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

Viktor Sersik [1]

Supervisors: Andreea Costea[1], Jaro Reinders[1]

[1]EEMCS, Delft University of Technology, The Netherlands

Goal of This Research

The goal of this research is to answer the following question: *Can we make the Hylo compiler robust against incomplete programs? Can we continue parsing even if an error is found in unrelated code?*

By answering the question, we hope to ...

- ... enable a **smoother, more supportive programming environment** for *Hylo* developer
- ... make the *Hylo* compiler **robust against incomplete programs**
- ... allow **parsing to continue** even if an error is found

Research Method

Our research followed a design-oriented and iterative approach, structured into **three key phases**:

1. **Survey**: Analysed existing *Hylo* compiler architecture and surveyed state-of-the-art error-tolerant techniques from compilers like *Roslyn* and *IntelliJ*
2. **Design**: Selected and adapted suitable techniques for *Hylo*'s parser, prioritising AST placeholders for initial implementation due to architectural alignment and lower complexity.
3. **Implementation Evaluation**: Developed a proof-of-concept prototype in *Hylo*'s *Swift* codebase to demonstrate the viability of the Design

Why Error-Tolerant Parsing?

The implementation of error-tolerant parsing into the *Hylo* compiler is crucial for modern IDEs because it ...

- ... enable **real-time feedback** for developers.
- ... enables support for **intelligent code completion**.
- ... maintains **structural correctness** of the Abstract Syntax Tree (AST), essential for subsequent compiler phases.
- ... captures **comprehensive diagnostics** even from incomplete code.

Key Techniques Explored

Four prominent techniques to make parsers - and therefore compilers - more error-tolerant were explored. All of them seem to be viable with the parser's architecture for *Hylo*.

Phrase-Level Recovery.

If an error occurs, a **local fix** is inserted. Afterwards, the parser continues to parse. Mostly, the fixes involve adding symbols at the end of lines, such as missing semicolons or closing parentheses.

```
1 fun greet(name: Str): Str = "Hello, " + name
2 print(greet("Viktor")
3 val nums: List<Int> = [1, 2, 3, 4]
4 // ...
```

Listing 1. Example of code with non-closed parenthesis

In this code example to the left, there is an unclosed parenthesis in line 2. With *Phrase-Level Recovery*, a parenthesis would be added at the end of the leading, correcting it to `print(greet("Viktor"))`

Combinator Wrappers with Recovery Logic.

When encountering faulty code within a **combinator wrapper**, correcting code is inserted. This technique is very similar to *Phrase-Level Recovery*, but is an extension to parser combinators.

```
1 // ...
2 val numbers = [1, 2, 3, 4]
3 val alphabet = ['a', 'b', 'c', 'd']
4 val numbers2 = [5 6 7 8]
5 // ...
```

Listing 2. Example of an array without comma separators

In line 4, we can see that there are no comma separators in the declaration of `numbers2` The *recovery logic* would add commas to make the line `val numbers2 = [5, 6, 7, 8]`. Afterwards, the parser can continue with parsing.

Token Synchronisation.

If an error is encountered, the parser **skips** to the next "synchronising token" and continues parsing there.

```
1 fun example() -> Int {
2     let x = 1
3     let y = * 2
4     return x + y
5 }
6
7 fun example2() -> Int {
8     let x = 1
9     let y = 2
10    return x + y
11 }
```

Listing 3. Example of incomplete code

In the case of a non-error-tolerant parser, the example on the left would halt when encountering the error in line 3. With *token synchronisation*, the parser would skip to the next synchronising token, in this example `}` in line 5, and continue parsing below.
The main downside here is that `example()` will now not be part of the scope for future compiling stages.

AST Placeholders.

Add special **placeholder nodes** into the Abstract Syntax Tree (AST) when an error is encountered.

```
1 // ...
2 let x = 1862 + ;
3 // ...
```

For the example above, the AST that is created is shown in Figure 1 to the right.

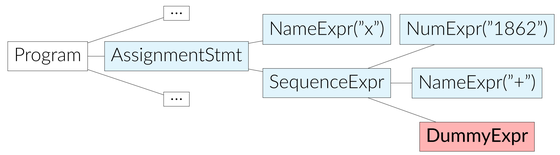

Figure 1. Example AST with Placeholder Node

Integration with *Hylo* Compiler

To enable error-tolerant parsing within the *Hylo* compiler, **two steps were added** to the structure of the already existing **parsing process**. The two steps marked in dark - *Phrase-Level Recovery* and *AST with Placeholders* - are the two added steps.

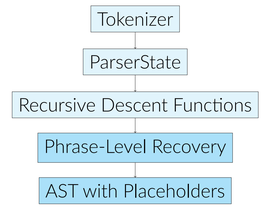

Figure 2. High-level parsing process

- The **Tokeniser** is responsible for producing a stream of tokens. 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.
- The **Phrase-Level Recovery** here also includes recovery logic for parser combinators. This step aims to recover errors that can be easily fixed.
- When an error cannot be recovered, a corresponding **placeholder** node is inserted into the **AST** where the error occurs.

Only the *AST Placeholders* have actually been implemented into a proof-of-concept prototype[a]

Prototype Implementation

To evaluate error-tolerant parsing in *Hylo*, a **proof-of-concept prototype** was developed, focusing on *AST placeholders*.

- When a parsing error occurs, a **dummy node is inserted** into the Abstract Syntax Tree (AST), containing metadata like source location and diagnostic message.
- **Parsing continues** uninterrupted after each error, accumulating all diagnostics for collective reporting.
- Four types of dummy nodes were introduced: `DummyDecl`, `DummyExpr`, `DummyPattern`, `DummyStmt`
- These placeholder nodes preserve **AST integrity** and provide specific diagnostic messages.

Conclusions

The integration of AST placeholders marks a significant **advancement towards error tolerance** in the *Hylo* compiler. This approach directly addresses limitations of traditional compilers in interactive development by:

- Enabling **continuous parsing and analysis** despite syntax errors.
- Maintaining **AST integrity**, which is crucial for subsequent compiler phases and IDE features.
- Significantly improving the **developer experience** by providing immediate and comprehensive feedback.

Future Work

Building upon the **foundational work** with AST placeholders, future efforts include:

- Integrating more complex **recovery mechanisms** (e.g., phrase-level recovery).
- Further refining **diagnostic reporting** to minimize the risk of developers not understanding error messages.
- Extend error tolerance beyond parsing into **semantic analysis** (e.g., type checking)

[a]https://github.com/viktorSrk/hylo-ast-placeholders