EVOLVING TXL

by

ADRIAN THURSTON

A thesis submitted to the

School of Computing

in conformity with the requirements for

the degree of Master of Science

Queen's University

Kingston, Ontario, Canada

January 2005

Copyright © Adrian Thurston, 2005

Abstract

This thesis is about evolving the TXL programming language. TXL was originally designed for small transformation tasks in support of an experimental approach to language design. Since its introduction, there has been a trend towards using TXL in large software analysis and renovation systems whose development require collaboration and maintainability. Also, there has been increasing diversity in TXL's application domain. Users have discovered applications unforeseen by the original designers. In response to application growth TXL must itself grow. This thesis presents nine enhancements to TXL that address problems that arise in TXL programming. We call the resulting language ETXL. Each change to the language is individually motivated using examples that detail what aspect of TXL programming we are targeting for improvement. We present the design of each solution, give the syntax and informally define the semantics. We provide examples of how the problems that motivated us are now solved in the new language and present some additional solutions that are gained by combining features. We introduce our fully functional prototype and explain how each change has been implemented. Finally, we present ETXL solutions to two problems that are representative of TXL's current usage.

Acknowledgments

I would like to extend my thanks to my supervisor Dr. James Cordy for lending his direction, deep experience, ideas and insight to the production of this work, to colleagues Richard Zanibbi, Jeremy Bradbury, Dean Jin and Derek Shimozawa for advice, direction and help with all kinds of problems, to my father, Paul for proof reading, to my entire family for their love and support, and to my friends for the good times that were had in parallel to the course of this work.

The original version of TXL was designed by Dr. Charles Halpern and Dr. Cordy. Over the years, contributions have been made by many individuals. Modern TXL was designed and developed by Dr. Cordy. To the designers and the many contributors I am grateful for the existence of an extremely enjoyable and useful language. The opportunity to base my investigation into language design on a language with an impressive history in academia and industry as well as a devout user base is indeed a privilege.

Contents

Abstract **i**

Acknowledgments **ii**

Contents **iii**

List of Figures **vii**

1 Introduction **1**

1.1 Introduction 1

1.2 Problems and Solutions 4

1.3 Thesis Outline 7

1.4 Terms and Conventions 8

2 Program Transformation Systems **9**

2.1 ASF+SDF 9

2.2 ELAN 12

2.3 Stratego Language 14

2.4 ANTLR Tree Parser Generator 16

2.5 TXL 17

2.6 Summary 23

3 Motivation **25**

3.1 Must-Match Rules 25

3.2 Objectless Rules 27

3.3 Strong Typing 28

3.4 Nested Rules 31

3.5 Rule Parameters 33

3.6 Type Parameters 35

3.7 If Clauses 36

3.8 Out Parameters 39

3.9 Modularity 41

3.10 Summary 43

4 Design **46**

4.1 Must-Match Rules 47

4.2 Objectless Rules 48

4.3 Strong Typing 50

4.4 Nested Rules 51

4.5 Rule Parameters 55

4.6 Type Parameters 58

4.7 If Clauses 61

4.8 Out Parameters 65

4.9 Scoping and Forward Variable Declarations 70

4.10 Modularity 73

4.11 Summary 76

5 Implementation **79**

5.1 Must-Match Rules 81

5.2 Objectless Rules 82

5.3 Strong Typing 83

5.4 Nested Rules 83

5.5 Rule and Type Parameters 84

5.6 If Clauses 87

5.7 Out Parameters 89

5.8 Scoping and Forward Variable Declarations 93

5.9 Modularity 94

5.10 Summary 97

6 Examples **99**

6.1 Expand Vector Components 99

6.2 Generic AVL Tree Module 102

6.3 Summary 106

7 Conclusion **108**

7.1 Summary 108

7.2 Future Work 111

7.3 Other Changes 113

7.4 Conclusion 115

A Unified ETXL Grammar **116**

B Example: Expand Vector Components **128**

C Example: AVL Tree Module **133**

Bibliography **141**

List of Figures

2.1 ASF+SDF Example 11

2.2 ELAN Example 13

2.3 Common Strategies in Stratego 15

2.4 ANTLR Tree Parser Example 17

2.5 TXL Grammar Example 20

2.6 TXL Ruleset Example 21

2.7 TXL Input and Output Example 22

3.1 Multiple-Stage Transformation 26

3.2 Unused Main Pattern 29

3.3 Rule Valid at Only One Root 30

3.4 Propagation of Variables 32

3.5 Custom Tree Traversal 34

3.6 List Reversal 36

3.7 Incorrect Branching 38

3.8 Variations of a Pattern 40

3.9 Reusable HTML Markup Code 42

4.1 Must-Match Rule Syntax 47

4.2 Must-Match Rule Example 48
4.3 Objectless Rule Example 49
4.4 Capturing Rule Failure 50
4.5 Strong Typing Syntax 51
4.6 Capturing Intent with Strong Typing 52
4.7 Nested Rule Syntax 52
4.8 Nested Rule Example 54
4.9 Rule Parameter Syntax 55
4.10 Generic Binary Search Tree Traversal 57
4.11 Type Parameter Syntax 59
4.12 Generic Sorting 60
4.13 If Clause Syntax 62
4.14 Binary Search Using If Clause 62
4.15 Incomplete Rule 63
4.16 If Test Indentation 65
4.17 Out Parameter Syntax 67
4.18 Abstracted Deconstruction 68
4.19 Binary Search Using Out Parameters 68
4.20 Parameterized Patterns 70
4.21 Forward Variable Declaration Syntax 71
4.22 Forward Variable Declaration Example 71
4.23 Ambiguous Pattern Binding 72
4.24 Module Syntax 74
4.25 HTML Markup Module 75

5.1 Implementation Architecture 80
5.2 Must-Match Rule Implementation 82
5.3 Nested Rule Implementation 85
5.4 Generic Sorting Implementation 86
5.5 If Clause Implementation 88
5.6 Manual Failing of a Pattern 90
5.7 Out Parameter Implementation 92
5.8 Scoping Implementation 94
5.9 Modularity Implementation 96

6.1 Type Conversion 100
6.2 Extracting Initializations from Declarations 101
6.3 AVL Module Find Function 103
6.4 Private Max Function 106

Chapter 1

Introduction

1.1 Introduction

Source analysis and transformation systems are characterized by an ability to define a grammar and a set of rules. Such a system constructs a parser from the user-supplied grammar, runs the parser on the program input to construct an internal representation, applies the rules describing the intended analysis or transformation, and finally writes out the transformed structure. Though principally used for operating on programming languages, source analysis and transformation systems are not restricted to processing source code. They can in fact be used to perform structural analysis or transformation on any data that can be represented by a tree structure.

One such system is the TXL programming language [Cordy04]. User input is parsed according to a free-form extended BNF-like grammar, the parsed structure is processed by a program given in TXL's rule specification language, and the output is guided by formatting cues embedded in the grammar. In this thesis we study the rule-based analysis and transformation phase of TXL.

TXL's rule specification system is a hybrid rule and functional programming language. At the heart of it is the by-example pattern and replacement rewrite system. Built on top of that is a purely functional programming language in which users can specify program logic to solve programming problems. Though not always obvious upon first impression, TXL is a Turing-complete programming language.

The TXL language was originally intended for small programming language transformation tasks [Cordy88, Cordy90]. Since its inception, TXL has matured from a tool for experimenting with language variations to a development environment in which large software renovation programs [Dean01] and language implementations are written. The applications to which users apply TXL have grown from hundreds of lines to many thousands of lines. When using TXL for such applications it becomes clear that it was not intended to be used as the main environment in which a large solution is coded. Provisions for addressing the maintainability requirements of huge TXL code bases on which multiple developers collaborate were never made. Large applications have put TXL's design to the test and it is clear that we must introduce software engineering practices such as modularity and generic programming into the TXL language in order to support the writing of maintainable large-scale applications.

Over the years TXL has also seen a growth in its application domain. Originally intended to aid in the development of the Turing programming language, TXL is now used for multilingual parsing [Syn03], web page analysis [Ric01], handwritten formulae recognition [Zan02], speech understanding [Cet96], program optimization and translation between programming languages [Cordy04]. TXL is also well suited to many tasks in software engineering [Cordy02], including design recovery [Dean02, Lin03], design pattern analysis [Rad99], metaprogramming, refactoring [Grant03], and maintenance. These many new

uses for TXL have certainly put a strain on its existing feature set and now TXL requires help from the very techniques that it supports.

This thesis is about improving the TXL programming language to meet the needs of growing application sizes and an expanding application domain. As TXL's uses evolve, so must the language in order to keep pace. We wish to make TXL a more useful tool, to cater to the needs of existing users and to gain new users.

We present a series of changes to the TXL programming language. Each enhancement introduces a new paradigm to TXL that should enable the writing of more intuitive, more easily understood, or more maintainable TXL programs. The end result will be the next iteration in the evolution of TXL as a source code analysis and transformation programming language. We call the new language we have developed ETXL.

In our approach we will forego strict syntactical backward compatibility. Doing so will give us the freedom to choose the most natural syntax, even if it invalidates some existing programs. We do, however, strive to preserve the existing semantics. We must not render any existing programs inexpressible and we must not reduce functionality. We aim to choose our language changes carefully, such that any existing programs we have invalidated are easily ported to the new syntax.

Once digesting our design, the reader should see that porting a TXL program to ETXL mostly involves escaping the new keywords. There are some scenarios that require special attention, for example an unescaped colon can no longer be passed as an argument because a colon is now used to separate in parameters from out parameters. In practice these scenarios occur infrequently.

Each language change described in this thesis has been prototyped using TXL itself. The design and implementation of new features were done in parallel. This enabled us to

experiment and acquire instant feedback on the utility of our design changes. Once design flaws have been worked out and it is clear that the new changes are fit for production, they can be incorporated directly into the TXL engine. In the meantime, it is our hope that TXL programmers will try out the prototype, which is available from the project's home page [1]. Feedback on the prototype is encouraged.

1.2 Problems and Solutions

In general, we wish to improve TXL in response to its growing use. On a closer look, we have some more specific motivations for improving the language. Our changes may be roughly characterized as follows.

Abstraction and Modularity

Directly tied into the fact that TXL is increasingly used for larger projects that require collaboration is an observed need for generic programming, abstraction and modularity facilities. When developing large systems, code reuse and organization is important to developers. We wish to promote code reuse in TXL by adding language constructs that permit the writing of generic data structures and algorithms. We want a mechanism for grouping code into logical components. Our work on rule, type and out parameters as well as on modularity supports this need.

General Purpose Language Features

Some changes are centered around taking proven characteristics of general-purpose languages and incorporating them into the unique TXL environment. We strive to introduce

features that TXL programmers expect to see based on their previous programming experience and incorporate them into TXL within the bounds of its unique semantics. Doing so should make the language more accessible. For example, we are interested in increasing the type safety of the language and we want to better support complex decision making.

Increasing Expressiveness

Other parts of this work have to do with increasing the overall expressiveness of the language. We aim to provide users with the proper tools to express their intent. We believe that the more a developer's intentions can be captured by a programming language the better the language is. A programming environment that is better informed is able to do more for the programmer in the form of managing the mundane but very necessary details of a complex problem. As expressiveness increases, programmers become more confident that their program accurately depicts the ideal solution to their problem. In a sense this is trying to narrow the gap between expected program behaviour and actual program behaviour.

Minimizing use of Globals

Some changes presented in this thesis are motivated by a desire to minimize the use of global variables for storing data that is not global in nature. For example, modularity enables global variables that are used only in a subset of the program to be made private to the relevant code. The introduction of out parameters helps to eradicate the use of global variables used strictly to transfer data between rules. Global variables used for these purposes inappropriately use up names in the global name space and since they never go out of scope they are not garbage collected. Using globals for data transfer results in wasted memory.

Supporting TXL's Unique Paradigms

On one hand we wish to introduce features present in general-purpose languages and thereby make TXL more accessible, but on the other hand TXL is a unique programming language and we have no intention of grinding away the characteristics that make it stand out as a useful tool. Therefore we also aim to support the unique paradigms of the TXL language. For example, we have noticed that in TXL many variables are often propagated between rule invocations due to the fact that TXL programs are centred around deconstructing and reconstructing parse trees. To accommodate this programming pattern we have added nested rules so that users may take advantage of implicitly passed variables.

Use-Driven Design

We strive to adapt the language to reflect the ways in which it is used in practice. It is true that a language is best designed by its users. Language designers are in fact the principal users of a language and therefore drive its design, however, they are not the only users, and so should not be the only designers. Regular users with specific applications in mind often come up with ways of using a language that were not foreseen by the original designer and so features that support such applications do not exist. Best practices evolve and bring to light language deficiencies. Some language changes described in this thesis have come about from noticing how the language is used in practice and adjusting it to reflect those conventions.

Support

Some language changes we propose are in support of the new features we have introduced and do not constitute new features on their own. They are, however, necessary elements of

the new design and serve to tie together the mix of features we have proposed. For example, variable name scoping and forward variable declarations have been introduced in support of the block structure that if clauses and nested rules introduce.

1.3 Thesis Outline

In order to get a better feel for how the proposed changes fit into the transformational programming paradigm as a whole, background research into other transformation systems was performed. TXL indeed has competitors and our work on modularity for instance has been partially motivated by the existence of modularity features in other source transformation systems. Chapter 2 surveys existing source transformation systems, including TXL.

In preparation for this work, considerable study of TXL syntax and semantics [Cordy03] was performed. An in-depth knowledge of TXL must be obtained before suggesting language changes in order to avoid proposing features whose functionality are already covered by existing features. Background study in this regard is not directly discussed in Chapter 2, although it is evident throughout in the form of assertions and arguments regarding the syntax and semantics of the language.

Chapter 3 brings to light in detail the motivation behind each language change. We look at the current limitations of the language and show why a change is necessary. Chapter 4 proposes each language change, summarizes the syntax and semantics and highlights example uses. Also, any associated implications or problems are discussed. The prototype implementation of each feature is discussed in Chapter 5.

Chapter 6 discusses two real-world examples of ETXL programs that make use of the language features we have developed. The first is an optimization problem that is typical

of the kind of problems that TXL is used for. It expands C++ expressions involving vectors into multiple instances, one for each vector component. The second example demonstrates the use of new features in the creation of a reusable dictionary module that is able to store any type of data, while retaining a type-safe interface.

Chapter 7 summarizes this work, discusses future directions for the design of ETXL and mentions some other changes that were not included in the design. The appendices contain the full ETXL grammar as well as the full listings of the examples described in Chapter 6.

1.4 Terms and Conventions

In this thesis the term *parameter* is taken to mean the named entity that exists inside of a rule. The term *argument* is taken to mean the value passed to a rule invocation. We say that a value is passed as an argument and received as a parameter.

The term *scope* has two uses in this thesis. It is used to refer to the parse tree to which a rule is applied. Here we say the scope of a rule and refer to the parse tree which the rule can act on. Secondly, we use it to refer to the subset of a rule's clauses in which a particular variable is visible. Here we say the scope of a variable and refer to the set of clauses in which the variable may be referenced.

Chapter 2

Program Transformation Systems

This chapter reviews other source analysis and transformation systems, then gives a brief introduction to TXL. The other systems mentioned here all differ considerably in appearance from TXL, yet at their core, they are quite similar to TXL. Common between all systems is the process of matching instances of structured patterns and possibly replacing them with new tree constructions. The differences between these systems are in how the grammar and transformation facilities are expressed, controlled and implemented.

2.1 ASF+SDF

ASF+SDF [Bran02, Bran03] is a transformation system based on term rewriting. Term rewriting is directly analogous to tree transformation, a term simply meaning a node in a parse tree. SDF is the syntax definition component of the system. It allows the programmer to specify the lexical, context-free, and abstract syntax of the language to be processed. ASF is the algebraic specification formalism, which is the rule-based term rewriting component. Programming transformations in ASF entails the composition of a set of rewrite

rules in the form of `LHS = RHS` equations.

The grammar is specified in a form that has the syntax of each production given on the left hand side and the type given on the right. This is a notable difference from most language description tools. If the user's grammar is ambiguous, it is possible to resolve the ambiguities using priority and associativity attributes on productions.

The rewrite rules specify how to make reductions, also known as simplifications. In term rewriting this refers to matching a pattern and making a replacement. Each rule is composed of a simple equation. On the left hand side is the pattern to match and on the right hand side is the replacement. The left hand side may bind variables that may be used on either side of the equation. If used on the left hand side they will require equality of expressions. If used on the right hand side they will transfer an expression to the replacement. Once a set of term rewriting rules have been declared, program transformation becomes the problem of the simplification of terms until no more rules can succeed and a normal form is achieved.

Figure 2.1 shows the use of ASF+SDF to compute the derivative of an expression with respect to a specific variable. The comments in the example highlight the various sections of the specification. These sections are common ingredients in all program transformation systems.

In the domain of program transformation, the repeated application of all rules to a parse tree until a normal form is achieved is not very useful. We need an ability to specify to which sections of a tree certain rules should be applied. This effect can be achieved in ASF+SDF by defining custom grammar productions that encapsulate a set of types with a new type and then targeting rewrite rules to match the new type. By constructing the new type in a rule's RHS, we are able to invoke the collection of rules by name. This allows us

Figure 2.1: ASF+SDF Example

```
%% The token definitions.
lexical syntax
   [0-9]+  -> nat
   [XYZ]   -> var

%% The grammar.
context-free syntax
   nat                 -> expr
   var                 -> expr
   expr "+" expr       -> expr {left}
   expr "*" expr       -> expr {left}
   "(" expr ")"        -> expr {bracket}
   "d" expr "/" "d" var -> expr

%% Priority specification for resolving the ambiguity
%% inherent in the grammar.
context-free priorities
   expr "*" expr -> expr > expr "+" expr -> expr

%% The types of the variables used in rewrite rules.
hiddens
   variables
      "N"       -> nat
      "V"[0-9]* -> var
      "E"[0-9]* -> expr

%% The rewrite rules.
equations
   [ 1] dN/dV = 0                     [ 2] dV/dV = 1
   [ 3] V1 != V2 ==> dV1/dV2 = 0
   [ 4] d(E1+E2)/dV = dE1/dV + dE2/dV
   [ 5] d(E1*E2)/dV = dE1/dV * E2 + E1 * dE2/dV
   [ 6] E + 0 = E                     [ 7] 0 + E = E
   [ 8] E * 1 = E                     [ 9] 1 * E = E
   [10] 0 * E = 0                     [11] E * 0 = 0
```

Use of ASF+SDF to compute the derivative of an expression with respect to a specific variable. In this example all rules are repeatedly applied until a normal form is arrived at, yielding the solution.

to localize the application of rules to specific subtrees.

To further control the rewrite process, ASF provides a mechanism for attaching conditions to rules. Conditions can be used to refine the pattern or to construct values that are used in the replacement. Rule labelling and conditions are useful in controlling the rewrite process, however they are somewhat lacking with respect to flexibility and generality. ASF+SDF supports some standard tree traversals for visiting nodes of a parse tree in a predictable order, however tree traversals other than standard top-down and bottom-up traversals and their variants must be explicitly programmed using the types involved. Therefore other means of controlling the rewrite process are sought. Other systems, notably ELAN and Stratego, aim to fix this deficiency.

2.2 ELAN

ELAN [Boro98] is a rule-based programming language designed for developing theorem provers, logic-based programming languages, constraint solvers and decision procedures. ELAN was not specifically designed for writing program transformations, however it is useful in the problem domain. One of the first applications of ELAN was to prototype a functional programming language.

The syntax definition component of ELAN is similar to SDF. An example is shown in Figure 2.2. Types, known as sorts in the rewrite literature, are defined using a format that is slightly different from SDF. Instead of the name of a type appearing in the syntax definition of a production, the symbol `@` is used. Following the syntax definition, in parenthesis, are the corresponding names of the types used. Then comes the type of the production itself. Finally, after the production type, attributes that guide the parsing process may be given. As in SDF, these attributes are priority and associativity directives.

Figure 2.2: ELAN Example

```
// Syntax definition.
operators global
   x : variable;
   y : variable;
   z : variable;
   @ : ( variable ) expr pri 3;
   @ : ( int ) expr pri 3;
   ( @ ) : ( expr ) expr;
   @ * @ : ( expr expr ) expr assocRight pri 2;
   @ + @ : ( expr expr ) expr assocRight pri 1;
end

// Types of variables used in rewrite rules.
rules for expr
   N, N1, N2 : int;
   V : variable;

// Rewrite rules.
global
   [] N1 + N2 => N where N:=() N1 + N2 end
   [] N1 * N2 => N where N:=() N1 * N2 end
   [] ( V ) => V end
   [] ( N ) => N end
end
end
```

Use of ELAN to simplify operations on constants in arithmetic expressions.

ELAN rewrite rules very much resemble those of ASF. Rules have a pattern on the left hand side with corresponding replacement on the right hand side. They may be followed by conditions which can be used to further refine the rules or to construct values used in the replacement. They may also be labelled so they can be explicitly referenced in other rules.

From a program transformation point of view, ELAN is very similar to ASF+SDF. We mention it here because of one important difference. It introduced a novel concept to rewrite systems which has since made its way into rewrite systems targeted towards

program transformation.

The contribution of ELAN was motivated by a need to manage rule applications. Large rewrite applications usually have many sections and subtasks. An ability to say for certain what is the order of operations in a large program is certainly necessary. Also, since rules in a rewrite system may be written such that more than one rule may succeed at rewriting any given input, a collection of rewrite rules may yield more than one result. Such programs require an ability to declare that certain rule applications should happen before others in order to narrow down the set of results to those that are desired.

ELAN addresses this problem of managing rule applications by introducing a new entity into rewrite systems called a strategy. Strategies bring order to the otherwise non-deterministic process of making rule applications. They allow the programmer to specify how rules may be combined together. They can be used to explicitly declare how trees are walked and which rules are applied during a walk. ELAN is not specifically targeted towards program transformation, however strategies in ELAN provided the motivation for their introduction into program transformation rewriting, via the Stratego language. Strategies are discussed further in the next section covering the Stratego language.

2.3 Stratego Language

Recognizing the need for better control of the application of rewrite rules in rule-based program transformation systems, the Stratego language [Vis04] introduces the concept of rewrite strategies to rule-based program transformation. Stratego is a direct descendant of ASF+SDF and ELAN, combining the program transformation facilities of ASF+SDF with the strict separation of rules from strategies that was introduced by ELAN.

Stratego uses the SDF component of ASF+SDF for defining syntax. Rule syntax also

Figure 2.3: Common Strategies in Stratego

```
module traversals
import lists
strategies
    try(s)            = s <+ id
    repeat(s)         = rec x(try(s; x))
    topdown(s)        = rec x(s; all(x))
    bottomup(s)       = rec x(all(x); s)
    downup(s)         = rec x(s; all(x); s)
    downup2(s1, s2)   = rec x(s1; all(x); s2)
```

Some common rule application strategies. Strategies do not explicitly mention the terms that they operate on. They take other strategies (a rule is a simple form of a strategy) as parameters and implicitly deconstruct terms. For example, the `all(s)` strategy applies `s` to all subterms of the current term.

---

borrows very much from ASF. The distinguishing characteristic of Stratego is therefore the strategy driven transformation. A strategy is a declarative construct which allows careful control over the dependency relationships between rules. A strategy takes as a parameter a transformation task that is to be applied to a parse tree. Strategies look very much like rules, however, they do not explicitly mention the terms that they modify. Instead, they reference other strategies and implicitly operate on terms.

The most simple form of a strategy is a rule. More complex strategies are constructed using a set of operators that combine strategies together. For example, the sequence operator `s1;s2` causes the application of one strategy, then the next. The `s1+s2` operator attempts to apply one of two strategies. A set of built-in strategies define basic traversal behaviour. The `rec x(s)` strategy defines a symbol, `x`, whose reference in `s` represents a recursive application of the strategy. The `all(s)` strategy says apply the strategy `s` to all subterms of the current term being processed (i.e. all subtrees of the current tree). These operators and basic strategies allow the programmer to specify a traversal without giving

the specific forms involved.

Figure 2.3 shows some common strategies that may be programmed in Stratego. The `try(s)` strategy first applies strategy `s`, if that fails then it applies the identity strategy. The `repeat(s)` strategy applies `s` until it fails. The `topdown(s)` strategy applies `s` to the current term then recurses. The `bottomup(s)` strategy recurses first then applies `s`. Lastly, the `downup` strategies do a top-down and bottom-up application.

Stratego represents the most recent developments in the term-rewriting family of program transformation systems. Stratego has many other interesting features such as dynamic rules which allow a program to define rules at run-time. Overlays permit patterns to be abstracted so language syntax need not be directly specified in a rule. Our work on out parameters resembles overlays in Stratego. Research into how to better express transformations through control of the rewrite process is ongoing in the Stratego community.

2.4 ANTLR Tree Parser Generator

The ANTLR Tree Parser Generator [Parr94] (formerly known as SORCERER) is an approach to the problem of program transformation aimed at being of practical importance to programmers by integrating with general-purpose programming languages.

ANTLR allows the specification of a grammar plus tree patterns and associated actions. In the first stage the ANTLR system parses the input into an internal representation. In the second stage the internal representation is parsed in a top-down fashion using the tree patterns. Upon a pattern match, an associated action, coded in a host language such as C++, is triggered. ANTLR provides escapes from action code back into the ANTLR domain for accessing the parts of a pattern match. Using these escapes it is possible to read or write parse tree components. This mechanism makes it possible to explicitly program arbitrary

Figure 2.4: ANTLR Tree Parser Example

```
expr:
        ID
    |   FLOAT
    |   <<int n;>>
        fc > [n]
        <<printf("function call has %d arguments\n", n);>>
;

fc > [int nargs]:
        <<int i = 0;>>
        #( FUNC ID ( . <<i++;>> ) * )
        <<nargs = i;>>
;
```

An example of ANTLR tree parser patterns and associated actions. The `expr` *pattern matches expressions of type identifier, float, or function call. The function call form is separated out into a pattern of its own that counts the number of function call arguments and returns the result to the main pattern. The symbols <<>> delimit pattern action code.*

---

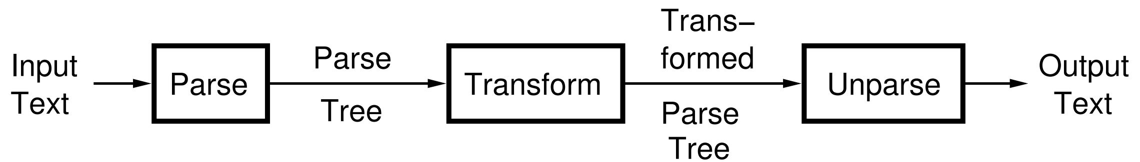
tree traversals and transformations. ANTLR does not have an unparsing phase. Outputting a transformed tree requires the writing of output statements in action code. Figure 2.4 shows an example of tree patterns used to match expressions.

ANTLR also supports semantic predicates for refining the pattern matching process. A semantic predicate is a user-defined expression that evaluates to either true or false and that indicates the validity of a pattern match. Semantic predicates provide the same function as conditions on rewrite rule.

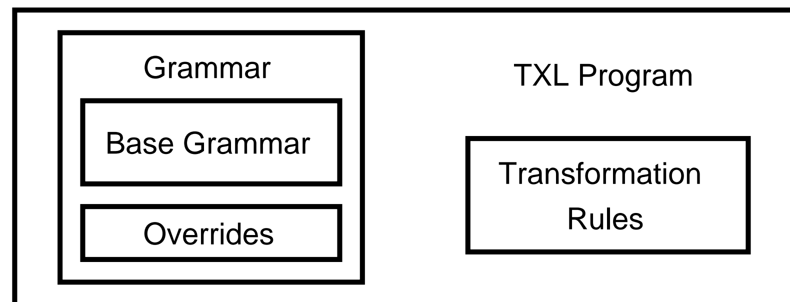
2.5 TXL

TXL, the Tree Transformation Language, is a hybrid functional and rule-based program transformation system. The processing of input is divided into three phases: parsing of the

input text into a structured parse tree, transformation of the parse tree and unparsing of the new tree to the output text.



A program written in the TXL Language is divided into a grammar specification and a set of structural transformation rules. These specify the language to be processed and the transformation instructions, respectively. A TXL grammar is normally divided into a base grammar and grammar overrides.



The base grammar defines the lexical and syntactic forms of the language to be processed. The base grammar is usually included by the main program because language grammars are often reused. The lexical forms are given in a regular expression notation. The syntactic forms are defined using a free-form extended BNF-like grammar. Ambiguities in the grammar are handled via a backtracking parsing algorithm.

Grammar overrides can be used to extend or modify the base grammar, allowing new language forms to be accepted during the parse or generated by the transformation. For example, if extending a programming language to include a new kind of statement, one will override the statement non-terminal.

Program transformation behaviour is defined by the set of transformation rules, which are able to search parse trees for by-example style patterns and provide replacements. Whether or not a rule is applied may be controlled by further pattern matching on pattern components as well as by conditions programmed in a functional style. These facilities are analogous to rule conditions in the term rewriting systems previously described.

TXL provides a built-in tree traversal mechanism. If the default tree searching is inadequate, arbitrary tree traversals may be developed by explicitly coding the traversal order in a functional style. Similarly, the order of rule application is controlled by the explicit invocation of rules using TXL's function application mechanism.

To fully take advantage of TXL's abilities, grasping the hybrid nature of rules is essential. TXL Rules have dual personalities. A rule exhibits characteristics of term rewriting rules in the sense that a rule repeatedly searches its scope for instances of its pattern and makes a replacement. On the other hand a rule is a functional style program that looks and behaves like a statically typed lisp function that can be implicitly iterated. Knowing when to view a rule as which is the key to fully harnessing TXL's abilities. In some problems it is advantageous to believe that you are writing a rewrite rule and at other times it is necessary to be thinking in terms of functional programming.

In the following figures we show an example of a TXL program that moves else-if tests and all following cases into an if statement nested in an else block. A section of the C++ grammar relevant to the transformation is shown in Figure 2.5. The complete TXL ruleset is shown in Figure 2.6. Finally, sample input and output texts are shown in Figure 2.7.

Figure 2.5: TXL Grammar Example

```
define statement
        [labelled_statement]
    |   [expression_statement]
    |   [selection_statement]
    |   [iteration_statement]
    |   [jump_statement]
    |   [declaration_statement]
    |   [try_block]
    |   [compound_statement]
    |   [comment]
end define

define selection_statement
        'if ( [condition] )
            [statement]
        [repeat else_if_part]
        [opt else_part]
    |   'switch ( [condition] )
            [statement]
end define

define else_if_part
        'else 'if ( [condition] )
            [statement]
end define

define else_part
        'else
            [statement]
end define

define condition
        [expression]
    |   [repeat type_specifier+] [declarator] = [assignment_expression]
end define
```

A section of the C++ grammar relevant to the TXL transformation example in Figure 2.6.

Figure 2.6: TXL Ruleset Example

```
include "cpp.grm"

define program
       [cpp_program]
end define

rule nestElseIfs
   replace [selection_statement]
      'if ( Condition [condition] )
         IfBranch [statement]
      ElseIfPart [repeat else_if_part]
      OptElsePart [opt else_part]
   deconstruct ElseIfPart
      'else 'if ( ElseIfCond [condition] )
         ElseIfBranch [statement]
      ElseRest [repeat else_if_part]
   by
      'if ( Condition )
         IfBranch
      'else {
         'if ( ElseIfCond )
            ElseIfBranch
         ElseRest
         OptElsePart
      }
end rule

function main
   replace [program]
      P [program]
   by
      P [nestElseIfs]
end function
```

A TXL transformation that moves else-if tests and all following cases to a nested else branch.

Figure 2.7: TXL Input and Output Example

```
bool b1, b2;
int main(int argc, char **argv)
{
    if ( argc == 1 )
        cout << "There were no arguments." << endl;
    else if ( argc == 2 )
        cout << "There was one argument: '" << argv[1] << "'." << endl;
    else {
        cout << "There was more than one argument: ";
        for ( int a = 1; a < argc; a++ )
            cout << (a == 1 ? "'" : ", '") << argv[a] << "'";
        cout << "." << endl;
    }
}
```

```
bool b1, b2;
int main( int argc, char **argv )
{
    if ( argc == 1 )
        cout << "There were no arguments." << endl;
    else {
        if ( argc == 2 )
            cout << "There was one argument: '" << argv[1] << "'." << endl;
        else {
            cout << "There was more than one argument: ";
            for ( int a = 1; a < argc; a++ )
                cout << (a == 1 ? "'" : ", '") << argv[a] << "'";
            cout << "." << endl;
        }
    }
}
```

Sample input and output texts showing the result of the transformation in Figure 2.6. The else-if test has been moved to a new if statement in an else block.

2.6 Summary

Source transformation systems are in wide use in many areas of computer science and software engineering. We have looked at several popular systems. The first of the term rewriting family of program transformation systems is ASF+SDF. It supports conditional rewrite rules and a fixed set of generic traversal schemes. Creating a custom traversal scheme requires the node visitation order to be explicitly programmed.

ELAN is a rule-based rewrite language that is targeted towards writing deduction systems. It is not specifically designed for program transformation and therefore can be awkward when used to analyze source code. ELAN introduced the concept of strategies to rewrite systems. Strategies are declarative constructs that can specify traversal order and rule application order. Strategies have proven useful in the area of program transformation.

The Stratego language is the descendent of ASF+SDF and ELAN. It combines the program transformation facilities of ASF+SDF with the strategy oriented traversal introduced by ELAN. It also introduces innovative features of its own such as dynamic rules and overlays.

ANTLR takes a different approach to source analysis and transformation, one which aims at easy integration with popular host programming languages. Though useful for making transformations, ANTLR is better described as a tree parser generator, rather than a transformation system. ANTLR tree patterns invoke actions which are coded in a host language. Escapes from the action code back into ANTLR permit read and write access to the components of a pattern match, enabling the transformation of parse trees.

TXL, also in a category of its own, is a hybrid rule and functional programming language. Rewrite rules in the form of pattern and replacement pairs compose the base language. A full functional programming language is built on top, which enables arbitrary

traversal and program logic. The goal of this thesis is to improve the program transformation facilities of TXL. The next chapter looks at the motivations behind the improvements that will be described in subsequent chapters.

Chapter 3

Motivation

This chapter looks at the specific motivations behind the language changes that we propose. In each section we illustrate how the solutions to certain problems that arise in program transformation are expressed in TXL. We point out current limitations of the language with respect to approaching these problems and suggest ways in which the language may be altered so that solutions may be better expressed. The design of each proposed change follows directly from the motivations we illustrate in this chapter.

3.1 Must-Match Rules

The central paradigm of TXL programming is by-example pattern and replacement. If the tree to which a rule is applied matches the rule's pattern then the result is the replacement. If the tree does not match the pattern then the original is silently returned. In most cases this is the desired behaviour because TXL is most often used to find and modify instances of a particular problem in arbitrary source code. If a problem instance is not found then the work of the TXL engine is complete and the programmer need not be informed.

Figure 3.1: Multiple-Stage Transformation

```
% Find function declarations that return vectors by value.
rule moveRetToParams
    replace $ [function_definition]
        'Vector Declarator [declarator] Body [function_def_body]
    by
        'void Declarator [insertParam] Body [rewriteReturn]
end rule
```

The principal pattern of a multiple-stage transformation. This example locates functions that return vectors by value and rewrites them to return their value in the first argument. Once the pattern has made a match and the return type has been changed, `insertParam` *must succeed at inserting a parameter and* `rewriteReturn` *must successfully rewrite all the return values it has found otherwise the transformation is incomplete.*

However, sometimes a problem is such that certain patterns can be expected. This may arise if the text being processed is automatically generated or if processing a program known to be legal and the pattern involves a necessary condition for legality, such as the existence of a `main()` routine. Rules written for such cases should always match, otherwise some error condition exists.

Rules that are expected to match also arise when writing multiple-stage transformations. These transformations require an all-or-nothing approach. Once the first stage of a multiple-stage transformation has taken effect it is expected that the following stages will also succeed, otherwise some error condition exists. Multi-stage transformations are characterized by the existence of a principal pattern which is responsible for identifying instances of the problem and worker rules which are employed to handle the various stages of the transformation. An example of a multiple-stage transformation is given in Figure 3.1.

In practice, rules that are expected to match are frequently written, however, whether or not these rules actually match often goes unchecked. There is no easy way of learning

when a rule that should have matched failed to match anything. At present, it is necessary for the programmer to identify test cases that may be problematic and verify the results or to otherwise notice that something is wrong. In many cases this involves actively pursuing the possibility that a particular rule is not matching.

When it is apparent that some rule is failing to match, but it is not obvious which rule is the source of the problem, tracking down the faulty rule can be very time consuming and inconvenient. This can involve inserting print statements throughout the program to identify the rules that are taking effect. After the problem has been identified the print statements must be deleted or commented out. If another bug surfaces later on then the procedure of sprinkling print statements throughout the program must be repeated. Alternatively, program flow may be traced. Which approach is best depends on the nature of the problem; print statements provide good summary information whereas debugging provides precise information. In either case effort must be expended towards tracking down the source of the problem.

We believe that whether or not a rule should match is a static property of a rule. To address this we wish to add to TXL a mechanism for explicitly stating as an attribute of a rule, whether or not the rule is expected to match. The TXL engine is then capable of verifying our assumptions.

3.2 Objectless Rules

Practical experience with TXL has shown that it is sometimes not necessary for a rule to be applied to anything in particular. For example, it may be the case that a rule simply queries or sets a global variable. This kind of rule does not operate on anything, does not need to be passed any parameters, nor does it return a result. In another example, a programmer may

prefer as a matter of style to pass all data on which the rule operates as parameters. This can happen if a rule must take many parameters and it does not make sense stylistically to use any one of the parameters as the object of the rule, thereby suggesting that the parameter is the focus of a replace or matching rule.

In TXL, rules that do not need to operate on any parse tree are written as polymorphic rules without a named binding in the pattern. Omitting a binding for the `[any]` type is not required; it is done because the anonymous type is not used and it is poor style to add unused pattern bindings to the local namespace.

These types of polymorphic rules can be applied to anything and always leave the parse tree to which they are applied unchanged. Such a rule can be considered an identity function. Since these kinds of rules do not perform a deconstruct, it is best to invoke them by applying them to an anonymous construct of type `[any]` in order to avoid falsely suggesting that they read or write the parse tree to which they are applied. Figure 3.2 shows a rule that simultaneously walks two lists. As a matter of style, both lists are passed as parameters.

The existence of this practice of writing anonymous polymorphic matching rules that operate only on globals or parameters challenges the existing TXL design, suggesting that every rule is not strictly either a matching or replacing rule. In practice, some rules are simply a sequence of operations. Therefore we wish to permit rules to be written with no replace or match clause. We call this new kind of rule an objectless rule.

3.3 Strong Typing

In TXL, rules may be applied to any tree type. Some rule applications are nonsensical and TXL warns about these. If a rule can never match a tree because the grammar does not

Figure 3.2: Unused Main Pattern

```
function typecheckLists ParamTypes [repeat decl_specifier]
       ArgTypes [repeat decl_specifier]
   match [any]
       _ [any]
   % Take apart each parameter and argument.
   deconstruct ParamTypes
       Param [decl_specifier] RestParams[repeat decl_specifier]
   deconstruct ArgTypes
       Arg [decl_specifier] RestArgs [repeat decl_specifier]
   % Type check the first pair and recurse.
   construct _ [any]
       _ [typecheck Param Arg] [typecheckLists ParamTypes ArgTypes]
end function
   ...
   construct _ [any]
       _ [typecheckLists ParamTypes ArgTypes]
```

This rule is responsible for checking that in a function call the argument types match the parameter types. Neither `ParamTypes` nor `ArgTypes` are considered the object of the type check. Instead they are both parameters. The function is invoked in an anonymous construct of type `[any]`.

---

derive the type for which the rule is searching, then a warning message is printed. Though in general, TXL can search for a pattern match from any tree root. TXL is this way because the structure of code, specifically how a transformation is grouped into rules and patterns, is driven by the structure of the grammar that is being operated on. In practice, grammar productions are often reused, meaning that instances of patterns can occur rooted at various different types, therefore the ability to search for patterns from different parse tree roots is a useful feature of TXL.

However, it is also true that even though a rule may succeed at rewriting subtrees beneath many different tree roots, it may be a valid transformation only when applied to a specific tree root. This can happen because languages are often designed such that syntactically valid code is a superset of semantically valid code. Therefore a transformation

Figure 3.3: Rule Valid at Only One Root

```
rule forLoopMakeUnaryInc
    skipping [statement]
    replace [assignment_expression]
         CastExpr [cast_expression] += '1
    by
        ++ CastExpr
end rule
```

In this example, the programmer's intent is to replace by-one increments of for loop variables in loop iteration code with the prefix operator. Though this rule would succeed when applied to various kinds of statements, it should only be applied to iteration statements. In TXL this requirement must be stated in a comment and explicitly heeded by the user of the rule.

---

can be syntactically valid but not necessarily semantically valid. It may also be the case that a programmer simply intends a rule to be used rooted at a specific tree root and is not interested in matching the other cases.

Figure 3.3 shows an example of a transformation that is syntactically valid when applied to various different tree roots, but is a valid transformation, as prescribed by the programmer, only when applied to a single tree root. This transformation is intended to rewrite loop increment operations to use the increment operator. The rule could successfully transform expression statements for example, but in this application they are outside the rule's domain.

In an environment where code is written and used by different parties and must be maintainable over time, the ability to communicate in the language that a rule should be applied only to a specific type can enable more effective collaboration. The reason is that the compiler is able to enforce the property laid out by the programmer. Type safeness helps to ensure that code is used as intended.

3.4 Nested Rules

In a TXL program, parse trees are frequently taken apart and reconstructed. Much of TXL's power comes from the ease of locating instances of problems and binding their parts to variables for later use in constructing a solution. The process involves the management of many variables containing parts of parse trees. These variables are a key part of the pattern matching paradigm around which TXL is built.

Due to the fact that grammars are essentially tree data structures, TXL programmers are often faced with the need to propagate data collected at the upper levels of a traversal down to the lower levels. This is generally accomplished by accumulating parameters as rules deconstruct and recurse deeper into a tree structure. A complex problem can involve propagating an increasingly large number of parameters down many levels of a rule hierarchy.

For example, suppose a rule needs to manipulate expressions in a class member function of a C++ program. The manipulation requires the class name, the class variables, the function name and the function's local parameters. A top level rule is written for locating classes, extracting the class name and collecting the class variables. A second level rule locates functions, extracts the function name and collects local parameters. The actual worker searches a function's expressions and performs the rewrite, making use of the propagated variables. An example of a transformation of this nature is given in Figure 3.4.

Manually propagating data in this fashion is time consuming, error prone and somewhat inconvenient. It requires extra work during initial programming as well as during maintenance. Due to the deconstruct and reconstruct nature of TXL programming, the manual propagation of data down a traversal surfaces often. We wish to relieve this necessity by providing nested rules that support the referencing of local data originating from a

Figure 3.4: Propagation of Variables

```
% Use class and function properties to determine if we should prefix.
function meetsPrefixCriteria ClassKey [class_key] ClassId [id]
       OptBase [opt base_clause] FuncDeclSpec [repeat decl_specifier]
       FuncId [id]
    ...
end function

% Visit each declaration in the function.
rule prefixInFunc ClassKey [class_key] ClassId [id] OptBase [opt base_clause]
       FuncDeclSpec [repeat decl_specifier] FuncId [id]
   replace $ [init_declarator]
       Id [id] OptInit [opt initializer]
   where
       Id [meetsPrefixCriteria ClassKey ClassId OptBase FuncDeclSpec FuncId]
   by
       ClassId [_ FuncId] [_ Id] OptInit
end rule

% Visit each function in the class.
rule prefixInClass ClassKey [class_key] ClassId [id] OptBase [opt base_clause]
   replace $ [function_definition]
       FuncDeclSpec [repeat decl_specifier] FuncDeclarator [declarator]
       FuncBody [function_def_body]
   deconstruct * [id] FuncDeclarator
       FuncId [id]
   by
       FuncDeclSpec FuncDeclarator
       FuncBody [prefixInFunc ClassKey ClassId OptBase FuncDeclSpec FuncId]
end rule

% Visit each class.
rule prefixLocals
   replace $ [class_specifier]
       ClassKey [class_key] ClassId [id] OptBase [opt base_clause]
          { MemberSpec [opt member_specification] }
   by
       ClassKey ClassId OptBase
          { MemberSpec [prefixInClass ClassKey ClassId OptBase] }
end rule
```

An example scenario requiring the propagation of parse tree components to the lower levels of a tree traversal. The top level rule `prefixLocals` collects class properties and pushes them down. The second level rule `prefixInClass` collects function properties and adds them to the collection. The real workers `prefixInFunc` and `meetsPrefixCriteria` utilize the propagated data to perform the transformation.

parent rule. Instead of propagated data dominating parameter lists, we wish it to be made implicitly available.

3.5 Rule Parameters

TXL rules provide a built-in mechanism for searching parse trees. The leftmost, shallowest search strategy is appropriate in most cases. However, at other times a custom search strategy must be written. Since TXL is a full functional programming language, writing a custom search strategy does not pose a problem itself. The general recipe is to encode the search into a recursive form by deconstructing the parse tree and recursing on its parts in a way that yields the desired pattern test order. The order could be dictated by the contents of nodes as in the case of a binary search. Alternatively, the search could simply follow a predefined order other than leftmost, shallowest. For example, the evaluation of arithmetic expressions represented in parse tree form requires a leftmost, deepest search strategy. Another example is the need to visit the elements of a sequence in reverse order. Figure 3.5 shows the use of binary search to locate a node by key, then apply a worker rule.

Sometimes it is the case that the same search strategy is required multiple times in a TXL program. For example, it may be the case that processing list elements in reverse order is the normal way to handle a specific type of input. Or in the case of a binary tree implementation of a symbol table, it may be desirable to modify the contents of a named node using various different rules from different points in the program.

In the case of binary search, one might be inclined to solve this problem by first retrieving the node of interest using a generic find then modifying it. This would permit using the same find routine no matter which rule is used to modify the node of interest. Since TXL employs copy-on-write semantics this approach will not work. Retrieving a node and

Figure 3.5: Custom Tree Traversal

```
function findLeft Key [number]
   replace [node]
      NodeKey [number] NodeVal [value] Left [node] Right [node]
   where
      Key [< NodeKey]
   by
      NodeKey NodeVal Left [findAndIncRef Key] Right
end function

function findRight Key [number]
   replace [node]
      NodeKey [number] NodeVal [value] Left [node] Right [node]
   where
      Key [> NodeKey]
   by
      NodeKey NodeVal Left Right [findAndIncRef Key]
end function

function findHere Key [number]
   replace [node]
      NodeKey [number] NodeVal [value] Left [node] Right [node]
   where
      Key [= NodeKey]
   by
      NodeKey NodeVal [incRef] Left Right
end function

function findAndIncRef Key [number]
   replace [node]
      Node [any]
   by
      Node [findLeft Key] [findRight Key] [findHere Key]
end function
```

An example of a custom tree traversal in the form of a binary search. Once the target of the binary search is found the worker rule `incRef` *is applied.*

then modifying it will leave the version in the symbol table unchanged. The node must afterwards be reinserted into the tree in order to store the result of the change. This approach works but is more cumbersome and expensive than modifying the node in place. For other search strategies, extraction then reinsertion may not even be possible if the data structure was previously organized in a fashion that cannot be reproduced with an insertion routine. In these cases we have no choice but to embed the rewrite rule directly in the search strategy.

In TXL, reusing the same search strategy with various different rewrite rules requires us to duplicate the code implementing the search. To remove this requirement for code duplication, we wish to add to TXL a mechanism for writing rule-independent search strategies. It should be possible to write a search strategy once and parameterize the rules that the search strategy applies.

3.6 Type Parameters

Similar to the need for rule-independent search strategies, there is also a need for type-independent rules. Often, a search strategy or transformation is written multiple times with the only difference between the versions being the tree types on which they operate. This can happen when writing common list processing algorithms such as sort routines or list reversals. Figure 3.6 gives an example of list reversal. Notice that one could substitute any type for `[id]` and still have a valid reversal.

A need to duplicate code and differentiate it only by type can arise along with the need to specialize types. A type specialization in grammar programming involves duplicating and slightly modifying a section of a grammar with possibly a smaller, larger or just plain different set of allowable strings. Type-independent rules would also find use in the writing

Figure 3.6: List Reversal

```
function reverse List [repeat id]
   deconstruct List
      First [id] Rest [repeat id]
   replace [repeat id]
      SoFar [repeat id]
   construct Result [repeat id]
      First SoFar
   by
      Result [reverse id Rest]
end function
```

An example of a rule that could be made to operate on any list by simply changing the type of the list element.

---

of generic algorithms over homogeneous structures such as trees composed of a single type.

As discussed in the previous section, an ability to parameterize a rule would enable us to write search strategies that are independent of the specific transformations that they perform. If we extend this idea of parameterization of language constructs to include types as well, we find that more generic programming is possible. Now we are also able to write a search strategy that generalizes to multiple tree types.

3.7 If Clauses

TXL has control flow implicitly built into the language. If any clause in a rule fails then the entire rule is considered to have failed. In this sense TXL rules are a series of nested if statements. With this in mind one can implement branching by applying a different rule for each case in the branch. The binary search of Figure 3.5 is implemented in this way.

This mechanism of branching does not mirror the semantics of an if statement as most programmers know it. An important difference is that the cases must be explicitly written

such that they are mutually exclusive. Since cases are encoded in rule applications, which in TXL are independently applied, then even if a particular case succeeds, all successive cases are still tested and may also succeed if not explicitly programmed to exclude the previous cases. Most languages that implement if statements do not behave this way. TXL is this way because rule application semantics have been borrowed for the purpose of branching.

The explicit mutual exclusion of cases can be seen in Figure 3.5. Normally, if both less-than and greater-than tests fail then equality is assumed to hold. However, with rule-based cases as in this example, the equality test is required. Without this clause the binary search would apply its rule to every node it encountered on the way to the intended target of the rule.

Guaranteeing mutually exclusive branches in the case of binary searching is easy. However, as the complexity of tests increases the task gets harder. In the event that tests involve multiple deconstructions and where clauses, the only manageable solution is to introduce a global variable that indicates when a branch of the if statement has been taken. In the event that an if branch modifies data that is read in other tests of the if clause, then this solution is necessary, otherwise successive tests will exclude previous tests based on a different program state.

The difficulties in explicitly programming mutually exclusive branches is shown by the subtle incorrectness of Figure 3.7, an example of explicitly walking a statement list. This kind of processing is necessary when data must be propagated down a list of statements. In this example there is a rule to handle each type of statement. The cases for handling label and expression statements are shown. Note that since each rule must propagate the walk data down to the next statement, each case is responsible for the recursive call to `walkStmts`, the rule that ties all the cases together. This example is incorrect because

Figure 3.7: Incorrect Branching

```
function walkLabeledStatement WalkData [walk_data]
   replace [repeat statement]
      Stmt [labeled_statement] Rest [repeat statement]
   by
      Stmt Rest [walkStmts WalkData]
end function

function filterExpr Expr [expression_statement] WalkData [walk_data]
   replace [repeat statement]
   deconstruct not Expr
      'assert ( AssertExpr [expression] ) ;
   by
      Expr
end function

function walkExpressionStatement WalkData [walk_data]
   replace [repeat statement]
      Stmt [expression_statement] Rest [repeat statement]
   construct ResStmt [repeat statement]
      _ [filterExpr Stmt WalkData]
   construct Tail [repeat statement]
      Rest [walkStmts WalkData]
   by
      ResStmt [. Tail]
end function

function walkStmts WalkData [walk_data]
   replace [repeat statement]
      Statements [repeat statement]
   by
      Statements
         [walkLabeledStatement WalkData]
         [walkExpressionStatement WalkData]
         ...
end function
```

This example is incorrect because in the event that `filterExpr` removes the expression statement, all successive statements may be processed twice depending on whether the following statement is handled by any of the cases after the expression statement. The only way to guard against this is to introduce a global indicating that a case was taken.

when `filterExpr` removes an expression statement, the remaining statements may get unknowingly processed twice. This kind of problem requires the use of global variables because different cases see different program state and therefore cannot correctly exclude each other.

Though using global variables is a workable solution, in the event that recursion is required the global variable must in fact be a stack of globals in order to be correct, making the implementation far more tedious. Since one could easily argue that recursion is inevitably required in every non-trivial TXL program, the global variable solution to mutual exclusion leaves much to be desired.

3.8 Out Parameters

The principal method of analyzing a parse tree in TXL is the pattern match. A pattern match is used to both require that a parse tree contain specific subtrees and to extract subtrees from it. If a pattern that expresses the entire direct subtree is suitable for extracting the subtrees of interest, then a simple deconstruct will do. If a pattern rooted deeper in the tree is necessary then a searching rule or deconstruct can be used.

Deconstruction requires exact knowledge of the structure of the parse tree being taken apart. In TXL, there is currently no way of deconstructing a tree without having access to the associated grammar definitions. This fact makes it difficult to modularize a deconstruction by inserting an interface between the code that needs to extract subtrees for later use and the patterns used to extract them. In the event that different parts of a grammar are maintained by different developers, such an interface would be necessary to modularize programming tasks.

An ability to abstract away the deconstruction of a tree would allow TXL programmers

Figure 3.8: Variations of a Pattern

```
rule rewriteAssignment1
   replace $ [statement]
      Id [id] = AssignExpr [assignment_expression] ;
   by
      Id [_ 'set] ( AssignExpr ) ;
end rule

rule rewriteAssignment2
   replace $ [statement]
      assign( Id [id], AssignExpr [assignment_expression] );
   by
      Id [_ 'set] ( AssignExpr ) ;
end rule
```

An example of a pattern that can take on multiple forms. These rules both find variable assignments. Though the forms they search for differ in syntax, they are identical in semantics. In TXL there is no way to factor out a pattern from otherwise identical rules.

---

to better organize code much the same way that a function call in an imperative language is used to abstract away a task. A function can be specified once and invoked multiple times. Similarly, a pattern abstraction mechanism would allow a pattern to be specified once and used multiple times.

In some scenarios, several rules may differ only by the patterns that they use to take apart trees. In this case the ability to write the rule once and pass in as a parameter the pattern used to take apart the tree being operated on would be useful. Figure 3.8 gives an example of two rules whose patterns differ only in syntax. In cases such as these, an ability to abstract away the deconstruction would avoid the need to duplicate rules.

We wish to add to TXL a mechanism for deconstructing a tree without immediately specifying a pattern, but instead use some object that represents a pattern. With this goal in mind we have introduced the concept of out parameters.

3.9 Modularity

TXL was originally designed for small rapid prototyping tasks. Now it finds use in larger production systems with many thousands of lines of code, on which multiple developers work. As the size of TXL programs grow, a mechanism for information hiding becomes increasingly necessary. The ability to maintain a section of a program independent of the remainder of the program is very important when developing large systems. Collaborating developers must be concerned about such issues as avoiding name collisions and specializing rule names to reflect the areas of the TXL program in which they are meaningful.

Developers that write reusable code modules which are intended to be dropped into any TXL program must also heed the name collision problem. If they are to claim that they have written a truly reusable module, it cannot be the case that it must be modified to work with a program that uses the same names.

In existing TXL programs, avoiding name collisions is accomplished by employing a naming convention whereby all entities are prefixed with a qualifying name. Private entities can be further distinguished by prepending an underscore to the name. This convention works, but it is somewhat tedious as every name definition and associated reference must be prefixed. Figure 3.9 gives an example of a reusable code module in which all entities are prefixed with this scheme.

Naming conventions succeed at avoiding name collisions but do not permit the true hiding of names. That is, there can be no real distinction between entities that are private and entities that are public. Name hiding is very commonplace in large-scale systems because its use can dramatically simplify the comprehension of a program by hiding internals. Name hiding is a key requirement when producing a library that requires entry strictly through a carefully chosen interface.

Figure 3.9: Reusable HTML Markup Code

```
define _HTML_item
       [_HTML_begin] [any] [opt _HTML_end]
end define

define _HTML_begin
       < [id] [repeat _HTML_option] >
end define

define _HTML_end
       < / [id] >
end define

% Make an item bold.
function HTML_boldize
   replace [any]
      A [any]
   construct BoldTag [_HTML_begin]
      <B>
   by
      A [_HTML_tagwith BoldTag]
end function

% Markup with any tag.
function _HTML_tagwith Tag [_HTML_begin]
   deconstruct Tag
      < TagId [id] TagOptions [repeat _HTML_option] >
   replace [any]
      Anything [any]
   construct TaggedThing [_HTML_item]
      <TagId TagOptions> Anything </TagId>
   deconstruct TaggedThing
      TaggedAnything [any]
   by
      TaggedAnything
end function
```

An example of a reusable code module that can be easily dropped into any TXL program for the purpose of marking up output with HTML tags. To avoid name conflicts, type definitions and rules must be prefixed with the module name. Private names are signified by a leading underscore.

With a simple naming convention as the one described above, it is not possible to bypass the use of qualifying names as a matter of convenience when there happens to be no name collisions. This is another feature that is common in modularization schemes.

We wish to provide a facility for defining abstraction layers that allow developers to either hide or expose grammar definitions, rules and global variables as well as group code into logical program modules. A modularization feature would aid in collaboration on large projects, ease the writing of generic libraries and improve the quality of TXL software from a software engineering point of view.

3.10 Summary

Often rules are written which are expected to match. Unfortunately, whether or not these rules actually match often goes unchecked. When a transformation goes wrong because rules are failing, tracking down the faulty rule can be time consuming. To defend against this scenario, it is desirable to be able to declare as a part of the language that a rule is to always match. This empowers the TXL engine to verify the programmer's intentions on their behalf.

There are cases in TXL programming where it is not necessary for a rule to have a main pattern. Some rules are not really rules, but are merely sequences of operations. In these cases, the common practice is to write anonymous polymorphic matching rules. This suggests that perhaps a new type of rule is in order, one that has no match or replace clause.

TXL allows rules to search for patterns from various different tree roots. This is a useful feature of TXL, however, sometimes we want a rule to be relevant from only a single root. Some mechanism of expressing in the language that a rule is to be applied only to a tree of a specific type would enable the TXL engine to enforce rule application restrictions.

Transformations often require context information. In TXL, the way to propagate context is by passing data to subrules with parameters. As tree traversals go deeper and deeper into a parse tree, the amount of data propagated increases. To free the programmer from explicitly passing many parameters we wish to add an ability to implicitly make values available to subtasks.

In program transformation, the ability to write rules that specify how to walk a tree independent from the rules that specify how to transform it, is a desirable feature. An ability to pass rules as parameters enables this abstraction. Similarly, we want to be able to write rules that are aware of the structure of the trees that they operate on, but are independent of the specific types. Type parameters satisfy this requirement.

In TXL, rule application and pattern matching semantics have been borrowed for the purpose of branching. The techniques employed for selecting cases work, but are unfamiliar to most programmers, contain subtle pitfalls and are somewhat tedious for complicated tests. Therefore a native branching facility in TXL would improve programs that require complex decision making.

In TXL, there is no way to define an abstraction layer between code that needs to deconstruct a tree and the code that does the deconstruction, without using global variables. An ability to abstract away the task of taking apart a tree would enable developers to draw clean program modularization boundaries between those two tasks.

Due to the increasing size of TXL programs and requirement for developer collaboration that accompanies larger programs, software engineering practices are being drawn on for managing large code bases. Programmers are required to emulate modularity features that are common in other systems. Therefore we wish to introduce language support for organizing code into modules.

We have shown in detail the motivations behind our language changes. With each change we aim to address specific language deficiencies. In the following chapter we present the design of the language changes that we have developed in response to the problems presented in this chapter.

Chapter 4

Design

In this chapter we present the language changes that we have developed. We define the syntax of our new features relative to the existing TXL syntax, which is described in detail in the TXL language specification [Cordy03]. We use TXL's language definition syntax as the notation for our changes. This is also described in the TXL language specification. For those readers who wish to see how each syntax change fits into the full solution, the entire ETXL grammar is given in Appendix A.

For each new feature we informally define its semantics and then describe, with examples, how our solution solves the problems we set out to address in Chapter 3. We aim to show that each new feature is an improvement in its own right, and in some cases how features may be combined effectively. Some of our features bring added benefits and these are mentioned. Also, limitations and implications of our solutions are discussed.

Figure 4.1: Must-Match Rule Syntax

```
redefine replace_clause
       [opt 'must] 'replace [opt replace_modifier] [nonterm_ref]
          [pattern]
end redefine
```

The replace clause has been redefined to include the optional `must` *keyword.*

4.1 Must-Match Rules

Some TXL rules are written with the expectation that they will always match the tree to which they are applied. However, the default behaviour in TXL is for a rule to silently fail. We wish to add to TXL an optional attribute associated with a rule that indicates it should always match, thus enabling the TXL engine to assert such rules indeed match.

The design of must-match rules is fairly straightforward. By definition, matching-only rules (rules that contain only a match clause) communicate whether or not they have matched to their caller, and so we need not be concerned with them. The must-match property is applicable only to rules with replace clauses, therefore we have made it a part of the replace clause. Figure 4.1 shows the syntax we have chosen. A rule is given as a must-match rule by prefixing the replace clause with the keyword `must`. This syntax is simple and reads naturally.

Once a rule has been specified with the must-match property, the ETXL engine is capable of identifying rules that should have matched but did not. Rules may unexpectedly fail perhaps because of an improperly specified pattern or because the programmer has made an assumption of the input that is too strong and needs to be relaxed by covering additional cases. In any case, the failure of a must-match rule should be made known to the user and so we cause it to generate a run-time error.

Figure 4.2: Must-Match Rule Example

```
function saveAndReturn
    must replace [statement]
        'return AssignmentExpression [assignment_expression] ;
    by
        {
            '__ret = AssignmentExpression ;
            'return ;
        }
end function
```

Worker component of Figure 3.1. The transformation component `saveAndReturn` *would be applied to all return statements after transforming a function's return value to a by-reference parameter. Any return statement that failed to be rewritten would constitute a failure of the transformation.*

An important aspect of must-match rules is that the TXL engine is able to check that rules are behaving as expected on every run of the program. This automated checking provides assurance to the programmer that if an assumption fails to hold then the error will not go unnoticed.

Must-match rules permit the writing of better multiple-stage transformations. Once a problem instance has been found and committed to, worker rules that implement the various components of the transformation can be made must-match rules. In the event that a transformation has only partially completed, a run-time error will draw attention to the problem. Figure 4.2 shows the worker rule that would be applied to all return statements found by the rule `rewriteReturn` of Figure 3.1.

4.2 Objectless Rules

In practice, not all rules are strictly either a replace or match rule. Some rules can be considered merely a sequence of operations. In support of this, we have added objectless

Figure 4.3: Objectless Rule Example

```
function noGlobalErrors
   import GblSyntaxErrors [number]
      0
   import GblSemanticErrors [number]
      0
end function
   ...
   where
      _ [noGlobalErrors]
```

In this example, an objectless function is used in a where clause to require that no errors were encountered during the processing of input.

---

rules, also known as identity rules, to TXL. Objectless rules have neither a replace clause nor a match clause and are semantically equivalent to an anonymous polymorphic matching rule. They can be applied to any parse tree and their success is determined only by contained deconstruct, import and where clauses.

It is possible for clauses other than match and replace to fail. An objectless rule may query a global for a specific pattern or deconstruct its arguments. Therefore, even though an objectless rule always matches the parse tree it is applied to, it may still be useful in a where clause. In existing TXL, this is not possible without first constructing a dummy variable simply for the purpose of applying the objectless rule in a where clause. To accommodate this minor discrepancy, objectless rules may be applied to the anonymous variable '_' in where clauses. We must restrict this facility however. Objectless rules are the only type of rules that may be applied to the anonymous variable in where clauses. The application of a non-objectless rule would be the equivalent of searching an empty, typeless value, therefore we do not permit it. Figure 4.3 shows an example of an objectless rule that does nothing but query globals. The result of the rule is verified in a where clause.

Figure 4.4: Capturing Rule Failure

```
rule removeReturnVals
   if
      replace [jump_statement]
         'return _ [expression] ;
      by
         'return ;
   else
      export NoReturnValsRemoved [number]
         1
   end if
end rule
```

If the pattern does not succeed then print a warning and apply the identity function.

Objectless rules can be used in combination with if clauses (described in Section 4.7) to make alternatives to replacements in a manner that preserves the meaning of the rule. TXL semantics dictate that if a rule's pattern does not match the parse tree it is applied to, then the parse tree is returned unchanged. Preserving this property and catching the case that a rule's pattern does not match can be accomplished by wrapping the pattern in an if test and using an objectless rule in the else clause. Should the pattern fail, the parse tree will always be returned unchanged because the else case is the identity function. Figure 4.4 gives an example.

4.3 Strong Typing

TXL permits a rule to be applied to any tree root. This is useful because grammar productions are often reused in grammar design and so rules are useful when applied to various different types. On the other hand some TXL rules may be valid only when applied to a

Figure 4.5: Strong Typing Syntax

```
redefine rule_stmt
       'rule [opt nonterm_ref] [id] [param_list]
          [repeat rule_clause+]
       'end 'rule
end redefine
```

The new rule syntax with strong typing. Function syntax has been modified in the same way.

---

specific type, despite the fact that they succeed in their transformation when applied to various different types. There is no facility in the language for communicating the requirement that a rule should only be applied to a specific type.

Strongly typed rules accommodate this need by permitting the programmer to specify an exclusive type to which a rule may be applied. Strong typing is optional and specified immediately before a rule name. The syntax is shown in Figure 4.5.

Once a rule has been designated as strongly typed, the ETXL compiler is able to verify that it is applied to parse trees of the correct type. Rules applied to the wrong type cause a compile-time error.

Strongly typed rules permit a programmer's intent to be captured more accurately. The example in Figure 3.3 highlights a case where a rule is syntactically valid at many different statement types, but is semantically valid only when applied to the iteration statement. Figure 4.6 shows the strongly typed version of this example.

4.4 Nested Rules

TXL programs frequently involve the deconstruction and reconstruction of parse trees. Inherent in analysis and transformation is a need to propagate data deeper into a parse tree

Figure 4.6: Capturing Intent with Strong Typing

```
rule [iteration_statement] forLoopMakeUnaryInc
   skipping [statement]
   replace [assignment_expression]
       CastExpr [cast_expression] += '1
   by
      ++ CastExpr
end rule
```

The strongly typed version of Figure 3.3. The strong typing captures the requirement that the rule be applied to iteration statements only.

Figure 4.7: Nested Rule Syntax

```
redefine rule_clause
      ...
   |  [rulefunc_stmt]
end redefine
```

The change required to support nested rules. Rule and function statements are now permitted in clause lists.

---

during a traversal. The way to do this in TXL is by passing data as parameters to rules. We have noticed that in some cases data is propagated deeper and deeper though several levels of a traversal. The amount of data tends to increase as the traversal proceeds.

The solution we have devised is to allow the nesting of rules. Figure 4.7 shows the syntax change required; a rule or function statement is now able to appear in a clause list. A nested rule in ETXL is permitted to access variables bound in its ancestors. Since TXL variables must be declared before use, variables that are to be accessed by a nested rule must be bound before the declaration of the nested rule.

Nested rules are not visible outside of their parent and its descendants. This makes it impossible for a rule invocation to skip a generation. As a result, variables bound in an

ancestor are guaranteed to exist on the program stack. In the event that a parent rule has multiple invocations on the stack (perhaps because of a recursive call), the youngest stack frame is used for accessing the parameter. Since bound variables are only ever read, a function would never modify its parent's stack frame, only read from it.

In addition to reducing the amount of parameter passing required, this nested rule solution also serves as a means to better organize code. In existing TXL, private rules that are only ever used by one function are often written. Nesting helps to communicate the context of a rule. Figure 4.8 shows the example of Figure 3.4 implemented with nested rules. All explicit propagation of data is eliminated.

Other languages that have nested functions include Pascal, Algol-60, Algol-68, PL/1 and Ada. In the past there has been debate over the value of nested functions [Han81]. There has been criticism especially in the domain of systems programming, where it is said that nested functions hinder separate compilation and require an unacceptable amount of function call overhead. TXL, being a high-level, directly interpreted language is free from these concerns, however there is some merit to the argument that nested rules make programs less readable by further separating the declaration of variables from their use. Also, that nested rules do not cooperate very well with other language features. For example, in our implementation a nested rule cannot be passed as an argument to a rule parameter.

Despite these concerns we believe that nested rules are a worthwhile addition to TXL because they reduce the need to repeatedly pass as parameters the many read-only variables that are often in use in TXL programs.

Figure 4.8: Nested Rule Example

```
% Visit each class.
rule prefixLocals
   replace $ [class_specifier]
      ClassKey [class_key] ClassId [id] OptBaseClause [opt base_clause]
         { MemberSpec [opt member_specification] }

   % Visit each function in the class.
   rule prefixInClass
      replace $ [function_definition]
         FuncDeclSpecifier [repeat decl_specifier] FuncDeclarator [declarator]
         FuncBody [function_def_body]
      % Retrieve the function's name
      deconstruct * [id] FuncDeclarator
         FuncId [id]

      % Use context to determine if we should prefix.
      function meetsPrefixCriteria
         ...
      end function

      % Visit each declaration in the function.
      rule prefixInFunc
         replace $ [init_declarator]
            Id [id] OptInit [opt initializer]
         where
            Id [meetsPrefixCriteria]
         by
            ClassId [_ FuncId] [_ Id] OptInit
      end rule

      by
         FuncDeclSpecifier FuncDeclarator FuncBody [prefixInFunc]
   end rule

   by
      ClassKey ClassId OptBaseClause
         { MemberSpec [prefixInClass] }
end rule
```

This example shows Figure 3.4 implemented with nested rules. The worker rules `prefixInFunc` and `meetsPrefixCriteria` implicitly utilize variables bound in their ancestors. The need to explicitly propagate class and function data has been eliminated.

Figure 4.9: Rule Parameter Syntax

```
redefine parameter_type
       ...
    |  [rule_type]
end redefine

define rule_type
       '[ [opt nonterm_ref] [rule_or_func] [rule_type_param_list] ']
end define

define rule_type_param_list
       [repeat parameter_type] [opt rule_type_out_param_list]
end define

define rule_type_out_param_list
       ': [repeat parameter_type]
end define
```

A summary of rule parameter syntax. Parameter types have been redefined to allow the rule type. Rule types require any strong typing and parameter types that argument rules might have. Since rules are referenced by name, no change to argument syntax is required.

4.5 Rule Parameters

To enable the writing of generic search strategies that can be used to traverse a parse tree in a manner that is independent of specific rewrite rules, we propose adding rule parameters. Rule parameters enable the writing of custom generic search strategies. Figure 4.9 shows the ETXL syntax of rule parameters.

Rule parameters must fully specify the signature of the rules they accept as arguments. This includes parameter types and any strong typing specification. This requirement is justified by the fact that in TXL, rule signatures are static, meaning that a single rule passed in as an argument can only be used with one set of argument types passed to it. Requiring the full signature in the type of the rule parameter ensures that the parameters and any

strong typing a rule argument must have is effectively communicated. Without the full signature, the programmer would need to examine the contents of the rule to which a rule argument is passed to see what the parameters must be and what type it is applied to. This would work for determining the required type of rule arguments, but would be inconvenient and likely error prone.

The full specification of the rule parameter signature also simplifies the implementation. It implies that the body of a rule containing a rule parameter can be validated for correctness and compiled independent of any specific invocations. Not requiring the full signature would mean that a rule with rule parameters must be validated for each set of rules passed as arguments that are found in the program. Since an invocation point may itself be in a rule with rule parameters, any of which may be passed as the rule argument in the invocation, finding all unique sets of rules passed as parameters to any given rule involves a recursive search through all rules. Requiring the full specification of rule parameter signatures means we do not have to do this.

At invocation point, a rule is passed as an argument either by referencing a local rule parameter or by naming a global rule. No change to the syntax is required to support the passing of rules as arguments. Inside a rule, a rule parameter name is available for use in the rule body as if it were a globally defined entity. Figure 4.10 shows a generic binary search that is able to apply any rule to a node found by key.

The implementation of this mechanism is simple and the paradigm can be used to write any generic search strategy. One remaining issue to address is that of passing data arguments along with a rule argument and later passing them to the rule argument when it is invoked. The most simple solution, the one which we employ, is to require the programmer to pass the data arguments to the rule accepting the rule parameter, then later explicitly pass

Figure 4.10: Generic Binary Search Tree Traversal

```
function findLeft Key [key] Rule [rule]
   replace [node]
      NodeKey [key] NodeVal [value] Left [node] Right [node]
   where
      Key [< NodeKey]
   by
      NodeKey NodeVal Left [findAndApply Key Rule] Right
end function

function findRight Key [key] Rule [rule]
   replace [node]
      NodeKey [key] NodeVal [value] Left [node] Right [node]
   where
      Key [> NodeKey]
   by
      NodeKey NodeVal Left Right [findAndApply Key Rule]
end function

function findHere Key [key] Rule [rule]
   replace [node]
      NodeKey [key] NodeVal [value] Left [node] Right [node]
   where
      Key [= NodeKey]
   by
      NodeKey NodeVal [Rule] Left Right
end function

function findAndApply Key [key] Rule [rule]
   replace [node]
      Node [any]
   by
      Node
         [findLeft Key Rule] [findRight Key Rule] [findHere Key Rule]
end function
```

A generic version of the binary search in Figure 3.5 that is able to apply any rule to the target node.

the data arguments to the rule argument in all calls to it.

Another approach might be to specify the rule plus its arguments as a single unit in the argument list. An invocation of `findAndApply` may look as in the following example.

```
SymbolTable [findAndApply VariableName [addReference ExpressionId]]
```

The problem with this solution is that it may be confused with an ability to apply rules to rule arguments. In this proposed scheme, the above example is read as the application of `findAndApply` to `SymbolTable` with two arguments, `VariableName` and `addReference` plus its associated parameter `ExpressionId`. However, given existing TXL semantics a programmer's intuition may suggest the above is read as the application of `addReference` to `VariableName` then the application of `findAndApply` to `SymbolTable` with the modified `VariableName` as an argument. Despite the fact that ETXL does not support the application of rules to rule arguments, the syntax certainly suggests that it does. Given this potential ambiguity as well as a desire to reserve the right to introduce the application of rules to rule arguments, this solution is unacceptable.

4.6 Type Parameters

A mechanism to parameterize types would permit us to write type-independent rules. Such rules are useful when writing algorithms over various different types that have similar properties. Type parameters free us from writing multiple rules that differ only by the types that they operate on.

Type parameters are specified in a similar fashion to rule parameters. They are given in the parameter list with the keyword `type` or `define` in place of the parameter type. Figure 4.11 shows the syntax changes we have made. At invocation point, a type is passed as an argument either by referencing a local type parameter or by literally specifying the type as

Figure 4.11: Type Parameter Syntax

```
redefine parameter_type
        ...
    |   [type_type]
end redefine

define type_type
        '[ 'define ']
    |   '[ 'type ']
end define
```

```
redefine function_arg
        ...
    |   [nonterm_ref]
end redefine
```

Syntax modifications necessary to support type parameters. The parameter type has been redefined to permit the type form. The function argument syntax now permits literal types.

---

in a define statement or a pattern. The TXL grammar has been modified to permit literal references to types in the argument list. A literal type may either be a pure type such as `[declarator]` or it may contain a modifier as in `[repeat statement]`.

Inside a rule, a type parameter may be used anywhere a globally defined type may be used, with a restriction. A type with a modifier cannot be passed as an argument to a parameter that is in turn used with a modifier. Type parameters used with a modifier must accept only pure types as arguments. If the definition of a type parameter in a parameter list implied the definition of a new type that contained the argument type then this restriction would not exist, however no such implication exists. In ETXL, type parameters merely associate a name with an existing type. The author concedes that it would be beneficial to somehow specify which type parameters accept only pure types and which type parameters accept modifiers in order to enforce the previously mentioned restriction. This is left for future work.

Type parameters allow the expression of a generic sort as shown in Figure 4.12. This sort can operate on any type for which a less-than operator can be written. In general, type

Figure 4.12: Generic Sorting

```
rule sort T [type] LessThan [[T] rule [T]]
   replace [repeat T]
      N1 [T] N2 [T] Rest [repeat T]
   where
      N2 [LessThan N1]
   by
      N2 N1 Rest
end rule
   ...
   construct Sorted [repeat pair]
      Pairs [sort [pair] pairLess]
```

A generic sorting routine that can be applied to a list of any type. The sort routine requires a less-than operator for comparing instances of the type.

---

parameters aid in writing generic algorithms and data structures. Type parameters are most useful in combination with rule parameters.

Type parameters introduce the nesting of `[]` delimiters for the purpose of containing type names. This nesting would cause problems if implementing the application of rules to rule arguments or if implementing nameless constructors, which are both mentioned as possible future changes in Section 7.3. In terms of parsing this is not a fatal ambiguity because type names can be distinguished from rule names. In terms of readability though, a sufficient naming convention would be required in order to easily decipher the meaning without looking up the name in question.

Unfortunately type parameters do not permit as simple an implementation as rule parameters. A rule with rule parameters may be compiled once and the same code used for all invocations due to the statically typed nature of rule parameters. In the run-time system, a rule parameter may be represented with a single value identifying the rule. Unfortunately the same cannot be said of type parameters. Since the structure of a pattern must be parsed

before the pattern is used and the result of the parse is directly determined by the types in the pattern, a rule must therefore be validated for correctness and compiled once for each set of types passed as parameters. This requires a recursive search through all rules in the program.

4.7 If Clauses

In order to permit branching in TXL without requiring the explicit programming of mutually exclusive branches, we propose adding a native branching construct to TXL. The construct should behave similar to if statements in general-purpose programming languages, but should not represent a significant deviation from existing TXL conventions. The most obvious choice for achieving this goal is the syntax shown in Figure 4.13. This syntax fits in well with existing TXL syntax and is fully compositional. It allows if tests and then blocks to be any sequence of clauses, including an if clause itself.

If clauses behave as follows. If all clauses in an if test succeed, the then block is entered and no more if tests are considered. Once a then block has been entered, the success of the clause list containing the if clause is determined by the clauses in the then block. If all clauses succeed, control is passed to the code following the if clause.

If any clause in an if test fails, the then block is abandoned and the test block of the next else-if clause is tried. If no test block succeeds, control is transfered to the else clause. If there is no else clause, the clause list containing the if clause fails. The reason for this is discussed later in this section. Figure 4.14 demonstrates the use of an if clause to implement the binary search of Figure 4.10.

In order for a rule that contains an if clause to be legal, every control flow path through the rule must be a correct rule in its own right. This means that in every path there must be

Figure 4.13: If Clause Syntax

```
define if_clause
        'if
            [repeat clause]
        [opt then_block]
        [repeat elseif_clause]
        [opt else_clause]
        'end 'if
end define

define then_block
        'then
            [repeat clause]
end define

define elseif_clause
        'else 'if
            [repeat clause]
        [opt then_block]
end define

define else_clause
        'else
            [repeat clause]
end define
```

Syntax of if clauses. This syntax allows any clause in if tests, then blocks and else blocks.

Figure 4.14: Binary Search Using If Clause

```
function findAndApply Key [number] Rule [rule]
    replace [node]
        N [number] Left [node] Right [node]
    if where
        Key [< N]
    then by
        N Left [findAndApply Key Rule] Right
    else if where
        Key [> N]
    then by
        N Left Right [findAndApply Key Rule]
    else by
        N [Rule] Left Right
    end if
end function
```

Using if clauses to implement the binary search of Figure 4.10. The size of the code is reduced significantly.

Figure 4.15: Incomplete Rule

```
if replace [repeat number]
    1
then construct _ [stringlit]
    SawOneMsg [print]
else if
    ...
end if
by
    Repl
```

An example of why an empty else block must induce failure or we must forbid match, replace and by clauses from appearing in an if clause. If no if test is able to succeed then transferring control to after the if clause would imply that the rule is able to make a replacement without first succeeding in matching any pattern.

---

only one replace or match clause and replace clauses must have a corresponding by clause which is not followed by anything. It is worth noting that it is possible for a path to partially complete. For example, a replace in an if test may succeed, but a following where clause or deconstruct may fail. In this case, control is transfered to the next test or else block as if the replace never succeeded. Another replace clause must be present in whichever path is taken.

As previously mentioned, if clauses in ETXL must be complete. This means that if none of the if tests succeed, and there is no else clause, then control flow cannot proceed past the if clause. There are two reasons for this. The first has to do with the fact that if any branch of an if clause contains a replace or by clause then all branches must, otherwise it would be possible to derive a control flow path through a rule that did not have a proper pattern and replacement pair. A rule containing the code in Figure 4.15 would not make any sense if upon failure of all if tests, control were passed to after the if clause. Allowing control flow to proceed past an if clause when an empty else clause is encountered would

require us to disallow the use of match, replace and by clauses in if clauses. Since doing so would dramatically reduce the utility of if clauses, we choose to fail when an empty else clause is encountered.

The second reason that empty else clauses cause failure has to do with the fact that variables have no default value. Either they are bound or do not exist. Quite often, the purpose of an if clause is to select a value for a variable that is to be used later in the rule. Suppose that all if tests fail and the empty else branch is taken, then the variable is not bound and according to TXL semantics, does not exist. This constitutes a compile-time error that must be corrected by explicitly programming an else case that binds some default value. Instead of forcing the programmer to add a dummy else case that constructs some default value merely to satisfy the compiler, we allow them to be omitted and fail the containing clause list when an empty else clause is encountered.

We have found that when using the if clause, it is sometimes the case that no statements are required in the then block. To accommodate this, the `then` keyword has been made optional. A branch is interpreted as having an empty then block when the keyword is not present.

The syntax we have chosen is not without its faults. It is somewhat awkward with respect to readability, leading to a lack of a consistent indentation convention. For example, the `if where` clause of Figure 4.14 reads clumsily. If we move the `where` to the next line and add a level of indentation as shown in Figure 4.16 the awkwardness of `if where` goes away. Unfortunately it makes the code look unnaturally sparse and adds a second level of indentation to a construct that by convention requires only one.

If the test contains more than one clause as in the second part of Figure 4.16 then we find that adding a level of indentation is necessary in order to ensure readability. Since each

Figure 4.16: If Test Indentation

```
if
   where
      Key [< N]
then
```

```
if
   construct Dist [number]
      First2 [- First1]
   where
      Dist [< Max] [= Max]
then
```

At left: an alternative to writing `if where`*. At right: when the if test contains more than one clause, adding a level of indentation is necessary to ensure readability.*

---

clause in the test (with the exception of clauses that cannot fail) contributes to the result, it would be misleading to misalign the clauses.

4.8 Out Parameters

In TXL there is no way to abstract away the deconstruct of a parse tree. It is not possible to put an interface in between code that requires a tree to be broken down into parts and the code that performs the breakdown. Also, there is no way of parameterizing a rule by the patterns it uses. We wish to bring these abilities to TXL by introducing out parameters.

An out parameter binds a value from a called rule to a variable name in a calling rule. Inside the called rule, an out parameter must be bound as either the target of a construct or an element in a pattern or be returned by another rule invocation as an out parameter. The behaviour of out parameters was designed to follow the behaviour of pattern matching. If a pattern fails to find a match then the variables of the pattern are not bound and control cannot proceed past the pattern. In our design of out parameters, we require that if a rule fails to bind an out parameter then the caller is unable to proceed.

Note that the successful transfer of an out parameter to the caller is independent of the success or failure of the rule in which it was bound. A rule that succeeds in binding an out parameter but fails in making a replacement will not fail the caller clause list. Maintaining these two mechanisms as independent processes helps to keep the design simple.

TXL rules can have iteration implicitly built-in. Any clause that follows the replace clause of a searching rule may be executed more than once during a single invocation. This means that an out parameter may be bound in the called rule more than once. In this case the last value to be bound in the called rule is the value that is returned to the caller.

Syntactically, a colon is used to separate in parameters from out parameters in both parameter and argument lists. Figure 4.17 summarizes the required syntax changes to support out parameters.

In TXL, the introduction of a new variable into the current scope is always accompanied by a type. This design choice is motivated by the fact that TXL programs typically manipulate many variables. Keeping track of the meaning of a program with many variables is made easier by more explicit typing of those variables. The introduction of new variables in out argument lists maintains this design principle. An out argument whose type has not been previously defined in the current scope by a forward clause (described in Section 4.9) must have its type explicitly given in the out argument list. Even though it is possible to infer the type of a newly bound variable from the signature of the rule being called, we require the type to be explicitly given in order to help promote better program comprehension.

Out parameters are a convenient way of abstracting away the deconstruct of a parse tree without deviating far from existing TXL syntax and semantics. The use of out parameters to abstract away a deconstruct is demonstrated in Figure 4.18.

Figure 4.17: Out Parameter Syntax

```
redefine param_list
       [repeat parameter] [opt out_param_list]
end redefine

define out_param_list
       ': [repeat parameter]
end define

redefine function_app
       '[ [function_name] [repeat function_arg] [opt out_arg_list] ']
end redefine

define out_arg_list
       ': [repeat out_arg]
end define

define out_arg
       [id] [opt nonterm_ref]
end define
```

Parameter and argument lists now have an optional out parameter component that both begin with a colon. Out parameters are specified in a manner identical to in parameters. Out arguments are given by an identifier and an optional type.

In another example, consider the problem of returning a value from a find routine. In TXL, writing a retrieval function without the use of a global variable requires the value sought to be returned via the object on which the function operates. The tree to be searched is passed in as a parameter. The function matches an empty value and replaces it with the result of the find. After the find call, the value must be extracted from the optional result type.

Using out parameters, the find routine instead operates on the tree being searched and returns its result via an out parameter. This eliminates the need for a deconstruct of the result immediately following the find call. The find can instead be invoked with a single

Figure 4.18: Abstracted Deconstruction

```
function deconstructElseIf : Condition [condition] Statement [statement]
          Rest [repeat else_if_part]
   match [repeat else_if_part]
      'else 'if ( Condition [condition] )
         Statement [statement]
      Rest [repeat else_if_part]
end function
...
   where
      ElseIfParts [deconstructElseIf : Condition [condition]
         Statement [statement] Rest [repeat else_if_part]]
```

Use of out parameters to abstract away the deconstruct of an else-if clause. The caller is able to retrieve the critical parts of the clause without being concerned about the specific syntax.

Figure 4.19: Binary Search Using Out Parameters

```
function find Key [number] : Value [number]
   match [node]
      NodeKey [number] NodeValue [number] Left [node] Right [node]
   if where
      Key [< NodeKey]
   then where
      Left [find Key : Value [value]]
   else if where
      Key [> NodeKey]
   then where
      Right [find Key : Value [value]]
   else construct Value [value]
      NodeValue
   end if
end function
...
   where
      Tree [find Key : Value]
```

A binary search that returns its result via an out parameter. This permits the find to be invoked using a where clause and avoids the need to extract the resulting value from an optional type.

---

where clause that reads naturally. A solution to the problem of binary search using out parameters is shown in shown in Figure 4.19.

In general, the ability to move the return value from the object of the rule application to an out parameter is useful because it allows the developer to take advantage of TXL's built-in searching capabilities and to return any number of results from the same rule while avoiding the use of global variables.

Out parameters also find a use in propagating information through the walk of a parse tree. It is very often the case that during a walk, information must be propagated back up a parse tree. In traditional compiler design these values are called synthetic attributes. A downside to using out parameters to this end is that the data structure used to pass the information around will be given twice in parameter lists, once in the in parameter list for propagating the structure down the tree and a second time as the out parameter list for propagating the data back up.

A nice result of out parameters is that when used in combination with rule parameters, we gain the ability to parameterize patterns for the purpose of information hiding or otherwise generic programming. For example, a developer may need to duplicate a rule many times, varying only the pattern used to deconstruct the tree on which it operates. If the pattern could be parameterized then a single generic rule could be written, passing the method of deconstruction in as a parameter. In Figure 4.20, a rule parameter binds the LHS and RHS of a value assignment using out parameters and thereby acts as a parameterized pattern. The generic replacement routine can easily be used with various other patterns that locate value assignments in other forms provided that the pattern binds the necessary LHS and RHS values.

Figure 4.20: Parameterized Patterns

```
function deconstructAssignment : Id [id] Expr [expression]
   match [statement]
      Id [id] = AssignExpr [assignment_expression] ;
   construct Expr [expression]
      AssignExpr
end function

rule genericReplace AssignmentPattern [rule : [id] [expression]]
   replace $ [statement]
      Stmt [statement]
   where
      Stmt [AssignmentPattern : Id [id] Expr [expression]]
   by
      Id [_ 'set] ( Expr ) ;
end rule
   ...
   by
      Program [genericReplace deconstructAssignment]
```

An example of the parameterization of patterns using out parameters in combination with rule parameters. The generic replace routine expects to be passed a pattern parameter that finds forms of assignment and returns their LHS and RHS.

4.9 Scoping and Forward Variable Declarations

TXL rules consist of a single local variable scope. In our design we have added a block structure in the form of nested rules and if statements. If we are to not stray too far from common programming language design practices, then with this block structure comes a requirement for nested scopes that implement name hiding. ETXL therefore treats nested rules and blocks of clauses within an if clause as a new variable name scope, allowing variables in parent scopes to be accessed, or hidden by a local variable name.

Since all local variables in TXL are read-only, it is not possible for a branch in an if clause to modify a variable that exists in a parent scope, as is commonly done in block

Figure 4.21: Forward Variable Declaration Syntax

```
define forward_clause
       'forward [id] [nonterm_ref]
end define
```

The new clause type for making forward variable declarations.

Figure 4.22: Forward Variable Declaration Example

```
forward ExprStmt [expression_statement]
if
   deconstruct UseExpr1
      'true
then
   construct ExprStmt
      Expr1
else
   construct ExprStmt
      Expr2
end if
...
match [statement]
   ExprStmt
```

An example of a forward clause used to predeclare a variable that is assigned a value from within an if clause. Without the forward declaration the variable would not be visible outside the if clause.

---

structured languages. This inhibits a very common use of the if construct; which is to conditionally construct a variable. Therefore we require an ability to pre-declare variable names that may possibly be constructed in a child block. To satisfy this requirement we have added the forward clause.

Forward variable declarations associate a variable with a type but do not give it a value. The syntax is shown in Figure 4.21. The variable must be bound to a value in following code before it may be referenced. Except in the case of patterns, the binding need not

Figure 4.23: Ambiguous Pattern Binding

```
forward ExprStmt [expression_statement]
...
deconstruct StmtList
    ExprStmt ExprStmt RestStmt [repeat statement]
```

An example of why we cannot omit a type from a binding in a pattern. The meaning of the deconstruct pattern cannot be deciphered without determining if the expression statement was bound following the forward declaration.

---

include a type since the variable has already been assigned a type. Forward variable declarations enable a clean and consistent syntax for pre-declaring the type of variables bound by out parameters as well as for constructing a value based on the result of a condition in an if clause. An example of a forward variable declaration in a conditional variable construction is shown in Figure 4.22.

Due to the similarity between the semantics out parameters and forward variable declarations, out parameters also function as forward variable declarations.

Forward variable declarations allow types to be omitted from construct clauses and out parameter lists. It seems natural to generalize this to all variable binding points in order to gain a simpler and cleaner design. In the name of consistency, it is desirable to support the omitting of the type from a parameter binding as well, but this is not practical. The problem is that the meaning of a pattern can become dependent on the existence of a binding in prior code.

Figure 4.23 illustrates this problem. Deciphering the meaning of the deconstruct of `StmtList` requires not only locating the declaration of `ExprStmt`, but also searching forward from the declaration to verify that the variable is indeed bound. If bound, then the `StmtList` deconstruct means to match an existing expression statement twice. If

not bound, then the deconstruct means to match any two identical expression statements. Therefore it is not possible to easily understand the meaning of a deconstruct without first studying the entire rule. Due to the potential for error inherent in this requirement we have chosen to leave out the ability to omit a type from a pattern.

4.10 Modularity

As TXL has gained acceptance, there has been a growth in the average size of TXL applications. In order to accommodate this growth, language features that permit developers to organize code into coherent modules are necessary. Therefore we have added a modularity system to TXL.

We chose to implement the module system at the language level, rather than at the file level because TXL already has a useful include system. Figure 4.24 summarizes the new statements introduced by modularity. Rules, global variables and grammar defines may be modularized by wrapping them in a module statement. All other TXL language constructs are considered entities in the global name scope because they are closely tied to the parsing phase, which is global in nature. By default, an entity within a module is private and as such is not accessible outside of the module. An entity may be made public by listing it in a public statement. A public entity may then be referenced outside of its module by qualifying it with its module name and a dot. Figure 4.25 shows the HTML markup code of Figure 3.9 in a module form.

The need to qualify a public entity can be alleviated by importing its module with the using statement. The using statement is permissible in the top level scope as well as in a module. Once a module has been imported, all its public entities are accessible without qualification as if they were defined in the name scope. Consequently, no collisions

Figure 4.24: Module Syntax

```
redefine statement
        ...
    |   [module_stmt]
    |   [using_stmt]
end redefine

define module_stmt
        'module [id]
            [repeat statement_inmod]
        'end 'module
end define

define statement_inmod
        [define_stmt]
    |   [rulefunc_stmt]
    |   [public_stmt]
    |   [using_stmt]
end define

define using_stmt
        'using [id]
end define

define public_stmt
        'public
            [repeat id]
        'end 'public
end define
```

A summary of modularity syntax. A module is a construct at the language level that is used to wrap a collection of statements. Also shown is the using statement, which imports all the names of a module, and the public statement, which is used to make modularized entities public.

---

between the public names of the imported module and the existing names in the scope can exist in order for the import to succeed. Previously imported names are included in this collision check. That is, it is not possible to import the same name into a module more than once.

Entities are modularized strictly at the name level. For example, in our scheme a public grammar define that is composed of private grammar defines may be fully deconstructed using only token types, which must always be public. This means that it is possible to take apart a private grammar define without naming it. In a sense this could be considered to be circumventing the module system we have devised.

Figure 4.25: HTML Markup Module

```
module HTML
   public
      boldize
   end public

   define item
      [begin_tag] [any] [opt end_tag]
   end define

   define begin_tag
      < [id] [repeat option] >
   end define

   define end_tag
      < / [id] >
   end define

   function boldize
      replace [any]
         A [any]
      construct BoldTag [begin_tag]
         <B>
      by
         A [tagwith BoldTag]
   end function

   function tagwith Tag [begin_tag]
      deconstruct Tag
         < TagId [id] TagOptions [repeat option] >
      replace [any]
         Anything [any]
      construct TaggedThing [item]
         <TagId TagOptions> Anything </TagId>
      deconstruct TaggedThing
         TaggedAnything [any]
      by
         TaggedAnything
   end function
end module
   ...
   replace * [id]
      Id [id]
   by
      Id [HTML.boldize]
```

The HTML markup code of Figure 3.9 in a module form. Using modularity features, private names are truly hidden and there is no need to qualify names within the HTML module.

Enforcing a strict boundary between public and private entities at the grammar structure level would involve the addition of an accessibility designation that represents the boundary. A grammar define with such a designation, perhaps termed a visible-only define, could be referenced in a name binding but not deconstructed. This kind of define would closely resemble opaque types of common programming languages. To close the gap in our module system, public defines would be required to be composed strictly of either other public defines or visible-only defines. Verifying that visible-only defines are never deconstructed in a pattern would entail the analysis of the parse tree of every pattern to ensure that the current set of visible-only defines exist in the parse tree only as leaf nodes.

Though visible-only defines are attractive from a design standpoint because they enable the precise definition of the boundary between a module's public and private grammar definitions, the little practical gain they bring does not warrant the effort required of their implementation, which at our prototype level is substantial. If we consider that private module defines are not likely to be composed of public defines, coming from inside the module or elsewhere, then essentially the purpose of this kind of check is merely to protect against a rather useless fringe case: which is the ability to take apart a type that cannot be named, and doing so with only token types. Therefore we have chosen to omit visible-only defines.

4.11 Summary

This chapter presents the design of the new language features that are introduced in this thesis. The first of the changes, must-match rules, permit the prefixing of a replace clause with the `must` keyword, which indicates to TXL that a rule should always make a match. Must-match rules that fail to match generate a run-time error.

To accommodate the writing of rules that are merely a sequence of operations, we now permit rules with no replace or match clause. This new kind of rule, an objectless rule, is semantically equivalent to an anonymous polymorphic matching rule.

Strong typing enables the programmer to specify the type of variables to which a rule may be applied. This enables programmers to more accurately communicate their intentions. An application of a strongly typed rule to a variable of the wrong type results in a compile-time error.

In our design we permit rules to be given in the clause list of another rule. A rule nested in another may reference variables defined in any of its ancestors. This can alleviate the need to explicitly propagate data down a traversal.

To make generic programming possible we have added a new parameter of type `rule` for passing rules as arguments. Rule parameters must fully specify the type of arguments they accept. To accompany rule parameters in the introduction of generic programming facilities to TXL, we have also added a new parameter of type `type` that permits grammar definitions to be passed as arguments.

A native branching construct that supports full if-elseif-else functionality has been added. Any clause may be nested in an if clause provided that every path through the rule is a valid rule in its own right. If no test succeeds and there is no else block, control flow cannot proceed past the if clause.

Out parameters have been introduced, which enable the transfer of a value from a called rule back to the caller. If the called rule fails to bind an out parameter then the caller cannot proceed past the point of invocation. If an out parameter is bound more than once due to implicit iteration then the last value is returned to the caller. Combining out parameters with rule parameters enables us to create pattern parameters that can be used to deconstruct

a tree in manner that is independent of the specific pattern used.

To support nested block structures introduced by if statements and nested rules, we have added variable scoping and forward declarations. Each nested block introduces a new variable scope in which existing ancestor variables may be accessed, or hidden by new variables. Since variables are read-only, constructing a variable that is to exist in a parent scope requires us to introduce forward variable declarations.

We have introduced a modularity statement that can be used to organize code into modules with independent name spaces. A rule, define and global variable may be encapsulated in a module. By default, module entities are private. They may be made public by including them in a public statement. Outside of a module, public module entities can be accessed by qualifying the entity with the module name. The need to do this can be alleviated with a using statement.

In presenting the design of these new features we have discussed our approach, presented the syntax, informally defined the semantics and given examples that show how our new features address the problems outlined in the previous chapter. In the next chapter we discuss the implementation of our prototype.

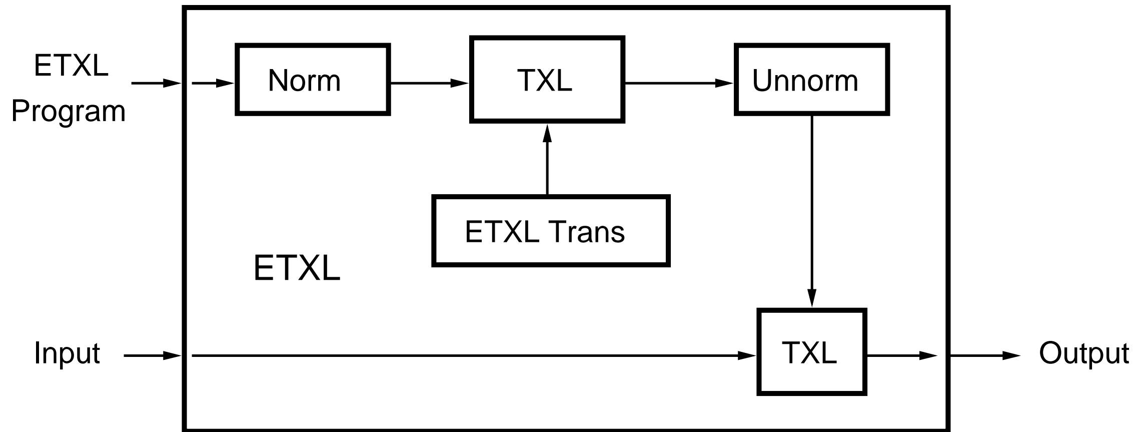
Chapter 5

Implementation

TXL was originally designed to aid in the development of the Turing programming language [Cordy04]. The Turing language developers wanted a tool for specifying and prototyping new language features in a by-example manner in support of the user-driven development of Turing. Turing developers needed a tool that would allow a maximum amount of experimentation with new features without the development time required by common compiler development tools, which in contrast are largely designed for developing production compilers for fixed languages. Born from this need and evolved over many years into a general-purpose source analysis and transformation tool, TXL has found uses in many language development problems.

In a testament to the usefulness of TXL as a language design tool we have chosen to rapidly prototype ETXL using a TXL transformation. It is, by design, a natural choice for this problem. Since we are extending the TXL language, we also use it as the target of the transformation. We transform ETXL to pure TXL in a manner that preserves all existing language features as well as implements the new features we have designed. Figure 5.1 illustrates this approach. Our prototype aims to be complete and usable by the TXL

Figure 5.1: Implementation Architecture



The ETXL program is first transformed lexically to simplify the processing of certain TXL constructs such as comments and compounds that are difficult to deal with in TXL. The ETXL transformations are then applied to produce a TXL program that implements the new features. The reverse of the lexical transformation is then performed. Finally, the transformed program and the original input is passed to the TXL engine for processing.

---

community. Once our new language features have shown to be practical and worthwhile in a production environment they can be implemented directly in the TXL engine. It is our hope that user feedback can be collected and heeded as the features proposed in this thesis are considered for transfer into the TXL engine.

Much of the implementation of this prototype relies on the intricacies of the TXL implementation. For example, we rely upon the fact that in a rule, any clauses preceding the replace clause are executed only once, no matter how many times the rule matches the parse tree to which it is applied. This is not specifically part of the TXL language, but rather a property of the implementation. In another example, if clauses rely on the fact that matching rules in where clauses never short circuit. Only one of the matching rules needs to succeed, but all are always applied. Since the TXL engine is itself evolving, it is not possible for us to guarantee the correctness of our prototype implementation into the

future. The assumptions we make and rely on may at some point no longer hold.

5.1 Must-Match Rules

Must-match rules use a flag variable to indicate to their caller whether or not the rule has succeeded. The flag is initialized to false before the execution of any clauses that may fail. The rule executes as normal and just before it returns, at a point at which success of the rule is guaranteed, a value of true is written to the flag. This point may not necessarily be the last statement due to the requirement that the by clause be the last statement. The caller, upon return, is then responsible for examining the flag and issuing an error message if the must-match rule did not match anything.

Implementing must-match rules is complicated by the fact that we wish them to work in conjunction with recursion. A global variable unique to each rule will not work for us because a recursive call would tamper with the value before it can be read by the caller. Also, tree expressions permit a rule to be invoked multiple times against a single variable. Using a global variable, we would be forced to split up such expressions in order to read the result of an invocation before a subsequent invocation of the same rule interfered with the result. To address both of these issues, we require each invocation to have a flag variable unique to it. To accomplish this, flags are managed using a stack. Upon entry, the flag initialization is in fact a push to the must-match stack. Before return, success is always written to the top of the stack. Figure 5.2 shows the implementation of Figure 4.2.

Figure 5.2: Must-Match Rule Implementation

```
function returnByReferenceParameter
   construct _ [any]
      _ [_mm_stack_push 0]
   replace [statement]
      'return AssignmentExpression [assignment_expression] ;
   construct _ [any]
      _ [_mm_stack_set_top 1]
   by
      {
         '__ret = AssignmentExpression ;
         'return ;
      }
end function
```

The implementation of the must-match rule example in Figure 4.2. Upon entry a value of false is pushed to the must-match stack. At the end of the rule a value of true is written to the top of the stack. The caller is responsible for popping the flag from the stack and generating a run-time error if false.

5.2 Objectless Rules

Implementing objectless rules entails the addition of a match any clause to the rule. The match any clause may go anywhere in the rule because it always matches irrespective of the value or type its containing rule is applied to and therefore cannot cause the rule to fail. Furthermore, matching rules and functions both quit after the first match is made. This is in contrast to rules with replacements, which execute the code following the replace clause for each match to the replace clause. Consequently, it does not matter if the rule's statements go after or before the match clause. We chose to insert the match any clause at the end of the rule.

```
   match [any]
      _ [any]
```

As described in Section 4.2, we have added the ability to apply objectless rules to the anonymous variable '_' in where clauses. To implement this feature we first enforce that every rule applied to the anonymous variable in a where clause is indeed an objectless rule. A failure in this regard generates a compile-time error. Then, to accommodate for the fact that in TXL the anonymous variable is not permitted as the target of a where clause, we create a variable of type `[any]` and use it as the object of the where clause.

5.3 Strong Typing

The strong typing implementation is strictly a compile-time feature. Once strong typing has been checked the type specifiers are removed and the program is left as is. Enforcing strong typing involves verifying the legality of the application of each strongly typed rule. This verification is done during the scoping pass when a dictionary of local variables and their types is available. Before the scoping pass begins, a global dictionary of strongly typed rules is built. If the name of a rule in an application is contained in the dictionary, or the name is a strongly typed rule parameter, its type is verified against the type of the variable to which it is applied. Any type mismatch results in a compile-time error.

A strong typing check is also incorporated into the verification of rule argument types in the implementation of rule and type parameters. Any mismatch in the strong typing between argument and parameter results in a compile-time error.

5.4 Nested Rules

In block structured languages such as Pascal, access to an ancestor's stack is provided by the display variable, which is an array of frame pointers. Since at this prototype level we do

not have access to frame pointers, we cannot implement nested rules in the traditional way. Instead, we take advantage of the fact that the values stored in TXL variables are never modified. Once a variable is bound to a value, it is a read-only entity until the variable goes out of scope and its value is garbage collected. This property allows us to implement the parent stack access by implicitly passing needed values from parent to child as regular parameters. There is no need to propagate changed values from the child back to the parent since variables are never modified.

For every child rule encountered in a parent clause list we first determine which variables defined in the parent are referenced in the child or any of its descendants. For each variable found, we add a parameter of the same type to the child's parameter list. Then for every call to the child rule from the parent or any of its descendents we add the variable to the argument list. Note that these are the only locations from which a child rule may be called. The child rule is then moved to the top level scope and its clause list is scanned in search of nested rules. This recursive approach permits variable references across multiple levels in the nesting hierarchy to work. Figure 5.3 shows the implementation of the `prefixLocals` routine in Figure 4.8, which is an example of the need to propagate variables down a traversal.

5.5 Rule and Type Parameters

Both rule and type parameters are implemented with a simple substitution mechanism. The program is searched for applications of rules where rule or type arguments are passed. ETXL first performs some simple validity checks on each application. Rule arguments have their type checked against the specification of the rule parameter. This involves checking

Figure 5.3: Nested Rule Implementation

```
function prefixLocals_prefixInClass_meetsPrefixCriteria ClassKey [class_key]
       ClassId [id] OptBaseClause [opt base_clause]
       FuncDeclSpecifier [repeat decl_specifier] FuncDeclarator [declarator]
   match * [any]
   ...
end function

rule prefixLocals_prefixInClass_prefixInFunc ClassKey [class_key] ClassId [id]
       OptBaseClause [opt base_clause] FuncDeclSpecifier [repeat decl_specifier]
       FuncDeclarator [declarator] FuncId [id]
   replace $ [init_declarator]
      Id [id] OptInit [opt initializer]
   where
      Id [prefixLocals_prefixInClass_meetsPrefixCriteria ClassKey ClassId
         OptBaseClause FuncDeclSpecifier FuncDeclarator]
   by
      ClassId [_ FuncId] [_ Id] OptInit
end rule

rule prefixLocals_prefixInClass ClassKey [class_key] ClassId [id]
       OptBaseClause [opt base_clause]
   replace $ [function_definition]
      FuncDeclSpecifier [repeat decl_specifier] FuncDeclarator [declarator]
      FuncBody [function_def_body]
   deconstruct * [id] FuncDeclarator
      FuncId [id]
   by
      FuncDeclSpecifier FuncDeclarator
         FuncBody [prefixLocals_prefixInClass_prefixInFunc ClassKey ClassId
         OptBaseClause FuncDeclSpecifier FuncDeclarator FuncId]
end rule

rule prefixLocals
   replace $ [class_specifier]
      ClassKey [class_key] ClassId [id] OptBaseClause [opt base_clause]
         { MemberSpec [opt member_specification] }
   by
      ClassKey ClassId OptBaseClause
         { MemberSpec [prefixLocals_prefixInClass ClassKey ClassId OptBaseClause] }
end rule
```

The implementation of Figure 4.8. Nested rules are moved to the top level scope, have their name prefixed with the name of their parent, and their parameter lists are filled in with the implicitly passed variables.

Figure 5.4: Generic Sorting Implementation

```
rule sort__pair__pairLess
   replace [repeat pair]
      N1 [pair] N2 [pair] Rest [repeat pair]
   where
      N2 [pairLess N1]
   by
      N2 N1 Rest
end rule
   ...
   construct Sorted [repeat pair]
      Pairs [sort__pair__pairLess]
```

The implementation of the generic sorting routine and call point in Figure 4.12.

---

any strong typing and parameter types of the rule argument versus those of the rule parameter. In general, a type parameter will accept any argument, however if a type argument with a modifier, for example `[repeat id]`, is passed to a type parameter that is referenced in conjunction with a modifier then a compile-time error is generated.

The substitution works as follows: rules with rule or type parameters are removed from the program and constitute a collection of templates. For each distinct set of rule and type arguments encountered in all invocations of a rule template, the template is instantiated by substituting the argument values in place of their corresponding parameter references. In the case of rule parameters, the argument value is simply the name of a rule. In the case of type parameters the argument value is the name of a type plus a possible modifier.

The name of an instantiated rule is chosen by appending the argument values to the root name. New instantiations are prepended to the list of rules. Figure 5.4 shows the instantiation and corresponding call point of the generic sorting rule of Figure 4.12.

5.6 If Clauses

If clauses are implemented by splitting each if test into two function calls, one to go down the if true branch and possible then block, and another to go down the else-if or else branch. Control does not return to the caller at the point of the split. Instead each branch is responsible for executing its case and the rest of the clauses that follow the if clause.

As in the implementation of must-match rules, the implementation of if clauses uses a stack of flags for indicating when a control flow path has been taken. The rule implementing the if branch first pushes a value of false to the stack that stores if test status flags, then proceeds to evaluate the test. Upon success of the test, it writes a value of true to the top of the stack and proceeds down the then branch. Since the rule implementing the else branch will be executed regardless of the outcome of the if branch, it first pops the top of the stack and requires it to be false. This mechanism ensures that else-if tests and else clauses are executed only when the previous if test fails.

Since each if test transfers control flow to the two rules implementing its cases, all local variables must be propagated to both branches. Local variables are propagated in the form of implicitly passed arguments. Figure 5.5 shows the TXL code implementing the if-elseif-else construct of the binary search shown in Figure 4.14.

The implementation we have described so far is not yet complete. It requires the tie-up of a loose end. In the case of rules that repeatedly search the trees to which they are applied, we require a mechanism to manually fail a search when a pattern and corresponding replacement is split across multiple rules by an if clause.

For example, consider a rule in which a pattern is given in the top level scope of the rule and the correct replacement is selected by an if clause. The top level scope needs to stop searching for patterns when no if branch succeeds at making a replacement. In

Figure 5.5: If Clause Implementation

```
function findAndApply Key [number]
    Rule [rule]
  replace [node]
    _This [node]
  deconstruct _This
    N [number] Left [node] Right [node]
  by
    _This
      [_branch1 Key Rule N Left Right]
      [_branch2 Key Rule N Left Right]
end function

function _branch1 Key [number]
    Rule [rule] N [number]
    Left [node] Right [node]
  construct _ [any]
    _ [_if_stack_push_false]
  replace [node]
    _This [node]
  where
    Key [< N]
  construct _ [any]
    _ [_if_stack_set_true 0]
  by
    N Left [findAndApply Key Rule] Right
end function

function _branch2 Key [number]
    Rule [rule] N [number]
    Left [node] Right [node]
  construct _if_stack_top_1 [number]
    _ [_if_stack_pop]
  deconstruct _if_stack_top_1
    0
  replace [node]
    _This [node]
  by
    _This
      [_branch3 Key Rule N Left Right]
      [_branch4 Key Rule N Left Right]
end function

function _branch3 Key [number]
    Rule [rule] N [number]
    Left [node] Right [node]
  construct _ [any]
    _ [_if_stack_push_false]
  replace [node]
    _This [node]
  where
    Key [> N]
  construct _ [any]
    _ [_if_stack_set_true 0]
  by
    N Left Right [findAndApply Key Rule]
end function

function _branch4 Key [number]
    Rule [rule] N [number]
    Left [node] Right [node]
  construct _if_stack_top_2 [number]
    _ [_if_stack_pop]
  deconstruct _if_stack_top_2
    0
  replace [node]
    _This [node]
  by
    N [Rule] Left Right
end function
```

Implementation of Figure 4.14. The invocation of branch one and two in `findAndApply` *signifies the start of the if construct. Branch one implements the if test and the first case. Branch two ensures that the if test failed, then proceeds with the remaining cases. Branch three implements the else-if test. Branch four ensures that the else-if test failed then executes the else part.*

our implementation of if clauses, the top level scope invokes the if branches via two function applications. Since the TXL function application mechanism cannot indicate when an applied function fails to make a replacement, an explicit boolean indicator is necessary.

The need for explicit failure also arises when a complete if clause separates a replace and its by clause. This is due to the fact that all clauses following an if clause are pushed into the rules implementing the branch cases.

Our prototype detects when manual failure may be required and adds an explicit check for this special case. Again, a stack of boolean flags is appropriate for solving our control flow problem. Immediately before the first branch point separating the pattern and replacement, a flag is pushed to the replace stack. All branches of control flow that can be derived from the branch point write a success indicator to the top of the stack upon success in rewriting. This will happen immediately before the by clause. The rule that invokes the branch point and thus contains the search mechanism that may need to be failed ensures that success was written to the flag before it continues searching for the pattern.

Figure 5.6 shows an example of a pattern and replacement that gets split across multiple rules by an if clause and consequently the search must be manually failed when the replacement fails. In this example, the result of a deconstruct is used to determine what the replacement should be. The corresponding implementation, also in Figure 5.6, shows the use of a stack of flags to explicitly fail the replacement.

5.7 Out Parameters

In the high-level TXL programming environment, it is not possible for a rule to access the local variables of a rule it has invoked, therefore some transfer point must be employed for passing values from a called rule back to its caller. For this we again employ a stack

Figure 5.6: Manual Failing of a Pattern

```
rule iVarNeverZero
   replace [statement]
      'if ( Cond [condition] )
         Stmt [statement]
   if
      deconstruct Cond
         'i != 0
      by
         Stmt
   else if
      ...
end rule
```

```
rule iVarNeverZero
   replace [statement]
      _This [statement]
   deconstruct _This
      'if ( Cond [condition] )
         Stmt [statement]
   construct _ [any]
      _ [_repl_stack_push 0]
   construct _result_1 [statement]
      _This
         [_branch1 Cond Stmt]
         [_branch2 Cond Stmt]
   construct _repl_top_1 [number]
      _ [_repl_stack_pop]
   deconstruct _repl_top_1
      1
   by
      _result_1
end rule
```

```
function _branch1 Cond [condition]
      Stmt [statement]
   construct _ [any]
      _ [_if_stack_push_false]
   replace [statement]
      _This [statement]
   deconstruct Cond
      'i != 0
   construct _ [any]
      _ [_if_stack_set_true 0]
   construct _ [any]
      _ [_repl_stack_set_top 1]
   by
      Stmt
end function
```

An ETXL example and corresponding implementation showing the manual failing of a pattern search. The purpose of this example rule is to remove tests that are known to be true. The principal pattern is contained in the top level scope and the replacements are contained in the sub rules implementing the if cases. Success of the replacement must be explicitly communicated to the rule in which the pattern is searched. Only the first case is shown.

mechanism.

As noted in Section 4.8, we wish that the transfer of out parameters be independent of the success or failure of a rule. Since any statement in a rule may cause termination, it is simply not practical to write values to the out parameter stack at the return point of a rule. Every clause in a rule is potentially a return point. Instead, we initialize out parameter values to a default value and then write them to their position on the stack immediately after they are bound. Each out parameter is represented on the stack by a boolean flag that indicates whether or not the value has been bound, and a variable of type `[any]` that stores the out parameter value.

At entry point, a rule that binds out parameters for its caller will immediately push a flag value of false and an empty value to the stack. One default value will be pushed for every out parameter the rule is expected to return. If the rule should fail before an out parameter is bound then the parameter's flag value will read false and the caller will be aware the out parameter was never bound. As the rule proceeds through clauses and variables that name out parameters are bound in the local scope, they are immediately written to their position in the current stack frame along with a flag value of true.

On the invocation side, the calling rule is responsible for popping values from the stack and creating local bindings. All out parameter flags and values are popped from the stack immediately after the invocation. After this, the values may be examined. These two tasks must be separated because examination of the value may cause the calling rule to fail. If this were to happen before all expected out parameters were popped from the stack then the out parameter stack and TXL's built-in stack would become out of sync. From that point on in the program's execution, out parameters would not work correctly.

Once all values have been popped, then it is verified that each value was bound using

Figure 5.7: Out Parameter Implementation

```
function deconstructElseIf_outp
   construct _ [any]
      _ [_op_stack_push] [_op_stack_push] [_op_stack_push]
   match [repeat else_if_part]
      'else 'if ( Condition [condition] )
         Statement [statement]
      Rest [repeat else_if_part]
   construct _ [any]
      _ [_op_stack_set_top 2 Rest]
   construct _ [any]
      _ [_op_stack_set_top 1 Statement]
   construct _ [any]
      _ [_op_stack_set_top 0 Condition]
end function
...
   construct _ [repeat else_if_part]
      ElseIfParts [deconstructElseIf_outp]
   construct Condition_stack_obj [opt _op_stack_obj]
      _ [_op_stack_pop]
   construct Statement_stack_obj [opt _op_stack_obj]
      _ [_op_stack_pop]
   construct Rest_stack_obj [opt _op_stack_obj]
      _ [_op_stack_pop]
   deconstruct Condition_stack_obj
      1 Condition_any_type [any]
   deconstruct * [condition] Condition_any_type
      Condition [condition]
   deconstruct Statement_stack_obj
      1 Statement_any_type [any]
   deconstruct * [statement] Statement_any_type
      Statement [statement]
   deconstruct Rest_stack_obj
      1 Rest_any_type [any]
   deconstruct * [repeat else_if_part] Rest_any_type
      Rest [repeat else_if_part]
```

The implementation of Figure 4.18. The rule `deconstructElseIf` *is responsible for first pushing empty values to the out parameter stack. Upon variable binding, out parameters are written into their position in the stack along with a flag indicating that they have been bound. After the rule invocation, which is shown here as a construct clause for simplicity, the caller pops the stack objects containing the values and flags. Then for each parameter the caller ensures it was written and extracts the value into a local variable of the appropriate type.*

the parameter's status flag. The out parameter value, which has been retrieved from the stack into a variable of type `[any]`, is then extracted and bound to a variable of the correct type. Figure 5.7 shows the TXL implementation of the ETXL code in Figure 4.18.

Notice that the implementation in Figure 5.7 shows the invocation as an anonymous construct when in fact in the original example the rule is invoked using a where clause. Our example implementation has been modified for simplicity because the implementation of an invocation via a where clause is indeed more complex. By definition, a where clause fails the clause list in which it is contained if the condition it implies is not met. This property would prevent the caller from popping default or otherwise bound values from the out parameter stack when the where clause failed, putting the out parameter stack out of sync with TXL's built-in rule invocation stack.

To solve this problem, we put where clauses that involve the transfer of out parameters into an if clause. The original where clause is put into the if test. The code for popping and binding the out parameters is inserted in the then block, followed by all the original clauses that were after the where clause. Finally, code that pops unread out parameters is written into the else case. This ensures that following a where clause, the out parameter stack is always in sync with TXL's built-in rule invocation stack.

5.8 Scoping and Forward Variable Declarations

Scoping is implemented by a dedicated scoping pass, during which dictionaries of names defined in each scope are maintained. As the scoping pass walks over clauses, it looks up every name encountered, associates the name with its origin scope and mangles the name with a number representing the scoping depth. The implementation of forward variables is incorporated into this scoping pass. Found variables that do not have types (as in construct

Figure 5.8: Scoping Implementation

```
function removeAssigns : _sc1_Assigns [repeat assignment_expression]
   if
      replace * [list init_declarator]
         _sc2_DeclId [id] = _sc2_AssignExpr [assignment_expression]
            , _sc2_Rest [list init_declarator]
   then
      construct _sc2_Result [list init_declarator]
         _sc2_DeclId , _sc2_Rest [removeAssigns :
            _sc2_TailAssigns [repeat assignment_expression]]
      construct _sc1_Assigns [repeat assignment_expression]
         _sc2_DeclId = _sc2_AssignExpr _sc2_TailAssigns
      by
         _sc2_Result
   else
      construct _sc1_Assigns [repeat assignment_expression]
         _
   end if
end function
```

Implementation of `removeAssigns` found in Figure 6.2. The `Assigns` variable is the only top level scope variable. All others are created in the second level scope.

---

clauses and out argument lists) have their types inserted if their type has been previously declared in a forward clause or if their name is an out parameter.

For example, Figure 5.8 shows the implementation of `removeAssigns`, found in Appendix B on page 130. Names have been mangled with their scope number. Types have been looked up and inserted.

5.9 Modularity

The modularity implementation is strictly a compile-time feature. It is based on name prefixing, a practice often adopted by TXL programmers wishing to emulate modularity.

Specifically, all entities inside of a module have their name prefixed with `_m_` and the module name. This ensures that names will not collide with entities of the same name that exist in other modules or in all likelihood with names that exist in the top level scope. All references to a module entity, explicitly qualified or otherwise implied, are located and also prefixed.

The modularity implementation is divided into several passes. On the first pass we build a list of names contained in each module, as well as in the top level scope. Then for each using statement we verify its legality. The public names imported by a using statement must not collide with any name that already exists in the scope. The collision check is performed for all items defined in the scope as well as all the public names already imported into the scope by previous using statements. Any failure in this regard causes a compile-time error.

On the second pass we look up unqualified names that refer to module entities and give them the proper qualification. Unqualified names are first looked up in the local module. If found they are qualified with the module name. If not found they are then looked up within the public entities of all imported modules. If still not found, names will go unchanged. Names that are intended to refer to an entity within a module but are never found will be interpreted as a reference to an entity that does not exist when the program is passed to the TXL engine.

On the final pass all qualifications are looked up, validated and mangled. When a reference to a qualified name is encountered, the ETXL compiler looks up the name in the list of the module's names and if found and is public, appropriately mangles the name by removing the dot and prepending the qualification to the name. If the name is not found within the module, or if it is private, the improper reference will generate a compile-time error. Validation is relevant to manually programmed qualifications only, as those inserted

Figure 5.9: Modularity Implementation

```
define _m_HTML_item
      [_m_HTML_begin_tag] [any] [opt _m_HTML_end_tag]
end define

define _m_HTML_begin_tag
      < [id] [repeat option] >
end define

define _m_HTML_end_tag
      < / [id] >
end define

function _m_HTML_boldize
   replace [any]
      A [any]
   construct BoldTag [_m_HTML_begin_tag]
      <B>
   by
      A [_m_HTML_tagwith BoldTag]
end function

function _m_HTML_tagwith Tag [_m_HTML_begin_tag]
   deconstruct Tag
      < TagId [id] TagOptions [repeat option] >
   replace [any]
      Anything [any]
   construct TaggedThing [_m_HTML_item]
      <TagId TagOptions> Anything </TagId>
   deconstruct TaggedThing
      TaggedAnything [any]
   by
      TaggedAnything
end function
   ...
   replace * [id]
      Id [id]
   by
      Id [_m_HTML_boldize]
```

The implementation of Figure 4.25. All module entities are moved to the top level scope and have their names prefixed with the module name.

in the previous pass are guaranteed to be correct. Figure 5.9 shows the implementation of the HTML markup module of Figure 4.25.

5.10 Summary

We have rapidly prototyped ETXL as a transformation. Since TXL is well suited to language-to-language translation we use it as the transformation tool. Since our new language is an extension of TXL we also use it as the target of the transformation. Using TXL has afforded us the time to concentrate on experimenting with the design of the new features. Once the new features have been proven in a production environment they may be transferred to the TXL engine.

Must-match rules use a stack of flags for indicating if a rule has made a match. Immediately upon entry a must-match rule pushes a false value. Immediately before returning it writes a true value to this flag. The caller pops the flag from the top of the stack and generates a run-time error if it is false.

The implementation of objectless rules involves detecting when a rule has no replace or match clause and inserting a match clause at the end of the rule that will match any type.

Strong typing is implemented during the scoping pass when the type of each variable is known. If a strongly typed rule is applied to a variable of the wrong type then a compile-time error is generated.

Implementing nested rules involves determining which variables that originate in a parent rule are accessed in a child. These variables are added to the child's parameter list and corresponding arguments are added to all invocations. The rule is moved to the top level scope.

Rule and type parameters are both implemented by substitution. For each set of rule

and type arguments passed to a rule, the rule is instantiated by replacing occurrences of the parameters with the literal values of the arguments.

Each if and else-if test is implemented by branching the rule into two subrules that are both executed unconditionally. The true branch uses a stack of flags to indicate success as in the implementation of must-match rules. The false branch verifies that the true branch failed before executing its clauses. Control does not return to the original rule. Both branches are responsible for implementing all following clauses.

The out parameter implementation uses a stack for returning out parameters from a called rule back to its caller. The called rule first pushes default values for each out parameter, then immediately after the parameter is bound it is written into its position on the stack. The caller pops values from the stack, ensures that they were written, then binds them to local values.

Scoping and forward variable declarations are implemented during a dedicated scoping pass, which is responsible for determining the originating declaration of each referenced variable and mangling the name with the scope depth.

All module entities are prefixed with their module name. References to module entities from within the module are also prefixed. References to public entities from outside of the module are located and again prefixed.

In describing the implementation of our new features, this chapter has discussed the overall approach of the transformation, gone into the details of the compile-time and run-time implementations, and given examples of the code generated by the transformation. The next chapter presents two real-world examples of ETXL features in use.

Chapter 6

Examples

This chapter explains two example programs that make use of ETXL features. The full listings of the examples are found in Appendices B and C. These examples solve real-world problems while making use of ETXL features where they fit naturally. They were chosen from a larger selection of examples on the grounds that they each showcase several ETXL features in combination.

6.1 Expand Vector Components

The first example, with the full listing found in Appendix B, finds expression statements where all terminals are either constants or three-dimensional vector objects and expands them into three statements, one for each vector component. This is useful in C++ for avoiding the creation of temporary variables used to store the result of each vector operation. The optimized code generated by this transformation is instead able to run the expression to completion on each vector component before moving to the next. Temporary values never leave the registers.

Figure 6.1: Type Conversion

```
function constructStatements FromType [type] From [repeat FromType]
    deconstruct From
        First [FromType] Rest [repeat FromType]
    construct Tail [repeat statement]
        _ [constructStatements FromType Rest]
    replace [repeat statement]
    by
        First ; Tail
end function
```

This example uses type parameters to construct a list of statements from any list of objects that may be promoted to a statement by appending a semi-colon. The rule is parameterized by the type that the conversion is made from and the list of items to be promoted. It returns a list of statements.

Type parameters are used in this example to write a generic type conversion. The rule `constructStatements`, shown in Figure 6.1, constructs a list of statements from a generic list of any objects that could be individually used to construct a statement. Since statements derive expressions and expressions derive many forms, there are a number of different types that could be converted to the statement type. For example, function calls and assignment expressions may both be promoted to the statement type. Normally a type change of a sequence of items requires a conversion to be written for each pair of types. Using type parameters we can eliminate that need.

This example makes heavy use of nested rules both for organizing code into sections and to a lesser extent for implicitly passing parameters. This example is divided in three major sections that are each encapsulated into a top level rule. The sections are: collecting `Vector` declarations (page 129), extracting initialization expressions from variable declarations (page 130) and expanding vector expressions into components (page 131). Most of the worker subroutines associated with each of these sections are expressed as nested rules.

This example uses objectless rules in combination with if clauses for catching the case

Figure 6.2: Extracting Initializations from Declarations

```
function removeAssigns : Assigns [repeat assignment_expression]
   if
      replace * [list init_declarator]
         DeclId [id] = AssignExpr [assignment_expression]
            , Rest [list init_declarator]
   then
      forward TailAssigns [repeat assignment_expression]
      construct Result [list init_declarator]
         DeclId , Rest [removeAssigns : TailAssigns]
      construct Assigns
         DeclId = AssignExpr TailAssigns
      by
         Result
   else
      % Got to the end of the list, return empty assignments.
      construct Assigns
         _
   end if
end function
```

The `removeAssigns` *routine removes initialization expressions from a list of variable declarations and returns them as a list of assignment expressions.*

---

that a rule fails to match anything. In the rule `removeAssigns`, shown in Figure 6.2, if the main pattern does not match an item in the list it is traversing, then it is assumed that the end of the list is reached and the rule defaults to the else case, which initializes the out parameter it is expected to return. In this branch there is no pattern match. In existing TXL, catching when the end of a list is arrived at during a walk requires a separate rule. In ETXL we are able to express the task of walking the list with a single rule.

The rule `removeAssigns` is interesting for another reason; it achieves an effect that is not possible in existing TXL. This rule removes initializers from declarations and returns them in an out parameter. In general terms, it simultaneously modifies a parse tree and

returns a result based on the modification without using any global variables in the process. This cannot be achieved in TXL. The need to simultaneously modify and return values from a tree is a task that surfaces often. Using out parameters we are easily able to program it without the use of globals variables.

6.2 Generic AVL Tree Module

This example, with the full listing found in Appendix C, shows a generic AVL tree module written in ETXL. It is a good example of using the new features to create a module that integrates easily with any program. The AVL tree supports insert and find, and because it is a balanced binary tree, they both run in logarithmic time. This module would be used to implement a symbol table.

Since the tree is used for storing symbols, the type of its key is fixed to the identifier type. However, the value that it stores can be of any type. Type independence in the value is useful because often the type of the data associated with an entry in a symbol table is application specific. Therefore the example makes use of type parameters. All routines that deal with the value type are parameterized by it.

A key feature of this example is the use of modularity. All code is encapsulated in the AVL module. Only the tree data type and the insert and find functions are made public. Everything else is hidden from the user. This minimal interface makes the module fairly easy to understand and use.

The find routine, shown in Figure 6.3, operates on a tree and utilizes an out parameter to return its result. As mentioned in the out parameter discussion of Section 4.8, moving the result to an out parameter enables a query function to be coded as a matching function and consequently used in a where clause. If the find function fails to locate a given key

Figure 6.3: AVL Module Find Function

```
% Find function. Fails the match if the item is not in the tree.
% Always binds OptValue.
function [tree] find Key [id] ValType [type] : OptValue [opt ValType]
   if
      match [tree]
         'nil
   then
      % Construct an empty result, but fail the function.
      construct OptValue _
      deconstruct not Key Key
   else
      match [tree]
         NodeKey [id] NodeVal [any] Height [number] Left [tree] Right [tree]
      if where
         Key [< NodeKey]
      then
         % Go left.
         where
            Left [find Key ValType : OptValue]
      else if where
         NodeKey [< Key]
      then
         % Go right.
         where
            Right [find Key ValType : OptValue]
      else
         % Hit the target.
         deconstruct * [ValType] NodeVal
            TypedVal [ValType]
         construct OptValue
            TypedVal
      end if
   end if
end function
```

The module's find routine. It makes use of strong typing, out parameters and if clauses. In this example of find the out parameter is always bound, regardless of the result of the find. The first if test catches the case that the key is not located and binds an empty return value. Since the failure case is explicitly handled the function must be deliberately failed.

then it will fail any where clause it is in.

The out parameter that find uses to return its value is an optional type. If a search key is not found then an empty result is returned. This means that if the find routine is used in a construct clause then it will never fail the calling clause list. This allows a user to search for a value and continue onward regardless of whether or not a value was returned. Processing the return value can be handled elsewhere in the program. The find routine is therefore flexible in how it may be used. Calling it from a where clause will cause the find to fail the clause list if the key is not found and calling it from a construct clause will always return a result that may be processed at a later time.

The strong typing feature that we have introduced is also utilized in this example. The insert and find routines are both strongly typed. The use of strong typing makes it easy to see how the insert and find are intended to be used because the type of the rule is shown directly in the signature. Note however, that using strong typing here is only a matter of good style. It does not affect the compiler's ability to report errors. This is because insert and find are both non-searching rules. Without the strong typing, the TXL engine would still produce an error if insert or find were applied to a non-tree type because their patterns could never match.

Following the module, in the top level name space, are driver insert and find routines (page 140) that test the AVL functionality. The insert driver reads key and value pairs from a list and inserts some of them into the tree. The find driver looks up every key from the same list and reports on the results. Some of the keys will not be found. Notice how the top level scope can have names that also exist in the module without problems. Access to the public names of the module is accomplished by qualifying them with the module name.

The find driver takes advantage of if clauses in order to implement more compact code

than previously possible. The find function is invoked in a where clause, which is in turn put into an if clause. Should the key be found, the associated value is printed. If the find fails, the case is handled immediately by printing a message. In existing TXL, the handling of this case would need to be moved to a separate rule, making the logic of the driver more difficult to follow.

If clauses are also prevalent in inserting, finding and rebalancing. Insert and find can both be expressed as single routines using if clauses. Previously, each case would need to be expressed in a function of its own. This improves readability considerably.

The rebalancing code searches for tree nodes that are lopsided in height and corrects them by rearranging the node and its descendents in a manner that levels out the tree while preserving the ordering of nodes. There are four possible tree configurations that require reblancing. Consider the rule `rebalCase1` (page 135), which is responsible for handling two of these cases. This rule first creates many variables that are used later in both cases. If we were using the old form of selection, the creation of all of these variables would need to be duplicated, once for each case. Using if clauses we can instead create the variables once, select on their values, then use them in constructing the replacement.

Lastly, this example makes use of nested rules for grouping code. The `calcHeight` function shown in Figure 6.4 requires the support of a function for calculating the maximum of two integers. Though a max routine could certainly be a reusable function, the goal of this example is to provide a generic dictionary, not a math library. Therefore the max function is considered private to `calcHeight`. Writing it as a nested rule helps to show this property.

Figure 6.4: Private Max Function

```
% Calculate the height of a single tree node, assuming all
% children have correct height.
function calcHeight
   % Return the max of two numbers.
   function max N2 [number]
      replace [number]
         N1 [number]
      where
         N1 [< N2]
      by
         N2
   end function

   replace [tree]
      Key [id] Val [any] Height [number] Left [tree] Right [tree]
   construct LH [number]
      _ [getHeight Left]
   construct RH [number]
      _ [getHeight Right]
   by
      Key Val LH [max RH] [+ 1] Left Right
end function
```

An example of declaring small, private helper routines as nested rules.

6.3 Summary

This chapter highlights the use of our new features in the solution of two real-world problems. The first example is a tool for optimizing expressions involving numerical vector objects in C++. It makes use of objectless rules and if clauses to handle the case that a rule fails to match anything. It uses nested rules for implicitly passing parameters and grouping code. Type parameters are used to write a type conversion from a list of any type to a list of statements. Out parameters are used to write a rule that simultaneously transforms a tree and returns data based on the transformation without the use of global variables.

The second example is a generic AVL tree module. It makes use of strong typing for indicating the use requirements of the interface directly in the rule signatures. It uses nested rules for grouping code. It uses type parameters for making a type-independent data structure. It uses if clauses for complex decision making throughout. It uses out parameters for returning values from the interface and modularity for encapsulating the code into a module that can be easily integrated with any existing program.

These examples show how ETXL is useful in solving traditional program transformation problems, as well as in solving problems that arise in general-purpose programming, such as a generic dictionary. In the following chapter we summarize this work, suggest future directions for the design of ETXL and give several other changes that were not included in this work.

Chapter 7

Conclusion

7.1 Summary

The goal of this thesis is to improve the TXL programming language. We set out to enhance the language in response to growing TXL application sizes and an expanding application domain. In some cases, TXL programs have become huge and help from the best practices of software engineering is needed. In other cases, novel uses for TXL have been found and TXL must grow to keep pace with these new demands. We hope to make TXL easier to use, TXL programs more concise and easier to understand, and to make it easier for programmers to express their intent.

We have introduced nine features that independently enhance TXL. These features introduce new paradigms and enable new kinds of TXL programs to be written. By combining the features, we find that more language effects are possible, such as parameterizable patterns, or rules that simultaneously modify a tree and return data based on the modification.

In TXL programming, it is not uncommon to write rules that are expected to match.

There is no facility in the language for declaring this property of rules. Therefore, it is up to the programmer to see to it that rules that should always be matching are actually matching. If we are able to declare as a part of the language that a rule is to always match then we can instead have the engine assert this property. To this end we have added must-match rules. The must keyword, prepended to a replace clause, can now be used to declare that a rule is to always match. This frees the programmer from writing verification code.

Often, rules are written that do not transform or match any tree in particular. Instead they operate only on parameters or global variables. Therefore we recognize that not all rules are strictly either a replace or a match rule. Some rules are objectless rules and do not have a match or replace clause. We implement objectless rules by inserting a match clause that will match any tree to which the rule is applied.

An ability to apply a rule to any tree root is a useful feature and coincides with typical grammar construction practices. Sometimes a rule is written such that it should only be applied to a single type. In an environment where code is written and used by different people, specifying the type that a rule may be applied to communicates the intent of the programmer. Therefore we introduce the ability to strongly type rules. An application of a strongly typed rule to the wrong type generates a compile-time error.

In TXL, the need to propagate data down the traversal of a parse tree surfaces often. This is typically handled by explicitly propagating data through parameters. In some cases this can be many variables. We have added nested rules with the ability to reference variables originating in an ancestor in an effort to reduce the need to explicitly pass parameters. In the implementation of this feature, referenced variables are implicitly passed as parameters.

In TXL programming, it is not uncommon to write multiple rules that differ only by

the rules that they apply or by the types that they operate on. This can happen, for example, when writing traversals over data structures or when writing algorithms over types that share structural properties. To meet these needs we have introduced rule and type parameters, which allow a rule to be parameterized by the entities that vary. This allows the code to be written once and reused. Both rule and type parameters are implemented by substitution.

TXL finds use in a variety of programming situations. The need for complex decision making often arises. In existing TXL, rule application semantics have been borrowed for the purpose of branching, which works, but often is awkward and error prone. We have added a native branching clause that allows complex decision making. The if clause may contain any clause provided that every path through the rule is a correct pattern-replace rule or matching rule. If clauses are implemented by utilizing subrules to handle the cases and a stack of boolean variables for communicating if test status.

In TXL there is no way to abstract away a pattern. A developer must have access to type definitions in order to take apart a tree. We have added out parameters for transferring data from a called rule back to a caller. Out parameter semantics follow pattern matching semantics. If a rule fails to bind an out parameter then the caller cannot proceed past the point of invocation. This mechanism is independent of a rule's success or failure. Out parameters are implemented using a dedicated stack for returning values back to the caller.

If we define rule parameters that are expected to return results via out parameters then in effect we have gained the ability to parameterize patterns. This is useful if the semantics of a particular deconstruct is fixed, but the specific syntax varies among different patterns.

To accommodate the block structure we have introduced via if clauses and nested rules, we have added variable name scopes. A variable name in a nested block, for example in an

if test, may reference a variable declared in a parent. A newly bound variable in a nested block may hide a variable declared in a parent block. Scoping is implemented by mangling variable names with their scoping depth number.

In order to permit nested blocks to declare a variable that is accessed in a parent, for example for selecting variable values using an if clause, we have added the forward variable declaration. Forward variable declarations affect the scoping depth number used to mangle names during the scoping pass.

To address larger programs on which multiple developers work we have introduced modularity capabilities to TXL. Modularity allows a developer to wrap a section of code in module opening and closing statements and thereby make the code private. Interfaces may be declared by designating rules, type defines and global variables as public. Public entities may be accessed outside of the module by qualification with the module name. Alternatively, a commonly used module may have its public names imported with the using statement.

We have presented the solution to two real-world problems that take advantage of our new features. The first is a classic example of program transformation. The second is an implementation of a data structure that is a rare task for program transformation systems.

Our language changes have been implemented in a working prototype which is available from the ETXL home page [1]. We encourage users to download it, try it out and submit feedback to the author.

7.2 Future Work

This section describes future directions for the design of ETXL. Some design issues were not resolved and therefore the design would benefit from further study. In this section we

also address moving the existing design into the engine, which is a more concrete goal.

As noted in the design discussion on type parameters in Section 4.6, an ability to specify which type parameters accept only pure types and which type parameters accept any type at all may be beneficial. This would communicate in the signature of the type parameter which types may be passed as arguments.

An argument to a type parameter is passed using `[]` delimiters. This is consistent with the use of these delimiters whenever referencing a type in TXL. It may be worthwhile to investigate the passing of type parameters without these delimiters in order to resolve the problems associated with nesting them, as discussed in Section 4.6 on type parameter design.

Again concerning type parameters, in large type-independent modules there can be frequent type parameter passing. Every rule that needs to reference the parameterized types must take the type as a parameter and must have the type passed to it by all callers. It may worthwhile if a type parameter could act across a collection of rules. The solution might be to pass type parameters to modules. The same argument may be applied to rule parameters.

Another problem that might be solved in a similar way is the fact that type parameters can only be passed to rules. If writing a module that has a private data type such as the dictionary module described in Section 6.2, it would be helpful if types could be parameterized in grammar definitions as well.

Unfortunately, out parameters in their current design cannot be used in combination with TXL's built-in iteration for the purpose of collecting a list of items. Out parameters are only useful for collecting lists if programming explicit iteration using recursion. This limitation can be seen in `getDeclarations` of Appendix B (page 129). This example

uses TXL's built-in search facility to avoid the explicit traversal of a large subset the C++ grammar and so it must use global variables. A change to the out parameter design to permit this may therefore be worthwhile.

It is our hope that existing TXL programmers will try out our prototype. Any developer feedback we are able to acquire will be very useful in working out design bugs and determining the practical utility of new features. We are especially interested in how our changes will fair in a production environment. Once the features are deemed production quality, their design is settled and all the inconsistencies are ironed out, we hope to transfer this work directly into the TXL engine.

The transfer process should be incremental so that the correctness of the engine may be frequently asserted. Some of the new features are easily implemented independently. Objectless rules, strong typing, modularity, must-match rules and out parameters are good examples these. Other features must be transfered together. For example scoping and if clauses should be implemented together. Initial analysis of the dependencies between the features would help to avoid unnecessarily adding several at once.

7.3 Other Changes

This section briefly describes other proposed language changes. These changes are largely cosmetic in nature and do not constitute new features. Being a matter of convenience, these changes were not covered by our research goals, however they are interesting from a usability point of view. These may be worthy of future pursuits, especially if one set out to update TXL's syntax, a task we have not tackled.

- Change the `define` keyword to `type` to better reflect that a grammar definition represents a data type in the TXL programming language. Much of the TXL documentation describes grammar definitions as types and thus has shown the keyword `type` to be a natural name.

- Allow the application of rules to rule arguments. In existing TXL, the application of a rule to a tree which is to be passed as an argument is handled by simply constructing the argument in a clause immediately preceding the rule application. Allowing the application inline enables the programmer to express simple applications more concisely.

- Allow the use of nameless constructors in tree expressions. Like the application of functions to function arguments, nameless constructors enable code to be expressed more concisely. Nameless constructors would be usable anywhere a typed variable is allowed and would take the following form in tree expressions:

```
[type arg1 arg2 arg3]
```

- Enumerate all the different search characteristics of rules, such as searching rule, one-pass rule, function and searching function, and permit the declaration of the search type of a rule using only a single keyword. The current combination of keyword and symbol modifier is unnecessarily complex.

- Allow arbitrary boolean expressions using `AND` and `OR` operators in where clauses. In TXL, composing complex boolean expressions from where clauses using the non-standard forms `where all` and `where not` usually requires careful attention. The functionally equivalent boolean operators are far more familiar to programmers.

7.4 Conclusion

As the size of TXL programs grow and the breadth of applications increases, it becomes necessary to respond to the needs of TXL programmers. This work attempts to do just that by adding features to the TXL programming language that aid in the writing of large programs, that encourage common software engineering practices and that otherwise facilitate the writing of better TXL programs. In addressing these needs we have developed ETXL, the next generation in TXL's evolution.

Appendix A

Unified ETXL Grammar

This appendix presents the entire ETXL grammar. Modifications necessary to support ETXL features have been integrated into the base TXL grammar. We use TXL's syntax definition facilities as the notation for describing the ETXL syntax. This form is described in detail in the TXL language specification [Cordy03]. This grammar is functionally equivalent to the grammar used in the prototype implementation.

```
% Grammar for ETXL processing. The input ETXL program must first
% be filtered through the norm program and then afterwards
% through unnorm before being processed by TXL. These filters
% handle constructs requiring special attention such as pragmas,
% comment declarations and compounds. Both filters are found
% in the ETXL distribution.

tokens
   quotedlit
      % Gets the more common case. Note that brackets
      % can only occur as the first char
      "'#[ \t\n'\"]#[ \t\n\[\]'\"]*" |

      % Get the special case '"string"
      "'\"[(\\\c)#\"]*\"" |
```

```
       % Get the special case ''string'. Note that the
       % single quoted string cannot have a newline.
       "''[(\\\c)#['\n]]*'" |

       % Get the special case '' or ' '
       "'[ \t\n]*'" |

       % And finally if all else fails.
       "'"
end tokens

comments
   '%
end comments

compounds
   ... <= >= ~=
end compounds

keys
   'all 'assert 'attr 'by 'comments 'compounds 'construct
   'deconstruct 'define 'each 'end 'export 'external 'function
   'import 'include 'keys 'list 'match 'not 'opt 'redefine
   'repeat 'replace 'rule 'see 'skipping 'tokens 'where 'module
   'public 'using 'forward 'type 'must 'if 'then 'else
end keys

define txl_program
       [repeat statement]
end define

%
% Statements
%

define statement
       [include_stmt] | [pragma_stmt] | [comment_stmt]
    |  [keys_stmt] | [define_stmt] | [rulefunc_stmt]
    |  [tokens_stmt] | [compounds_stmt] | [module_stmt]
    |  [using_stmt]
end define
```

```
define include_stmt
       'include [stringlit]
end define

define pragma_stmt
       '# 'pragma [stringlit]
end define

define comment_stmt
       'comments
          [repeat comment_inst]
       'end 'comments
end define

define comment_inst
       [stringlit]
end define

define keys_stmt
       'keys
          [repeat keys_token]
       'end 'keys
end define

define keys_token
       [token]
   |   [not 'end] [key]
end define

define define_or_redefine
       'define
   |   'redefine
end define

define define_stmt
       [define_or_redefine] [id]
          [productions]
       'end [define_or_redefine]
end define

define rulefunc_stmt
       [rule_stmt]
   |   [func_stmt]
```

```
end define

define rule_stmt
       'rule [rule_def_sig]
          [repeat rule_clause+]
       'end 'rule
end define

define func_stmt
       'function [rule_def_sig]
          [repeat rule_clause+]
       'end 'function
end define

define rule_def_sig
       [opt nonterm_ref] [id] [param_list]
end define

define tokens_stmt
       'tokens
          [repeat token_define]
       'end 'tokens
end define

define token_define
       [id] [opt token_sep] [token_patterns]
end define

define token_patterns
       [token_pattern_item] [token_sep] [token_patterns]
    |  [token_pattern_item]
end define

define token_pattern_item
       [stringlit]
    |  '...
end define

define token_sep
       '|
    |  '+
end define
```

```
define compounds_stmt
       'compounds
          [repeat stringlit]
       'end 'compounds
end define

define compounds_token
       '... | '. | [token]
end define

define module_stmt
       'module [id]
          [repeat statement_inmod]
       'end 'module
end define

define statement_inmod
       [define_stmt] | [rulefunc_stmt]
    |  [public_stmt] | [using_stmt]
end define

define public_stmt
       'public
          [repeat id]
       'end 'public
end define

define using_stmt
       'using [id]
end define

%
% Productions
%

define productions
       [production]
       [repeat more_productions]
end define

define more_productions
       '| [production]
end define
```

```
define production
       [repeat production_el]
end define

define production_el
       '...
    |  [nonterm_ref]
    |  [term_ref]
end define

%
% Terminals and Non-Terminals
%
define nonterm_ref
       '[ [nonterm_ref_cont] ']
end define

define nonterm_ref_cont
       [id] '. [id]
    |  [id]
    |  [nonterm_premod] [nonterm_id_or_term] [opt nonterm_postmod]
    |  '!
    |  [nonterm_depreciated]
end define

define nonterm_depreciated
       [nonterm_id_or_term] '?
    |  [nonterm_id_or_term] '*
    |  [nonterm_id_or_term] '+
    |  [nonterm_id_or_term] ',
    |  [nonterm_id_or_term] ', '+
    |  ': [nonterm_id_or_term]
    |  '~ [nonterm_id_or_term]
end define

define nonterm_id_or_term
       [id] '. [id]
    |  [id]
    |  [quotedlit]
end define

define term_ref
```

```
        [id]
    |   [term_quote_not_needed]
    |   [number]
    |   [stringlit]
    |   [quotedlit]
end define

define term_quote_not_needed
        '! | '# | '$ | '& | '( | ') | '* | '+ | ',
    |   '- | '. | '/ | ': | '; | '< | '= | '> | '?
    |   '@ | '\ | '^ | '_ | '` | '{ | '} | '~
    |   '<= | '>= | '~=
end define

define nonterm_premod
        'opt | 'repeat | 'list | 'see | 'not | 'attr
end define

define nonterm_postmod
        '+
end define

%
% Rule Clauses
%

define rule_clause
        [replace_clause] | [match_clause] | [where_clause]
    |   [assert_clause] | [skipping_clause] | [import_clause]
    |   [export_clause] | [deconstruct_clause] | [construct_clause]
    |   [forward_clause] | [rulefunc_stmt] | [if_clause]
    |   [by_clause]
end define

define replace_clause
        [opt 'must] 'replace [opt replace_modifier] [nonterm_ref]
            [pattern]
end define

define match_clause
        'match [opt replace_modifier] [nonterm_ref]
            [pattern]
end define
```

```
define replace_modifier
       '* | '$
end define

define where_clause
       'where [opt 'not] [opt 'all]
          [tree_expression]
end define

define assert_clause
       'assert [opt 'not] [opt 'all]
          [tree_expression]
end define

define skipping_clause
       'skipping [nonterm_ref]
end define

define import_clause
       'import [global_id] [opt nonterm_ref]
          [opt pattern]
end define

define export_clause
       'export [global_id] [opt nonterm_ref]
          [opt replacement]
end define

define global_id
       [opt module_qual] [id]
end define

define module_qual
       [id] '.
end define

define deconstruct_clause
       'deconstruct [opt 'not] [opt '*] [opt nonterm_ref] [id]
          [pattern]
end define

define construct_clause
```

```
       'construct [id] [opt nonterm_ref]
          [replacement]
end define

define forward_clause
       'forward [id] [nonterm_ref]
end define

define if_clause
       'if
          [repeat rule_clause]
       [opt then_block]
       [repeat elseif_clause]
       [opt else_clause]
       'end 'if
end define

define elseif_clause
       'else 'if
          [repeat rule_clause]
       [opt then_block]
end define

define then_block
       'then
          [repeat rule_clause]
end define

define else_clause
       'else
          [repeat rule_clause]
    |  'else 'else
          [repeat rule_clause]
end define

define by_clause
       'by
          [replacement]
end define

%
% Parameter List
%
```

```
define param_list
       [repeat parameter] [opt out_param_list]
end define

define out_param_list
       ': [repeat parameter]
end define

define parameter
       [id] [parameter_type]
end define

define parameter_type
       [nonterm_ref]
    |  [rule_type]
    |  [type_type]
end define

define rule_type
       '[ [opt nonterm_ref] [rule_type_kw] [rule_type_param_list] ']
end define

define rule_type_kw
       'rule
    |  'function
end define

define rule_type_param_list
       [repeat parameter_type] [opt rule_type_out_param_list]
end define

define rule_type_out_param_list
       ': [repeat parameter_type]
end define

define type_type
       '[ 'define ']
    |  '[ 'type ']
end define

%
% Patterns
```

```
%

define pattern
       [repeat pattern_el]
end define

define pattern_el
       [id]
    |  [pattern_lit]
    |  [id] [nonterm_ref]
end define

define pattern_lit
       [term_quote_not_needed]
    |  [number]
    |  [stringlit]
    |  [quotedlit]
end define

%
% Replacements
%

define replacement
       [repeat replacement_el]
end define

define replacement_el
       [replacement_lit]
    |  [tree_expression]
end define

define replacement_lit
       [term_quote_not_needed]
    |  [number]
    |  [stringlit]
    |  [quotedlit]
end define

%
% Tree Expressions
%
```

```
define tree_expression
       [id] [repeat function_app]
end define

define function_app
       '[ [function_name] [repeat function_arg] [opt out_arg_list] ']
end define

define out_arg_list
       ': [repeat out_arg]
end define

define out_arg
       [id] [opt nonterm_ref]
end define

define function_name
       [id] '. [id]
    |  [id]
    |  [term_quote_not_needed]
end define

define function_arg
       'each
    |  [function_arg_lit]
    |  [id] '. [id]
    |  [id]
    |  [nonterm_ref]
end define

define function_arg_lit
       [term_quote_not_needed]
    |  [number]
    |  [stringlit]
    |  [quotedlit]
end define
```

Appendix B

Example: Expand Vector Components

This appendix contains the full listing of the example described in Section 6.1. This example finds expression statements where all terminals are either constants or three-dimensional vector objects and expands them into three expressions, one for each vector component.

```
include "cpp.grm"

define program
       [cpp_program]
end define

% Construct a list of statements from a generic list of objects.
% Assumed that the objects are not terminated with a semi-colon.
function constructStatements Type [type] From [repeat Type]
   deconstruct From
      First [Type] Rest [repeat Type]
   construct Tail [repeat statement]
      _ [constructStatements Type Rest]
   replace [repeat statement]
   by
      First ; Tail
end function
```

```
% Return a list of identifiers representing all declarations of
% type DeclSpec regardless of scope. Because the rules that
% search for declarations use TXL's built-in search, we cannot
% eradicate all use of global variables.
function getDeclarations DeclSpec [id] : Declarations [repeat id]

   % Retrieve declarations from parameter lists.
   rule getParams
      replace $ [parameter_declaration]
         DeclSpec ParamDeclarator [parameter_declarator]
            ParamInit [opt parameter_init]
      deconstruct ParamDeclarator
         ParamDeclaratorId [id]
      import GetDeclarations_VectorDecls [repeat id]
      export GetDeclarations_VectorDecls
         ParamDeclaratorId GetDeclarations_VectorDecls
      by
         DeclSpec ParamDeclarator ParamInit
   end rule

   % Retrieve identifiers from a list of init declarators.
   rule getInitDecls
      replace $ [init_declarator]
         Declarator [declarator] OptInit [opt initializer]
      deconstruct Declarator
         DeclaratorId [id]
      import GetDeclarations_VectorDecls [repeat id]
      export GetDeclarations_VectorDecls
         DeclaratorId GetDeclarations_VectorDecls
      by
         Declarator OptInit
   end rule

   % Get declarations from a simple declaration.
   rule getDecls
      replace $ [simple_declaration]
         DeclSpec InitDeclaratorList [list init_declarator] ;
      by
         DeclSpec InitDeclaratorList [getInitDecls] ;
   end rule

   match [any]
```

```
        Any [any]
    export GetDeclarations_VectorDecls [repeat id] _
    construct _ [any]
        Any [getParams] [getDecls]
    import GetDeclarations_VectorDecls
        Declarations [repeat id]
end function

% Move initializer expressions to following statements.
rule breakInitializers
    % Remove assignments and return them in the assignments out
    % parameter.
    function removeAssigns : Assigns [repeat assignment_expression]
        if
            replace * [list init_declarator]
                DeclId [id] = AssignExpr [assignment_expression]
                    , Rest [list init_declarator]
        then
            forward TailAssigns [repeat assignment_expression]
            construct Result [list init_declarator]
                DeclId , Rest [removeAssigns : TailAssigns]
            construct Assigns
                DeclId = AssignExpr TailAssigns
            by
                Result
        else
            % Got to the end of the list, return empty assignments.
            construct Assigns
                _
        end if
    end function

    replace $ [repeat statement]
        'Vector InitDeclaratorList [list init_declarator] ;
        RestStmt [repeat statement]

    % Pull out the inits of simple identifiers and move them to
    % following statements.
    forward Assigns [repeat assignment_expression]
    construct StrippedDecl [statement]
        'Vector InitDeclaratorList [removeAssigns : Assigns] ;
    construct InitExprs [repeat statement]
        _ [constructStatements [assignment_expression] Assigns]
```

```
    by
        StrippedDecl InitExprs [. RestStmt]
end rule

% Find expressions where all postfix expressions are either
% constants or vectors and expand the expression into three
% expressions, one for each component.
rule componentExpand VectorDecls [repeat id]

    % Is a postfix expression a reference to a vector identifier?
    function isVectId
        match [postfix_expression]
            Id [id]
        deconstruct * [id] VectorDecls
            Id
    end function

    % Is a postfix expression a literal number?
    function isNumber
        match [postfix_expression]
            _ [number]
    end function

    % Is a postfix expression a paren expression?
    function isParen
        match [postfix_expression]
            ( _ [expression] )
    end function

    % Look for postfix expressions that are not references to
    % vectors or not literal numbers.
    function nonId
        match * [postfix_expression]
            PostfixExpression [postfix_expression]
        where not
            PostfixExpression [isParen] [isVectId] [isNumber]
    end function

    rule appendComponent Component [id]
        replace $ [postfix_expression]
            Id [id]
        by
            Id . Component
```

```
    end rule

    replace $ [statement]
        Expression [expression] ;

    % We are interested in expressions in which there is not one
    % postfix_expression that does not have the necessary
    % property of being either a reference to a vector or being a
    % number.
    where not
        Expression [nonId]
    by
        {
            Expression [appendComponent 'x] ;
            Expression [appendComponent 'y] ;
            Expression [appendComponent 'z] ;
        }
end rule

rule componentExpandInFunctions
    replace $ [function_definition]
        FunctionDefinition [function_definition]
    % Get the local declarations of vector objects.
    forward VectorDecls [repeat id]
    construct _ [function_definition]
        FunctionDefinition
            [getDeclarations 'Vector : VectorDecls]
    by
        FunctionDefinition
            [breakInitializers]
            [componentExpand VectorDecls]
end rule

function main
    replace [program]
        P [program]
    by
        P [undoDisambig] [componentExpandInFunctions]
end function
```

Appendix C

Example: AVL Tree Module

This appendix contains the full listing of the example described in Section 6.2. This is an example of a generic AVL tree module that is useful as a symbol table. The tree supports insert and find. It can be easily dropped into any application.

```
% Generic AVL tree that supports insert and find. Can be used to
% store any type.

module AVL

public
   tree insert find
end public

% Key, Value, Height, Left, Right
define tree
      [id] [any] [number] [tree] [tree]
   |  'nil
end define

% Getting a node's height
function getHeight Tree [tree]
   replace [number]
      N [number]
```

```
    if deconstruct Tree
        'nil
    then by
        0
    else if deconstruct Tree
        _ [id] _ [any] Height [number] _ [tree] _ [tree]
    then by
        Height
    end if
end function

% Calculate the height of a single tree node, assuming all
% children have correct height.
function calcHeight
    % Return the max of two numbers.
    function max N2 [number]
        replace [number]
            N1 [number]
        where
            N1 [< N2]
        by
            N2
    end function

    replace [tree]
        Key [id] Val [any] Height [number] Left [tree] Right [tree]
    construct LH [number]
        _ [getHeight Left]
    construct RH [number]
        _ [getHeight Right]
    by
        Key Val LH [max RH] [+ 1] Left Right
end function

%
% Rebalancing
%

function leftLong RH [number] DesiredDiff [number]
    match [number]
        LH [number]
    where
        RH [< LH]
```

```
   construct Diff [number]
      LH [- RH]
   where
      Diff [= DesiredDiff]
end function

function rightLong RH [number] DesiredDiff [number]
   match [number]
      LH [number]
   where
      LH [< RH]
   construct Diff [number]
      RH [- LH]
   where
      Diff [= DesiredDiff]
end function

% Handle an imbalance on the left.
function rebalCase1
   replace [tree]
      Key [id] Val [any] Height [number] Left [tree] Right [tree]
   deconstruct Left
      LKey [id] LVal [any] LHeight [number] LLeft [tree] LRight [tree]
   construct LLH [number]
      _ [getHeight LLeft]
   construct LRH [number]
      _ [getHeight LRight]
   if
      %     gp
      %    /
      %   p
      %  /
      % n
      where
         LLH [leftLong LRH 1]
   then
      deconstruct LLeft
         LLKey [id] LLVal [any] LLHeight [number]
                LLLeft [tree] LLRight [tree]
      construct NewLeft [tree]
         LLKey LLVal 0 LLLeft LLRight
      construct NewRight [tree]
         Key Val 0 LRight Right
```

```
      by
          LKey LVal LHeight NewLeft [calcHeight] NewRight [calcHeight]
   else if
      %     gp
      %    /
      %  p
      %   \
      %    n
      where
          LLH [rightLong LRH 1]
   then
      deconstruct LRight
          LRKey [id] LRVal [any] LRHeight [number]
                 LRLeft [tree] LRRight [tree]
      construct NewLeft [tree]
          LKey LVal 0 LLeft LRLeft
      construct NewRight [tree]
          Key Val 0 LRRight Right
      by
          LRKey LRVal LRHeight NewLeft [calcHeight] NewRight [calcHeight]
   end if
end function

% Handle an imbalance on the right.
function rebalCase2
   replace [tree]
      Key [id] Val [any] Height [number] Left [tree] Right [tree]
   deconstruct Right
      RKey [id] RVal [any] RHeight [number] RLeft [tree] RRight [tree]
   construct RLH [number]
      _ [getHeight RLeft]
   construct RRH [number]
      _ [getHeight RRight]
   if
      %  gp
      %     \
      %      p
      %     /
      %    n
      where
          RLH [leftLong RRH 1]
   then
      deconstruct RLeft
```

```
            RLKey [id] RLVal [any] RLHeight [number]
                   RLLeft [tree] RLRight [tree]
         construct NewLeft [tree]
            Key Val 0 Left RLLeft
         construct NewRight [tree]
            RKey RVal 0 RLRight RRight
         by
            RLKey RLVal RLHeight NewLeft [calcHeight] NewRight [calcHeight]
   else if
      % gp
      %  \
      %   p
      %    \
      %     n
      where
         RLH [rightLong RRH 1]
   then
      deconstruct RRight
         RRKey [id] RRVal [any] RRHeight [number]
                RRLeft [tree] RRRight [tree]
      construct NewLeft [tree]
         Key Val 0 Left RLeft
      construct NewRight [tree]
         RRKey RRVal 0 RRLeft RRRight
      by
         RKey RVal RHeight NewLeft [calcHeight] NewRight [calcHeight]
   end if
end function

%
% Rebalance, select between:
%
%    gp    gp
%   /   or   \
%  p          P
%
function rebal
   replace [tree]
      Tree [tree]
   deconstruct Tree
      _ [id] _ [any] _ [number] Left [tree] Right [tree]
   construct LH [number]
      _ [getHeight Left]
```

```
    construct RH [number]
        _ [getHeight Right]

    % Check for imbalance towards the left side.
    if where
        LH [leftLong RH 2]
    then by
        Tree [rebalCase1]

    % Check for imbalance towards the right side.
    else if where
        LH [rightLong RH 2]
    then by
        Tree [rebalCase2]
    end if
end function

% Insert function. If the node is in the tree already then this
% will silently fail.
function [tree] insert Key [id] ValType [type] Value [ValType]
    forward Result [tree]
    if
        % Hit a leaf, this is our insert point.
        replace [tree]
            'nil
    then
        % Init the new node.
        deconstruct Value
            ValueAny [any]
        construct Result
            Key ValueAny 0 'nil 'nil
    else
        % Everything else is a node, decide whether to go left or
        % right.
        replace [tree]
            NodeKey [id] NodeVal [any] Height [number] Left [tree] Right [tree]
        if where
            Key [< NodeKey]
        then construct Result
            NodeKey NodeVal Height Left [insert Key ValType Value] Right
        else if where
            NodeKey [< Key]
        then construct Result
```

```
            NodeKey NodeVal Height Left Right [insert Key ValType Value]
        end if
    end if

    % On the way back up we check for the rebalance condition and
    % recalculate heights. Rebal will only ever happen once due
    % to properties of the avl tree.
    by
        Result [rebal] [calcHeight]
end function

% Find function. Fails the match if the item is not in the tree.
% Always binds OptValue.
function [tree] find Key [id] ValType [type] : OptValue [opt ValType]
    if
        match [tree]
            'nil
    then
        % Construct an empty result, but fail the function.
        construct OptValue _
        deconstruct not Key Key
    else
        match [tree]
            NodeKey [id] NodeVal [any] Height [number] Left [tree] Right [tree]
        if where
            Key [< NodeKey]
        then
            % Go left.
            where
                Left [find Key ValType : OptValue]
        else if where
            NodeKey [< Key]
        then
            % Go right.
            where
                Right [find Key ValType : OptValue]
        else
            % Hit the target.
            deconstruct * [ValType] NodeVal
                TypedVal [ValType]
            construct OptValue
                TypedVal
        end if
```

```
    end if
end function

end module

%
% Module usage.
%

% Insert items that are to go into the dict.
rule insert
    replace $ [item]
        Key [id] Value [number] 'yes
    import Tree [AVL.tree]
    export Tree
        Tree [AVL.insert Key [number] Value]
    by
        Key Value 'yes
end rule

% Find all items.
rule find
    replace $ [item]
        Key [id] Value [number] ShouldInsert [key]
    import Tree [AVL.tree]
    forward Result [opt number]
    if
        where
            Tree [AVL.find Key [number] : Result]
    then
        construct _ [opt number]
            Result [print]
    else
        construct msg [stringlit] ""
        construct _ [stringlit]
            msg [+ "find failed on "] [quote Key] [print]
    end if
    by
        Key Value ShouldInsert
end rule
```

Bibliography

[1] http://www.cs.queensu.ca/home/thurston/etxl/.

[Boro98] P. Borovansky, C. Kirchner, H. Kirchner, P.E. Moreau and C. Ringeissen. An Overview of ELAN. *Proceedings of the 2nd International Workshop on Rewriting Logic and its Applications*, September 1998.

[Bran02] M.G.J. van den Brand, J. Heering, P. Klint and P.A. Oliver. Compiling Language Definitions: The ASF+SDF Compiler. *ACM Transactions on Programming Languages and Systems*, pages 334–368, July 2002.

[Bran03] M.G.J. van den Brand, P. Klint and J. Vinju. Term Rewriting with Traversal Functions. *ACM Transactions on Software Engineering and Methodology*, 12(2):152–190, April 2003.

[Cet96] M. Cettolo and A. Corazza and R. D. Mori. A Mixed Approach to Speech Understanding. In *Proc. ICSLP '96*, volume 2, pages 845–848, Philadelphia, PA, 1996.

[Cordy88] J.R. Cordy, C.D. Halpern and E. Promislow. TXL: A Rapid Prototyping System for Programming Language Dialects. *Proc. of IEEE 1988 International Conference on Computer Languages*, pages 280–285, October 1988.

[Cordy90] J.R. Cordy and E.M. Promislow. Specification and Automatic Prototype Implementation of Polymorphic Objects in Turing Using the TXL Dialect Processor. *Proc. IEEE 1990 International Conference on Computer Languages*, pages 145–154, March 1990.

[Cordy02] 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, October 2002.

[Cordy03] J.R. Cordy, I.H. Carmichael and R. Halliday. *The TXL Programming Language*. STL, School of Computing, Queen's University and TXL Software Research Inc., 1987-2003.

[Cordy04] J.R. Cordy. TXL - A Language for Programming Language Tools and Applications. *Proc. of the ACM 4th International Workshop on Language Descriptions, Tools and Applications (LDTA 2004)*, pages 1–27, April 2004.

[Dean01] T.R. Dean, J.R. Cordy, K.A. Schneider and A.J. Malton. Experience Using Design Recovery Techniques to Transform Legacy Systems. *ICSM 2001 - IEEE International Conference on Software Maintenance*, pages 622–631, November 2001.

[Dean02] T.R. Dean, J.R. Cordy, A.J. Malton and K.A. Schneider. Grammar Programming in TXL. *IEEE 2nd International Workshop on Source Code Analysis and Manipulation (SCAM'02)*, pages 93–102, October 2002.

[Grant03] S. Grant and J.R. Cordy. An Interactive Interface for Refactoring Using Source Transformation. *1st International Workshop on Refactoring: Achievements, Challenges, Effects (REFACE'03)*, pages 30–33, November 2003.

[Han81] David R. Hanson. Is Block Structure Necessary? *Software: Practice and Experience*, 11:853–866, 1981.

[Lin03] Yuan Lin and Richard C. Holt and Andrew J. Malton. Completeness of a Fact Extractor. In *WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering*, page 196. IEEE Computer Society, 2003.

[Parr94] Terence J. Parr. An Overview of SORCERER: A Simple Tree-Parser Generator, 1994.

[Rad99] Ansgar Radermacher. Support for Design Patterns Through Graph Transformation Tools. In *AGTIVE*, pages 111–126, 1999.

[Ric01] Filippo Ricca and Paolo Tonella. Analysis and testing of Web applications. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 25–34. IEEE Computer Society, 2001.

[Syn03] N. Synytskyy, J.R. Cordy and T.R.Dean. Robust Multilingual Parsing Using Island Grammars. *CASCON 2003, 13th IBM Centres for Advanced Studies Conference*, pages 149–161, October 2003.

[Vis99] Eelco Visser. Strategic Pattern Matching. In *Rewriting Techniques and Applications (RTA'99)*, volume 1631 of *Lecture Notes in Computer Science*, pages 30–44. Spinger-Verlag, July 1999.

[Vis01] Eelco Visser. Stratego: A Language for Program Transformation based on Rewriting Strategies. System Description of Stratego 0.5. In *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Spinger-Verlag, May 2001.

[Vis04] Eelco Visser. Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Spinger-Verlag, June 2004.

[Zan02] 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.