Project Eia Vision

So far I've built the best interpreter implementation I could.
Now it's probably time to plan about the compiler and runtime design of the language.

The last time I tried building a compiler was just a few months ago, that too in C++. Although it was quite a successful project in terms of me getting to know C++ more with a learning experience. But it wasn't as good as I wanted it to be in terms of implementation.

I'll be focusing on designing a byte-code interpreter for the language. Compiler for now will be in Kotlin. Runtime (VM) probably will be in either C++ or Kotlin itself.

(this document will be mostly many ideas and scrambles from my head that i want to write down somewhere, so that i can revisit them later)

How I want it to be (what? the bytecode and compiler design)

1. Do not make it fancy or complex, keep each instruction really simple.

```
println(8 + 2)
```

Would get broken down into:

```
Push Int 8
Push Int 2
Plus
Println
```

We should dump the bytecode in a non recursive and linear way.
Means each instruction only does one task, it does not trigger the next task until the runtime moves to next instruction.

For a code like:

```
2 + 3 + 4
```

Should broken down this way:

```
Plus Int 2
Plus Int 3
Add
Plus Int 4
Add
```

And not this way:

`Plus Plus Int 2 Int 3 Int 4`

Plus reads two ints -> (Plus reads to Ints -> 3, 4), 4
Which is really ineffective, bytecode shouldn't be like a lisp language.

2. First design the bytecode, also most importantly there SHOULD be a way to represent the bytecode in text like form as the above and not just pure raw bytes for sake of readability. Also additionally make it such that You can compile the raw bytecode text file → into Eia Bytecode! (Since thats just like mapping of the keywords) it would be a fun challenge to begin with.

Ok so goals are

1. Keep bytecode extremely simple
2. Start with designing bytecode, dont directly integrate it with the language, design a simple bytecode converter from raw human files →raw bytecode files and get a good runtime VM working.
3. We need to ensure that Eia optimizes the code before dumping bytecode. Like eliminating un-observed expressions, etc.
4. Start slow integration with Eia Code

Imagining Eia Code

My imagination process, how I envision.

```
let text = "meow"
println(text.len + 8)
```

```
String meow\0
Var
Get Var [0, 0]
StrLen
Int 8
Plus
Println
```

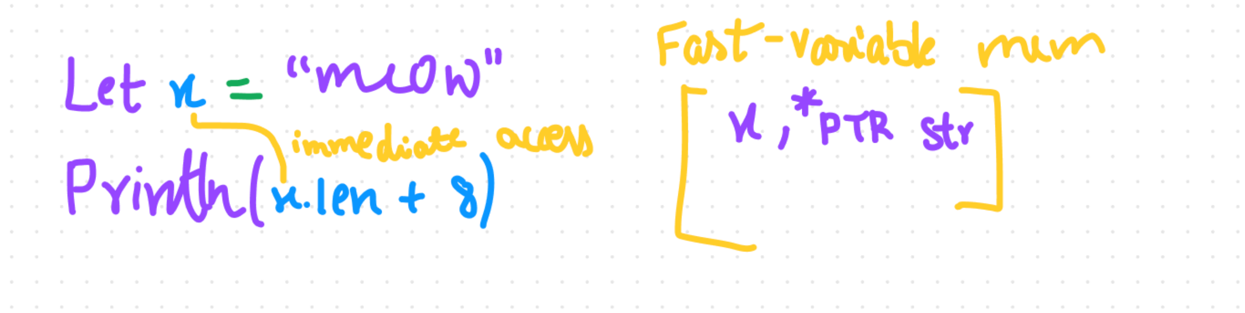
Hey wait?? What If I was somehow able to link the last variable declaration to the upcoming property access. We could optimize the branching so that it's already in the stack

How:

> If a variable is declared, and suddenly the next statement, one of it's property is accessed, then it's best for the variable's value to remain in stack so that we dont need to fetch it from the memory (that takes time)

So our memory spaces:

1. Linear memory (to do stuff like +, -) a stack memory
2. Scope memory (Variables, functions) are declared
3. Immediate variable memory (where most recently accessed variable memories are stored)



An image representing usage of Fast-Variable-Memory. Ok maybe FVM is limited to that particular scope only?

Other small things:

- All the variables are pre-indexed (scope travelling indices) like the Eia Interpreted Runtime scope implementation

Current mem impl:
https://github.com/XomaDev/Eia64/blob/main/src/main/java/space/themelon/eia64/runtime/Memory.kt

Kita Mem impl: https://github.com/XomaDev/kita/blob/dev/kita/runtime/name_resolver.cpp
https://github.com/XomaDev/kita/blob/dev/kita/runtime/memory_manager.cpp