Wye: A Compiled,, Functional, Object-Oriented Language

Version 2.1.0

Aditya Gomatam

December 25, 2024

Abstract

One day, I thought to myself, “I’m going to write a compiler." But, for what? This document answers that question.

Wye is a statically-, strongly-, and structurally-typed, functional, object-oriented language, heavily inspired by Haskell, Rust, and OCaml. Wye tries to prevent some of the nuisances that arise in modern industry-grade languages. Nevertheless, it retains a firm basis in functional programming, indeed alluding in its name to the upside-down `y` of the lambda calculus.

This document compiles the full specification of the Wye language: its syntax, semantics, type system, and intermediate representation.

1 Introduction

A programming language should enable the programmer, not hinder them. Overly permissive language constructs such as inheritance and duck typing can make programs behave in unexpected ways; clunky syntax, such as fully-qualified names, simply distract from and obstruct the design process, preventing programmers from focusing on their work. Wye strives to avoid these encumbrances, to be simple yet still useful.

To this end, Wye features straightforward syntax, emphasizing functional over object-oriented programming. Wye invests heavily in its type system, as powerful and expressive types not only accelerate the development of correct programs, but also showcase the true beauty of type theory. Wye extends the Hindley-Milner type system (which underlies languages like OCaml and Haskell), with abstract data types, nominal and structural records, and bounded interfaces, all while retaining perfect type inference.

Wye does not innovate anything. It is not intended to push the envelope or to be at the bleeding edge of programming language research. Even though it originates from

deep study of programming language and type theory, Wye emphasizes usability over theoretical significance. It does not include esoteric types, nor language constructs that are sparingly useful, and is kept in check by real-world programming use-cases.

Most importantly, Wye was designed to be compiled. Not to be compiled *efficiently* – just to be compiled. This project is considered incomplete, or proceeding in the wrong direction, as long as Wye has no working compiler.

2 Syntax

A Wye program is stored in a `.wye` file and is, in its most abstract view, simply a sequence of Wye statements.

Written below in Backus-Naur form is roughly the Wye grammar. The start symbol is *Program*. Nonterminals start with a capital letter. Terminals are written in `this font`. $pat*$ denotes the Kleene star: zero or more repetitions of the pattern pat. $pat+$ is a shorthand for $pat\ pat*$. $pat?$ denotes that pat may or may not occur. $(pat_1 \ldots pat_n)$ groups $pat_1, \ldots, pat_n$ into a new pattern in which each pat_i is to occur in the specified sequence. $pat1|pat2$ means that exactly one of $pat1$ or $pat2$ may occur. Beware that `|` and | are different – the first is a Wye token. ⟨These brackets⟩ are used to annotate certain grammar rules.

2.1 Grammar

2.1.1 Top-level non-terminals

$$
\begin{aligned}
\mathit{Program} &\rightarrow \mathit{Statement}+ \\
\mathit{Statement} &\rightarrow \mathit{Expr} \mid \mathit{EnumDecl} \mid \mathit{StructDecl} \\
&\quad \mid \mathit{SigDecl} \mid \mathit{ImplBlock}
\end{aligned}
$$

2.1.2 Expressions

$$
\begin{aligned}
\mathit{Expr} \rightarrow\ & \mathit{IntLiteral} \mid \mathit{FloatLiteral} \mid \mathit{StringLiteral} \\
& \mid \mathit{List} \mid \mathit{Tuple} \mid \mathit{AnonRecord} \\
& \mid \mathit{Id} \mid \mathit{BuiltinOp} \\
& \mid \texttt{print} \mid \texttt{error} \\
& \mid \mathit{TypeId}.\mathit{TypeId}\ (\texttt{with}\ \mathit{Expr})\ \langle\text{enum variant}\rangle \\
& \mid \mathit{Expr}\ \mathit{Expr}\ \langle\text{function application}\rangle \\
& \mid \mathit{Expr}\ \mathit{BuiltinOp}\ \mathit{Expr}\ \langle\text{reserved binary op}\rangle \\
& \mid \texttt{match}\ \mathit{Expr}\ \texttt{\textbackslash n}\ (\mathit{Pat}\ \texttt{=>}\ \mathit{Expr}\ \texttt{\textbackslash n}) * \mathit{Pat}\ \texttt{=>}\ \mathit{Expr}\ \texttt{end}
\end{aligned}
$$

$$
\begin{array}{rcl}
 & & \mid \texttt{\textbackslash}\ Id\ \texttt{->}\ Expr\ \langle\text{lambda expression}\rangle \\
 & & \mid Id\texttt{.}Id\ \langle\text{member access}\rangle \\
 & & \mid Id\texttt{\#}Id\ \langle\text{method access}\rangle \\
 & & \mid \texttt{(}\ Expr\ \texttt{)} \\
 & & \mid LetExpr\ (\texttt{in}\ Expr)? \\
 & & \mid AttrSet \\
List & \rightarrow & \texttt{[}\ (Expr\ \texttt{,})*\ Expr\ \texttt{]} \mid \texttt{[]} \\
Tuple & \rightarrow & \texttt{(}\ (Expr\ \texttt{,})+\ Expr\ \texttt{)} \\
AnonRecord & \rightarrow & \texttt{\{}(Id\texttt{:}Expr\texttt{,})+(Id\texttt{:}Expr\ \texttt{,}?)\texttt{\}} \\
BuiltinOp & \rightarrow & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{//} \mid \texttt{::} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{==} \mid \texttt{!=} \\
Pat & \rightarrow & \texttt{\_} \mid IntLiteral \mid FloatLiteral \mid StringLiteral \mid Id \\
 & & \mid Id\ \texttt{::}\ Id\ \langle\text{head::tail list destructuring}\rangle \\
 & & \mid TypeId\ (\texttt{with}\ Pat)? \\
 & & \mid \texttt{[}\ (Pat\ \texttt{,})*\ Pat\ \texttt{]} \mid \texttt{[]} \\
 & & \mid \texttt{(}\ (Pat\ \texttt{,})+\ Pat\ \texttt{)} \\
 & & \mid \texttt{\{}\ (Id\texttt{:}Pat\texttt{,})*\ (Id\texttt{:}Pat\ \texttt{,}?)(\texttt{,}\ \texttt{\_})?\ \texttt{\}} \\
 & & \mid\ \sim\ Pat\ \langle\text{pattern complement}\rangle \\
 & & \mid Pat\ (\texttt{|}\ Pat)+\ \langle\text{pattern union}\rangle \\
 & & \mid Pat\ \texttt{if}\ Expr\ \langle\text{guarded pattern}\rangle \\
 & & \mid \texttt{case}\ Expr\ \langle\text{check boolean } Expr\rangle \\
LetExpr & \rightarrow & \texttt{let}\ Id\ (\texttt{:}Type)?\ \texttt{=}\ Expr \\
 & & \mid \texttt{let}\ Id\ Id+\ \texttt{=}\ Expr \\
 & & \mid \texttt{let}\ Id\ (\ (Id\texttt{:}Type)\ \texttt{->}\ )+Type\ \texttt{=}\ Expr \\
AttrSet & \rightarrow & \texttt{set}\ Id\texttt{.}Id\ \texttt{=}\ Expr
\end{array}
$$

2.1.3 Records and Signatures

$$
\begin{array}{rcl}
EnumDecl & \rightarrow & \texttt{enum}\ TypeId\ (\texttt{'}\ TypeId?\ Id)*\ \texttt{=}\ (TypeId\ (\texttt{with}\ Type)?) \\
 & & (\texttt{|}\ TypeId\ (\texttt{with}\ Type)?)* \\
StructDecl & \rightarrow & \texttt{struct}\ Id\ (\texttt{'}\ TypeId?\ Id)*\ \texttt{\textbackslash n}\ (Id\texttt{:}Type\ \texttt{\textbackslash n})+\ \texttt{end} \\
SigDecl & \rightarrow & \texttt{sig}\ Id\ (\texttt{implies}\ (Id\ \texttt{+})*\ Id)?\ \texttt{\textbackslash n} \\
 & & (ValDecl \mid MethodImpl \mid MethodDecl)+\ \texttt{end} \\
ImplBlock & \rightarrow & \texttt{impl}\ Id(\texttt{'}\ TypeId?\ Id)*\ \texttt{:}Id(\texttt{'}\ TypeId?\ Id)*\ \texttt{\textbackslash n} \\
 & & (AttrSet \mid MethodImpl)+\ \texttt{end}
\end{array}
$$

$$
\begin{aligned}
ValDecl &\rightarrow \texttt{val}\ Id\ \texttt{:}\ Type \\
MethodDecl &\rightarrow \texttt{method}\ Id\ \texttt{:}\ Type \\
MethodImpl &\rightarrow \texttt{method}\ Id\ Id*\ \texttt{=}\ Expr \\
&\quad \mid \texttt{method}\ Id\ \texttt{(}\ \texttt{(}Id\texttt{:}Type\texttt{)}\ \texttt{->}\ \texttt{)} + Type\ \texttt{=}\ Expr
\end{aligned}
$$

2.1.4 Types and type expressions

$$
\begin{aligned}
Type \rightarrow\ & \texttt{int} \mid \texttt{float} \mid \texttt{string} \\
& \mid TypeId\ (Type)* \\
& \mid \texttt{[}\ Type\ \texttt{]} \\
& \mid \texttt{(}\ (Type\ \texttt{,})+\ Type\ \texttt{)} \\
& \mid TypeId?\ \texttt{\{}\ \texttt{method}?\ Id\ \texttt{:}\ Type\ \texttt{\}} \\
& \mid \texttt{'}\ TypeId?\ Id\ \langle\text{(bounded) type variable}\rangle \\
& \mid Type\ \texttt{->}\ Type\ \langle\text{function type}\rangle
\end{aligned}
$$

2.1.5 Notes on the grammar

The exact substitution rules for the *Literal* patterns are omitted to avoid boring the reader. If you are reading this, you know what a float is. Similarly, the rules for *Id* and *TypeId* are omitted, though they are the same: a string of digit, lowercase, underscore, or uppercase ASCII characters, not starting with a digit. Note that while Wye uses ASCII as its character set for identifiers and type names, strings may contain Unicode characters.

*Id*s and *TypeId*s should not be any of the builtin keywords such as `int`, `float`, `string`, `print`, `match`, `with`, `requires`, and so on. *Id*s and *TypeId*s must not conflict with each other within their scope.

Notice that `bool` is not a builtin type. This is because the *TypeDeclaration* system of Wye is used to define it in the Wye prelude.

One-line Wye comments begin with `;` and mark all following text until the next carriage return or newline as whitespace. Multiline Wye comments begin with `(*` and end with `*)`.

In Wye, the application of functions is always written in postfix notation, except for certain reserved binary operators (such as `+` and `::`) that may be written in infix notation. These binary operations are, under the hood, translated into postfix notation.

3 Semantics

A Wye program is parsed, as noted earlier, as a sequence of Wye statements. A Wye program is then executed top-to-bottom. Every statement has the following result:

3.1 Expressions

3.1.1 Patterns

3.2 Enums

3.3 Structures, interfaces and implementations

Methods that define shared code cannot be overridden. If you want to override them, that indicates you actually want to implement a different interface than what is being shared. That is, your code either should not be shared, or the implementation you want to provide in the override should have a different name.

4 Future plans

4.1 List Comprehension

List comprehension should not be too hard to implement compared to the rest of this project, but it will require some additions to the grammar, and is not a strong focus of the initial versions of Wye.

4.2 User Input

In Wye version 2.1.0, users may only output to the screen. Taking user input is slightly more complicated and is thus deferred to a future version of Wye.

4.3 Type Aliases

Similar to TypeScript, it would be nice if the user could do something like the following to create an abbreviation for a type:

```
; just a tuple-type of length 2
type Pair 'a 'b = ('a, 'b);
```

4.4 Modules

4.5 Intersection and Union types

5 Acknowledgements

The syntax and semantics of Wye are based on Rust, Haskell, and OCaml. I thank the developers of these languages for making it much easier for me to learn about and write a compiler. I thank Jens Palsberg and Carey Nachenberg of UCLA for teaching me the fundamentals of compiler implementation and programming languages.