3. Visiting Trees Overview

Computations in structure trees may serve any suitable purpose, e.g.

- **compute or check properties of language constructs**, e. g. types, values
- **determine or check relations in larger contexts,** e.g. definition - use
- **construct data structure or target text**

Formal model for specification: attribute grammars (AGs)

Generator Liga transforms

a specification of computations in the structure tree
(an AG written in the specification language Lido)

into

a tree walking attribute evaluator that executes the specified computations
for each given tree in a suitable order.

Computations in Tree Contexts Specified by AGs

Abstract syntax is augmented by:

Attributes associated to **nonterminals**:
e.g. Expr.Value   Expr.Type   Block.depth used to

store values at tree nodes, representing a property of the construct,
propagate values through the tree,
specify dependences between computations

Computations associated to **productions** (RULEs) or to nonterminals (SYMBOL):

Compute attribute values
using other attribute values of the particular context (RULE or SYMBOL), or

cause effects, e.g. store values in a definition table,
check a condition and issue a message, produce output

Each **attribute** of every node is **computed exactly once**.
Each **computation** is **executed exactly once** for every node of the RULE it is specified for.

The **order of the computation execution** is **determined by the generator**. It obeys the **specified dependences**.

Dependent Computations

```
SYMBOL Expr, Opr: value: int SYNT;
SYMBOL Opr: left, right: int INH;
TERM Number: int;

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

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;

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

typed attributes of symbols

terminal symbol has int value

SYNThesized attributes are computed in lower contexts,
INHerited attributes in upper c..

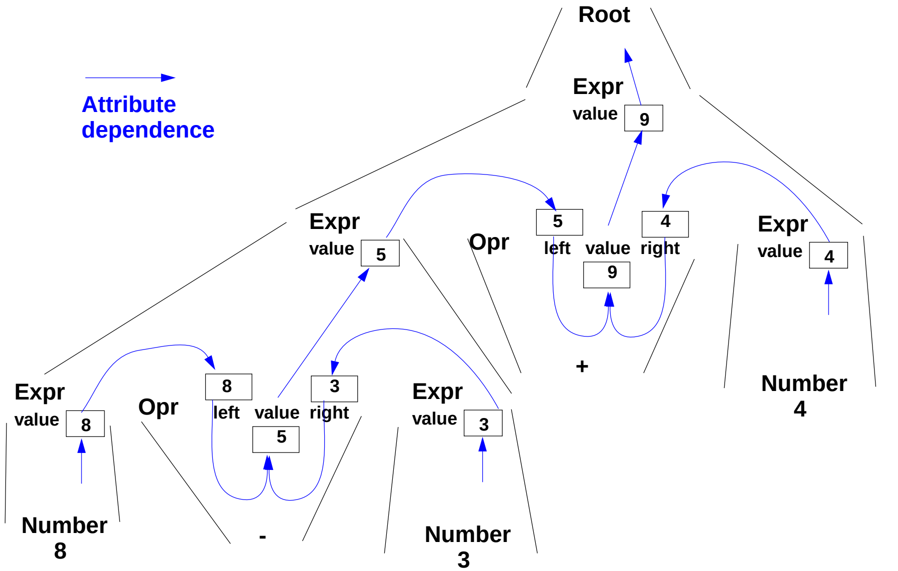
SYNT or INH usually need not be specified.

Generator determines the **order of computations** consistent with dependences.

Example:

Computation and output of an expression's value

An Attributed Structure Tree



Pre- and Postconditions of Computations

```
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: 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;

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

Attributes `print` and `printed` **don't have values** (type **`VOID`**)

They describe states being **pre- and postconditions** of computations

`Expr.print`:

Postfix output up to this node is completed.

`Expr.printed`:

Postfix output up to and including this node is completed.

Example:

Expression is printed in postfix form

Pattern: Dependences Left-to-Right Depth-First Through the Tree

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

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

CHAIN specifies **left-to-right depth-first** dependence.

CHAINSTART in the **root context** of the CHAIN (initialized with an irrelevant value)

Computations are inserted between **pre- and postconditions of the CHAIN**

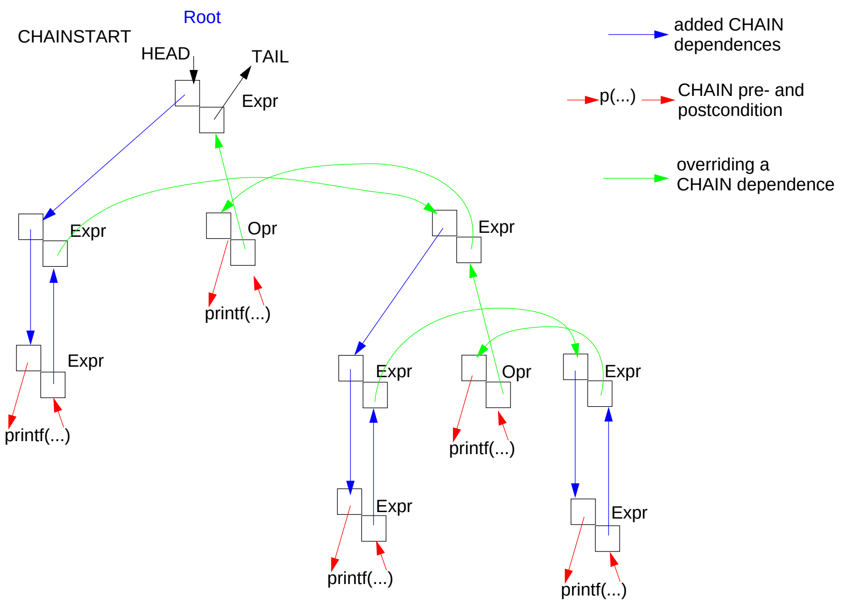
CHAIN order can be overridden.

Omitted CHAIN computations are added automatically

Example:

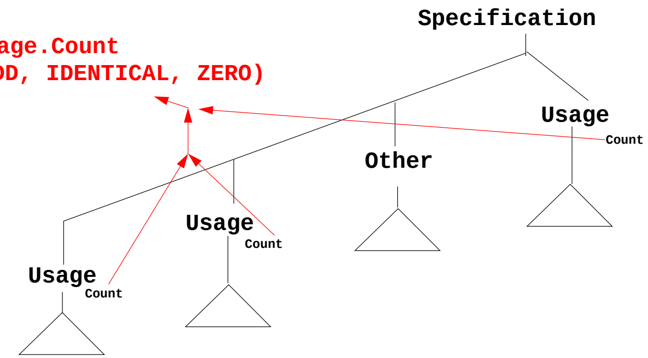
Output an expression in postfix form (cf. GSS-3.4)

Pattern: Dependences Left-to-Right Depth-First Through the Tree



Pattern: Combine Attribute Values of a Subtree

```
CONSTITUENTS Usage.Count
   WITH (int, ADD, IDENTICAL, ZERO)
```



`CONSTITUENTS` combines certain attributes of a subtree, here **`Usage.Count`**

	`WITH (int,`	**`ADD,`**	**`IDENTICAL,`**	**`ZERO)`**
Meaning:	type	binary function	unary function, applied to every attribute	constant function for optional subtrees

Pattern: Use an Attribute of a Remote Ancestor Node

```
SYMBOL Block: depth: int INH;

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

RULE: Block ::= '(' Sequence ')' END;
RULE: Sequence LISTOF
         Definition / Statement 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;
```

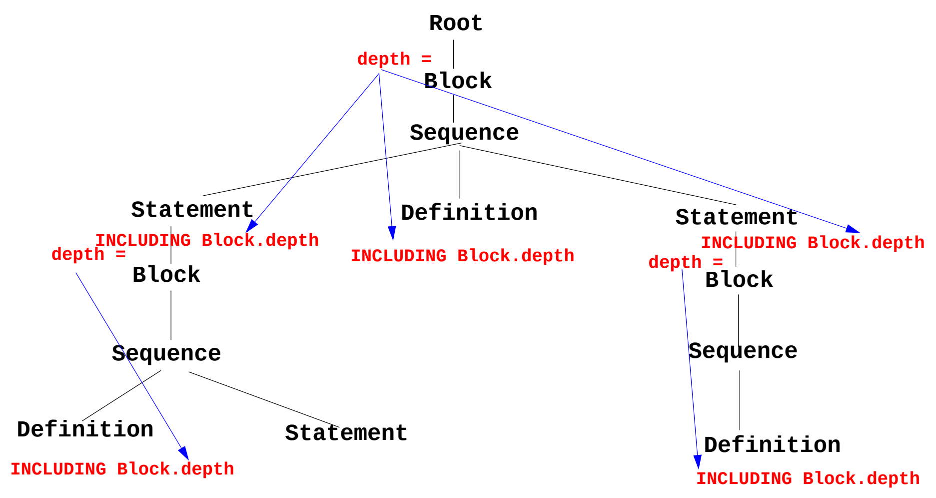
Example:

Compute nesting depth of blocks

INCLUDING Block.depth refers to the **depth** attribute of the next ancestor node (towards the root) that has type **Block**

The **INCLUDING** attribute is **automatically propagated** through the contexts between its definition in an ancestor node and its use in an **INCLUDING** construct.

Example for INCLUDING in a Tree



Pattern: Combine Preconditions of Subtree Nodes

```
SYMBOL Block: DefDone: VOID;

RULE: Root ::= Block END;

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

Example:

Output all definitions before all uses

The attributes **DefDone** do not have values - they specify **preconditions** for some computations

This **CONSTITUENTS** construct does not need a **WITH clause**, because it does not propagate values

Typical combination of a CONSTITUENTS construct and an **INCLUDING** construct:

Specify the order side-effects are to occur in.

Computations Associated to Symbols

Computations may be associated to **symbols**; then they are executed for **every occurrence** of the symbol in a production.

```
SYMBOL Expr COMPUTE
   printf ("expression value %d in line %d\n", THIS.value, LINE);
END;
```

Symbol computations may contain **INCLUDING**, **CONSTITUENTS**, and **CHAIN** constructs:

```
SYMBOL Block COMPUTE
   printf ("%d uses occurred\n",
      CONSTITUENTS Usage.Count WITH (int, ADD, IDENTICAL, ZERO);
END;
```

SYNT.a resp. **INH.a** indicates that the computation belongs to the **lower** resp. **upper context** of the symbol:

```
SYMBOL Block COMPUTE
   INH.depth = ADD (INCLUDING Block.depth);
END;
```

Computations in **RULE contexts override computations** for the same attribute **in SYMBOL context**, e.g. for begin of recursions, defaults, or exceptions:

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

Reuse of Computations

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

SYMBOL DefVarIdent INHERITS IdOcc END;
SYMBOL DefTypeIdent INHERITS IdOcc END;
SYMBOL UseVarIdent INHERITS IdOcc END;
SYMBOL UseTypeIdent INHERITS IdOcc END;


CLASS SYMBOL CheckDefined COMPUTE
   IF (EQ (THIS.Key, NoKey),
   message ( ERROR,
            "identifier is not defined",
            0, COORDREF);
END;

SYMBOL UseVarIdent
   INHERITS IdOcc, CheckDefined END;
SYMBOL UseTypeIdent
   INHERITS IdOcc, CheckDefinedEND;
```

Computations are associated to **CLASS** symbols, which do not occur in the abstract syntax.

INHERITS binds **CLASS** symbols to tree symbols of the abstract syntax.

Reuse of Pairs of SYMBOL Roles

```
CLASS SYMBOL OccRoot COMPUTE
   CHAINSTART HEAD.Occurs = 0;
   SYNT.TotalOccs = TAIL.Occurs;
END;
CLASS SYMBOL OccElem COMPUTE
   SYNT.OccNo = THIS.Occurs;
   THIS.Occurs = ADD (SYNT.OccNo, 1);
END;


SYMBOL Block      INHERITS OccRoot END;
SYMBOL Definition INHERITS OccElem END;

SYMBOL Statement INHERITS OccRoot END;
SYMBOL Usage     INHERITS OccElem END;
```

CLASS symbols in cooperating roles, e.g. count occurrences of a language construct (**OccElem**) in a subtree (**OccRoot**)

Restriction:
Every **OccElem**-node must be in an **OccRoot**-subtree.

Reused in pairs:

Block - Definition and

Statement - Usage

must obey the restriction.

Library modules are used in this way (see Ch. 6)

Design Rules for Computations in Trees

1. Decompose the task into **subtasks**, that are small enough to be solved each by only a few of the specification patterns explained below.d
   Develop a `.lido` fragment for each subtask and explain it in the surrounding `.fw` text.
2. Elaborate the **central aspect of the subtask** and map it onto one of the following cases:
   A. The aspect is described in a natural way by **properties of some related program constructs**,
   e.g. types of expressions, nesting depth of blocks, translation of the statements of a block.
   B. The aspect is described in a natural way by **properties of some program entities,**
   e.g. relative addresses of variabes, use of variables before their definition.

   Develop the computations as described for A or B.
3. Step 2 may exhibit that further aspects of the subtask need to be solved (attributes may be used, for which the computations are not yet designed). Repeat step 2 for these aspects.

A: Compute Properties of Program Constructs

Determine the **type of values**, which describe the property. Introduce **attributes of that type for all symbols**, which represent the **program constructs**. Check which of the following cases fits best for the computation of that property:

A1: Each **lower context** determines the property in a different way:
Then develop **RULE computations for all lower contexts**.

A2: As A1; but **upper context**.

A3: The property can be determined **independently of RULE contexts**, by using only attributes of the symbol or attributes that are accessed via INCLUDING, CONSTITUENT(S), CHAIN:
Then develop a **lower (SYNT) SYMBOL computation**.

A4: As A3; but there are a **few exceptions**, where either lower of upper (not both) RULE contexts determine the property in a different way:
Then develop a upper (INH) or a lower (SYNT) **SYMBOL computation** and **override it in the deviating RULE contexts**.

A5: As A4; but for **recursive symbols**: The begin of the recursion is considered to be the exception of A4, e.g. nesting depth of Blocks.

If none of the cases fits, the design of the property is to be reconsiderd; it may be too complex, and may need further refinement.