Birch Language Specification

January 22, 2024

1 Purpose

Birch sets out to be a language that merges the best of functional and OOP into one style, rather than merely supporting both paradigms, cutting down the breadth of possible ways to write a given piece of code. Statically typed, AOT compiled, and as minimal as possible while being reasonably safe.

2 Overview

Modules Modules define all structural scope in Birch. Any Birch file is a module, and any public module may be imported by any other module using the import keyword. Modules also can define data structures, which is dicussed in User Defined Types.

User Defined Types

Modules Any module may have an internal data structure defined, which will always be an algebraic type. If a module has a data structure, it may be instantiated. (If tuples removed, then modules will need destructuring syntax)

Tuples Tuples may be prototyped, and elements given names. A function returning a tuple, may also give names to the elements of the tuple, with out prototyping the tuple ahead of time. Elements may be accessed by name or index. Tuples may be automatically destructured, which will not preserve the element names. (Depending on how stream lined modules become, perhaps tuples will be removed entirely)

Aliasing Any type may be aliased, though no further functionality will be added to the type.

Functions

3 Types

u8-64
i8-32
f32-64
usize
Future specifications would add homogenous vector types that will natively support SIMD operations.

4 Reserved Words

let
priv
expose
import
pub
if
else
match or case tbd
for
while
mod
data or type tbd
tup may not be needed depending on how type is implemented
to
as
unsafe or trust
none
self
Self

5 Operators

. . = + − ∗/|& $(opfromabove) = <, >, = (opfromabove) =$

6 Symbols

[]()′” ; ?_

7 Tokens

8 Grammar

prog mod_list main decl_list decl decl_list decl

9 Notes

for consideration, making some form of packed data that is native to the language consider making it such that the / op actually just makes a num effectivley 1/x figure out scoping symbol useage may change it such that modules can be compiled at run time, in which case the syntax for a mod would change to mod

_ = ... perhaps files wont be treated as modules, thus importing the explicit module will be required will need to resolve the piping idea with the .= operation consider making the top level mod in a file, if the only one, not require any scoping, maybe signify it as such some how modules and funcions are pre annotated
mod data Self = x: i32
mod data Self = & x: i32 h: i32 SomeType | y: i32 z: i32 OtherType | y: i32 z: i32
func f(self)
... type =, 1, 1 -> f -> ...