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 https://ohmjs.org/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.

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 {
```

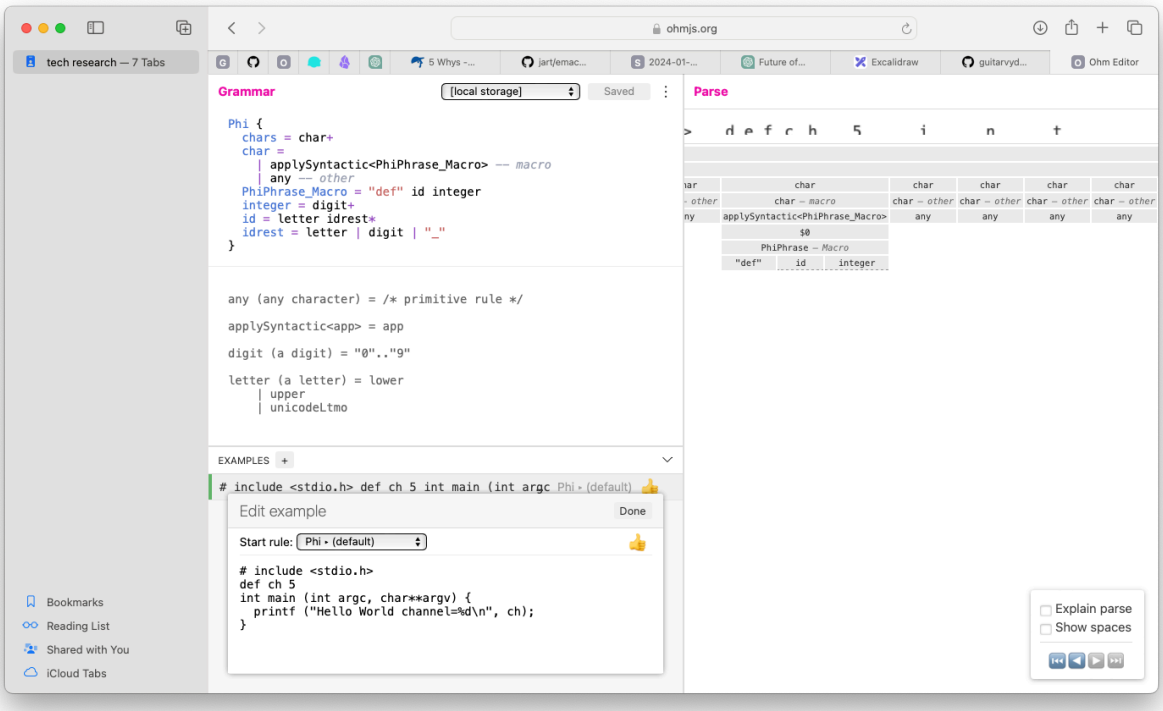
```
  chars = char+
  char =
    | applySyntactic<PhiPhrase_Macro> -- macro
    | any -- other
  PhiPhrase_Macro = "def" id integer
  integer = digit+
  id = letter idrest*
  idrest = letter | digit | "_"
}
```

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.



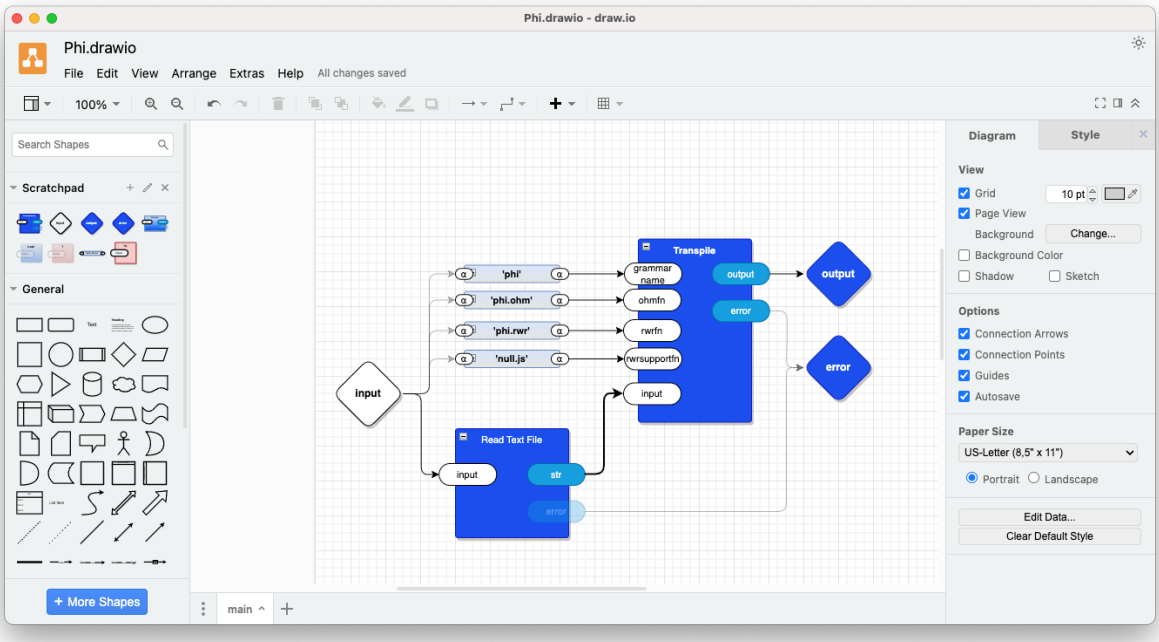
The above grammar has some flaws, but, is good enough for now.

RWR File

This .rwr file is for the C version. Modify it for Go.

```
phi {
  chars [char+] = '«char»'
  char_macro [phrase] = '«phrase»'
  char_other [c] = '«c»'
  PhiPhrase_Macro [_def id integer] = '\nint «id» = «integer»;\n'
  integer [digit+] = '«digit»'
  id [letter idrest*] = '«letter»«idrest»'
  idrest [c] = '«c»'
}
```

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

nextgen_parsing: $(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 nextgen_parsing nextgen_parsing.dSYM
```

null.js

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

test.txt

Create the file `test.txt` with the the contents of the before code from above:

C version, replace with Go version:

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

Cheat by Copying Bits from ...

https://github.com/guitarvydas/hamburger

What's Wrong With This Grammar?

The grammar is close, but can give bad results if the letters "def" appear anywhere else, like in a comment or a string. Due to space-skipping the grammar accepts and parses "defch5" the same as"def ch 5". So, if you ever create a variable with a strange name like "defx123", the grammar - as given - will macro-expand it instead of leaving it alone.

My solution to this problem is to preprocess all input with a low-level tokenizer that inserts Unicode brackets around all identifiers. The low-level tokenizer must be written using Lexical rules in OhmJS. It is probably possible to do all of this preprocessing in a single Ohm grammar, but, I find it easier to think about as a separate problem. Making it a completely separate component means that I can debug it, then put it away and never have to think about the problem again. Trying to write this all in one grammar causes little unexpected gotchas, i.e. when you make a change to single grammar, the change might cause the preprocessing step to act differently due to "hidden" dependencies between rules in the grammar. In my mind, it is better to spend time on bigger problems than spending time to figure out how to tweak the grammar to fold all parsing into one common grammar.

Furthermore, space-skipping will delete whitespace from comments.

Both of these problems can be easily fixed by creating small preprocessor components. I call them "delineate words" and "escape whitespace".

"Delineate words" pattern matches for identifiers and wraps Unicode brackets around each identifier. Components further down in the chain can be written as Syntactic rules (much cleaner for readability) and don't need to worry about the nuances of recognizing legal identifiers. This component might need to be tweaked for each input language to accommodate differences in definitions of legal identifiers. For example, Common Lisp accepts many more characters in identifiers than does Javascript. A lot of languages have similar rules for identifiers, hence, in many cases this component does not need to be customized. I use Common Lisp and Prolog in many projects. These languages have identifier naming rules that are not the same as in most modern languages.

"Escape whitespace" simply uses Javascript's "encodeURIComponent()" subroutine to turn all whitespace in strings and comments into printable ASCII. This encoded form is acceptable to a wide range of languages, even old ones based on ASCII such as Prolog, and, it is obviously an acceptable way to send strings in HTTP requests. One needs to remember to unencode the strings and comments in a final cleanup step at the end of the pipeline. This is fairly straight-forward and can be done using Javascript without any Ohm code.

A side-effect of using encoding, is that it becomes possible to insert whitespace into names of components. Name of Components on draw.io diagrams can contain whitespace. The 0D

engine sees only the encoded names and does not get confused. So, human-readable names can contain whitespace, whereas machine-readable names never include unprintable characters. Humans never need to look at the encoded versions (well, except for systems programmers who need to debug at a low level), so everyone is happy.

Comments

I don't - yet - explain the syntax for .rwr. You'll have to puzzle it out, if you get that far. I think that it is straight-forward. I use some Unicode characters in the rewrite strings, so that we don't need to use wierd escape strategies for most ASCII characters. I use a different quote for the front of strings than for the back of strings. That helps preserve my sanity and reduces the need for esoteric escape sequences.

String interpolation is done by transpiling the inside of Unicode double-angle brackets into Javascript. Again different characters for front and back. PEG is easier when you use different characters for front and back.

The RWR transpiler emits tree-walking code based on the LHS (Left Hand Side) parameter lists (square brackets). On the RHS there must be only one rewrite string that contains raw ASCII characters or string interpolation sequences, e.g. '«c»'. The stuff inside of RHS brackets is pure Javascript. This - usually - consists of a reference to a Javascript variable name (like "c" in '«c»') but can actually contain any Javascript code. The Javascript variable contains the tree-walked value of the the like-named rule parameter. In more rare cases, I write calls to Javascript functions in the "_" namespace. The "_" functions are defined in the support code passed into the transpiler on the 'rwrsupportfn' pin of Transpiler. Most often, this isn't used and I simply pass "null.js" into the 'rwrsupportfn' pin. [Note that my convention is to use the suffix 'fn' to mean "filename". In general, it is expected to pour source code directly into pins. Filenames are used only in these edge cases where 0D code meets non-0D code. I doubt that developers will see too many of such edge cases, in the long run.]

***Aside**: It would seem that what I'm doing is patently unsafe - I simply pass Javascript from .rwr into node.js without doing any serious error checking. I've been using this stuff for several years now and haven't suffered much from this decision. Users don't get to see this stuff, only developers use it. It appears that developers don't need as much hand-holding as we would provide to users. My conclusion is that DX doesn't need to be as bullet-proof as UX, or, that DX issues are quite different from UX issues. Developers are smart enough to use dangerous tools like this without blowing their own feet off, and, when they do hit problems, they can figure out the cause. This trade-off meant that I could whip up the RWR DSL in a short amount of time - by cheating and avoiding work - and use it to do interesting kinds of programming. Instead of taking time to make RWR bullet-proof, I used the left-over time to do other kinds of things. This is probably a variant of MVI style thinking instead of MVP style thinking. Or, it may be a manifestation of the anti-waterfall approach - I assume that I don't really know what's best and don't waste time fixing things that I don't know need to be fixed and tightened up. I assume that, if we encounter severe problems, the fix will be obvious - later, not now. Or, maybe this is just my mindset. In my mind, I'm not using an off-the-shelf tool, I'm writing Javascript. RWR just makes the job easier, kind of like having shortcut keys in my text editor. The RWR transpiler generates all of the boiler-plate for me, and I just fill in the blanks where it can't easily infer what I want. The shortcut keys in my editor aren't type-checked. Where is the tipping point? This sounds like an interesting avenue for further research. How do you measure effectiveness? How do you compare time spent on bullet-proofing vs. time spent making progress on other fronts?*

The End (Part 1)

Are we disappointed yet?

We're done.

What did we see?

---

1. How to create a macro

We built a simple - very simple - macro preprocessor for a textual language.

A larger macro processor doesn't need to be much more than what we saw here, but, just more of the same.

---

2. How to use a DPL

We used a simple - very simple - Diagrammatic Programming Language to create code that can be executed by a computer.

Instead of needing to use a text editor, like *VSCode* or *Emacs* or ..., we used *draw.io*, to edit source code.

The diagrammatic programming language is a VPL (Visual Programming Language), except that, instead of compiling Rembrandt masterworks to code, we used a language composed only of a few symbols
- rectangles
- Rhombuses
- Ellipses
- Arrows
- Grouping (called "scoping" in TPL (Textual Programming Languages))
- Drawing more than one diagram, but signifying one diagram as the top-most one
  - in this case, we called it "main", but this can be changed

We saw two (2) kinds of *Components*
1. Containers
   - Drawings that contain other drawings
   - Nesting, "scoping", relativity
   - Contains other *Components* (other Containers, and/or other Leaves)
2. Leaves
   - Code
     - In this case, Leaf code is written in the Odin and Javascript languages
     - The DPL does not *define* ways to write code, you just write code in some pre-existing language (like Odin, /bin/bash, Javascript) and put the code in text boxes (or external files accessible from the DPL).
   - Leaf code must be explicitly declared in *main.odin.*
     - This can probably be improved and automated sometime in the future.
   - A shorthand for creating constant strings `std.string_constant(...)`
   - A shorthand for shelling out to the shell (`$ ...`)

- Input and output ports
  - Rhombuses,
    - Top level ports to/from the diagram
  - Rounded pills
    - Ports of children components

Using the tools that exist, i.e. text parsing tools, makes it easy to build DPLs based on diagrams that are saved in XML format.

A difference between the DPL we used here and traditional TPLs (Textual Programming Languages), is that our symbols (glyphs) are vector-based, instead of being based on small bitmaps (called "characters"). Vector-based symbols can be resized and skewed and transformed, yet still be recognizable as being the same symbols, whereas TPLs use absolute binary codes, e.g. ASCII, to represent symbols. The binary codes cannot be transformed without losing their meaning. In 1950, computer hardware could only handle absolute binary codes, but, in 2024, computer hardware can easily handle vector-based data.

We saw that *leaf components* cannot refer directly to other components, leaf or container. Arrows on the diagrams "belong" to the parent components, not to the children components. Container components cannot refer directly to their peers, but can reference *only* their direct children components (leaf or container).

Arrows on the diagram are one-way message paths. They are drawn explicitly on the diagrams and cannot change at runtime, i.e. self-modifying code is prohibited, dynamic programs are prohibited. Note that this is opposite to the commonly-held belief in concepts such as publish/subscribe and *becomes* operations on classes.

It turns out that bi-directional - two-way - message paths are simply an optimization of one-way arrows. It requires more code to create bi-directional arrows, and, using bi-directional arrows tends to obfuscate DI - Design Intent.

Arrows primarily show data-flow, but, event arrival on input ports causes a very primitive form of control flow. Event arrival triggers event-handling in components, but, the actual timing of event arrival is arbitrary and determined by the system event dispatcher.

Message passing is like GOTO. Message passing is an extremely useful low level concept that needs to be further structured to allow programming and composition of larger systems. [*see my other essays about Structured Message Passing for suggestions*].

Components are completely *asynchronous*. They cannot rely on message ordering. In textual programming languages, it is understood that lines of code are executed in top-to-bottom, left-to-right order. This kind of implicit ordering is not guaranteed by our DPL. Components are completely isolated from one another.

The only guarantees provided by our DPL are
- that a single output event from a component will be delivered atomically to all receivers, i.e. events cannot interleave, if A sees *X* followed by *Y*, B cannot see *Y* followed by *X*
- *t*hat containers are busy and cannot consume another input event, if any child is busy.

Note that messages can be queued up (in order of arrival) for later processing, if a component is busy.

A component can react to only one input event at a time. Leaf components simply react to one incoming event and produce a response(s). Container components, though, pass input messages down to their children for processing. Container components cannot process another input message unless all of its children are quiescent.

A component reacts to a single input event by processing the input data, then, generating zero (0) or more output events. Note that this is contrary to how functions work in most programming languages. A component is not required to produce any output for a given input event, but, when a component does produce outputs, the outputs are not required to go back to the sender. Components simply generate responses (or not) and drop the responses, as events, onto their own output queues. Their parents decide where the output messages should go, children components have no say in this matter.

This example shows that main.odin is compose of three (3) main sections. Production Engineers might be able to reduce the number of sections.
1. Main
2. start_function
3. components_to_include_in_project

- ~~vector-based, not bitmap~~
- ~~Basics~~
  - ~~Ownership of arrows and routing~~
- ~~Arrows~~
  - ~~One-way message paths~~
    - ~~Making bi-directional paths is an (visual) optimization~~
    - ~~Making bi-directional paths requires more code~~
  - ~~Control flow - data flow~~
  - ~~A container is busy if any child is busy, recursively~~
  - ~~asynch~~
  - ~~Event atomicity~~
- main.odin, 3 sections
- feedback
  - Not shown
  - Not the same as recursion, due to use of ordered queues instead of stacks.
- not class-based, inheritance breaks the rule of locality-of-reference

---

4. All of FP in a Nutshell

FP is Functional Programming.

The goal of FP is to allow “referential transparency”, i.e. replacing textual code by other textual code. This is what *macros* do. In fact, Microsoft Word can do this, except that its pattern matching is not nestable (as are REGEXs and CFGs).

All of the “rules” of FP - e.g. “no side effects” - are put in place to enable “referential transparency”.

Hygienic macros come for “free” when the macros are done in a preprocessing manner, without conflating the language used for macros with the language used for programming. In the past, the issues and arguments about hygienic macros were caused by the insistence of forcing macros into the same programming language as is used for programming, e.g. Lisp and Scheme.

Compilers are text-to-text transpilers. They inhale high-level source code and exhale low-level assembler source code. Some compilers go an extra step and transpile the low-level assembler source code to even lower-level binary code. Some compilers go even further by transpiling high-level source code directly into lower-level binary code.

When you apply the concepts of Functional Programming rigidly, you get very small languages, like Sector Lisp. Code bloat in other languages (Python, Rust, etc.) comes from forcing mutation into the same programming language that is functional at heart. As soon as you need to have a heap, the language contains many more edge cases and you get unintended code bloat.

---

5. Efficiency

The code, by O(N) standards, is highly inefficient.

Why do we care, though? It runs fast enough on our development machine.

There is a difference between production-quality “efficiency” and developer-level “efficiency”. These two concerns have two completely different goals, that cannot be unioned together to make a “one language to rule them all”. An effort union-together all concerns invariably results in a lowest-common-denominator approach - mediocre at best.

- Production-Level Efficiency
  - Reducing final cost is the main goal.
  - “Engineering” consists of “Production Engineering”.
    - Refactoring designs to reduce hardware costs.
    - Refactoring designs to reduce memory footprint.
    - Assumption that spending more development effort results in lower amortized cost of final product.
- Development Efficiency
  - Save development time.
  - Assume that expensive, fast, big, computers are used by developers.
  - “Engineering” consists of building an MVI - Minimum Viable Implementation - by trading off efficiency for speed of development.

---

6. OhmJS

Grammar, pattern-matching specification for building parsers.

Inhale. Grok the input text (code).

---

7. RWR

ReWrite specification.

Exhale - write out text (code) based on what was matched.

6. Type Checking

Not needed when you keep things small and simple.

Type checking is only of interest to developers.

Users don’t care if the developers used type checking or wrote raw assembler, as long as the result is robust. The use of CD - Continuous Delivery - tools implies that the result is far from robust.

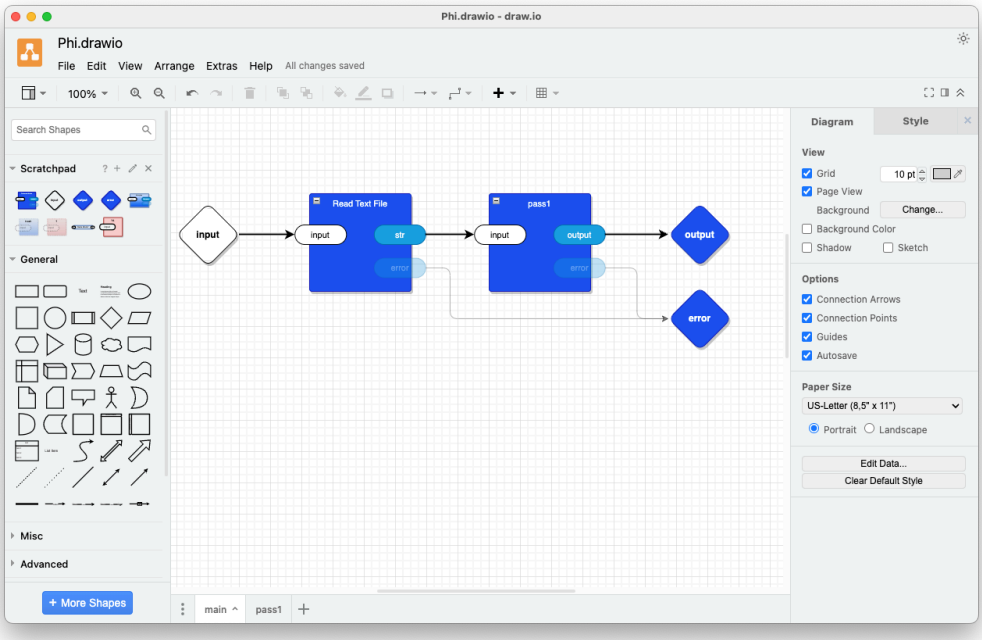
Type checking is needed by developers only when they can’t see all of the code. Hence, the goal should be to keep things small and simple and nested, instead of adding more elaborate type checking to allow larger code flat bases.

What’s Next? Syntax-Based Translation.

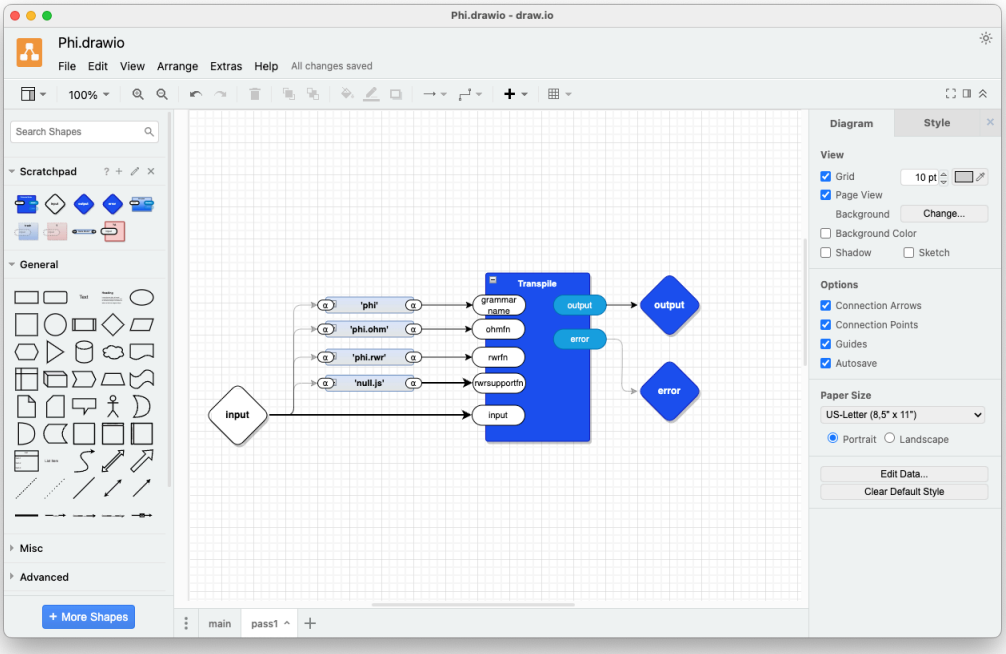
Adding Preprocessing Before Pass1

Step 1

1. Right-click on the tab “main”, and select “duplicate”.
2. Check that you have 2 tabs, double-click on the “Copy of main” tab and rename it to “pass1”
3. Go back to the main tab and revamp it to look like this...



And make the other tab, “pass1” looks like:



Rebuild.

```
$ make.
```

The result should be the same.

Explanation: the 0D compiler compiles all of the diagrams in the `.drawio` file. Then, it looks for a diagram called "main". When found, it instantiates "main". During the instantiation, "main" refers to a Component called "pass1". That component must be a Leaf or a Container. In this case it's a Container, defined in the diagram "pass1". That diagram was compiled and stuffed into the "database" under the name "pass1". Diving deeper... leaves are lumps of Odin code (in this version) and are defined in the Odin proc called "`components_to_include_in_project (...)`" in the file `main.odin`. The Container named "`Transpile`" is in `libsrc/`. The `Makefile` ensures that it is included in the project. The "`Transpile`" Component accepts 5 inputs (grammar name, ohm filename, rwr filename, rwr support (JS) filename, and a string to be transpiled) and produces *one* output - either a transpiled string, or, an error. It only produces one of those outputs at a time, keeping the other output output silent (not *nil*, not anything, just nothing at all).

If you wish, you can keep digging deeper and you will find that "Transpile" invokes a piece of Javascript called "`ohmjs.js`". As you might guess, this is where ohm-js is invoked. Twice. Once to read the `.rwr` spec and once more to parse and transpile the input string. As it stands, we simply pass big strings between the passes. This can be optimized, if necessary. You could re-write this stuff to be "more efficient" by rewriting "`Transpile`" in Javascript, then rewriting "`main`" in Javascript and so on. But, why bother? If it runs "fast enough" on your development system, why would you bother making it run a few nano-seconds faster? Save your neurons for something more important.

Basically, "main" gets a filename on its input, reads the file into a string, then passes the string into "pass1".

Step 2

Add a Component called "pass2" on yet another tab in draw.io, maybe duplicating the "pass1" tab. Modify the "main" diagram to show the output of "pass1" being fed input "pass2".

Modify the `.rwr` of `pass1` and create a new grammar for `pass2` that parses the output of `pass1`. `Pass1` should not emit valid C code, but should insert some sort of annotation into the code. `Pass2` should accept the annotated code and produce valid C code.

... *TBD ...*

Macro-Grammars for All of Phi

TBD: delineate words - to be supplied later - I need to drag the Components in from other projects
TBD: escape whitespace - to be supplied later - I need to drag the Components in from other projects
TBD: ???