Non-Pass-Oriented Evaluation: Visit-Sequences

A **visit-sequence** vs_p **for each production** of the tree grammar:

p: X_o ::= X_1 ... X_i ... X_n

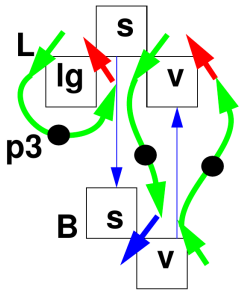
A visit-sequence is a **sequence of operations**:

↓ 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

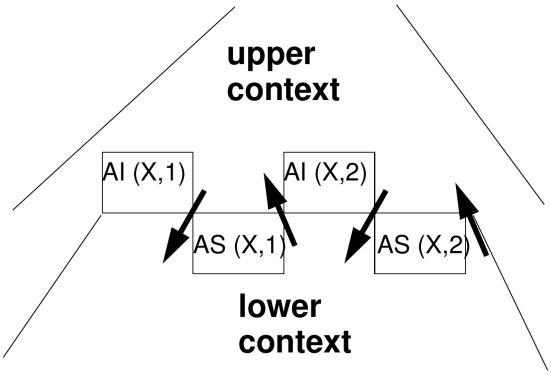
L.lg=1; ↑1; B.s=L.s; ↓B,1; L.v=B.v; ↑2



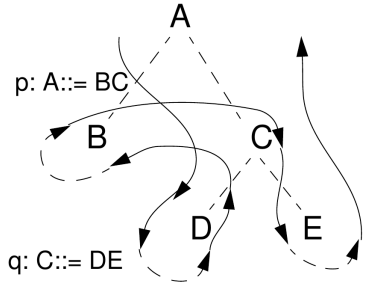
Interleaving of Visit-Sequences

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:

vs_p: ... ↓C,1 ...↓B,1 ...↓C,2 ...↑1

vs_q: ... ↓D,1 ... ↑1 ... ↓E,1 ... ↑2

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

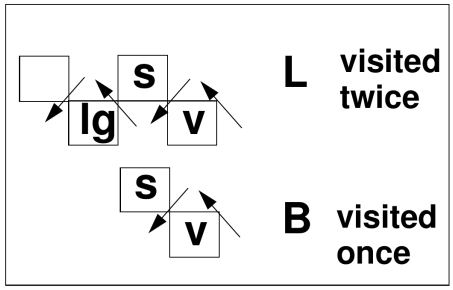
L.lg=1; ↑1; B.s=L.s; ↓B,1; L.v=B.v; ↑2

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

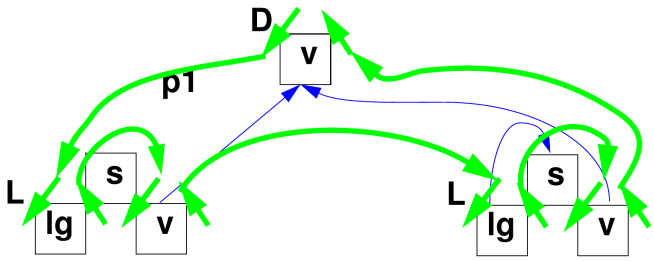
B.v=0; ↑1

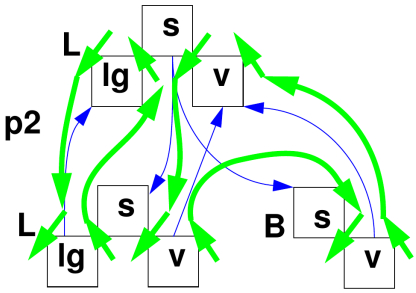
vs_{p5}: B ::= '1'

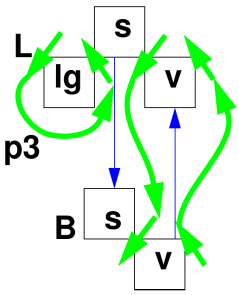
B.v=Power2(B.s); ↑1

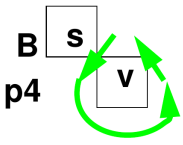


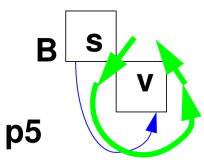
Visit-Sequences for AG "Binary Numbers" (tree patterns)



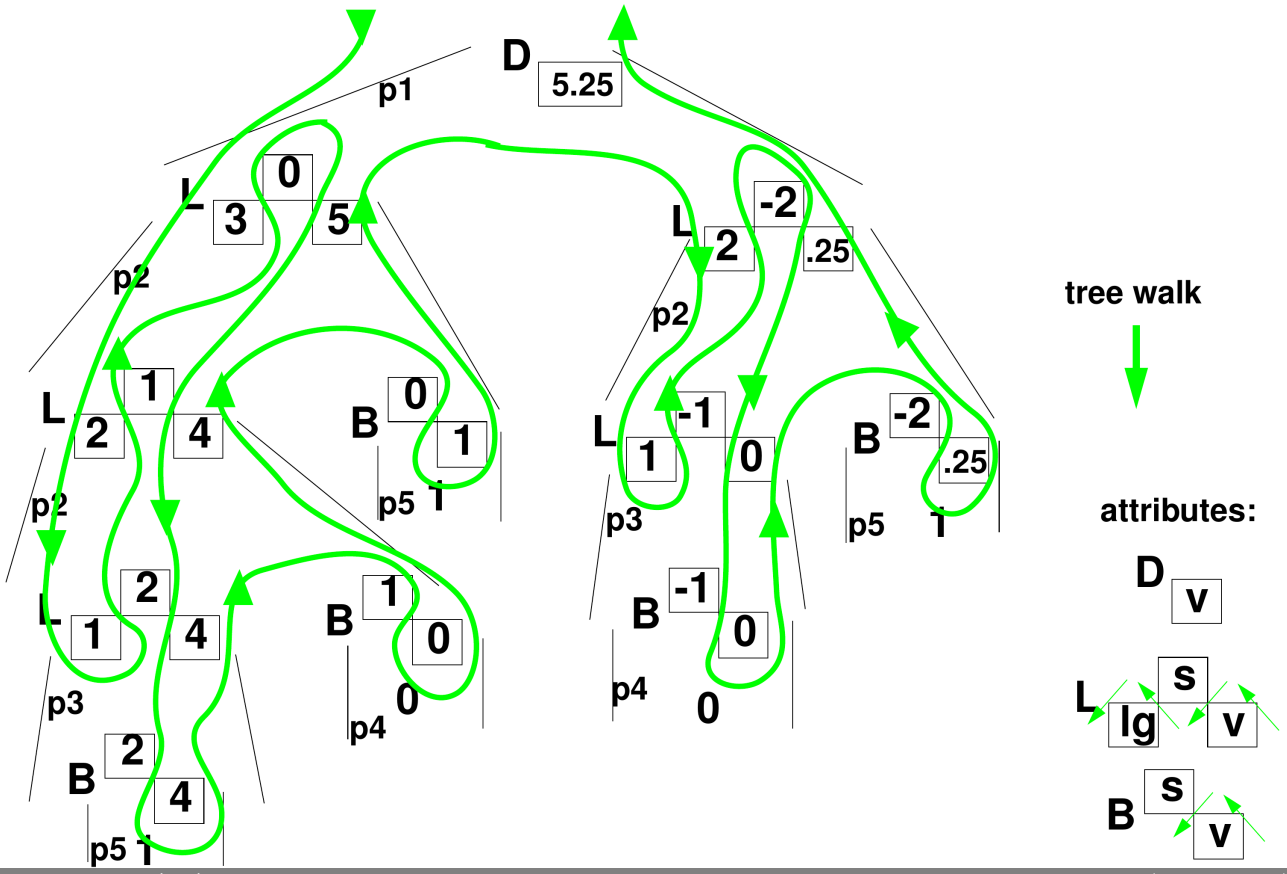








Tree Walk for AG "Binary Numbers"



Generators for Attribute Evaluators

LIGA, University of Paderborn, OAG
CoCo/R, University of Linz, LAG(k)
JastAdd, University of Lund, dynamic evaluation

Properties of the Attribute Evaluator Generator LIGA

- integrated in the Eli system cooperating with the other tools
- attribute grammar specification language *Lido*
- modular and reusable AG components
- symbol computations and reuse mechanism
- computations call functions implemented outside the AG
- advanced notations for remote attribute access
- visit-sequence controlled attribute evaluators, implemented in C

Examples of Liga specifications are given on the following slides.

LIGA: State attributes without values

```
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.

LIGA: Dependency pattern 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.

LIGA: 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.

LIGA: 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.

LIGA: Symbol Computations

Computations can be associated to symbols occurring in the tree (*TREE symbols*). They are executed for every node which represents that symbol in a particular tree.

Symbols may be introduced which do not belong to the tree grammar. They are called *CLASS symbols* and represent computational roles. Their computations may be inherited by grammar symbols.

TREE symbol computations

```
TREE SYMBOL Expr COMPUTE
   SYNT.coercion =
      coerce(THIS.pre,THIS.post);
   INH.IsValContext = true;
   chkLegal(THIS.coercion);
END;
```

CLASS symbols and inheritance

```
CLASS SYMBOL IdOcc COMPUTE
   SYNT.Sym = TERM;
END;

SYMBOL VarDef INHERITS IdOcc
END;
```

The `TREE` and `CLASS` markers can be omitted. In attribute denotations the symbol names are replaced by `SYNT` , `INH` , `THIS` , or `TERM` , `TERM[1]`, ...

Symbol computations can be provided in libraries and are a powerful mechanism for reuse.