SOL

As our goal is to compile down to a more imperative language, we'll use the most simple form of an Object-Oriented language. Please keep in mind that we are only interested in the capability to encapsulate a piece of memory managed exclusively by the methods of the class.

$\langle dec \rangle ::=$ `machine` $f =$
    (`interface` t)?
    `memory` $\langle var\_decs \rangle$
    `instances` $\langle mach\_dec \rangle$
    `reset`$() = \langle inst \rangle$
    `step`$(\langle var\_decs \rangle)$
    `returns` $(\langle var\_decs \rangle) =$
    `var` $\langle var\_decs \rangle$ `in` $\langle inst \rangle$

$\langle inst \rangle ::= x = \langle exp \rangle$
  | `state`$(x) = \langle exp \rangle$
  | $\langle inst \rangle$; $\langle inst \rangle$
  | `skip`
  | o`.reset`
  | $(x, ..., x) = o$`.step`$(\langle exp \rangle, ..., \langle exp \rangle)$
  | `case`(x) { $\langle C \rangle$: $\langle inst \rangle$ | ... | $\langle C \rangle$: $\langle inst \rangle$}

$\langle exp \rangle ::= x$
  | $\langle value \rangle$
  | `state`(x)
  | $op(\langle exp \rangle, ..., \langle exp \rangle)$

$\langle value \rangle ::= C$
  | i

$\langle mach\_dec \rangle ::= o : f, ..., o : f$

$\langle var\_decs \rangle ::= x : t, ..., x : t$

Figure 1: Simple Object Language Grammar

ADT

Algebraic Data Types exhibit some properties: * Composite: Definition by cases - each case is composed into a single type * Closed: A finite set of cases

The simplest ADT is Haskell's Boolean.

```
data Boolean = True | False
```

Here, a variable of type `Boolean` has to be either `True` or `False`. However, a variant type can also contain inner variables.

```
type event = Nothing | GainPoints(amount: int)
```

A variable of type `event` will then be of type `Nothing` or `GainPoints` based on what happened in the game.

To access a fieled contained in a variant type, we use a syntax similar to deconstructive pattern matching in ML languages.

```
step(e : event) returns () = var in
case (e) {
  Nothing: {- We do nothing-} |
  GainPoints(amount): state(points) = state(points) + amount
  }
```

Syntax

$\langle$*type_dec*$\rangle$ ::= `type` *id* '`=`' $\langle$*type_constr*$\rangle$

$\langle$*type_constr*$\rangle$ ::= *constr_id*
  | *constr_id*($\langle$*param*$\rangle$)
  | $\langle$*type_constr*$\rangle$ '`|`' $\langle$*type_constr*$\rangle$

$\langle$*param*$\rangle$ ::= *id* '`:`' *type_id*
  | $\langle$*param*$\rangle$ '`,`' $\langle$*param*$\rangle$

Figure 2: Algebraic Data Type Grammar

Compilation

However, Javascript doesn't have Algebraic Data Types. Thus, some trick is required. Fortunately, this has been the center of some research, as this is know as The Expression problem. While we'll not dive into detail here, know that an ADT can be represented using Object-Oriented Programming.

We use a class's static methods to represent the construction of a variant type. If a variant type is also a product type, that is it contains variables, the static method will take as arguments those variables in order.

We use object litterals with a bit set to the enumerated type corresponding. For example, the aforementioned `event` type will become:

```
var event_enum = Object.freeze({
  Nothing: 1,
  GainPoints: 2
});
```

$$
\begin{aligned}
TC_{(id)}(C_n(i_1 : t_1, ..., i_n : t_n)) \quad &= \quad id.C_n = \texttt{function}(i_1, ..., i_n)\ \{ \\
&\qquad \texttt{return}\ \{\texttt{id} : id, i_1 : i_1, ..., i_n : i_n\}; \\
&\quad \}
\end{aligned}
$$

$$
\begin{aligned}
TT(\texttt{type}\ \ id = C_1|...|C_n) \quad &= \quad \texttt{var}\ \ id\texttt{\_enum} = \{ \\
&\qquad C_1 : 1, \\
&\qquad ..., \\
&\qquad C_n : n \\
&\quad \}; \\
&\quad TC_{(id)}(C_1); \\
&\quad ...; \\
&\quad TC_{(id)}(C_n);
\end{aligned}
$$

Figure 3: Type Translation Function

```
function event_type() {}

event_type.Nothing = function() {
  return {id: event_enum.Nothing}
}

event_type.GainPoints = function(amount) {
  return {id: event_enum.GainPoints, amount:amount}
}
```

Formal translation

Interface

Our model concentrates on the implementation of an `interface` node - or machine - that will be used by the game engine's javascript implementation. Such a node will typically take an unique parameter, in the form of an `event` variable.

Such an `event` will leverage the power of `Algebraic Data Types` to describe the event.

```
type event = Move | ChangeDir(x: int, y: int)
```

In order to represent such a data structure in Javascript, we leverage the ADT/Object duality (described in Cook's paper) and convert the event type to a Javascript object.

The previously mentioned `event` type will be compiled to the following Javascript object.

```
var event_enum = Object.freeze({
  Move: 1,
  ChangeDir: 2
});

function event_type() {}

event_type.Move = function() {
  return {id: event_enum.Move}
}

event_type.ChangeDir = function(x, y) {
  return {id: event_enum.ChangeDir, x:x, y:y}
}
```

The type variant is idntified by the `id` field present in each object variation. Each variant then has its specific properties added in its object litteral.

Interface

```
machine main =
  interface event
  memory
  instances
  reset () = skip
  step(e : event) returns (x: int, y: int) =
    var in
```

```
function main() {}

main.prototype.reset = function() {}

main.prototype.step = function(e) {
  return this;
}
main.prototype.move = function () {
  this.step(event_type.Move());
  return this;
}

main.prototype.changeDir = function (x, y) {
  this.step(event_type.ChangeDir(x, y));
  return this;
}
```

which allows the engine's code to simply write:

```
main.move();
main.changeDir(10, 10);
```

Effectively providing a nice interface to the dataflow code.