The TXL *Programming Language*

Version 10.5

November 2007

James R. Cordy

Ian H. Carmichael • Russell Halliday

Highlighted sections/subsections are under required readings.

James R. Cordy et al.

The TXL Programming Language
Version 10.5

© 1991-2007 James R. Cordy, Ian H. Carmichael and Russell Halliday

Versions and portions have also appeared
© 1991 Gesellschaft für Mathematik und Datenverarbeitung mbH,
© 1991-2004 Queen's University at Kingston,
© 1995-2000 Legasys Corporation, and others. Used by permission.

November 2007

Software Technology Laboratory
School of Computing
Queen's University at Kingston
Kingston, Ontario K7L 3N6
Canada

http://www.cs.queensu.ca/~stl

Table of Contents

1. Introduction

This document 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 text into a single output text. Sorting a list of numbers, processing data to generate statistics, formatting text, or even 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 the output text.

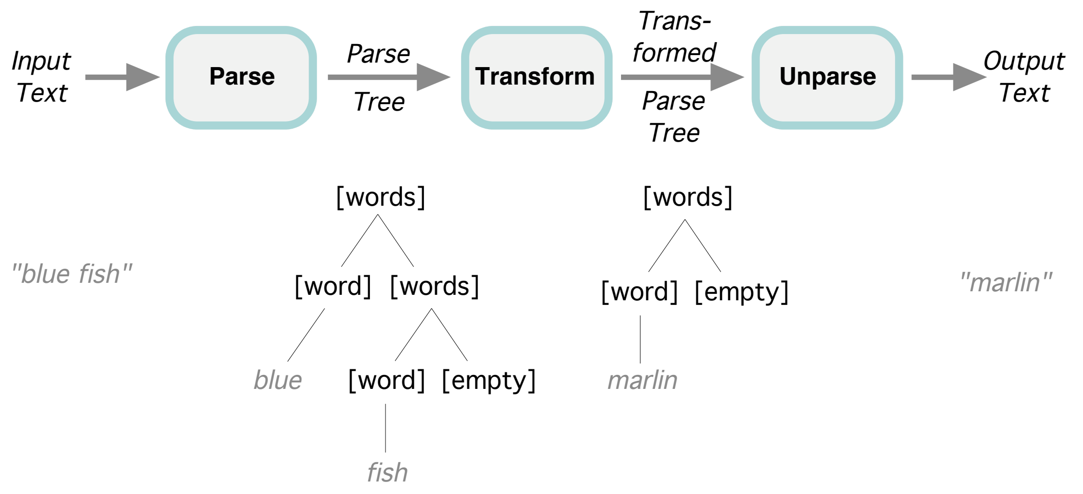

Figure 1. The Three Phases of TXL

3. The Parsing Phase

The parsing phase is responsible for parsing the input to the TXL program according to the given grammar. The grammar is specified in a notation similar to Extended Backus-Nauer Form (BNF). TXL's parser can handle virtually any context-free grammar, no matter the form. While the general parsing strategy is top-down with full backtracking, general left- and right-recursion is supported using a combination of a hybrid bottom-up parsing heuristic and a set of highly efficient built-in repetition primitives. For this reason, good TXL grammars use repetition in preference to recursion when describing sequences of items.

TXL is comfortable with ambiguous grammars, which it resolves by exploring alternatives in the order they are given in the grammar. This allows use of more "natural" user-oriented grammars to describe input structure rather than traditional compiler-style "implementation" grammars. As a matter of fact, TXL works better and more efficiently with user-level grammars.

3.1 Grammar Definition

The basic unit of a TXL grammar is the *define* statement. Each define statement gives an ordered set 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 form, 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 alternatives.

For example, the TXL nonterminal definition:

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

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 the keywords of TXL (see Appendix A) must be quoted (preceded with a single quote, e.g. '[ ) if they are intended to be terminals in a nonterminal definition. For example, the square brackets used for array subscripting in the C language would have to appear as '[and '] respectively in a TXL grammar for the syntax of that language. Any terminal symbol may be quoted if desired; it is considered good TXL style to quote *all* terminal symbols.

By convention, the nonterminal type [*program*] is the goal symbol (the type as which the entire input must be parsed) for every TXL program. Every TXL program must contain a definition for *program*.

The general form of a define statement is

```
define name
         alternative1
     |   alternative2
     |   alternative3
              .
              .
     |   alternativeN
end define
```

Where each *alternative* is any sequence of terminal symbols and nonterminal types enclosed in square brackets [ ]. Each define statement introduces a nonterminal type of the given name. The defined type can be used anywhere in the TXL program; defines need not appear in any particular order (however, by convention, the definition for *program* often appears first).

Previously defined nonterminal types can be *overridden* using the *redefine* statement. A redefine statement gives a new definition for a previously defined nonterminal name. For example, if the redefine statement

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

appeared later in the TXL program than the *define* statement shown in Figure 2, then this new definition would override the original, and *all* references to the nonterminal type [expression] would refer to this new type, even if the reference appeared before the override. Overrides are most useful when defining grammars for dialects of existing languages, in which some features may have a new or extended syntax, while the majority of others retain their original definition.

Previously defined nonterminals may also be *extended* using a redefine. An extension is simply an override that begins or ends with an ellipsis ... followed by any number of additional alternatives separated by or bars |. For example, we could extend the definition for [expression] above to allow multiplication and division by including the redefine statement

```
redefine expression
          ...
     |    [expression] * [expression]
     |    [expression] / [expression]
end redefine
```

later in the TXL program.

The general form of a redefine statement is the same as the define statement, except that the ellipsis may be used as either the first or the last alternative form. In both cases, the ellipsis represents the alternative form(s) of the previous (re-)definition of the nonterminal. There may be any number of redefines of the same nonterminal in a program; each redefine overrides or extends the textually previous (re-)definition.

In contrast to other parsing tools, the ambiguity associated with the above nonterminal definitions being both direct left-recursive and direct right-recursive is typical of TXL grammars. In general, the simplest possible grammar is the best one for use with TXL. Ambiguity (or rather the lack thereof) is not an issue, and in many cases can be exploited to simplify a transformation.

When parsing input using such a grammar, TXL resolves ambiguities by choosing the first alternative of each nonterminal definition that can match the input, backtracking to try the next alternative if necessary. Alternatives are always considered in the order that they are given in the original grammar, and TXL will always choose to parse using the first matching alternative. Thus TXL grammars are *ordered* grammars.

3.2 Predefined Nonterminal Types

Certain predefined nonterminal types match the basic grammatical classes of input items (known as *tokens* in TXL). By default, TXL recognizes the following set of input item types. In every case, the longest possible sequence of input characters is matched to the type. Once matched to a type, an input item is permanently typed and its type cannot be changed. Input items are classified by attempting matches to the types in the order given below. In cases of ambiguity, the first matching type is used.

Type	Matches
`[id]`	Any identifier beginning with a letter or underscore and continuing with any number of letters, digits and underscores. e.g., `ABC, abc, a9b56, aBc, AbC, a_7bc, Ab_C_, _abc, _22, _` The character set to be recognized in identifiers can be extended using the *–idchars* command line option, e.g. *–idchars '$#'* will cause both $ and # to be accepted in identifiers.
`[number]`	Any unsigned integer or real number beginning with a digit and continuing with any number of digits, an optional decimal point followed by at least one more digit, and an optional exponent beginning with the letter E or e and followed by an optional sign and at least one digit. e.g., `123, 12.34, 123.45e22`
`[stringlit]`	Any double quoted string beginning and ending with a double quote (") and consisting of any number of characters between. Any character is allowed in a string except the character " itself and the ASCII character NUL (Hex 00). If a string escape character such as "\" has been specified using the *–esc* command line option, then " may appear in the string escaped (e.g., "\""). e.g., `"Hi there", "Hello \"There\"", ""`
`[charlit]`	Any single quoted string beginning and ending with a single quote (') and consisting of any number of characters between. Any character is allowed in a string except the character ' itself and the ASCII character NUL (Hex 00). If a string escape character such as "\" has been specified using the *–esc* command line option, then ' may appear in the string escaped (e.g., '\'). e.g., `'Hi there', 'Hello \'There\'', ''`
`[comment]`	Any input language comment as defined in the **comments** section of the TXL program (see "Comments Statement" below). Comment tokens are ignored by TXL unless the *–comment* command line option is used, in which case they are accepted as input items. If comments are to be accepted as input items, then the grammar defined by the TXL program must be prepared to accept them wherever they may occur in the input.

Any input item that is not an [id], [number], [stringlit], [charlit] or [comment] as defined above represents a literal terminal symbol. Normally such items consist of the next non-blank single character of input unless the character begins an input language compound token as defined in the **compounds** section of the TXL program (see "Compounds Statement" below) or a user-defined token pattern as defined in the **tokens** section of the TXL program (see "Tokens Statement" below). Most punctuation and special characters fall into this category.

Input identifiers that are explicitly listed in the **keys** section of the TXL program (see "Keys Statement" below) are not accepted as [id]'s, but rather represent literal terminal keyword symbols.

By default all TXL input is treated as case-sensitive, that is, *Abc, ABC* and *abc* are treated as distinct input items. This behaviour can be changed using the command line options *–upper, –lower* and *–case.* Input can be processed in a completely case-insensitive way using –*case*, which specifies that input items differing only by the case of their letters are to be treated as equivalent, so for example *Abc, ABC* and *abc* are all treated as the same item, while retaining their own distinct forms in output. In the case of –*upper* and –*lower*, all items are converted to upper case or lower case respectively on input and retain their converted form in output.

By default [stringlit], [charlit], [comment] and user defined input token types may be multi-line (that is, may contain newline characters) unless the –*nomultiline* command line option is specified, in which case they are limited to a single line.

Except insofar as it serves as a delimiter, all white space in the input is ignored except when embedded in a [stringlit], [charlit], [comment] or user-defined input token type. By default TXL treats spaces, newlines (ASCII CR and/or LF), tab characters (ASCII HT) and form feeds (ASCII FF, Ctrl-L) outside of these input types as white space, and all other characters as input. The character set to be ignored as white space can be extended using the –*spchars* command line option, e.g. –*spchars ';,'* will cause all semicolons and commas to be ignored as white space.

The –*char* command line option changes the default behaviour by making all white space characters and line boundaries significant, so that all input characters become part of the input to be parsed. In this mode the grammar must be prepared to accept line boundaries and white space as input tokens wherever they may occur in the input. When –*char* is specified, the following additional predefined nonterminal types are recognized:

Type	**Matches**
`[space]`	A white space token - that is, any sequence of blank (ASCII SP) and tab (ASCII HT) characters. If white space is to be accepted as input, then the grammar defined by the TXL program must be prepared to accept [space] tokens wherever they may occur in the input.
`[newline]`	A newline token - that is, an NL (ASCII LF) character in Unix/Linux and MacOSX, or an end of line (ASCII CR LF) in Windows. If line boundaries are to be accepted as input, then the grammar defined by the TXL program must be prepared to accept [newline] tokens wherever they may occur in the input.

The –*newline* command line option changes the default behaviour by making only line boundaries significant, so that newlines become part of the input to be parsed. Other white space characters (spaces and tabs) continue to act only as delimiters and are ignored as usual. When –*newline* is used, the grammar must be prepared to accept [newline] tokens wherever they may occur in the input, but need not account for spacing.

In addition the the basic nonterminal types, TXL implements a number of predefined types that further constrain the characters in the input items to be recognized. When used in a grammar, these types act like the corresponding basic types except that only input items matching the constraints are accepted.

Type	**Matches**
`[upperlowerid]`	Any [id] beginning with an upper case letter. e.g., `AbCdE, ABCDE, A_bcd`
`[upperid]`	Any [id] containing only upper case letters. e.g., `ABCDE`

`[lowerupperid]`	Any [id] beginning with a lower case letter. e.g., `aBcDe, abcde, a_BCde`
`[lowerid]`	Any[id] containing only lower case letters. e.g., `abcde`
`[floatnumber]`	Any [number] with an explicit exponent. e.g., `12.3e22, 12345E22, 1e2`
`[decimalnumber]`	Any [number] with an explicit decimal point. e.g., `123.45, 1.`
`[integernumber]`	Any [number] with no exponent and no decimal point. e.g., `12345, 95, 2`

The special nonterminal [empty] always matches regardless of the input and never accepts anything. It is most useful in crafting grammars with optional or deletable items.

<u>**Type**</u>	<u>**Matches**</u>
`[empty]`	Always recognized regardless of input, but consumes no input items.

Although not of interest to the majority of users, sophisticated TXL programmers may find the following additional type classes to be useful in achieving special effects.

<u>**Type**</u>	<u>**Matches**</u>
`[key]`	Any input language keyword as defined in the **keys** section of the TXL program (see "Keys Statement" below). Normally keywords appear in the grammar as literal terminal symbols representing themselves. This nonterminal on the other hand accepts <u>any</u> keyword of the input language at all, and is not normally used in any but the most sophisticated TXL programs.
`[token]`	Any input item which is not a keyword as defined in the **keys** section of the TXL program (see "Keys Statement" below). This nonterminal accepts <u>any</u> input item at all, including all [id]'s, [number]'s, [stringlit]'s, [charlit]'s, [comment]'s, and literal terminal symbols, and is not normally used in any but the most sophisticated TXL programs.

The set of input nonterminals recognized by TXL can be extended to include other types of input items or modified to recognize the predefined nonterminal types differently using the *tokens* statement (see "The Tokens Statement" below). TXL is "8-bit clean", which means that it can handle any 8-bit character set, including all of the ISO high-bit European and Cyrillic extensions, and all ASCII-based multi-byte UTF-8 character sets.

3.3 Nonterminal Modifiers

Any nonterminal type name enclosed in square brackets may be modified by a nonterminal modifier. The five possible modifiers are *opt*, *repeat*, *list, see* and *not*. 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 *opt*, *repeat* or *list* modified nonterminal is translated by TXL into a predetermined set of internal nonterminal definitions. The structure of these is documented in Appendix B, "Detailed Semantics of *opt, repeat, list* and *attr*".

The nonterminal modifiers *see* and *not* are used to specify lookahead constraints on the input. [**see** X] succeeds if an [X] can be matched in the prefix of the remaining input at this point in the parse, and fails otherwise. [**not** X] fails if an [X] can be matched in the prefix of the remaining input at this point in the parse, and succeeds otherwise. In both cases, if the lookahead succeeds, then the parse continues, otherwise the parser backtracks to see if there is another parse of the previous nonterminals that can change the input such that the lookahead succeeds.

The lookahead modifiers only test the input for a match, they do not actually accept any input. Lookahead modifiers are most useful in implementing robust parsers, in which unrecognized input must be flushed until a recognized form is seen. For example:

```
define flush_until_END
         [repeat anything_but_END]
end define

define anything_but_END
         [not 'END] [token_or_key]
end define

define token_or_key
         [token] | [key]
end define
```

The nonterminal modifiers *push* and *pop* provide the ability to do local context-sensitive matching in a nontermnal definition. For example, [**push** id] matches and saves any [id] in the input for later matching by a [**pop** id]. The nonterminal type modified by *push* or *pop* must be a predefined or user token type. An example of the use of *push* and *pop* is the matching of tag identifiers in XML:

```
define matched_tag
        < [push id] > [repeat content] </ [pop id] >
end define
```

The [matched_tag] nonterminal above will parse only elements whose opening and closing tag identifiers match, and will yield a syntax error for those that do not.

For conciseness, each of the nonterminal modifiers has an equivalent and convenient short form notation that many TXL programmers may prefer. In each case, X may be any nonterminal type or quoted terminal symbol, and T must be a predefined or user token type.

Modifier	Equivalent short form	Example
[**opt** X]	[X?]	[elseClause?]
[**repeat** X]	[X*]	[id*]
[**repeat** X+]	[X+]	[statement+]
[**list** X]	[X,]	[argument,]
[**list** X+]	[X,+]	[formalParameter,+]
[**see** X]	[:X]	[: key]
[**not** X]	[-X]	[~'END]
[**push** T]	[>T]	[>id]
[**pop** T]	[<T]	[<id]

3.4 The Universal Nonterminal [any]

The special nonterminal type [any] is a universal nonterminal that matches any nonterminal type when used as a pattern or replacement (see "Polymorphic Rules" in section 4 for details). Items of type [any] appearing in the grammar can never match input and always cause a syntax error if they are required (that is, if there are no other alternatives allowed). They can however be used in the grammar to allow insertion of arbitrary parse trees by type-cheating rules and functions. See "Polymorphic Rules" in section 4 for details.

3.5 Limiting Backtracking

The special nonterminal type [!] can be used to limit backtracking in a highly ambiguous grammar to speed detection of syntax errors. [!] is used in a sequence of one or more nonterminals to indicate a point at which the parse of the sequence should be committed. If the parser backtracks to a [!], the entire sequence is rejected without retrying the previous nonterminals in the sequence, and the parse proceeds to the next alternative immediately. If there is no other alternative, the nonterminal in which the [!] appears fails, and backtracking continues as usual at the point where it was used in the parse.

3.6 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 predefined nonterminal [id] and its variants. Input keywords are only matched by explicit literal occurrences of the keyword in a nonterminal definition or pattern and by the special predefined nonterminal [key] (see "Predefined Nonterminal Types" above). Keywords are defined using the *keys* statement. A keys statement simply lists the identifiers to be treated as keywords. For example:

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

Any number of keys statements may appear anywhere in a TXL program, although they normally appear at the beginning of a grammar or program. If a keyword of TXL itself (e.g. **end**) is to be a keyword of the TXL program's grammar, it must be quoted every time it occurs in the program (e.g., 'end) including inside the keys statement (as shown above).

It is also possible to specify literal tokens other than identifiers as keywords in the *keys* statement. This is used to specify that the tokens should not be accepted by the [token] nonterminal (and should be accepted by the [key] nonterminal), and has no other effect.

3.7 The Compounds Statement

By default the TXL input scanner treats each special (punctuation) character in the input as a separate literal 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').

Compound tokens are defined using the *compounds* statement. A compounds statement simply lists the sequences of characters to be treated as compound tokens. For example:

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

Any number of compounds statements may appear anywhere in a TXL program, although they normally appear at the beginning of a grammar or program. 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 inside the compounds statement itself (as shown above).

3.8 The Comments Statement

The *comments* statement is used to describe the commenting conventions of the input language so that TXL can either ignore comments (the default) or recognize them as [comment] nonterminals (if the command line option *–comment* is used). The comments 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
```

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 the 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.

Any number of comments statements may appear anywhere in a TXL program, although they normally appear at the beginning of a grammar or program. Several different commenting conventions may be given, specifying that several different kinds of comments are to be accepted in the input. Comment brackets consisting of more than one character are automatically considered to be compound tokens (see "The Compounds Statement" above).

TXL normally ignores all comments when parsing the input language, unless the *–comment* command line option is used. When *–comment* is specified as a command line option, comments in the input are not ignored, but rather are treated as input items of the nonterminal 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 when the *–comment* option is used.

Like all predefined nonterminal types, the [comment] token type may be extended or overridden in a *tokens* statement, see "The Tokens Statement" below. This allows for more complex commenting conventions than can be specified in the *comments* statement.

3.9 The Tokens Statement

The *tokens* statement is used to extend or modify the recognition of input item types ("tokens") to reflect the lexical conventions of the input language to be processed. The tokens statement can be used to add new user-defined input item types as well as to extend or modify the predefined nonterminal types of TXL. Any number of tokens statements may appear in a TXL program, each adding its new input token types to the set of recognized types.

The tokens statement lists the new input item types one per line, each line consisting of an identifier naming the new type and a string literal giving a regular expression pattern for the sequences of input characters to be recognized. For example, the following tokens statement adds the user-defined input item type `[hexnumber]` to the set of recognized input items:

```
        tokens
            hexnumber     "0[Xx][\dABCDEFabcdef]+"
        end tokens
```

The identifier on the left defines the nonterminal name to be used in the grammar to refer to input items of the type, in this case `[hexnumber]`. The string literal gives a regular expression pattern for the sequences of characters to be recognized as input items of the type. In this case the pattern requires that items begin with the character 0 (zero) followed by either an upper- or lower-case X and a sequence of at least one or more digits or upper- and lower-case letters A through F. The complete syntax of regular expression patterns is given in the section "Token Patterns" below.

The standard lexical conventions of the TXL predefined nonterminal types apply to user-defined input types as well. In every case, the longest possible sequence of input characters is matched to the type, and white space is ignored except insofar as it serves as a delimiter or is explicitly specified in the regular expression pattern for an input item type. Once matched to a type, an input item is permanently typed and its type cannot be changed. See the section "Predefined Nonterminal Types" for details.

Input items are classified by attempting matches to the regular expression patterns of defined token types in the order given in the *tokens* statements of the TXL program. In cases of ambiguity, where more than one input token type may match an input item, the first matching type is used. If no user-defined input type's regular expression matches, then the input item is typed using the standard TXL input typing described in the section "Predefined Nonterminal Types".

If the name given for a token pattern is the name of a predefined nonterminal type or a previously user-defined token type, then the new definition overrides the old. For example, the following tokens statement overrides the `[hexnumber]` token type of the previous example to allow Z as well as X as the hexadecimal indicator, and overrides the predefined nonterminal type `[number]` to restrict it to unsigned integer numbers only.

```
tokens
     hexnumber      "0[XxZz][\dABCDEFabcdef]+"
     number         "\d+"
end tokens
```

An empty pattern can be used to explicitly undefine a predefined or previously user-defined token type. For example, the following token definition explicitly undefines the predefined nonterminal type `[charlit]`, for example when defining the grammar of a language that uses the single quote (') for other purposes.

```
tokens
     charlit        ""
end tokens
```

Several different alternative patterns for the same token type may be given, separated by the or bar |. For example, we could define the `X` and `x` alternatives for the `[hexnumber]` type separately, as follows:

```
tokens
     hexnumber      "0X[\dABCDEFabcdef]+"
                  | "0x[\dABCDEFabcdef]+"
end tokens
```

Tokens statements can also be used to extend the definition of the predefined nonterminal types or previous user-defined input types, using an ellipsis `...` followed by the or bar | to indicate that the definition is an extension of a previously defined input type rather than an override.

For example, the following tokens statement extends our original definition for the `[hexnumber]` type to allow Z or z as well as X or x, and additionally extends the predefined nonterminal `[id]` to allow `$`, `@` and `&` as the first character in an identifier.

```
tokens
     hexnumber      ... | "0[Zz][\dABCDEFabcdef]+"
     id             ... | "[$@&]\i*"
end tokens
```

The ellipsis symbol `...` indicates that we want to preserve the previous patterns for the input type, and the or bar | indicates the beginning of an additional alternative pattern for inputs of that same type. For brevity, the ellipsis is optional.

In order to insure that user token definitions do not interfere with the lexical conventions of TXL itself, the source of the TXL program is processed using only the predefined token types `[id]`, `[number]`, `[charlit]` and `[stringlit]`, and the user's overrides and extensions to them. Care must be taken when overriding the definition of `[id]` to allow TXL keywords to continue to be recognized as identifiers, and not to allow the TXL metacharacters `[, ]` and | as identifier characters. If user defined tokens are to appear in the TXL program, they must be quoted using a single leading quote `'` .

3.10 Token Patterns

Regular expression patterns for user-defined tokens are built from the following set of metacharacters and operators. Any single character that is not one of the pattern metacharacters `[`, `]`, `(`, `)`, \ or # and is not immediately preceded by a \ or # simply represents itself. For example, the regular expression pattern `"x"` allows only the literal character `x`. A metacharacter preceded by \ or # defeats any special meaning and simply represents the character itself. For example, the regular expression pattern `"\(x\)"` allows only the sequence of characters `(x)` . Similarly, the string literal quote `"` is represented as `\"` when required as part of a token pattern.

Single character patterns - when used in a regular expression pattern, the following match a single input character of the indicated class.

`\d`	a digit, i.e. any one of 0, 1, 2, 3, 4, 5, 6, 7, 8 or 9.
`\a`	an alphabetic character, i.e., any one of a-z or A-Z.
`\u`	any alphabetic character or _ (underscore)
`\i`	an identifier character, defined as: an alphabetic character, an _ (underscore), a digit, or any character specified using the –*idchars* command line option (see the section "Predefined Nonterminal Types")
`\A`	any uppercase alphabetic character
`\I`	any uppercase identifier character
`\b`	any lowercase alphabetic character
`\j`	any lowercase identifier character
`\s`	any special character, defined as: a non-white space character that is not an identifier character, bracket or parenthesis
`\n`	the newline character(s) of the host system (when using –*char* or –*newline*)
`\t`	a tab character
`\c`	any character at all (be careful using this!)
#*C*	any character that is not the character *C*, where *C* is not \
#\C	any character that is not in character class *C*, where *C* is one of d, a, t, i, A, I, b, j, s, as listed above
#\M	where *M* is one of the metacharacters [,], (,), \ or # or a string quote ", means any character but the literal character *M*
\M	where *M* is one of the metacharacters [,], (,), \ or # or a string quote ", means the literal character *M* itself
C	where *C* is not a metacharacter or string quote, means the literal character *C* itself.

Regular expression operators - the single character patterns can be combined with the following pattern operators to build regular expression patterns of arbitrary complexity.

`[`*PQR*`]` any one of *P*, *Q*, or *R* where *P*, *Q* and *R* are regular expression patterns
`(`*PQR*`)` the sequence *P* followed by *Q* followed by *R* where *P*, *Q* and *R* are regular expression patterns (parentheses are required only for grouping the sequence for use with other operators)
P`*` zero or more repetitions of pattern *P*
P`+` one or more repetitions of pattern *P*
P`?` zero or one of pattern *P* (i.e., *P* is optional)

Perhaps the most interesting examples of regular expression patterns are the actual patterns for the predefined nonterminal types of TXL. These patterns precisely specify the input items recognized by the predefined nonterminal types.

```
[id]          "\u\i*"
[number]      "\d+(.\d+)?([eE][+-]?\d+)?"
[stringlit]   "\"[(\\\c)#\"]*\""
[charlit]     "'[(\\\c)#']*'"
```

The first pattern, for `[id]`, specifies an alphabetic character or underscore (`\t)` followed by any number of identifier characters (`\i*`). The more complex pattern for `[number]` specifies one or more digits (`\d+`) followed by an optional fraction part `(.\d+)?` consisting of a decimal point and at least one or more digits (\d+), optionally followed by an exponent part `([eE][+-]?\d+)?` consisting of either an e or an E (`[eE]`), an optional plus or minus sign (`[+-]?`) and at least one or more digits (`\d+`).

The regular expression patterns for `[stringlit]` and `[charlit]` are identical except for the quotes used. In the case of `[stringlit]`, the pattern begins with a string quote (`\"`), followed by any number of characters `[(\\\c)#\"]*` that are either escaped, i.e. consist of the sequence \ followed by any character (`\\\c`) or are not a string quote (#`\"`), and ending with a string quote (`\"`).

When the –*char* command line option is specified, the following additional predefined nonterminal types are defined by the patterns:

```
[space]       "[ \t]+"
[newline]     "\n"
```

The regular expression pattern for `[space]` allows 1 or more repetitions of either a blank (ASCII SP) character or a tab (ASCII HT) character. The expression for `[newline]` allows only the newline character(s) of the host system.

When the –*newline* command line option is specified but not –*char*, the [newline] predefined type is active but not [space]. When either –*newline* or –*char* is specified, token patterns may use the "\n" character pattern, which matches the newline character(s) of the host system (e.g., ASCII LF on Unix/Linux and MacOSX, CR-LF on Windows).

For example, the following definitions accept and ignore (unless –*comment* is specified) Snobol-style "asterisk in column 1" comments:

```
% Make the input newline-sensitive
#pragma -newline
```

```
% Treat all asterisk-in-column-1 lines and lone newlines as comments
tokens
    comment           "\n\*#n*"
                  |   "\n"
end tokens
```

TXL is "8-bit clean", which means that it can handle any 8-bit character set, including all of the ISO high-bit European and Cyrillic extensions, and all ASCII-based multi-byte UTF-8 character sets. Thus token patterns can be used to specify language-dependent characters of the user's own language. In particular, the identifier character set can be extended to allow for both input and TXL programs to be written in the user's native language using the *–idchars* command line option, for example:

```
% Add French extended characters to the valid identifier set
% Ajouter les caractères prolongés français à l'ensemble valide
#pragma -idchars "àâäçéèêëîïôöûùü'"

tokens
    charlit ""        % Disable charlits since French words may contain "'"
                      % Neutraliser les cordes citées simples puisque
                      %     les mots français peuvent contenir «'»
end tokens

define program
    [repeat id]
end define

function main
    replace [program]
        UnPeuD'entréeGelée [program]
    construct LaMêmeChose [program]
        UnPeuD'entréeGelée
    by
        LaMêmeChose
end function
```

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 functions 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 nonterminal type of parse tree that the function transforms (the *target* type), *pattern* is a pattern which the function'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 is consistent with other pure functional languages. 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. Because they explicitly return the original tree if it does not match the pattern, TXL functions are also always *total* - that is, they produce a result for any argument of the appropriate type.

A TXL function is applied in postfix form, using the function name enclosed in square brackets following the first argument. 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 of such a sequence is function composition. That is, *f* is applied first, then *g* to the result of *f*, then *h* to the result of *g*. In more conventional functional notation the composition above would be written *h (g (f (X))*.

"Searching" functions search their scope of application for the first match (leftmost shallowest in the scope 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 the scope 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 transformation *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 X is the name of a tree of type [**repeat** number] containing the number values :

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

the rule application

```
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 6 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 document the term 'rule' refers to <u>both</u> rules and functions except as noted.

4.3 The Main Rule

Every TXL program must have a rule or function named *main.* Execution of a TXL program consists of applying this rule to the entire input parse tree. If other rules are to be applied, then the main rule 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 main
    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 parse tree parameters. The general 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 argument 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 nameless TXL variable. 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. Anonymous variables can be constructed using an explicit constructor (e.g., to achieve the side effects) but cannot be referenced.

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 parsing the pattern, considering variables to be terminal symbols of their nonterminal type. 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
        T * 2
end rule
```

if *foo* is given an argument 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, and each new (i.e., explicitly typed) variable in the pattern is matched to a subtree of the variable's type. Subsequent references to the variable, if any, must be matched to identical subtrees for the pattern match to succeed.

For example, if we apply the rule:

```
rule foo
    replace [expression]
        T1 [term] - T2 [term]
    by
        -(T2) + T1
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
```

Anonymous variables ('_') may be used in patterns as placeholders for matched parts that we are not interested in. Every anonymous variable must be explicitly typed and represents a unique new variable. For example, in the following rule replaces all whole expressions consisting of an addition of any two constant numbers by the number 2, regardless of what numbers were originally being added.

```
rule additionsAllTwo
    replace [expression]
        _ [number] + _ [number]
    by
        2
end rule
```

Anonymous variables cannot be referenced.

When the target type allows, a pattern can be completely empty, denoted by the absence of any pattern at all. For example, the following rule finds every empty argument list and replaces it by a default argument list containing the single argument 1.

```
rule replaceEmptyArgumentLists
    replace [list argument]
    by
        1
end rule
```

It is considered good style to make intentionally empty patterns explicit using a comment. For example, the above rule would be better coded as:

```
rule replaceEmptyArgumentLists
     replace [list argument]
          % one with no arguments
     by
          1
end rule
```

4.7 Replacements

You at least need to know the highlighted part

Like the pattern, the replacement of a rule is defined using a sequence of terminals and variables. However, 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. TXL creates a replacement parse tree by parsing the replacement, considering variables to be terminal symbols of their nonterminal type.

Each variable reference in the replacement may have one or more subrules 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 vector dot products:

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

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

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. (Note: In the present implementation of TXL, at most two parameters are allowed following each.) All affected parameters must be lists or repeats, 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
```

As a shorthand, when the nonterminal type being replaced derives `[empty]` (for example if it is a `[repeat]` or `[list]` type), the entire replacement can consist of a reference to the empty variable '_' and rules applied to it, with the semantics that it is a variable of the target type bound to an empty match. For example, the following function replaces the sequence of statements by the same sequence in reverse, by prepending each statement to the result sequence of previous ones, beginning with an empty sequence denoted by the empty variable '_'.

```
function reverseStatements
    replace [repeat statement]
        Statements [repeat statement]
    by
        _ [prependStatement each Statements]
end function

function prependStatement  Statement [id]
    replace [repeat statement]
        PreviousStatements [repeat statement]
    by
        Statement
        PreviousStatements
end function
```

The empty variable can also be used as the replacement for any predefined or user token nonterminal type, for example `[stringlit]` or `[id]`, with the semantics that it denotes an empty string (`""`) or token (see "Constructors" for details and an example).

When the target type allows, a replacement can also be completely empty, denoted by the absence of any replacement at all. For example, this rule finds every argument list and replaces it with an empty one.

```
rule makeArgumentListsEmpty
    replace [list argument]
        _ [list argument]
    by
end rule
```

It is considered good style to make intentionally empty replacements explicit using a comment. For example, the above rule would be better coded as:

```
rule makeArgumentListsEmpty
    replace [list argument]
        _ [list argument]
    by
        % an empty one
end rule
```

4.8 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 (i.e., the *replace* or *match* clause of the rule) 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.

4.9 Deconstructors We haven't talked about it yet, but good to know

Deconstructors are used to break 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. The rule matches and a replacement can be made only if the main pattern is matched and all of the 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
```

When the target type allows, the deconstructor pattern can be completely empty, denoted by the absence of any pattern at all. For example, if *Tail* is a bound variable of type `[repeat number]` as in the example above, the following deconstructor would match only number sequences bound to *Tail* that are completely empty:

```
deconstruct Tail
    % none
```

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 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 [procedure_declaration]
        procedure P [id]
            Body [repeat statement]
        'end P
    % 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
end rule
```

The scope of a searching deconstructor's search can be limited to the higher levels of a tree in the same way that we limit the scope of application of a rule. 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 *deconstruct*.

For example, if we are interested only in outer-level expressions that are not nested inside another expression, then we can use the deconstructor:

```
    skipping [expression]
    deconstruct * [expression] Scope
        Expn [expression]
```

The sense of a deconstructor or searching deconstructor can be negated using the keyword *not* following *deconstruct*:

```
deconstruct not varName
    pattern

deconstruct not * [type] varName
    pattern
```

In this case the rule is considered not to have matched its pattern (i.e., the main pattern match is discarded and a new match is searched for) if the deconstructor does match its pattern. That is, the rule continues with its main pattern match only if no match can be found for the pattern of the deconstruct. Because the rule continues in this case only if the deconstructor does not match its pattern, no variables are bound by a *deconstruct not.*

4.10 Conditions

Try to work with all. These are really interesting!

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

c.    where all
          conditionalExpression

d.    where not all
          conditionalExpression
```

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

```
where
    X [containsANumber] [containsAnIdentifier]
```

A condition of form (a) *succeeds* if and only if at least 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). Thus the condition above can be read as "where either X contains a number or X contains an identifier or both", assuming that the rules [containsANumber] and [containsAnIdentifier] check what their names imply.

A condition of form (c) succeeds if and only if all of the rules applied find a match to their pattern (or in the case of form (d), if and only if at least one of the rules applied does not find a match). The condition :

```
where all
    X [containsANumber] [containsAnIdentifier]
```

can be read as "where X contains a number and X contains an identifier".

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).

Several of the built-in functions (see 4.16, "Built-in Functions") 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 function that matches
    by                  %     iff the numeric value of N1 < N2
        N2 N1 Rest
end rule
```

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.

This kind of "trailing subsequence" pattern is a standard paradigm of TXL programming, and one worth remembering. It is interesting to consider what would happen if the trailing subsequence were not allowed in the pattern - that is, if the pattern of the sorting rule did not allow for *Rest*. In that case, the pattern would only be able to match a sequence of exactly length two - normally only the last pair of numbers in a sequence. The rule would sort only that pair and never match any other pair in the scope.

Assertions are identical to conditions except that they must succeed. The syntax of assertions is identical to conditions except that the the keyword *where* is replaced with *assert*. For example:

```
assert
     X [hasNoSubscript]
```

Assertions are used primarily in development, to check that assumptions under which the program was developed actually hold true during execution. If an assertion fails, the TXL run immediately ends with an error message.

4.11 Constructors

Try to play with all. However, at least read the highlighted part

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. Constructors that appear before the main pattern are evaluated once for each rule invocation; constructors appearing after the main pattern are re-evaluated for each pattern match. 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 special variable name '_' denotes the nameless anonymous variable, which can be constructed for the side effects of its replacement. Any number of anonymous variables may be constructed in a rule, but they cannot be referenced.

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.

As a shorthand, when the nonterminal type being constructed derives `[empty]` (for example if it is a `[`**`repeat`**`]` or `[`**`list`**`]` type), the constructed replacement can consist of a reference to the empty variable '_' and rules applied to it, with the semantics that it is a variable of the target type bound to an empty match. For example, the following constructor converts a list of numbers *NumberList* of type `[`**`list`**` number]` to a sequence of type `[`**`repeat`**` number]` by appending each element starting with an empty sequence:

```
construct NumberSequence [repeat number]
     _ [. each NumberList]
```

The empty variable '_' is commonly used in conjunction with the "extract" [^] built-in function. For example, given a variable *Proc* bound to a procedure definition, the following constructor makes a sequence containing a copy of every `[expression]` used in the procedure:

```
construct AllExpressionsInProc [repeat expression]
     _ [^ Proc]
```

The empty variable can also be used as the replacement for any predefined or user defined token nonterminal type, for example `[stringlit]` or `[id]`, with the semantics that it denotes an empty instance of the token, that is, an empty string for `[stringlit]` and `[charlit]` (that is, `""` and `''` respectively), and an empty token (that is, a token with no characters in it) for all other token types. For example, given bound variables *Formal*, *Actual* and *Type* of types `[id]`, `[expression]` and `[typedef]` respectively, the following constructor makes a `[stringlit]` by appending their text to the empty string into a message of the form `"size:int=x+5"`.

```
construct DebuggingString [stringlit]
     _ [unquote Formal] [+ ":"] [unquote Type] [+ "="] [unquote Actual]
```

When the target type allows, a constructed replacement can also be completely empty, denoted by the absence of any replacement at all. For example, the following constructor makes an empty sequence of numbers:

```
construct EmptySequence [repeat number]
     % none
```

It is considered good style to make intentionally empty replacements explicit using a comment (as shown above).

4.12 Global Variables, Import and Export

We haven't talked about these but great to know.

Rules can *import* and *export* global variables. Global variables are bound when a rule exports them, and their current value is accessed when a rule imports them. TXL uses a "bulletin board" model of global variables - rules "post" variable bindings to the bulletin board using *export*, and "read" variable values from the bulletin board using *import*. No explicit declaration of global variables is required - a global variable comes into existence when it is first exported from a rule, and retains its binding until a variable of the same name is exported from another rule. Import of a global variable is legal only if the variable has been previously exported by some rule.

The name of a global variable is visible in a rule only if the rule either imports or exports the variable. Inside the importing or exporting rule, the variable acts like a locally constructed variable. Global variable names not explicitly imported or exported from a rule can be used as local variable names independent of any global variable of the same name.

Like all TXL variables, global variables are strongly typed. The type of a global variable is permanently determined by the first export of the variable, and the type must be identical in every subsequent import and export of the variable.

Global variables are created and bound using an *export* clause. Inside the rule, an export acts exactly like a constructor, that is, it introduces a new local variable name and binds the constructed tree to it. Like constructors, exports may appear at any point before the replacement in a rule body. Export clauses take the form:

```
export 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 exported global variable is bound on each execution of the export clause to the newly constructed tree. Other rules, including subrules of the exporting rule, can import the global variable to access its new binding. The exported binding of the global variable remains on the bulletin board until another export clause for the same variable name is executed by this or some other rule.

Local variables, including imported variables, that have already been bound in a rule can also be exported. Because previously bound local variables already have a type, no type is given in this case. Export of a previously bound local variable takes the form:

```
export varName
     replacement
```

where *varName* is the name of the local variable and *replacement* has the same syntax as a main replacement. In this form the replacement is optional. If no replacement is given, then the variable is exported with its current binding. If a replacement is given, then the variable is bound to the newly constructed tree and exported with its new binding.

Access to global variables from other rules is provided by the *import* clause. An import clause acts like a constructor that introduces a new local variable of the global variable's name and binds it to the current binding of the global variable. Like constructors, imports may appear at any point before the replacement in a rule body. Import clauses take the form:

```
import varName [type]
       pattern
```

where *varName* is the name of a bound global variable, *type* is the nonterminal type of the variable, and *pattern* has the same syntax as a main pattern. The pattern is optional. If a pattern is given, the semantics are identical to that of a deconstruct of the imported variable (see "Deconstructors").

The imported global variable is re-evaluated to the current binding of the global variable at each execution of the import clause. A previously imported or exported variable may be (re-)imported later in the rule to explicitly force a re-evaluation. Because previously bound local variables already have a type, no type is given in this case. Import of a previously imported or exported variable takes the form:

```
import varName
      pattern
```

Imported variables may be used exactly like other local variables. In particular, they may be exported later in the rule, either with their original binding (which sets the global variable back to its binding at import time regardless of any intervening new bindings), or with an explicitly constructed new binding. If an imported variable is exported, no type is given in the export clause.

A global variable may imported or exported from the same rule any number of times, with the semantics that the global variable is bound at each execution of an export clause, and re-evaluated to the global's current binding at each execution of an import clause. The current binding of an imported global variable seen by a rule is affected only by the import and export clauses of that rule.

For example, if a subrule called in a constructor or replacement of a rule exports a new binding for a global variable previously imported into the rule, then the binding for that variable name seen by the rule remains unchanged until the next execution of an import or export clause for the variable in the rule. Thus the binding seen by a rule for any global variable can only be changed by the rule itself.

4.13 Working with Global Variables

It is really great to know these as well.

Global variables are a rich and powerful feature that can be used for many distinct purposes, including global tables, multiple results from a rule, "deep" parameters, and message-passing communication between rules in a rule set.

Global tables of information can be set up using an *export* clause before the *replace* clause in the main rule of a program. For example,

```
define table_entry
    [stringlit] '-> [stringlit]
end define

function main
    export Table [repeat table_entry]
        “Veggie” -> “Celery”
        “Fruit” -> “Apple”
        "Fruit" -> “Orange”
        “Fruit” -> “Pear”
        “Veggie” -> “Cabbage”
    replace  [program]
        P [program]
    by
        P [R1] [R2] [R3]
end function
```

Global tables created in this way can be accessed from any rule in the program using the following *import* clause before the *replace* clause of the rule:

```
import Table [repeat table_entry]
```

Global tables can be easily queried using searching deconstructors. For example, if we wish to know what kind of thing "Orange" is, after importing Table we can use the deconstructor:

```
deconstruct * [table_entry] Table
    Kind [stringlit] -> "Orange"
```

If Table had the original binding shown in the main rule above, then the binding for Kind will be the `[stringlit]` "Fruit". If no match were to be found, then the deconstructor would fail.

Global tables can be modified by exporting a new binding for the table based on the imported original binding. For example, the following function, when applied to a scope of type `[table_entry]`, adds the new entry to the global table of the example above:

```
function addTableEntry
    import Table [repeat table_entry]
    match [table_entry]
         NewEntry [table_entry]
    export Table
        Table [. NewEntry]
end function
```

Note however that modification of global tables is likely to be an expensive operation if the table is used extensively in the program, because the old value of the table must be preserved for previous imports of the variable that are still visible in other rules (whose value is not changed by the *export*!).

Multiple results from a rule can be achieved by exporting results other than the changed scope to global variables that are immediately imported by the calling rule. The additional result is exported immediately before the replacement of the called rule, so that it takes effect if and only if the rule succeeds in replacing its scope (i.e., if it returns a main result also). For example, the following simple rule returns all but the first element of a sequence of numbers as its main result, and the first element as its secondary result, in the global variable *HeadTailFirst* :

```
function headTail
    replace [repeat number]
        HeadTailFirst [number]
        HeadTailRest [repeat number]
    export HeadTailFirst     % additional result
    by
        HeadTailRest         % main result
end function
```

The calling rule should call from within a constructor, and then follow the constructor with an import of any additional results, for example :

```
construct RestX [repeat number]     % main result
    X [headTail]
import HeadTailFirst [number]       % additional result
```

Any number of results can be returned from a rule simply by exporting additional global variables.

Deep parameters allow the passing of parameters down to the bottom of a long chain of rule calls without explicitly handling the parameters in the intermediate rules. This clerical overhead of passing a parameter through all the intermediate levels of subrules until it finally gets down to the rule that actually needs the parameter can be annoying and error-prone. Using global variables, parameters can be passed directly from a rule to rules at levels of call far below its immediate subrules. The parent rule simply exports a global variable which is imported by the deep subrule that needs it.

For example, the following rule makes its entire original scope available to its deeply embedded subrule, even though the subrule itself actually works on only a tiny part of it :

```
function parentRule
    replace [repeat number]
         AllNumbers [repeat number]
    export AllNumbers
    by
         AllNumbers [transformEqualPairs]
end function

rule transformEqualPairs
    replace [repeat number]
         N1 [number] N2 [number] Rest [repeat number]
    deconstruct N1
         N2
    by
         N1 N2 [replaceByZeroIfEqual N1]
               [replaceByInfinityIfNotZero]
        Rest
end rule

function zeroIfEqual N [number]
    % replace the second number of an equal pair by zero
    replace [number]
         N
    % but only if the whole set of numbers doesn't have
    % a zero in it already
    import AllNumbers [repeat number]
    deconstruct not * [number] AllNumbers
         0
    by
         0
end function

function replaceByInfinityIfNotZero
    replace [number]
         N [number]
    deconstruct not N
         0
    by
         999999999
end function
```

<u>Communication between parent and subrules</u> : Global variables can be used to set up a message communication link between a parent rule and its subrules. In the following example, the subrule exports a flag to tell whether or not it actually did anything, indicating whether its result is meaningful:

```
rule SingleStepBubbleSort
    replace [repeat number]
         Numbers [repeat number]
    export Swapped [yesno]
         ‘no
    construct NewNumbers [repeat number]
        Numbers [SwapSomeMisorderedPair]
    import Swapped [yesno]
    deconstruct Swapped
        ‘yes
    by
        NewNumbers
end rule

function SwapSomeMisorderedPair
    replace [repeat number]
        N1 [number] N2 [number] Rest [repeat number]
    where
        N1 [> N2]
    export Swapped
        ‘yes
    by
        N2 N1 Rest
end function
```

Before calling the subrule, the parent rule exports the default value *no* for the global variable *Swapped*. The subrule then exports a new value *yes* for *Swapped* if and when it actually is able to make a replacement. In this way, the parent rule can efficiently check whether the subrule actually made a change without the computational overhead of comparing the original with the result. In the example above, the parent rule will halt when the subrule fails to find a replacement to make.

Communication between sibling subrules : Global variables can also be used to set up a message communication link between two or more subrules of a common parent rule. This can be useful when making sure that exactly one subrule of a set of subrules applies. For example, the following simple example rule has subrules that replace 1 by 2, and 2 by 3, without interference :

```
function shiftByOne
    replace [number]
        Number [number]
    by
        Number [replaceOneByTwo] [replaceTwoByThree]
end function

function replaceOneByTwo
    export Flag [id]
        'not_found
    replace [number]
        1
    export Flag
        'found
    by
        2
end function

function replaceTwoByThree
    import Flag [id]
```

```
        deconstruct Flag
            'not_found
        replace [number]
            2
        by
            3
    end function
```

The deconstruct of *Flag* in function [`replaceTwoByThree`] insures that it will not overwrite any work done by [`replaceOneByTwo`], even though its pattern matches the result of [`replaceOneByTwo`]. This is a trivial example of the much broader general problem of avoiding interference between subrules.

4.14 Limiting the Scope of Application

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 the statements in the body of a particular procedure, but not in any nested sub-procedures of 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 declaration_or_statement]
            S [statement]
            D [declaration]
            RestOfScope [repeat declaration_or_statement]
        by
            D
            S
            RestOfScope
    end rule
```

When applied to the code in the body of a procedure, this rule will sort the declarations before the 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 sub-procedures.

4.15 One-Pass Rules

Because the semantics of rule application requires that the rule is re-applied to the result of each replacement, certain kinds of rule can be difficult to write. For example, the following rule, which outwardly appears to be correct, will never terminate in TXL because every replacement it makes matches its original pattern, and it will go on replacing X with X forever.

```
rule anonymizeIdentifiers
    replace [id]
        OldId [id]
    by
        'X
end rule
```

The problem is that while TXL rules are in general intended to be applied recursively in order to find every match, including those created as part of the transformation, this kind of rule is intended to be applied to each matching item exactly once. In TXL we call this kind of rule a *one-pass* rule.

TXL has a special notation to specify that a rule should be one-pass. One-pass rules are specified by a $ following the keyword replace. For example, the rule above can be written as the one-pass rule:

```
rule anonymizeIdentifiers
    replace $ [id]
        OldId [id]
    by
        'X
end rule
```

The $ specifies that the rule is to be applied to each matching subtree of the original scope exactly once. This strategy is implemented by limiting the application of the rule to one pass of the scope tree in which the rule is applied only to the subtrees of each replacement, and not to its root. In this way each potential match is examined exactly once. Note that it is still possible to make a non-terminating one-pass rule, if the replacement of the rule always creates a new instance of the rule's pattern.

One-pass rules are often significantly faster than regular rules, and can be used in tuning TXL programs for speed (once they are debugged). However, when tuning in this way, care must be taken to insure that rules changed to be one-pass retain their originally intended semantics.

For example, the following rule, which reduces any sequence of repeated instances of same numbers in a sequence of numbers, will not behave correctly if changed to be one-pass, because the replacement may itself be a new instance of the pattern that we intend to be processed.

```
rule removeRepetitions
    replace [repeat number]
        N [number] N Rest [repeat number]
    by
        N Rest
end rule
```

4.16 Built-in Functions

Built-in functions provide a set of common operations that are difficult, awkward or inefficient to implement directly in TXL. TXL predefines the following built-in functions. For technical details and examples of the use of these functions, see the "Guide to TXL Built-In Functions".

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
N1 [div N2]	integer numeric quotient N1 / N2
N1 [rem N2]	integer numeric remainder N1 / N2
N1 [round]	round N1 to integer
N1 [trunc]	truncate N1 to integer
<u>Text Operations</u>	(T1, T2 must be of type [stringlit], [charlit], [id], [comment] or any other predefined or user token type, N1, N2 of type [number] or other numeric type)
T1 [+ T2]	concatenation of the text of T1 and T2
T1 [: N1 N2]	substring of the text of T1 from char N1 through char N2 inclusive (1-origin)
N1 [# T1]	length of text of T1
N1 [index T1 T2]	index of the first instance of the text of T2 in T1, or zero if none found
<u>Text Case Conversion</u>	(T1, T2 must be of type [stringlit], [charlit], [id], [comment] or any other predefined or user token type)
T1 [toupper]	replaces T1 with the text of T1 in upper case
T1 [tolower]	replaces T1 with the text of T1 in lower case
<u>Operations on Identifiers</u>	(ID1, ID2 must be of type [id] or any other identifier type)
ID1 [_ ID2]	identifier concatenation of ID1, underscore, and ID2
ID1 [!]	unique new identifier beginning with ID1, e.g. if ID1 is 'Bob', the result may be 'Bob27'
<u>Operations on Sequences</u>	(R1 must be of type [**repeat** T] for some type [T], R2 must be either of the same type [**repeat** T] or its element type [T], N1, N2 must be of type [number], and X1 can be of any type at all)
R1 [. R2]	replaces R1 with the sequence concatenation of R1 and R2
R1 [^ X1]	("extract") replaces R1 of type [**repeat** T] with a sequence consisting of every subtree of type [T] contained in X1
N1 [length R1]	replaces N1 with the length of the sequence R1
R1 [select N1 N2]	replaces R1 with the subsequence of R1 from element N1 through N2 inclusive (1-origin)
R1 [head N1]	replaces R1 with the subsequence of R1 from element 1 through N1 inclusive (1-origin)
R1 [tail N1]	replaces R1 with the subsequence of R1 from element N1 to the end of R1 inclusive (1-origin)
<u>Operations on Lists</u>	(L1 must be of type [**list** X] for some type [X], L2 must be either of the same type [**list** X] or its element type [X])
L1 [, L2]	replaces L1 with the list concatenation of L1 and L2
N1 [length L1]	replaces N1 with the length of the list L1

`L1 [select N1 N2]`	replaces L1 with the sublist of L1 from element N1 through N2 inclusive (1-origin)
`L1 [head N1]`	replaces L1 with the sublist of L1 from element 1 through N1 inclusive (1-origin)
`L1 [tail N1]`	replaces L1 with the sublist of L1 from element N1 to the end of L1 inclusive (1-origin)
Numeric Comparisons	(N1, N2 must both be of type [number] or other numeric type)
`N1 [= N2]`	succeeds if N1 is numerically equal to N2
`N1 [~= N2]`	succeeds if N1 is numerically not equal to N2
`N1 [> N2]`	succeeds if N1 is numerically greater than N2
`N1 [>= N2]`	succeeds if N1 is numerically greater than or equal to N2
`N1 [< N2]`	succeeds if N1 is numerically less than N2
`N1 [<= N2]`	succeeds if N1 is numerically less than or equal to N2
Text Comparisons	(T1, T2 must be of type [stringlit], [charlit], [id], [comment] or any other predefined or user token type)
`T1 [= T2]`	succeeds if the text of T1 is identical to the text of T2
`T1 [~= T2]`	succeeds if the text of T1 is not identical to the text of T2
`T1 [> T2]`	succeeds if the text of T1 is alphanumerically greater than the text of T2
`T1 [>= T2]`	succeeds if the text of T1 is alphanumerically >= the text of T2
`T1 [< T2]`	succeeds if the text of T1 is alphanumerically less than the text of T2
`T1 [<= T2]`	succeeds if the text of T1 is alphanumerically <= the text of T2
General Comparisons	(X1, X2 must both be of the same type, which is not a built-in or user token type)
`X1 [= X2]`	succeeds if X1 is treewise alphanumerically identical to X2
`X1 [~= X2]`	succeeds if X1 is not treewise alphanumerically identical to X2
Substring Search	(T1, T2 must both be of type [stringlit], [charlit], [id] or any other identifier or user token type)
`T1 [grep T2]`	succeeds if the text of T2 is a substring of the text of T1
Fast Ground Substitute	(Y1, Y2 must both be of any same type, X1 can be any other type)
`X1 [$ Y1 Y2]`	result is X1 with Y2 substituted for every occurrence of Y1 (i.e., a fast one-pass substitution of Y2 for Y1)
Type Conversions	(T1, T2 must be of type [stringlit], [charlit], [id], [comment] or any other predefined or user token type, S1 must be of type [stringlit] or [charlit], and X1, X2 can be any type at all)
`T1 [quote X1]`	appends the output text of X1 to the text of T1 (same as [unparse])
`T1 [unquote S1]`	replaces the text of T1 with the unquoted text of S1
`X1 [parse T1]`	replaces X1 of any type [T] with a parse of the text of T1 as a [T]

`T1 [unparse X1]` appends the output text of X1 to the text of T1 (same as [quote])

`X1 [reparse X2]` replaces X1 of any type [T] with a parse of the leaves (terminal symbols) of X2 as a [T]

Operations on Types (polymorphism operations for use with [any] - ID1 must be of type [id] or any other identifier type, X1 of any type at all)

`ID1 [typeof X1]` replaces ID1 with the nonterminal type name of the type of X1

`X1 [istype ID1]` succeeds if the nonterminal type name of X1 is ID1

Input/Output Operations (S1 must be of type [stringlit], [charlit], [id] or any other identifier type, F1 must be of type [stringlit] or [charlit], X1 can be any type)

`X1 [read S1]` replaces X1 of any type [T] with a parse of the text file S1 as a [T]

`X1 [write S1]` writes the output text of X1 as the text file S1 (which is opened or created if necessary, written and closed)

`X1 [get]` inputs one line of text from the standard input and replaces X1 of any type [T] with a parse of the input text as a [T]

`X1 [fget F1]` inputs one line of text from file F1 (which is opened if necessary) and replaces X1 of any type [T] with a parse of the input text as a [T]

`X1 [getp S1]` outputs the text of S1 as a prompt on the standard error stream, then inputs one line of text from the standard input and replaces X1 of any type [T] with a parse of the input text as a [T]

`X1 [put]` appends the text of X1 to the standard error stream

`X1 [fput F1]` appends the text of X1 to file F1, which is opened or created if necessary

`X1 [putp S1]` appends the text of S1 to the standard error stream, with the output text of X1 inserted at the point of the first "%" character in S1

`X1 [fputp F1 S1]` appends the text of S1 to file F1, with the output text of X1 inserted at the point of the first "%" character in S1

`S1 [gets]` replaces S1 with one line of raw input text from the standard input

`S1 [fgets F1]` replaces S1 with one line of raw input text from file F1 (which is opened if necessary)

`S1 [puts]` outputs the raw text of S1 as a line to the standard error stream

`S1 [fputs F1]` outputs the raw text of S1 as a line to file F1, which is opened or created if necessary

`X1 [fclose F1]` closes file F1 - only necessary if the file is to be reopened

`X1 [fopen F1 M1]` opens file F1 in the indicated mode M1, which must be one of "get", "put" or "append" - only necessary if the file is to be opened "append"

Debugging Aids (X1, X2 can be any type at all)

`X1 [message X2]` outputs the output text of X2 to the standard error stream;
if X2 is of type [stringlit] or [charlit], it is unquoted

`X1 [print]` outputs the output text of X1 to the standard error stream;
if X1 is of type [stringlit] or [charlit], it is unquoted

`X1 [printattr]` outputs the output text of X1 to the standard error stream with
attributes visible; if X1 is of type [stringlit] or [charlit], it is unquoted

`X1 [debug]` outputs the internal tree format of X1 to the standard error stream

`X1 [breakpoint]` temporarily halts execution until a carriage return character is
received from the standard input

Execution Control (S1 must be of type [stringlit] or [charlit], N1 must be of type [number]
or other numeric type, X1 can be any type)

`X1 [pragma S1]` interpret the TXL command line options specified in S1
to dynamically change TXL options during execution

`X1 [quit N1]` immediately abort TXL execution with return code N1

System Interface (S1, S2 must both be of type [stringlit] or [charlit],
X1 can be any type at all)

`X1 [system S1]` invoke /bin/sh to run the text of S1 as a shell command line;
X1 is ignored and unchanged
(most useful when used in conjunction with [write] and [read]
to manipulate data in files)

`S1 [pipe S2]` invoke /bin/sh to run the shell command
echo "*text of S1*" | *text of S2*
and replace the text of S1 with the first line of the result

4.17 External Functions

Great to know

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

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

Where *ruleName* is an identifier, and *parameterList* is a list of typed identifiers (as in a function definition). The choice of whether an external function is declared to be a function or a rule 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.

External functions are implemented by the user by modifying the file 'user.c' and re-linking TXL; see the comments in the file 'user.c' for details. See also the built-in functions [system] and [pipe] in the section "Built-in Functions" above. External functions are available only in compiled TXL programs, and are not supported in all implementations of TXL.

4.18 Condition Rules Try to understand what is going on.

Rules and functions 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 allowed.

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 [type_spec]
         RestOfScope [repeat statement]
    where not
         RestOfScope [references X]
    by
         RestOfScope
end rule

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

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.

Condition rules can also be used in place of transformation rules in order to achieve a useful side-effect. For example, the following debugging function can be called in any context to print a debugging message if the identifier "OOPS" appears anywhere in its scope:

```
function checkForOOPS
    match * [id]
        'OOPS
    construct PrintMessageDummy [id]
        _ [message "Found an OOPS!"]
          [breakpoint]
end function
```

4.19 Complex Conditions

Arbitrary Boolean conditions can be built up using combinations of *where* clauses. The *or* of a set of conditions is represented by the semantics of the *where* clause itself, which succeeds if any of the sequence of condition rules applied succeeds, and the *and* of a set of conditions can be represented using the *where all* form, which succeeds only if all 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 can be represented either by using the *where all* form:

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

or perhaps more elegantly by using 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* as above, we can also 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 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
```

4.20 Polymorphic Rules

Great to know.

Rules and functions can implement a limited form of universal polymorphism using the universal nonterminal type [any]. The nonterminal type [any] has some obvious properties, and some not-so-obvious ones. The main thing to know is that [any] matches any tree at all, so the pattern:

```
match [any]
     Any [any]
```

will match any parse tree of any type at all. The matched tree retains its own nonterminal type and structure, even though it has been matched using [any]. A TXL variable *X* of any particular type [T] can be recast as an [any] using the deconstruct:

```
deconstruct X
     AnyX [any]
```

This deconstruct never fails, no matter what *X* is, and always matches all of *X*, that is, the tree bound to *AnyX* is exactly the same as the tree bound to *X*.

Because a tree matched by [any] retains its internal structure, we can still use deep deconstructs to match internal parts of that structure. For example, if we want to constrain ourselves to only those matches of trees that contain a subtree of type [T], we can write:

```
match [any]
     Any [any]
deconstruct * [T] Any
     X [T]
```

The deconstruct will succeed if and only if the tree matched by the [any] pattern is itself a [T] or contains a [T] embedded somewhere in it. The deconstruct must be a searching deconstruct, that is, it must have a '*', because the type [any] provides no fixed grammatical structure by which TXL can parse the pattern of the deconstruct. For this reason, a deconstruct cannot be used to constrain the matched [any] to be entirely a tree of a certain type [T].

However, the built-in functions [istype] and [typeof] allow for explicit testing of the type of a tree originally matched as [any]. For example, to match any tree and then constrain it to be either a [declaration] or a [statement], we can write:

```
match [any]
     Any [any]
where
     Any [istype 'declaration] [istype 'statement]
```

We can also discover the type of a tree matched as [any] using [typeof], and then deconstruct the type name itself to see if it is what we want. For example, if we're interested only in [declaration] trees, we can use this:

```
match [any]
     Any [any]
construct TypeOfAny [id]
     _ [typeof Any]
deconstruct TypeOfAny
     'declaration
```

Because there is no fixed grammatical definition of the structure of the type [any], it is also not possible to construct a variable of type [any] directly. Thus the construct statement:

```
construct A [any]
    replacement
```

is always an error, unless the *replacement* consists exactly of a reference to a single bound variable already of type [any]. Instead, the paradigm for constructing arbitrary trees to be used as type [any] is the following:

```
construct B [T]
    replacement
deconstruct B
    AnyB [any]
```

Because the deconstruct cannot fail no matter what the type and structure of *B*, this has the effect of allowing us to construct any tree at all as an [any].

This construction is most useful when making a *replace* rule or function of type [any]. Because TXL constrains us to use a replacement of the same nonterminal type as the pattern, the type of the replacement in a rule or function whose pattern is of type [any] must also be [any]. Thus the usual paradigm for such a rule or function is:

```
function replaceAnyTree
    replace [any]
        AnyTree [any]
    % Constraints on the matched tree go here
    construct NewTree [newtreetype]
        % The intended replacement tree goes here
    deconstruct NewTree
        AnyNewTree [any]
    by
        AnyNewTree
end function
```

"Replace" rules and functions of type [any] are very dangerous, and should be used with care.

4.21 Working with Polymorphism

Rules using [any] can (obviously) lead to unexpected results. Since the trees resulting from a replace rule or function targeting type [any] need not be structured according to the language grammar, the patterns of subsequent rules, functions and deconstructs may fail unexpectedly on the resulting trees. This is because TXL parses all patterns according to the language grammar at compile time, creating pattern trees structured according to the grammar. A tree not structured in exactly the same way, even if its terminal contents (leaves) are the same, cannot match the pattern tree.

This phenomenon can happen in TXL even without use of [any], if the program exploits ambiguity in the grammar by using explicit constructs to create trees that use alternative (second or subsequent alternative) parses. However, this technique is relatively uncommon and in general is done intentionally to *prevent* subsequent pattern matches. In the case of replace rules and functions of type [any], it is much more likely to happen unintentionally and lead to unexpected pattern match failures and thus unexpected results.

For this reason, it is recommended that polymorphic replace rules and functions be avoided unless absolutely necessary. They are simply too error prone for common practice.

On the other hand, replace rules and functions of type [any] can be very powerful, and can conveniently achieve some otherwise awkward or difficult transforms when used with discipline. Condition rules and functions of type [any] are never dangerous, and can be very useful as well. In this section we explore some paradigms for the disciplined use of rules and functions of type [any] that are very powerful and relatively benign.

Generic property testing functions : The safest useful thing that can be done with [any] is to write generic property testing functions, avoiding the necessity of writing many functions of the same shape. The following example tests for containment of any object of any type in any other object of any type:

```
function contains Object [any]
    match * [any]
        Object
end function
```

On the face of it, this function is much like a searching deconstruct. It's only when used with *each* that we really see its usefulness:

```
% Does this declaration contain any of these attributes?
construct InterestingAttributes [repeat attribute]
    'int  'int2  'int4  'nat  'nat2  'nat4  'real  'real8
where
    Declaration [contains each InterestingAttributes]

% Does this procedure use any of these built-in functions?
construct StringBuiltinIdentifiers [repeat id]
    'substr  'length  'index
where
    Procedure [contains each StringBuiltinIdentifiers]
```

Rule set abstraction : It's also safe to use [any] to write abstraction functions that gather a set of rules that are to be applied together to the scope, when the scope can vary in type. Again, this avoids the necessity of writing multiple functions of the exact same shape for different scope types.

```
function applyMyRules
    replace [any]
        Scope [any]
    by
        Scope
            [myRule1]
            [myRule2]
            [myRule3]
            [myRule4]
end function
```

Of course, if this abstraction is to be effective, each of the subrules must be a rule or searching function.

These first two paradigms are always safe because they both preserve grammatical structure. The next paradigm of use is not quite so restrictive, because it does violate grammatical structure, but it preserves behaviour for most rule sets.

Nested factoring : The basic idea of this paradigm is that the changes we make are constrained to include the unchanged original structure inside the changed structure when modifying the tree. By constraining ourselves in this way, we insure that all searching functions, rules and searching deconstructs with patterns that used to target the original structure will continue to do so, hopefully minimizing the effect of the [any] cheat on the rest of the program.

The best example of this technique is generic markup. In generic markup, we define a generic markup type that can mark up any nonterminal, and then use it to do all our markups, preserving the original item inside the markup at all times so that it can be found by other markup rules and functions.

```
define markup
    % The [any] here allows us to mark up anything at all
    '< [id] '>  [any]  '</ [id] '>
end define
```

Markup rules can then use this type to mark up anything at all, saving us the chore of changing the grammar every time we choose to make a markup rule for a new nonterminal.

```
function markupWith Tag [id]
    replace [any]
        Any [any]
    % Make a marked-up copy of the scope ...
    construct Markup [markup]
        '< Tag '>  Any '</ Tag '>
    % ... and recast it as an [any] so that it can replace it.
    deconstruct Markup
        MarkupAny [any]
    by
        MarkupAny
end function
```

If tomorrow we decide we need to mark up [statement]'s, then we can do so without changing either the grammar, the overrides, or the [markupWith] rule:

```
rule markupStatementsMentioning Ids [repeat id] Markup [id]
    % Don't re-mark anything we have already marked up
    skipping [markup]

    % Find every statement once
    replace $ [statement]
        Stmt [statement]

    % Mark up the statement only if it uses at least one of the Id's
    % (We re-use the generic [contains] function defined above)
    where
        Stmt [contains each Ids]
    by
        Stmt [markupWith Markup]
end rule
```

Because we always preserve the entire tree structure of the marked-up [statement] intact in the result, other rules in the program will in general not be affected by this use of [any].

5. The Unparsing Phase

The unparsing phase is the simplest of the three phases of TXL. 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 all following output lines by four (more) spaces

`[EX]` exdent all following output lines by four (fewer) spaces

`[IN_`*`NN`*`]` indent all following output lines by *NN* (more) spaces

`[EX_`*`NN`*`]` exdent all following output lines by *NN* (fewer) spaces

As an example, the following 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 procedure_declaration
    procedure [id] [formal_parameter_list] ;    [NL][IN]
         [declarations]                         [EX]
    begin                                       [NL][IN]
         [statements]                           [EX]
    'end [id]
end define
```

Grammars can also contain explicit tabbing using the built-in formatting nonterminal [TAB], which explicitly directs the formatting of output as follows:

`[TAB]` start the next output item at the next ANSI standard tab stop -
standard tab stops are set in intervals of 8 columns starting at column 1

`[TAB_`*`NN`*`]` start the next output item in column *NN* by adding spaces -
if the current output column is already >= *NN*, then output continues
in column *NN* of the next line, unless the *–notabnl* command line option
is given, in which case exactly one space is added to the current line

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 *–raw* command line option. When *–raw* is given as a command line option, no spacing at all is done in the output unless it is explicitly specified in the grammar using the built-in nonterminal [SP], or the [TAB] directives shown above.

`[SP]` append a space to the output

`[SP_`*`NN`*`]` append *NN* spaces to the output

For example, the following definition, when used with *-raw*, specifies that the output is to have a spaces around '+' operators but none around '*' operators in the output.

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

[SP] can also be used when the *–raw* option is not on, to add explicit additional spacing to output.

When character-level input is specified using the *–char* command line option, *–raw* is on by default. If two adjacent identifiers appear in *–raw* output, then they are separated by a space to distinguish them, unless *–char* is on, in which case they are concatenated instead.

Local spacing of output can also be explicitly controlled by the TXL program using the built-in nonterminals [SPOFF] and [SPON], which dynamically turn default output spacing off and on respectively during output.   For example, default spacing would normally put spaces around < and >, which is not appropriate for XML tags, so we can temporarily suspend default spacing in tags like so:

```
define XML_tag
    < [SPOFF] [id] > [SPON]         % e.g., <x>
        [repeat content]
    < [SPOFF] / [id] > [SPON]       % e.g., </x>
end define
```

5.2 Attributes

TXL also allows a grammar definition to have attributes associated with it. Attributes act like optional nonterminals, but normally do not appear in the output. Attributes are not required in the input, but may be added to the parse tree during transformation. Attributes are denoted in the grammar by the nonterminal modifier *attr*, which can be added to any nonterminal type.

For example, the following definition of *typed_id* describes an identifier with a type as an attribute. The type can be *int* or *string*, with untyped identifiers represented by an empty attribute.

```
define type
    'int | 'string
end define

define typed_id
    [id] [attr type]    % [attr] means that the [type] is an optional
                        % attribute
end define
```

This could be used in a situation where the type of an identifier needs to be inferred from its associated value expression, as in the following example:

```
function inferTypeFrom Expn [expression]
    replace [typed_id]
        Id [id]
    deconstruct Expn
        % checks to see whether the expression is a
        % binary operation acting on two numbers
        First [number] Op [binary_operator] Second [number]
    by
        Id 'int
end function
```

Subsequent rules can then deconstruct the [typed_id] to get its inferred type attribute. In all other respects the type [typed_id] acts exactly like [id], including being output as the [id] without the attribute.

Attributes can be of arbitrarily complex nonterminal types containing any amount of information. Thus transforms requiring the collection and reuse of complex attributes such as those often stored in symbol tables can be encoded directly in the parse tree.

Attributes can be forced to print in the output, for example for debugging purposes, using the *–attr* command line option. When *–attr* is given as a command line option, all attributes are printed in the output. See also [printattr] in the section "Built-in Functions".

6. TXL Programs

A TXL program combines a set of nonterminal type definitions with a set of rules and functions. The nonterminal types are defined using *define*, *keys*, *compounds*, *comments* and *tokens* statements, and the rules are defined using *rule*, *function* and *external* rule statements. In general 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 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 *main*, which is automatically applied to the parse tree of the input.

6.1 Comments

TXL end-of-line 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 (e.g. '%) each time it appears to distinguish it from the comment marker.

TXL also supports bracketing comments, which begin with `%(` and end with `)%`, to allow for long comments and commenting out of rules or blocks of code. For example:

```
%( All of the text
   in these lines, including this comment and program text:

        % Create Id
        construct Id [id]
            'foo

   is commented out. )%
```

The comment brackets `%{` and `}%` are also accepted.

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 string literal containing the (possibly system-dependent) name of the file to be included. By default, file names without directory specifiers are assumed to be in the same directory as the TXL source file containing the include statement. If the file is not found there, then TXL uses an include file search path to find the file. By default, the include file search path is: the present working directory (i.e., the directory from which the TXL command was run), the *Txl* subdirectory of the present working directory (if one exists), and the TXL library directory, configured when the TXL processor is installed on each system. Additional directories can be prepended to the include file search path using the *–I* command line option (see the "User's Guide to the TXL Compiler/Interpreter" for further information).

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.

6.3 Preprocessor Directives

The TXL preprocessor allows conditional compilation of TXL programs depending on parameters specified using the *–d* command line option or explicit **#define** preprocessor statements. The TXL preprocessor uses Turing (Modula-3 style) preprocessor directives. For compatibility with C and other Unix tools, TXL also accepts an equivalent subset of the ANSI C preprocessor directives.

The complete set of Turing (Modula-3 style) preprocessor directives is accepted. Because of its similarity to the syntax of TXL itself, this is the preferred form for TXL preprocessor directives.

The following directives form the Turing preprocessor command set. In the following, items enclosed in square brackets [ ] indicate optional items and items enclosed in brace brackets { } are items that may be repeated zero or more times.

```
      #pragma   { command line flags }

      #define   SYMBOL
      #undefine SYMBOL

      #if [not] SYMBOL1 [then]
           source lines
   {  #elsif [not] SYMBOLn [then]
           source lines            }
   [  #else
           source lines            ]
      #end [if]
```

For compatibility with other Unix and C tools, the following equivalent subset of the ANSI C standard preprocessor directives is also accepted.

```
      #pragma   { command line flags }

      #define  SYMBOL
      #undef[ine] SYMBOL

      #if[n][def] SYMBOL1
           source lines
   {  #elif[n][def] SYMBOLn
           source lines        }
   [  #else
           source lines        ]
      #endif
```

Note that these commands include all of those generated by the Unix *diff –D* command, so different versions of the same TXL program can be automatically integrated and conditionally compiled from the same source using *diff –D* .

Any mix of ANSI C and Turing-style preprocessor syntax is accepted by the TXL preprocessor, although due to readability concerns, such mixing is discouraged as bad style.

TXL preprocessor statements are interpreted line-wise, which means that the # character must be the first non-blank character on the line (otherwise the line is treated as a regular TXL source line), and any trailing characters on the same line as the preprocessor directive are ignored as comments.

For example, the following line does not contain a preprocessor directive :

```
'[ Oldid [id] #end if Jimbo Jambo ']
```

The following preprocessor line is legal and does contain a preprocessor directive :

```
#end if JIMBO @#$%!(*& JAMBO or #define MUMBO JUMBO
```

This line is exactly equivalent to :

```
#end if
```

Any number of spaces may appear between the leading # and the preprocessor command, and between items in a preprocessor command. Thus the preprocessor directive :

```
#     define   FOO
```

is exactly equivalent to :

```
#define FOO
```

As in Turing and C, preprocessor symbols are by convention all upper case, but are not required to be so. Any TXL identifier can be used as a TXL preprocessor symbol.

In general, the interpretation of TXL preprocessor directives is intended to be identical to the interpretation of the corresponding directives in ANSI C and Turing. Preprocessor directives are interpreted line-by-line from beginning to end of the TXL source file, with **include** statements interpreted as if replaced by the source lines in the included file. Non-preprocessor source lines (those that do not begin with a # as the first non-space character) are passed on unchanged to the TXL compiler except as noted below.

```
#pragma { command line flags }
```

The #pragma directive allows command line flags and their arguments to be specified from within the TXL program itself. The effect of the flags is exactly as if they had been specified on the command line.

```
#define   SYMBOL
#undefine SYMBOL
#undef    SYMBOL
```

TXL preprocessor symbols can only be *defined* or *undefined*, but cannot be associated with any value. Preprocessor symbols are by default undefined unless explicitly defined in one of three ways :

1. Explicit definition using the #**define** preprocessor directive, for example :

```
#define SYMBOL
```

2. Command line definition using the *–d* command line option, for example :

```
    txl -d SYMBOL myinput.input mytxlprogram.Txl
```

3. Pragma definition using the #**pragma** preprocessor directive :

```
    #pragma -d SYMBOL
```

Once defined, preprocessor symbols remain defined until explicitly undefined using the #**undef** or #**undefine** preprocessor directives, or until end of the compilation, whichever comes first. Certain symbols (e.g., MAC, UNIX, AIX, TXLDB, etc.) may be predefined to allow conditional compilation based on target machine, debugger or other environmental factors.

```
    #if [not] SYMBOL1 [then]
         source lines
{   #elsif [not] SYMBOLn [then]
         source lines              }
[   #else
         source lines              ]
    #end [if]

    #if[n][def] SYMBOL1
         source lines
{   #elif[n][def] SYMBOLn
         source lines      }
[   #else
         source lines      ]
    #endif
```

These directives allow conditional inclusion of source lines in the compilation depending on whether or not certain preprocessor symbols are currently defined. In each case, the source lines following the first **if**/**elif**/**elsif** that holds are interpreted if and only if the specified symbol is defined (or is not defined, if **n** or **not** is specified in the **if**/**elif**/**elsif**), and the source lines following all the other **if**/**elif**/**elsif**/**else** 's are discarded.

If none of the **if**/**elif**/**elsif** 's hold and **else** is specified, then the source lines following the **else** are interpreted, otherwise they are discarded. If none of the **if**/**elif**/**elsif** 's hold and no **else** is specified, then all of the nested source lines are discarded.

#**if** and any other preprocessor directives may be nested in the conditionally included source lines. These directives are interpreted if and only if interpretation of the #**if** directive results in the interpretation of the group of source lines in which they appear, otherwise they are discarded with the group.

6.4 Predefined Global Variables

Great to know.

When TXL programs are run, certain global variables are pre-initialized to import information from the environment of the run. In particular, the TXL global variable *TXLargs* of type [**repeat** stringlit] is predefined in all implementations of TXL to import the command line arguments to the TXL program. Command line arguments to the program are specified using a single "-" to separate them from TXL's own arguments. For example, if the command line to run the program is:

```
    txl -s 20 myinput.input mytxlprogram.Txl - -myoption 2 -myotheroption
```

Then *TXLargs* will automatically be preinitialized to:

```
export TXLargs [repeat stringlit]
    "-myoption" "2" "-myotheroption"
```

The program can import *TXLargs* to test its command line arguments, for example:

```
import TXLargs [repeat stringlit]
deconstruct * TXLargs
    "-myoption" Value [stringlit] MoreOptions [repeat stringlit]
deconstruct * [stringlit] TXLargs
    "-myothereoption"
```

While individual platforms may vary in their predefined variables, the following TXL global variables are preinitialized in all TXL implementations.

`TXLargs [repeat stringlit]`	TXL program arguments (see above)
`TXLprogram [stringlit]`	file name of the TXL program being run
`TXLinput [stringlit]`	file name of the main input to the TXL program
`TXLexitcode [number]`	exit code for the TXL run - can be exported to change the exit code to be returned at the end of the run

For additional details and examples using predefined variables and command line arguments in TXL programs, see the "User's Guide to the TXL Compiler/Interpreter".

6.5 Version Compatibility

For compatibility with previous versions of TXL, some obsolete forms not documented in this manual continue to be accepted by the TXL processor. In particular, `'+'` is accepted in place of `'...|'` in token definitions.

Appendix A. Formal Syntax of TXL

We use the TXL grammar notation to formally specify the syntax of TXL. The precise definition of the predefined nonterminals `[id]`, `[number]`, `[stringlit]`, `[charlit]`, `[token]` and `[key]` used below is given in section 3.2, "Predefined Nonterminal Types".

<u>Note</u>: This grammar does not include the syntax of preprocessor directives - see "Preprocessor Directives" in section 6 for the syntax of preprocessor statements.

```
% TXL comments begin with a % character and end at the end of the line.
% The % character must be quoted if it appears as a terminal symbol
% in a TXL program (as it does here)
comments
     '%
end comments

% The following are keywords of TXL and must be quoted if they appear
% as terminal symbols in a TXL program (as they do here)
keys
     'all 'assert 'attr 'by 'comments 'compounds 'construct 'deconstruct
     'define 'each 'end 'export 'external 'function 'import 'include
     'keys 'list 'match 'not 'opt 'push 'pop 'redefine 'repeat 'replace
     'rule 'see 'skipping 'tokens 'where
end keys

% A TXL program consists of a sequence of TXL statements
define program
     [repeat statement]
end define

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

define includeStatement
     'include [stringlit]          % string literal gives 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 (to end of line)
     |   [literal] [literal]        % start / end symbol pair
end define

define tokensStatement
     'tokens
         [repeat tokenPattern]      % one pattern per line
     'end 'tokens
end define

define tokenPattern
         [typeid] [stringlit]       % new token type, or override of
                                    %     an existing token type
     |         '| [stringlit]       % extension of the immediately
                                    %     preceding token type
     |   [typeid] '... '| [stringlit] % extension of an existing
                                    %     token type
     |   [typeid] '+ [stringlit]    % extension of an existing
                                    %     token type (deprecated)
end define

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

define redefineStatement
     'redefine [typeid]
         [opt dotDotDotBar]         % postextension of existing define
         [repeat literalOrType]
         [repeat barLiteralsAndTypes]
         [opt barDotDotDot]         % preextension of existing define
     'end 'redefine
end define
```

```
define dotDotDotBar
     '... '|
end define

define barDotDotDot
     '| '...
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] ']
     |   '[ 'attr [typeidOrQuotedLiteral] ']
     |   '[ 'see [typeidOrQuotedLiteral] ']
     |   '[ 'not [typeidOrQuotedLiteral] ']
     |   '[ 'push [typeidOrQuotedLiteral] ']
     |   '[ 'pop [typeidOrQuotedLiteral] ']
     |   '[ '! ']

     % Alternative short forms for the above
     |   '[ [typeidOrQuotedLiteral] '? ']       % opt
     |   '[ [typeidOrQuotedLiteral] '* ']       % repeat
     |   '[ [typeidOrQuotedLiteral] '+ ']       % repeat +
     |   '[ [typeidOrQuotedLiteral] ', ']       % list
     |   '[ [typeidOrQuotedLiteral] ', '+ ']    % list +
     |   '[ ': [typeidOrQuotedLiteral] ']       % see
     |   '[ '~ [typeidOrQuotedLiteral] ']       % not
     |   '[ '> [typeidOrQuotedLiteral] ']       % push
     |   '[ '< [typeidOrQuotedLiteral] ']       % pop
end define

define plusOrStar
     '+ | '*
end define

define typeidOrQuotedLiteral
         [typeid]
     |   [quotedLiteral]
end define

define ruleStatement
         'rule [ruleid] [repeat formalArgument]
             [repeat constructDeconstructImportExportOrCondition]
             [opt skippingType]
```

```
              'replace [opt '$] [type]
                  [pattern]
              [repeat constructDeconstructImportExportOrCondition]
              'by
                  [replacement]
         'end 'rule

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

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

     |   'function [ruleid] [repeat formalArgument]
              [repeat constructDeconstructImportExportOrCondition]
              [opt skippingType]
              'match [opt '*] [type]
                  [pattern]
              [repeat constructDeconstructImportExportOrCondition]
         '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 constructDeconstructImportExportOrCondition
         [constructor]
     |   [deconstructor]
     |   [condition]
     |   [import]
     |   [export]
end define
```

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

define deconstructor
     [opt skippingType]
     'deconstruct [opt 'not] [opt '*] [opt type] [varid]
         [pattern]
end define

define condition
         'where [opt 'not] [opt 'all]
             [expression]
     |   'assert [opt 'not] [opt 'all]
             [expression]
end define

define import
     'import [varid] [opt type]
         [opt pattern]
end define

define export
     'export [varid] [opt type]
         [opt replacement]
end define

define skippingType
     'skipping [type]
end define

define pattern
     [repeat literalOrVariable]
end define

define literalOrVariable
         [literal]
     |   [varid] [type]       % bind of a new variable
     |   [varid]              % use of a previously bound variable
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 varidOrLiteral]
         [opt 'each] [repeat varidOrLiteral] ']
end define

define varidOrLiteral
     [varid] | [literal]
end define

define literal
     [quotedLiteral] | [unquotedLiteral]
end define

define quotedLiteral
     ' ' [unquotedLiteral]    % note: ' ' means a single quote mark
end define

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

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

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

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

define ruleid
     [id]                     % identifier that is a rule/function name
end define
```

Appendix B - Detailed semantics of opt, repeat, list and attr

The nonterminal modifiers *opt*, *repeat*, *list* and *attr* are implemented by generating internal nonterminal definitions corresponding to the meaning of the modifiers, as shown in the following sections. Note that the internal nonterminal type names given in the defines below are examples only and may or may not be present in any particular implementation of TXL. Use of these internal type names in TXL programs is never necessary and not recommended.

B.1 Implementation of [opt] and [attr]

The nonterminal modifier `[opt X]` generates the define statement:

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

Every reference to `[opt X]` is implemented as a reference to `[opt__X]`.
The implementation of `[attr X]` is similar, using internal type name `[attr__X]`.

B.2 Implementation of [repeat]

The nonterminal modifier `[repeat X]` generates the define statements:

```
define repeat_0_X
        [repeat_1_X]
    |   [empty]
end define

define repeat_1_X
    [X] [repeat_0_X]
end define
```

Every reference to `[repeat X]` is implemented as a reference to `[repeat_0_X]` .
`[repeat X+]` generates the same set of defines but uses `[repeat_1_X]` as the implementation.

A tree of type `[repeat X+]` always matches a pattern requiring `[repeat X]`, and a nonempty tree of type `[repeat X]` always matches a pattern requiring `[repeat X+]`.

B.3 Implementation of [list]

The nonterminal modifier `[list X]` generates the define statements:

```
define list_0_X
        [list_1_X]
    |   [empty]
end define
```

```
define list_1_X
    [X] [list_0_X]
end define
```

Every reference to the nonterminal `[list X]` is implemented as a reference to `[list_0_X]`. `[list X+]` generates the same set of defines but uses `[list_1_X]` as the implementation.

A tree of type `[list X+]` always matches a pattern requiring `[list X]`, and a nonempty tree of type `[list X]` always matches a pattern requiring `[list X+]`.

The separating commas of the list are not explicitly represented in the implementation of `[list X]`. This saves space and speeds up pattern matching, but does not change the requirement for explicit commas in input lists and list patterns.

The implementation of lists is such that the pattern:

```
First [X] , Rest [list X]
```

matches any nonempty list, including the singleton list (even though it does not have a comma).
This property simplifies working with lists and avoids the need to have separate rules for singleton lists.