Initial tools setup

- make an empty directory for this project
- clone 0d somewhere else alongside the new project,
  - `$ git clone https://github.com/guitarvydas/0d`
- cd into the new project directory
- copy the 0d engine into your new project, `$ cp -R ../0d/0d .`
- copy the 0d library into your new project, `$ cp -R ../0d/libsrc .`
- make sure that you have access to `draw.io`. I tend to download the offline app instead of using the online version. It doesn't matter. Just become comfortable in creating a drawing and saving it. The saved `.drawio` file should be uncompressed XML (graphML, I think) and you should be able to load it into a programming editor of your choice and see it in ASCII text. It looks like you can get this from https://www.drawio.com.
- make sure that you can bring up and use the Ohm-Editor.
- load the necessary Ohm files
  - ensure that you are in the new project directory, then `$ npm install ohm-js yargs`
- ensure that you have `node.js` installed.

The copying is clearly kinda clunky, but, it ensures that we have a version of the engine that won't change under our feet. Ultimately, we would like the 0D engine to be a single LEGO block that we just snap into any project, but, we're not there yet. We will start by making LEGO blocks for our project and defer considering how to make the engine itself a LEGO block.

The copying does something else: it strips away all of 0D demo stuff that we don't need for this project.

Agenda

We want to work towards getting a set of nano-grammars for Phi working. We'll build up slowly, then go to town.

1. Write the simplest grammar possible.
2. Write a corresponding RWR specification for the simplest grammar.
3. Create a diagram containing 1 Transpiler component that uses the above grammar and RWR spec.
4. Test that the 0D / drawware environment works from one end to the other.
5. Work towards understanding how to build a set of nano-grammars by building the simplest, useless nano-grammar pipeline possible, i.e. a diagram containing 2 Transpilers chained together in a pipeline - see below steps 6, 7, and, 8.
6. Rewrite the above RWR spec so that it does not emit C code, but annotates the Go code with semantic information
7. Write another grammar that parses the output from the previous step (6).
8. Write an RWR spec that consumes the semantic info and produces valid Go code.

Since I'm rusty with Go, but remember too much about how to write C, I will use C for the examples below. You will have to convert the examples into Go and re-test everything.

Note that step 1 involves writing a *simple* grammar. I believe that the simplest grammar is a macro processor that converts a single phrase from Phi into Go. The macro processor (steps 1-4) will input pseudo-Go, that contains mostly Go code plus some simple phrase in Phi, and outputs compilable Go code, by transpiling, or macro-expanding, the Phi syntax into legal Go code.

The result is useless and should be thrown away, but, this work accomplishes:
1. Check that all of the tools are in-place and working.
2. Demo of a macro for a non-Lisp language (i.e. Go).

Once you've added macro grammars to your tool-belt, you can make chains of nano-grammars.

There is no new technology involved in creating nano-grammars. It's just a shift in mind-set. Most often we think of grammars as only for human consumption. If you can knock off transpilers *quickly*, then you can afford to think of adding grammars tuned only for machine consumption. Using machine-oriented grammars, you can write LEGO blocks, chopping up a given problem (e.g. building a compiler) into smaller, more easily implemented pieces that use syntax that is most appropriate to a sub-problem. Instead of writing only one grammar meant to appease human readers, you can write a zillion grammars most of which are meant to make processing at each step easier. For example, if you want to do semantic analysis in one pass of the compiler, e.g. type checking, then you can annotate the source code with type-checking details. Such code looks ugly to human readers, but the machine doesn't care.

Passes further down the line can use the type-checking annotations to analyze whether the source code has errors in it.

Likewise, you can fine-tune intermediate grammars to add annotations for information like scoping and lifetime (answering questions like "should this variable go on the stack, or does it need to be in the heap?"). And so on. As an example, the PT Pascal compiler chops up its work into 4 passes:

1. Parsing and syntax checking
2. Semantic analysis - type-checking, etc.
3. Allocation - figuring out where to put each variable and function (register, stack, heap, ROM, RAM, etc).
4. Emitting assembler code for some specific CPU architecture.

Note that, by the time the program code reaches step 4 (emission), all of the syntactic noise has been stripped out and no more error checking needs to be done. The programmer writing the emitter can concentrate solely on outputting correct sequences of assembler instructions.

Recap: What is the actual goal, here?

We already know how to build a compiler from scratch, so, let's do something drastically different.

Instead, let's build a bunch of nano-grammars that glue new features onto Go. E.g a workbench for experimenting with Phi, that uses Go as its "assembler". Let's strive for

an MVI - Minimum Viable Implementation. Let's cheat as much as possible - let the Go environment do all of the heavy lifting for us. “MVI” means that we scrimp on optimization instead of scrimping on features.

Advantages:
- we can use the Go build environment and workflow, without having to manually build all that stuff from scratch
- quick turn-around for trying out and revamping features of Phi

Disadvantages:
- lack of integration, e.g. the Go compiler will produce error messages that might not relate directly back to the Phi source code
  - I know from experience that we will get fresh ideas on how to fix these problems - if we really want to fix them - as we go, let's get *something* working, then worry about the niggly details like integration, optimization, etc.
  - In my experience, the stuff that I first thought was absolutely necessary turned out to dissolve and wasn't really all that necessary. I tend not to be able to predict how best to cheat and how to avoid unnecessary work. I figure that kind of thing out as I work towards the main goal - the MVI. I rely on this effect. Build it and ideas will come.

Macro-Grammar First Light

To get our feet wet, let's just build a quickie macro-grammar for some feature of Phi.

The goal should be to write so little code that we have no compunction about throwing it all away and doing it better the 2nd time around.

Fred Brooks says that you need to take 3 tries. The 3rd try throws all of the 2nd try away and builds a better version using the knowledge we've gained while doing the 1st and 2nd tries. Again, this sounds like excess work given that we've been steeped in writing code the hard way, but, if we ensure that we write only small amounts of code, then we will have no problem with simply scrapping it all and staring over again. Lispers have this attitude, but C-ers resist this kind of thing. Every line of C code represents a huge investment in terms of thinking and debugging and twisty little dependencies that intertwine every line of C code to every other line of C code (eg. the data structures). From the C perspective, it seems blasphemous to throw code away.

Our goal, though, is to aim at throwing code away. How? Cheat as much as possible at first, then write slightly better versions (and throw *them* away). The goal is to understand what you want to do. If you are already sure that you already know what you want to do, you are engaging in a practice called "The Waterfall Method".

Using function-based thinking, it is not possible to cleave code in this way. Using 0D it is possible. The difference is tiny. The effect is huge. It’s not technology so much as mind-set.

In fact, using function-based thinking is even worse than it seems. We *think* that we have LEGO blocks when we build and use code libraries, but, we don't really have LEGO blocks, because of various kinds of hidden dependencies. This mis-belief leads us to build code that only looks like LEGO blocks, but, requires all sorts of work-arounds. We waste our time inventing workarounds instead of dealing with more important issues, like solving users' problems instead of solving meta-problems.

Goal #1

Ensure that all of the machinery, the engine, the environment, is working.

How?

Build a small macro-grammar that emits Go.

Use a small test - a complete Hello-World level Go program that includes some new piece of Phi syntax.

1.1
Just build a Hello-World-ish Go program and build it and run it. Ensure that the Go program employs some piece of code that we're going to override with a layer of Phi syntax later.

This should be a total no-brainer for someone who knows how to set up and run a simple Go program. I am unfamiliar with Go. It would take me many hours / days to just get this bit running. Out of laziness, I will leave it up to you to do this.

Looking at SPEC.md, my eyes see a simple global-variable channel definition as a possible candidate for this first pass, e.g.

```
def ch(chan num)
```

What is this supposed to translate to in Go? If this isn't easy to do in Go, or you see something even simpler, then by all means let's go with that. You know more about this stuff and the ultimate goal than I do.

You can educate me by writing a markdown file containing the "before" and "after" versions of the simple test code.

Example of Before and After
When you get good at this, you will "see" the before and after code in your head and you won't need to bother writing it out for me.

I know C, so I'll write the example in C, not Go. You can re-create this using Go:

Before

```
# include <stdio.h>
def ch 5
int main (int argc, char**argv) {
  printf ("Hello World channel=%d\n", ch);
}
```

After

```
# include <stdio.h>
int ch = 5;
int main (int argc, char**argv) {
  printf ("Hello World channel=%d\n", ch);
}
```

Recap

This accomplishes: Just getting the machinery to work. Checking to see that I haven't forgotten to push vital stuff, etc.

Next - A First Macro-Grammar

This is the big leap, but, when you understand it, it looks trivially easy. I find it very difficult to put ideas, that I think are trivially easy, into words. If what I say doesn't make sense, keep prodding me.

Note that we first look at how to build a macro-grammar, then we will extend the concept to include macro-grammars.

Getting the macro-grammar to work is the "hard part". Any problems at this stage are probably caused by setup and tool issues.

Once you have a macro-grammar working, the rest is easy.

1.2 A Macro-Grammar
Focussing on the C code I wrote above, start the Ohm-JS editor.

Delete the default grammar and the default test code from the editor.

Paste the following macro-grammar into the `grammar` window.

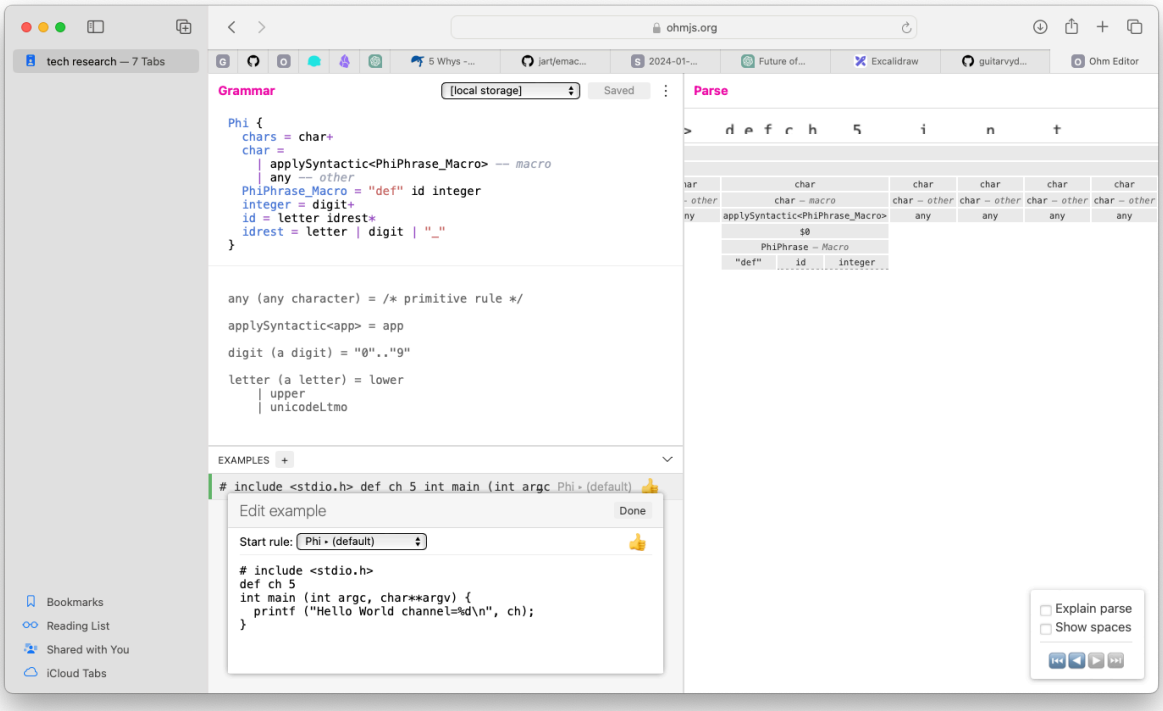
```
Phi {
  chars = char+
  char =
    | applySyntactic<PhiPhrase_Macro> -- macro
    | any -- other
  PhiPhrase_Macro = "def" id integer
  integer = digit+
  id = letter idrest*
  idrest = letter | digit | "_"
}
```

If there are any problems with the grammar, the editor will display an error message in red in the `grammar` window.

Fool with the grammar until there are no more errors and the bottom half displays - in gray - the builtin rules that OhmJS is using.

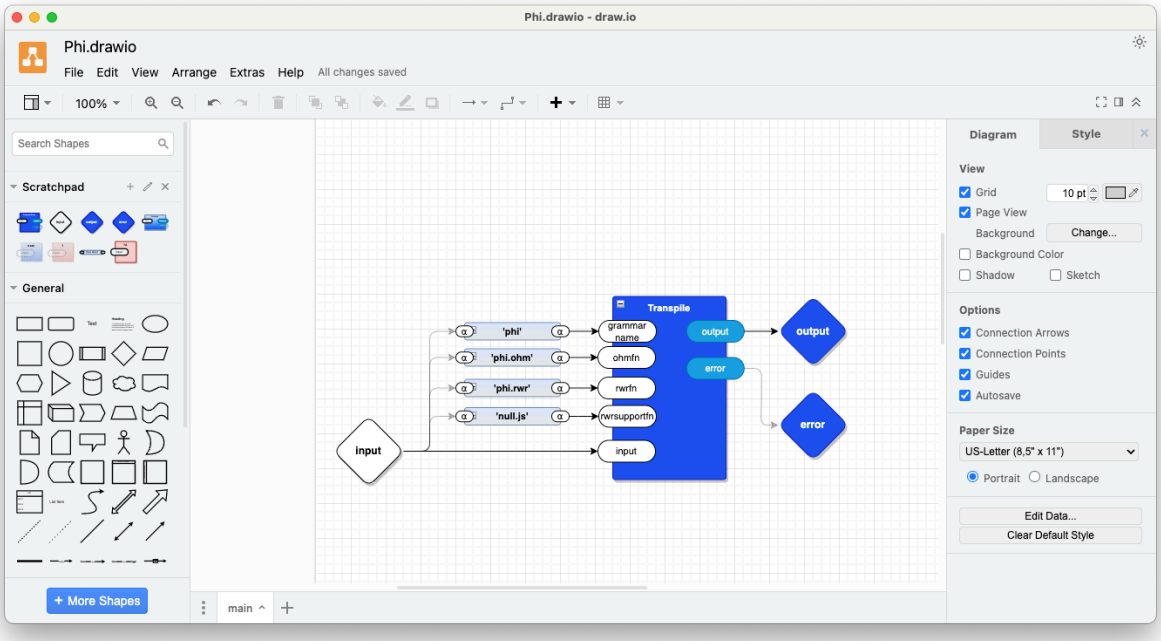
Now, paste the above `Before` code into a fresh `Example`.

After fooling around and fixing things, you should see a green bar on the left side of the example, meaning, it parses OK. Because this parses and accepts every character, you need to double-check the `Parse` window to see that the macro has, indeed, been recognized as a macro instead of just as a bunch of characters.



The above grammar has some flaws, but, is good enough for now.

RWR File
Source Code



Main.odin

Copy from https://github.com/guitarvydas/hamburger
Then modify:

```
package nextgen_parsing

import zd "0d/odin/0d"
import "0d/odin/std"

main :: proc() {
    main_container_name, diagram_names := std.parse_command_line_args ()
    palette := std.initialize_component_palette (diagram_names, components_to_include_in_project)
    std.run (&palette, main_container_name, diagram_names, start_function)
}

start_function :: proc (main_container : ^zd.Eh) {
    filename := zd.new_datum_string ("test.txt")
    msg := zd.make_message("input", filename, zd.make_cause (main_container, nil) )
    main_container.handler(main_container, msg)
}


components_to_include_in_project :: proc (leaves: ^[dynamic]zd.Leaf_Template) {
    zd.append_leaf (leaves, std.string_constant ("phi"))
    zd.append_leaf (leaves, std.string_constant ("phi.ohm"))
    zd.append_leaf (leaves, std.string_constant ("phi.rwr"))
    zd.append_leaf (leaves, std.string_constant ("null.js"))
}
```

Makefile
Copy and modify

```
LIBSRC=libsrc
ODIN_FLAGS ?= -debug -o:none
0D=0d/odin/0d/*.odin 0d/odin/std/*.odin
D2JDIR=0d/odin/das2json
D2J=$(D2JDIR)/das2json

dev: clean run

run: phi
	./phi main phi.drawio $(LIBSRC)/transpile.drawio

phi: $(D2J) phi.drawio
	$(D2J) phi.drawio
	$(D2J) $(LIBSRC)/transpile.drawio
	odin build . $(ODIN_FLAGS)

$(D2J):
	echo 'Please remake das2json'

$(0D):
	echo 'Please remake 0D'

clean:
	rm -rf phi phi.dSYM
```

null.js
Copy from https://github.com/guitarvydas/hamburger

ohmjs.js
TBD

Cheat by Copying Bits from ...
https://github.com/guitarvydas/hamburger

Macro-Grammars for All of Phi
TBD: delineate words
TBD: escape whitespace
TBD: ???