Excerpts from The TXL Cookbook, Part I

TXL Basics

James R. Cordy

School of Computing

Queen's University at Kingston, Canada

Agenda

- In this tutorial we will be exploring a set of *excerpts* from the **TXL** Cookbook
    - Some representative problems and solutions in program processing and analysis using **TXL**
- The tutorial will proceed in *three parts*:
    - A *basic introduction* to **TXL** (for those new to it)
    - *Parsing and restructuring* problems and recipes for **TXL** (foundations for many solutions)
    - *Transformation and analysis* problems and recipes for **TXL** (selections from the **TXL** Cookbook)

goal: A basic understanding of using **TXL** effectively in restructuring, analysis and transformation tasks

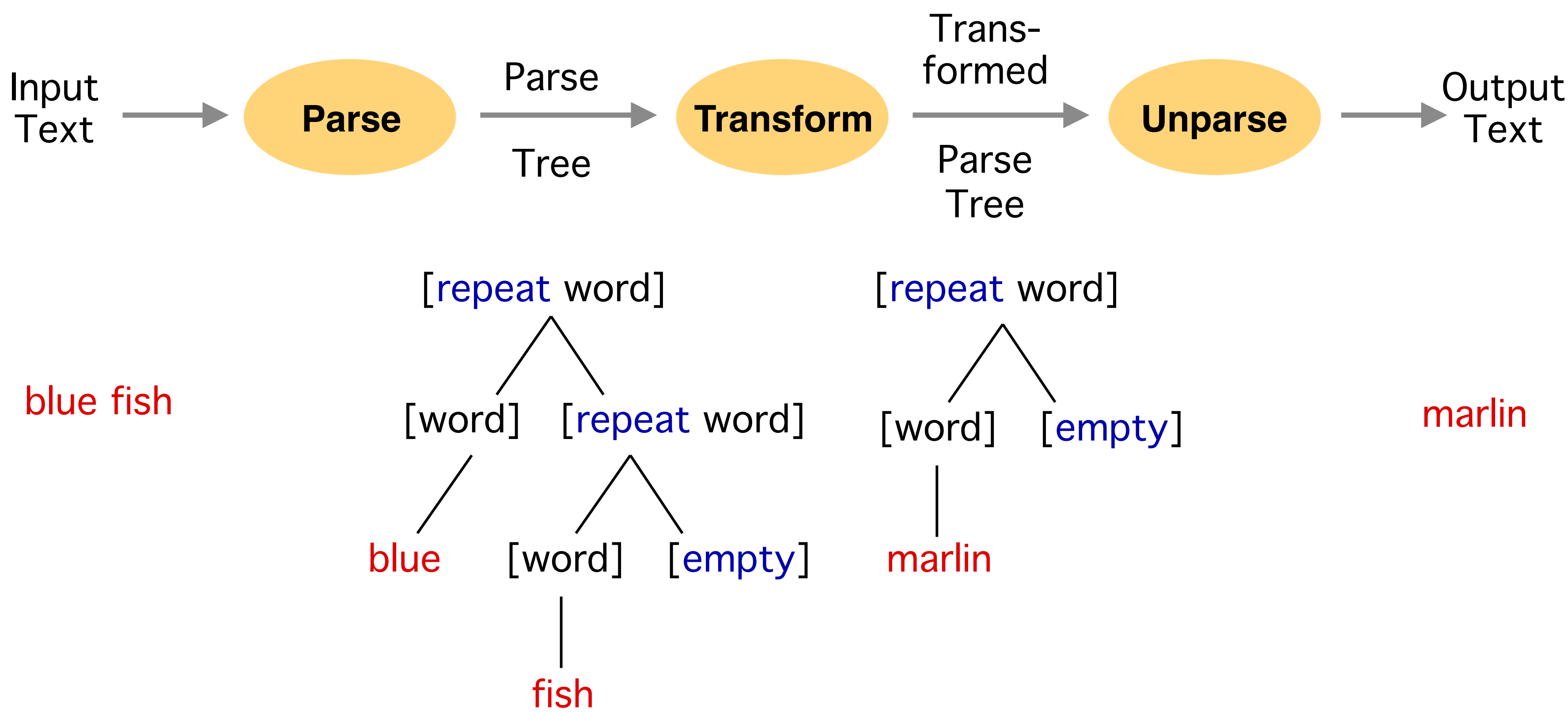
The TXL Programming Language

- *Original purpose* (1983, the golden age of PL's):
  DSL for experiments in language notations, *dialects* and *extensions*
  - Variants, *DSLs* of *Turing*
  - *OOT* (OO variant), and *NT* (numerical computation variant) of *Turing* originally rapid prototyped using **TXL**

- *Actual uses* (1990-present, the dark age of PLs):
  Source analysis, software renovation, system migration,
  generative programming, security analysis, clone detection, MDE
  - Code generation from models (1992)
  - Design recovery (1994)
  - Financial systems, *Y2K* (1997)
  - Airline mergers (2000)
  - Secuirty analysis and risk prevention (2003-)
  - UML model extraction and transformation (2005-)
  - Clone detection and resolution (NICAD, Simone) (2007-)
  - Over 250 *companies* and *universities* using in last 10 years
  - Over 100 refereed papers on uses in *last 5 years*

The TXL Paradigm

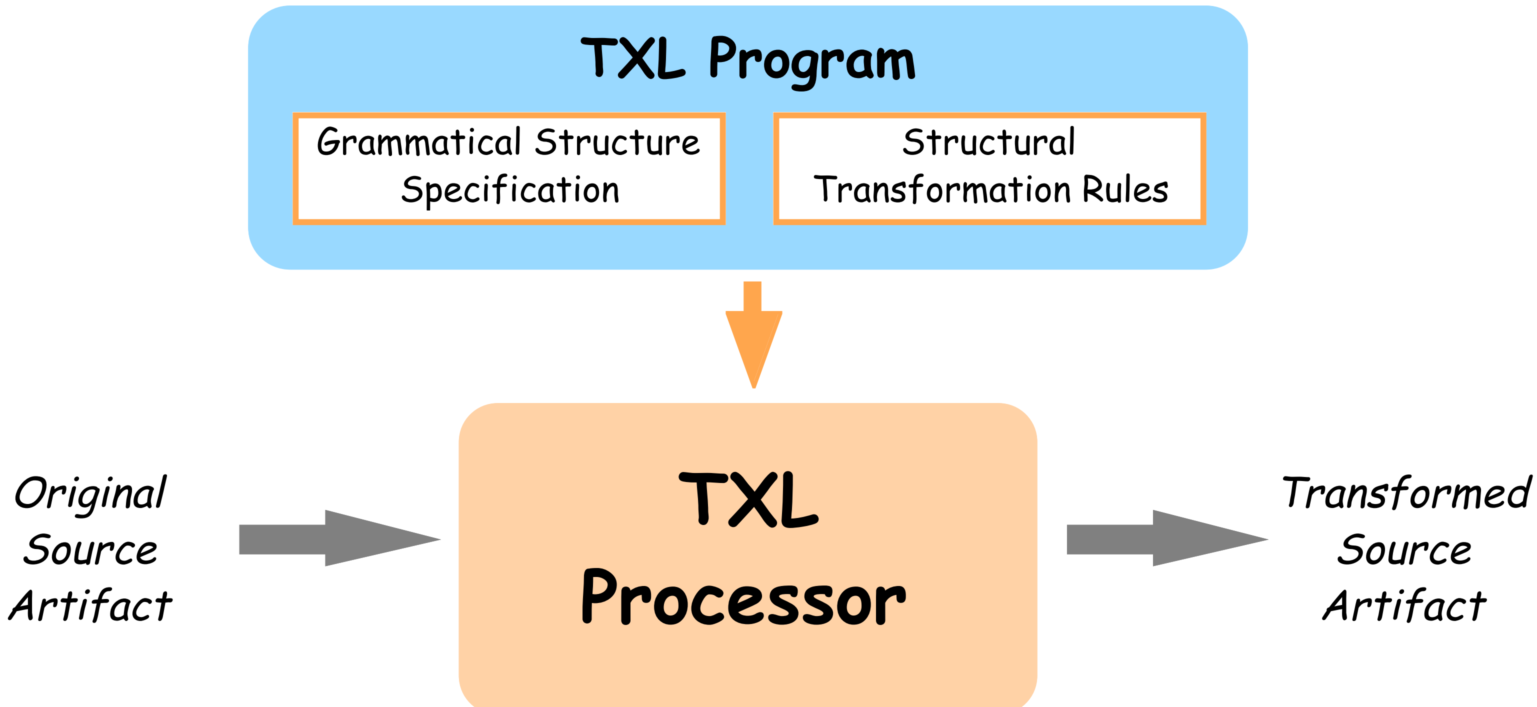
- the TXL paradigm consists of *parsing* the input text into a structure tree, *transforming* the tree to create a new structure tree, and *unparsing* the new tree to a new output text



tip: You can think of TXL in this way at every level

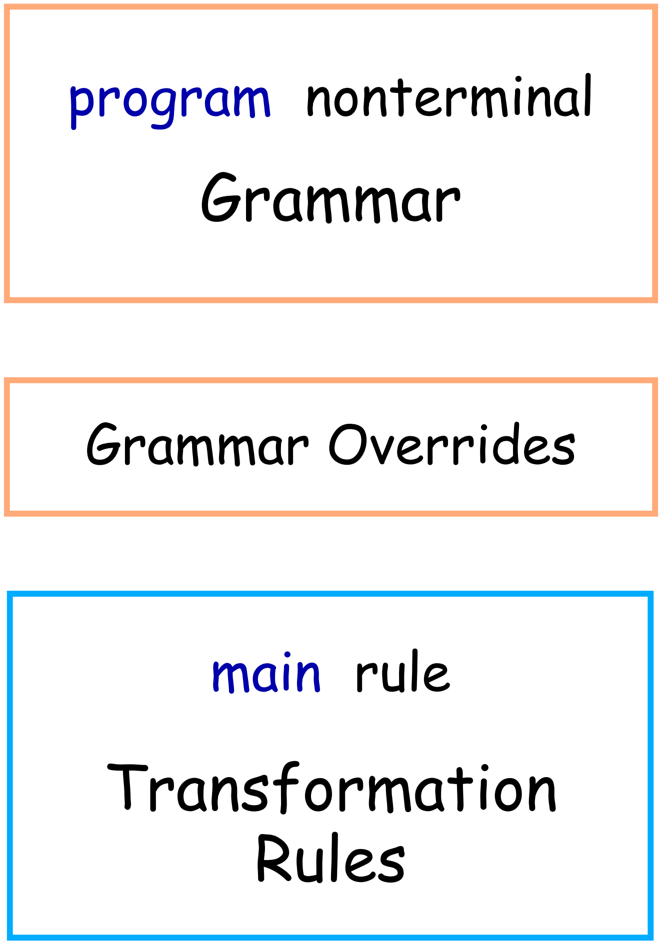
The TXL Processor

- Grammars and transformation rules are specified in the TXL *language*
- the TXL *processor* efficiently implements the TXL language



tip: One transformation at a time - think cascaded sequence

Anatomy of a TXL Program



- The *base grammar* defines the lexical forms *(tokens)* and the rooted set of syntactic forms *(nonterminals* or *types)* of the input language - usually an *include* statement
- The optional grammar *overrides* extend or modify types of the grammar to allow output and intermediate forms of the transformation
- The *ruleset* defines the rooted set of transformation *rules* and *functions*

tip: Keep grammar and ruleset in topological order to aid readability

The Grammar: Lexical Forms

- The *tokens* statement gives regular expressions for each class of token in the input language

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

- Predefined defaults include C-style identifiers *[id]*, integer and float numbers *[number]*, string literals *[stringlit]*, *[charlit]*

tip: The predefined defaults are often sufficient for a first version

The Grammar: Lexical Forms (cont'd)

- The *comments* statement specifies the commenting conventions of the input language

```
comments
    /*   */
    //
end comments
```

- By default, comments are *ignored* (treated as white space) by **TXL**, but they can be treated as significant symbols if desired

tip: Most tasks can ignore comments

The Grammar: Lexical Forms (cont'd)

- The *keys* statement specifies that certain identifiers are to be treated as unique special symbols (and not as identifiers)

```
% keywords of Pascal
keys
    program procedure function
    repeat until for while do begin 'end
end keys
```

- The *compounds* statement specifies character sequences to be treated as a single character

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

(Really just a shorthand for an unnamed token definition)

tip: TXL comments start with % to end of line

The Grammar: Syntactic Forms

- Syntactic forms (*nonterminal symbols* or *types*) specify how sequences of input symbols are grouped into the structures of the input language

- Specified using an (almost) unrestricted ambiguous *context free grammar* in extended BNF notation, where

`X`	*terminal* symbols represent themselves (optional `'X`)
`[X]`	*nonterminal* types appear in brackets
|	*or bar* separates alternative syntactic forms

tip: Each TXL program defines its own symbols and type system

The Grammar: Syntactic Forms (cont'd)

- Each nonterminal type is specified using a *define* statement
- The special type *[program]* describes the structure of the entire input

```
define program                        % goal symbol of input
    [expression]
end define

define expression
    [term]
  | [expression] + [term]
  | [expression] – [term]
end define

define term
    [primary]
  | [term] * [primary]
  | [term] / [primary]
end define

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

tip: Grammars are most efficient and natural when most user-oriented
- avoid Yacc-style "implementation" grammars

The Grammar: Syntactic Forms (cont'd)

- Extended BNF-like sequence notation

```
[repeat X]   or   [X*]       % sequence of zero or more (X*)
[repeat X+]  or   [X+]       % sequence of one or more (X+)
[list X]     or   [X,]       % comma-separated list zero or more
[list X+]    or   [X,+]      % comma-separated list one or more
[opt X]      or   [X?]       % optional (zero or one)
```

tip: For more natural patterns, always use repeat and list for sequences

tip: Use less restrictive grammars rather than syntax checkers

The Grammar: Syntactic Forms (cont'd)

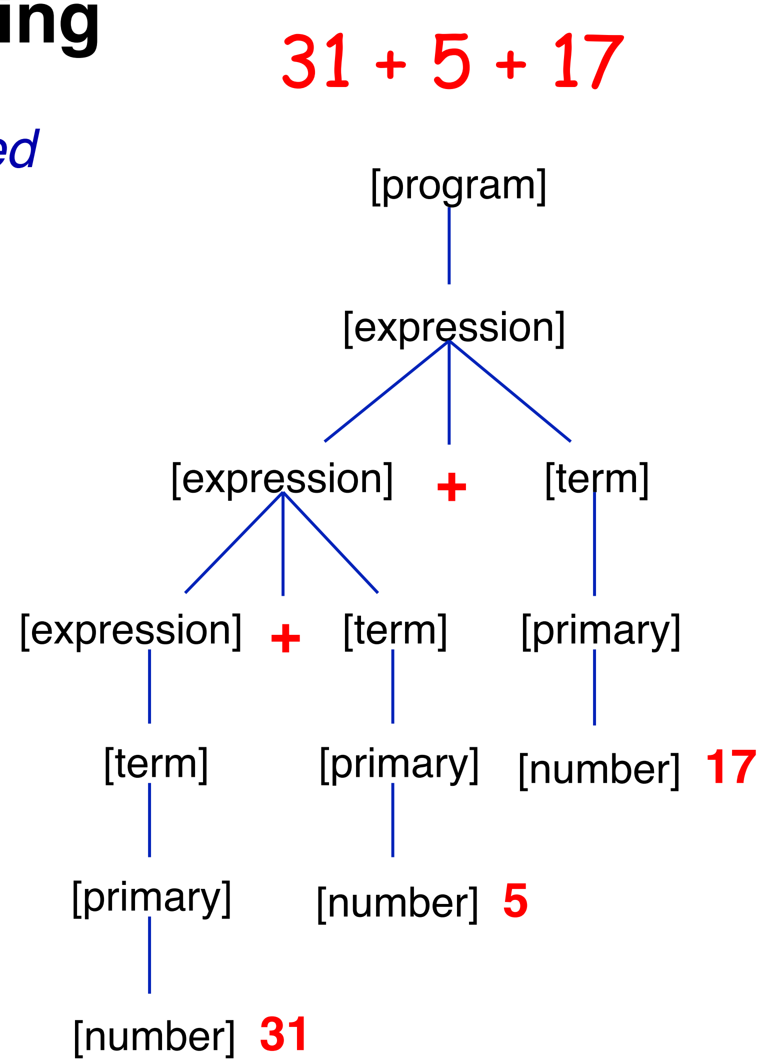
- Formatting cues in defines specify how to format output

`[NL]`	*newline* in unparsed output
`[IN]`	*indent* unparsed output by four spaces
`[EX]`	*exdent* unparsed output by four spaces

tip: Formatting cues have no effect on input parsing

Input Parsing

- Input is automatically *tokenized* and *parsed* according to the grammar
- The entire input must be recognizable as the type *[program]*
- The result is represented internally as a *parse tree*
- All pattern matching and transformation operations work on the parse tree



tip: Syntax errors may indicate an incorrect grammar rather than malformed input

Base Grammars and Overrides

- The *base grammar* for the syntax of the input language is normally kept in a separate grammar file which is rarely if ever changed, and is *included* in the **TXL** program
- Dialects and extra output forms are added to the base grammar using *grammar overrides*, which modify or extend the base grammar's lexical and syntactic forms

```
% The original example grammar
include "Expr.grm"

% Override to allow identifiers and lists of expressions
redefine primary
      [id]
   |  [number]
   |  ( [list expression+] )
end redefine
```

tip: The crafting of grammars is the most critical step in the success of a TXL project!

Base Grammars and Overrides (cont'd)

- Grammar overrides can also be used to *extend* the existing forms of a nonterminal type
  - Using "`...`" to refer to the original definition

```
% The C language grammar
include "C.grm"

% Override to allow statements to have XML markup
redefine statement
    ...
  | <[id]> [statement] </[id]>
end redefine
```

tip: Grammar extensions can be independent of most changes to the base grammar

Transformation Rules

- The actual input to output source transformation is specified using a rooted set of *transformation rules*

- Each transformation rule specifies:
    - A *target type* to be transformed
    - A *pattern* (example of the instances that we want to replace)
    - A *replacement* (example of the result we want when we find one)

```
% replace every 1+1 expression by 2

rule addOnePlusOne
   replace [expression]   % target type to search for
      1 + 1               % pattern to match
   by
      2                   % replacement to make
end rule
```

tip: TXL rules are strongly typed - the replacement must be of the same type as the pattern

Transformation Rules (cont'd)

- The pattern can be thought of as an actual source text *example* of the instances we want to replace
- Patterns consist of *tokens* (terminal symbols which represent themselves) and named *variables* (nonterminal types which match any instance of the type)

```
rule optimizeAddZero
   replace [expression]
      N1 [number] + 0
   by
      N1
end rule
```

- When the pattern is matched, variable names are *bound* to the corresponding instances of their types in the match
- Variables can be used in the replacement to *copy* their bound instance into the result

tip: Think by example, not by parse tree

Transformation Rules (cont'd)

- Similarly, the replacement is a source text *example* of the desired result
- Replacements consist of *tokens* and *references* to bound variables, whose bound instance is *copied* into the result
- References to variables can be optionally transformed by *subrules* (other transformation rules), which transform (only) the copy of the variable's bound instance before it is copied into the result
- Subrules are applied to a variable reference using square bracket notation ***X[f]***, which in function notation would be ***f(X)***

```
rule resolveAdditions
   replace [expression]
      N1 [number] + N2 [number]
   by
      N1 [+ N2]      % [+] is one of TXL's built-in functions
end rule
```

tip: X[f][g] denotes functional composition - g(f(X))

Transformation Rules (cont'd)

- When a rule is applied to a variable, we say that the variable's copied value is the rule's *scope*
- A rule application only transforms *inside* the scope it is applied to
- The distinguished rule called *main* is automatically applied to the entire input as its scope
    - any other rules must be *explicitly applied* as subrules

```
function main
   replace [program]
      EntireInput [program]
   by
      EntireInput [resolveAdditions]
                  [resolveSubtractions]
                  [resolveMultiplys]
                  [resolveDivisions]
end function
```

tip: Often the main rule is a simple function to apply other rules

Rules and Functions

- **TXL** has two kinds of transformation rules, *rules* and *functions*, which are distinguished by whether they should transform only *one* (for functions) or *many* (for rules) occurrences of their pattern
- By default, *rules* repeatedly *search* their scope for the first instance of their target type matching their pattern, transform it in place to yield a *new scope*, and then reapply to the entire new scope until no more matches are found
- By default, *functions* do not search, but attempt to match only their *entire scope* to their pattern, transforming it if it matches

```
function resolveEntireAdditionExpression
   replace [expression]
      N1 [number] + N2 [number]
   by
      N1 [+ N2]
end function
```

tip: Use functions to apply several rules to a single scope

Rules and Functions (cont'd)

- *Searching* functions, denoted by *replace ** , search to find the first occurrence of their pattern in their scope but do not repeat

```
function resolveFirstAdditionExpression
   replace * [expression]
      N1 [number] + N2 [number]
   by
      N1 [+ N2]
end function
```

tip: Use searching functions when only one match is expected

Rules and Functions (cont'd)

- Subrules and functions may be passed *parameters*, which bind the values of variables in the applying rule to the formal parameters of the subrule

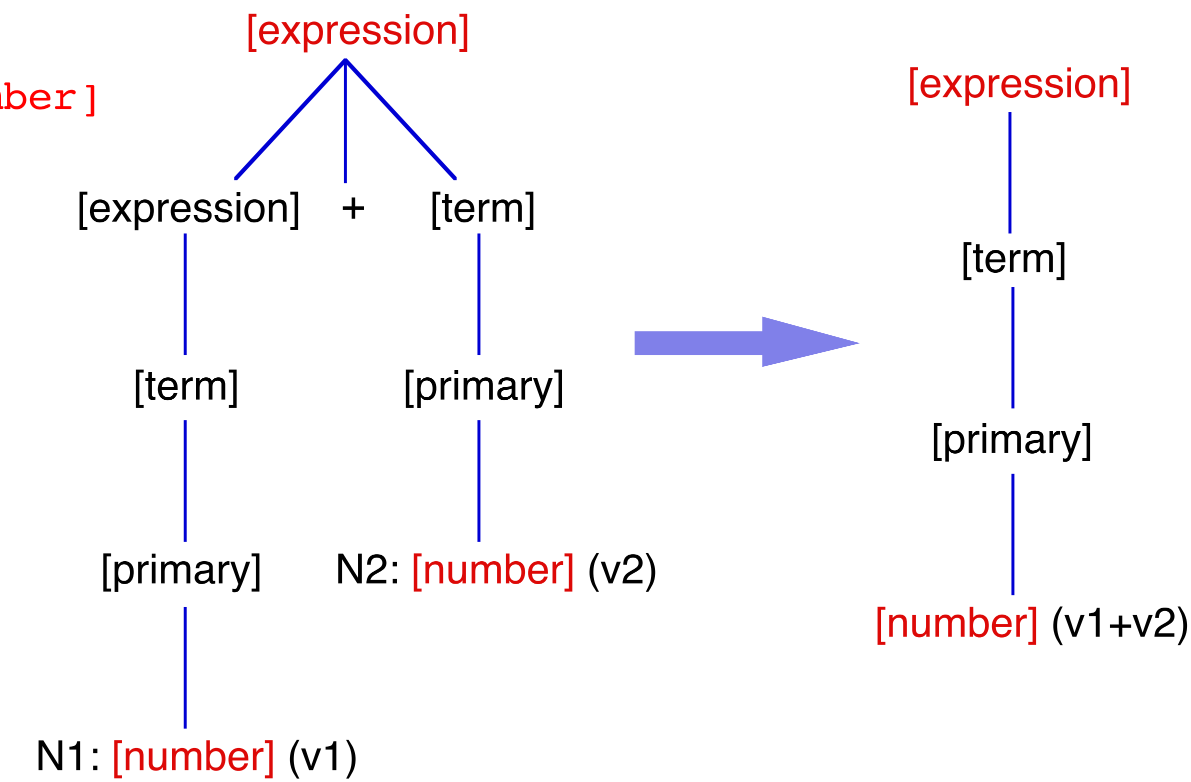
```
rule resolveConstants
   replace [repeat statement]
      const C [id] = V [expression];
      RestOfScope [repeat statement]
   by
      RestOfScope [replaceByValue C V]
end rule

rule replaceByValue ConstName [id] Value [expression]
   replace [primary]
      ConstName
   by
      ( Value )
end rule
```

tip: Use parameters to build transformed results from many parts

Patterns and Replacements

- Patterns and replacements are parsed in the same way as the input, to make *pattern tree* => *replacement tree* pairs

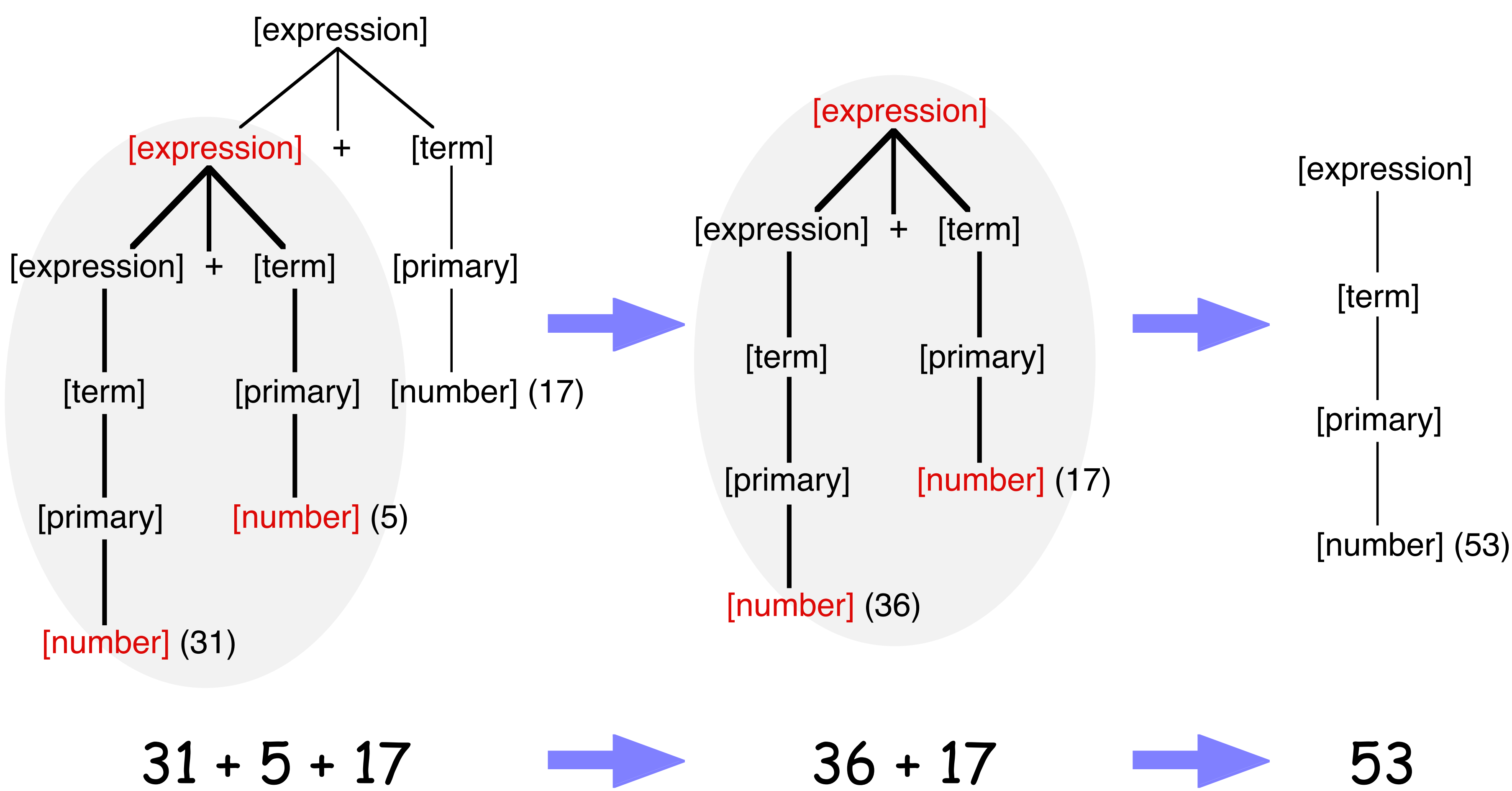
```
rule resolveAdditions
   replace [expression]
      N1[number] + N2[number]
   by
      N1 [+ N2]
end rule
```



tip: But think by example when authoring rules, not about the trees!

Patterns and Replacements (cont'd)

- Rules are implemented by searching the *scope parse tree* for tree pattern matches of the *pattern tree* , and replacing instances with corresponding instantiations of the *replacement tree*



Patterns and Replacements (cont'd)

- Patterns may use previously bound variables later in the pattern (*strong pattern matching*)
- This effectively *parameterizes* the pattern with a copy of the bound variable, to specify that two parts of the matching instance must be the *same* to have a match

```
rule optimizeDoubles
   replace [expression]
      E [term] + E
   by
      2 * E
end rule
```

- Patterns can also be parameterized by formal *parameters* of the rule, or other bound variables, to specify that matching instances must contain an *identical copy* of the variable's bound value at that point in the pattern

tip: References to a variable always mean a copy of its bound value, no matter what the context

Deconstructors and Constructors

- Patterns may be piecewise *refined* to more specific patterns using *deconstruct* clauses

```
rule optimizeFalseIfs
   replace [statement*]
      IfStatement [if_statement] ;
      RestOfStatements [statement*]
   deconstruct * [if_condition] IfStatement
      IfCond [if_condition]
   deconstruct IfCond
      false
   by
      RestOfStatements
end rule
```

- Deconstructors specify that the deconstructed variable's bound value must match the given pattern - if not, the *entire* pattern match fails
- Deconstructors act like functions - by default, the entire bound value must match the deconstructor's pattern, but *deconstruct ** (a *deep* deconstruct) searches within the bound value for a match

Deconstructors and Constructors (cont'd)

- Pattern matches can also be constrained using *where* clauses
    - Allows arbitrary matching conditions tested by subrules

```
rule vectorizeScalarAssignments
   replace [statement*]
       V1 [variable] := E1 [expression];
       V2 [variable] := E2 [expression];
       RestOfScope [statement*]
   where not
       E2 [references V1]
   where not
       E1 [references V2]
   by
       < V1,V2 > := < E1,E2 > ;
       RestOfScope
end rule
```

tip: It's always better to use a deconstruct than a where clause

Deconstructors and Constructors (cont'd)

- Where clauses use a special kind of rule called a *condition* rule
- Condition rules have only a (possibly very complex) pattern, but no replacement - they simply *succeed* or *fail*

```
function references V [variable]
   deconstruct * [id] V
       Vid [id]
   match * [id]
       Vid
end function
```

Deconstructors and Constructors (cont'd)

- Replacements can also be piecewise refined to *construct* results from several independent pieces

```
rule addToSortedSequence NewNum [number]
    replace [number*]
        OldSortedSequence [number*]
    construct NewUnsortedSequence [number*]
        NewNum OldSortedSequence
    by
        NewUnsortedSequence [sortFirstIntoPlace]
end rule
```

- Constructors allow partial results to be bound to *new variables*, allowing subrules to further transform them

tip: In complex rules, liberal use of constructs aids readability

Authoring TXL Programs

- TXL is primarily intended as a rapid prototyping platform, and is ideally suited to *extreme programming*
- Begin with an explicit set of *test cases*, and treat these as the *specification* of your transformation
- Program your transformation *incrementally*, as a sequence of *successive approximations* to the final result
- Actually *run* your partial transforms against the test cases to keep track of your progress and *test as you go*
- Always write the *simplest possible* transformation rules to achieve the result - don't worry about efficiency until you are done
- Begin each rule with an *explicit example* pattern and replacement, and generalize from there

tip: TXL programs tune incredibly well - factors of 10 to 100 are common

Authoring TXL Programs (example)

- Step 1 - Start with an explicit *concrete example case*

```
rule convertAddIJK
   replace [statement]
      ADD I TO J GIVING K               % COBOL
   by
      K = I + J;                        % PL/I
end rule
```

- Step 2 - *Generalize* by introducing pattern variables

```
rule convertAddGiving
   replace [statement]
      ADD I [operand] TO J [operand] GIVING K [operand]
   by
      K = I + J;
end rule
```

tip: Test at every stage!

Authoring TXL Programs (example)

- Step 3 - *Specialize* by identifying, testing and generalizing *special cases* in the same way

```
rule convertAddNoGiving
   replace [statement]
      ADD I [operand] TO J [operand]
   by
      J = J + I;
end rule
```

- Step 4 - *Integrate* by abstracting and prioritizing cases

```
rule convertAdds
   replace [statement]
      AddStatement [COBOL_add_statement]
   by
      AddStatement [convertAdd1]
                   [convertAddNoGiving]
                   [convertAddGiving]
                   [checkAddConverted]
end rule
```

Authoring TXL Programs (example)

- Step 5 - *Constrain* to semantically precise conditions
  (get the details right!)

```
rule convertAddBinaryOnly
   replace [statement]
      ADD I [identifier] TO J [identifier]
   where
      I [FB_hasFactWithAttribute 'FieldSize 'COMP]
   where
      J [FB_hasFactWithAttribute 'FieldSize 'COMP]
   by
      J = J + I;
end rule
```

Understanding TXL

- **TXL** is not really a source transformation system
  - It is a *language for authoring* source transformation systems

- A **TXL** *"grammar"* is not really a grammar
  - **TXL** has no grammar analyzer or parser generator
  - The grammar is a *functional program* for parsing the input
  - *Direct control* over parse - flexibility vs automation

- A **TXL** transformation *"rule set"* is not really a term rewriting system
  - No globally applied rules, no traversals or strategies
  - The rules are a *functional program* for transforming the input
  - *Direct control* over traversal and strategy - flexibility vs automation

tip: Nothing is hidden in TXL - no magic

Understanding TXL (cont'd)

- TXL programs are completely *self-contained*
  - No dependence on external parsers, frameworks, tools, libraries, other languages or notations
  - Everything is in the TXL program source
- TXL programs are *interpreted directly*
  - No compile step, just run directly from source
  - No portability issues
- TXL *processor* also has no dependencies
  - Install and go, requires nothing else

tip: Install TXL and run

That's It!

- Basically, that's **TXL**
    - Everything else is in how you use it – the *recipes*
- Next:
    - **TXL** practice lab #1– get started,
      try a simple transformation
- Then:
    - Part II: Some Recipes for Parsing and Language
      Manipulation Problems using **TXL**