HONK

FINAL PROJECT FOR COMPILERS

JUNE 3, 2020

The project

The purpose and objective

Here we are, the end of the infamous Compilers class. So, what did we learn?
Compilers are so much more pain than I initially thought, that's one.

The point of this class was to set out on a learning journey to understand how compilers as a whole functioned at its core, and then use those learnings to make a *pseudo*-compiler of our own (I say "pseudo" because it's technically still an interpreter, but it's simulating lower-level instructions through "quadruples" and virtual memory instead of an easier architecture). And I, for one, did learn quite a bit of those inner workings, and since I'm writing this document, I think it's safe to assume I managed to reach that goal of a functioning compiler!
Was it painful? *Very*. *More than I'd like*. But I guess no pain, no gain. The whole scope of this entire project was to learn those components of compilers to make our own and that's how far we reached. While ***Honk*** is still somewhat barebones, it's what this project was set out to achieve, learn and reproduce.

Requirements and use cases

The requirements of ***Honk***'s functionality are based on a set of requirements for a proposal called *Patito++,* with a number of changes and touches to make it more unique and a little more functional. These requirements will be explained more in detailed further below, but before we continue, it is important to state the following.

Honk has <u>two</u> syntaxes.

This was a challenge I imposed on myself because I thought about how it'd be funny if that the language could be a joke language. And ***Honk*** has a little extra because of that. With that said, Honk has a *regular syntax* and a *goose syntax (yes, I called it that)*. The language functionally works the same, but one is tailored to a regular developer experience, *and the other is if you hate yourself and/or want to become one with the goose*.
For the rest of the document, it'll describe both syntaxes of Honk. But I highly recommend you focusing on the regular syntax when first reading this (and you, the teacher, will probably have an easier time revising this). The diagrams and code segments on the `left` side of the page will represent the regular syntax, while the `right` side holds the *goose* syntax.
With that said, the program will have a general structure as such (**bold text** means it's required):

```
Program <program_name>;
<global variable declarations>
<function declarations>

%% Comments
main() {
  <statements>
}
```

```
Untitled <program_name> game HONK
<global variable declarations>
<function declarations>

%% Comments
Press y to honk
OPEN FANCY GATE
  <statements>
CLOSE FANCY GATE
```

The first line will identify the program's name, followed by an optional set of global variable declarations and function declarations, each with their own formats specified below. `main()` must be the last function declared and it is, fittingly, the main function of the program. Only the code ran in `main()` is executed.

Variables can be declared in a global or local scope. The following is the structure of how variables are declared for both cases:

```
var
  <type> <id><dimensions>;
  <type> <id>, <id2>, ...;
  <...more variables>
```

```
pond
  <type> <id><dimensions> HONK
  <type> <id> MOAR <id2> MOAR ... HONK
  <...more variables>
```

Each variable declaration requires a `type`, an `id` for identification, & an optional `dimensions` field. The available variable types are `int`/`WHOLE GOOSE` for integer values, `float`/`PART GOOSE` for floating point values, `char`/`LETTER GOOSE` for single characters, & `bool`/`DUCK OR GOOSE` for true/false values. There can be multiple variable declarations per type as long as they're separated by commas (or `MOAR` statements), & similarly, multiple type declarations are possible when separated by semicolons (or `HONK` statements).

Upon specifying a variable name for `id`, you can declare dimensions to make it an array or matrix. Up to 2 dimensions are allowed.

NOTE: It's worth noting that Boolean types are an extra thing I added in here.

For function declarations, they follow the following syntax:

```
function <type> <name>(<params>)
<local variable declarations>
{
  <statements>
}
```

```
task <type> <name> HONK <params> HONK
<local variable declarations>
OPEN FANCY GATE
  <statements>
CLOSE FANCY GATE
```

Functions must be unique & can also be of type `void`\`my soul` which doesn't require a value to be returned. A function can also have any number of parameters of any type, separated by commas (or `MOAR` statements) and can be of any available number of dimensions.

Also worth noting that using arrays and matrixes as return values and parameters for functions is an extra thing I added.

The following statements are within the language's capabilities (and will be explained in greater detail later on):

- **Assignment**: A variable can be assigned to an expression's value (as long as it is compatible in type and dimension). This can be done and combined using different methods:
    - **Arithmetic** (+, -, *, /, %)
    - **Comparison** (==, !=, <, <=, >, >=)
    - **Logic** (&, |)
    - **Matrix operations** (determinant - $, transpose - !, inverse - ? , dot product - .)
    - **Function calls** that yield a value
- **Return**: Returns a value from a function that's non-void
- **User input:** Reads user input and assigns to 1+ variable(s)
- **Print**: Prints 1+ variable values
- **Conditional statements**: An 'if' statement that enters a code block when a condition is met. An optional 'else' statement is also allowed for when said condition fails.
- **Loops**: A 'loop' statement that repeats a block of code until a specified condition fails. A **break** statement is also available to "break" out of a loop at any point. There are two variations:
    - **While loop**: Standard 'while' loop reminiscent to languages like C++.
    - **From loop**: Similar to a 'for' loop. A temporary iterator variable is set (or an existing one is set) and increments by 1 with each loop. Starting from a specified value and ending in another.

Some special features included in this language, besides the ability of matrix operations, is the ability to do *batch* operations with certain operations, such as summing one matrix with another, dot product, inputting an array without the need of a loop, etc. Bear in mind that most operations require that the matrix is "compatible". For example, you can only assign an array or matrix with another if they have the same dimensions & calculating the determinant of a matrix requires it to be a square. These restrictions and better explanations of these operations will be explained later below.

Use cases and test cases

The language aims to do *most* basic operations that you can normally do on a standard programming language. With that in mind, some of the use cases with ***Honk*** are reminiscent to objectives with regular languages, with a small emphasis in the usage of array/matrix operations and/or potentially doing certain programming problems in pain using the *goose* syntax.
Before we dive into the finer details of ***Honk,*** here are some test cases to demonstrate some of its functionality:

Read two numbers and print its sum

The following pieces of code will print out a message saying, “Sum 2 numbers:”, followed by 2 prompts. The sum of those two numbers will be calculated and printed on screen.

```
Program sumTwoNums;
var int a, b, result;

main() {
  print(“Sum 2 numbers:”);
  read(a);          %% 1
  read(b);          %% 2
  result = a + b;
  print(result);    %% 3
}
```

```
Untitled sumTwoNums game HONK
pond WHOLE GOOSE a MOAR b MOAR result HONK

Press y to honk
OPEN FANCY GATE
  SHOW ON TV "Sum 2 numbers:" HONK
  HO- a -ONK HONK          %% 1
  HO- b -ONK HONK          %% 2
  result AM GOOSE a MORE GOOSE b HONK
  SHOW ON TV result HONK   %% 3
CLOSE FANCY GATE
```

Loop through 0 through n

The following code will ask for a number from the user & sends it to a void function to later print out all the numbers from 0 to (including) said number. The function will simply not print anything if the user sends a 0 or a negative value.

```
Program loopThroughN;
var int a;

function void loop(int n) {
  from (i = 0 to n) do {
    print(i);
  }
}

main() {
  read(a);
  loop(a);
}
```

```
Untitled loops game HONK
pond WHOLE GOOSE a HONK

task my soul loop HONK WHOLE GOOSE n HONK
OPEN FANCY GATE
  inhales i AM 0 HOOOONK n HOONK
  OPEN FANCY GATE
    SHOW ON TV i HONK
  CLOSE FANCY GATE
CLOSE FANCY GATE

Press y to honk
OPEN FANCY GATE
  HO- a -ONK HONK
  HOOONK loop OPEN GATE a CLOSE GATE HONK
CLOSE FANCY GATE
```

Check if all numbers in a matrix are larger than the numbers on another matrix

This will prompt the user to input the numbers of 2 2x2 matrixes (so 8 values total). You’ll notice we put a comparison operator between two matrixes in an ‘if’ statement. What this will first do is check if A[0][0] > B[0][0], A[0][1] > B[0][1], and so forth. This will produce a matrix of booleans. Then the conditional will act as an AND gate, only entering the block if *all* elements are True. Inputting [[8, 7], [6, 5]], [[4, 3], [2, 1]] will result in a printed message, for example.

```
Program checkMatrix;
var int A[2][2], B[2][2];

main() {
  read(A, B);
  if (A > B) then {
    print("All of A is
bigger than B!");
  }
}
```

```
Untitled checkMatrix game HONK
pond WHOLE GOOSE A OPEN BOX 2 CLOSE BOX OPEN
BOX 2 CLOSE BOX MOAR B OPEN BOX 2 CLOSE BOX
OPEN BOX 2 CLOSE BOX HONK

Press y to honk
OPEN FANCY GATE
  HO- A MOAR B -ONK HONK
  HONK? A SUPERIOR B HONK!
  OPEN FANCY GATE
    SHOW ON TV "All of A is bigger than B!"
HONK
  CLOSE FANCY GATE
CLOSE FANCY GATE
```

Get the 3rd row's 2nd element of a transposed matrix

Let's say you have a really weird case where you have to get that value for whatever reason. A matrix operation and then accessing it is all you need.

```
Program getSpecificElement;
var int arr[4][3],
trans[3][4];

main() {
  read(arr);
  trans = arr!;
  print("Transposed
matrix:", arr);
  print(trans[2][1]);
}
```

```
Untitled getSpecificElement game HONK
pond WHOLE GOOSE arr OPEN BOX 4 CLOSE BOX OPEN
BOX 3 CLOSE BOX MOAR trans OPEN BOX 3 CLOSE
BOX OPEN BOX 4 CLOSE BOX HONK

Press y to honk
OPEN FANCY GATE
  HO- arr -ONK HONK
  trans AM arr SURPRISE HONK
  SHOW ON TV "Transposed matrix:" MOAR trans
HONK
  SHOW ON TV trans OPEN BOX 2 CLOSE BOX OPEN
BOX 1 CLOSE BOX HONK
CLOSE FANCY GATE
```

These are just a few of the test cases that you can try out with ***Honk***, but for creative experimentation and painful eyesores!

The process

I'll be honest, most of the plan was to follow along with the teacher's teachings and get a bit of help from friends to better understand how to get this done (not copying though! I was just lost in a few places and needed some pointers). Once some things clicked, advancements and regressions were made. For the most part, it was step by step catching up to the week advancements during the weekend (because we started a week late) until around the past few weeks where I could add a couple of details.

Advancement Reports

- **Advance 2** *(we missed week 1)*
  - Start using RPLY
  - Do lexical analysis
  - Start base structure of compiler
  - Do sums and substractions
  - Start function directory
- **Advance 3**
  - Switch from RPLY to PLY (there was like zero documentation for RPLY)
  - Improve lexer productions
  - Improve function directory
  - Start quad manager and semantic cube
- **Advance 4**
  - Replace use of lists in favor of dictionaries
  - Implement semantic cube
  - Add Boolean vartype
  - Improve functions and local variables
  - Implement better quads for arithmetic operations
- **Advance 5**
  - Improve expression functionality to work with groups
  - Implement constants
  - Add quads for conditionals, 'while' loop, & functions
  - Improve semantic cube so it isn't ginormous anymore
  - Improve input read through files
- **Advance 6**
  - *Teammate dropped out. Now it's a solo project!*
  - Implement quad for 'for' loop, printing, reading, ERA & GoTo
  - Implement virtual addresses
  - Complete quads for ERA and functions
  - Implement array/matrix declaration and access
  - Start with virtual machine
- **Advance 7**
  - Make constants global
  - Implement .obj file creation
  - Run PRINT, READ, ERA, PARAM, GoSub, RETURN, EndFunc, END, and more quads in virtual machine
  - Create new quads (+->, =>) to complete other quads functionalities
  - Use constant for 'for' loop

    - Adjust memory ranges
    - Fix pointers
    - Redo memory structure to work with functions
- **Final advance**
    - *Lots of fixes*
        - Fix Booleans, pointer access, & local memory in VM
        - Fix function calls with local variables
        - Fix function parameters
        - Fix matrix access
        - Fix negative indexes
    - Redo variable stack
    - Redo local memory
    - Add matrix validation
    - Disallow int = float operations
    - Run more dual-op operations
    - Implement and run matrix operation quads
    - Implement and run 'batch' quads for certain operations
    - Implement and run dot product quad
    - Refactor filenames
    - Implement command line options
    - Start documentation
- **18 hours to go!**
    - Refactor for English tokens
    - Implement BREAKs in loops
    - Fix bugs
    - Honkify

Commit history

Copy and pasted from the git log –all –decorate –one-line –graph

```
* 3af3820 (HEAD -> master, origin/master) Update README
* 9c7d53b Typo in semantic cube
* 92cf3ff Code cleanup
* 93fde69 Third pass of goose syntax
* 793d276 Second pass on honk syntax
* e47ceb3 First pass of goose syntax?!
* c2fc99d Begin honk tokens
* 62a9d6a Fix printing empty strings
* cb68ed1 Fix break
* e207638 Implement loop break
* 42c9c2a Optimize GoToF quad calls
* 15751c9 Improve functions params & allow array/matrix params
* ef6a6aa Allow functions to return arrays/matrixes
* 19b3b3d Fix function param
* 0e4a503 Add validation for return in main()
```

```
* 854e45f Alter vars declaration structure
* aa0dbe2 Alter `from` structure
* 0ca384c Allow from iterators to use existing vars
* b65c446 Refactor for English tokens
* 398756d Implement CLI arguments
* 731cddc Refactor a few filenames
* d6e71b0 Implement and run dot product quads
* ce459f3 Remove unnecesary bulk
* e06470f Improve mono-op parsing
* 5eb98c1 Implement and run batch quads!
* abe484e Implement and run matrix operation quads
* 4ad64c3 Rethink local/temp memory
* 9fc0a27 Fix more pointer-related issues
* f2787c1 Run implement operations
* f51663f Disallow int = float
* d1ddf36 Add validation for dimensions and fix matrix access
* 6074b31 Fix negative array indexes
* aceffa0 Refactor var stack
* f2932f2 Fix getting global arrays/matrixes from local function
* f33774a Fix function parameters being processed early
* 88d0556 Fix function calls with local vars
* e04139e Fix local and temporary memory structure
* 4f967ca Fix pointer access in VM
* 9bd0202 Improve debug log in VM
* cbb148f Fix boolean retrieval in VM
* 0d312f8 Configure nodemon
* 085fe33 Complete function calling!
* 46132d4 Fix bug between constants of equal value but different types
* b2b65e1 Redo virtual machine memory
* 1abd699 Implement ERA quad
* 116b8a2 Fix param validation with multiple functions
* e73fda2 Improve array debug log
* 9c53c69 Remove unnecesary var property
* e4234bf Fix variable names of longer than 1 char
* 6601b4a Fix pointers
* 8b36e54 Move 'for' quad creation methods
* 5417097 Run VERIFY quad
* 53295a3 Run READ quad
* 0c552d7 Make memory based on numbers
* ff3395e Adjust ranges
* f634960 Fix end of ranges
* d441acb Use constant for 'for' loops
* 55cf257 Fix loops
* 49a6c26 Run PRINT quad
* 77d5656 Run more dual-ops quads and GoToF quad
* bd823e9 It runs!!!
* a15c746 Ignore .txt files
* 480c488 Prepare virtual machine
* 3efed38 Add ranges to .o
* 8d0430a Implement building
* f07a4c6 Make ctes global
* 4b9ba5c Add main quads
* 763df82 Implement array access
* 733385a Reimplement arrays and vector declaration
* f08c80c Add debug logs
* 3f13604 Regress vars from arrays for now
* 390f2d7 Complete function quads
* 3f32253 Complete ERA quad
* 5a28784 Implement return quad
* cacdad6 Implement virtual addresses
* e9bfaf2 Progress for virtual addresses
```

```
* 98af858 Implement read quad
* 5dd30c1 Implement print quad
* f584fac Add string type
* 58f7942 Implement for loop
* 622bbd6 Code cleanup
* 5111e24 Slight refactor
* 5b23c2a Set temporal count for functions
* 1bf3163 Refactor
* a10f47d Implement functions
* 53f79f8 Fix expression group priority
* 8dce68c Improve semantic cube
* 45cc6e6 Implement file reading
* 54ef915 Add end quad
* c8c71dd Add quads for `while` blocks
* ae6d49b Remove unnecessary comment token
* 805d14b Add quads for if/else
* b6cda63 Improve expression
* c723286 Add constants
* 36505b5 Add dummy code for missing functions
* dca8686 Code cleanup
* c94e02d Improve quadruples
* 5098eaf Bugs squashed and refinement
* d6c0638 Implement quads
* 2245514 Implement semantic cube
* cacf0f6 Add boolean type
* bb9e442 Add functions and local vars (preliminary)
* 7508b0c replaced lists with dictionary for faster lookups
* 3aa1d7c Update gitignore
* 655ae8e Implement preliminary function directory
* 1e215a4 Refine p_variable usage
* aab62a6 Remove rply attempt
* 7a9731e Switch parser to PLY
* 5f52fc9 Switch lexer to PLY
* 7df92a4 Add README
* 5d27cd4 Initial commit
```

Now a word from our suffering student

Compilers are tiring, haha. I did learn quite a bit from this, and I don't think I'd be as *completely* lost if I were to try making another compiler for whatever reason. So much abstract thinking to get everything working, and heck, this isn't even a full-on compiler; it's still an interpreter simulating things like quadruples and memory allocation. Gives you a nice perspective on how these things work, but well, I'm just hoping this compiler will do for now.

The language

It's called Honk!

I already mentioned it a few times, but the name of this language is called ***Honk***! The best way to describe the language really boils down to "a programmer's first attempt at making a language", but ***Honk*** has some interesting things up its sleeve.

Honk can be seen as a similar language to something like C++, with rather similar structures and limitations. It aims to be able to fulfill most of the basic functions from a language like that, but also with a few extra goodies. This language has 3 rather distinct characteristics:

1. **Matrix operations**

Honk has some extra operators to work with compatible matrixes. It can find the *determinant, transpose, inverse, & dot product* of matrixes. They all have their validation checks (like how you can't calculate the inverse if the determinant is zero or using incompatible matrixes for dot products), and can work well with other arrays or matrixes, because you can do regular arithmetic with compatible arrays or matrixes, for example. Which brings us to the second point…

2. **Batch processes**

Some operations allow arrays and matrixes to be processed in *batches*. This is something reminiscent to running them through a loop, but all in one easy line of code. You can for example run `print(arr)`/`SHOW ON TV arr HONK` to print an entire array instead of processing it through a loop; or read into one instead. This allows just a bit quicker development speed.

3. **Goose syntax**

Okay this is just for fun, there's a second syntax available in Honk, known as *goose syntax*. Inspired by joke languages like LOLCODE, and *this tweet*, I set out to make an obnoxious, hard-to-read syntax. And it works, so if you want to melt your brain trying to write in this, be my guest. As mentioned before, both syntaxes will be described here, but it *is* recommended that you understand the regular syntax first (and if you're someone like, I dunno, my teacher grading this, it'll be easier on you, trust me).

Before we set off talking about the compiler, this is an important piece of info for your testing. I'll describe how to run your files later, but for now, remember that **.honk** files will run through the *goose syntax* lexer and parser, while every other file will run through the *regular syntax* counterparts.

Caveats

There *are* a few caveats in this language due to how it's currently built, so please take these into consideration as you write your code in ***Honk***:

- Variables can live in only 2 scopes: global or local. This is determined if they are in the main function or their own function. Local function scopes are still unique between functions but bear in mind that loops do not give you a higher scope. Therefore, you cannot overwrite variables with the help of a loop because this isn't a change of scope.
- Every argument in a conditional are always evaluated. This usually is no problem, but in the case of certain logical expressions, short-circuiting doesn't exist. So be aware about how you build your conditionals with this in mind.

Compilation Errors

When you attempt to compile your code, there may be errors that can be caught in the parser. These are the different kinds you can bump into:

Syntax error at x

This error simply states that there's an invalid token x at a certain line (this error is prefixed with the line number). To fix, check your code and re-write accordingly.

Variable x already exists

This states that a variable of the same name (x) already exists in the current scope. While it is possible to create a variable of the same name at a higher function scope, this does not apply for the current scope. Also bear in mind that in this language, loops do **not** equate to a higher scope, they are still subject to the global/local scope. Please look at the Caveats section for a bit more detail. Change your variable name to fix this.

Function x already exists

Similarly, as above, a function must always have a unique name, and this function warns about the violation of this rule. Fix this by changing your function's name to something else.

Zero or negative indexes are not allowed

When creating arrays or matrixes, this validation warns about the usage of size 0 or less, as this isn't allowed.

Multiple declaration of x

When creating functions with parameters, each parameter must be a unique name. Change your parameter name when this error occurs.

x has y dimension(s)!

When accessing arrays or matrixes, this error occurs when you access a "dimension" that doesn't exist. For example, if you attempt to access the first element of an atomic variable, this error will pop up. It will also show you the dimensions the variable holds for cross-checking. Avoid accessing invalid dimensions to stop this error.

Function x does not exist!

Simple, the function you tried to call doesn't exist. Make sure that the function call is written correctly to avoid this error.

This function is non-void, therefore it can't be used as an expression!

This is a straight-forward one. When you call a non-void function as a statement, this error will warn you about, as it is not allowed outside of an expression. Use the function in an expression and avoid using it outside of one to avoid this error.

This function is missing a return statement!

Also straight-forward, a non-void function must have a return statement, so make sure there is one to avoid this error.

This function is void and cannot be used as an expression!

And on the contrary, a function that is void cannot be used as part of an expression, but only as a statement, so make sure that is the case.

Wrong number of parameters in `x`!

When calling a function `x` with the wrong number of parameters, this error will warn you about it. Make sure you're using the correct number of parameters to continue working properly.

Wrong param type! `x != target_type`

When calling a function with parameters, this error warns about the usage of a parameter that isn't of the same type as the expected type.

Wrong param dimensions! `x != target_dims`

Similarly as above, this warns about the mismatch between the parameter's dimensions and the expected number of dimensions.

Type mismatch! `x op y`

This error happens when you are trying to carry out an invalid expression based on the types of variables used. When this happens, the error will show the types and operator used so that you can debug the problem.

Type mismatch! `x op`

Same as above, but for mono-operand operators.

Type mismatch! `x != bool`

Also similar as above, but when in a conditional, this error is thrown when the resulting value of a conditional is not of Boolean type. Check to make sure you're writing your conditionals correctly.

`x` must be a matrix to use the `op` operator!

If one of the matrix operations operators are used (`$`, `!`, `?`, `.`) and the operand is not a matrix, this will halt the program and warn the user about it.

`x` must be a square matrix to use the `op` operator!

For the `$` & `?` operators, a square matrix must be used to utilize the operator. Otherwise, this error is thrown. This is because it's impossible to calculate the determinant and inverse of a matrix that isn't square, respectively.

Dimension mismatch! `[x] != [y]`

Similarly, as above, this error is thrown when the dimensions of the variables involved aren't equal with each other. The error will show the operands' dimensions to allow easier debugging.

Incompatible dimensions! `[x] [y]`

This is similar to the error above but is specific to the dot product operator (`.`). Incompatible dimensions mean not equal dimensions but simply "not compatible". For a dot product of matrixes to work, *the number of columns of the first matrix must be equal to the number of rows of the second matrix*. The other dimensions do not affect compatibility, which is why this is its own error.

You can't have a return statement on main()!

Return statements are not allowed in `main()`, so avoid doing so.

There can't be return statements in non-void functions!

And also, return statements are not allowed in non-void functions, as they are specific to functions with a return value.

Returned variable doesn't match return type! -> `x = return_type`

This happens when a user codes in a return statement that uses a variable of a different type than the function's return type. Be wary of which variables you use to return from a function.

Returned variable doesn't match return dimensions! -> `[x] != [return_dims]`

Similarly, this error is thrown when the dimensions don't match with the return dimensions of a function. Same principle as above, simply check and make sure you're returning the correct variable and dimensions.

Can't use `BREAK` outside of a loop!

A `BREAK` statement can only be used when inside a loop, so this error warns about the violation of such.

Array indexes must be an atomic value!

When using variables to access arrays/matrixes, they must be an atomic value, not an array or matrix itself. Be sure to avoid doing such to not throw this error.

Out of bounds! `x in scope`

This error happens when the virtual memory for a specific type and scope has run out. Either adjust accordingly or increase the size if you know your way around the code.

Invalid vartype/scope?!

A rare error that occurs when a variable isn't able to allocate memory into the virtual directory. This is probably due to something wrong with the source code itself, sorry about that.

Execution Errors

Besides the common errors you'll find when using the language, as it is built above Python, these errors can also appear during execution:

quack has commit die

This is a general error that occurs when the virtual machine was unable to be initialized. This can likely be due to the .o file being corrupted or created incorrectly.

Var is not assigned in memory?!

This error occurs when a variable hasn't been assigned a value in memory, so the likely cause is that you haven't assigned a value to it upon reaching this part of the code.

Accessing prohibited memory! *or* Getting from out of bounds! *or* Setting in invalid memory!

This happens when the VM attempts to get or set data in memory that's invalid to access and write to. This is likely something wrong with the VM's code.

How did we get here??

Related to the error above, this is actually supposed to be an *impossible* error to reach & is only there for code-reading purposes. If you actually see this error, I'm genuinely curious on how that happened.

This matrix can't be inverted!

This error happens when you attempt to invert a matrix whose determinant is equal to 0, which makes it impossible to invert.

! Invalid type! Try again…

This common error happens during user input. When the input doesn't match a given type, it warns the user and prompts to try again.

`x` is out of bounds of range `0-limit`

This error occurs when attempting to access out of an array's range. Make sure the index is valid.

The Compiler

The nitty-gritty now!

Here we go! Where all the *dark* magic happens! The next few sections will explain more in depth about how the compiler works and functions. To preface, ***Honk*** is a language that runs on top of Python and simulates a compiler by using lower-level instructions (called quadruples) & virtual memory allocation. While this has been tested on a MacOS system utilizing a UNIX terminal, this should theoretically work just fine on any system that runs *Python 3.6 or above*. The following modules have been using to carry out the entire compilation process:

- `sys` (for argument passing through command line)
- `os` (for file path operations)
- `argparse` (for a neat CLI interface)
- `ply` (Lex and Yacc wrappers for Python)
- `collections` (primarily use for its dictionaries and stacks)

Lexical Analysis

As previously described, there are *2 syntaxes for **Honk***, so this document will do its best to describe them both. While they are functionally the same, they have wildly different lexical structures (but somewhat similar syntax structure). In an attempt to avoid less confusion, this document will pair together related tokens between syntaxes; `left` being the regular syntax (this lexer is run from the `lexer.py` file), & `right` being *goose syntax* (`lexhonker.py`). The following lists shows pairs where it means **regex used : token name**.
(also yes, I'm aware how weird the *goose syntax* is going to look, it's a joke syntax)

Regular syntax	Goose syntax
`Program : PROGRAM`	`Untitled : UNTITLED`
	`game : GAME`
`main : MAIN`	`Press : PRESS`
	`y : Y`
	`to : TO`
	`honk : HONK_LOWERCASE`
`var : VAR`	`pond : POND`
`int : INT`	`WHOLE : WHOLE`
`float : FLOAT`	`PART : PART`
`char : CHAR`	`LETTER : LETTER`
`bool : BOOL`	`DUCK : DUCK`
	`OR : OR`
`function : FUNCTION`	`task : TASK`
`void : VOID`	`my : MY`
	`soul : SOUL`
`return : RETURN`	`GOT : GOT`

```
read : READ


print : PRINT


if : IF
then : THEN
else : ELSE
while : WHILE
do : DO
from : FROM
to : TO
break : BREAK




[a-zA-Z_][a-zA-Z0-9_]* : ID
\-?\d+\.\d+ : CTE_FLOAT
\-?\d+ : CTE_INT
\'.\' : CTE_CHAR
(True|False) : CTE_BOOL
\".*\" : STRING
= : =
== : IS_EQUAL
!= : IS_NOT_EQUAL
< : <
<= : LESS_THAN_OR_EQUAL
> : >
>= : MORE_THAN_OR_EQUAL
\+ : +
\- : -
\* : *
\/ : /
\% : %
\$ : $
\! : !
\? : ?
\& : &
```

```
BELL : BELL
HO : HO
- : -
ONK : ONK
SHOW : SHOW
ON : ON
TV : TV
\? : ?
\! : !
BONK : BONK


inhales : INHALES

peace : PEACE
was : WAS
never : NEVER
an : AN
option : OPTION
[a-zA-Z_][a-zA-Z0-9_]* : ID
\-?\d+\.\d+ : CTE_FLOAT
\-?\d+ : CTE_INT
\'.\' : CTE_CHAR
(Goose|Duck) : CTE_BOOL
\".*\" : STRING
AM : AM
GOOSE : GOOSE
NOT : NOT
INFERIOR : INFERIOR
maybe : MAYBE
SUPERIOR : SUPERIOR

MORE : MORE
LESS : LESS
GOOSETIPLY : GOOSETIPLY
GOOSIVIDE : GOOSIVIDE
LEFTOVERS : LEFTOVERS
GOOSECOIN : GOOSECOIN
SURPRISE : SURPRISE
wh : WH
TOGETHER : TOGETHER
```

```
\| : |
\( : (
\) : )
\[ : [
\] : ]
\{ : {
\} : }
\. : .
\, : ,
\: : :
\; : ;
```

```
FOREVER : FOREVER
POLE : POLE
GATE : GATE

BOX : BOX

FANCY : FANCY

doot : DOOT
MOAR : MOAR

HONK : HONK
HOONK : HOONK
HOOONK : HOOONK
HOOOONK : HOOOONK
OPEN : OPEN
CLOSE : CLOSE
```

It also bears mentioning that whitespace and tabs are ignored, including anything after **%%** as there are registered as comments. Also, the **CTE** and **ID** tokens have the following functions associated with them:

```
def t_CTE_FLOAT(t):
  r'\-?\d+\.\d+'
  t.value = float(t.value)
  return t

def t_CTE_INT(t):
  r'\-?\d+'
  t.value = int(t.value)
  return t

def t_CTE_CHAR(t):
  r'\'.\''
  t.value = t.value[1]
  return t

def t_CTE_BOOL(t):
  r'(true|false)'
  t.value = (t.value == "true")
  return t

def t_ID(t):
  r'[a-zA-Z_][a-zA-Z0-9_]*'
  t.type = reserved.get(t.value, 'ID')
  return t
```

```
def t_CTE_FLOAT(t):
  r'\-?\d+\.\d+'
  t.value = float(t.value)
  return t

def t_CTE_INT(t):
  r'\-?\d+'
  t.value = int(t.value)
  return t

def t_CTE_CHAR(t):
  r'\'.\''
  t.value = t.value[1]
  return t

def t_CTE_BOOL(t):
  r'(Goose|Duck)'
  t.value = (t.value == "true")
  return t

def t_ID(t):
  r'[a-zA-Z_][a-zA-Z0-9_]*'
  t.type = reserved.get(t.value, 'ID')
  return t
```

Syntax Analysis

The syntax structures between the two syntaxes are actually somewhat similar, but it can be hard to tell because of the vastly different tokens used. For simplicity's sake, we'll first show the regular syntax's formal grammar, followed by the *goose syntax*'s counterpart.

REGULAR SYNTAX

PROGRAM -> Program *id* ; VARS FUNCTIONS *main* ( ) BODY
VARS -> VAR_DECLARE | ε
VAR_DECLARE -> TYPE VAR_NAME ; VAR_DECLARE'
VAR_DECLARE' -> VAR_DECLARE | ε
VAR_NAME -> *id* VAR_DIMS VAR_NAME'
VAR_NAME' -> , VAR_NAME | ε
VAR_DIMS -> DIM DIM | DIM | ε
DIM -> *[cte_int]*
TYPE -> *int* | *float* | *char* | *bool*
FUNCTIONS -> *function* FUNC_TYPE *id* FUNC_DIMS *(* FUNC_PARAMS *)* VARS BODY
  FUNCTIONS' | ε
FUNCTIONS' -> FUNCTIONS | ε
FUNC_TYPE -> TYPE | *void*
FUNC_DIMS -> DIM DIM | DIM | ε
FUNC_PARAMS -> TYPE *id* PARAM_DIMS FUNC_PARAMS' | ε

```
FUNC_PARAMS' -> , FUNC_PARAMS | ε
PARAM_DIMS -> DIM DIM | DIM | ε
BODY -> { STATEMENTS }
STATEMENTS -> STMT STMT' | ε
STMT -> ASSIGN | CALL_FUNC | RETURN | PRINT | IF | FROM | WHILE | BREAK
STMT' -> STMT | ε
ASSIGN -> EXPR_VAR = EXPR ;
RETURN -> return ( EXPR ) ;
READ -> read ( READ_PARAMS ) ;
READ_PARAMS -> EXPR_VAR READ_PARAMS'
READ_PARAMS' -> , READ_PARAMS | ε
PRINT -> print ( PRINT_PARAMS ) ;
PRINT_PARAMS -> EXPR PRINT_PARAMS' | STRING PRINT_PARAMS'
PRINT_PARAMS' -> , PRINT_PARAMS | ε
IF -> if ( EXPR ) then BODY ELSE
ELSE -> else BODY | ε
FROM -> from ( id = EXPR to EXPR ) do BODY
WHILE -> while ( EXPR ) do BODY
BREAK -> break ;
EXPR -> EXPR_CMP EXPR'
EXPR' -> & EXPR | '|' EXPR | ε
EXPR_CMP -> EXPR_ARITH EXPR_CMP'
EXPR_CMP' -> == EXPR_CMP | != EXPR_CMP | < EXPR_CMP | <= EXPR_CMP | >
  EXPR_CMP | >= EXPR_CMP | ε
EXPR_ARITH -> EXPR_FACT EXPR_ARITH'
EXPR_ARITH' -> + EXPR_ARITH | - EXPR_ARITH | ε
EXPR_FACT -> EXPR_MONO EXPR_FACT'
EXPR_FACT' -> * EXPR_FACT | / EXPR_FACT | % EXPR_FACT | . EXPR_FACT | ε
EXPR_MONO -> EXPR_ATOM EXPR_MONO_OP
EXPR_MONO_OP -> ! EXPR_MONO_OP' | ? EXPR_MONO_OP' | $ EXPR_MONO_OP'
EXPR_MONO_OP' -> EXPR_MONO_OP | ε
EXPR_ATOM -> EXPR_GROUP | EXPR_VAR | EXPR_CALL_FUNC
EXPR_GROUP -> ( EXPR )
EXPR_VAR -> id EXPR_VAR_DIMS | CTE
EXPR_VAR_DIMS ->  EXPR_VAR_DIM EXPR_VAR_DIM | EXPR_VAR_DIM | ε
EXPR_VAR_DIM -> [ cte_int ]
CTE -> cte_int | cte_float | cte_char | cte_bool
EXPR_CALL_FUNC -> id ( CALL_FUNC_PARAMS )
CALL_FUNC -> id ( CALL_FUNC_PARAMS ) ;
CALL_FUNC_PARAMS -> EXPR CALL_FUNC_PARAMS' | ε
CALL_FUNC_PARAMS' -> , CALL_FUNC_PARAMS | ε
```

GOOSE SYNTAX

PROGRAM -> UNTITLED *id* GAME *HONK* VARS FUNCTIONS MAIN BODY

VARS -> *pond* VAR_DECLARE | ε
VAR_DECLARE -> TYPE VAR_NAME *HONK* VAR_DECLARE'
VAR_DECLARE' -> VAR_DECLARE | ε
VAR_NAME -> *id* VAR_DIMS VAR_NAME'
VAR_NAME' -> *MOAR* VAR_NAME | ε
VAR_DIMS -> DIM DIM | DIM | ε
DIM -> *OPEN BOX cte_int CLOSE BOX*
TYPE -> *WHOLE GOOSE* | *PART GOOSE* | *LETTER GOOSE* | *DUCK OR GOOSE*
FUNCTIONS -> *task* FUNC_TYPE *id* FUNC_DIMS *HONK* FUNC_PARAMS *HONK* VARS BODY FUNCTIONS' | ε
FUNCTIONS' -> FUNCTIONS | ε
FUNC_TYPE -> TYPE | *my soul*
FUNC_DIMS -> DIM DIM | DIM | ε
FUNC_PARAMS -> TYPE *id* PARAM_DIMS FUNC_PARAMS' | ε
FUNC_PARAMS' -> *MOAR* FUNC_PARAMS | ε
PARAM_DIMS -> DIM DIM | DIM | ε
MAIN -> *Press y to honk*
BODY -> *OPEN FANCY GATE* STATEMENTS *CLOSE FANCY GATE*
STATEMENTS -> STMT STMT' | ε
STMT -> ASSIGN | CALL_FUNC | RETURN | PRINT | IF | FROM | WHILE | BREAK
STMT' -> STMT | ε
ASSIGN -> EXPR_VAR *AM* EXPR *HONK*
RETURN -> *GOT BELL* EXPR *HONK*
READ -> *HO -* READ_PARAMS *– ONK HONK*
READ_PARAMS -> EXPR_VAR READ_PARAMS'
READ_PARAMS' -> *MOAR* READ_PARAMS | ε
PRINT -> *SHOW ON TV* PRINT_PARAMS *HONK*
PRINT_PARAMS -> EXPR PRINT_PARAMS' | STRING PRINT_PARAMS'
PRINT_PARAMS' -> *MOAR* PRINT_PARAMS | ε
IF -> *HONK ?* EXPR *HONK !* BODY ELSE
ELSE -> *BONK* BODY | ε
FROM -> *inhales id AM* EXPR *HOOOONK* EXPR *HOONK* BODY
WHILE -> *HONK HONK* EXPR *HOONK* BODY
BREAK -> *peace was never an option HONK*
EXPR -> EXPR_CMP EXPR'
EXPR' -> *TOGETHER FOREVER* EXPR | *POLE* EXPR | ε
EXPR_CMP -> EXPR_ARITH EXPR_CMP'
EXPR_CMP' -> *AM GOOSE ?* EXPR_CMP | *NOT GOOSE ? !* EXPR_CMP | *INFERIOR* EXPR_CMP | *INFERIOR maybe* EXPR_CMP | *SUPERIOR* EXPR_CMP | *SUPERIOR maybe* EXPR_CMP | ε
EXPR_ARITH -> EXPR_FACT EXPR_ARITH'
EXPR_ARITH' -> *MORE GOOSE* EXPR_ARITH | *LESS GOOSE* EXPR_ARITH | ε
EXPR_FACT -> EXPR_MONO EXPR_FACT'

```
EXPR_FACT' -> GOOSETIPLY EXPR_FACT | GOOSIVIDE EXPR_FACT | LEFTOVERS
EXPR_FACT | doot EXPR_FACT | ε
EXPR_MONO -> EXPR_ATOM EXPR_MONO_OP
EXPR_MONO_OP -> SURPRISE EXPR_MONO_OP' | wh EXPR_MONO_OP' | GOOSECOIN
  EXPR_MONO_OP'
EXPR_MONO_OP' -> EXPR_MONO_OP | ε
EXPR_ATOM -> EXPR_GROUP | EXPR_VAR | EXPR_CALL_FUNC
EXPR_GROUP -> OPEN GATE EXPR CLOSE GATE
EXPR_VAR -> id EXPR_VAR_DIMS | CTE
EXPR_VAR_DIMS -> EXPR_VAR_DIM EXPR_VAR_DIM | EXPR_VAR_DIM | ε
EXPR_VAR_DIM -> OPEN BOX cte_int CLOSE BOX
CTE -> cte_int | cte_float | cte_char | cte_bool
EXPR_CALL_FUNC -> HOOONK id OPEN GATE CALL_FUNC_PARAMS CLOSE GATE
CALL_FUNC -> HOOONK id OPEN GATE CALL_FUNC_PARAMS CLOSE GATE HONK
CALL_FUNC_PARAMS -> EXPR CALL_FUNC_PARAMS' | ε
CALL_FUNC_PARAMS' -> MOAR CALL_FUNC_PARAMS | ε
```

Some key points to notice between them is how a few productions discarded their "counterpart" token. This is for the sake of not making the code too obnoxiously long, as it's hard to develop and test as I went on.

Semantic Analysis and Code Generation

Now the meat of it! For brevity's sake, we aren't going to go through every single detail, so we'll go over the most important parts of how semantics are validated and how code is generated. The good thing about this part is beside structure changes in syntax, both syntaxes share the majority of the logic involve (keyword: majority, there are still a few differences because of how they're structured).
The way it generally functions is utilizing the parser (`parser.py` for the regular syntax & `parshonker.py` for *goose syntax*) in conjunction with helper classes:

- `functionDirectory.py`: Keeps a record of functions, variable tables, constant tables, & other helper objects for keeping track of them all.
- `quadManager.py`: In charge of keeping track of quads and certain stacks to aid quadruple creation. Also keeps a reference to the function directory for quicker access.
- `virtualDirectory.py`: Manages the virtual memory addresses of variables, temporary values, and constants. Also defines the "virtual memory" bounds for the program & allows saving the *ERA* size of a function as it's processed.
- `semanticCube.py`: Holds the result types of dual-operand and mono-operand operations (or lack thereof). Very useful for operations.

For now, we will focus on `parser.py` to look at certain functionality, but most of its contents are the same as the *goose* counterpart.

Let's first see how variable declaration works:

Variable declaration

```
# parser.py
def p_variable_declare(p):
  "variable_declare : ID"
  var = p[1]
  if funcDir.varAvailable(var):
    funcDir.addVar(var,
quads.vDir.generateVirtualAddress(funcDir.currentFunc, funcDir.currentType))
    funcDir.setVarHelper(var)
  else:
    s_error(f'Variable "{var}" already exists!"')
```

The `variable_declare` token is called during global and local variable declarations, before the function or `main()` is parsed. Upon this function's call, it grabs the name within the `ID` token and checks if it already exists in the scope. `funcDir.varAvailable()` will only check the current scope to see if the variable is available to declare, and errors out if needed.

The next step is setting up a virtual address for a new variable, which we use `quads.vDir.generateVirtualAddress()` for. The following statements shown below are rather long, but this is the gist of it:

- In the virtual directory, we store the memory ranges for global, local, temporary, and constant values. Each of those ranges have their own *subranges* that hold a type. This is defined by a 5-element tuple where each pair represents their space (0-1 is for ints, 1-2 for floats, etc.)
- We also keep a counter of how many times a variable is used (scope-wise and type-wise) to later calculate a function's ERA size.
- When we call `generateVirtualAddress()`, we specify a scope and vartype to assign this variable to (which we have stored in our function directory). Using this data, we check if there is available space and "allocate" it. We return the variable's *virtual address* and make the necessary changes to our virtual address

```
# virtualDirectory.py
def generateVirtualAddress(self, scope, vartype):
  return self.setSpace(scope, vartype, 1)
```

```
# Set space and counters based on a variable's needs
  def setSpace(self, scope, vartype, space):
    # Select variable type
    v = None
    if vartype == 'int':
      v = 0
    elif vartype == 'float':
      v = 1
    elif vartype == 'char':
      v = 2
    elif vartype == 'bool':
      v = 3

    # Update total counter
    self.totalCounter += space

    # Select scope (with validation checks)
    if scope == 'main':   # Global
      if self.globalRanges[v] + self.globalCounter[v] + space >= self.globalRanges[v
+ 1] - 1:
        raise Exception(f"Out of bounds! {vartype} in {scope}")
      self.globalCounter[v] += space
      return self.globalRanges[v] + self.globalCounter[v] - 1
    elif scope == 'temp':   # Temp
      if self.tempRanges[v] + self.tempCounter[v] + space >= self.tempRanges[v + 1] -
1:
        raise Exception(f"Out of bounds! {vartype} in {scope}")
      self.tempCounter[v] += space
      return self.tempRanges[v] + self.tempCounter[v] - 1
    elif scope == 'cte':    # Constants
      if self.cteRanges[v] + self.cteCounter[v] + space >= self.cteRanges[v + 1] - 1:
        raise Exception(f"Out of bounds! {vartype} in {scope}")
      self.cteCounter[v] += space
      return self.cteRanges[v] + self.cteCounter[v] - 1
    else:                   # Local (any local function)
      if self.localRanges[v] + self.localCounter[v] + space >= self.localRanges[v +
1] - 1:
        raise Exception(f"Out of bounds! {vartype} in {scope}")
      self.localCounter[v] += space
      return self.localRanges[v] + self.localCounter[v] - 1

    raise Exception(f'Invalid vartype/scope?! -> {vartype}, {scope}')
```

Now back to where we started, we see we use `funcDir.addVar()` after getting our virtual address. This function is simply adding our variable into the corresponding var table in our function directory for later use. This is really useful as we need the virtual address later, since we want to do stuff with our variable.

```
# parser.py
def p_variable_declare(p):
  "variable_declare : ID"
  var = p[1]
  if funcDir.varAvailable(var):
    funcDir.addVar(var,
quads.vDir.generateVirtualAddress(funcDir.currentFunc, funcDir.currentType))
    funcDir.setVarHelper(var)
  else:
    s_error(f'Variable "{var}" already exists!"')
```

Function declaration

The full declaration of a function is as follows:

```
# parser.py
def p_functions(p):
  """functions : FUNCTION func_type ID found_func_name func_dims '('
func_params ')' vars found_func_start body found_func_end functions
               | empty"""
  pass
```

Function declaration works a bit similarly where upon finding its name, we check if it already exists, error out if it does, and initialize it. The function will be registered into the function directory's `directory` dictionary, which is amazingly fast and useful.

```
def p_found_func_name(p):
  "found_func_name : empty"
  func = p[-1]
  if funcDir.functionExists(func):
    s_error(f'Function "{func}" already exists!"')
  else:
    funcDir.addFunction(func)
    funcDir.createVarTable()
```

Though we have to do extra stuff besides that. For example, ***Honk*** supports functions that can return arrays or matrixes, so we also have to store the return *dimensions* along with its type. We use `setReturnDims()` to set the `returnDims` property of our function after declaring it:

```
def p_func_no_dims(p):
  "func_dims : empty"
  pass

def p_func_dims(p):
  """func_dims : dim dim
               | dim"""
  if len(p) == 3:
    funcDir.setReturnDims([p[1], p[2]])
  else:
    funcDir.setReturnDims([p[1]])
```

And besides that, we have to check and register any parameters the function may have, and also check if *those* parameters are arrays/matrixes (since ***Honk*** also supports array/matrix parameters). But I'm getting ahead of myself here. We do a similar process, see if it's available and add it into our function directory accordingly:

```
def p_func_params(p):
  """func_params : func_param
                 | empty"""
  pass

def p_func_param(p):
  """func_param : type ID found_func_param param_dims ',' func_param
                | type ID found_func_param param_dims"""
  pass

def p_found_func_param(p):
  "found_func_param : empty"
  param = p[-1]
  if funcDir.varAvailable(param):
    funcDir.setVarHelper(param)
    vAddr = quads.vDir.generateVirtualAddress(funcDir.currentFunc,
funcDir.currentType)
    funcDir.addVar(param, vAddr)
    funcDir.addFuncParam(vAddr)
  else:
    s_error(f'Multiple declaration of "{param}"!')
```

And finally, we want to mark the start and end point of our function. So before parsing the body, we set its `quadStart` property to the current quad count; & once we finish parsing our function, we add an `EndFunc` quad to mark it and delete its var table since we don't need it anymore.

```
def p_found_func_start(p):
  "found_func_start : empty"
  funcDir.setQuadStart(quads.getQuadCount())

def p_found_func_end(p):
  "found_func_end : empty"
  quads.addEndFuncQuad()
  funcDir.deleteVarTable()
```

Operations

We can't do stuff without these! One thing to remember is that *every* time an expression is parsed, it's pushed into a stack called `sVars` in our quadruple manager. Once we have to do something like a multiplication with it, we'll have it on the ready.

```
# parser.py
def p_expr_var_name(p):
  "expr_var_name : ID"
  var = funcDir.getVar(p[1])
  funcDir.setVarHelper(var.name)
  quads.pushVar(var)
```

Now say it's time to do a multiplication. The following command when it finds the operator is what triggers a check (which we'll suppose is allowed), and then we proceed to enter our `addDualOpQuad()`.

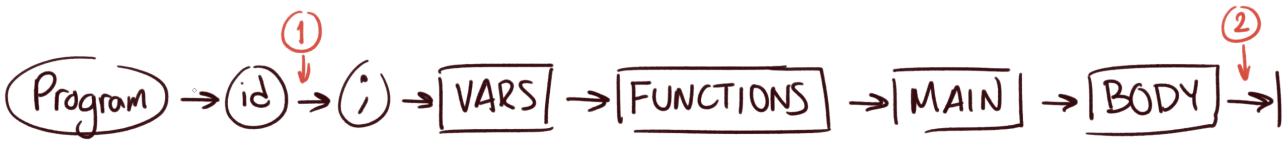
```
def p_expr_factor(p):
  "expr_factor : expr_mono found_expr_factor
expr_factor2"
  pass

def p_found_expr_factor(p):
  "found_expr_factor : empty"
  quads.addDualOpQuad(['*', '/', '%', '.'])
```

When we enter this function, we make various checks to see if the quad is allowed. First, we check if the operator is allowed to proceed (as there is a stack that accumulates operators), then we check if we can multiply the two types into a result type, using `semanticCube.py`, as well as if they have compatible dimensions. We also check if we can do a *batch operation* if the result is an array or matrix, using a `MAT` quad. This makes it easy to do tasks in bulk in the virtual machine when needed. Finally, we create our temporary variable, assign it to our virtual directory, create our new multiplication quad and push it as our new result to our `sVars` stack.

```
  # quadManager.py
  def addDualOpQuad(self, ops):
    # If operators stack is empty, stop here
    if not self.sOperators or self.sOperators[-1] not in ops:
      return

    # Special case for dot product operator ('.')
    if self.sOperators[-1] == '.':
      self.addDotProdQuad()
      return

    right = self.sVars.pop()
    left = self.sVars.pop()
    operator = self.sOperators.pop()

    result_type = getDuoResultType(left.vartype, right.vartype, operator)
    if not result_type:
      raise Exception(f'Type mismatch! {left.vartype} {operator}
{right.vartype}')

    if not self.areCompatibleDims(left.dims, right.dims):
      raise Exception(f'Dimension mismatch! {left.dims} != {right.dims}')

    if left.dims:
      self.addMatQuad(left.dims)

    result = self.vDir.generateVirtualAddress('temp', result_type)
    self.addQuad((operator, left.vAddr, right.vAddr, result))
    self.pushTemp(result, result_type, left.dims)

    if self.debug:
      print(f'\t\t\t\t\t> TMP: t{self.tempCount} - {result_type} -> {result}')

    self.tempCount += 1
```

There's a number of operations going on behind the scenes, but this is a general view of it. Now what follows are the syntax diagrams with their semantic actions. For brevity's sake, since most of the functionality is shared between them, only the regular syntax diagrams will be shown (not to mention the *goose* diagrams are incredibly confusing at a glance, and it *is* an extra thing I'm doing).

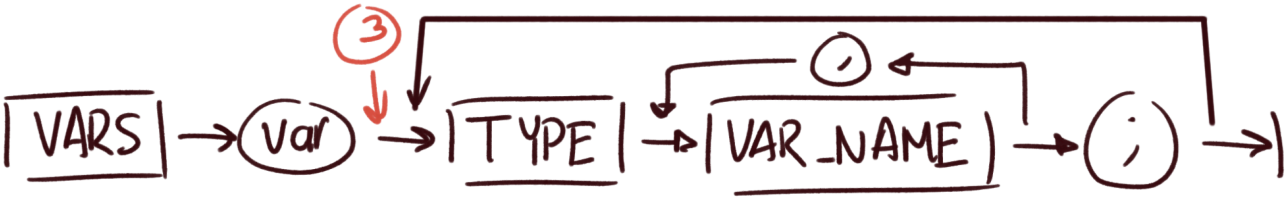
Syntax diagrams



1. Find program name: Registers global function and prepares `GOTO` quad towards `main()`
2. Complete program: Adds `END` quad and prepares to build .o file

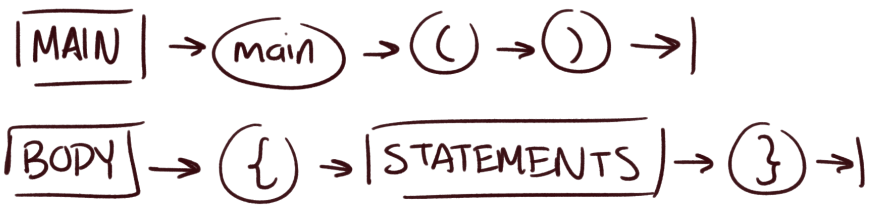
```
# 1. Find program name
def p_found_program_name(p):
  'found_program_name : empty'
  if quads.debug:

print("\n|==|==|==|==|==|==|==|==|
START DEBUG LOG
|==|==|==|==|==|==|==|==|\n")

  funcDir.addFunction('main')
  funcDir.setGlobalFunction()
  quads.addMainQuad()
```

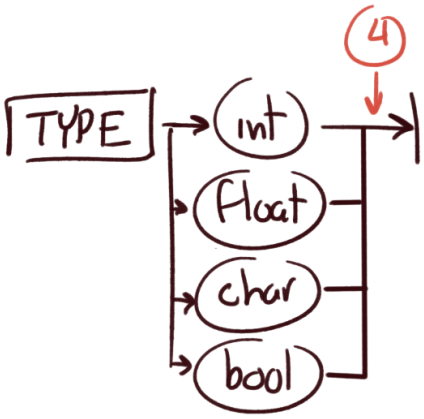
```
# 2. Complete program
def p_program(p):
  "program : PROGRAM ID
found_program_name ';' vars
functions main body"
  # Finish parsing
  quads.addEndQuad()
  p[0] = quads

  # Build .o
  quads.build()
```



3. Create var table: Simple create a var table when finding a variable

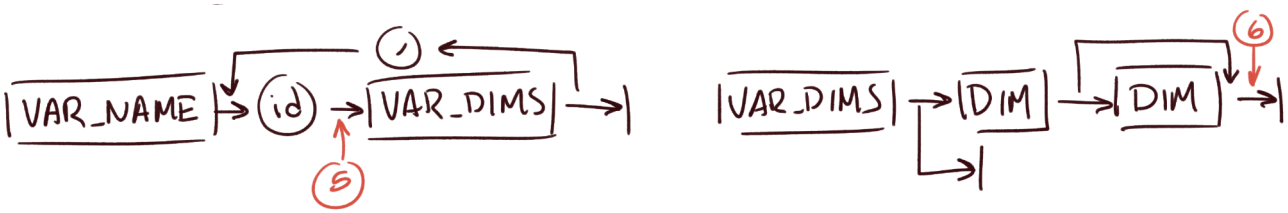
```
# 3. Create var table
def p_found_var(p):
  "found_var : empty"
  funcDir.createVarTable()
```





4. Set current vartype

```
# 4. Set current vartype
def p_type(p):
  """type : INT
          | FLOAT
          | CHAR
          | BOOL"""
  funcDir.setCurrentType(p[1])
  p[0] = p[1]
```



5. Declare variable: Checks if the variable name is available and create it
6. Set variable dimensions: Set dimensions to a given variable and update its virtual address and space

```
# 5. Declare variable
def p_variable_declare(p):
  "variable_declare : ID"
  var = p[1]
  if funcDir.varAvailable(var):
    funcDir.addVar(var,
quads.vDir.generateVirtualAddress(fu
ncDir.currentFunc,
funcDir.currentType))
    funcDir.setVarHelper(var)
  else:
    s_error(f'Variable "{var}"
already exists!"')
```

```
# 6. Set variable dimensions
def p_var_dims(p):
  """var_dims : dim dim
              | dim"""
  if len(p) == 3:
    funcDir.setVarDims([p[1], p[2]])
    p[0] = p[1] * p[2]
  else:
    funcDir.setVarDims([p[1]])
    p[0] = p[1]


quads.vDir.makeSpaceForArray(funcDir.cu
rrentFunc, funcDir.currentType, p[0])
```

7. Check dimension: Checks if a dimension index is valid for creation

```
# 7. Check dimension
def p_dim(p):
  "dim : '[' CTE_INT ']'"
  if p[2] <= 0:
    s_error(f'Zero or negative indexes
are not allowed! [{p[0]}]')
  p[0] = p[2]
```

8. Make function void: If ‘void’ set the function to void
9. Create function: Checks if function is available and creates it in the var table
10. Set function start: Sets the starting quad for function
11. Set function end: Creates `EndFunc` quad and deletes var table

```
# 8. Make function void
def p_func_type(p):
  """func_type : type
               | VOID"""
  if p[1] == 'void':
    funcDir.setCurrentType('void')
```

```
# 9. Create function
def p_found_func_name(p):
  "found_func_name : empty"
  func = p[-1]
  if funcDir.functionExists(func):
    s_error(f'Function "{func}"
already exists!"')
  else:
    funcDir.addFunction(func)
    funcDir.createVarTable()
```

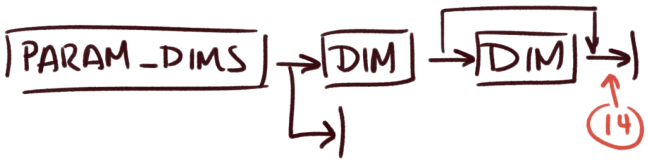
```
# 10. Set function start
def p_found_func_start(p):
  "found_func_start : empty"

funcDir.setQuadStart(quads.getQuadCou
nt())
```

```
# 11. Set function end
def p_found_func_end(p):
  "found_func_end : empty"
  quads.addEndFuncQuad()
  funcDir.deleteVarTable()
```

12. Set function return dimensions: Set the required dimensions for the function's return value
13. Add function param: Validate parameter and add it to the function's parameter and variable table

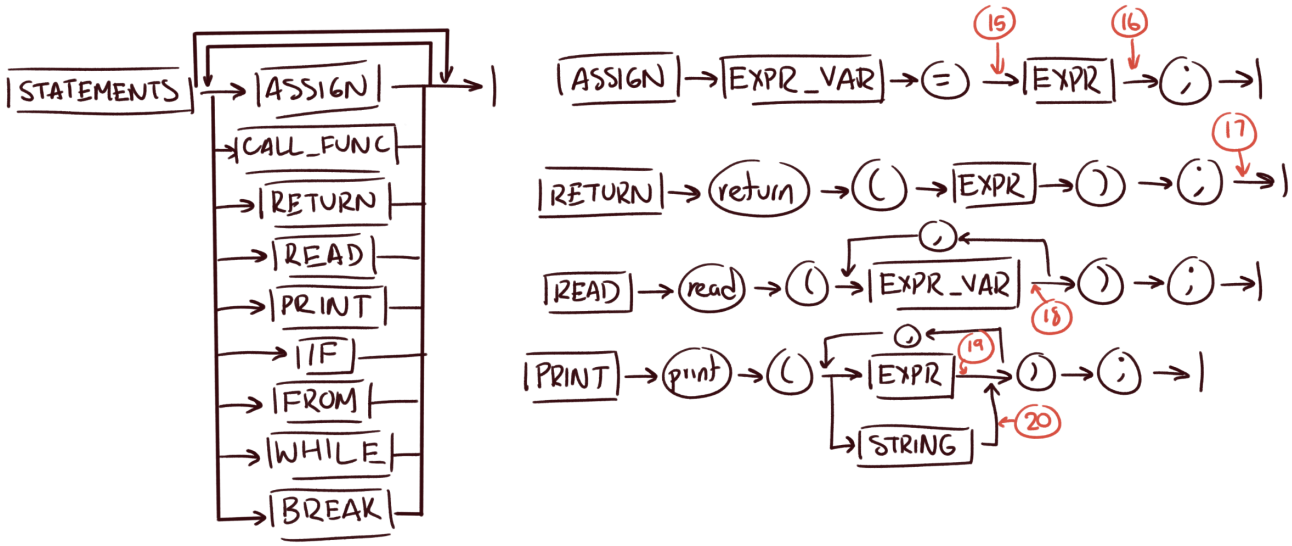
```
# 12. Set function return dimensions
def p_func_dims(p):
  """func_dims : dim dim
               | dim"""
  if len(p) == 3:
    funcDir.setReturnDims([p[1],
p[2]])
  else:
    funcDir.setReturnDims([p[1]])
```

```
# 13. Add function param
def p_found_func_param(p):
  "found_func_param : empty"
  param = p[-1]
  if funcDir.varAvailable(param):
    funcDir.setVarHelper(param)
    vAddr =
quads.vDir.generateVirtualAddress(fu
ncDir.currentFunc,
funcDir.currentType)
    funcDir.addVar(param, vAddr)
    funcDir.addFuncParam(vAddr)
  else:
    s_error(f'Multiple declaration
of "{param}"!')
```



14. Set dimensions to parameter: Sets a parameter's required dimensions to a specified set of values and make space for it

```
# 14. Set dimensions to parameter
def p_param_dims(p):
  """param_dims : dim dim
                | dim"""
  space = None
  if len(p) == 3:
    p[0] = [p[1], p[2]]
    space = p[1] * p[2]
  else:
    p[0] = [p[1]]
    space = p[1]

  funcDir.addFuncParamDims(p[0])
  funcDir.setVarDims(p[0])
  quads.vDir.makeSpaceForArray(funcDir.currentFunc,
funcDir.currentType, space)
```



15. Push operator to `sOperators` stack
16. Add assignment quadruple
17. Add return quadruple
18. Found `read()` parameter and add read quadruple

19. Add print quad using expression
20. Add print quad using a string

```
# 15. Push operator
def p_found_expr_duo_op(p):
  "found_expr_duo_op : empty"
  quads.pushOperator(p[-1])
```

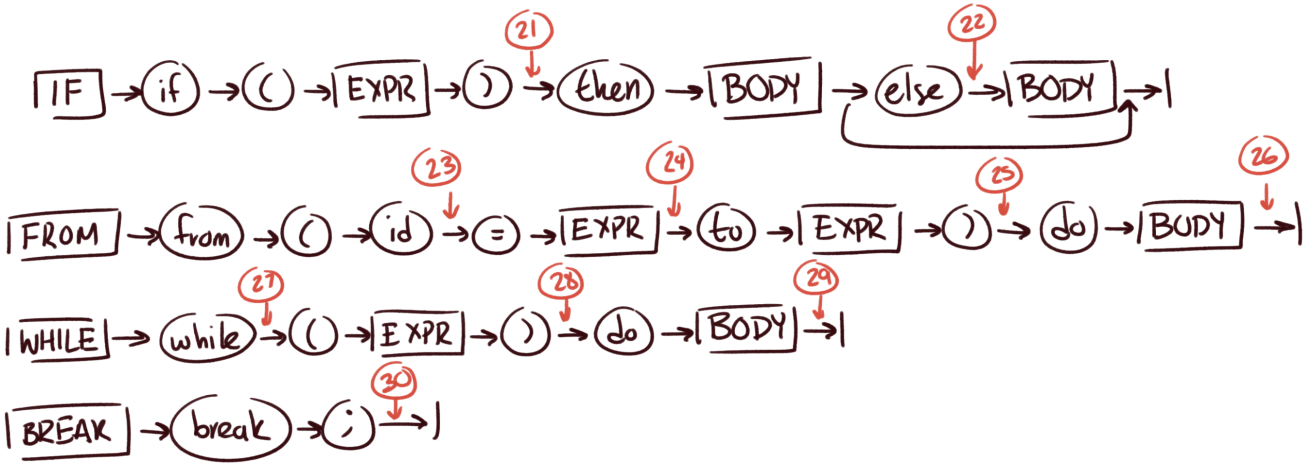
```
# 16. Add assignment quad
def p_found_assignment_end(p):
  "found_assignment_end : empty"
  quads.addAssignQuad()
```

```
# 17. Add return quad
def p_return(p):
  "return : RETURN '(' expr ')' ';'"
  quads.addReturnQuad()
```

```
# 18. Add read quad
def p_found_read_param(p):
  "found_read_param : empty"
  quads.addReadQuad()
```

```
# 19. Add print quad with expression
def p_print_param(p):
  "print_param : expr"
  quads.addPrintQuad(False)
```

```
# 20. Add print quad with string
def p_print_string(p):
  "print_param : STRING"
  quads.addPrintQuad(p[1])
```



21. Add `GoToF` quad for if statement
22. Add else-related quadruple
23. Add quads for ‘from iterator
24. Add quads related to the start of a ‘from’ loop
25. Add quads related to the conditional statement of a ‘from’ loop
26. Complete quad for end of loop
27. Prepare quads for ‘while’ loop
28. Add conditional quad for while loop expression

29. Complete quad for end of loop
30. Add `BREAK` quad

```
# 21. Add GoToF quad for if
def p_found_if_expr(p):
  "found_if_expr : empty"
  quads.addGoToFQuad()
```

```
# 22. Add else quad
def p_found_else(p):
  "found_else : empty"
  quads.addElseQuad()
```

```
# 23. Add quads for ‘from’ iterator
def p_found_from_iterator(p):
  "found_from_iterator : empty"
  quads.addFromIteratorQuads(p[-1])
```

```
# 24. Add quads related to the start
of a ‘from’ loop
def p_found_from_start(p):
  "found_from_start : empty"
  quads.addFromStartQuad()
```

```
# 25: Add quads related to the ‘from’
conditional statement
def p_found_from_cond(p):
  "found_from_cond : empty"
  quads.addFromCondQuads()
```

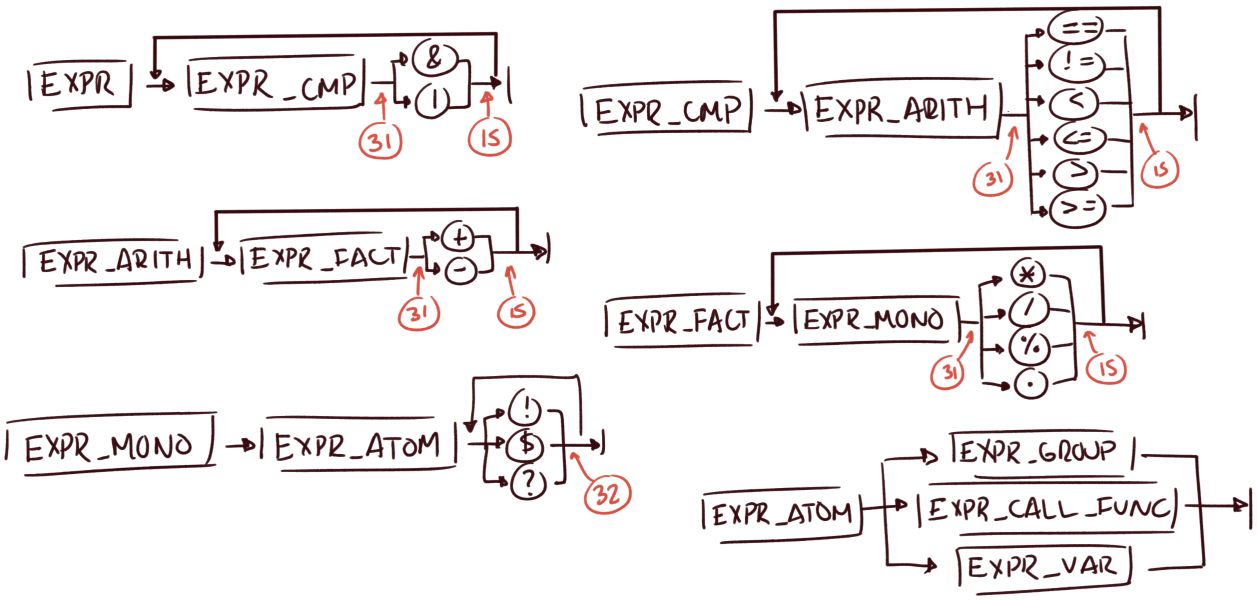
```
# 26: Add quads for end of ‘from’
loop
def p_from(p):
  "from : FROM '(' ID
found_from_iterator '=' expr
found_from_start TO expr ')'
found_from_cond DO body"
  quads.addFromEndQuads()
```

```
# 27: Prepare for ‘while’ loop
def p_found_while(p):
  "found_while : empty"
  quads.prepareLoop()
```

```
# 28: Add GoToF for ‘while’
conditional
def p_found_while_expr(p):
  "found_while_expr : empty"
  quads.addGoToFQuad()
```

```
# 29: Complete quad for end of loop
def p_while(p):
  "while : WHILE found_while '(' expr
')' found_while_expr DO body"
  quads.completeLoopQuad()
```

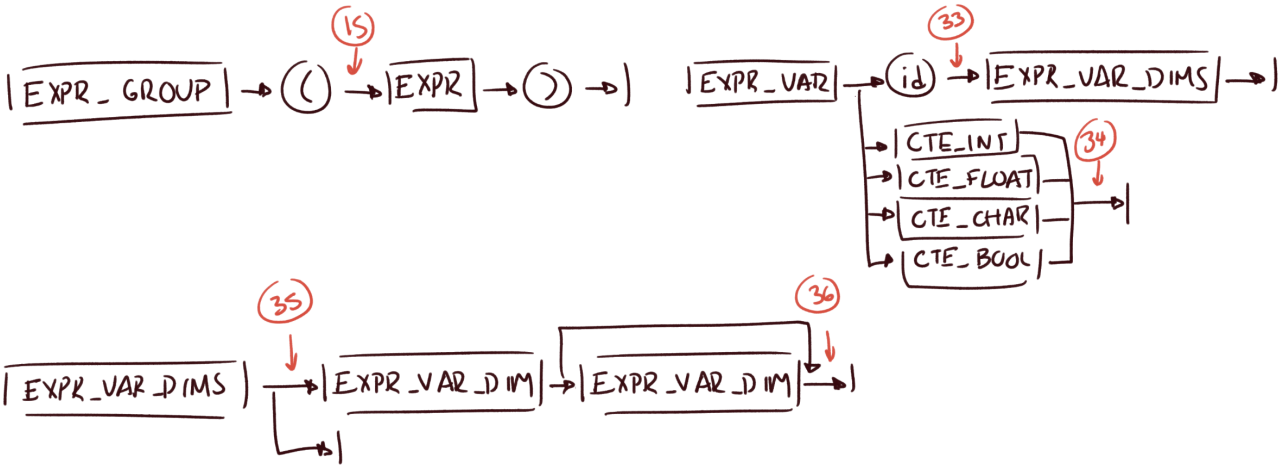
```
# 30: Add BREAK quad
def p_break(p):
  "break : BREAK ';'"
  quads.addBreakQuad()
```



31. Add dual-op operation quad when possible
32. Add mono-op operation quad

```
#31: Add dual-op quad when possible
def p_found_expr_<op>(p):
  "found_expr_compare : empty"
  quads.addDualOpQuad(ops)
```

```
#32: Add mono-op quad
def p_found_expr_mono_op(p):
  "found_expr_mono_op : empty"
  quads.addMonoOpQuad(p[-1])
```



33. Push variable to `sVars` stack
34. Push constant to `sVars` stack
35. Prepare array access for variable

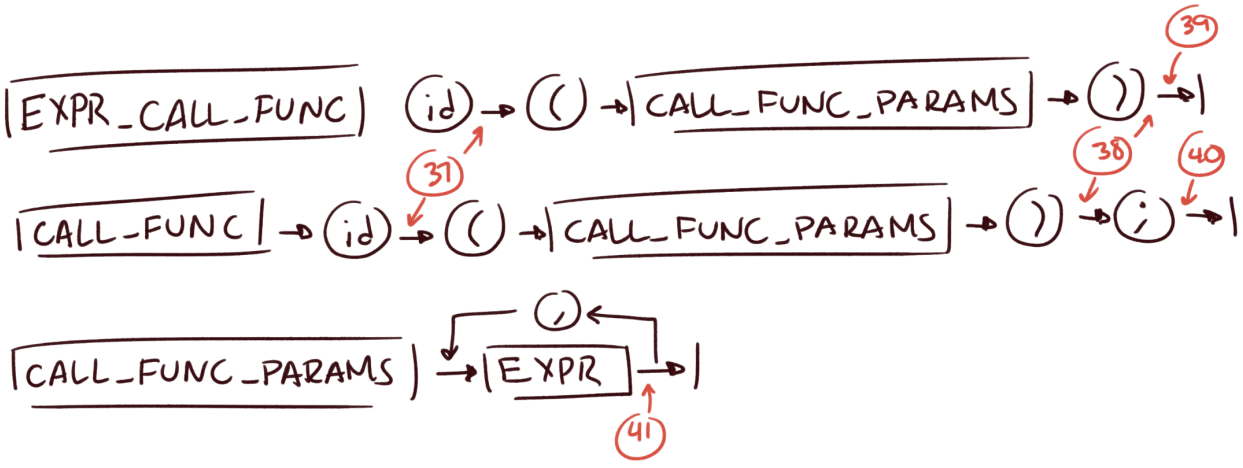
36. Add base address to array access

```
# 33: Push variable to sVars stack
def p_expr_var_name(p):
  "expr_var_name : ID"
  var = funcDir.getVar(p[1])
  funcDir.setVarHelper(var.name)
  quads.pushVar(var)
```

```
# 34: Push constant to sVars stack
def p_cte(p):
  """expr_var : CTE_INT
              | CTE_FLOAT
              | CTE_BOOL
              | CTE_CHAR"""
  t = None
  if (type(p[1]) is int):
    t = 'int'
  elif (type(p[1]) is float):
    t = 'float'
  elif (type(p[1]) is str):
    t = 'char'
  elif (type(p[1] is bool)):
    t = 'bool'

  cte = quads.upsertCte(p[1], t)
  quads.pushCte(cte)
```

```
#35: Prepare array access for var
def p_found_expr_var_dims(p):
  "found_expr_var_dims : empty"

quads.sDims.append((funcDir.varHelper
, 0))
```

```
#36: Add base address
def p_expr_var_dims(p):
  """expr_var_dims :
found_expr_var_dims expr_var_dim
expr_var_dim
                   |
found_expr_var_dims expr_var_dim"""
  quads.addBaseAddressQuad()
```



37. Prepare ERA for calling function

38. Verify parameter count and add `GoSub` quad
39. Assign return value of function call
40. Check if function isn’t void
41. Add parameter `PARAM` quad for calling function

```
# 37: Prepare ERA for calling
function
def p_found_call_func_name(p):
  "found_call_func_name : empty"
  func = p[-1]
  if funcDir.functionExists(func):
    quads.addEraQuad(func)
    quads.pushFunction(func)
    quads.sOperators.append('(')
  else:
    raise Exception(f'Function
{func}() does not exist!')
```

```
# 38: Verify param count and add
GoSub quad
def p_found_call_func_end(p):
  "found_call_func_end : empty"
  func = quads.getTopFunction()
  if funcDir.verifyParamCount(func):
    funcDir.resetParamCount()
    quads.addGoSubQuad(func,
funcDir.getQuadStartOfFunc(func))
    quads.popOperator()
  else:
    raise Exception(f'Wrong number
of parameters in {func}!')
```

```
# 39: Assign return value of
function call
def p_expr_call_func(p):
  "expr_call_func : ID
found_call_func_name '('
call_func_params ')'
found_call_func_end"
  quads.addAssignFuncQuad()
```

```
# 40: Check if function isn’t void
def p_call_func(p):
  "call_func : ID
found_call_func_name '('
call_func_params ')'
found_call_func_end ';'"
  func = quads.popFunction()
  if
quads.funcDir.getReturnTypeOfFunc(fu
nc) != 'void':
    raise Exception(f"This function
is non-void, therefore it can't be
used as an expression! -> {func}")
```

```
# 41: Add parameter and PARAM quad
def p_func_single_step(p):
  "func_single_step : empty"
  target_param =
funcDir.getParamOfFunc(quads.getTopF
unction())
  quads.addParamQuad(target_param,
funcDir.paramCount)
  funcDir.incrementParamCount()
  pass
```

Semantic table/cube

The following table shows the valid dual-operand operations, depending on the types and operator used. If a combination is not found in this table, then it yields an error:

Left operand	Operator	Right operand	Result
int	=	int	int
int	+	int	int
int	-	int	int
int	*	int	int
int	/	int	int
int	%	int	int
int	.	int	int
int	==	int	bool
int	!=	int	bool
int	<	int	bool
int	<=	int	bool
int	>	int	bool
int	>=	int	bool
int	+	float	float
int	-	float	float
int	*	float	float
int	/	float	float
int	.	float	float
int	==	float	bool
int	!=	float	bool
int	<	float	bool
int	<=	float	bool
int	>	float	bool
int	>=	float	bool
float	=	int	float
float	+	int	float
float	-	int	float
float	*	int	float
float	/	int	float
float	.	int	float
float	==	int	bool
float	!=	int	bool
float	<	int	bool
float	<=	int	bool
float	>	int	bool
float	>=	int	bool
float	=	float	float
float	+	float	float

float	-	float	float
float	*	float	float
float	/	float	float
float	.	float	float
float	==	float	bool
float	!=	float	bool
float	<	float	bool
float	<=	float	bool
float	>	float	bool
float	>=	float	bool
char	=	char	char
char	==	char	bool
char	!=	char	bool
bool	=	bool	bool
bool	&	bool	bool
bool	|	bool	bool
bool	==	bool	bool
bool	!=	bool	bool

This table is similar, but handles mono-operand operations, specifically for matrix operations (bear in mind this doesn't take into account the matrix's dimensions, as that is checked after this validation):

Operand	Operator	Result
int	$	int
int	!	int
int	?	float
float	$	float
float	!	float
float	?	float
char	!	char
bool	!	bool

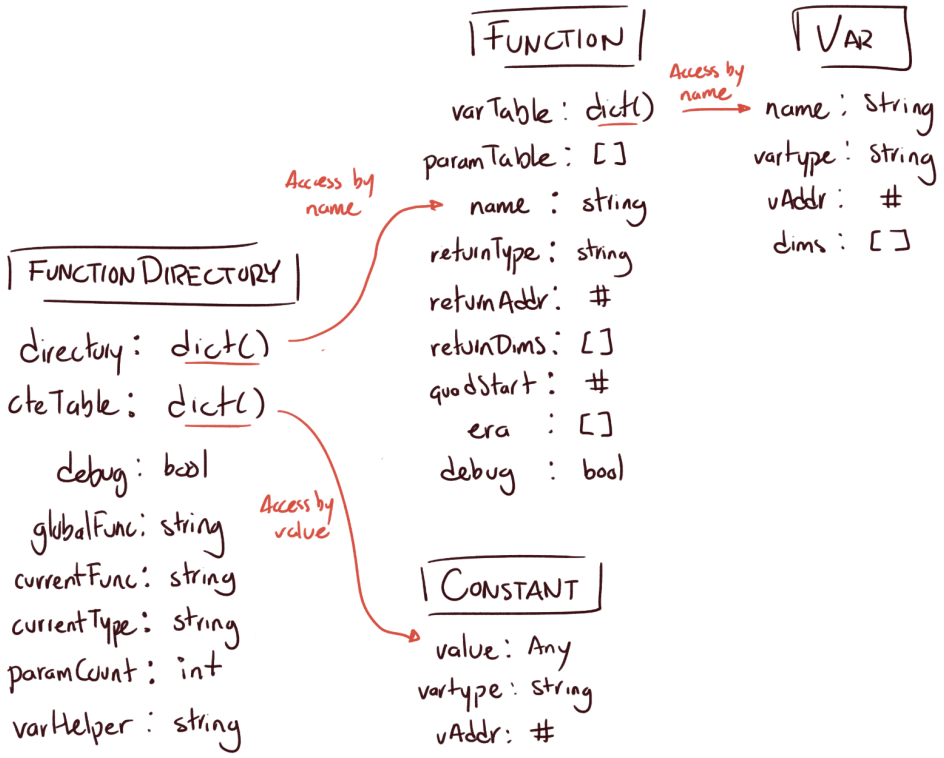
Memory in Compilation

As mentioned before, the main brains of the operation are the *quadruple manager* and the *function directory*. First and foremost a global function is registered into the function directory. This assigns a `Function` object to the `directory` dictionary (using the function's name as the key) of the `FunctionDirectory` instance, initialized with a `name` and a `return_type`. From that point forward is a matter of how the user codes a function to populate it with different pieces of data, such as local variables, parameters, quad start, era size, etc.

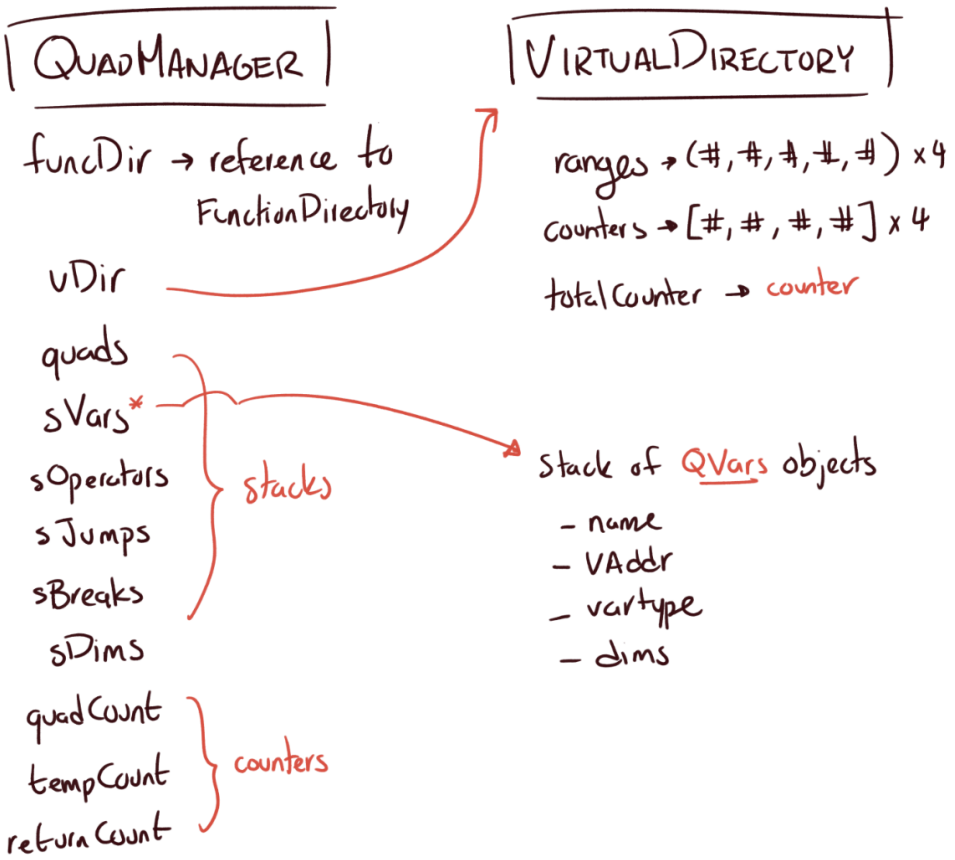
Variables are declared right after a function's declaration to assign it to its var table (or before any other function during the global variable declaration). The var table is also a dictionary that uses the variable's `name` value as the key for access, and the variable itself is stored as a `Var` object. Both of these objects are similar with the exception of Constants not having a dimensions property.

It is also worth mentioning that the var table is stored *within the Function object*. This was done as a preventative measure at first, but it has stayed throughout development ever since. Constants behave differently as they are stored "globally" in the function directory in its own table, called the `cteTable`. Works almost the same way as a var table, except it uses the constant's value as the key to that dictionary.

The following diagram shows how the FunctionDirectory instance is made up of:



The quad manager, when given the indication to create quads, uses stacks and a `VirtualDirectory` instance to keep track of variables and addresses, while being the headquarters of operations as it holds a reference to the function directory as well. Besides the usage of the VirtualDirectory and the reference to the FunctionDirectory, the memory is pretty straightforward with the usage of stacks and counters. It's worth noting that the `sVars` stack is a stack of variables, constants, and temporary values wrapped in a singular class for easier access and manipulation. On the other hand, the VirtualDirectory instance can be described as a primitive version of memory usage, as it uses regular tuples and lists to keep track of memory usage. It keeps a 4 tuples of 5 elements each to keep track of the bounds of memory for each range and type, plus keeping a counter of the amount of variable types used locally and temporarily for calculating ERA function size. The structure is as followed:



Also, finally, the semanticCube.py is simply two dictionaries that uses as keys variable types and operators for the sake of easily and quickly checking for result types for operations.

The Virtual Machine

HonkVM, your virtual companion

Now on to the virtual machine! The virtual machine runs on the same version and platform as the compiler: Python 3.6. The only difference is that the usage of **numpy** is required for this to work correctly, as it uses some functions for matrix operations. Other than that, it's practically the same.

Memory in Execution

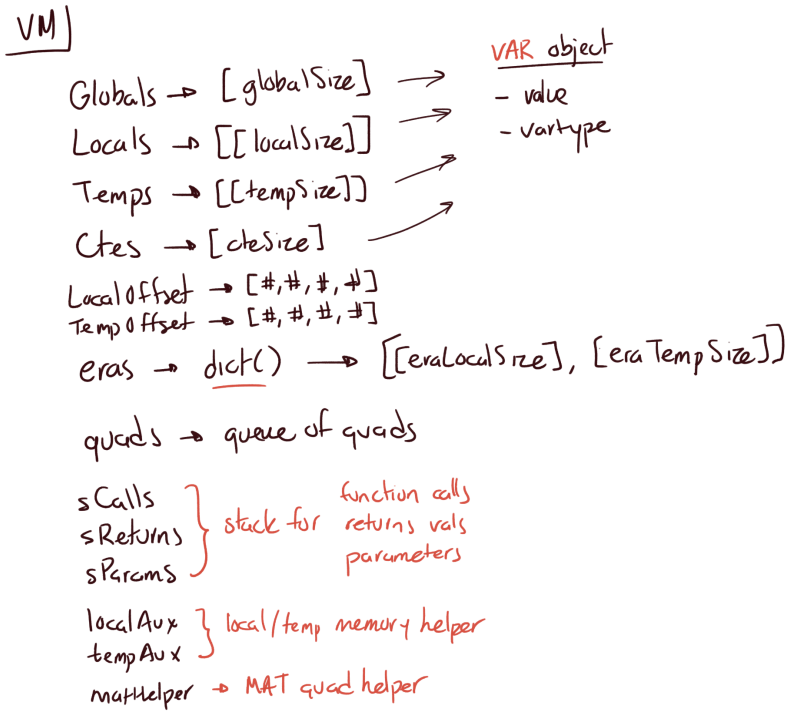
When it initializes, it'll read the data from the built .o file from the compiler, & it'll set itself up using that data. The first thing it does is read the memory ranges from the file & creates its own "memory" with those sizes. Global and constants create a list of that size, BUT locals and temporaries are placed inside a **stack** with that size, as it'll be used and replaced for function calls. A list of *offsets* are also created for these function calls in relation to how ERA sizes are handled later on

Function calls are also going to create "function memory" equal to their ERA size, so the VM reads the ERA sizes and stores them in a dictionary for later use, with the key being the function's name and the value being the local and temporary size in a list.

Afterwards, quads are later placed into a queue and constants are stored in the Ctes memory for execution, & a set of stack and helpers are initialized right after.

During execution, constants are loaded to be stored in `Var` objects, in charge of giving accurate values and vartypes to the virtual machine.

The structure of the VM is as follows:



How memory works in the virtual machine is a tad different to the addresses set in the compiler. Global and constants memory are fixed to their sizes, & the addresses set in the compiler are equal to the VM. Simple.
However, local and temporary values work differently, as they have to be “saved” between function calls. With this in mind, local and temporary memory works like a *stack*, where the topmost element is the current instance of local and temporary memory. Whenever a function call is called, it prepares to allocate a piece of memory equal to its ERA size. This pushes a new set of memory into the `Locals` and `Temps` memory stacks, as well as pushes new offsets to the `LocalOffsets` and `TempOffsets` stacks. These offsets determine where the “new” memory is stored and any changes to that segment of memory is adjusted accordingly.
Temp memory is a special case because it’s accessible in *both* global and local memory, so it acts as a hybrid between stacked memory and fixed memory, depending on the situation.

The testing

Here are a few tests to run the compiler through and show some of its capabilities… (and for simplicity's sake, we are only going to use the regular syntax for this)

Sort and find (sort an array of 7 number and find an index of a value)

```
Program sortAndFind;
var int a, ind, arr[7];

%% Insertion Sort
function void ins_sort()
var int key, j;
{
  from (i = 1 to 6) do {
    key = arr[i];
    j = i - 1;
    while (j >= 0) do {
      if (arr[j] < key) then {
        break;
      }
      arr[j + 1] = arr[j];
      j = j - 1;
    }
    arr[j + 1] = key;
  }
}

%% Find index of value
function int index(int val) {
  from (i = 0 to 6) do {
    if (arr[i] == val) then {
      return(i);
    }
  }

  print("Value doesn't exist in array!");
  return(-1);
}

main() {
  read(arr);
  print("Original array:", arr);
  ins_sort();
  print("Sorted array:", arr);

  print("Search for what value?");
  read(a);
  ind = index(a);
  print("Index:", ind);
  if (ind != -1) then {
    print("Element from index:", arr[ind]);
  }
}
```

Results - Quadruples

```
-- main - void
0:      GoTo    None    None    None
```

```
                                    > VAR: a - int -> 1000
                                    > VAR: ind - int -> 1001
                                    > VAR: arr - int -> 1002
                                    >> VAR: arr - int[7] -> 1002 - 1008
-- ins_sort - void
                                    > VAR: key - int -> 9000
                                    > VAR: j - int -> 9001
                                    > VAR: i - int -> 20000
                                    > CTE: 1 - int -> 31000
1:      =       31000   None    20000
                                    > CTE: 6 - int -> 31001
2:      <=      20000   31001   29500
                                    > TMP: t0 - bool -> 29500
3:      GoToF   29500   None    None
4:      VERIFY  20000   None    7
5:      +->     20000   1002    20001
                                    > PTR: t1 - int -> (20001)
6:      =       (20001,)        None    9000
7:      -       20000   31000   20002
                                    > TMP: t2 - int -> 20002
8:      =       20002   None    9001
                                    > CTE: 0 - int -> 31002
9:      >=      9001    31002   29501
                                    > TMP: t3 - bool -> 29501
10:     GoToF   29501   None    None
11:     VERIFY  9001    None    7
12:     +->     9001    1002    20003
                                    > PTR: t4 - int -> (20003)
13:     <       (20003,)        9000    29502
                                    > TMP: t5 - bool -> 29502
14:     GoToF   29502   None    None
15:     GoTo    None    None    None
                                    ! Completed quad #14 with jump to 16
16:     +       9001    31000   20004
                                    > TMP: t6 - int -> 20004
17:     VERIFY  20004   None    7
18:     +->     20004   1002    20005
                                    > PTR: t7 - int -> (20005)
19:     VERIFY  9001    None    7
20:     +->     9001    1002    20006
                                    > PTR: t8 - int -> (20006)
21:     =       (20006,)        None    (20005,)
22:     -       9001    31000   20007
                                    > TMP: t9 - int -> 20007
23:     =       20007   None    9001
24:     GoTo    None    None    9
                                    ! Completed quad #10 with jump to 25
                                    ! Completed quad #15 with jump to 25
25:     +       9001    31000   20008
                                    > TMP: t10 - int -> 20008
26:     VERIFY  20008   None    7
27:     +->     20008   1002    20009
                                    > PTR: t11 - int -> (20009)
28:     =       9000    None    (20009,)
29:     +       20000   31000   20010
                                    > TMP: t12 - int -> 20010
30:     =       20010   None    20000
31:     GoTo    None    None    2
                                    ! Completed quad #3 with jump to 32
32:     EndFunc None    None    None
-- index - int
```

```
                                        > VAR: val - int -> 9000
                                        > VAR: i - int -> 20000
33:     =       31002   None    20000
34:     <=      20000   31001   29500
                                        > TMP: t0 - bool -> 29500
35:     GoToF   29500   None    None
36:     VERIFY  20000   None    7
37:     +->     20000   1002    20001
                                        > PTR: t1 - int -> (20001)
38:     ==      (20001,)        9000    29501
                                        > TMP: t2 - bool -> 29501
39:     GoToF   29501   None    None
                                        ! Set the function's return address to (9001)
40:     =       20000   None    9001
41:     RETURN  None    None    9001
                                        ! Completed quad #39 with jump to 42
42:     +       20000   31000   20002
                                        > TMP: t3 - int -> 20002
43:     =       20002   None    20000
44:     GoTo    None    None    34
                                        ! Completed quad #35 with jump to 45
45:     PRINT   None    None    "Value doesn't exist in array!"
                                        > CTE: -1 - int -> 31003
46:     =       31003   None    9001
47:     RETURN  None    None    9001
48:     EndFunc None    None    None
                                        ! Completed quad #0 with jump to 49
--- main
                                        ! Preparing for matrix of [1][7]
49:     MAT     1       7       None
50:     READ    None    None    1002
51:     PRINT   None    None    "Original array:"
                                        ! Preparing for matrix of [1][7]
52:     MAT     1       7       None
53:     PRINT   None    None    1002
54:     ERA     None    None    ins_sort
55:     GoSub   None    None    1
56:     PRINT   None    None    "Sorted array:"
                                        ! Preparing for matrix of [1][7]
57:     MAT     1       7       None
58:     PRINT   None    None    1002
59:     PRINT   None    None    "Search for what value?"
60:     READ    None    None    1000
61:     ERA     None    None    index
62:     PARAM   1000    9000    0
63:     GoSub   None    None    33
64:     =>      None    None    20000
                                        > RET: t0 - int -> 20000
65:     =       20000   None    1001
66:     PRINT   None    None    "Index:"
67:     PRINT   None    None    1001
68:     !=      1001    31003   29500
                                        > TMP: t1 - bool -> 29500
69:     GoToF   29500   None    None
70:     PRINT   None    None    "Element from index:"
71:     VERIFY  1001    None    7
72:     +->     1001    1002    20001
                                        > PTR: t2 - int -> (20001)
73:     PRINT   None    None    (20001,)
                                        ! Completed quad #69 with jump to 74
74:     END     None    None    None
```

Results – Execution

```
Honk on  master
➜  python3 honk.py sortAndFind.txt
> 6
> 3
> 2
> 4
> 5
> 7
> 8
Original array:
[6, 3, 2, 4, 5, 7, 8]
Sorted array:
[2, 3, 4, 5, 6, 7, 8]
Search for what value?
> 7
Index:
5
Element from index:
7
```

Factorial and Fibonacci sequence (cycle and recursive)

```
Program factFibo;
var
  int a;

%% Factorial - Cycle
function int factCycle(int x)
var int ret;
{
  if (x <= 0) then {
    print("Can't print a factorial of 0 or less!");
    return(-1);
  }
  ret = 1;
  from (i = 1 to x) do {
    ret = ret * i;
  }
  return(ret);
}

%% Factorial - Recursion
function int factRecur(int x) {
  if (x <= 0) then {
    print("Can't print a factorial of 0 or less!");
```

```
    return(-1);
  }
  if (x == 1) then { return(x); }
  return(x * factRecur(x - 1));
}

%% Fibonacci - Cycle
function int fiboCycle(int x)
var int first, second, ret;
{
  if (x < 0) then {
    print("Can't use a negative index!");
    return(-1);
  }
  if (x <= 1) then {
    return(x);
  }

  first = 0;
  second = 1;
  from (i = 2 to x) do {
    ret = first + second;
    first = second;
    second = ret;
  }

  return(ret);
}

%% Fibonacci - Recursion
function int fiboRecur(int x) {
  if (x < 0) then {
    print("Can't use a negative index!");
    return(-1);
  }
  if (x == 0) then { return(0); }
  if (x == 1) then { return(1); }
  return(fiboRecur(x - 1) + fiboRecur(x - 2));
}

main() {
  read(a);
  print(factCycle(a));
  print(factRecur(a));

  read(a);
  print(fiboCycle(a));
  print(fiboRecur(a));
}
```

Results - Quads

```
-- main - void
0:     GoTo   None   None   None
                                  > VAR: a - int -> 1000
```

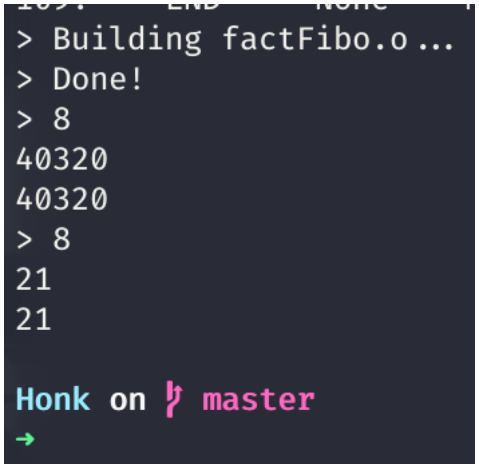
```
-- factCycle - int
                                > VAR: x - int -> 9000
                                > VAR: ret - int -> 9001
                                > CTE: 0 - int -> 31000
1:     <=     9000   31000  29500
                                > TMP: t0 - bool -> 29500
2:     GoToF  29500  None   None
3:     PRINT  None   None   "Can't print a factorial of 0 or less!"
                                > CTE: -1 - int -> 31001
                                ! Set the function's return address to (9002)
4:     =      31001  None   9002
5:     RETURN None   None   9002
                                ! Completed quad #2 with jump to 6
                                > CTE: 1 - int -> 31002
6:     =      31002  None   9001
                                > VAR: i - int -> 20000
7:     =      31002  None   20000
8:     <=     20000  9000   29501
                                > TMP: t1 - bool -> 29501
9:     GoToF  29501  None   None
10:    *      9001   20000  20001
                                > TMP: t2 - int -> 20001
11:    =      20001  None   9001
12:    +      20000  31002  20002
                                > TMP: t3 - int -> 20002
13:    =      20002  None   20000
14:    GoTo   None   None   8
                                ! Completed quad #9 with jump to 15
15:    =      9001   None   9002
16:    RETURN None   None   9002
17:    EndFunc       None   None   None
-- factRecur - int
                                > VAR: x - int -> 9000
18:    <=     9000   31000  29500
                                > TMP: t0 - bool -> 29500
19:    GoToF  29500  None   None
20:    PRINT  None   None   "Can't print a factorial of 0 or less!"
                                ! Set the function's return address to (9001)
21:    =      31001  None   9001
22:    RETURN None   None   9001
                                ! Completed quad #19 with jump to 23
23:    ==     9000   31002  29501
                                > TMP: t1 - bool -> 29501
24:    GoToF  29501  None   None
25:    =      9000   None   9001
26:    RETURN None   None   9001
                                ! Completed quad #24 with jump to 27
27:    ERA    None   None   factRecur
28:    -      9000   31002  20000
                                > TMP: t2 - int -> 20000
29:    PARAM  20000  9000   0
30:    GoSub  None   None   18
31:    =>     None   None   20001
                                > RET: t3 - int -> 20001
```

```
32:    *      9000   20001  20002
                                  > TMP: t4 - int -> 20002
33:    =      20002  None   9001
34:    RETURN None   None   9001
35:    EndFunc       None   None   None
-- fiboCycle - int
                                  > VAR: x - int -> 9000
                                  > VAR: first - int -> 9001
                                  > VAR: second - int -> 9002
                                  > VAR: ret - int -> 9003
36:    <      9000   31000  29500
                                  > TMP: t0 - bool -> 29500
37:    GoToF  29500  None   None
38:    PRINT  None   None   "Can't use a negative index!"
                                  ! Set the function's return address to (9004)
39:    =      31001  None   9004
40:    RETURN None   None   9004
                                  ! Completed quad #37 with jump to 41
41:    <=     9000   31002  29501
                                  > TMP: t1 - bool -> 29501
42:    GoToF  29501  None   None
43:    =      9000   None   9004
44:    RETURN None   None   9004
                                  ! Completed quad #42 with jump to 45
45:    =      31000  None   9001
46:    =      31002  None   9002
                                  > VAR: i - int -> 20000
                                  > CTE: 2 - int -> 31003
47:    =      31003  None   20000
48:    <=     20000  9000   29502
                                  > TMP: t2 - bool -> 29502
49:    GoToF  29502  None   None
50:    +      9001   9002   20001
                                  > TMP: t3 - int -> 20001
51:    =      20001  None   9003
52:    =      9002   None   9001
53:    =      9003   None   9002
54:    +      20000  31002  20002
                                  > TMP: t4 - int -> 20002
55:    =      20002  None   20000
56:    GoTo   None   None   48
                                  ! Completed quad #49 with jump to 57
57:    =      9003   None   9004
58:    RETURN None   None   9004
59:    EndFunc       None   None   None
-- fiboRecur - int
                                  > VAR: x - int -> 9000
60:    <      9000   31000  29500
                                  > TMP: t0 - bool -> 29500
61:    GoToF  29500  None   None
62:    PRINT  None   None   "Can't use a negative index!"
                                  ! Set the function's return address to (9001)
63:    =      31001  None   9001
64:    RETURN None   None   9001
```

```
                                    ! Completed quad #61 with jump to 65
65:    ==     9000   31000  29501
                                    > TMP: t1 - bool -> 29501
66:    GoToF  29501  None   None
67:    =      31000  None   9001
68:    RETURN None   None   9001
                                    ! Completed quad #66 with jump to 69
69:    ==     9000   31002  29502
                                    > TMP: t2 - bool -> 29502
70:    GoToF  29502  None   None
71:    =      31002  None   9001
72:    RETURN None   None   9001
                                    ! Completed quad #70 with jump to 73
73:    ERA    None   None   fiboRecur
74:    -      9000   31002  20000
                                    > TMP: t3 - int -> 20000
75:    PARAM  20000  9000   0
76:    GoSub  None   None   60
77:    =>     None   None   20001
                                    > RET: t4 - int -> 20001
78:    ERA    None   None   fiboRecur
79:    -      9000   31003  20002
                                    > TMP: t5 - int -> 20002
80:    PARAM  20002  9000   0
81:    GoSub  None   None   60
82:    =>     None   None   20003
                                    > RET: t6 - int -> 20003
83:    +      20001  20003  20004
                                    > TMP: t7 - int -> 20004
84:    =      20004  None   9001
85:    RETURN None   None   9001
86:    EndFunc       None   None   None
                                    ! Completed quad #0 with jump to 87
--- main
87:    READ   None   None   1000
88:    ERA    None   None   factCycle
89:    PARAM  1000   9000   0
90:    GoSub  None   None   1
91:    =>     None   None   20000
                                    > RET: t0 - int -> 20000
92:    PRINT  None   None   20000
93:    ERA    None   None   factRecur
94:    PARAM  1000   9000   0
95:    GoSub  None   None   18
96:    =>     None   None   20001
                                    > RET: t1 - int -> 20001
97:    PRINT  None   None   20001
98:    READ   None   None   1000
99:    ERA    None   None   fiboCycle
100:   PARAM  1000   9000   0
101:   GoSub  None   None   36
102:   =>     None   None   20002
                                    > RET: t2 - int -> 20002
103:   PRINT  None   None   20002
```

```
104:   ERA    None   None   fiboRecur
105:   PARAM  1000   9000   0
106:   GoSub  None   None   60
107:   =>     None   None   20003
                                  > RET: t3 - int -> 20003
108:   PRINT  None   None   20003
109:   END    None   None   None
```

Results – Execution



Fun with arrays and matrixes

```
Program arraysAndMatrixes;
var
  int atomIa, atomIb, det;
  float atomFa, atomFb;
  int arrIa[3], arrIb[3];
  float arrFa[3], arrFb[3];
  int twosome[2][2], twosometwin[2][2], tester[2][2];
  int tower[3][1], matIa[3][3], matIb[4][3], matTrans[3][4];
  float inverted[2][2];
  float matFa[3][3], matFb[4][3];
  bool arrBool[3];

function void fillTwosometwin() {
  from (i = 0 to 1) do {
    from (j = 0 to 1) do {
      twosometwin[i][j] = i * 2 + j + 1;
    }
  }
}

main() {
  print("Write to a variable:");
  read(atomIa);
  print(atomIa);

  print("Write to an array [3]:");
```

```
  read(arrIa);
  print(arrIa);

  print("Write to a matrix [2][2]:");
  read(twosome);
  print(twosome);

  fillTwosometwin();
  print(twosometwin);

  print("You can do batch processes with equal matrixes and lists, like sums, multiplication,
comparisons, quite a few things:");
  print("");
  print("Your matrix + [[1, 2], [3, 4]]");
  print(twosome + twosometwin);

  print("");
  print("Your matrix * [[1, 2], [3, 4]]");
  print(twosome * twosometwin);

  print("");
  print("Your matrix . [[1, 2], [3, 4]] (dot product)");
  print(twosome . twosometwin);

  print("");
  print("Your matrix > [[1, 2], [3, 4]]");
  print(twosome > twosometwin);

  print("You can even put a matrix of booleans through a conditional and it'll act like an
AND gate");
  print("");
  if (twosome > twosometwin) then {
    print("Every element in twosome beat twosometwin!");
  } else {
    print("At least one element in twosome got beat from twosometwin!");
  }

  print("");
  print("Get the determinant of a matrix:");
  print(twosome$);

  print("");
  print("Get the transpose of a matrix:");
  print(twosome!);

  print("");
  print("Get the inverse of a matrix");
  print(twosome?);
}
```

Results – Quads

```
-- main - void
0:     GoTo   None   None   None
                                  > VAR: atomIa - int -> 1000
```

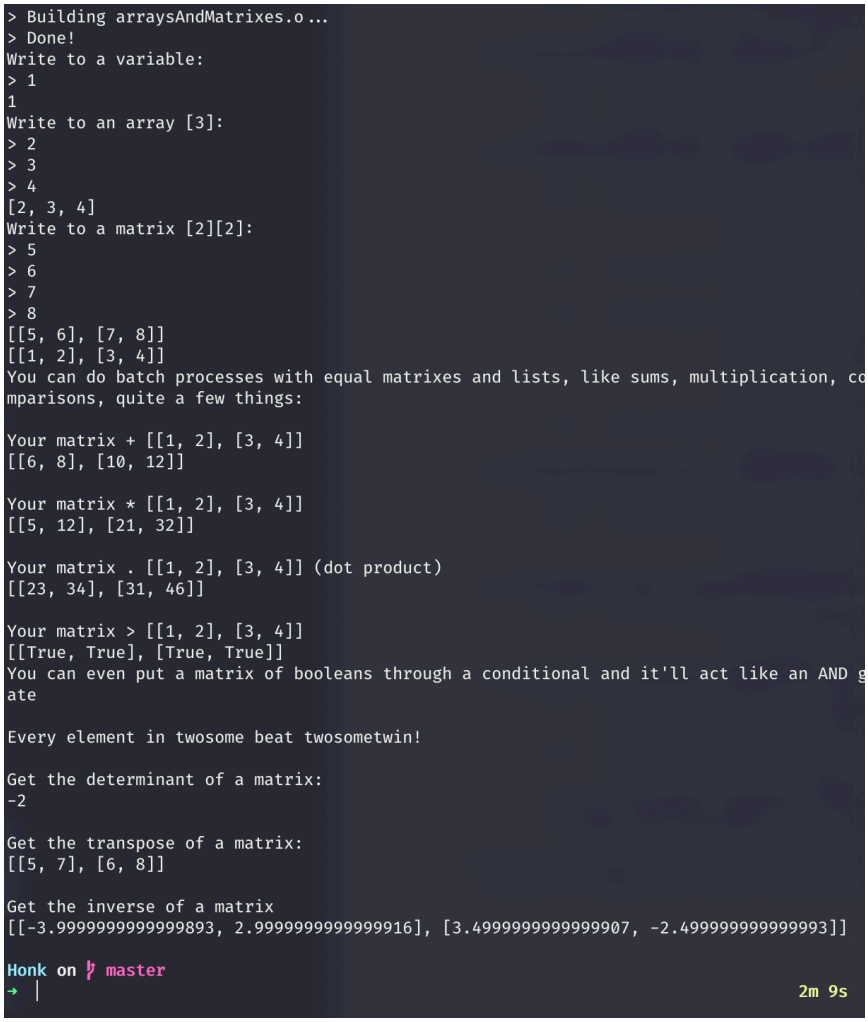
```
                                    > VAR: atomIb - int -> 1001
                                    > VAR: det - int -> 1002
                                    > VAR: atomFa - float -> 4000
                                    > VAR: atomFb - float -> 4001
                                    > VAR: arrIa - int -> 1003
                                    >> VAR: arrIa - int[3] -> 1003 - 1005
                                    > VAR: arrIb - int -> 1006
                                    >> VAR: arrIb - int[3] -> 1006 - 1008
                                    > VAR: arrFa - float -> 4002
                                    >> VAR: arrFa - float[3] -> 4002 - 4004
                                    > VAR: arrFb - float -> 4005
                                    >> VAR: arrFb - float[3] -> 4005 - 4007
                                    > VAR: twosome - int -> 1009
                                    >> VAR: twosome - int[2, 2] -> 1009 - 1012
                                    > VAR: twosometwin - int -> 1013
                                    >> VAR: twosometwin - int[2, 2] -> 1013 - 1016
                                    > VAR: tester - int -> 1017
                                    >> VAR: tester - int[2, 2] -> 1017 - 1020
                                    > VAR: tower - int -> 1021
                                    >> VAR: tower - int[3, 1] -> 1021 - 1023
                                    > VAR: matIa - int -> 1024
                                    >> VAR: matIa - int[3, 3] -> 1024 - 1032
                                    > VAR: matIb - int -> 1033
                                    >> VAR: matIb - int[4, 3] -> 1033 - 1044
                                    > VAR: matTrans - int -> 1045
                                    >> VAR: matTrans - int[3, 4] -> 1045 - 1056
                                    > VAR: inverted - float -> 4008
                                    >> VAR: inverted - float[2, 2] -> 4008 - 4011
                                    > VAR: matFa - float -> 4012
                                    >> VAR: matFa - float[3, 3] -> 4012 - 4020
                                    > VAR: matFb - float -> 4021
                                    >> VAR: matFb - float[4, 3] -> 4021 - 4032
                                    > VAR: arrBool - bool -> 8000
                                    >> VAR: arrBool - bool[3] -> 8000 - 8002
-- fillTwosometwin - void
                                    > VAR: i - int -> 20000
                                    > CTE: 0 - int -> 31000
1:     =      31000  None   20000
                                    > CTE: 1 - int -> 31001
2:     <=     20000  31001  29500
                                    > TMP: t0 - bool -> 29500
3:     GoToF  29500  None   None
                                    > VAR: j - int -> 20001
4:     =      31000  None   20001
5:     <=     20001  31001  29501
                                    > TMP: t1 - bool -> 29501
6:     GoToF  29501  None   None
7:     VERIFY 20000  None   2
                                    > CTE: 2 - int -> 31002
8:     *      20000  31002  20002
                                    > TMP: t2 - int -> 20002
9:     VERIFY 20001  None   2
10:    +      20002  20001  20003
                                    > TMP: t3 - int -> 20003
```

```
11:    +->    20003  1013   20004
                                  > PTR: t4 - int -> (20004)
12:    *      20000  31002  20005
                                  > TMP: t5 - int -> 20005
13:    +      20005  20001  20006
                                  > TMP: t6 - int -> 20006
14:    +      20006  31001  20007
                                  > TMP: t7 - int -> 20007
15:    =      20007  None   (20004,)
16:    +      20001  31001  20008
                                  > TMP: t8 - int -> 20008
17:    =      20008  None   20001
18:    GoTo   None   None   5
                                  ! Completed quad #6 with jump to 19
19:    +      20000  31001  20009
                                  > TMP: t9 - int -> 20009
20:    =      20009  None   20000
21:    GoTo   None   None   2
                                  ! Completed quad #3 with jump to 22
22:    EndFunc       None   None   None
                                  ! Completed quad #0 with jump to 23
--- main
23:    PRINT  None   None   "Write to a variable:"
24:    READ   None   None   1000
25:    PRINT  None   None   1000
26:    PRINT  None   None   "Write to an array [3]:"
                                  ! Preparing for matrix of [1][3]
27:    MAT    1      3      None
28:    READ   None   None   1003
                                  ! Preparing for matrix of [1][3]
29:    MAT    1      3      None
30:    PRINT  None   None   1003
31:    PRINT  None   None   "Write to a matrix [2][2]:"
                                  ! Preparing for matrix of [2][2]
32:    MAT    2      2      None
33:    READ   None   None   1009
                                  ! Preparing for matrix of [2][2]
34:    MAT    2      2      None
35:    PRINT  None   None   1009
36:    ERA    None   None   fillTwosometwin
37:    GoSub  None   None   1
                                  ! Preparing for matrix of [2][2]
38:    MAT    2      2      None
39:    PRINT  None   None   1013
40:    PRINT  None   None   "You can do batch processes with equal matrixes and lists, like
sums, multiplication, comparisons, quite a few things:"
41:    PRINT  None   None   ""
42:    PRINT  None   None   "Your matrix + [[1, 2], [3, 4]]"
                                  ! Preparing for matrix of [2][2]
43:    MAT    2      2      None
44:    +      1009   1013   20000
                                  > TMP: t0 - int -> 20000
                                  ! Preparing for matrix of [2][2]
45:    MAT    2      2      None
```

```
46:    PRINT  None   None   20000
47:    PRINT  None   None   ""
48:    PRINT  None   None   "Your matrix * [[1, 2], [3, 4]]"
                                ! Preparing for matrix of [2][2]
49:    MAT    2      2      None
50:    *      1009   1013   20001
                                > TMP: t1 - int -> 20001
                                ! Preparing for matrix of [2][2]
51:    MAT    2      2      None
52:    PRINT  None   None   20001
53:    PRINT  None   None   ""
54:    PRINT  None   None   "Your matrix . [[1, 2], [3, 4]] (dot product)"
                                ! Preparing dot product for dims [2, 2] · [2, 2]
55:    MAT·   2      2      2
56:    ·      1009   1013   20002
                                ! Preparing for matrix of [2][2]
57:    MAT    2      2      None
58:    PRINT  None   None   20002
59:    PRINT  None   None   ""
60:    PRINT  None   None   "Your matrix > [[1, 2], [3, 4]]"
                                ! Preparing for matrix of [2][2]
61:    MAT    2      2      None
62:    >      1009   1013   29500
                                > TMP: t3 - bool -> 29500
                                ! Preparing for matrix of [2][2]
63:    MAT    2      2      None
64:    PRINT  None   None   29500
65:    PRINT  None   None   "You can even put a matrix of booleans through a conditional and
it'll act like an AND gate"
66:    PRINT  None   None   ""
                                ! Preparing for matrix of [2][2]
67:    MAT    2      2      None
68:    >      1009   1013   29501
                                > TMP: t4 - bool -> 29501
                                ! Preparing for matrix of [2][2]
69:    MAT    2      2      None
70:    GoToF  29501  None   None
71:    PRINT  None   None   "Every element in twosome beat twosometwin!"
72:    GoTo   None   None   None
                                ! Completed quad #70 with jump to 73
73:    PRINT  None   None   "At least one element in twosome got beat from twosometwin!"
                                ! Completed quad #72 with jump to 74
74:    PRINT  None   None   ""
75:    PRINT  None   None   "Get the determinant of a matrix:"
                                > TMP: t5 - int -> 20006
                                ! Preparing for matrix of [2][2]
76:    MAT    2      2      None
77:    $      1009   None   20006
78:    PRINT  None   None   20006
79:    PRINT  None   None   ""
80:    PRINT  None   None   "Get the transpose of a matrix:"
                                > TMP: t6 - int -> 20007
                                >> TMP: t6 - int[2, 2] -> 20007 - 20010
                                ! Preparing for matrix of [2][2]
```

```
81:    MAT    2      2      None
82:    !      1009   None   20007
                                ! Preparing for matrix of [2][2]
83:    MAT    2      2      None
84:    PRINT  None   None   20007
85:    PRINT  None   None   ""
86:    PRINT  None   None   "Get the inverse of a matrix"
                              > TMP: t7 - float -> 24000
                              >> TMP: t7 - float[2, 2] -> 24000 - 24003
                              ! Preparing for matrix of [2][2]
87:    MAT    2      2      None
88:    ?      1009   None   24000
                              ! Preparing for matrix of [2][2]
89:    MAT    2      2      None
90:    PRINT  None   None   24000
91:    END    None   None   None
```

Results – Execution



The details

There are some rather important functions used in this compiler. Of course, there are too many to describe in detail, but let's go over a few:

AddDualOpQuad(ops)

```
def addDualOpQuad(self, ops):
    # If operators stack is empty, stop here
    if not self.sOperators or self.sOperators[-1] not in ops:
      return

    # Special case for dot product operator ('.')
    if self.sOperators[-1] == '.':
      self.addDotProdQuad()
      return

    right = self.sVars.pop()
    left = self.sVars.pop()
    operator = self.sOperators.pop()

    result_type = getDuoResultType(left.vartype, right.vartype, operator)
    if not result_type:
      raise Exception(f'Type mismatch! {left.vartype} {operator}
{right.vartype}')

    if not self.areCompatibleDims(left.dims, right.dims):
      raise Exception(f'Dimension mismatch! {left.dims} != {right.dims}')

    if left.dims:
      self.addMatQuad(left.dims)

    result = self.vDir.generateVirtualAddress('temp', result_type)
    self.addQuad((operator, left.vAddr, right.vAddr, result))
    self.pushTemp(result, result_type, left.dims)

    if self.debug:
      print(f'\t\t\t\t\t> TMP: t{self.tempCount} - {result_type} -> {result}')

    self.tempCount += 1
```

This is used all over the code. This function validates and creates a dual-operand operation quad. In other words, it's in charge of making quads for sums, multiplications, divisions, comparisons, you name it. It receives a list of operators to check first (because it must validate that it's ready in the stack), and continues from there. It'll make a check if the operation is valid based on their types and dimensions, and if so, create the quad and continue on. The reason why this is so important is because over half of

the quads produced are going to be dual-operand ones, and even some compiler functions use this in the background. Whenever a sum or division or anything with two operands is parsed, this will likely be called.

AddMatQuad(dims)

```
def addMatQuad(self, dims):
    if len(dims) == 1:
      rows = 1
      cols = dims[0]
    if len(dims) == 2:
      rows = dims[0]
      cols = dims[1]

    if self.debug:
      print(f'\t\t\t\t\t! Preparing for matrix of [{rows}][{cols}]')

    self.addQuad(('MAT', rows, cols, None))
```

This is a slightly smaller quad implementation, but this is actually quite important for array/matrix/batch operations. What this does (likely after validation was made) is take a dimensions list and find its rows and columns, then create a quad using those pieces of data.
This may look rather simple, but this quad serves as an *immense* helper to certain operations possible, because the virtual machine receives this quad and keeps it stored in its `matHelper` property. The very next quad will see this and adapt to a different kind of functionality to work with a collection of data instead of an atomic variable.

SetSpace(scope, vartype, dims)

```
def setSpace(self, scope, vartype, space):
    # Select variable type
    v = None
    if vartype == 'int':
      v = 0
    elif vartype == 'float':
      v = 1
    elif vartype == 'char':
      v = 2
    elif vartype == 'bool':
      v = 3

    # Update total counter
    self.totalCounter += space

    # Select scope (with validation checks)
    if scope == 'main':   # Global
      if self.globalRanges[v] + self.globalCounter[v] + space >=
self.globalRanges[v + 1] - 1:
        raise Exception(f"Out of bounds! {vartype} in {scope}")
      self.globalCounter[v] += space
      return self.globalRanges[v] + self.globalCounter[v] - 1
    elif scope == 'temp':   # Temp
      if self.tempRanges[v] + self.tempCounter[v] + space >= self.tempRanges[v + 1]
- 1:
        raise Exception(f"Out of bounds! {vartype} in {scope}")
      self.tempCounter[v] += space
      return self.tempRanges[v] + self.tempCounter[v] - 1
    elif scope == 'cte':    # Constants
      if self.cteRanges[v] + self.cteCounter[v] + space >= self.cteRanges[v + 1] -
1:
        raise Exception(f"Out of bounds! {vartype} in {scope}")
      self.cteCounter[v] += space
      return self.cteRanges[v] + self.cteCounter[v] - 1
    else:                   # Local (any local function)
      if self.localRanges[v] + self.localCounter[v] + space >= self.localRanges[v +
1] - 1:
        raise Exception(f"Out of bounds! {vartype} in {scope}")
      self.localCounter[v] += space
      return self.localRanges[v] + self.localCounter[v] - 1

    raise Exception(f'Invalid vartype/scope?! -> {vartype}, {scope}')
```

The setSpace() function found in `virtualDirectory.py` is in charge of assigning virtual addresses to variables and constants accordingly. The way this works is it takes 3 parameters: `scope`, `vartype` and `space`. The first two will help determine “where” to allocate the variable, based on what scope and type to place it. Once it finds it, it’ll check to see if there’s available space and throw an error if there’s no more room (which is unlikely, but you have to prepare). If all goes well, the counter for that space will increase by `space` amount and it’ll return the virtual address to that space.

This is *exceedingly* important for the compiler to work with addresses instead of variable names. Every variable assignment and temporal creation involves this function because they all need a virtual address to work with. It also is used for arrays/matrixes by later asking for more space once the parser finds out it's a collection of data. The trick with that is that since it already has its virtual address, it just needs to call the function before another variable does. The two functions that are directly used for virtual addresses (`generateVirtualAddress()` & `makeSpaceForArray()`) use this function in the background.

SetValue(value, addr, matOffset=0)

```
def setValue(self, value, addr, matOffset=0):
    if self.isPointer(addr):
      ptr = addr[1:-2]
      self.setValue(value, self.getValue(ptr, matOffset))
    else:
      addr = int(addr) + matOffset
      if addr < self.globalRanges[0] or addr >= self.tempRanges[4]:
        return Exception(f'Setting in invalid memory! -> ({addr}{f" + {matOffset}"
if matOffset else ""})')

      elif addr < self.globalRanges[4]:
        rangeAddr = addr - self.globalRanges[0]
        self.Globals[rangeAddr] = Var(value, self.getTypeByRange(addr,
self.globalRanges))
      elif addr < self.localRanges[4]:
        rangeAddr = None
        if len(self.Locals) == 1:
          rangeAddr = addr - self.localRanges[0]
        else:
          rangeAddr = self.getLocalIndex(addr, self.localRanges,
self.LocalOffsets[-1])
        self.Locals[-1][rangeAddr] = Var(value, self.getTypeByRange(addr,
self.localRanges))
      elif addr < self.tempRanges[4]:
        rangeAddr = None
        if len(self.Temps) == 1:
          rangeAddr = addr - self.tempRanges[0]
        else:
          rangeAddr = self.getLocalIndex(addr, self.tempRanges, self.TempOffsets[-
1])
        self.Temps[-1][rangeAddr] = Var(value, self.getTypeByRange(addr,
self.tempRanges))
      else:
        rangeAddr = addr - self.cteRanges[0]
        self.Ctes[rangeAddr] = Var(value, self.getTypeByRange(addr,
self.cteRanges))
```

`SetValue()`, alongside `getVar()` & `getValue()`, are the most used functions in the virtual machine, and for good reason because this is the only way to set and get memory in said machine. It takes 2 parameters (`value`, `addr`)and 1 *optional* parameter (`matOffset`, defaults to 0). The first thing it does is check if it's a pointer, so it can get the value of that first to later set the value in the right spot. Outside of that, the function will attempt to save the `value` into a specified address `addr`. It'll set the value accordingly by finding the right range, getting the type (based on the address given and the ranges saved in the VM). It's of upmost importance that this works properly because the virtual machine depends on this for memory to be read and written by any of the quadruples' instructions.

Final words

Well, that's it.

If you're looking for the "second part", you can find a README zipped together with this document that shows how you can run and write in ***Honk.*** And if not, you can check out the honk repo for that same README.
There is also a demonstration video showing you a small look at how to run Honk~.

That's it…I'm tired…hope you have a good day today. I'm going to bed.