1 Goals

Provide optimized code while retaining ability to utilize functional list paradigms.

2 Algorithm

At a high level, the algorithm will transform a series of standard JavaScript *Array* operators (*filter*, *map*, *reduce*, *forEach*, *some*, *every*) into a standard for loop notation.

The full algorithm can be broken into the following process:

2.1 Transpile

When transpiling the source files, we look for possible transformation patterns

1. Look for every possible indication that Array list operators are being used
   (a) Only *map* and *filter* reliably return new Arrays, whereas *some*, *forEach* and *every* reliably return non-arrays. *reduce* can return any value (and so we treat it as unknown)
2. When a potential Array operator site is found, modify source to
   (a) Wrap the Array expression with a $WRAP$ invocation, to provide an Array surrogate that can collect the operators
   (b) Wrap every function used by the operators with a TAG invocation. This provides additional meta-data used for the compilation process
   (c) Wrap the potential Array expression and the subsequent operators with an $EXEC$ invocation to allow for compilation and execution of the collected operators

Sample Transpilation input

```
let sumAndCount = findPeople()
  .filter(p => p.age > 35 && p.gender == 'male')
  .map(p => p.name)
  .reduce((acc, p) => [acc[0] + p.income, acc[1] + 1], [0,0])

let avgIncome = sumAndCount[0]/sumAndCount[1];
```

Sample Transpilation output

```
let sumAndCount = EXEC(WRAP(findPeople())
  .filter(TAG(p => p.age > 35 && p.gender == 'male'), UNIQUE_ID_1, true)
  .map(TAG(p => p.name), UNIQUE_ID_2, true)
  .reduce(TAG((acc, p) => [acc[0] + p.income, acc[1] + 1], [0,0]), UNIQUE_ID_3, true))

let avgIncome = sumAndCount[0]/sumAndCount[1];
```

2.2 Runtime

At runtime time, the functions $WRAP$, TAG, and $EXEC$ will perform all the necessary work to produce and execute the optimized code

1. When transpiling a function or program context, allow for control of process by pragma
2. When $WRAP$ is invoked, and the argument is an array, will produce a wrapper object that will collect all the tagged operations (*filter*, *map*, *reduce*, etc.).
3. When $EXEC$ is invoked, if the input is a wrapper object, it will compile and execute the code.
4. When $WRAP$ is invoked with a non-array or $EXEC$ is invoked with a non-wrapper object, it will return the input as is. This allows for runtime type detection, and will not fail if types are not-known at compile-time.

2.3 Compilation

When compiling the code, we analyze, and transform the operators into a for loop. There are some caveats with transforming the code as some constructs prevent rewriting. The steps we follow are:

1. Generate the unique key to represent the combination and sequence of the operators in the series.
2. If the key has been seen before, return compiled output
3. Determine operator validity
   (a) Check for operator functions for intermediate array references. *map*, *reduce*, *filter*, etc have a form in which the intermediate array is accessible in a read only fashion. If referencing, disqualify entire chain as it cannot be supported.
   (b) When the array operator is a variable (and not a function literal), we need to check for closed variables, since we are rewriting the function, any closed variable we do not have access to will be lost.
      i. There is provision to allow for commonly used globals to not be considered closed (and thusly disqualify rewriting).
      ii. All variables can shadowed at runtime, and we cannot determine the actual value looking at source alone.
      iii. Many global static functions can, and should assume will not be overwritten (e.g. String.fromCharCode, or Math.min)
   (c) When the array operator is a function literal, it is in the scope of the current function, this gives us some leeway.
      i. Closed variables generally are free to be ignored since we can account for them.
         A. Pass in all closed variables into generated function
         B. Re-assign all closed variables that were written to, on completion of function
4. If operator series is valid, build a new function by transforming the operators into a for loop
5. Also note, we have the potential issue that that we will be modifying the function invocation order:

```
[1,2].map(a).filter(b)
```

will essentially turn into

```
let out = []
let af = map(1)
if (filter(af)) out.push(af);
let bf = map(2)
if (filter(bf)) out.push(bf);
```

This generally shouldn't matter,

3 Evaluation

The current goal is to find existing code bases that this can be run on, and evaluate overall performance. The main hinderance to this, is that converting functional list form to for-loop is a common hand optimization. The places where this would be utimately useful will more than likely already have been optimized.

The alternate to finding an existing code base, is to create a program, only using the functional list format, and then run with optimizations, verifying output and timings. The program would have to be of sufficient complexity.

Generally these optimizations make sense at the framework level even more (though finding places to optimize will be harder).

What I may do is also look into CPU intensive applications, rewriting into functional list form, and evaulating the ability to test and verify the program. The functional form should be better tested vs a hand optimized form.