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

Contents

Chapter 1

Introduction and motivation

- debugger for autumn
- not a debugger a proprement parler
- it actually is more like a code anylizer but for the rest of this paper we will refer to it as a “debugging tool”
- autumn is a parser library
- what is special about autumn ?
- Ok autumn is cool, but have no debugger
- debugger are important why ?
- ok lets read about it
- language tools = pretty cool
- programmers are building domain-specific languages
- many mainstream programming and markup language possess context-sensitive features.
- however few solution exists for true context-sensitive parsing
- Autumn is a context sensitive parser combinator developped in Kotlin.
- particularity : it can deal with context sensitive grammars.
- debuggers = important tool = expose the underlying execution and helps detect issues and correct them
- traditional debuggers = generic mechanisms to explore and exhibit the execution stack and system state
- grammar developpement and parsing developper = domain-specific reasoning
- creates gap between the debugging needs and the debugging support.

- Writting and debugging grammar is tricky business.
- Errors throwed while executing grammar are rarely useful for debuging.
- we don’t know if the error comes from the grammar or the input.
- we propose a code inspection tool to help expose those higher level concepts
- code inspection tool and intellij IDE plugin designed to work with Autumn Grammars.
- provide the user with relevant information he can easily reason with to track and identify potential issues in a much easier way.
- to convince ourselves, lets have a look at an example. (example showing unhelpful error message and highlighting what kind of information would be more relevant)

The goal is to simplify the life of the developper when writting grammars. Just as developers use IDEs (integrated development environments) to dramatically improve their productivity, programmers need a sophisticated development environment for building, understanding, and debugging grammars.

Chapter 2

Based Material

2.1 Parsing expression grammar

- PEG = type of analytic formal grammar closely related to the family of top-down parsing languages
- similar to context-free grammars (CFGs) but PEGs cannot be ambiguous; if a string parses, it has exactly one valid parse tree.
- Formally, a parsing expression grammar consists of: A finite set N of nonterminal symbols. A finite set $\sum$ of terminal symbols that is disjoint from N. A finite set P of parsing rules. An expression eS termed the starting expression.
- 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, which is constructed in the following fashion:
  - An atomic parsing expression consists of: any terminal symbol, any nonterminal symbol, or the empty string.
  - Given any existing parsing expressions e, e1, and e2, 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
- desirable properties: closure under composition, built-in disambiguation, unifica- tion of syntactic and lexical concerns, and closely matching programmer intuition.
- struggle with left-recursive grammar rules : infinite recursion
- These include parser generators (like the venerable Yacc) and more recently parser combinator libraries [5].

The fundamental difference between context-free grammars and parsing expression grammars is that the PEG's choice operator is ordered.

- DSLs are generally very high-level languages tailored to specific tasks.

- DSLs are particularly important to software development because they represent a more natural, high-fidelity, robust, and maintainable means of encoding a problem than simply writing software in a general- purpose language.

Parsing 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 recog- nizer 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.

Grammar To understand grammars and to understand their capabilities and limitations, you need to learn about the nature of computer languages.

Represent grammar as a state machine; states and transitions labeled with vocabulary symbols. The transitions are directed connections that govern navigation among the states. Machine execution begins in state s0, the start state, and stops in s4, the accept state.

The machine can also generate invalid sentences, such as "Your truck is sad and sad." Grammatical does not imply sensible. For example, "Dogs revert vacuum bags" is grammatically OK but doesn't make any sense. The difference between the regular and context-free languages is the differ- ence between a state machine and the more sophisticated machines in the next section. The essential weakness of a state machine is that it has no memory of what it generated in the past. Trees Such trees are called derivation trees when generating sentences and parse trees when recognizing sen- tences. It turns out that the humble stack is the perfect memory structure to solve both word dependency and order problems.4 Adding a stack to a state machine turns it into a pushdown machine. A state machine is analogous to a stream of instructions trapped within a single method, unable to make method calls. A pushdown machine, on the other hand, is free to invoke other parts of the machine and return just like a method call. The stack allows you to partition a machine into submachines. These submachines map directly to the rules in a grammar.

Ambiguousity As we all know, English and other natural languages can be delight- fully ambiguous. Any language with an ambiguous sentence is consid- ered ambiguous, and any sentence with more than a single meaning is ambiguous. Sentences are ambiguous if at least one of its phrases is ambiguous. Here is an ambiguous faux newspaper headline: "Bush appeals to democrats." In this case, the verb appeals has two mean- ings: "is attractive to" and "requests help from." This is analogous to operator overloading in computer languages, which makes programs hard to understand just like overloaded words do in English.

For example, a syntax diagram for C can generate statement i*j; following the path for both a multiplicative expression and a variable definition (in other words, j is a pointer to type i). To learn more about the relationship of ambiguous languages to ANTLR grammars,

Complex language generation enforces word dependencies and order requirements. Your brain enforces these constraints by subconsciously creating a tree structure. It does not generate sentences by thinking about the first word, the second word, and so on, like a simple state machine. It starts with the overall sentence concept, the root of the tree structure. From there the brain creates phrases and subphrases until it reaches the leaves of the tree structure. From a computer scientist's point of view, generating a sentence is a matter of performing a depth-first tree walk and "saying" the words represented by the leaves. The implicit tree structure conveys the meaning. Sentence recognition occurs in reverse. Your eyes see a simple list of words, but your brain subconsciously conjures up the implicit tree structure used by the person who generated the sentence.

2.2 difficulties

there are two very common difficulties encountered by grammar developers: Understanding why a grammar fragment results in a parser nondeterminism and determining why a generated parser incorrectly interprets an input sentence.

2.3 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.

2.3.1 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)

2.3.2 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.

2.4 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 ?

2.4.1 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.

Chapter 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.

3.1 side note

- at first : we wanted to work on top of the IDE debugger
- but from plugin it is difficult to access the implementation for the debugger
- made me think about what is important for debugging grammar ?
- important thing is to be able to see the execution trace of the parsers
- being able to look at the parse state the system was in when the parser was called
- to be able to see the portion of the input matched by the parser
- it is possible to access all this information without stopping the execution
- the syntax tree generated by the parse can by analyze and we can simulate time traveling in the parse execution

3.2 Overview of the fonctionality

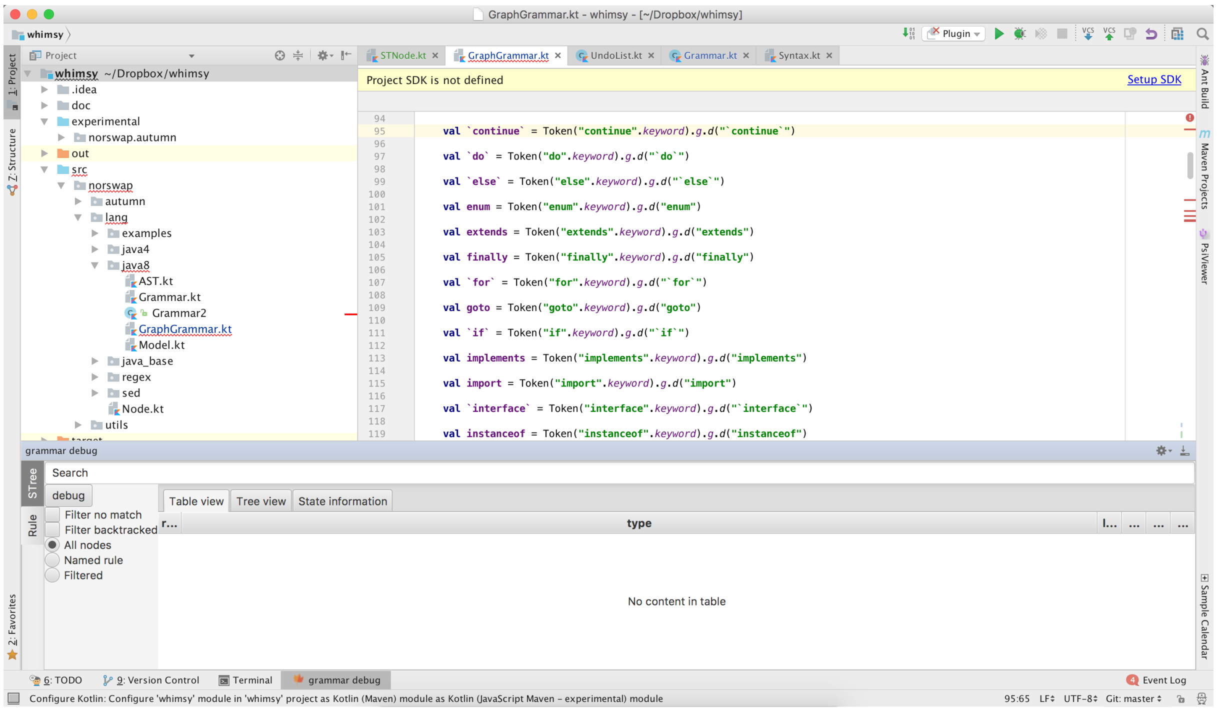
At this point we know the context, we know autumn and we know the motivations. From here we don't discuss why but how.

- Autumn implementation of parser as function -> changed into objects to create hooks for debugger.
- neet to rewrite java grammar to reflect changes in the parsers implementation
- debugger is a function that is hooked on the execution of each parsers
- during execution a syntax tree is build
- each node of the syntax tree contains informations about the parse state at the moment of the parser invocation

- those information can then be processed and displayed on a GUI for the developper to reason about
- ULM chart of the debugger and how it connects to autumn
- debuging tool = hook on top of parser's definition
- builds up a syntax tree
- which is then read and displayed by a GUI
- the GUI is implemented as a IDE plugin

3.2.1 Intellij plugin

- intellij plugin is essentially a panel that can be docked on top of the IDE
- it presents the data in several different forms
- the data can then be filtered to pinpoint specific problems



Project SDK is not defined

```
    val `continue` = Token("continue".keyword).g.d("`continue`")

    val `do` = Token("do".keyword).g.d("`do`")

    val `else` = Token("else".keyword).g.d("`else`")

    val enum = Token("enum".keyword).g.d("enum")

    val extends = Token("extends".keyword).g.d("extends")

    val finally = Token("finally".keyword).g.d("finally")

    val `for` = Token("for".keyword).g.d("`for`")

    val goto = Token("goto".keyword).g.d("goto")

    val `if` = Token("if".keyword).g.d("`if`")

    val implements = Token("implements".keyword).g.d("implements")

    val import = Token("import".keyword).g.d("import")

    val `interface` = Token("interface".keyword).g.d("`interface`")

    val instanceof = Token("instanceof".keyword).g.d("instanceof")
```

No content in table

Configure Kotlin: Configure 'whimsy' module in 'whimsy' project as Kotlin (Maven) module as Kotlin (JavaScript Maven – experimental) module

Figure 3.1: IDE GUI for Autumn's debuggin tool

3.2.2 Views

Table view - Execution trace The table view highlight the execution sequence of the parsers

- execution trace
- allow to see the sequence of parsers that were called chronologically

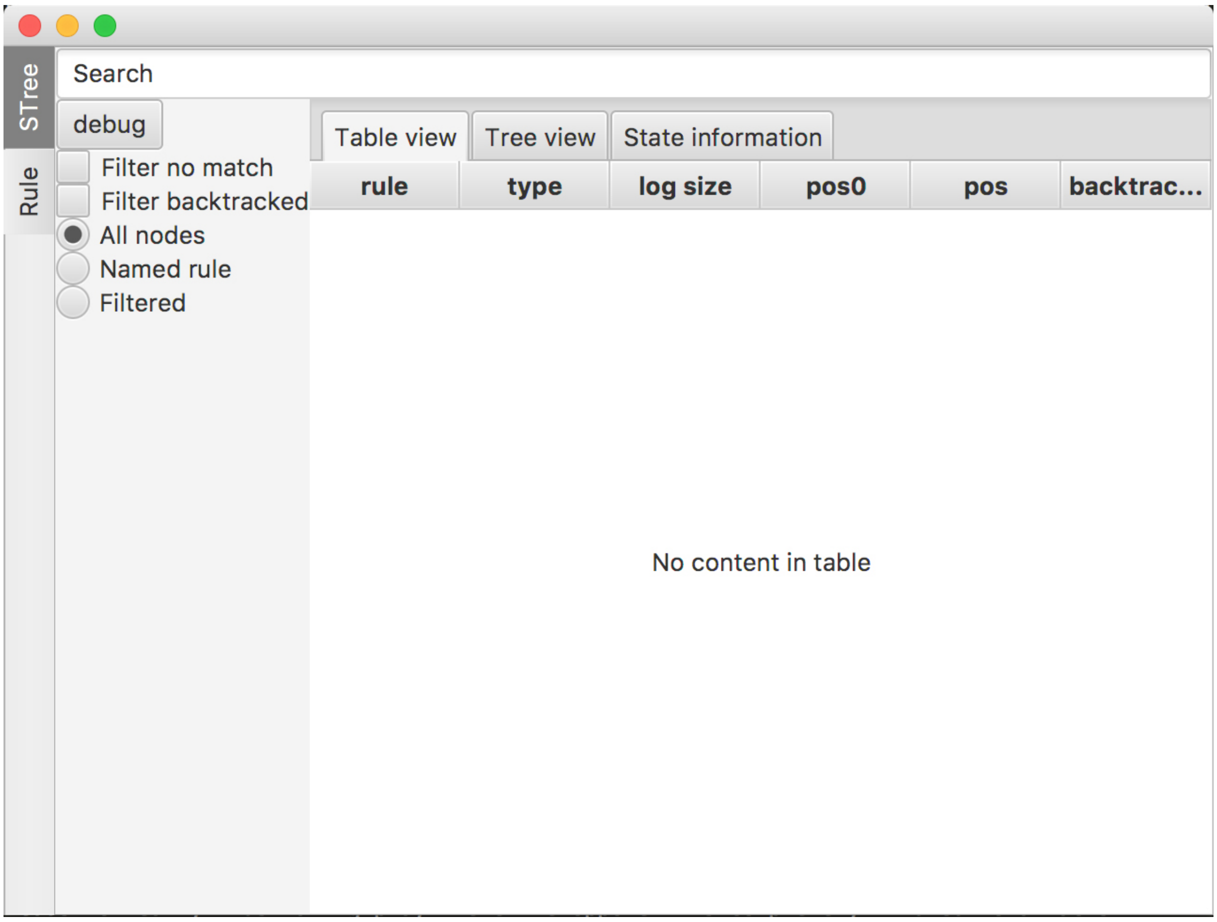


Figure 3.2: Stand alone view - the debugger GUI can be executed independetly from the IDE albeit some restrictions

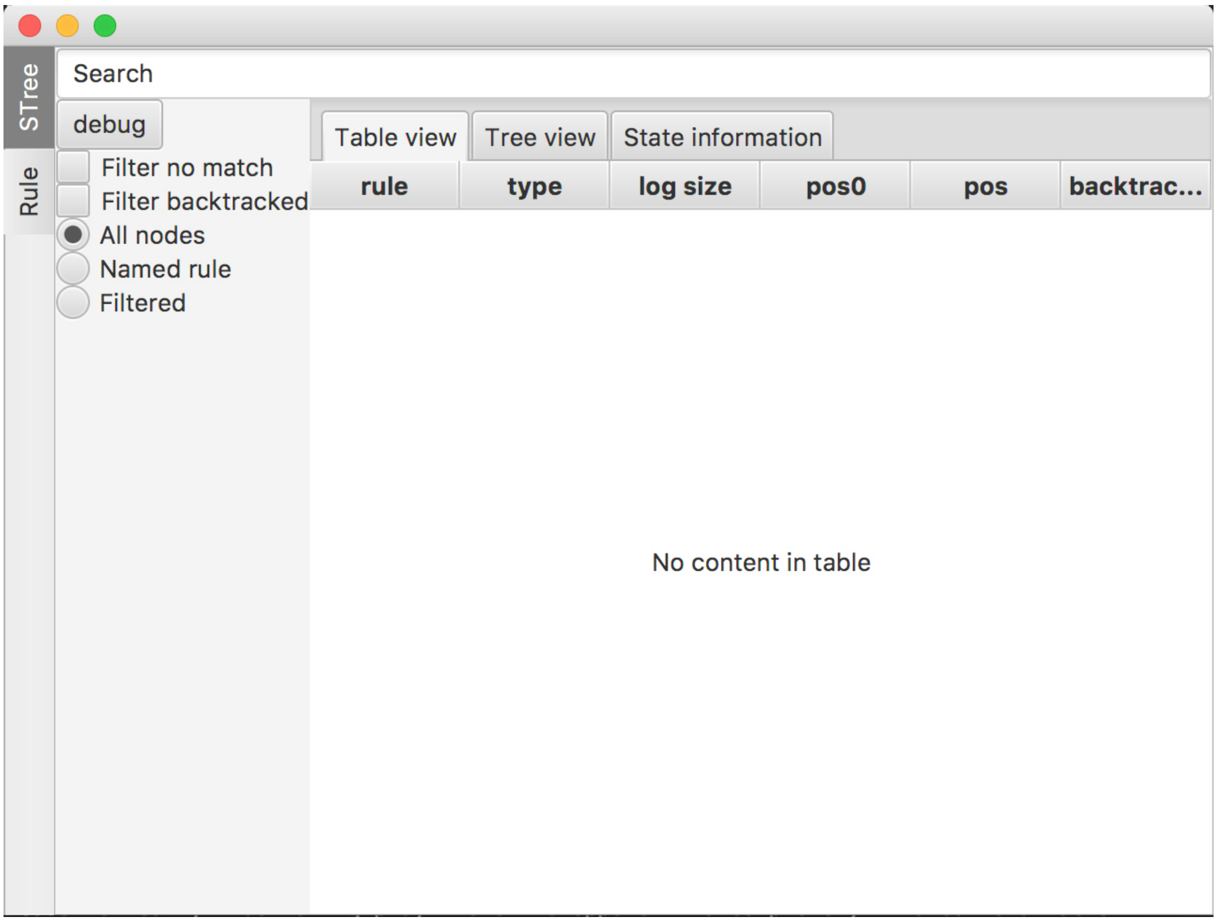
Tree view - Syntax tree The tree view highlight the hierarchy of the parsers

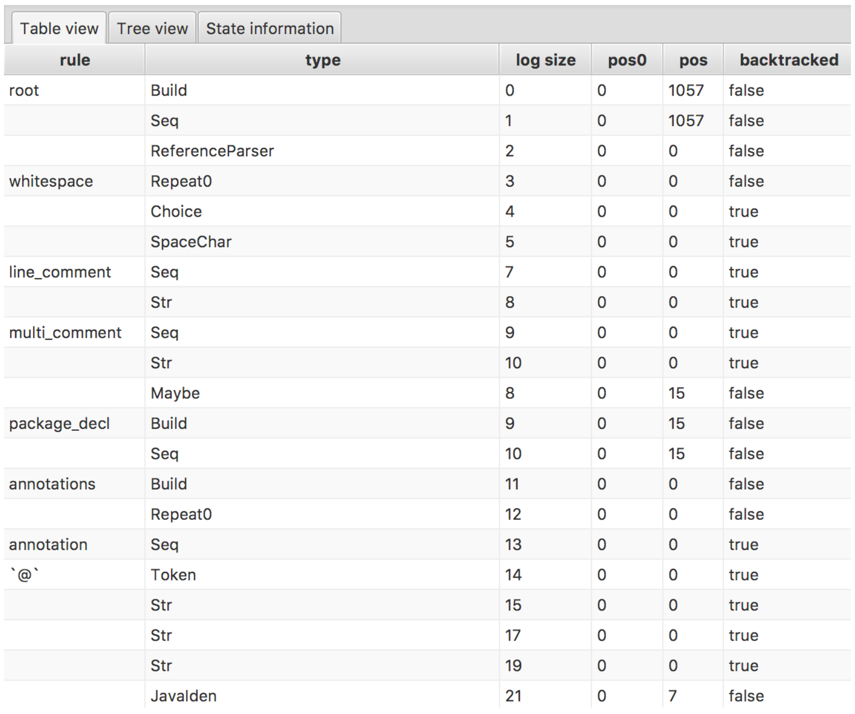
- highlight the hierarchy between parsers and rules

State information This tab display the current state

3.2.3 Filtering the informations

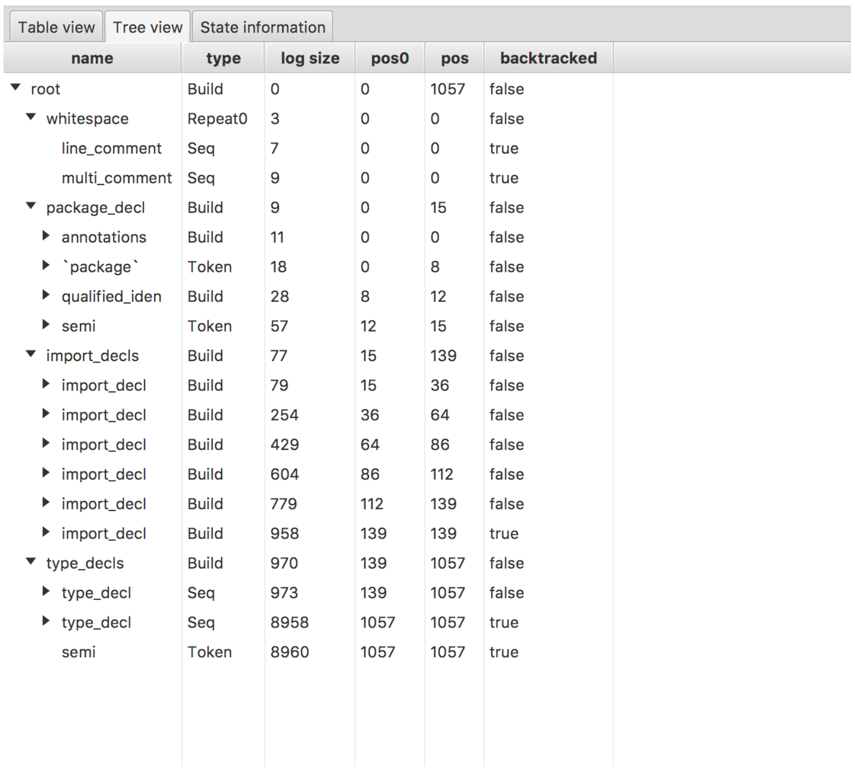
- all this information is overwhelming and not very usefull (like a dump of info)
- filter allow to select the relevant informations the user might be reasoning with
- filter possibilities
- unamed: unamed parser which doesn't match the rule one to one
- backtracked
- no match : nodes that didn't match anything in the input but succeded





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 3.3:



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 3.4:

Chapter 4

Implementation

4.1 Debuggin tool

4.1.1 New implementation for the parsers

- autumn parsers were first implemented as boolean functions
- it was difficult to create a debugger for this implementation
- parsers are now objects that inherit from a “parser” super class
- each parser object wrap around the parser function
- this implementation was called “naive” because performance issues due to megamorphic call sites were expected with the creation of so many objects.
- benchmark revealed that the performance difference was neglectable.

4.1.2 New grammar and model Compiler

- because of the new implementation of the parsers, a new grammar needed to be written
- writting grammar is time consuming and prone to error business
- also because we expected performances issues with the new implementation of parsers, we wanted to compare both implementation
- to make sure that both implementation was comprehensive and correct, a single model of the grammar as beed created
- this model has been feeded to two compilers
- the first regenerated the same grammar using the parser functions
- the other generated a new grammar using the parser objects.
- the validation section will discuss the results of the benchmark.

- note : this structure model/compiler allow us to change the implementation of some/all the parsers without the need to modify or rewrite the grammar from scratch which would be very tedious.

- schemas qui explique la structure model/grammar.

4.1.3 Syntax Tree generation

- autumn doesnt build a syntax tree, instead it builds an AST

- the AST abstract some of the nodes together to reduce the general size of the tree

- for debuging purposes, it is needed to have a direct mapping between the nodes of the tree and the rules of the grammar.

- the syntax tree is build by the debuging logic, separately from autumn AST

- if we do not debug, the ST is not built

- It has been made that way because I wanted to isolate the implementation of the debugger from the implementation fo autumn

- 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.

4.1.4 Debug Nodes

- Debugger logic is hooked on the parsers invocation

- at each invocation the debugger create a node with information about the state of the parse as well as the position in the input.

- There is therefore a node per parser that form the syntax tree

- the state is not copied in each node of course

- the state works as a log with each entry the modification made to the parse state

- each entry can be undone and redone at will

- each node remember the size of the log at the invocation of the corresponding parser

- the state can therefore be regenerated for analyze purposes

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

4.1.5 Syntax tree nodes

- Each nodes of the syntax tree is created by the debugger at the invocation of the parsers
- 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
- the nodes works as a linked list. it has a reference to its parent and children.

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[**?**] 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

Chapter 5

Future work

5.1 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.

5.2 Debugger extension

5.2.1 Adding hooks

- access more informations during debug
- create new state structure

5.3 Autumn related

5.3.1 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

Chapter 6

validation

6.1 Benchmark

Benchmark with consequent java code. Because of megamorphic call sites : expected the new implementation of the parsers to be much slower. in practice its not the case.

6.2 Debugging Benchmark

Avec l'ajout de la logique de debug, voila les perfs:

- vitesse d'execution comparée avec et sans le debug
- memoire consomée par les structures de debug. (discuté si c'est un gros desavantage ou non, si oui apporter une solution pour travailler avec de gros fichier input.)

6.3 Debugger in practice

6.3.1 Examples

- case study 1 : error in grammar
- case study 2 : error in input
- case study 3 : error in parser definition
- show how the debugger helps detecting errors in the differrent case studies

Chapter 7

Technical background

This chapter will present the important technical aspects involved in the developpement of the solution. Feel free to skip it and come back to it as needed. (I will reference the different section at the relevant places throughout the text) (for this chapter, Im not sure if I should have it or directly take its section and put them in the relevant places (i.e. put the IDE plugin section in the plugin chapter))

- not sure if I shouldnt just discart this entire chapter because its not really relevant and the reader can find those information himself ?
- anyhow, clearly the less important chapter.

7.0.1 Plugin developpement for Intellij IDE

Present briefly the structure of Intellij IDE and plugin structure, limitation and difficulties ?

7.1 Kotlin

Might not be super relevant either, but since everything is made in kotlin, maybe its interesting to at least mention it, and quickly highlight strenght of the language VS java ?

7.2 TornadoFX framework

High level presentation of important features - how it works and difference with javafx / swing ?

Chapter 8

Related work

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

8.2 Antlr [13]

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

8.2.1 Antlrworks

- development environment for ANTLR grammars - parser nondeterminism visualizer (syntax diagrams and a time-traveling)
- grammar-aware editor with refactoring and navigation features,
- a grammar interpreter
- domain-specific grammar debugger.

RAPID PROTOTYPING - time travel My solution doesnt actually do time travel but provides as relevant information by being able to regenerate a state for a given parser and analyze the state of the system at the time of its execution

8.3 Ohm

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