Wye: A Compiled, Functional, Object-Oriented Language

Version 2.1.0

Aditya Gomatam

January 19, 2025

Abstract

Wye is a statically-, strongly-, and structurally-typed, functional, object-oriented language, heavily inspired by Haskell, Rust, and OCaml. Wye recognizes there is a conceptual burden imposed the type system in many languages that can inhibit the development of programs. In response to these issues, as well to explore the beauty of type theory in both its mathematical and practical incarnations, Wye centers around structural object typing and type inference. Nevertheless, Wye 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

Wye is a statically-, strongly-, and structurally-typed functional, object-oriented language, heavily inspired by Haskell, Rust, and OCaml. In recognition that the type system can have a drastic impact on the tendency of programmers to write maintainable or unintelligible programs, Wye invests a great deal in building a safe, expressive, yet straightforward type system.

A specific point of interest is in facilitating design work through the language. Design work is inherently unaware of its final form; however, explicitly-typed languages like C++ require that you know the types of your entities before you've completely fleshed out your thoughts. On the other hand, dynamically-typed languages like JavaScript and Python can be especially convenient during the design phase, affording an open space to play; but, they readily become unintelligible, unmaintainable, messy, and sorts of other undesirable adjectives.

Wye seeks to strike a balance between these two extremes. How can a language be designed so that it both supports design work, while also encouraging maintainability and intelligibility? I currently do not know the answer. But a hypothesis is examined in this work – extend basic functional programming (which is generally very academic, but which has perfect type inference) with structural object types and interface programming with bounds (features present in most industry-grade object-oriented languages).

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

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

$$
\begin{array}{rcl}
 & & \mid \texttt{match}\ Expr\ \texttt{\textbackslash n}\ (Pat\ \texttt{=>}\ Expr\ \texttt{\textbackslash n}) * Pat\ \texttt{=>}\ Expr\ \texttt{end} \\
 & & \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

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

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

2.1.4 Types and type expressions

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

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. The semantics of each line of the grammar are described in the subsections of this section.

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 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);
```

4.2 Dependent types

Introduce a `'a val` type that can be used to compile-time create conditions for typing dependent on that value. The value should not be computed via an expression but should be listed explicitly - there will have to be a notion of an "explicit" expression.

4.3 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.

4.4 Modules

4.5 Intersection and Union types

4.6 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.

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.