A Code inspection tool for debugging Autumn's Context-Sensitive Grammars

Gerard Nicolas

A thesis submitted in partial fulfillment for the degree of Master in Computer Sciences

in the

Faculty EPL
University UCL

Supervisors:
Prof. Kim Mens,
Phd. student Nicolas Laurent

August 2017

Abstract

The debuging process is a crutial part of the lifetime of any projects. Unfortunately general purpose debuggers cannot provide the level of abstraction needed to reason efficiently on grammar developpement related errors.

We introduce a debugging tool to Autumn context-sensitive grammar. The goal is to provide the developpers with a tool that expose high level abstractions that represents the structure he reason about more closely, allowing him to track errors and resolve them more easily.

Acknowledgements

I would like to thank my mentor Kim Mens and PHD student Nicolas Laurent for their support and understanding.

Contents

1

Introduction

This paper will present a *debugging tool* developped to help troubleshooting the design of Autumn grammars. First of all, I'd like to stress the fact that we are discussing a debugging tool as opposed to a *"debugger"* strictly speaking. The reader might expect a debugger to be able to follow the flow of excecution in real time and to be able to stop it at will. Instead, our tool explore another approach: it first let the execution of the parse run completely, gathering all the relevant information needed to reason about the grammar down the road. In that quality, it is more like an code anylizer rather than a debugger stricly speaking.

Notherless, the object of this tool is to help debug a grammar, therefore, we will simply refer to our debugging tool with the word *"debugger"* for the rest of this paper. Please take note of this nomenclature specificity.

1.1 Motivation

During the lifetime of any project, the chunk dedicated to debugging is usually important and represent a crucial step in the developpement of a solution. Traditionnally, debuggers have given the developpers access to the running systems and general mechanism that expose the raw system state. Although this solution gives a universal framework to debug any project, the fact is developpers think and reason in term of high level abstractions while the debuggers only expose raw data which force the developpers to mentally refine their high level questions into low level ones making the debugging session an unnecessarily difficult and error prone endeavor which in term can increase the developpement duration and cost [20].

When applied to our domain of parsing and language recognition, this problem becomes all the more obvious. For example, consider the parse of an input, one might be interested to analyze the behavior of the parser past a certain position in the input stream. Since general purpose debuggers have no knowledge about parsing or input streams, the developper would have to manually single step through the entire execution until we reach the affordmentionned input position.

Moreover, writting grammars is a tricky and error probed business. Errors could come from a mistake in the definition of the grammar or from a mistake in the input stream or even from a mistake in some user defined parser. The highly recursive nature of grammars makes it so that errors reported by the general purpose debugger are rarely useful. Indeed, the debugger having no knowledge about those higher level abstractions cannot inform the developpers with clear information to reason with.

Other debugging solution has been developped for other parsing library, what makes our solution different is the simplicity of our approach combined with the power of Autumn's parsing library to express true context sensitivity.

Nowadays, most modern programming languages exhibit context sensitivity features[1], for example, python exhibits context sensitivity through significant whitespace. However, true context-sensitivity has rarely handled by parsing libraries. The reason lies with the difficulties to express context sensitivity with current grammar formalisms. As a result most solutions lack context transparancy (Grammatical construct needs to be unaware of the context shared between its ancestors and its descendants) and instead intertwin context and grammar making it difficult to maintain or eaven reason about.

In the other hand, Autumn is a context sensitive parser combinator library that approach the issue of context transparancy by intoducing the **"principled stateful parsing"** discipline. It is based on a *stateful parsing* approach in which the context is passed implicitely through a parse wide mutable state. This approach introduce its own set of problems that Autumn solve by introducing a set a primitive operations designed to formally interact with the parse state.

We propose a code inspection tool for Autumn's grammar designed as an Intellij plugin user interface that allow the exposure of higher level concepts relevant to parsing theory such as syntax tree inspector in all simplicity. With this tool, the developper can test grammar rules and track the invocation sequence of the parsers helping him to detect errors accross the project.

To convince ourselves, I will refer to section 5.2 in which we present several case studies highlighting the possibilities of our solution in comparison to general purpose debuggers.

Our goal is to simplify the life of Autumn grammar developpers. Just as developers use IDEs to dramatically improve their productivity, programmers need a sophisticated development environment for building, understanding, and debugging grammars.

[1]TODO: add ref

2

Background Material

This chapter will review some important theoritical topics necessary to fully appreciate the work presented in this paper.

2.1 Parsing expression grammar

Parsing expression grammar, or "PEG" is a formalism designed by `B.Ford` [15] to describe machine-oriented syntax. To give a proper overview of parsing expression grammars, it is necessary to come back on the notion of `N.Chomsky`'s Context-free grammar (CFG [16]) as they are very closely related.

2.1.1 Context Free Grammar

A *context-free gramma*r is described by a set of production rules and terminal symbols. A production rule is defined by a sequence of *terminal symbols*. The left hand side of the production rule is usually refered to as a *non-terminal symbol* Such rules describe the set of all possible strings of the language defined by the grammar.

More formally :

$G = (V, \Sigma, R, S)$

with V, a finite set of non-terminal character or variable.

Σ, a finite set of terminals.

R, a finite relation from $V \to (V \cup \Sigma)^*$

S, the starting symbol

Originally made to model natural (human) languages, the ability to express ambiguity was a crucial point to their original purpose that increased its expressive power. Ambiguity, however, is the very reason why CFGs makes it so difficult do express and parse machine-oriented languages. Such languages are intended to be precise and unambiguous by definition.

2.1.2 PEG

Parsing Expression Grammars provide a better framework for describing machine-oriented langages by avoiding ambiguity altogether. PEG is very similar to context-free grammars in the sense that it can be considered an extention of it.

The most important difference between the two is that production rules in CFG features non-deterministic choice between alternatives while PEG introduce prioritized choice order. A consequence of that is PEG is unambiguous by construction, each rule can indeed produce but a single outcome, therefore, a successful parse can result in only one valid parse tree. On the flip side of the coin, PEG users have to deal with "language hiding". Consider the simple choice rule $a|aa$, in the PEG formalism, we will always match the left side a and never the right side aa. Trying to parse the input "aa" with this rule will fail.

Formally, a parsing expression grammar consists of:

- A finite set N of nonterminal symbols.
- A finite set Σ of terminal symbols that is disjoint from N
- A finite set R of parsing rules
- A starting expression eS

Each parsing rule in P has the form A ← e, where A is a nonterminal symbol and e is a parsing expression.
A parsing expression is a hierarchical expression similar to a regular expression that can either be atomic or a combination of parsing expressions.

- An atomic parsing expression is either any terminal symbol, any nonterminal symbol, or the empty string.
- Given any existing parsing expressions, a new parsing expression can be constructed using the following operators: *Sequence, Ordered-choice, Zero-or-more, One-or-more, Optional, And-predicate, Not-predicate*

2.1.3 Parsing

As `Terence Parr` [13] put it, parsing can be explained using the maze metaphore. Imagine a maze with a single entrance and single exit that has words written on the floor. Every path from entrance to exit generates a sentence by "saying" the words in sequence. In a sense, the maze is analogous to a grammar that defines a language.

You can also think of a maze as a sentence recognizer. Given a sentence, you can match its words in sequence with the words along the floor. Any sentence that successfully guides you to the exit is a valid sentence in the language defined by the maze.

At almost every word, the recognizer must make a decision about the interpretation of a phrase or subphrase. Sometimes these decisions are very complicated. For example, some decisions require information about previous decision choices or even future choices. Most of the time, however, decisions need just a little bit of lookahead information.

2.2 Autumn Parsing Library

Autumn is a context sensistive parsing library.

- Autumn is different from other parsing lib because it deals with true context sensitivity.
- context-sensitive in parsing = tricky (context transparency) however lots of mainstream languages exhibit context sensitive features.
- most grammar formalism lack context transparency, they handle context sensitivity with ad-hoc code outside of the scope of parsing theory.
- custom memoization and error han- dling strategies.

Autumn's parsing library is based on Grammar parsing expression,

2.2.1 Context sensitivity

- grammar needs to remember what was previously matched to make a decision
- Autumn uses a parse wide state to remember context
- problem with stateful parsing : context transparency : Sometimes parse has to backtrack (explain here what backtracking is) and therefore need to undo changes made to the state.
- One might think that it can be easily done with a simple construct that undo the change on failure
- tricky part is : some parsers might be success full but because one of its ancestor failed, its changes has to be reverted anyway.
- Autumn's approach to deal with context transparancy : principled stateful parsing
- Principled stateful parsing define a set of primitive state manipulation operations that allows us to get an image of the parse state at a specific time and allow us to revert the state to that of the image. (consequence : we can backtrack safely)

A grammatical construct is context-transparent if it is unaware of the context shared between its ancestors and its descendants.

Stateful parsing is not enough to deal with context sensitivity as it is not context-transparent. (We need to make sure that parsers combinator doesnt backtrack)

2.2.2 Principled Stateful Parsing

Parse State

- state = passing context around implicitly
- if execution = linear, reading/writting to this state would suffice.
- althought because of backtracking (speculative execution) : we need to reverse the state.

Parsers

A parser represents a computation over the parse state that either succeeds or fails and has side effects on the parse state.

Primitive Operations

To deal with backtracking we need operations that can take a picture of the state at a particular time and restore it. 4 operations are used to manipulate states:

- snapshot : capture of the state at a specific point during the execution.
- restore : The restore operation takes a snapshot as input and returns a transformation that brings the state to that described by the snapshot.
- diff : The diff operation returns a DELTA object representing the difference between a snapshot and the current state
- merge : The merge operation takes a delta as input and returns a transformation that appends this delta to the input state.

2.3 Implementation

Explain how Autumn work from a high level "intuitive" level.

Highlight the main classes and explain how they work together. Maybe a UML chart ?

2.3.1 Important classes

Parsers Parsers are function that return boolean, their implementation can be found in the parsers package and are implemented as extension functions of the grammar class

Grammar Main class: Explain how the main parsing algo works. And that every structures(maybe explain what are those structure and their purpose) are defined here.

State/side effects/Undo structures Explain how state is handled through undo structures, present the basic undo structures that are implemented and explain that the users can create custom structures for his needs.

3

Overview of the solution

This chapter presents the main functionality of the debugging tool in all generalities and presents how it has been integrated with Intellij IDE as a GUI plugin. At this point I'd like to remind the reader that our solution is not a *debuger* in the traditional meaning of the word. Dispite this fact, we will refer to it as a debuger for simplicity.

3.1 Overview

To empower Autumn's parsing library with a proper debugging tool, we developped an IDE plugin designed to be a straple for the developper to interact with the grammar's output and track down errors. During a debugging session, Autumn will attempt to parse the entire input. Weather it is successful or not, it produces a parse tree during its invocation.

The generated parse tree is then passed to the plugin's GUI. Each node of the tree represents the invocation of a single parser and contains information about the circumstances surrounding its invocation that can be used to assess the correctness of its behavior

The tree itself can be seen as a list representing the history of invocation, the first entry being the starting expression. Each entries representing a moment in time during the parse, one can retrace the execution of the entire parse, stepping forward or backward simply by navigating up and down the list.

The debugging effort can therefore be summurized as a search in a tree of event.

3.1.1 Why not a "normal" debuger ?

From the start, we wanted to create a plugin that would serve as a GUI for our debugger. The original idea was to create a more conventional debuger by overlaying our logic on top of Intellij's general purpose debugger to do achieve a more traditional approach allowing us to step live into the execution. The problem is that Intellij doesn't expose the implementation of its debugger through the plugin interface, making it a much more difficult endeavor. To access the IDE's debugger, one could work directly with the community version of the IDE which is open source, but we decided to stick with our plugin interface.

Facing this problem made us think the solution over in term of practical use. We tried to put ourselves in the shoes of grammar developpers and asked ourselves questions like:

- What problem will I face while developping a grammar ?
- What information do I need to track errors down and take decisions ?
- How can I get those informations in the simplest possible way ?

3.1.2 Difficulties of grammar development

The most common problem encountered by grammar developpers is to determine why a generated parser incorrectly interprets an input sentence. Generally, an incorrect parse can be reduced to three different cases:

- The grammar contains a certain number of wrong production rules that leads to a wrong interpretation of the input.
- The input itself contains incorrect parts with respect to the specification of the grammar which in turns lead to a wrong interpretation of the input.
- There is a mistake in the definition of some user custom parser.

TODO Having some examples of bad error reporting and highliting the difficulties of grammar debugging here might be a good place

To deal with those issues, the most important properties that we need are :

- to able to expose chronologically call sequence of each parser.
- the ability to manipulate the input stream and identify what portion of the input a particular parser has matched. Additionally to be able to analyze the parsing behavior at a certain position in the input is also very valuable.
- lastly, to be able to inspect the condition of the general parse state during the call of a particular parser.

As we discussed before, general purposed debugger fall short of dealing with those particular points. Indeed, general purpose debuggers don't have any knowledge of input stream manipulation or parse state, therefore the only way to gather those information using a general purposed debugger is to manually step into the code and monitor mentally the mutation of some low level data structures which is far from convenient.

We wanted to be able to observe the flow of execution of the parse at a higher level of abstraction than instruction by instruction. We wanted to be able to travel through time at the parser level, being able to see which parser was called in which order, being able to see where a parse failed and to be able to go back in time to analyze if the condition for it to succeed was theorically met or if the problem came from elsewhere. We wanted to be able to jump immediately to a certain point in the input stream and see how the parser matched a particular portion of it. And finally, we wanted to be able to analyze the parse state at any point during the execution of any parser.

The question was then, instead of enhancing the general debuger with the required abstractions, could we do that in another way ? Our alternate approach proved to be much simpler indeed. By building a parse tree that can be navigated, we meet the first requirement of analyzing the flow or sequence of invocation of the parsers as well as exposing the hierachy of the parsers.

Autumn's formalism can be specified as a "functional flow" where one can plug an arbitrary function that consults the input and push the parse forward or fail. The only information required to simulate input stream manipulation is the position in the input that the parser consulted. It is not unreasonable to reccord a single integer for each node of the tree, that leaves the lasst problem of being able to analyze the parse state at any point.

When manipulating the parse state, autumn keep a log of every mutation a parser applied to it. This structure is maintained primarely to revert the mutation a backtracking parser applied to the state. This structure serves our purpose as well, all we have to do is to record the size of the log for each parser. From this information, it is easy to regenerate the parse state as it were during the invocation of a specific parser.

Our alternate solution to traditional debugging is build on this idea, building a parse tree and recording the input position as well as the log size for each parser. Navigating this tree allow us to observe the flow of execution just as we were travelling back and forth through time at the parser's invocation.

3.2 Functionality

Syntax tree Originally, autumn doesn't build a syntax tree like it is generally done in other parsing libraries. Instead it directly generate a abstract syntaxt tree (AST) which contract nodes together to make them easier to read from a human stand-point. Such an AST doesn't represent the grammar as closely as a regular syntax tree as we no longer have a one to one relationship between the grammar rules and nodes of the tree.

Having a one to one relationship between the rules of the grammar and the node of the tree is a crutial property for our debugging purpose. Indeed, we navigate through the tree to observe the invocation of the parsers through time, so if a particular node is actually the abstraction of several children parser, it can be that one node doesn't correspond to one rule of the grammar but to several. To be able to tell which one contains errors would be made much more difficult and less relevant.

The debugger assumes the task of building the required syntax tree. Since both trees contain essentially the same information, we could easily have had the Autumn's logic build the syntax tree at the same time it does the AST to improve the performances and avoid uneccessary work. Nonetheless, we choose to seperate the debugging logic from autumn's implementation as best we could to minimize coupling between the debugger and Autumn. Autumn's logic doesn't need to build the syntax tree and so this effort should be undertaken only in the debugging context.

Gathering relevant information The syntax tree is build implicitely during the invocation of each parser. Every time a parser is called, a new node is create in the tree and some information about the context is recorded within the node. The information reccorded in the node are:

- the **name of the production rule** the parser is defining, if any. Each production rule being defined by a combination of parsers, some parser are just a part of the definition of such rule.
- the **type of the parser**. (i.e. Seq, Str, . . .)
- its *parent* and *children*
- and the **log size** to regenerate the parse state

3.3 Intellij plugin

The plugin we developped is essentially composed of a panel that can be docked on any side of the IDE window, although it is recommanded to dock in at the bottom or on floating on another screen for maximum clarity. Figure 3.1 shows how the plugin integrate within the IDE.

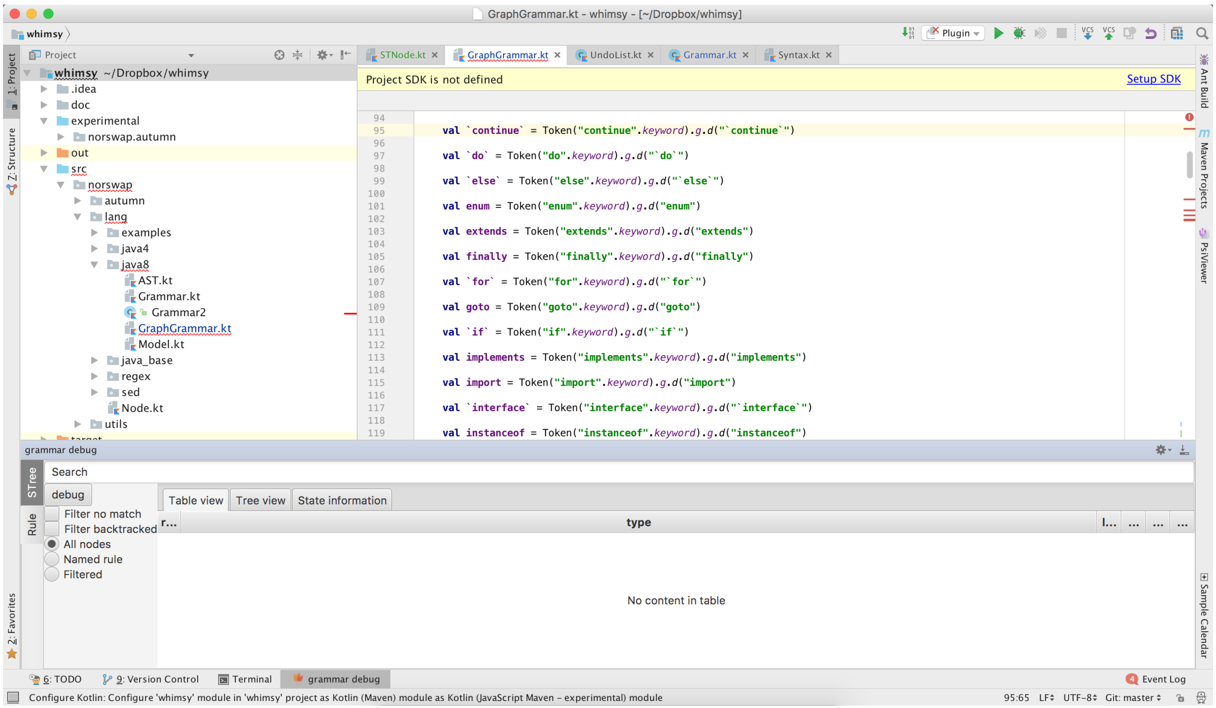


Figure 3.1: IDE GUI for Autumn's debuggin tool

The parse tree that has been build can be displayed in two different forms. A table form and a tree form, the former represents the execution trace, it displays each parser in the sequence they were called in a simple list. The later displays the same information as a tree, highlighting the hierachy between parsers.

It can be argued that the tree view would be enough and that having a table view would be redundant. While this is true, given the highly recursive nature of grammar rules, navigating the tree can quickly lead to very high level of nesting, while it makes the hierachy between parsers obvious it also makes it more difficult to have a clear image of the sequence of calls. On the other hand, the table view being a simple list makes it really easy to follow the sequence of call one by one, the downside being that it hides the parser hierachy. So even if it can argued that both views are redundant and unecessary, we think that in some cases, it can provide a small improvement in the quality of life of the developper which is what this tool is all about.

It can also be argued that this information as is wouldn't be very useful to the developper as it is quiet a dump of information. When trying to debug a grammar, most of the time we are interested in the behavior of a restricted number of rules that we suspect are carrier of mistakes. In traditional debugging, this is done by setting a serie of breakpoints and executing the code. Once the breakpoint occurs, the programmer will generally single step through the code instructions by instructions. We achieve this by introcing a serie of filters and search field. By filtering the information, one can really easily isolate the execution of a parser or rule he is interested to verify the correctness and behavior. From there he can easily navigate the tree up and down to see how the grammar parsed the input around those particular parsers and reason about them.

Note that the plugin can also run as a stand alone application as shown on figure 3.2, albeit some disabled functionnalities (those that are tightly coupled to the IDE, like jumping to the rule definition). The usefulness of such feature can be discussed, it could possibly serve as a starting point to develop a full fledged independent debugging environment for autumn grammars.

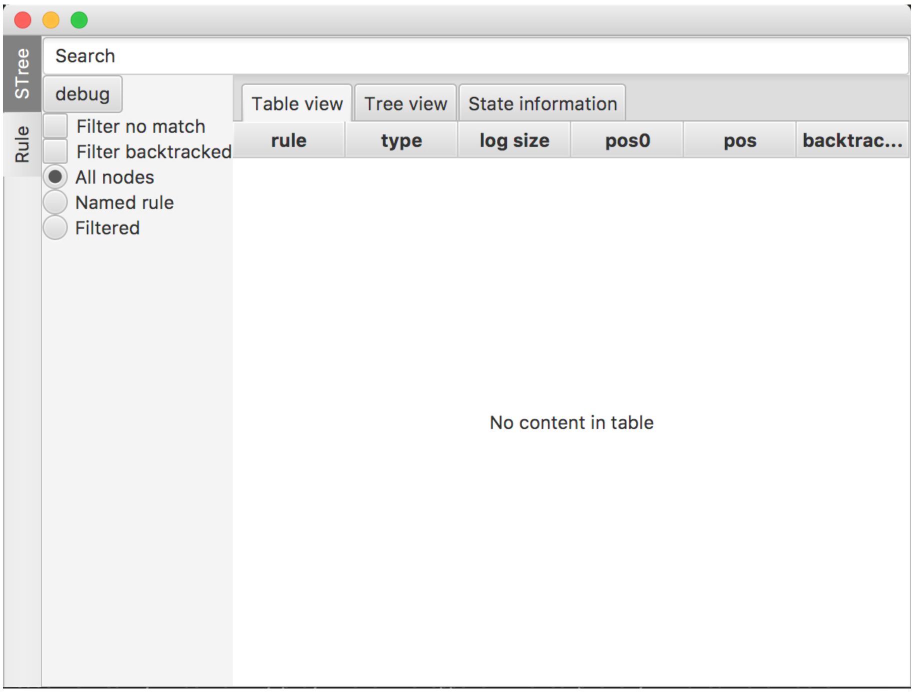


Figure 3.2: Stand alone view - the debugger GUI can be executed independetly from the IDE albeit some restrictions

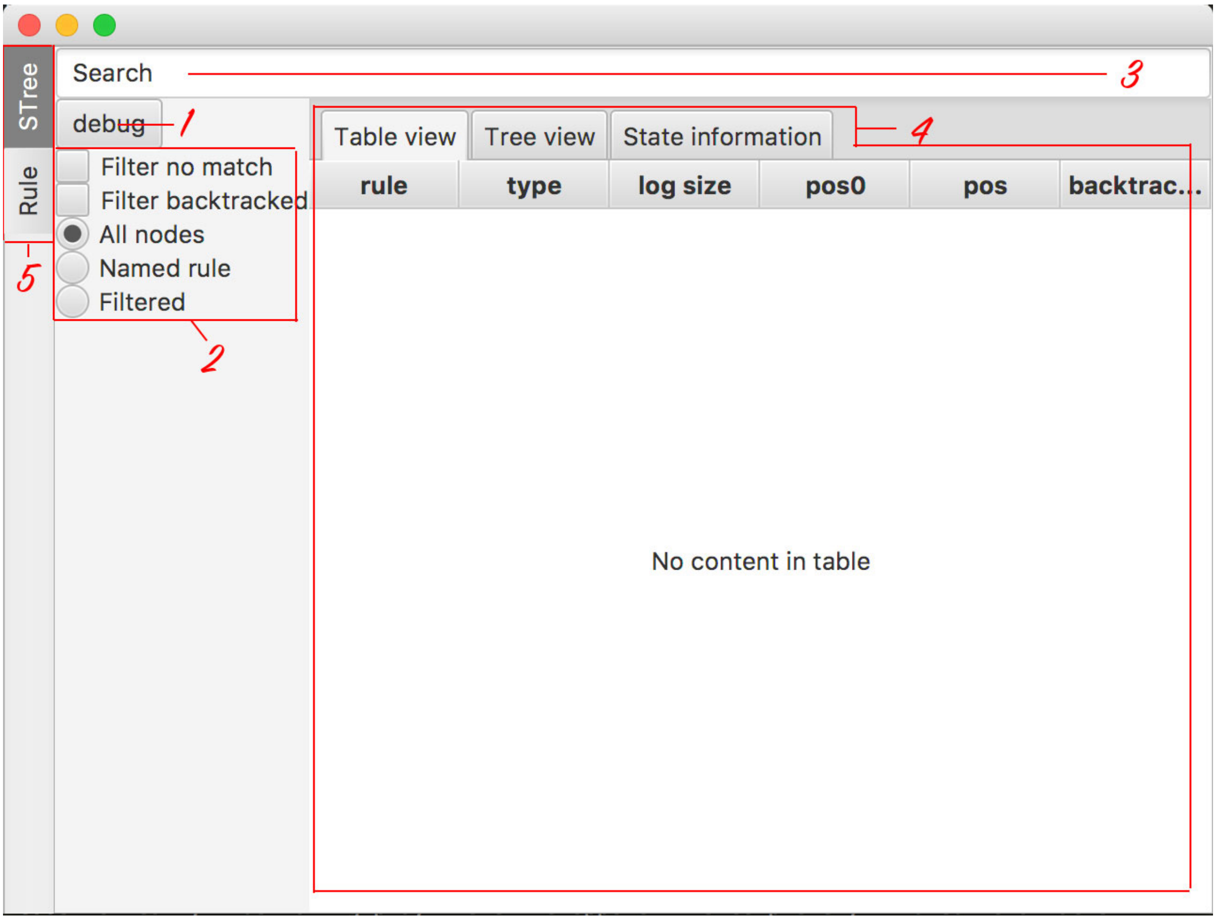


Figure 3.3: Close up view of the plugin in detail

3.3.1 Views

This section describe the content of the different views in a little more details.

Figure 3.3 presents the different elements of the GUI.

- 1. Debug button, used to start the debugging process
- 2. Filters buttons, used to apply filters to the displayed data
- 3. Search field, used to filter the parsers by name
- 4. Main information display, 3 tabs are available : Table view, Tree view and finally state view
- 5. Side tab panel, used to switch from the main view to the rule view.

Table view - Execution trace

An example of the table view displaying information of a small parse example can be seen on figure 3.4. As discussed before, the table representation of information allow for clearer display of the sequence of parser calls, it countains the parser calls in chronological order.

The table is divided in a serie column:

- Column rule lists the name of the production rule that the parse is defining
- Column type lists the actual parser type
- log size represents as its name indicate the size of the log, the number of mutation applied to the parse state so far.
- pos0 represents the position in the input before the parser was invoked while pos indicate the position of the input after the parser was executed.
- finally, backtracked indicates with a boolean property if the parser was successful or if it backtracked.

Table view | Tree view | State information

rule	type	log size	pos0	pos	backtracked
root	Build	0	0	1057	false
	Seq	1	0	1057	false
	ReferenceParser	2	0	0	false
whitespace	Repeat0	3	0	0	false
	Choice	4	0	0	true
	SpaceChar	5	0	0	true
line_comment	Seq	7	0	0	true
	Str	8	0	0	true
multi_comment	Seq	9	0	0	true
	Str	10	0	0	true
	Maybe	8	0	15	false
package_decl	Build	9	0	15	false
	Seq	10	0	15	false
annotations	Build	11	0	0	false
	Repeat0	12	0	0	false
annotation	Seq	13	0	0	true
\`@\`	Token	14	0	0	true
	Str	15	0	0	true
	Str	17	0	0	true
	Str	19	0	0	true
	JavaIden	21	0	7	false

Figure 3.4: Table view: Chronological parser call on a trivial parse example

Context menu In this view there is a single context menu called "show rule info". When clicked, it displays the complete definition of the grammar rule as well as highlighting the portion of the input that was matched during the parse.

Tree view - Syntax tree

The tree view displays the data as a tree like structure. It highlights the hierarchy between parsers making it obviously easy to trace where the invocation call came from. When first generated, the structure opens two level of nesting by default. The children of a particular node can be displayed by double clicking on

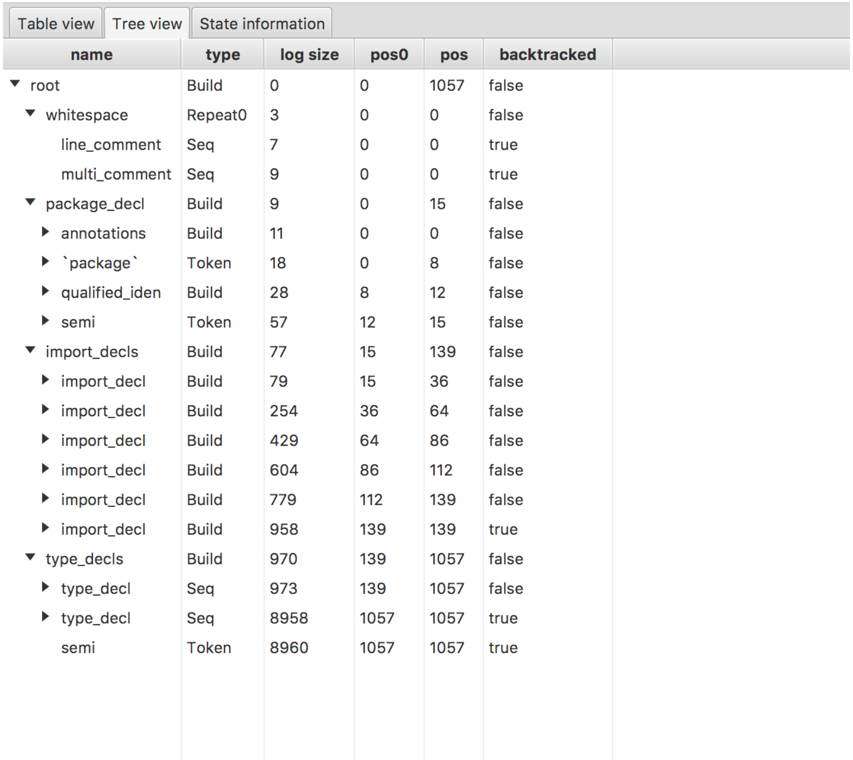


Table view | Tree view | State information

name	type	log size	pos0	pos	backtracked
root	Build	0	0	1057	false
whitespace	Repeat0	3	0	0	false
line_comment	Seq	7	0	0	true
multi_comment	Seq	9	0	0	true
package_decl	Build	9	0	15	false
annotations	Build	11	0	0	false
`package`	Token	18	0	8	false
qualified_iden	Build	28	8	12	false
semi	Token	57	12	15	false
import_decls	Build	77	15	139	false
import_decl	Build	79	15	36	false
import_decl	Build	254	36	64	false
import_decl	Build	429	64	86	false
import_decl	Build	604	86	112	false
import_decl	Build	779	112	139	false
import_decl	Build	958	139	139	true
type_decls	Build	970	139	1057	false
type_decl	Seq	973	139	1057	false
type_decl	Seq	8958	1057	1057	true
semi	Token	8960	1057	1057	true

Figure 3.5:

it, as a side effects, the column gets resized to accomodate the new nesting level. An example of this view can be seen on figure 3.5 for a simple parse example.

The context menu for this view is a little bit more complex and contains four different actions.

Context menu - Set as root Because of the reccursive nature of the parse tree, high level of nesting can quickly disrupt the clarity of the display. This option does exactly what it suggest it does, it sets the selected rule as the root of the tree, displaying only its children. It is useful when analyzing the parse of a particular rule deep inside the tree to isolate it from the rest of its siblings. The resulting view is a subset of the original tree, improving the clarity of the display by focusing on the elements one is reasoning about.

Context menu - Set parent as root This option works similarily to the previous one, it take the parent of the root and sets it as the new root. This is useful when one is tracking an error, one analyzes the parse of one rule, and after making sure of its correctness, one might go one step back and analyze the parse of it's parent.

Context menu - Reset root Finally, at any point, one might want to display the entire tree again. This option reset the root to the starting expression.

Context menu - Analyze entry The last context menu option is used to display the definition of the rule and highlight the matched input.

State information

The last tab of the main view's purpose is to display the parse state associated to the parser currently inspected. In the current implementation of Autumn, the parse state is simply used to record the position of the input and the abstract syntax tree built during the parse. Both information are already displayed in the other views, therefore, it doesn't display information for now. This tab remains relevant notherless because users can implement their own structures to store custom parse state providing they respect Autumn's prescription for such structure. Those custom structure could contain relevant information that has not been captured by the other views. The existance of this tab, and a hook that has been implemented in those structure can greatly facilitate the display of such custom information.

Rule side tab

The last section of the plugin we didn't discuss yet is the side tab displays. When one is analyzing a parser using the context menu of the other views, this panel's information is populated with the rule definition of the parser and the portion of input matched by it. Figure 3.6 shows an example of that. The top part of the panel, above the line, is the definition of the production rule linked to the parser. Below the line is the portion of the input matched by the parser.

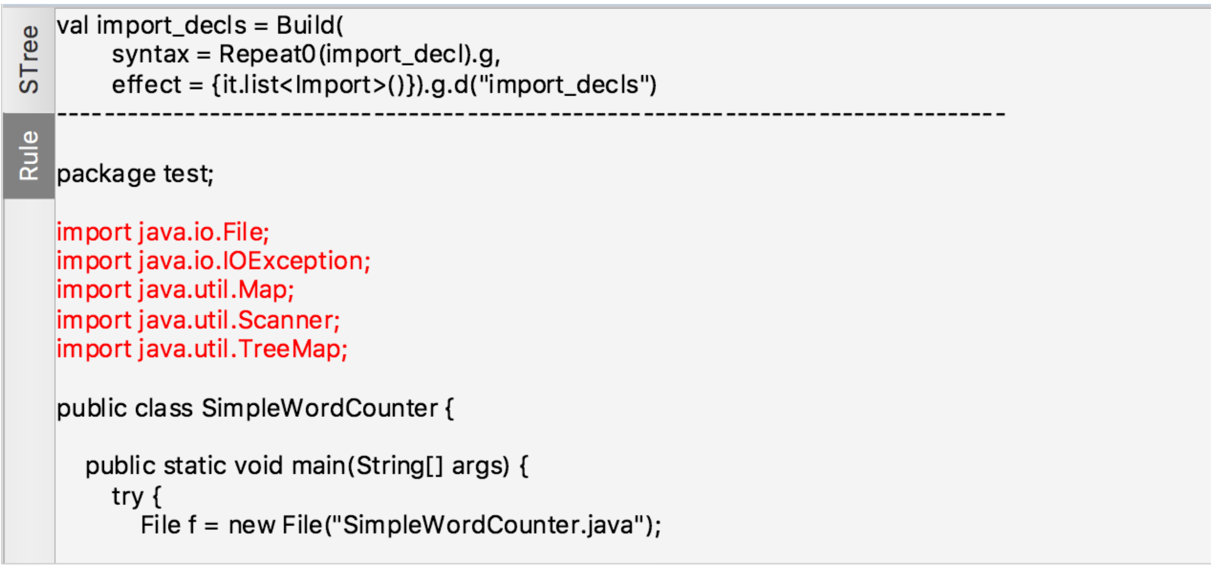


Figure 3.6: Example of grammar rule information and matched input for a particular parser

3.3.2 Filtering the informations

As we hinted before, the fact that the system display all the information at once can be quite overwhelming and difficult to use in practice. That is without considering the different filters that helps narrowing down the information to exactly the one we need to detect mistakes in our grammar.

The different filters buttons can be observed on the number 2 highlight on figure 3.3.

- “Filter no match”, due to the nature of parser definition borrowed from the PEG formalism, certain parsers can potentially return successfully although they didn’t advance the parse in any ways. An example would be the “repeat0” parser which return successfully uppon matching 0 or more times the content of its body. Such parser might get in the way of clarity when we analyze how the parse actually matched the input. These parser can be identified by the fact that the position before and after their invocation is the same. For this reason we can filter them out using this button.

- “Filter backtracked”, due to context sensitivity, some parsers might match the input partially and then backtracked upon failure. Similarily to successful parsers that didn’t match the input, depending on the situation, those entries can get in the way of understanding how the input was matched. It is although obviously important to keep those parsers around within the syntax tree to debug our solution. A particular rule we wrote could backtrack unexpectedly, we therefore might want to analyze why it failed to match the input.

- “All nodes”, as its name indicate, this radio button displays all the nodes without filtering them based on their name or type.

- “Named rule”, this radio button filter the tree to leave only the parsers that are the begining of a rule definition. Filtering named parsers is particularly useful as each nodes of the tree now match directly with the rules of the grammar, making it really easy to understand how the parse matched the input file.

- “Filtered”, Finally, this radio button works together with the search field marked as the number 3 on figure 3.3. One can type rule names or parser type name in the search field to filter out from the tree all the parsers that doesn’t match this definition. Switching to this radio button simply filter the tree with whatever’s in the search field. Intrinsicly, this radio button doesn’t do much except indicating that the display is driven by the search field, which overides the other name based filters.

3.3.3 Jump to code

A really interesting feature for our plugin is the ability to jump directly in the code for the definition of the grammar rule and/or the input file. Our implementation doesn’t allow us to do it in a neat way. Right now, we achieve this feature by selecting one rule from the table or tree view, then going to the “tool” menu and selection “jump to rule definition”. Alternatively, there is a keyboard shortcut to do this : command-alt-A then C.

The reason for this will be discussed in the chapter 4.

4

Implementation

This chapter will present the practical implementation of the solution.

In section 2.2, we presented the overall structure of the Autumn parsing libraries.

The debugging logic has been implemented using hooks attached to the parsers.

4.1 The debuging tool implementation

4.1.1 Overview of the structure

- model
  - builder
  - model compilers
    - model compiler
    - graph model compiler
- parsers interface, subparsers
- syntax tree STnode
- autumn
  - parse state
    - undoable list
    - undoable map
  - grammar

4.1.2 Hooking into the parser implementation

The debuging logic presented in the previous chapter has been implemented through the creation of a hook inside the parsers. During a debuging session, each parser invocation will instead call the debug function that will handle the debugging logic.

Originally, the parsers were implemented in a strictly functionnal fashion. Autumn defines a parser simply as a boolean function taking some input, optionally modifying the parse state, advancing the input position and finally returning a boolean indicating the success or failure of its execution.

To implement our hook, we changed the parsers from a functional implementation to an object oriented implemention by simply wrapping the parsers function inside objects. A general parser interface has been created to serve as an entry point for our hook.

We called this new implementation "naive parsers" because we suspected that it would cause some performances issues due to constant creation of new objects during the parse, this issue was referred to as megamorphic call sites. Interestingly, after comparing the benchmark, it has been shown that the impact on the execution time wasn't significant. The results of these benchmark is discussed in chapter 5

4.1.3 Rewritting the grammar, the notion of model and model Compiler

The changes made to the parsers had the consequence that we had to rewrite the grammar to reflect those modifications. As discussed before, writing grammar without proper debugging tools can be a time consuming and error prone business. Moreover, to investigate the suspected performances issues related to the megamorphic call sites, we wanted to make sure that we could recreate this new grammar in such a way that we could effectively compare its performances with the previous implementation.

To do so, we created an **object graph model** representing the grammar. This model would be used to generate actual grammars using a model compiler. The first challenge was to regenerate the exact same java grammar we had before based on the model. The reason for this step was to ensure the correctness of the model, if we can regenerate the grammar from the model, then we can be sure that another grammar generate from the same model will be correct as well.

The second compiler, baptise "graph model compiler" aimed to generate a java grammar based on the object oriented implementation of the parsers. It's name comes from the fact that the grammar generated is represented by a graph of connected objects. Figure 4.1 illustrate the relationship between model, model compiler and grammar.

Then, both grammar was tested on a consequential java corpus. It appeared that the performances, although slightly affected by the creation of so many objects wasn't impacted in a significant way. The results will be discussed more deeply in the chapter 5.

The other motivation we had to create this framework was to simplify the writting of grammar. Autumn grammars are indeed constrained by the inlining notation of the Kotlin language that introduce a less desirable verbosity. The notation for rule definition can be greatly simplified by using this framework as one can see in the following code sample.

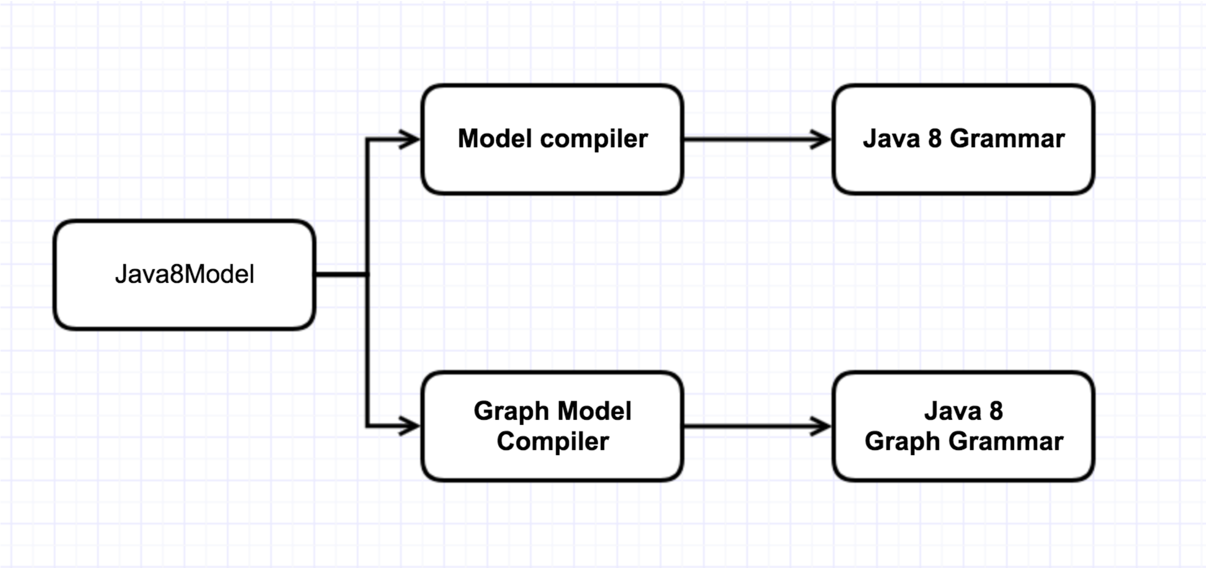


Figure 4.1: Chart illustrating model-grammar generation

```
val method_ref_suffix = Build(2,
    syntax = iden,
    effect = {MaybeBoundMethodReference(it(0), it(1),
                                        it(2))}).g.d("method_ref_suffix")
```

```
val method_ref_suffix
    = iden
    .build (2, "MaybeBoundMethodReference(it(0), it(1), it(2))")
```

4.1.4 Syntax Tree generation

The AST autumn is building during the parse is abstracting some of the nodes together to reduce the size of the tree and make it more readable from a human stand-point. A consequence of this is that we don't have a direct mapping between tree nodes and grammar rules.

This is a critical part of the debuging effort. The syntax tree is build by the debuging logic, separately from autumn AST and is only build during in debugging mode. For debugging purposes, backtracking nodes are kept within the tree as well. Backtracking nodes are of course marked as backtracked

The implementation of the syntax tree is contain in the STNode class. It is the structure that holds all the informations needed to debug the grammar.

4.1.5 Debug logic and syntax nodes

When called the debuging logic will start building up the syntaxt tree as the different parsers get invoked. For each invoked parser, a syntax node is created graft onto the tree. The final syntax tree counts one node per parser invoked.

One might wonder what happens when a particular parser fail and backtracked over. Nothing happens, we keep this node right inside the tree, because for debugging purposes, it can be interesting to see which node backtracked as a rule could fail to match an input while its intended purpose was to match it.

The syntax tree itself is represented by a chained structure called syntax node, or STNode. Each parser call is represented by an STNode in the tree. It holds information about the parser:

- each nodes knows information about the parser it is linked to
- its type
- if it is the definition of a rule
- the position in the input
- the size of the state log to be able to regenerate the state
- its parent and children

In the context sensitivity setting, it might be important to be able to display information related to the global parse state. The way autumn recall context is to register it within a mutable parse state. This parse state essentially works as a log whose entries represent mutations applied to this state. Each entry stores the mutation as well as a way to revert it, and later on reapply it. Therefore, it is possible by reccording the size of the log to actually regenerate the parse state corresponding to a specific parser call.

This state represent the context shared between different parsers, as such, when one parser backtracked, the mutations it applied to the parse state are reverted and removed from the log. Therefore, to be able to regenerate the parse state for backtracked parsers, we need to adapt our implementation a bit.

The debug logic is contained in the *debugNode* class. Any number of additional functionality can be added by creating new hooks to the implementation of the debug function

4.2 Plugin Implementation

Intellij IDE's interace uses java SWING. Considering that the entire project has been code in Kotlin, we were reluctant to use this out of style implementation of GUI.

Because we needed to our implementation to interoperate with the SWING components of the IDE, we considered javaFX [11] which is an evolution of SWING by Oracle, but like its predecessor it's written in java and still very verbose. In the end, we were seduced by the tornadoFX [12] framework, coded in kotlin, it is build on top of of javaFX, assuring seamless interoperability with SWING as well as providing the powerful syntax of Kotlin.

TornadoFX framework - reasons why we used tornadofx

Present the strenght of tornadoFX vs standard SWING. Highlight the fact that it is written in kotlin and therefore can harness the strenght of the language and at the same time, since its based on javaFX it guaranteed its interoperability with SWING and therefore blend seemlessly with the IDE.

Present some example of how it can dramatically reduce the effort needed to build up a GUI compared to SWING.

4.2.1 Event Handler & Message passing

implementation details of the GUI that are maybe not so relevant as we discussed ... I suppose Im gonna remove this section.

4.2.2 Limitations

4.2.3 execution thread

- intellij doesnt allow certain instructions to be called from another thread than the one expects.
- tornadoFX works in its own thread, so some functionallity are not possible as it is

5

validation

5.1 Benchmarking

5.1.1 Profiling excution time

As mentionned earlier, one interesting aspect to look at in term of performances was the megamorphic call sites. To perfom a meaningful Benchmarking, a consequent java framework [18] has been used as a input corpus for Autumn to parse. The framework itself contains 7800 files and more than a million lines, therefore representing a fairly big project.

Because of megamorphic call sites, it was expected that the new implementation of the parsers would be much slower. It indeed does have an impact on performances increasing the relative execution time by an order of magnitude of 25%, which might be considered significant, although considering the overall execution time with respect to the size of the input (1.2 Million lines), it can be argued that this loss of performances doesnt impact the project so much.

Figures 5.1 provide an illustration of the profiler for the functional parser implementation while figure 5.2 illustrate the results of the profiler for the object oriented implementation of the parsers.

Additionally, we also ran the benchmark for this corpus in debug mode for sake of comprehensivity. Although it is expected that our debugger will be used on smaller test inputs to verify the correctness of a portion of the grammar at a time, it is still relevant to measure the impact the overhead introduced by the debugger has on performances. As figure 5.3 demonstrate, the overhead introduced increase the execution time by an order of 10. Considering that debuggers are usually used to quickly test a solution rather than mentally check the correctness of the code, an execution time of 2 minutes is not acceptable. It is although not reasonable to think that one would use the debugger in that way for huge project like this one.

5.1.2 Profiling memory consumption

In this section we are interested in the memory consumption overhead the debugger introduced. Generally, we expect that one would use this tool on very small examples to provide actionnabe feedback as he or she design the grammar. Therefore, the memory consumption on such small examples, which would represent the majority of the usual use-cases, isn't really critical.

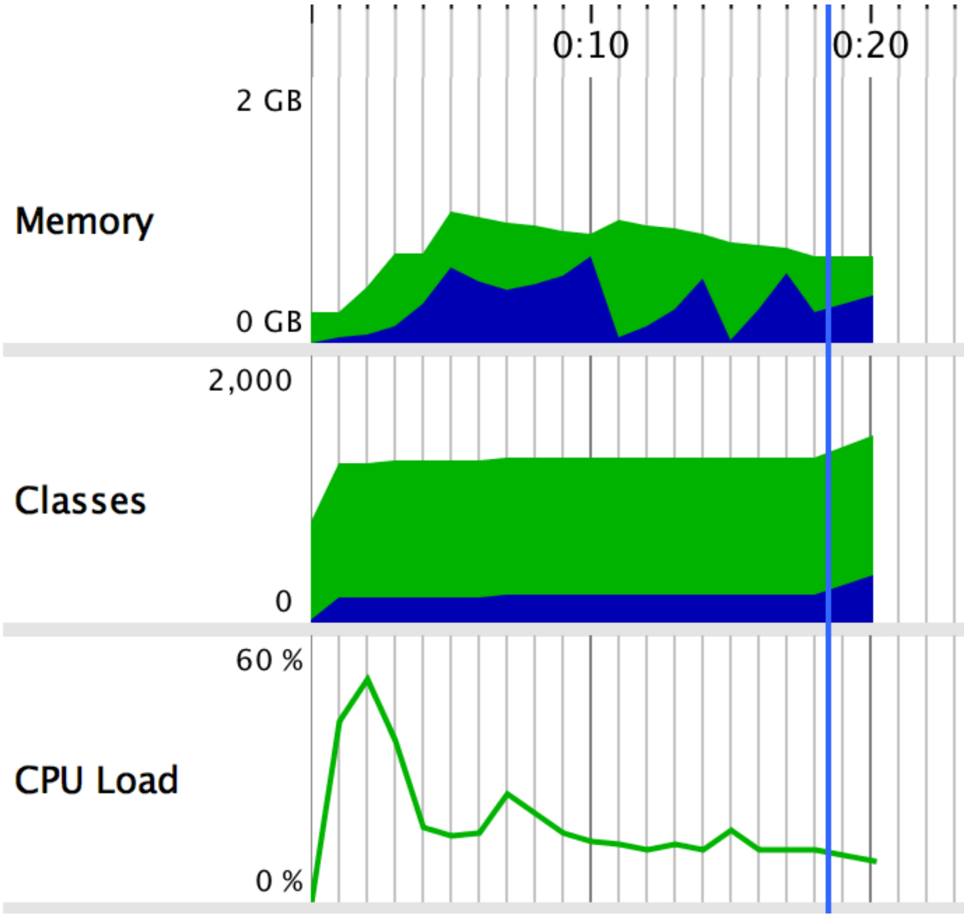


Figure 5.1: Profiler for the benchmark with functional parser implementation

Despite this assumption, if for any reason, one would be interested in running the debugger on a large project, generating and maintaining a large amount of structures could represent an issue for memory consumption and prevent the algorithm to run altogether. To study this issue, we used JProfiler [19] to analyze the memory consumption of the solution.

Figure 5.4 and Figure 5.5 presents the live memory consumption after parsing the same corpus used in the previous section (Java spring framework containing 1.2 million lines).

We can observe that the structure maintained by the debugger increase the memory consumption significantly. The syntax tree alone generate more than 300MB of data, not considering the increased memory cost in the form of extra appliedSideEffects. Of course, the is no way around an increased overhead in memory consumption. Despite those observation, one might argue that, even thought the memory consumption has been increased significantly, the cheer size of the framework used for the benchmark rule out the need for additional modification for our debugger to run on bigger projects.

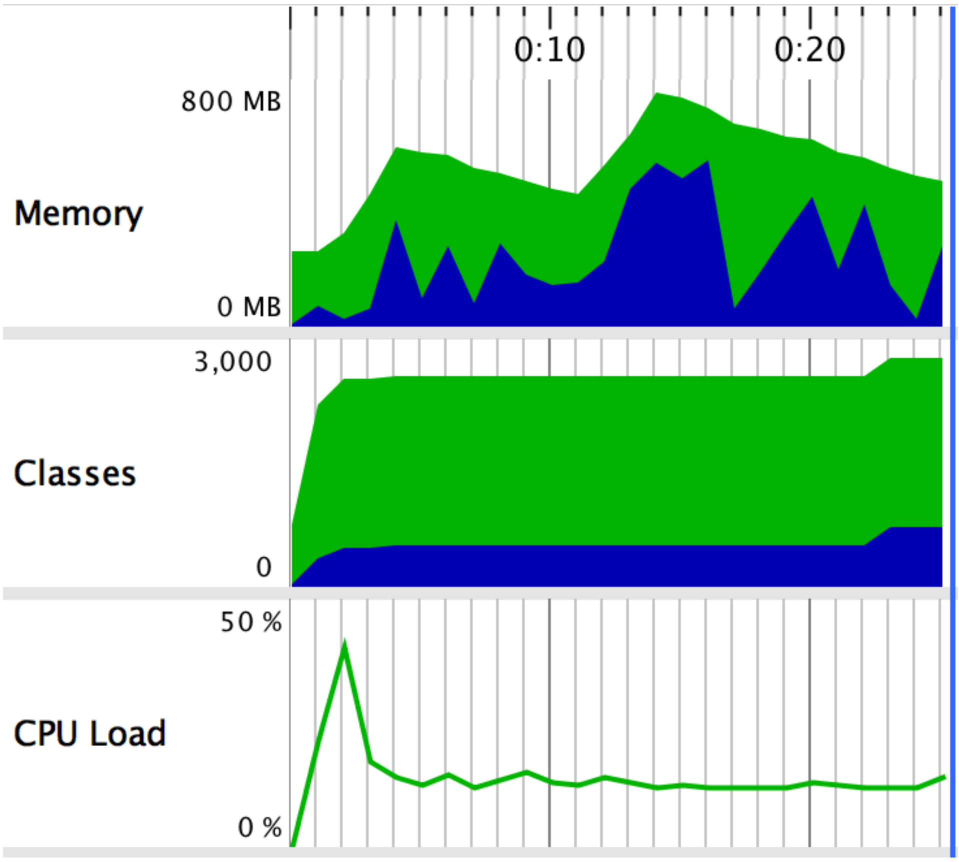


Figure 5.2: Profiler for the benchmark with object oriented implementation

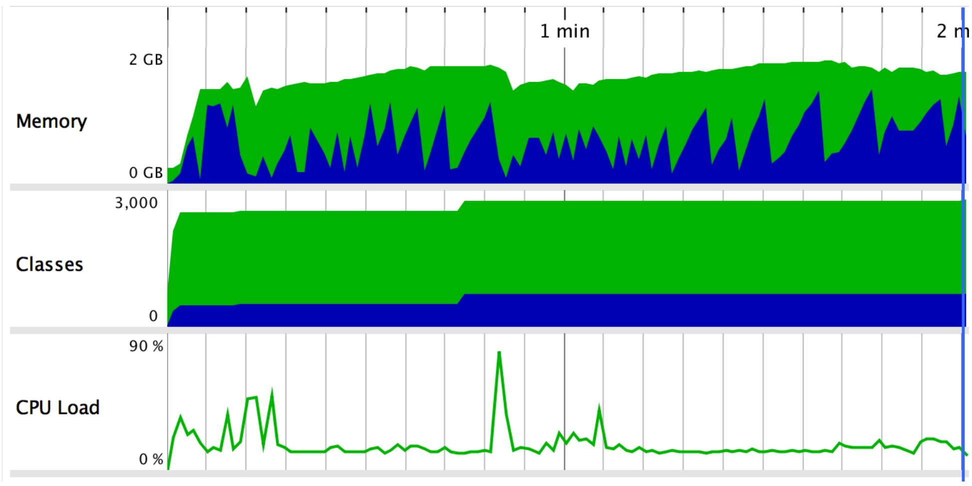
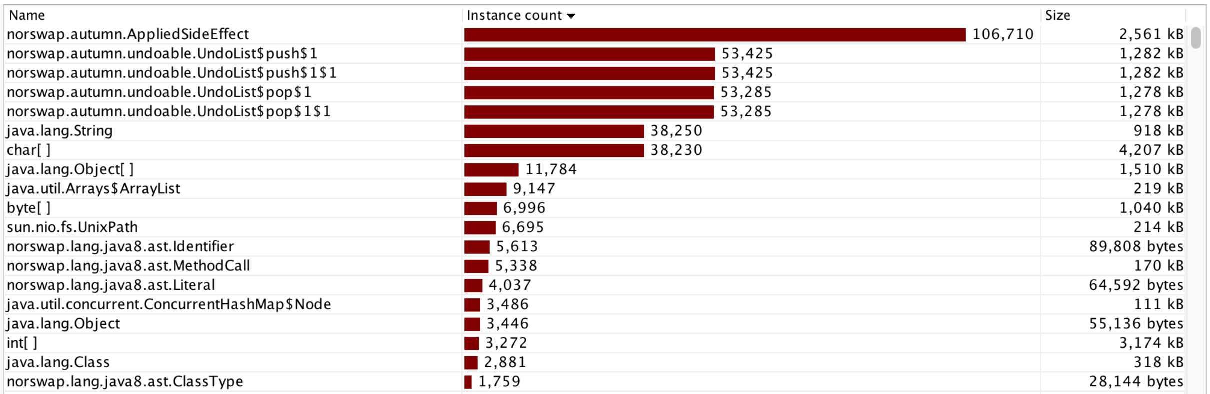


Figure 5.3: Profiler for the benchmark in debug mode

5.2 Debugger in practice

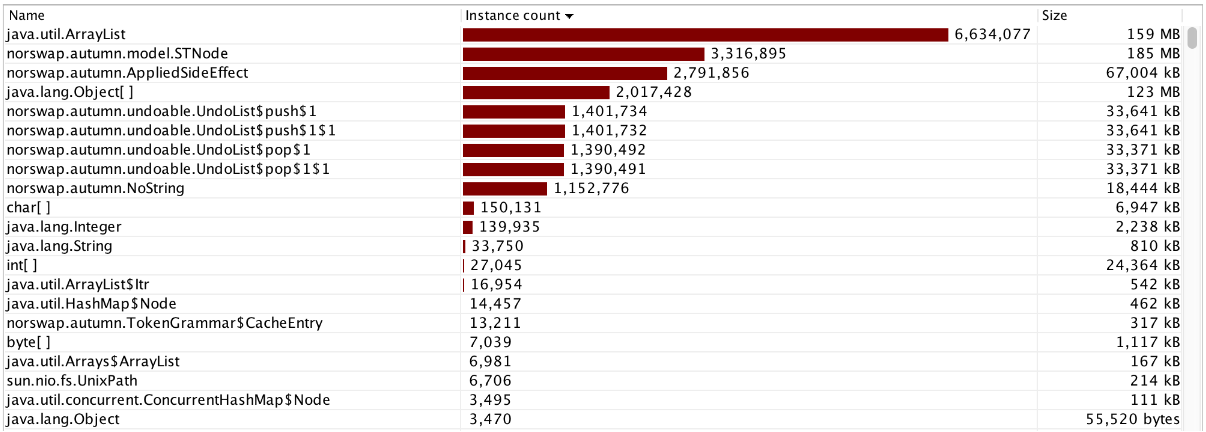
5.2.1 Examples

case study 1 : error in grammar For this example, we will show how to go about to detect an error in the grammar and how our tool helps to resolve the issue.



Name	Instance count	Size
norswap.autumn.AppliedSideEffect	106,710	2,561 kB
norswap.autumn.undoable.UndoList$push$1	53,425	1,282 kB
norswap.autumn.undoable.UndoList$push$1$1	53,425	1,282 kB
norswap.autumn.undoable.UndoListpop1	53,285	1,278 kB
norswap.autumn.undoable.UndoListpop1$1	53,285	1,278 kB
java.lang.String	38,250	918 kB
char[ ]	38,230	4,207 kB
java.lang.Object[ ]	11,784	1,510 kB
java.util.Arrays$ArrayList	9,147	219 kB
byte[ ]	6,996	1,040 kB
sun.nio.fs.UnixPath	6,695	214 kB
norswap.lang.java8.ast.Identifier	5,613	89,808 bytes
norswap.lang.java8.ast.MethodCall	5,338	170 kB
norswap.lang.java8.ast.Literal	4,037	64,592 bytes
java.util.concurrent.ConcurrentHashMap$Node	3,486	111 kB
java.lang.Object	3,446	55,136 bytes
int[ ]	3,272	3,174 kB
java.lang.Class	2,881	318 kB
norswap.lang.java8.ast.ClassType	1,759	28,144 bytes

Figure 5.4: Memory consumption without debugging



Name	Instance count	Size
java.util.ArrayList	6,634,077	159 MB
norswap.autumn.model.STNode	3,316,895	185 MB
norswap.autumn.AppliedSideEffect	2,791,856	67,004 kB
java.lang.Object[ ]	2,017,428	123 MB
norswap.autumn.undoable.UndoList$push$1	1,401,734	33,641 kB
norswap.autumn.undoable.UndoList$push$1$1	1,401,732	33,641 kB
norswap.autumn.undoable.UndoListpop1	1,390,492	33,371 kB
norswap.autumn.undoable.UndoListpop1$1	1,390,491	33,371 kB
norswap.autumn.NoString	1,152,776	18,444 kB
char[ ]	150,131	6,947 kB
java.lang.Integer	139,935	2,238 kB
java.lang.String	33,750	810 kB
int[ ]	27,045	24,364 kB
java.util.ArrayList$Itr	16,954	542 kB
java.util.HashMap$Node	14,457	462 kB
norswap.autumn.TokenGrammar$CacheEntry	13,211	317 kB
byte[ ]	7,039	1,117 kB
java.util.Arrays$ArrayList	6,981	167 kB
sun.nio.fs.UnixPath	6,706	214 kB
java.util.concurrent.ConcurrentHashMap$Node	3,495	111 kB
java.lang.Object	3,470	55,520 bytes

Figure 5.5: Memory consumption with debugging

case study 2 : error in input This time, the grammar will be correct, but we will introduce a mistake in the parsed input and see how the tool helps to find out where the problem is.

case study 3 : error in parser definition Finally, because a user could define a custom parser to handle some custom/domain specific parse state, we will introduce a mistake in the definition of a parser and see how it can be detected.

6

Future work

6.1 Debugger extension

6.1.1 Adding hooks

- access more informations during debug
- create new state structure

6.1.2 Turning the plugin into a full fledge independent software

6.1.3 Working in tandem with the general purpose debugger

6.2 Autumn related

6.2.1 Parser implementation

- as discussed, parsers as been reimplemented as objects.
- the implementation was meant to be temporary because of the suspected performances issues
- since this issues are neglictable, the functions can be directly integreted in the object implementation
- better management of memory

6.3 GUI extentions

Making use of tornadoFX powerful verboseless feature, we implemented the plugin strictly following the MVC paradigm, decoupling the functionalities from the display. We worked with easy maintainability and extendability in mind at all time.

The code is divided in 3 different classes, there is the Model which stores all the data. Then there is the views that define the way data will be displayed. And finally, there is a controller, the middleman that request information to the model and dispatch them to the views. The implementation make use of an event system which help decouple the controller and the view furthermore. It is therefore very easy to create new views to hook on the masterview, or replace the masterview all together, the only thing needed is to listen to the event and display the received data in the chosen way. Symmetrically, it is as easy to create new filters, or provide new information to the views by creating new events and new methods in the controller.

Because Autumn implements stateful parsing, the users may define custom states for his parsers. It is then his responsability to create a display for it and to feed it to the plugin views. For this purpose I added an interface for the states that define a “getRepresentation” method.

7

Related work - state of the art

7.1 The Moldable Debugger: a Framework for Developing Domain-Specific Debuggers

The moldable debugger is a framework to develop domain specific debugger.

There exist two main approaches to address, at the application level, the gap between the debugging needs and debugging support: – supporting domain-specific debugging operations for stepping through the execution, setting breakpoints, checking invariants [10,11,12] and querying stack-related information [13,14,15]. – providing debuggers with domain-specific user interfaces that do not neces- sarily have a predefined content or a fixed layout [16].

We start from the realization that the most basic feature of a debugger model is to enable the customization of all aspects, and we design a debugging model around this principle. We call our approach the Moldable Debugger.

The Moldable Debugger decomposes a domain-specific debugger into a domain- specific extension and an activation predicate. The domain-specific extension customizes the user interface and the operations of the debugger, while the activation predicate captures the state of the running program in which that domain-specific extension is applicable. In a nutshell, the Moldable Debugger model allows developers to mold the functionality of the debugger to their own domains by creating domain-specific extensions.

Then, at run time, the Moldable Debugger adapts to the current domain by using activation predicates to select appropriate extensions.

A domain-specific extension consists of (i) a set of domain-specific debugging operations and (ii) a domain-specific debugging view, both built on top of (iii) a debugging session. The debugging session abstracts the low-level details of a domain. Domain-specific operations reify debugging operations as objects that control the execution of a program by creating and combining debugging events. We model debugging events as objects that encapsulate a predicate over the state of the running program (e.g., method call, attribute mutation) [17]. A domain- specific debugging view consists of a set of graphical widgets that offer debugging information. Each widget locates and loads, at run-time, relevant domain-specific operations using an annotation-based approach.

- domain-specific user interfaces: User interfaces of software development tools tend to provide large quantities of information, especially as the size of systems increases. This in turn, increases the navigation effort of identifying the information relevant for a given task.

- To address this concern an infrastructure for developing domain-specific debuggers should:
  - allow domain-specific debuggers to have domain-specific user interfaces dis- playing information relevant for their particular domains;
  - support the fast prototyping of domain-specific user interfaces for debugging.
- domain-specific debugging operations : Debugging is viewed as a laborious activity requiring much manual and repet- itive work. This idea of having customizable or programmable debugging operations that view debugging as an event-oriented activity has been supported in related works [10,11,12,23]. Mainstream debuggers like GDB have, to some extent, also incorporated it.
- automatic discovery
- dynamic switching
- This framework inspired our first idea of debugger

7.2 Antlr [13]

- ANTLR parser generator accepts a larger class of grammars than LL(k)

7.3 Ohm

Ohm is a parser generator consisting of a library and a domain-specific language. You can use it to parse custom file formats or quickly build parsers, interpreters, and compilers for programming languages. The Ohm language is based on parsing expression grammars (PEGs), which are a formal way of describing syntax, similar to regular expressions and context-free grammars. The Ohm library provides a JavaScript interface (known as Ohm/JS) for creating parsers, interpreters, and more from the grammars you write.

Like its older sibling OMeta, Ohm supports object-oriented grammar extension. One thing that distinguishes Ohm from other parsing tools is that it completely separates grammars from semantic actions. In Ohm, a grammar defines a language, and semantic actions specify what to do with valid inputs in that language. Semantic actions are written in the host language — e.g., for Ohm/JS, the host language is JavaScript. Ohm grammars, on the other hand, work without modification in any host language. This separation improves modularity, and makes both grammars and semantic actions easier to read and understand. Currently, JavaScript is the only host language, but as the API stabilizes, we hope to have implementations for other languages.

To use Ohm, you need a grammar that is written in the Ohm language. The grammar provides a formal definition of the language or data format that you want to parse. There are a few different ways you can define an Ohm grammar:Ohm has two tools to help you debug grammars: a text trace, and a graphical visualizer. The visualizer is still under development (i.e., it might be buggy!) but it can still be useful.

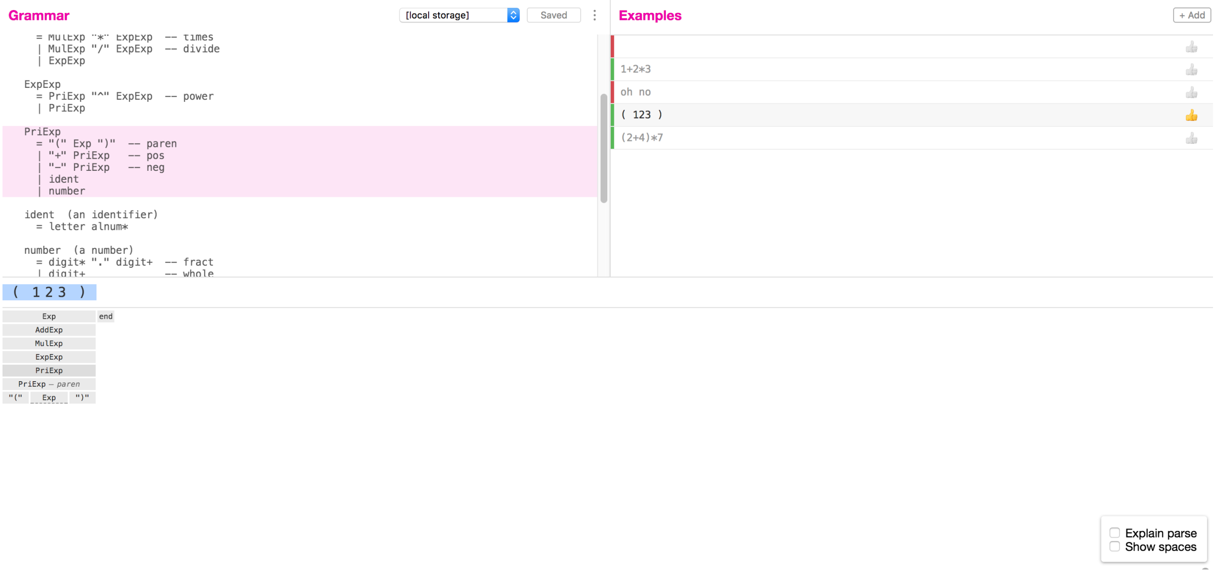


Figure 7.1:

7.4 Debugger Canvas: Industrial Experience with the Code Bubbles Paradigm

Debugger Canvas opens each executed method in its own bubble and draws arrows to represent method calls. Each bubble has a pop-up that shows the current value of the local variables, which can be snapshotted for compari- son over time. Because the task of debugging also involves code exploration and trying potential bug fixes, Debugger Canvas also supports code navigation features, like go-to definition, and editing with the bubble. Each bubble is a full-fledged Visual Studio editor, with all the typical features like tooltips and code completion.

7.5 Grammar-Driven Generation of Domain-Specific Language Debuggers

Their approach is to generate a debugger from and for a domain specific language. They state that most end user developpers are in fact not trained computer scientists but instead specialists of other domain developping programs using DSL to abstract the lower level language knowledge empowering them to develop softwares without knowledge of actual computer language. They propose a tool that can modify the grammar to insert a hook that provides an entry point for the debugging.

- imperative DSL: An imperative programming language is based on the von Neumann concept that is centered on assignment expressions and control flow statements [52], which allows a program to change the content of cells in memory. In an imperative language, the state change of variable values is a central feature of interest. Therefore, for imperative languages, debuggers are designed around capabilities to examine the value of variables at run-time.

- declarative DSL: A declarative programming language is based on declarations that state the relationship between inputs and outputs. Declarative programs consist of declarations rather than assignment or control flow statements. The declaration semantics have a precise interpretation that is closer to the problem domain. Such programs do not state how to solve a problem, but rather describe the essence of a problem and let the language environment determine how to obtain a result [52]. Instead of assessing the value of individual variables, a declarative DSL debugger needs to evaluate the relationships between each declaration, which are often represented as data structures with symbolic logic.
- hybrid DSL

Bibliography

[1] The jetbrains intelliji platform documentation, http://www.jetbrains.org/intellij/sdk/docs/basics/architectural_overview/file_view_providers.html, June 2017.

[2] Ohm, a parser generator consisting of a library and a domain-specific language. https://github.com/harc/ohm, 2017.

[3] The Moldable Debugger: A Framework for Developing Domain-Specific Debuggers https://link.springer.com/chapter/10.1007/978-3-319-11245-9_6.

[4] A graph grammar approach to graphical parsing http://ieeexplore.ieee.org/document/520809/.

[5] Laurent Nicolas and Mens Kim. Parsing Expression Grammars Made Practical. SLE, 2015.

[6] Laurent Nicolas and Mens Kim. Taming Context-Sensitive Languages with Principled Stateful Parsing. SLE, 2016.

[7] David R. Hanson and Jeffrey L. Korn. : A Simple and Extensible Graphical Debugger. Proceedings of the USENIX Annual Technical Conference, January 6-10, 1997. 173–184

[8] petit parser 28. Renggli, L., Ducasse, S., G^ırba, T., Nierstrasz, O.: Practical dynamic grammars for dynamic languages. In: Proc. DYLA. (2010)

[9] TornadoFX https://edvin.gitbooks.io/tornadofx-guide/content/4.%20Basic%20Controls.html

[10] psi cookbook http://www.jetbrains.org/intellij/sdk/docs/basics/psi_cookbook.html

[11] Oracle, javaFX http://docs.oracle.com/javase/8/javase-clienttechnologies.htm

[12] TornadoFX guide https://edvin.gitbooks.io/tornadofx-guide/content/1.%20Why%20TornadoFX.html

[13] Terence J. Parr. The Definitive ANTLR Reference: Building Domain-Specific Languages. The Pragmatic Programmers, 2007. ISBN 0-9787392-5-6.

[14] Terence Parr, Jean Bovet. ANTLRWorks: An ANTLR Grammar Development Environment, UNPUBLISHED DRAFT.

[15] Bryan Ford. Parsing Expression Grammars: A Recognition-based Syntactic Foundation. In POPL, pages 111–122. ACM, 2004.

[16] Theodore Norvell. A Short Introduction to Regular Expressions and Context Free Grammars. Software Engineering 7893, Typseset November 8, 2002 http://www.engr.mun.ca/~theo/Courses/fm/pub/context-free.pdf

[17] DeLine, R., Bragdon, A., Rowan, K., Jacobsen, J., Reiss, S.P.: Debugger canvas: industrial experience with the code bubbles paradigm. In: ICSE. (2012) 1064–1073.

[18] The Spring Framework, a comprehensive programming and configuration model for modern Java-based enterprise applications. https://github.com/spring-projects/spring-framework

[19] EJTechnology, JProfiler. Java JVM all in one profiler http://www.ej-technologies.com/products/jprofiler/overview.html?gclid=EAIaIQobChMImda0lYLD1QIVjxbTCh1qxAXMEAAYASAAEgKca_D_BwE

[20] The Economic Impacts of Inadequate Infrastructure for Software Testing