TXL Introduction

1. What is TXL?
   1. TXL is a programming language specifically designed to support
      1. Software analysis
      2. Program transformation
   2. TXL is a hybrid functional / rule-based language with
      1. Implied iteration
      2. Deep pattern match
2. Program Transformations
   1. Program transformation is the act of change one program into another
3. What is TXL good for
   1. Translations
      1. E.g.
         1. High-level language migration to another high-level language
         2. High-level language analysis to an aspect language
         3. High-level language synthesis to a low-level language
         4. Low-level language reversing engineering to high-level language
      2. Synthesis/code generation
         1. Derivation of a program from a specification
      3. Program migration
         1. Porting a Pascal program to C or C++ to Java
         2. Translation between dialects
      4. Reverse engineering
         1. Architecture extraction
         2. Design recovery
      5. Program analysis
         1. Clone detection and analysis
      6. Program Verification/Formal methods
   2. Normalization
      1. Reduces a program to a program in a sub-language to decrease its syntactic complexity
         1. Pascal to core Pascal
   3. Optimization
      1. Improve the run-time and/or space performance of a program
         1. Code motion optimization
   4. Refactoring or restructuring
      1. Changes the structure of the program to make it easier to understand
         1. Goto 🡪 while
   5. Renovation
      1. Error pair and change requirements
4. TXL Program Components
   1. Each TXL program has 2 components
      1. Grammar: description of the structure/input language to be transformed specified as Extended-BNF grammar, in context-free ambiguous form
      2. Rules: a set of transformation rules specified by example, using pattern/replacement pairs
   2. TXL Processor
      1. Grammars and rules are specified in the TXL language
      2. The TXL processor efficiently implements the TXL language
   3. How TXL works
      1. The TXL paradigm consists of parsing the input text into a structure tree, transforming the tree to create a new structure tree, and unsparing the new tree to a new output text
5. Writing TXL programs
   1. 2 main parts
   2. need to do
      1. define the grammar of the input/output language/structure 1
      2. write necessary transformation rules 2
6. Defining Input Grammar
   1. E.g. (20+6)\*10=260
   2. Should write:
      1. Define program [Expr] end define
      2. Define Expr [number] | [Expr]’ + [Expr] | [Expr] ‘\* [Expr] | ’([Expr] ‘) end define
   3. Each TXL program defines its own symbols and type system
   4. Nonterminal symbols or types (or syntactic forms) specify how sequences of input symbols are grouped into the structures of the input languages
   5. These forms the structured types of the TXL program
   6. These are specified using an(almost) unrestricted ambiguous context free grammar in extended BNF notation
7. Input Parsing
   1. Input is automatically tokenized and parsed (structured) according to the grammar
   2. The entire input must be recognizable as the type [program]
   3. The result is represented internally as a parse tree
   4. Parse tree represents the structural understanding of the input according to the grammar
   5. All pattern matching and transformation operations work on the parse tree
8. Transformation Rules
   1. The actual input to output source transformation is specified using a rooted set of transformation rules
   2. Each transformation rule specifies a target type to be transformed, a pattern and a replacement
      1. Pattern: an example of particular instance of the type that we are interested in replacing
      2. Replacement: an example of the result we want when we find such an instance
   3. Resolve bracketed expressions: (N) 🡪 N
      1. Rule resoveBracketedExpr
         1. Replace [Expr]
            1. ‘N[number]’)
         2. by
            1. N
         3. End rule
   4. Resolve additions: N1+N2 🡪 N3
      1. Rule resolveAdditions
         1. Replace [Expr]
            1. N1[number] + N2[number]
         2. By
            1. N1[+N2]
         3. End rule
   5. Resolve multiplications: N1 \* N2 🡪 N3
      1. Rule resoveMultiplications
         1. Replace [Expr]
            1. N1[number] \* N2[number]
         2. By
            1. N1 [\*N2]
         3. End rule
   6. The Main Rule
      1. Rule main
         1. Replace [Expr]
            1. E[Expr]
         2. Construct NewE [Expr]
            1. E[resolveAdditions]
            2. [resolveMutiplications]
            3. [resolveBracketedExpr]
         3. where not
            1. NewE [=E]
         4. By
            1. newE
      2. end rule
9. Authoring TXL rules
   1. Think by example, not by parse tree
   2. Begin each rule with an explicit example pattern and replacement, and generalize from there
   3. Example
      1. Start with an explicit concrete example case
      2. Generalize by introducing pattern variables
      3. Specialize by identifying, testing and generalizing special cases in the same way
      4. Integrate by abstracting and prioritizing cases
10. Software Clone or Code Clone?
    1. A code fragment which has identical or similar code fragments in source code
       1. Type 1: Changes in layout and formatting
          1. Changes in whitespace
          2. Changes in comments
          3. Changes in formatting
       2. Type 2: Renaming Identifiers and Literal Values
          1. Renaming of identifiers
          2. Renaming of Literal and types
       3. Type 3: Statements added/deleted/modified in copied fragments
          1. Modification of lines
          2. Addition of new of lines
          3. Deletions of lines
       4. Type 4: Statement recording/ control replacements
          1. Reordering of statements
          2. Control replacements
11. Why Clones in Software
    1. Copy and paste is a common practice in software development
       1. Existing code is often used as templating
    2. Time limit assigned to programmers
    3. Risk in developing new code
    4. Language limitations
12. Do Clones Matter?
    1. It is harmful
       1. Lead to server unexpected behaviour
       2. Number 1 bad smell in the community
    2. Increase cognitive effort
    3. Code bloat
13. Application of Code Similarity Detection
    1. Software Maintenance and Evolution
    2. Software reuse process
    3. Program comprehension
    4. Detecting Library Candidates
    5. …
14. TXL Clone Detection
    1. Code Clone – Duplicated/similar fragments of code in software system
15. C language Grammar
    1. Keywords
       1. Do, if, static, while, for, int, float …
    2. Operators
       1. &&, ||, \*=, /=, etc…
    3. Commenting sytles
       1. //
       2. /\*\* … \*\*/
    4. Statements
       1. If statement
       2. For loop
       3. Assignment statement
16. C Grammar in TXL
    1. Key statement specifies that certain identifiers are to be treated as unique special symbols
       1. Keys
          1. Do if static while for int float double long
       2. End keys
    2. The compounds statement specifies character sequences to be treated as a single character
       1. Compounds
          1. && || \*= /=
       2. end compounds
    3. The comments statement specifies the commenting conventions of the input language
       1. Comments
          1. /\* \*/
          2. //
       2. end comments
    4. The statement defines the all different kinds of statements
       1. Define statements
          1. [if\_statement]
          2. | [for\_head]
       2. end define
    5. The if-statement as TXL grammars
       1. Define if\_statement
          1. If([expr]){
             1. [statements]
             2. }
          2. [opt else\_statement]
          3. end define
       2. define else\_statement
          1. else {
             1. [statements]
             2. }
          2. end define
    6. the for-head as TXL Grammar
       1. define for\_head
          1. for([opt expr]; [opt expr]; [opt expr])
       2. end define
17. Standard Pretty-Printing
    1. Standard pretty-printing for if-statement
       1. Redefine if\_statement
          1. If( [expr] ){ [IN][NL]
             1. [statement] [EX]
             2. }
          2. [opt else\_statement]
       2. end redefine
       3. redefine else\_statement
          1. else{ [IN][NL]
             1. [statement] [EX]
             2. }
       4. end redefine
    2. [NL] new line is unparsed output
    3. [IN] indent unparsed output by four space
    4. [EX] extent unparsed output by four space
18. Context-sensitive Normalization
    1. We use TXL rules for context-sensitive structural
    2. Language and possibly project specific
       1. Rule ifElseNormalization
          1. Replace $ [statement]
             1. If( Expr1 [expr])

ThenPart [statement]

OptElsePart [opt else\_statement]

* + - 1. By
         1. If(‘ anyIfControl)

ThenPart

OptElsePart

* + - 1. End rule

1. Flexible code filtering
   1. Flexible code filtering allows us to elide unimportant parts for comparison
      1. Optionally on a contextual basis
         1. Rule declarationFiltering
            1. Relace [declaration\_or\_statement\*]

DeclarationPart [declaration]

RestOfScope [declaration\_or\_statement\*]

* + - * 1. By

RestOfScope

* + - * 1. End rule

1. Flexible Pretty-printing
   1. Input Grammar:
      1. Define for\_head
         1. For([opt expr]; [opt expr]; [opt expr])
      2. End efine
   2. Redefined Grammar
      1. Redefine for\_head
         1. For( [NL]
            1. [opt expr]; [NL]
            2. [opt expr]; [NL]
            3. [opt expr]) [NL]
         2. end redefine
2. TXL in Academia
   1. DRACULARE: a system for recognizing and understanding handwritten mathematics
   2. Analysis and understanding of Italian speech
   3. Re-desgin of legacy application
   4. Formal testing methods
   5. Database integration
   6. TXL is used in research project on every continent in the world, except one
3. Conclusion
   1. TXL is an executable specification language and rapid prototyping syste for soruce transformation task of all kinds
   2. Most automatable software engineering tasks can be modeled as soruce transformations
   3. Rapid prototypes of these tasks can be quickly developed using TXL and later used as design specifications for a production implementation
   4. TXL is a pure functional language that demonstrates the power, simplicity and practicality of the functional programming paradigm