Wye: a Functional Language

Version 1.0.0

Aditya Gomatam

July 21, 2024

Abstract

To learn more about functional languages and compilers, I decided to build a functional language. This language is intended not only to have a minimal set of features so that it be relatively easy to implement, but also to be useful. As functional programming is based upon the lambda calculus, I have decided to name the language Wye, as `y` is λ upside-down.

This document will compile 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 functional language based on the lambda calculus. It utilizes a Hindley-Milner type system, which enables both parametric polymorphism and type inference. For clarity of your code, and to aid the compiler, Wye allows the annotation of types of Wye variables. Wye also allows the declaration of custom types.

1.1 Features that Wye does not have

1.1.1 Type Classes

The original ideas for Wye included the ability to create Haskell-like Type Classes, between which one could establish subtype relations, and moreover, by which type variables could be bound (for example, one could annotate that a function argument generalize over all `'Ordered` types, instead of all types). Due to the added complexity of implementing such a system, Type Classes are deferred to a later version of Wye.

1.1.2 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 so in keeping with the idea of a minimal set of features, are also deferred to a future version.

1.1.3 User Input

In Wye version 1.0.0, users may only output to the screen. Taking user input is slightly more complicated and is thus deferred to a future version of Wye.

1.1.4 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 `.y` file and is, in its most abstract view, simply a sequence of Wye statements.

Witten 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

$$
\begin{array}{rcl}
\mathit{Program} & \rightarrow & \mathit{Statement}+ \\
\mathit{Statement} & \rightarrow & \mathit{LetStatement} \mid \mathit{TypeDeclaration} \\
\mathit{LetStatement} & \rightarrow & \texttt{let}\ \mathit{Id}\ (\texttt{:}\ \mathit{Type} \mid \mathit{Id}* \mid ((\texttt{(}\mathit{Id}\ \texttt{:}\ \mathit{Type}\texttt{)}\ \texttt{->}) +\ \mathit{Type})?\ \texttt{=}\ \mathit{Expr}\ \texttt{;} \\
\mathit{TypeDeclaration} & \rightarrow & \texttt{type}\ \mathit{TypeId}\ (\texttt{'}\ \mathit{Id})*\ \texttt{=}\ (\mathit{TypeId}\ (\texttt{with}\ \mathit{Type})?) \\
 & & (\texttt{|}\ \mathit{TypeId}\ (\texttt{with}\ \mathit{Type})?)*\ \texttt{;} \\
\mathit{Expr} & \rightarrow & \mathit{IntLiteral} \mid \mathit{FloatLiteral} \mid \mathit{StringLiteral} \mid \mathit{List} \mid \mathit{Tuple} \\
 & & \mid \mathit{Id} \mid \mathit{BuiltinOp} \mid \texttt{print} \mid \texttt{error}\ \langle\text{variable within scope}\rangle \\
 & & \mid \mathit{TypeId}\ (\texttt{with}\ \mathit{Expr})?\ \langle\text{Type variant}\rangle \\
 & & \mid \mathit{Expr}\ \mathit{Expr}\ \langle\text{function application}\rangle \\
 & & \mid \mathit{Expr}\ \mathit{BuiltinOp}\ \mathit{Expr}\ \langle\text{reserved binary operator}\rangle \\
 & & \mid \texttt{match}\ \mathit{Expr}\ \texttt{\{}\ (\mathit{Pat}\ \texttt{=>}\ \mathit{Expr}\ \texttt{,})*\ \mathit{Pat}\ \texttt{=>}\ \mathit{Expr}\ (\texttt{,}?)\ \texttt{\}} \\
 & & \mid \texttt{\textbackslash}\ \mathit{Id}+\ \texttt{->}\ \mathit{Expr}\ \langle\text{lambda expression}\rangle \\
 & & \mid \texttt{(}\ \mathit{Expr}\ \texttt{)} \\
 & & \mid \texttt{\{}\ \mathit{Statement}*\ \mathit{Expr}\texttt{\}}\ \langle\text{let .. in block}\rangle \\
\mathit{Type} & \rightarrow & \texttt{int} \mid \texttt{float} \mid \texttt{string} \\
 & & \mid \mathit{TypeId}\ (\mathit{Type})* \\
 & & \mid \texttt{[}\ \mathit{Type}\ \texttt{]} \\
 & & \mid \texttt{(}\ (\mathit{Type}\ \texttt{,})+\ \mathit{Type}\ \texttt{)} \\
 & & \mid \texttt{'}\ \mathit{Id}\ \langle\text{type variable}\rangle \\
 & & \mid \mathit{Type}\ \texttt{->}\ \mathit{Type}\ \langle\text{function type}\rangle \\
\mathit{List} & \rightarrow & \texttt{[}\ (\mathit{Expr}\ \texttt{,})*\ \mathit{Expr}\ \texttt{]} \mid \texttt{[]} \\
\mathit{Tuple} & \rightarrow & \texttt{(}\ (\mathit{Expr}\ \texttt{,})+\ \mathit{Expr}\ \texttt{)} \\
\mathit{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{!=} \\
\mathit{Pat} & \rightarrow & \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 \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
\end{array}
$$

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`, `if`, 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 operations (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 Statements

There are two types of statements in Wye – let statements and type declarations. The following are valid examples of Wye statements:

3.1.1 Let Statements

Wye `let` statements declare constants (referred to throughout this document as variables for consistency with common notions of `let`) at the top-level, or within blocks. In Wye, functions are first-class – that is, they occupy the same status as "regular" types of values such as `int`s. They can be declared as variables, passed as function arguments, and so on.

Variables declared using `let` statements are visible throughout their entire scope, both backwards and forwards. Blocks create nested scopes. The variables declared in such nested scopes are not visible in outer scopes. Function arguments shadow over the outer scope. Here are some examples:

```
# variable (actually a constant)
let x = 4;
# type-annotated variable
let y: int = 4;
```

```
# function
let plus_4 = (+) 4;
# type-annotated functions
let plus_4 (x: int) -> int = (+ x 4);
let id: (x: 'a) -> 'a = x;
let into_tup (x: int) -> (y: float) -> (z: bool) -> (int, float, bool)
  = ((plus_4 x), y, (id z));
# blocks
let y = { 5 };
let doublesum lst: [int] -> int = {
  let sum lst: [int] -> int = match lst {
    [] => 0;
    x :: xs => (+ x (sum xs));
  };
  (* 2 (sum lst)) # no semicolon
}; # semicolon
# function argument type annotation
let func (g: int -> int) -> (x: int) -> int = g x;
```

It would be slightly uncomfortable to express this through the grammar, but Wye reserves a let statement of the following form at the top-level to be its Main statement – that is, the expression on the right hand side is run when the `.y` file is executed from the command line:

```
let Main = print (plus_4 6);
```

Apart from `Main`, every function in Wye is pure, i.e., it will not affect the scope outside of the function. There are no global mutable variables, and the only function that is allowed to call other functions that take input or output to the terminal is `Main`.

3.1.2 Type Declarations

Type declarations in Wye are akin to enums in Rust and abstract data types in Haskell. Variants of the type are separated by `|` and one may choose to give any variant an optional field. Unlike Rust, wherein enums can have multiple fields, Wye allows only one field. However, one can declare this field to have a tuple type, which will allow the variant to behave as if it holds multiple fields. Moreover, user-defined types can be *polymorphic* – i.e., they may have a type argument. For example:

```
type bool = false | true;
type Option 'a = None | Some with 'a;
type binary_tree 'a = Leaf
  | Node with ('a, binary_tree 'a, binary_tree 'a);
```

One can think of polymorphic user-defined types as “type functions" that take in a type parameter and output an instance of a type. In the above example `Some with 4` would take in the type `int` of the expression `4` and output an instance of `Option int`.

3.2 Expressions

The right hand side of every let statement in Wye must be an expression. Wye version 1.0.0 supports only a few expressions. Most of them exist in other languages and are fairly self-explanatory. This section will go over the less familiar ones.

3.2.1 Function Application

As a language that implements the lambda calculus, Wye supports function application, which expressed by the following substitution rule:

$$\mathit{Expr} \quad \rightarrow \quad \mathit{Expr}\ \mathit{Expr}$$

For example:

```
let f (x: int) -> (y: int) -> int = (+ x y);
let plus_4 (y: int) -> int = f 4;
```

Functions in Wye are *curried*. That is, the definition of `func` actually looks something like this under the hood:

```
function f(int x) {
  function g(int y) {
    return x + y;
  }
  return g;
}
```

Thus, in defining `plus_4` as `f 4`, `plus_4` is actually like the function `g`, which has captured the value of `4` from an outer scope, and which on input `y` returns `4 + y`.

This is also the reason why Wye's function types include an arrow between arguments. The function `f` has type `int -> int -> int`, the function `f 4` has type `int -> int`, and the value obtained from computing `plus_4 5` has type `int`.

Moreover, because of currying, Wye can safely evaluate functions left to right. That is, Wye functions are *left-associative*. So, for instance, the expression `f g 4` is evaluated as `((f g) 4)`. As a concrete example:

```
let f (g: int -> int) -> (x: int) -> int = (g x);
let double (x: int) -> int = 2 * x;
let z = f double 4;
```

Here, the function `f` takes in a function argument `g` of type `int -> int` and an integer argument `x`. The function `double` is a function of type `int -> int`. Most importantly, the expression assigned to `z` is evaluated as `((f double) 4)`. If we were to evaluate as `(f (double 4))` then this would supply the value 8 as the first argument to `f`, which is expected to be a function of type `int -> int`.

As mentioned earlier, under the hood, `f` is curried, and thus is really a function of *one* argument only. When we write `f double` we return the result of `partially applying` `f`. That is, `f double` returns a function that takes an integer argument (`x`). Then, `(f double) 4` actually computes the result intended to be stored in `z`, applying the partial function obtained from `(f double)`.

Now, if we were to have this example:

```
let f (x: int) -> int = x;
let double (x: int) -> int = 2 * x;
let z = f double 4;
```

then this would not compile, as `double`, which is of type `int -> int` is *not* of the type `int` that was expected as the first argument of `f`. Nevertheless, the following compiles:

```
let f (x: int) -> int = x;
let double (x: int) -> int = 2 * x;
let z = f (double 4);
```

as the parentheses make it clear that `double 4` is one expression intended to be supplied to `f`.

3.2.2 List Construction

Just like Lisp, OCaml, Haskell, and many other functional languages, lists are an important part of Wye. As such, Wye provides list construction via the "cons" operator `::`. Wye also supports pattern matching of lists using cons notation, which is talked about in 3.2.5.

The cons operator operates as follows: if x is a value of type t and l is a list of type `[`t`]`, with value `[`$a_1, a_2, ..., a_n$`]` where $n \geq 0$, then `x :: l` denotes the list `[`$x, a_1, ..., a_n$`]`.

The list construction expression falls under the *Expr BuiltinOp Expr* grammar substitution rule for *Expr*. Moreover, like other functions, `::` is itself curried.

3.2.3 Using variants of a user-defined type

If t is a custom type declared via a *TypeDeclaration* statement, with variants $t_1, ..., t_n$, then one may access the variant t_i as t_i. In version 1.0.0 of Wye, for simplicity of the parser, variants with the same name across different user-defined types is not allowed. Thus, the following Wye code compiles:

```
# synonyms of many
type SynMany = Numerous | Several | Plenty;
# synonyms of abundant
type SynAbundant = Copious | Profuse;
let x = Plenty;
let x = Copious;
```

but this does not:

```
# synonyms of many
type SynMany = Numerous | Several | Plenty;
# synonyms of abundant. ERROR - conflicting variants
type SynAbundant = Plenty | Copious | Profuse;
```

If the type variant has a field, then one should use `with` syntax to define the value of the field. For example:

```
type Option 'a = None | Some with 'a;
let x: Option int = Some with 4;
```

3.2.4 Match constructs

The `match` construct allows control flow of the program to be determined dynamically at runtime, based on whether or not a particular expression matches a pattern. The technical syntax of the `match` construct is as follows, with match arms separated by a comma (,):

$$MatchExpression \quad \rightarrow \quad \texttt{match}\ \mathit{Expr}\ \{\ (\mathit{Pat} \Rightarrow \mathit{Expr}\ ,)*\ \mathit{Pat}\ \Rightarrow\ \mathit{Expr}\ (,?)\ \}$$

or equivalently:

$$MatchExpression \quad \rightarrow \quad \texttt{match}\ \mathit{Expr}\ \{\ \mathit{Pat}_1 \Rightarrow \mathit{Expr}_1\ ,\ \ldots\ ,\ \mathit{Pat}_n \Rightarrow \mathit{Expr}_n\ (,?)\ \}$$

where $n \geq 1$.

The operation of the `match` construct is as follows: the *Expr* following `match` is first evaluated to a value v. Wye then checks at runtime whether v matches each of Pat_1 to Pat_n in order. Let i be the smallest number from 1 to n such that v matches Pat_i (what it means to "match" a pattern will be discussed later). This branch will be taken, and only Expr_i will be evaluated. The value to which Expr_i evaluates is then the value of the entire `match` construct. If v matches *none* of the patterns $\mathit{Pat}_1, ..., \mathit{Pat}_n$, Wye will throw a runtime error.

Each of $\mathit{Expr}_1, ..., \mathit{Expr}_n$ must evaluate to a value of the same type under all possible executions. Thus, the `match` expression will evaluate to something of exactly one type, under all possible executions. Note that Wye does not check for exhaustiveness of the patterns.

3.2.5 Patterns in a Match construct

The patterns that one can match against in a `match` are described by the following substitution rules:

$$\begin{aligned} Pat \quad &\to \quad \_\ |\ IntLiteral\ |\ FloatLiteral\ |\ StringLiteral\ |\ Id \\ &\quad\ |\ Id \texttt{ :: } Id \\ &\quad\ |\ TypeId\, \texttt{with}\, Pat \\ &\quad\ |\ \texttt{[}\ (Pat\ \texttt{,})*\ Pat\ \texttt{]}\ |\ \texttt{[]} \\ &\quad\ |\ \texttt{(}\ (Pat\ \texttt{,})+\ Pat\ \texttt{)} \\ &\quad\ |\ \sim\ Pat \\ &\quad\ |\ Pat\ (\texttt{|}\ Pat)+ \\ &\quad\ |\ Pat\ \texttt{if}\ Expr \end{aligned}$$

Throughout the subsequent discussion, let v denote the value to which the *Expr* after the `match` keyword evaluates. The following list describes when a particular pattern is matched:

- If $Pat \to IntLiteral|FloatLiteral|StringLiteral$, then the expression on the right is evaluated to some w, and v matches this expression iff there is exact equality of v and w. Note equality in Wye is always *by value*. This implies that if two expressions do not have the same type, they cannot ever be equal. Moreover, equality of two functions is *by identity*, not by the mathematical notion of function equivalence. However, this is consistent with by-value equality semantics, as functions are stored via pointer under the hood.

- If $Pat \to Id$, let x denote the Id that occurs on the right. If this x is a variable already present in the outer scope of the `match` construct, then the value of this x is taken and exact equality is checked between the value of x and v, as described above. If such x is the name of a variant of a declared type, we check that v is a type variant with this type. If x is the name of a type variant that had a field (which is not specified in this pattern), then a type error is thrown. Otherwise, if x does not exist in the outer scope as either a variable or a variant of some declared type, Wye will catch this as an undefined symbol error.

- If $Pat \to \_$ then v always matches the pattern. _ is a wildcard in Wye.

- If $Pat \to Id :: Id$, then v matches the pattern iff v is a list of type `[`t`]` with at least one element. This pattern type is better explained with the following example:

```
match e1 {
  x :: s => e2
}
```

Suppose the expression `e1` evaluates to the value v. Then v matches `x :: s` iff:

1. v is of some list type `[`t`]`
2. v has at least one element

If v matches the pattern, then the *new variable* `x` will be set to the first element of v (the "head" of v) and the *new variable* `s` will be set to the list containing the remaining elements of v, in the original order (that is, `s` will be set to the "tail" of v). `x` and `s` will be available for use in the scope of `e2` ONLY, and will shadow over variables with the same name from the outer scope. v will always fail to match such a "list-construction" pattern if it is a list of 0 elements. Note that if v is a list of 1 element, `x` will always be set to that element, and `s` will be an empty list.

- if $Pat \rightarrow TypeId$ `with` Pat, then b must evaluate to a variant of a type defined in a $TypeDeclaration$ statement *that has a field*. Now, if the Pat after the `with` is an identifier Id, the semantics are similar to the list destructuring pattern `x :: s`. That is, if the Id that occurs is `x`, then `x` is a new variable visible in the scope of only the expression of that `match` arm. For example:

```
type Option 'a = None | Some with 'a;
let print_optional (opt: Option string) -> string = match opt {
  None => print "nothing!",
  # assume that cat concatenates two strings
  Some with x => print (cat "some with" x),
}
```

  This Id will shadow over a variable with the same name from the outer scope. On the other hand, if the Pat after the `with` is a integer, float, or string literal, v is checked not only to be of the specified variant type, but to have a field that is exactly the specified literal. Similarly,

- If $Pat \rightarrow$ `[]` then v matches the pattern iff it is an empty list (and it type checks).

- If $Pat \rightarrow$ `[`$(Pat$ `,`$)*$ Pat`]`, then v must be a list type in order to match the pattern. Let p denote this pattern. Then v matches p iff:

  1. v is a list of the same length as p (note in this case, $|p| \geq 1$)
  2. if v_i is the i^{th} element of v and p_i is the i^{th} pattern in p, then for each i, v_i must match p_i

  If any variable-introducing patterns occur as elements of the above list, such as $TypeId$ `with` Id or Id `::` Id, then such variable names must not conflict across list elements, and all are available for use in ONLY the scope of the expression that follows the `=>`.

- If $Pat \to$ `(`$(Pat$`,`$) + Pat$`)`, then v must be a tuple type in order to match the pattern. Let p denote this pattern.. Then, v matches p iff:

  1. v is a tuple of the same length as p (note in this case, $|p| \geq 1$)
  2. if v_i is the i^{th} element of v and p_i is the i^{th} pattern in p, then for each i, v_i must match p_i (in type as well).

  If any variable-introducing patterns occur as elements of the above tuple, such as $TypeId$`with`Id or Id`::`Id, then such variable names must not conflict across tuple elements, and all are available for use in ONLY the scope of the expression that follows the `=>`.

- If $Pat \to \sim Pat$, let p be the LHS Pat and q be the RHS Pat. Then v matches p iff v does not match q.

- If $Pat \to Pat$ (`|` Pat)+, i.e., if $Pat \to Pat_1$ `|` ... `|` Pat_n where $n \geq 2$, then v matches Pat iff v matches some Pat_i where $1 \leq i \leq n$. Note that variable-introducing patterns such as Id `::` Id are NOT allowed in the union pattern, as such variables will have no meaning if v does not match that particular pattern of the union.

- If $Pat \to Pat$ `if` $Expr$ then, letting p denote the Pat on the left hand side, q the Pat on the right hand side, and e the $Expr$, v will match p iff v matches q and e evaluates to the `bool` variant `true`. Note that the `bool` type is declared in the Wye prelude and is thus available for use like this. Moreover, note that Wye employs short-circuiting in the evaluating of `match` guards – e is not evaluated if v does not match q.

3.2.6 Function overloading

Wye does not support function overloading of user-defined functions. However, certain functions, such as `+` are defined on multiple types, which the compiler is able to handle. For a reader interested in how one could handle function overloading in a functional language, look up Haskell's type classes and kinds.

3.2.7 Builtin Operations

There are a few builtin operations in Wye:

`+` adds two `int`s or `float`s or concatenates two `string`s or lists. Wye does not support addition of `int` and `float`. An explicit cast must occur (more on this later).

`-` an expression `a - b` results in the subtraction of the value `b` from the value `a`. Subtraction is supported only on `int`s and `float`s.

`*` multiplication of `int`s or `float`s.

`/` an expression `a / b` results in the division of the value `a` by the value `b`. In Wye, division is supported *only for* `float`*s*. If `b` is the value 0, then Wye will throw a division-by-zero error at runtime.

`//` an expression `a // b` results in the floor division of the value `a` by the value `b`. In Wye floor division is supported *only when* `a` *is a* `float` *and* `b` *is an* `int`. If `b` is the integer 0, then Wye will throw a division-by-zero error at runtime.

`==` an expression `a == b` results in the `bool` variant `true` if the value `a` and the value `b` are exactly the same integer, float, or string literal, or exactly the same function pointer, or lists or tuples whose elements are equal according to this rule. Otherwise, this expression results in the `bool` variant `false`. Wye will throw a type error if the arguments to `==` are of dfiferent types.

`!=` an expression `a != b` is `true` iff `a == b` is `false`. Wye will throw a type error if the arguments to `==` are of dfiferent types.

`<` an expression `a < b` results in the `bool` variant `true` if `a` is an integer or float that is less than the integer or float `b`. Otherwise, it results in the `bool` variant `false`. `a` is a float iff `b` is a float; `a` is an integer iff `b` is an integer. If either `a` or `b` is not an integer, then a type error is thrown. Note that Wye does not support the float `NaN`, and thus a total ordering of the float type is possible.

`>` an expression `a > b` results in `true` iff `a < b` is not true and `a == b` is not true. Note that this operation is supported only on integers or floats, and a float cannot be compared with an integer with `>`.

`<=` an expression `a <= b` is `true` iff `a > b` is `false`, otherwise it is `false`.

`>=` an expression `a >= b` is `true` iff `a < b` is `false`, otherwise it is `false`.

`::` as discussed in 3.2.2, `::` is the list construction operator in Wye for constructing a list via specifying its head and tail.

Wye's builtin binary operations can be written both in postfix notation (for example, division can be written as `(/) a b`), or in infix notation (`a / b`). The parentheses around these binary operators in postfix notation aids the parser in disambiguating.

3.3 Types

3.3.1 Basic types

Wye has essentially 3 builtin primitive types, with support for constructing list and tuple data structures. The three primitive types are:

`int`: a 64-bit signed integer.
`float`: a 64-bit floating point number.
`string`: essentially a list of 4-byte unicode characters.

An expression `[e_1, ..., e_n]` is of list type `[t]` if e_i is of type t for all $1 \leq i \leq n$. An expression `(e_1, ..., e_n)` is of tuple type $(t_1, ..., t_n)$ if each e_i has type t_i.

3.3.2 Polymorphic types

Just like OCaml and Haskell, Wye's type system is based on the Hindley-Milner type system and thus supports polymorphic types. A polymorphic type is essentially a type of the form $\forall \alpha\, \sigma$ where σ is a type signature in which α is a free variable. Such type signatures have been referred to earlier, for example in the `Option` type:

```
type Option 'a = None | Some with 'a;
```

Or, in the identity function:

```
let id (x: 'a) -> 'a = x;
```

In Wye, the ' token specifies that the Id immediately following it is a *type variable* which is implicitly quantified over in a type expression (an expression that evaluates to a type). In this way, the function `id` can take in `any value` of `any type`! The function's argument is polymorphic.

Whenever the `id` function is called, it actually implicitly takes in a *type argument* – namely, the type of its actual argument! Due to Wye's type system, this type is actually known at compile-time, and thus type checking can be adequately performed via substitution of the quantified type variable `a` with whatever type the function is called with. For example, the following code type-checks, with the type variable `a` being substituted with `string` and the type variable `b` being substituted with `int`:

```
type OptionalTuple 'a 'b = None | Some with ('a, 'b);
let f (x: 'a) -> (y: 'b) -> OptionalTuple 'a 'b = Some with (x, y);
# j is of variant Some with (string, int)
let j = f "hello" 4;
```

3.3.3 Type inference

Wye is a statically typed language, and due to its Hindley-Milner type system, it is able to infer the types of expressions (and thus variables) at compile-time. The exact rules for the type system will be discussed in greater detail in the Type Checking section of this document. However, it is important to note that Wye expressions, when not annotated with a type, are typed to the *most general possible type*. For example, in the following code:

```
let cons x lst = x :: lst;
```

the function `cons` type checks to the functional type `'a -> ['a] -> 'a`. For more information, check out the the Hindley-Milner type system.

4 Type Checking

In order to verify at compile-time that a program accomplishes the author's intention, it is helpful to type-check the program at compile-time. Type checking is the process of verifying that all operations in the program are performed on values of appropriate types. Type checking can be performed at compile-time only if the types of every Wye expression are known at compile-time. Thus, if the Wye compiler cannot figure out the type of a Wye expression, the compiler will refuse to compile the input program. Type checking is achieved via deductive reasoning on the input program based on certain deductive rules. In version 1.0.0 of Wye's compiler, type checking is implemented via a depth-first traversal of the Abstract Syntax Tree of an input Wye program.

4.1 Notation

The rules by which Wye performs type checking are written in the following form:

$$\frac{hypothesis_1, ..., hypothesis_n}{conclusion}$$

In order to perform typing, a context is necessary. For example, if a function `f` is applied in an expression, then `f` must be defined somewhere. The set of variables, functions, and types available in scope for use in an expression is referred to as the *context* and notated as Γ.

The symbol $\vdash$ means “entails" or “proves." We say $\Gamma \vdash e : \tau$ if the context Γ and the deductive rules of Wye's type system prove that the expression e has type τ. If interested in the origins of this symbol, the reader should look into soundness and completeness in mathematical logic.

A comma between hypotheses indicates a logical and. That is, $hypothesis_1, hypothesis_2$ is logically equivalent to $hypothesis_1 \wedge hypothesis_2$. Moreover, Γ can be considered a hypothesis itself, an assertion that all declared variables and functions have their specific types, along with assertions about the available types. Thus, the notation $\Gamma, hypothesis \vdash e : \tau$ may be used to express that $\Gamma \wedge hypothesis$ proves that $e : \tau$.

In general, τ will be used to indicate a monomorphic type (i.e., a “concrete" type like `int`), and σ will be used to indicate a polymorphic type. Any type with a type variable, i.e., with an implicit quantifier over a type parameter, is a polymorphic type.

Also important to consider are the free type variables in a type expression. For example in the identity function:

```
let f x: 'a -> 'a = x;
```

within the expression on the right hand side of the `=` (namely, `x`), the type of the parameter x is free. It is equal to some `a` which is only concretized when `f` is applied to some parameter.

On the other hand, there are no free variables in `f`'s type. `f` has the type $\forall a\, a \to a$ – all the type variables in this function definition have been fully quantified over.

The set of free variables in an expression or set of expressions S is notated as $\text{free}(S)$. free is defined as follows:

1. if α is a type variable, $\text{free}(\alpha) = \alpha$.
2. if C is a type function (a variant of a user-defined type with a field), then $\text{free}(C\ \tau) = free(\tau)$
3. if Γ is a context, then $\text{free}(\Gamma) = \bigcup_{x:\sigma\in\Gamma} free(\sigma)$ where the union is taken over all distinct polymorphic types of variables or functions in Γ.
4. for polymorphic types, $\text{free}(\forall\alpha\ \sigma) = \text{free}(\sigma) - \{\alpha\}$
5. for contexts, $\text{free}(\Gamma \vdash e : \sigma) = \text{free}(\sigma) - \text{free}(\Gamma)$

4.2 Type order

Before we give the rules for type inference, it is important to define an ordering on types. This allows Wye to infer the most general type possible for an expression (the "least" potential type in this type ordering), which creates a highly flexible yet still strict type system.

4.3 Typing expressions

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 of UCLA for teaching me the fundamentals of compiler implementation.