Syntactic Analysis

$Revision: 1.26 $

Compiler Tools Group
Department of Electrical and Computer Engineering
University of Colorado
Boulder, CO, USA
80309-0425

Table of Contents

Syntactic Analysis

The purpose of syntactic analysis is to determine the structure of the input text. This structure consists of a hierarchy of *phrases*, the smallest of which are the *basic symbols* and the largest of which is the *sentence*. The structure can be described by a tree with one node for each phrase. Basic symbols are represented by values stored at the nodes. The root of the tree represents the sentence.

This manual explains how to use a '`.con`' specification to describe the set of all possible phrases that could appear in sentences of a language. It also discusses methods of resolving ambiguity in such descriptions, and how to carry out arbitrary actions during the recognition process itself. The use of '`.perr`' specifications to improve the error recovery of the generated parser is described as well.

Computations over the input can be written with attribute grammar specifications that are based on an abstract syntax. The abstract syntax describes the structure of an abstract syntax tree, much the way the concrete syntax describes the phrase structure of the input. Eli uses a tool, called *Maptool*, that automatically generates the abstract syntax tree based on an analysis of the concrete and abstract syntaxes and user specifications given in files of type '`.map`'. This manual will describe the rules used by Maptool to determine a unique correspondence between the concrete and abstract syntax and the information users can provide in '`.map`' files to assist in the process.

This manual will also discuss how Maptool makes it possible to only partially specify the concrete and abstract syntaxes, as long as together they specify a complete syntax.

Although Maptool simplifies the task of relating the phrase structure of a language to the abstract syntax on which a computation is based, it is cometimes necessary to use a parser that was not generated by Eli to analyze phrase structure. In that case, the relationship between phrase structure and abstract syntax must be embedded in a hand-coded tree construction module. The last section of this manual explains how such a module is implemented, and describes the way in which Eli supports that implementation and manages its integration with the generated tree computations.

1 Context-Free Grammars and Parsing

A *context-free grammar* is a formal system that describes a language by specifying how any legal text can be derived from a distinguished symbol called the *axiom*, or *sentence symbol*. It consists of a set of *productions*, each of which states that a given symbol can be replaced by a given sequence of symbols. To derive a legal text, the grammar is used as data for the following algorithm:

1. Let `text` be a single occurrence of the axiom.
2. If no production states that a symbol currently in `text` can be replaced by some sequence of symbols, then stop.
3. Rewrite `text` by replacing one of its symbols with a sequence according to some production.
4. Go to step (2).

When this algorithm terminates, `text` is a legal text in the language. The *phrase structure* of that text is the hierarchy of sequences used in its derivation.

Given a context-free grammar that satisfies certain conditions, Eli can generate a *parsing routine* to determine the derivation (and hence the phrase structure) of any legal text. This routine will also automatically detect and report any errors in the text, and repair them to produce a correct phrase structure (which may not be that intended by the person who wrote the erroneous text).

1.1 How to describe a context-free grammar

Each production of a context-free grammar consists of a symbol to be replaced and the sequence that replaces it. This can be represented in a type-'`con`' file by giving the symbol to be replaced, followed by a colon, followed by the sequence that replaces it, followed by a period:

```
Assignment: Variable ':=' Expression.
StatementList: .
Statement:
   'if' Expression 'then' Statement
   'else' Statement.
```

The first production asserts that the symbol `Assignment` can be replaced by the sequence consisting of the three symbols `Variable`, `':='`, and `Expression`. Any occurrence of the symbol `StatementList` can be replaced by an empty sequence according to the second production. In the third production, you see that new lines can be used as separators in the description of a production. This notation is often more commonly referred to as *Backus Naur Form*, or just *BNF*.

Symbols that are to be replaced are called *nonterminals*, and are always represented by *identifiers*. (An identifier is a sequence of letters and digits, the first of which is a letter.) Every nonterminal must appear before a colon in at least one production. The axiom is a nonterminal that appears before the colon in exactly one production, and does not appear between the colon and the period in any production. There must be exactly one nonterminal satisfying the conditions for the axiom.

Symbols that cannot be replaced are called *terminals*, and may be represented by either identifiers or *literals*. (A literal is a sequence of characters bounded by apostrophes ('). An apostrophe appearing within a literal is represented by two successive apostrophes.) No terminal may appear before a colon in any production. Terminals represent character strings that are recognized by the lexical analyzer (see Section "Specifications" in *Lexical Analysis*).

Extended BNF allows the use of certain operators on the right hand side of a production. These operators are designed to be short-hands to simplify the grammar description. Rules with extended BNF operators can be translated into rules which use only the strict BNF constructs described so far. While the use of extended BNF constructs is supported for the concrete syntax description in Eli, only strict BNF constructs are allowed in the abstract syntax. When it comes time to deduce the correspondence between the concrete and abstract syntax, Maptool operates on the abstract syntax and a version of the concrete syntax in which all rules containing extended BNF constructs have been translated into equivalent strict BNF rules.

The remainder of this section is devoted to describing how each of the extended BNF constructs are translated to their strict BNF equivalents. Note that most of the EBNF constructs require the introduction of generated symbols for their strict BNF translation. Users are strongly discouraged from using these constructs in instances where attribution is required for those contexts, because changes in the grammar will change the names of the generated symbols used.

The most appropriate use of EBNF constructs that introduce generated symbols is when matching the LIDO `LISTOF` construct, since the `LISTOF` construct makes no assumptions about the phrase structure of the list. For a description of the `LISTOF` construct, see Section "Productions" in *LIDO - Reference Manual*.

When a grammar contains many productions specifying replacement of the same nonterminal, a slash, denoting *alternation* can be used to avoid re-writing the symbol being replaced:

```
Statement:
   Variable ':=' Expression /
   'if' Expression 'then' Statement 'else' Statement /
   'while' Expression 'do' Statement .
```

This alternation specifies three productions. The nonterminal to be replaced is `Statement` in each case. Possible replacement sequences are separated by slashes (/). The strict BNF translation for the above example is:

```
Statement: Variable ':=' Expression .
Statement: 'if' Expression 'then' Statement 'else' Statement .
Statement: 'while' Expression 'do' Statement .
```

Alternation does not introduce any generated symbols and has a very straight-forward translation. As a result, it is the most heavily used of the EBNF constructs.

Square brackets are used to denote that the set of symbols enclosed by the brackets are optional. In the following example, `Constants` and `Variables` are optional, but `Body` is not:

```
Program: [Constants] [Variables] Body .
```

The strict BNF translation of this construct is to generate a rule for each possible permutation of the right hand side. In the case of the above example, the following four rules would result:

```
Program: Body .
Program: Variables Body .
Program: Constants Body .
Program: Constants Variables Body .
```

While the translation doesn't introduce any generated symbols, indiscriminate use of this construct may lead to less readable specifications.

An asterisk (or star) is used to denote zero or more occurrences of the phrase to which it is applied. In the following example, `Program` consists of zero or more occurrences of `Variable` followed by `Body`:

```
Program: Variable* Body .
```

The strict BNF translation of this construct requires the introduction of a generated symbol. Generated symbols begin with the letter `G` and are followed by a unique number. Generated symbols are chosen to not conflict with existing symbols in the concrete syntax. No check is performed to ensure that the generated symbols do not conflict with symbols in the abstract syntax, so users should avoid using symbols of this form in their abstract syntax. The translation for the above example is as follows:

```
Program: G1 Body .
G1: G1 Variable .
G1: .
```

A plus is used to denote one or more occurrences of the phrase to which it is applied. In the following example, `Program` consists of one or more occurrences of `Variable` followed by `Body`:

```
Program: Variable+ Body .
```

The strict BNF translation of this construct is similar to the translation of the asterisk (see ⟨undefined⟩ [Asterisk], page ⟨undefined⟩). The translation for the above example is as follows:

```
Program: G1 Body .
G1: G1 Variable .
G1: Variable .
```

A double slash is used to denote one or more occurrences of a phrase separated by a symbol. In the following example, `Input` is a sequence of one or more `Declaration`'s separated by a comma:

```
Input: Declaration // ',' .
```

The strict BNF translation for the above example is as follows:

```
Input: G1 .
G1: G2 .
G1: G1 ',' G2 .
G2: Declaration .
```

Note that all of the EBNF constructs, except the single slash (for alternation) have higher precedence than the separator construct.

Parentheses are used to group EBNF constructs. This is used primarily to apply other EBNF operators to more than a single symbol. For example:

```
Program: (Definition Use)+ .
```

In this example, we want to apply the Plus operator to the concatenation of a `Definition` and a `Use`. The result denotes one or more occurrences of `Definition`'s followed by `Use`'s. The strict BNF translation for the above is:

```
Program: G2 .
G1: Definition Use .
G2: G1 .
G2: G2 G1 .
```

This is identical to the translation for the Plus operator operating on a single symbol, except that another generated symbol is created to represent the parenthetical phrase.

Note that a common error is to introduce parentheses where they are not needed. This will result in the introduction of unexpected generated symbols.

1.2 Using structure to convey meaning

A production is a construct with two components: the symbol to be replaced and the sequence that replaces it. We defined the meaning of the production in terms of those components, saying that whenever the symbol was found in `text`, it could be replaced by the sequence. This is the general approach that we use in defining the meaning of constructs in any language. For example, we say that an assignment is a statement with two components, a variable and an expression. The meaning of the assignment is to replace the value of the variable with the value resulting from evaluating the expression.

The context-free grammar for a language specifies a "component" relationship. Each production says that the components of the phrase represented by the symbol to be replaced are the elements of the sequence that replaces it. To be useful, the context-free grammar for a language should embody exactly the relationship that we use in defining the meanings of the constructs of that language.

1.2.1 Operator precedence

Consider the following expressions:

```
A + B * C
(A + B) * C
```

In the first expression, the operands of the addition are the variable `A` and the product of the variables `B` and `C`. The reason is that in normal mathematical notation, multiplication takes precedence over addition. Parentheses have been used in the second expression to indicate that the operands of the multiplication are the sum of variables `A` and `B`, and the variable `C`.

The general method for embodying this concept of operator precedence in a context-free grammar for expressions is to associate a distinct nonterminal with each precedence level, and one with operands that do not contain "visible" operators. For our expressions, this requires three nonterminals:

`Sum`	An expression whose operator is `+`

`Term`	An expression whose operator is `*`
`Primary`	An expression not containing “visible” operators

The productions that embody the concept of operator precedence would then be:

```
Sum: Sum '+' Term / Term.
Term: Term '*' Primary / Primary.
Primary: '(' Sum ')' / Identifier.
```

1.2.2 Operator associativity

Consider the following expressions:

```
A - B - C
A ** B ** C
A < B < C
```

Which operator has variable `B` as an operand in each case?

This question can be answered by stating an *association* for each operator: If `-` is “left-associative”, then the first expression is interpreted as though it had been written `(A-B)-C`. Saying that `**` is “right-associative” means that the second expression is interpreted as though it had been written `A**(B**C)`. The language designer may wish to disallow the third expression by saying that `<` is “non-associative”.

Association rules are embodied in a context-free grammar by selecting appropriate nonterminals to describe the operands of an operator. For each operator, two nonterminals must be known: the nonterminal describing expressions that may contain that operator, and the nonterminal describing expressions that do not contain that operator but may be operands of that operator. Usually these nonterminals have been established to describe operator precedence. Here is a typical set of nonterminals used to describe expressions:

`Relation`	An expression whose operator is `<` or `>`
`Sum`	An expression whose operator is `+` or `-`
`Term`	An expression whose operator is `*` or `/`
`Factor`	An expression whose operator is `**`
`Primary`	An expression not containing “visible” operators

The association rules discussed above would therefore be expressed by the following productions (these are *not* the only productions in the grammar):

```
Sum: Sum '-' Term.
Factor: Primary '**' Factor.
Relation: Sum '<' Sum.
```

The first production says that the left operand of `-` can contain other `-` operators, while the right operand cannot (unless the subexpression containing them is surrounded by parentheses). Similarly, the right operand of `**` can contain other `**` operators but the left operand cannot. The third rule says that neither operand of `<` can contain other `<` operators.

1.2.3 Scope rules for declarations

Identifiers are normally given meaning by declarations. The meaning given to an identifier by a particular declaration holds over some portion of the program, called the *scope* of that declaration. A context-free grammar for a language should define a phrase structure that is consistent with the scope rules of that language.

For example, the declaration of a procedure `P` within the body of procedure `Q` gives meaning to the identifier `P`, and its scope might be the body of the procedure `Q`. If `P` has parameters, the scope of their declarations (which are components of the procedure declaration) is the body of procedure `P`.

Now consider the following productions describing a procedure declaration:

```
procedure_declaration: 'procedure' procedure_heading procedure_body.
procedure_heading:
   ProcIdDef formal_parameter_part ';' specification_part.
```

Notice that the phrase structure induced by these productions is inconsistent with the postulated scope rules. The declaration of `P` (`ProcIdDef`) is in the same phrase (`procedure_heading`) as the declarations of the formal parameters. This defect can be remedied by a slight change in the productions:

```
procedure_declaration: 'procedure' ProcIdDef ProcRange.
ProcRange:
   formal_parameter_part ';' specification_part procedure_body.
```

Here the formal parameters and the body have both been made components of a single phrase (`ProcRange`), which defines the scope of the formal parameter declarations. The declaration of `P` lies outside of this phrase, thus allowing its scope to be differentiated from that of the formal parameters.

2 The Relationship Between Phrases and Tree Nodes

`RULE` declarations in files of type '`lido`' describe the structure of the abstract syntax tree over which computations are performed. Eli will create a routine to construct an abstract syntax tree if any tree computations are specified (see Section "Tree Structure" in *LIDO – Computations in Trees*). In order to do this, Eli must be able to deduce a unique correspondence between the concrete and abstract syntaxes, such that for each rule in the concrete syntax it is possible to uniquely determine what abstract syntax tree fragment to build. The tool within Eli that does this is called Maptool. In addition to generating a routine to construct the abstract syntax tree, Maptool will also deduce complete versions of the concrete and abstract syntaxes if only incomplete versions of each are provided by the user. This can only be done if the two syntaxes can together form a complete syntax.

The concrete syntax is provided by the user in files of type '`con`'. Since EBNF constructs are allowed in these files, they are first translated into their strict BNF equivalents before being processed by Maptool (see ⟨undefined⟩ [EBNF], page ⟨undefined⟩). The abstract syntax is extracted from the `RULE` declarations made in files of type '`lido`' (see Section "Rule Specifications" in *LIDO - Reference Manual*).

The remainder of this section will discuss how Maptool deduces the correspondence between the two syntaxes, the use of files of type '`map`' to influence the mapping process, and some usage hints.

2.1 Syntax mapping process

Maptool begins by matching any `LISTOF` constructs that appear in the abstract syntax to any appropriate concrete rules. The next phase examines each concrete rule not matched in the previous phase and tries to find a matching abstract syntax rule. After all matching is complete, unmatched concrete rules are added to the abstract syntax and unmatched abstract rules are added to the concrete syntax. There are a few exceptions to this as are noted in the remainder of this section.

While the most obvious benefit to having Maptool deduce syntax fragments from one syntax and place them in the other is to reduce the amount of typing required, the more important advantage is the support it gives for incremental development. It allows the user to only specify those portions of the syntax with which they are concerned at the moment.

2.1.1 Chain rule definitions

Chain rules have different behavior than other rules during the matching process. Descriptions for three different kinds of chain rules are given here to assist in the explanations given in the remainder of this section:

Chain Rule

A normal chain rule is a rule in which there is exactly one symbol on the right hand side of the rule that is not equivalent to the left hand side. For example, '`X ::= Y`' where X is not equivalent to Y is a chain rule.

Trivial Chain Rule

A trivial chain rule is a chain rule in which the left hand side is equivalent to the right hand side. This typically happens when a symbolic equivalence class is defined that includes both the left hand side symbol and the right hand side symbol (see Section 2.2.1 [Symbol Mapping], page 12).

Literal Chain Rule

A literal chain rule is similar to a trivial chain rule, except that it also has literal symbols on its right hand side. A typical example of this is the rule '`Expr ::= '(' Expr ')'`'.

Based on the above definition for normal chain rules, we define *coercions* between symbols. A symbol `X` can be coerced to a symbol `Y` if there is a chain rule with `X` on the right hand side and `Y` on the left hand side. Coercions are also transitive. If `X` is coercible to `Y` and `Y` is coercible to `Z`, then `X` is also coercible to `Z`. A symbol is also considered coercible to itself.

2.1.2 Matching the `LISTOF` construct

The `LISTOF` construct denotes zero or more occurrences of the elements that appear on its right hand side. It does not dictate the ordering of those right hand side symbols or any delimiters that may be used to separate them. The ordering and delimiters are determined by concrete rules. In simple terms, Maptool begins with the left hand side of the `LISTOF` and recursively matches rules until it finds the right hand side elements. The next paragraph gives a more precise description.

An abstract `LISTOF` construct is matched by starting with the symbol on the left hand side of the LISTOF. All concrete rules with equivalent left hand side symbols are added to the set of matched rules. For each rule added to the set, the right hand side symbols are examined. Of these symbols, literal symbols are ignored. If terminal symbols are encountered that aren't coercible to the symbols appearing on the right hand side of the `LISTOF`, an error is signalled, because the left hand side of the `LISTOF` may not derive symbols other than those that appear on the right hand side. For each nonterminal symbol that isn't coercible to one of the right hand side symbols, the concrete rules that have that symbol on their left hand side are added to the set. The process continues until no more rules can be added to the set.

The intermediate nonterminal symbols that are encountered as new concrete rules are added to the set may not appear on the right hand side of other concrete rules.

If Maptool doesn't find any concrete rules to match a `LISTOF`, it will generate a canonical left recursive representation. For the list:

```
RULE: Program LISTOF Declaration | Statement END;
```

Maptool would generate the following:

```
Program: LST_Program .
LST_Program: LST_Program Declaration .
LST_Program: LST_Program Statement .
LST_Program: .
```

This specifies zero or more occurrences of `Declaration`'s and `Statement`'s.

There is one other important thing to note about the `LISTOF` construct. Attribute computations associated with a `LISTOF` construct can just as easily be written as symbol computations on the symbols of the `LISTOF`. The advantage to using the `LISTOF` construct is that it becomes possible to generate an abstract syntax tree structure which allows for more efficient traversal. In order to construct this special tree structure, it is sometimes necessary to insert an additional chain rule into the concrete syntax at the root of the LISTOF.

This is the case when the rules matching the LISTOF have a recursive occurrence of the left hand side symbol. As an example, the LISTOF construct shown above might be written as follows in the concrete syntax:

```
Program: Program Declaration .
Program: Program Statement .
Program: .
```

As you can see, the root of the LISTOF, Program is used both on the left hand side and right hand side of rules that match the LISTOF construct, meaning that it is used recursively. If the LISTOF construct is provided in a '.lido' file, Maptool must introduce the chain rule 'Program ::= LST_Program' and change other occurrences of Program to LST_Program in order to build the efficient tree structure.

Users should be aware that it is possible for the addition of this chain rule to cause LALR(1) conflicts for the parsability of the concrete syntax that do not appear in the absence of the LISTOF construct. In these cases, users must either rewrite the concrete syntax or avoid the use of the LISTOF construct to avoid the problem.

2.1.3 Matching remaining rules

After all LISTOF constructs have been matched, Maptool attempts to match the remaining concrete rules to rules given in the abstract syntax. A match is determined if the signature of the concrete rule is equivalent to the signature of an abstract rule or coercions (see Section 2.1.1 [Chain Rules], page 9) exist between any symbols which differ in the signatures. Remember that symbolic equivalence classes are applied to concrete rules before this matching takes place, so symbols in the signatures are considered equivalent if they belong to the same equivalence class.

For example, consider the following abstract rules:

```
RULE: Declaration ::= IdDef Type END;
RULE: IdDef ::= Identifier END;
```

The following concrete rule will match the first of the above abstract rules, because of the coercion defined between Identifier and IdDef:

```
Declaration: Identifier Type .
```

The reason for doing this is to distinguish semantically between occurrences of Identifier's in different contexts. In the above example, we have used IdDef to represent a definition of an Identifier. In another place in the grammar, we may want to refer to uses of identifiers instead and use the symbol IdUse. Note that use of chain rules in the manner just described makes it impossible to perform attribute computations during tree construction (see Section 2.4.2 [Constraints], page 15).

It is possible for Maptool to detect multiple possible matching abstract rules for a single concrete rule. Maptool signals an error in this case that must be fixed by changing the grammar to disambiguate the contexts.

2.1.4 Complete generated concrete and abstract syntaxes

After rule matching is complete, unmatched concrete rules, except trivial chain rules and literal chain rules (see Section 2.1.1 [Chain Rules], page 9) are added to the abstract syntax. The reason for this is that trivial chain rules are meaningless in the abstract syntax and literal chain rules are only meaningful if they have attribute computations associated with them, in which case they would already have been specified as part of the abstract syntax.

Sometimes it is desirable to include literal chain rules in the abstract syntax even when the user has not explicitly included them there. A typical situation where this occurs is when generating output conforming to the concrete syntax using the Idem tool (see Section "Textual unparser" in *Abstract Syntax Tree Unparsing*). In this situation the output must contain all literals hence the literal chain rules must be in the abstract syntax so that Idem can generate output patterns for them. To preserve the literal chain rules in the abstract syntax use the `MAPCHAINS` keyword in a specification (see Section 2.2.3 [Mapping Chain Rules], page 14).

Unmatched abstract rules are included in the concrete syntax except in the following instances:

- The rule is a chain rule whose left hand side is not a symbol in the concrete syntax. Adding the rule to the concrete syntax in this case would cause the concrete syntax to be disconnected.
- The rule can only be part of a computed subtree (see Section "Computed Subtrees" in *LIDO - Reference Manual*). This is true if the rule is only reachable from the root symbol if symbols preceded by a *$* are included.

Users can use the `:consyntax` product (see Section "consyntax" in *Products and Parameters*) to view the complete version of the concrete syntax.

The `:abstree` product (see Section "abstree" in *Products and Parameters*) is used to view the complete abstract tree grammar. The `:absyntax` product (see Section "absyntax" in *Products and Parameters*) by contrast only shows the abstract syntax rules which are not part of computed subtrees.

2.2 User mapping specifications

Files of type '`map`' can be provided by the user to influence the way in which certain rules are matched. The syntax of map files can be found with other grammar description towards the end of this document (see Appendix A [Grammars], page 41).

There are currently three ways in which the mapping can be affected. The first are symbolic equivalence classes, which group together symbols that have the same semantic meaning. The second method is to map specific rules. Using this method, concrete rules can be rewritten and/or reordered to match a specific abstract rule. The third method controls the elimination of literal chain rules.

2.2.1 Specifying symbolic equivalence classes

Symbolic equivalence classes are used to group together symbols appearing in the concrete syntax because the semantics of the symbols are equivalent. As a result, a single symbol can be used to represent all of the members of the symbolic equivalence class in the abstract syntax. This representative symbol can either be one of the concrete symbols or a new symbol altogether. Symbolic equivalence classes are specified in files of type '`map`'. A series of symbolic equivalences must be preceded by the keyword `MAPSYM`. An equivalence class is then specified by giving the representative symbol (the symbol to appear in the abstract syntax), followed by *::=* and the list of symbolically equivalent symbols from the concrete syntax terminated by a period. For example, the following specification says that a `Primary`, `Factor`, and `Expr` belong to the same equivalence class:

```
MAPSYM
Expr ::= Primary Factor .
```

Application of symbolic equivalence classes to rules in the concrete syntax is done before the matching process begins. Symbolic equivalence classes can only be created for symbols which are either all nonterminals or all terminals (see Section 1.1 [Notation], page 3). An error message will also be issued if a symbolic equivalence class specification includes abstract syntax symbols on the right hand side, since each abstract syntax symbol represents its own equivalence class.

For backward compatibility with previous releases of Eli, symbolic equivalence classes may also be specified in files of type '`sym`'.

2.2.2 Specifying rule mappings

Rule mapping allows users to rewrite a concrete rule for the purposes of matching it to a specific abstract rule. This is useful in cases where two syntactically different constructs are semantically equivalent. Consider the following expression language with bound identifiers:

```
Computation: LetExpr / WhereExpr .
LetExpr: 'let' Definitions 'in' Expr .
WhereExpr: Expr 'where' Definitions .
```

In this example, `LetExpr` and `WhereExpr` are semantically equivalent constructs, but the ordering of `Definitions` and `Expr` are reversed and they use different literal symbols. We'd like to only specify the semantic computations for the two constructs once. To do this, we can define a symbolic equivalence class for `LetExpr` and `WhereExpr`:

```
MAPSYM
BoundExpr ::= LetExpr WhereExpr .
```

The abstract rule that we can use to represent the two constructs is:

```
RULE: BoundExpr ::= Definitions Expr END;
```

Finally, we must use rule mapping specifications to rewrite the two concrete rules to match the abstract rule:

```
MAPRULE
LetExpr: 'let' Definitions 'in' Expr < $1 $2 > .
WhereExpr: Expr 'where' Definitions < $2 $1 > .
```

The keyword `MAPRULE` precedes a group of rule mapping specifications in the map file. Each rule mapping begins with the concrete rule to be rewritten followed by its rewritten form in angle brackets. In angle brackets, nonliteral symbols appear as positional parameters. A positional parameter is specified with a *$* followed by a number indicating which nonliteral symbol from the concrete rule is to be used. Any literal symbols may also appear between the angle brackets.

An abstract syntax will sometimes have several rules with different names but identical signatures. For example, consider the case where dyadic expressions are represented by abstract rules that do not contain operators:

```
RULE Add: Expression ::= Expression Expression END;
RULE Mul: Expression ::= Expression Expression END;
...
```

In this case, the rule mapping must specify the abstract rule name explicitly in order to disambiguate the pattern match:

```
MAPRULE
Expression: '(' Expression '+' Expression ')' < $1 $2 >: Add .
Expression: '(' Expression '*' Expression ')' < $1 $2 >: Mul .
...
```

Rule names are optional, and may be omitted when the pattern match is unambiguous (as in the bound variable example).

When rule matching proceeds, the concrete rule is seen in its rewritten form. An abstract syntax rule must exist in a LIDO specification that corresponds to the rule mapping specification given. Note that the use of rule mapping makes it impossible to perform attribute computations during tree construction (see Section 2.4.2 [Constraints], page 15).

2.2.3 Preserving literal chain rules

The mapping process normally does not include literal chain rules in the complete abstract syntax unless they appear in the user-supplied abstract syntax (see Section 2.1.4 [Completion], page 12). Sometimes it is desirable to preserve literal chain rules even if the user has not included them in the abstract syntax. To force literal chain rules to be included in the abstract syntax, use the `MAPCHAINS` keyword. The behavior is unchanged if all literal chain rules already appear in the abstract syntax.

Care should be taken when using `MAPCHAINS` in conjunction with attribution. A specification using this keyword may require more attribution than the same specification without it, because it may be necessary to transfer attribute values from the child to the parent or vice versa. The presence of symbol computations for the symbols occurring in the chain rules without the transfer computations just mentioned may result in incorrect attribution without warning.

2.3 Influences of BOTTOMUP specifications on mapping

The generation of the parsing grammar (the input to the parser) may be influenced by `BOTTOMUP` specifications (see Section "Computations" in *LIDO - Reference Manual*) specified in your attribute grammar. This is because the parsing grammar must ensure that the nodes of the abstract syntax tree are constructed in a particular order in the presence of `BOTTOMUP` constraints.

In order to deal with this, Maptool must sometimes inject generated chain rules into the parsing grammar to which tree building actions can be attached. These injected chain rules may cause the parsing grammar to exhibit LALR(1) conflicts. If so, an error will be reported to indicate that the `BOTTOMUP` constraints you have provided cause your grammar to not be parsable.

In trying to resolve such a conflict, it is useful to use the `:pgram` derivation (see Section "pgram" in *Products and Parameters*) to be able to view the parsing grammar that is submitted to the parser generator and contains the injected chain rules. It is also useful to use the `:OrdInfo` derivation to get more information about how `BOTTOMUP` constraints were introduced for specific rules. Approaches to resolving such a problem include eliminating unnecessary `BOTTOMUP` constraints from the attribute grammar or making changes to the concrete syntax that allow the chain rules to be injected without causing LALR(1) conflicts.

2.4 Syntax development hints

This section begins by describing typical patterns of syntax development. This is followed by two more specific examples of how to use the mapping techniques described in the previous sections.

2.4.1 Typical patterns of syntax development

When developing a translator for an existing language, the complete concrete syntax is typically already available. In these cases, it is advantageous to start with the complete concrete syntax and add symbolic equivalences and rule mapping specifications to suit the attribute computations as they are being developed.

On the other hand, when designing a new language, it is easier to start work by specifying attribute computations and adding concrete syntax rules as necessary to resolve issues of precedence, associativity, and other parsing ambiguities.

When errors relating to the syntax appear, it is strongly recommended that the first course of action be to look at the complete generated versions of the syntaxes by using the `:consyntax`, `:absyntax`, and `:abstree` products (see Section "Specifications" in *Products and Parameters*). Very often these problems are simply a result of not correctly anticipating the matching process.

2.4.2 Constraints on grammar mapping

The LIGA attribute grammar system allows users to specify that the first pass of computations are to be performed as the abstract syntax tree is being built. This is specified either by an option given in a LIGA control specification see Section "Order Options" in *LIGA Control Language* or by using an additional keyword in an attribute grammar computation see Section "Computations" in *LIDO - Reference Manual*.

Combining computations with tree construction, however, requires that the tree be constructed in strict left-to-right and bottom-to-top order. In the presence of more advanced grammar mappings, it is not possible to maintain this strict ordering. For this reason, Maptool generates the LIGA control directive:

```
ORDER: TREE COMPLETE ;
```

when it detects that one of these grammar mappings is required. The control directive indicates that the tree should be constructed completely before any computations take place.

The grammar mappings which cause Maptool to emit these directives are the use of chain rules in the abstract syntax that do not exist in the concrete syntax (see Section 2.1.3 [Rule Matching], page 11) and any use of rule mapping (see Section 2.2.2 [Rule Mapping], page 13). Aside from symbolic mappings (see Section 2.2.1 [Symbol Mapping], page 12) and the use of LISTOF constructs, the generated concrete and abstract syntaxes need to be identical in order to allow computations to take place during tree construction.

2.4.3 Abstracting information from literals

Literal terminals often distinguish phrases whose structures are identical except for the particular literal terminal. For example, in a normal arithmetic expression the phrase describing addition and the phrase describing subtraction are identical except for the literal `+` or `-`. Taking nonterminal equivalence classes into account, it may be that *all* phrases representing operations with two operands are identical except for the operator literal.

When phrases have identical structure except for one or more literals, the tree computations carried out at the nodes corresponding to those phrases are often identical except for some parameter that depends on the particular literal. It is then useful to abstract from the distinct literals, obtaining a single phrase with which to associate the computation and a set of phrases with which to associate the parameter evaluation. The key point here is that in many cases the computation will apply to a wide variety of translation problems, whereas the particular set of literals characterizes a single translation problem. By abstracting from the distinct literals, the computation can be reused.

To abstract from a specific literal, simply replace that literal with a nonterminal and add a production that derives the literal from that nonterminal. This added production represents the phrase with which the parameter evaluation would be associated. The computation for the phrase in which the literal was replaced by the nonterminal will now obtain the parameter value from the corresponding child, rather than evaluating it locally.

2.4.4 Mapping expressions for overload resolution

It is quite common for a single operator to have different meanings that depend on the types of its operands. For example, in Pascal the operator `+` might mean integer addition, real addition or set union. There are well-known techniques for deciding what is meant in a particular context, and these techniques depend only on the particular set of operators and operand types. The computations themselves are parameterized by this information (see Section “Selecting an operator at an expression node” in *Type Analysis*).

In order to reuse the tree computation to resolve overloading, abstract from the particular set of literals that represent the operators of the language. Then define equivalence classes in which every nonterminal representing an expression is replaced by `Expr` and every nonterminal representing an operator by `Op`. Finally, associate the appropriate computations with the following rules:

```
Expr: Expr Op Expr.
Expr: Op Expr.
Expr: Identifier.
Expr: Integer.
...
```

(Here `...` indicates rules for other denotations, such as floating-point numbers, `true`, etc., defined in the language.)

As an example of the process, consider a language with integer and Boolean expressions in the style of Pascal.

The literals that represent operators in this language are `+`, `-`, `*`, `/`, `div`, `mod`, `and`, `or` and `not`. Define a new nonterminal for each precedence level of the dyadic operators, one for the unary arithmetic operators, and one for `not`:

```
Addop: '+' / '-' / 'or' .
Mulop: '*' / '/' / 'div' / 'mod' / 'and' .
Sign: '+' / '-' .
Notop: 'not' .
```

These productions abstract from the literals, and embody the information about the precedence and association (all operators are left-associative) needed to determine the phrase structure.

Using these new nonterminals, define the phrase structure of an expression:

```
SimpleExpression: Sign Sum / Sum .
Sum: Sum Addop Term / Term .
Term: Term Mulop Factor / Factor .
Factor: Notop Factor / Primary .
Primary: Integer / Id / 'true' / 'false' / '(' SimpleExpression ')' .
```

(Here `Integer` is a terminal representing arbitrary digit sequences and `Id` is a terminal representing arbitrary identifiers. These symbols will be recognized by the lexical analyzer.)

All of the dyadic operators fall into the same equivalence class, which should be represented by the symbol `Binop`. `Sign` and `Notop` both belong to the `Unop` class, and `SimpleExpression`, `Sum`, `Term`, `Factor`, `Primary` are in the `Expr` class. Here is a type-'`map`' file defining these classes:

```
MAPSYM
Op ::= Addop Mulop Sign Notop .
Expr ::= SimpleExpression Sum Term Factor Primary .
```

3 How to Resolve Parsing Conflicts

Eli attempts to construct a particular kind of parser from the context-free grammar specifying the desired phrase structure. If this attempt fails, Eli reports that failure by describing a set of *conflicts*. In order to understand what these conflicts mean, and to understand how they might be resolved, it is necessary to have a rudimentary idea of how the constructed parser determines the phrase structure of the input text.

A context-free grammar is said to be *ambiguous* if it permits more than one phrase structure to describe a single input text. Most conflicts are the result of such ambiguities, and there are three ways of resolving them:

1. Change the grammar so that only one phrase structure is possible.
2. Provide additional information that causes the parser to select one of the set of phrase structures.
3. Change the form of the input text to avoid the ambiguity.

Note that all of these methods result in the parser recognizing a different language than the one described by the original grammar.

3.1 How the generated parser determines phrase structure

The generated parser is a finite-state machine with a stack of states. This machine examines the input text from left to right, one basic symbol at a time. The current state of the machine is the one at the top of the stack. It defines the set of productions the parser might be recognizing, and its progress in recognizing each. For example, consider the following trivial grammar:

```
Sentence: Expression.
Expression: Primary.
Expression: Expression '+' Primary.
Primary: Integer.
Primary: Id.
```

Initially, the parser might be recognizing the first production, but in order to do so it must recognize either the second or the third. In order to recognize the second production, it must recognize either the fourth or fifth. Finally, because we are considering the initial situation, no progress has been made in recognizing any of these productions. All of the information expressed by this paragraph is represented by the initial state, which is the only element of the stack.

On the basis of the state at the top of the stack, and the basic symbol being examined, the machine decides on one of two moves:

`Shift`	Accept the basic symbol as the corresponding terminal, push a new state onto the stack, and examine the next basic symbol.
`Reduce`	Note that a specific phrase has been recognized, remove a number of states equal to the number of symbols in the sequence of the corresponding production from the stack, push a new state onto the stack, and examine the current basic symbol again.

The parser halts after the reduce move noting that the production containing the axiom has been recognized.

If the first basic symbol of the text were an identifier, a parser for the sample grammar would make a shift move. The new state would be one in which the parser had completely recognized the fifth production. Regardless of the next basic symbol, the parser would then make a reduce move because the fifth production has been recognized. One state would be removed from the stack, and a new state pushed in which the the parser had completely recognized the second production. Again the parser would make a reduce move, removing one state from the stack and pushing a state in which the parser had either completely recognized the first production or recognized the first symbol of the third production.

The parser's next move is determined by the current input symbol. If the text is empty then the parser makes the reduce move noting that the first production has been recognized and halts. If the current symbol of the text is '+' then the parser makes a shift move.

A conflict occurs when the information available (the current state and the basic symbol being examined) does not allow the parser to make a unique decision. If either a shift or a reduce is possible, the conflict is a *shift-reduce conflict*; if more than one phrase could have been recognized, the conflict is a *reduce-reduce conflict*.

The classic example of a shift-reduce conflict is the so-called "dangling else problem":

```
Statement: 'if' Expression 'then' Statement.
Statement: 'if' Expression 'then' Statement 'else' Statement.
```

A parser built from a grammar containing these productions will have at least one state in which it could be recognizing either, and has just completed recognition of the `Statement` following `then`. Suppose that the current basic symbol is `else`; what move should the parser make next?

Clearly it could shift, accepting the `else` and going to a state in which it is recognizing the second production and has just completed recognition of the `else`. It could also reduce, however, recognizing an instance of the first production, popping four elements from the stack and returning to the current state. Thus there is a shift-reduce conflict.

The conflict here is due to an ambiguity in the grammar. Consider the following input text (E1 and E2 are arbitrary expressions, S1 and S2 are statements that do not contain `if`):

```
if E1 then if E2 then S1 else S2
```

There are two possible phrase structures for this text, depending on whether the `else` is assumed to belong with the first or second `if`:

```
if E1 then {if E2 then S1} else S2
if E1 then {if E2 then S1 else S2}
```

In each case the bracketed sub-sequence is a `Statement` according to one of the given rules, and the entire line is a `Statement` according to the other. Both are perfectly legal phrase structures according to the grammar.

The following description of integer denotations in various bases leads to a reduce-reduce conflict:

```
Denotation: Seq / Seq Base.
Seq: Digit / Seq Next.
Next: Digit / Hexit.
```

```
Digit: '0' / '1' / '2' / '3' / '4' / '5' / '6' / '7' / '8' / '9'.
Hexit: 'a' / 'b' / 'c' / 'd' / 'e' / 'f'.
Base: 'b' / 'o' / 'e' / 'x'.
```

When `Base` is omitted, the integer is assumed to be decimal if it contains no `Hexit` and hexadecimal otherwise. An explicit `Base` indicates the base of the digits to be 2, 8, 10 or 16 respectively.

One of the states of the parser constructed from this grammar indicates that either the `Hexit b` or the `Base b` has been recognized. If the input is not empty then the parser has recognized a `Hexit`, but either is possible if the input is empty. Thus the parser cannot determine the production by which to reduce, and the conflict arises.

This conflict indicates an ambiguity in the grammar, exemplified by the input text "1b". Two phrase structures are possible, one yielding the value "1 base 2" and the other yielding the value "1b base 16".

3.2 Conflict resolution by changing the grammar

An ambiguity can sometimes be resolved by changing the grammar. The altered grammar must define exactly the same set of input texts as the grammar that gave rise to the conflict, but it cannot describe more than one phrase structure for any particular text. That phrase structure must reflect the meaning of the text as defined by the language design.

Most languages solve the dangling else problem by associating an `else` with the closest `if`. Here is an unambiguous grammar describing that phrase structure:

```
Statement: matched / unmatched.
matched:
   'if' Expression 'then' matched 'else' matched /
   Others.
unmatched:
   'if' Expression 'then' matched 'else' unmatched /
   'if' Expression 'then' Statement.
```

(`Others` stands for all sequences by which `Statement` could be replaced that contain no `if`.)

If the identifiers `Statement`, `matched` and `unmatched` are placed in an equivalence class, then this grammar yields exactly the same phrase structure as the ambiguous grammar given in the previous section. It is therefore acceptable as far as the remainder of the translation problem is concerned.

3.3 Conflict resolution by ignoring possible structures

When Eli is constructing a parser from a grammar, it computes a set of symbols called the *exact right context* for each production in each state. The exact right context of a production in a state contains all of the symbols that could follow the phrase associated with that production in that state. It is possible for the parser to reduce by a production if the current state indicates that all of the symbols in the production's sequence have been accepted, and the next basic symbol of the input is a member of the exact right context of that production in that state.

By adding a *modification* to the description of a production in a type-‘`con`’ file, the user can specify that a particular symbol be deleted from one or more exact right contexts. The user is, in effect, telling Eli that these symbols cannot follow the phrase associated with that production in that state. In other words, the parser is to ignore phrase structures in which the specified symbol follows the phrase.

A modification is a sequence consisting of either a dollar (`$`) or at-rate-of (`@`) followed by a terminal. It can be placed anywhere within a production, and more than one modification can appear in a single production. If a modification is introduced but no conflict is resolved thereby, an error is reported.

Suppose that a modification `$S` is introduced into a production `P`. The effect of this modification is to delete the symbol `S` from the exact right context of production `P`. This kind of modification can be used to solve the dangling else problem:

```
Statement: 'if' Expression 'then' Statement $'else'.
Statement: 'if' Expression 'then' Statement 'else' Statement.
```

The modification introduced into the first production removes `else` from the exact right context of that production, and therefore makes a reduce move impossible for the parser when it is in the state indicating that it is recognizing one of these productions and has just recognized the first `Statement`. Since the reduce move is impossible, there is no shift-reduce conflict.

Suppose that a modification `@S` is introduced into a production `P`. The effect of this modification is to delete the symbol `S` from the exact right context of any production involved in a reduce-reduce conflict with production `P`. This kind of modification can be used to solve the integer denotation problem:

```
Denotation: Seq / Seq Base.
Seq: Digit / Seq Next.
Next: Digit / Hexit.
Digit: '0' / '1' / '2' / '3' / '4' / '5' / '6' / '7' / '8' / '9'.
Hexit: 'a' / 'b' / 'c' / 'd' / 'e' / 'f'.
Base: 'b' @EOF / 'o' / 'e' @EOF / 'x'.
```

The two modifications introduced into the productions remove `EOF` (the empty input text) from the exact right contexts of the `Hexit` productions that conflict with these two `Base` productions, and therefore make it impossible to reduce the `Hexit` productions when the parser is in the state indicating it has completed recognizing either a `Hexit` or `Base` and the input is empty. A *`b`* or *`e`* at the end of an input text will thus always be interpreted as a marker: “1b” means “1 base 2”, not “1b base 16”. (“1b base 16” would have to be written as “1bx”.)

4 Carrying Out Actions During Parsing

In some cases the translation problem being solved requires that arbitrary actions be carried out as the parser is recognizing the phrase structure of the input, rather than waiting for the complete phrase structure to be available. Most of those cases can be classified either as interactive applications or as complex structuring problems in which contextual information is needed to determine the phrase structure.

An arbitrary action is specified by a fragment of C code. None of the data accessed by this code is provided by Eli; it is the responsibility of the writer of the arbitrary actions to manage any data they manipulate. The simplest approach is to implement all actions as invocations of operators exported by a library or user-defined abstract data type. If these invocations have arguments, they are either constant values characteristic of the particular invocation or references to an entity exported by some (possibly different) abstract data type.

An action is a sequence consisting of an ampersand (`&`) followed by a literal. The content of the literal is the C code fragment to be executed. Actions can be placed anywhere in a production, and will be executed when all of the symbols to the left of the action's position have been recognized. Thus an action placed at the end of the production would be executed when all of the symbols in the sequence have been recognized.

Here is a fragment of a grammar describing a desk calculator; actions are used to compute subexpression values as the expression is parsed:

```
expression:
   term /
   expression '+' term &'ExprPlus();' /
   expression '-' term &'ExprMinus();' .

term:
   primary /
   term '*' primary &'ExprTimes();' /
   term '/' primary &'ExprDiv();' .
```

The C code fragments invoke operations of a module that maintains a stack of integer values.

If an action is placed anywhere other than the end of the production, it may lead to conflicts. Suppose that an action is placed between the first and second symbols of the sequence in a production `P`. Suppose further that there is another production, `Q`, whose sequence begins with the same two symbols but does not contain the same action. If one of the states of the parser could be recognizing either `P` or `Q`, and has recognized the first symbol, it would not be able to decide whether or not to execute the action.

5 Improving Error Recovery in the Generated Parser

In some cases, the same pattern in the input text may represent different tokens in the grammar. Knowing which token the pattern represents may be based on other available information. When the parser determines that it cannot accept the next look-ahead token, the boolean function `Reparatur` is called:

```
int Reparatur (POSITION *coord, int *syncode, int *intrinsic);
```

This allows the user to change the look-ahead token based on other available information. If the function returns `0`, then the token has not been altered and the generated parser continues with its normal error recovery. If the function returns `1`, it is assumed that the passed in attributes of the token have been changed (in particular `syncode`), and the generated parser rechecks the look-ahead token to see if it can accept it.

By default, the Eli system provides file '`dfltrepar.c`' containing a definition of the function `Reparatur` that always returns `0`. To override the default, the user must provide a new definition of the function `Reparatur` in some C file.

In case of erroneous input the generated parser invokes its error recovery. The error recovery works completely automatically and usually behaves satisfactorily, in that it produces a tree that is close to the one that might be expected if there were no syntactic errors. This enables the compiler to go on and detect additional semantic errors.

It is also possible to generate a program that will terminate after parsing if syntactic errors were detected. To generate a program with this property, simply add the following parameter to the request for derivation (see Section "define" in *Products and Parameters Reference*):

```
+define='STOPAFTERBADPARSE'
```

There are a few possibilities to control the error recovery in order to improve its behavior. To understand the control facilities it is necessary to know how the error recovery works in principle.

If an error in the input is detected two methods for error repair are used. The first method tries to "correct" the error by deleting, inserting, or replacing one input symbol. The repair is considered successful, if the next 4 parsing steps don't lead to another error. The use of this method is optional. If the first method is not used or if it failed the second method performs a complete correction without backtracking. It skips input symbols until a so-called *restart point* is reached. The restart point is a symbol where normal parsing can be resumed. Before normal parsing resumes error correction takes place. Input symbols are inserted in order to construct a syntactically correct input and the associated semantic actions are executed. The intention is to pass consistent information to the following compiler phases, which therefore do not have to bother with syntax errors.

The second method for error recovery can be controlled by providing additional information. The intention is to decrease the probability of error avalanches caused by wrong error repair decisions. As a running example, we use an ALGOL-like language defined by the following grammar:

```
block : 'begin' declarations statements 'end' .
declarations : declarations declaration ';' / .
declaration : 'real' 'identifier' /
```

```
                / 'procedure' 'identifier' ';' statement .
statements : statement / statements ';' statement .
statement : 'identifier' / block .
```

Three types of error recovery information can be specified by the user in files of type '`.perr`':

The error recovery has a major drawback when applied to errors in lists, defined, e.g., as

```
statements : statement / statements ';' statement .
```

A missing delimiter ';' cannot be inserted in order to parse the rest of the list. This could lead to an infinite loop in the parser. Therefore errors like

```
begin identifier begin identifier ; ...
```

cannot be repaired by inserting the semicolon ';' but by deleting the two symbols 'begin' and 'identifier'.

The following specification in a '`.perr`' file defines the mentioned terminals as list separators.

```
$SEPA ';' . ',' .
```

A list separator will always be inserted if a restart point can be found immediately behind it. In this case the rest of the list can be parsed without the danger of getting into an infinite loop.

Programming languages have bracketed structures like 'begin' and 'end' which delimit not only the syntactic structure of "block" but also the scope of identifiers. Deleting or inserting such semantically significant parentheses is highly probably to cause avalanches of syntactic and semantic errors. Therefore, the error recovery should not change the structures of a program as far as it concerns scopes of identifiers or similar semantic concepts.

Consider the following erroneous input:

```
begin
  procedure identifier ;
  begin
    real identifier ;
    identifier ;
    real identifier ;
    identifier ;
```

Inserting the terminal 'end' before the second "real declaration" corrects the program syntactically but may lead to a semantic error in the last line, as the scope structure is changed.

The specification

```
$BRACKET 'begin' . 'end' .
```

in a file of type '`.perr`' declares the mentioned terminals to be delimiters of semantically significant regions (semantic delimiters). These terminals are not inserted unless the restart point is end of input or the restart point itself is specified as such a delimiter.

Usually there are a few terminals not suited as restart points. The reason is that is programming languages terminals like 'identifier' or 'number' occur in many different syntactic positions. Consider the error

```
begin real identifier identifier ; real identifier ...
```

There is no safe way to tell whether the second identifier belongs to a statement or to a declaration. If it is used as a restart point, the error is corrected to

```
begin real identifier ; identifier ; real identifier ...
```

This corresponds to a transition from the declaration part into the statement part of the block, a frequent cause for error avalanches. In general, terminals like 'identifier' or 'number' are not feasible as restart points.

The specification

```
$SKIP 'identifier' . 'integer_number' . 'real_number' .
```

in a type '`.perr`' file defines the mentioned terminals as unsafe restart points. Unsafe restart points are skipped in case of an error in order to search for restart points more feasible.

With the above specification the second identifier in the mentioned example will be skipped. Parsing resumes at the following semicolon without carrying out a transition to the statement part.

6 Using Foreign parsers

When Eli is used to generate a parser, Maptool is able to relate the concrete syntax to the abstract syntax and create all of the code necessary to build a tree representing the input text. If a parser is generated by other tools, or written by hand, tree-building code must be created manually. In this section, we assume that a parser for the source language exists, and that Eli is being used to generate code from a LIDO specification of the abstract syntax and desired tree computations.

The interface specification of any parser designed to support tree computation defines a set of function invocations that will occur as parsing of the input text proceeds. If the parser has been generated, these function invocations are included in the grammar as semantic actions (see Chapter 4 [Carrying Out Actions During Parsing], page 23).

The code generated from a LIDO specification includes a set of tree construction functions, one for each rule context (see Section "Tree Construction Functions" in *LIDO - Reference Manual*). These functions must be invoked at appropriate times with appropriate arguments during the course of the parse. In order to use an existing parser, therefore, we must implement a module obeying the interface specification of that parser and correctly invoking the tree construction functions generated from the LIDO specification.

It would be possible to develop the module in isolation and then integrate it with the foreign parser, but a better approach is to use the foreign parser as part of the Eli specification of the complete program. Development can then proceed incrementally using Eli tools like execution monitoring to track down errors and verify correct behavior.

6.1 Building tree nodes

A typical parser interface specifies a data structure to define text fragments, in addition to the semantic actions:

```
typedef struct {  /* Basic symbol */
  int line;       /*   Source line containing the symbol */
  int col;        /*   Column containing the first character */
  int type;       /*   Classification code of the symbol */
  char *text;     /*   Symbol text */
} Token;
                  /* Symbol classification codes */
#define ETXT 1    /*   End of the shource file */
#define LPAR 2    /*   Left parenthesis */
#define RPAR 3    /*   Right parenthesis */
#define PLUS 4    /*   Plus */
#define STAR 5    /*   Asterisk */
#define INTG 6    /*   Integer */
...
```

Each tree construction function generated by Eli from a LIDO specification must be invoked with pointers to its children, and therefore the tree must be built bottom-up. The usual strategy is to store pointers to constructed nodes on a stack until their parent node is built. Eli provides a stack module for this purpose:

```
NODEPTR *_nst;         /* Stack array: _nst[...] are the elements */
```

```
int _nsp;              /* Stack index: _nst[_nsp] is the top element */
void _incrnodestack(); /* Push an empty element onto the stack */
```

Elements of the stack are `_nst[_nsp]`, `_nst[_nsp-1]`, etc. The statement `_nsp-=k;` pops `k` elements off of the stack, and the statement `_incrnodestack();` pushes an empty element onto the stack. To make the stack visible, include the file '`treestack.h`'.

The behavior of the functions called by the parser is determined primarily by the needs of the abstract syntax. We'll consider two LIDO specifications, one for computing the value of an integer expression involving addition and multiplication and the other for carrying out overload resolution in more general expressions.

6.1.1 Tree designed for expression evaluation

Consider the following LIDO specification, which evaluates an integer expression involving addition and multiplication. It assumes each `Integer` terminal is represented by the value of the corresponding integer:

```
ATTR val: int;

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

RULE Add: Expr ::= Expr '+' Expr COMPUTE
  Expr[1].val=ADD(Expr[2].val,Expr[3].val);
END;

RULE Mul: Expr ::= Expr '*' Expr COMPUTE
  Expr[1].val=MUL(Expr[2].val,Expr[3].val);
END;

RULE Num: Expr ::= Integer COMPUTE
  Expr.val=Integer;
END;
```

Eli generates four node construction functions from this specification:

```
NODEPTR MkTop(POSITION *_coord, NODEPTR _d1);
NODEPTR MkAdd(POSITION *_coord, NODEPTR _d1, NODEPTR _d2);
NODEPTR MkMul(POSITION *_coord, NODEPTR _d1, NODEPTR _d2);
NODEPTR MkNum(POSITION *_coord, int _TERM1);
```

To make the node construction functions visible, include the file '`treecon.h`'.

The `_coord` parameter will be discussed in detail in the next section; here we will always supply `NoPosition` as the value of this argument (see Section "Source Text Coordinates and Error Reporting" in *The Eli Library*).

Our module must call `MkNum` whenever the parser recognizes an integer, and we must provide the internal value of that integer as the second argument of that call. The result of the call must be pushed onto the top of the stack.

Suppose that by looking at the code of the parser, or the grammar from which the parser was generated, we determine that when the parser recognizes an integer in the input text it calls the function `int_literal_constant` with the `Token` describing that integer as its argument. We might then implement `int_literal_constant` as follows:

```
void int_literal_constant(Token *t)
{ _incrnodestack();
  _nst[_nsp]=MkNum(NoPosition,atoi(t->text));
}
```

Note that this code does *not* check for an error in the conversion of the string. That might or might not be reasonable, depending upon how careful the parser was in accepting a string as a representation of an integer value.

Further examination of the parser might show that it calls the function `mult_operand` with no arguments when it has recognized an expression involving two operands and an asterisk operator. In this case, the nodes for the two operand expressions are already on the stack. They must be removed and replaced by a `Mul` node:

```
void mult_operand(void)
{ _nst[_nsp-1]=MkMul(NoPosition,_nst[_nsp-1],_nst[_nsp]);
  _nsp--;
}
```

Implementation of the action when the parser recognizes an expression involving two operands and a plus operator is identical except that `MkAdd` is invoked instead of `MkMul`.

If the parser invokes `level_0_expr` with no arguments when it has completed recognition of the input text, the implementation of that function might be:

```
void level_0_expr(void)
{ _nst[_nsp]=MkTop(NoPosition,_nst[_nsp]);
}
```

Suppose that the parser invokes `level_3_expr` with no arguments when it has recognized an expression in parentheses. There is no corresponding rule in the abstract syntax, because parentheses serve only to override operator precedence and do not affect the computation. In that case, the routine does nothing:

```
void level_3_expr(void)
{ }
```

An expression language usually has precedence levels containing several operators. For example, dyadic `+` and `-` operators usually have the same precedence, as do dyadic `*` and `/`. A parser may invoke a single function when it recognizes *any* dyadic expression whose operator is at a specific precedence level. In that case, some indication of the operator must be passed to that function. For example, the parser might call `mult_operand` with a pointer to the operator token. The implementation of `mult_operand` must then use the token type to select the correct node construction function:

```
void mult_operand(Token *o)
{ if (o->type == STAR)
    _nst[_nsp-1]=MkMul(NoPosition,_nst[_nsp-1],_nst[_nsp]);
  else
    _nst[_nsp-1]=MkDiv(NoPosition,_nst[_nsp-1],_nst[_nsp]);
  _nsp--;
}
```

This assumes that the parser will not invoke `mult_operand` unless the operator is either `*` or `/`, and therefore no error checking is required. (If the number of operators at the precedence level were larger, then a switch statement might be preferable to the conditional.)

The code for `mult_operand` also assumes that the division is implemented by a LIDO rule named `Div`:

```
RULE Div: Expr ::= Expr '/' Expr COMPUTE
  Expr[1].val=DIV(Expr[2].val,Expr[3].val);
END;
```

6.1.2 Tree designed for overload resolution

Consider the following LIDO specification, which provides a structure to analyze the result type of an expression involving addition, subtraction, multiplication, and division of integers and floating point numbers (see Section "Operator Overloading" in *Tutorial on Type Analysis*). It assumes that each `Integer` and `Float` terminal is represented by the string definining the corresponding number:

```
RULE Top: Root ::= Expr END;

RULE Dya: Expr ::= Expr BinOp Expr END;

RULE Pls: BinOp ::= '+' END;
RULE Min: BinOp ::= '-' END;
RULE Str: BinOp ::= '*' END;
RULE Sls: BinOp ::= '/' END;

RULE Ntg: Expr ::= Integer END;
RULE Flt: Expr ::= Float   END;
```

Eli generates eight node construction functions from this specification. The first six are:

```
NODEPTR MkTop(POSITION *_coord, NODEPTR _d1);
NODEPTR MkDya(POSITION *_coord, NODEPTR _d1, NODEPTR _d2, NODEPTR _d3);
NODEPTR MkPls(POSITION *_coord);
NODEPTR MkMin(POSITION *_coord);
NODEPTR MkAst(POSITION *_coord);
NODEPTR MkSls(POSITION *_coord);
```

To make the node construction functions visible, include the file '`treenode.h`'.

In this example, we would like to represent the integer and floating-point constants in the tree by the strings that represent them in the source text. There are two possibilities:

1. If the foreign parser stores permanent copies of token strings, then pointers to those strings can be stored in the tree nodes.
2. If the foreign parser points to token strings in the input buffer, then our module must store them permanently for reference by the tree nodes.

In case 1, a specification must be added to the LIDO description of the tree:

```
TERM Integer, Float: CharPtr;
```

`CharPtr` is the LIDO name for the C type `char *`. The definition of `CharPtr` is made available by including file '`strings.h`'.

The `TERM` specification causes the two functions `MxNtg` and `MkFlt` to be defined as follows:

```
NODEPTR MkNtg(POSITION *_coord, CharPtr t);
NODEPTR MkFlt(POSITION *_coord, CharPtr t);
```

Suppose that, as discussed in the last subsection, the parser calls `int_literal_constant` when it recognizes an integer in the source text. That routine could be implemented as:

```
void int_literal_constant(Token *t)
{ _incrnodestack();
  _nst[_nsp]=MkNtg(NoPosition,t->text);
}
```

In case 2, we can make use of Eli's `MakeName` module (see Section "Generating Optional Identifiers" in *Solutions of common problems*). It provides a function to store a string uniquely and return an integer-valued hash table index to that unique representation:

```
int MakeName(char *c);
```

Because the default type of a LIDO terminal is `int`, we can omit the `TERM` specification and the two functions `MxNtg` and `MkFlt` will be defined as follows:

```
NODEPTR MkNtg(POSITION *_coord, int t);
NODEPTR MkFlt(POSITION *_coord, int t);
```

The implementation of `int_literal_constant` would be:

```
void int_literal_constant(Token *t)
{ _incrnodestack();
  _nst[_nsp]=MkNum(NoPosition,MakeName(t->text));
}
```

The `MakeName` module must be instantiated in order to gain access to the `MakeName` function. This is done by adding the following line to a '`.specs`' file:

```
$/Tech/MakeName.gnrc:inst
```

No `+instance` parameter should be supplied, because scanning and parsing are provided by the foreign code. Once the module has been instantiated, the definition of the `MakeName` function is made available to the tree construction module by including file '`MakeName.h`'.

Let's assume that the parser invokes a single function when it recognizes any dyadic expression whose operator is at a specific precedence level, passing the operator token to that function. For example, `+` and `-` might both be operators at precedence level 1:

```
void level_1_operator(Token *o)
{ NODEPTR op;
  if (o->type == PLUS) op=MkPls(NoPosition);
  else                 op=MkMin(NoPosition);
  _nst[_nsp-1]=MkDya(NoPosition,_nst[_nsp-1],op,_nst[_nsp]);
  _nsp--;
}
```

This assumes that the parser will not invoke `level_1_operator` unless the operator is either `+` or `-`, and therefore no error checking is required. (If the number of operators at the precedence level were larger, then a switch statement might be preferable to the conditional.)

If, on the other hand, the parser invokes a function `add_operand` with no arguments when it has recognized an expression involving two operands and an addition operator then `add_operand` can be implemented as:

```
void add_operand(void)
{ _nst[_nsp-1]=
    MkDya(NoPosition,_nst[_nsp-1],MkPls(NoPosition),_nst[_nsp]);
  _nsp--;
}
```

Note that the operator node implied by the `add_operand` call must be explicitly created in this case; it is only implicit in the parse.

6.1.3 Tree nodes for chain rules

Recall that a chain rule has the form '`X ::= Y`', where '`X`' differs from '`Y`' (see Section 2.1.1 [Chain rule definitions], page 9). Such a rule will always result in a tree node with a single child, and if the rule name is '`Ch`' then the constructor function will be:

```
NODEPTR MkCh(POSITION *_coord, NODEPTR _d1);
```

With the exception of the root node of the tree, it is never necessary to explicitly invoke the constructor of a chain rule node. This is actually a very important property of the tree construction module. For example, consider the following fragment of a LIDO specification:

```
RULE SimpleVar: Var ::= VrblIdUse                        END;
RULE SubscrVar: Var ::= Var '[' Exp ']'                  END;
RULE VarExp:    Exp ::= Var                              END;
RULE ArrayExp:  Exp ::= TypeIdUse '[' Exp ']' 'of' Exp END;
RULE Typ: TypeIdUse ::= Symbol                           END;
RULE Var: VrblIdUse ::= Symbol                           END;
RULE Idn:    Symbol ::= Identifier                       END;
```

`Identifier` is a terminal symbol, represented by a unique permanent string (see Section 6.1.2 [Tree designed for overload resolution], page 32).

The problem here is that, given the input sequence '`a[`', a parser would have to look beyond the matching '`]`' in order to decide whether '`a`' was a `VrblIdUse` or a `TypeIdUse`. But because the rules `Typ` and `Var` are chain rules, their constructor functions don't need to be called. That means the parser can construct an `Idn` node for '`a`' and leave it on the stack. If that node is later used as the left child of a `SubscrVar` node, the tree construction module will insert the necessary `Var` and `SimpleVar` nodes. If, on the other hand, the `Idn` node is used as the left child of an `ArrayExp` node then the tree construction module will insert the necessary `Typ` node. There is no need for the parser to look ahead.

6.2 Adding coordinate information

LIDO computations may access the coordinates of the first character of the source text region represented by a node. Usually, these computations are used to attach error reports to appropriate text locations. Many of the modules that implement common computations use this facility for error reporting (for an example, see Section "Verifying typed identifier usage" in *Type Analysis*).

Execution monitoring is provided by Noosa, a separate process that can display the abstract syntax tree and graphically relate it to the source text (see Section "Trees and

Attribute Values" in *Execution Monitoring Reference*). Noosa requires that both the source text coordinates of the first character of a tree context and those of the first character *beyond* that context be supplied to its construction function.

Specific source text coordinates are represented by a `POSITION` (see Section "Source Text Coordinates and Error Reporting" in *The Eli Library*). This data type and the operations upon it are made visible by including the file '`err.h`'. An appropriate `POSITION` value must be created from parser data and a pointer to that data passed to the tree construction function.

6.2.1 Supplying coordinates for computation

LIDO provides three names that can be used in computations to obtain source text coordinates of a tree context (see Section "Predefined Entities" in *LIDO - Reference Manual*):

`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.

If any of these three names appear in the LIDO computation, the source text coordinates of the first character of each tree context must be supplied to its node construction function. That information must be extracted from the parser.

In order to support the use of coordinates in computation, the tree construction function must have access to the location of the first character of its tree context. We have assumed that each token provided by the parser specifies the line and column of the first character of the corresponding input string (see Section 6.1 [Building tree nodes], page 29). This information can be used to build a `POSITION` value:

```
POSITION curpos;

void int_literal_constant(Token *t)
{ LineOf(curpos) = t->line; ColOf(curpos) = t->col;
  _incrnodestack();
  _nst[_nsp]=MkNum(&curpos,atoi(t->text));
}
```

Notice that the address of `curpos`, rather then `curpos` itself, is passed to the node construction function `MkNum`.

Unfortunately, this information isn't sufficient. We must not only pass the coordinates to `MkNum`, we must also save them on the stack in case this node is the left child of another node. At that point, the coordinates of the first character of this token would be the coordinates of the first character of the larger tree context.

The Eli stack module actually provides two parallel stacks, `_nst` for nodes and `_pst` for positions. Thus the complete code for integer literal constants would be:

```
void int_literal_constant(Token *t)
{ LineOf(curpos) = t->line; ColOf(curpos) = t->col;
  _incrnodestack();
```

```
    _pst[_nsp]=curpos;
    _nst[_nsp]=MkNum(&curpos,atoi(t->text));
  }
```

The position value, not a pointer to that value, is saved on the stack. That frees `curpos` to be used in constructing other values.

When a node whose children are all on the stack is constructed, the coordinates are obtained from the leftmost child:

```
void mult_operand(void)
{ _nst[_nsp-1]=MkMul(&_pst[_nsp-1],_nst[_nsp-1],_nst[_nsp]);
  _nsp--;
}
```

Generally speaking, the stack location for the left operand becomes the stack location for the result. Because the coordinates of the result are the coordinates of the left operand, there is no need for an assignment to `_pst`.

6.2.2 Supplying coordinates for Noosa

Noosa requires the coordinates of the first character of a tree context and also the coordinates of the first character beyond the end of that context. The additional coordinates should be supplied, however, *only* if execution monitoring has actually been specified for the particular run. This is because the `POSITION` value will only have the necessary space if monitoring has been specified.

The simplest strategy is to define a routine to compute the appropriate `POSITION` value for a given token:

```
POSITION PositionOf(Token_t *token)
{ POSITION curpos;

  LineOf(curpos) = token->line; ColOf(curpos) = token->col;
#ifdef RIGHTCOORD
  RLineOf(curpos) = LineOf(curpos);
  RColOf(curpos) = ColOf(curpos) + strlen(token->text);
#ifdef MONITOR
  CumColOf(curpos) = ColOf(curpos); RCumColOf(curpos) = RColOf(curpos);
#endif
#endif
  return curpos;
}
```

`RIGHTCOORD` and `MONITOR` are defined by Eli for each C compilation if the user specifies the `+monitor` parameter to the derivation (see Section "monitor" in *Products and Parameters Reference*).

A node for an integer literal constant would then be built by:

```
void int_literal_constant(Token *t)
{ curpos = PositionOf(t);
  _incrnodestack();
  _pst[_nsp]=curpos;
  _nst[_nsp]=MkNum(&curpos,atoi(t->text));
}
```

The construction of a node whose children are on the stack becomes more complex, because the coordinates of the constructed node involve the coordinates of the first character of the leftmost child node and the coordinates of the first character beyond the end of rightmost child node. The tree stack module provides a function, `SpanOf`, to compute the correct coordintes:

```
POSITION SpanOf(POSITION left, POSITION right);
```

Using `SpanOf`, the `mult_operand` routine would be written as:

```
void mult_operand(void)
{ curpos=SpanOf(_pst[_nsp-1],_pst[_nsp]);
   _pst[_nsp-1]=curpos;
   _nst[_nsp-1]=MkMul(&curpos,_nst[_nsp-1],_nst[_nsp]);
  _nsp--;
}
```

6.3 Building LISTOF constructs

There are *three* tree construction functions associated with a LISTOF construct with the rule name '`Ex`' (see Section "Tree Construction Functions" in *LIDO - Reference Manual*):

```
NODEPTR MkEx(POSITION *_coord, NODEPTR _d1);
NODEPTR Mk0Ex(POSITION *_coord);
NODEPTR Mk2Ex(POSITION *_coord, NODEPTR _d1, NODEPTR _d2);
```

Arguments '`_d1`' and '`_d2`' may be:

- the result of `Mk0Ex`, which represents an empty portion of the list (any call to `Mk0Ex` can be replaced by the constant `NULLNODEPTR`)
- the result of `Mk2Ex`, which represents a portion (possibly empty) of the list
- any node that can be made a list element subtree by implicit insertion of chain contexts, which represents a single element of the list

The node representing the complete '`Ex`' construct is the one resulting from a call of `MkEx`.

LISTOF constructs always involve either looping or recursion in a parser. For example, consider a language in which a block consists of an arbitrary non-empty sequence of declarations and statements. The LIDO specification for the abstract syntax might contain the rule:

```
RULE Blk: Block LISTOF Declaration | Statement END;
```

Suppose that the parser calls `declaration_action` after each `Declaration` has been recognized and `statement_action` after each `Statement` has been recognized. Moreover, it calls `block_begin` prior to beginning analysis of the list and `block_end` when the end of the block has been reached:

```
void block_begin(void)
{ _incrnodestack();
  _nst[_nsp]=Mk0Blk(&curpos);
}

void declaration_action(void)
```

```
{ curpos=SpanOf(_pst[_nsp-1],_pst[_nsp]);
  _pst[_nsp-1]=curpos;
  _nst[_nsp-1]=Mk2Blk(&curpos,_nst[_nsp-1],_nst[_nsp]);
  _nsp--;
}

void statement_action(void)
{ declaration_action(void) }

void block_end(void)
{ _nst[_nsp]=MkBlk(&_pst[_nsp],_nst[_nsp]); }
```

6.4 Running a foreign parser under Eli

There are two distinct possibilities for the implementation of a foreign parser:

- The foreign parser exists as a collection of C/C++ source files and/or object files that can be linked with the tree construction and computation modules. (A scanner/parser created by LEX/YACC or FLEX/Bison would have this property.)
- The foreign parser exists as an executable file that expects to load a shared library containing the tree construction and computation modules. (A scanner/parser created by a Java-based tool like ANTLR would have this property.)

6.4.1 The parser is a collection of routines

When the parser is a collection of routines, whether in source or object form, the files containing those routines can be listed in a '`.specs`' file (see Section "Descriptive Mechanisms Known to Eli" in *Guide for New Eli Users*). The name of that file then appears in the overall specification. For example, suppose that all of the components of the foreign parser are listed in file '`parser.specs`' and the tree computations are defined by the file '`treecomp.lido`'. Then the overall specification of the program might be '`prog.specs`' with the content:

```
parser.specs
treecomp.lido
```

(Normally the tree computation would involve a number of different specifications rather than a single '`.lido`' file, so a more realistic example would use '`treecomp.specs`' or '`treecomp.fw`' to specify it.)

Eli normally generates a parser from every specification. When a parser is supplied, this behavior must be suppressed by adding the parameter `+parser=none` to the derivation (see Section "How to Request Product Manufacture" in *Guide for New Eli Users*):

```
prog.specs +parser=none :exe
```

Eli also normally provides the following main program:

```
int main(int argc , char *argv[])
{
#ifdef MONITOR
  _dap_init (argv[0]);
  _dapto_enter ("driver");
#endif

  ParseCommandLine(argc, argv);

#include "INIT.h"

  TREEBUILD();

#ifdef STOPAFTERBADPARSE
  if (ErrorCount[ERROR] == 0)
#endif
  ATTREVAL();

#include "FINL.h"

#ifdef MONITOR
  _dapto_leave ("driver");
#endif
  return (ErrorCount[ERROR] > 0);
}
```

One possible strategy is to write a wrapper procedure named `TREEBUILD` that carries out all of the setup operations needed for the foreign parser and then invokes it. This can often be done by renaming a main program provided with the foreign parser and making a few changes to it.

If it is not feasible to modify the main program of the foreign parser, then production of Eli's main program must be suppressed by adding the parameter `+nomain` to the derivation:

```
prog.specs +nomain +parser=none :exe
```

In this case, however, the interface module must:

1. include the initialization code file '`INIT.h`',
2. invoke `ATTREVAL` after the tree has been built,
3. and include the finalization code file '`FINL.h`'.

If the parser executes a function call '`begin_parse();`' before invoking any other functions of the interface, and a function call '`end_parse();`' when it has completed recognition of the input text, then the implementation of these two functions might be:

```
void begin_parse(void)
{
#ifdef MONITOR
  _dap_init ("");  /* Argument is generally the program name */
#endif

#include "INIT.h"
}

void end_parse(void)
{  _nst[_nsp]=MkRoot(&_pst[_nsp],_nst[_nsp]);
```

```
  ATTREVAL();

#include "FINL.h"
}
```

Replace "Root" by the name of the rule that creates the root node of the tree. If the root node is created by another function, omit the Mk-function call. ATTREVAL assumes that the root node is at the top of the stack; if this pre-condition is not satisfied then the computation will silently do nothing.

6.4.2 The parser is an executable file

When the parser is an executable file that expects to load a shared library, that library must be built from the specifications of the tree construction and computation (see Section "so" in *Products and Parameters Reference*). The library must not contain a parser or a main program:

```
treecomp.specs +nomain +parser=none :so
```

Here we assume that all of the components of the LIDO specification, tree construction interface, and supporting modules are listed in 'treecomp.specs'.

The simplest approach to integrating the foreign parser with the shared library is to copy it to a file with the name that the foreign parser expects. For example, if the parser program expects to load a shared library named 'ParserActions.so', then use the following derivation to make the library available under that name:

```
treecomp.specs +nomain +parser=none :so > libParserActions.so
```

(See your system documentation for the placement and naming of shared library files.)

Appendix A Grammars for the Specification Files

TypeConFile ::= *Production**.

Production ::= *Identifier Delim Alternatives* '`.`'.

Delim ::= '`:`' / '`::=`'.

Alternatives ::=
  Alternatives '`/`' *Alternative* /
  Alternatives '`//`' *Separator* /
  Alternative .

Alternative ::= *Element**.
Separator ::= *Symbol*.

Element ::=
  Symbol /
  Connection /
  Modification /
  '`(`' *Alternatives* '`)`' /
  '`[`' *Alternatives* '`]`' /
  Element '`*`' /
  Element '`+`' .

Connection ::= '`&`' *Symbol*.
Modification ::= '`@`' *Symbol* / '`$`' *Symbol*.

Symbol ::= *Identifier* / *Literal*.

TypeMapFile ::=
  ('`MAPSYM`' *SymbolMapping*`+` / '`MAPRULE`' *RuleMapping*`+` / '`MAPCHAINS`')`+` .

SymbolMapping: *Identifier* '`::=`' *Members* '`.`' .
Members: *Identifier*`+` .

RuleMapping: *Rule Rewrite RuleName* '`.`' .
Rule: *Identifier Delimiter RHS* .
Delimiter: '`:`' / '`::=`' .
RHS: *Element** .
Element: *Identifier* / *Text* .
Rewrite: '`<`' *RewriteRHS* '`>`' .
RewriteRHS: (*Position* / *Text*)`+` .
Position: '`$`' *Integer* .

RuleName: / ‘`:`’ *Identifier* .

TypePerrFile ::= *ErrorSpecs*.

ErrorSpecs ::= *ErrorSpecs SeparatorSpecs* /
ErrorSpecs BracketSpecs /
ErrorSpecs SkipSpecs / .

SeparatorSpecs ::= ‘`$SEPA`’ *Symbols* .

BracketSpecs ::= ‘`$BRACKET`’ *Symbols* .

SkipSpecs ::= ‘`$SKIP`’ *Symbols* .

Symbols ::= *Symbols Symbol* .

Symbol ::= *Identifier* / *Literal* .

Index