LIDO - Reference Manual

Uwe Kastens

Compiler and Programming Language Group
Faculty of Electrical Engineering, Computer Science and Mathematics
University of Paderborn, Germany

Table of Contents

1 Introduction

This is a reference manual for LIDO, a language for the specification of computations in trees. It is used to specify all computations of the analysis phase and the translation phase of a language processor, which are to be executed on the abstract tree. The main purpose of a LIDO specification is to describe which computations have to be executed in which tree context, how those computations depend on each other, and which values are propagated from one computation to another. The functions called in computations and the types of propagated values are implemented in C; those implementations are not part of a LIDO specification.

The LIGA system processes a LIDO specification and generates an evaluator in form of a C module from it. LIGA automatically determines a tree walk strategy and the evaluation order of computations on the base of the specified dependencies. Attribute grammars are the formal model for this process.

This document is intended to provide precise definitions of LIDO constructs and of rules of the language LIDO. For studying the use of LIDO in more complex and complete translation specifications we recommend to read the explained example specifications in $/Name/Examples/AlgLike.fw and in $/Type/Examples/Type.fw.

Other documents related to LIDO are:

- Section "top" in *LIDO - Computation in Trees*. Introduces and explains typical uses of LIDO constructs.
- Section "top" in *LIGA - Control Language*. Describes how variants in LIGA's processing can be controlled.
- Section "top" in *Show*. Describes how to obtain debugging information for LIDO.
- Section "top" in *GORTO - Graphical Order Tool*. Describes how to trace dependencies graphically.
- Section "top" in *ModLib - Specification Module Library*. Describes how to use precoined solutions of common problems.

2 Overall Structure

A LIDO text specifies an evaluator for executing computations driven by a tree walk. A tree grammar specifies the structure of trees. Computations are associated with rules and symbols of the tree grammar. Computations may depend on one another via typed attributes.

In Eli a LIDO specification is usually composed of several components supplied by the user, derived from libraries, or generated by Eli tools. The components are combined into one file and then processed by LIGA.

Syntax

```
LIDOSpec       ::=  Specification
Specification  ::=  Specification Specification |
                 |  RuleSpec ';' | SymComp ';'
                 |  SymSpec ';'  | TermSpec ';'
                 |  AttrSpec ';' | ChainSpec ';'
```

Examples

```
RULE p: Stmt  ::= 'while' Expr 'do' Stmt COMPUTE
  Expr.postType = boolType
END;

SYMBOL Expr COMPUTE
  Compatible (THIS.preType, THIS.postType);
END;

ATTR preType, postType: DefTableKey;
```

There is no restriction on the order of specifications. Any permutation of specifications has the same meaning.

LIDO objects such as rules, symbols, or attributes are identified by their names. They are introduced by using them in LIDO constructs. There are no explicit declarations in LIDO.

Specifications associate certain properties with an object, e. g. computations are associated with a rule, or a type with an attribute name. There may be several specifications for the same object as long as the specified properties are not contradictory.

In the syntax of this document we distinguish names for objects of different kinds, e. g. `RuleName`, `SymbName`, `TypeName`. The syntax rules for names are omitted in the rest of this document. The following rules are assumed for `XYZNames`

```
XYZName  ::= Identifier
XYZNames ::= XYZName | XYZNames ',' XYZNames
```

All names are written as identifiers in C.

Restrictions

It is strongly recommended *not* to use names that begin with an underscore or which have the form `rule_i` where i is a number, in order to avoid interference with identifiers generated by LIGA.

`RuleNames`, `SymbNames`, and `TypeNames` must be mutually distinct. `AttrNames` must be different from `ChainNames`.

3 Rule Specifications

A rule specification specifies a production of the tree grammar, and may associate some computations with the rule context. They are executed in every context which represents that rule in a particular tree.

Syntax

```
RuleSpec ::= 'RULE' [RuleName] ':' Production Computations 'END'
```

Example:

```
RULE p: Stmt  ::= 'while' Expr 'do' Stmt COMPUTE
  Expr.postType = boolType
END;
```

There may be several rule specifications that refer to the same rule. In that case the associated computations are accumulated.

The set of productions of all rules forms the tree grammar. It must have exactly one root symbol that does not occur on any right-hand side of a production.

Eli usually generates some rule specifications (without computations) from the concrete grammar in order to complete the tree grammar.

In general the `RuleName` is omitted. The rule is then identified by the production. LIGA generates a name of the form `rule_i`, with a unique number `i` for such a rule. A meaningful `RuleName` should be specified for rules that are part of computed subtrees, since the name of the tree construction function is derived from it (see Chapter 10 [Computed Subtrees], page 31). Also using the `RuleFct` feature may give rise to explicitly name rules (see Chapter 12 [Predefined Entities], page 37).

Restrictions

Two unnamed rule specifications refer to the same rule if their productions are identical.

A named rule specification and an unnamed one refer to the same rule if their productions are identical. In that case there must not be another rule specification with the same production but a different name.

Two named rule specifications with the same `RuleName` must have the same production.

Note: Two rule specifications with different names, but equal productions, are only reasonable if they belong to computed subtrees rather to subtrees constructed by a parser.

3.1 Productions

A production as part of a rule specification describes the structure of the rule context. Computations associated with the rule may use or define attributes of nonterminal symbols that occur in the production. The set of all productions in a LIDO specification defines the tree grammar.

Syntax

```
Production     ::= SymbName '::=' Symbols
                 | SymbName 'LISTOF' Elements

Symbols        ::= Symbols Symbols |
                 | SymbName | Literal

Elements       ::=  Elements '|' Elements |
                 | SymbName

TermSpec       ::= 'TERM' SymbNames ':' TypeName
```

Examples

```
Stmt ::= 'while' Expr 'do' Stmt
DefIdent ::= Identifier
Declarations LISTOF ProcDecl | VarDecl
TERM Identifier: int;
```

Productions are composed of nonterminal symbols, named terminal symbols, and literal terminals.

The `SymbName` on the left-hand side of a production is a nonterminal. A `SymbName` that does not occur on the left-hand side of any production denotes a named terminal. A nonterminal symbol that does not occur on the right-hand side of any production is the root of the tree grammar.

We say the rule context is a *lower context* for the left-hand side nonterminal, and an *upper context* for any right-hand side nonterminal.

Literal terminals are denoted by arbitrary non empty strings enclosed in single quotes. A quote that is part of such string is denoted by two single quotes.

Literal terminals do not contribute to the trees specified by the tree grammar. They only relate tree productions to concrete productions describing the input text, and distinguish otherwise equal productions.

Named terminal symbols do not contribute to the trees specified by the tree grammar. They are related to named terminal symbols of corresponding concrete productions describing the input text. A value derived from such an input token may be used in computations which are associated with the rule of the production or with the symbol on the left-hand side of the production. (If the tree context is constructed by a computation, rather than by parsing the input text, then that value is supplied as an argument to the call of the construction function (see Section 10.1 [Tree Construction Functions], page 32).)

The type of the value provided by a named terminal symbol is specified by a `TERM` specification. If there is no such specification the type `int` is assumed.

There are two forms of productions: plain productions and `LISTOF` productions.

A plain production defines tree contexts with a node for the left-hand side nonterminal having a sequence of subtrees, one for each nonterminal on the right-hand side.

Computations may refer to any attribute of any nonterminal in the production. If one nonterminal occurs more than once in the production references to the occurrences in computations are distinguished by indices (starting from 1).

A LISTOF production defines tree contexts with a node for the left-hand side nonterminal having an arbitrary long sequence of subtrees each rooted by a nonterminal specified as a LISTOF element. That sequence may be empty, even if there is no empty LISTOF element specified.

Computations associated with the rule of a LISTOF production may only refer to attributes of the left-hand side symbol. Attributes of the element subtrees are referenced only by remote attribute access (see Chapter 9 [Remote Attribute Access], page 25).

Restrictions

There must be exactly one root nonterminal which does not occur on any right-hand side of a tree grammar production.

If X is the left-hand side symbol of a LISTOF production, then there may not be a different production (neither LISTOF nor plain) that also has X on its left-hand side.

Named terminals may not be LISTOF elements.

A literal terminal may not be the empty string.

4 Symbol Specifications

A symbol specification associates computations with a symbol. They are executed for every node which represents that symbol in a particular tree.

Symbols may be introduced which do not occur in the tree grammar. They are called `CLASS` *symbols* and represent a computational role. Their computations may be inherited directly or indirectly by grammar symbols. Symbols that do occur in the tree grammar are called `TREE` *symbols*.

Syntax

```
SymComp    ::= SymbKind SymbName [ Inheritance ] Computations 'END'

SymbKind   ::= 'SYMBOL' | 'CLASS' 'SYMBOL' | 'TREE' 'SYMBOL'
```

Example:

```
TREE SYMBOL Expr COMPUTE
  SYNT.coercion = coerce (THIS.preType, THIS.postType);
  INH.IsValContext = true;
  Compatible (THIS.preType, THIS.postType);
END;
```

A symbol specified `TREE` occurs in a tree grammar production, a symbol specified `CLASS` does not. If neither is specified the symbol kind is determined by its occurrence in the tree grammar. (Only the computations of `CLASS` symbols may be inherited by other symbols.)

The `CLASS` symbol `ROOTCLASS` is predefined. It is implicitly inherited by the root of the tree grammar. Hence, any computation associated with `ROOTCLASS` is inherited by the root context. This facility is to be used to specify computational roles for the root which are grammar independent, and which need not be inherited explicitly.

Note: There may be `TREE` symbols that do not occur in the user supplied rules, but only in those generated from the concrete grammar. In those cases it is recommended to explicitly specify their kind to be `TREE`, in order to get more specific error reports in cases of accidental mismatches.

Two sets of computations are associated with a symbol: the *lower computations*, which are executed in every lower context of the symbol, i. e. in a context whose production has the symbol on its left-hand side, and the *upper computations*, which are executed in every upper context, i. e. in a context whose production has the symbol on its right-hand side. The upper computations are executed once for each right-hand side occurrence of the symbol.

Each symbol has two disjoint sets of attributes: *synthesized* (`SYNT`) attributes that are defined by computations in lower contexts of the symbol, and *inherited* (`INH`) attributes that are defined by computations in upper contexts of the symbol.

In a symbol computation only attributes of that symbol may be used or defined (except the use of remote attributes). Those attributes are denoted `SYNT.a` if `a` is a synthesized attribute, `INH.b` if `b` is an inherited attribute. An attribute of the symbol may also be denoted `THIS.c`. In this case the attribute class must be specified in another occurrence of that attribute.

A computation that defines a synthesized (an inherited) attribute of the symbol belongs to the set of lower (upper) computations. A *plain computation* defining no attribute belongs to the set of lower computations (see Chapter 5 [Computations], page 11).

There may be several symbol specifications for one symbol. In that case the associated computations are accumulated.

If both a symbol computation and a rule computation define the same attribute of that symbol, the rule computation will be executed in that context, overriding the symbol computation.

Plain computations can not be overridden.

Restrictions

The kind of a symbol, `TREE` or `CLASS` may not be specified contradictory.

`CLASS SYMBOL`s may not be used in productions.

`TREE SYMBOL`s may not be used in `INHERITS` clauses (see Chapter 8 [Inheritance of Computations], page 23).

5 Computations

Computations are associated with rules or with symbols. Each computation (that is not overridden) is executed exactly once for every instance of its context in a particular tree. A computation may yield a value denoted as an attribute which may be used by other computations. Computations may also be specified as depending on one another without passing a value in order to specify dependences on side-effects of computations. (see Section 7.1 [Dependent Expressions], page 19).

Syntax

```
Computations ::=  [ 'COMPUTE' Computation ]

Computation  ::=  Computation Computation |
                | Attribute '=' Expression Terminator
                | Expression Terminator
                | Attribute '+=' Expression Terminator

Terminator   ::= ';'
                | 'BOTTOMUP' ';'
```

There are three forms of computations: *attribute computations* denoted as an assignment to an attribute, *plain computations* that are simple expressions, and *accumulating computations* which are a special variant of attribute computations, distinguished by the `+=` token.

5.1 Attribute Computations and Plain Computations

The following example shows a sequence of two attribute computations and two plain computations:

Examples

```
COMPUTE
  Expr.postType = boolType;
  Stmt[1].code = PTGWhile (Expr.code, Stmt[2].code);
  printf ("while loop in line %d\n", LINE);
  printf ("value = %d\n", Expr.val) BOTTOMUP;
END;
```

A computation is executed by evaluating its expression. It depends on every attribute that occurs in the expression regardless whether the attribute is used for the evaluation. We say those attributes are the *preconditions* of the computation. The attribute on the left-hand side of an attribute computation represents the *postcondition* of that computation. Plain computations do not establish a postcondition for any other computation. The evaluator is generated such that the computations are executed in an order that obeys these dependencies for any tree of the tree grammar.

If both a symbol computation and a rule computation define the same attribute of a symbol, the rule computation will be executed in that context, overriding the symbol computation.

An expression may occur in *value context*, where it must yield a value, or it may occur in `VOID` *context*, where it may or may not yield a value. If it does yield a value in `VOID` context, the value is discarded. These terms will be used in sections below where further constructs are introduced which contain expressions.

If the left-hand side attribute of an attribute computation has a type different from `VOID` the right-hand side expression is in value context; the result of the expression evaluation is assigned to the attribute. If the left-hand side attribute has the type `VOID` the right-hand side expression is in `VOID` context. In this case the attribute simply states the postcondition that the computation has been executed.

A plain computation is in `VOID` context, i. e. it may or may not yield a value.

Computations may be specified to be executed `BOTTOMUP`, that means while the input is being read and the tree is being built. LIGA then tries to arrange the computations such that those are executed already when their tree node is constructed. This facility is useful for example if the generated language processor is to produce output while its input is supplied (like desktop calculators), or if a computation is used to switch the input file.

Note: A `BOTTOMUP` computation may depend on other computations. These dependencies should be specified the usual way. Such precondition computations should NOT be specified `BOTTOMUP` unless they themselves are to be related to input processing. Without such an over-specification LIGA can apply more sophisticated means to correctly schedule the precondition computations automatically.

Note: Due to the parser's lookahead, one token beyond the last token of the context of the `BOTTOMUP` computation is read before before the computation is executed.

Restrictions

If the attribute in an attribute computation has a non-`VOID` type the evaluation of the expression must yield a value of that type. This condition is not checked by LIGA. It is checked by the compiler that compiles the generated evaluator.

Multiple symbol computations that define the same attribute are forbidden.

There must be exactly one attribute computation for each synthesized attribute of the left-hand side nonterminal and for each inherited attribute of each nonterminal occurrence on the right-hand side in the production of a rule context, or such a computation is inherited in the rule context. (For accumulating computations a different rule applies.)

There may not be any cyclic dependencies between computations for any tree of the tree grammar.

Contexts that may belong to subtrees which are built by computations (see Chapter 10 [Computed Subtrees], page 31) may not have computations that are marked `BOTTOMUP` or contribute to `BOTTOMUP` computations.

LIGA may fail to allocate `BOTTOMUP` computations as required due to attribute dependencies or due to LIGA's evaluation strategy. In such cases messages are given.

5.2 Accumulating Computations

There are situations where a `VOID` attribute, say `Program.AnalysisDone`, represents a computational state which is reached when several computations are executed, which conceptually belong to different sections of the LIDO text. Instead of moving all these computations to the only place where `Program.AnalysisDone` is computed, several accumulating computations may stay in their conceptual context and contribute dependences to that attribute.

A computation is marked to be accumulating by the `+=` token. The following example demonstrates the above mentioned use of accumulating computations:

```
RULE: Program ::= Statements COMPUTE
   Program.AnalysisDone += DoThis ( );
END;
  ....
RULE: Program ::= Statements COMPUTE
   Program.AnalysisDone += DoThat ( ) <- Statements.checked;
END;
```

Two accumulating computations contribute both to the attribute `Program.AnalysisDone`, such that it represents the state when the calls `DoThis ( )` and `DoThat ( )` are executed after the pre-condition `Statements.checked` has been reached. The two accumulating computations above have the same effect as if there was a single computation, as in

```
RULE: Program ::= Statements COMPUTE
   Program.AnalysisDone = ORDER (DoThis ( ), DoThat ( ))
                          <- Statements.checked;
END;
```

The order in which `DoThis ( )` and `DoThat ( )` are executed is arbitrarily decided by the Liga system.

Accumulating computations may be formulated in rule context or in the context of `TREE` or `CLASS` symbols. Rule attributes may also be computed by accumulating computations.

Only `VOID` attributes may have accumulating computations. If an attribute has an accumulating computation, it is called an accumulating attribute, and all its computations must be accumulating. Attributes are not explicitly defined to be accumulating. If an attribute is not defined explicitly, it has the type `VOID` by default. Hence, accumulating attributes need not be defined explicitly, at all.

The set of accumulating computations of an attribute is combined into a single computation, containing all dependences and function calls of the contributing accumulating computations, as shown above.

Accumulating computations may be inherited from `CLASS` symbols. In contrast to non-accumulating computations, there is no hiding for accumulating computations: All accumulating computations that lie on an inheritance path to an accumulating attribute in a rule context are combined. For example, add the following specifications to the above example:

```
SYMBOL Program INHERITS AddOn COMPUTE
  SYNT. AnalysisDone += AllWaysDo ( );
END;
CLASS SYMBOL AddOn COMPUTE
  SYNT. AnalysisDone += AndAlsoDo ();
END;
```

Then all four computations for `Program.AnalysisDone` (two in the RULE context above, one in the TREE symbol context `Program`, and one inherited from the `CLASS` symbol `AddOn`) will be combined into one. It characterizes the state after execution of the four function calls and the computation of `Statements.checked`.

Restrictions

If an attribute has an accumulating computation, it is called an accumulating attribute, and may not have or inherit non-accumulating computations.

An accumulating attribute must have type `VOID`.

Let `X` be the left-hand side nonterminal in a rule `r` and `X.s` an accumulating synthesized attribute, then there must be at least one accumulating computation for `X.s` in `r` or inherited there.

Let `X[i]` be an occurrence of the nonterminal `X` on the right-hand side of the rule `r` and `X.s` an accumulating inherited attribute, then there must be at least one accumulating computation for `X[i].s` in `r` or inherited there.

`CHAIN` computations and `CHAIN` attributes may not be accumulating.

6 Attributes

Attributes are associated with symbols and with rules. They are defined and used in rule computations and in symbol computations.

Each symbol has two disjoint sets of attributes: *synthesized* (`SYNT`) attributes that are defined by computations in lower contexts of the symbol, and *inherited* (`INH`) attributes that are defined by computations in upper contexts of the symbol.

Attributes are introduced by their occurrence in computations. They are not explicitly declared. How types and classes of attributes are determined is described in Section 6.1 [Types and Classes of Attributes], page 16.

Syntax

```
Attribute ::= SymbolRef '.' AttrName
            | RuleAttr
RuleAttr  ::= '.' AttrName
SymbolRef ::= SymbName
            | SymbName '[' Number ']'
RhsAttrs  ::= 'RHS' '.' AttrName
```

Examples

```
RULE: Stmt ::= 'while' Expr 'do' Stmt COMPUTE ...
  ... Expr.postType ...
  ... Stmt[1].code ...
  ... .label ...
  ... RuleFct ("PTG", RHS.Ptg) ...
END;
SYMBOL Expr COMPUTE
  ... SYNT.preType ...
  ... INH.postType ...
  ... THIS.preType ...
  ... RuleFct ("PTG", RHS.Ptg) ...
END;
```

Attributes in rule computations have the form `X.a` or `X[i].a` where `X` is a nonterminal in the production of the rule. They refer to the attribute `a` of the tree node corresponding to `X`. The index distinguishes multiple occurrences of the nonterminal in the production, counting from left to right starting at 1.

Rule attributes of the form `.b` may be used in rule computations, to simplify reuse of computed values. They are defined and used within the computations of a single rule. They are not associated with any symbol.

In symbol computations attributes of the considered symbol are denoted using `SYNT`, `INH`, or `THIS` instead of the `SymbName`: `SYNT.a` for a synthesized attribute, `INH.b` for an inherited attribute, or `THIS.c` leaving the attribute class to be specified elsewhere.

A `RhsAttrs` construct, such as `RHS.a`, is a shorthand for a sequence of attributes all named `a`, one for each right-hand side nonterminal of the rule context associated with the computation. If there is more than one such nonterminal the construct may only occur in function calls, where it contributes part of the argument sequence, or in `DependsClauses`(see Section 7.1 [Dependent Expressions], page 19). If a symbol computation contains a `RhsAttrs` its sequence of attributes is determined for each rule context of the symbol individually. In combination with the predefined function `RuleFct` a `RhsAttrs` construct may be used to specify a call pattern that is instantiated differently for each rule context (see Chapter 12 [Predefined Entities], page 37).

Restrictions

The `SymbolRef` must occur in the production of the rule.

The `SymbolRef` must be indexed if and only if the symbol occurs more than once in the production.

The index of a `SymbolRef` must identify an occurrence of the symbol in the production.

`SymbNames` and indices may not be used in attributes of symbol computations.

Rule attributes may not be used in symbol computations.

6.1 Types and Classes of Attributes

Each attribute has a certain type characterizing the values propagated by the attribute. Attributes that describe only postconditions of computations without propagating a value have the predefined type `VOID`. Non-`VOID` types must be specified explicitly.

Each attribute has either the class synthesized (`SYNT`), if it is computed in all lower contexts of its symbol, or it has the class inherited (`INH`), if it is computed in all upper contexts of its symbol. Attribute classes are usually derived from computations without explicit specifications.

Syntax

```
AttrSpec     ::= 'ATTR' AttrNames ':' TypeName [ AttrClass ]

SymSpec      ::= SymbKind SymbNames ':' [ AttrSpecs ]

AttrSpecs    ::= AttrSpecs ',' AttrSpecs
               | AttrNames ':' TypeName [ AttrClass ]

AttrClass    ::= 'SYNT' | 'INH'
```

Examples

```
ATTR code: PTGNode SYNT;
SYMBOL Expr, UseIdent: preType, postType: DefTableKey;
```

An attribute name specification (`ATTR`) determines the type and optionally the class of all attributes having one of the `AttrNames`.

An `AttrSpec` for a nonterminal determines the type and optionally the class of attributes given by the `AttrNames` for all nonterminals given by `SymbNames`. These specifications override the type and the attribute class stated by `ATTR` specifications.

If the type of an attribute is left unspecified it is assumed to be `VOID`.

Note: Misspelling of an attribute name in a computation leads to introduction of a `VOID` attribute, and is usually indicated by messages on missing computations for that attribute or illegal use of a `VOID` attribute.

Note: The type of a non-`VOID` rule attribute has to be specified by `ATTR` specifications.

Restrictions

There may be several `ATTR`specifications for the same `AttrName` provided their properties are not contradictory.

A specified attribute class must be consistent with all computations of that attribute.

`VOID` attributes may not be used in value contexts.

The type specified for an attribute must denote an assignable C type that is available in the generated evaluator. LIGA does not check whether non-`VOID` attributes are used consistently with respect to their types. Violations will be indicated when the generated evaluator is compiled.

7 Expressions

An expression is evaluated as part of a computation. The evaluation may yield a value, cause an effect, or both.

7.1 Dependent Expressions

The evaluation of an expression depends on all attributes to which it refers. The expression is evaluated only after all those attributes are evaluated.

Further attributes may be added as preconditions for expression evaluation without using their values for computing the expression's result. The additional attributes may describe a computational state that has to be reached before the expression is evaluated. These attributes are specified by a `DependsClause`.

Syntax

```
Expression    ::= SimpExpr [ DependsClause ]
DependsClause ::= '<-' DepAttrList
DepAttrList   ::= DepAttr
                | '(' DepAttrs ')'
DepAttrs      ::= DepAttrs ',' DepAttrs
                | DepAttr
DepAttr       ::= Attribute | RemoteAccess | RhsAttrs
```

Examples

```
GetProp (UseId.Key,0) <- UseId.PropIsSet
printf ("%s ", Opr.String) <- (Expr[2].printed, Expr.[3].printed)
```

A `DependsClause` has a `VOID` context, i.e. its attributes may have any type; their values are discarded.

7.2 Terminal Access

Named terminal symbols that occur in a production represent values that are usually obtained from corresponding input tokens when the tree node is constructed. Those values can be used in both rule and symbol computations.

Syntax

```
SimpExpr ::= SymbolRef
           | 'TERM' [ '[' Number ']' ]
```

Examples

```
RULE:   DefIdent ::= Ident COMPUTE
  DefIdent.Key = DefineIdn (DefIdent.Env, Ident);
END;
RULE:   Point ::= '(' Numb Numb ')' COMPUTE
  printf ("X = %d, Y = %d\n", Numb[1], Numb[2]);
END;
```

```
SYMBOL Point COMPUTE
  printf ("X = %d, Y = %d\n", TERM[1], TERM[2]);
END;
```

In rule computations the value of a terminal in the production is denoted by the `SymbName`, which is indexed if and only if there are multiple occurrences of the `SymbName` in the production.

Note: In a rule computation a non-indexed identifier that is not a name of a symbol in the production of this rule denotes some entity of the generated C program, even if it coincides with the name of a terminal that occurs in other productions.

In lower computations of a symbol `X` terminal values are accessed by `TERM` or `TERM[i]`, where `TERM` is equivalent to `TERM[1]`. `TERM[i]` denotes the `i`-th terminal in each production that has `X` (or a symbol that inherits `X`) on its left-hand side, regardless of the terminal's name.

Restrictions

`TERM` must not be used in rule computations or in upper symbol computations.

A terminal accessed in a symbol computation must exist in every production the computation is associated with.

7.3 Simple Expressions

Expressions are written as nested function calls where the basic operands are attributes, C identifiers and C literals. The functions are either predefined in LIDO or their definitions are supplied by the user in the form of C functions or macros outside the LIDO specification. There is no operator notation for expressions in LIDO.

Syntax

```
SimpExpr  ::= C_Name | C_Integer | C_Float | C_Char | C_String
            |  Attribute | RemoteAccess | RhsAttrs
            |  FunctionName '(' [ Arguments ] ')'
Arguments ::= Arguments ',' Arguments
            |  Expression
```

Examples

```
printf ("Val = %d\n", Expr.val)
IF (LT (Expr.val, 0), 0, Expr.val)
```

Evaluation of a function call notation in LIDO has the same effect and result as the equivalent notation in C.

There are some predefined `FunctionNames` that have a special meaning in LIDO (see Chapter 12 [Predefined Entities], page 37).

Function calls need not yield a value if they are in a `VOID` context. All arguments of a function call are in a value context.

`C_Name`, `C_Integer`, `C_Float`, `C_Char`, `C_String` are names and literals denoted as in C.

Restrictions

Every FunctionName and C_Name must be predefined in LIDO or supplied by a user definition.

All arguments of non-predefined functions must yield a (non-VOID) value. For predefined LIDO functions specific rules apply (see Chapter 12 [Predefined Entities], page 37).

Type consistency for non-VOID types is not checked by LIGA. Those checks are deferred to the compilation of the generated evaluator.

A C_Name or a FunctionName should not begin with an underscore, in order to avoid conflicts with LIGA generated identifiers.

8 Inheritance of Computations

A set of related computations can be associated with a CLASS symbol describing a certain computational role. It can be inherited by TREE symbols or by other CLASS symbols, thus specifying that they play this role and reusing its computations. A symbol can play several roles at the same time (multiple inheritance). Inherited computations can be overridden by other computations of attributes having the same name. CLASS specifications have the same notation and meaning as SYMBOL specifications.

Syntax

```
Specification ::= SymbKind SymbName [ Inheritance ]
                   Computations 'END' ';'
Inheritance   ::= 'INHERITS' SymbNames
```

Example:

```
CLASS SYMBOL RootSetLine COMPUTE
  SYNT.GotLine = CONSTITUENTS KeySetLine.GotLine;
END;

CLASS SYMBOL KeySetLine COMPUTE
  SYNT.GotLine = ResetLine (THIS.Key,LINE);
END;

CLASS SYMBOL KeyPrintLine COMPUTE
  printf ("identifier in Line %d defined in line %d\n",
          LINE, GetLine (THIS.Key,o))
     <- INCLUDING RootSetLine.GotLine;
END;

SYMBOL VarDefId  INHERITS KeySetLine   END;
SYMBOL ProcDefID INHERITS KeySetLine   END;
SYMBOL UseIdent  INHERITS KeyPrintLine END;
SYMBOL Program   INHERITS RootSetLine  END;
```

CLASS computations obey the same rules as symbol computation.

The sets of lower and upper class computations may be accumulated from several CLASS specifications for the same class.

CLASS computations may be inherited by TREE symbols or by other CLASS symbols.

A CLASS or a TREE symbol Target inherits the computations from a CLASS Source if there is a Target INHERITS Source relation specified. The complete inheritance relation is accumulated by all INHERITS specifications.

A computation is inherited only once even if there are several paths to it in the inheritance relation.

A computation for an attribute a associated with a CLASS or a TREE symbol overrides any computation for a inherited from a CLASS symbol.

Note: Plain computations can not be overridden.

The computations inherited by a `CLASS` symbol belong to the computation sets of the `CLASS` symbol and may be subject to further inheritance.

Restrictions

`TREE` symbols and `CLASS` symbols may not inherit from `TREE` symbols.

The inheritance relation must not be cyclic.

If `C` inherits from `CLASS` symbols `C1` and `C2`, and if both `C1` and `C2` have computations for an attribute `a`, it is undefined which one is inherited by `C`.

9 Remote Attribute Access

Remote access constructs are used to relate computations that belong to distant contexts in the tree, rather than those of adjacent contexts. The `INCLUDING` construct accesses attributes of symbols that are further up in the tree (i. e. closer to the root). The `CONSTITUENT(S)` construct accesses attributes of symbols that are further down in the tree (i. e. closer to the leaves). The `CHAIN` construct relates computations in a left-to-right depth-first order within subtrees.

These constructs may propagate values or simply specify dependencies between computations.

Remote access constructs are used to abstract from the particular tree structure between related computations. Computational patterns can be specified independent of the particular grammar using remote access in combination with symbol computations and `CLASS` symbols. Reusable specification modules are based on that technique.

9.1 INCLUDING

The `INCLUDING`-construct accesses an attribute of a symbol that is on the path towards the tree root. Hence, several computations in a subtree may depend on an attribute at the subtree root.

Syntax

```
RemoteAccess ::= 'INCLUDING' RemAttrList
RemAttrList  ::=  RemAttr | '(' RemAttrs ')'
RemAttrs     ::=  RemAttr ',' RemAttrs '|' RemAttr
RemAttr      ::=  SymbName '.' AttrName
```

Examples

```
INCLUDING Range.Env
INCLUDING (Block.Scope, Root.Env)
```

The `RemAttrList` specifies the set of attributes referred to by the `INCLUDING` construct, called the *referred set*. On evaluation it accesses an attribute of the first symbol on the path to the root which is in that set.

An `INCLUDING` in a rule computation accesses an attribute of a symbol above the current context, even if the left-hand side symbol is in the `RemAttrList`.

An `INCLUDING` in a symbol computation accesses an attribute of a symbol above the current one, even if the current one is in the `RemAttrList`.

An attribute of a `CLASS` symbol `C.a` in the `RemAttrList` contributes attributes `X.a` to the referred set for all `TREE` symbols `X` by which `C` is inherited.

An `INCLUDING` in a `VOID` context does not cause a value to be propagated; it just states a dependency.

Restrictions

The referred set may not be empty, unless the computation which contains it is not part of or inherited by any rule context.

The tree grammar must guarantee that in every tree there is at least one of the symbols of the referred set above the context of the `INCLUDING`.

The referred set must not contain different attributes of the same symbol.

The types of the attributes in the referred set must be equal, unless `INCLUDING` is in a `VOID` context.

9.2 CONSTITUENT(S)

The `CONSTITUENTS`-construct accesses attributes of symbols that are in the subtree of the current context. Hence, it may depend on several computations in the subtree. If values are to be propagated they are combined by user defined functions.

The `CONSTITUENT`-construct accesses a single attribute instance of a symbol that is in the subtree of the current context.

Syntax

```
RemoteAccess ::= [ SymbolRef ] 'CONSTITUENT'
                  RemAttrList [ ShieldClause ]
                | [ SymbolRef ] 'CONSTITUENTS'
                  RemAttrList [ ShieldClause ] [ WithClause ]
ShieldClause ::= 'SHIELD' SymbNameList
SymbNameList ::= SymbName | '(' SymbNames ')' | '(' ')'
WithClause   ::= 'WITH' '(' TypeName ',' CombFctName ','
                        SingleFctName ',' NullFctName ')'
```

Examples

```
CONSTITUENT Declarator.type
Declarations CONSTITUENTS DefIdent.GotType
CONSTITUENTS Range.GotLocKeys SHIELD Range
CONSTITUENTS Stmt.code SHIELD Stmt
   WITH (PTGNode, PTGSeq, IDENTICAL, PTGNull)
```

The `RemAttrList` specifies the set of attributes referred to by the `CONSTITUENT(S)` construct, called the *referred set*. On evaluation it accesses all instances of attributes of that set which are in a certain range of the subtree of the current context. That range is determined by its root node, which itself does not belong to the range, and by the set of shield symbols. The tree nodes below a shield symbol are excluded from that range.

In a rule computation the root of the tree range is the node corresponding to the left-hand side of the production. The optional `SymbolRef` may restrict the root of the tree range to a node corresponding to a symbol of the right-hand side of the production.

In a (lower or upper) symbol computation the root of the tree range is the node corresponding to that symbol.

If the optional `ShieldClause` is given it specifies the set of shielded symbols. If an empty `ShieldClause` is given, no symbols are shielded from the tree range. If the `ShieldClause` is omitted then the root symbol of the tree range (as described above) is shielded from the range.

An attribute of a `CLASS` symbol `C.a` in the `RemAttrList` contributes attributes `X.a` to the referred set for all `TREE` symbols `X` to which `C` is inherited.

A CLASS symbol C in the ShieldClause contributes symbols X to the set of shielded symbols for all TREE symbols X to which C is inherited.

A CONSTITUENT(S) in a VOID context simply states a dependency and does not cause a value to be propagated.

For a CONSTITUENTS that is not in VOID context a WithClause specifies how the values of the accessed attribute instances are combined into one value.

The given TypeName specifies the type of the result and of intermediate values.

The CombFctName specifies a function (or macro) that is applied to two values of the given type and yields one value of that type.

The SingleFctName specifies a function (or macro) that is applied to each accessed attribute instance and yields a value of the given type.

The NullFctName specifies a function (or macro) that has no argument and yields an intermediate value. It is called for every node in the tree range that could have referred attribute instances below it according to the tree grammar, but for the particular tree it has none. Hence, the result of this function should be neutral with respect to the combine function.

It is guaranteed that the combine function is applied to intermediate values according to a post-order projection of the accessed tree nodes. It is left open in which associative order that function combines intermediate values.

The referred set of a CONSTITUENTS may be empty if no attributes of the RemAttrList are reachable in the subtree or if CLASS symbols in the RemAttrList are not inherited to any TREE symbol. In that case a VOID CONSTITUENTS is ignored, and a value CONSTITUENTS results in a call of the NullFctName.

Restrictions

A SymbolRef must denote a right-hand side symbol of the production. It must not be specified in symbol computations.

A CONSTITUENTS in a value context must have a WithClause.

For a CONSTITUENT the tree grammar must guarantee that the accessed attribute instance is uniquely determined for every tree.

The RemAttrs must have the same type if the CONSTITUENT(S) is in value context.

9.3 CHAIN

Chains relate computations in left-to-right depth-first order within certain subtrees. A chain may propagate values or just specify dependencies in that order. Only effective computations, that compute a new chain value or a new post-condition need to be specified. They are automatically linked in the described order.

The basic idea is captured by the following diagram representing the way of a chain through the tree context of a rule graphically:

```
RULE: LhsSym ::= RhsSym1 RhsSym2 END;

           |                 ^
           v                 |
           u      LhsSym     d
```

```
    /                  ^
   /                    \
  /        -----         \
 /         |   |          \
v          |   v           \
d  RhsSym1 u   d  RhsSym2   u
|          ^   |            ^
|          |   |            |
------------   --------------
```

The arcs represent the path of the chain through this context, coming in from the upper context of `LhsSym`, going through the two subtrees, and leaving to the upper context. That chain propagation is established automatically if the chain is not used in this context. Usually, some of the three arcs inside the the context may be specified by explicit computations that use and define the chain at a certain symbol occurrence. The `u` and `d` in the graphic stand for usable and definable chain accesses respectively.

Chain accesses are denoted like attribute accesses with a `ChainName` instead of an attribute name.

Syntax

```
ChainSpec     ::= 'CHAIN' ChainNames ':' TypeName

Computation   ::= 'CHAINSTART' Attribute '=' Expression Terminator

Attribute     ::= SymbolRef '.' ChainName
```

Examples

```
CHAIN cnt : int
RULE: Block ::= '{' Decls Stmts '}' COMPUTE
  CHAINSTART Stmts.cnt = 0;
  printf ("Block has %d statements\n", Stmts.cnt);
END;
RULE: Stmt ::= Var '=' Expr ';' COMPUTE
  Stmt.cnt = ADD (Stmt.cnt, 1);
END;

CHAIN codeseq: PTGNode;
SYMBOL Block COMPUTE
  CHAINSTART HEAD.codeseq = PTGNULL;
  SYNT.transl = TAIL.codeseq;
END;
SYMBOL Stmt COMPUTE
  THIS.codeseq = PTGSeq (THIS.codeseq, THIS.transl);
END;
```

A `CHAIN` specification introduces the name and the type of a chain. Any attribute notation using a `ChainName` denotes a chain access.

A chain states a precondition and a postcondition for each symbol node on the chain. The precondition is set by the upper context of the symbol, the postcondition by its lower context. They can be understood as an implicitly introduced pair of attributes, an inherited one for the precondition and a synthesized one for the postcondition.

A computation is allocated on the chain if it depends on the chain and its result contributes to the chain. Such computations are automatically linked in left-to-right depth-first order. A computation is only linked in chain order if it defines the chain and depends directly or indirectly on it. A computation that only accesses the chain without defining it is not necessarily executed in chain order.

A computation that defines a chain without directly or indirectly accessing it breaks the chain, i. e. the execution order of subsequent chain computations is independent of those prior to this computation.

There may be several instances of a chain that have the same name and type. Each instance is identified by a context that contains a `CHAINSTART` computation for that chain. Chain references in subtrees of such a `CHAINSTART` context belong to that instance, unless they belong to a nested instance of `CHAINSTART` context deeper in the tree. Different instances of a chain are not related to each other, regardless of whether they are nested or separate. However, they may be explicitly connected by computations. The structure of the tree grammar must ensure that there is a `CHAINSTART` context above any computation that refers to the chain.

A `CHAINSTART` computation defines the initial value of a chain. The chain is started at the symbol specified as the destination of the `CHAINSTART` computation. It must be the leftmost of the right-hand side symbols which the chain is to be passed through. `HEAD.c` may be used for a chain `c` to denote the leftmost symbol of the right-hand side, in symbol computations as well as in rule computations.

A computation may refer to a chain `c` by one of the following notations: `X.c` in rule computations, `THIS.c`, `SYNT.c`, `INH.c` in symbol computations, `HEAD.c`, and `TAIL.c` in both rule and symbol computations.

The notations `X.c` and `THIS.c` have different meanings depending on their occurrence in a defining position of an attribute computation or in an applied position within an expression:

In rule computations the following holds: If `X` is the left-hand side symbol of the production, then an applied occurrence `X.c` denotes the chain's precondition at `X`; a defining occurrence `X.c` denotes the chain's postcondition at `X`. If `X` is a right-hand side symbol of the production, then a defining occurrence `X.c` denotes the chain's precondition at `X`; an applied occurrence `X.c` denotes the chain's postcondition at `X`.

In symbol contexts only lower computations may access or define a chain. An applied occurrence of `THIS.c` denotes the chain's precondition of that symbol; `INH.c` may be used instead. A defining occurrence of `THIS.c` denotes the chain's postcondition of that symbol; `SYNT.c` may be used instead.

The notation `HEAD.c` can be used to define the chain's precondition of the leftmost subtree. The notation `TAIL.c` can be used to access the chain's postcondition of the rightmost subtree. These notations can be used in symbol computations and in rule computations. If used in a rule computation that rule must have at least one subtree.

If HEAD.c, TAIL.c, or CHAINSTART is used in a symbol computation that is inherited by a rule which has no subtree, they have the same effect as if there was a subtree which passes the chain dependency and the chain value, if any, unchanged.

In the following example a chain c is used in symbol computations. They state that the functions Prefix and Suffix are called on the chain for every Expression context. The Prefix call is applied to the incoming chain and specifies the chain precondition for the leftmost subtree of Expression. The Suffix call is applied to the result of the rightmost subtree and specifies the chain postcondition of this Expression:

```
SYMBOL Expression COMPUTE
    HEAD.c = Prefix (THIS.c);
    THIS.c = Suffix (TAIL.c);
END;
```

Restrictions

Every ChainName must be different from any attribute name and any AttrName.

The tree grammar must guarantee that each access of a chain is in a subtree of a CHAINSTART context for that chain. Furthermore that subtree may not be to the left of the symbol where the CHAINSTART initiates the chain.

None of THIS.c, SYNT.c, INH.c, TAIL.c may be used in upper symbol computations.

HEAD.c must not be used in applied positions.

TAIL.c must not be used in defining positions.

Chains can not be accessed in INCLUDING or CONSTITUENT(S) constructs.

10 Computed Subtrees

In general the tree represents the abstract structure of the input text and is built by scanning and parsing the input. That initial tree may be augmented by subtrees which result from certain computations. This feature can be used for translation into target trees which also contain computations that are executed in the usual way.

The tree construction functions generated by LIGA are used to build such subtrees. They are inserted into the initial tree at certain positions specified in productions.

Syntax

```
Symbols ::= '$' SymbName
```

Examples

```
RULE: Block ::= '{' Decl Stmts '}' $ Target COMPUTE
  Target.GENTREE =
       MkTBlock (COORDREF, Dels.tcode, Stmt.tcode);
END;
RULE TBlock: Target ::= TSeq TSeq COMPUTE ... END;
```

Trees may be the result of computations using LIGA's tree construction functions as described below (see Section 10.1 [Tree Construction Functions], page 32).

Tree values may be propagated between computations using attributes of the predefined type `NODEPTR` (see Chapter 12 [Predefined Entities], page 37).

Tree values are inserted into the tree in contexts where the right-hand side of the production specifies *insertion points* of the form `$ X` where `X` is a nonterminal name.

The insertion is specified by a computation of the attribute `X.GENTREE` where `X` is the insertion point symbol and `GENTREE` is a predefined attribute name for inherited attributes of insertion symbols (see Chapter 12 [Predefined Entities], page 37). The computation must yield a value of type `NODEPTR` that is a legal tree with respect to the tree grammar for `X`: LIGA guarantees that the computations in the inserted tree are not executed before the tree is inserted.

The tree grammar productions for computed trees may be disjoint from or may overlap with the productions for the initial tree.

Computed trees may again have insertion points in their productions.

Restrictions

There must be exactly one insertion computation for each insertion point of a rule context.

There may not be an insertion computation for a symbol that is not an insertion point.

Inserted trees must be legal with respect to the tree grammar. This property is checked at runtime of the evaluator.

No computation that establishes a precondition for a tree insertion may depended on a computation within the inserted tree.

Contexts that may belong to subtrees which are built by computations may not have computations that are marked `BOTTOMUP` or contribute to `BOTTOMUP` computations (see Chapter 5 [Computations], page 11).

10.1 Tree Construction Functions

LIGA generates a set of tree construction functions, one for each rule context. They may be used in computations to build trees which are then inserted at insertion points. Their names and signatures reflect the rule name and the right-hand side of the production.

For a rule

```
RULE pBlock: Block::= '{' Decls Stmts '}' END
```

there is a function

```
NODEPTR MkpBlock (POSITION *c, NODEPTR d1, NODEPTR d2)
```

The function name is the rule name prefixed by `Mk`. Hence, it is recommended not to omit the rule name when its construction function is to be used.

LIGA's tree construction functions are ready to be used in attribute computations. If they are to be applied in user-supplied C-code an include directive

```
#include "treecon.h"
```

has to be used to make the function definitions available.

The first parameter of every function is a pointer to a source coordinate. That argument may be obtained from the coordinate of the context where the function is called. It is used for error reporting, see Chapter 12 [COORDREF], page 37.

The following parameters correspond to the sequence of non-literal symbols of the right-hand side of the production. For each nonterminal in the production there is a parameter of type `NODEPTR`. Its argument must be a pointer to the root node of a suitable subtree, built by node construction functions. For each insertion point in the production there is a parameter of type `NODEPTR`. Its argument should be `NULLNODEPTR`, since that subtree is inserted later by a computation. For each named terminal in the production there is a parameter of the type of the terminal. Its argument is the value that is to be passed to terminal uses in computations.

Functions for chain productions, the right-hand side of which consists of exactly one non-terminal, need not be called explicitly. The nodes for those contexts are inserted implicitly when the upper context is built.

`LISTOF` productions have a specific set of tree construction functions: For a rule like

```
RULE pDecls: Decls LISTOF Var | Proc | END;
```

the functions

```
NODEPTR MkpDecls (POSITION *c, NODEPTR l)
NODEPTR Mk2pDecls (POSITION *c, NODEPTR ll, NODEPTR lr)
```

are provided, where `Mk2pDecls` constructs internal list context nodes and `MkpDecls` builds the root context of the list.

The arguments for each of the parameters `l`, `ll`, and `lr` can be `NULLNODEPTR` representing an empty list, a pointer to a list element node, a node that can be made a list element subtree by implicit insertion of chain contexts, or the result of a `Mk2`-function call representing a sublist.

The `Mk2`-functions concatenate two intermediate list representations into one retaining the order of their elements.

`Mk0`-macros are generated. They take only the `POSITION` but no tree as argument, and return `NULLNODEPTR` representing an empty list. These macros usually need not be used.

The `LISTOF` subtree must be finally built by a call of the root context function.

11 Iterations

The general principle of computations in trees guarantees that the computations specified for each tree node context are executed exactly once. The iteration construct allows to specify cyclic dependencies that may cause certain computations to be iterated until a specified condition holds.

Syntax

```
Computation ::= Iteration Terminator
              | Attribute '=' Iteration Terminator
Iteration   ::= 'UNTIL' Expression
                'ITERATE' Attribute '=' Expression
```

Example:

```
ATTR cnt, incr: int;
RULE: R ::= X COMPUTE
  X.cnt = 1;
  R.done = UNTIL GT (X.cnt, 10) ITERATE X.cnt = X.incr;
END;
RULE: X ::= SomeThing COMPUTE
  X.incr = ORDER (printf ("%d\n", X.cnt), ADD (X.cnt, 1));
END;
```

The execution of an iteration establishes the postcondition specified by the `UNTIL` expression.

The attribute defined in the `ITERATE`-clause is the iteration attribute. The expression of that definition usually depends cyclically on the iteration attribute itself. There has to be another non cyclically dependent computation for the iteration attribute, which is executed initially before the iteration. The iteration attribute may be a `VOID` attribute. All computations that depend on the iteration attribute are executed at least once.

The `ITERATE` computation and all computations that depend on it are reexecuted if the `UNTIL` condition does not hold.

Restrictions

The `UNTIL` condition must yield an `int` value being used as a conditional value.

There must be an initializing non-cyclic definition for the iteration attribute.

The cyclic dependencies involved in the iteration may not include computations of upper contexts of the iteration context.

Some computations that do not lie on the iteration cycle may also be reexecuted on iteration if not specified otherwise. This effect can be avoided by specifying the initial iteration attribute computation to depend on them, or by specifying them to depend on the postcondition of the iteration.

There may be several iterations for the same iteration attribute. The so specified iterations may be arbitrary merged if not otherwise specified. In any case the `UNTIL` conditions hold after completion of the iterations.

Termination of iterations has to be ensured by suitable `UNTIL` conditions and computations.

The iteration attribute may not be a chain attribute.

12 Predefined Entities

The names described in this chapter have a predefined meaning in LIDO specifications.

The following types are predefined in LIDO:

`VOID` Attributes of this type describe a computational state without propagating values between computations. Those attributes do not occur as data objects in the generated evaluator.

`int` The terminal type.

`NODEPTR` Attributes of this type represent computed subtrees.

The predefined value `NULLNODEPTR` of type `NODEPTR` denotes no tree.

The `CLASS` symbol `ROOTCLASS` is predefined. It is implicitly inherited by the root of the tree grammar (see Chapter 4 [Symbol Specifications], page 9).

The following attribute is predefined in LIDO:

`GENTREE` Every insertion point symbol has an attribute `GENTREE` of type `NODEPTR`.

The following functional notations have a specific meaning in LIDO. They are translated into suitable C constructs rather than into function calls:

`IF (a, b, c)`
: denotes a conditional expression. At runtime either `b` or `c` is evaluated, if `a` yields a non-zero or a zero value. For determination of the static evaluation order each of `a, b, c` contribute to the precondition of the computation that contains the `IF` construct. If it occurs in value context `b` and `c` are in value context, too. Then `b` and `c` have to yield values of the same type (not checked by LIGA). Otherwise `b` and `c` are in `VOID` context and may or may not yield a value of some type.

`IF (a, b)` is a conditional computation of `b`, which is executed only if `a` yields a non-zero value. For determination of the static evaluation order both `a` and `b` contribute to the precondition of the computation that contains the `IF` construct. This `IF` construct must occur in `VOID` context. `b` is in VOID context, too.

`ORDER (a, b, ..., x)`
: The arguments are evaluated in the specified order. If it occurs in `VOID` context all arguments are in `VOID` context. If it occurs in value context it yields the result of the last argument `x`. The others are in `VOID` context and may or may not yield a value. For determination of the static evaluation order all arguments of the `ORDER` construct contribute to the precondition of the computation containing it. Any nesting of `ORDER`, `IF`, function calls, and other expressions is allowed, as long as the stated conditions for `VOID` and value contexts hold.

`RuleFct (C_String, arguments ...)`
: A call of this function is substituted by a call of a function whose name is composed of the `C_String` and the name of the rule that has (or inherits) this call. The remaining arguments are taken as arguments of the substituted call. E.g. in a rule named `rBlock` a call `RuleFct ("PTG", a, b)` is substituted by `PTGrBlock (a, b)`.

`RhsFct (C_String, arguments ...)`
: A call of this function is substituted by a call of a function whose name is composed of the `C_String` and and two numbers that indicate how many non-terminals and terminals are on the right-hand side of the rule that has (or inherits) this call. The remaining arguments are taken as arguments of the substituted call. E.g. in a rule `RULE: X ::= Id Y Id Z Id END;`, where `Y, Z` are nonterminals, and `Id` is a terminal, a call `RhsFct ("PTGChoice", a, b)` is substituted by `PTGChoice_2_3 (a, b)`. Usually, `RhsFct` will be used in symbol computations, having arguments that are obtained by the `RHS` construct and by a `TermFct` call.

`TermFct (C_String, arguments ...)`
: A call of this function is substituted by a comma separated sequence of calls of functions whose names are composed of the `C_String` and the name of the non-literal terminals in the rule that has (or inherits) this call. The remaining arguments are taken as arguments of the substituted calls. E.g. the following symbol computation

```
SYMBOL X COMPUTE
  SYNT.Ptg = f (TermFct ("ToPtg", TERM));
END;
RULE: X ::= Y Number Z Ident ';' END;
```

yields the following rule computation

```
RULE: X ::= Y Number Z Ident ';' COMPUTE
  X.Ptg = f (ToPtgNumber (Number), ToPtgIdent (Ident));
END;
```

The order of the calls corresponds to the order of the terminals in the rule. The `TermFct` call must occur on argument position if there is more than one terminal in the rule.

The following names can be used in computations to obtain values that are specific for the context in the abstract tree in which the computation occurs:

`LINE` the source line number of the tree context.

`COL` the source column number of the tree context.

`COORDREF` the address of the source coordinates of the tree context, to be used for example in calls of the message routine of the error module or in calls of tree construction functions.

`RULENAME` a string literal for the rule name of the tree context, to be used for example in symbol computations.

Note: These names are translated by LIGA into specific constructs of the evaluator. Hence, they can not be used with this meaning in macros that are expanded when the evaluator is translated. (That was allowed in previous versions of LIGA.)

The following C macros are defined as described for the generated evaluator, and can be used in the LIDO text:

```
APPLY (f, a, ... )  (*f) (a, ... )   a call of the function f
```

```
                                        with the remaining arguments

CAST(tp,ex)          ( (tp) (ex) )
SELECT(str,fld)      ( (str).fld )
PTRSELECT(str,fld)   ( (str)->fld )
INDEX(arr,indx)      ( (arr)[indx] )

ADD(lop,rop)         ( lop + rop )
SUB(lop,rop)         ( lop - rop )
MUL(lop,rop)         ( lop * rop )
DIV(lop,rop)         ( lop / rop )
MOD(lop,rop)         ( lop % rop )
NEG(op)              ( -op )

NOT(op)              ( !op )
AND(lop,rop)         ( lop && rop )
OR(lop,rop)          ( lop || rop )

BITAND(lop,rop)      ( lop & rop )
BITOR(lop,rop)       ( lop | rop )
BITXOR(lop,rop)      ( lop ^ rop )

GT(lop,rop)          ( lop > rop )
LT(lop,rop)          ( lop < rop )
EQ(lop,rop)          ( lop == rop )
NE(lop,rop)          ( lop != rop )
GE(lop,rop)          ( lop >= rop )
LE(lop,rop)          ( lop <= rop )

VOIDEN(a)            ((void)a)

IDENTICAL(a)         (a)
ZERO()               0
ONE()                1
ARGTOONE(x)          1
```

The last four macros are especially useful in `WITH` clauses of `CONSTITUENTS` constructs.

13 Outdated Constructs

The following constructs are still supported to achieve compatibility with previous LIDO versions. Their use is strongly discouraged.

13.1 Terminals

In previous versions of LIDO terminal symbols could have attributes, at most one synthesized and several inherited. They were associated explicitly by specifications of the form

```
TERM Identifier: Sym: int;
```

Attributes of terminals could be used in attribute notations or `CONSTITUENT(S)` constructs:

```
Identifier.Sym
CONSTITUENT Identifier.Sym
```

If the above constructs occur in a specification a new nonterminal symbol that has the specified attributes is introduced by LIGA, as well as a production that derives to the terminal.

Terminal symbols could be element of a `LISTOF` production:

```
Idents LISTOF Identifier
```

This facility is NOT allowed anymore. It is indicated by an error message, and has to be transformed explicitly.

13.2 Keywords

The key word `DEPENDS_ON` introducing a `DependsClause` is now abbreviated by the token `<-`.

The key word `NONTERM` should be replaced by `SYMBOL`.

```
NONTERM Stmt: code: PTGNode;
NONTERM Stmt COMPUTE ... END;
```

13.3 Pragmas

The pragma notations are substituted by simpler notations:

Calling a function the name of which is composed from a string and the rule name, e.g.

```
LIGAPragma (RuleFct, "PTG", ...)
```

is now achieved by

```
RuleFct ("PTG", ...)
```

See see Chapter 12 [Predefined Entities], page 37.

A pattern for the sequence of right-hand side attributes, e.g

```
LIGAPragma (RhsAttrs, Ptg)
```

is now written

```
RHS.Ptg
```

Hence a combination of both features above, like

```
SYMBOL Reproduce COMPUTE
```

```
      SYNT.Ptg = LIGAPragma (RuleFct, "PTG", LIGAPragma (RhsAttrs, Ptg));
   END;
```

is now written

```
   SYMBOL Reproduce COMPUTE
      SYNT.Ptg = RuleFct ("PTG", RHS.Ptg);
   END:
```

See see Chapter 6 [Attributes], page 15.

Computations were specified to be executed while the input is being read and the tree is being built using a pragma

```
LIGAPragma (BottomUp, printf("early output\n"));
```

Now the keyword BOTTOMUP is added to the computation:

```
printf("early output\n") BOTTOMUP;
```

See see Chapter 5 [Computations], page 11.

14 Syntax

This section contains an overview over the LIDO Syntax. Outdated LIDO constructs described in the previous chapter are left out in this grammar. For further explanations refer to previous chapters.

```
LIDOSpec        ::=  Specification
Specification   ::=  Specification Specification |
                  |  RuleSpec ';' | SymComp ';'
                  |  SymSpec ';'  | TermSpec ';'
                  |  AttrSpec ';' | ChainSpec ';'

RuleSpec        ::= 'RULE' [RuleName] ':' Production Computations 'END'
SymComp         ::= SymbKind SymbName [ Inheritance ] Computations 'END'
TermSpec        ::= 'TERM' SymbNames ':' TypeName
SymSpec         ::= SymbKind SymbNames ':' [ AttrSpecs ]
AttrSpec        ::= 'ATTR' AttrNames ':' TypeName [ AttrClass ]
ChainSpec       ::= 'CHAIN' ChainNames ':' TypeName

AttrSpecs       ::= AttrSpecs ',' AttrSpecs
                  | AttrNames ':' TypeName [ AttrClass ]

SymbKind        ::= 'SYMBOL' | 'CLASS' 'SYMBOL' | 'TREE' 'SYMBOL'
AttrClass       ::= 'SYNT' | 'INH'

Production      ::= SymbName '::=' Symbols
                  | SymbName 'LISTOF' Elements

Symbols         ::= Symbols Symbols |
                  | SymbName | Literal | '$' SymbName

Elements        ::= Elements '|' Elements |
                  |   SymbName

Computations    ::= [ 'COMPUTE' Computation ]

Computation     ::= Computation Computation |
                  | Expression Terminator
                  | Attribute '=' Expression Terminator
                  | 'CHAINSTART' Attribute '=' Expression Terminator
                  | Iteration Terminator
                  | Attribute '=' Iteration Terminator
Terminator      ::= ';'
                  | 'BOTTOMUP' ';'

Iteration       ::= 'UNTIL' Expression
                    'ITERATE' Attribute '=' Expression
```

```
Attribute      ::= SymbolRef '.' AttrName
                 | SymbolRef '.' ChainName
                 | RuleAttr
RuleAttr       ::= '.' AttrName
SymbolRef      ::= SymbName
                 | SymbName '[' Number ']'

Expression     ::= SimpExpr [ DependsClause ]

DependsClause  ::= '<-' DepAttrList
DepAttrList    ::= DepAttr
                 | '(' DepAttrs ')'
DepAttrs       ::= DepAttrs ',' DepAttrs
                 | DepAttr
DepAttr        ::= Attribute | RemoteAccess | RhsAttrs

SimpExpr       ::= C_Name | C_Integer | C_Float | C_Char | C_String
                 | Attribute | RemoteAccess
                 | RhsAttrs
                 | FunctionName '(' [ Arguments ] ')'
                 | SymbolRef
                 | 'TERM' [ '[' Number ']' ]

RhsAttrs       ::= 'RHS' '.' AttrName

Arguments      ::= Arguments ',' Arguments
                 |  Expression

Inheritance    ::= 'INHERITS' SymbNames

RemoteAccess   ::= 'INCLUDING' RemAttrList
                 | [ SymbolRef ] 'CONSTITUENT'
                   RemAttrList [ ShieldClause ]
                 | [ SymbolRef ] 'CONSTITUENTS'
                   RemAttrList [ ShieldClause ] [ WithClause ]

RemAttrList    ::=  RemAttr | '(' RemAttrs ')'
RemAttrs       ::=  RemAttr ',' RemAttrs
RemAttrs       ::=  RemAttr
RemAttr        ::=  SymbName '.' AttrName

ShieldClause   ::= 'SHIELD' SymbNameList
SymbNameList   ::= SymbName | '(' SymbNames ')' | '(' ')'

WithClause     ::= 'WITH' '(' TypeName ',' CombFctName ','
                            SingleFctName ',' NullFctName ')'
```

Literals in expressions (`C_Name`, `C_Integer`, `C_Float`, `C_Char`, `C_String`) are written as in C.

Literals in productions (`Literal`) are written as strings in Pascal.

This syntax distinguishes names for objects of different kinds, e. g. `RuleName`, `SymbName`, `TypeName`. The syntax rules for names are omitted. The following rules are assumed for `XYZNames`:

```
XYZName  ::= Identifier
XYZNames ::= XYZName | XYZNames ',' XYZNames
```

Identifiers are written as in C.

LIDO text may contain bracketed non nested comments in the style of C or Pascal

```
/* This is a comment */
(* This is a comment *)
```

or line comments like

```
% The rest of this line is a comment
```

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

Z