Error-Tolerant Parsing and Compilation for Hylo: Enabling Interactive Development

Research Project CS3000

Viktor Seršiḱ[1]
Supervisors: Andreea Costea[1]**, Jaro Reinders**[1]
[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 22, 2025

Name of the student: Viktor Seršiḱ
Final project course: CSE3000 Research Project
Thesis committee: Andreea Costea, Jaro Reinders, Harm Griffioen

An electronic version of this thesis is available at http://repository.tudelft.nl/.

Abstract

Traditional compilers assume complete and syntactically correct input, making them ill-suited for modern interactive programming environments, where code is often incomplete or erroneous. This paper examines how error-tolerant parsing, crucial for modern Integrated Development Environment (IDE) support, can be implemented in the *Hylo* programming language, which currently lacks this capability.

We identify key techniques from state-of-the-art compilers such as Roslyn and IntelliJ, including *phrase-level recovery*, *token synchronization*, *combinator resilience*, and *Abstract Syntax Tree (AST) placeholders*. We evaluated their suitability for *Hylo*'s architecture and implemented a prototype demonstrating AST placeholder integration.

This approach moves beyond the 'halt-on-first-error' model, enabling the parser to continue processing despite errors, producing a structurally sound abstract syntax tree annotated with diagnostics.

Our work highlights the potential of error-tolerant parsing to enhance developer experience in emerging languages and lays the foundation for IDE support and interactive tooling in the *Hylo* ecosystem.

1 Introduction

In traditional compiler design, a foundational assumption is that the input source code is complete and syntactically correct [1]. This assumption holds well for batch-style compilation and production pipelines, where correctness is a prerequisite. However, modern software development workflows, particularly within Integrated Development Environments (IDEs), frequently involve incomplete, temporarily incorrect, or refactored code [10]. To support these dynamic workflows effectively, compilers must offer robustness in the face of incomplete or erroneous input. [12, 8]

Robust compiler tooling for incomplete programs enables real-time feedback, intelligent code completion, syntax-aware navigation, and early error detection, even while the developer is midway through writing code. Achieving this requires compilers to recover gracefully from parsing errors, generate partial intermediate representations like abstract syntax trees (ASTs), and continue collecting meaningful diagnostics across the file. This concept is referred to as error-tolerant or resilient compilation. [16, 5]

State-of-the-art IDEs such as *Microsoft*'s *Visual Studio* (via *Roslyn*) and *JetBrains*' *IntelliJ* platform implement sophisticated error recovery techniques that allow them to function smoothly over incomplete or invalid programs. [12, 8] These techniques have proven critical in enhancing developer productivity and usability.

For emerging languages like *Hylo*, building such robust compiler support is a timely challenge. Currently, a single syntax or type error in *Hylo* can prevent any further analysis, blocking IDE features and degrading the developer experience. [4] This research project aims to explore how *Hylo*'s compiler can be extended to handle incomplete programs, including parsing and type checking where possible. This includes investigating existing techniques from other compilers and IDEs, evaluating Hylo's architecture, and potentially designing a prototype that demonstrates partial, error-tolerant compilation. Such efforts will help enable a smoother, more supportive programming environment for *Hylo* developers.

To address this timely challenge and enable a smoother, more supportive programming environment for *Hylo* developers, this research project aims to explore how *Hylo*'s compiler can be extended to handle incomplete programs smoothly, including parsing, where possible. Specifically, this work is guided by the following central research question:

Can we make the Hylo compiler robust against incomplete programs? Can we continue parsing even if an error is found in unrelated code?

This question breaks down into several sub-questions:

- Can the parser continue processing even if errors are found in unrelated parts of the code?
- What techniques exist for error-tolerant parsing in other compilers and IDEs?
- How can we adapt or integrate these techniques into the Hylo compiler?
- Can a prototype be developed that demonstrates partial parsing and diagnostic collection for incomplete code?

2 Background

The development of error-tolerant compilers is a response to the growing demand for responsive, interactive programming environments. Traditional compilers, designed for batch-style workflows, are optimized for complete and correct programs. In contrast, modern IDEs serve developers during active code construction, where syntax errors, incomplete declarations, and unresolved references are common [7]. To maintain functionality such as real-time diagnostics, code completion, and refactoring tools, these environments require compilers that can parse, analyse, and provide feedback on incomplete or partially incorrect code.

Error-tolerant parsing serves two primary roles:

1. Maintaining structural correctness, so the rest of the toolchain can operate
2. Capturing rich diagnostics, to inform the developer without blocking the workflow

Several well-established compilers have long evolved to support such interactive needs. For example:

- The *Roslyn* compiler for *C#* is designed to work incrementally and supports speculative parsing and typed holes to maintain semantic analysis in the presence of incomplete constructs. [9]
- *JetBrains' IntelliJ* uses layered parsing and fallback strategies to recover from syntax errors without compromising AST integrity. [8]

These systems emphasize *incremental compilation*, *error recovery mechanisms*, and *intermediate representations* that tolerate missing or malformed code segments. [14]

Parsing strategies like *recursive descent*, often used in hand-written compilers for new languages, can be adapted to support resilience through techniques such as *phrase-level recovery*, *AST placeholders*, or *token synchronization* [1]. These methods allow the parser to skip over or fill in problematic areas, enabling the rest of the code to remain analysable. Moreover, partial ASTs can feed into later stages such as type checking, symbol resolution, and code transformation-further improving the development experience. [5, 2]

In the case of *Hylo*, supporting incomplete programs poses unique challenges. The existing compiler halts on minor syntax errors, limiting its integration with modern tooling. Addressing this gap involves rethinking how *Hylo* handles errors during parsing and analysis, and adopting techniques proven in mature language ecosystems to support interactive and resilient compilation.

3 Methodology

This research followed a design-oriented and iterative approach to extend the *Hylo* compiler with error-tolerant capabilities, focusing on its parsing phase as a foundational step. The methodology was divided into three key phases: Survey, Design, Implementation/Evaluation.

The first phase was the **Survey** phase. This phase aimed to analyse the current *Hylo* compiler. Furthermore, it looked into compilers for other languages that are robust against incomplete programs. Based on the findings, possible techniques were evaluated for their suitability for the *Hylo* compiler. The evaluation method involved a literature review of compiler error recovery strategies and a comparative analysis based on criteria such as complexity of implementation, compatibility with *Hylo*'s recursive descent parser, and potential for IDE integration. The results of this phase could then be used during the next phase to ideally integrate the found key techniques in the Hylo compiler.

The second phase was the **Design** phase. Based on the review of the previous phase, suitable techniques were selected for integration into *Hylo*'s compiler. These techniques were mapped directly onto *Hylo*'s Parser. This was a crucial step before implementation.

The **Implementation/Evaluation** phase developed a proof-of-concept implementation in the *Hylo* compilers *Swift* codebase. This prototype was designed to continue parsing even after encountering an error of incomplete code. This phase inherently evaluated the viability of the design of the previous phase. *AST placeholders* were chosen for the initial implementation due to their strong alignment with *Hylo*'s architecture, allowing for continuous parsing and maintaining AST integrity with relatively low implementation complexity, serving as a foundational step before integrating more complex recovery mechanisms.

4 Design and Integration of Error-Tolerant Parsing in *Hylo*

This section presents the main contribution of this research: a set of parser techniques implemented in the *Hylo* compiler to support error-tolerant parsing of incomplete or incorrect programs, along with their integration strategy for the *Hylo* compiler.

4.1 Selected Techniques

The following four techniques were chosen to be viable to be integrated into the parsing layer of the *Hylo* compiler: *Phrase-Level Recovery*, *Combinator Wrappers with Recovery Logic*, *Token Synchronization*, and *AST Placeholders.*

These techniques were selected based on their proven effectiveness in tools like *ANTLR*, *Roslyn*, or *IntelliJ*, as well as their potential compatibility with *Hylo*'s recursive descent parser. All four aim to allow continued parsing in the presence of syntactic errors, but they differ in how they balance recovery accuracy, structural fidelity, and implementation complexity.

4.1.1 Phrase-Level Recovery

Phrase-Level Recovery is one of the most basic recovery techniques used in many parsers like *ANTLR* [11], or *Roslyn* [9]. The way it works is simple, but can vary in complexity: If an er-

```
1 fun greet(name: Str): Str = "Hello, " + name
2
3 print(greet("Viktor")
4
5 val nums: List<Int> = [1, 2, 3, 4]
6
7 // ...
8 // ...
```
Listing 1: Example of code with non-closed parenthesis

```
// ...
val numbers = [1, 2, 3, 4]
val alphabet = ['a', 'b', 'c', 'd']
val numbers2 = [5 6 7 8]
// ...
```
Listing 2: An example of an array without comma separators

ror occurs, a local fix is inserted. Afterwards, the parser continues to parse. Most commonly, phrase-level recovery inserts semicolons at the ends of lines or closes open parentheses. [13]

In the code example in Listing 1, without Phrase-Level Recovery, the parser would stop at line 3 for unclosed parentheses. With Phrase-Level Recovery, however, line 3 would get changed to `print(greet("Viktor"))`, adding a parenthesis.

4.1.2 Combinator Wrappers with Recovery Logic

Combinator Wrappers with Recovery Logic are very similar in principle to Phrase-Level Recovery. This technique is an extension to parser combinators, for example, for arrays. If faulty code is encountered within a combinator wrapper, it is corrected, similarly to *Phrase-Level Recovery* from Subsubsection 4.1.1. [3, 13]

In the code example in Listing 2, we can see that `numbers2` is declared wrong, i.e., without commas between its elements. The recovery logic would add commas to `val numbers2 = [5, 6, 7, 8]`, and the parser could continue.

4.1.3 Token Synchronization

Token Synchronization is possibly one of the simplest techniques for making a parser more robust against incomplete programs, but it mostly leads to unexpected behaviour due to its nature. If an error occurs, the parser merely jumps to the next synchronization token and ignores anything in the context of where the error occurred up to the synchronization token. [11, 6]

In the code in Listing 3, for example, there is a mistake in line 3. Therefore, the parser jumps to the next synchronization token `}`, in line 5. Due to the jump, the function `example` is never declared.

```
fun example() -> Int {
    let x = 1
    let y = * 2
    return x + y
}

fun example2() -> Int {
    let x = 1
    let y = 2
    return x + y
}
```

Listing 3: Example of incomplete code

```
//...
x = 1862 + ;
//...
```

Listing 4: Example of incomplete code

4.1.4 AST Placeholders

AST Placeholders is the most effective technique when it comes to making the parser continue parsing after an error, while keeping the result predictable. This technique requires the construction of an Abstract Syntax Tree (AST) during parsing: When an error occurs during parsing, a placeholder node is inserted into the AST, and the parsing continues [5, 2]. Furthermore, the placeholder nodes may hold additional info of what type of error occurs - e.g. `MissingExpr`, `ErrorNode`, `CursorTyping`, etc. - and are used in a parser like *Tree-Sitter* [15].

For example, the code in Listing 4 creates the AST seen in Figure 1:

4.2 Integration with the *Hylo* Compiler

To enable error-tolerant parsing within the *Hylo* compiler, a layered architecture was adopted to structure the parsing process. This pipeline consists of five core components: Tokenizer, ParserState, Recursive Descent Functions, *Phrase-Level Recovery* (Including Recovery for *Combinator Wrappers*), and *AST with Placeholders*. The flow of the parsing can be seen in Figure 2. While the Tokenizer, ParserState, and the Recursive Descent Functions are pre-

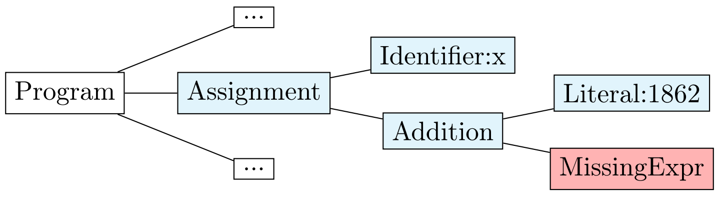

Figure 1: Example AST with Placeholder Node

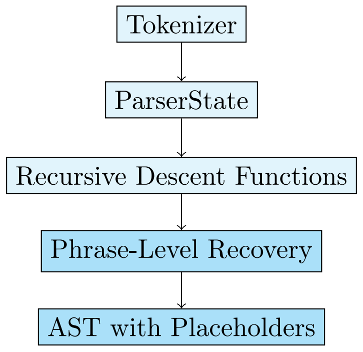

Figure 2: High-level parsing pipeline

```
fun main() {
    string1 = "Hello";
    string2 = string1 + "!";
}
```

Listing 5: Example of complete *Hylo* code

existing parsing stages, the other steps are enhancements to the current state of the parser for *Hylo*. *Token Synchronization* was omitted from the final pipeline because its behaviour of skipping to the next synchronization token would often discard valuable contextual information, leading to less precise diagnostics compared to the structural preservation offered by AST placeholders, making it less suitable for *Hylo*'s goal of enabling rich IDE features.

The Tokenizer is responsible for producing a stream of tokens, using key synchronization tokens such as `;` for the end of lines, `}` for the end of statements or some declarations, `)` for the end of functions, `,` as a separator in combinators like lists, arrays, or tuples, `]` as the end of arrays. These tokens are fundamental for identifying structural boundaries in the code. The ParserState tracks the parser's current position, maintains diagnostics, and contains the AST.

The Recursive Descent Functions implement the grammar of the *Hylo* language and can be extended to support *phrase-level recovery*, including recovery logic for *combinator wrappers*. This recovery mechanism attempts to gracefully handle missing or misplaced tokens by heuristically inserting expected closing delimiters (e.g., parentheses, brackets, commas).

Lastly, whenever an error cannot be recovered, a corresponding placeholder node is inserted into the Abstract Syntax Tree (AST) where the error occurs. These placeholder nodes are structured to represent the kind of construct that failed to parse. Refer to Section 5 for more details.

Looking at the code in Listing 5, the AST in Figure 3 is created. The AST looks about as expected from any AST. Knowing that, the integration of placeholder nodes is rather easy. If the parser stumbles upon faulty code, merely a placeholder node has to be placed in the position where the error occurs, while the rest stays the same. For example, modifying line 3 to `string2 = string1 + ;` would produce the AST shown in Figure 4

It is key for the nodes to preserve critical metadata such as the source location and

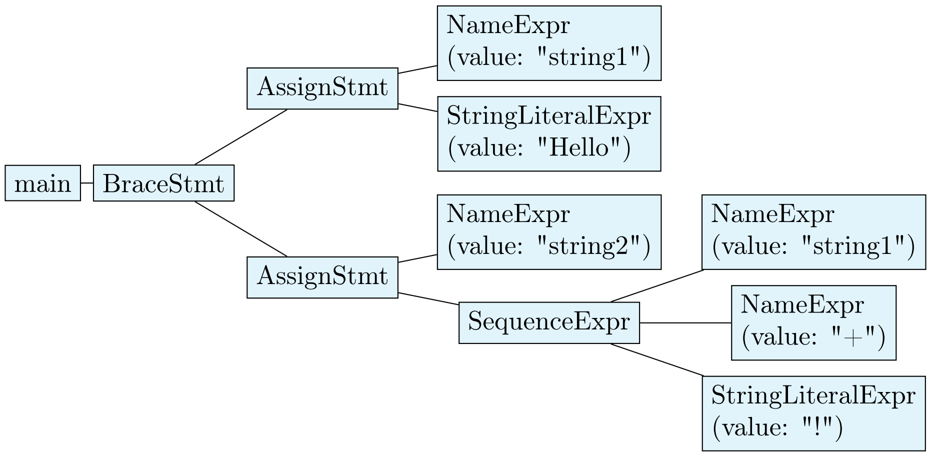
main
BraceStmt
AssignStmt
NameExpr (value: "string1")
StringLiteralExpr (value: "Hello")
AssignStmt
NameExpr (value: "string2")
SequenceExpr
NameExpr (value: "string1")
NameExpr (value: "+")
StringLiteralExpr (value: "!")

Figure 3: Regular AST

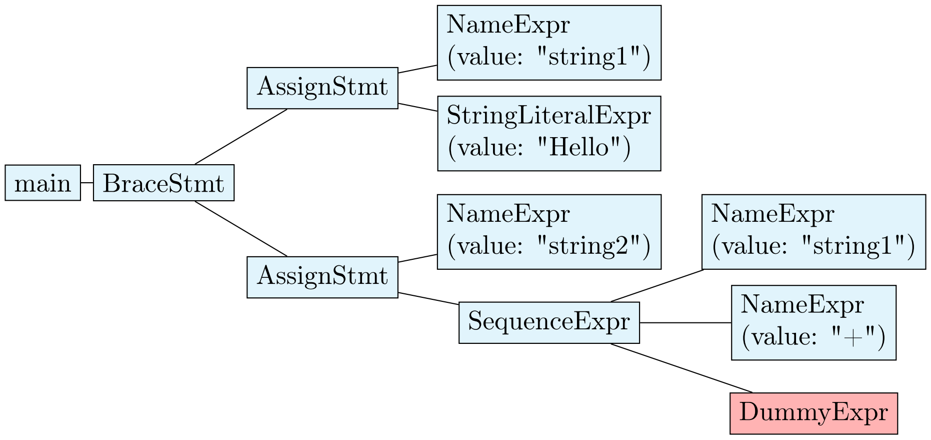
main
BraceStmt
AssignStmt
NameExpr (value: "string1")
StringLiteralExpr (value: "Hello")
AssignStmt
NameExpr (value: "string2")
SequenceExpr
NameExpr (value: "string1")
NameExpr (value: "+")
DummyExpr

Figure 4: AST with Placeholder Node

diagnostic information. This information can be used for providing rich error messages and enabling downstream compiler phases or IDE features to precisely pinpoint and highlight issues to the user.

A key design goal of this integration is to preserve the well-formedness of the AST. Even when code is incomplete or syntactically invalid, the AST produced must remain syntactically correct in structure so that it can be consumed by downstream phases such as type checking or IDE tooling. Placeholder nodes ensure that the tree remains navigable and analyzable, making the parser both robust and developer-friendly.

4.3 Summary of Contributions

This research makes the following key contributions:

- **Identification and evaluation of error-tolerant parsing techniques:** We systematically surveyed state-of-the-art compiler error recovery strategies (*phrase-level recovery*, *token synchronization*, *combinator resilience*, and *AST placeholders*) and assessed their applicability to *Hylo*'s recursive descent compiler architecture. This directly addresses the research sub-questions: *What techniques exist for error-tolerant parsing in other compilers and IDEs?* and *Can the parser continue processing even if errors are found in unrelated parts of the code?*

- **Design of a resilient parsing pipeline for *Hylo*:** We proposed a layered architecture integrating selected techniques into *Hylo*'s existing compiler structure, outlining how these components interact to achieve error tolerance. This directly addresses the research sub-question: *How can we adapt or integrate these techniques into the Hylo compiler?*

This leaves open the last sub-question: *Can a prototype be developed that demonstrates partial parsing and diagnostic collection for incomplete code?* This sub-question is answered during the Prototype Implementation and Viability Evaluation in Section 5

5 Prototype Implementation and Viability Evaluation

To evaluate the viability of error-tolerant parsing in the *Hylo* compiler, a proof-of-concept implementation was developed focusing on a single technique: *AST placeholders* as described in Subsubsection 4.1.4. This approach was chosen due to its simple implementation with *Hylo*'s existing recursive descent parser and its ability to preserve structural integrity in the presence of syntax errors.

The implementation was carried out in the existing *Swift* codebase of the *Hylo* compiler. In the prototype, modifications were made to the parser such that when a parsing error occurs - whether due to an unexpected token, missing construct, or invalid syntax - a *dummy node* is immediately inserted into the Abstract Syntax Tree (AST). These dummy nodes contain metadata, including the source location where the error occurred and a corresponding diagnostic message. This ensures that the resulting AST remains structurally valid and that information about all encountered issues is retained.

Parsing continues uninterrupted after each error, and the parser accumulates all diagnostics as the AST is being constructed. Once parsing completes, the diagnostics collected via these placeholder nodes are reported collectively, rather than halting after the first error.

```
1 fun main() {
2     let x = 1 + ;
3     let y = 5;
4     if (x > y)
5 }
```

Listing 6: Example of incomplete *Hylo* code

This design enables better IDE feedback and allows developers to address multiple issues in one pass.

To maintain the parser's structure of the AST, four distinct types of placeholders - or *dummy nodes* - were introduced:

- `DummyDecl` - a placeholder for malformed or absent declarations (i.e., var-declarations, function-declarations, etc.)
- `DummyExpr` - a placeholder for invalid, incomplete, or absent expressions (i.e., numeric-literal-expression, cast-expression, boolean-literal-expression, etc.)
- `DummyPattern` - inserted when pattern constructs fail to parse (i.e., binding-pattern, tuple-pattern, name-pattern, etc.)
- `DummyStmt` - a placeholder for malformed statements (i.e., conditional-statement, for-statement, while-statement, etc.)

For example, consider the incomplete *Hylo* code snippet in Listing 6. In this example, the parser encounters errors at line 2 and line 4. Without error tolerance, the AST would be incomplete for the stages following parsing. With the prototype implementation, a `DummyExpr` and `DummyStmt` node are inserted at the corresponding locations, resulting in an AST as seen in Figure 5

The placeholder nodes not only preserve the AST's integrity, allowing the compiler to continue to the next stage of the toolchain, but also carry a diagnostic message indicating the type of error: `"expected expression"` and `"expected proper statement"`

6 Responsible Research

In the development of error-tolerant compiler infrastructure for the *Hylo* language, careful attention has been paid to responsible research practices. This includes evaluating the broader ethical implications of the work and ensuring the reproducibility and transparency of the methods used.

6.1 Ethical Considerations

The core goal of this research is to improve the developer experience and enable more robust tooling for incomplete or evolving code. As such, the project does not involve human subjects, sensitive data, or applications with direct ethical risks (e.g., medical, surveillance, or AI-based decision-making systems). Nonetheless, it is important to recognize that compiler infrastructure plays a critical role in software correctness and developer trust.

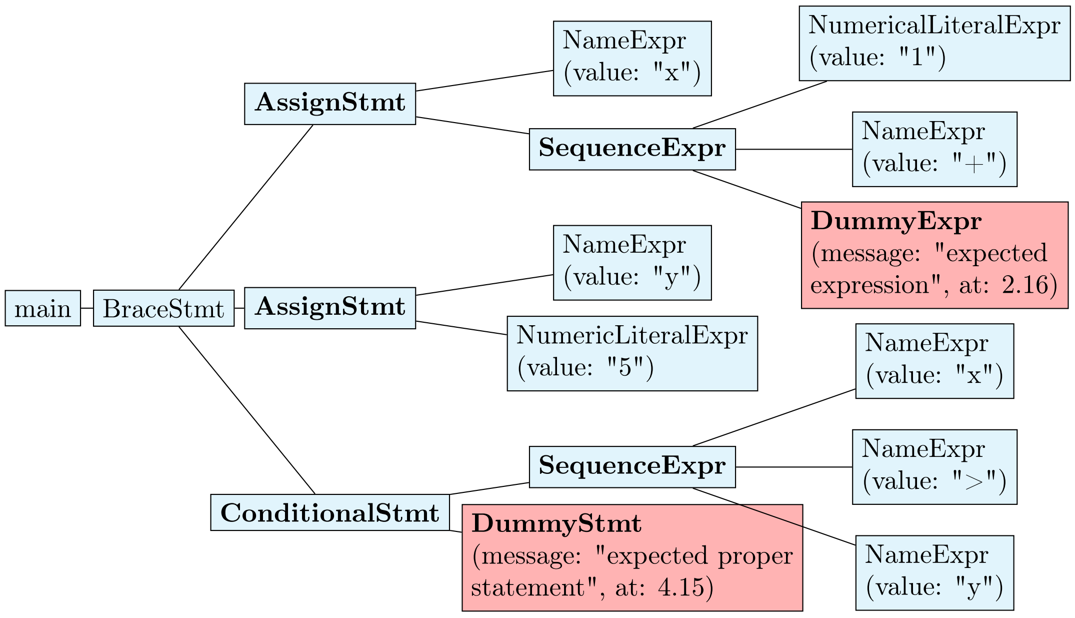

Figure 5: Resulting AST from Listing 6

By designing a compiler that accepts and tolerates incomplete code, there is a risk that developers might overlook certain errors during early development stages. To mitigate this, the techniques implemented (such as *AST placeholders* and *diagnostics*) are explicitly designed to preserve and report all encountered issues rather than silently masking them. The goal is not to "ignore" errors but to postpone fatal failure and enable progressive correction with full transparency. All placeholder nodes are annotated in the AST and linked to clear diagnostics, ensuring that developers are always aware of potential issues.

6.2 Reproducibility

This project emphasizes transparency and reproducibility. All prototype implementations are built directly into the publicly accessible *Hylo* compiler codebase. The changes made are uploaded to a fork[1] of the official GitHub repository[2] for *Hylo* to allow future contributors to reproduce and extend the results.

7 Discussion

The implementation of *AST Placeholders* in the *Hylo* compiler represents a significant step towards achieving error tolerance. This approach directly addresses the limitations of traditional compilers in interactive development environments, where immediate feedback and continuous analysis are paramount. By inserting dummy nodes into the AST, the parser can effectively "skip over" syntax errors without halting, thereby maintaining the structural integrity of the abstract syntax tree. This is crucial because a well-formed (even if partially

[1]https://github.com/viktorSrk/hylo-ast-placeholders
[2]https://github.com/hylo-lang/hylo

incorrect) AST is a prerequisite for subsequent compiler phases, such as type checking, and for various IDE features like syntax highlighting, code completion, and refactoring.

The choice of *AST Placeholders* as the primary technique for the proof-of-concept was strategic. Its relative simplicity of integration into *Hylo*'s recursive descent parser, combined with its impact on the compiler's resilience, made it an ideal starting point. Unlike simpler *Token Synchronization*, which might discard valuable contextual information, *AST Placeholders* explicitly mark the location and type of error, preserving a more complete semantic understanding of the codebase. This allows for more precise diagnostics and enables the compiler to report multiple errors in a single pass, enhancing developer efficiency.

However, the current implementation, while effective for demonstrating the core concept, has limitations. The diagnostics currently collected through placeholder nodes are basic, indicating only the presence and general type of error. To truly rival the developer experience offered by mature compilers like Roslyn, these diagnostics need to be significantly enriched. This includes providing more context-aware error messages, suggesting potential fixes, and categorizing errors based on severity.

The decision to focus solely on *AST Placeholders* in the prototype was driven by time constraints, but it underscores the need for future work to integrate other identified techniques. *Phrase-level Recovery*, for example, could proactively "fix" minor errors, reducing the number of placeholder nodes generated and resulting in a cleaner AST for further analysis. This would be particularly beneficial for common typos or missing delimiters, allowing the parser to produce an even more accurate representation of the developer's intent.

Overall, this research confirms that it is indeed possible to make the *Hylo* compiler robust against incomplete programs by continuing parsing even when errors are found. The prototype successfully demonstrates the feasibility of this approach and highlights *AST Placeholders* as a foundational technique. The insights gained lay a solid groundwork for future enhancements (see Section 9), pushing *Hylo* closer to providing the sophisticated interactive development experience expected of modern programming languages.

8 Conclusions

This research addressed the critical challenge of making the *Hylo* compiler robust against incomplete and erroneous programs, a necessity for modern interactive development environments. Our central research question was: *Can we make the Hylo compiler robust against incomplete programs? Can we continue parsing even if an error is found in unrelated code?*

We conclusively demonstrate that it is indeed possible. Through a systematic survey of state-of-the-art error-tolerant compilation techniques, including *Phrase-level Recovery*, *Token Synchronization*, and *AST Placeholders*, we identified the most suitable candidates for *Hylo*'s recursive descent architecture. Our prototype implementation, focusing on *AST Placeholders*, proves that the parser can continue processing code and build a structurally sound Abstract Syntax Tree (AST) even in the presence of syntax errors. This is achieved by inserting specialized dummy nodes (`DummyDecl`, `DummyExpr`, `DummyPattern`, `DummyStmt`) at error locations, which also carry diagnostic information.

This approach offers several key contributions. Firstly, it moves *Hylo* beyond the traditional "halt-on-first-error" compilation model, enabling a more fluid and responsive developer experience. Secondly, by maintaining a complete, although error-annotated, AST, it lays the essential foundation for rich IDE features such as real-time diagnostics, intelligent code completion, and syntax-aware navigation, even when code is in an incomplete state. This capability is paramount for emerging languages like *Hylo* seeking broader adoption and a supportive tooling ecosystem.

In conclusion, this research provides a vital blueprint for building resilient compiler infrastructure for *Hylo*. By embracing error tolerance from the parsing stage, we significantly enhance the developer experience and pave the way for a vibrant and interactive *Hylo* ecosystem.

9 Future Work

The current implementation as part of the evaluation is merely a proof-of-concept implementation. This means that it opens several avenues for future research and development:

- **Integration of Additional Recovery Techniques:** While *AST Placeholders* ensure structural integrity, integrating phrase-level recovery and combinator wrappers could proactively correct minor errors, leading to fewer placeholder nodes and a more "correct" partial AST. This would further reduce noise in diagnostics and improve the accuracy of subsequent analyses.
- **Enriching Diagnostics:** The current dummy nodes collect basic error information. Future work should focus on enhancing these diagnostics to provide more context-rich, user-friendly error messages, potentially suggesting common fixes or pointing to relevant documentation
- **Partial Semantic Analysis:** A significant next step is to extend error tolerance beyond parsing into semantic analysis (e.g., type checking, symbol resolution). This would involve designing mechanisms to perform speculative or partial semantic analysis on ASTs containing placeholder nodes, allowing the compiler to provide feedback on semantic correctness even in incomplete programs.
- **Incremental Compilation:** For truly responsive IDEs, the compiler needs to process small, incremental changes efficiently. Investigating how to combine error-tolerant parsing with incremental compilation techniques (e.g., re-parsing only changed regions) would be crucial for large *Hylo* projects.
- **Tooling and IDE Integration:** The ultimate goal of this work is to enable a robust IDE experience. Future efforts should involve integrating the error-tolerant compiler into a Language Server Protocol (LSP) implementation for *Hylo*, allowing it to seamlessly power features in various IDEs.

References

[1] V Aho Alfred, S Lam Monica, and D Ullman Jeffrey. *Compilers principles, techniques & tools.* pearson Education, 2007.

[2] Luís Eduardo de Souza Amorim, Sebastian Erdweg, Guido Wachsmuth, and Eelco Visser. Principled syntactic code completion using placeholders. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, pages 163–175, 2016.

[3] Martin Bravenboer and Eelco Visser. Parse table composition. separate compilation and binary extensibility of grammars. In D. Gasevic and E. van Wyk, editors, *Proceedings of the First International Conference on Software Language Engineering (SLE 2008)*, Lecture Notes in Computer Science, Heidelberg, October 2008. Springer.

[4] Crystal Community Forum. Why isn't there an lsp for crystal? Crystal Lang Forum, 2025.

[5] Maartje de Jonge, Lennart CL Kats, Eelco Visser, and Emma Söderberg. Natural and flexible error recovery for generated modular language environments. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 34(4):1–50, 2012.

[6] Lukas Diekmann and Laurence Tratt. Don't panic! better, fewer, syntax errors for lr parsers. *arXiv preprint arXiv:1804.07133*, 2018.

[7] Sebastian Erdweg, Lennart CL Kats, Tillmann Rendel, Christian Kästner, Klaus Ostermann, and Eelco Visser. Growing a language environment with editor libraries. In *Proceedings of the 10th ACM international conference on Generative programming and component engineering*, pages 167–176, 2011.

[8] JetBrains IntelliJ Platform Team. *Grammar and Parser (IntelliJ Platform Plugin SDK)*, 2025.

[9] Microsoft. *Microsoft.CodeAnalysis.CSharp.SyntaxFactory.MissingToken Method*, 2025. Microsoft .NET Documentation (syntax recovery), accessed 2025.

[10] Microsoft Visual Studio Code Documentation. *Language Server Extension Guide*, 2025.

[11] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007.

[12] Roslyn Team. Handling program that is not syntactically correct is confusing (github issue #22755). GitHub, dotnet/roslyn repository, 2017. Comment by Cyrus on Roslyn's error-tolerant design.

[13] S Doaitse Swierstra and Luc Duponcheel. Deterministic, error-correcting combinator parsers. In *International School on Advanced Functional Programming*, pages 184–207. Springer, 1996.

[14] Microsoft Team. Tolerant php parser, 2016.

[15] Tree-sitter Developers. *Tree-sitter: Basic Syntax (ERROR and MISSING nodes)*, 2025. Tree-sitter documentation, accessed 2025.

[16] Tim A Wagner and Susan L Graham. Efficient and flexible incremental parsing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(5):980–1013, 1998.