4GL Code Transformation

Modernize 4GL Code Using Fully Automated Transformation

Greg Shah
Golden Code Development

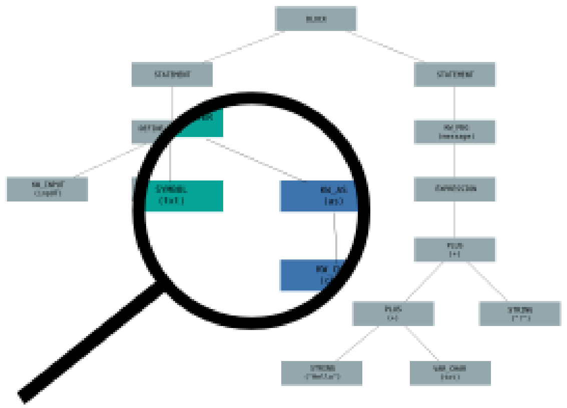
Tuesday October 8, 2019

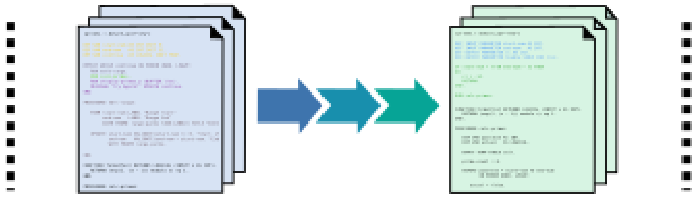
Agenda

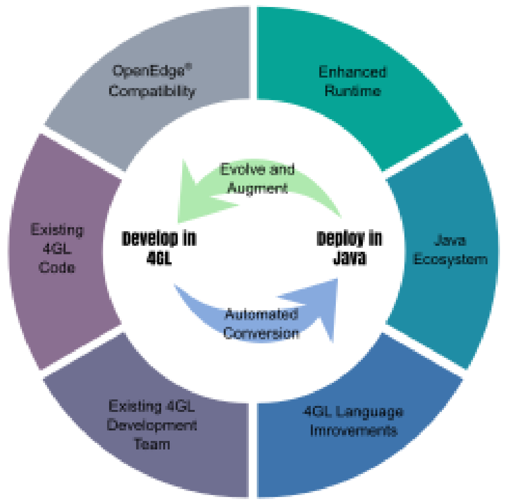
- Background
- Handling Flexibility and Scale
- Transformation Process
- TRPL Primer
- Usage Tips
- How to Get Started
- Planned Improvements

Background

FWD is an **open source** toolset for **modernizing** 4GL applications.







Explore

Tools for reporting, statistics and analysis, which expose the inner workings of applications.

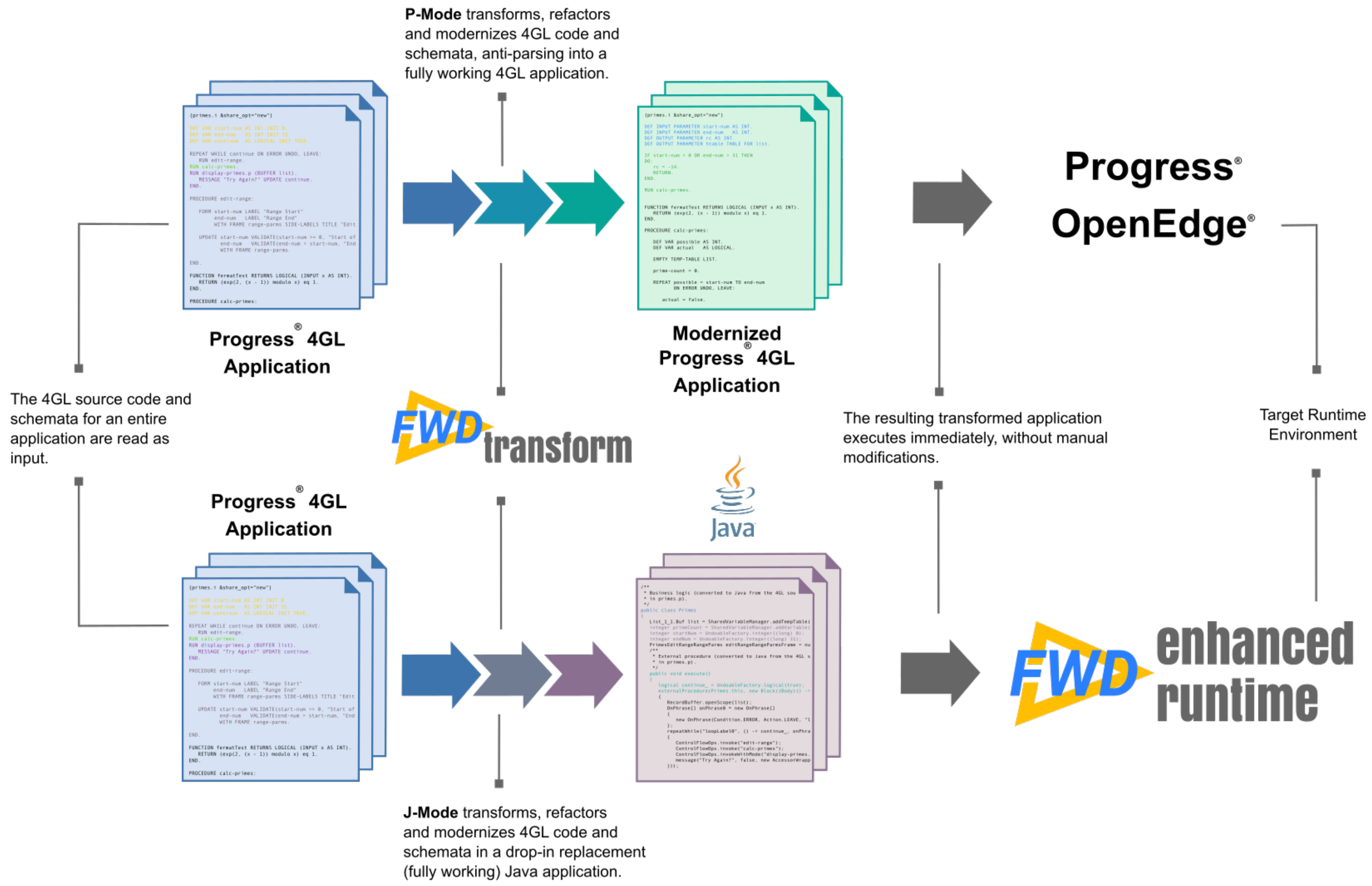
Transform

Fully automated transformation and modernization of entire applications without a manual rewrite.

Transcend

Compatible replacement for OpenEdge® which leverages an enhanced runtime, an upgraded 4GL language and Java to modernize and evolve applications.

Fully automated transformation and **modernization of entire applications without a manual rewrite.**


P-Mode transforms, refactors and modernizes 4GL code and schemata, anti-parsing into a fully working 4GL application.
Progress® 4GL Application
Modernized Progress® 4GL Application
Progress® OpenEdge®
The 4GL source code and schemata for an entire application are read as input.
FWD transform
The resulting transformed application executes immediately, without manual modifications.
Target Runtime Environment
Progress® 4GL Application
Java
FWD enhanced runtime
J-Mode transforms, refactors and modernizes 4GL code and schemata in a drop-in replacement (fully working) Java application.

Why Use FWD Transform?

- Enables fully automated transformation of Progress® 4GL code. Transforms an entire application in one run.
- Powerful enough to handle the most extreme refactoring and transformation projects.
- Can be used to separate business logic from UI.
- Handles changes at scale, even on the largest of applications. Tested on applications well over 10 million lines of code.
- Proven technology used to convert entire applications into fully compatible Java versions.
- By far, this is the most capable transformation toolset for Progress® 4GL code.
- Open source.
- No manual rewrite! **Eliminates YEARS of wasted effort.**

FWD transform

Handling Flexibility and Scale

Your Source Is Not Helping

- Programmatic analysis of an application needs to be aware of the 4GL language syntax.
- Your source code is text. That text is non-regular and ambiguous.
  - different text, same meaning (non-regular code)
  - same text different meaning (ambiguous code)
- The 4GL suffers from this problem more than most languages.

Non-Regular Code

Non-Regular Code Feature	Description	Different Text, Same Meaning

Case Insensitivity	Source code may be entered with any case (all upper, all lower, mixed). This frees the programmer from having to think about or match specific case of keywords and symbols (including user defined symbols).	`Display`
		`DISPLAY`
		`display`
		`dIsplay`
		`disPlay`

Keyword Abbreviations	Some keywords (not all of them are documented) are allowed to be arbitrarily abbreviated to some minimum number of characters.	`display`
		`displa`
		`displ`
		`disp`

Database Symbol Abbreviations	4GL source code will include direct references to database table names and database field names. Both table and field names are user defined symbols. The references can be arbitrarily abbreviated to the smallest form of the symbol that does not overlap with any other field. What is actually allowed will depend on the context of where the name references occur. Changes in surrounding code can make some abbreviations possible that would not be valid in other places of the same program.	Given two tables cust and customer, all of the following refer to the customer table:
		`customer`
		`custome`
		`custom`
		`custo`
		
		Given two fields (not necessarily in the same table) name and number, all the following refer to number:
		`number`
		`numbe`
		`numb`
		`num`
		`nu`
		
		If cust and customer both have name and number, then all of these refer to
		`customer.number:`
		`customer.number`
		`custome.number`
		`custom.num`
		`custo.nu`
		`...`

Non-Regular Code

Non-Regular Code Feature	Description	Different Text, Same Meaning

Optional Database Name Qualifiers	Database tables and fields can appear in either a qualified or unqualified form. What is actually allowed will depend on the context of where the name references occur. Changes in surrounding code can make some unqualified forms possible that would not be valid in other places of the same program.	Table:
		`mydb.customer`
		`Customer`
		
		Field:
		`mydb.customer.number`
		`customer.number`
		`number`

Ordering of Options/Clauses	The ordering of keywords, options and clauses in most statements is very flexible. This was probably an unavoidable consequence of the keyword-heavy nature of the 4GL syntax. Since there is a low ratio of punctuation to keywords, there are relatively fewer natural cues in the language to differentiate the grammatical structure of “sentences” (statements). Without the flexible ordering, the syntax would be harder to remember.	`def var num format “999” init 14 extent 4.`
		
		`def var num init 14 extent 4 format “999”.`

Synonyms	Some punctuation and keywords can be used interchangeably. Often these are undocumented capabilities.	Most code blocks can have their header ended with either a . (period) or a : (colon).
		
		`function help returns int ().`
		`end.`
		`function help returns int ():`
		`end.`
		
		Keywords `WAIT` and `WAIT-FOR` are synonyms even though `WAIT-FOR` cannot be abbreviated (`WAIT-FO` is not valid):
		
		`WAIT go of my-widget.`
		`WAIT-FOR go of my-widget.`

Ambiguous Code

Ambiguous Code Feature	Description	Same Text, Different Meaning
Unreserved Keywords	With roughly 2000 keywords in the language, not all of them could be made into reserved keywords (keywords whose text cannot be a user-defined symbol). Over 400 keywords are reserved, leaving the other 1600 (approximately) as unreserved. The keywords can appear as source text for user-defined names.	A user defined function encrypt() can be named the same as the built-in function encrypt() that uses an unreserved keyword: `some-var = encrypt(some-data).`
Overloading Punctuation and Keywords	Some punctuation and keywords are used for multiple purposes, whose function is differentiated by the context of the program.	The . (period or dot) is used for many purposes: statement terminator, qualified database name separator, date separator, decimal point, package name separator, included in a unquoted filename. The : (colon) is similarly overloaded, including many cases where it can be substituted for the . (period). The ERROR keyword can be used as part of an ON ERROR clause, as a handle-based attribute, as a key function, as a type of alert-box, as a built-in function or as part of the RETURN statement.
Many Namespaces	The 4GL contains 19 different namespaces, some of which are “flat” and some of which are scoped to code blocks. The same user defined symbols can appear in any and all of these namespaces in the same program.	A variable can be named the same thing as a stream which can be the same name as a buffer, and so forth.

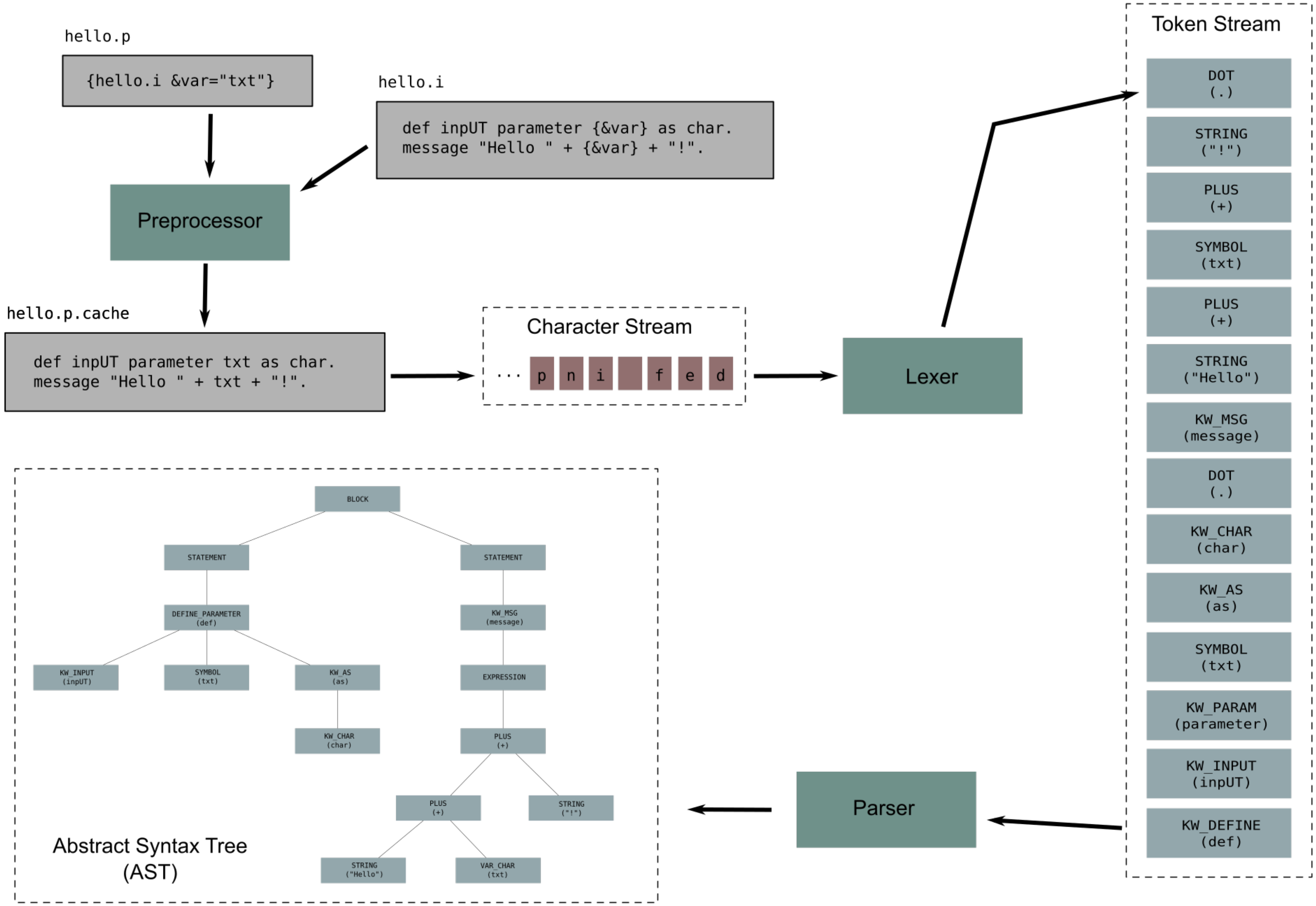
Syntax Flexibility Increases Cost

- The more flexible the language’s grammar, the larger the set of possible valid constructs that can be written by the programmer.
- This may lead to marginal time savings when creating new programs.
- That increased flexibility magnifies the long term cost of reading, maintenance, debugging, support and refactoring.
- **Greater flexibility in syntax results in greater long term cost over the life cycle of an application.**

Abstract Syntax Trees (ASTs)

- To enable proper analysis of code, we must transform the text into a data structure that represents the purest form of the code.
- ASTs represent the code's language syntax without syntactic sugar. The result is regular and unambiguous.
- This allows the **meaning of the code** (its semantics) to be separated from the messiness of the representation (the highly varying syntax).

File -> Char -> Token -> Tree



hello.p

```
{hello.i &var="txt"}
```

hello.i

```
def inpUT parameter {&var} as char.
message "Hello " + {&var} + "!".
```

hello.p.cache

```
def inpUT parameter txt as char.
message "Hello " + txt + "!".
```

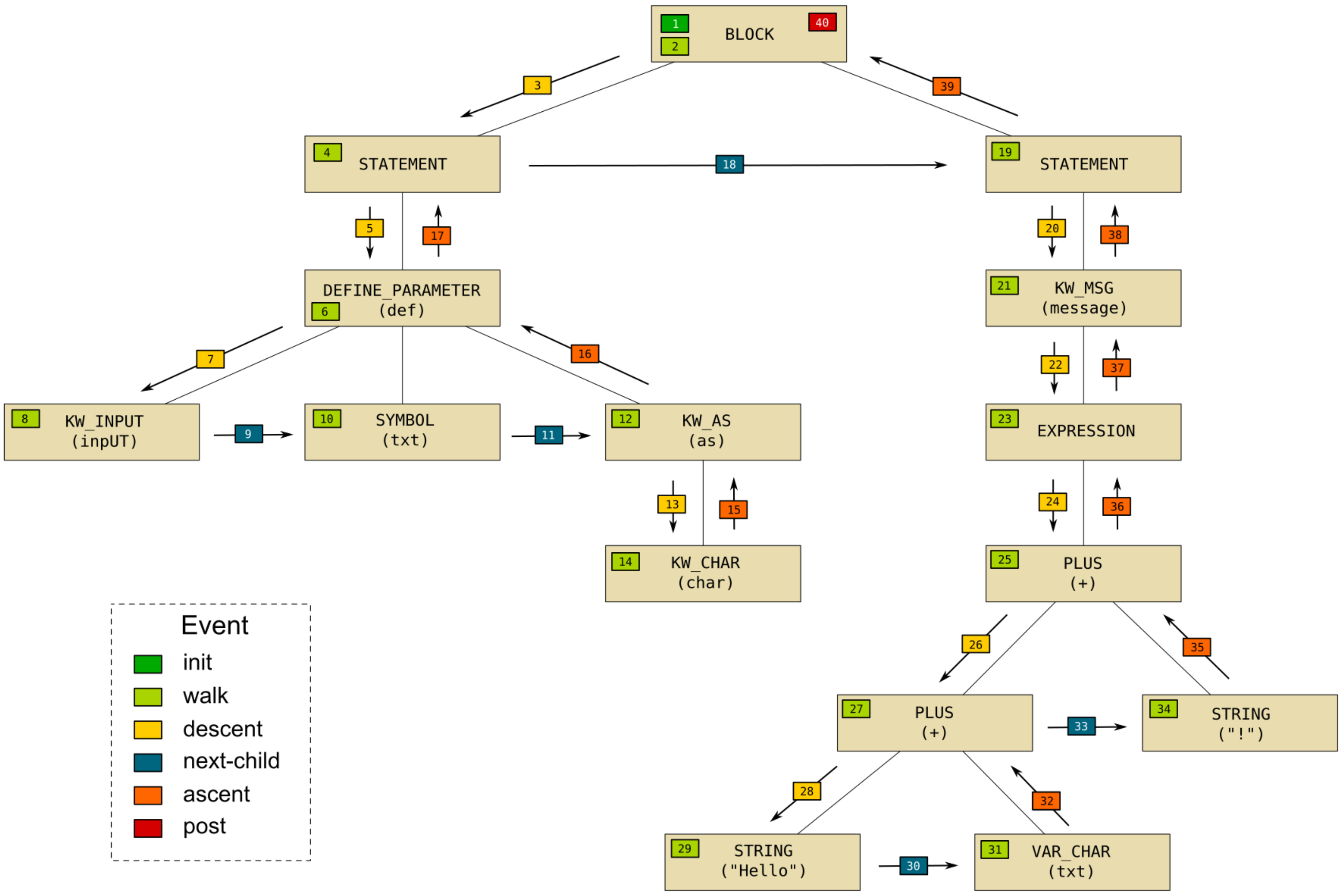
Scale Multiplies Issues

- Spread that syntax flexibility across tens of thousands of files and millions of lines of code.
- How do you find specific patterns?
- How do you even know if you have found all matches?
- Brute force is not enough.
- **Scale makes hard problems impossible (or at least impractical).**
- Automation is the only practical solution.

TRee Processing Language (TRPL)

- FWD provides tools to parse an entire application.
- Each source file and each schema file (.df) will be represented as an AST.
- TRPL is the analysis and transformation toolset in FWD which can operate on the entire set of ASTs as a batch.
- When you process trees, it is commonly called a tree walk.
- TRPL includes an engine that handles the tree walking for programs written in the TRPL language.

TRee Processing Language (TRPL) Event Model



Event	
🟩	init
🟩	walk
🟨	descent
🟦	next-child
🟧	ascent
🟥	post

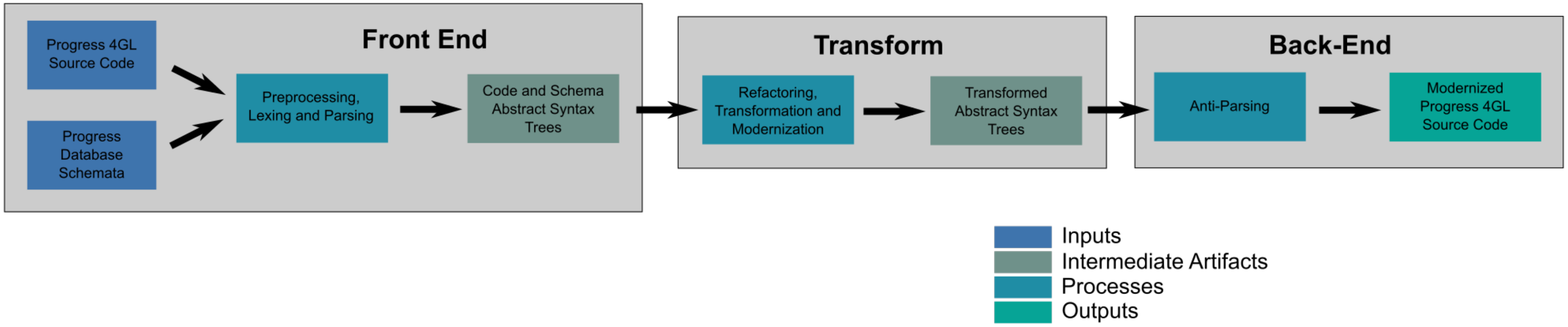
AST Designed for Transformation

- At parse time, there is a great deal of knowledge about the code. Encoding that knowledge into the tree makes downstream work easier.
- Resolving data types of each expression component is very important. This allows downstream code to calculate the type of each subexpression or expression in the application.
- By tracking resources by scope and creating linkages between the references and the definition, it becomes easier to work with these resources later.
- Structuring the tree is important. This can make it easier to walk the tree, match patterns and transform.
  - Multiple nodes can be rewritten as a single unambiguous node (e.g. KW_DEFINE KW_PARAMETER can be written as DEFINE_PARAMETER).
  - Artificial nodes can be inserted to group multiple related nodes.
- Calculated values and context-specific information are stored in the associated nodes as annotations.
- The ASTs created by FWD were designed with these issues (and others) in mind.

FWD transform

Transformation Process

Progress® 4GL to Progress® 4GL Transformation


Front End
Progress 4GL Source Code
Progress Database Schemata
Preprocessing, Lexing and Parsing
Code and Schema Abstract Syntax Trees
Transform
Refactoring, Transformation and Modernization
Transformed Abstract Syntax Trees
Back-End
Anti-Parsing
Modernized Progress 4GL Source Code
Inputs
Intermediate Artifacts
Processes
Outputs

How It Works

- 3 phase process:
  - Front End - Reads 4GL source code and handles the preprocessing, lexing and parsing to generate the “pristine” ASTs.
  - Transform - Executes your custom TRPL rules to analyze, refactor and transform the ASTs.
  - Back End - Anti-parses the transformed ASTs into 4GL source code.
- This is a non-interactive process that runs the same TRPL programs against the entire application.
- This can take seconds/minutes for a small project or hours for a large project.
- The resulting output (if done correctly) is syntactically correct 4GL code.

Command Line

- The common way to run the transformation process (all 3 phases):

```
     java -classpath p2j/build/lib/p2j.jar
          com.goldencode.p2j.convert.ProgressTransformDriver
          F2+TR+AP
          <transformation_ruleset>
          <4gl_program> ...
```

- The phases are controlled by the `F2+TR+AP` (front_end+transform+anti_parsing).
- You define the transformation ruleset to run (this is your TRPL program).
- Although the transformation ruleset filenames normally have the extension `.xml`, the extension is not specified in the command line.
- Specifying Programs to Transform
  - The default approach is to provide an explicit list of one or more files on the command line (see above).
  - `-F <whitelist>` is an explicit whitelist of the files to process.
  - `-S <directory> <filespec>` is a filespec form to specify wildcard matches.
  - `-X <directory> <filespec> <blacklist>` is a filespec form combined with a blacklist to exclude files.

Transformation Artifacts

- Given 4GL code in a file named `program.p`, the most important artifacts created:
  - **`program.p.cache`** contains the fuly preprocessed 4GL source code (all includes, conditional expansions and arg/name references expanded)
  - **`program.p.ast`** is the AST in XML format
  - **`program.p.mod`** is the transformed 4GL source code created
- The **`.mod`** file is comparable to the **`.cache`** file. It is a fully preprocessed version of the code, but with all changes applied.

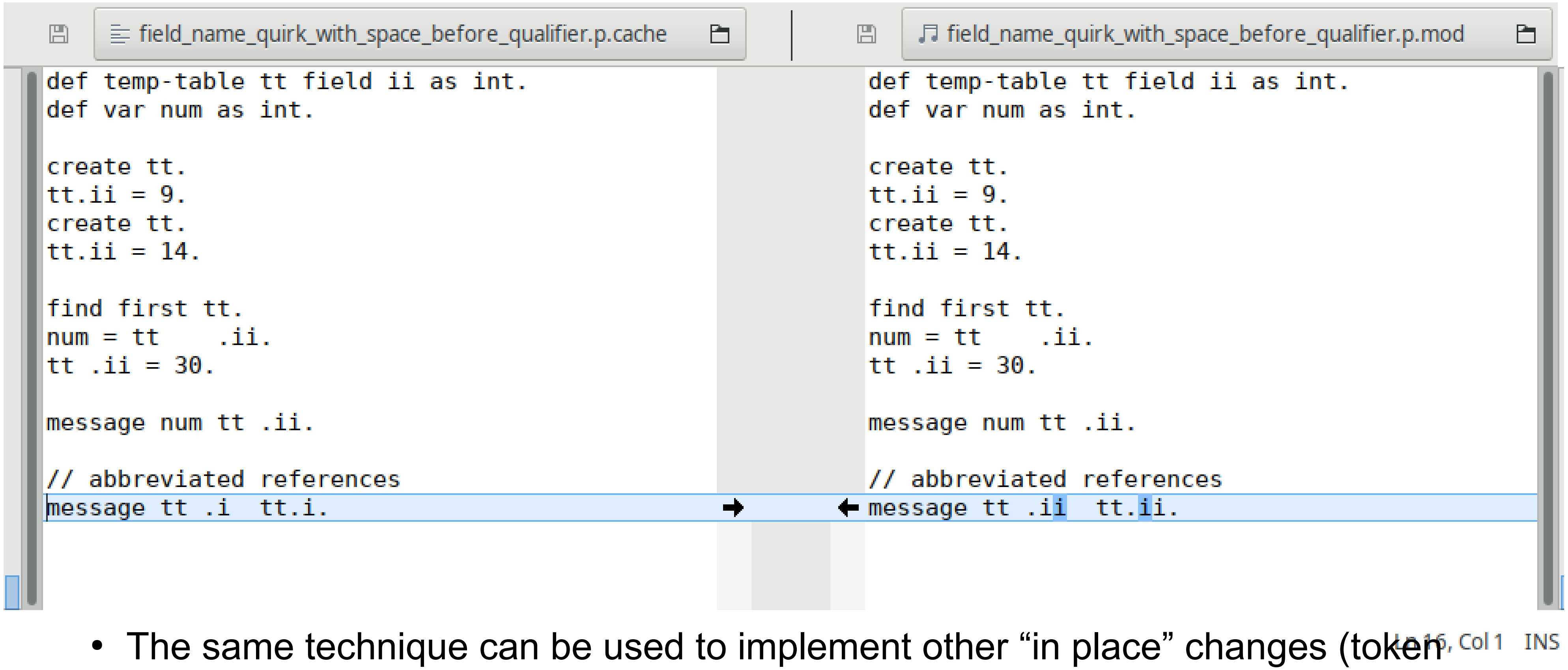
Example 1 Eliminate Field Abbreviations

- Use `progress/eliminate_field_abbreviations` as the TRPL rule set.

```
java -classpath p2j/build/lib/p2j.jar
     com.goldencode.p2j.convert.ProgressTransformDriver
     F2+TR+AP
     progress/eliminate_field_abbreviations
     <4gl_program>
```

- Matches on any static field reference (doesn’t matter if it is on a buffer, table, temp-table or work-table).
- Examines the `schemaname` annotation (fully qualified and unabbreviated field name) and compares field portion with field portion of the reference.
- Replaces the AST node’s text if it is abbreviated.
- The only tricky part of the TRPL rules is the part that handles the quirk where “qualified field names can have some whitespace inside”. Such nodes must have the `original-text` annotation set to the new value.
- This ruleset is generic and can be run on your projects without changes.

Example 1 Eliminate Field Abbreviations


field_name_quirk_with_space_before_qualifier.p.cache
field_name_quirk_with_space_before_qualifier.p.mod
Ln 16, Col 1
INS

```
def temp-table tt field ii as int.
def var num as int.

create tt.
tt.ii = 9.
create tt.
tt.ii = 14.

find first tt.
num = tt    .ii.
tt .ii = 30.

message num tt .ii.

// abbreviated references
message tt .i  tt.i.
```

```
def temp-table tt field ii as int.
def var num as int.

create tt.
tt.ii = 9.
create tt.
tt.ii = 14.

find first tt.
num = tt    .ii.
tt .ii = 30.

message num tt .ii.

// abbreviated references
message tt .ii  tt.ii.
```

- The same technique can be used to implement other “in place” changes (token types or other state for nodes).
- Does not change the structure of the AST.
- Does not rely upon fragile text matching or replacement behavior.

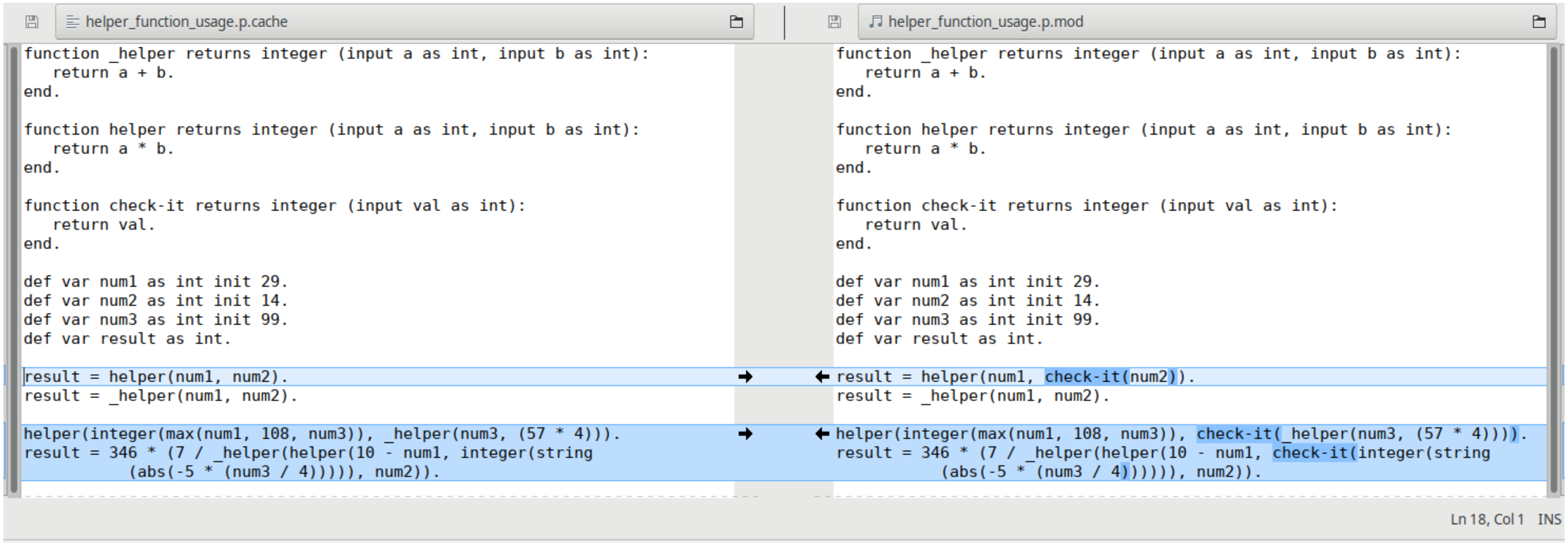
Example 2 Rewrite Function Params

- Use `progress/rewrite_function_signature` as the TRPL rule set.

```
java -classpath p2j/build/lib/p2j.jar
     com.goldencode.p2j.convert.ProgressTransformDriver
     F2+TR+AP
     progress/rewrite_function_signature
     <4gl_program>
```

- Matches all function calls to user-defined `helper(integer, integer)` that returns `integer`.
- This is not a generic ruleset. It can be used with the example 4GL programs `helper_function_usage.p` and `non_helper_function_usage.p` (provided by Golden Code).
- The matching logic:
  - `FUNC_INT` token type (function call that returns integer)
  - `text` is case-insensitively equal to “helper”
  - `builtin` annotation does not exist or is set to `false`
  - Takes exactly 2 parameters, each subtree evaluates to `integer` type.
- When a match is found, it:
  - Creates a new AST node (`createProgressAst()`) with text “check-it” and type of `FUNC_INT`.
  - Inserts it into the tree in between the 2nd parameter and the matched “helper” `FUNC_INT` node.
  - This shows structural change of the tree AND it is written to be independent of the complexity of the parameter sub-expression.
  - Inserts “shadow nodes” to handle hidden syntactic sugar of program (the LPARENS and RPARENS of the `check-it()` function call.
- The only tricky part of the TRPL rules is the part that handles the shadow nodes using (`insertInStream()`).

Example 2 Rewrite Function Params



helper_function_usage.p.cache

```
function _helper returns integer (input a as int, input b as int):
   return a + b.
end.

function helper returns integer (input a as int, input b as int):
   return a * b.
end.

function check-it returns integer (input val as int):
   return val.
end.

def var num1 as int init 29.
def var num2 as int init 14.
def var num3 as int init 99.
def var result as int.

result = helper(num1, num2).
result = _helper(num1, num2).

helper(integer(max(num1, 108, num3)), _helper(num3, (57 * 4))).
result = 346 * (7 / _helper(helper(10 - num1, integer(string
           (abs(-5 * (num3 / 4))))), num2)).
```

helper_function_usage.p.mod

```
function _helper returns integer (input a as int, input b as int):
   return a + b.
end.

function helper returns integer (input a as int, input b as int):
   return a * b.
end.

function check-it returns integer (input val as int):
   return val.
end.

def var num1 as int init 29.
def var num2 as int init 14.
def var num3 as int init 99.
def var result as int.

result = helper(num1, check-it(num2)).
result = _helper(num1, num2).

helper(integer(max(num1, 108, num3)), check-it(_helper(num3, (57 * 4)))).
result = 346 * (7 / _helper(helper(10 - num1, check-it(integer(string
           (abs(-5 * (num3 / 4)))))), num2)).
```

Ln 18, Col 1   INS

- Does change the structure of the AST, this is very important to the result.
- The same technique can be used to insert other structural changes.
- Notice that there is no dependency on string processing for the result.
- ARBITRARILY complex parameter sub-expressions are handled as easily as the single node case.
- The `non_helper_function_usage.p` case has virtually identical text but very slight changes that cause the 2nd parameter to be a `logical`, which means no changes are applied!

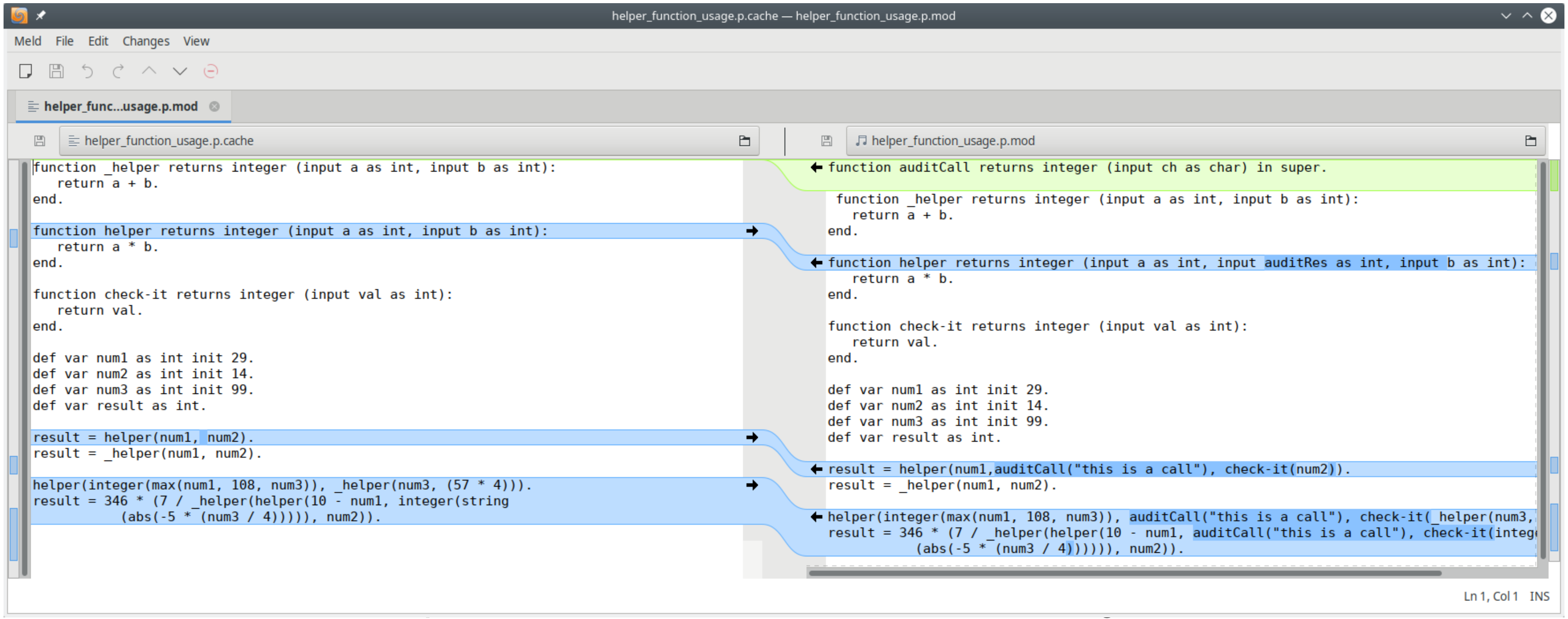
Example 3 Insert Template

- Use `progress/rewrite_function_signature_2` as the TRPL rule set.

```
java -classpath p2j/build/lib/p2j.jar
     com.goldencode.p2j.convert.ProgressTransformDriver
     F2+TR+AP
     progress/rewrite_function_signature_2
     <4gl_program>
```

- Matches all function calls to user-defined `helper(integer, integer)` that returns `integer`.
- This is not a generic ruleset. It can be used with the example 4GL programs `helper_function_usage.p` and `non_helper_function_usage.p` (provided by Golden Code).
- The matching logic:
  - `FUNC_INT` token type (function call that returns integer)
  - `text` is case-insensitively equal to “helper”
  - `builtin` annotation does not exist or is set to `false`
  - Takes exactly 2 parameters, each subtree evaluates to `integer` type.
- When a match is found, it:
  - Inserts a sub-tree created from a templ`ate named "audit_function_call_v1".
  - Inserts it into the tree in between the existing 1st and 2nd parameters as a child of the matched “helper” `FUNC_INT` node.
  - This shows structural change of the tree AND it is written to be independent of the complexity of the parameter sub-expression.
  - Inserts “shadow nodes” to fixup the hidden syntactic sugar on either side of the inserted template (the COMMA after the new parameter.
- If it inserts that code, it will ALSO (in “post” rules):
  - Insert a FUNCTION IN SUPER declaration from a template.
  - Modify the helper() function signature to add a new parameter using a template.``

Example 3 Insert Template


helper_function_usage.p.cache — helper_function_usage.p.mod
Meld File Edit Changes View
helper_func...usage.p.mod
helper_function_usage.p.cache
helper_function_usage.p.mod
Ln 1, Col 1 INS

```
function _helper returns integer (input a as int, input b as int):
   return a + b.
end.

function helper returns integer (input a as int, input b as int):
   return a * b.
end.

function check-it returns integer (input val as int):
   return val.
end.

def var num1 as int init 29.
def var num2 as int init 14.
def var num3 as int init 99.
def var result as int.

result = helper(num1, num2).
result = _helper(num1, num2).

helper(integer(max(num1, 108, num3)), _helper(num3, (57 * 4))).
result = 346 * (7 / _helper(helper(10 - num1, integer(string
           (abs(-5 * (num3 / 4)))))), num2)).
```

```
function auditCall returns integer (input ch as char) in super.

 function _helper returns integer (input a as int, input b as int):
   return a + b.
end.

function helper returns integer (input a as int, input auditRes as int, input b as int):
   return a * b.
end.

function check-it returns integer (input val as int):
   return val.
end.

def var num1 as int init 29.
def var num2 as int init 14.
def var num3 as int init 99.
def var result as int.

result = helper(num1,auditCall("this is a call"), check-it(num2)).
result = _helper(num1, num2).

helper(integer(max(num1, 108, num3)), auditCall("this is a call"), check-it(_helper(num3,
result = 346 * (7 / _helper(helper(10 - num1, auditCall("this is a call"), check-it(intege
           (abs(-5 * (num3 / 4)))))), num2)).
```

- Templates are XML “snippets” of an AST subtree, optionally including shadow nodes.
- The same structure as the persisted AST is used.
- Templates support replacements syntax.
  - Any string value in an XML node attribute will be searched for references like ${ttype}.
  - If present, the value will be replaced with the text provided in the graft/graftAt.

Limitations

- Does not (yet) emit changes into the original source files (includes and non-preprocessed procedures or classes). The changes are emitted in fully preprocessed form.
  - It is OK to run the fully preprocessed versions.
  - OR the user can apply the changes back as patches.
- Although the J-Mode (Progress® 4GL to Java) has rules available to completely refactor and convert entire applicatons, the P-Mode (Progress® 4GL to Progress® 4GL) currently does not have pre-defined transformation rules.
- Whitespace handling is more effort than necessary (shadow node processing).
- Line and column numbers are not automatically calculated for any changes.

FWD transform

TRPL Primer

Introduction

- TRPL expression syntax largely the same as Java (as of J2SE version 1.4)
- All symbols are case-sensitive
- Scalar expressions
- Method invocation using familiar Java syntax (`object.method()`)
- Type casting possible (using a modified syntax)
- Primitive values represented using Java wrapper types (e.g., `java.lang.Integer`)
- https://proj.goldencode.com/projects/p2j/wiki/Writing_TRPL_Expressions for more detail

Features

- **Auto-boxing**
  - Automatic conversion between primitive data types and Java wrapper types.
  - E.g., `int → java.lang.Integer`, `java.lang.Boolean → boolean`
- **Automatic Type Conversion**
  - Widening/narrowing numeric conversions and object reference data type conversions.
  - Allow narrowing conversion with care, precision loss can occur!
- **Automatic Null Checking**
  - Object references in boolean expressions checked for null before being dereferenced.
  - Equality/range comparisons to null evaluate false; inequality to null evaluates true.
  - Method parameters **not** null-checked prior to derefence.
- **Property Notation**
  - Properties of an object exposed via a bean-like API (e.g., `foo.getBar()` and `foo.setBar(int)`) are also exposed via shorthand, “property” notation.
  - E.g., `foo.bar > 10` (getter) and `foo.bar = 55` (setter).

Literals

- **String** (e.g., "Hello World"): auto-boxed to `java.lang.String`.
- **Integral** (e.g., 100, -9999): signed, base 10 numbers without a decimal point, mapped Java primitive `long`, auto-boxed to `java.lang.Long`.
- **Hexadecimal** (e.g., 0x01, 0xffff): base 16 numbers with a `0x` prefix, mapped to Java primitive `long`, auto-boxed to `java.lang.Long`.
- **Floating point** (e.g., 1.234, -56.7): signed, base 10 numbers with a decimal point, mapped to Java primitive `double`, auto-boxed to `java.lang.Double`.
- **Boolean** (e.g., true, false): logical constants mapped to Java primitive `boolean`, auto-boxed to `java.lang.Boolean`.
- The **null** literal represents the `null` value. Typically used with the == or != operator to test whether opposite operand is or is not the `null` value, respectively.

Object References

- Java objects are created in the following ways:
  - as the return value of a method call on an existing object
  - as the return value of a property accessor shorthand call on an existing object
  - as the return value of a user-defined function
  - as an exported resource from a registered worker library
  - as the return value of one of the special create* functions (built-in functions in TRPL)
    - stringBuilder = create("java.lang.StringBuilder", 64)
    - myListOfStrings = createList("a", "b", "c", "one", "two", "three"));
  - by de-referencing a variable (e.g., myVar);
  - by auto-boxing a literal value;
  - by auto-boxing a primitive value returned by a method call.

Method Invocation

- Java-like syntax for method invocation:
  - `<object reference>.<method name>([param1 [, …]])`
  - No implicit `this` reference. TRPL expressions do not run in the context of an object instance.
  - Static method invocation qualified by a class name does NOT work (no implicit class name resolution).
- Given a variable `string1` of type `java.lang.String`, initialized to "Hello World", the following invocation is valid:
  - `string1.indexOf('He')`
  - would return 0 upon execution.
- Libraries of worker functions can be imported as global methods.

Differences from Java

- No implicit `this` reference
- No direct static references
- Special typecast syntax
- No inline assignments
- No `new` keyword
- Arrays not supported
- String delimiters
- No `char` literals
- String literals can not be dereferenced
- Several operators not available
- Method call chaining is limited
- No import statement
- Generics not (currently) supported
- https://proj.goldencode.com/projects/p2j/wiki/Writing_TRPL_Expressions#DifferencesLimitations-Compared-to-Java-Syntax for more detail

Operators

- Operators in this table are listed in order of their precedence, from those evaluated first to those evaluated last.
- Operators which have the same precedence are grouped together and are evaluated left to right.
- Parentheses (()) may be used to group operations which must be evaluated in a different order.

Precedence	Symbol	Type	Operands	Operation Performed
0	`.`	Special	Binary	Method invocation
1	`!` or `not`	Logical	Unary	Logical complement
2	`~`	Bitwise	Unary	Bitwise complement
3	`-`	Arithmetic	Unary	Negation
4	`*`	Arithmetic	Binary	Multiplication
4	`/`	Arithmetic	Binary	Division
4	`%`	Arithmetic	Binary	Modulo/Remainder
5	`+`	Arithmetic	Binary	Addition
5	`-`	Arithmetic	Binary	Subtraction
6	`<<`	Bitwise	Binary	Left shift
6	`>>`	Bitwise	Binary	Right shift with sign extend
6	`>>>`	Bitwise	Binary	Right shift with zero extend

Operators (continued)

Precedence	Symbol	Type	Operands	Operation Performed
7	`<`	Logical	Binary	Is less than
7	`<=`	Logical	Binary	Is less than or equal to
7	`>`	Logical	Binary	Is greater than
7	`>=`	Logical	Binary	Is greater than or equal to
8	`==`	Logical	Binary	Is equal to (primitive or object identity)
8	`!=`	Logical	Binary	Is not equal to (primitive or object identity)
9	`&`	Bitwise	Binary	Bitwise AND
10	`^`	Bitwise	Binary	Bitwise XOR (exclusive OR)
11	`|`	Bitwise	Binary	Bitwise OR (inclusive OR)
12	`&& and`	Logical	Binary	Logical AND
13	`|| or`	Logical	Binary	Logical OR

Token Types

- Each AST node has a token type. This is an integer value that identifies the meaning of the node.
- In a TRPL expression, a human-readable token name is used instead of the integer value. Different types of ASTs use different sets of token type to token name mappings.
- FWD separates these by namespaces:
  - `prog` - Progress token types (see ProgressParserTokenTypes). This namespace is available when using the Code Analytics tool set and during code and schema conversion.
  - `java` - Java token types (see JavaTokenTypes). This namespace is available during code and schema conversion.
  - `xml` - XML grammar token types (see XmlTokenTypes). This namespace is available during code and schema conversion.
  - `data` - Data model token types (see DataModelTokenTypes). This namespace is available during schema conversion.
- Namespaces are used as qualifiers in TRPL expressions, so the correct token type to name mapping is used. For example, `prog.kw_for` specifies the token type for the Progress 4GL FOR statement; `java.kw_for` specifies the token type for the Java for statement.

AST Node Text

- Each AST node which was parsed from source code or schema text, the `text` property of an AST node usually represents the text from the input file.
- This might be the name of a variable or field, or a keyword in the language.
- This text is usually left in its original form.
- Since it is not normalized into any regular form, the node text is not always the most reliable property of an AST on which to match a pattern.
- This text is the original, non-regular or ambiguous features which we are trying to avoid.

Annotations

- **Arbitrary information** added to an AST node with a given **label**, so it can be retrieved later.
- **Added by the parser** to store information not inherently available from the tree structure or other standard properties of an AST.
- **Added by downstream TRPL programs** to store computed/derived information.
- Each annotation has a **data type**.
- Can be **queried** using getter methods in `Aast` interface.
- Extremely useful for **pattern matching**.
- Example: **`schemaname`** is a qualified, canonical name attached to the AST node of every database table and field reference in an application's business logic.

Structural Relationships

- The structure of an AST is key to its meaning.
- The tree structure represents **the syntax of the language in its most pure form**.
- The relationships between nodes when combined with token types, encode the meaning of language constructs the AST represents.
- It is important to understand the structure of an AST in order to write a TRPL expression which describes the meaning of the underlying code construct.

Structural Relationships

- Where the placeholder `<ref>` is used in an example expression, it represents an `Aast` object reference.
- **`node`** - One point or location in the tree which represents a single token of code. Every node has a token type and text (could be an empty string) associated with it. Every node except the root of the tree has a single parent; each node has 0 or more direct child nodes. Every node of type `Aast` has a unique numeric identifier associated with it.
- **`this`** - The "active" node; the node which is the focus of the current event during a tree walk. This is an implicitly available object reference.
- **`root`** - The single ancestor of the entire tree. The root is the only node in a tree which has no parent. Thus, a call to `<ref>.getParent()` at the root node will return `null`. Retrieved with `<ref>.getRoot()` or `<ref>.root`.
- **`depth`** - The number of generations or levels a node is removed from the root node. The depth of the root node is 0; the depth of its immediate child nodes is 1; the depth of its grandchildren is 2; and so on. Depth is retrieved with `<ref>.getDepth()` or `<ref>.depth`.
- **`leaf`** - A node which has no children. "Leaf-ness" can be tested with `<ref>.isLeaf()` or `<ref>.leaf` or the helper function `leaf()`.

Structural Relationships

- **`ancestor`** - A node which has a direct lineage to another node and which has a lower depth. There are several variants of the `getAncestor()` and `ancestor` methods and helper functions to retrieve ancestor nodes or test whether an ancestor node with certain properties exists, respectively.
- **`descendant`** - A node which has a direct lineage to another node and which has a higher depth. There are several variants of the `descendant()` methods and helper functions to test whether a descendant node with certain properties exists.
- **`parent`** - The direct ancestor of another node. A node can have 1 or 0 parents. Multiple nodes may have the same parent. Retrieved with the method `<ref>.getParent()` or the shorthand `<ref>.parent`, or simply the helper function `parent`.
- **child** - The direct descendant of another node. A node can have 0 or more children. The first of a node's children can be retrieved (as a BaseAST object) with `<ref>.getFirstChild()` or `<ref>.firstChild`. An arbitrary child node can be retrieved (as an `Aast` object) with `<ref>.getChildAt(N)`, where N is a zero-based index.
- **sibling** - A different child node with the same parent. Siblings are retrieved with calls to `<ref>.getPrevSibling()` or `<ref>.prevSibling` (as an `Aast` object) or with `<ref>.getNextSibling()` or `<ref>.nextSibling` (as a `BaseAST` object), relative to the active node.
- The authoritative reference for the Progress ASTs created by the FWD parser is the parser's grammar definition, located in the source code at `src/com/goldencode/p2j/uast/progress.g`.

Editing Tools

- **`ProgressPatternWorker`**
  - **`createProgressAst()`** - Creates a new AST node which is completely disconnected from the tree. Set its state and then insert it using `Aast.graft()` or `Aast.graftAt()`.
  - **`createShadow()`** - Creates a new shadow node which is completely disconnected from the tree. Set its state and then insert it using `insertInStream()`.
  - **`insertInStream()`** - Add shadow nodes (hidden nodes like whitespace and other punctuation which is syntactic sugar) maintaining the proper left/right linkages so that the output will be emitted with proper formatting. Also allows AST nodes to be inserted relative to the stream while maintaining the shadow node linkages.
  - **`removeFromStream()`** - Remove shadow nodes or AST nodes from their position in the left/right output stream.
- **`AnnotatedAst`** (it is the parent class of every node which is a **`ProgressAst`** and implements **`Aast`**)
  - **remove() -** Disconnect the subtree rooted at a given node from the main tree. It can be grafted back somewhere else, or discarded entirely.
  - **graft()** and **graftAt() -** Attach the sub-tree rooted the given node into the target tree, optionally at a specific child index position. Fixup any parent/child linkages and set node ids to match the tree.

Editing Tools

- **`AnnotatedAst`** `(continued)`
  - **`duplicate()`** and **`duplicateFresh()`** - Duplicate entire sub-trees rooted at a given node. These are disconnected from the given tree and can be grafted in.
  - Moving can be achieved by `remove()` followed by `graft()` or `graftAt()`.
- **`ExpressionConversionWorker`**
  - **`expressionType()`** - Calculates the type of a sub-expression as represented by a specific AST node.
- **`TemplateWorker`**
  - **`load()`** - Load pre-defined sub-tree snippets of ASTs from an XML template file.
  - **`graft()`** and **`graftAt()`**- Attach a named template to the given tree (optionally at a specified index position) with a set of text replacements applied before grafting.
  - This is a useful way to build and attach very complex sub-trees with very little code.

FWD transform

Usage Tips

Writing TRPL Expressions

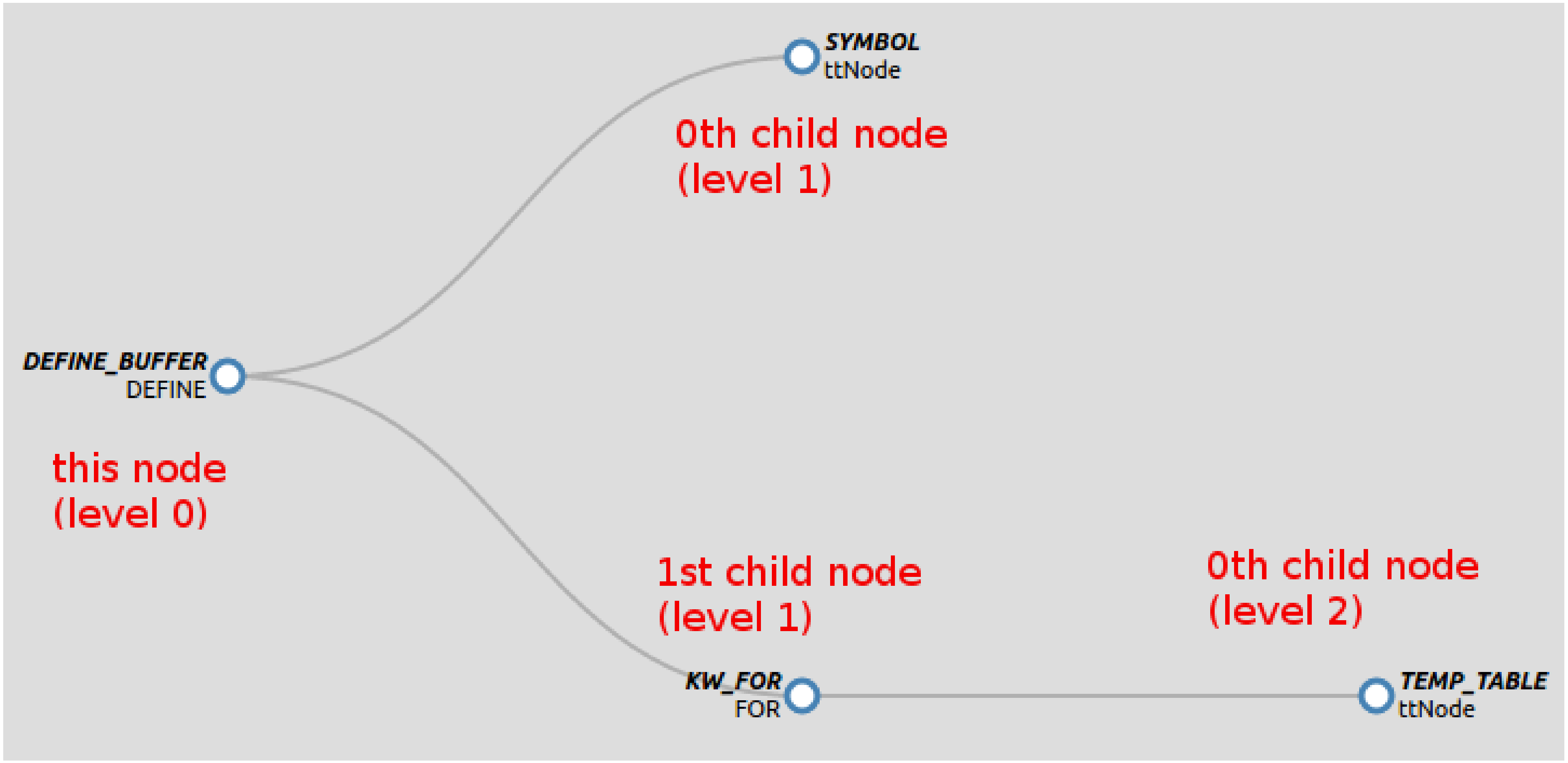
- Look at the AST structure that corresponds to the code you are trying to match.
    - Write a code snippet and parse it, then view it.
    - Use the predefined reports in FWD Analytics to find locations that already exist and look at the Source/AST View.
- Decide which node is the best situated. Usually this is about finding the node that is most “centrally” located.
- All the context for the expression is written from that node’s “perspective”.
- Use the token type first, to roughly match a set of possible nodes.
- Refine this to get an exact match by adding use of tree structure, annotations and text.

Copy or This?

- TRPL is designed for a highly pipelined approach, where multiple (maybe hundreds of) rulesets are processed in sequential order.
- The output of one ruleset (an optionally modified tree) is the input to the next.
- To facilitate this, the input tree is completely duplicated at the time each ruleset runs. The ruleset has access to both the input and output trees.
- In each ruleset, there are 2 representations of the current node in the tree:
  - **`this`** – A reference to the node in the input tree.
  - **`copy`** – A reference to the duplicate of **`this`** but in the output tree.
- At the beginning, these trees are identical. This means that **`this`** and **`copy`** are often interchangeable.
- The TRPL engine walks the tree using **`this`** and edits should ONLY EVER be made to **`copy`**. **NEVER EDIT `this`.**

Look at the AST

- Tree visualization of DEFINE BUFFER



Don’t Fight the Tree!

- Let the structure of the AST solve the problem for you.
- TRPL will walk the tree for you.
- Your expression is being executed at each possible location in the entire application.
- It is a “callback” model with the events determined by the tree structure.
- The tree structure is the pure form of the language syntax as represented in your code.
- Matching on the tree is matching on the syntax.
- If you are finding yourself doing something “unnatural”, ask: how can the tree structure help me?

FWD transform

How to Get Started

How to Get Started

- Download and install FWD.
- Download one of the sample template projects (there is one for ChUI and one for GUI).
- Follow the “Getting Started” instructions to get the template project installed and configured for your application code, including placing your code and schemata into the template project.
- Run the `ProgressTransformDriver` in F2 mode to test the parsing of the project. Please refer to https://proj.goldencode.com/projects/p2j/wiki/Conversion_Handbook chapters 6 through 12 for details on how to work through parsing issues. You MUST have your 4GL code parsing properly before you can try transformation rules.
- Write your transformation rules using the examples in the FWD distribution (see `rules/progress/*`) and the documentation in https://proj.goldencode.com/projects/p2j/wiki/Understanding_ASTs_and_TRPL.
- From there you can test your rules and iterate rapidly until you reach your objectives.
- Register an account in Redmine (https://proj.goldencode.com/account/register) and post in the Conversion forum (https://proj.goldencode.com/projects/p2j/boards/2) for help.

FWD transform

Planned Improvements

Planned Improvements - P-Mode

- Phase 2
  - Need to automatically calculate the line/column numbers of changes to the tree. Some of our processing depends upon this to work. For example, artificial nodes don't get anti-parsed AND some processing depends on calculating which nodes are left/right of each other so the relative line numbers must be consistent.
  - Provide a default formatting/whitespace output for nodes that were created via program (TRPL rules) instead of being read from the input source code. In the current parser, in order to have formatted output, hidden nodes with the whitespace would have to be inserted in between nodes and linked properly. The core issue here is that simple heuristics may not be enough since there is so much different syntax in the language.
  - Add shadow node support to the `TemplateWorker` for loading and grafting. This will also require changes in `XmlFilePlugin`.
  - Add more tools (in `ProgressPatternWorker`) for insert/delete/move of nodes and shadow nodes (keeping all the shadow node linkages intact).
  - Provide tools for the TRPL user to control/edit/specify the formatting, without hard coding the whitespace as hidden nodes.
  - Create a tool to help apply changes to the original files by writing the changes as patches that could be applied via diff. In combination with the current output, this could make it unnecessary to implement phase 3.
- Phase 3
  - Implement rules to separate business logic from user interface. This will include the creation of APIs for the business logic.
  - Implement an option to flow edits back to the original source files (even include files). This is not guaranteed to work in all cases since changes may stretch across the boundaries of preprocessor expansions, conditional preprocessor directives and nested includes. However, it is also possible that many or most changes could be calculated safely.

Planned Improvements - TRPL

- Move our existing transformation rules that calculate important properties to an early enough location that it can be integrated into reporting. This would include things like buffer scoping, frame scoping, index selection, transaction/block properties and more.
- Duplicate Code Identification. We can identify arbitrary code matches across the entire application using a bottom-up fingerprinting approach for each unique sub-tree in the application. By using fuzzy logic, we can match code that is the same whether it was cut and pasted or just independently coded the same way. Using these fingerprints we can turn duplicated code into common code.
- Improved TRPL syntax and structure, source level debugging. This will likely be done by shifting to a Scala implementation.

FWD

Find Us On the Web!

www.beyondabl.com

facebook.com/beyondabl

twitter.com/beyondabl

linkedin.com/company/fwd-project

youtube.com/channel/
UCk3pga7EKxAQVOV_CiYOR7g