4GL Code Analytics

Unique Tools for Deep Code Analysis and Inspection

Greg Shah
Golden Code Development

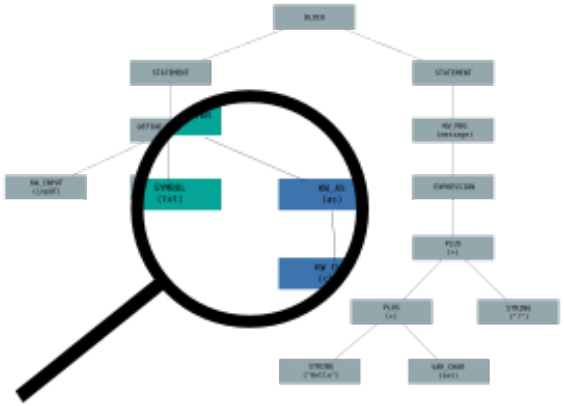
Wednesday October 9, 2019

Agenda

- Background
- Handling Flexibility and Scale
- Standard Reports
- Search
- Custom Reports
- Usage Tips
- How to Get Started

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

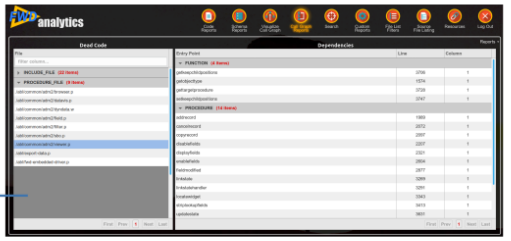
Fully automated transformation and modernization of entire applications without a manual rewrite.

Transcend

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

FWD analytics Tools for reporting, statistics and analysis, which **expose the inner workings of an application**.

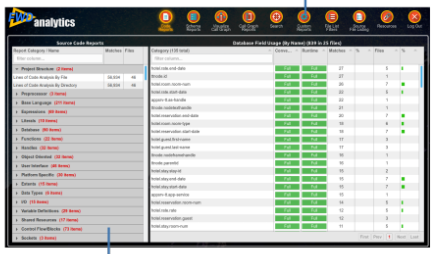
Search for arbitrary 4GL syntax to find exact answers to questions that were previously difficult or impossible to answer.



Call Graph Analysis determines program dependencies, missing code and explores all reachable code paths. Reports allow identification and removal of dead code reducing application size by 25% to 40%

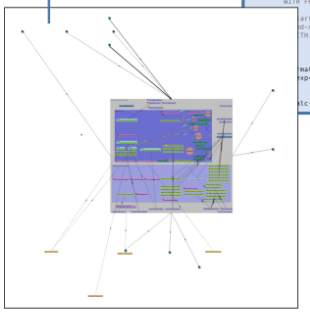
Explore

Build custom reports to meet specific needs.

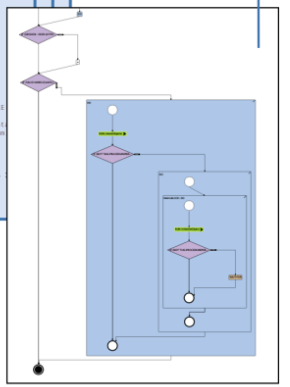


Built-in reports and statistics provide hundreds of predefined views of the code and schemata.

Visualize and interactively explore the call graph.



Generates logic flow charts for all code in the application.



Advantages

- Reduce development effort.
- Improve code quality.
- Deeply understand existing code.
- Compensate for missing documentation,
- Empower developers to more capably handle:
  - The most complex refactoring and modernization projects.
  - Making changes at scale, even with the largest of applications.

Why Use FWD Analytics?

- Reduce development effort.
- Improve code quality.
- Deeply understand and explore existing code.
- Compensate for missing documentation.
- Empower developers to more capably handle:
  - The most complex refactoring, transformation and modernization problems; AND
  - Making changes at scale, even with the largest of applications.

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.

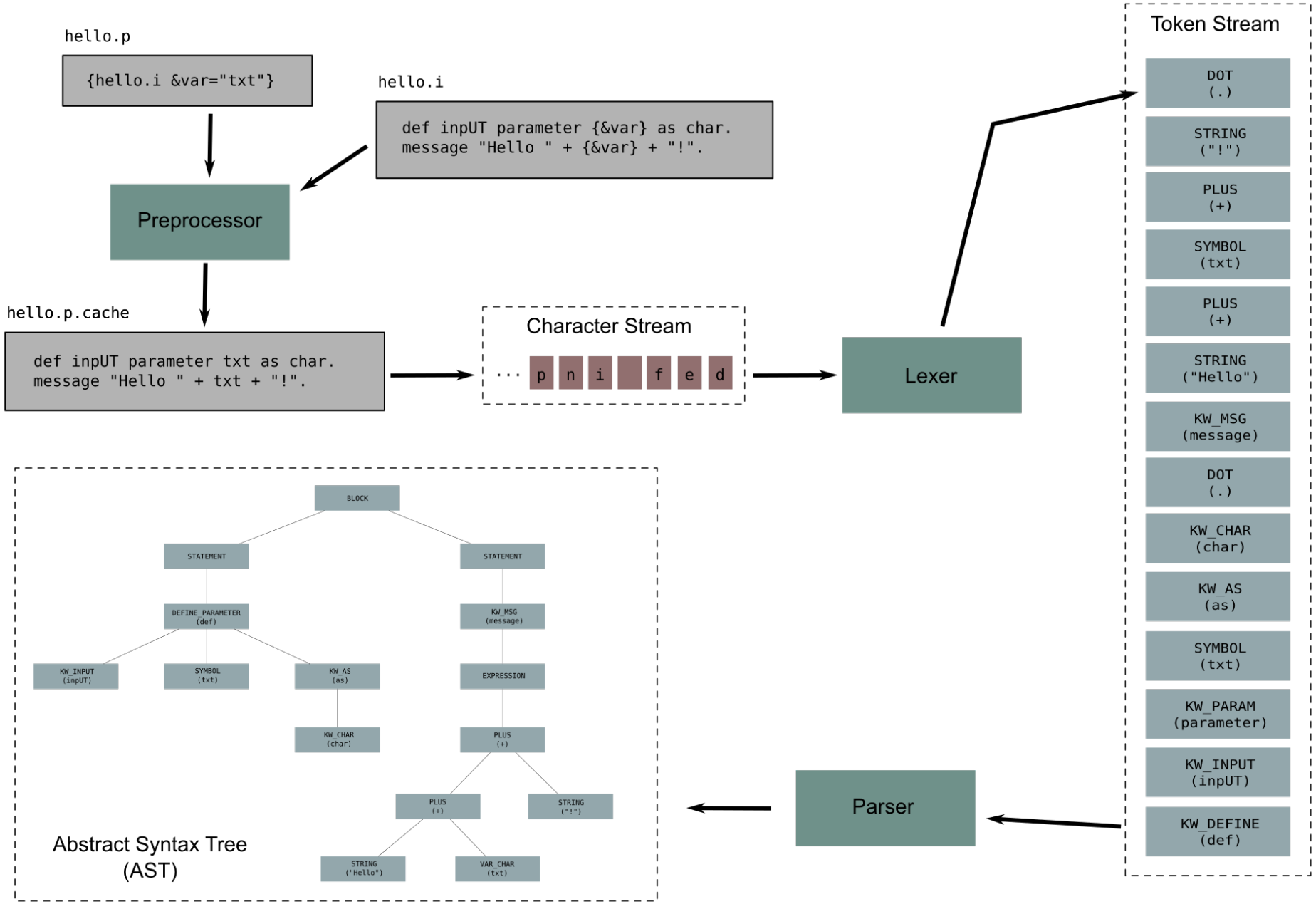
Syntax Flexibility Increases Cost

- 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



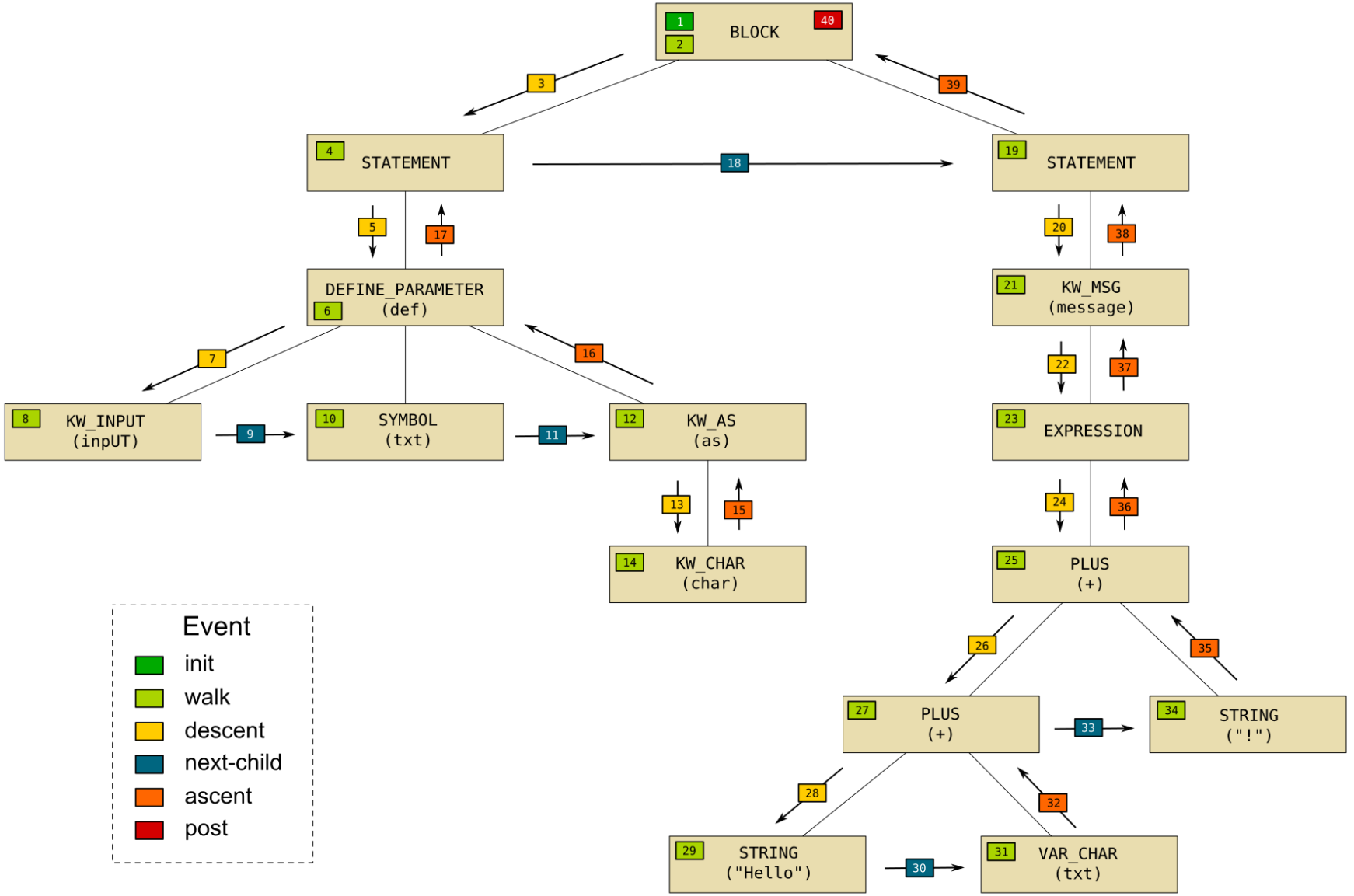
Scale Multiplies Issues

- 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



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.

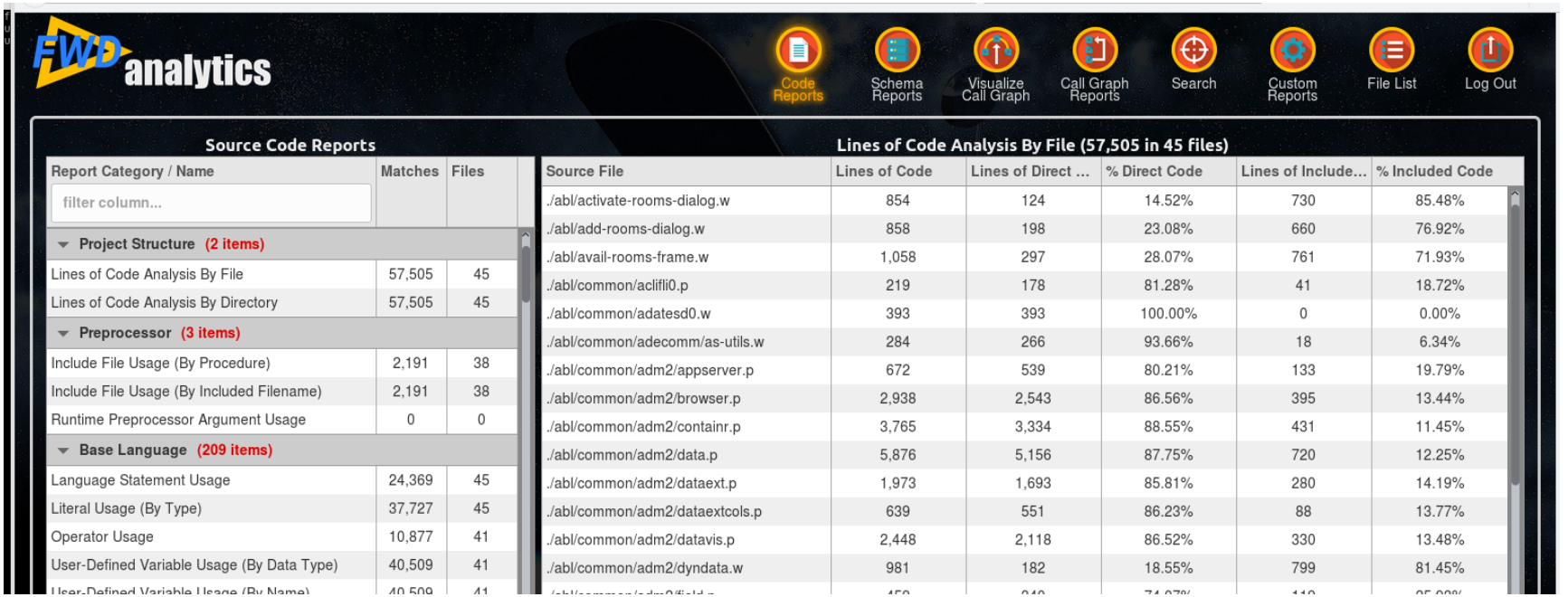
Report Generation

- After the entire application has been parsed, we can run the report generation step.
- This is a non-interactive process that runs a set of pre-defined TRPL programs to calculate a few hundred reports.
- This can take minutes for a small project or hours for a large project.
- Both the parsing and the report generation can be scripted and used in CI or build servers.
- After the reports are generated, they can be accessed via an interactive web interface.

Standard Reports

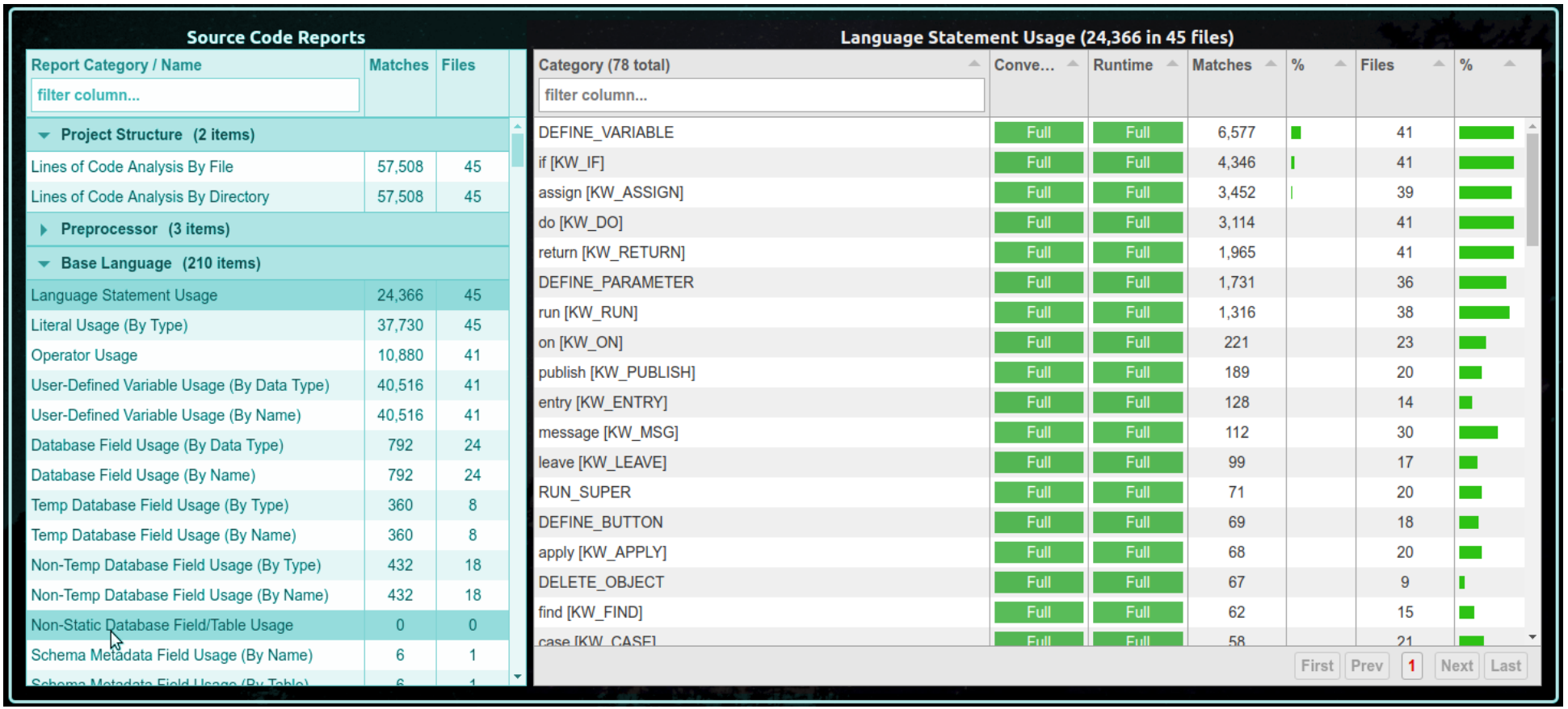
Initial Reports Screen

- List of predefined reports on left
- Currently viewed report on right
- Most reports are a set of mutually exclusive categories
- Summary statistics for the report at the top
- Individual categories have their own statistics
- Filter and sort columns using the column header
- Click on a row in the current report to see the exact list of matches
- Pagination controls at the bottom



Summary Report

- When a report is selected from the available reports list, the selected Summary Report is loaded into the right side of the screen.
- Each Summary Report organizes the matches based on categories, one row per category.



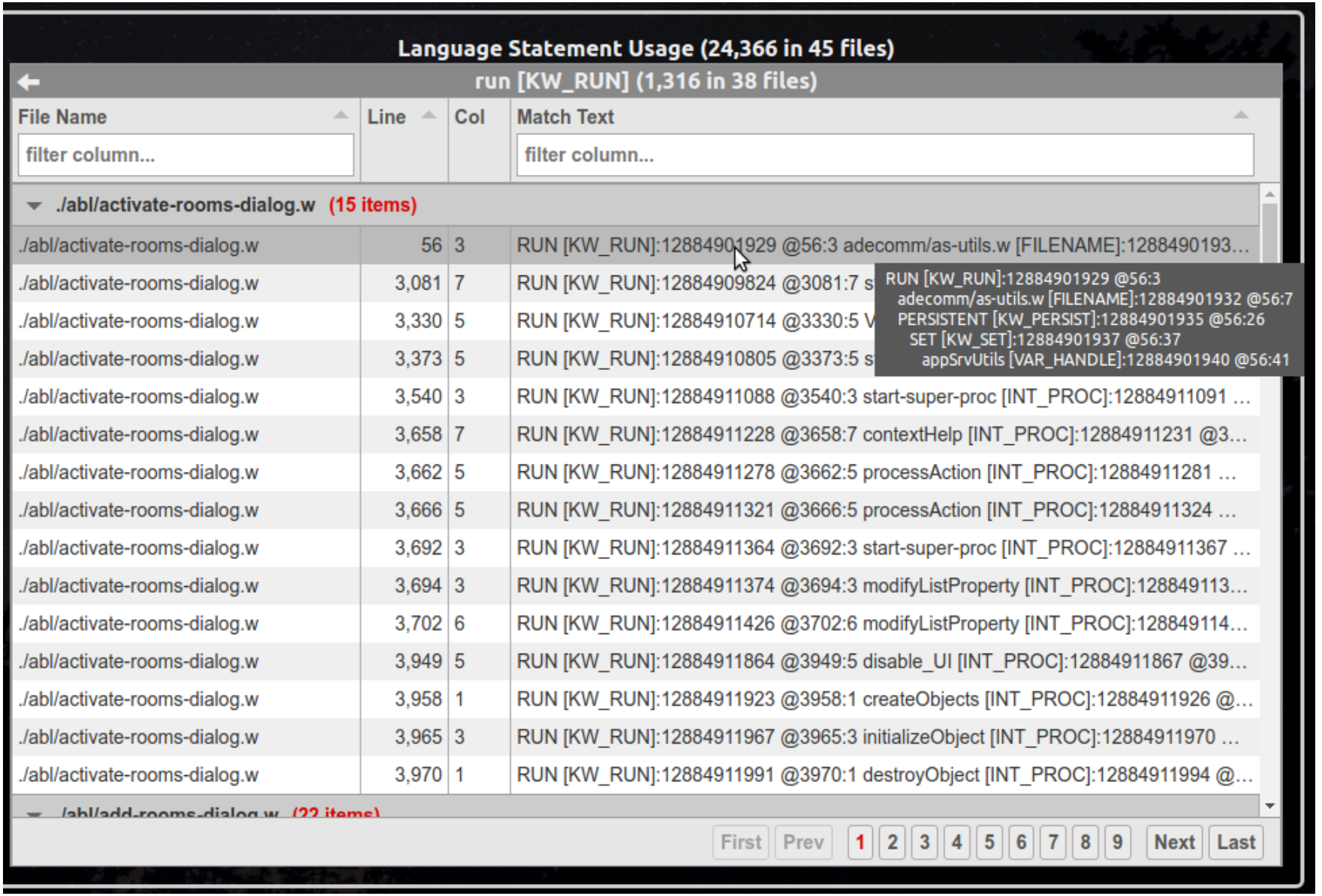
Summary Report

- The title is the report name (e.g. "Language Statement Usage") followed by the total number of matches (e.g. there are 24,366 matches to a Language Statement) and the number of files that contained matches (e.g. 45).
- Each report has a match condition that is being used to determine whether an AST node (from the code or a schema) should be included in the report. This match condition is a single boolean expression, which can be of arbitrary complexity.
- By default the table is sorted based on the number of matches, descending. The user can override the sorting using the up/down pointing triangles in the header of each column.
- To improve performance, the number of categories shown at once is limited. For reports that have large numbers of categories, there will be multiple "pages" available via the pagination controls in the lower right corner.
- The CSV link at the top right of the report allows the report contents to be exported to a CSV format.

Summary Report Columns

- **Category**
  - Each match in a report will be categorized into a single *mutually exclusive* named "bucket".
  - The matches in one category can never appear in another category for the same report.
  - The total number of categories is in parenthesis in the header of the column (e.g. 78 in this example).
  - If you want to quickly jump to a specific category name, case-insensitive filtering is available.
  - The tooltip for the header of this column will show the TRPL expression being used as the condition match for the report's results.
- **Conversion Support Level/Runtime Support Level**
  - Optional column describes the level of support in FWD for converting/running the associated 4GL language feature.
  - If this column does not exist, it means that the AST nodes being matched in this report do not yet have gap analysis marking rules.
- **Matches** - The number of AST node matches for this category.
- **%** - The percentage column just to the right of the Matches column is the category's matches / total matches expressed as a percentage.
- **Files** - The number of files that have AST node matches for this category.
- **%** - The percentage column just to the right of the Files column is the category's files with matches / total files with matches expressed as a percentage.

Details Report



Details Report

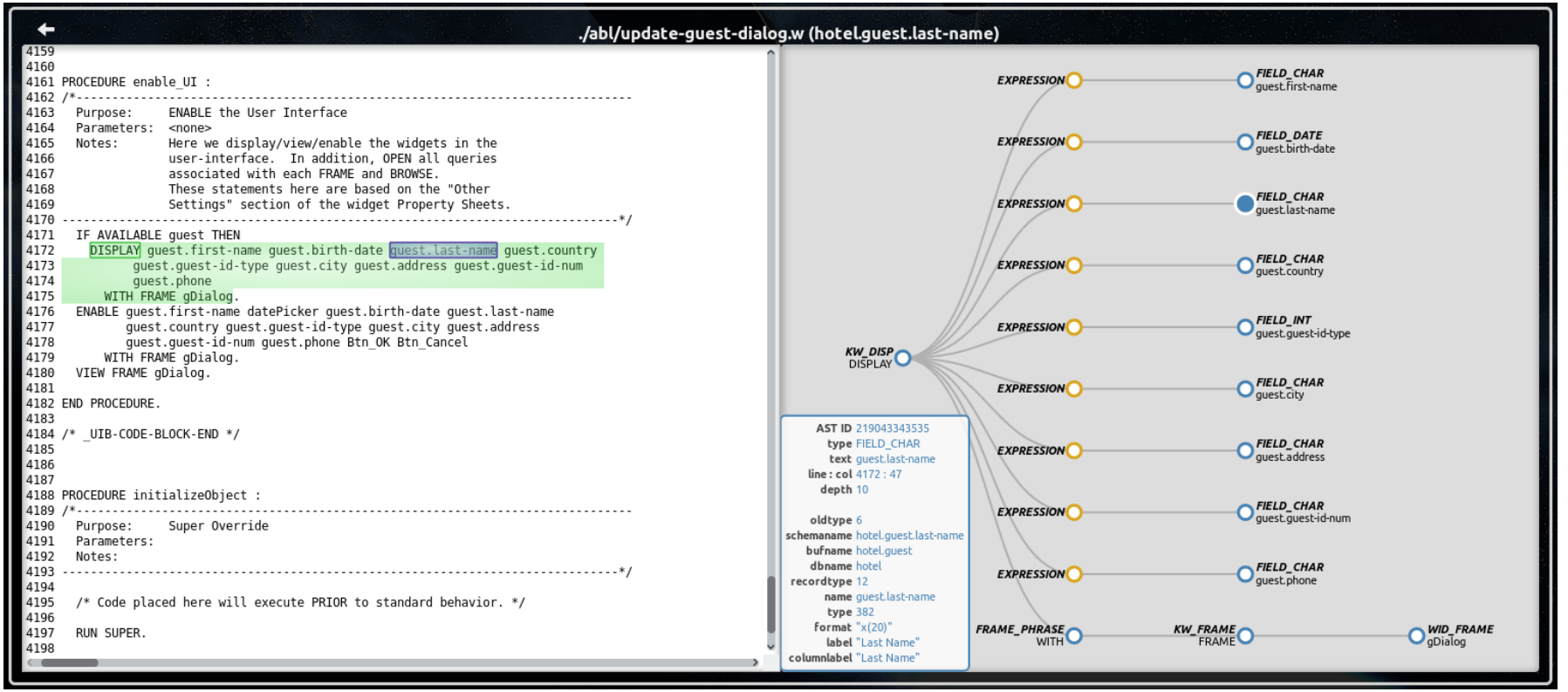
- The Details Report will be loaded by clicking on the Category row of the Summary Report. When the Details Report loads, it replaces the content table of the "parent" Summary Report. It is loaded into that same space formerly occupied by the Summary Report contents.
- Notice that the parent Summary Report's title remains visible at the top, but the Details Report has been loaded where the Summary Report table was previously displayed.
- The Details Report has a title of its own that is the category name from the Summary Report (e.g. "run [KW_RUN]") followed by the number of matches in the category (e.g. there are 1,316 RUN statements in the project) and the number of files that contained matches (e.g. 38 files had RUN statements).
- Each match in the Details Report is a single row. That means that a row is specific to a specific AST node location in a single file (4GL code or schema).
- Match Text - Formatted text showing some useful information about the match. The content of this will vary by report. Some may be simple text from the source files and others will show a text rendering of a node or subtree from the AST.
- Hovering the mouse pointer over the Match Text cell of a specific row will show a tooltip with the a potentially larger, multi-line version of the Match Text.
- **Click on a row to display the Source/AST View for the specific match being referenced.** The Source/AST View is a tool for reviewing and exploring the preprocessed source code (or schema) text and the associated AST.
- The CSV link at the top right of the report allows the report contents to be exported to a CSV format.

Support Levels

Support Level	Conversion or Runtime	Description
Unknown	Both	The support level is not known.
None	Both	There is no support for this feature.
Stubs	Runtime Only	Runtime functionality stubbed out, but not implemented. This means that there are placeholder classes and methods in the Java code, sufficient for any converted code to compile. BUT these classes and methods have no real implementation inside, they are just there to satisfy the javac compiler.
Untested	Runtime Only	A runtime implementation exists, but it needs testing. It is likely there are issues to resolve to achieve compatibility.
Partial	Both	Partial runtime support is implemented but there is at least some implementation missing.
Basic	Both	A basic implementation is available but there are limitations which require more work before having a complete implementation. This is typically a statement of more compatibility testing being needed.
Full (Restricted)	Both	Full support is implemented, but there are permanent restrictions. This is typlically due to some feature that does not make sense or is not needed in Java.
Full	Both	Full support is implemented. It is expected to be fully compatible.

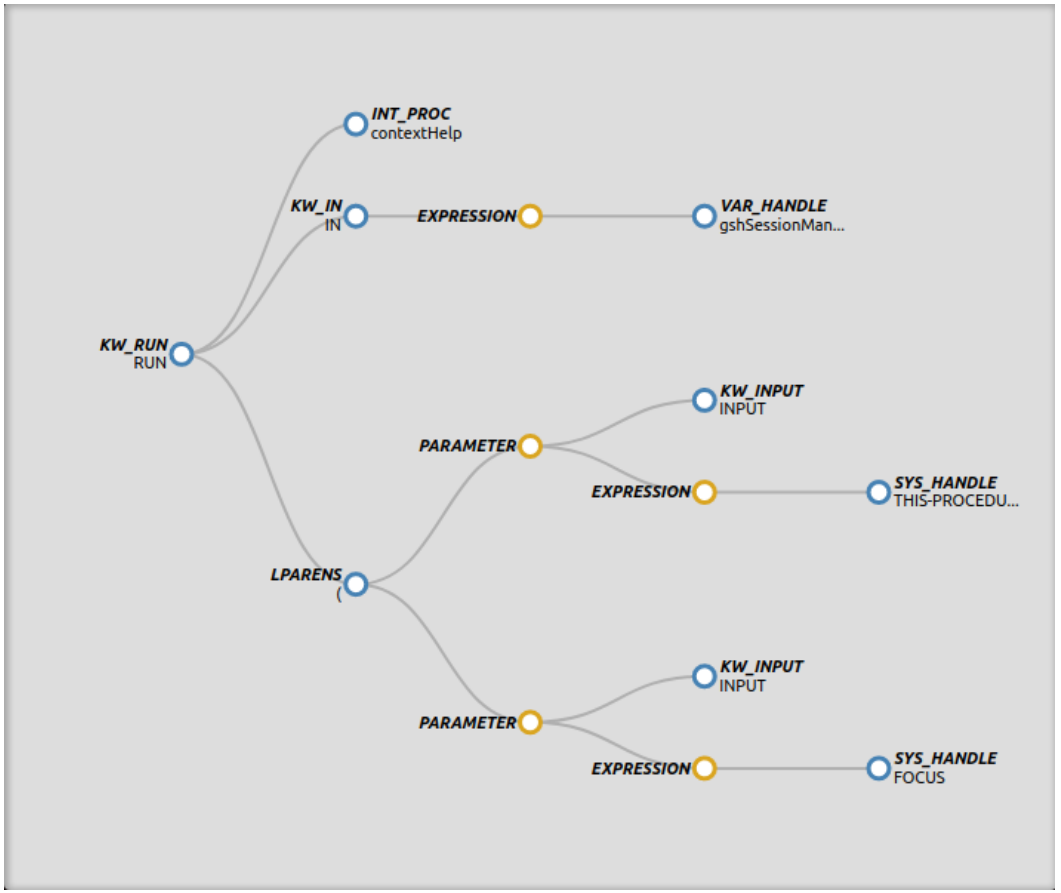
Source/AST View

- Fully preprocessed file on left with the match selected in pink.
- Current selection in the AST on the right.
- Source and AST views are linked, a selection on either side is highlighted and made visible on the other side.
- Hover mouse over an AST node to get details.
- Shift-click on the “root” node of the subtree to traverse up the tree.
- Ctrl-click on a child node to traverse down the tree.
- Zoom (mouse wheel scrolling) and Pan (left mouse dragging) the AST pane.



AST Pane

- When first entered, the AST Pane will display a sub-tree that corresponds with the magenta colored initial selection in the Code Pane.
- Subsequent left-clicks in the Code Pane may change the green colored current selection which will lead to the AST Pane displaying a different sub-tree.
- Consider this code: `RUN contextHelp IN gshSessionManager (INPUT THIS-PROCEDURE, INPUT FOCUS).`



- This is just one line of code of hundreds or thousands in a given file.
- When the entire program is parsed into AST form, there will be a branch of the tree that is created from this line of code.
- The branch is considered a sub-tree that is rooted at an AST node associated with the text RUN.
- The rest of the language statement is represented as child nodes, grand-child nodes as so forth as needed to reflect the structure of the statement.
- Each AST node is displayed as a circle. Blue are real AST nodes that correspond with text in the source code. Yellow are “artificial nodes” which only exist to structure the tree.
- Parent/child relationships are represented with a black line connecting the parent (left side node) to the child (right side node).

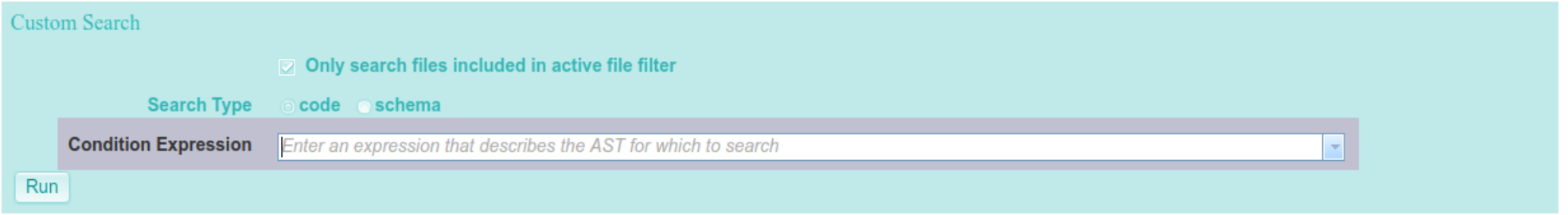
Search

Syntax-Aware Search

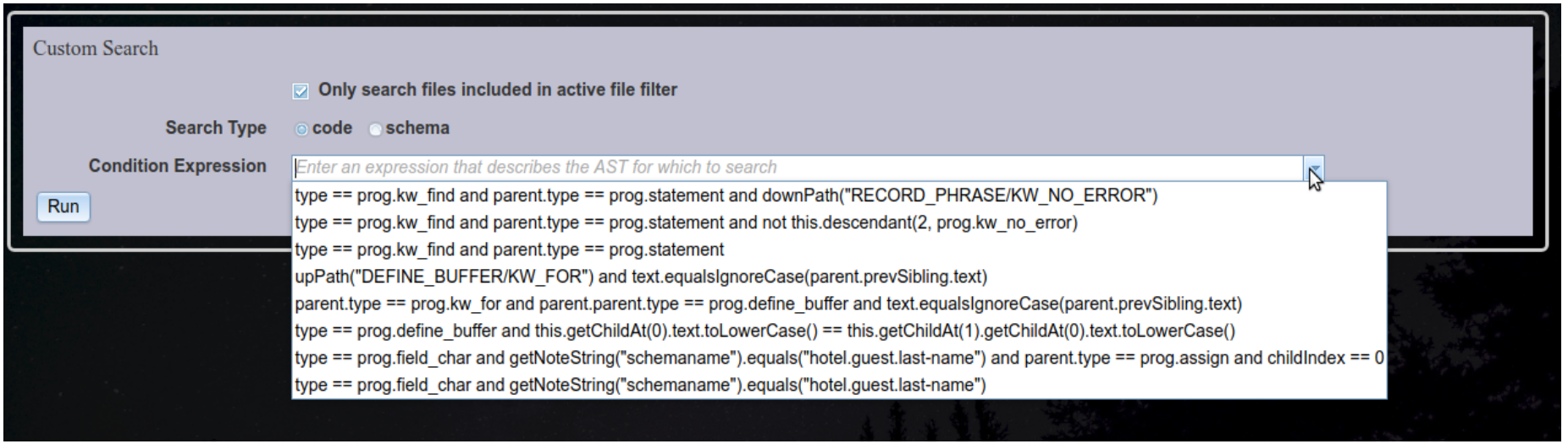
- If grep (regex searching) was fully aware of 4GL syntax it would still not be as good as this.
- Write expressions or arbitrary complexity that match based on the full richness of the AST.
- The TRPL engine does the tree walk, you just specify exactly what you want to match.
- The TRPL expression syntax has many features that make it easier to process AST concepts, including the knowledge of the current AST node being visited.
- Code that cannot be implemented in a single expression can be put into a callable TRPL function and accessed from expressions.
- All AST nodes and other data being accessed are actually Java objects. You can call Java instance methods (no statics or generics at this time) on these objects and you can pass those same objects to Java methods or to TRPL functions.
- TRPL has a wide range of advanced AST processing features that can be leveraged.

Condition Expression

- There is a text entry field for typing the **boolean** TRPL expression that specifies the search condition.

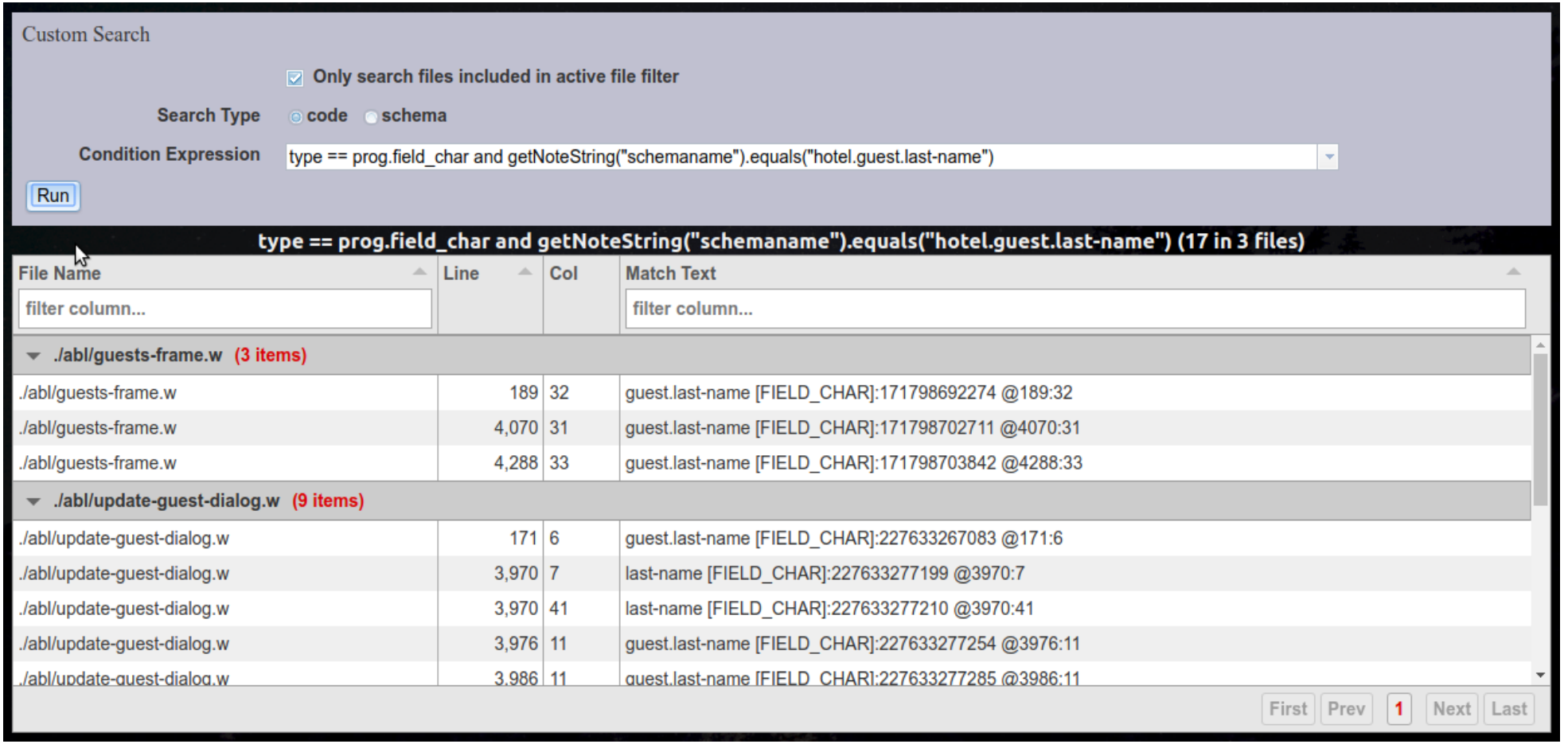


- The user can type directly into this field. As an alternative, pressing the arrow down button on the right will show the entry field's drop down list. This list shows any previous expressions that ran successfully. This is shared between all users and will be persisted across report server restarts.



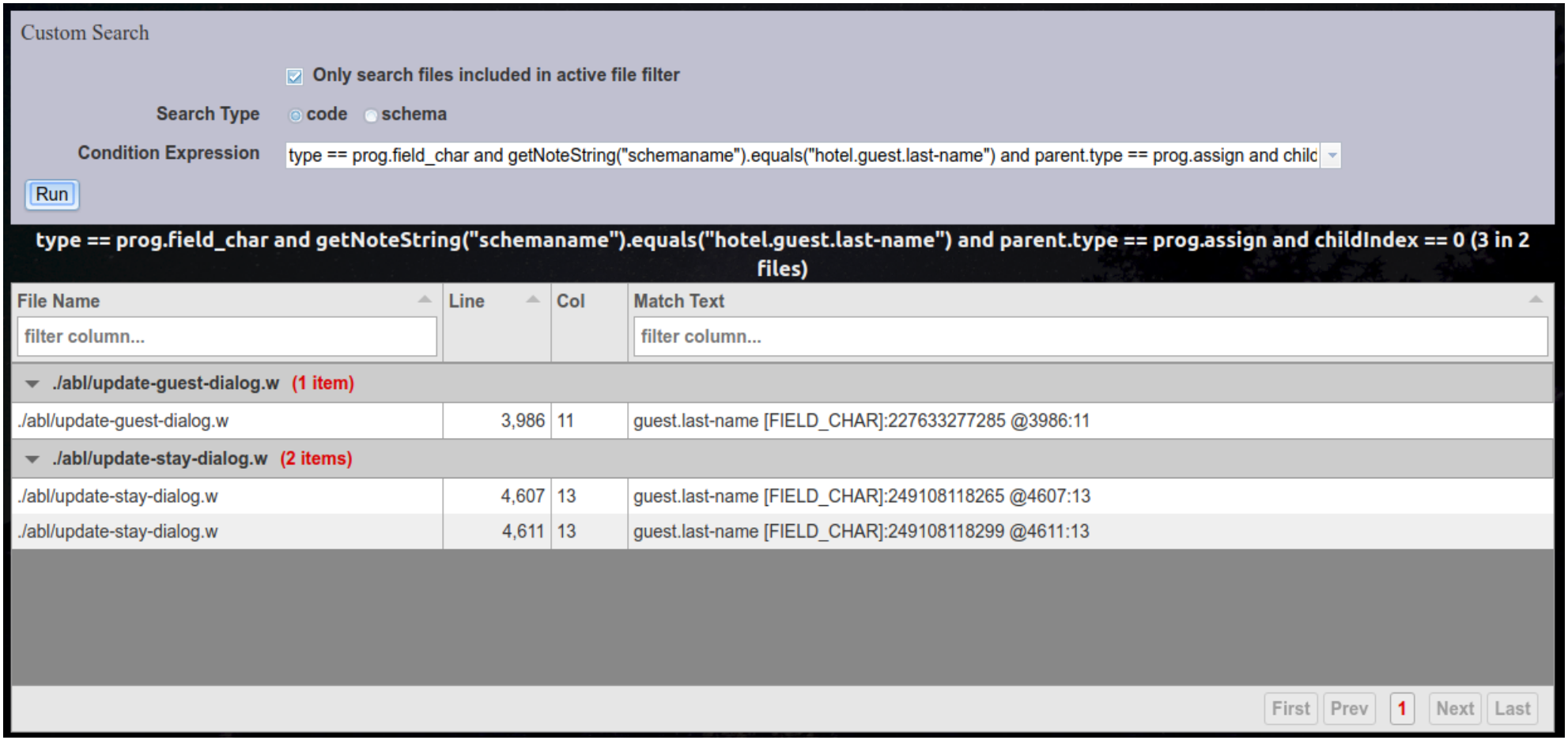
Search Results

- The following is an example of how to search for all references to the `hotel.guest.last-name` field.
- After typing this expression, press the Run button to execute the search. The report server will dynamically search the Code ASTs or the Schema ASTs for matches to this expression. All matches will be listed in the results at the bottom of the screen.



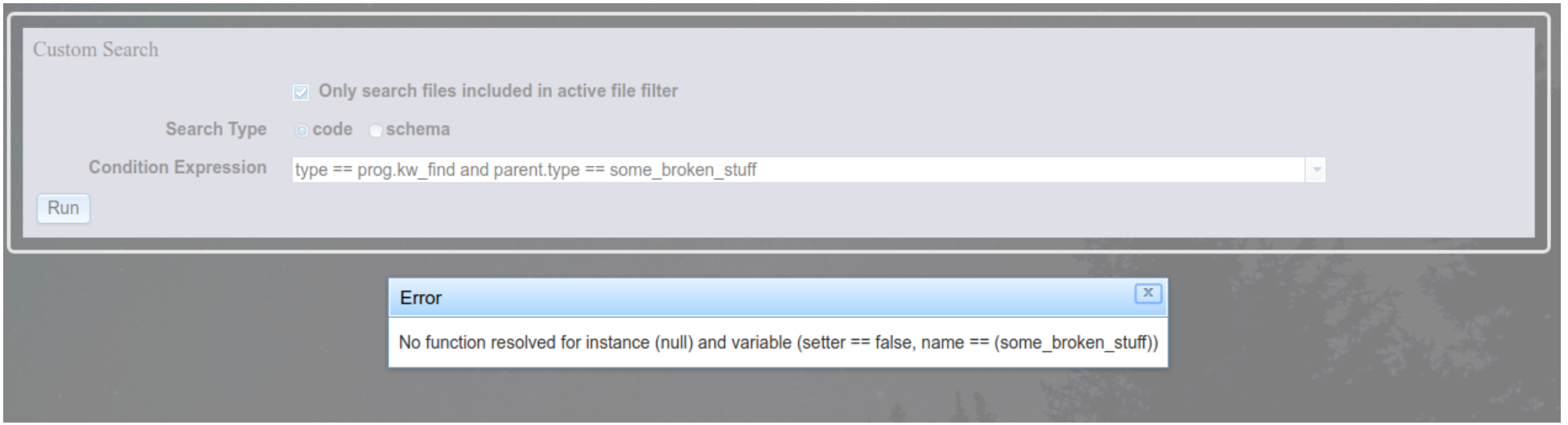
Condition Editing

- The search condition field can be edited and enhanced in an iterative manner.
- Pressing Run will execute the search using the latest condition expression and will load the results into the screen.



Errors

- If the expression is invalid (does not parse or pass validation) or if it results in a runtime error, the user will be notified:



- Dismiss the error notification and edit the expression to resolve the error.
- It should be noted that at this time, the error reporting is a bit "raw", as the error message displayed in the notification is taken directly from the TRPL engine when it parses, compiles, or applies a search expression.
- These messages originally were designed for TRPL programmers and are not necessarily user friendly.
- The message `No matches found for the given criteria` is not an error. It indicates that your search expression was valid, but the condition it represented simply did not match any of the ASTs searched.

Search: Field References

All references to guest.last-name:

```
type == prog.field_char and
getNoteString("schemaname").equals("hotel.guest.last-
name")
```

Assignments to guest.last-name:

```
type == prog.field_char and
getNoteString("schemaname").equals("hotel.guest.last-
name") and parent.type == prog.assign and childIndex == 0
```

Search: Buffers That Hide Buffers

Version 1:

```
type == prog.define_buffer and
this.getChildAt(0).text.toLowerCase() ==
this.getChildAt(1).getChildAt(0).text.toLowerCase()
```

Version 2:

```
parent.type == prog.kw_for and parent.parent.type ==
prog.define_buffer and
text.equalsIgnoreCase(parent.prevSibling.text)
```

Version 3:

```
upPath("DEFINE_BUFFER/KW_FOR") and
text.equalsIgnoreCase(parent.prevSibling.text)
```

Search: FIND and NO-ERROR

- All FIND statements (70 matches):

```
type == prog.kw_find
```

- FIND statements **without** NO-ERROR (26 matches)

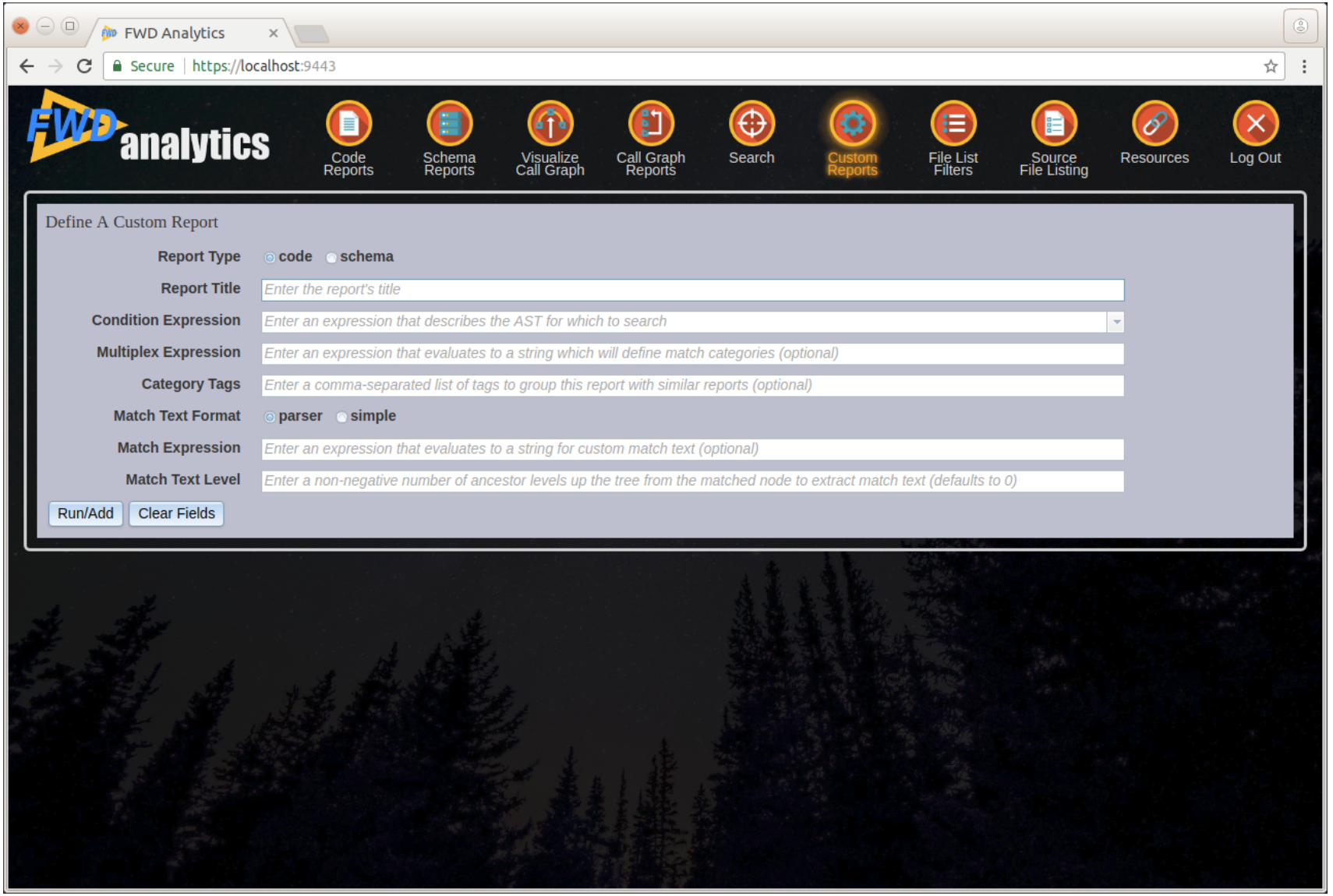
```
type == prog.kw_find and not
this.descendant(2, prog.kw_no_error)
```

- FIND statements **with** NO-ERROR (44 matches)

```
type == prog.kw_find and
downPath("RECORD_PHRASE/KW_NO_ERROR")
```

Custom Reports

Report Definition User Interface



Custom Reports

- Practice first with Custom Search, this is used to define the **Condition Expression**.
- Refine output with Custom Reports
  - **Multiplex Expression** to define “buckets”
  - Specify “details” text using the **Match Text Format** or **Match Text Expression**
  - Organize by **Category Tag** and **Report Title**
- Persist the report definitions you find useful.
- Planned: Edit and Delete of custom reports.

Custom Reports Example

- Report Title:

  FIND without NO-ERROR (by Buffer Name)

- Condition Expression:

```
type == prog.kw_find and parent.type == prog.statement
and not this.descendant(2, prog.kw_no_error)
```

- Multiplex Expression:

```
this.getImmediateChild(prog.record_phrase,
null).getChildAt(0).getAnnotation("schemaname")
```

- Category:

  Database

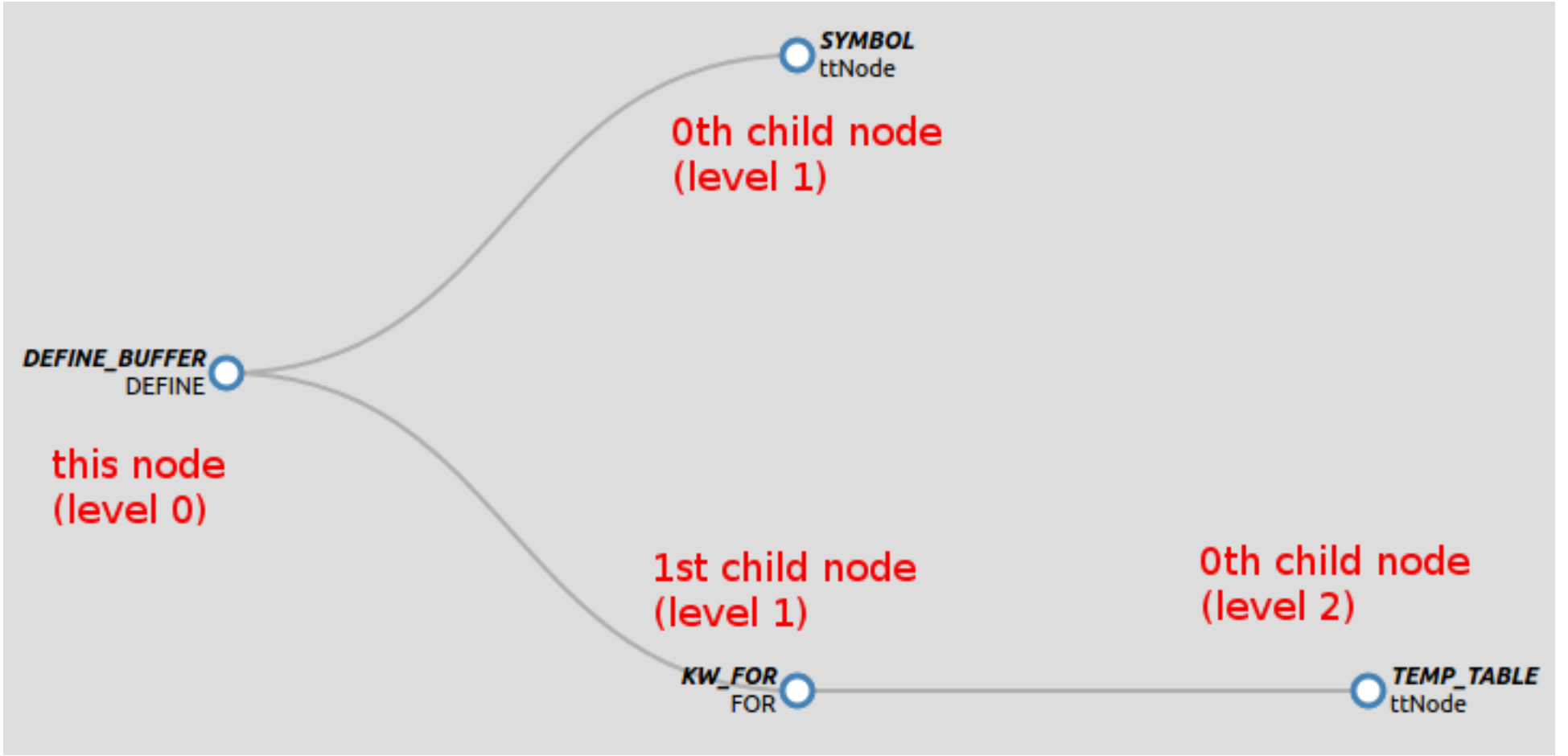
Usage Tips

Writing a Search Expression

- 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 in the source/AST view.
    - Use the predefined reports to find locations that already exist.
- 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.

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?

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 `ant report_server` target.
- Start the report server.
- Access the server at port 9443 via a browser.
- Full details of this process and all documentation: **https://proj.goldencode.com/projects/p2j/wiki/Code_Analytics**

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