Hazel Programming Language Definition

1 What Is Hazel

Hazel is an declarative, Purely Functional Paradigm Programming Language. Similar to other pure functional paradigm languages, like Haskell for example. Hazel runs on pure functions, all computations happen via functions with no mutation.

However unlike Haskell, Hazel takes a more verbose approach. It expects you to derive features Haskell provides for free using existing language features. This is intentional as it meant to be less intimidating then Haskell. You can know what exactly is going on.

1.1 Origins

Hazel first appeared as a pseudo code language, I created improving my old language, perennial. I realized how verbose perennial is and removed some of the verbosity as well as making it more clearly pseudo code and got rid of the abbreviations and replaced them with more strongly worded keywords.

It evolved to a functional paradigm language. and I gave it the name "hazel" as its a softer name then "Haskell"

2 Comments

Comments are parts of your source file the compiler will ignore. in hazel they are defined as pascal style, enclosed in parentheses followed by a star.

```
1         (*this is a comment*)
2
3         (* They Also
4             can be multi line *)
```

3 Types

Types are how you represent different data in hazel. All types are immutable and are Optional(T), they are either have a Some(T) or a None(T).

3.1 Native Types

native Types are the simplest types. The smallest native Type is boolean and the biggest is unsigned_integer or a function type.

Type	Accepting Values
boolean	The smallest type, it only stores a true or a false value. as well as an expression that evaluates to true or false
byte	stores an integer whole number ranged from $\pm 2^7$
integer	stores an integer whole number ranged from $\pm 2^{63}$
decimal	stores a floating point decimal number, that has 15-17 digits in floating point precision and ranges to $\pm 1.8 * 10^{308}$
character	stores a singular letter that is equal in size and range to an unsigned 8 bit integer, you define them via single quote
string	an array of characters, you define them via double quote
unsigned_byte	integer whole number ranged from 0 to 2^8, wraps around to 0 if reaches overflow
unsigned_integer	integer whole number ranged from 0 to 2^{64}, wraps around to 0 if it reaches overflow
(type of parameter...) : return type	this denotes the type of a function. or a function pointer in other languages, the type of the parameters and the colon return type designates the return type of the function pointer your computing.

3.2 None

None is null in hazel. All types are optional types. And can have some-value or None value

You denote None types with the none keyword, and Some types with the actual value

```
module hazel_lists

let none_function : integer => None (*this is none*)
let some_function : integer => 1 (*this is some*)
```

Listing 1: How Lists are defined in hazel

for some types that are none. mainly numerical it will instead be denoted as a default value. if you try to reference the value.

however for non numerical types. it will be null if you reference the value. so it may crash.

3.3 Lists

Lists are Hazel's primary data structures. They are ordinary lazy linked-lists. Lazy meaning they are computed and allocated efficiently.

too have a list to type, they are defined as brackets with the name of the type you want in the list. so an integer list will be denoted as, [integer]

```
module hazel_lists

let get_list: [integer] => [1,2,3,4,5]
```

Lists, are 0 based. The start index is 0, are fixed size to the max elements you appended into the list, and are immutable. meaning if you concat to another a whole new list pointer is created.

Too index into a list you use the "!!" operator. which looks like this

```
module hazel_lists

let get_list: integer => [1,2,3,4,5] !! 1
```

You can also have a ranged list, which is defined as an open bracket with the start and an end. the start and end must be constant integers. and they must be an enclosed range.

and what ever your start and end is, is the size of the list.

```
module hazel_lists

let range_list_foo: [integer] => [1..4]
let range_list_bar : [integer] => [4..1]
```

ranged lists are allocated up to the amount you access and computed up to the amount you access. this is because lazy eval. Hazel also supports infinite and data structures.

```
module hazel_lists

let infinite_list_foo: [integer] => [1..]
```

Listing 2: Infinite List

3.4 Function Types

a function type is a type to represent a function pointer. they are denoted as (type of parameters...) : return type

```
module hazel_lists

let function_type: (integer, integer) : integer
    let add(let a : integer, let b : integer) => a + b
    return add
```

Listing 3: A function that returns a function

it doesn't compute add but returns a pointer to add which you can call if you call the function

```
module hazel_lists

let function_type: (integer, integer) : integer
    let add(let a : integer, let b : integer) => a + b
    return add
let do_stff : integer => function_type. (1,1)
```

Listing 4: A function that returns a function

3.5 Structures

structure, is a singular type container that stores different functions You define them with the "Structure" keyword followed by the name, which will be the type.

and you use the instance function to decide what they contain

```
module hazel_lists

structure Node
    instance(let left: Node, let right: Node, let data: integer)
```

Listing 5: A function that returns a function

to create an instance of a structure you write the name of the type followed by the parameters of the instance method

```
module hazel_structures

structure Number
    instance(let data : integer)

let get_node : Node => Number(1)
```

you can also have predetermined functions inside a structure you cant set these functions. and, you can overload the instance method. when you overload it all values in the structure in the other instance method that aren't set in the current will be set to none

```
module hazel_structures

structure Number
    instance(let data : integer)
    instance(let data2 : decimal)
    let add_one : integer => data + 1
```

to avoid ambiguity with this. there is a this keyword which looks like this

```
module hazel_structures

structure Number
    instance(let data : integer)
    instance(let data2 : decimal)
    let add_one : integer => this.data + 1
```

3.6 Enumerated Types

Enumerated Constants, are types that are just integer constants. From 0 to how many constants you have.

to define them. you use the "type" keyword followed by the name of the type and the values associated with that type like this

```
module enumarations

type Colors => [red, blue, green]

(* to use the type this is how you use it *)
let get_color : Colors => red
```

3.7 Generics

Generics let you have type parameters. Which could be useful to avoid overloading too much They are defined via the "Type" keyword. And you can reference them with the angle brackets.

like this

```
module generics

let generic_function(Type T, let args:<T>): <T>
let multi_generic_function(Type T, Type K, let args:<T>): <T>
```

This function takes a type parameter generic T and the type of argument is whatever you set Type T is How the call of this function would look like. you just add the type you want.

```
module generics

generic_function(integer, 1)
```

you can also add them to structures via the Instance function you just put the generic as Type T in the instance function and any function that gets defined by the generic you define it like this

```
module generics

structure ListStructure
    instance(Type T, let list : [<T>])
    let get_list : [<T>] => this.list
```

4 Operators

4.1 Math Operators

Math operators in hazel will work with any 2 numbers of the same type yield a number that is the same type of the 2 numbers you provided, so an integer + integer yields an integer

Unlike most functional languages. there is no function notation for operators and no operator overloads. You must do lhs operator rhs for binary operators or operator value. for unary operators

Operator	**Function**	**Example**
+	Addition, Adds any 2 numbers together	1 + 1
-	Subtraction, subtracts 2 numbers together	1 - 1
*	Multiplication, Multiplies 2 numbers together	1 * 1
/	Division, Divides 2 numbers, assuming the denominator is not 0	1/2
>>, <<	bit-shift left and bit-shift right, this is an integer exclusive operator that shifts the bits of a number, yielding a result that's divide by 2 or multiply by 2	$1 << 2, 1 >> 2$
∼	bit wise not ,unary operator, that works on all integers and boolean. It performs bit wise not on the bits of a number	∼ 1, ∼true
and	performs bit-wise "AND" between left hand side and right hand side. it works on both Integers and boolean types	1 and 1, true and false
or	performs bit-wise "OR" between left hand side and right hand side. it works on both Integers and boolean types	1 or 1, true or false
^	performs bit-wise "XOR" between left hand side and right hand side. it works on Integer types only	1 ^ 1

4.2 Comparison Operators

Comparison operators in hazel, will work with any 2 numbers of the same type, however it yields a value that is type boolean. so true and/or false

Example 1 = 1 yields true

Operator	Function	Example
=	Equals, returns true if the 2 numbers are equal	1 = 1
=/	Not Equals, returns true if the 2 numbers are not equal	1 =/ 1
>	Greater Than, returns true if left hand-side is greater then right hand-side	1 > 1
<	Less Than, returns true if left hand-side is less then right hand-side	1 < 1
>=	Greater than or equal to, returns true if left hand-side is greater then or equal to right hand-side	1 >= 1
<=	Less than or equal to, returns true if left-handsize if less then or equal to right hand size 0	1 <= 1

Table 1: Types in hazel

4.3 List and String Operators

Hazel also has operators for lists and strings they only work if and only if the lists holds the same type.

and they don't mutate the type. they create a new new list or string after every concatenation

Operator	Function	Example
++	Concatenation, concats the left hand side with the right hand side	[1,2] ++ [1] = [1,2,1], "hello" ++ "world" = "helloworld", "world" ++ "hello" = "world-hello"
!!	Index in. returns the value at the index of the right hand size.	[1,2,3,4] !! 0 = 1, "hello world" !! 1 = 'e'
!! (start : end)	Sub-list, returns a sub list at the range	[1,2,3,4] !! (0:2) = [1,2], "hello world" !!(0:2) = "he"

Table 2: Types in hazel

5 Functions

Functions in hazel are used for computations, they are strictly pure functions, a pure function meaning "a function is a pure function iff, for every same input

“a” yields the same output “b””. with no side affects or mutations

a function in hazel is defined by key word ”let” followed by the name of the function. you define the return type of a function with the ”colon” character followed by the name of the type. all functions must have a return type.

you use a new line or an arrow to define the scope. if you want the scope to be one line. use the arrow, if not then use a new line. and you return with the keyword return

```
module hazel_functions

let foo : integer => 1 (* one line *)

let bar : integer (* 2 lines *)
    return 1
```

Listing 6: Function definition

for functions that take parameters you add parenthesis followed by the functions you want to take as input

```
module hazel_functions

let foo(let a : integer, let b : integer) : integer => 1
```

Listing 7: Functions with parameters

Functions are also memoized, the value of the return of a function is cached. to decrease performance overhead, and the amount of function calls.

5.1 Function calls

too call a function that takes parameters you do this

the name of the function if it takes no parameters add no parenthesis if it does take parameters add parenthesis

all parameters in a function call are not computed unless you actually use it

```
module hazel_functions

let bar : integer => 1
let foo(let a : integer, let b : integer) : integer => 1
let baz : integer => foo(1,2)
let b : integer => bar
```

Listing 8: How to define a function call

5.2 Anonymous Functions

Anonymous Functions, or Functions that can be denoted as an expression, and are not of Scope. meaning unless you set it as a parameter or have a function that returns it you cant call them

they look just like Haskell’s anonymous function. you denoted via a backslash, parameters and an arrow for the body like this

```
module hazel_functions

(* consider map *)
let v : integer ...

let map(...)

let function: integer => map(v, \(let n : integer) => n + 1)
```

you can use the indentation scope for more complex anonymous functions, however its recommended you only have it do one expression for neatness. and avoid defining functions or anything else more complex in side an anonymous function

6 Conditionals

Conditionals, are similar to match in other languages. But you can't project outputs based on an expression. Instead it has a list of boolean expressions and if one happens to be true it terminates while executing the function. you cant define a conditional in a statement and must be defined in a return

conditionals are sort of a, type less function that takes 0 parameters, meaning you cant have it be a function pointer. or have it be a function type. to define a conditional you use the conditional keyword, followed by the type and give it different boolean expressions and expected output. if no boolean expression is true the computer evaluates the $default keyword branch. you must have a default branch. and The default branch must be defined after you define your other conditions.

```
module program

let factorial(let n: integer) : integer =>
conditional: integer
    n = 1 => n
    $default => n * factorial(n-1)
```

Listing 9: How to define a conditional

you can let it be multiple lines, and you can have conditionals that return other conditionals as well as define functions.

```
module program

let traverse(let node: Node) : Node =>
conditional: integer
    node = None => None
    $default
        return
        traverse(node.left).
        traverse(node.right)
```

Listing 10: Multi-line conditional branches

7 Modules

A module is used too have more organized code, reusable code by allowing you to create isolated and controlled name spaces. to define a module. You use the "module" keyword and the name of your module.

Modules also don't require indentation. and a module ends when there is another module keyword or the compiler runs out of valid hazel code

```
module program
```

Listing 11: How to define a module

You can also import modules into other modules. The way to do that is by adding parenthesis next to the name of the module and adding the name of the modules you want to import.

```
module program(foo_module,...)
```

Listing 12: How to import Other modules

when you import, it only imports into the module you are importing into. and you Will not be able to access the module via dependency. (meaning if C depends B and A depends on B. A will not access module C, unless you import C into A)

All functions and types are private in a module unless you add the export attribute right before you add the function definition or type definition. You do that by adding the export keyword right before the let keyword.

```
module program(foo_module,...)

export let foo : integer => 1

export structure bar_struct
    instance(let f : integer)
```

Listing 13: exporting

the export keyword is for only global scope. if you define the export keyword in a local scope it will be an error

8 Blocks

hazel uses indents and de-dents to define blocks. one block equals one indent. and it goes until there is a de-dent.

one indent is 1 tab or 4 spaces and 1 de-dent is minus 4 spaces or minus 1 tab

```
module program

let bar :integer
    let function : integer
        return 1
    return 1
```

blocks can be one line with the arrow operator, defined by "equals" on the left and greater then symbol on the right.

```
module program

let function : integer => 1
```

if you need to you can define a new line for it

```
module program

let function : integer =>
1
```

9 Scope

Hazel utilizes Lexical Scope in local scope, meaning all inner functions can reference the functions defined before it in outer functions. however inner functions defined after. cant reference the functions in defined in the inner functions before.

```
module program

let foo : integer =>
    (* its scope is foo *)
    let bar : integer =>
        (* its scope is bar, foo and baz *)
        let baz : ineteger =>
            return bar
        (* its scope is just foo bar and baz *)
        return baz
    (* its scope is bar *)
    return bar
```

In global scope, you can define any function inside the module even before it is defined. and any function exported to the module.

```
module A

export let someCalcA : integer => b
let b : integer => 2

module B(A)

let function : integer => someCalcA
```

10 Lazy evaluation

Lazy evaluation, Nothing is computed until it's used. (computed meaning the actual math computation. not the compilation) Its most famously seen in languages such as Haskell. and is an efficient way of using computational resources

Typically most languages such as C are static evaluation, meaning all computations are bound to variable expressions, and are computed even if your not

using it. This is sufficient for C, as C is for more lower level design. so it works as a sort of error checker. "fail fast". h

However static evaluation is not sufficient for hazel, because everything is a function. all computations happen with in functions. and you cant static evaluate a function. suppose it takes parameters and does computations with the parameters.

```
let bar : integer => bar

let foo(let a: integer, b: integer) : integer =>
conditional: integer
    a = 1 => a
    a = 2 => b
    $default => 0

let baz : integer => foo(1, bar)
```

Listing 14: consider these 2 functions. bar a recursive function and foo that takes parameters a and b and compares a results

In a static-evaluated language, this would be a stack overflow error. bar, would of been computed already and set the parameter to b. however in a lazy evaluation language, this wouldn't be a problem. It would see a = 1 and return 1. not bar so it wouldn't compute bar.

Consider another example

```
module lists

let foo : [integer] => [1,2,3,4]
let bar c : integer => foo !! 2
```

Listing 15: consider a function that returns a list and your indexing the list

in a normal static evaluated language this would allocate and compute 4 integers. however in Hazel it only allocates 2 integers and computes 1 integer. This is due to lists being allocated up to what you need, since lists in hazel are linked list and to access the nth element in a linked list, you must allocate the list up to the nth element to access the nth element, in that case its 2 integers.