Progger: Programming by Errors (Work In Progress)

Alan T. McCabe
Lund University
alan.mccabe@cs.lth.se

Emma Söderberg
Lund University
emma.soderberg@cs.lth.se

Luke Church
Lund University / University of Cambridge
luke@church.name

Abstract

This paper describes a work in progress implementation of a programming tool that puts errors and their provenance at the forefront of the interaction between a developer and a compiler. We discuss the motivation for such a tool, it's design and implementation, and reflect upon avenues for further research which it can facilitate.

Keywords

Error messages, Compilers, Conversations, Developer Tools, Usability, Communication breakdown, Conversational repair

1. Introduction

For novice programmers, compiler errors are a common occurrence as they work through the process of familiarising themselves with the syntax of a programming language. They write some code, execute it, and any errors that result can then be used to inform their next steps. Sometimes, however, unclear error messages can leave them at a loss as to how to proceed or, even worse, lead them down a wrong path altogether if the actual source of the error ends up being in a different location in the code from the line that the compiler flagged as problematic (Becker et al., 2019). In this case, the respective understanding of developer and compiler are no longer in alignment (Henderson & Harris, 2011) - looking at this interaction through the lens of a conversation (Dubberly & Pangaro, 2009), it can be said that a communication breakdown (Beneteau et al., 2019) has occurred. This conceptual approach is explored at length in (Church et al., 2021), which proposes a theoretical 'tool for thinking with', this paper explores a possible application of that framework.

In the event of a breakdown in understanding between two human participants, the focus would shift towards a meta-conversation about the conversation itself (Dubberly & Pangaro, 2009). The participants would attempt to "repair" the breakdown, bringing all parties back into alignment before returning to the primary topic. A compiler, however, is much less forgiving. If a developer fails to understand a statement from the compiler, the onus is on the developer to seek out clarification and decipher exactly what is meant, with nothing from the compiler by way of assistance.

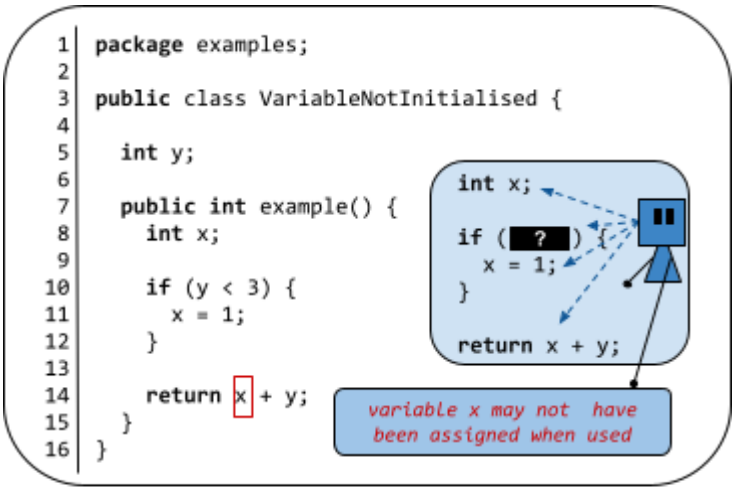

Figure 1. Java code example with the compiler and its view illustrated with blue figures.

As an example, if we consider the Java code in Figure 1, it has a small method (`example`) using two variables; one field (`y`) declared in the class and one local variable (`x`) declared in the method.

The local variable is “possibly assigned” before it is used in the return statement (on line 14). If we know that the field (`y`) is given a default value of zero we can see that the condition in the `if` statement will always be true, and consequently the local variable will be assigned.

However, the compiler (the little blue robot-figure operating in the blue box) has a different view, where it considers the declaration and uses of the local variable (indicated with dashed lines) to see if it is assigned before it is used. When considering this, the compiler can not determine the value of the condition[1] (the black box with the question mark), and as a consequence it reports an error where the local variable is used. If we let the javac compile process the code example via its command line interface, we get the following result, providing a message and pointing to a position:

```
VariableNotInitialised.java:14: error: variable x might not have been initialized
    return x + y;
           ^
1 error
```

This is where the interaction with `javac` ends - we get no more assistance and the burden of determining that the fix may be to assign a value declared on line 8 is on us. If we consider this as a communication breakdown, how can we provide assistance for repair?

With this in mind, we have developed a prototype for the exploration of the provenance of errors that extracts extra information from the compiler about the source of an error and presents it to the user. Given this relative wealth of information, we will discuss how it can be used to repair a breakdown in understanding and ask the question “how would the interaction look if we explicitly designed for mitigation of breakdowns in the conversational alignment between the developer and the compiler, as well as support for repair?” - In other words, can we design a tool that enables the user to engage in “programming by errors”?

2. Background

For ease of access to additional information within the compiler, we decided to build a small extension of a compiler based on a *reference attributed grammar* (RAG). An explanation of this formalism will be provided in the following section, as well as the motivation behind our selection of a RAG-based compiler as a basis for the Progger system.

2.1 Attribute Grammars

The formalism of *attribute grammars* (AGs), as introduced in (Knuth, 1968), is a means of extending a context-free grammar to allow for a declarative description of context-sensitive elements of the grammar. This is achieved by attaching *attributes* to non-terminal nodes of the abstract syntax tree with rules for their evaluation, with attributes categorised as either *synthesised* or *inherited* depending on whether they are used to propagate information upwards or downwards through the tree respectively. For example, consider a simple context-free grammar describing addition and subtraction expressions. The notation used here is a simplified form of the JastAdd notation used to specify the abstract syntax tree (AST) model. Some notation from the concrete grammar is also used for clarity, such as the explicit inclusion of ‘`+`’ and ‘`-`’ tokens which would normally be omitted from the abstract grammar. Certain object-oriented concepts are used, such as abstract classes and the use of inheritance, denoted here in the form `<Subclass>: <Superclass>`.

```
abstract Expr
Add: Expr     → Left:Expr + Right:Expr
```

[1] The compiler is performing a static analysis without running the code. One consequence is that it has little knowledge about values of variables and results of expressions. It could possibly infer the result of the condition in the example code but this kind of analysis is typically not done by compilers for the Java language.

```
Sub: Expr     → Left:Expr - Right:Expr
Numeral: Expr → ℕ
```

Where ℕ is the set of all natural numbers, ℕ = {0, 1, 2, 3, 4, …}. To represent the numerical value of an expression, a *synthesised attribute* (val) may be attached to each of the **Expr** nodes and a corresponding *equation* defined for each production in the grammar as follows:

```
syn int Expr.value();[2]
eq Add.value()      = getLeft().value() + getRight().value();
eq Sub.value()      = getLeft().value() - getRight().value();
eq Numeral.value()  = ℕ;
```

Given the input string "1+2-3" within this grammar, the syntax tree in Figure 2 may be derived with the appropriate attribute values as defined by these equations.

The *synthesised* attribute, `value`, can be seen to be calculated based on information provided by the child of the node it is attached to, thereby propagating information up the tree. The evaluation of attributes occurs on-demand, therefore the values of the child nodes would not be calculated until the result is required by the parent node. It is also of interest to propagate information downwards through the tree, however. This is achieved using *inherited* attributes, where the equation is defined in an ancestor of the node containing the attribute. This will be explained in greater detail in the next section.

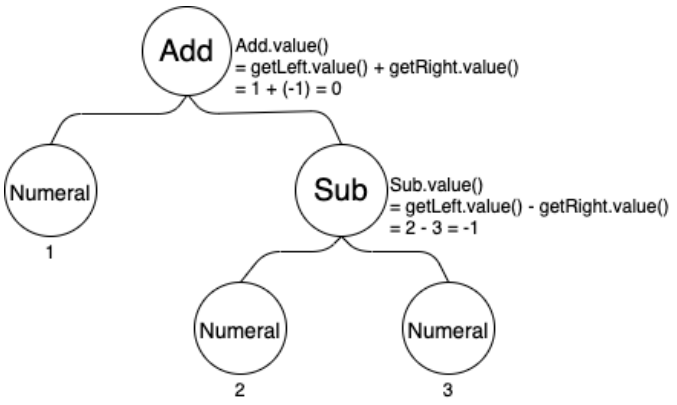

Figure 2. Derived abstract syntax tree.

2.2 Reference Attribute Grammars

The AG formalism has been further extended by the introduction of *reference attributed grammars* (RAGs) as described in (Hedin, 2000). The primary addition in this extension is to facilitate the referencing of objects by attributes, thereby allowing a *reference attribute* to form a link between one node of the tree and another node at an arbitrary distance from it. This allows for multiple benefits, such as the superposition of graphs over chains of use-def relationships or inheritance structures.

Consider an extension to the previous example that allows for the assignment of values to variables and a predefined print function. This can be achieved by introducing the concept of a block (**`Block`**) composed of a list of statements (**`Stmt`**). A statement may be a variable declaration (**`Decl`**), an assignment of a value to a previously declared variable (**`Assign`**), a call to print the result of an expression (**`Print`**), or an expression (**`Expr`**). We also introduce a **`Use`** node to represent calling a variable by reference, and let ID correspond to a string of arbitrary length:

```
abstract Stmt
Block: Stmt  → Stmt*
Decl: Stmt   → ID Expr
Assign: Stmt → ID Expr
Print: Stmt  → Expr
```

```
abstract Expr
Add: Expr     → Left:Expr Right:Expr
Sub: Expr     → Left:Expr Right:Expr
Use: Expr     → ID
Numeral: Expr → ℕ
```

[2] The notation presented here is a simplified form of the JastAdd syntax. A detailed description can be found in the JastAdd reference manual: https://jastadd.cs.lth.se/web/documentation/reference-manual.php

In order to associate variable uses with their declarations, we must introduce several new attributes. All **`Stmt`** nodes will require an attribute (`declares`), that, given an ID as a parameter, will return true if evaluated to a **`Decl`** statement with a matching `ID` node, or false for all other **`Stmt`** nodes:

```
syn boolean Stmt.declares(String id)     = false;
eq Decl.declares(String id)              = getID().equals(id);
```

Another equation may now be defined on **`Use`** nodes, decl, that can be used to calculate a reference to the variable declaration and thereby obtain it's value. This may be accomplished by the use of an *inherited attribute*, lookup, which we can use to implement the lookup pattern which was previously defined for JastAdd-style RAGs (Fors et al., 2020). This attribute is attached to the **`Use`** node, however its equation is defined further up the tree, within the **`Block`** node:

```
eq Use.value()      = decl().val();
syn Decl Use.decl() = lookup(getID());
inh Decl Use.lookup(String id);
inh Decl Block.lookup(String id);
eq Block.getStmt(int i).lookup(String id) = {
  for (Stmt s : getStmts()) {
    if (s.declares(id)) {
      return s;
    }
  }
  return super.lookup(id);
}
```
[3]

As several attributes are dependent upon each other in this example, the dynamic dependencies must be calculated on-demand when a reference attribute is evaluated. An evaluation stack is used during this calculation which can effectively be compared to a call stack, where an attribute pushed on to the top of the stack can be understood as having been called by the attribute immediately below it.

Consider an input to the compiler in the form of a small program and the resulting syntax tree as shown in Figure 3, with additional information included to highlight dependencies. In the interest of clarity only the dependencies for a single attribute are presented: those which can be used to return the matching declaration for the last reference to "a" in the line: "`print b + a;`".

[3] As can be seen, this code snippet includes some imperative code. This is considered to be valid within the JastAdd system, as long as there are no externally visible side-effects.

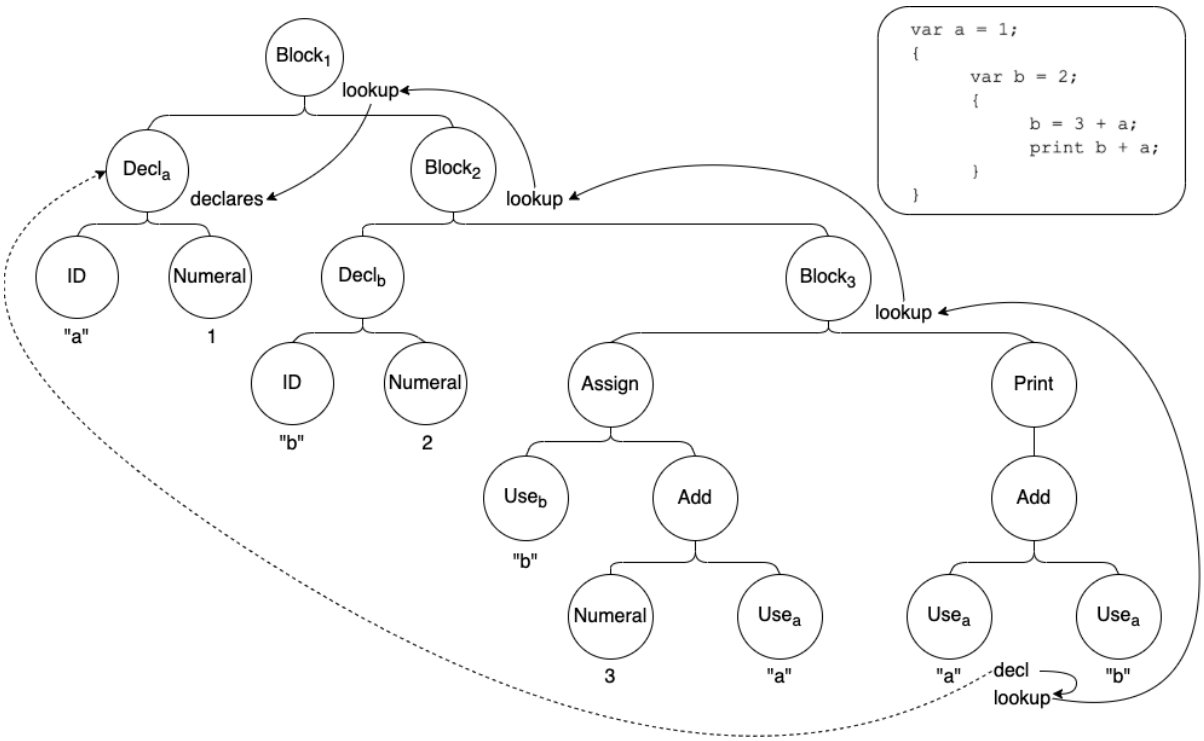

Figure 3. Extended abstract syntax tree with arrows showing dependencies.[4]

The ability of *inherited* attributes to propagate information down the tree is highlighted here. The `ID` value "a" is first propagated up through the **`Block`** nodes via the `lookup(String id)` attribute until a local declaration can be found. A reference to the **`Decl`** node is then passed back down the tree and assigned to the `decl()` attribute in the **`Use`** node. The call stack for this Use.decl attribute instance is presented in Figure 4, where the stack is shown at points in the evaluation immediately preceding the top-most object on the stack being popped. As the computation and evaluation of attributes can be expensive in a large syntax tree, *caching* has also been introduced in order to increase the efficiency of subsequent calls to an already accessed attribute. It assumed for this example that none of the attributes on the evaluation stack have been previously cached.

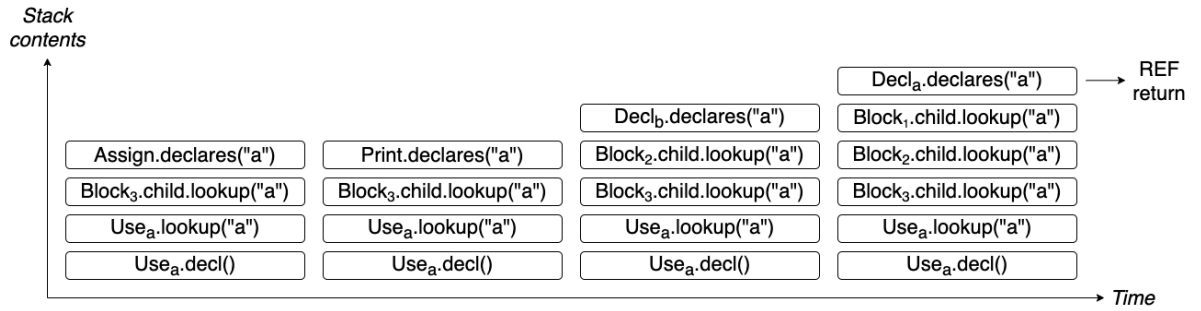

Figure 4. Evaluation stack

Reference attributes can be used for various semantic analyses, for instance, in the evaluation of compile time errors. RAGs have also been extended with several additional concepts, for instance, that of the *collection attribute* (Magnusson et al., 2009). These are attributes where the value is

[4] The subscript numerals 1, 2, and 3 on the `Block` nodes in this diagram correspond to the level of nesting that the respective `Block` occurs at. Subscripts a and b on `Decl` and `Use` nodes refer to the variable name that is declared or used.

defined by a combination of contributions from other nodes within the tree. Other extensions include circular attributes, rewrites, higher-order attributes, and incremental evaluation, however as they are not pertinent to this paper they will not be discussed here.

2.3 The JastAdd System

The efficacy of RAGs has been demonstrated by their implementation in the meta-compilation system JastAdd (Hedin & Magnusson, 2003) and the subsequent implementation of, for instance, the extensible Java compiler, ExtendJ (Ekman & Hedin, 2007), built upon the JastAdd system. Due to the extensibility and modularity of JastAdd, the resultant implementation of the Java specification in ExtendJ provides a convenient entry point to attribute evaluation within a full Java compiler, and is therefore a valuable tool when seeking greater insight into error provenance.

The JastAdd system supports *aspect-oriented programming* (Kiczales et al., 1997), which is reflected in the organisation of the Java specification implementation within ExtendJ. Extension of existing classes within the AST, and the addition of new ones, is supported by the use of inter-type declarations in JastAdd aspect modules, defined in files using the .jrag extension. Aspects use a Java-like syntax to allow for additional classes to be defined within the AST while *attributes* weave additional code into existing generated class files. Aspects are used to group common behaviour together under an easily recognisable descriptor - for example, type checking behaviour within the ExtendJ compiler is grouped together in the TypeCheck.jrag aspect file.

The tracing system within JastAdd, first introduced in (Söderberg & Hedin, 2011), provides trace events generated at various stages of compilation to perform this attribute tracking. For the previous example in Figure 4, this may be illustrated as in Figure 5 by including the trace events generated at each stage of the stack operations. A TOKEN_READ event is first generated upon scanning of the token. Subsequently, as attribute evaluation occurs, a COMPUTE_BEGIN is generated as each new attribute is placed on the stack. Once the reference is calculated, the attributes are popped one by one, generating a COMPUTE_END event as they are to signify that a value has been obtained. At any point in the evaluation an attribute that has been pre-calculated may have it's value read from the cache, at which point a CACHE_READ event would be triggered instead.

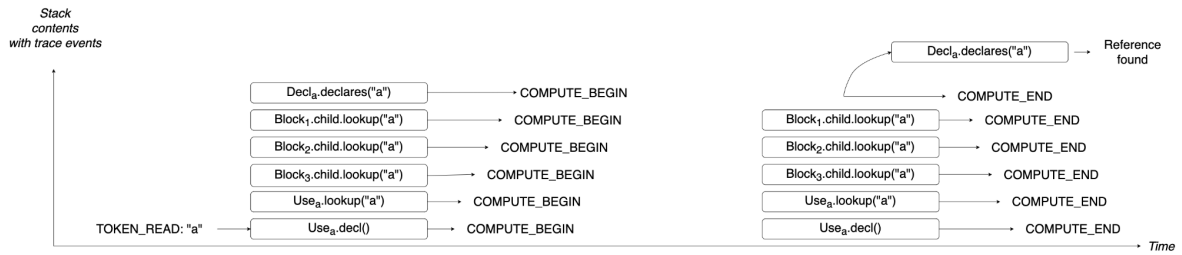

Figure 5. Evaluation stack with trace events

One benefit of using a RAG-based compiler to build the prototype is the aspect-oriented nature of JastAdd, which groups computations by behaviour, the utility of which we will elaborate on in Section 3.3. Another benefit is the evaluation and subsequent tracing mechanism that is available. By being able to hook into the evaluation stack, this allows us to present to the user not only an error message, as in a conventional system, but also a tree structure showing exactly *where* in the code the compiler was looking when the error occurred.

3. The Progger Prototype

In this section, we will explain how we designed the prototype. A literal application of the conversational metaphor in this design would result in an interaction that was similar to a conversational agent, whilst interesting as a possibility this would create a very significant implementation challenge to avoid uncanny valley effects (Mori et al., 2012). Instead we aim here to implicitly support the conversational nature of the interaction outlined in the breakdown and repair properties in Section 1. We present the prototype in terms of its client-server architecture (Section 3.1), how we extract error details in the server (Section 3.2), and then how we bring out the details from the compiler to the user in the client (Section 3.3).

3.1 Architecture

To facilitate experimentation in conversational compilers, we elected to build an architecture based on the client-server model, illustrated in Figure 6. The client is a simple web application with a file picker which allows the user to select a .java file, which is then rendered in the client with some simple syntax highlighting to emphasise Java keywords. When the "Compile" button is clicked, the file is uploaded to the compiler service via REST which then compiles the source code and returns a data structure with key information accessed during compilation and the corresponding token locations in the code.

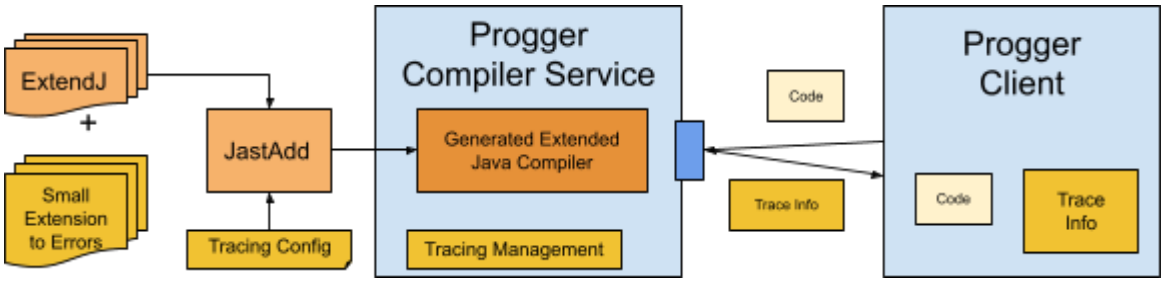

Figure 6: Architecture overview of the Progger prototype.

3.2 The Progger Compiler Service

To extract and present a greater amount of information with regards to error provenance, it is necessary to establish an understanding of what information the compiler was making use of when it encountered the error. For this purpose, a Java compiler based on RAGs was selected. Where a non-RAG based compiler may give us access to just the call stack, RAG based compilers make use of declaratively defined objects called *attributes* to perform computation, and consequently can allow access to the attribute evaluation stack. This presents a finer level of information about the computation, with dependencies between units of computation being exposed.

In the RAG-based compiler that we have selected, these computation units are also grouped into aspects, which are clearly labelled collections of attributes and behaviour. This allows us to link operations on the attribute evaluation stack to logically named stages of the compilation process, and thus to map these operations to conversational statements that allow for a greater degree of exploration by users regardless of their level of knowledge about compilers. The following sections will give an overview of RAGs, as well as the attribute grammars that act as their foundation.

3.2.1 Extracting Compiler Trace Details

During development of the prototype, the tracing system in JastAdd was updated to contain additional *aspect* information within the trace events, as well as TOKEN_READ and several events related to the evaluation of collection contributions. As attributes are defined within aspects, COMPUTE events generated by the tracer are now able to report the name of the aspect from which the attribute that is

being computed originates. This allows us to link more generic attribute names to the context in which they are being accessed - any attribute defined in the TypeCheck aspect, for example, may be easily inferred to have been accessed by the compiler when checking the type of an object.

A key component of the prototype described in this paper is the ability to "hook into" a compiler in order to extract more information relating to errors. Specifically, within ExtendJ a "problems" *collection attribute* is defined in the root node of each compilation unit, which is typically a representation of the code within a Java file. Upon failure of some check in the compiler, a Problem object is created and contributed to the problems collection containing information such as an error message, location where the error was discovered, severity etc. By tracking the attributes that are evaluated in the calculation of a contribution to the problems collection, it becomes possible to link an error to various contributing locations in the code.

The prototype achieves this by listening for events triggered at the beginning of a collection contribution check, signified by a CONTRIBUTION_CHECK_BEGIN trace event. For example, a node in the tree may contribute a Problem to the problems collection in the event that a return type does not match the method signature. Upon reaching the return statement in a method block, the compiler will begin evaluating this contribution and trigger a trace event signifying so. Once a contribution check begins, we start constructing a tree of the attributes that the contribution evaluation is dependent upon. To do this, any other attribute that is calculated or that has it's cached value accessed in the process of evaluation is stored in an internal data structure. At the conclusion of the contribution check a different event is generated depending on whether or not the contribution condition is matched. In the event of a match, the dependency tree is saved and returned to the application once compilation is complete, otherwise it is simply discarded.

3.2.2 Service REST API

Progger makes use of a simple REST service built upon the lightweight Spark framework[5] to convey the results of a compilation back to the client. This consists of an array of any errors encountered by the compiler mapped to a JSON format as follows:

```
{
 "message":  [compiler error message],
 "fileName": [file name],
 "location": [line number],
 "severity": [warning/error],
 "rootNode": [originating attribute node]
}
```

The root serves as the starting point for the attribute dependency tree, with each node containing the following information:

```
{
 "name":     [attribute name],
 "aspect":   [name of aspect the attribute is associated with],
 "location": [Token locations associated with the attribute],
 "children": [array of child attribute nodes]
}
```

The location data of attributes within the dependency tree can be used to render annotations over the source code within the development environment. These annotations show us exactly where the compiler was "looking", in terms of the lexical tokens in relation to the AST nodes hosting the attribute instance, when it encountered the error (illustrated in Figure 7) - effectively a visual representation of the code in a manner similar to that seen by the compiler robot in Figure 1.

[5] https://sparkjava.com/

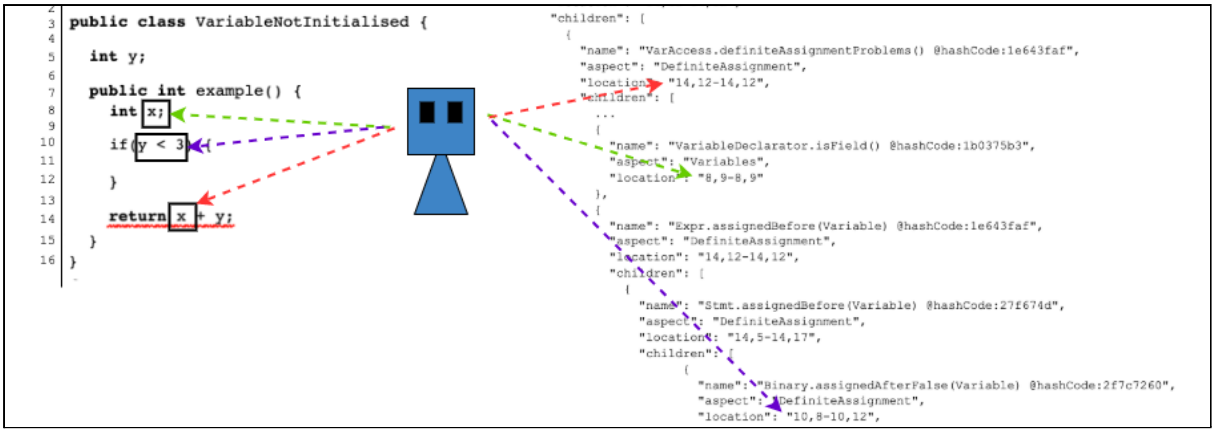

Figure 7. Example JSON excerpts of an attribute error tree. The coloured lines indicate which sections of the source code each attribute refers to.

3.3 The Progger Client

The Progger client is implemented as a web UI. Appendix A, Figure 8 shows a screenshot with the example presented in Figure 1. Again we have the error from earlier (yet with a different formulation of the error message due to a different Java compiler being used "under the hood"), but now we also have a *"Tell me more about this"* button. If this button is clicked, we can get more details about what the compiler was considering (approximately the dashed blue arrows in Figure 1) in Appendix A, Figure 9.

Here, we are seeing that the compiler considered several aspects of the code while trying to investigate whether the local variable had been assigned before its first use (`definiteAssignmentProblems`). As we hover over the "error details" to the right, the code related to the aspect being considered is highlighted in the code to the left, for instance, at one point (marked by the cursor) the compiler considered whether the local variable declaration was a field (`isField`).

The error nodes provided at the top level of the JSON output from the compiler service act as a starting point for the visualisation of a conversational interaction with the compiler. The error is rendered to the right of the code in an information box, with the line of code corresponding to the error underlined in red when the error element is moused over. At first glance, the meaning of the error may be unclear to an inexperienced programmer, and it may be useful for them to be able to ask a question of the compiler to help align their mental model - for example "What were you doing when you encountered this problem?". To facilitate this discussion, the client provides an option on the error element to *"Tell me more about this"* (Figure 8).

On click, the first layer of the attribute tree is expanded and displayed to the user (Figure 9). In keeping with the conversational tone, the aspect information that is supported in JastAdd-style RAGs are used to more clearly explain what the compiler was doing at each stage of the evaluation process. Since aspects are used to group attributes by common functionality, we have mapped each aspect name to a conversational statement giving a general overview of the purpose of that aspect. In this example, the compiler encountered a problem while checking the definite assignment of `x + y` to the return value of the method. This problem was flagged in the `DefiniteAssignment` aspect, which is mapped to a clarifying statement in the client: "I was checking if this definite assignment is valid", shown in Appendix A, Figure 10. This "error details box" can then be clicked for more details, at which time additional information from the trace is displayed (Figure 9) and can be explored by further expansion of the error details boxes.

With this presentation of the error details, the interaction continues after the error is presented. When the error does not make sense (conversation breaks down) the developer can ask for more information ("tell me more about this") which then results in a display of a list of the steps the compiler took to detect the error. The user can follow the "train of thought" of the compiler by following along the list of error details and hover over the boxes to see what parts of the code that were considered, while also considering the names of the aspect and attribute of the computational unit.

Appendix A, Figure 11 includes the end of the error details from Figure 8 along with the code highlighting connected to those error detail boxes. As the user gradually hovers over the list of error details (all concerned with "definite assignment") from top to bottom, the "return statement" is highlighted (where the error is marked), then the if statement, the assignment, and then the condition. The final box then highlights the declaration. The compiler is trying to determine whether the "x" variable has been assigned. With the highlighting we can follow along to the areas in the code (away from the error location) that it had to consider.

4. Discussion and Implications for Future Work

This paper outlines a strategy for making the interaction around errors a foreground part of the experience of a developer. In doing so it practically demonstrates the work that is needed in order to create an environment with a closer alignment between a developer and a compiler. This work proceeds on two fronts. Firstly, the internal processes and data structures of the compiler have to be exposed, and, where possible, mappings created on top of them that are likely to be closer to the model by which the developer thinks of what is happening in the compiler. Secondly, the user interface by which the developer interacts has to be brought closer to the compiler, with additional elements added to enable requests for additional information in particular cases. This engineering work to 'meet in the middle' needs to be built on top of an architecture that can support allocation of different pieces of work to the various components in the system. Doing this results in a more conversation-like interaction with the compiler shifting away from an idempotent input/output model, to a question/answer model, and in doing so highlights a number of possibilities for future work.

In order to empirically characterise the effects of this to a more conversational interaction mode it will be necessary to support a wider range of features within Progger, for example saving changes, introducing syntax highlighting and multiple files in order to create a more representative interaction context. More significantly, there are further refinements that can be made to the presentation of the errors themselves, hiding extraneous information and supporting the developer using Progger by focussing on the conversational interaction. Once these improvements have been completed we propose to study in a representative commercial context how developers go about doing their work when the focus is shifted to a conversational interaction around error messages.

The tool in its current state may also lend itself well to an educational context. While the information presented in the attribute tree view may not necessarily solve a problem by itself, it will point the user towards relevant places in the code - potentially highlighting to novice programmers the most relevant areas for review when trying to resolve an error. In this way, Progger acts not as a problem-solving tool, in that it does not actively suggest fixes, but rather as one that aids our understanding and facilitates learning.

Acknowledgements

This work is supported by the Swedish Foundation for Strategic Research under Grant No. FFL18-0231 and the Swedish Research Council under Grant No. 2019- 05658.

References

Becker, B. A., Denny, P., Pettit, R., Bouchard, D., Bouvier, D. J., Harrington, B., Kamil, A., Karkare, A., McDonald, C., Osera, P., Pearce, J. L., & Prather, J. (2019). *Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research.* In Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education, pp. 177-210, ACM.

Beneteau, E., Richards, O., K., Zhang, M., Kientz, J. A., Yip, J., & Hiniker, A. (2019). *Communication Breakdowns Between Families and Alexa.* In proceedings of the 2019 CHI Conference on Human Factors in Computing Systems, ACM.

Church, L., Söderberg, E., & McCabe, A. T. (2021). *Breaking down and making up - a lens for conversing with compilers.* In proceedings of the 32nd Annual Workshop of the Psychology of Programming Interest Group (PPIG).

Dubberly H., & Pangaro, P. (2009). *What is conversation? How can we design for effective conversation?* Interactions Magazine, XVI(4):22-28.

Ekman, T., & Hedin, E. (2007). *The JastAdd Extensible Java Compiler.* SIGPLAN Notices 42(10):1-18.

Fors, N., Söderberg, E., & Hedin, G. (2020). *Principles and Patterns of JastAdd-style Reference Attribute Grammars.* In proceedings of the 13th International Conference on Software Language Engineering, ACM.

Hedin, G. (2000). *Reference Attributed Grammars*. Informatica, 24(3):301–317.

Hedin, G., & Magnusson, E. (2003). *JastAdd - an aspect-oriented compiler construction system*. Science of Computer Programming, 47(1):37-58.

Henderson A., & Harris, J. (2011). *Conversational Alignment.* Interactions Magazine, 18(3):75-79, ACM.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingier, J., & Irwin, J. (1997). *Aspect-oriented programming.* In proceedings of the European Conference on Object-Oriented Programming (ECOOP), pp. 220-242, Springer.

Knuth, D. (1968). *Semantics of context-free languages*. Mathematical Systems Theory, 2(2):127-145.

Magnusson, E., Ekman, T., & Hedin, G. (2009). *Demand-driven evaluation of collection attributes*. Automated Software Engineering, 16(2):291-322.

Mori, M., MacDorman, K. F., & Kageki, N. (2012). *The Uncanny Valley [From the Field]*. IEEE Robotics & Automation Magazine, 19(2):98-100.

Söderberg, E., & Hedin, G. (2011). *Automated Selective Caching for Reference Attribute Grammars*. In proceedings of the International Conference on Software Language Engineering, pp 2-21, Springer Berlin Heidelberg.

Öqvist, J. (2018). *ExtendJ: Extensible Java Compiler*. In the conference companion of the 2nd International Conference on Art, Science, and Engineering of Programming, pp. 234–235, ACM.

Appendix A: Screenshots

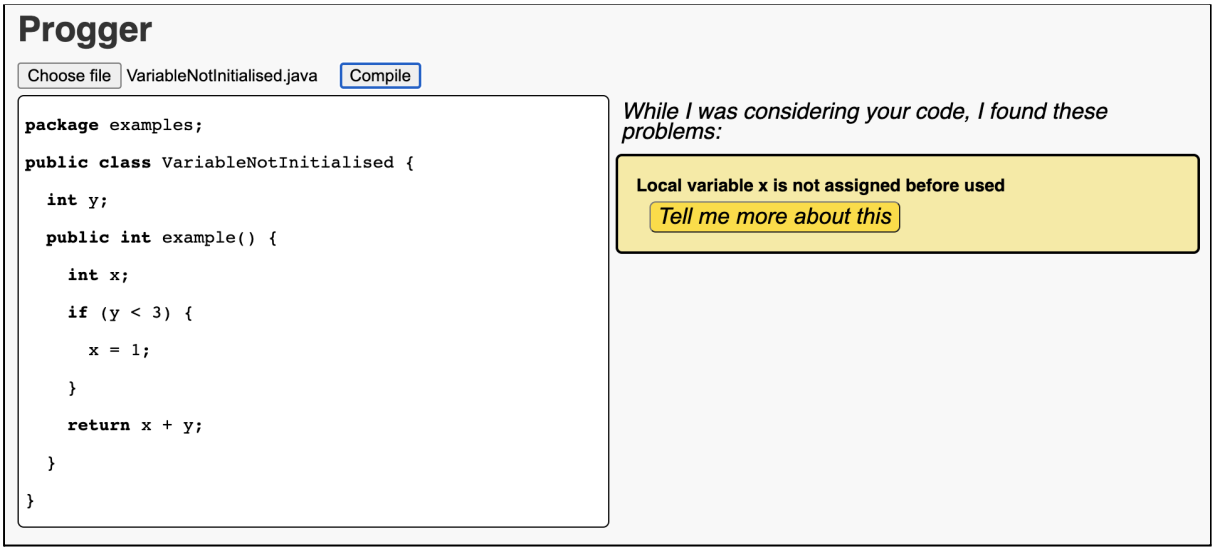

Figure 8. Screenshot of the Progger prototype with code to the left and error information to the right.

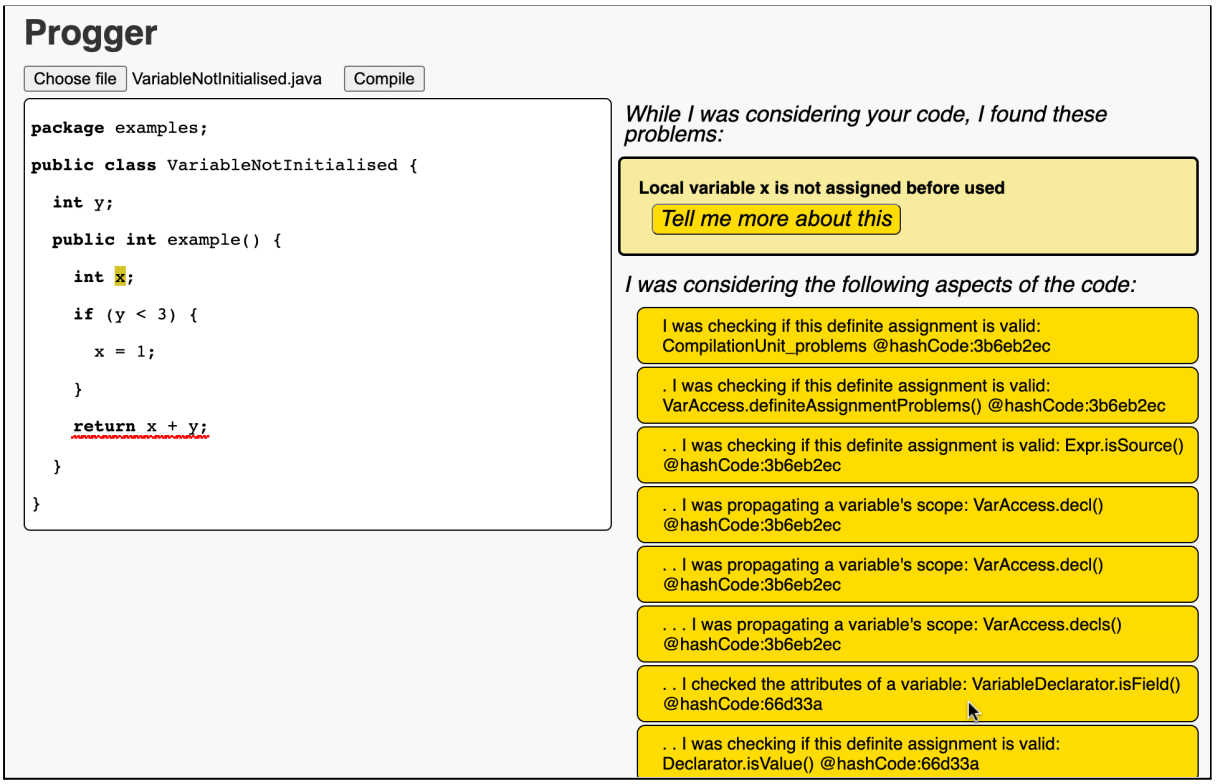

Figure 9. Screenshot of the Progger prototype with the error details from Figure 8 expanded. The variable, x, highlighted in the code pane corresponds to the location investigated while evaluating the attribute that the cursor is over.

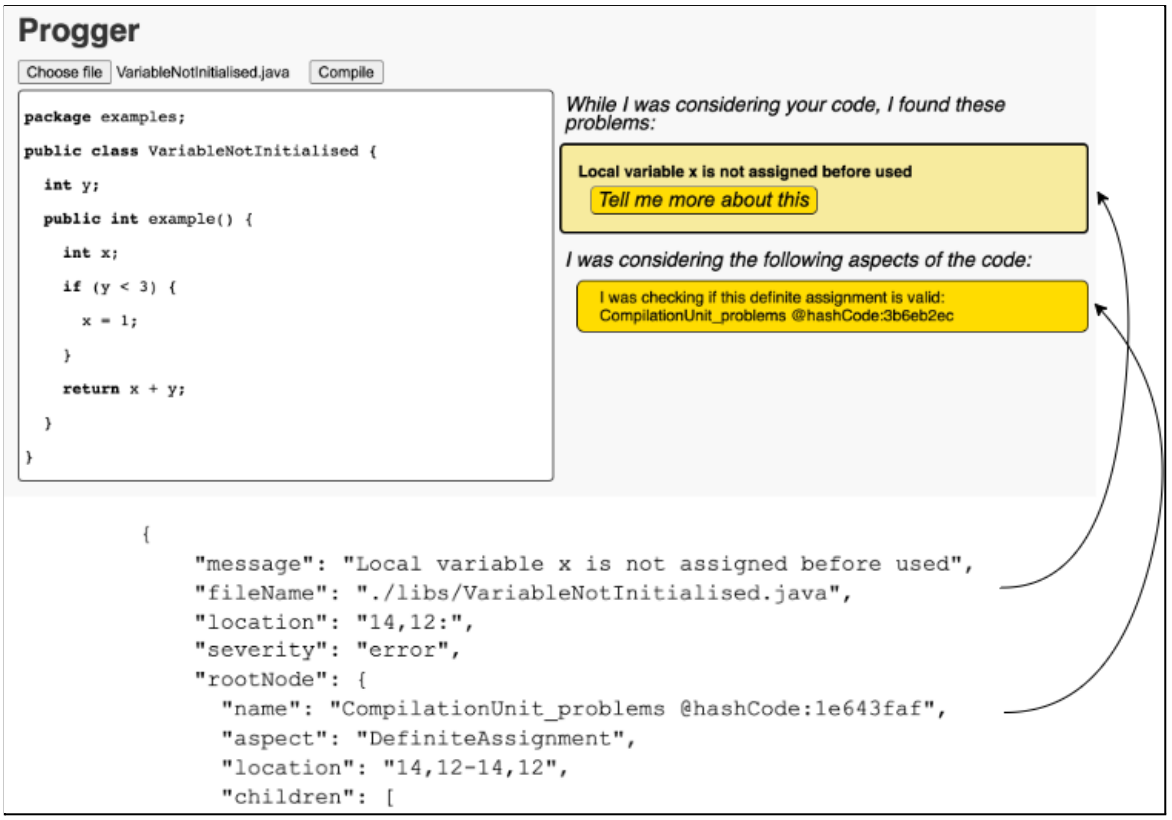

Figure 10. Mapping of errors and attribute details to conversational statements

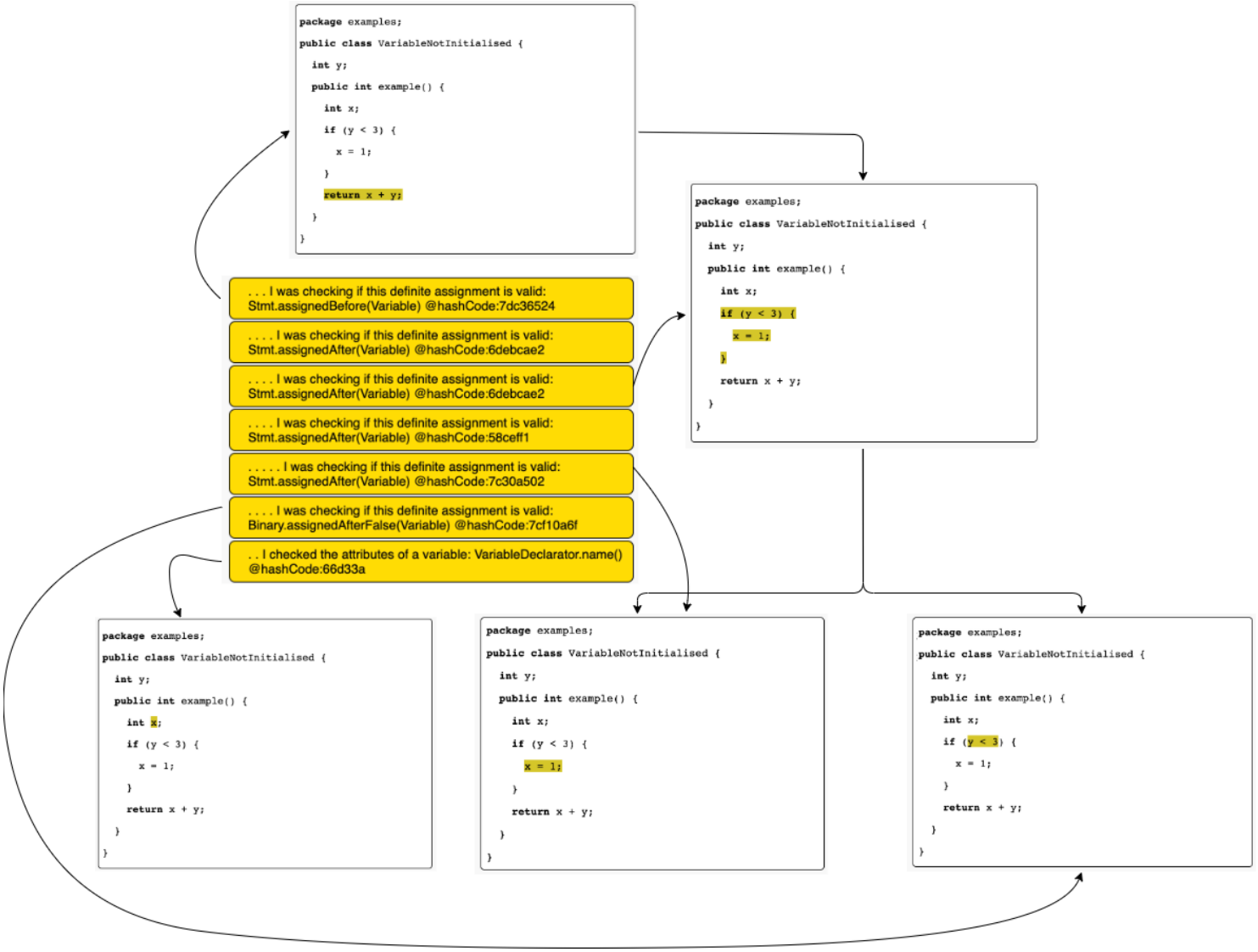

Figure 11. The error trace details shown in order of occurrence in the computation with arrows pointing to how the highlighting in the code changes on hover over error details.