Wye: an Interface-Oriented Language

Version 2.0.0

Aditya Gomatam

November 26, 2024

Abstract

Wye started out as a project to learn more about compilers. It turned into an epic journey through the beautiful areas of type theory, the lambda calculus, and programming language design. Though originally intended to be a compiled, pure-functional language, Wye succumbed to my ideas about "interface-orientation," as opposed to object-orientation. Wye retain features of functional languages such as its basis in the lambda calculus, fast type inference, and a focus on functional rather than imperative design. Nevertheless, its functional origins are indicated in its name, as `y` is λ upside-down.

This document compiles the specification of Wye: its syntax, semantics, type system, intermediate representation, register allocation, and finally its translation into an assembly language of choice.

1 Introduction

Wye is a statically-typed, interface-oriented, functional language. Wye is heavily inspired by Rust, Haskell, and OCaml, as well as object-oriented languages like C++. It utilizes a Hindley-Milner type system augmented with basic record types (`structs`), bounded interfaces, enumerations, pattern matching, and so on. Wye's "interface-orientation" combined with its underlying Hindley-Milner type system, enables perfect type-inference simultaneously with encapsulation and code-sharing, while preventing as much as possible the the messiness that OOP can become.

For clarity of your code, and to aid the compiler, Wye allows the annotation of types of variables.

1.1 Features that Wye does not have

1.1.1 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.

1.1.2 User Input

In Wye version 2.0.0, users may only output to the screen. Taking user input is slightly more complicated and is thus deferred to a future version of Wye.

1.1.3 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);
```

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. $\backslash n$ refers to a newline or carriage return.

2.1 Grammar

2.1.1 Top-level non-terminals

$$
\begin{aligned}
Program &\rightarrow Statement+ \\
Statement &\rightarrow Expr \mid EnumDecl \mid StructDecl \\
&\quad \mid InterfaceDecl \mid ImplBlock \mid Main \\
Main &\rightarrow \texttt{main}\ \backslash\texttt{n}\ Expr\ \texttt{end}
\end{aligned}
$$

2.1.2 Expressions

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

2.1.3 Structures and interfaces

$$
\begin{aligned}
EnumDecl &\rightarrow \texttt{enum}\ TypeId\ (\texttt{'}\ TypeId?\ Id) * \texttt{=}\ (TypeId\ (\texttt{with}\ Type)?) \\
&\quad (\texttt{|}\ TypeId\ (\texttt{with}\ Type)?) * \\
StructDecl &\rightarrow \texttt{struct}\ Id\ (\texttt{'}\ TypeId?\ Id) * \ \texttt{\textbackslash n}\ (Id\texttt{:}Type\ \texttt{\textbackslash n}) +\ \texttt{end} \\
InterfaceDecl &\rightarrow \texttt{interface}\ Id\ (\texttt{requires}\ (Id\ \texttt{+}) *\ Id)?\ \texttt{\textbackslash n} \\
&\quad (ValDecl \mid MethodImpl \mid MethodDecl) +\ \texttt{end} \\
ImplBlock &\rightarrow \texttt{impl}\ Id(\texttt{'}\ TypeId?\ Id) * \ \texttt{:}Id(\texttt{'}\ TypeId?\ Id) * \ \texttt{\textbackslash n} \\
&\quad (AttrSet \mid MethodImpl) +\ \texttt{end} \\
ValDecl &\rightarrow \texttt{val}\ Id\ \texttt{:}\ Type \\
MethodDecl &\rightarrow \texttt{method}\ Id\ \texttt{:}\ Type \\
MethodImpl &\rightarrow \texttt{method}\ Id\ Id * \ \texttt{=}\ Expr \\
&\quad \mid \texttt{method}\ Id\ (\ (Id\texttt{:}Type)\ \texttt{->}\ ) + Type\ \texttt{=}\ Expr
\end{aligned}
$$

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

3.1 Expressions

3.1.1 Patterns

3.1.2 Main

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 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.