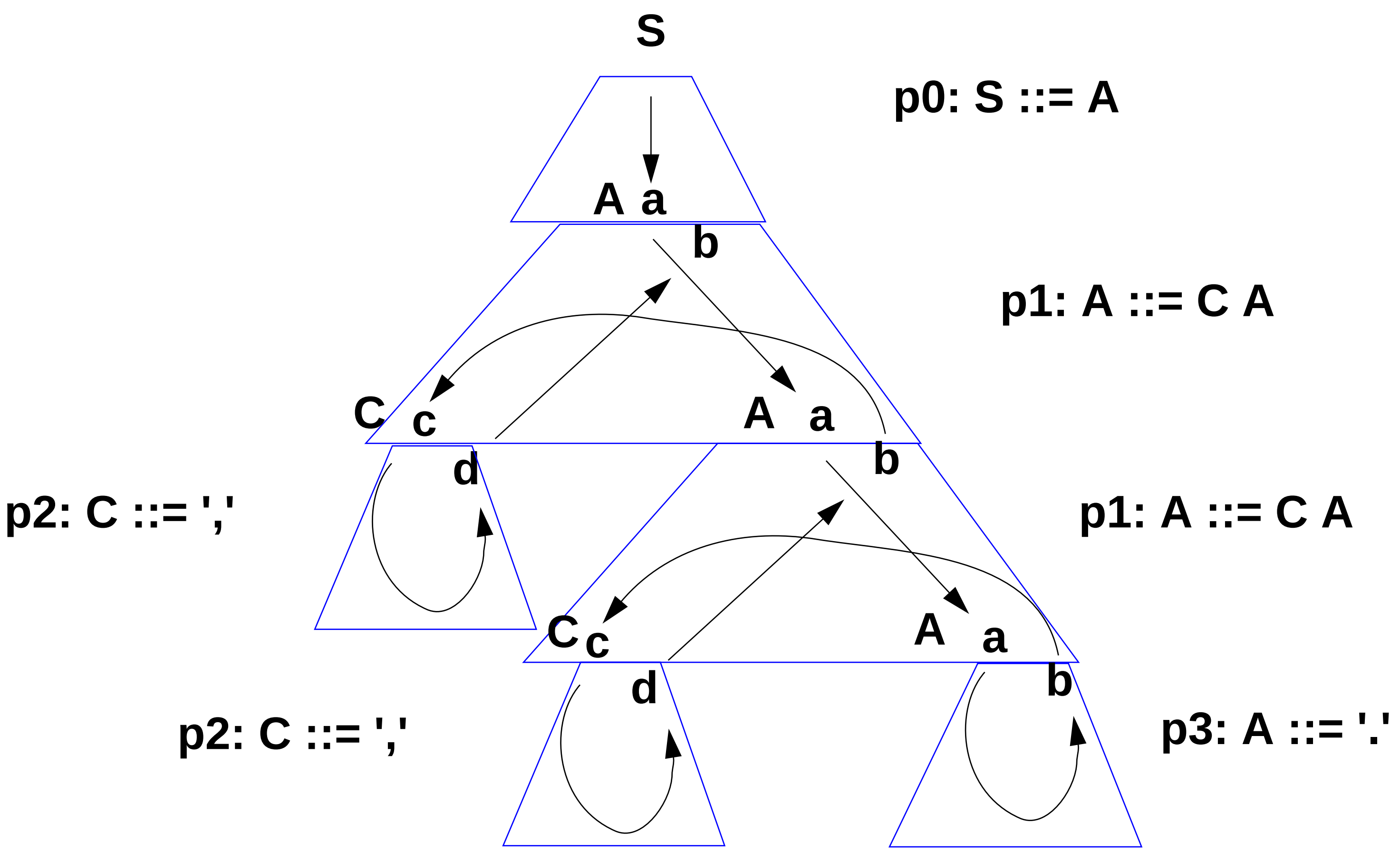
`X` and **`Y`** are on the right-hand side, **`Y`** to the right of **`X`** and **`Y.a`** in **`A(i)`** or

`X.a` -> **`X.b`** s.t. **`X`** is on the right-hand side and **`X.a`** is in **`A(i)`**

- **`X.b`** depends on an attribute that is not yet in any **`A(i)`**

if **`Cand`** is empty:	exit: the AG **is LAG(k)** and all attributes are assigned to their passes
if **`A(i)`** is empty:	exit: the AG **is not LAG(k) for any k**
else:	set i = i + 1

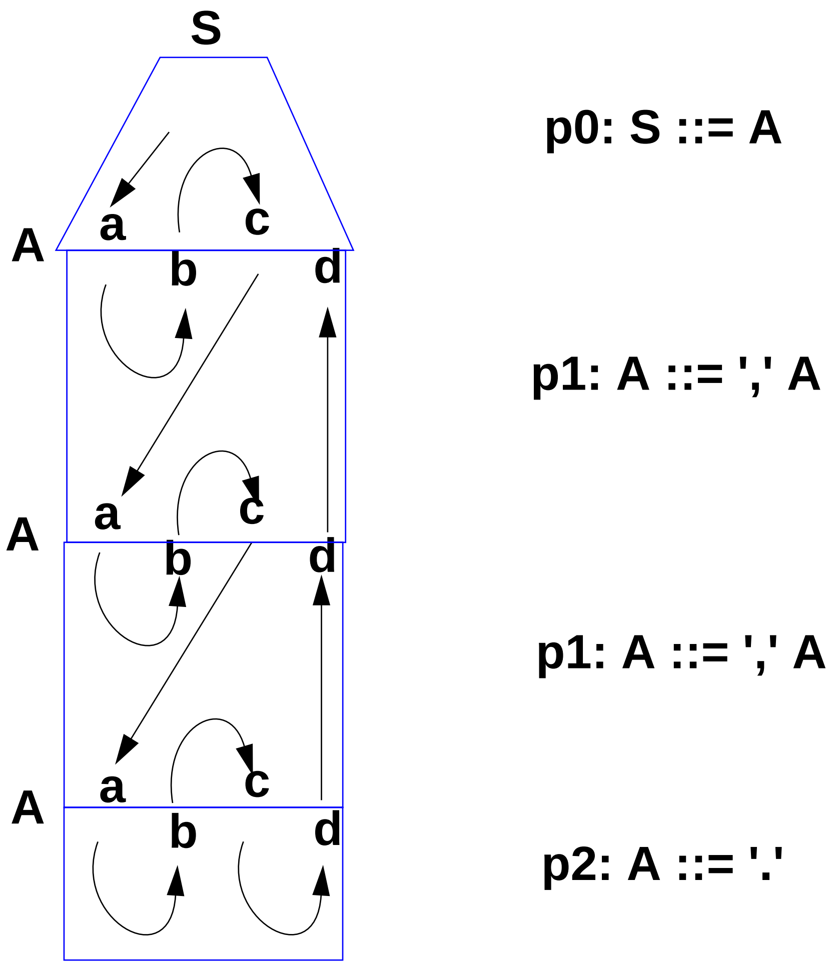
AG not LAG(k) for any k



A.a can be allocated to the first left-to-right pass.
C.c, C.d, A.b can not be allocated to any pass.

The AG is RAG(1), AAG(2) and
can be evaluated by visit-sequences.

AG not evaluable in passes



No attribute can be allocated to any pass for any strategy.

The AG can be evaluated by visit-sequences.

Generators for attribute grammars

LIGA	University of Paderborn	OAG
FNC-2	INRIA	ANCAG (superset of OAG)
CoCo	Universität Linz	LAG(k)

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**

Explicit left-to-right depth-first propagation

```
ATTR pre, post: int;
RULE: Root ::= Block COMPUTE
  Block.pre = 0;
END;
RULE: Block ::= '{' Constructs '}' COMPUTE
  Constructs.pre = Block.pre;
  Block.post = Constructs.post;
END;
RULE: Constructs ::= Constructs Construct COMPUTE
  Constructs[2].pre = Constructs[1].pre;
  Construct.pre = Constructs[2].post;
  Constructs[1].post = Construct.post;
END;
RULE: Constructs ::= COMPUTE
  Constructs.post = Constructs.pre;
END;
RULE: Construct ::= Definition COMPUTE
  Definition.pre = Construct.pre;
  Construct.post = Definition.post;
END;
RULE: Construct ::= Statement COMPUTE
  Statement.pre = Construct.pre;
  Construct.post = Statement.post;
END;
RULE:Definition ::= 'define' Ident ';' COMPUTE
  Definition.printed =
     printf ("Def %d defines %s in line %d\n",
             Definition.pre, StringTable (Ident), LINE);
  Definition.post =
     ADD (Definition.pre, 1) <- Definition.printed;
END;
RULE: Statement ::= 'use' Ident ';' COMPUTE
  Statement.post = Statement.pre;
END;
RULE: Statement ::= Block COMPUTE
  Block.pre = Statement.pre;
  Statement.post = Block.post;
END;
```

`Definitions` are enumerated and printed from left to right.

The next **`Definition`** number is propagated by a pair of attributes at each node:

`pre` (inherited)
`post` (synthesized)

The value is initialized in the **`Root`** context and

incremented in the **`Definition`** context.

The computations for propagation are systematic and redundant.

Left-to-right depth-first propagation using a CHAIN

```
CHAIN count: int;

RULE: Root ::= Block COMPUTE
   CHAINSTART Block.count = 0;
END;

RULE: Definition ::= 'define' Ident ';'
COMPUTE
   Definition.print =
      printf ("Def %d defines %s in line %d\n",
              Definition.count, /* incoming */
              StringTable (Ident), LINE);

   Definition.count = /* outgoing */
      ADD (Definition.count, 1)
      <- Definition.print;
END;
```

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

One CHAIN name; attribute pairs are generated where needed.

CHAINSTART initializes the CHAIN in the root context of the CHAIN.

Computations on the CHAIN are **strictly bound** by dependences.

Trivial computations of the form X.pre = Y.pre in CHAIN order can be **omitted**. They are generated where 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;

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

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

The nesting depths of **Blocks** are computed.

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

Propagation from computation to the uses are generated as needed.

No explicit computations or attributes are needed for the remaining rules and symbols.

Dependency pattern CONSTITUENTS

```
RULE: Root ::= Block COMPUTE
   Root.DefDone =
      CONSTITUENTS Definition.DefDone;
END;

RULE: Definition ::= 'define' Ident ';'
COMPUTE
   Definition.DefDone =
      printf ("%s defined in line %d\n",
               StringTable (Ident), LINE);
END;

RULE: Statement ::= 'use' Ident ';' COMPUTE
   printf ("%s used in line %d\n",
            StringTable (Ident), LINE)
   <- INCLUDING Root.DefDone;

END;
```

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

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

Propagation from computation to the **`CONSTITUENTS`** construct is generated where needed.

The shown combination with **`INCLUDING`** is a common dependency pattern.

All **`printf`** calls in **`Definition`** contexts are done before any in a **`Statement`** context.