Translation

To present the translation functions, we'll use the following format. On the left side is the SOL program, and its translation is found on the right.

A machine will be translated to a object (function) declaration, with two methods added to its prototype, `step()` and `reset()`

```
machine <id> =
  memory <mach_dec>
  instances <insts>
  reset() =
      <exps>
    step(<in_var_decs>) returns (<
         out_var_decs>) =
    var <var_dec> in
    <step_exps>
```

```
function <id>() {
  translate_mem(<machdec>);
  translate_inst(<insts>);
}

<id>.prototype.reset = function() {
  translate_exps(<exps>)
}

<id>.prototype.step = function(<
     in_var_decs>) {
  translate_exps(<step_exps>);
}
```

A memory is translated as a member variable.

```
translate_mem(<var_id> : <var_ty>)
```

```
this.x = undefined;
```

A node instance is translated as a member variable holding a node object instance.

```
translate_inst(<var> : <node_id>)
```

```
this.<var> = new <node_id>()
```

A variable assignment is translated as an Javascript assignment and the tranlation of the right-hand side exp.

```
translate_inst(<var_id> = <exp>)
```

```
<var_id> = translate_exp(<exp>);
```

Skip expression amounts to nothing.

```
translate_inst(skip)
```

```
;
```

A reset instruction is translated as a `reset()` call on the member node variable.

```
translate_inst(<id>.reset)
```

```
this.<id>.reset();
```

A state assignment is translated as an assignment of the translated exp to the member variable.

```
translate_inst(state(<var_id>) = <exp
     >)
```

```
this.<var_id> = translate_exp(<exp>);
```

A step instruction is translated as a `step()` call on the node member variable. The tuple assignment is translated to array assignment, which is available in ES6.

```
translate_inst(
  (<var_id>, ..., <var_id>) =
  <node_id>.step(<value>, ..., <value
       >))
```

```
[<var_id>, ..., <var_id>] =
  this.<node_id>.step(
    translate_val(<value>), ...,
         translate_val(<value>))
```

Translation of a sequence of inst is translated as the sequence of the translations.

```
translate_inst(<inst>;<inst>)
```

```
translate_inst(<inst>);
translate_inst(<inst>);
```

The case instruction is converted as the corresponding switch instruction in Javascript. However, a special case is made when the variable switched on is part of the node interface.

```
translate_inst(case(<var_id>)
  {<constr> : <inst>,
   ...,
   <constr> : <inst>})
```

```
switch(<var_id>) {
  case <constr>:
    translate_inst(<inst>);
  ...
  case <constr>:
    translate_inst(<inst>);
}
```

A variable id is simply left as such.

```
translate_exp(<var_id>)
```

```
<var_id>
```

A value translation as expression is translated as a value.

```
translate_exp(<value>)
```

```
translate_val(<value>)
```

State variable access is translated as an access to a member variable

```
translate_exp(state(<var_id>))
```

```
this.<var_id>
```

Operator translation amounts to use the same operator with each value argument translated.

```
translat_exp(<op_id>(<value>, ..., <
     value>))
```

```
<op_id>(translate_val(<value>),
        ...,
        translate_val(<value>))
```

A constructor has to be a type's variant. Thus, it is translated as the value of that type's enum. See the section on ADT for more information.

```
translate_val(<constr_id>)
```

```
<type_id>_enum.<constr_id>
```

An immediate is simply left as such.

```
translate_val(<immediate>)
```

```
<immediate>
```

Full example

On the next page is a full example.

```
machine condact =
memory x2: int
instances x4: count
reset () =
    state(x2) = 0
step (c : bool,i:int) returns (o:int)
      =
    var x3 : int in
    case(c) { Empty: o = x4.step(i) |
              Full: o = state(x2)};
    state(x2) = o
```

```
function condact() {
  this.x2 = undefined;
  this.x4 = new count();
}

condact.prototype.reset = function()
     {
  this.x4.reset();
  this.x2 = 0;
  return this;
}

condact.prototype.step = function(c,
     i) {
  var x3 = undefined;
  switch(c) {
    case inside_enum.Empty:

      o = this.x4.step(i);
      break;
    case inside_enum.Full:

      o = this.x2;
      break;
  };
  this.x2 = o;
  return o;
}
```