LIDO - Computations in Trees

Uwe Kastens

University of Paderborn
D-4790 Paderborn
F.R.G

$Revision: 4.3 $

Table of Contents

Introduction

Language processors generated by Eli consist of a structuring phase and a transformation phase. The first reads the input, checks whether its structure fulfills the language requirements and builds a tree representing that structure. The transformation phase performs any kind of computations on such trees necessary for analysis of the input structure and for computing some output as required by the language processor tasks. Both phases are generated from user supplied specifications.

This document introduces the techniques used to specify the transformation phase. Its central concepts are computations in trees. The subsequent sections introduce the most important techniques and their notation in the specification language LIDO on the base of very simple examples.

Chapter 2 [Depend], page 5 to Chapter 4 [Symbols], page 17 should be read in that order to get a complete overview. It should be pointed out, that this document is not intended to define the language LIDO. The LIDO Reference manual should be consulted for specific questions on language constructs. It also describes language facilities that are not introduced here. See LIDO – Reference Manual.

Chapter 5 [Specification], page 23 of this document describes how to use LIDO specifications within Eli, how they interact with other specifications and how to get more information when errors are reported on the specification. The first part (Section 5.1 [LIGA Files], page 23 to Section 5.3 [Implementing], page 24) should be read while examples are practically exercised. Section 5.4 [Errors], page 25 should be consulted initially when errors occur that can not be immediately traced and corrected.

1 Tree Structure

The central data structure of a specified language processor is a tree. It usually represents the abstract structure of the particular input text and is built by actions of the scanner and parser. The trees a language processor operates on are specified by a context-free grammar, the tree grammar. It is part of the `lido` specification. Figure 1 shows a tree grammar for simple expressions of numbers and binary operators.

```
RULE:    Root ::= Expr                END;
RULE:    Expr ::= Expr Opr Expr       END;
RULE:    Expr ::= Number              END;
RULE:    Opr  ::= '+'                 END;
RULE:    Opr  ::= '*'                 END;
```

Figure 1: Expression Tree Grammar

`Root`, `Expr`, and `Opr` are the nonterminals of this context-free grammar. `Number`, `'+'` and `'*'` are its terminals. Trees are built such that their nodes represent occurrences of nonterminals of the tree grammar. Terminals are not represented in the tree. Each production specifies that the symbol on the left-hand side has a sequence of subtrees according to the nonterminals on the right-hand side. In our example the first production specifies one, the second three, and the others no subtree. `Expr` and `Opr` have two alternative productions each.

Figure 2 shows an example for a tree specified by this grammar, which may represent the input expression `1 + 2 * 3`. (The terminals in the bottom line do not belong to the tree.)

```
                 Root
                  |
                  |
                 Expr
                  |
      -----------|-----------
      |           |          |
     Expr        Opr      Expr
      |           |          |
      -           -     ------------
                        |    |      |
                      Expr  Opr   Expr
                        |    |      |
                        -    -      -

     Number       +   Number  *   Number
```

Figure 2: An Expression Tree

A tree node together with its immediate descendent nodes represent the application of a production, called a rule context. In Figure 2, there are for example two instances of the rule context for the second rule of the grammar. The two productions for `Opr` describe different rule contexts, although both have no subtrees.

If we consider a node of a tree, then it connects two "adjacent contexts", an "upper context" and a "lower context". For example the upper context of an `Expr` node may be an application of the first or the second rule, and the lower context may be an application of the second or third rule.

Rule contexts and adjacent contexts are the central concepts for association of computation, for dependencies between computations, and for the tree walk executing them.

Two kinds of terminals are distinguished: Literal terminals like `'+'` and `'*'` do not carry any information. They are only used to identify the production and to relate it to the concrete grammar. Named terminals like `Number` may carry some information usually computed from the input token by the scanner, e. g. the value of the number. That information may be used in computations of the rule context where the terminal occurs.

2 Dependent Computations

Language processing requires certain computations to be executed for each instance of a language construct. Hence the specification associates computations to rule contexts. Usually a computation depends on values or effects yielded by other computations. Such dependencies are specified by "attributes" associated to grammar symbols. It should be emphasized that only the necessary dependencies are specified, rather than a complete execution order. A tree walking algorithm that executes the computations in a suitable order is generated automatically.

In this section we introduce the basic concepts and notations for specification of dependent computations in trees. The examples refer to the tree grammar of the previous section. It will be shown how expression evaluation and a simple transformation is specified.

2.1 Value Dependencies

Let us first describe computations that evaluate any given expression tree and print the result:

```
ATTR value: int;

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

The above `RULE` associates the `printf` computation to the rule context. The notation repeats the rule as shown in the tree grammar and adds any set of computations between `COMPUTE` and `END`. Computations are denoted like calls of C functions. The arguments are C literals, again function calls, or attributes. (User defined functions are implemented in specifications files separate from the .lido specification, See Chapter 5 [Specification], page 23.)

The above computation uses the `value` attribute of the `Expr` subtree of this context. In general any attribute of any symbol that occurs in the rule context may be used in a computation of that context.

In this case the `value` attribute is the integral value computed for the expression. The `ATTR` construct states that its type is `int`. In fact it specifies that type for any attribute named `value` that is just used with a symbol. Any C type name may be specified. Such a type association is valid throughout the whole specification.

The above computation depends on a computation that yields `Expr.value`. Since the internal structure of the `Expr` subtree determines how its value is to be computed, those computations are associated to the two lower adjacent contexts that have `Expr` on the left-hand side of their production, as shown in the computations of Figure 3.

```
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;
```

Figure 3: Computation of Expression Values

The `TERM` construct states that the terminal `Number` carries a value of type `int` to be used in computations like that of the first rule.

Computations that yield a value to be used in other computations are denoted like an assignment to an attribute. But it must be emphasized that they have to obey the single assignment rule: There must be exactly one computation for each attribute instance in every tree.

The value of binary expressions is computed in each of the two `Opr` contexts and passed to the root of the binary subtree. The `ADD` and `MUL` operations are predefined macros in LIDO. `Opr` has three attributes, the values of the left and right operands and the result of the operation. The attributes `left` and `right` are associated to `Opr` by the `SYMBOL` construct which states their type to be `int`. As only the `Opr` symbol has attributes of these names thy are not introduced by an `ATTR` construct. The attribute `value` is associated to `Opr` by just using it in a computation. Its type is specified by the `ATTR` construct explained above.

The three attributes belong to two conceptually different classes: `Opr.value` is computed in the lower context, as `Expr.value` (called synthesized or `SYNT` attribute), whereas `Opr.left` and `Opr.right` are computed in the upper context (called inherited or `INH` attributes). Any attribute must belong to either of the classes in order to obey the single assignment rule.

There are three (in this case trivial) computations specified in the context for binary expressions. It should be pointed out that their textual order is irrelevant: The execution order is determined by their dependencies. In this case the computation of `Expr[1].value` will be executed last. (The attribute notation requires indexing of symbol names, if a symbol occurs more than once in a production, like `Expr`.)

2.2 State Dependencies

Our second example specifies how to print expressions in postfix notation, e. g. `1 2 3 * +` for the given expression `1 + 2 * 3`. It demonstrates how computations that yield an effect rather than a value are specified to depend on each other.

We may start from a specification that just outputs each number and operator, given in Figure 4. It causes each instance of numbers and operators in the tree being printed. Since no dependencies are specified yet, they may occur in arbitrary order in the output.

```
RULE: pRoot: Root  ::= Expr  COMPUTE
  printf ("\n");
END;

RULE: pNumb: Expr ::= Number COMPUTE
  printf ("%d " , Number.Sym)
END;

RULE: pAdd: Opr   ::=  '+' COMPUTE
  printf ("+ ");
END;

RULE: pMul:  Opr   ::= '*' COMPUTE
  printf ("* ");
END;
```

Figure 4: Output of Expression Components

In order to achieve the desired effect we have to specify that a computation is not executed before certain preconditions hold which are established by a postcondition of some other computations. We specify such conditions by attributes that do not have values, but describe a computational state. In Figure 5 we associate attributes `print` and `printed` to `Expr` and `Opr`. `Expr.print` describes the state where the output is produced so far such that the text of the `Expr` subtree can be appended. `Expr.printed` describes the state where the text of this subtree is appended (correspondingly for `Opr.print` and `Opr.printed`).

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

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

Figure 5: Dependencies for Producing Postfix Expressions

The general form of dependent computations as used in Figure 5 is

postcondition = *computation* `<-` *precondition*

If the postcondition is not used elsewhere, it (and the =) is omitted. If the postcondition is directly established by another condition, the computation and the `<-` are omitted. If a condition initially holds it is denoted by some literal, like "yes" in `pRoot`. A computation may depend on several preconditions:

```
<- (X.a, Y.b).
```

Computations may also depend on the computation of value carrying attributes without using their value, or computations may yield a value and also depend on some preconditions.

State attributes which do not carry a value have the type `VOID`. They need not be introduced by a `SYMBOL` or an `ATTR` construct. They may be just used in a computation. But the same consistency and completeness requirements apply for them as for value carrying attributes.

It should be noted that the specifications of several tasks, e. g. computing expression values and producing postfix output may be combined in one specification for a language processor that solves all of them. It is a good specification style to keep the specifications of different tasks separate (in different files), rather than to merge the computations for each single context. You may specify several sets of computations at different places. They are accummulated for the same context.

See Section 3.3 [Chain], page 15, for an example of expressing the above example using Left-to-Right Depencencies.

3 Remote Dependencies in Trees

In the previous section we considered dependencies between computations within one rule context or computations that are associated to pairs of adjacent contexts. It is often necessary to specify that a precondition of a computation is established rather far away in the tree, e. g. a value computed in the root context is used in several computations down in the tree. Instead of propagating it explicitly through all intermediate contexts it may be accessed directly by notations for remote dependencies.

In the following we introduce three constructs for remote dependency specification:

access to a subtree root from contexts within the subtree (`INCLUDING` construct),

access to contexts within a subtree from its root context (`CONSTITUENTS` construct),

computations at certain subtree contexts that depend in depth-first left-to-right order on each other (`CHAIN` construct).

These constructs can be used for value dependencies as well as for state dependencies. Since these constructs avoid specifications in the contexts between the remote computations, they abstract from the particular tree structure in between: It may be designed according to other aspects or be altered without invalidating those remote dependencies.

3.1 Access to a Subtree Root

Assume that we have a language where `Blocks` are arbitrarily nested. We want to compute the nesting depth of each `Block`, and mark each `Definition` with the nesting depth of the smallest enclosing `Block`.

```
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;
```

Figure 6: Nesting Depth of Blocks

The specification of Figure 6 solves the stated problem by remote dependencies (`INCLUDING`). The tree contexts between `Block` and `Statement` or `Definition` are not mentioned. The expression

```
INCLUDING Block.depth
```

in the two computations accesses the `depth` value of the next enclosing `Block`.

In general alternative subtree root symbols may be specified, e. g.

```
INCLUDING (Block.depth, Procedure.depth, Module.depth)
```

Then the root of the smallest enclosing subtree is accessed which represents one of the given symbols. The tree grammar must guarantee that such a subtree root can always be found.

`INCLUDING` constructs may also be used as preconditions in `<-` constructs, and they may refer to state attributes.

3.2 Access to Contexts within a Subtree

Assume that we have a language for sequences of definitions and uses of names in arbitrary order. We want to produce an output text for each definition and each use, such that definition texts precede the use texts in the output. No specific order is required within the two text blocks.

```
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;
```

Figure 7: Sequencing Classes of Computations in a Subtree

The solution of the problem given in Figure 7 uses a state attribute `Block.DefDone`. It describes the state where all definition texts are printed. Hence in that state the condition `Definition.DefDone` must hold at each `Definition` in the subtree below the `Block` context, as stated by the `CONSTITUENTS` construct. The state `Block.DefDone` in turn is the precondition for the print computation in the `Usage` context.

The following example demonstrates the remote access to values within a subtree. We simply want to compute the number of `Usage` constructs in a program of the above language.

```
ATTR Count: int;

RULE: Block ::= '{' Sequence '}' COMPUTE
  printf  ("%d uses occurred\n ",
           CONSTITUENTS Usage.Count
           WITH (int, ADD, IDENTICAL, ZERO));
END;

RULE: Usage ::= 'use' Ident COMPUTE
  Usage.Count = 1;
END;
```

Figure 8: Adding Values of Subtree Components

The `CONSTITUENTS` construct in Figure 8 combines the values `Usage.Count` of each `Usage` node within the `Block` subtree. The `WITH` clause specifies how the values are combined, in this case they are added yielding an `int`-value.

The `WITH` clause is a scheme to combine an arbitrary number of values by a binary function. The general form is

```
WITH (t, combine, single, none)
```

where `single` is a function that yields a value of type `t` applied to an attribute accessed by the `CONSTITUENTS`. The function `combine` yields a `t` value applied to two `t` values. `none` is a constant function yielding a `t` value applied to no argument. (It is applied at subtrees that do not contain the accessed symbol, although the tree grammar would allow them to contain it). Typical examples for `WITH` clauses are given in Figure 9.

```
WITH (int, ADD, IDENTICAL, ZERO)
WITH (int, Maximum, IDENTICAL, ZERO)
WITH (int, OR, IDENTICAL, ZERO)
WITH (int, AND, IDENTICAL, ONE)
WITH (listtype, Append, SingleList, NullList)
```

Figure 9: Typical `WITH` Clauses

The applications of the `combine` functions obey the left-to-right order of the tree nodes where their arguments stem from. The `combine` function should be associative, the `none` function should not affect the resulting value. The calls of the three functions may occur in any suitable order; hence one should not rely upon side-effects.

Suitable implementations of the functions and the type must be made available for the evaluator. (see Section 5.3 [Implementing], page 24)

We must be aware that in our example the `CONSTITUENTS` is applied in a recursive tree structure, i. e. blocks are nested in our language. In fact the above specification causes that `Usage` constructs in inner blocks do not contribute to the `CONSTITUENTS` in outer `Block` context: Inner `Block` subtrees are shielded from it. We better make that explicit by

```
CONSTITUENTS Usage.Count SHIELD Block WITH (...)
```

In general we may shield any class of subtrees from the `CONSTITUENTS`-access, e. g. by

```
SHIELD (Block, Procedure, Module) ...
```

If no subtree should be shielded an empty `SHIELD` clause is used:

```
... SHIELD () ...
```

In this case our example would count the `Usage` constructs of all inner blocks too.

In general several attributes may be specified for being accessed by a `CONSTITUENTS`:

```
CONSTITUENTS (X.a, Y.b)
```

3.3 Left-to-Right Dependencies

As an example for a simple left-to-right dependent computation we rewrite the translation of expressions into postfix form of Figure 4.

In Figure 10 the `CHAIN print` specifies a sequence of computations that depends on each other in left-to-right depth-first order throughout the tree. It takes over the role of the pair of state attributes `print` and `printed` in Figure 5. Hence the `CHAIN` is introduced with type `VOID`.

```
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. Sym)
               <- 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;
```

Figure 10: `CHAIN` for Producing Postfix Expressions

The `CHAIN` computations are initiated in the `Root` context; `HEAD.print` refers to the `CHAIN` at the leftmost subtree. `TAIL.print` refers to the end of the `CHAIN` at the rightmost subtree (again `Expr`). It is the precondition for printing the final newline.

In the second context the `printf` computation is specified to lie on the `CHAIN` by stating `Expr.print` to be a precondition (“incoming” `CHAIN`) as well as to be a postcondition (“outgoing” `CHAIN`).

The `Opr` context specifies that operators are not printed in `CHAIN` order, but are appended after the right operand is printed. For that purpose two state attributes `Opr.pre` and `Opr.post` are used, as in Figure 5.

If it is necessary to locally deviate from `CHAIN` order, like here in case of the binary operation context, it has to be made sure, that the chain is not cut into separate pieces which are not linked by dependencies: If by some reason we would add another computation to the `Opr` context of our exapmle, e. g.

```
Opr.print = printf("Operator encountered\n")
            <- Opr.print;
```

it looks like being integrated into the print `CHAIN`. But the two computations of the binary `Expression` context shortcut the `CHAIN` across the `Opr` symbol. Hence, this computation may be executed later than intended.

The above example specifies a single `CHAIN` of computations through the tree. In general there may be several instances of a `CHAIN` in several subtrees, which may be nested, too. For example, we may allocate variable definitions to storage addresses relative to their smallest enclosing `Block`, as shown in Figure 11. Here the `CHAIN` computations propagate values in depth-first left-to-right order.

```
CHAIN RelAdr: int;

RULE: Block ::= '{' Sequence '}' COMPUTE
  CHAINSTART HEAD.RelAdr = 0;
END;

RULE: Definition ::= 'define' Ident COMPUTE
  Definition.RelAdr = ADD (Definition.RelAdr, VariableSize);
END;
```

Figure 11: Computing Addresses of Variables

An individual `CHAIN` is started for each `Block`. In the computation of the Definition context the two occurrences of `Definition.RelAdr` refer to different values on the `CHAIN`: The access in the `ADD` computation is the incoming current `CHAIN` value (the address of this variable), the result left to the `=` symbol denotes the outgoing next `CHAIN` value.

4 Symbol Computations

In this section we introduce constructs that associate computations to tree grammar symbols rather than to rule contexts. They can be used for computations which are conceptually connected with symbols, i. e. they have to be executed once for each such symbol node and they are not distinguished for the contexts in which the symbol occurs.

The use of symbol computations makes specifications even more independent of the particular tree grammar, and hence reduces the chance to be invalidated by grammar changes. Well designed symbol computations may be reused at different places in one specification, and even in specifications of different language processors.

In the following we demonstrate the use of symbol computations, and introduce a construct for their reuse.

4.1 Basic Symbol Computations

Consider the expression grammar given in the example of Section 2.1 [Value], page 5. For the purpose of this example assume that we want to trace the computation of expression values, and print `Expr.value` for each `Expr` node. We could associate that computation to both lower `Expr` context. But this computation does not refer to those particular contexts, it only depends on one `Expr` attribute. Hence we better associate it to the `Expr` symbol:

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

Symbol computations may use attributes of the symbol by the notation `THIS.AttributeName`.

The next example in Figure 12 shows how attributes are computed by symbol computations. It rewrites the usage count example of Figure 8. Both of its computations are in fact independent of the particular rule context.

```
ATTR Count: int;

SYMBOL Usage COMPUTE
  SYNT.Count = 1;
END;
```

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

Figure 12: Symbol Computations for Usage Count

An attribute that is defined by a symbol computation has to be classified `SYNT` or `INH`, like `SYNT.Count` above (instead of `THIS.Count` if the attribute were used in the computation). It determines whether the computation is intended for the lower `SYNT` or the upper `INH` contexts of the symbol.

The symbol computation of `Block` above shows that `CONSTITUENTS` constructs may be used in symbol computations as well as in rule contexts. The same applies to `INCLUDING` and `CHAIN` as shown below.

The above example reduces the amount of key strokes only slightly compared with that of Figure 8. But it makes this part of the specification invariant to modifications of the contexts where `Usage` and `Block` occur. The productions for `Block` and for `Usage` may be modified without affecting these symbol computations.

Now consider the computation of nesting depth of blocks in Figure 6. The computation of `Block.depth` can also be specified as a symbol computation:

```
SYMBOL Block COMPUTE
  INH.depth = ADD (INCLUDING Block.depth, 1);
END;
```

In this case the computation is intended to go to the upper contexts of `Block`, indicated by `INH.depth`. That is correct for any context where `Block` is a descendant of a `Statement`. But in the `Program` context the `depth` of the root `Block` must be computed to 0, rather than by the above symbol computation. So we keep the computation of Figure 6:

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

It overrides the above symbol computation: A rule computation overrides a symbol computation for the same attribute.

The example demonstrates a design rule:

> General context independent computations are specified by symbol computations. They may be overridden in special cases by computations in rule contexts.

The technique of overriding can also be used to specify default computations: A symbol computation specifies an attribute value for most occurrences of the symbol. In some contexts it is overridden by rule specific computations.

Finally we demonstrate how `CHAINS` are used in symbol computations. In Figure 11 addresses are computed for variable definitions. It is rewritten, as shown in Figure 13. In the second computation `THIS.RelAdr` occurs twice with different manings: it refers to the incoming `CHAIN` value in the call of `ADD`, whereas the outgoing `CHAIN` value is defined on the lefthand-side of the computation. `CHAIN` accesses are distinguished by their use or definition, rather than by `SYNT` and `INH` as in case of attributes.

```
CHAIN RelAdr: int;

SYMBOL Block COMPUTE
  CHAINSTART HEAD.RelAdr = 0;
END;

SYMBOL Definition COMPUTE
  THIS.RelAdr = ADD (THIS.RelAdr, VariableSize);
END;
```

Figure 13: Symbol Computations for Addresses of Variables

4.2 Reuse of Symbol Computations

Symbol computations are a well suited base for reuse of specifications: A computational concept is specified by a set of symbol computations. Then it is applied by inheriting it to grammar symbols. (Here the term inheritance is used in the sense of object oriented programming; it must not be confused with the class of inherited attributes.)

Assume we want to enumerate occurrences of non-recursive language constructs, e. g. definitions in a block and variable uses in each single statement. We first describe this computational concept by computations associated to new `CLASS` symbols that do not occur in the tree grammar. In Figure 14 use a `CHAIN` as in the examples of Section 3.3 [Chain], page 15.

```
CHAIN Occurrence: int;
ATTR OccNo, TotalOccs: int;

CLASS SYMBOL OccRoot COMPUTE
  CHAINSTART HEAD.Occurrence = 0;
  THIS.TotalOccs = TAIL.Occurrence;
END;

CLASS SYMBOL OccElem COMPUTE
  SYNT.OccNo = THIS.Occurrence;
  THIS.Occurrence = ADD (SYNT.OccNo, 1);
END;
```

Figure 14: Computationel Concept Occurrence Count

The above computations correspond to those for computing addresses in Figure 11. They are extended by computations of attributes `TotalOccs` (for the total number of enumerated constructs) and `OccNo` (for the current number of the enumerated element). Further computations may use these attributes, instead of referring to the `CHAIN`, which can be considered as an implementation mechanism of the enumeration computation.

The `CLASS` symbols `OccRoot` and `OccElem` represent two roles of this computational concept: The root of a subtree where elements are counted, and the elements to be counted. They are distinguished from symbols of the tree grammar by specifying them `CLASS SYMBOL`.

We now apply this count specification to symbols of our tree grammar:

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

`Block` inherits the role `OccRoot` and `Definition` inherits the role `OccElem`. Those constructs yield the same effect as if the computations for `OccRoot` (`OccElem`) were associated to `Block` (`Definition`).

As a consequence further computations may use the attributes `Definition.OccNo` (the number of a definition in a block), and `Block.TotalOccs` (the total number of definitions in that block). The second enumeration application is specified in the same way:

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

Of course we have to make sure that different applications do not interact. For example a third application enumerating the variable assignments would collide with the definition enumeration. This computational concept is not applicable to the enumeration of blocks which are recursive constructs. The specification module library of Eli provides more general applicable modules, and a mechanism that avoids such collisions. The application of those library modules is based on the technique of inheritance of computational roles as described here.

We finally show how several computational concepts may be combined: Assume that we want to print the total number of enumerated constructs. We again introduce a `CLASS` symbol for this computations.

```
CLASS SYMBOL PrintTotalOccs COMPUTE
   printf ("construct in line %d has %d elemts\n ",
           LINE, THIS.TotalOccs);
END;
```

This computation is applied by adding

```
SYMBOL Block INHERITS PrintTotalOccs END;
```

to the specification and correspondingly for `Definition`. The two `INHERITS` constructs can also be combined to one:

```
SYMBOL Block INHERITS OccRoot, PrintTotalOccs END;
```

We alternatively could extend the role of the enumeration root `OccRoot` such that the total number is always printed by

```
CLASS SYMBOL OccRoot INHERITS PrintTotalOccs END;
```

In this case each symbol that inherits the computation of `OccRoot` also inherits the computation of `PrintTotalOccs`.

5 Interactions within Eli

This section gives initial information how specifications of computations in trees interact with other Eli facilities. It should be sufficient for getting started. Other documents have to be consulted for a deeper understanding of those facilities and the interaction.

5.1 Supplying Tree Computation Specifications to Eli

Specifications as described in this document are written in files named `x.lido` where `x` is an arbitrary name. The specification of larger tasks should be decomposed into single subtasks specified in separate files each.

Further contributions to LIDO specifications are obtained from instantiation of library modules. The components of a LIDO specification are comprised by enumerating their names in a `.specs` file or by generating them from a `.fw` file. Section 5.2 [Tree Construction], page 23 describes another contribution to the set of `.lido` files for tree grammar specification.

It should be pointed out that `RULE`, `SYMBOL`, `ATTR`, and `CHAIN` constructs for the same names may occur arbitrary often in several or one single file as long as they do not specify contradicting properties. Computations specified for one `RULE` or `SYMBOL` name are accumulated. The LIGA system processes the concatenation of all `.lido` files. Hence the reference manual for the language LIDO (see LIDO - Reference Manual) refers to that compound specification disregarding the composition of single files.

It is highly recommended to simplify the development of specifications by the use of precoined solutions provided by the library of specification modules. For that purpose applicable tasks have to be identified in the module library, see Specification Module Library. The inheritance mechanism as introduced in section Section 4.2 [Inheritance], page 19 has to be applied, and the use of the module has to be stated in a `.specs` file as described in that document.

5.2 Tree Construction

The specification of computations in trees assume that a tree according to the tree grammar exists. Usually it is constructed by the structuring phase (scanner and parser) of the language processor. There are two different starting points for the design of the tree structure specifications: the tree grammar or the concrete grammar for the input language.

In general there may be parts of the language that need more attention to the concrete grammar and others where the computations in the tree grammar should be considered first. That may give rise to a mixed strategy: Supply the concrete grammar specification (`.con` and `.sym` files) for those parts of the language which are known and fixed, specify tree grammar rules (in `.lido` files) where computations are already known to be associated to, and take care that the whole grammar is covered by either of them. During the refinement of the computations further tree grammar rules may be added without updating the concrete grammar specification.

An Eli tool (Maptool, see Syntactic Analysis Manual) combines both grammar specification fragments, completes each of them, and relates concrete productions to tree grammar rules such that the parser builds the required tree. That relation is usually not a 1:1 mapping: Some concrete chain productions are left out in the tree grammar, e. g. those which describe operators precendences in expressions. The tree grammar may have chain context which have no correspondence in the concrete grammar, e. g. those which distinguish different classes of identifier occurrences. The latter may even be introduced to the tree grammar when their are needed during the refinement of the computations without updating the concrete grammar specification.

Both the concrete and the tree grammar distinguish literal terminals and named terminals. If the scanner is generated by Eli no further specification is needed for literal terminals, like `'begin'` or `':='`. For each named terminal, like `Name` or `Number`, a `.gla` specification has to describe its notation. The named terminals usually carry token specific information to be used in tree computations, e. g. the encoding of an identifier token or the value of a number. The LIDO specification should state the type of that information using a `TERM` construct, e. g.

```
TERM Name: int;
```

Since GLA generated scanners pass such token information by values of type `int`, LIGA assumes that type if the `TERM` construct is omitted for a named terminal. Hence, the above `TERM` construct is redundant for terminals created by GLA. It is needed for terminal created by other scanners or by computed tree extension.

5.3 Implementing Tree Computations

The implementation of functions, types, constants, and variables used in tree computations is not specified within the `.lido` specification. They have to be made available to the generated evaluator. No further user action is necessary if they are defined in C (like the basic C types) or in the standard I/O library `stdio.h` (like `printf`), or if they are predefined in LIDO (like `ADD`, see Section 'Predefined Entities' in LIDO – Reference Manual, or if they are provided by Eli tools

(e. g. PDL, PTG). Otherwise the user has to supply implementations of the used entities by C definition.

It is recommended to apply a modular style for those implementations: Supply C modules consisting of a `m.c` and a `m.h` file each, where the latter describes the objects exported by the module. (It is also possible to implement computations by CPP-macros.)

The file names `m.h` and `m.c` of all such user supplied C-modules have to be mentioned in some `.specs` file. Furthermore one or several `.head` files or `.HEAD.phi` files have to be provided. They have to contain a line

```
#include "m.h"
```

for each module `m`, making it available to the generated evaluator. It is recommended to protect each `m.h` file against multiple inclusion by suitable CPP-commands.

If a module needs some operations for initialization or finalization they can be written (as function calls) into files `m.init` or `m.finl` (or into `.INIT.phi` or `.FINL.phi` files).

5.4 Specification Errors

Eli checks the whole set of specifications extensively. It generates a language processor only if no errors are found. Error reports and warnings are obtained by a derivation like

```
x.specs:exe:warning
```

The error reports are related to the specification file (and line and column coordinates in it) where Eli found the error symptom, if that is possible. In the following we give hints how to react on the most common classes of errors. As a general rule one can obtain more information about an error symptom by applying the derivation

```
x.specs:exe:help
```

Violations of the LIDO specification language definition are reported with references to the `.lido` files. In most cases one should be able to deduce the correction from the report text, consulting the LIDO reference manual if necessary. Additionally the following information might be helpful:

A report saying

```
VOID attribute not allowed here
```

in most cases indicates that an attribute is used without specifying its type, `VOID` is assumed then. The reason of such an error often is a misspelled attribute name.

It may be helpful to derive

```
x.specs:showFe
```

and look at the file `attr.info`. It gives an overview on all attributes the system found so far in the `.lido` files. (Attributes that stem from inheritance are not yet found in this phase.)

A report saying

```
attribute class in conflict
```

indicates that computations in lower contexts and in upper contexts define that attribute. One has to rewrite them such that only one class is used.

If problems are reported with remote dependencies one should check the use of those constructs within the tree grammar structure. In special difficult cases more information can be obtained by deriving

```
x.specs:ExpInfo
```

That file describes how each remote access construct can be replaced by a set of equivalent computations propagating the accessed values through adjacent contexts. (The coordinates given in that information do not refer to single `.lido` files, hence are not directly usable.)

LIGA also checks whether the dependencies between the computations are acyclic for any tree, and reports if they are cyclic. In that case more information can be obtained by deriving

```
x.specs:OrdInfo
```

or by using the tool `gorto` (see GORTO - Graphical Dependency Analyzer) for tracing dependencies graphically.

In rather seldom cases LIGA may report that it could not find an evaluation order, although the dependencies are acyclic. If such a situation occurs it is usually caused by several sets of far ranging dependencies where the computations in one set are independent of those of the other sets. Adding additional dependencies that specify some computation sets to depend on others often solves the problem. More information on the problem is obtained by using `gorto`. It is highly recommended NOT to try to avoid such situations before they are reported, since they occur rather seldom.

In the following situations errors may be reported with respect to the generated C files rather than to the `.lido` specifications. A report like

```
visitprocs.c, line 27,0: XYZ undefined
```

stems from the C compiler. It indicates that the name XYZ is not made available to the evaluator, e. g. a type name or a variable name used in a `.lido` specification. Such an error is corrected by supplying the necessary information in `.head` files. In case of undefined type names often avalanche errors are reported by the C compiler with respect to several product files.

If functions are used in a `.lido` file but are not made available for the evaluator, the error might not be reported before the whole program is linked.

It should be pointed out that the computations in the `.lido` files are not checked for type and parameter consistency of function calls. Such errors may be reported by the C compiler. It may even be necessary to trace such errors by applying `lint` to the complete set of source files generated from the specification, or by debugging the program.

Index

P

R

S

T

U

V

W