The TXL Source Transformation Language

James R. Cordy[1]

School of Computing, Queen's University, Kingston, Canada

Abstract

TXL is a special-purpose programming language designed for creating, manipulating and rapidly prototyping language descriptions, tools and applications. TXL is designed to allow explicit programmer control over the interpretation, application, order and backtracking of both parsing and rewriting rules. Using first order functional programming at the higher level and term rewriting at the lower level, TXL provides for flexible programming of traversals, guards, scope of application and parameterized context. This flexibility has allowed TXL users to express and experiment with both new ideas in parsing, such as robust, island and agile parsing, and new paradigms in rewriting, such as XML markup, rewriting strategies and contextualized rules, without any change to TXL itself. This paper outlines the history, evolution and concepts of TXL with emphasis on its distinctive style and philosophy, and gives examples of its use in expressing and applying recent new paradigms in language processing.

Key words: source transformation, functional programming, term rewriting, grammars

1 What is TXL?

TXL[20,21] is a programming language specifically designed for manipulating and experimenting with programming language notations and features using source to source transformation. The motivating paradigm of TXL consists of beginning with a grammar for an existing language, specifying syntactic modifications to the grammar representing new language features or extensions, and rapidly prototyping these new features by source transformation to the original language.

[1] E-mail: cordy@cs.queensu.ca

```
% Trivial coalesced addition dialect of Pascal

% Based on standard Pascal grammar
include "Pascal.Grm"

% Overrides to allow new statement forms
redefine statement
        ...
    |   [reference] += [expression]
end redefine

% Transform new forms to old
rule main
    replace [statement]
       V [reference] += E [expression]
    by
       V := V + (E)
end rule
```

Fig. 1. An Example TXL Program

While TXL was originally designed to support experiments in programming language design, it has proven much more widely applicable. It has been used in a range of applications in programming languages, software engineering, database applications, structured documents, web technology and artificial intelligence among many others, and with a range of programming languages including C, C++, Java, COBOL, PL/I, RPG, Modula 2, Modula 3, Miranda, Euclid, Turing and many others. In particular it was used as the core technology in the LS/2000 analysis and remediation system[24], which processed over 4.5 billion lines (Gloc) of source code.

TXL programs (Figure 1) normally consist of three parts, a context-free “base” grammar for the language to be manipulated, a set of context-free grammatical “overrides” (extensions or changes) to the base grammar, and a rooted set of source transformation rules to implement transformation of the extensions to the base language.

2 How TXL Came to Be

This paper considers the TXL language from an historical perspective, tracing from its roots in the rapid prototyping of language dialects to its present use as a generalized source transformation system. It is not intended to explore the formal semantic properties of the language, to comprehensively catalogue its paradigms of use, or to demonstrate its application to real problem domains. These issues are addressed in many other papers[35,25,21,24,23,36,58].

TXL has a different heritage than most other language manipulation and transformation tools, and its goals are different. TXL does not originate with parsing, term rewriting or attribute grammar technology - rather its heritage is rapid prototyping and first order functional programming. It was born in the early 1980's, in a time when the study of programming language design was an active and productive area. Experimentation with new programming languages and features was the order of the day, and many languages, including C++, Modula 3, Eiffel, Ada, Perl, Prolog and Miranda have their roots in that time. One such language was Turing[29].

2.1 The Turing Language Project

The goal of the Turing project was to design a general purpose language with excellent ease-of-use, lightweight syntax and formal axiomatic semantics that was also very accessible and easy to learn. The design of Turing was heavily influenced by the "programming as a human activity" philosophy of Gerald Weinberg's *Psychology of Computer Programming*[57]. As a result the Turing project adopted a "design by use" philosophy - when users made errors by writing what they thought "ought to work", we would study these errors to look for opportunities to make the language more like what the users expected.

An example of this was the design of the substring features of the *string* type in Turing. Original syntax to choose a character out of a string was simple subscripting - so for example if the string variable `s` has value `"hello"`, then `s(1)` chooses the character `"h"`. Because Turing has the notion of a subrange of integers, for example `1..10`, users naturally fell into writing `s(1..4)` to get longer substrings, and this was the feature added to the language.

Turing uses an asterisk (`*`) to denote the upper bound of a parameter array (as in `array 1..* of int`). Users therefore began to write `s(3..*)` to mean the substring from position 3 to the end of the string, `s(1..*-1)` to mean the substring from the first position to the second last, `s(*-1..*)` to mean the substring consisting of the last two characters, and so on. As these forms evolved, the language was modified to adapt to the users' expectations.

This experimental style of language design proved very successful - the features of the Turing language seemed "natural" because the users helped to design them. Users would explain what they meant by showing an equivalence - for example, when asked what `s(2..*)` meant to them, they would say `s(2..length(s))`. This led to an example-based understanding of meaning - a *this-means-that* style. Turing language proposals therefore most often consisted of a pair drawn on the board - the syntax of an example use of the new feature on one side, and its corresponding meaning in the syntax of the current language on the other (Figure 2).

Adapting Turing to these new ideas involved the heavyweight process of rebuilding each of the phases of the compiler to add the lexical, syntactic, se-

Proposal for an Object-Oriented extension to Turing

```
                                        module ID
type ID:                                    IMPORTS
    class                                   EXPORTS
        IMPORTS                             export DataRecord
        EXPORTS          means              type DataRecord:
        FIELDS                                  record
        METHODS                                     FIELDS
    end ID                                      end record
                                            METHODS  (fix field references)
                                        end ID

                                        (fix variable declarations and references)
```

Fig. 2. A Turing “This-means-that” New Feature Proposal

mantic and code generation changes for each new feature. This tended to discourage experimentation, commit us too early to features we weren’t sure about, and slow down the rapid evolution that we had in mind.

2.2 The Turing eXtender Language

Ideally what we wanted to have was something that would allow us to instantly try out what we were writing on the board - simply show what we had in mind by example, and *presto!* a rapid prototype should appear. Thus the TXL idea was born - the *Turing eXtender Language*, a language for specifying and rapidly prototyping new language ideas and features in an example-like style. As we shall see, this vision drives all of the design decisions of TXL and its implementation.

It was clear that such a language could not be based in the compiler technology of the time - we wanted true rapid prototyping, with no generation or build steps, and a cycle time measured in seconds. This implied a direct interpretive implementation, and we therefore looked to Lisp for inspiration. In particular, MkMac[32], a language extension facility for the Scheme variant of Lisp, seemed to be something like what we had in mind.

Lisp[37] is a functional programming language based on one simple data structure: nested first-rest (*car-cdr*) lists. Lisp has a fast interpretive full backtracking implementation that is widely used in artificial intelligence and well suited to rapid prototyping. Its implementation is well understood and heavily optimized for list processing. For these reasons we chose Lisp as the model for the underlying semantics of TXL, using Lisp list structures as the basis of its parse trees, grammars and patterns; pure value semantics with no assignment or variables; function composition as the main control structure; and functional programming with full backtracking for both the parser and the transformer aspects of the language.

3 Design of the TXL Language

The design of the TXL language was driven almost entirely by the example-based rapid prototyping goal. In this section we introduce the basic features and properties of the TXL language in terms of the design goals that they meet.

3.1 Goal: Rapid Prototyping

The Lisp heritage of TXL led to a parsing model similar to that often used in Lisp and Prolog: direct top-down functional interpretation of the grammar. Beginning with the goal nonterminal `[program]`, a TXL grammar is directly interpreted as a recursive functional program consuming the input as a list of terminal symbols (*tokens*). The structure of the grammar is treated as a combination of two kinds of lists: *choice lists*, representing alternation, and *order lists*, representing sequencing. Alternate forms in choice lists are interpreted in the order they are presented in the grammar, with the first matching alternative taken as a success. List representation makes backtracking easy: when a choice alternative or sequence element fails, we simply backtrack one element of the list to retry previous elements until a full parse is obtained.

The result of a TXL parse is a parse tree represented in the same nested list representation. This representation is used throughout TXL to represent the grammar, parse trees, rules, patterns and replacements and is one of the main reasons that TXL is so fast. Because direct top-down backtracking interpretation of grammars has difficulty with left recursion, TXL recognizes and interprets left-recursive definitions as a special case, effectively switching to bottom-up interpretation of these productions on the fly. Nevertheless it is still quite possible to write a TXL grammar that is slow or impractical to use because of too much backtracking - this is the price we pay for being able to directly interpret the grammar, which as we will see plays a large role the power and flexibility of the language.

Specification of the grammar (Figure 3) uses a simple notation similar to BNF, with nonterminals referenced in square brackets (e.g., `[expression]`) and unadorned terminal symbols directly representing themselves. Terminals may be quoted using a single prefix quote (e.g., `'end`) as in Lisp, but only when necessary to distinguish them from a TXL keyword. In keeping with the example-based goal, the contents of a TXL nonterminal define statement are the direct unadorned sentential forms of the target language.

Because the grammar is interpreted in the order presented, the user has control over how input is parsed. Alternatives are ordered, with earlier forms

```
% Trivial statement language grammar
define program
      [repeat statement]
end define

define statement
      var [id];
    | [reference] := [expression];
    | { [repeat statement] }
    | if [expression] then
         [statement]
      [opt else_statement]
    | while [expression] do
          [statement]
end define

define else_statement
      else [statement]
end define
```

```
define expression
      [primary]
    | [expression] [op] [expression]
end define

define op
      + | - | * | /
    | = | > | < | >= | <=
end define

define primary
      [id]
    | [number]
    | ( [expression] )
end define
```

Fig. 3. An Example TXL Grammar

taking precedence over later ones. Since the grammar is effectively a program for parsing under user control, no attempt is made to analyze or check the grammar - any grammar that can be written has some interpretation. In particular, since the grammar is now a programming language, TXL does not attempt to restrict it in any way, and nonterminating parses are intentionally the responsibility of the programmer.

Ambiguity in the grammar is allowed, and as we shall see, is very important to the TXL paradigm. Because the grammar is interpreted in ordered fashion, resolution of ambiguities when parsing is automatic. However, ambiguous forms are not necessarily redundant, because transformation rules may force construction of any tree structure allowed by the grammar (including those that would never be the result of an input parse) at transformation time. Advanced programming techniques in TXL frequently exploit ambiguity in this way.

Several standard extended BNF structures are built in to TXL, notably `[opt X]`, which means zero or one items of nonterminal type `[X]`, `[repeat X]`, meaning a sequence of zero or more `[X]`s, and `[list X]`, meaning a comma-separated sequence of zero or more `[X]`s. An important property of the `[repeat X]` structure is that it is right-recursive, defined as either *empty* or `[X]` followed by `[repeat X]` in Lisp first-rest style. This matches the natural interpretation of declarations and statements in many programming languages. For example, the scope of a declaration in Turing is from the declaration itself to the end of the scope, captured by the parser as the rest of the statements following the declaration.

The naive unrestricted form of TXL grammars is essential to the goal of rapid prototyping - working grammars can be crafted quickly, often directly from

```
% Some example grammar overrides based on the Java grammar
include "Java.Grm"

% Distinguish assignments from other expression statements
redefine expression_statement
       [assignment_statement]     % preferred new form takes precedence
    |  [expression];              % original form, ambiguous with new
end redefine

define assignment_statement
       [assignment_expression];
end define

% Add optional XML tags on expressions
redefine expression
       ...
    |  [xmltag] [expression] [xmlendtag]
end redefine

% Distinguish JDBC method calls from others
redefine method_call
       [jdbc_call]
    |  ...
end redefine
```

Fig. 4. TXL Grammar Overrides Using Redefines

user-level reference manuals, without the necessity of removing ambiguities, dealing with shift-reduce conflicts or restructuring to adapt to parser restrictions. A grammar for a substantial new language can be crafted and working in TXL in less than a day, and the parse trees created can be in the natural concrete form of users of the language rather than the abstract implementation grammar form used by compilers, making it easier to understand and remember forms when crafting patterns and transformation rules.

3.2 Goal: Language Experimentation

The main TXL goal of language experimentation requires that we have some way to add new forms and modify old forms in an existing grammar. TXL captures this idea with the notion of *grammar overrides*. TXL programs normally begin with a *base grammar* which forms the syntactic basis of the original language we are experimenting with. The base grammar is then modified by *overriding* nonterminal definitions to change or extend their form using grammar *redefines* (Figure 4).

Redefines replace the existing nonterminal definition of the same name in the base grammar with the new definition, effectively making a new grammar from the old. Overrides can either completely replace the original definition of the nonterminal, or they can refer to the previous definition using the “. . .” notation, which is read as “what it was before” (Figure 4). So for example the redefinition “. . . | [X]” simply adds a new alternative form [X] to the nonterminal, as when adding a new statement to a language. Because TXL definitions are interpreted sequentially, new forms may be added as either pre-extensions ("`[X] | ...`") or post-extensions ("`... | [X]`"), corresponding to the new form being preferred over old ones in the former and old forms being preferred over the new in the latter.

Redefinitions are interpreted in the order that they appear, which means that later redefinitions can extend or modify previous redefinitions, allowing for dialects of dialects and extensions of previous language extensions. The effective grammar is the one formed by substituting each of the redefinitions into the grammar in the order that they appear in the TXL program.

Grammar overrides are the key idea that distinguishes TXL from most other language tools. They allow for independent exploration of many different dialects and variants of a language without cloning or modifying the base grammar or other existing dialects. As we shall see, they also allow for *agile parsing* - the ability to independently modify grammars to suit each particular transformation task.

3.3 Goal: Example-like Patterns and Replacements

The this-means-that idea on which TXL is based requires an example-like style for transformation rules, in which both patterns and replacements (post-patterns) are specified in the concrete syntax of the target language, the style recently referred to as *native patterns*[49]. TXL patterns are effectively unadorned sentential forms (examples) of the things we want to change and what we should change them to (Figure 5).

TXL rules specify a *pattern* to be matched, and a *replacement* to substitute for it. The nonterminal type of the pattern (the *target* type) is given at the beginning of the pattern, and the replacement is implicitly constrained to be of the same type. In this sense TXL is strongly typed, using the grammar as the type system of the TXL program. Patterns and replacements are parsed using the same direct interpretive execution of the grammar that the input is parsed by, compiling them into parse tree *schemas* in the same list form as the parse tree of the input. Transformation rules are executed by searching their input (the *scope* of the rule) for parse subtrees matching their pattern tree, and replacing them with a copy of their replacement tree with parts captured in the pattern copied into the result. The process is repeated on the result until no new matches can be found.

In patterns and replacements as in grammar defines, terminal symbols simply represent themselves, quoted only when necessary to avoid conflict with TXL keywords, and nonterminals are referenced using square brackets (e.g.,

```
% Part of transformation to implement OO extension to Turing

rule transformClasses
   replace [repeat declaration_or_statement]
      type ClassId [id] :
         class
             Imports [repeat import_list]
             Exports [repeat export_list]
             Fields [repeat variable_declaration]
             Methods [repeat procedure_delaration]
         'end ClassId
      RestOfScope [repeat declaration_or_statement]
    by
      module ClassId :
         Imports
         export DataRecord
         Exports
         type DataRecord:
             record
                Fields
             'end record
         Methods [fixFieldReferences each Fields]
                 [makeConstructorMethod]
                 [addObjectParameterToMethods]
      'end ClassId
      RestOfScope [transformClassReferences ClassId]
end rule
```

Fig. 5. The TXL Example-like Style (adapted from [19])

```
rule simplifyAssignments
   replace [statement]
      V [reference] := V + E [term]
    by
      V += E
end rule
```

Fig. 6. A Rule Using a Non-linear Pattern

`[expression]`). Pattern nonterminals are "captured" in TXL variables by labelling them with a variable name (e.g., `Expn [expression]`). Variables are explicitly typed only at their first occurrence, which on each pattern match binds them to the corresponding part of the matched input. Subsequent references to a variable refer to its bound value.

Bound variables may be referred to in replacements, which allows for copying parts of the matched input to the substituted output, but they may also be referred to later in the pattern in which they are bound or in other subsequent patterns, allowing for non-linear pattern matching[46]. References to bound variables have copy semantics, that is, they can only be matched by an exact copy of their bound subtree (Figure 6). For efficiency reasons, TXL provides only *one-way pattern matching*, that is, the binding occurrence of a pattern variable must be the first occurrence.

3.4 Goal: Context-dependent Transformations and Relationships

A common difficulty with source transformation systems is control over the scope of application of rules. It is frequently the case that desired transformations are phrased in terms such as *"this means that, except within that we substitute ..."* or *"this means that, except outside this we substitute ..."*. An example of this is the object-oriented Turing language extension of Figure 5. In this transformation, once the basic substitution has been made, other transformations need to be applied, some of which must be limited to the scope inside the transformed part, and some of which must be limited to the scope outside and following the transformed part. This limitation of scope of application can be difficult to express in a pure term rewriting system, requiring complex guards on rewrite rules.

In TXL, such scope limitations fall naturally from the decompositional style of the functional paradigm. Rules are structured into a rooted pure functional program in which lower level rules are applied as functions of subscopes captured by higher level patterns. Higher level rules capture in their pattern variables the subparts to which lower level rules are explicitly applied as part of the construction of their replacement.

Invocation of a subrule is denoted by the subrule name in square brackets following the name of the variable capturing the subtree to which it is to be applied, for example `Thing [changeit]` where *changeit* is the name of the subrule and *Thing* is the pattern variable containing the context within which it is to be applied. In keeping with pure functional value semantics, the result of a subrule invocation is a copy of the bound subtree as changed by the subrule. Subrules may be applied to the result of a subrule invocation by invoking another subrule on the result, as in `X[F][G]`, denoting the function composition `G(F(X))`.

The semantics of an entire TXL transformation is the application of the distinguished rule called *main* to the entire input. The main rule typically simply captures the highest level structure to be transformed (often the entire input) and invokes several composed subrules on it to do the real work. In complex transformations, this same paradigm is used again in the subrules, and so on, to decompose and modularize the transformation.

3.5 Goal: Complex Scalable Transformations

TXL was expected to allow easy rapid prototyping of any possible Turing language dialect or extension that could be imagined. As a result, it was designed to allow for easy user refinement of patterns and replacements in order to scale

```
% Remove all literally false if statements
rule foldFalseIfStatements
   replace [repeat statement]
      IfStatement [if_statement] ;
      RestOfStatements [repeat statement]
   % Pattern match deeply (*) to find the if condition -
   % matches the first [if_condition] in IfStatement,
   % which is of course the one guarding the statement
   deconstruct * [if_condition] IfStatement
      IfCond [if_condition]
   % Pattern match to see if it is false -
   % this deconstruct is not deep, so it matches only
   % if the entire IfCond is exactly the word "false"
   deconstruct IfCond
      false
   by
      RestOfStatements
end rule
```

Fig. 7. Pattern Refinement Using Deconstructs

up to complex multi-stage transformations without losing readability. For this reason, *deconstructors* and *constructors* were added to the language.

Deconstruct clauses constrain bound variables to match more detailed patterns (Figure 7). Deconstructors may be either shallow, which means that their pattern must match the entire structure bound to the deconstructed variable, or deep, which means that they search for a match embedded in the item. In either case, deconstructors act as a guard on the main pattern - if a deconstructor fails, the entire main pattern match is considered to have failed and a new match is searched for.

Replacements can also be stepwise refined, using *construct* clauses to build results from several independent pieces (Figure 8). Constructors provide the opportunity to build partial results and bind them to new variables, thus allowing subrules to further transform them in the replacement or subsequent constructs. They also provide the opportunity to explicitly name intermediate results, aiding the readability of complex rules.

Complex transformations may depend not only on the point of their application, but also on properties of other contexts remote from it. Thus a transformation rule may depend on many parts of the input captured from many different patterns. TXL allows for this using subrule parameters, which play the same role as additional function parameters in standard functional notation (Figure 9). Bound variables may be passed to a TXL subrule by adding them to the subrule invocation using the notation `X[F A B C]` where A, B and C are additional bound variables on which the subrule F may depend.

Inside the subrule, deconstructs can be used to pattern match the additional parameters in the same way that the main pattern matches the scope. This

```
% Minimize adjacent Modula VAR declarations

rule mergeVariableDeclarations
    replace [repeat declaration]
        VAR VarDeclarations1 [repeat var_decl]
        VAR VarDeclarations2 [repeat var_decl]
        OtherDeclarations [repeat declaration]

    % First simply concatenate into one list
    construct NewVarDeclarations [repeat var_decl]
        VarDeclarations1 [. VarDeclarations2]

    % Then use subrule to merge the lists if types are the same
    by
        VAR NewVarDeclarations [mergeSameTypeLists]
        OtherDeclarations
end rule
```

Fig. 8. Replacement Refinement Using Constructs

```
% Eliminate named constants by replacing all references
% with their (compile-time) values

rule resolveConstants
   replace [repeat statement]
      % Capture name and value of constant declaration
      const C [id] = V [expression];
      RestOfScope [repeat statement]
   by
      % Pass them to subrule for expansion
      RestOfScope [replaceByValue C V]
end rule

rule replaceByValue ConstName [id] Value [expression]
   % Expand references given constant name and value
   replace [primary]
      ConstName
   by
      ( Value )
end rule
```

Fig. 9. Subrule Parameters

allows the subrule to restrict its application based on the properties of many different contexts, and generalizes transformation rules to handle transformations based on arbitrary combinations of information spread across the input.

4 User Refinement of the TXL Language

In keeping with the user-oriented design philosophy of the Turing project from which it sprang, TXL was allowed to evolve for some years based on user feedback. In this section we briefly outline some of the language refinements that have come about due to user experience with TXL. With these refinements, the TXL language has been more or less stable since about 1995.

```
% Ruleset to create a new Turing module for a given set of variables
function createModule ModuleId [id] VarsToHide [repeat id]
   replace [repeat statement]
      Scope  [repeat statement]
   by
      Scope [createEmptyModule          ModuleId]
            [hideVarsInModule           ModuleId VarsToHide]
            [createAccessRoutines       ModuleId each VarsToHide]
            [moveRoutinesIntoModule     ModuleId VarsToHide]
            [qualifyExportedReferences ModuleId VarsToHide]
            [createImportExports        ModuleId VarsToHide]
            [relocateModuleInProgram    ModuleId VarsToHide]
end function
```

Fig. 10. Ruleset Abstraction

4.1 Functions and Rulesets

TXL rules by default use the fixed-point compositional semantics of pure rewriting systems. That is, a rule searches its scope for the first instance of its pattern, makes a replacement to create a new scope, and then re-searches the result for the next instance, and so on until it can no longer find a match. In most cases, this is the most general and appropriate semantics for source transformations. However, as TXL began to be used for more and more complex transformations, the limitations of this single rule semantics began to be stretched. In particular, the need for pure (apply once only) functions and for modular rule abstractions was quickly evident.

Both of these needs were met by a single new feature: *functions*. TXL functions act like functions in any other language - they simply match their arguments (i.e., scope and parameter patterns), compute a result value (i.e., make a replacement) and halt. Like rules, TXL functions are *total* - that is, if their pattern does not match then they simply return their unchanged scope as result. With the addition of functions, TXL provides four separate basic transformation semantics: match and transform the entire scope once (a function), match and transform within the scope once (a deep function), match and transform the entire scope many times (a recursive function), and match and transform searching within the scope many times (a rule).

One of the most common uses for functions in TXL is *rule abstraction*, in which a function is used to gather a number of related rules to be applied to a scope together (Figure 10). In TXL such a function is often referred to as a *ruleset*, with the semantics that application of the function to a scope applies the composition of all of the rules in the ruleset. Combinations of functions and rules allow for complex programmed control over application and scoping of transformation rules.

```
% Base case of a vectorizing ruleset
rule vectorizeScalarAssignments
   replace [repeat statement]
       V1 [id] := E1 [expression];
       V2 [id] := E2 [expression];
       RestOfScope [repeat statement]

   % Can only vectorize if independent
   where not
       E2 [references V1]
   where not
       E1 [references V2]

   by
       < V1,V2 > := < E1,E2 > ;
       RestOfScope
end rule
```

```
% Condition rule to check
rule references V [id]
   match [primary]
       V
end rule
```

Fig. 11. A Guarded Rule Using **where**

4.2 Explicit Guards

Complex transformations often require computed constraints on the application of a rule even when the scope matches its pattern. For example, a sorting rule may match pairs of elements of a sequence, but should make its transformation only if the values of the elements are misordered. In general, such constraints may be very complicated, involving significant additional computation or information gathered remotely from other sources.

To meet this need, *where* clauses, which can impose arbitrary additional constraints on the items bound to pattern variables, were added to TXL. Where clauses use a new special kind of TXL rule called a *condition rule*. Condition rules have only a pattern, usually with additional refinements and constraints, but no replacement - they simply succeed or fail (that is, match their pattern and constraints, or not). A number of built-in condition rules provide basic semantic constraints such as numerical and textual value comparison of terminal symbols. Figure 11 shows an example assignment vectorizing rule that uses a simple condition rule to test whether an expression references a variable.

Because condition rules are themselves TXL functions or rules, they may use additional *deconstructs*, *constructs*, subrules, *where* clauses and so on, allowing for arbitrary computation in guards, including tests involving global or external information (Section 4.4).

4.3 Lexical Control

TXL was originally designed to support dialects and experiments with only one language - Turing. For this reason, the lexical rules of Turing were originally

```
% Part of the TXL lexical specification of C
comments
    //
    /* */
end comments

% Token definitions for C-like identifiers, integer numbers, string and
% character literals are predefined in TXL and need not be repeated here
tokens
    hexint    "0[xX][\dAaBbCcDdEeFf]+[LUlu]*"
    dotfloat  ".\d+([eE][+-]?\d+)?[FLfl]?"
    float     "\d+.\d*([eE][+-]?\d+)?[FLfl]?"
          |   "\d+(.\d*)?[eE][+-]?\d+[FLfl]?"
          |   "\d+(.\d*)?([eE][+-]?\d+)?[FLfl]"
    longint   "\d+[LUlu]+"
end tokens

compounds
    ->  ++  --  <<  >>  <=  >=  ==  !=  &&  ||  ...  *=  /=
   '%=  +=  -=  <<=  >>=  &=  ^=  |=  :=  ..  'not=
end compounds

keys
    auto       double     int        struct     break      else
    long       switch     case       enum       register   typedef
    char       extern     return     union      const      float
    short      unsigned   continue   for        signed     void
    default    goto       sizeof     volatile   do         if
    static     while
end keys
```

Fig. 12. Specification of Lexical Rules in TXL

built in to TXL. Once it began to be used more generally for implementing source transformations of other languages such as Pascal, C, and so on, the need to allow for specification of other lexical conventions became clear.

As a result, features were added to TXL to allow specification of lexical rules in terms of *keywords* (reserved identifiers), *compounds* (multi-character sequences to be treated as a unit), *comments* (specification of commenting conventions) and most generally *tokens*, regular expression patterns for arbitrary character sequences (Figure 12). Like nonterminal definitions, token definitions may be ambiguous and are interpreted in the order they are specified, with earlier patterns taking precedence over later.

For some input languages, it is most convenient to work directly at the character level, using the power of the parser to process input directly. This technique, recently known as *scannerless parsing*, has other advantages as well[56]. To facilitate character level processing in TXL, a *char* mode provides for character-by-character parsing of input. When combined with token definitions, this mode allows for parser processing of raw input by either character, line or character class (e.g., alphabetic, numeric, space, etc.).

4.4 Global Variables and Tables

Perhaps the most extensive user addition to the TXL language has been global variables. Many transformation tasks are most conveniently expressed using some kind of symbol table to collect information which is then used as a reference when implementing the transformation rules. Implementation of symbol tables in pure functional languages is problematic, involving passing the structure around explicitly as an additional parameter (although one can hide this using monadic style).

In order to allow TXL to more easily handle this class of transformation and avoid the overhead and inefficiency associated with extra rule parameters and complex guards, global variables were added. TXL globals are modelled after the Linda *blackboard* style of message passing[27]. In this style, bound local variables are *exported* to the global scope by a rule or function for later *import* by some other rule or function. Exported variables may be of any nonterminal type, including new types not related to the main grammar, and when a variable is imported in another rule it must be as the same type.

TXL globals have a great many uses in transformations, but the most common is the original use: symbol tables. Symbol tables in TXL are typically structured as an associative lookup table consisting of a sequence of *(key, information)* pairs. Both the key and the information can be of any nonterminal type, including new types defined solely for the purpose. Often the key is of type `[id]` (i.e., an identifier). TXL deconstructs are used to associatively look up the information given the key (Figure 13). Because they use pattern matching, table lookups are also two-way; if one wants to know the key associated with some information, the deconstruct can just as easily pattern match that way also.

In applications where tables can be large, the linear search implied by the associative lookup of a TXL deconstruct can be prohibitively expensive. TXL programmers address this issue using AVL-tree[1] structured global tables.

With the addition of functions, guards, lexical control and global variables, the TXL language was essentially complete - a general purpose language for programming source transformations. In the rest of this paper we demonstrate this generality by showing how TXL has been able to express new ideas in language processing, source analysis and source transformation.

```
% Simple example global table
% The type of entries (can be anything)
define table_entry
   [stringlit] -> [stringlit]
end define

% Export initial table from main rule
function main
   export Table [repeat table_entry]
      "Veggie" -> "Celery"
      "Veggie" -> "Broccoli"
      "Fruit" -> "Orange"
      "Fruit" -> "Pear"
   replace [program]
      P [program]
   by
      P [Rule1] [Rule2] [Rule3]
end function
```

```
% Updating the global table
function addAsFruit
   match [stringlit]
      NewFruit [stringlit]
   import Table [repeat table_entry]
   export Table
      "Fruit" -> NewFruit
      Table
end function

% Querying the global table
function isAVeggie
   match [stringlit]
      Item [stringlit]
   import Table [repeat table_entry]
   deconstruct * [table_entry] Table
      "Veggie" -> Item
end function
```

Fig. 13. A Global Table in TXL

5 Expressing New Paradigms in TXL

Because of its fully programmable nature, new ideas and paradigms in source manipulation can be experimented with directly by TXL users, without the need to change TXL or its implementation. The interpretive parser means that this applies as well to new ideas in parsing as it does to transformation. In this section we look at a number of recently popular new ideas in grammars, parsing and transformation and their implementation in TXL.

5.1 Robust Parsing

In recent years source code analysis and manipulation techniques have been widely applied to large scale legacy systems written in Cobol, PL/I and RPG. A difficulty with such languages is that they are challenging to parse because of the wide range of dialects, variants, preprocessors and local enhancements. It is frequently the case that analysis tools fail due to a parse error on these differences. In most cases such differences are minor, and the main problem is simply coming up with a parse.

Robust parsing[3] is a method for automatically providing the ability to complete a parse even in the presence of sections of input that cannot be interpreted. The original method for robust parsing involved a customized LL(1) algorithm[4] to correct syntax errors in input by substituting or ignoring a minimal section of input to continue the parse. For example, when coming to a statement of an unrecognized form, the method might simply ignore the input symbols in the statement up to the next semicolon or other end marker.

```
% Example of robust parsing in TXL
% This time for C dialects with strange new statements
include "C.Grm"

% If all statement forms fail, fall throught to unknown
redefine statement
        ...
    |   [unknown_statement]
end redefine

% Accept anything at all before the next semicolon or brace
define unknown_statement
        [repeat not_semicolon_brace]
end define

define not_semicolon_brace
        [not ';] [not '}] [token] % any single token not ; or }
    |   [key]                     % any keyword
end define
```

Fig. 14. Example of Robust Parsing in TXL

Grammar overrides allow the TXL user to directly program robust parsing without any change to the TXL parser. For example, we can extend the nonterminal definition for *statement* to include an additional uninterpreted case that accepts anything at all until the next end of statement marker (Figure 14). This solution takes advantage of two properties of direct interpretation of the grammar: ordered alternatives (because it is the last alternative, the uninterpreted case will never be used unless no other statement form can match) and ambiguity (because the uninterpreted case is ambiguous with respect to all other statement forms).

5.2 Island Grammars

Island grammars[26,38] are a related idea borrowed from natural language processing. Island grammars allow for robust, efficient semi-parsing of very large inputs when we are only interested in parts of them. Island grammars are used to pick out and parse only those items of interest (the *islands*) in a stream of otherwise uninteresting input (the *water*). This idea is extended to multiple levels, in which islands may contain uninterpreted *lakes* which in turn may contain smaller islands and so on. Island parsing is particularly useful when we are interested in only one aspect of a complex input, for example, if we are only interested in processing the embedded ASP aspect of HTML web pages, or if we are only interested in embedded SQL aspect of Cobol programs.

Island grammars can be coded in TXL either directly or as dialects of a base language in which the islands are embedded. Figure 15 shows a TXL grammar that uses an island grammar to process embedded SQL in Cobol programs as uninterpreted lakes (the SQL code) containing interesting islands (SQL refer-

```
% Begin with Cobol
include "Cobol.Grammar"

% Extend to allow SQL
redefine statement
    ...
  | [sql_statement]
end redefine

define sql_statement
    EXEC SQL
      [repeat sql_item]
    [end_exec]
end define

define end_exec
    END-EXEC
end define
```

```
% Use lake and island parsing to parse
% only parts of SQL we're interested in
define sql_item
    [host_variable]
  | [water]
end define

define host_variable
    : [ref_name]
end define

define water
    % Bounded by END-EXEC shoreline
    [not end_exec] [token_or_key]
end define

define token_or_key
    % TXL idiom for "any input"
    [token] | [key]
end define
```

Fig. 15. Island Grammar for Embedded SQL in Cobol (adapted from [25])

ences to Cobol host variables). The key feature in this grammar is the nonterminal modifier *not*. The TXL expression [not end_exec] tells the parser that the following grammatical form must not match the same sequence of tokens that the nonterminal [end_exec] matches. [not] is essentially a lookahead check; it does not consume any input. This prevents the parser from consuming non-SQL tokens in error. In island grammar terminology, this can be thought of as a breakwater that prevents the lake from consuming the shoreline.

5.3 Union Grammars

Due to concerns about "legacy languages" and migration to the world wide web, source-to-source translation has been a very hot topic in recent years. Unlike the language extension tasks for which TXL was designed, this requires transformations that deal with not one language grammar, but two - the *source* language and the *target* language. Moreover, because TXL rules are constrained to be homomorphic (grammatical type preserving), it is not obvious how TXL can serve this kind of multi-grammar task.

One solution is *union grammars*, which mix the nonterminals of the two languages at "meet" points appropriate to natural levels of translation - for example procedures, statements and expressions. In a union grammar, the [statement] nonterminal allows both the input language statement forms and the output target language statement forms, with the parse of input being constrained to the former and the resulting output being constrained to the latter.

```
% Start with both base grammars
include ”Pascal.Grm”
include ”C.Grm”

% In the union we accept either
% kind of program
redefine program
    [pascal_program]
  | [c_program]
end redefine

define pascal_program
  ’program [id] [file_header]
    [repeat decl]
    [block] ’.
end define

define c_program
  [repeat decl]
end define
```

```
% Either kind of block
redefine block
  [begin_or_brace]
    [repeat decl]
    [repeat statement]
  [end_or_brace]
end redefine

define end_or_brace
  ’end | ’}
end define

define begin_or_brace
  ’begin | ’{
end define

% Either kind of if statement
redefine if_statement
  ’if [expression] [opt ’then]
    [statement]
  ’else
    [statement]
end redefine
```

Fig. 16. Part of a Union Grammar for Pascal and C (adapted from [25])

Union grammars can be coded as TXL grammar overrides, for example by redefining the `[statement]` nonterminal to list the input language alternatives first and the output language alternatives second. Because the grammar is directly interpreted in ordered fashion, the parse of the input will be as input language statements even if the output language statements are ambiguously similar. However, because the nonterminal `[statement]` allows both input and output language forms, statement transformation rules can move freely between the two. Figure 16 shows a part of a language translation from Pascal to C using this technique.

5.4 Agile Parsing

Agile parsing[25] refers to the idea of overriding a base grammar to provide a parse more appropriate or convenient to each individual application. This idea can radically simplify software analysis and transformation tasks by using a custom grammar that structures the parse of the input into an ideal form for the task at hand, rather than the usual standard form for the language.

Figure 17 shows a very simple example using agile parsing to identify and isolate the JDBC (database) aspect of Java programs by overriding the grammar to categorize and parse JDBC method calls differently from other method calls. Again, this solution exploits the programmable handling of ambiguity in TXL to modify the grammar to the task. Using the power of the parser to identify items of interest and abstract them into custom grammatical categories

```
% Java base grammar
include "Java.Grm"

% Use parser to identify JDBC calls for us
% (simplified for demonstration purposes)
redefine method_call
       [jdbc_call]
    |  ...
end redefine

define jdbc_call
    [jdbc_name] [arguments]
end define

define jdbc_name
    ‘createStatement | ‘prepareStatement
  | ‘executeUpdate | ‘executeQuery | ‘getRow
end define
```

Fig. 17. Customizing Grammar to Task Using Agile Parsing (adapted from [25])

can significantly reduce the cost and complexity of an analysis ruleset.

5.5 Parse Tree Annotations

Parse tree annotations[45] is an idea that has recently gained new attention in the software re-engineering community[33]. The challenge is to provide the ability to add, preserve and manipulate complex annotations in parse trees in order to allow for concerns such as layout preservation, reversible preprocessing and other separate factors of the source code[36] in reverse- and re-engineering transformations.

TXL's ordered ambiguity makes it easy to specify and manipulate parse tree annotations. Using grammar overrides, optional annotations can be added to nonterminals of an existing base grammar. The annotations can be of any structure at all, specified using new nonterminal definitions, and can be manipulated either separately or together with the items they annotate using standard TXL patterns and replacements.

Figure 18 uses overrides to allow for addition of statistical annotations on method declarations in Java. Normal rules can be used to add or manipulate these annotations. Such annotations can later be gathered (extracted) from the parse tree to form a table of information using TXL's *extract* built-in function and then used in guards on later transformations of the methods or written to a file.

An example application of parse tree annotations is source *fact extraction*, also known as *design recovery*[7,22]. Design recovery analyzes a software system's source to identify and extract a database of data and program *entities* such

```
% Java base grammar
include "Java.Grm"

% Structure of statistical information annotation
% (syntactic sugar optional)
define method_stats
       { [list method_stat] }
end define

define method_stat
       [method_label] = [number]
end define

define method_label
       ‘static_calls | ‘indirect_static_calls
    |  ‘fan_in | ‘fan_out | ‘in_depth | ‘out_depth
end define

% Allow optional statistics annotation on methods
redefine method_declaration
       ...
    |  [method_declaration] [opt method_stats]
end redefine
```

Fig. 18. Parse Tree Annotations

as variables, classes and methods, and the higher level *design relationships* between these entities, such as containment, use, calling, reading, writing or parameterizing of one entity by another. The result is a high level design database representing the actual architecture of the software system.

When it was first proposed to apply TXL to this problem it was not at all obvious how it could be done. TXL’s search and pattern match capabilities could encode the complex interrelationships that indicate the presence of the required relationships, but it had no notion of output of facts representing the result. In retrospect the solution to this is remarkably simple - use grammar overrides to allow for design fact annotations in the source code itself, and then extract the facts when done. Higher level rules and patterns establish the context for each inference, and then annotate the evidence for each relationship with its fact using a local pattern match (Figure 19).

5.6 Source Code Markup and XML

One of the most important new ideas in source code analysis in recent years is the advent of source code markup and the introduction of the standard markup notation XML[14]. From the TXL standpoint, XML is just another language whose grammar can be described, and source code markup is simply another kind of grammar override, so programmers could begin generating and working with XML markup without any change to TXL (Figure 20). TXL’s polymorphism (using the universal nonterminal type [any]) allows for the definition of generic XML markup that can be added to any language

```
% Simple example of design recovery in TXL
rule processProcedureRefs
   % Because this rule’s pattern directly matches its replacement,
   % there is no natural termination point, so we use a one-pass ($) traversal
   replace $ [declaration]
      procedure P [id] ParmList [opt parameter_list]
         Scope [repeat statement]
      'end P
   by
      procedure P ParmList
         Scope [embedProcCalls P]
               [embedFuncCalls P]
               [embedVarParmRefs P]
               [embedPutRefs P]
               [embedGetRefs P]
      'end P
end rule

% Annotate embedded argument uses with design fact giving procedural context
rule embedVarParmRefs ContextId [id]
   replace $ [argument]
      ReferencedId [id] Selectors [repeat selector] : var FormalId [id]
   by
      ReferencedId Selectors : var FormalId [id]
      $ ‘vararguse (ContextId, ReferencedId, FormalId) $
end rule
```

Fig. 19. Design Recovery (adapted from [22])

as an independent subgrammar. Rules to create either full or partial XML markup of simple parse trees or complex source inferences can then be coded in a fashion similar to the inference of facts in design recovery[23].

5.7 Traversals

Control of traversal of the parse tree when applying source transformations can be an important issue. For example, in a transformation that resolves references to declarations, the traversal must proceed from the bottom up, whereas in a transformation that restructures architecture, we normally want to proceed from the top down. Similarly, some transformations should apply only once, some only at a single level and not below, and so on. Both ASF+SDF[6] and Stratego[53,54] provide explicit facilities for defining and using generic traversals to control transformations[12].

In TXL the notion of traversal is in general under programmed user control using standard functional programming style (Figure 21). Traversals are implicitly programmed as part of the functional decomposition of the transformation ruleset, which controls how and in which order subrules are applied. Bottom-up traversal is simply a directly recursive function or rule, apply-once rules are simply TXL functions, single level traversal is explicit recursion on

```
% Simple example of XML markup using TXL

% This time we're marking up C++
include "Cpp.Grm"

% Simplified syntax of XML tags
define xmltag
    < [id] >
end define

define endxmltag
    </ [id] >
end define

% Allow statements to be marked up
redefine expression
        ...
    |  [xmltag] [expression] [endxmltag]
end define

% Example rule to mark up interesting statements
rule markExpressionsUsing InterestingId [id]
    % Mark only outermost expressions, and only once
    skipping [expression]
    replace $ [expression]
        E [expression]
    % It's an interesting one if it uses the interesting thing
    deconstruct * [id] E
        InterestingId
    by
        <interesting> E </interesting>
end rule
```

Fig. 20. Generic XML Source Markup (adapted from [23])

a sequence, and so on. In general, any required traversal can be programmed directly and compactly in traditional recursive functional programming style.

Generic traversals are a major advantage of ASF+SDF and Stratego over TXL, since in TXL traversal paradigms are not generic and must be reused by hand cloning. However it does have the advantage that custom traversals are easily made transformation-sensitive to each application.

5.8 Rewriting Strategies and Scoped Application of Rules

As the sophistication and complexity of source transformation tasks has grown, the necessity of providing some method for limiting the scope of rewrite rules to only a part of the input in response to previous analysis has become increasingly important. One of the important innovations in the recent Stratego language[54] was to address this issue in term rewriting. Stratego uses the powerful notion of *rewriting strategies*[55] for this purpose.

In TXL the scoping of rules (limitation of rewriting to a particular context)

```
function toplevelleftright
    % Left-right top level no rescan
    replace [repeat T]
      Instance [T]
      RightContext [repeat T]
    by
      Instance [dotransform]
      RightContext [toplevelleftright]
end function

rule bottomupleftrightrescan
    % Bottom-up left-right rescan
    replace [repeat T]
      Instance [T]
      RightContext [repeat T]
    by
      Instance [bottomupleftrightrescan]
               [dotransform]
      RightContext
end rule
```

```
rule topdownleftrightrescan
    % Top-down left-right rescan
    replace [T]
      Instance [T]
    by
      Instance [dotransform]
end rule

rule bottomuprightleftrescan
    % Bottom-up right-left with rescan
    replace [repeat T]
      Instance [T]
      RightContext [repeat T]
    construct NewRightContext [repeat T]
      RightContext [bottomuprightleftrescan]
    by
      Instance [bottomuprightleftrescan]
               [dotransform]
      NewRightContext
end rule
```

Fig. 21. Sample Traversal Paradigms

falls out naturally from the functional programming paradigm. TXL functions and rules are applied explicitly to scopes consisting of bound variables selected from the patterns matched by the functions and rules that invoke them. As pure functions these subrules cannot see any other part of the input, and their scope is necessarily limited to the subtree to which they are applied.

In TXL rewriting strategies are intentionally expressed as an integral part of the functional decomposition of the rules. While generalized abstract strategies and traversals are a certainly a valuable and important new concept, TXL has no ability to directly express them in the reusable sense of Stratego. In future it would be natural to address this by adding higher-order functions and rules (using function and rule parameters) to the language. A first implementation of this idea has recently been demonstrated in ETXL[50].

5.9 Contextualized Rules

It is frequently the case that rules need to be parameterized by a previous context, for example in a transformation that inlines functions, traces dataflow or folds expressions. Stratego[54] has recently introduced the notion of *dynamic rules*[10] to address this situation by allowing for rules parameterized by context to be generated and applied on the fly as part of a transformation.

As we have already seen (Figure 9), in the functional programming paradigm of TXL parameters bound from previous contexts in higher level rules or patterns can be explicitly passed to subrules, allowing for arbitrary contextualization in the natural functional programming style.

5.10 Native Patterns

Traditional term rewriting and program transformation tools express their rewriting rules using internal abstract syntax, which can become cumbersome and difficult to understand when patterns are large or complex. For this reason there has been much recent interest in the notion of *native patterns*[49], the idea that patterns and replacements should be expressed in the concrete syntax of the target language, and modern transformation systems such as ASF+SDF and Stratego support this notion. TXL takes the idea to the limit, in that it consistently uses only native patterns in all contexts. Patterns in concrete syntax were of course the original goal of TXL, and the coming of age of the example-based paradigm (which brings us up to date, almost 20 years later).

6 Transformation as a Programming Paradigm

As the range of applications of source transformation languages grows, the role of *transformational programming* as a general purpose computing paradigm for a range of applications becomes an increasingly interesting possibility. TXL has been used in many applications outside the domain of programming languages and software engineering, including VLSI layout, natural language understanding, database migration, network protocol security and many others.

Perhaps the most unusual application of TXL is its recent use in the recognition and analysis of two dimensional mathematical formulas from hand-written graphical input[58]. In this application TXL is used in several stages: to analyze two dimensional image data for baseline structure, to associate symbols into local structural units such as subscripted symbols, to combine these units into higher level mathematical structures such as summations and integrals, to associate meaning with these structures based on domain knowledge, and to render this meaning into equivalent LaTeX formulas and Mathematica or Maple programs. This work has been generalized into a transformational paradigm for diagram recognition tasks[8].

The surprising and highly successful application of TXL to a range of very different problem domains in electrical engineering, artificial intelligence, database applications and so on, and the success of other transformational tools and languages in applications to biology and medicine, lead one to wonder if there are not many other problems for which this paradigm might serve. Work in the TXL project has begun on the next generation of such languages, with the aim of a more generally accessible and usable general purpose transformational programming paradigm. In the meanwhile, we continue to explore the use of TXL itself in a wide range of new and diverse applications.

7 Related Work

Many other tools and languages are similar to TXL in various ways. ASF+SDF [6,11] is a very general toolset for implementing programming language manipulation tools of many kinds, including parsers, transformers, analyzers and many other tools. While it is very different in its methods and implementation, using a GLR parsing algorithm, providing grammar-based modularity and so on, most tasks appropriate to TXL can be expressed in ASF+SDF.

Stratego[53,54] is a modern language aimed at the same kinds of problems as TXL. Stratego augments pure rewriting rules with the separate specification of generic rewriting strategies, an idea adapted from the Elan[9] deduction metasystem. This separation can lead to a more compact and modular transformation specification compared to TXL, although it can be more difficult to see the overall effect of a rule combined with its application strategy. From an execution efficiency standpoint, there is little difference between the two.

Both ASF+SDF and Stratego support the notion of traversal independently of the types to be traversed, whereas in TXL it is most natural to program traversal as an inherent part of the functional decomposition of the rules. Like TXL, both ASF+SDF and Stratego support specification of patterns in concrete syntax, and Stratego's *overlays* support the notion of application-specific pattern abstractions, which play a role somewhat similar to agile parsing in TXL.

ANTLR[39] is an LL-based language manipulation system that grew out of the PCTSS compiler project and is primarily aimed at implementing compilers, interpreters and translators. ANTLR's tree construction and walking capabilities can be used to assist in tasks often done using TXL, and ANTLR's SORCERER[42] tree walker generator can be used to facilitate similar parse tree manipulations, albeit in a radically different way.

TXL's top down parser can be compared to ANTLR's generalized LL and other top-down parsing methods. In particular, the use of Definite Clause Grammars (DCG's)[43] in Prolog bears a resemblance to TXL's backtracking parsing method, including the resolution of left-recursive productions by left factoring, either on-the-fly or using grammar transformations[47]. Functional parsers (also known as *combinatory parsers*[52]) are built by composing elementary parsing functions according to the context-free grammatical structure of the language using a small set of higher-order *parser combinators*, resulting in a pure functional parser that acts very like TXL's direct functional grammar interpretation. However, TXL's parser is purely context-free, leaving context dependencies for later transformation rules, whereas these more general parsing methods can handle context dependencies directly.

Most modern source transformation tools, such as ASF+SDF, Stratego and DMS[5], use generalized LR (GLR) parsers[51]. GLR parsers, and in particular scannerless GLR (SGLR)[56] parsers, have many advantages (e.g., no problems with left recursion) and have been shown to be well suited to rewriting systems. A major difference with the top-down methods is in the handling of ambiguity. Because GLR parsing yields all possible derivations, much effort in these systems is devoted to the problem of disambiguation[13]. By contrast TXL's direct ordered interpretation of the grammar automatically yields a deterministic unambiguous parse in all cases, while still allowing for exploitation of grammatical ambiguity in the rule set.

While grammar overrides are an inherent and convenient feature of TXL, their effect, and agile parsing in general, can also be implemented using generative techniques such as grammar transformation and in particular grammar adaptation[34]. The primary difference is that TXL interprets overrides directly, whereas adaptation generates a whole new grammar for the task. In both cases the original "base" grammar is unaffected by the customization, which is the most important point.

APTS[41] is a very general transformation system based on parse tree rewriting primarily aimed at "transformational programming", the derivation of efficient programs from simple but correct specifications by the application of correctness-preserving transformations. It is particularly well suited to expressing constraint-based transformations. While the notation and control structures of APTS are quite different from TXL, it shares as its basis non-linear tree pattern matching. However, where TXL and most other systems use top-down tree matching, representing broadest-first match, better suited to structural transformation, APTS uses bottom-up tree matching, finding deepest-first match, better suited to program generation tasks. APTS has been used to implement complex algorithms by correctness-preserving transformation from high level specifications to executable C code that can be more efficient than hand-coded Fortran[15,40].

XSLT[17] is the W3C standard for source transformation of XML documents. While not a general purpose source transformation system (and not intended to be one), XSLT nevertheless shares many ideas with TXL and its related systems. In particular, XSLT is a user programmable transformation language, it is primarily a pure functional language, and it uses the notion of pattern-replacement pairs applied in term rewriting style.

Other related work includes Rigal[2], a language for implementing compilers that shares with TXL a list-oriented implementation, transformation functions and non-linear pattern matching, Gentle[48], a comprehensive compiler toolkit that supports source to source transformation, and the commercial DMS toolkit and its Parlanse language[5], which uses a very different paradigm to implement similar software analysis applications. Many other source transformation tools and languages can be found on the program transformation wiki, `http://www.program-transformation.org` .

8 Conclusion

From its roots in experimental language design 20 years ago[28], TXL has grown into a powerful general purpose source transformation programming system. It has been used in a wide range of applications, including industrial transformations involving billions of lines of source code. TXL's flexible general purpose functional programming style distinguishes it from most other source to source transformation systems in that it leaves all control over parsing and transformation in the hands of the programmer. While not without its drawbacks, this flexibility has proven very general, allowing TXL users to express and experiment with evolving new ideas in parsing and transformation on their own, without the necessity of moving to new languages and tools.

9 Acknowledgments

The original *Turing eXtender Language*[28] was designed by Charles Halpern-Hamu and the author at the University of Toronto in 1985, and the first practical implementations were developed by Eric Promislow[44] and Ian Carmichael [16] at Queen's University between 1986 and 1990. The modern TXL language [18] was designed and implemented by the author at GMD Karlsruhe and Queen's University between 1990 and 1995. Andrew Malton developed the formal semantics of TXL[35] at Queen's University in 1993. Development of TXL has been funded by the Natural Sciences and Engineering Research Council of Canada, Communications and Information Technology Ontario, ESPRIT project REX, GMD Karlsruhe, the University of Toronto, Queen's University and IBM Research. This paper has been technically corrected and significantly improved by the thorough and detailed reviews of the three referees and the insightful guidance of the guest editors, for which I thank them.

References

[1] G. Adelson-Velskii and E. Landis, "An Algorithm for the Organization of the Information", *Soviet Mathematics Dokay* **3**, 1259-1263 (1962).

[2] M. Auguston, “RIGAL - A Programming Language for Compiler Writing”, *Lecture Notes in Computer Science* **502**, 529–564 (1991).

[3] D.T. Barnard and R.C. Holt, “Hierarchic Syntax Error Repair for LR Grammars”, *International Journal of Computing and Information Sciences* **11**(4), 231–258 (1982).

[4] D.T. Barnard, “Automatic Generation of Syntax-Repairing and Paragraphing Parsers”, Technical Report CSRG-52, Computer Systems Research Group, University of Toronto, 132 pp. (1975).

[5] I.D. Baxter, “Parallel Support for Source Code Analysis and Modification”, Proc. IEEE 2nd International Workshop on Source Code Analysis and Manipulation, 3–15 (2002).

[6] J.A. Bergstra, J. Heering and P. Klint, *Algebraic Specification*, ACM (1989).

[7] T. J. Biggerstaff, “Design Recovery for Maintenance and Reuse”, *IEEE Computer* **22**(7), 36–49 (1989).

[8] D. Blostein, J.R. Cordy and R. Zanibbi, “Applying Compiler Techniques to Diagram Recognition”, Proc. 16th IAPR International Conference on Pattern Recognition, **3**, 127–130 (2002).

[9] P. Borovansky, C. Kirchner, H. Kirchner, P.E. Moreau and C. Ringeissen, “An Overview of ELAN”, Proc. 2nd International Workshop on Rewriting Logic and its Applications (WRLA’98), *Electronic Notes in Theoretical Computer Science* **15** 55–70 (1998).

[10] M. Bravenboer, A. van Dam, K. Olmos and E. Visser, “Program Transformation with Dynamically Scoped Rewrite Rules”, *Fundamenta Informaticae* **69**(1) 1–56 (2005).

[11] M. van den Brand, J. Heering, P. Klint and P.A. Olivier, “Compiling Language Definitions: the ASF+SDF Compiler”, *ACM Transactions on Programming Languages and Systems* **24**(4), 334–368 (2002).

[12] M. van den Brand, P. Klint and J.J. Vinju, “Term Rewriting with Traversal Functions”, *ACM Transactions on Software Engineering and Methodology* **12**(2), 152–190 (2003).

[13] M. van den Brand, J. Scheerder, J.J. Vinju and E. Visser, “Disambiguation Filters for Scannerless Generalized LR Parsers”, Proc. 11th International Conference on Compiler Construction, 143–158 (2002).

[14] T. Bray, A. Paoli and C.M. Sperberg-McQueen (*eds.*), *Extensible Markup Language (XML) 1.0*, `http://www.w3.org/TR/1998/REC-xml-19980210.pdf` (1998).

[15] J. Cai and R. Paige, “Towards Increased Productivity of Algorithm Implementation”, *ACM Software Engineering Notes* **18**(5), 71–78 (1993).

[16] I. Carmichael, “TXL: Experiments with Pattern-Directed Tree Transformation as a Programming Paradigm”, M.Sc. thesis, Department of Computing and Information Science, Qujeen’s University at Kingston (1990).

[17] J. Clark (ed.), *XSL Transformations (XSLT) Version 1.0’*, W3C Recommendation, `http://www.w3.org/TR/1999/REC-xslt-19991116` (1999).

[18] J.R. Cordy, I.H. Carmichael and R. Halliday, *The TXL Programming Language*, Queen’s University at Kingston (1988, rev. 2005).

[19] J.R. Cordy and E.M. Promislow, “Specification and Automatic Prototype Implementation of Polymorphic Objects in Turing Using the TXL Dialect Processor”, Proc. 1990 IEEE International Conference on Computer Languages, 145–154 (1990).

[20] J.R. Cordy, C.D, Halpern and E. Promislow, “TXL: A Rapid Prototyping System for Programming Language Dialects”, *Computer Languages* **16**(1), 97–107 (1991).

[21] J.R. Cordy, T.R. Dean, A.J. Malton and K.A. Schneider, “Source Transformation in Software Engineering using the TXL Transformation System”, *Journal of Information and Software Technology* **44**(13), 827–837 (2002).

[22] J.R. Cordy and K.A. Schneider, “Architectural Design Recovery Using Source Transformation”, Proc. CASE’95 Workshop on Software Architecture, (1995).

[23] J.R. Cordy, “Generalized Selective XML Markup of Source Code Using Agile Parsing”, Proc. IEEE 11th International Workshop on Program Comprehension, 144-153 (2003).

[24] T.R. Dean, J.R. Cordy, K.A. Schneider and A.J. Malton, “Experience Using Design Recovery Techniques to Transform Legacy Systems”, Proc. 2001 IEEE International Conference on Software Maintenance, 622-631 (2001).

[25] T.R. Dean, J.R. Cordy, A.J. Malton and K.A. Schneider, “Agile Parsing in TXL”, *Journal of Automated Software Engineering* **10**(4), 311–336 (2003).

[26] A. van Deursen and T. Kuipers, “Building Documentation Generators”, Proc. 1999 International Conference on Software Maintenance, 40–49 (1999).

[27] G. Gelernter, “Generative Communication in Linda”, *ACM Transactions on Programming Languages and Systems* **7**(1), 80-112 (1985).

[28] C. Halpern, “TXL: A Rapid Prototyping Tool for Programming Language Design”, M.Sc. thesis, Department of Computer Science, University of Toronto (1986).

[29] R.C. Holt and J.R. Cordy, “The Turing Programming Language”, *Communications of the ACM* **31**(12), 1410–1423 (1988).

[30] R.C. Holt, P.A. Matthews, J.A. Rosselet and J.R. Cordy, *The Turing Programming Language - Design and Definition*, Prentice-Hall (1987).

[31] M.A. Jenkins, “Q’Nial: A Portable Interpreter for the Nested Interactive Array Language, Nial”, *Software - Practice and Experience* **19**(2), 111–126 (1989).

[32] E. Kohlbecker, “Using MkMac”*,* Computer Science Technical Report 157, Indiana University (1984).

[33] J. Kort and R. Laemmel, “Parse-Tree Annotations Meet Re-Engineering Concerns”, Proc. IEEE 3rd International Workshop on Source Code Analysis and Manipulation, 161–171 (2003).

[34] R. Laemmel, “Grammar Adaptation”, Proc. International Symposium on Formal Methods Europe (FME 2001), *Lecture Notes in Computer Science* **2021** 550-570 (2001).

[35] A.J. Malton, “The Denotational Semantics of a Functional Tree Manipulation Language”, *Computer Languages* **19**(3), 157–168 (1993).

[36] A.J. Malton, K.A. Schneider, J.R. Cordy, T.R. Dean, D. Cousineau and J. Reynolds, “Processing Software Source Text in Automated Design Recovery and Transformation”, Proc. IEEE 9th International Workshop on Program Comprehension, 127-134 (2001).

[37] J. McCarthy *et al.*, *LISP 1.5 Programmer’s Manual*, MIT Press (1962).

[38] L. Moonen, “Generating Robust Parsers using Island Grammars”, Proc. IEEE 8th Working Conference on Reverse Engineering, 13–22 (2001).

[39] T.J. Parr and R. W. Quong, “ANTLR: A Predicated LL(k) Parser Generator,” *Software, Practice and Experience* **25**(7), 789–810 (1995).

[40] R. Paige, “Viewing a Program Transformation System at Work”, Proc. Joint 6th International Conference Programming Language Implementation and Logic Programming, and 4th International Conference on Algebraic and Logic Programming, *Lecture Notes in Computer Science* **844**, 5–24 (1991).

[41] R. Paige, “APTS External Specification Manual”, Unpublished manuscript, available at `http://www.cs.nyu.edu/~jessie` (1993).

[42] T.J. Parr, “An Overview of SORCERER: A Simple Tree-parser Generator”, Technical Report, `http://www.antlr.org/papers/sorcerer.ps` (1994).

[43] F. Pereira and D. Warren, “Definite Clause Grammars for Language Analysis - A Survey of the Formalism and a Comparison with Augmented Transition Networks”, *Artificial Intelligence* **13**(3), 231–278 (1980).

[44] E. Promislow, “A Run-Time Model for Generating Semantic Transformations from Syntactic Specifications”, M.Sc. thesis, Department of Computing and Information Science, Qujeen’s University at Kingston (1990).

[45] J.J. Purtilo and J.R. Callahan, “Parse-Tree Annotations”, *Communications of the ACM* **32**(12), 1467–1477 (1989).

[46] R. Ramesh and I. V. Ramakrishnan, “Nonlinear Pattern Matching in Trees”, *Journal of the ACM* **39**(2): 295–316 (1992).

[47] J. Sarbo, “Grammar Transformations for Optimizing Backtrack Parsers”, *Computer Languages* **20**(2), 89–100 (1994).

[48] F. Schroer, *The GENTLE Compiler Construction System*, Oldenbourg (1997).

[49] M.P.A. Selink and C. Verhoef, “Native Patterns”, Proc. IEEE 5th Working Conference on Reverse Engineering, 89–103 (1998).

[50] A. Thurston, “Evolving TXL”, M.Sc. thesis, School of Computing, Queen’s University at Kingston (2005).

[51] M. Tomita, “An Efficient Augmented Context-free Parsing Algorithm”, *Computational Linguistics* **13**(1–2), 31–46 (1987).

[52] K. Vijay-Shanker and D. Weir, “Polynomial Time Parsing of Combinatory Categorical Grammars”, Proc. 28th International Meeting of the Association for Computational Linguistics,1–8 (1990).

[53] E. Visser, “Stratego: A Language for Program Transformation based on Rewriting Strategies”, Proc. Rewriting Techniques and Applications (RTA’01), *Lecture Notes in Computer Science* **2051**, 357–361 (2001).

[54] E. Visser, “Program Transformation in Stratego/XT: Rules, Strategies, Tools and Systems in Stratego XT/0.9”, Proc. Domain Specific Program Generation 2003, *Lecture Notes in Computer Science* **3016**, 216–238 (2004).

[55] E. Visser, Z. Benaissa and A. Tolmach, “Building Program Optimzers with Rewriting Strategies”, Proc. ACM 3rd SIGPLAN International Conference on Functional Programming (ICFP’98), 13–26 (1998).

[56] E. Visser, “Scannerless generalized-LR parsing”, Technical Report P9707, Programming Research Group, University of Amsterdam (1997).

[57] G.M. Weinberg, *The Psychology of Computer Programming*, Dorset House (1971).

[58] R. Zanibbi, D. Blostein and J.R. Cordy, “Recognizing Mathematical Expressions Using Tree Transformation”, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **24**(11), 1455–1467 (2002).