Guide to TXL Built-in Functions

Version 10.5
November 2007

James R. Cordy

James R. Cordy

Guide to TXL Built-in Functions Version 10.5

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. Built-in Functions in TXL 4
2. Arithmetic Operations 4
    2.1 Real Arithmetic [+], [-], [*], [/] 4
    2.2 Integer Arithmetic [div], [mod] 6
    2.3 Numeric Rounding [round], [trunc] 7
3. Text Operations 7
    3.1 Concatenation, Substring and Length [+], [:], [#] 7
    3.2 Substring Search [index], [grep] 9
    3.3 Text Case Conversion [tolower], [toupper] 10
4. Operations on Identifiers 11
    4.1 Identifier Concatenation [+], [_] 11
    4.2 Unique Identifier Generation [!] 11
5. Operations on Sequences and Lists 12
    5.1 Sequence and List Concatenation [.], [,] 12
    5.2 Subsequence Selection [length], [select], [head], [tail] 12
    5.3 Type Extraction to a Sequence [^] 15
6. Comparison Operations 17
    6.1 Numeric Comparisons [=], [~=], [>], [>=], [<], [<=] 17
    6.2 Text Comparisons [=], [~=], [>], [>=], [<], [<=] 18
    6.3 General Comparisons [=], [~=] 18
7. Fast Ground Substitute [$] 19
8. Type Conversion 19
    8.1 Conversion to Text [quote], [unquote] 19
    8.2 Dynamic Parsing [parse], [reparse], [unparse] 21
    8.3 Generic programming [typeof], [istype] 23
9. Programmed Input/Output 25
    9.1 File-oriented Input/Output [read], [write] 25
    9.2 Interactive Input/Output [get]/[fget], [put]/[fput], [getp], [putp]/[fputp], [gets]/[fgets], [puts]/[fputs], [fopen], [fclose] 27
10. Debugging Aids 30
    10.1 Programmed Messages [message], [print], [printattr] 31
    10.2 Interactive Debugging [debug], [breakpoint] 32
11. Execution Control 34
    11.1 Dynamic Run Options [pragma] 34
    11.2 Programmed Abort [quit] 34
12. System Interface 35
    12.1 Execute System Command [system], [pipe] 35

1. Built-in Functions in TXL

TXL built-in functions provide a set of common operations that are difficult, awkward, or inefficient to implement directly in TXL. These consist of the arithmetic, text, and identifier operations and comparisons, list and sequence operations, interactive and file input/output, system access and debugging operations.

The TXL built-in functions are built in to the TXL engine and are automatically available in all TXL programs. They need not be imported or included explicitly. With the exception of *[system]* and *[pipe],* which are platform dependent, all built-in functions are available in all implementations of TXL.

2. Arithmetic Operations

The arithmetic operations allow for numerical computation in TXL. While TXL is not designed or recommended for numerical applications, many analyses and transformations appropriate to TXL require some numerical computation. However, TXL is a symbolic computation system and all numeric tokens are internally represented by their text, not their value. These operations convert the text to numeric values, perform the operations numerically, and convert the result back to text as a new token. Thus TXL is not suitable for implementing programs that are primarily numerical in nature.

All arithmetic operations are performed in double precision floating point arithmetic, although the result is represented in simplest terms, so *X [+ 1]* where X is 1 yields a result of 2, not 2.0e00. The range of numerical computation in a TXL implementation is determined by double precision floating point representation of the host computer. For example, on the x86 series (e.g., Pentium, Core 2 Duo), the range of values is approximately -1.8e+308 to +1.8e+308 with an exponent range of -308 to +308.

2.1 Real Arithmetic

The basic arithmetic operations are [+] (addition), [-] (subtraction), [*] (multiplication), and [/] (division). The operators all take a scope and one argument of type [number] (or equivalently, any other numeric type) and yield a number of the same type as result.

Although the operation is performed in double precision floating point, the result is always represented in simplest form - that is, as an integer if the result is an integer (e.g., 2, not 2.0e+00 and 35000, not 3.5e+04), and as a rounded decimal number if the result has a fractional part (e.g., 37.5, not 3.749999999e+01). Results less than or equal to -1,000,000 or greater than or equal to +1,000,000 are represented in floating point notation, that is, 1.0e+06, not 1000000.

For example, if A, B and C are [number]s bound to the values A=1, B=2.5 and C=3 respectively, then:

```
construct Bsquared [number]
    B [* B]                             % yields 6.25
construct FourAC [number]
    A [* C] [* 4]                       % yields 12
construct BsquaredMinusFourAC [number]
    Bsquared [- FourAC]                 % yields -5.75
```

Note that the constructor *FourAC* cannot be written as *4 [* A] [* C],* since TXL functions must be applied to a bound variable and cannot be applied directly to a terminal token as scope. If the order of computation were important and the 4 must be first, then we would have to construct a variable with the value 4:

```
    construct Four [number]
         4
    construct FourAC [number]
         Four [* A] [* C]              % still yields 12
```

In the examples above we have used traditional TXL format in the function applications, separating the argument from the function name (e.g., *A [* C]*). However, because the arithmetic functions use operator symbols rather than function names, argument spacing is redundant and we can also write things like

```
    construct CsquaredMinus1 [number]
         C [*C][-1]                    % yields 8
```

While TXL is not designed for numerical computation, when necessary it is possible to write most numerical computations (albeit in a wordy form). As an example, the function *[factorial]* below takes a number N as its scope, and recursively calculates N! (read 'N-factorial'), where N! = N * (N-1) * (N-2)...* 2 * 1, for N > 0.

```
    function factorial
         replace [number]
              N [number]
         where
              N [> 1]
         construct NminusOneFactorial [number]
              N [- 1] [factorial]  % subtract 1, then recursively
                                   % apply the function [factorial]
         by
              N [* NminusOneFactorial]
    end function
```

This function works by taking a number N, then checking to ensure that the N is strictly greater than 1. It then proceeds to calculate the value of (N-1)! recursively. This result is then multiplied to the original value of N, to yield the result. (Note that this function should really check for the 0! case (0! = 1), but this case has been left out for simplicity.)

An issue with numerical computation in TXL is the creation of new tokens for each result. Since TXL is s symbolic manipulation languages, each result is converted to a text token of type [number] and entered into the TXL hash table. If an excessive amount of numerical computation is carried out, the hash table can quickly fill to exceed TXL limits. Thus TXL is not well suited as a numerical computation language, and the arithmetic functions should only be used as necessary in support of a source transformation.

The four real arithmetic operations are summarized below:

<u>Real Arithmetic Operations</u> (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]`	real numeric quotient N1 / N2

2.2 Integer Arithmetic

The arithmetic operations [div] (integer division) and [rem] (integer remainder) take a scope and argument of type [number] (or equivalently, any other numeric type) and yield an integer result of the same type.

For example, if N is a [number] bound to the value 10, then

```
construct Nslash3 [number]
    N [/ 3]                 % yields 3.333333333333, but
construct Ndiv3 [number]
    N [div 3]               % yields 3, and
construct Nmod3 [number]
    N [rem 3]               % yields 1
```

As demonstrated by the *[factorial]* function in Section 2.1, the real arithmetic [+], [-], and [*] operations all yield integer results for integer operands. However, while the results look and act as integers, the operations are in fact implemented in double precision floating point , and if their result values become large (i.e., smaller than -1,000,000 or larger than 1,000,000), they will change to exponent notation. For example, if N is bound to the value 10, then

```
construct Amillion [number]
    N [* 100000]            % yields 1.0e6, not 1000000
construct ALittleLess [number]
    N [* 99999]             % yields 999990, not 9.9999e5
```

Normally this difference is undetectable unless the number is output. However, because TXL is a symbolic language and the symbol '1.0e6' is not identical to the symbol '1000000', the two will not match in a pattern. So for example,

```
deconstruct Amillion
    1000000
```

will fail, whereas using numerical comparison

```
where
    Amillion [= 1000000]
```

will succeed because the comparison is carried out in double precision floating point and 1,000,000 can be represented exactly in that form.

The entire set of integer arithmetic operations is summarized below:

<u>Integer Arithmetic Operations</u> (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 [div N2]`	integer quotient N1 / N2
`N1 [rem N2]`	integer remainder N1 / N2

2.3 Numeric Rounding

The arithmetic operations [round] and [trunc] provide explicit conversion of real results to integer. [round] rounds its scope [number] to the nearest integer, with ties rounding up (e.g. rounding 1.25 down to 1, 1.75 up to 2, and 1.5 up to 2) and [trunc] truncates the fractional part (e.g., truncating 1.95 to 1).

For example, if we wish to compute the percent of total statements that are if statements, given *Nstatements* and *NifStatements*, both of type [number}, then

```
construct PercentIfStatements [number]
    NifStatements [/ Nstatements] [* 100] [round]
```

Numeric Rounding Operations (N1 must be of type [number] or any other numeric type)

`N1 [round]`	rounded integer value of real number N1
`N1 [trunc]`	truncated integer part of real number N1

3. Text Operations

The text operations provide for direct manipulation of the text of tokens of any predefined or user token type. Text operations act directly on the text of the token, and are not constrained to valid results for the token's type, so for example *X [+ ".y"]* where X is of type [id] and bound to the identifier *foo*, will yield an identifier with the text *foo.y* even if *foo.y* is not a valid input identifier in the target language.

When applied to tokens of type [stringlit] or [charlit], token text is converted to unquoted evaluated form before the operation and back to quoted form afterwards. In unquoted form all escaped characters are converted to their output representation (e.g, if the language escape convention is *-esc '\'*, then "\n" becomes one newline character in the text), and any special characters in the quoted result are automatically escaped (e.g., a newline character in the quoted result becomes "\n").

Token types can be mixed arbitrarily using the text operations - for example, an [id] can be concatenated to a [stringlit], and a [number] can be concatenated to a [charlit]. The [+] operation is overloaded, meaning arithmetic addition when its scope is a [number] or other numeric type, and text concatenation otherwise.

3.1 Concatenation, Substring and Length

The basic text operations are concatenation [+], substring [:], and length [#].

The concatenation operation [+] is used to append the text of one token onto the end of another. When used on scopes of type [stringlit] or [charlit], the string quotes are stripped from the text before the operation and added to the result afterward.

For example, if S is a [stringlit] variable bound to the string *"Hello there"*, Id is an [id] bound to the identifier *John*, and N is a [number] bound to the value 4, then:

```
construct Greeting [stringlit]
      S [+ " "] [+ Id] [+ "!"]       % yields "Hello there John!"
construct StrangeId [id]
      Id [+ ", "] [+ S]              % yields 'John, Hello there'
                                     % as the text of an [id]
```

```
        construct Fact [stringlit]
              _ [+ Id] [+ " has "] [+ N] [+ " siblings"]
                                           % yields "John has 4 siblings"
```

The text of tokens of different types may be combined using [+] to create, for example, meaningful error messages from a TXL program. For example, given *VarId* and *ProcId* of type [id], the following outputs a message of the form *"*** Error: XXX has not been defined in PPP"* on the standard error stream:

```
        construct Message [stringlit]
              _ [+ "*** Error: '"] [+ VarId]
                [+ "' has not been defined in'"] [+ ProcId] [+ "'"]
                [print]
```

Concatenations are commonly coded beginning with the anonymous variable '_' as shown above to denote beginning with an empty text. This has the advantage that we do not need a bound TXL variable of the same type to seed the sequence.

Tokens of any kind can be manipulated, for example, if the *-comment* flag is used to make *[comment]* a valid token, then we can create comments in our output. The following creates a C comment of the form *"/* XXX was here */".*

```
        construct Comment [comment]
              _ [+ "/* "] [+ VarId] [+ " was here */"]
```

The substring operation [:] takes the form S1 [: N1 N2]. It is used to make a substring of the text of any token bound to S1 and change it into the substring of S1 from character N1 through N2 inclusive (where N1, N2 are numbers N1 < N2). Character positions in TXL are indexed starting from 1.

The substring built-in function normally expects N1 and N2 to be within the bounds of the string. If N1 < 1 or N2 is greater than the length of the string, TXL will truncate them to fit by setting N1 < 1 to 1 and N2 > length of string to the length of the string. If N2 < N1, then the result is the empty string.

For example, if *S* is a TXL variable of type [stringlit] bound to the string *"crystalline vase"* , then

```
        construct Glass [stringlit]
              S [: 1 7]              % yields "crystal"
        construct Flagon [stringlit]
              S [: 13 999]           % yields "vase"
        construct Oops [stringlit]
              S [: 3 1]              % yields "" (empty string)
```

The construct of *Flagon* above demonstrates a common trick when using the substring function in TXL - using a large number (conventionally 999) as the second argument to mean "to end of string".

The length function {#] returns the length of the text of a token or string. The scope of the operation must be a number, with a single token or string as a argument. Using the length function, a cleaner and more general implementation of the construct of *Flagon* might be:

```
        construct LenS [number]
              _ [# S]                % yields 16
        construct Flagon [stringlit]
              S [: 13 LenS]          % still yields "vase"
```

This table summarizes the basic token text operations:

Basic Text Operations	(S1 must be of type [stringlit] or any other non-numeric token type, S2 of type [stringlit] or any other token type, N1 of type [number])
`S1 [+ S2]`	concatenation of S1 and S2
`S1 [: N1 N2]`	substring of S1 from char N1 through char N2 inclusive (1-origin)
`N1 [# S2]`	length of S2

3.2 Substring Search

The text operations [index] and [grep] provide the ability to search within the text of a string or other token's text representation. Like other token text built-in functions, these can work on the text of [stringlit], [charlit] and any other token type.

The [index] function is used to replace the scope (which must be of type [number] or other numeric type) with the 1-origin index of the first occurrence of the second argument (of type [stringlit] or any other token type) within the first (also of type [stringlit] or any other token type). If no occurrence is found then the function returns 0.

Most frequently the [index] function is used on with newly constructed number (denoted by the anonymous variable _) as scope. For example, the following function takes a paragraph in English, parsed as a single [stringlt], and returns the first sentence of the paragraph.

```
function getFirstSentence
    replace [stringlit]
        Paragraph [stringlit]

    % Find the first period in the paragraph,
    % which is where the first sentence ends
    construct DotIndex [number]
        _ [index Paragraph "."]

    % Make sure we actually found one
    deconstruct not DotIndex
        0

    % Take from the first character to the first
    % period in the paragraph as result
    by
        Paragraph [: 1 DotIndex]
end function
```

The [grep] function performs a similar text search but simply returns success or failure depending on whether an occurrence of the second argument's text is found in the first. [grep] is a condition function, intended to be used as part of a TXL *where* clause, for example:

```
function warnIfFriendMentioned
    % Given an email message text as a string,
    % print a warning message if one of my friends is mentioned

    construct Friends [repeat id]
        'Jim 'Jane 'John 'Judy
    % Given an email message
    match [stringlit]
        EmailMessage [stringlit]
    % See if any of my friends is mentioned in it
    where
        EmailMessage [grep each Friends]
    % If so print a message
    construct _ [id]
        _ [message "Message mentions a friend!"]
end function
```

Summary of text searching operations:

Text Search Operations	(S1 must be of type [stringlit] or any other non-numeric token type, S2 of type [stringlit] or any other token type, N1 of type [number])
`N1 [index S1 S2]`	1-origin index of text of S1 in text of S2, or 0 if not found
`S1 [grep S2]`	true if text of S2 is a substring of text of S1

3.3 Text Case Conversion

The text operations [tolower] and [toupper] convert the alphabetic characters of the text of any token to lower case or upper case respectively. Like all text functions, they can be applied to a scope of type [stringlit], [charlit] or any other token type. For example, to normalize an HTML tag to standard lower case form, we might have the following:

```
    function NormalizeTags
          replace [html_tag]
                < Tagid [id] TagAttributes [repeat html_attribute] >
                      TagBody [repeat html_element]
                </ TagId >
          by
                < Tagid [tolower] TagAttributes >
                      TagBody
                </ TagId [tolower] >
    end function
```

Summary of text case conversion operations:

Text Case Conversion Operations	(S1 must be of type [stringlit] or any other token type)
`S1 [tolower]`	lower case version of S1
`S1 [toupper]`	upper case version of S1

4. Operations on Identifiers

As TXL is most often used in conjunction with other programming languages, special operations for identifiers have also been included among the built-in functions. These include two different forms of identifier concatenation (that is, building a new identifier name out of two other identifiers), as well as a function to take an identifier and create a unique new identifier from it.

4.1 Identifier Concatenation

Identifier concatenation [+] uses the token text concatenation operator described in Section 3 with identifiers as scope and argument. For example, given *ProcId* and *VarId* both of type [id], we can concatenate them with an underscore:

```
% make a globally unique identifier for a local variable
construct GlobalId [id]
      ProcId [+ "_"] [+ VarId]      % given X and Y, yields X_Y
```

Because concatenating identifiers using underscores is a common transformation, the [_] built-in function is provided as a shorthand:

```
construct GlobalId [id]
ProcId [_ VarId]        % given X and Y, yields X_Y
```

4.1 Unique Identifier Generation

Some transformations require the introduction of new identifiers in the transformed result. The unique identifier function [!] assists in this. Given an identifier *Id* of type [id] as scope, [!] creates a unique new identifier using *Id* as the base. For example, if *BaseId* is a TXL variable of type [id] bound to the identifier *Gollum*, then:

```
construct UniqueId [id]
      BaseId [!]        % yields a globally unique new identifier
                        % of the form Gollum234
```

The base identifier need not come from the program, it can be specified as part of the constructor. In this way a transformation can generate meaningful identifiers for its result.

```
construct ObjectId [id]
      _ [+ "GenObject"] [!]   % yields unique new identifier
                              % of the form GenObject123
```

The following is a summary of the identifier functions and how they are used. Note that all of the token text operations of section 3 can also be used with identifiers as well.

Operations on Identifiers	(ID1, ID2 must be of type [id] or any other identifier type)
`ID1 [+ ID2]`	identifier concatenation of ID1 and ID2
`ID1 [_ ID2]`	identifier concatenation of ID1, underscore, and ID2
`ID1 [!]`	unique new identifier beginning with ID1

5. Operations on Sequences and Lists

These operations are used to create and manipulate trees of type [**repeat** X] or [**list** X] for any type [X]. the operations are analogous to the token text operations, but for sequences of arbitrary type. The result of each function is always a new sequence or list, but optimized to minimize element copying.

5.1 Sequence and List Concatenation

The sequence splice function [.] is used to splice an object of type [**repeat** X] for any type [X] to either an object of type [X] or an object of type [**repeat** X]. It is analogous to the string concatenation function [+] but for sequences rather than text.

Suppose, for example, that we wanted to design a function that constructs a sequence of even numbers: 2 4 6 8 ... 100. We could do this in the following way:

```
function makeEvens PreviousNum [number]
      replace [repeat number]
            SequenceSoFar [number]
      where
            PreviousNum [< 100]
      construct ThisNum [number]
            PreviousNum [+2]
      by
            SequenceSoFar [. ThisNum]
                          [buildSequence ThisNum]
end function
```

This function can be used in a **construct** beginning with an empty sequence (denoted by the anonymous variable _) like this:

```
construct NumberSequence [repeat number]
      _ [buildSequence 0]     % yields 2 4 6 8 10 ... 100
```

The list splice function [,] is the analogous function for lists, used to splice objects of [**list** X] to objects of type [X] or [**list** X] for any type [X]. It works by taking the first list, adding a comma, then concatenating the second list or element to it, yielding a concatenated new list.

For example, if *FirstArgs* and *LastArgs* are both TXL variables of type [**list** argument] bound to the lists *a,b,c* and *1,2,3* respectively, and *Arg* is a TXL variable of type [argument] bound to the argument *f(x)*, then:

```
construct NewArgs [list argument]
      FirstArgs [, Arg] [, LastArgs]   % yields a,b,c,f(x),1,2,3
```

As in the case of other special symbol built-in functions, spacing is optional.

5.2 Subsequence Selection

Several built-in functions are provided for efficient manipulation of subsequences in indexed array-like fashion. While any of these can be implemented recursively in TXL itself, the built-in functions are optimized to be significantly more efficient for large sequences.

The [length] function is analogous to the string length operator [#]. It takes as argument a tree of type [**repeat** X] or [**list** X] for any type X, and, when applied to a scope of type [number], returns the number of elements in the argument repeat. Frequently the scope of the function is a newly constructed number (denoted by _ in a **construct**).

In this example, the [length] built-in function is used to check that the length of an argument list matches the length of a formal parameter list.

```
function checkArgumentListLength
        FormalParameters [list formal_parameter]
    match [list argument_expn]
        CallArguments [list argument_expn]

    % Get the lengths of the formal and actuals lists
    construct LengthFormals [number]
        _ [length FormalParameters]
    construct LengthArguments [number]
        _ [length CallArguments]

    % If they are not the same, print an error message
    deconstruct not LengthFormals
        LengthActuals
    construct Message [stringlit]
        _ [message "Error: wrong number of arguments"]
end function
```

The subsequence built-in function [select] and analogous to the substring operation [:]. When applied to a scope of type [**repeat** X] or [**list** X] for any type [X] and two arguments of type [number] N1 and N2, it returns the subsequence or sublist of the scope from position N1 through position N2 inclusive. As always in TXL, positions are 1-origin.

```
% Get the next five statements in the sequence Statements
construct NextFiveStatements [repeat statement]
    Statements [select 1 5]

% Or the last five
construct NumberOfStatementsMinus4 [number]
    _ [length Statements] [- 4]
construct LastFiveStatements [repeat statement]
    Statements [select NumberOfStatementsMinus4 99999]
```

The construct of *LastFiveStatements* above demonstrates a common trick in TXL - using a large number (conventionally 99999) as the second argument to mean "to end of the sequence". A more elegant way to do this is to use the [tail] function - see below.

One way that the [select] function can be used is to treat a sequence [**repeat** X] as an array of type [X]. For example:

```
function subscript Array [repeat element] Index [number]
    % Result is optional
    replace [opt element]
        % none

    % Get the subsequence of length 1 at Index
    construct ElementSubsequence [repeat element]
        Array [select Index Index]
```

```
          % Get the element in the subsequence
          deconstruct ElementSubsequence
               Element [element]
          by
               Element
     end function
```

Given a sequence such as *Squares* of type [**repeat** element] (where in this case [element] is defined as [number], perhaps initialized to the squares of the numbers 1 to N for some N) and an index of type [number], this function might be used something like this:

```
     % Look up the square of 17
     construct SquaresSub17 [opt element]
          _ [subscript Squares 17]
     deconstruct SquaresSub17
          Square17 [number]
```

The [select] function normally expects positions N1 and N2 to be within the bounds of the sequence or list. If N1 < 1 or N2 is greater than the length of the sequence or list, TXL will truncate them to fit by setting N1 < 1 to 1 and N2 > length of sequence / list to the length of the sequence / list. If N2 < N1, then the result is the empty sequence / list.

This total behavior of the [select] function means that the array indexing function above really should check that the index is within the bounds of the "array". We can check that using the [length] function:

```
     function safeSubscript Array [repeat element] Index [number]
          % Result is optional
          replace [opt element]
               % none
          % Get the upper bound of the "array"
          construct UpperBound [number]
               _ [length Array]

          % Check Index is in bounds using TXL assertions
          assert
               Index [>= 1]
          assert
               Index [<= UpperBound]

          % Get the subsequence of length 1 at Index
          construct ElementSubsequence [repeat element]
               Array [select Index Index]
          % Get the element in the subsequence
          deconstruct ElementSubsequence
               Element [element]
          by
               Element
     end function
```

The [head] and [tail] built-in functions are specializations of [select] that can be used partition a scope of type [**repeat** X] or [**list** X], for any type [X.] [head N]selects the first N elements of the sequence or list, and [tail N] selects the elements from element N to the last element of the sequence or list.

We can use [head] and [tail] to make a more elegant version of our previous example taking the first and last five statements in a sequence:

```
% Get the next five statements in the sequence Statements
construct NextFiveStatements [repeat statement]
     Statements [head 5]
% Or the last five
construct NumberOfStatementsMinus4 [number]
     _ [length Statements] [- 4]
construct LastFiveStatements [repeat statement]
     Statements [tail NumberOfStatementsMinus4]
```

In this example, given a position N, we return a result that deletes the Nth element from the middle of a sequence.

```
function deleteElement N [number]
     replace [repeat id]
          Ids [repeat id]
     construct NminusOne [number]
          N [- 1]
     construct NplusOne [number]
          N [+ 1]
     construct IdsBeforeElement [repeat id]
          % Take identifiers in positions 1 through N-1
          Ids [head Nminus1]
     construct IdsAfterElement [repeat id]
          % Take identifiers in positions N+1 through to the end
          Ids [tail Nplus1]
     by
          % Stick them together
          IdsBeforeElement [. IdsAfterElement]
end function
```

5.3 Type Extraction to a Sequence

The element extract function [^] is used to create a [**repeat** X] sequence of all occurrences of items of type [X] in the tree bound to a TXL variable (which can be any type at all). The type [X] to be extracted is specified by the element type of the scope of the function application, which must be of type [**repeat** X] for some type [X]. The first argument of the function application specifies the tree from which elements of type [X] are to be extracted.

The elements of the extracted sequence appear in the same left-right order as they appear in the original tree, with repetitions preserved. If the grammar for type [X] is recursive (i.e., allows embedded [X]'s) then embedded [X]'s are not included in the sequence (thus each subtree of type [X] in the original appears exactly once in the result).

This function is normally used in a new **construct** beginning with an empty sequence (denoted by the anonymous variable _). For example, suppose that we wanted to extract all the identifiers from a program, and display them on the screen. We would have a TXL program that looks like this:

```
function main
    replace [program]
        P [program]
    construct AllIds [repeat id]
        _ [^ P] [print]      % extract all identifiers and print them
    by
        P
end function
```

The extract function has a wide range of applications, and can be used effectively in combination with agile parsing to mark and collect interesting items in a program, as in this example, which identifies and prints out all calls to JDBC methods in a Java program:

```
% Use agile parsing to identify JDBC calls in Java
include "Java.Grm"

redefine method_call
    [jdbc_call] [NL]   % [NL] so calls output on separate lines
   | ...
end redefine

define jdbc_call
    [jdbc_name] [arguments]
end define

define jdbc_name
    'createStatement | 'prepareStatement
   | 'executeUpdate | 'executeQuery | 'getRow
end define

function main
    match [program]
        P [program]

    % Collect and output all the JDBC calls anywhere in the program
    construct JDBCaspect [repeat jdbc_call]
        _ [^ P] [print]
end function
```

The following is a summary of the built-in functions for sequences and lists:

Operations on Sequences and Lists	(R1 must be of type [**repeat** T] or [**list** T] for some type [T], R2 must be either of the same type [**repeat** T] or [**list** T] or its element type [T], N1, N2 must be of type [number], and X1 can be of any type at all)
`R1 [. R2]`	sequence / list concatenation of R1 and R2
`R1 [^ X1]`	("extract") where R1 of type [**repeat** T], returns a sequence consisting of every subtree of type [T] contained in X1
`N1 [length R1]`	length of the sequence R1
`R1 [select N1 N2]`	subsequence of R1 from element N1 through N2 inclusive (1-origin)

`R1 [head N1]`	subsequence of R1 from element 1 through N1 inclusive (1-origin)
`R1 [tail N1]`	subsequence of R1 from element N1 to the end of R1 inclusive (1-origin)

6. Comparison Operations

TXL provides comparison built-in functions to allow for order-dependent transformation of numeric and alphanumeric text data.

The inequality operators [>], [>=], [<] and [<=] compare two arguments of the same type, which can be type [number] or any other numeric type (for arithmetic ordering), or of type [stringlit], [charlit] or [id] (for alphanumeric ordering). The equality operators [=] and [~=] can be used with any two arguments of the same type. Comparison operators can only be used in a **where** or **where not** clause to guard a replacement.

6.1 Numeric Comparisons

When used with numeric arguments, the comparison built-in functions [=], [~=], [>], [>=], [<], [<=] convert to double precision floating point representation for the comparison, for example, if X and Y are of type [number] bound to the values 2.54 and 5.75 respectively, then:

```
where
     X [< Y]              % succeeds since 2.54 < 5.75
where
     X [>= Y]             % fails since 2.54 is not >= 5.75
where
     X [+ 1] [> X]        % succeeds since 3.54 > 2.54
where not
     X [* 2] [< Y]        % fails since (2.54 * 2) = 5.08 is < 5.75,
                          % but the condition is negated
```

The inequality operators can be used to order items in a sequence or table, for example, the following rule, when applied to a TXL variable bound to a sequence of type [**repeat** number], will sort it into ascending order:

```
rule bubbleSortNumbers
     replace [repeat number]
          N1 [number] N2 [number] Rest [repeat number]
     where
          N1 [> N2]
     by
          N2 N1 Rest
end rule
```

(Note that if the rule above had no *Rest* in its pattern, it would match only a subsequence of length 2, and therefore would only sort the last two numbers in a sequence.)

The equality comparisons [=], [~=] are defined for any two items of the same type. If the compared type is [number] or any other numeric type, then the comparison is done by double precision floating point value. For non-numeric types, the semantics of [=] is identical to that of a **deconstruct** (which is the preferred and more efficient form). For numeric types, the two are different, for example, if N1 is a [number] bound to 0, and N2 is a [number] bound to 00, then

```
deconstruct N1
    N2           % fails since 0 is not identically 00
where
    N1 [= N2]    % succeeds since the numeric value of both
                 % 0 and 00 is 0
```

The distinction between a [number] token and its value can be even more clearly seen in the case of -0, which is distinct from 0 when bound to a TXL [number] variable, but has the same value.

6.2 Text Comparisons

When applied to operands of type [stringlit], [charlit] or [id], the inequality operators [>], [>=], [<] and [<=] use ASCII alphanumeric (dictionary) text order, with longer strings being greater when the rest is equal. For example, if *F* is a [stringlit] bound to *"the fox"*, *C* is a [stringlit] bound to *"the cat"*, and *FS* is a [stringlit] bound to *"the foxes"*, then

```
where
    C [>= F]     % fails since "the cat" < "the fox"
where
    F [< FS]     % succeeds since "the fox" < "the foxes"
```

For [stringlit], [charlit] and [id] operands, the semantics of [=] and [~=] is identical to **deconstruct** and **deconstruct not**, which are the preferred forms. Thus the following are equivalent:

```
where
    S1 [= S2]    % S1 identical to S2?
deconstruct S1
    S2           % S1 identically S2?
```

6.3 General Comparisons

The equality operators [=] and [~=] are defined for any two operands of the same type. When applied to non-numeric operands, the semantics of [=] and [~=] are identical to **deconstruct** and **deconstruct not**, which are the preferred forms.

Following is a summary of the comparison operators:

Inequality Comparisons	(X1, X2 must both be of type [stringlit], [charlit] or [id], or X1, X2 must both of type [number] or other numeric type)
`X1 [> X2]`	succeeds if X1 > X2
`X1 [>= X2]`	succeeds if X1 >= X2
`X1 [< X2]`	succeeds if X1 < X2
`X1 [<= X2]`	succeeds if X1 <= X2

Equality Comparisons (N1,N2 must both be of type [number] or other numeric type, X1, X2 must both be of the same non-numeric type)

`N1 [= N2]`	succeeds if the value of N1 is equal to the value of N2
`N1 [~= N2]`	succeeds if the value of N1 is not equal to the value of N2
`X1 [= X2]`	succeeds if X1 is identical to X2 (same as **deconstruct**)
`X1 [~= X2]`	succeeds if X1 is not identical to X2 (same as **deconstruct not**)

7. Fast Ground Substitute

The fast ground substitute [$] function is simply a generic shorthand for a very common TXL operation - replace all occurrences of an item in a scope by another of the same type, for example, all occurrences of one identifier in a program by different identifier. [$] takes a scope of any kind, and two arguments, both of the same type. It returns a copy of the scope with all occurrences of the first argument replaced by the second argument.

For example, suppose we want to rename the formal parameter of a monadic function to include the function name in its name. We could write a TXL function to do this like this:

```
function renameParameterWithFunctionName
    replace [function_declaration]
        FunctionName [id] ( ParameterName [id] )
        FunctionBody [block]
    construct NewParameterName [id]
        FunctionName [_ ParameterName]   % e.g., f_p
    by
        FunctionName ( NewParameterName )
        FunctionBody [$ ParameterName NewParameterName]
end function
```

<u>Fast Ground Substitute</u>	(Y1, Y2 must both be of the same type, X1 can be any other type)
`X1 [$ Y1 Y2]`	result is X1 with Y2 substituted for every occurrence of Y1

8. Type Conversion

TXL rules and functions are constrained to be type-preserving so that they can produce only well-formed results according to the grammar. However, at times it is necessary to manipulate input in a non-structural or untyped way to achieve the desired result. To support this possibility, TXL provides predefined functions to assist in disciplined dynamic type conversion, both to and from text and between types. Using these functions, rules can dynamically invoke the TXL parser and unparser as part of the transformation.

8.1 Conversion to Text

Conversion to and from text form is supported by the [quote] and [unquote] predefined functions, which take an item of any type to and from [stringlit] or [charlit] of its text respectively. The length of the text is limited by the TXL [stringlit] / [charlit] implementation maximum, typically 32768 characters. See the [read] and [write] functions for handling larger conversions using temporary files.

The [quote] function takes a argument of any type, and turns it into either a [stringlit] or a [charlit], depending on the scope type. It concatenates the text of the argument to the scope string, so that text of several items can be concatenated easily, for example when making an error message. Often [quote] is used in a *construct* statement with the empty variable (_) as scope, in which case the new string is the quoted version of the argument.

[quote] is very convenient n the creation of customized conditions and error messages, for example, if *Expn* is of type [expression], bound to the expression *f(x)+y(z)/3* :

```
% Outputs a message of the form:
%    *** Error, expression 'f(x) + y(z) / 3' may have a side effect
construct Message [stringlit]
     _ [+ "*** Error, expression '"] [quote Expn]
       [+ "' may have a side effect"] [print]
```

[quote] is particularly useful in transformations to self-documenting programs. For example, the following function turns a C statement into a self-tracing version of itself :

```
function traceStatement
     replace [statement]
          Stmt [statement]
     construct QuoteStmt [stringlit]
          _ [quote Stmt]
     by
          {
               fprintf (stderr, ">>> %s\n",  QuoteStmt);
               Stmt
          }
end function
```

[quote] can be used in conjunction with the [parse] function to conveniently convert between types. See [parse] below.

The [unquote] function takes a argument of type [stringlit] or [charlit], and turns it into an identifier or any other token type (essentially just removing the quotes). Note that the text of the argument does not necessarily have to be a legal identifier or instance of the token type. For example, if *S* is a [stringlit] bound to the string *"hi there"*, then :

```
construct FunnyId [id]
     _ [unquote S]       % makes the strange identifier 'hi there'
construct FunnyAssignment [statement]
     FunnyId := 5;       % makes the strange statement 'hi there := 5;'
```

Use of [unquote] is now deprecated, since the generalization of [+] makes it equivalent to :

```
construct FunnyId [id]
     _ [+ S]             % makes the strange identifier 'hi there'
```

The following summarizes the text conversion functions.

Text Conversion Operations	(S1 must be of type [stringlit] or [charlit], T1 can be of type [id] or any other token type, and X1 can be of any type at all)
`S1 [quote X1]`	appends the output text of X1 to the text of S1 (same as [unparse])
`T1 [unquote S1]`	replaces the text of T1 with the unquoted text of S1

8.2 Dynamic Parsing

Some transformations need the ability to view the same source text from different perspectives, using different grammars or subgrammars, or by forcing a different parse than the default for some instance of an input form. Others may benefit from the ability to parse text that is constructed as part of the transformation. The ability to invoke the parser as part of a transformation is called *dynamic parsing*, and is supported by the TXL predefined functions [parse], [reparse] and [unparse].

The [parse] function takes an arbitrary [stringlit], [charlit], [id] or any other token type and attempts to scan and parse it as the scope type as if it were input. For example, suppose that the TXL variable *Text* is of type [stringlit] and bound to the string *"int Counter;"* (the text of an integer variable declaration in C). If the grammar has type [declaration] for C variable declarations, then the string could be parsed like so:

```
construct Decl [declaration]
     _ [parse Text]
```

If the string cannot be parsed as the given type, then a fatal error is raised and the TXL program is terminated, which can be inconvenient. Using robust parsing, there is s simple paradigm to avoid this and make the parse robust:

```
% See if we can parse Text as a declaration
construct OptDecl [declaration_or_not]
     _ [parse Text]
% Check if we got one
deconstruct OptDecl
     Decl [declaration]
```

Where [declaration_or_not] is defined as:

```
define declaration_or_not
     [declaration]
   | [not_declaration]
end define

define not_declaration
     [repeat token_or_key]     % any sequence of input at all
end define

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

The [reparse] function takes an argument of any nonterminal type,, and attempts to reparse the leaves (tokens) of its parse tree as the scope type as if it were input. It is similar to [parse] except that it reparses directly from one type to another without converting to a string or rescanning the text. The leaves parsed are not converted to text or rescanned, so they retain their original types and contents in the new parse. If a rescan is required, for example when reparsing an item to character mode (–*char*), then explicit conversion to text using [quote] and reparsing using [parse] will be necessary.

As an example, the small program below defines two kinds of inputs, [**repeat** thing] and [**repeat** thang]. Both allow inputs consisting of sequences of numbers, identifiers and parentheses, but [thing] enforces matched parentheses whereas [thang] does not. Because it is the first alternative, the program normally takes in a structured input parsed as [**repeat** thing], requiring fully nested parentheses.

```
define program
     [repeat thing]
   | [repeat thang]
end define

define thing
     [id] | [number] | ( [repeat thing] )  % note nested parentheses
end define

define thang
     [id] | [number] | ( | )               % note arbitrary parentheses
end define
```

In the example below, in order to allow manipulation of the input in unstructured fashion (i.e., without the requirement of keeping parentheses nested and matched while transforming), the program uses [reparse] to construct the parse *Thangs* which views the input as the unstructured type [**repeat** thang].

With this reparse the program is then free to manipulate the *Thangs* in unstructured fashion, in this case by adding unmatched parentheses one by one. The program then uses [reparse] once again, this time to reparse the unstructured *NewNewThangs* as the required result type, the structured type [**repeat** thing], to make certain that the parentheses we added are once again legally matched.

```
function main
     replace [program]
          Things [repeat thing]

     % reparse to get unnested form
     construct Thangs [repeat thang]
          _ [reparse Things]

     % add an unbalanced parenthesis at the head
     construct NewThangs [repeat thang]
          ( Thangs

     % and another at the tail
     construct CloseParen [thang]
          )
     construct NewNewThangs [repeat thang]
          NewThangs [. CloseParen]

     % reparse the result to check balanced
     construct NewThings [repeat thing]
          _ [reparse NewNewThings]
     by
          NewThings
end function
```

The [unparse] function is the inverse of [parse], converting any parsed argument to its text and appending it to the [stringlit], [charlit], [id] or any other token type given as scope. The semantics and use of [unparse] is identical to [quote].

The following summarizes the dynamic parsing operations:

Dynamic Parsing Operations	(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)
`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]

8.3 Generic Programming

Using the general polymorphic nonterminal type [any] in TXL, it is possible to program many things generically that would otherwise involve many type-specific copies of widely used rules. For example, a simple containment testing function can be made very general using [any] :

```
% Given an item of any nonterminal type,
% test whether the scope contains an instance of it
function contains Object [any]
    match * [any]
        Object
end function
```

On the face of it, this function looks much like any use of it could be replaced by a deep deconstruct. It's only when used with *each* that we really see its usefulness:

```
% Does this declaration have any of these attributes?
where
    Declaration [contains each InterestingAttributes]

% Does the this method mention any of these variables?
where
    Method [contains each Variables]
```

Another obvious use of [any] is to enable writing 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 copies of the same function for different scope types.

```
function applyRules
    replace [any]
        Scope [any]
    by
        Scope [rule1]
              [rule2]
              [rule3]
end function
```

Use of [any] creates untyped TXL variables that are opaque - that is, we don't know he nonterminal type of tree they are bound to. The [typeof] and [istype] predefined functions are designed to support a limited amount of type-dependent generic programming with [any] variables.

The [typeof] function takes a scope of type [id] and replaces it with the nonterminal type name of the type of its argument, which should be of type [any]. This allow us to use the type in various ways, including for example making the parse of a tree explicit in XML :

```
% syntax of simple XML markup ([SPOFF] and [SPON] control output spacing)
define xml_markup
     < [SPOFF] [id] > [SPON]
         [any]
     < [SPOFF] / [id] > [SPON]
end define

% rule to mark up only specified nonterminals with their type
rule markupSpecifiedStructures
     % things to mark up - in this case all unstructured control
     % statements in Java
     construct SpecifiedTypes [repeat id]
         'break_statement 'continue_statement 'return_statement
         'throw_statement
     % look at every node in the program's parse tree,
     % but don't look inside already marked things
     skipping [markup]
     replace [any]
         Subtree [any]
     % get the type of this subtree
     construct SubtreeType [id]
         _ [typeof Subtree]
     % see of it is one of those we're interested in
     deconstruct * [id] SpecifiedTypes
         SubtreeType
     % if so mark it up
     construct MarkedUpSubtree [xml_markup]
         < SubtreeType > Subtree </ SubtreeType >
     % get generic version of marked up node
     % (since this rule transforms [any])
     deconstruct * [any] MarkedUpSubtree
         AnyMarkedUpSubtree [any]
     by
         AnyMarkedUpSubtree
end rule
```

The [istype] predefined function is a shorthand for testing the type of the tree bound to an [any]. For example, the statements above:

```
% get the type of this subtree
construct SubtreeType [id]
     _ [typeof Subtree]
```

```
        % see of it is one of those we're interested in
        deconstruct * [id] SpecifiedTypes
             SubtreeType
```

Can be replaced with this simpler version using [istype] :

```
        % see of it is one of those we're interested in
        where
             Subtree [istype each SpecifiedTypes]
```

Summary of generic programming functions:

Operations on Types	(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

9. Programmed Input / Output

The input/output predefined functions allow TXL programs to deal with multiple input and output files and to interactively accept input or print output to the terminal or other files. In general, files are automatically opened and closed as used, but explicit operations for programmed control over opening and closing of files is also provided.

Terminal input/output operations normally use the standard input stream (*stdin* on Unix/Linux) for input and the standard error stream (*stderr* on Unix/Linux) for terminal output. However, console behavior can be platform dependent, and users should refer to the section on interactive TXL programs in the *TXL User's Guide* for specific platform details.

9.1 File-oriented Input / Output

At times a TXL program may need to deal with more than one input or output file, for example when processing *include* statements of a language, when merging two or more programs, or when splitting a program into aspects. TXL provides the [read] and [write] predefined functions to support dynamic input and output file parsing and unparsing for this purpose. Both [read] and [write] take as argument the name of a text file which is automatically opened or created and closed as necessary before and after the operation.

The [read] function opens and parses a file whose filename is specified as a [stringlit] argument, replacing the scope with a parse of the file text as the scope type. [read] can be particularly useful in writing a TXL program to parse and transform several programs in the same run, and in processing include files.

For example, consider the following TXL function to compare two programs to see if they are clones modulo formatting, commenting and renaming :

```
        function compareProgs FileName [stringlit]
             % the first program is the scope -
             % parsing removes formatting and comments
             match [program]
                 P1 [program]

             % normalize P1 to anonymous identifiers
             construct NormalizedP1 [program]
                 P1 [normalize]

             % the second program is read from the given file name
             % again parsing removes formatting and comments
             construct P2 [program]
                 P1 [read FileName]

             % normalize P2 to anonymous identifiers
             construct NormalizeP2 [program]
                 P2 [normalize]

             % compare the normalized forms
             deconstruct NormalizedP2
                 NormalizedP1

             % print a message if they are the same
             construct Message [id]
                 _ [message "The programs are clones!"]
        end function

        % Simple normalization rule -
        % just make all identifiers the same
        rule normalize
             replace [id]
                 Id [id]
             % don't go on forever!
             deconstruct not Id
                 'x
             by
                 'x
        end rule
```

The [write] function is the inverse of [read]. It takes as argument a [stringlit] giving the filename of a file, and writes (as text) the leaves (tokens) of the scope to this file as unparsed text. The scope may be of any type, and is not changed. Output is formatted according to the grammar exactly as it would be in the main output of the TXL program.

[write] can be used to create multiple files as output, and can be useful in debugging, as results of individual transforms can be written to a file for future reference. For example, we might have something like this:

```
    function applyFunctions File1 [stringlit] File2 [stringlit]
        replace [program]
            P1 [program]
        construct TransformedP1 [program]
            P1 [DoSomething]     % Do a series of transformations here
               [write File1]     % Write intermediate results to a file
               [DoSomethingElse] % More transformations
               [write File2]     % Second intermediate results
               [DoTheRest]       % Final transformations
        by
             TransformedP1
    end function
```

The file input/output operations can be summarized as:

File Input/Output Operations	(S1 must be of type [stringlit], [charlit], [id] or any other identifier type, X1 can be of any type)
`X1 [read S1]`	replaces X1 of any type [T] with a parse of the text file S1 as a [T], automatically opening and closing the file before and after the operation
`X1 [write S1]`	writes the output text of X1 as the text file S1, automatically opening / creating and closing the file before and after the operation

9.2 Interactive Input / Output

The interactive input/output functions provide line-oriented access to the terminal and text files. The [get], [getp], [gets], [put], [putp] and [puts] functions interact with the terminal console, and the corresponding [fget], [fgets], [fput], [fputp] and [fputs] functions allow line-oriented input/output to named files. Line-oriented input/output is linewise synchronous, which means that the [get]/[fget] functions will wait until a complete line is available in the input on each operation, and [put]/[fput] functions will flush the output file following each output.

Files are automatically opened on the first access, and remain open until the TXL program finishes or the file is explicitly closed using [fclose]. Up to 5 line-oriented files may be simultaneously open, and use of [fclose] to explicitly close a file is unnecessary unless the program needs to reuse the 5 slots over the length of a run.

The [get] function takes no arguments, and reads one line of text from the terminal, scanning and parsing it as the scope type (which can be of any type).

```
    % interactively get a number from the terminal
    construct InputNumber [number]
        _ [get]
```

The input must be parseable as the scope type. For terminal input ([get], [getp]), if the input cannot be parsed then the user will be prompted for another input with a message of the form:

```
    [number] input expected - try again (y/n)?
```

If the response is "y" then then another input line is read from the terminal, otherwise a fatal error is raised.

[fget] is similar to [get], reading the next line of the argument text file and scanning and parsing it as the scope type. It is always a fatal error if the input to [fget] cannot be parsed as the scope type. The argument file name is automatically opened for text input if it is not already open, and is kept open until the end of the run or it is explicitly closed using the [fclose] function.

Lines of the input file are read one-by-one sequentially, returning an empty line at end of file. Here is an example rules using [fget] to read an entire file of numbers, one per line, and returning their sum:

```
rule sumFile FileName [stringlit]
    % The sum so far - we should be called beginning with 0
    replace [number]
        SumSoFar [number]
    % Try to get the next number
    construct OptNextNumber [opt number]
        _ [fget FileName]
    % If there wasn't one we're done
    deconstruct OptNextNumber
        NextNumber [number]
    % Otherwise add it in and continue
    by
        SumSoFar [+ NextNumber]
end rule

% Example use of this rule
construct SumOfMarks [number]
    _ [sumFile "marks.txt"]
```

[getp] differs from [get] in that it takes and argument [stringlit] which is used as the input prompt. For example, the following function interactively prompts the user to input an expression to evaluate, using the prompt *"Please input an expression to evaluate: "*. A line of input will be read from the terminal and parsed as an [expression] according to the program's grammar.

```
function getExpnAndEvaluate
    replace [expression]
        Expn [expression]
    % ask the user for an expression
    construct ExpnToEvaluate [expression]
        _ [getp "Please input an expression to evaluate: "]
    % evaluate and output the result
    by
        ExpnToEvaluate [evaluate]
end function
```

There is no corresponding [fgetp] since prompts don't make sense for file input.

The [put] and [putp] functions are the corresponding output functions to [get] and [getp], providing output of the unparsed text of the scope to the terminal. The [put] function takes no arguments, and simply outputs the unparsed text of the scope (which may be of any type) to the terminal console. If the scope is a [stringlit] or [charlit], then the text is output in unquoted form.

```
construct Assignment [statement]
    x := 27;
construct Output [statement]
    Assignment [put]         % outputs "x := 27;" on a line
```

Output is formatted according to the grammar in the same way it would be in the unparsed output of the program, and is not limited to a single line.

The [putp] function takes as a argument a [stringlit] pattern of the form *"Here is some output % in the* *middle of the string"*, where the % symbol marks the position to embed the scope text in the output. If there is no % in the argument string, then the end of the string is assumed. As an example, consider the following function:

```
construct ResultOfExpn [expression]
     Expn [evaluate]
          [putp "The result of the expression was:  %."]
```

The output of this example would be a line on the terminal something like:

```
The result of the expression was: 42.
```

While the [get]/[fget] and [put]/[fput] functions use the TXL parser to convert items of any type to and from text form, the [gets]/[fgets] and [puts]/[fputs] input/output uniterpreted text lines directly as [stringlit] or [charlit] values. This can be useful when writing TXL programs that require character-level text processing of interactive input, for example when interpreting another language using a TXL transformation.

The [gets] predefined function reads one line of input from the terminal as text and replaces its scope, which must of of type [stringlit] or [charlit], with the raw text as a string. [fgets] is the corresponding operation for reading raw text lines from files. In both cases the input line is uninterpreted and all spacing and special characters, such as spaces and tab characters, are input intact. An empty line is returned at end of file.

[puts] and [fputs] are the corresponding output operations, writing the raw text of their [stringlit] or [charlit] scope unchanged to the terminal or specified file respectively. As an example, the following function copies the lines of a given file to another, stopping at the first empty line.

```
rule CopyLines InputFile [stringlit] OutputFile [stringlit]
     % we ignore our scope, so just pass it through
     replace [any]
          Scope [any]
     % get the next line from our input file
     construct NextInputLine [stringlit]
          _ [fgets InputFile]
     % if it's empty we're done
     deconstruct not NextInputLine
             ""
     % output it to our output file
     construct NextOutputLine [stringlit]
          NextInputLine [fputs OutputFile]
     by
          Scope
end rule
```

While in most cases the automatic opening and closing of files by the TXL programmed input/output functions is sufficient, at times it is necessary to have greater control. For example, if a great many files are to be interactively read, because they are not automatically closed until the end of the run, the TXL program can run out of file descriptors in a Unix/Linux system. To assist with this, the predefined functions [fopen] and [fclose] are provided to allow for explicit programmed opening and closing of files.

[fopen] takes two arguments, a [stringlit] file name and a [stringlit] file mode. The file mode may be one

of "get" for input (Unix/Linux "r" mode), "put" for output (Unix/Linux "w" mode), and "append" for append-to-end output (Unix/Linux "a" mode). In all cases the file is opened in text mode rather than binary. "append" mode provides the ability to concatenate output to the end of an existing file, which can be useful in a multi-stage transformation or when logging TXL actions.

[flcose] takes a [stringlit] file name as argument and immediately closes the file, whatever its mode. It is most useful in programs that must open many interactive files, where it can be used to free up file descriptors to allow for an unbounded number of files, and in programs that wish to reopen for input a line-oriented file it has previously written to. Note that the file input/output functions [read] and [write] do not require [fopen] or [fclose] since they automatically open and close their files for each operation.

Summary of interactive input/output operations:

<u>Interactive Input/Output Operations</u>

Note: All operations automatically open the input/output file if it is not already open.

(S1 must be of type [stringlit], [charlit], [id] or any other identifier type,
F1 and M1 must be of type [stringlit] or [charlit], and X1 can be of any type)

`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 [fopen F1 M1]`	opens file F1 in the indicated mode M1, which must be one of "get", "put" or "append" - only necessary if the mode is "append"
`X1 [fclose F1]`	closes file F1 - only necessary if short of file descriptors or the file is to be reopened (e.g., for input) in the same run

10. Debugging Aids

While the TXL Debugger *txldb* can be used to help debug small programs, it is often too detailed or at too low a level for debugging complex TXL transformations. The debugging predefined functions provide the ability to instrument and control run-time execution of TXL programs as the user wishes by inserting calls where they are needed.

10.1 Programmed Messages

The [message] function is one of several ways to output text to the console terminal. The function prints the leaves of the tree given as argument to the debugging output. The argument may be of any type, but [message] is most often used with a [stringlit]. The scope that the function is applied to is ignored and unchanged.

The following program, for example, uses [message] as a means of telling the user the current status of the transformation:

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

The [message] function is also the usual way to output error messages from the transform itself, as in:

```
function checkCompatibleTypes T2 [primitive_type]
    match [primitive_type]
        T1 [primitive_type]
    deconstruct not T1
        T2
    construct Message [id]
        _ [message "Error, types do not match"]
end function
```

The [print] function prints out the leaves of the scope tree to the console terminal. The function can be applied to any scope, and the scope tree itself remains unchanged. It is very similar to the [message] function described above except that it prints its scope rather than its argument.

In this example, we trace the progress of the transformation rule by showing the intermediate result after each subrule is applied:

```
rule reduceExpression
    replace [expression]
        E [expression]
    by
        E [reduceMultiplications]
          [message "After [reduceMultiplications] we have:]
```

```
                 [print]
                 [reduceAdditions]
                 [message "After [reduceAdditions] we have:]
                 [print]
        end rule
```

This function is also useful when two or more separate transform outputs from one TXL program are required, as shown below. Under Unix, the two outputs can be separated using output redirection.

```
        function main
            replace [program]
                P [program]
            construct P1 [program]
                P [transformSetOne]
                  [print]                % to standard error stream
            by
                P [transformSetTwo]      % to standard output stream
        end function
```

[printattr] is identical to [print] except that attributes (e.g. [attr X]), which are normally not visible in TXL output, are printed as well to aid debugging of transformations that require attributes.

See also the [put] and [putp] functions for standard error stream output.

Programmed Messages	(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

10.2 Interactive Debugging

The [debug] and [breakpoint] operations provide some of the functionality of the TXL Debugger *txldb* under programmed control. The [debug] function prints out the internal tree representation of the scope tree that the function is applied to to the standard error output in XML form. The scope remains unchanged. It is similar to [print] except that it outputs the entire tree structure rather than just the leaves of the tree. This function's primary use is in viewing intermediate trees when debugging, as seen below:

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

The output produced by this program as it runs looks something like this:

```
Reducing multiplications in:

--- DEBUG expression ---
<expression>
   <expression><term>
      <term><primary><number>2</number></primary></term>
         <mulop>*</mulop>
         <primary><number>4</number></primary>
      </term>
   </expression>
   <addop>+</addop>
   <term><primary><number>3</number></primary></term>
</expression>

Reducing additions in:

--- DEBUG expression ---
<expression>
   <expression><term><primary><number>8</number>
      </primary></term></expression>
      <addop>+</addop>
   <term><primary><number>3</number></primary></term>
</expression>

Reducing multiplications in:

--- DEBUG expression ---
<expression>
   <term><primary><number>11</number></primary></term>
</expression>

Reducing additions in:

--- DEBUG expression ---
<expression>
   <term><primary><number>11</number></primary></term>
</expression>
```

The [breakpoint] function provides the ability to temporarily stop the transformation at a given point. It works by printing a message on the debugging output, and temporarily suspending the program until a carriage return is pressed on the keyboard. The scope tree remains unchanged.

[breakpoint] is most useful in a program that being instrumented using [message], [print] and [debug] to allow the user to look at the debugging information at a given point before going on. For example:

```
rule reduceExpression
    replace [expression]
        E [expression]
    by
```

```
         E [reduceMultiplications]
           [message "so far, we have:"] [print] [breakpoint]
           [reduceAdditions]
end rule
```

Summary of interactive debugging functions:

Interactive Debugging Aids	(X1, X2 can be any type at all)
`X1 [debug]`	outputs the internal tree format of X1 to the standard error stream
`X1 [breakpoint]`	temporarily halts execution until a carriage return is received from the standard input

11. Execution Control

These functions allow dynamic control over TXL options and results normally handled at the command line level.

11.1 Dynamic Run Options

The [pragma] predefined function allows run-time changes to command line options such as *–char, –case* and *–raw,* for example allowing the program to dynamically scan and parse items using the [read], [parse] and [quote] predefined functions using different options than for the main input, or to dynamically output items using [write], [put] or [print] using different output spacing options than the main output.

[pragma] takes a [stringlit] command line options string as argument, and can be applied to a scope of any type, which is unchanged.

```
% Get actual carriage return and tab characters as strings
construct CarriageReturn [stringlit]
     _ [pragma "-char"]
       [quote "^M"]          % actual carriage return
       [pragma "-nochar"]
construct Tab [stringlit]
     _ [pragma "-char"]
       [quote "  "]          % actual tab character
       [pragma "-nochar"]
```

11.2 Programmed Abort

The [quit] function allows the TXL program to abort with a given return code at any time. It takes an integer return code as argument and can be applied to any scope, which is unchanged. The return code can be used to indicate success or failure to the batch or shell script that invoked the TXL program. TXL programs normally return with a 0 return code, indicating success. To return another code, they can use [quit], for example:

```
function checkCompatibleTypes T2 [primitive_type]
    match [primitive_type]
        T1 [primitive_type]
    deconstruct not T1
        T2
    construct Message [id]
        _ [message "Error, types do not match"]
          [quit 99]
end function
```

Summary of execution control predefined functions:

<u>Execution Control Operations</u>	(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

12. System Interface

The system interface built-in functions [system] and [pipe] allow the TXL program to run other programs or commands as part of a TXL transformation, for example to manage files or use editor scripts to assist in the transformation. The syntax and semantics of the commands are platform dependent, and the system interface functions are not supported on all platforms. On Unix/Linux/MacOSX systems, the system commands invoke the standard *system()* library routine to run a shell command line, and on Windows systems, they invoke the Windows *system()* routine to run a DOS command line.

12.1 Execute System Command

The [system] command takes as argument a [stringlit] shell command line to execute and invokes the host system's *system()* library routine to execute it. The scope, which can be of any type, is ignored and unchanged. [system] is typically used in conjunction with [write] and [read], as for example here:

```
% Reformat expressions one per line
redefine expression
      ... [NL]
end redefine

function AllUniqueExpressions Program [program]
      % Finds and returns all unique expressions in the program

      % First get all the expressions in the program
      construct AllExpressions [repeat expression]
            _ [^ Program]

      % use “sort -u” to order and uniquify the set
      construct UniqueExpressions [repeat expression]
            AllExpressions [write “/tmp/expns”]
                           [system “sort -u /tmp/expns > /tmp/uexpns”]
                           [read “/tmp/uexpns”]
                           [system “rm -f /tmp/expns /tmp/uexpns”]
```

```
        replace [repeat expression]
              % scope should be empty when called
        by
              UniqueExpressions
    end function
```

[system] succeeds or fails depending on the return code of the command, and can be used in a **where** clause to check for success.

```
    % Are we on a Unix-style system?
    where
          Something [system "ls > /dev/null"]
```

The [pipe] function takes as scope a [stringlit] or [charlit] string of text to be piped into the command specified by its [stringlit] or [charlit] argument. The scope is replaced with the result of the command as a text string. For example:

```
    % Use the Unix sed command to expand tabs in a string literal
    construct ExpandedString [stringlit]
          String [pipe "sed 's/\t/    /'"]
```

Summary of system interface built-in functions:

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 S*" | *text of S2* and replace the text of S1 with the first line of the result