The No Spoon Series : Building Slang

Srikumar K. S.

21 Feb 2017 - 26 Apr 2017

Contents

1 Introduction

Today, thanks to the wide availability of sharing and publication tools, software engineers are being bombarded by new technology at a pace that many are finding hard to keep up with. This is even if they've already had the clarity to choose a narrow field to focus on, such as "front end engineering", for which the pace is particularly unsettling. To top it, engineers enter the field armed with a reasonable grasp of one major programming language that they can use on their job. Once settled thus, I've seen many get stuck in the "idiomatic" ways of thinking within their accidentally chosen toolsets, unable to raise their head above the waves of technologies sweeping past them.

Redeeming ourselves from this frog-in-the-well situation is not that hard as the metaphor might suggest is the case in the physical world. What we need is a focus on "what exactly is the new idea here?". If there is any, dive in. If there is none, move on. This series is a spontaneous creation to address this educational issue for software engineers to show them the nature of the matrix and how to escape from it using the tools they're already familiar with.

I use a sane and simple subset of Javascript here, but it can be any main stream language and the concepts should be easy enough to translate. One reason for choosing Javascript is that JS programmers seem to be the ones struggling the most to keep up with all the "trendy stuff". As we work within this JS subset, you'll see how quickly we can rise up conceptually to what has taken years to come down through the EcmaScript standardization pipeline. You'll also see how some "modern" concepts are really decades old as we go on.

The series is titled after the "The Matrix" trilogy. Not only will you learn how to bend spoons, I hope you'll see that there is really no spoon.

> **Disclaimer**: This is a work-in-progress writing that is almost written by flow of thought with a little non-linear editing thrown in. I do not guarantee that the existing structure or code will remain as is. However, I will strive to have it all be self contained and comprehensive to the extent I can. Also, this series is **not** about data structures and algorithms, but is about system building and modeling "how to knowledge".

Hope you have fun exploring the rabbit hole!

-Srikumar

1.1 Primary reference material

There are two primary textbooks relevant to this series. They are -

1. The Structure and Interpretation of Computer Programs by Hal Abelson and Gerald Sussman.
2. Concepts, Techniques and Models of Computer Programming by Peter Van Roy and Seif Haridi

Other references will be mentioned as we go along.

2 Building Slang

In this series, we will gradually work through implementing an interpreter for a small programming language in Javascript. The goal is to understand language "features" whose needs arise at various points and the how various languages trade off these features in their implementations. We aim to understand them so that we can invent the most appropriate tool for the job at hand in any occasion instead of boxing ourselves into the limitations of a design made for another context. We'll be working towards a pretty powerful feature set, but efficiency will not be one of our initial concerns. The goal will be to understand how to think out of the box no matter what programming language or toolkit you're working with.

> **Note**: This file is written in a "stream of thought" style so you can follow along from top to bottom as we add more detail. You can pretty much draw a line at any point in the code, copy

> paste everything before that line into a javascript console and run it. I say "pretty much" because some concepts and implementations will take a few steps to flesh out, so "pretty much" means "as long as everything you need has been defined".
>
> -Srikumar

We use ES6 features like `let`.

```
"use strict";
```

2.1 Application area

While working through this, we'll keep in mind a simple application of the language we'll be building - a drawing tool that works within web browsers. As we do more advanced stuff, we'll be going beyond this application area, but those transition points will be well defined.

2.2 Approach

Normally, when learning a programming language, we learn first about its *syntax* - which is how the language looks, then delve a little deeper into its *semantics* - which is how the language works and behaves and later on work out a *mental model* for the language so we can understand programs in it and use the language syntax and semantics for effective system design.

In this series, we'll be going the other way around. We won't bother ourselves with pretty syntax, but we'll start with a mental model of programming and work out a viable semantics for a language and only then slap a syntax on it purely for convenience.

2.3 The mental model

Program: Our program will be .. a sequence of instructions we give our interpreter to perform. Simple eh? We read instructions one by one and "execute" them. This will be how our "interpreter" will work.

Environment: The instructions our program runs will do their job within the context of an environment. Making this environment explicit is very useful to making multiple runtimes co-exist by creating different environments for them.

Stack: Our programs will operate on a *stack* of values. Programs will have immediate access to the top elements of the stack, but will have to pop out elements in order to look any deeper.

2.4 Values

We break down our language into the values that our programs work with and choose basic data structures for representing our program as well as the stack that they operate on.

So what values will our program need to deal with? If you think about drawing applications, at the minimum we need numbers to represent coordinates and strings to include text. We're also likely to encounter repeated patterns. So we need some ways of giving names to these patterns so we can reuse them. We'll also have some operations that we initially won't be able to perform in our language and will need to dig into the "host language" - in this case Javascript - to perform. We'll call these "primitives".

We'll represent values of these types using a simple Javascript Object with two fields `t` for "type" and `v` for "value".

We can use these functions to make values that our programs can consume. By using these functions, we're guaranteeing that we'll supply proper argument types so our programs can, for example, trust that the `v` field will be a number if the `t` field has the value `"number"`.

```
let number = function (v)  { return {t: 'number', v: v};  },
    string = function (v)  { return {t: 'string', v: v};  },
    word   = function (v)  { return {t: 'word',   v: v};  },
    prim   = function (fn) { return {t: 'prim',   v: fn}; };
```

If you look carefully into what we've done here, we've already committed to something pretty big! These are the entities using which we'll be expressing the values that our programs will operate on. They are also the entities using which we'll express our programs! Though we'll be expanding on this set, we'll try and preserve this symmetry as far down the line as makes sense for our purpose.

2.5 Running a program

Recall that we said our program is a sequence of instructions we process one by one. We can represent our program therefore using a plain old Javascript array, along with a "program counter" which is an index into the array of instructions to execute next. The stack that our program needs to work on can also be represented by an array.

```
let run = function (env, program, pc, stack) {
```

So how do we run our program? It is just as the Red Queen said to the White Rabbit in Alice in Wonderland - "Start at the beginning, go on until you reach the end, then stop."

```
    for (; pc < program.length; ++pc) {
        let instr = program[pc];
```

When an instruction is a "word", we need to use it as a key to lookup a value in our environment. Once we look it up, we have to treat it as though this value occurred in our program as a literal, which means treating it as an instruction and processing it.

```
        if (instr.t === 'word') {
            let deref = lookup(env, instr);
            if (!deref) {
                console.error('Undefined word "' + instr.v + '" at instruction ' + pc);
            }
            instr = deref;
        }

        switch (instr.t) {
```

When we encounter a primitive operation given as a Javascript function, we have to pass it our stack so that it can do whatever it needs to do with the values stored on the stack.

```
            case 'prim':
                stack = apply(env, instr, stack);
                break;
```

In all other cases we just store the value on the stack.

```
            default:
                push(stack, instr);
                break;
        }
    }

    return stack;
};
```

... and that's it for our first interpreter!

2.6 Looking up words and performing primitive operations

Since our simple idea of an environment is as a key-value lookup table, we use a plain Javascript object as our environment. We'll capture this assumption in a function to create a new environment from scratch.

```
let mk_env = function () {
    return {}; // A new hash map for key-value associations.
};
```

With such an "environment", we get a simple lookup function -

```
let lookup = function (env, word) {
    console.assert(word.t === 'word');
    return env[word.v];
};
```

Associate the value with the given key and returns the environment.

```
let define = function (env, key, value) {
    env[key] = value;
    return env;
};
```

For generality, we can model primitive operations as functions on our stack.

```
let apply = function (env, prim, stack) {
    console.assert(prim.t === 'prim');
    return prim.v(env, stack);
};
```

Question: What limitations does this definition impose on what a primitive function can do?

We'll also abstract the stack operations to keep things flexible.

```
let push = function (stack, item) {
    stack.push(item);
    return stack;
};

let pop = function (stack) {
    return stack.pop();
};

let topitem = function (stack) {
    return stack[stack.length - 1];
};
```

It is useful to look a little deeper into the stack. So we add another function to peek deeper than the topmost element.

```
let topi = function (stack, i) {
    return stack[stack.length - 1 - i];
};

let depth = function (stack) {
    return stack.length;
};
```

For simplicity, we assume that our primitives do not throw exceptions. In fact, we will not bother with exceptions at all. Forget that they were even mentioned here!

2.6.1 Testing our mini language

Let's quickly write a few functions to express how we intend to run our programs and what we'll expect of them.

We'll hold all our tests in a single hash table mapping the test name to the test function to be called.

```
let tests = {};
```

The smoke_test function should produce a stack with a single item on it - the number 3.

```
tests.smoke = function () {
```

We start with an empty environment, load our standard library of routines into it and use it to run our "program" that adds 1 and 2 and returns the stack with the result.

```
    let env = load_stdlib(mk_env());

    let program = [
        number(1),      // Push 1 on to the stack
        number(2),      // Push 2 on to the stack
        word('+')       // Apply '+' operation on top two elements.
    ];

    return run(env, program, 0, []);
};
```

2.6.2 Displaying the stack for debugging

A helper function to show the top n elements of the stack on the console. The count defaults to 20.

```
let show = function (stack, n) {
    n = Math.min(n || 20, depth(stack)); // Default to 20 elements.

    for (let i = 0; i < n; ++i) {
        show_item(topi(stack, i));
    }
};

let show_item = function (item) {
    switch (item.t) {
        case 'string':
```

We need to properly escape characters, so we use stringify only for strings.

```
            return console.log('string(' + JSON.stringify(item.v) + ')');
        default:
```

Everything else, we let the default display routine take over.

```
            return console.log(item.t + '(' + item.v + ')');
    }
};
```

2.7 Standard library

We'll choose a very basic standard library consisting of 4 arithmetic operations to start with. We'll expand this set, but we're too impatient to get to try out our new fangled "language" that we're willing to wait for that coolness.

```
let load_stdlib = function (env) {
```

Basic arithmetic operators for starters. Note the order in which the arguments are retrieved.

```
    define(env, '+', prim(function (env, stack) {
        let y = pop(stack), x = pop(stack);
        return push(stack, number(x.v + y.v));
    }));

    define(env, '-', prim(function (env, stack) {
        let y = pop(stack), x = pop(stack);
        return push(stack, number(x.v - y.v));
    }));

    define(env, '*', prim(function (env, stack) {
        let y = pop(stack), x = pop(stack);
        return push(stack, number(x.v * y.v));
    }));

    define(env, '/', prim(function (env, stack) {
        let y = pop(stack), x = pop(stack);
        return push(stack, number(x.v / y.v));
    }));

    return env;
};
```

2.7.1 Test distance calculation

To calculate the distance between two points on a 2D plane, we need a new primitive - the square root function.

First, a small utility to add new definitions to our `load_stdlib` function. `new_defns` is expected to be a function that takes an environment, defines some things into it and returns the environment.

```
let stddefs = function (new_defns) {
    load_stdlib = (function (load_first) {
        return function (env) {
            return new_defns(load_first(env)) || env;
        };
    }(load_stdlib));
};
```

Augment our "standard library" with a new 'sqrt' primitive function.

```
stddefs(function (env) {
```

We'll not be able to express x * x without the ability to duplicate the top value on the stack. We could also add 'pow' as a primitive for that specific case.

```
    define(env, 'dup', prim(function (env, stack) {
        return push(stack, topitem(stack));
    }));
```

Similarly, we could use 'drop' also to take off elements from the stack without doing anything with them.

```
    define(env, 'drop', prim(function (env, stack) {
        pop(stack);
        return stack;
    }));

    define(env, 'sqrt', prim(function (env, stack) {
        let x = pop(stack);
        return push(stack, number(Math.sqrt(x.v)));
    }));
});
```

We always want the standard library for tests, so simplify it with a function.

```
let test_env = function () {
    return load_stdlib(mk_env());
};
```

Now we can finally calculate the distance between two points.

```
tests.distance = function (x1, y1, x2, y2) {
    let program = [
        number(x1),     // Store x1
        number(x2),     // Store x2
        word('-'),      // Take the difference
        word('dup'),    // 'dup' followed by '*' will square it.
        word('*'),
        number(y1),     // Store y1
        number(y2),     // Store y2
        word('-'),      // Take the difference
        word('dup'),    // 'dup' followed by '*' will square it.
        word('*'),
        word('+'),      // Sum of the two squares.
        word('sqrt')    // The square root of that.
    ];

    return run(test_env(), program, 0, []);
};
```

3 Developing Abstractions

We now have a simple language "slang" in which we can give a sequence of instructions to be "performed" by the interpreter and it will faithfully run through and do it one by one.

This is great for a basic calculator, but for anything with a bit of complexity, this is inadequate since we **want** to be able to build higher levels of abstraction as we work with on more complicated problems.

Let's recall the distance calculation program we wrote earlier -

```
let program = [
number(x1),     // Store x1
```

```
number(x2),     // Store x2
word('-'),      // Take the difference
word('dup'),    // 'dup' followed by '*' will square it.
word('*'),
number(y1),     // Store y1
number(y2),     // Store y2
word('-'),      // Take the difference
word('dup'),    // 'dup' followed by '*' will square it.
word('*'),
word('+'),      // Sum of the two squares.
word('sqrt')    // The square root of that.
];
```

This is cheating a bit actually. We're inserting values directly into a program as though they were constants. If we were to truly express this as a reusable function, then we'll need to make it operate on the top 4 values on the stack - `[y2, x2, y1, x1, ...]` and produce the distance as a result on the stack.

Since our environment can hold key-value associations, we can add a primitive that will insert these mappings into the current environment.

We need two things for that - a) a way to specify a "symbol" and b) a primitive to assign the symbol to a value in the current environment.

We introduce a new value type called 'symbol' which is similar to a 'word', except that it will always refer to itself when "evaluated" - i.e. if you ask our interpreter the value of a symbol, it will simply return the symbol itself, unlike a word.

```
let symbol = function (name) { return {t: 'symbol', v: name}; };
```

At this point, our interpreter only has special dealings for 'word' and 'prim' types. Everything else gets pushed on to the stack. So the interpreter already knows how to work with symbols. What we need is a way to use a symbol name and bind it to a value. For this, we add a new primitive to our standard library called 'def'.

```
stddefs(function (env) {
    define(env, 'def', prim(function (env, stack) {
        let sym = pop(stack), val = pop(stack);
        console.assert(sym.t === 'symbol');
        define(env, sym.v, val);
        return stack;
    }));
});
```

> That seems fine and will also work fine in basic tests. However, there is a subtlety here. The way we've implemented the `def` primitive, the value assignment will *always* happen in the top level environment! For the moment, that is ok. We'll improve this soon as we realize that programs become hard to reason about if references are implemented globally all the time. This is the same as the "no global variables" rule most of you follow.

With symbols and `def`, we can now implement a proper distance calculator. The new tests.distance will need to be called like this - `tests.distance2([2,4,6,7].map(number))`

```
tests.distance2 = function (stack) {
    let program = [
```

We name our four arguments and consume them from the stack.

```
        symbol('y2'), word('def'),
        symbol('x2'), word('def'),
```

```
        symbol('y1'), word('def'),
        symbol('x1'), word('def'),
```

Calc square of x1 - x2

```
        word('x1'), word('x2'),
        word('-'), word('dup'), word('*'),
```

Calc square of y1 - y2

```
        word('y1'), word('y2'),
        word('-'), word('dup'), word('*'),
```

Sum and sqroot.

```
        word('+'),
        word('sqrt')
    ];

    return run(test_env(), program, 0, stack);
};
```

Now you see that the entire program within `tests.distance` is a **constant** array, unlike what it was before when parts of the program depended on the arguments to the distance function. We can actually store this into disk and load it and use it at will.

3.1 Blocks

We see a sequence occuring twice exactly as is -

```
let program = [
...
word('-'), word('dup'), word('*'),
...
word('-'), word('dup'), word('*')
...
];
```

Any time you see this specific sequence, you know that the top two elements of the stack will be differenced and squared. In other words, this sequence behaves like the following function - a primitive, if we were to implement in javascript -

```
let diffsq = prim(function (env, stack) {
let x2 = pop(stack), x1 = pop(stack);
let dx = x1.v - x2.v;
push(stack, number(dx * dx));
});
```

We could replace those three sequences with `diffsq` and nobody would notice anything different .. except maybe the processor which would have to do more work with our interpreter.

It therefore seems pertinent to introduce a block of instructions as a type of thing we can store on the stack and assign to symbols in an environment, so we can reuse such blocks whenever we need them instead of having to copy paste the code like we've done here.

We have to introduce a new value type called 'block' for this purpose, which holds a program to jump into part-way when the interpreter encounters it. We also have to introduce a primitive for "performing" a block encountered on a stack. We'll call this primitive `do`.

```
let block = function (program) { return {t: 'block', v: program}; };

stddefs(function (env) {
    define(env, 'do', prim(function (env, stack) {
        let program = pop(stack);
```

If we've been given a primitive, we call it directly.

```
        if (program.t === 'prim') {
            return program.v(env, stack);
        }

        console.assert(program.t === 'block');
        return run(env, program.v, 0, stack);
    }));
});
```

We can now rewrite our distance function with some more abstraction.

```
tests.distance3 = function (stack) {
    let program = [
```

We define a "difference and square" subprogram

```
        block([word('-'), word('dup'), word('*')]), symbol('dsq'), word('def'),
```

We name our four arguments and consume them from the stack.

```
        symbol('y2'), word('def'),
        symbol('x2'), word('def'),
        symbol('y1'), word('def'),
        symbol('x1'), word('def'),
```

Calc square of x1 - x2

```
        word('x1'), word('x2'), word('dsq'), word('do'),
```

Calc square of y1 - y2

```
        word('y1'), word('y2'), word('dsq'), word('do'),
```

Sum and sqroot.

```
        word('+'), word('sqrt')
    ];

    return run(test_env(), program, 0, stack);
};
```

4 Scoping and Control Structures

Date: 28 Feb 2017

What we had to do with `do` is still not quite satisfactory. We're now forced to distinguish between when a word is a primitive and when it is a "user-defined" block to be run using `do`. We don't really want this distinction.

We'll define a separate primitive called `defun` which will behave just like def, but must be used with blocks so that referring to these "defined functions" does not require an explicit `do` to perform them.

The disadvantage of `defun` folding the `do` operation is that we can no longer treat the block as a value. The only thing we can do with a `defun`d block is to execute it.

```
stddefs(function (env) {
    define(env, 'defun', prim(function (env, stack) {
        let sym = pop(stack), program = pop(stack);
        console.assert(sym.t === 'symbol');
        console.assert(program.t === 'block');
        define(env, sym.v, prim(function (env, stack) {
            return run(env, program.v, 0, stack);
        }));
        return stack;
    }));
});
```

With this `defun`, we can simplify the distance calculation program to -

```
tests.distance4 = function (stack) {
    let program = [
```

We define a "difference and square" subprogram

```
        block([word('-'), word('dup'), word('*')]), symbol('diffsq'), word('defun'),
```

We name our four arguments and consume them from the stack.

```
        symbol('y2'), word('def'),
        symbol('x2'), word('def'),
        symbol('y1'), word('def'),
        symbol('x1'), word('def'),
```

Calc square of x1 - x2

```
        word('x1'), word('x2'), word('diffsq'),
```

Calc square of y1 - y2

```
        word('y1'), word('y2'), word('diffsq'),
```

Sum and sqroot.

```
        word('+'), word('sqrt')
    ];

    return run(test_env(), program, 0, stack);
};
```

4.1 Control structures

We now have a concept of blocks and we can use them to implement simple control structures like if-then-else and while. We're going to need some comparison operators to work with and we need a boolean type.

```
let bool = function (b) { return {t: 'bool', v: b}; };
```

Languages differ in how they choose to define a boolean type based on what notion of "type" is implemented. In our case, we're treating a boolean like an enumeration where the `v:` part tells what the value actually is. Dynamic object-oriented languages may implement this idea by defining a separate 'true' type and a 'false' type.

```
stddefs(function (env) {
    define(env, 'true', bool(true));
    define(env, 'false', bool(false));

    define(env, '>', prim(function (env, stack) {
        let y = pop(stack), x = pop(stack);
        return push(stack, bool(x.v > y.v));
    }));

    define(env, '<', prim(function (env, stack) {
        let y = pop(stack), x = pop(stack);
        return push(stack, bool(x.v < y.v));
    }));

    define(env, '>=', prim(function (env, stack) {
        let y = pop(stack), x = pop(stack);
        return push(stack, bool(x.v >= y.v));
    }));

    define(env, '<=', prim(function (env, stack) {
        let y = pop(stack), x = pop(stack);
        return push(stack, bool(x.v <= y.v));
    }));

    define(env, '=', prim(function (env, stack) {
        let y = pop(stack), x = pop(stack);
        return push(stack, bool(x.v === y.v));
    }));

    define(env, '!=', prim(function (env, stack) {
        let y = pop(stack), x = pop(stack);
        return push(stack, bool(x.v !== y.v));
    }));
});
```

We'll also need some binary combining operators for booleans.

```
stddefs(function (env) {
    define(env, 'and', prim(function (env, stack) {
        let y = pop(stack), x = pop(stack);
        return push(stack, bool(x.v && y.v));
    }));

    define(env, 'or', prim(function (env, stack) {
        let y = pop(stack), x = pop(stack);
        return push(stack, bool(x.v || y.v));
    }));

    define(env, 'not', prim(function (env, stack) {
        let x = pop(stack);
        return push(stack, bool(!x.v));
    }));

    define(env, 'either', prim(function (env, stack) {
```

```
        let y = pop(stack), x = pop(stack);
        return push(stack, bool(x.v ? !y.v : y.v));
    }));
});
```

> **Question**: What would be some advantages/disadvantages of defining the boolean combining operators in the above manner? For example, in the expression "a and b", if the "a" part ends up being false, then the "b" part doesn't even need to be evaluated. This is called *short circuiting*. A similar behaviour holds for the "or" and "either" operators too. Such short circuiting prevents unnecesary computations from happening. How would you re-design and re-implement these operators to have short circuiting behaviour?

We'll implement a generic branching mechanism that checks a set of conditions associated with actions and performs the first set of actions whose condition evaluates to a true value. This will be implemented using a `cond` primitive. However, contrary to other primitives we've implemented thus far, this one will take a block argument with a specific structure - a sequence of condition/consequence pairs each of which is itself represented using a block. The condition blocks are expected to produce a boolean on the stack which `branch` will examine to determine whether to execute the corresponding block or not. If none of the conditions were satisfied, no action is taken.

Note that for this to work properly, the condition and conseuence blocks must themselves behave properly and leave the stack in the right state.

```
stddefs(function (env) {
    define(env, 'branch', prim(function (env, stack) {
        let cond_pairs = pop(stack);
        console.assert(cond_pairs.t === 'block');
        console.assert(cond_pairs.v.length % 2 === 0);
        for (let i = 0; i < cond_pairs.v.length; i += 2) {
            console.assert(cond_pairs.v[i].t === 'block');
            console.assert(cond_pairs.v[i+1].t === 'block');
```

Check the condition.

```
            stack = run(env, cond_pairs.v[i].v, 0, stack);
            let b = pop(stack);
            console.assert(b.t === 'bool');
            if (b.v) {
```

Condition satisfied. Now evaluate the "consequence" block corresponding to that condition.

```
                return run(env, cond_pairs.v[i+1].v, 0, stack);
            }
        }
        return stack;
    }));
});
```

4.1.1 Test branch

Run this like `tests.branch([number(2), number(3)])`.

```
tests.branch = function (stack) {
    let program = [
        symbol('y'), word('def'),
        symbol('x'), word('def'),
        block([
```

```
            block([word('x'), word('y'), word('<')]),
            block([string("less than")]),

            block([word('x'), word('y'), word('>')]),
            block([string("greater than")]),

            block([word('true')]),
            block([string("equal")])
        ]),
        word('branch')
    ];

    return run(test_env(), program, 0, stack);
};
```

Concept: We've been writing programs simply using Javascript arrays and plain objects. Our program is itself an array. With the newly introduced "blocks", our data structure for representing programs just gained some self referential characteristics. With the recent `branch`, programs are now starting to look like trees with some amount of nesting of blocks. This data structure that we're using to represent our "programs" is an example of **Abstract Syntax Trees** or **AST** for short.

We've been writing ASTs all along! There is little else to it.

4.1.2 Test fibonacci series

Are we Fibonacci yet? Not quite .. 'cos we can't print anything yet :)

```
stddefs(function (env) {
    define(env, 'print', prim(function (env, stack) {
        console.log(pop(stack).v);
        return stack;
    }));
});
```

Ok now let's try again.

```
tests.fibonacci = function (n) {
    let program = [
```

We define a 'fib' word that uses 3 values from the stack i n1 n2 It prints n1, updates these values as (i-1) n2 (n1+n2) and calls itself by name again, until i reaches 0.

```
        block([
```

Load up our three arguments.

```
            symbol('n2'), word('def'),
            symbol('n1'), word('def'),
            symbol('i'), word('def'),

            block([
```

The recursion breaking condition.

```
                block([word('i'), number(0), word('<=')]),
                block([]),
```

```
                block([bool(true)]),
                block([
                    word('n1'), word('print'),
```

Leave three values on the stack just like when we started. Then call ourselves again.

```
                    word('i'), number(1), word('-'),
                    word('n2'),
                    word('n1'), word('n2'), word('+'),
```

Recursively invoke ourselves again.

```
                    word('fib')
                ])
            ]),
            word('branch')
        ]),
        symbol('fib'), word('defun'),
```

The number n is assumed to be available on the stack.

```
        number(0), number(1), word('fib')
    ];

    return run(test_env(), program, 0, [number(n)]);
};
```

4.2 Detour: Argument binding

It is kind of getting to be a tedious ritual to bind arguments passed on the stack into "variables" we can then refer to in our programs.

```
block([
symbol('n2'), word('def'),
symbol('n1'), word('def'),
symbol('i'), word('def'),
...
])
```

Let's simplify this using an argument binding primitive we'll call `args`. We'll make args take a block given on the stack with a special structure - the block must consist of a sequence of symbols. i.e. If you executed the block you'd get a bunch of symbols on the stack. We scan this block from the end to beginning, pop off matching elements from the stack and `def` them into our environment.

This will now permit us to simplify that ritual above to the following -

```
block([
block([word('i'), word('n1'), word('n2')]), word('args'),
...
])
```

```
stddefs(function (env) {
    define(env, 'args', prim(function (env, stack) {
        let args = pop(stack);
        console.assert(args.t === 'block');
        for (let i = args.v.length - 1; i >= 0; --i) {
            console.assert(args.v[i].t === 'word');
```

```
            define(env, args.v[i].v, pop(stack));
        }
        return stack;
    }));
});
```

Notice that our functions are now getting to be more self describing. If we use `args` in the opening part of all our functions by convention, the "code" of each function can be examined to tell how many arguments it needs from the stack - i.e. its arity. By then connecting these symbols to what the function does with them, we can infer a lot more about the function. For example, if `n1` and `n2` are the names of two arguments, and the function calculates `n1 n2 +` at some point, we know that these two must be numbers without executing the function.

Question: Can you relate this to main stream programming languages?

4.3 Implementing `if`

`branch` is general enough that we don't need any other conditional handling in our language. However, for one or two branches, all that nested blocks seems a bit too much. So just for convenience and minimalism, we can implement an `if` operator as well. `if` will pop two elements off the stack - a boolean and a block. It will execute the block like `do` if the boolean happened to be true. This way, you can chain a sequence of `if`s to get similar behaviour to `branch`.

```
stddefs(function (env) {
    define(env, 'if', prim(function (env, stack) {
        let blk = pop(stack), cond = pop(stack);
        console.assert(blk.t === 'block');
        console.assert(cond.t === 'bool');
        if (cond.v) {
```

Note that we're again making the choice that the if block will get evaluated in the enclosing scope and won't have its own scope. If you want definitions within `if` to stay within the `if` block, then you can make new environment for the block to execute in.

```
            return run(env, blk.v, 0, stack);
        }
        return stack;
    }));
});
```

We can now rewrite the fibonacci in terms of `if` like this -

```
tests.fobonacci_if = function (n) {
    let program = [
```

We define a 'fib' word that uses 3 values from the stack i n1 n2 It prints n1, updates these values as (i-1) n2 (n1+n2) and calls itself by name again, until i reaches 0.

```
        block([
```

Load up our three arguments.

```
            symbol('n2'), word('def'),
            symbol('n1'), word('def'),
            symbol('i'), word('def'),
```

The loop exit condition.

```
            word('i'), number(0), word('>'),
            block([
                word('n1'), word('print'),
```

Leave three values on the stack just like when we started. Then call ourselves again.

```
                word('i'), number(1), word('-'),
                word('n2'),
                word('n1'), word('n2'), word('+'),
```

Recursively invoke ourselves again.

```
                word('fib')
            ]), word('if')
        ]),
        symbol('fib'), word('defun'),
```

The number n is assumed to be available on the stack.

```
        number(0), number(1), word('fib')
    ];

    return run(test_env(), program, 0, [number(n)]);
};
```

4.4 Scope of variables

It looks like we're comfortably using variables, binding them to values in our environment and even doing recursion with them. In other words, when we execute a block, its input not only consists of the stack contents, but also the bindings in the environment. This means a block can both access variables from the environment as well as clobber it!

> **Question**: What are the consequenecs of this approach if we adopted it for large scale software development? Do you know if this approach is used anywhere currently?

Here is a "weird" implementation of `fib` to prove the point. Try to trace what it does.

```
tests.fibonacci2 = function (n) {
    let program = [
        block([
            block([
```

The recursion breaking condition.

```
                block([word('i'), number(0), word('<=')]),
                block([]),

                block([bool(true)]),
                block([
                    word('n1'), word('print'),
```

Redefine the three variables we use for our iteration.

```
                        word('i'), number(1), word('-'),
                        symbol('i'), word('def'),

                        word('n2'),
                        word('n1'), word('n2'), word('+'),
```

```
                    symbol('n2'), word('def'),
                    symbol('n1'), word('def'),
```

Recursively invoke ourselves again.

```
                    word('fib')
                ])
            ]),
            word('branch')
        ]),
        symbol('fib'), word('defun'),
```

Store number(n) available on the stack into 'i'.

```
        symbol('i'), word('def'),

        number(0), symbol('n1'), word('def'),
        number(1), symbol('n2'), word('def'),
        word('fib')
    ];

    return run(test_env(), program, 0, [number(n)]);
};
```

The above implementation of `fib` clearly shows that all our so called "functions" are using and clobbering what are essentially "global variables". This is not a great idea for programming in the large. It means you'll need to reserve some names specially for use within functions as "temporary" variables so that functions don't assume that they have valid values. It also means you can now look at the internal state of a function after it has executed by examining the environment. You cannot also transfer the source code of a function between two programs, for fear that the other program may be using some names that your function relies on and might clobber, causing the other program to break.

In short, we need our functions to **encapsulate** the computation they perform and communicate with the outside world only via the stack.

4.4.1 Design choices

When we want to implement such an encapsulation, we need to take a step back and think about the design choices we have at hand and the consequences of these designs. Indeed, different programming languages may make different choices at this point, ending up with varied behaviours. If we understand the choices at hand, then we'll likely be able to quickly construct mental models of execution of programs in a given language and be effective in exploiting the design.

4.4.1.1 Option: Scoping by copy

One basic choice at hand here is to lift the notion of "environment" from "set of variable bindings" to a "stack of set of variable bindings". This way, when we enter a "scope" in which we wish that local changes don't affect the enclosing scope, we can simply make a **copy** of the current environment before entering the scope, push it on to this stack, and pop it back once the scope ends.

An important consequence of this choice is that it will **not** be possible for a scope to influence another scope through variables. We may want that in our language, we may not. It is not any inevitable law, but simply a choice we get to make at design time. With this approach, it is clear what should be done when a block "sets a variable to a value".

When accessing variables not `def`d within a scope (these are called "free variables"), this implementation will result in the block picking up values of free variables from the environment in which it is *executed*. Such free variables are said to be "dynamically scoped". This implementation limits dynamic scoping to permit reading such variables, but invoking `def` on them won't cause changes in the executing environment of a scope irrespective of where it is defined.

> **Term**: "Free variables" are variables in a block that are referred to in the code without being defined in it as local.

4.4.1.2 Option: Scoping by live reference chain

Instead of making a *copy*, we could turn the variable lookup process to walk a chain of environments. When looking up a variable, we'd check the head of the chain first. If it isn't found, we check its "parent", then the parent's parent and so on until we find a reference or declare the variable to not be found. This way, we could just push an empty environment at the end of the chain to limit the scope of variables used by a block or function.

This is a bit more efficient than the environment copy option. However, it influences the meaning of blocks and functions. With this implementation, the notion of "setting a variable" can have two meanings. One is to introduce the binding in the innermost scope where the variable may not exist at `def` time. Another is to figure out which scope in the chain has the variable defined and make the `def` operate on that scope's environment. With the first option, we get a behaviour similar to (1), but with the second option, blocks and functions get to modify their environments in a more controlled manner than through global variables.

Another consequence of this chain approach is how it lets us deal with blocks that are defined in one scope but are evaluated in another. We can inject a block into a different scope by manipulating the chain of environments. The story with free variables is different in this case. A block that `def`s a free variable *can* modify the variable in the scope chain in which it is executed.

4.4.1.3 Option: Scope stored on the stack

Another implementation choice is to manage the scopes along with the data on the same stack. In this case, the "environment" will simply be an index into a stack at which we begin the lookup process. The above two implementation choices of copying bindings or linking them into a scope chain apply in this implementation too.

As of now, this implementation choice only implies some additional book-keeping on our part with no meaning difference with the other implementation. However, if we change the execution model (to asynchronous, for example), then this will impact the kinds of programs we can write and how they will behave.

4.4.1.4 Option: Splitting away `set` from `def`

With the three options above, we've pretended that we'll be using `def` to both *introduce* a new variable as well as to *set* an existing variable. This need not be the case. We're free to differentiate between the two operations, which leads to further bifurcation of implementation choices and consequences.

4.4.1.5 Option: Accessing definition scope in execution scope

When a block is defined, it may wish to refer to bindings in its *definition* scope and recall them in its *execution* scope. Supporting this feature requires blocks to be created with the necessary bindings captured when they're pushed on to the stack. Then at execution time, these bindings need to be injected into the environment chain so that the blocks have access to the definition scope as well as the execution scope.

We have a couple of options here too - where we preserve the definition environment between invocations and where we simply copy the bindings, thereby losing any modifications a block may do to is definition scope.

These two options determine whether a block can communicate with itself between two executions or not. If we preserve the definition environment by reference, then any `def`s executed will modify it and the changes will be visible to subsequent runs. If we copy the definition environment into the execution scope, then the modifications won't sustain.

4.4.1.6 Our choice

For the moment, we'll take and follow through on approach (2), with a basic implementation of (5) also thrown in.

This means free variables referenced in a function will be able to read variables available in their definition and execution environments, but won't be able to modify them.

We'll need to modify our notion of "environment" which now needs to be turned into a "stack of variable bindings" instead of "variable bindings". This means we need to change the definitions of `mk_env`, `lookup` and `define` to reflect this. While `lookup` will now scan the entire stack to find a value, `define` will modify it only in the inner-most environment. Each scope will maintain a reference to its "parent scope" in a special reserved key called "[[parent]]". Though we'll be doing this, we'll also maintain the parent relationship via a stack of environments as well for the moment.

```
let parent_scope_key = '[[parent]]';

mk_env = function () {
    return [{}]; // We return a stack of environments.
};

lookup = function (env, word) {
    for (let i = env.length - 1; i >= 0; --i) {
        let scope = env[i];
        while (scope) {
            let val = scope[word.v];
            if (val) { return val; }
            scope = scope[parent_scope_key];
        }
    }
    return undefined;
};

define = function (env, key, value) {
    env[env.length - 1][key] = value;
    return env;
};
```

We also need new operations for entering and leaving a local environment. This local environment needs to have all the info available in its parent environment, so we copy the contents of the parent into the local environment at creation time. With this, when we leave a local environment, any bindings effected by `define` will be lost.

```
let enter = function (env) {
    let localEnv = {};
    localEnv[parent_scope_key] = current_bindings(env);
    env.push(localEnv);
    return env;
};

let leave = function (env) {
```

```
    env.pop();
    return env;
};

let current_bindings = function (env) {
    return env[env.length - 1];
};

let copy_bindings = function (src, dest) {
    for (let key in src) {
        dest[key] = src[key];
    }
    return dest;
};
```

This change affects how the `do` and `defun` words function as well. We don't want to change how `branch` behaves just yet.

Before we get to any of that though, we must modify `run` to handle creation of blocks. When a block is about to be pushed on to the stack, a copy of its definition environment must be made and stored along with the block. When `do` and `defun` evaluate blocks, they will restore this definition environment copy and then execute it.

```
run = function (env, program, pc, stack) {
    for (; pc < program.length; ++pc) {
        let instr = program[pc];

        if (instr.t === 'word') {
            let deref = lookup(env, instr);
            if (!deref) {
                console.error('Undefined word "' + instr.v + '" at instruction ' + pc);
            }
            instr = deref;
        }

        switch (instr.t) {
            case 'prim':
                stack = apply(env, instr, stack);
                break;
```

Special case for block definition. We capture the definition environment and store it along with the block structure on the stack. There is a subtlety here - we're making a new block value instead of reusing the block value in the program directly by reference. This is necessary because the same block may end up being defined multiple times with different sets of bindings from its environment.

One subtlety is that copying the bindings at this point as opposed to referencing the current bindings have different effects on program behaviour. When copying, further alterations to the parent environment will not be available to the block's environment. If we reference instead, then we can not only save copy time, but also use the environment sharing as a way to communicate between blocks in the same environment. Briefly, forward references to definitions won't work within a block.

We'll choose the stricter form and do copying for now. Later we'll relax this constraint.

We've implemented a "closure"!

> **Question**: Comment on the efficiency of doing this as opposed to copying everything as might've happened had we used approach (1).

```
            case 'block':
                let bound_block = block(instr.v);
                bound_block.bindings = instr.bindings || copy_bindings(current_bindings(env), {});
                push(stack, bound_block);
                break;
```

In all other cases we just store the value on the stack.

```
            default:
                push(stack, instr);
                break;
        }
    }

    return stack;
};
```

`do` and `defun` need to be redefined to support the block-level bindings we wish to incorporate. But just so we don't repeat ourselves again, we'll pull out the common functionality in `do` into a function we can customize later.

```
let do_block = function (env, stack, block) {
```

This is the crucial bit. We enter a local environment when evaluating the block and leave it once the evaluation is complete. Notice that `copy_bindings` will copy all the bindings including the reference to the parent. This means that the block will not have access to bindings visible from the execution environment. So if a block makes a reference to a "free" variable named 'x', then the meaning of 'x' will always be either in the context of where the block was created, or local to the block.

> **Question**: What consequence does this choice have on program behaviour? Consider program predictability, debuggability, etc. You'll need to check this from both perspectives - i.e. what happens when we include the execution environment's bindings as well as what happens when we exclude them.

```
    enter(env);
    copy_bindings(block.bindings, current_bindings(env));
    stack = run(env, block.v, 0, stack);
    leave(env);
    return stack;
};

stddefs(function (env) {
    define(env, 'do', prim(function (env, stack) {
        let block = pop(stack);
```

If we've been given a primitive, we call it directly.

```
        if (block.t === 'prim') {
            return block.v(env, stack);
        }

        console.assert(block.t === 'block');
        return do_block(env, stack, block);
    }));

    define(env, 'defun', prim(function (env, stack) {
        let sym = pop(stack), block = pop(stack);
```

```
        console.assert(sym.t === 'symbol');
        if (block.t === 'prim') {
            define(env, sym.v, block);
        } else {
            console.assert(block.t === 'block');
            define(env, sym.v, prim(function (env, stack) {
                return do_block(env, stack, block);
            }));
        }
        return stack;
    }));
});
```

With the above re-definitions for local environments, we'll find that `tests.fibonacci2` no longer works, but `tests.fibonacci` does work.

```
tests.fibonacci2 = function (stack) {
    console.error("tests.fibonacci2 will work only before the support for local scope was added.\n" +
                  "tests.fibonacci will continue to work though.");
    return stack;
};
```

5 A Parser

Requires slang.js

```
"use strict";
```

In the No Spoon series, we want to start with a mental model, and then move up through semantics into syntax. The motivation for this approach is that too often devs get caught up in the syntactic aspects of a language, and if they're lucky, some semantics which hinder them from reaping the benefits of the language. By starting with the mental model, our intention is to ensure that we introduce no new concepts as we move through semantics to syntax.

In that spirit, we'd like to start with our now clear mental model as expressed in our programs and then make a syntax that is closely aligned with it with the sole purpose of saving some typing. We will introduce no new concepts along the way.

```
let fibonacci_program = [
    block([
        symbol('n2'), word('def'),
        symbol('n1'), word('def'),
        symbol('i'), word('def'),

        word('i'), number(0), word('>'),
        block([
            word('n1'), word('print'),

            word('i'), number(1), word('-'),
            word('n2'),
            word('n1'), word('n2'), word('+'),

            word('fib')
        ]),
```

```
        word('if')
    ]),
    symbol('fib'), word('defun'),

    number(0), number(1), word('fib')
];
```

5.1 String to entity mapping

To start with, our "program" is just a normal Javascript array. Each entry of the array uses a value constructor to determine what instruction occurs at that location. The following types of values need to be supported in our programs.

- words
- numbers
- strings
- symbols
- blocks

To work out a syntax for all instruction types other than blocks, it would suffice if we're able to work out the constructor that we must use given a string representation of the argument of the constructor. i.e. Given "123", if we can infer it is a number, given "abc" that it is a word, given "hello world" that it is a string, and given "x1" that it is a symbol.

Of these, we have an overlap between words, symbols and strings. Numbers are straightfoward. For blocks, we can just use arrays anyway.

So we can add something special to identify a symbol given its string representation. For example, we could infer that all strings of the form ":abcxyz123" will be symbols (not including the ":"), that strings are enclosed in double quotes, with everything else being a word.

For strings, we want to keep it simple and say that the double quote character won't be used within the string, but we'll permit URL-style encoding instead. This makes it easy to parse out a string.

5.2 Commitments made already

The problem with starting out with syntax starts very early. We see already how we're limiting the shape of symbols, how we're going to represent strings in serialized form and the specific numeric format we'll use for numbers.

Thankfully, this is where it ends for now. We can make a nice parser already. Here is our ideal version of our fibonacci program that would save typing.

```
let fibonacci_program_rep = [
    [ ':n2', 'def',
      ':n1', 'def',
      ':i', 'def',

      'i', '0', '>',
      [ 'n1', 'print',

        'i', '1', '-',
        'n2',
        'n1', 'n2', '+',

        'fib'
```

```
      ], 'if',
    ],
    ':fib', 'defun',

    '0', '1', 'fib'
];
```

Now let's write a function to turn each of those array entries into a valid slang entity.

```
let parse_entity = function (item) {
    if (typeof item === 'string') {
        if (item[0] === ':') { // Symbol
            return symbol(item.substring(1));
        }
        if (item[0] === '"' && item[item.length-1] === '"') { // String
            return string(item.substring(1, item.length-1));
        }
        let n = parseFloat(item);
        if (!isNaN(n)) { // Number
            return number(n);
        }
        return word(item);
    } else if (item instanceof Array) {
        return block(item.map(parse_entity));
    } else {
        throw new Error('Invalid item serialization');
    }
};

fibonacci_program = fibonacci_program_rep.map(parse_entity);
```

5.3 Flattening to a single string

Now that we've represented each entity in serialized form, we can just flatten out our array to make a serialized representation of our entire program. So if we can parse this flattened out form, then we're done with our parser.

```
let parse_slang = function (program) {
    let result = [];
    let glyph = /^[^\s\"\[\]]+/;
    do {
```

Kill prefix spaces.

```
        program = program.replace(/^\s+/, '');

        if (program.length === 0) {
            return result;
        }
```

Check for number.

```
        let n = parseFloat(program);
        if (!isNaN(n)) {
            result.push(number(n));
            program = program.replace(glyph, '');
```

```
            continue;
        }
```

Check for symbol.

```
        if (program[0] === ':') {
            let sym = glyph.exec(program);
            if (sym) {
                result.push(symbol(sym[0].substring(1)));
                program = program.substring(sym[0].length);
                continue;
            }

            console.error('Hm? Check this out - {' + program.substring(0, 30) + ' ...}');
            program = program.replace(glyph, '');
            continue;
        }
```

Check for string.

```
        if (program[0] === '"') {
            let i = program.indexOf('"', 1);
            if (i >= 0) {
                result.push(string(decodeURIComponent(program.substring(1, i))));
                program = program.substring(i+1);
                continue;
            }

            console.error('Bad string - {' + program.substring(0, 30) + ' ...}');
            result.push(string(program.substring(1)));
            program = '';
            continue;
        }
```

Check for block start.

```
        if (program[0] === '[') {
            let b = parse_slang(program.substring(1));
            result.push(block(b.block));
            program = b.rest;
            continue;
        }
```

Check for block end.

```
        if (program[0] === ']') {
            return { block: result, rest: program.substring(1) };
        }

        let w = glyph.exec(program);
        result.push(word(w[0]));
        program = program.substring(w[0].length);
    } while (program.length > 0);

    return result;
};
```

This makes our fibonacci program quite simple indeed.

```
fibonacci_program = parse_slang(`
    [ [i n1 n2] args
      i 0 >
      [ n1 print
        i 1 -
        n2
        n1 n2 +
        fib
      ] if
    ] :fib defun
    0 1 fib
`);
```

5.4 Printing an item

Given that we can parse a serialized program form, it'll be great to be able to write out a data item in serialized form so we can do both.

```
let show_slang = function (term) {
    if (term instanceof Array) {
        return term.map(show_slang).join(' ');
    }

    switch (term.t) {
        case 'number': return '' + term.v;
        case 'word':   return term.v;
        case 'symbol': return ':' + term.v;
        case 'string': return '"' + encodeURIComponent(term.v) + '"';
        case 'block':  return '[' + show_slang(term.v) + ']';
        case 'prim':   return '<<prim>>';
        default:
            return '<<err>>';
    }
};
```

So `show_slang` and `parse_slang` are expected to be nearly inverses of each other.

5.5 Adding comments to our parser.

Normally in programming languages, a "comment" is a piece of text that doesn't add any meaning to the program and can be stripped out without loss for the computer .. though usually at a great loss to humans.

It would be natural to expect our syntax to support such comments too. However, instead of adding it at the parser level, we'll just add a comment primitive that will drop the latest item on the stack. This way, we can write -

```
"Some code below" ;
```

and the string will in essence be of no use to the program. Here, we're using the word ';' to mean `comment`. However, this way costs some runtime overhead to process comments. Since we can easily remedy that before passing the program to our interpreter, we won't bother about that.

That said, we need to clarify one behaviour of such a comment that might be unexpected to those familiar with main stream langauges - that it comments out the value effect of the *most recent computation* and not

the *most recent term*. So as long as we use `comment` such that the most recent computation is also the most recent term, programs won't cause any ambiguity there.

Note that this implementation of commenting is the same as the `drop` primitive … just with better wording.

```
stddefs(function (env) {
    define(env, ";", lookup(env, symbol('drop')));
});
```

6 Vocabularies

Date: 15 March 2017

Requires slang.js

```
"use strict";
```

We've thus far done the traditional stuff with scopes. In most languages, your power stops pretty much there .. unless you're using something from the LisP family. We'll do something slightly unusual to demonstrate what having control over this process can let us do.

So far, we can define "words" to mean whatever we want them to mean and we can do that in a local environment without affecting the surrounding environment. This ability lets us reason about blocks in isolation without worry about how its execution environment is going to influence the behaviour of the block.

One of the most powerful things a language can provide you is to make whatever facilities it provides in what is called a "reified" or "first class" manner. If we can have local variables, what if we can introduce a type using which we can capture the local variables introduced by a block and use it wherever we want later on even without the block? In other words, what if we could define and use environments *within* our little language?

We can call this a "vocabulary", since we're interested in meanings assigned to a set of words. For example, we can use a block to define a set of functions that will work with 2D points as xy coordinates pushed on to the stack. We can store away these definitions in a vocabulary and call on them only when we need them. We introduce a new type called `vocab` for this purpose.

```
block([...]), word('vocab')
```

When the `vocab` word is interpreted, we'll get an object on the stack that captures all the bindings that were created within the scope of the block, in addition to evaluating the block just like `do` would.

```
let vocab = function (bindings) { return {t: 'vocab', v: bindings}; };
```

We re-define test_env so that the stdlib becomes a common entity instead of being copied over and over.

```
test_env = function () {
    return enter(load_stdlib(mk_env()));
};
```

When we're introducing such a "reification" in our system, it is also useful to think about what is called the "dual" operation. In our case, `vocab` captures a set of bindings made within a block and pulls it into an object accessible to our programs. The "dual" or "inverse" of this operation would be to take such an object and re-introduce the bindings that it captured into an environment. After such a step, the vocabulary's bindings would be available like normal within the current block.

We could call this inverse operation `use` because it offer a way to "use" the words that a vocabulary defines.

> **Question**: Can you think of other such "reification-dual" pairs in languages that you know?

Whenever we have this kind of matched pair - an operation and its inverse where the operation is a communication between two different layers of a system, we open ourselves to powerful composition possibilities. In our case, for example, we can capture two or more vocabularies, `use` them within a block in order to make a composite vocabulary consisting of all the words in those vocabularies. This gives us an "algebra of vocabularies". We'll see later how we can put this algebra to good use.

What are other such pairs? In Java, for example, the reflection API lifts what is normally accessible only to the JVM - the notion of classes, methods, properties, etc. - into the Java language, permitting programmers to invoke methods and introspect objects without prior knowledge about their classes or properties. One "inverse" of this lifting is a way to take some data produced by a Java program and reintroduce it to the JVM as a class. This is nothing but the "class loader" mechanism. Many frameworks in Java exploit class loaders to make programming certain kinds of systems simple.

> **Question**: If you consider the concept of an "iterator" or "enumerator" in languages like C++, Java and C#, what would be the "dual concept" of an iterator?

```
tests.vocab = function () {
    let program = [
        block([
            block([
                block([word('x1'), word('y1'), word('x2'), word('y2')]), word('args'),
                word('x1'), word('x2'), word('-'),
                word('y1'), word('y2'), word('-'),
                word('length')
            ]), symbol('distance'), word('defun'),

            block([
                block([word('dx'), word('dy')]), word('args'),
                word('dx'), word('dy'), word('dx'), word('dy'), word('dot'),
                word('sqrt')
            ]), symbol('length'), word('defun'),

            block([
                block([word('x1'), word('y1'), word('x2'), word('y2')]), word('args'),
                word('x1'), word('x2'), word('*'),
                word('y1'), word('y2'), word('*'),
                word('+')
            ]), symbol('dot'), word('defun')
        ]), word('vocab'), symbol('point'), word('def'),

        block([
            word('point'), word('use'),
            number(2), number(3), number(5), number(7), word('distance')
        ]), word('do')
    ];

    return run(test_env(), program, 0, []);
};
```

Now for the definitions of `vocab` and `use`.

```
stddefs(function (env) {
```

`vocab` needs to run the block on the stack just like `do` does, except that before the local environment of the block is thrown away, it is captured into a separate `bindings` hash and stored away as part of the vocabulary.

```
    define(env, 'vocab', prim(function (env, stack) {
        let defs = pop(stack);
        console.assert(defs.t === 'block');
```

Execute the block and capture its definitions before we leave it.

```
        enter(env);
        stack = run(env, defs.v, 0, stack);
        let bindings = copy_bindings(current_bindings(env), {});
        leave(env);
```

We don't want to preserve the scope chain in this case, so delete the parent scope entry.

```
        delete bindings[parent_scope_key];

        return push(stack, vocab(bindings));
    }));
```

The way we're implementing `use`, the vocabulary is "immutable" - i.e. you cannot change the bindings in a vocabulary in a block that "uses" a vocabulary. Once the block in which the `use` operation occurs finishes, the introduced bindings will no longer be in effect. So the effect of `use` is said to be "locally scoped", just like `def` and `defun`.

```
    define(env, 'use', prim(function (env, stack) {
        let vocab = pop(stack);
        console.assert(vocab.t === 'vocab');

        copy_bindings(vocab.v, current_bindings(env));
        return stack;
    }));
});
```

> **Concept**: Such a "vocabulary" is equivalent to "modules" or "packages" in many languages. However, many languages don't let modules be "first class" in that they cannot be passed around. Javascript is a language which permits you to pass around modules defined in a certain way. Languages like C++ ("modules" = "namespaces") and Java ("modules" = "pakages") don't. You could consider "classes" to be modules in a twisted sense, but the notion of a class has additional machinery that doesn't befit the notion of a vocabulary.
>
> **Question**: How would you implement a "mutable" vocabulary in *slang*?

6.1 The structure of vocabularies

While it looks like we've introduced the "vocabulary" concept to illustrate that we can now play with scope in our language, we've already done what would be considered to be "fantastic" features in some programming languages.

1. Our vocabularies are "first class".
2. Our vocabularies can be combined to make new vocabularies.
3. Our vocabularies can be parameterized.

We already talked about (1). What is worth pointing out though is that we can pass vocabularies to functions/blocks to customize their behaviour by injecting values into the function scope.

(2) is simply the fact that invoking the `use` word introduces a vocabulary into the current scope. This means we can invoke more than one vocabulary and have them all combine in the order of invocation. If this were itself within a block, then we can use that block to define a new vocabulary, like this -

```
block([word('a'), word('use'), word('b'), word('use')]), word('vocab')
```

This gives us a kind of "inheritance" like the way object oriented languages combine classes to form new ones.

(3) is a consequence of the way we chose to define our vocabularies - by evaluating blocks. This means we can formulate a vocabulary that uses values on the stack to customize what gets defined. Our vocabularies don't even need to have names because they can be passed around by value. Parameterized modules are a powerful feature of the language OCaml.

Golden rule: Whenever you come up with an aspect of your system which has this characteristic, you know you have something powerful on your hands. The characteristic is that you have some operation using which you an combine two or more entities of a type to form a new entity of the same type.

7 Object Oriented Programming

Requires slang.js and slang_vocab.js

```
"use strict";
```

7.1 The essence of objects

We're going to *implement* an object oriented system in our language using the "vocabulary" mechanism we've just invented. To do this, we need to conceptualize what an object oriented system is. If we're to go by Alan Kay's expression of the essence of object orientation as "messaging" between encapsulated computational processes, we need to be able to express such a "message" in our system.

For starters, we don't really need a separate type for messages. We can use our good old symbols.

```
let message = symbol;
```

We also need a type for objects which can have a vocabulary to act on them and have properties to which we associate values. We make a simple generalization here as well - we'll treat any type which has a 'vocab' field whose value is a `vocab()` type as though it were an object to which we can send "messages".

```
let object = function (properties, vocabulary) {
    console.assert(vocabulary.t === 'vocab');
    return {t: 'object', v: properties, vocab: vocabulary};
};
```

A "message" is identified by a name. Given a message, the only thing we can logically do with it is to send it to an object. So we need to implement a "send" word which takes a message and a thing to which it can be sent to and "sends" it to it. For instructional purposes, we'll implement message sending as equivalent to method invocation just so it is in territory that everyone is familiar with.

```
stddefs(function (env) {
```

To "send a message to an object", we look up its vocabulary for the message symbol, take the defined handler and invoke it. If the vocabulary specifies a normal value, we just push it onto the stack.

Usage: `thing :message send`

```
    define(env, 'send', prim(function (env, stack) {
        let msg = pop(stack), thing = pop(stack);
        console.assert(msg.t === 'symbol');
```

We check for the `vocab` field as an indication of "objecthood". This means we can use `send` with other types which have been given a vocabulary also and not just those with type `object`.

```
        if (!thing.vocab || !thing.vocab.v[msg.v]) {
            console.error('No vocabulary relevant to message');
            return stack;
        }

        let method_or_val = thing.vocab.v[msg.v];
        if (!method_or_val) {
            console.error('Vocabulary doesn\'t accept message "' + msg.v + '"');
            return stack;
        }

        switch (method_or_val.t) {
            case 'prim':
                push(stack, thing);
                return apply(env, method_or_val, stack);
            default:
                return push(stack, method_or_val); // Normal value.
        }
    }));
```

To get a property of an object, we use a symbol to lookup its property list. This also works with vocabularies.

Usage: `thing :key get`

```
    define(env, 'get', prim(function (env, stack) {
        let name = pop(stack), thing = pop(stack);
        console.assert(name.t === 'symbol');
        console.assert(thing.t === 'object' || thing.t === 'vocab');
        return push(stack, thing.v[name.v]);
    }));
```

To change a property of an object, we modify its property list at the given key. At the end, we leave the object on the stack so that multiple put operations can be done.

Usage: `thing val :key put`

```
    define(env, 'put', prim(function (env, stack) {
        let name = pop(stack), val = pop(stack), thing = pop(stack);
        console.assert(name.t === 'symbol');
        console.assert(thing.t === 'object');
        thing.v[name.v] = val;
        return push(stack, thing);
    }));
```

`vocab new`

We certainly need some way to create objects! `new` takes the vocabulary on top of the stack and makes an object out of it. If the vocabulary includes a function named `make`, then it will invoke it on the object to build it up.

```
    define(env, 'new', prim(function (env, stack) {
        let voc = pop(stack);
        console.assert(voc.t === 'vocab');

        let thing = object({}, voc);
        push(stack, thing);

        let make = voc.v['make'];
```

```
        if (make && make.t === 'prim') {
            return apply(env, make, stack);
        }

        return stack;
    }));
});
```

That's about it. We're done implementing an "object oriented" language! It has objects. You can send messages to these objects and invoke code that will respond to these messages. Objects have properties that you can access and modify - though this is not really a necessity for them. You can construct vocabularies for objects to model their behaviour. You can combine these vocabularies to form new vocabularies.

> **Discussion**: What kind of an object oriented system have we just created? Does this correspond to any system you know of? Can we change the implementation to support other kinds of "object orientation"?

7.2 Invoking super

If you noticed, our mechanism doesn't give us an explicit notion of "inheritance", and so "invoking super" becomes a problem. This is even more of a problem if there is "inheritance" from multiple vocabularies involved. This is surprisingly easy to get around.

When we make a new vocabulary by combining existing vocabularies, we can refer to them directly, perhaps bound to a symbol. So all we need is a `send*` which will explicitly target a vocabulary on a given object, even irrespective of whether that vocabulary is part of the object's behaviour.

With `send*`, the invocation of a "super vocabulary" can be done using -

```
word('thing'), word('voc'), word('msg'), word('send*')
```

where `thing`, `voc` and `msg` are variables bound to values of appropriate types. `thing` should be an object, `voc` should be a vocabulary to direct when dealing with the object and `msg` is a message to send to the object as interpreted by the vocabulary.

> **Note**: The need for such a "send to super" surfaces in an object system purely because the concept of "message sending" has been interpreted as "method invocation". Therefore the question of "which method?" arises, hence the usual inheritance, etc. If message sending is interpreted as a message that is sent between two asynchronous *processes*, then this anomaly vanishes. In fact, in that case, the notion of inheritance, containment, etc. just disappear and merge into the concept of "object networks".

```
stddefs(function (env) {
    define(env, 'send*', prim(function (env, stack) {
        let msg = pop(stack), voc = pop(stack);
        console.assert(msg.t === 'symbol');
        console.assert(voc.t === 'vocab');

        let method_or_val = voc.v[msg.v];
        if (!method_or_val) {
            console.error('Vocabulary doesn\'t accept message "' + msg.v + '"');
            return stack;
        }

        if (method_or_val.t === 'prim') {
            return apply(env, method_or_val, stack);
        }
```

```
        return push(stack, method_or_val);
    }));
});
```

Alternatively, we can provide the ability to cast a spell on an object so that it can speak a given vocabulary. This would also permit us to make explicit super message sends.

`thing vocab cast`

Endows the `thing` with the given vocabulary and leaves it on the stack for further operations.

Options -

1. We can superimpose the given vocabulary on top of an existing vocabulary by adding words.
2. We can replace existing vocabulary entirely.

We choose the latter for simplicity. If you want (1), you can always make a new vocabulary that mixes multiple vocabularies and then use `cast`.

`cast` is intended to sound like "cast spells" rather than "type cast", though you could also think of it as the latter.

Once a thing has been casted, you can `send` messages to it using the vocabulary which it was casted with. This adds some cheap object orientation to the existing types we've defined. For example, you can do -

`2.34 complex cast`

where `complex` is a vocabulary for complex number operations. We don't touch the original value. This way, we can use this mechanism to do a "super send" in the case where multiple vocabularies are given to an object.

`thing super2 cast :message send`

```
stddefs(function (env) {
    define(env, 'cast', prim(function (env, stack) {
        let voc = pop(stack), thing = pop(stack);
        console.assert(voc.t === 'vocab');
        let copy = copy_bindings(thing, {});
        copy.vocab = voc;
        return push(stack, copy);
    }));
});
```

7.3 Making a complete object system

We have an asymmetry within our system. We have our new-fangled "objects" on the one hand, and on the other we have our numbers, strings, symbols and such "primitive objects". This is quite an unnecessary distinction as we can merge the two systems into a single system with the following rule -

> Every value is associated with a vocabulary.

In object-oriented languages like Smalltalk and Ruby, this principle is usually articulated as -

> Everything is an object.

So how do we convert our system into such a unified system?

As a first step, we can just modify all our "primitive object" constructors to produce entities which come along with a vocabulary. That would permit us to use `send` with all our primitive values as well. Our current implementation of `send` already accommodates this.

This seems simple, until we consider what we need to do with out "vocabulary objects". Since our vocabularies are also values, each vocabulary also need to have an associated vocabulary that tells the programmer how to talk to vocabularies!

- A value has a vocabulary that says how to talk to the value.
- A vocabulary is a value. So a vocabulary has a vocabulary that says how to talk to vocabularies.

```
n = {t: "number", v:5, vocab: howToTalkToNumber}
howToTalkToNumber = {t: "vocab", v: {..number methods..}, vocab: howToTalkToVocabulary}
howToTalkToVocabulary = {t: "vocab", v: {..vocab methods..}, vocab: ??}
```

What value should be in the `??` place? The simplest solution to that is -

```
howToTalkToVocabulary = {t: "vocab", v: {..vocab methods..}, vocab: howToTalkToVocabulary}
```

The snake must eat its own tail!

In a language like Smalltalk, the sequence is quite similar, and goes like -

- object is instance(class).
- class is instance(metaclass) is classOf(object).
- metaclass is instance(Metaclass) is metaclassOf(object).
- Metaclass is instance(Class)
- metaclassOf(Metaclass) is instance(Metaclass).

Ruby is less thorough in this matter, though it also adopts the "everything is an object" mindset. The notion of "metaclass" in Ruby is, for practical purposes, non-existent. Even so, Ruby still manages to expose considerable power to the programmer by working close enough to this territory.

> **Book**: The Art of the Metaobject Protocol details the value of having such a meta-object system in an "object oriented language". Though it discusses this in the context of the Common Lisp Object System (CLOS), which is one of the most thorough object-oriented systems designed to date, the concepts elucidated in it are generic enough to be applied to other languages and systems. Be warned that the book is not for the faint of heart, but if you survive it, you'll come out with a different brain :)

With such a unification via a metaobject mechanism, our system is further simplified by eliminating the `new` primitive. `new` can now be a message that we can `send` to a vocabulary object to make a new object that has that vocabulary.

8 Concurrency

Date: 3 April 2017

Requires slang.js, slang_vocab.js, slang_objects.js.

```
"use strict";
```

Most programming in today's environment involves concurrency. This means at any time, we can have more than one "process" running concurrently … from the perspective of the programmer/system designer. These concurrent processes may in some cases be executing in parallel on different processors or different machines or different cores on the same machine, while in other cases they may be multiplexed onto a single processor core. In all of these cases, we'd think of the processes as somehow independent of each other's execution. We'll also expect the environment to provide for these processes to maintain that independence.

To implement this kind of a mechanism, we have to radically change the way our "run" function is implemented … and actually a host of other things by consequence … so that we gain the ability to run "concurrent processes".

8.1 From sequentiality to concurrency as "beta abstraction"

When looking to model a domain, we can either come up with domain concepts top-down from an existing formalism, or try to figure out through trial and error in a bottom-up manner. While the former approach works for domains with a clear pre-existing computational formalism, we usually have to resort to bottom-up formalization in domains that we don't understand a priori.

The process of figuring out concepts in a bottom-up manner can appear quite haphazard. One technique that offer some guidance in such an effort is "beta abstraction". In fact, you could express many forms of abstraction as beta abstraction, so it is worth understanding beta abstraction.

The key idea is simple to express if we're not concerned about being mathematically rigorous.

> Beta abstraction is about pulling out as an argument, a symbol used within the body of a function definition.

Depending on which symbol you pull out, you gain different kinds of flexibility.

If we have a function like this -

```
function f(a1,a2,...) { .... S .... }
```

then the act of "beta abstracting on `S`" is rewriting this function as the application of another more abstract function to `S` - i.e.

```
(function g(a1,a2,...) { return function (S) { .... S .... }; })(S)
```

Or to keep it simple, you can just work with a function `g` that is the same as `f`, but with an additional argument `S`.

```
function g(a1,a2,...,S) { .... S .... }
```

The function `g` is considered to be more abstract than the function `f` because we can change what `S` is to achieve different ends.

Now, while many programming language allow you to pull out symbols standing for certain types of "values", they do not allow certain "reserved" symbols to be passed as arguments like this. However, for our purpose, we'll pretend that they do.

If we look at out interpreter, we have a rough structure as follows -

```
function run(env, program, pc, stack) {
for (; pc < program.length; ++pc) {
....
switch (instr.t) {
....
}
}

...
return stack;
}
```

We can abstract on many symbols, each giving us a different kind of "super power". For example, if we pull out `switch`, we gain the ability to customize our interpreter, a feature which we can then reintroduce into `slang` itself, to great effect. (Granted, Javascript doesn't let you pass `switch` as an argument.) If we abstract on the `pc = pc + 1` part, then we gain the ability to jump to different parts of a program, while we can currently only step through sequentially.

If we abstract on `return`, pretending that it is a function to which we pass the result stack as an argument, and that the `return` function itself never "returns" to the call site, we gain another such super power - the ability to direct program flow in a manner that can be exposed to `slang`. As it stands, the `return` statement

does something magical - you use it to specify that "wherever the function `run` gets called, now go back to that piece of code and *continue* with the value I'm giving you as an argument". That's a pretty special "function".

> **Term**: In CS literature, such a "function" is called a **continuation**.

If we pull out `return` as an argument, we gain the ability to pass whatever target function to which the `run` invocation must "return to". So our interpreter loop looks something like this -

```
function run(env, program, pc, stack, ret) {
... same same ...
ret(stack);
}
```

To truly ensure that we benefit from this change though, all the intermediate steps will need to be changed in the same manner to take an extra "return function" argument, where we pass different values depending on where we are. This is left as an exercise.

> **Term**: Such a rewrite of a normal function in terms of functions that take an extra "return function" argument is called in CS literature as "CPS transformation" where "CPS" stands for "Continuation Passing Style". The CPS transformation is a deep transformation of normal sequential code that affects every operation. Some compilers do this as a first step to providing advanced flow control operators, including concurrency.

While programming in the CPS style is tedious and verbose in most languages, supporting CPS in a language can give all sorts of cool super powers - like, for example, the ability to "return" as many times as you want to the call site, the ability to store away the "return function" and call it at a later point in time when a particular event arrives from a user, or the network, and so on.

So we're going to start with rewriting our interpreter to have our `run` function take such an extra "return function" argument that we'll be calling … `callback` :)

> **Note**: To Node.JS junkies, yes it's the same mundane "callback style" that gives you "callback hell" because you're programming in CPS style quite unnecessarily.

8.2 Basic asynchronous behaviour

Before we get to concurrent processes, we'll support asynchronous invocations so that we can at least use slang with Node.JS.

We'll use beta abstraction and modify our `run` function such that it will no longer return a value normally. Instead, it will provide its result via a callback function provided to it. The callback function, if required will need to be passed as the last argument to the `run` function, and must be a function that accepts a stack as its sole argument. At the end of the day, the callback function will be called with the current stack as the sole argument.

```
run = function (env, program, pc, stack, callback) {
    for (; pc < program.length; ++pc) {
        let instr = program[pc];

        if (instr.t === 'word') {
            let deref = lookup(env, instr);
            if (!deref) {
                console.error('Undefined word "' + instr.v + '" at instruction ' + pc);
            }
            instr = deref;
        }
```

```
        switch (instr.t) {
            case 'prim':
```

We identify a primitive that will complete only asynchronously by checking whether its function has arity 1 or 2. If it is 1, it is synchronous, if it is 2, it is asynchronous and the second argument needs to be the callback. If no callback is provided, we assume that all code is synchronous. The only place where this will throw an error is if you do an `await` within a synchronous `run` call.

```
                if (callback && instr.v.length === 3) {
                    return apply(env, instr, stack, function (stack) {
                        return run(env, program, pc+1, stack, callback);
                    });
                } else {
                    stack = apply(env, instr, stack);
                    break;
                }

            case 'block':
                let bound_block = block(instr.v);
                bound_block.bindings = instr.bindings || copy_bindings(current_bindings(env), {});
                push(stack, bound_block);
                break;
```

In all other cases we just store the value on the stack.

```
            default:
                push(stack, instr);
                break;
        }
    }
```

If the caller is thinking that this the run may operate asynchronously, we'll oblige by calling the supplied callback .. but in the next scheduler tick.

```
    if (callback) {
        return later(callback, stack);
    }

    return stack;
};
```

We define `later` to be a function that will postpone the immediate execution of the zero-argument function passed to it.

See slang_later.js for a faster implemetation.

```
let later = function (callback, value) {
    callback && setTimout(callback, 0, value);
    return value;
};
```

We used a four-argument version of `apply`. So we need to change its implementation to support primitives with callbacks too. We simply make use of the Javascript feature which lets us pass as many arguments as we want to any function. If a function receives more arguments than it knows about, it will just ignore them.

```
apply = function (env, prim, stack, callback) {
    console.assert(prim.t === 'prim');
```

```
    return prim.v(env, stack, callback);
};
```

8.3 Cooperative multi-tasking

While the ability to have our interpreter return asynchronously is the key step in our single threaded JS environment to enable concurrency, we need to explicitly use some scheduler in Javascript to enable cooperative multi-tasking between different sequences of interpreters.

We'll therefore need a way to spawn off such asynchronous processes which will not interfere with our spawning process. We'll add a primitive called `go` for that purpose. While `go` itself will be a synchronous operator, it will set up a process which will run at the next asynchronous opportunity. This will work to create concurrency only if there is enough opportunity for asynchronicity in each such process - i.e. if a spawned process doesn't have any async steps in it, it will hog the main loop and complete synchronously.

We first introduce a primitive type for processes. The notify array is intended to hold a bunch of callbacks to be called with the process's result value once the process finishes.

```
let process = function (env, block) {
    return {t: 'process', v: block, state: 'waiting', notify: [], env: env};
};

stddefs(function (env) {
```

Spawning a process will produce a process object on the stack. We can add notifiers to this process object so that the spawning process can await its result.

So, we have now implemented "go routines". We still don't have an error model in slang. We'll come to that. Since this is similar, we'll call the spawning operator as `go`.

```
    define(env, 'go', prim(function (env, stack) {
        let code = pop(stack);
        console.assert(code.t === 'block');
```

When spawning off a new process, we don't want the process to clobber the parent environment or stack. So we copy the environment and pass a fresh empty stack.

```
        let env2 = env.slice(0);
        enter(env2);
        copy_bindings(code.bindings, current_bindings(env2));

        let proc = process(env2, code);
```

A new process will receive a stack with two elements - the parent process, and the process itself. The block is free to pop it off and define it to a variable for multiple access. For example, a block may begin with -

```
[parent-process current-process] args ...
```

Using this technique, you can send/receive messages to yourself, or pass a reference to your process to another process you spawn, to enable two-way communication.

```
        let new_stack = [stack.process, proc];
        new_stack.process = proc;
        return later(function () {
            let env = env2;
            run(env, code.v, 0, new_stack, function (stack) {
```

```
                leave(env);
                proc.state = 'done';
                proc.result = pop(stack);
                if (proc.notify) {
                    for (let i = 0; i < proc.notify.length; ++i) {
                        proc.notify[i](env, proc.result);
                    }
                    proc.notify = [];
                }
            });
        }, push(stack, proc));
    }));
```

Since we're passing the process object as a property of the stack, we'll need to ensure that run function when called raw ensures that such a process object is available.

```
run = (function (old_run) {
    return function (env, program, pc, stack, callback) {
        if (!stack.process) {
            stack.process = process(env, block(program));
        }
        return old_run(env, program, pc, stack, callback);
    };
}(run));
```

Invoking `await` with a process object on the stack will result in the spawning process pausing until the spawned process finishes, and then it will end with the result of the spawned process on the parent process's stack. Notice that we define a two-argument function here, to indicate that `await` is an asynchronous primitive.

In this way, `await` has behaviour similar to "promises" or "futures". When a process is finished, the `await` will always fetch the result of the process, much like the way a promise immediately provides the value that is promised once the process that produces it has completed.

Another term for such an `await` in the concurrency jargon is "join". You may encounter phrases like "fork-join parallelism".

Though we've implemented these concepts in a single threaded system, we're not limited to it, given some basic coordination primitives.

```
define(env, 'await', prim(function (env, stack, callback) {
    let proc = pop(stack);
    console.assert(proc.t === 'process');
    console.assert(callback);
```

The process may be working or may be dead. If it is dead, then we need to immediately succeed with its result value.

```
    if (proc.state === 'done') {
        return later(callback, push(stack, proc.result));
    }
```

The process isn't done yet. So add ourselves to the notify list. Note that the notification callback itself will not immediately invoke the callback function to continue. But it will postpone it to the next scheduler cycle so that all the notifications stand some chance of running, and also we don't blow the javascript stack.

```
    proc.notify.push(function (result) {
        later(callback, push(stack, result));
    });
```

Return value will be discarded in the async case anyway.

```
        return stack;
    }));
```

Though we shouldn't need it in most cases, we'll add a `yield` operator which will give up time for another concurrency process to run. This takes no arguments.

```
    define(env, 'yield', prim(function (env, stack, callback) {
        console.assert(callback);
        return later(callback, stack);
    }));
```

We obviously need the quintessential asynchronous operation of sleeping for a given number of milliseconds! We use the word `after` instead of `sleep` to prevent the misconception that *all* processes would be asleep during this period.

```
    define(env, 'after', prim(function (env, stack, callback) {
        let ms = pop(stack);
        console.assert(ms.t === 'number');
        callback && setTimeout(callback, ms.v, stack);
        return stack;
    }));
});
```

We'll finally modify the main control structures to also support async operation. This includes `do`, `defun`, `if`, `branch` and `vocab`. We have another option here - to create new primitives like `do/async`, `if/async` and so on which will treat their blocks as async blocks. However, we want to avoid that extra hassle. The consequence of this choice is that there will always be a yield at block boundaries.

```
do_block = function (env, stack, block, callback) {
    enter(env);
    copy_bindings(block.bindings, current_bindings(env));
    if (callback) {
        return run(env, block.v, 0, stack, function (stack) {
            leave(env);
            callback(stack);
        });
    } else {
        stack = run(env, block.v, 0, stack);
        leave(env);
        return stack;
    }
};
```

So far, we've had to deal only with one environment. Now that we need to transport vocabularies between environments, we need to respond to the *current* environment when binding symbols to values instead of using the definition environment by default.

```
stddefs(function (env) {
    define(env, 'do', prim(function (env, stack, callback) {
        let code = pop(stack);
        if (code.t === 'prim') {
            return apply(env, code, stack, callback);
        }

        console.assert(code.t === 'block');
```

```
    return do_block(env, stack, code, callback);
}));

define(env, 'defun', prim(function (env, stack) {
    let sym = pop(stack), block = pop(stack);
    console.assert(sym.t === 'symbol');
    if (block.t === 'prim') {
        define(env, sym.v, block);
    } else {
        console.assert(block.t === 'block');
        define(env, sym.v, prim(function (env, stack, callback) {
            return do_block(env, stack, block, callback);
        }));
    }
    return stack;
}));

define(env, 'if', prim(function (env, stack, callback) {
    let blk = pop(stack), cond = pop(stack);
    console.assert(blk.t === 'block');
    console.assert(cond.t === 'bool');
    if (cond.v) {
```

Note that we're again making the choice that the `if` block will get evaluated in the enclosing scope and won't have its own scope. If you want definitions within `if` to stay within the `if` block, then you can make new environment for the block to execute in.

```
            return run(env, blk.v, 0, stack, callback);
        }

        if (callback) {
            return later(callback, stack);
        }

        return stack;
}));

define(env, 'branch', prim(function (env, stack, callback) {
    let cond_pairs = pop(stack);
    console.assert(cond_pairs.t === 'block');
    console.assert(cond_pairs.v.length % 2 === 0);
    if (callback) {
        let step = function (i, callback) {
            if (i >= cond_pairs.v.length) {
                return later(callback, stack);
            }
            console.assert(cond_pairs.v[i].t === 'block');
            console.assert(cond_pairs.v[i+1].t === 'block');
            return run(env, cond_pairs.v[i].v, 0, stack, function (env, stack) {
                let b = pop(stack);
                console.assert(b.t === 'bool');
                if (b.v) {
                    run(env, cond_pairs.v[i+1].v, 0, stack, callback);
                } else {
```

```
                        step(i+1, callback);
                    }
                });
            };
            return step(0, callback);
        } else {
            for (let i = 0; i < cond_pairs.v.length; i += 2) {
                console.assert(cond_pairs.v[i].t === 'block');
                console.assert(cond_pairs.v[i+1].t === 'block');
```

Check the condition.

```
                stack = run(env, cond_pairs.v[i].v, 0, stack);
                let b = pop(stack);
                console.assert(b.t === 'bool');
                if (b.v) {
```

Condition satisfied. Now evaluate the "consequence" block corresponding to that condition.

```
                    return run(env, cond_pairs.v[i+1].v, 0, stack);
                }
            }
            return stack;
        }
    }));

    define(env, 'vocab', prim(function (env, stack, callback) {
        let defs = pop(stack);
        console.assert(defs.t === 'block');
```

Execute the block and capture its definitions before we leave it.

```
        enter(env);
        if (callback) {
            return run(env, defs.v, 0, stack, function (stack) {
                let bindings = copy_bindings(current_bindings(env), {});
                leave(env);
                delete bindings[parent_scope_key];
                callback(push(stack, vocab(bindings)));
            });
        } else {
            stack = run(env, defs.v, 0, stack);
            let bindings = copy_bindings(current_bindings(env), {});
            leave(env);
```

We don't want to preserve the scope chain in this case, so delete the parent scope entry.

```
            delete bindings[parent_scope_key];

            return push(stack, vocab(bindings));
        }
    }));
});
```

8.4 Communicating between processes

Date: 11 Apr 2017

So far, we can spawn a process and the only way we can interact with it is to await its completion, upon which we'll get passed the result of that process on our stack.

We want to do more. Maybe we want a process that will never complete, but functions like a "server" which will receive requests from a port and keep responding to them.

To get this capability, we'll add a "mailbox" to each process. The mailbox canbe used to send and receive messages to the process. Since each process automatically gets a process-wide mailbox, we don't usually need anything more.

```
process = function (env, block) {
    return {t: 'process', v: block, state: 'waiting', notify: [],
            mailbox: mailbox(), env: env, pid: process.pid++};
};

process.pid = 1;
```

We'll introduce a slang type to represent the mailbox. We'll later transform this into something that can be used at the language level too.

```
let mailbox = function () {
    return {t: 'mailbox', v: [], receiver: null};
};

let mailbox_post = function (mb, msg) {
    mb.v.push(msg);
    if (mb.receiver) {
        later(mb.receiver);
        mb.receiver = null;
    }
    return msg;
};

let mailbox_receive = function (mb, stack, callback) {
    if (mb.v.length > 0) {
        let msg = mb.v.shift();
        mb.receiver = null;
        return later(callback, push(stack, msg));
    }

    mb.receiver = function () {
        let msg = mb.v.shift();
        callback(push(stack, msg));
    };

    return stack;
};
```

We need facilities to post messages to a process and for a process to receive and … um … process messages from another process.

```
stddefs(function (env) {
```

Posting a message to a process is a synchronous operation that places the message into the process' mailbox.

Usage: `proc msg post`

```
    define(env, 'post', prim(function (env, stack) {
        let msg = pop(stack), proc = pop(stack);
        console.assert(proc.t === 'process');
        mailbox_post(proc.mailbox, msg);
        return stack;
    }));
```

Receiving a message will wait for a messages to arrive into the current process' mailbox, and will place it on the stack when it does. Further code can then examine the message and take appropriate actions.

Usage: `receive`

```
    define(env, 'receive', prim(function (env, stack, callback) {
        let proc = stack.process;
        console.assert(proc.t === 'process');
        mailbox_receive(proc.mailbox, stack, callback);
        return stack;
    }));
```

Sometimes, we may not be able to process a received message until another one arrives, since messages arrive non-determinstically. In such cases, we'd need to postpone a message back into the process' mailbox.

Usage: `msg postpone`

```
    define(env, 'postpone', prim(function (env, stack) {
        let msg = pop(stack);
        console.assert(proc.t === 'process');
        console.assert(stack.process);
        stack.process.post(msg);
        return stack;
    }));
});
```

8.5 Processes as "objects"

If you recall, we said in the section on object oriented programming that that main concept behind OOP is objects which interact by passing messages between them. We highlighted at that point that there is no intrinsic requirement to make such message passing synchronous in the way most OOP languages including the seminal Smalltalk implement.

We now have processes that can pass messages between each other asynchronously - i.e. when a process passes another a message, it doesn't expect to get a "return value" immediately. Instead it is free to do other activities and *maybe* ask for one or more reply messages from the target process.

So, in a sense, our "processes" are better "objects" than our "objects". About the only programming language and runtime where this interpretation is close to feasible is Erlang - which doesn't have a notion of "objects", but has cheap isolated concurrent processes which can communicate with each other through message passing. If you think about it, objects in our real world are always concurrent - they have a clearly defined interaction boundary and have their own timeline of activities they may be engaged in. A clock keeps ticking, a fan keeps spinning, the TV keeps showing a video on a screen, the grinder keeps mashing up food, and so on. Very rarely do we have synchronicity among objects in our real world. One example is perhaps an electrical switch - whose response to making or breaking an electrical circuit is immediate .. for practical purposes.

8.6 Supporting objects

Let's return to our mundane symchronous world of objects for now.

We implemented message passing as method invocation in our object system. That doesn't yet support asynchronous operation. We'll need to fix that before we can use objects in this new circumstance.

We can add general asynchronous support to `send`, `send*` and `new` (for async constructor), but instead of doing that, we'll add a new message sending operator that will work asynchronously. We'll call them `send&` and and `send*&`. We'll use the `&` character to indicate that the message sending is being "backgrounded" like you might do in a shell.

Now, we have many choices here. We can simply implement async support for `send&` and `send*&`, but that is not a true message send because we'll have to wait for the send to finish before doing anything else. This prevents us from doing a sequence of message sends to various objects at one go. It also forces async operation for every object message send.

A better way to look at async object message sends is as though you're sending to another process, in which case if you intend to receive a message back, you'll send along your process id so that the process can communicate back to you whatever it needs to … or not. From the perspective of the sending process, the sending step appears synchronous, just like a `post`.

So an asynchronous message send to an object is a method invocation that accepts the sending process as the top-stack argument, after which the regular `send` arguments appear.

> **Note**: This means the appropriate vocabularies must be designed for such asynchronicity. No result should be expected from such an async send on the stack and the design of an async vocabulary should be such that it should pass results only by posting to the provided process object.

```
stddefs(function (env) {
    define(env, 'send&', prim(function (env, stack) {
        let msg = pop(stack), thing = pop(stack);
        console.assert(msg.t === 'symbol');

        if (!thing.vocab || !thing.vocab.v[msg.v]) {
            console.error('No vocabulary relevant to message');
            return stack;
        }

        let method_or_val = thing.vocab.v[msg.v];
        if (!method_or_val) {
            console.error('Vocabulary doesn\'t accept message "' + msg.v + '"');
            return stack;
        }

        switch (method_or_val.t) {
            case 'prim':
                push(stack, thing);
                push(stack, stack.process); // Extra process argument.
                return apply(env, method_or_val, stack);
            default:
                return push(stack, method_or_val); // Normal value.
        }
    }));

    define(env, 'send*&', prim(function (env, stack) {
        let msg = pop(stack), voc = pop(stack);
```

```
        console.assert(msg.t === 'symbol');
        console.assert(voc.t === 'vocab');

        let method_or_val = voc.v[msg.v];
        if (!method_or_val) {
            console.error('Vocabulary doesn\'t accept message "' + msg.v + '"');
            return stack;
        }

        if (method_or_val.t === 'prim') {
            push(stack, stack.process); // Extra process argument.
            return apply(env, method_or_val, stack);
        }

        return push(stack, method_or_val);
    }));
});
```

8.6.1 The trouble with such objects

Beyond this, we'll find that working with objects in an environment where such concurrency has prolific use is extremely hard to get right. In particular, designing a transport mechanism for objects that actually works is very hard. At best, objects can be exposed via communication port. In this sense, processes serve as better "objects" than objects themselves.

We'll satisfy ourselves with what we have for the moment and dive deeper into how to setup interplay between objects and concurrency later on.

8.7 Tests

8.7.1 Some debug utilities

```
stddefs(function (env) {
```

The top of the stack is expected to contain a message string that will be gobbled. The next stack item will be printed without popping it off the stack.

```
    define(env, 'debug', prim(function (env, stack) {
       let str = pop(stack).v;
       console.log(str, topitem(stack).v);
       return stack;
    }));
```

A `breakpoint` is useful to invoke the Javascript debugger's breakpoint without having any other impact on program behaviour.

```
    define(env, 'breakpoint', prim(function (env, stack) {
        debugger;
        return stack;
    }));
});
```

8.7.2 Test: Count down timer

Basic asynchronicity test where we run a count-down timer while the code looks like a normal recursive function.

```
tests.ticktock = function (n) {
    let program = parse_slang(`
        [ :i def
          i 0 >
          [ i print 1000 after
            i 1 - count-down
          ] if
        ] :count-down defun
        count-down
    `);

    return run(test_env(), program, 0, [number(n)], function (stack) {
        console.log("done");
    });
};
```

8.7.3 Test: Message passing

```
tests.passmsg = function () {
    let program = parse_slang(`
        [ "STARTED" debug
          "one" print receive print
          1000 after
          "two" print receive print
          1000 after
          "three" print receive print
          1000 after
          "four" print receive print
          1000 after
        ] go :target def
        target "ondru" post
        target "irandu" post
        target "moondru" post
        target "naangu" post
    `);

    return run(test_env(), program, 0, [], function (stack) {
        console.log("done");
    });
};
```

8.7.4 Test: Ping Pong

A basic ping pong messaging between two processes. The main process launches two processes first, it then tells them about each other and what messages they must print, and then gives the go to one of them who then starts the back-n-forth (pun unintentionally intended ;)

```
tests.pingpong = function (n) {
    let program = parse_slang(`
        :n def
        [ "PINGPONG" print

          "The first two messages tell us who we are
          and who to send messages to.";

          receive :target def
          receive :message def

          [ receive :i def
            message print 1000 after
            i 0 > [ target i 1 - post loop ] if
          ] :loop defun

          loop
        ] :pingpong def

        pingpong go :ping def
        pingpong go :pong def

        ping pong post     "Tell ping that pong is its target";
        ping "ping" post
        pong ping post     "Tell pong that ping is its target";
        pong "pong" post

        ping n post        "Start pingponging";
    `);

    return run(test_env(), program, 0, [number(n)], function (stack) {
        console.log("done");
    });
};
```

8.8 Scoping rules for concurrency

We've used blocks to represent the code that processes should run. So far, so good. If we want to expand the reach of processes to cover running them on other machines, then we need to change the meanings of our programs a bit to accommodate the fact that what is referred to within such blocks cannot be modified *after* the block is created. The reason for that is that, if the normal expectation is have bindings be modifiable after block creation, then the same expectation needs to be maintained when sending a block over to another machine for running there. That would, however, entail that variable assignments trigger network communication to the other machine.

We *could* implement such network communication when such "remote variables" change values, but then the programmer ceases to have control over the meanings of these variables because they may change at any point *within* a process. That is, within a process, you'll no longer be able to assume that the value of a captured variable will be stable within the process. Such unpredictability leads to considerable system complexity and bugs due to lapses in accounting for such changes.

To change this behaviour, we need to implement a different mechanism for *how* to save bindings at block creation time. One heuristic is to scan the block recursively for words and for any word that occurs, store the binding in the block's captured bindings. Below is an implementation of this scheme.

Note: With this change, we're significantly changing the meanings of our earlier programs in a subtle way. The reader's task is to revisit the earlier test cases and consider whether they are influenced by this change or not.

```
let copy_bindings_for_block = function (block, env, dest) {
    let scan = function (instructions) {
        for (let i = 0; i < instructions.length; ++i) {
            let word = instructions[i];
            if (word.t === 'word') {
                let val = lookup(env, word);
                if (val) {
                    dest[word.v] = val;
                }
            } else if (word.t === 'block') {
                scan(word.v);
            }
        }
    };

    scan(block.v);
    dest['self'] = block; // The word "self" refers to the block itself within the block.
    return dest;
};
```

Question: Scrutinize the above heuristic for determining what bindings to keep from the environment. In what cases does it keep more bindings than necessary? Are there cases where it may keep fewer bindings than necessary?

```
run = function (env, program, pc, stack, callback) {
    if (!stack.process) {
        stack.process = process(env, block(program));
    }

    for (; pc < program.length; ++pc) {
        let instr = program[pc];

        if (instr.t === 'word') {
            let deref = lookup(env, instr);
            if (!deref) {
                console.error('Undefined word "' + instr.v + '" at instruction ' + pc);
            }
            instr = deref;
        }

        switch (instr.t) {
            case 'prim':
                if (callback && instr.v.length === 3) {
                    return apply(env, instr, stack, function (stack) {
                        return run(env, program, pc+1, stack, callback);
                    });
                } else {
                    stack = apply(env, instr, stack);
                    break;
                }
```

```
            case 'block':
                let bound_block = block(instr.v);
```

We use `copy_bindings_for_block` instead of the earlier `copy_bindings` implementation. With this implementation, notice that the `[[parent]]` entry will **not** be copied. This has the effect of isolating the block's execution from the rest of the environment chain. This is our first baby step towards "process isolation", given that we intend to reuse blocks to describe processes.

```
                bound_block.bindings = instr.bindings || copy_bindings_for_block(bound_block, env, {});
                push(stack, bound_block);
                break;
```

In all other cases we just store the value on the stack.

```
            default:
                push(stack, instr);
                break;
        }
    }

    if (callback) {
        return later(callback, stack);
    }

    return stack;
};
```

We have to now examine the impact of this change on how we're spawning processes - using our `go` primitive. `go` currently copies the environment chain by reference - i.e. it isn't a deep copy. Now that the process of creating a block already captures whetever is necessary for the block to run, we can safely change the implementation of `go` to create a fresh new environment instead. This is also a significant step towards "process isolation".

```
stddefs(function (env) {
    define(env, 'go', prim(function (env, stack) {
        let code = pop(stack);
        console.assert(code.t === 'block');
```

When spawning off a new process, we don't want the process to clobber the parent environment or stack. So we create a *fresh* environment and store whatever the block needs in it. Note that only the creation of `env2` is different from the previous implementation, where we did `env.slice(0)`. We can change that to make a *blank* environment instead because everything that is needed to evaluate the block has already been captured into the `.bindings` hash. If that were not true, then this simplification won't be possible. That capturing step flattens the entire environment stack into a single hash.

```
        let env2 = mk_env();
        copy_bindings(code.bindings, current_bindings(env2));
        enter(env2);
```

> **Question**: What are the implications of this choice from the perspective of processes distributed across multiple machines, running within different interpreter processes? Is there anything that processes cannot do any more with this implementation that was possible before, for example?

```
        let proc = process(env2, code);
```

The rest remains the same.

```
        let new_stack = [stack.process, proc];
        new_stack.process = proc;
```

```
        return later(function () {
            let env = env2;
            run(env, code.v, 0, new_stack, function (stack) {
                leave(env);
                proc.state = 'done';
                proc.result = pop(stack);
                if (proc.notify) {
                    for (let i = 0; i < proc.notify.length; ++i) {
                        proc.notify[i](proc.result);
                    }
                    proc.notify = [];
                }
            });
        }, push(stack, proc));
    }));
});
```

One thing to notice with this new notion of capturing the bindings relevant to a block is that we can now simplify our implementation of environment with no change to behaviour. We can just maintain a single chain of scopes.

```
mk_env = function (base) { return { t: 'env', v: { env: {}, base: base && base.v } }; };

lookup = function (env, word) {
    for (let scope = env.v; scope; scope = scope.base) {
        let val = scope.env[word.v];
        if (val) { return val; }
    }
    return undefined;
};

define = function (env, key, value) {
    env.v.env[key] = value;
    return env;
};

enter = function (env) {
    env.v = { env: {}, base: env.v };
    return env;
};

leave = function (env) {
    env.v = env.v.base;
    return env;
};

current_bindings = function (env) {
    return env.v.env;
};
```

8.9 Data flow variables

Mutexes, semaphores and critical sections are common ways in which systems programming languages deal with concurrency control. Recently, promises and futures have turned up as useful abstractions in a variety of situations. A promise or a future is an immediate value that stands for the value that a process will *eventually* produce. Whenever a process needs the value of an unbound dataflow variable, it will suspend and wait for it to be bound in some other process. This way, two processes can coordinate their activities by synchronizing on such dataflow variables.

Let's try to model these in slang by introducing a new type for dataflow variables. With each `dfvar`, we need to store a list of callbacks to call when the dfvar becomes bound so that processes waiting on it can resume. We also keep around the name of the dfvar just for debugging purposes.

```
let dfvar = function (name) {
    return { t: 'dfvar', v: undefined, name: name, bound: false, resume: [] };
};

let df_is_bound = function (dfv) { return dfv.bound; };

let df_bind = function (dfv, val) {
    if (!dfv.bound) {
        dfv.v = val;
        dfv.bound = true;
    }
    while (dfv.resume.length > 0) {
        let fn = dfv.resume.shift();
        later(fn, dfv.v);
    }
    return dfv.v;
};

let df_val = function (dfv, callback) {
    if (dfv.bound) {
        return later(callback, dfv.v);
    }

    dfv.resume.push(callback);
    return dfv;
};
```

When a process tries to access the value of a dataflow variable that is not bound, it must suspend until it gets a value. We do this via the traditional `await` route.

We of course need a way for a process to bind an unbound dataflow variable with a value. We must insist that we do this only for unbound variables, or we'll end up with an asynchronous mess. We'll simply reuse `def` to include dataflow variables in the mix instead of introduce another word. We'll of course add a way to create new dataflow variables using a `dfvar` primitive. To wait for a dfvar to be bound, we'll repurpose `await`. This is particularly appropriate as the behaviour of await in the case of a process is similar - where it waits for a process to finish (if it is running) and resumes with the value it places on the top of the stack. If the process is already finished, then it resumes immediately. Similarly, if the dfvar is unbound, it waits for the dfvar to be bound by another process before continuing. If it is bound, then it replaces it by its value on the stack and continues.

> **Question**: What kind of a concurrency "mess" would we end up with if we permit bindings for already bound dataflow variables?

```
stddefs(function (env) {
    define(env, 'dfvar', prim(function (env, stack) {
        let sym = pop(stack);
        console.assert(sym.t === 'symbol');
        define(env, sym.v, dfvar(sym));
        return stack;
    }));

    define(env, 'def', prim(function (env, stack) {
        let sym = pop(stack), val = pop(stack);
        switch (sym.t) {
            case 'symbol':
                define(env, sym.v, val);
                break;
            case 'dfvar':
                df_bind(sym, val);
```

TODO: Note that we must fail in this case if the new value is not the same as an already bound value if that was the case.

```
                break;
            default:
                console.error('Unsupported binding type ' + sym.t);
        }
        return stack;
    }));

    define(env, 'await', prim(function (env, stack, callback) {
        let proc = pop(stack);

        if (proc.t === 'dfvar') {
            console.assert(callback);
            df_val(proc, function (val) {
                callback(push(stack, val));
            });
            return stack;
        }

        console.assert(proc.t === 'process');
        console.assert(callback);

        if (proc.state === 'done') {
            return later(callback, push(stack, proc.result));
        }

        proc.notify.push(function (result) {
            later(callback, push(stack, result));
        });
```

Return value will be discarded in the async case anyway.

```
        return stack;
    }));
});
```

```
tests.dfvar = function () {
    let program = parse_slang(`
        "We'll use a convention that capital letters indicate dfvars";
        :X dfvar
        [ 5000 after 42 X def ] go
        "Waiting for X" print
        X await print
    `);

    return run(test_env(), program, 0, [], function (stack) {
        console.log("done");
    });
};
```

Question: What do we have to do to implement dataflow variables such that we won't need `await` to get the value of such a variable? How would you design your primitives to support the creation of strcutures with unfulfilled dfvars that await their values only when requested?

Question: Supposing a dfvar never gets bound and a process is waiting for it. What facilities would we need to help design the process so that it won't be locked forever when such a thing happens?

8.10 Tests

8.10.1 Test prime number sieve

```
tests.primesieve = function (n) {
    let program = parse_slang(`
        [n] args                "We stop when we reach n";

        "We spawn one sieve process for each prime number we
         find. Each sieve process filters out the factors of
         that prime number and pipes its output to the higher
         prime processes.";

        [ receive :prime def    "The first number we get is prime";
          prime print
          self go :filter def   "Launch another filter process for
                                 the next prime";

          "Sieve out all factors of our prime and send it to
           the next sieve process";

          [ self :loop def
            receive :i def      "i will not have any factors < prime";
            i prime remainder 0 != [ filter i post ] if

            i n < [ loop do ] if
          ] do
        ] :sieve def

        sieve go :main def      "Start the first sieve";
```

```
        "Generate 2,3,4,5,... into the first sieve";
        [ [i target] args
          self :gen def
          target i post
          yield                 "Necessary for async generation";
          i n < [i 1 + target gen do] if
        ] :generate defun

        2 main generate
    `);

    return run(test_env(), program, 0, [number(n)], function (stack) {
        console.log("done");
    });
};
```

8.10.2 Math primitives needed for prime sieve

```
stddefs(function (env) {
    define(env, 'remainder', prim(function (env, stack) {
        let den = pop(stack), num = pop(stack);
        console.assert(den.t === 'number' && num.t === 'number');
        return push(stack, number(num.v % den.v));
    }));

    define(env, 'quotient', prim(function (env, stack) {
        let den = pop(stack), num = pop(stack);
        console.assert(den.t === 'number' && num.t === 'number');
        return push(stack, number(Math.floor(num.v / den.v)));
    }));
});
```

9 Non-deterministic programming

Date: 18 Apri 2017

(Requires slang_concurrency.js and whatever it requires.)

Now that we have control over control flow within "processes", we can start to play with it in ways we couldn't imagine doing in our base language.

Non-deterministic programming refers to programming operations with "choice points" such that at the time a choice is being made, we don't know which choice will succeed, as that is expected to be determined later on as the program continues to run. As the program runs, a particular choice may end up being seen as inappropriate and the program then "back tracks" to try another choice.

```
let failure = function (val) { return {t: 'fail', v: val}; };
```

9.1 The `choose` and `fail` operators

```
stddefs(function (env) {
```

The main ingredients of non-deterministic programming are a "choose" operator which choose one of several code paths in such a way that the whole program tries not to "fail". Correspondingly, there is a "fail" operator which informs the history of choice points whether the current code path satisfies the program's needs or not.

While this doesn't look very different from branching at the outset, the critical difference is that it is a runtime choice and that the criterion for choosing is not available to the function at the time it has to make a choice. This information is only available later on at a higher level in the program. Therefore with the `choose` operator, functions can now be designed in such a way that they can produce more than one possible outcome. In mathematical terms, the `choose` operator permits creating functions that map one set of inputs to more than one possible output - i.e. a one-to-many mapping. The value in such a one-to-many mapping is that the combinations of such choices that result in the satisfaction of some globally determined constraints can now be the result of the program, without the functions being forced to determine things that they are not well placed to determine.

Our `choose` operator takes a sequence of blocks or values and picks one that will cause the rest of the program to succeed if possible … or it will fail to an earlier choice point. This gives us "depth-first search" of the choice tree for possible solutions.

```
define(env, "choose", prim(function (env, stack, callback) {
    let options = pop(stack);
    let proc = stack.process;

    console.assert(callback);
    console.assert(options.t === "block");

    if (!proc.choice_points) {
        proc.choice_points = [];
    }

    let current_env = mk_env(env);
```

We model "making a choice" as a function that takes a single integer value representing which option among the ones it has been supplied with must be tried. When the function is called, it will assume that the choice corresponding to the integer supplied is available. We keep all of this as a stack of choice points.

```
    proc.choice_points.push({
        i: 0,
        count: options.v.length,
        fn: function (i) {
```

Fresh environment and stack, discarding all the other things possibly accumulated in prior choices.

```
            let cp_env = mk_env(current_env);
            let cp_stack = stack.slice(0);
            cp_stack.process = stack.process;
```

If given a block, we evaluate it. If given any other normal value, we push it on to the stack.

```
            if (options.v[i].t === "block") {
                run(cp_env, options.v[i].v, 0, cp_stack, callback);
```

Question: What happens if the block we're running itself creates choice points or triggers a fail? Does that need specific support in our code? Would it behave correctly? If not what kinds of

incorrect behaviour may result?

```
            } else {
                callback(push(stack, options.v[i]));
            }
        }
    });

    return back_track(stack, callback);
}));
```

Question: How will you implement a "breadth-first" search strategy? More generally, in the spirit of making implementation features available to our language itself, how can we make such search strategies programmable?

As a task, you can take on rewriting `choose` and `fail` to work using a breadth-first strategy, or add a new operator to use the breadth-first strategy.

Our `fail` operator just triggers the back tracking mechanism to try alternative choices. Typically, you'd use `fail` within some kind of a conditional so that the failure is triggered only when some condition isn't met.

```
define(env, "fail", prim(function (env, stack, callback) {
    return later(function (stack) { back_track(stack, callback); }, stack);
}));
```

9.2 Depth-first back tracking

The act of picking a choice involves checking a choice point and if it is exhausted, moving on to earlier choice points, and continuing that until we exhaust all choice points … at which point we give up.

```
function back_track(stack, callback) {
    let proc = stack.process;
    if (proc.choice_points.length === 0) {
```

All choices exhausted. Whole program failure.

```
        return later(callback, push(stack, failure(symbol('choose'))));
    }

    let choice_point = proc.choice_points[proc.choice_points.length - 1];
    if (choice_point.i < choice_point.count) {
        later(choice_point.fn, choice_point.i);
        choice_point.i++;
```

If we've initiated the last choice, we might as well remove the choice point from the back tracking history right away.

Question: Does this help? If so, how? If not, why?

```
        if (choice_point.i >= choice_point.count) {
            proc.choice_points.pop();
        }
    } else {
        proc.choice_points.pop(); // Choices exhausted.
        later(function () {
            back_track(stack, callback);
        });
    }
```

```
        return stack;
    }
});
```

9.3 Tests

9.3.1 Test: Constraints on two choice points

This code creates two choice points and the rest of the program decides to fail if the numbers chosen by these choice points don't satisfy some numeric criteria.

```
tests.two_constraints = function () {
    let program = parse_slang(`
        [1 2 3 4 5] choose :x def
        [1 2 3 4 5] choose :y def
        x y + 5 < [fail] if
        x print y print
        x y * 15 < [fail] if
        "result" print
        x print y print
    `);

    return run(test_env(), program, 0, [], function (stack) {
        console.log("done");
    });
};
```

> **Question**: How will you implement an operator that collects all possible "solutions" in the above example instead of just picking one?
>
> **Question**: Can you write a "choice point generator" that will try all natural numbers? How would you prevent the generation of infinite useless choices to try?

9.4 Non-determinism and data-flow-variables

We specified "data flow variables" to be analogous to boxes that can be filled only once with a value. If we introduce choice points affecting the contents of data flow variables, then the side effect of filling these boxes ripples over to other processes that share the DFVs.

> **Question**: How would the semantics of data flow variables work if the choice operator were to account for them too? In particular, how would invalidating the contents of a data flow variable in one process impact another process that doesn't have any choice points, but has proceeded because one DFV it was waiting on got filled?

In general, such choice points do not work well with operators that have side effects. At other times, the side effects are useful programming aids too. So actually exploiting such choice points in production code hasn't seen as wide an adoption as it might have gotten, had the separation of *specifying* side effecting actions from their actual performance to cause the side effects become more common.

> **Question**: What language feature that you're already familiar with do such choice points remind you of?

10 Finite domain constraint programming

Date: 18 April 2017
Note: DRAFT MODE. This section is INCOMPLETE.

Requires slang_nondet.js and whatever it requires.

In the previous section on non-deterministic programming, we saw how the `choose` operator can be used to generate possibilities and the `fail` operator can be used to limit these possibilities. This style of programming where you specify a problem not in terms of how to solve it, but in terms of what constraints must be met is referred to as "constraint programming". One particularly useful branch of this is "finite domain constraint programming", where the "domain" of variables can take on values only from a finite set. In particular, we can model such finite sets as variables which can take on a finite number of integer values.

To start with, we'll first model these "finite domain variables" as sets of integers. While a set is a powerful structure, we'll keep it simple here by using an integer so that our "sets" can have a maximum cardinality of 30 - i.e. we only permit integers in the range 0 to 29 (inclusive). Extending this with a data structure that supports larger finite domains is left as an exercise to the reader. The point of this section is to illustrate how to construct constraint solvers with programmable strategies.

```
let fdvar = function (dom) {
    if (typeof dom === 'number') {
        return {t: 'fdvar', v: dom}; // `dom` is a bit field.
    }
```

`dom` is an array of pairs of integers - like this - [[2,4],[7,13]] which mean the number from 2 to 4 (inclusive) and from 7 to 13 are to be included in the set.

```
    let bdom = 0; // The bit field.

    for (let i = 0; i < dom.length; ++i) {
        for (let j = dom[i][0]; j <= dom[i][1]; ++j) {
            bdom += 1 << j;
        }
    }

    return {t: 'fdvar', v: bdom};
};

let fd_const = function (n) {
    console.assert(n >= 0 && n < 30);
    return fdvar(1 << n);
};

let fd_bool = function () {
    return fdvar(3);
};
```

10.1 Basic operations of FDVars

We need some basic operations on these finite domain variables. At the minimum, we need union and intersetion of these sets, and to be able to work with them like sets.

```
let fd_universal = 1073741823;

let fd_union = function (v1, v2) {
```

```
    return fdvar(v1.v | v2.v);
};

let fd_intersection = function (v1, v2) {
    return fdvar(v1.v & v2.v);
};

let fd_complement = function (v) {
    return fdvar(fd_universal & ~v.v);
};
```

10.2 Finite domain constraints

Given that our variables are numeric, we can define numeric constraint operators on them. For example, "a < b" can e interpreted as a constraint on the two fdvars `a` and `b` such that they can only take on values that satisfy the constraint. Therefore, after such a constraint function is called, the values of both the fdvars may be modified to reflect the constraint.

10.2.1 a < b

Convention is that the last argument is the one that will be affected by the constraint. Implements v1 < v2.

```
let fdc_lt = function (v2, v1) {
    for (let i = 29; i >= 0; --i) {
        if (v2.v & (1 << i)) {
```

We have the highest set bit. Forbid all values above this for v1.

```
            v1.v &= fd_universal & ~((1 << i) - 1);
            break;
        }
    }
};
```

10.2.2 a == b

```
let fdc_eq = function (v1, v2) {
    v1.v = v2.v = (v1.v & v2.v);
};
```

10.2.3 a <= b

As per convention, implements v1 <= v2 where v1 is the last argument.

```
let fdc_lte = function (v2, v1) {
    for (let i = 29; i >= 0; --i) {
        if (v2.v & (1 << i)) {
```

We have the highest set bit. Forbid all values above or equal to this for v1.

```
            v1.v &= fd_universal & ~((1 << (i+1)) - 1);
            break;
        }
```

```
    }
};
```

10.3 Arithmetic constraints

Arithmetic constraints are somewhat complicated. They deal with three finite domain variables, ensuring that some arithmetic relation holds between them. Their constraining must necessarily affect all three variables. While that is true of general constraints, we're implementing primitive constraints here that constrain only the last argument.

10.3.1 a + b = c

```
let fdc_sum = function (a, b, c) {
    let cposs = 0;
```

This is a really stupid and inefficient algorithm intended to illustrate how the constraining is done.

```
    for (let i = 0; i < 30; ++i) {
        for (let j = 0; j < 30; ++j) {
            if ((a.v & (1 << i)) && (b.v & (1 << j))) {
                if (i + j < 30) {
                    cposs += (1 << (i + j));
                }
            }
        }
    }

    c.v &= cposs;
};
```

10.3.2 a - b = c

```
let fdc_sub = function (a, b, c) {
    let cposs = 0;
```

This is a really stupid and inefficient algorithm intended to illustrate how the constraining is done.

```
    for (let i = 0; i < 30; ++i) {
        for (let j = 0; j < 30; ++j) {
            if ((a.v & (1 << i)) && (b.v & (1 << j))) {
                if (i - j >= 0 && i - j < 30) {
                    cposs += (1 << (i - j));
                }
            }
        }
    }

    c.v &= cposs;
};
```

10.3.3 a * b = c

```
let fdc_prod = function (a, b, c) {
    let cposs = 0;
```

intended to illustrate how all three variables are constrained by the arithmetic condition.

```
    for (let i = 0; i < 30; ++i) {
        for (let j = 0; j < 30; ++j) {
            if ((a.v & (1 << i)) && (b.v & (1 << j))) {
                if (i * j < 30) {
                    cposs += (1 << (i * j));
                }
            }
        }
    }

    c.v &= cposs;
};
```

10.3.4 a / b = c

```
let fdc_div = function (a, b, c) {
    let cposs = 0;
```

intended to illustrate how all three variables are constrained by the arithmetic condition.

```
    for (let i = 0; i < 30; ++i) {
        for (let j = 0; j < 30; ++j) {
            if ((a.v & (1 << i)) && (b.v & (1 << j))) {
                if (i % j === 0 && i / j < 30) {
                    cposs += (1 << Math.round(i / j));
                }
            }
        }
    }

    c.v &= cposs;
};
```

10.3.5 a != b

This is an interesting case, because we can only do some constraining if one of the fdvars has a domain of cardinality 1.

```
let fdc_neq = function (a, b) {
    if (a.v & (a.v - 1) === 0) {
        b.v = b.v & ~a.v;
    }
};
```

10.4 Propagators

So far, we've introduced "constraints" which work to reduce the domains of one or more variables participating in the constraint. In a real problem, however, we have a network of these constraints. For example, we may have "a + b > 10" and "a - b > 3" both needing to be satisfied. If we evaluate these constraints in any one particular order, we may end up in a situation where one constraint affects another variable which in turn can help reduce another variable via some other constraint.

Constraints, therefore, need to be treated as a network and variable domain reductions must *propagate* through this network whenever some domain gets reduced.

We can model such propagators as processes which continuously maintain constraints on their dependent variables. When one propagator acts to reduce the domain of one of its variables, it triggers other propagators attached to that variable to try to reduce domains as well. Finally, all variables will settle to stable or failed domains, at which point we need to decide to do something about it outside the context of propagators. To start with, though, propagators are processes that continuously reinforce a constraint between their dependent variables.

For another example, our primitive `fdc_neq` constraint only constrains the second argument, but if after constraining the second argument, it becomes a singleton set, then ideally it should be used to constrain the first argument too. This triggering behaviour is what is implemented using propagators propagating constraints through a dependency network.

```
let propagator = function (constraint, variables) {
    let output = variables[variables.length - 1];
    let prop = {
        t: 'propagator',
        v: output,
        variables: variables,
        constraint: constraint,
        run: function () {
            let old_val = output.v;
            constraint.apply(this, variables);
            let new_val = output.v;
            return old_val !== new_val ? 1 : 0
        }
    };
```

Store a reference to the propagator in the variables that must trigger it when they change.

```
    for (let i = variables.length - 2; i >= 0; --i) {
        variables[i].prop = (variables[i].prop || []);
        variables[i].prop.push(prop);
    }

    return prop;
};
```

A simple loop for running constraint propagation until everything settles. `variables` is an array of variables to consider. Returns 'stable' or 'failed'.

```
let propagate = function (variables) {
```

We initially mark all variables as changed, so that we don't omit any propagators.

```
    for (let i = 0; i < variables.length; ++i) {
        variables[i].changed = 1;
    }
```

```
    let changes = 0;
```

We keep attempting to propagate changes until there are no changes.

```
    do {
        changes = 0;

        for (let i = 0; i < variables.length; ++i) {
            if (variables[i].changed) {
```

Every time we trigger a variable's propagators, we decrement its change count to account for it.

```
                variables[i].changed--;
                let props = variables[i].prop;
                for (let i = 0; i < props.length; ++i) {
                    let change = props[i].run();
                    props[i].v.changed += change;
                    changes += change;
                }
            }
        }
    } while (changes);
```

We'll come here once no variables change during one iteration. This could happen because the propagators couldn't proceed further, or because they've reached a point of no solution - i.e. at least one of the variables has a zero-sized domain.

```
    for (let i = 0; i < variables.length; ++i) {
        if (variables[i].v === 0) {
```

Failed propagation. Overconstrained.

```
            return 'failed';
        }
    }

    return 'stable';
};
```

11 Error conditions

Date: 26 April 2017
Note: DRAFT MODE. This section is INCOMPLETE.

Requires `slang_nondet.js` and everything else it requires.

In the section on "non-deterministic programming", we saw how the `choose` and `fail` operators can cooperate to implement a depth-first search of a tree of possible code paths in order for the whole program to meet some stipulated success criterion. While this is not the common way to approach successful program completion, good system design involves thinking through and accounting for all the kinds of error conditions that can occur in the course of its lifetime. The aim of this section is to give an overview of error management design choices made in various programming languages so you-the-reader can have some idea of the space of possibilities as well as the rationale for each.

11.1 What is an error condition?

At the basic level, an error condition occurs when a function is unable to return a value that will continue the flow of the program. It can express the fact that it cannot produce a normal value in a number of ways depending on the system we're looking at, but that's the kind of condition we're interested in dealing with.

Error conditions are usually associated with a *reason* for their occurrence. A function may not be able to compute its contract result if its input is in not valid, ex: dividing by zero, if some system state that it needs to cross reference its input with is invalid, or if an effect that a procedure is trying to have on its environment fails, ex: failure to send network packet, failure to open a file for writing.

A programming system that provides for handling such conditions involves both detecting the occurrence of an error and taking some action when a particular type of error occurs - usually referred to as a "handler". When an error simply cannot be recovered from (ex: system out of memory), the only possible recourse is to crash the program, which is often a default behaviour embedded into applications.

11.2 Encoding errors in values

The simplest approach that a function or procedure can take to signal an error condition is to encode the error condition as a special return value. This could be a simple numeric code or an entire object which captures more detailed context about the error so that appropriate action can be taken.

The C programming language and Go, for example, both take the option of returning an error code and expecting the caller to detect the condition and take corrective action if possible.

Below is a simple example in the C language.

```
int write_to_file(const char *message, const char *filename) {
FILE *f = fopen(filename, "w");
if (f != NULL) {
return -1; // Nothing written.
}

fprintf(f, "%s", message);
fclose(f);
return 0;
}
```

The idea of returning special error codes is also a common feature of languages featuring a strict type system, such as Haskell and Rust, which feature `Option` or `Maybe` and `Result` or `Either` types. Below is a trivial example of calculating the width in pixels of each column when the total width of all columns and the number of columns are known.

```
columnSize : Int -> Int -> Maybe Int
columnSize width numColumns =
if numColumns == 0 then
Nothing
else
Just (width `div` numColumns)
```

The `columnSize` function withh produce a `Nothing` when presented with the odd situation of no columns, and give `Just width` when presented with a valid situation. In this case, we didn't need any further explanation for the error condition perhaps. If we needed that, we could've used an `Either` type as follows

```
columnSize : Int -> Int -> Either String Int
columnSize width numColumns =
if numColumns == 0 then
```

```
Left "Can't give 0 columns"
else
Right (width `div` numColumns)
```

... where the `Left` value gives some explanation about the error condition instead of just saying `Nothing`.

At some level, all error management and recovery schemes boil down to representing error conditions in some data structure and passing it around to some piece of code which know what to do with it.

11.3 Errors at the system level

Most of our programs are launched by the OS kernel as processes which get their own address space. Since the kernel is expected to manage the resources allocated to the processes it launches, when a process exits due to an error condition, the kernel is expected to behave in such a way that system resources are not leaked.

Towards ensuring this, the kernel will release all the memory allocated to the process back to the pool, close all I/O handles such as files and sockets, release any shared memory or other resources and so on, when a process exits either in a controlled fashion or in an unexpected fashion.

A process may also register specific handlers for "signals". It may take specific actions, such as saving its current state, when asked to abruptly terminate itself due to some error condition. Thus, errors may also be injected into processes externally.

11.4 try-catch based error management

In languages such as Java, Javascript and C++, we can mark a piece of code as "error prone" by wrapping it inside a `try` block, with a `catch` clause specifying code to execute when error conditions occur. Some mock code -

```
int write_to_file(const char *message, const char *filename) {
File *f = NULL;

try {
FILE *f = fopen(filename, "w");
if (f == NULL) { throw -1; }
if (strlen(message) > 100) {
throw -2;
}
fprintf(f, "%s", message);
fclose(f);
return 0;
} catch (int e) {
fclose(f);
return e;
}
}
```

If the double `fclose` is bothering you as it should, you can use the `finally` clause to release resources whether the function succeeds or fails.

```
int write_to_file(const char *message, const char *filename) {
File *f = NULL;

try {
```

```
FILE *f = fopen(filename, "w");
if (f == NULL) { return -1; }
if (strlen(message) > 100) {
return -2;
}
fprintf(f, "%s", message);
return 0;
} finally {
fclose(f);
}
}
```

Such a `finally` clause is used in these languages to specify code that must be executed irrespective of whether the procedure returns normally or via some abnormal means such as an exception thrown using `throw`.

11.5 Resource Acquisition As Initialization (RAAI)

As we saw, resource cleanup is one of the important actions to be taken when errors happen. The above ways of testing for error conditions and managing resource cleanup is quite onerous on the programmer and usually leads to omissions that translate into resource leaks.

C++ provides an elegant solution to this problem through the idea of block scoping associated with objects allocated on the stack. When code within a block scope (portion within `{}` curly braces) completes, either abnormally or normally, the C++ compiler automatically inserts the necessary instructions to release all the objects that were allocated on the stack within that block. Since C++ has the notion of "destructor" for objects, which is a procedure called when an object is released or "destroyed", we can combine these two concepts to provide for automatic and almost code-free resource cleanup. The destructors of these stack allocated objects are called in the reverse allocation sequence automatically.

So what we have to do is to model resources as objects that can be allocated on the stack, and put in their cleanup code within the destructors of the classes of these objects. For example -

```
void write_to_file(const std::string &filename, const std::string &message) {
std::ostream file(filename);
if (message.length() > 100) {
throw std::error("Message too big");
}
file.write(message);
}
```

Since the `file` object is allocated on the stack, if the allocation failed, an exception will be thrown from within `write_to_file` and none of the other lines of code will run. When we `throw std::error("Message too big")` as well, that marks the end of the `file` object's scope and its destructor will be called before the `write_to_file` exits with an exception. If we get to the end of the function normally, the end of scope will take care of closing the file as well. This makes of succinct management of resource and leaves little room for programmer error.

Due to the determinate nature of C++ (i.e. non-garbage collected runtime), we can build robust systems layer by layer.

11.6 Concurrent error management

The above cases all treated error management in single-threaded programs. When we deal with highly concurrent systems, we need to resort to different strategies. The language Erlang is used to build massively

distributed and highly available telephony systems and it features an error management approach that is closer to the way the unix kernel treats processes than any of the other languages.

Erlang features light weight "processes" which are not the same as unix kernel level processed. A small amount of stack and heap is associated with an Erlang process when it starts (as little as 300 bytes) and multiple such processes can run on a single core machine. The Erlang runtime rations function call executions (called "reductions" in the Erlang world) between these processes so that fine grained concurrency is possible. Each Erlang process also has an associated "mailbox" into which it can receive messages from other processes, even if they're across the network.

One interesting feature of the Erlang system is that usually the code for processes will consist of only the happy path and processes are recommended to crash in the case of error! This may sound like blasphemy to experts in other languages, but this is the basis for the high availability strategy that is implemented in Erlang's OTP library. OTP is an application architecture that facilitates setting up Erlang processes in supervisor hierarchies. That is, if a process `S` is marked as the supervisor of another process `P`, then when `P` dies by crashing, `S` gets notified by a message, based on which it can choose a variety of actions. For example, `S` itself can choose to crash if it doesn't know how to recover from `P` crashing, or it may choose to restart `P` in a stable state and keep chugging along as though nothing happened. `S` itself may be supervised by another process which can apply similar logic to it child tree of processes. Therefore, garbage collection in Erlang happens through process exit, in addition to smaller collections within processes. Since processes share nothing between them (messages between them are copied), there is almost no inter-process analysis that needs to be done that can delay garbage collection.

In real world engineering, this "just crash and let the supervisor tree do its thing" strategy is surprisingly effective to ensure system robustness. Not all systems can be designed with this philosophy though, and Erlang's low latency scheduler is a critical component that makes this strategy work as effectively as it does.

11.7 Contextual recovery stratgies

We saw how in the C/C++/Java/Javascript language families exceptions propagate "up the call stack" until one reaches code that can do something about them. This implies that the local context in which the error occurs is no longer available when it comes time to determining what to do about the error.

There are many situations in programming where this is not acceptable. Many situations do not require the entire job to be restarted for certain kinds of errors, but need to do it for other kinds. For example, if writing to a log file raised an exception, it might be ignorable in a system where the log file is not a critical component, but cannot be ignored in cases where the log is, say, used as a transaction log. This means that the library that features such a logging cannot decide for itself what to do, and must necessarily delegate that responsibility to the caller. Local context destruction through stack unwinding is a bad fit for these kinds of problems.

A more general error handling approach is to maintain a stack of handlers and include error handling options as part of a function's calling interface. When an error occurs, the appropriate handler is selected and called **in-situ**, without local context destruction. The handler can then **choose** whether to unwind the call stack and destroy context, or adopt some other recovery strategy that permits the program to continue from the point at which the error occurs.

Common Lisp's "conditions" provide such a mechanism. This mechanism is strictly more general than the mechanisms offered by the other languages discussed thus far, apart from Erlang.

> For good examples of using Common Lisp's condition system, see the book "Practical Common Lisp" by Peter Seibel.

The error mechanism built into muSE is also similar in expressive power and is described here.

11.8 Error management in Slang

TODO: Implement Common Lisp-like error handling mechanism in Slang that is also process aware.