The TXL Programming Language Syntax and Informal Semantics Version 7

James R. Cordy

Ian H. Carmichael

External Technical Report 93-355
June 1993

Software Technology Laboratory
Department of Computing and Information Science
Queen's University
Kingston, Canada K7L 3N6

Telephone: (613) 545 6054 Email: txl@qucis.queensu.ca

Abstract

This report describes TXL, a programming language specifically designed to support transformational programming. The basic paradigm of TXL involves transforming input to output using a set of transformation rules that describe by example how different parts of the input are to be changed into output. Each TXL program defines its own context free grammar according to which the input is to be broken into parts, and rules are constrained to preserve gramm atical structure in order to guarantee a well-formed result.

TXL has been shown to be well suited to a wide class of computational problems. It has been used for rapid prototyping of new language parsers, semantic analyzers, translators, transliterators and interpreters; new and domain-directed features and dialects in existing languages; software code analysis and design recovery; software restructuring and remodularization; metaprogramming and retroactive software reuse; source-level optimization and parallelization; inter-paradigm program transformation; logical formula simplification and interpretation; program instrumentation and measurement; program normalization and structural comparison.

Table of Contents

1. Introduction

This paper describes the syntax and informal semantics of TXL, a programming language designed to support transformational programming. The basic paradigm of TXL involves transforming input to output using a set of transformation rules that describe by example how different parts of the input are to be changed into output. Each TXL program defines its own context free grammar according to which the input is to be broken into parts, and rules are constrained to preserve grammatical structure in order to guarantee a well-formed result.

2. Overview

Many programming problems can be thought as transforming a single input string into a single output string. Sorting a list of numbers, processing data to generate statistics, formatting text, or even a compiling a program to machine code can be thought of in this way. This is the basic model we use with TXL.

Every TXL program operates in three phases. The first of these is the parsing phase. The parser takes the entire input, tokenizes it, and then parses it according to the TXL program's grammar definitions to produce a parse tree. The second phase in a TXL program is the phase that does all of the "work". It takes the parse tree of the input, and transforms it into a new tree that corresponds to the desired output. The final phase simply unparses the tree produced by the transformer, producing an output string.

3. The Parsing Phase

The parsing phase is responsible for parsing the input string according to the given grammar. The grammar is specified in a notation similar to Backus-Nauer Form (BNF). The parser itself is a top-down, fully backtracking parser that can handle any context-free grammar[1] .

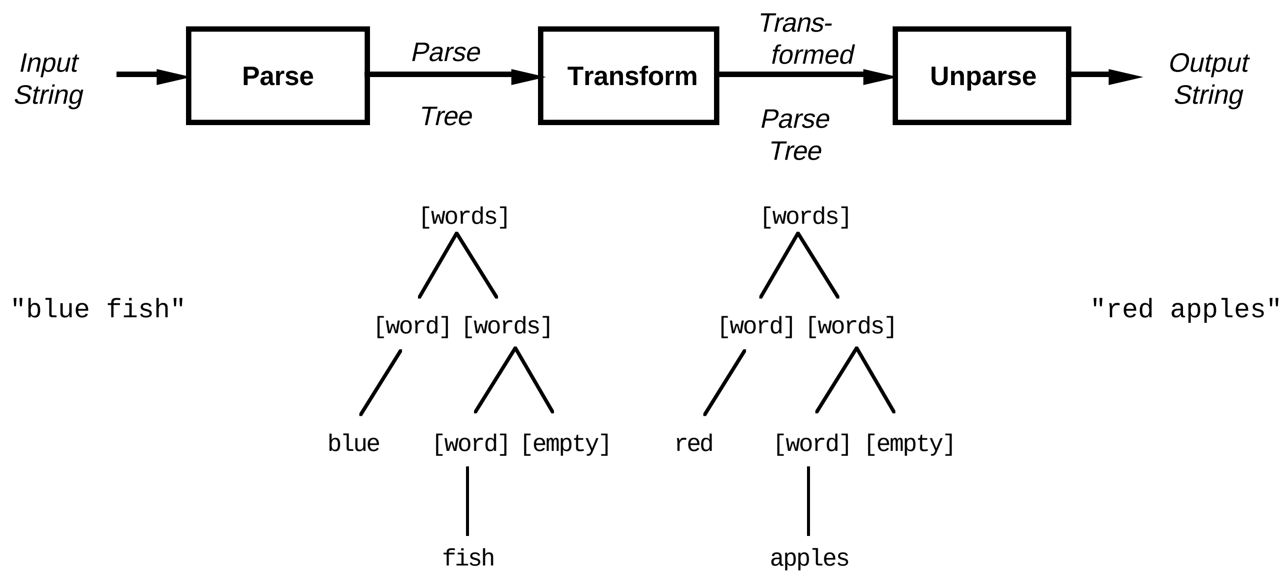

Figure 1. The Three Phases of TXL

[1] Left-recursion is handled by the simple but inefficient solution of bounding the recursive depth of the parser. Grammars that liberally use left recursion can, particularly for invalid input, occasionally require a long parsing time.

3.1 Grammar Definition

The basic unit in our grammars is a *define* statement. Each define statement gives all of the alternative forms for one nonterminal type, corresponding roughly to the set of productions for a single nonterminal in a BNF grammar. Each alternative form is specified as a sequence of *terminal symbols* (items that form part of the required syntax of the grammatical, such as brackets, special characters, keywords, etc.) and *nonterminals* (other grammatical forms from which the new one is built). The vertical bar, '|', is used to separate alternate forms.

For example, the TXL nonterminal definition in Figure 2 specifies that an item of type *expression* consists of either a number, or something that is already an *expression* followed by a plus sign followed by a number, or an *expression* followed by a minus sign followed by a number. For example, 2, 12+4 and 2+17–11+3 are all *expressions* according to the definition.

Nonterminals appearing in the body of a define statement must be enclosed in square brackets [ ]. Symbols not enclosed in brackets are terminals, representing themselves. Some symbols, such as the square brackets themselves, the vertical bar | and keywords of TXL must be quoted (preceded with a single quote) if they are intended to be terminals in a nonterminal definition. For example, the square brackets used for array subscripting in the Pascal language would have to appear as '[and '] respectively in a TXL grammar for the syntax of the programming language Pascal. Any terminal symbol may be quoted if desired; it is considered good TXL style to quote *all* terminal symbols.

By convention, the type *program* is the goal symbol (the nonterminal type as which the entire input to the program must be parsed) for every TXL program and must be defined.

```
define expression
        [number]
    |   [expression] + [number]
    |   [expression] - [number]
end define
```

Figure 2. An example TXL type definition corresponding to the BNF production

```
expression ::= number
             | expression + number
             | expression - number
```

'define' and 'end define' are required TXL punctuation; 'expression' following the first 'define' is the name of the type being defined.

3.2 Predefined Nonterminals

Certain predefined nonterminal types match common grammatical classes. These classes are organized into a hierarchy summarized by the table below. Indented classes are included in the exdented class above, for example, every item of type `[upperlowerid]` is also of type `[id]`, every `[floatnumber]` is also a `[number]`, and so on.

Type	Matches
`[id]`	any identifier, e.g., ABC, abc, aBc, AbC, a_bc, Ab_C_
`    [upperlowerid]`	any id beginning with upper case, e.g., AbCdE, ABCDE
`        [upperid]`	any upper case identifier, e.g., ABCDE
`    [lowerupperid]`	any id beginning with lower case, e.g., aBcDe, abcde
`        [lowerid]`	any lower case identifier, e.g., abcde

`[number]`	any unsigned real or integer, e.g., 123, 12.34, 123.45e22
`    [floatnumber]`	any unsigned real with exponent, e.g., 12.3e22, 12345E22
`    [decimalnumber]`	any unsigned decimal constant, e.g., 123.45
`    [integernumber]`	any unsigned integer constant, e.g.,12345
`[stringlit]`	any double quoted string, e.g. "Hi there"
`[charlit]`	any single quoted string, e.g. 'Hi there'
`[comment]`	any comment (see "Comments Statement" below)

The special nonterminal type `[empty]` always matches, and never accepts any input.

In order to allow the greatest flexibility, the `[id]` and other identifier forms do not allow identifiers to begin with an underscore (_). To allow identifiers beginning with an underscore, make a new nonterminal definition of the form:

```
define uid
         [opt '_] [id]
end define
```

and then use `[uid]` in place of `[id]`.

Similarly, the `[number]` forms do not allow a number to begin or end with a decimal point (.). For types that allow numeric constants to begin or end with a decimal point, an analogous definition to the one for `[uid]` above should be used.

3.3 Nonterminal Modifiers

Any nonterminal type name enclosed in square brackets may be modified by a nonterminal modifier. The three possible modifiers are *opt*, *repeat* and *list*. If the type name is preceded by 'opt', (e.g. `[opt elseClause]`), then the item is optional.

If the nonterminal is preceded by the word 'repeat', (e.g. `[repeat id]`), then zero or more repetitions of the nonterminal are matched. A repeat will always match something since, if nothing else, it will match zero repetitions (i.e., an empty string). If at least one item is required, the modifier '+' can be placed after the repeated type name (e.g. `[repeat statement+]`).

If the nonterminal is preceded by the word 'list', (e.g. `[list formalParameter]`), then a possibly empty, comma-separated list of the nonterminal is matched. As for repeats, if at least one item is required in the list then the modifier '+' can be placed after the type name. For example, the syntax of the formal parameters of a Pascal procedure might have a nonterminal definition like this one:

```
define formalProcedureParameters
         ( [list formalParameter+] )
    |    [empty]
end define
```

If the item to be modified is an explicit identifier or terminal symbol, then it must be quoted with a leading single quote. For example, `[opt ';]` denotes an optional semicolon, whereas `[opt ;]` is illegal.

Each modified nonterminal is translated by TXL into a predetermined set of extra nonterminal definitions. Under normal circumstances the user need not be concerned with these intermediate nonterminals and can treat them as invisible. As it is sometimes convenient to be able to use the intermediate nonterminals in sophisticated transformations, they are documented in detail in Appendix B.

3.4 The Keys Statement

It is possible to specify particular identifiers to be treated as keywords. Keywords differ from other identifiers only in that they are not matched by the built-in nonterminal `[id]` and its variants. Input keywords are only matched by explicit terminal occurrences of the keyword in a nonterminal definition and by the special predefined nonterminal `[key]`[2] .

Keywords are defined using the *keys* statement. A keys statement simply lists the identifiers to be treated as keywords. A total of up to 400 keywords may be given. For example:

```
keys
    program procedure function repeat until for while do 'end
end keys
```

Although any number of keys statements may appear anywhere in the TXL program, they normally appear at the beginning of a section and apply only to the definitions and rules following the statement, and of course to the entire input string. Keywords of TXL (e.g. `'end'`) which are to be keywords of the program's grammar as well must be quoted every time they occur in the TXL program (including in the keys statement, as shown above).

3.5 The Compounds Statement

The TXL input scanner normally treats each special (punctuation) character in the input as a separate terminal symbol. For example, it would treat '`:=`' as a sequence of the two symbols ':' and '='. Normally this does not matter in the course of a transformation task, but it is annoying in the output of TXL to see spaces between the characters, for example, '`x : = y ;`'. For this reason, it is possible to specify which sequences of special characters should be compounded together as single terminal symbols (also known as 'compound tokens').

Compounds are defined using the *compounds* statement. A compounds statement simply lists the sequences of characters to be treated as compound tokens. Each compound token may be up to four characters in length, and a total of up to 40 compound tokens may be given. For example:

```
compounds
    :=  <=  >=  ->  <->
end compounds
```

Although any number of compounds statements may appear anywhere in the TXL program, they normally appear at the beginning of a section and apply only to the definitions and rules following the statement, and of course to the entire input string. If the TXL comment character '%' is part of a compound token, then that token must be quoted everywhere it appears in the program (e.g. `'%=`), including the compounds section.

3.6 The Comments Statement

By default the TXL input scanner expects that the input has the same commenting conventions as TXL, that is, the TeX convention of '%' to end of line. Most input languages, however, have their own commenting conventions that differ from TXL's. The *comments* statement is used to describe the commenting conventions of the particular input language. The comment statement lists the comment brackets of the input language, one pair per line. For example, the C++ language conventions, which allow both '`//`' to end of line and C-style '`/**/`'comments, would be described as:

```
comments
    //
    /* */
end comments
```

[2] The predefined nonterminal [key] is designed primarily for use in transforming TXL itself and is not normally of interest to regular TXL programmers.

If only one comment symbol appears on a line, for example the '//' above, then it is taken to mean that comments beginning with that symbol end at end of line. If two comment symbols appear on a line, for example '/* */' above, then they are taken to be corresponding starting and ending comment brackets. Each comment symbol may be up to four characters long, and up to four different commenting conventions may be specified, specifying that several different kinds of comments are to be accepted in the input. Comment symbols consisting of two or more characters need not appear in a *compounds* statement.

TXL normally ignores all comments when parsing the input language, unless the '-comment' flag is given as a run option. When the '-comment' flag is used as a TXL run option, comments in the input are not ignored, but rather are treated as input tokens of the built-in type `[comment]` which are to be parsed as part of the input to the program. The main use of this feature is the preservation of comments when implementing pretty-printers or other formatters in TXL. Care must be taken to insure that the grammar allows comments to appear in all the expected places - inputs with comments unexpected by the grammar are treated as syntax errors in the input when the '-comment' run option is given.

4. The Transformation Phase

Once the input to a TXL program has been parsed into a parse tree according to the given grammar, the next phase is responsible for transforming the input parse tree into a parse tree for the desired output. The transformation is specified as a set of transformation *functions* and *rules* to be applied to parse trees. We will begin by describing basic transformation functions and rules, followed by the more sophisticated features and special classes of function and rules.

4.1 Transformation Functions

Each TXL function must have, at least, a *pattern* and a *replacement*. A function looks like this:

```
function name
    replace [type]
        pattern
    by
        replacement
end function
```

where *name* is an identifier, *type* is the type of parse tree that the function transforms, *pattern* is a pattern which the functions's argument tree must match in order for the function to transform it, and *replacement* is the result of the function when the tree matches.

The semantics of function application are: if the argument tree matches the pattern, then the result is the replacement, otherwise the result is the (unchanged) argument tree. For example, if the following function is applied to a parse tree consisting solely of the number 2, then the result is a tree containing the number 42, otherwise the result is the original tree.

```
function TwoToFortyTwo
    replace [number]
        2
    by
        42
end function
```

TXL functions are always homomorphic - that is, they always return a tree of the same type as their argument. This guarantees that transformation of an input always results in a well-formed output according to the grammar defined in the TXL program.

A TXL function is applied to its argument in postfix form, using the function name enclosed in square brackets. For example, the function application `f(x)` is written `x[f]` in TXL. When a function is applied to an argument tree, we call this tree the *scope of application* or simply the *scope* of the function, and we speak of the function *replacing* its scope with its result.

For example, if *X* is the name of a tree of type `[number]` whose value is the number 2, then the function application:

```
X [TwoToFortyTwo]
```

will replace the reference to `X` with the tree `[number]` 42.

Several functions may be applied to a scope in succession, for example:

```
X [f][g][h]
```

The interpretation is that *f* is applied first, then *g* to the result, then *h* to the result of that. In more conventional functional notation the applications would be written *h (g (f (X)).*

"Searching" functions search their scope of application for the first match (leftmost shallowest in the tree) to their pattern, and replace (only) the matching subtree with the replacement, leaving the rest of the scope tree untouched. A searching function is denoted by a * following the keyword *replace*.

For example, if we change the function *TwoToFortyTwo* to a searching function:

```
function FirstTwoToFortyTwo
    replace * [number]
        2
    by
        42
end function
```

The resulting function can be used to replace the first occurrence of the number 2 in any tree with the number 42. For example, if *X* is the name of a tree of type `[repeat number]` whose value is the sequence of numbers `1 3 5 7 9 2 4 6 8 2 9 4`, then the function application:

```
X [FirstTwoToFortyTwo]
```

will result in the `[repeat number]` tree `1 3 5 7 9 42 4 6 8 2 9 4`.

Technical details of the search are described in the section on transformation rules below.

4.2 Transformation Rules

A TXL *rule* has the same basic syntax as a function, except that the keyword **function** is replaced with **rule**.

```
rule name
    replace [type]
        pattern
    by
        replacement
end rule
```

The difference is that a rule searches the scope tree it is applied to for matches to its pattern (like a searching function), and replaces every such match (rather than just the first one). For example, if we change the function *TwoToFortyTwo* to a rule:

```
rule EveryTwoToFortyTwo
    replace [number]
        2
    by
        42
end rule
```

and if Y is the name of a tree of type `[repeat number]` containing the number values :

```
27 33 2 5 78 2 89 2
```

then the rule ap plication

```
X [EveryTwoToFortyTwo]
```

yields a `[repeat number]` result tree containing the values :

```
27 33 42 5 78 42 89 42
```

That is, every subtree of type `[number]` with value 2 in the scope has been replaced by a subtree with value 42 in the result.

Technically, a rule searches its scope of application (the tree to which it is applied) for nodes that are of the type of the rule. The search is always a preorder search, examining first each parent node in the tree and then each of its children, from left to right, recursively.

Each time a node of the rule's type is found, the tree rooted at that node is compared to the pattern to see if it matches. If no nodes can be found whose subtrees match the pattern then the rule terminates without changing the scope tree.

If a pattern match is found at a node, then the rule builds a replacement tree, and substitutes this for the node whose subtree was matched, yielding a new scope tree in which the replacement has been made. This new scope is then searched again, the first match in it replaced, and so on, until no more matches can be found.

Because the rule automatically searches the entire new scope tree after each subtree replacement, the replacement can itself create the next pattern match. For example, the rule:

```
rule AddUpNumbers
    replace [repeat number]
        N1 [number] N2 [number] MoreNs [repeat number]
    by
        N1 [+ N2] MoreNs
end rule
```

when applied to the `[repeat number]` input tree 5 7 6 1 3, will first match 5 to *N1*, 7 to *N2* and 12 1 3 to *MoreNs*, yielding the new scope 12 6 1 3. It then searches this new scope, matching 12 to *N1*, 6 to *N2* and 1 3 to *MoreNs*, yielding 18 1 3, and then 18 to *N1*, 1 to *N2* and 3 to *MoreNs*, yielding 19 3, and finally 19 to *N1*, 3 to *N2* and `[empty]` to *MoreNs*, yielding a final `[repeat number]` result tree containing only the number 22.

In the remainder of this paper the term 'rule' to refers to both rules and functions except as explicitly noted.

4.3 The Main Rule

Every TXL program must have one main rule named *mainRule*. Execution of a TXL program consists of applying the main rule to the entire input parse tree. If other rules are to be applied, then the main rule or function must explicitly invoke them.

For example, if rules *R1*, *R2* and *R3* are all to be applied to the entire input, then the main function would look like this:

```
function mainRule
    replace [program]
        P [program]         % i.e., the entire input parse tree
    by
        P [R1][R2][R3]      % apply R1, then R2, then R3 to it
end function
```

4.4 Parameters

Rules may be parameterized by one or more tree parameters. The syntax of a parameterized rule is:

```
rule name parameter1 [type1] parameter2 [type2] ...
    replace [type]
        pattern
    by
        replacement
end rule
```

where each *parameter$_i$* is an identifier, and each *type$_i$* is the nonterminal type of the parameter. The parameter names may be used anywhere inside the rule to refer to the corresponding trees passed as actual parameters to each application of the rule.

Parameterized rule applications must specify the names of the actual trees which are to be passed to the parameters. The type of an actual parameter tree must be the same as the the type of the corresponding formal parameter. For example, the following parameterized rule is like the old *TwoToFortyTwo* example except that it replaces all occurrences of the `[number]` 2 in its scope of application with the specified number *N*.

```
rule TwoToN  N [number]
    replace [number]
        2
    by
        N
end rule
```

When this rule is applied, an actual parameter tree corresponding to *N* must be given in the application. For example, if *Five* is the name of a tree of type `[number]` containing the value 5, and *X* is any tree containing `[number]` subtrees with value 2, then the rule application

```
X [TwoToN Five]
```

will replace every `[number]` 2 in *X* with 5. (Note: This application could also be written as `X [TwoToN 5]` since the terminal symbol 5 always represents the `[number]` tree of value 5.)

4.5 Variables

Parameters are one kind of TXL *variable*. Variables in TXL are names that are bound to (sub-) trees of a particular type which is explicitly given when the variable is introduced (for example, in a rule's formal parameter list, e.g. `N [number]` in the example above). New variables may be introduced either as formal parameters, as part of a pattern (see "Patterns" below), or using an explicit constructor (see "Constructors"). All TXL variables are local to the rule in which they are introduced.

Once introduced, a variable is used simply by name (e.g. `N`). In a pattern, the use of a bound variable means that we intend that part of the pattern to match the current value of the variable exactly (see "Patterns"). In a replacement, the use of a variable indicates that the tree bound to the variable is to be copied into the replacement tree.

The anonymous variable '_' represents a new TXL variable whose name will not be used. Anonymous variables play the role of placeholders in a pattern - they are always explicitly typed (i.e., are newly introduced variables) and cannot be referenced. Any number of anonymous variables may appear in a pattern, with the interpretation that each is a unique new variable.

4.6 Patterns

The pattern for a rule is defined using a sequence of terminal symbols and variables. The first occurrence of each new variable in the pattern is explicitly typed with a type following it (e.g., `X [expression]`). Subsequent uses of variables and formal parameters previously introduced in the rule must not be typed again.

The pattern defines the particular type and shape of the (sub-)tree that the rule is to match. TXL builds a pattern parse tree by (partially) parsing the pattern itself. The type of the pattern tree is the type of the rule. When a parameter or a subsequent use of a variable occurs in a pattern then the current subtree bound to that variable is substituted into the pattern in place of the variable before matching. For example, in the rule:

```
rule foo  T [term]
    replace [expression]
        T + T
    by
        ...
end rule
```

If *foo* is given a parameter which is a parse tree for the term '5' then the pattern tree will be the expression '5 + 5'. If *foo* is passed a tree for the term '3 * 4' then the pattern tree for that rule application will be the expression '3 * 4 + 3 * 4'.

When a pattern is compared to a tree, it matches only if every intermediate nonterminal node and every terminal symbol in the tree and the pattern match exactly. The first (explicitly typed) occurence of each new TXL variable introduced in the pattern, however, matches any subtree of the variable's type. The variable is bound to the subtree it matches. For example, if we apply the rule:

```
rule foo
    replace [expression]
        T1 [term] + T2 [term]
    by
        ...
end rule
```

to the parse tree corresponding to the expression '5*6+9' we get a match at the root of the tree. The variable T1 will be bound to the subtree corresponding to the term '5*6', and the variable T2 will be bound to the subtree corresponding to the term '9'.

A pattern may contain references to variables that are introduced earlier in the same pattern. For example, the pattern in the following rule looks for expressions of the form 'N+N' where N is any number, and replaces them by 'N*2'. Notice that the first occurrence of N in the pattern is explicitly typed, indicating that it is introducing a new variable, while the second reference to N is not typed, indicating that it is a reference to an already bound variable.

```
rule collapse
    replace [expression]
        N [number] + N
    by
        N * 2
end rule
```

4.7 Replacements

The replacement of a rule is also defined using a sequence of terminals and variables. Because all references to variables in a replacement are subsequent uses of variables previously introduced either as formal parameters or in the pattern of the rule, they must not have explicit types.

Each variable reference in the replacement may have one or more (sub-)rules applied to it. When a replacement tree is built, each variable reference is replaced with a copy of the subtree to which the variable is bound. If the variable has rules applied to it, they are applied before evaluating the rest of the replacement.

The syntax for rule application is to list each rule or function in square brackets following the variable name. The parameters to each rule appear within the square brackets. In general, only variables are passed as parameters to a rule, although terminal symbols may be passed if explicitly quoted.

As an example, consider the following rule (taken from a program that evaluates dot products of vectors):

```
rule evaluateAdditions
    replace [expression]
        N1 [number] + N2 [number]
    by
        N1 [+ N2]
end rule
```

The replacement in this rule will be built by applying the predefined rule + with parameter *N2* to the tree matched by *N1*.

4.8 Each

Rule applications involving parameters that are lists or repeats can be modified using the modifier *each*. The *each* keyword is inserted as an extra parameter and has the effect of reapplying the rule for each element of the following actual parameter, which must be a *list* or *repeat*. The type of the corresponding formal parameter of the rule must be the same as the element type of the list or repeat.

For example, the replacement of the following rule applies the subrule *expandConstant* once for each `[statement]` element of the `[repeat statement]` tree *ConstDefs*.

```
rule expandConstants
    replace [repeat statement]
        Block [repeat statement]
    construct ConstDefs [repeat statement]
        Block [deleteNonConstDefs]
    by
        Block [expandConstant each ConstDefs]
end rule
```

```
rule expandConstant ConstantDefinition [statement]
    deconstruct ConstantDefinition
        const ConstName [id] = ConstValue [expression] ;
    replace [primary]
        ConstName
    by
        ( ConstValue )
end rule
```

Each may appear only once in a parameter list, and all parameters following the *each* are affected. All affected parameters must be lists or repeats of the same length, and the rule is applied once for each pair of corresponding elements. For example, the replacement of the following rule applies the rule *substituteActual* once for each pair of corresponding *Formals* and *Actuals.*

```
rule expandInlineFunction  FunctionName [id]  Formals [list id]
                               FunctionExpn [expression]
    replace [primary]
        FunctionName ( Actuals [list expression] )
    by
        ( FunctionExpn [substituteActual each Formals Actuals] )
end rule

rule substituteActual  FormalName [id]  ActualExpn [expression]
    replace [primary]
        FormalName
    by
        ( ActualExpn )
end rule
```

4.9 Pattern and Replacement Refinement

This section describes the optional components of rules that give them more sophistication and power. These three components are *deconstructors*, *constructors*, and *conditions*. Any number of each of these may appear in a rule either before the pattern, or between the pattern and replacement. When more than one appears, they are interpreted in sequential order. Those that appear before the pattern are interpreted (only once) before the scope of application is searched for the pattern. Those that appear after the pattern are reinterpreted each time the pattern is matched.

Because deconstructors and constructors have, in themselves, patterns and replacements, we will use the phrases *main pattern* and *main replacement* to make it clear that we are talking about the pattern and replacement of the rule when just using *the pattern* or *the replacement* would be ambiguous.

When we add in all of these optional parts, the syntax for a rule definition becomes:

```
rule ruleName parameterList
    parts
    replace [type]
        pattern
    parts
    by
        replacement
end rule
```

where *parameterList* is zero of more of:

```
parameterName [type]
```

and where each *parts* is any number of deconstructors, constructors and conditions as defined below.

4.10 Deconstructors

Deconstructors are used to take variables (and parameters) apart into smaller pieces using a more refined pattern. They may appear at any point before the replacement in a rule body. A deconstructor takes the form:

```
deconstruct varName
    pattern
```

where *varName* is a subsequent reference to a variable previously defined in the rule, and *pattern* is a pattern like the main pattern of a rule.

The nonterminal type of the deconstructor's pattern is implicitly the type of the variable being deconstructed. The deconstructor pattern is compared to the *entire* tree bound to the variable. If the pattern matches, then any new variables in the deconstructor pattern are bound accordingly.

If a deconstructor pattern does not match, then the rule is considered to have not matched its pattern (i.e., the main pattern match is discarded and a new match is searched for). If a deconstructor appearing before the main pattern (i.e., a deconstruct of a formal parameter) does not match, then the rule is considered to have failed and no search for the main pattern is done. A tree matches a rule's pattern only when it matches the rule's main pattern <u>and</u> all of the rule's deconstructors match as well.

As an example of how we might use a deconstructor, consider the following rule that takes a sequence of numbers as parameter. The deconstructor splits this parameter sequence into its head (the first number in the sequence) and its tail (the rest of the numbers in the sequence). The rule can then go on to use these pieces in its pattern and/or replacement. In this case, the number at the head of the parameter list (*Head*) is passed as a parameter to the subrule *ruleThatUsesANumber*.

```
rule takesASequence  MySequence [repeat number]
    deconstruct MySequence
        Head [number] Tail [repeat number]
    replace [repeat number]
        OldList [repeat number]
    by
        OldList [ruleThatUsesANumber Head]
end rule
```

Searching deconstructors can be used to search for and take apart a subtree of the deconstructed tree. In this case, the deconstructor finds the first (leftmost shallowest) embedded subtree of the tree bound to the deconstructed variable that matches the pattern. A searching deconstructor takes the form:

```
deconstruct * [type] varName
    pattern
```

where *varName* and *pattern* are as before, and *type* is any nonterminal type name. The *[type]* is optional and defaults to the type of the deconstructed variable if omitted.

The pattern of a searching deconstructor must be of the given type. The deconstructor searches the tree bound to *varName* for the first subtree that matches the pattern, and binds the pattern variables accordingly.

Searching deconstructors can be very useful as guards in rules that are only interested in applying a set of rules to a tree when a given property is present in the tree. For example, the following rule applies the subrule *fixUpIfStatements* to a procedure body only if the procedure actually contains an if statement:

```
rule fixUpIfStatementsInProcedures
    replace [repeat statement]
        procedure P [id]
            Body [repeat statement]
        'end P
        Rest [repeat statement]

    % deep deconstruct Body to see if it has an if statement in it
    deconstruct * [statement] Body
        IfStmt [ifStatement]

    by
        procedure P
            Body [fixUpIfStatements]
        'end P
        Rest
end rule
```

4.11 Conditions

A *condition* is a sequence of rules that are applied to a single variable solely to yield success or failure. The condition must succeed for the rule to continue with a match. A condition takes one of the forms:

```
a.  where
        conditionalExpression

b.  where not
        conditionalExpression
```

where a *conditionalExpression* is a sequence of rules applied to a single variable, for example:

```
where
    X [isANumber] [orIsAnIdentifier]
```

A condition *succeeds* if and only if one of the rules applied finds a match to its pattern (or in the case of form (b), if and only if none of the rules applied finds a match to its pattern).

Conditions, like deconstructors, are considered to be refinements of the rule's pattern. If, for a particular main pattern match, any of the conditions fail, then we consider the main pattern not to have matched, and continue searching for another match. A tree matches a rule's pattern, then, only when it matches the rule's main pattern, and all of the rule's deconstructors match, and all of the rule's conditions succeed.

The rules that are applied in a condition must all be *condition rules*, that is, rules that do not do a replacement but simply try to match a pattern (see "Condition Rules" below).

There are several built-in condition rules that are intended to be used in conditions, including =, < and >. For example, the following rule will (bubble-)sort a sequence of numbers:

```
rule sort
    replace [repeat number]
        N1 [number] N2 [number] Rest [repeat number]
    where
        N1 [< N2]      % < is a built-in rule that matches iff N1 < N2
    by
        N2 N1 Rest
end rule
```

Built-in rules are discussed in detail in section 4.14.

The *sort* rule relies on the fact that every trailing subsequence of a `[repeat number]` sequence is itself a `[repeat number]` sequence and hence can be matched by the pattern of the rule. Thus the rule continues transforming matching trailing subsequences until there are no pairs of misordered adjacent numbers left in the result.

4.12 Constructors

Constructors are used to build intermediate subtrees for use later in a rule. A constructor explicitly introduces a new variable name and binds the constructed tree to it. Constructors may appear at any point before the replacement in the rule body. A constructor takes the form:

```
construct varName [type]
     replacement
```

where *varName* is the name of the new variable, *type* is the nonterminal type of the tree to be constructed and *replacement* has the same syntax as a main replacement.

The replacement of a constructor is evaluated in exactly the same way as the main replacement in a rule, except that it doesn't replace anything. The constructed tree is bound to the variable and can be used in the subsequent patterns and replacements in the rule.

Constructors are frequently used to allow application of a subrule to a replacement built out of many parts. For example, the rule:

```
rule addToSortedSequence NewNum [number]
     replace [repeat number]
          OldSequence [repeat number]

     construct NewSequence [repeat number]
          NewNum OldSequence
     by
          NewSequence [sort]
end rule
```

constructs a sequence called *NewSequence* in the middle of the rule. We can then invoke the *sort* rule on the new sequence we have built to sort the new number into its proper place in the result.

4.13 Limiting the Scope of Application of Rules

Sometimes we would like to limit the application of a rule to the higher levels of a tree. For example, we may want a rule to sort the declarations before statements in the body of a particular procedure, but not in any nested procedures in an input program. We do this by removing subtrees identified by a particular nonterminal type from the scope of application. When we say "remove a subtree from the scope of application", we mean that the subtree should not be searched for pattern matches - not that the tree cannot be changed. The syntax is to insert the keyword *skipping*, followed by the type of the subtrees to be removed from the search, immediately before the keyword *replace*. For example, a single level sort of declarations before statements can be written as:

```
rule sortDeclarationBeforeStatements
     skipping [declaration]
     replace [repeat declarationOrStatement]
          S [statement]
          D [declaration]
          RestOfScope [repeat declarationOrStatement]
     by
          D
          S
          RestOfScope
end rule
```

When applied to the code in the body of a procedure, this rule will sort declarations before statements in the body of the procedure itself, but will never search for matches inside any nested `[declaration]` in the procedure body, and hence will not sort the bodies of any nested procedures.

4.14 Built-in Rules

Built-in rules provide a set of common operations that are difficult, awkward or inefficient to implement directly in TXL. TXL predefines the following built-in rules:

Arithmetic Operations	(N1, N2 must be of type [number] or any other numeric type)
`N1 [+ N2]`	numeric sum N1 + N2
`N1 [- N2]`	numeric difference N1 - N2
`N1 [* N2]`	numeric product N1 * N2
`N1 [/ N2]`	numeric quotient N1 / N2
String Operations	(S1, S2 must be of type [stringlit] or [charlit], N1 of type [number])
`S1 [+ S2]`	concatenation of S1 and S2
`S1 [: N1 N2]`	substring of S1 from char N1 through char N2 inclusive (1-origin)
`N1 [# S2]`	length of S2
Operations on Identifiers	(ID1, ID2 must be of type [id] or any other identifier type)
`ID1 [+ ID2]`	identifier concatenation of ID1 and ID2
`ID1 [_ ID2]`	identifier concatenation of ID1, underscore, and ID2
`ID1 [!]`	unique new identifier beginning with ID1
Standard Constructors	
`R1 [. R2]`	R1 must be of type [repeat X] for some type [X], R2 must be either [repeat X] or [X] result is the splice of R1 and R2
`L1 [, L2]`	L1 must be of type [list X] for some type [X], L2 must be either [list X] or [X] result is the list splice of L1 and L2
`R1 [^ A2]`	R1 must be of type [repeat X] for some type [X], A2 can be any type at all result is a sequence containing every subtree of type [X] contained in A2
Comparisons	(X1, X2 must both be of type [stringlit] or [charlit], or X1, X2 must both of type [number] or other numeric type)
`X1 [> X2]`	succeeds if X1 > X2
`X1 [>= X2]`	succeeds if X1 >= X2
`X1 [< X2]`	succeeds if X1 < X2
`X1 [<= X2]`	succeeds if X1 <= X2

Equality Comparisons	(X1, X2 must both be of the same type)
`X1 [= X2]`	succeeds if X1 identical to X2
`X1 [~= X2]`	succeeds if X1 not identical to X2
Fast Ground Substitute	(Y1, Y2 must both be of the same type, X1 can be any other type)
`X1 [$ Y1 Y2]`	result is X1 with Y2 substituted for every occurrence of Y1

4.15 External Rules

An external rule is one that is implemented externally in some other language and used in a TXL program. External rules can be used to provide additional semantics that are awkward or impossible to implement directly in TXL. The syntax for declaring an external rule is:

```
a. external rule ruleName parameterList
b. external function ruleName parameterList
```

Where *ruleName* is an identifier, and *parameterList* is a list of typed identifiers (as in a rule definition).

The choice of whether an external is declared to be a rule or a function is arbitrary and has no effect. This is because the semantics of the external, and hence the question of whether it searches or repeats, is independent of TXL.

TXL provides a small library of useful pre-implemented external rules that can be declared and used in any TXL program. A complete list of these is given in Appendix C. Experienced users can add their own external rule implementations (see the *TXL User's Guide* for details).

4.16 Condition Rules

Rules used in conditions (*where* clauses) are required to be of a special kind called *condition rules.* Condition rules test for a match to their pattern and do nothing else. Syntactically, a condition rule is exactly like a regular rule or function except that the *replace* keyword is replaced by *match* and no replacement (*by* clause) is given.

For example, the following rule *eliminateRedundantDeclarations* removes all redundant variable declarations from a program. A declaration is redundant if there are no references to the variable in its scope of declaration. The condition that there be no references to the variable is tested by inverting the success of the condition rule *references.*

```
rule eliminateRedundantDeclarations
    replace [repeat statement]
         var X [id] : T [typeSpec]
         RestOfScope [repeat statement]
    where not
         RestOfScope [references X]
    by
         RestOfScope
end rule

% a condition rule to test if there are any references to X
rule references X [id]
    match [id]
         X
end rule
```

Transformation rules can be used as condition rules by prepending a ? onto the name of the rule in the rule application. For example, the following rule will never stop since it will continue to match its pattern forever regardless of what subrule R does:

```
rule doRuleR
    replace [repeat statement]
         Scope [repeat statement]
    by
         Scope [R]
end rule
```

What was intended is that the rule should continue as long as rule R continues to so something to the scope. This can be expressed using the condition:

```
    where
         Scope [?R]
```

The rule invocation [?R] invokes replacement rule R as a condition rule - thus it only tests to see if it can match its pattern, and does no replacing. If its pattern matches, then it succeeds and the condition passes, so the invoking rule continues, otherwise the condition fails and the invoking rule will stop.

4.17 Complex Conditions

Arbitrary Boolean conditions can be built up using combinations of *where* clauses.

The *or* of two conditions is represented by the semantics of the *where* clause itself, which succeeds if any of the sequence of condition rules applied succeeds. For example, if we want to specify the condition that a number *N* is less than or equal to another number *M*, we can specify the condition:

```
    where
         N [< M] [= M]  % less than M or equal to M
```

The *and* of two conditions is represented by two *where* clauses in a row, which together succeed only if both conditions succeed. For example, if we want to specify the condition that a number *N* is less than another number *M* and greater than a third number *K*, we use the two conditions:

```
    where
         N [< M]        % less than M
    where
         N [> K]        %     and greater than K
```

Complex conditions can be built up by combining these two forms with the use of *not*, for example, the condition that *M* be less than or equal to *N* and not greater than *K* can be expressed as:

```
    where
         N [< M] [= M]  % less than M or equal to M
    where not
         N [> K]        %     and  not greater than K
```

5. The Unparsing Phase

The unparsing phase is the simplest of the three phases. The unparser simply does an inorder (left subtree, this node, right subtree) walk of the transformed parse tree, writing the leaves to the output, to give an unparsed string representation of the result of the transformations.

5.1 Formatting of Unparsed Output

TXL normally tries to format the unparsed output in approximately 80 characters of width, with a two character indent for each continued line. The formatting of output can however be explicitly controlled using the built-in formatting nonterminals [NL], [IN] and [EX] to automatically produce pretty-printed output. [NL], [IN] and [EX] can be placed anywhere in a grammar and have no effect on either the parse or the transformation, but direct the formatting of unparsed output in the following way:

`[NL]`	force a new line of output
`[IN]`	indent following output lines four (more) spaces
`[EX]`	exdent following output lines four (fewer) spaces

As an example, the followin g definition causes output procedure declarations to be formatted in the standard Pascal pretty-printed way, while parsing exactly the same inputs as the same *define* with no formatting nonterminals.

```
define procedureDeclaration
    procedure [id] [formalParameterList] ;     [NL][IN]
        [declarations]                         [EX]

    begin                                      [NL][IN]
        [statements]                           [EX]
    'end [id]
end define
```

By default TXL uses a predefined spacing strategy for output that is appropriate for most Pascal- and C-like languages. It is also possible to take *total* control of output format by using the TXL run option '–raw'. When using the '-raw' flag is given as a TXL run option, no spacing at all is done in the output unless it is explicitly specified in the grammar using the built-in nonterminal [SP]. For example, the following definition, when used with the '-raw' flag, specifies that the output is to have a spaces around '+' operators but none around '*' operators in the output.

```
define binaryOperation
    % make spaces around the additive operators
    % in the output but none around multiplicative
        [value] [SP] + [SP] [value]
    |   [value] [SP] - [SP] [value]
    |   [value] * [value]
    |   [value] / [value]
end define
```

[SP] has no effect when the '-raw' flag is not given as a run option to TXL.

6. TXL Programs

A TXL program combines a set of nonterminal type definitions with a set of rules and functions. The types are defined using *define*, *keys* and *compounds* statements, and the rules are defined using *rule*, *function* and *external* rule statements. The order of statements is not important, other that if a name is multiply defined, the last occurrence of the name is taken to be the defining occurrence.

Every TXL program must contain a definition for the nonterminal *program*, which is always the name of the goal nonterminal of the TXL program, the type as which all inputs to the program must be parsed. The rule set must contain a definition for the rule or function *mainRule*, which is automatically applied to the parse tree of the input.

6.1 Comments

TXL comments begin with a percent symbol (%) and continue to the end of the line, as in TeX. If % is to be used as a terminal symbol in the TXL program, it must be quoted with a single quote each time it appears to distinguish it from a comment marker.

6.2 Include Files

A TXL program may include other TXL source files. Include files are equivalent to inserting the included file in the program at the point of the include. Included files may themselves include other source files. The syntax for include files is:

```
        include "filename"
```

where "filename" is a double quoted string containing the (system-dependent) name of the file to be included.

Note that the implementation of file inclusion is such that the keyword *include*, like *rule*, *function* and *define*, is reserved in TXL. If any of these words appear as terminal symbols in the grammar then they must be quoted with a single quote (e.g. `'include`) each time they appear. A complete list of TXL keywords is given in the *keys* section of Appendix A.

Appendix A - Formal Syntax for TXL Programs

We use the TXL grammar notation to define the syntax for TXL.

```
% TXL comments begin with a '%' and end at end of line.
% The '%' character must be quoted if it appears as part of a TXL program.
comments
    '%
end comments

% The following are keywords of TXL and must be quoted if they appear
% in a TXL program with other than their TXL meaning.
keys
    'by 'construct 'deconstruct 'define 'end 'external 'function
    'include 'keys 'list 'match 'not 'opt 'repeat  'replace 'rule
    'skipping 'where
end keys

define program
    [repeat statement]
end define

define statement
        [includeStatement]
    |   [keysStatement]
    |   [compoundsStatement]
    |   [commentsStatement]
    |   [defineStatement]
    |   [ruleStatement]
    |   [functionStatement]
    |   [externalStatement]
end define

define includeStatement
    'include [stringlit]            % string literal is file name
end define

define keysStatement
    'keys
        [repeat literal]
    'end 'keys
end define

define compoundsStatement
    'compounds
        [repeat literal]
    'end 'compounds
end define

define commentsStatement
    'comments
        [repeat commentConvention]       % one convention per line
    'end 'comments
end define
```

```
define commentConvention
         [literal]                  % start symbol (comment to end of line)
    |    [literal] [literal]        % start / end symbol pair
end define

define defineStatement
    'define [typeid]
         [repeat literalOrType]
         [repeat barLiteralsAndTypes]
    'end 'define
end define

define barLiteralsAndTypes
    '| [repeat literalOrType]
end define

define literalOrType
    [literal] | [type]
end define

define type
         '[ [typeid] ']
    |    '[ 'opt [typeidOrQuotedLiteral] ']
    |    '[ 'repeat [typeidOrQuotedLiteral] [opt plusOrStar] ']
    |    '[ 'list [typeidOrQuotedLiteral] [opt plusOrStar] ']
end define

define plusOrStar
    '+ | '*
end define

define typeidOrQuotedLiteral
         [typeid]
    |    [quotedLiteral]
end define

define ruleStatement
         'rule [ruleid] [repeat formalArgument]
              [repeat constructDeconstructOrCondition]
              [opt skippingType]
              'replace [type]
                   [pattern]
              [repeat constructDeconstructOrCondition]
              'by
                   [replacement]
         'end 'rule

    |    'rule [ruleid] [repeat formalArgument]
              [repeat constructDeconstructOrCondition]
              [opt skippingType]
              'match [type]
                   [pattern]
              [repeat constructDeconstructOrCondition]
         'end 'rule
end define
```

```
define functionStatement
         'function [ruleid] [repeat formalArgument]
              [repeat constructDeconstructOrCondition]
              'replace [opt '*] [type]
                   [pattern]
              [repeat constructDeconstructOrCondition]
              'by
                   [replacement]
         'end 'function

    |    'function [ruleid] [repeat formalArgument]
              [repeat constructDeconstructOrCondition]
              'match [opt '*] [type]
                   [pattern]
              [repeat constructDeconstructOrCondition]
         'end 'function
end define

define externalStatement
         'external 'rule [ruleid] [repeat formalArgument]
    |    'external 'function [ruleid] [repeat formalArgument]
end define

define formalArgument
    [varid] [type]
end define

define constructDeconstructOrCondition
         [constructor]
    |    [deconstructor]
    |    [condition]
end define

define constructor
    'construct [varid] [type]
         [replacement]
end define

define deconstructor
    'deconstruct [opt '*] [opt type] [varid]
         [pattern]
end define

define condition
    'where [opt 'not]
         [expression]
end define

define skippingType
    'skipping [type]
end define

define pattern
    [repeat literalOrVariable]
end define
```

```
define literalOrVariable
        [literal]
    |   [newVariable]       % introduction of a new variable
    |   [oldVariable]       % use of an already defined variable
end define

define newVariable
    [varid] [type]
end define

define oldVariable
    [varid]
end define

define replacement
    [repeat literalOrExpression]
end define

define literalOrExpression
        [literal]
    |   [expression]
end define

define expression
    [varid] [repeat ruleApplication]
end define

define ruleApplication
    '[ [ruleid] [repeat varid] [opt 'each] [repeat varid] ']
end define

define literal
    [quotedLiteral] | [unquotedLiteral]
end define

define quotedLiteral
    '' [unquotedLiteral]     % note: '' means a single quote, not two!
end define

define unquotedLiteral
        [id]
    |   [stringlit]
    |   [charlit]
    |   [number]
    |   [key]
    |   [repeat special+]   % sequence of contiguous special chars
end define

define special
        '! | '@ | '# | '$ | '^ | '& | '* | '( | ') | '_ | '+ | '{ | '}
    |   ': | '< | '> | '? | '~ | '\ | '= | '- | '; | ', | '. | '/
    |   '[ | '] | '|   % literals involving these three must be quoted
end define
```

```
define varid
    [id]                % identifier which is a variable name
end define

define typeid
    [id]                % identifier which is a type (define) name
end define

define ruleid
    [id]                % identifier which is a rule or function name
end define
```

Appendix B - Detailed semantics of opt, repeat and list

The notation `[opt X]` generates the define statement:

```
define opt__X
         [X]
    |    [empty]
end define
```

Every reference to `[opt X]` is equivalent to a reference to `[opt__X]` .

The notation `[repeat X]` generates the define statements:

```
define repeat__X
         [repeat_1_X]
    |    [empty]
end define

define repeat_1_X
    [X] [repeat__X]
end define
```

Every reference to `[repeat X]` is equivalent to a reference to `[repeat__X]` .
`[repeat X+]` generates the same set of defines but uses `[repeat_1_X]` as the equivalent.

The notation `[list X]` generates the define statements:

```
define list__X
         [list_1_X]
    |    [empty]
end define

define list_1_X
    [X] [list_opt_rest_X]
end define

define list_opt_rest_X
         [list_rest_X]
    |    [empty]
end define

define list_rest_X
    , [list_1_X]
end define
```

Every reference to the nonterminal `[list X]` is equivalent to a reference to `[list__X]`.
`[list X+]` generates the same set of defines but uses `[list_1_X]` as the equivalent.

Appendix C - Pre-implemented External Rules

TXL includes a small library of pre-implemented external rules and functions that can be used by any TXL program simply by including an external statement. The following is a list of the pre-implemented rules presently in the TXL library.

SYNTAX: `external function message M [any]`

SYNOPSIS: Prints the leaves of the parameter tree as a message on the debugging output. The parameter may be of any type, but is often most useful when a [stringlit] (see example below). The scope tree applied to is ignored and unchanged.

EXAMPLE: Useful in tracing the progress of a transform, as in:

```
rule reduceExpression
    replace [expression]
        E [expression]
    by
        E [message '"reducing additions"]
          [reduceAdditions]
          [message '"reducing multiplications"]
          [reduceMultiplications]
end rule
```

SYNTAX: `external function print`

SYNOPSIS: Prints out the leaves of the scope tree applied to on the debugging output. The scope tree applied to is unchanged.

EXAMPLE: Useful in viewing intermediate results when debugging, as in:

```
rule reduceExpression
    replace [expression]
        E [expression]
    by
        E [reduceAdditions] [print]
          [reduceMultiplications]
end rule
```

Also useful in creating two separate transform outputs from one TXL program, as shown below. Under Unix, the two outputs can be separated using output redirection.

```
rule mainRule
    replace [program]
        P [program]
    construct P1 [program]
        P [transformSetOne] [print]   % debugging stream output
    by
        P [transformSetTwo]           % standard stream output
end rule
```

SYNTAX: **external rule** debug

SYNOPSIS: Prints out the tree representation of the scope tree applied to on the debugging output. The scope tree applied to is unchanged.

EXAMPLE: Useful in viewing intermediate trees when debugging, as in:

```
rule reduceExpression
    replace [expression]
        E [expression]
    by
        E [message '"reducing additions in"] [debug]
          [reduceAdditions]
          [message '"reducing multiplications in"] [debug]
          [reduceMultiplications]
end rule
```

SYNTAX: **external rule** breakpoint

SYNOPSIS: Prints out a breakpoint message on the debugging output and temporarily halts the transform until a carriage return is pressed on the keyboard.
The scope tree applied to is ignored and unchanged.

EXAMPLE: Useful in conjunction with the other debugging rules, as in:

```
rule reduceExpression
    replace [expression]
        E [expression]
    by
        E [reduceAdditions] [message '"so far, we have:"]
                            [print] [breakpoint]
          [reduceMultiplications]
end rule
```

SYNTAX: **external function** quote X [any]

SYNOPSIS: Replaces a scope tree of type [stringlit] or [charlit] with a string containing the text of the parameter's value. The parameter must be of one of the predefined types.

EXAMPLE: Can be used to create customized conditions and error messages, as in:

```
function checkNotSpecial
    % check that an identifier does not begin with 'Z_'
    match [id]
        X [id]
    construct SX [stringlit]
        _ [quote X]
    % check if X begins with 'Z_'
    where
        SX [: 1 2] [= '"Z_"]
    % if we make it here then it does, so print an error message
    construct Message [stringlit]
        _ [+ '"Error: id "] [+ SX] [+ '" begins with 'Z_'"]
          [print]
end function
```

SYNTAX: **external function** unquote S [stringlit]

SYNOPSIS: Replaces a scope tree of type [id] with a the text of the parameter as an identifier. The parameter must be of type [stringlit] or [charlit]. The text of the parameter need not be a legal identifier.

EXAMPLE: Can be used to print or include arbitrary unquoted text in output, as in:

```
function checkNotSpecial
    % check that an identifier does not begin with 'Z_'
    match [id]
        X [id]
    construct SX [stringlit]
        _ [quote X]
    where
        SX [: 1 2] [= '"Z_"]    % does X begin with 'Z_' ?
    construct Message [stringlit]
        _ [+ '"Error: id "] [+ SX] [+ '" begins with 'Z_'"]
    construct UnquotedMessage [id]
        _ [unquote Message]
          [print]              % output message without quotes
end function
```

Index