Universite Catholique De Louvain

Ecole Polytechnique De Louvain

INGI department

A debugging tool for Autumn's Context-Sensitive Grammars

Author
Nicolas Gerard

Supervisors
Pr. Kim Mens
Phd Sdt. Nicolas Laurent

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

Academic year 2016-2017

Abstract

In grammar development, like any other project, the debugging process is a crucial part of its lifetime. Unfortunately, general purpose debuggers cannot provide the level of abstraction needed to reason efficiently on grammar development related errors. Concepts like input stream manipulation or production rule being unknown to it result in difficulties to track whether mistakes comes from the input or the grammar itself.

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

To achieve this, we developed an IntelliJ plugin that exposes the underlying structure of the grammar in a tree fashion and allows to observe the state of the parse during the invocation of any specific parser.

Acknowledgements

I would like to thank my mentor Kim Mens and PHD student Nicolas Laurent for their support and understanding throughout this year. I would also like to thank my friends and familly who will never read this paper but supported me emotionally throughout my entire scholar education nonetheless.

Contents

1

Introduction

This paper will present a *debugging tool* developed to help track issues related to the design of grammars for *Autumn*, a context sensitive parsing library.

Developing rules for grammars that specifies a language is a difficult and error-prone task. Errors can come from different source and without the proper tool to analyze the behavior of a parse, it is really difficult to determine whether a mistake comes from a rule, from the input or even from a parser.

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 execution in real time and to be able to stop it at will. Instead, our tool explore another approach: it first lets the execution of the parse run completely, gathering all the relevant information needed to reason about the grammar down the road.

Nonetheless, 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 time chunk dedicated to debugging is usually important and represent a crucial step in the development of a solution. Traditionally, debuggers have given the developers 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 that developers think and reason in term of high level abstractions while debuggers only expose raw data. This force developers 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. [22]

When applied to our domain of parsing, 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. Since general purposed debuggers have no knowledge about parsing or input streams, the developer would have to manually single step through the entire execution until we reach the aforementioned input position.

Moreover, writing 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 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 purposed debugger are rarely useful. Indeed, the debugger having no knowledge about those higher level abstractions cannot inform the developers with clear information to reason with.

Other debugging solution has been developed for other parsing library. However, the challenge of implementing this debugging solution lies in the way *Autumn* express context sensitivity through the manipulation of a parse wide 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. With this tool, the developer can track the invocation sequence of the parsers to test grammar rules and help him to detect errors across the project.

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

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

This paper will be divided in 6 different chapters. Chapter 2 will explain important concepts related to *Autumn's library* needed to understand the content of this paper. Chapter 3 will present the solution's functionality while Chapter 4 will present key aspects of its implementation. Chapter 5 will presents some statistics and use cases of our work. And finally, Chapter 7 will position our solution with respect to the state of the art.

2

Background Material

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

2.1 Parsing

As `Terence Parr` [19] put it, parsing can be explained using a 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."

If the grammar is a maze, then a parse is simply a walk through it.

Backtracking When navigating through the maze, sometimes it is possible that at a specific fork the words on the floor are identical on each path. Each path differentiate itself from the others by the sequence of words that is going to arise down its way but at this point there is no way to know which path is gonna be the right one. What is usually done is to try the first path, if at some point it fails to match the input, then we simply backtrack to the previous fork and try another one. This mechanism is called "backtracking"

2.2 Autumn Parsing Library

Autumn is a context sensitive parsing library inspired by PEG formalism, a formal way of describing syntax similar to regular expressions and context-free grammars. [12] PEG formalize a serie of operators, such as *Sequence* or *Ordered Choice* to define the semantic of grammars. More generally, Autumn formalize a functional flow of operations, users can define their own arbitrary functions that analyze the input and either advance the parse or fail.

2.2.1 Context sensitivity and its difficulties

Nowadays, most modern programming languages exhibit context sensitivity features[1].

> Here are a few examples of context sensitivity features in mordern programming languages :
>
> - Python uses significant whitespace to separate blocks of instructions, therefore, a parser needs to know the relative level of indentation to parse an input successfully.
> - In C, ambiguous statements like $x * y$ exists. To parse such statement correctly, one needs information about previously declared type definitions in order to predict if it is a product of two variables or a pointer of type x

Many more examples exists showing that previous knowledge of context is often required when parsing computer languages. However, true context-sensitivity is rarely handled by parsing libraries because of the difficulties of expressing it with current grammar formalisms.

The notion of *context transparency* [14], a property stating that context needs to be passed implicitly so that grammar constructs don't have to know about the context shared between its ancestors and descendants is a desirable property to decouple the grammar design from the context itself.

[1] Note that we are not talking about context sensitive grammar (CSG) as defined by Chomsky [9].

Unfortunately most solutions lack *context transperancy*, intertwining instead context with grammar making it difficult to maintain or even reason about.

Autumn, in the other hand, is a context sensitive parser combinator library that approach the issue of context transparency by introducing the **"principled stateful parsing"** discipline. It is based on a general notion of using a *global state* to pass the context around implicitly through a parse wide mutable state. To share context information through the manipulation of a global state introduces its own set of problems [16] that Autumn solve by introducing a set a primitive operations designed to formally interact with the parse state.

2.2.2 Global state and its shortcomings

The general approach relies on the use of a mutable parse wide state to pass the context between grammar rules around implicitly. While it might sounds like an obvious answer to the context sensitivity problematic, the manipulation a parse wide state introduces issues that are neither trivial nor obvious.

During its invocation, a parser can alter the context by mutating the state which will affect the results of the subsequent parse. The issues start to arise when a parser is faced with a choice. In this case, it has to perform a speculative execution, therefore one of its alternative might fails and need to backtrack. When this occurs, simply reverting the mutations applied by a parser when it fails is not enough. Indeed, let's consider the *Sequence* parser combinator. To succeed, each of its children has to succeed as well. Therefore, if some of its children return successfully but one of them fails at some point, only the last failing parser will revert its modification to the state, potentially leaving the state mutated by the other children. The *Sequence* parser needs to know how its children affected the parse state failing to achieve context transparency.

Autumn propose to solve this problem by introducing a set of primitive state manipulation operations to formally interact with the parse state, which was baptised "principled stateful parsing". Those primitive operations provides a set of building blocks which enables the user to follow a simple rule called transactionality rule. This rule states that either a parser succeeds, or it fails without modifying the state. By

definition, respecting this rule for every parser is enough to prevent the corruption the global state used to pass context around.

2.2.3 Principled Stateful Parsing

Because of the speculative execution and backtracking of parsers, we sometimes need to revert the mutation applied to the state. To achieve *context transperancy*, Autumn introduce the notion of *recall*. Upon failure, a parser needs to be able to recall the state it was in before its execution.

This is achieved by formalizing the interactions with the state through four primitive state manipulation operations.

- *snapshot* operation : capture an image of the state at a specific point during the execution
- *restore* operation : restore the state to match the one described by the snapshot
- *diff* operation : returns a DELTA object representing the difference between a snapshot and the current state
- *merge* operation : apply a DELTA to the state

Through those operation, Autumn creates a log representing the mutation of the parse state. A state can be reverted by simply reverting every entries of the log that has been added since the invocation of the parser. To reuse our previous example of the *sequence* parser, the parser can record the size of the log before calling its children, if one of them fails, recalling the state is only a matter of reverting the newly created entries of the log. That way, the *sequence* parser doesn't have know anything about the context shared by its children, achieving context transparency.

3

Overview of the solution

This chapter presents the main functionality of the debugging tool 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 *debugger* in the traditional meaning of the word. Despite this fact, we will refer to it as a debugger for simplicity.

3.1 Overview

To empower Autumn's parsing library with a proper debugging tool, we developed an IDE plugin designed to be a staple for the developer 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 structure akin to a traditional parse tree during its invocation.

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 is augmented by the addition of failing parsers. The resulting parse tree is then passed to the plugin's GUI.

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 summarized as a search in a tree of events.

3.2 Functionality

3.2.1 Overview of the GUI

The plugin we developed is essentially composed of a panel that can be docked on any side of the IDE window, although it is recommended 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.

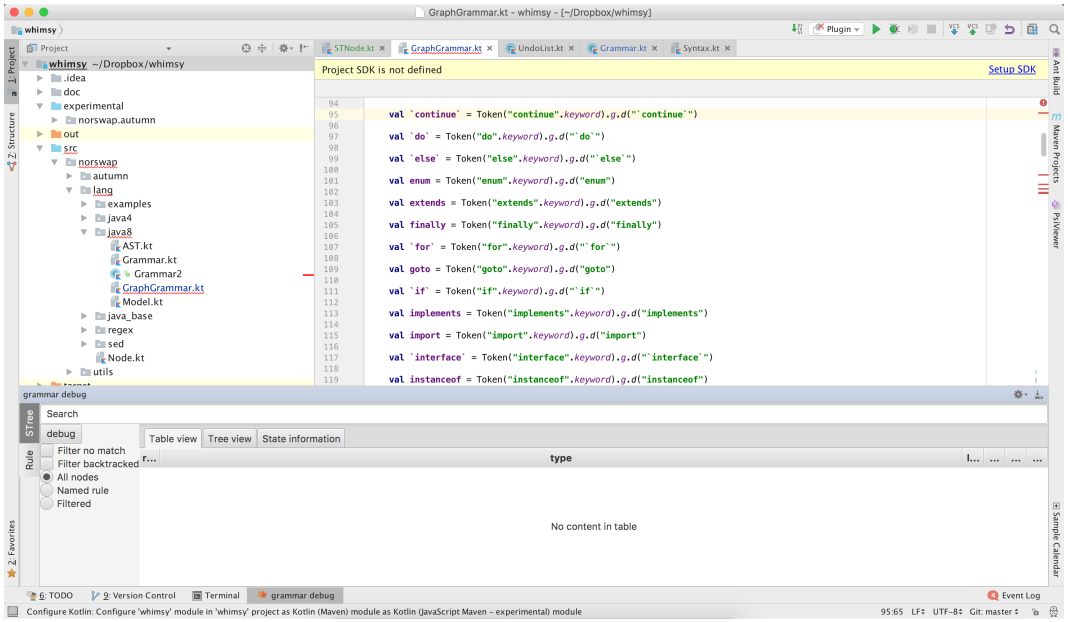

Figure 3.1: IDE GUI for Autumn's debuggin tool

The parse tree that is build during a debugging session 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 structure, highlighting the hierarchy between parsers. Each rows of both views then correspond to the invocation of a single parser.

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 hierarchy 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 hierarchy. So even if it can argued that both views are redundant and unnecessary, we think that in some cases, it can provide a small improvement in the quality of life of the developer 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 developer as it is quite 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 introducing 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 functionalities (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.

3.2.2 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

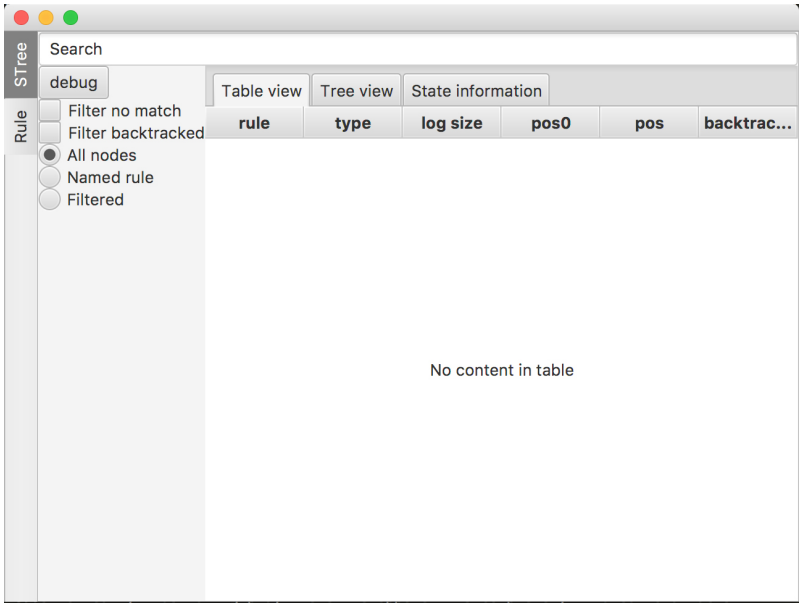

Figure 3.2: Stand alone view - the debugger GUI can be executed independently from the IDE albeit some restrictions

- 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.

Tableview - 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 contains 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.

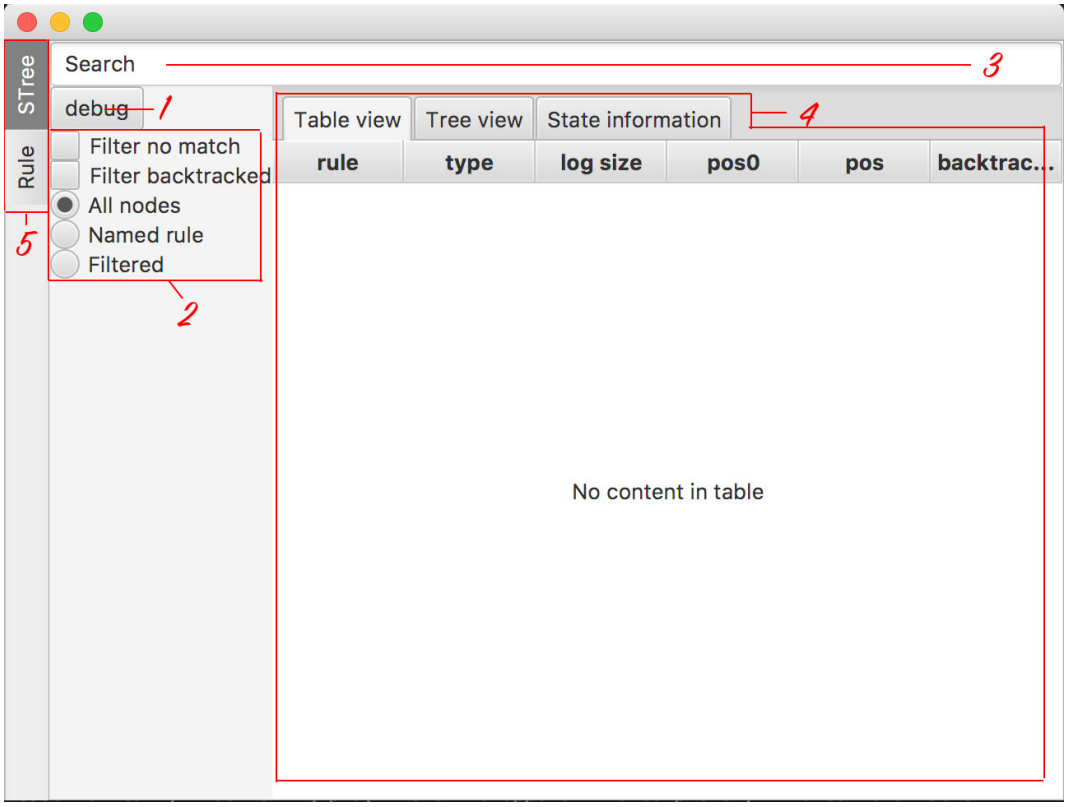

Figure 3.3: Close up view of the plugin in detail

- 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.

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

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
	Javalden	21	0	7	false

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

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 it, as a side effects, the column gets resized to accommodate 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 recursive 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 similarly to the previous one, it take the parent of the root and sets it as the new root. This is useful when

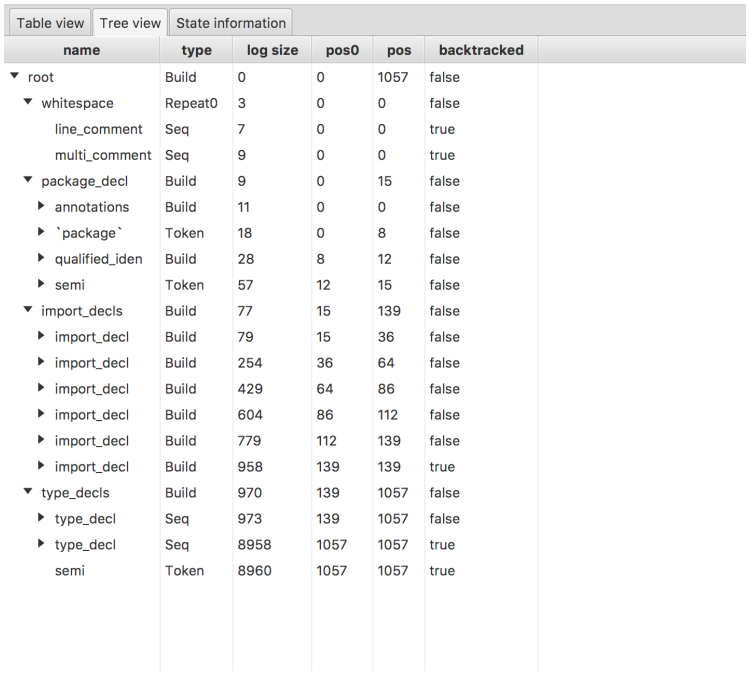

Figure 3.5

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 nonetheless 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 existence 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.

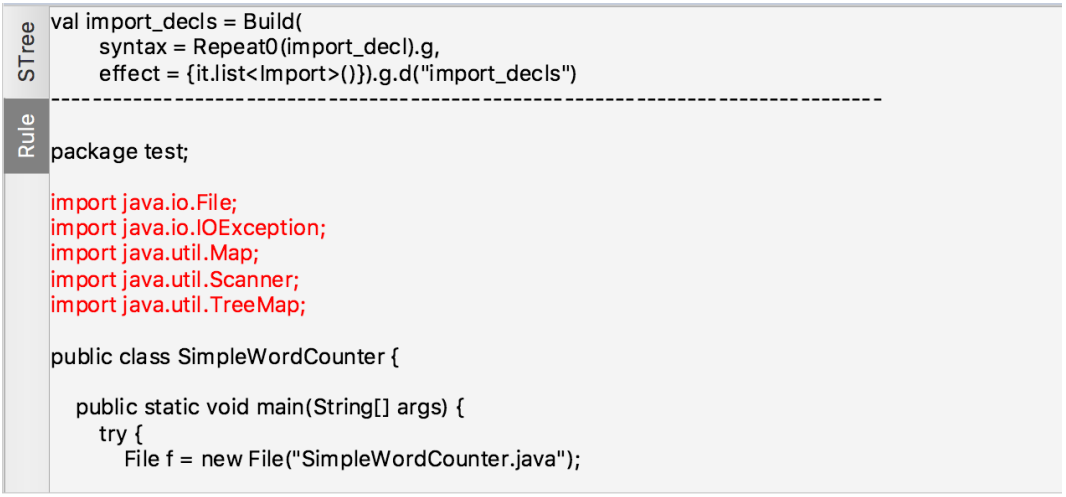

Figure 3.6: Example of grammar rule information and matched input for a particular parser

3.2.3 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 upon 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. Similarly 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 beginning 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. Intrinsically, this radio button doesn't do much except indicating that the display is driven by the search field, which overrides the other name based filters.

3.2.4 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.

4

Implementation

4.1 The debugging tool specificity

4.1.1 Why not a "normal" debugger ?

We discussed in chapter 3 that our solution for debugging Autumn grammars wasn't exactly a debugger strictly speaking but rather a tool that enable the developer to analyze the execution trace at the parser level.

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 debugger by overlaying our logic on top of Intellij's general purposed 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 developers and asked ourselves questions like:

- What problem will I face while developing 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 ?

The most common problem encountered by grammar developers 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.

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

1. the ability to expose chronologically the flow of execution of the parse at a higher level of abstraction than instruction by instruction, being able to see which parser was called in which order, 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 theoretically met or if the problem came from elsewhere..
2. 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.
3. lastly, the ability 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 purposed 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.

The question was then, instead of enhancing the general purposed debugger 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 hierarchy 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. In this context, it is not unreasonable to record a single integer for each node of the tree, that leaves the last 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 primarily 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[1]. From this information, it is easy to re-generate the parse state as it were during the invocation of a specific parser.

Our alternate solution to traditional debugging is based on this idea of 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.

4.1.2 Hooking into the parser implementation

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

Originally, the parsers were implemented in a strictly functional fashion. Autumn defines a parser simply as a boolean function taking some input, optionally modifying

[1]In practice, it isn't that simple as we need to account for the fact that when backtracking occurs, the entries from the log corresponding to the mutation the backtracking parser applied are reverted AND removed from the log. To fully enable the regeneration of the state, we need to record that information so it is not lost

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 implementation 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 referred to this new implementation as "naive parsers" for the reason that we suspected some performance issues due to the nature of its implementation. Since every parsers implement the same interface, it's impossible to know at the call site which parser will actually be called, therefore, there is a necessary operation that needs to occur to retrieve the implementation of the methods of the interface. This issue can be 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. A plausible explanation for this result would be that the JIT is able to optimize this issue to avoid the need for this lookup. The results of these benchmark is discussed in chapter 5

4.1.3 Rewriting 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 **??**.

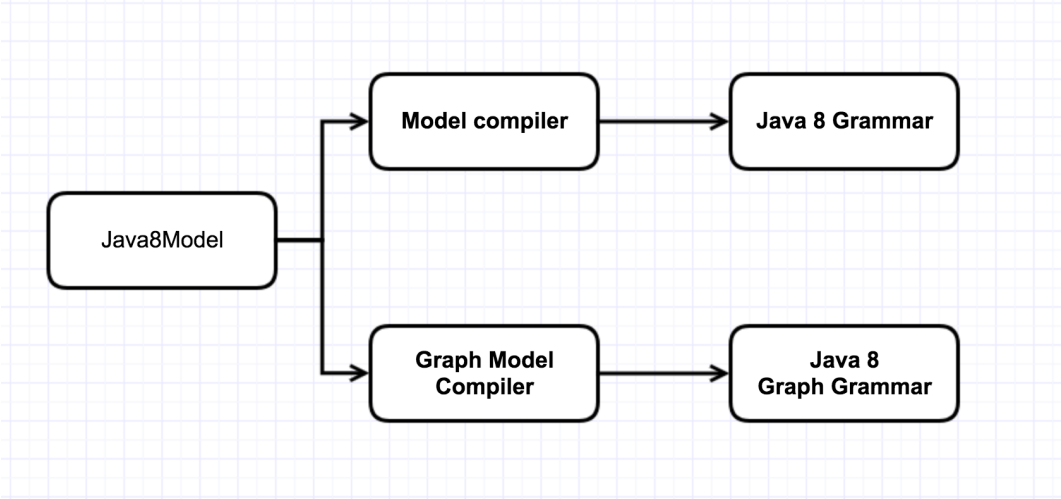

Figure 4.1: Chart illustrating model-grammar generation

The other motivation we had to create this framework was to simplify the writing 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 simplified by using this framework although it still present a consequent downside for the fact that the string cannot be verified by the kotlin compiler nor refactored easily. The following code samples illustrate this.

```
fun method_ref_suffix() = build(2,
    syntax = { iden() },
    effect = { MaybeBoundMethodReference(it(0),
                        it(1), it(2)) })
```

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

4.1.4 Syntax Tree generation

Syntax tree Unlike what is usually done in parsing library, autumn doesn't build a syntax tree. Instead it defines a serie of parsers design to help the user to generate their own abstract syntax tree (AST) build by contracting 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. Note that backtracking nodes are kept within the tree as well. Since a problematic parser could backtracked unintendedly, keeping them within the tree enable us to analyze their behavior and state to detect where the problem lies. Backtracking nodes are marked as such using a boolean property so they can be treated separately.

Having a one to one relationship between the rules of the grammar and the node of the tree is a crucial 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.

During a debugging session, Autumn will attempt to parse the entire input. Weather it is successful or not, it produces a structure akin to a traditional parse tree during its invocation.

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

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

4.1.5 Debug logic and syntax nodes

When running in debug mode, the syntax tree is getting build up 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 re-apply it. Therefore, it is possible by recording the size of the log to actually re-generate the parse state corresponding to a specific parser call.

This state represents 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

Despite the fact that Intellij IDE's interface uses java SWING, we decided to use TornadoFX instead which allowed for a much less verbose and easy to maintain implementation of the GUI. TornadoFX [4] is directly based on javaFX [1] which is an evolution of SWING by Oracle. Coded in Kotlin, it allowed us to leverage the strength of the Kotlin syntax while maintaining a good interoperability with the SWING components.

The GUI has been implemented the plugin using the MVC paradigm, decoupling the data from the display in order to provide easy maintainability and extendability.

The code is divided in 3 different classes, there is the Model which stores all the data, the views define the form of the display, and finally, the controller which plays the role of the middleman requesting information from the model and dispatching

them to the views. The communication between the different parts of the GUI happens through an event system.

Because it is less relevant with the scope of this paper, we won't dive in the implementation details of the GUI.

5

validation

5.1 Benchmarking

5.1.1 Profiling excution time

As mentioned earlier, comparing the performances of functional and object oriented implementation for the parsers was interesting to observe the impact of the megamorphic call sites. To perform a meaningful Benchmarking, a consequent java framework [3] 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 does not impact the project so much.

The following figures provide a comparison of the profiler for the functional parser implementation and the object oriented implementation of the parsers for memory consumption and CPU load.

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

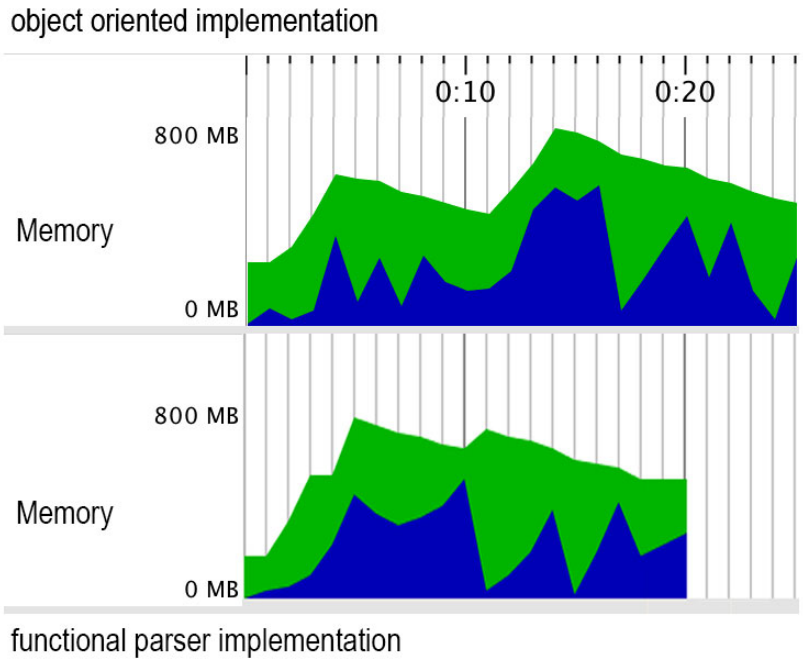

Figure 5.1: Memory consumption comparison. The blue area represents the memory consumed while the green area represents the free memory. The sum of both area represents the memory allocated to the computation effort. The x axis represents the amount of memory allocated while the y axis represents time.

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 one file at a time to provide actionable feedback as he or she design the grammar. Memory consumption should not be an issue when working with only one file, although, it is still interesting

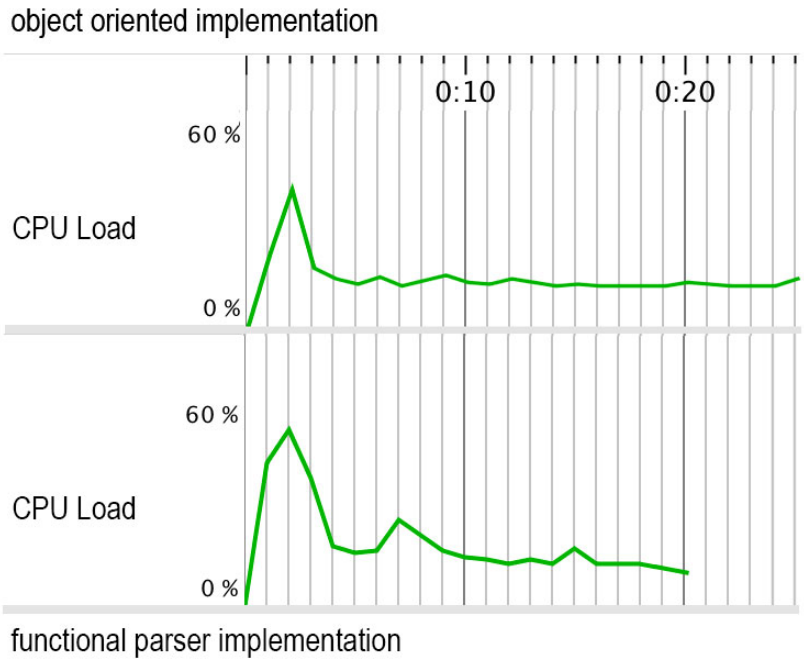

Figure 5.2: CPU load comparison in function of time.

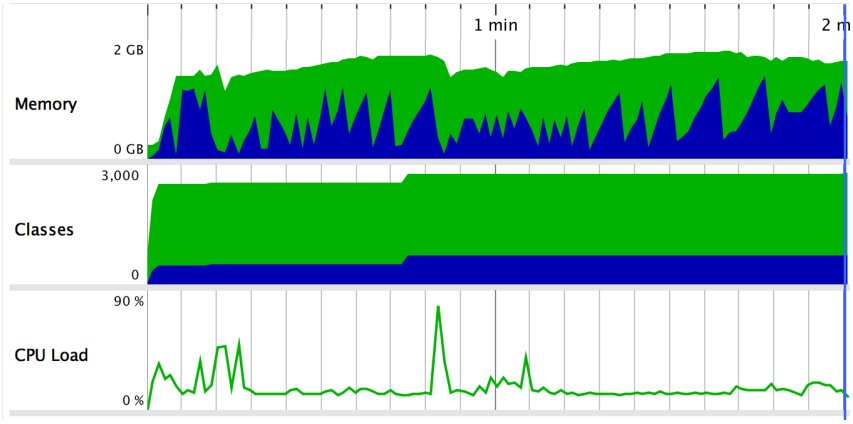

Figure 5.3: Profiler for the benchmark in debug mode

to analyze the behavior on larger files.

Despite this, 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 [11] 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, there's no way around an increased overhead in memory consumption. Despite those observation, one might argue that, even though 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.

That being said, given the overhead caused by the debugging effort, it doesn't really make sense to use it one more than one file at a time. A typical workflow would be to parse all the files and then start a debugging session one file at a time for each relevant files.

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

6

Future work

This chapter will discuss possible extensions and improvement upon the functionalities already available in our solution.

6.1 Debugger extensions

6.1.1 Enriching debug information with states

Autumn defines a special structure to handle global state manipulation in the form of a reversible stack representing the global state. Each entry represents an operation that manipulates the state. Every operation pushed on it can be reverted by popping it from the stack. The original implementation of Autumn uses this structure to represents an AST for java grammar. Since this information is already captured by the rest of the debug logic, it isn't useful to display it again in a separate view.

This global state can, however, be used by the user in any ways to capture custom context through the definition of custom parsers. Custom reversible structures can also be created to handle the state in different ways as well. For this reason, we created a hook in the form of an interface that requires the implementation of a function that returns the content of the state in a displayable format.

6.1.2 Turning the plugin into a full fledged independent software

As mentioned in section 3.2.1, the GUI implementation can run independently from the IDE, although it can't provide the functionalities that are derived from the IDE in

its current form (jumping to the definition, querying input, ...). Nonetheless, those functionalities could be implemented to be independent from the IDE altogether, turning the plugin into a staple for a full fledged debugging environment for Autumn grammars.

The benefits such environment would be to provide a solution for developing and debugging Autumn grammars that is not constrained by a single IDE anymore.

6.1.3 Working in tandem with the general purpose debugger

Another possible improvement would be to work directly with the IDE's implementation (community version of Intellij being open source) to layer the debugging logic on top of the general purpose debugger. An interesting behavior would be to display the information gathered by the debugger dynamically as it is being build and allow to stop the execution of the parse to then step inside the code instruction by instruction as needed.

As discussed in chapter 4, stepping through each instructions isn't a desirable property in general, but once the problematic parser has been identified through analysis of the higher level concepts that are parsers and production rules, providing the ability to step in the instructions themselves could provide the user with a more powerful, fine-grained tool to detect bugs.

6.1.4 Integrating object oriented implementation of the parsers

As presented in chapter 4, we implemented the hooks for the debug logic by rewriting the parsers from a pure functional definition to an object oriented one. This has been done by wrapping parser object definitions around their functional definitions. For performance issue in the form of megamorphic call sites where suspected, this implementation was meant to be temporary and therefore kept separated. In the light of the benchmark comparison between the two implementations done in chapter 5, the performance loss seems acceptable and so, both implementation could be merged together to reduce the overhead introduced by the new implementation.

7

Related work - state of the art

This chapter briefly discuss some of the related work and how they influenced our solution.

7.1 The Moldable Debugger

The moldable debugger is a framework to develop domain specific debugger. [8]

Their approach is heavily focused on the customizability of their solution. Developers develop their debugger by designing a domain-specific extension and an activation predicate. The domain-specific extension defines a custom user interface as well as debugging operations while the activation predicate detect when the domain-specific extension is applicable, allowing to switch from one domain to the other at run time.

While more general than our solution, the moldable debugger inspired our first idea of leveraging the power of the general purposed debugger and layering our solution on top. Chapter 4 discussed the it in more details.

7.2 Antlr Works

Antlr Works defines a debugging framework for Antlr grammars, a parser generator library. [18] [19]

It's main functionalities lies in the visualization of parser non-determinism and the definition of a time-traveling debugger that pays special attention to parser decision-making by visualizing lookahead usage and speculative parsing during backtracking.

The strength presented in this paper fueled a big part of our reflection about the specific needs of a developer for grammar debugging discussed in section 4.1.1 In particular, Antlr Works does :

- display the parse tree associated with a particular input sequence without requiring a complete grammar
- time travel: It can rewind and replay the parse multiple times without having to restart the actual parser.

As discussed before, our solution isn't a debugger strictly speaking, but it provides the developer with the same power as time travelling. By analyzing the parse trace and rebuilding the global state for a specific parser, one can effectively analyze the condition surrounding the invocation of the parser and the decision-making applied by the library to parse the input.

7.3 Ohm

Ohm is a parser generator consisting of a library and a domain-specific language.

The part that interested us was the debugging aspect of Ohm which propose an interesting visualizer illustrated by figure 7.1. The top left pane contains all the production rules defined by the grammar, the right pane displays the input while the bottom pane offer a visualization of the parse tree. It offer the ability to parse the input on the fly and allows to navigate the parse tree highlighting the rule chosen to parse each specific part of the input.

Ohm greatly influenced our choices for the GUI of our own implementation.

7.4 Debugger Canvas

Debugger canvas is a visual solution for debugging based on the Code Bubbles paradigm.. [10] It represents the executed method calls in the form of a directed

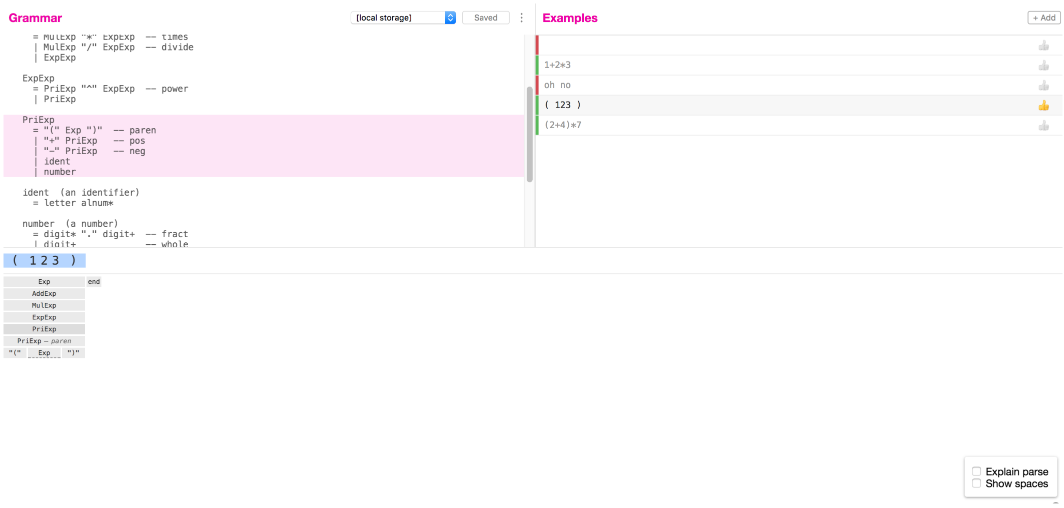

Figure 7.1: Ohm debugger visualization

graph, whose nodes, called bubble, displays the definition of the method. The current value of the different variables of a method can be accessed via a pop-up window. Each bubble is a Visual studio editor, allowing for modification on the fly. Figure 7.2

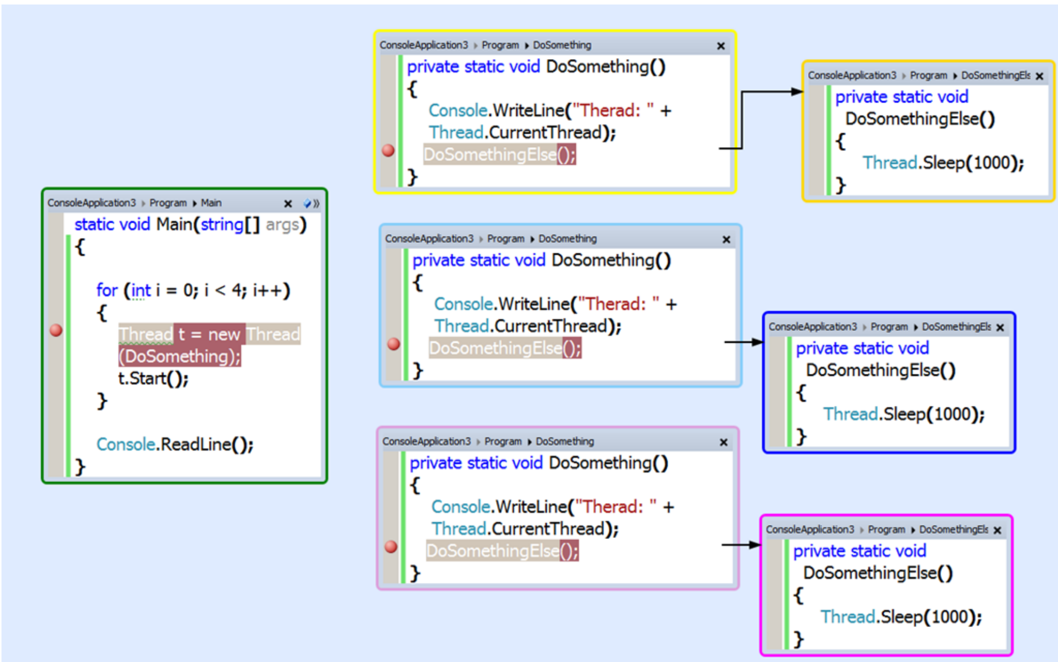

Figure 7.2: Debugger canvas example

Although it didn't directly influenced the development of our solution, Debugger canvas is nonetheless an interesting take on debugging visualization. We imagine it could be worth investigating further to see if some of its features couldn't represent the start of a new reflection for an improvement of our solution.

Bibliography

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

[2] *Psi cookbook: common operations for working with the PSI (Program Structure Interface).* http://www.jetbrains.org/intellij/sdk/docs/basics/psi_cookbook.html.

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

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

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

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

[7] Andrew Bragdon and et al. Code bubbles: rethinking the user interface paradigm of integrated development environments. *ICSE '10 Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, 1, May 01 - 08.

[8] Andrei Chiş, Tudor Gîrba, and Oscar Nierstrasz. The moldable debugger: A framework for developing domain-specific debuggers. *Combemale B., Pearce D.J., Barais O., Vinju J.J. (eds) Software Language Engineering. SLE 2014. Lecture Notes in Computer Science*, 8706, 2014.

[9] N. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2:113 – 124, September 1956.

[10] Robert DeLine, Andrew Bragdon, Kael Rowan, Jens Jacobsen, and Steven P. Reiss. Debugger canvas: Industrial experience with the code bubbles paradigm. *Software Engineering (ICSE), 2012 34th International Conference on Software Engineering (ICSE)*, 2-9 June 2012.

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

[12] Bryan Ford. Parsing expression grammars: A recognition-based syntactic foundation. *POPL '04 Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 39:111–122, 2004.

[13] David R. Hanson and Jeffrey L. Korn. A simple and extensible graphical debugger. *ATEC '97 Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 13–13, January 06 - 10, 1997.

[14] N. Laurent and K. Mens. Parsing expression grammars made practical. *SLE 2015 Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, pages 167–172, October 2015.

[15] N. Laurent and K. Mens. Taming context-sensitive languages with principled stateful parsing. *SLE 2016 Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, pages 15–27, October 2016.

[16] Nicolas Laurent and Kim Mens. Context-sensitive parsing through stateful parsing. *SPLASH 2016, Parsing@SLE workshop collocated with SLE 2016*, October 2016.

[17] Theodore Norvell. A short introduction to regular expressions and context free grammars.

[18] Terence Parr and Jean Bovet. Antlrworks: An antlr grammar development environment.

[19] Terrence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages (Pragmatic Programmers)*. Pragmatic Programmers. Pragmatic Bookshelf, 1 edition (may 27, 2007) edition, 2007.

[20] J. Rekers and A. Schurr. A graph grammar approach to graphical parsing. *VL '95 Proceedings of the 11th International IEEE Symposium on Visual Languages*, September 05 - 09, 1995.

[21] Lukas Renggli, Stéphane Ducasse, Tudor Gîrba, and Oscar Nierstrasz. Practical dynamic grammars for dynamic languages. *4th Workshop on Dynamic Languages and Applications (DYLA 2010)*, 2010.

[22] Social RTI, Health and Ph.D Economics Research, Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. *RTI Project Number 7007.011*, May 2002.