Working paper on normalizing all access to all data.

The purpose of normalization is to make it easier to automate generation of programs - to write program generators ("compilers" reimagined).

Data Descriptors

Data Descriptors were designed to normalize all access to all data to help with the gargantuan task of building software apps called "compilers".

As defined, DDs have the built-in assumption that all spaces are arrays of bytes.

DDs led to work on portable compiler technology such as the Orthogonal Code Generator .

That work was preceded by RTL which was used in the massively popular app called *gcc*.

What's Next?

Compilers have become a "solved problem".

Today, we want to build "transpilers" (source-to-source transformers) and we want to explore syntax elements other than textual characters.

We want to use lessons learned from the past

- normalize everything to make it possible to automate code generation
- use Orthogonal Code techniques for designing new languages.

Orthogonal Code Techniques

Orthogonal Code Generations subdivides code up into two broad categories:

- operands (AKA "data")
- operators (AKA "control flow").

Eschewing Control Flow

It is - currently - in vogue to treat "control flow" as being the same as "data flow".

This model of computation is useful for building complex calculators and has resulted in a flurry of languages of the Functional Programming ilk.

Electronic machines - AKA "computers" - are capable of doing more work than that required by mere calculators.

History - State

For example, "computers" are capable of sequencing events and machines. Sequencing requires the notion of *history*, and, *state*.

History and *state* is expressly ignored by Functional Programming notations.

Such abstraction leads to useful results, but, discourages results in other dimensions such as sequencing.

When all you have is functions, everything looks like a function.

Syntax is Control Flow

Control Flow is expressed by syntax.

(Data is expressed by OOP).

Up until now, we have been hampered by the fact that *syntax* has been difficult to deal with and to express.

Now, we have easy access to syntax and so-called "parser generators" in the form of Ohm-JS.

We can re-examine and re-express the relationships between *operands* and *operators*.

We already have OOP for expressing *operands*.

Now, we have Ohm-JS for expressing *operators*.

Now, we can do both, express data-flow *and* express control-flow - easily.

In designing transpilers,
SDs treat spaces like ordered lists, can be optimized to be mapped onto arrays of bytes

Where Can Operands Live?

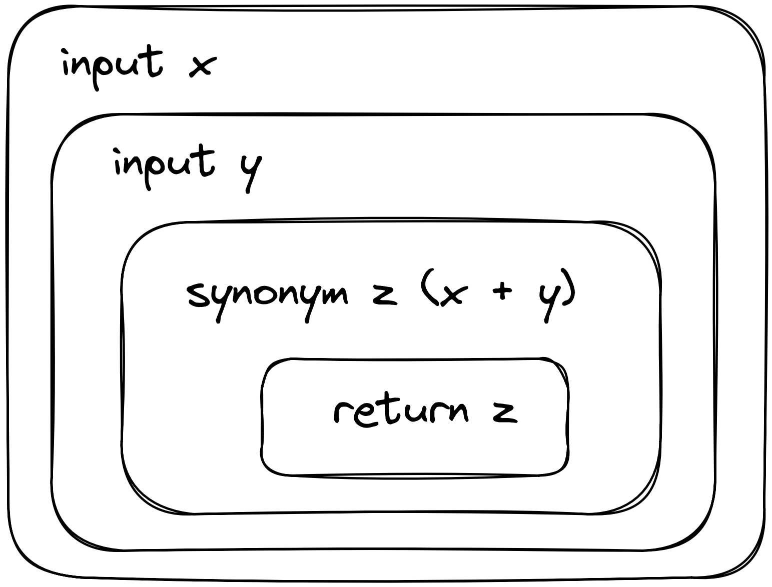
Imagine this bit of code:

```
(lambda (x)
  (lambda (y)
    (let ((z (add x y)))
      z)))
```

In JavaScript syntax, this might be written as:

```
function (x) {
  return function (y) {
    var z = x + y;
    return z;
    } (x);
}
```

As a drawing, we might write:



The variable `z` is an operand. It represents a temporary variable inside the inner-most expression. `z` is a *synonym* for the expression `x + y`.

The variable `y` is an operand. It is a parameter to the inner anonymous function (`lambda (y)`). The value of `y` is determined only at runtime.

The variable `x` is an operand. It is a parameter to the outer anonymous function (`lambda (x)`). The value of `x` is determined only at runtime.

The result `z` (the 4th line is an implicit `return` statement) assigns the value of the variable `z` to the return-variable of the complete expression. We don't bother to give a name to the return-variable for the expression.

- parameters
- temps
- actuals
- results

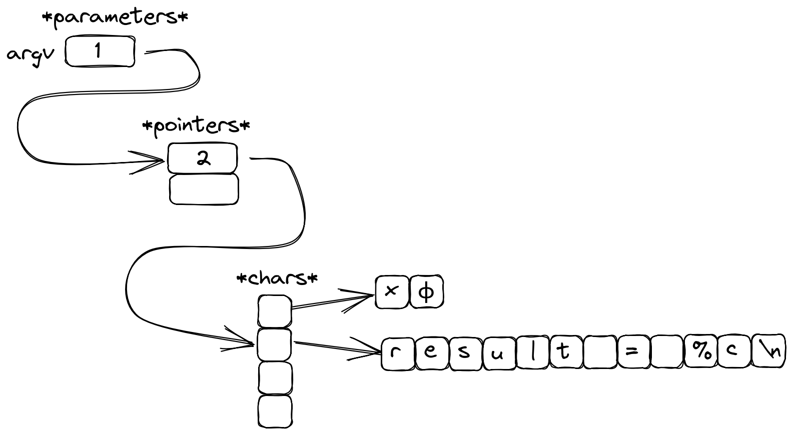
Not in the above, but, to be considered:

- pointers
- free, global
- messages (asynchronous)

Pointers

char *s = "result = %c\n";

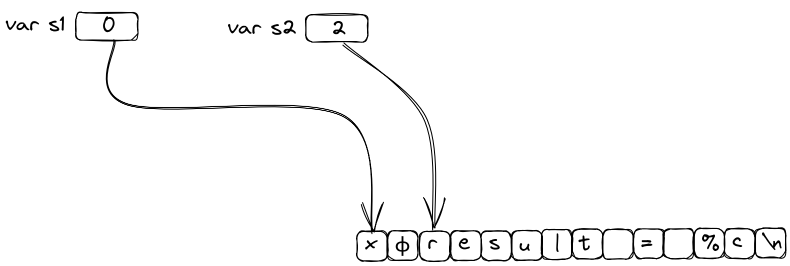
... f (... char **argv ...) { ... }



C Everything Is A Byte

c everything is a byte

c string



Operand Descriptors