The SMUDGE compiler is comprised of six passes:

- Parsing
- Clock checking
- Scheduling
- Normalization
- Sol codegen
- Javascript codegen

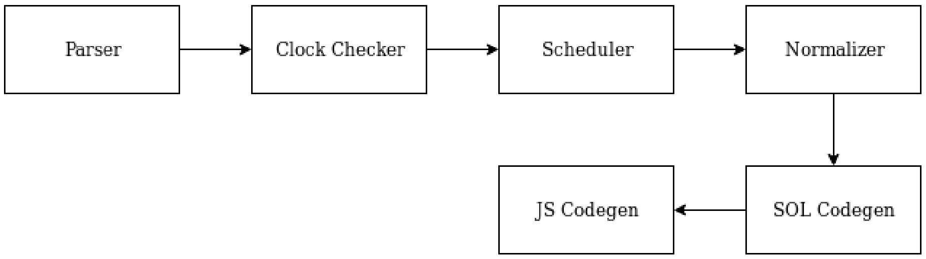

Figure 1: Compiler Passes

Each output is fed as input to the following pass.

For debug purposes, build artifacts are created in the `build/` folder. As such, we can find the following files in `build/`:

- name.sap
- name.clk
- name.scheduled
- name.nsap
- name.sol

If no `-o name` parameter is given, the compiler outputs the Javascript code in `out.js`

Scheduler

For more ease of use, SaP users are allowed to write their equations in whichever order they like. As such, there should be no noticeable difference in the execution of the following snippets.

```
node example() -> (x: int) with
  x = y + 1;
  y = 2
```

```
node example() -> (x: int) with
  x = y + 1;
  y = 2
```

This is the task of the Scheduler. We'll explain its operation using the following snippet as example.

```
node example() -> (x: int) with
  a = whatev(x);
  z = 3 fby plus(z, 1);
  b = 2 fby 1;
  x = plus(z, y);
  (c, d) = (x, y);
  y = 2
```

TL;DR

The scheduler must find dependencies between equations. In order to do so, it runs through node equations to find references to variables defined in-node. It has to check for circular dependency, and reorganize equation order based on those factors. We implement this by using graph algorithms.

Firstly, the scheduler builds an Hashmap containing the equations with the ids as index.

id	equations
a	a = op(x)
z	z = 3 fby plus(z, 1)
b	b = 2 fby 1
x	x = plus(z, y)
c	(c, d) = (x, y)
d	(c, d) = (x, y)
y	y = 2

Table 1: Hashtable Contents

The scheduler then establishes a list of ids that could be depended upon. This is to exclude input variables and delays. Indeed, input variables would not be present in the equations. We treat equations of the form `x = v fby a` as in this case `x` is a memory and thus must be scheduled after every computation reading variable `x`.

a	x	c	d	y

Table 2: Id List

The scheduler then moves on to creating the dependency graph. First, it adds a vertex for each id that isn't deducted from a delay expression.

Secondly, it adds an edge for each reference to an id in the right-hand side of the equation. For example, it would add an edge between `a` and `x`. We are presented with the following graph.

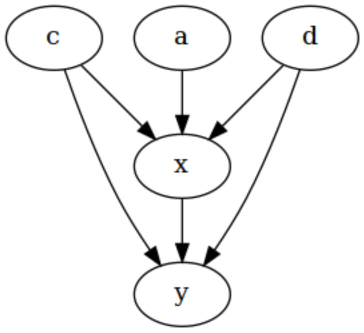

Figure 2: Dependency Graph

Once the dependency graph is in our possession, we can do a causality check. Indeed, a schedule only exists if the graph is acyclic. If no cycle is found, the scheduler moves on to reordering.

The scheduler then traverses the graph in order to find a vertex that has no successors, that is to say that eq has no more unsatisfied dependencies. When one is found, it is added to the ordered list, and the vertex is deleted. The scheduler goes on recursively until no vertex is left.

We are left with the following arrangement.

```
node test() -> () with
  y = 2;
  x = plus(z, y);
  a = whatev(x);
  (c, d) = (x, y);
  z = 3 fby plus(z, 1);
  b = 2 fby 1
```