The Hume Manual, Version 0.0

Greg Michaelson[1] Kevin Hammond[2]

[1]Department of Computing and Electrical Engineering
Heriot-Watt University
greg@cee.hw.ac.uk, +44 131 451 3422

[2]School of Computer Science, University of St Andrews
kh@dcs.st-and.ac.uk, +44 1334 463241

Contents

1 Introduction **2**

1.1 Overview 2

1.1.1 Acquiring Hume 3

2 Running Hume programs **4**

2.1 Execution Model 4

2.2 Runing Hume 4

2.3 Halting execution 5

2.4 Example 1: Counter 5

2.5 Example 2: Square and double 7

2.6 Example 3: Square and double with fair matching 11

3 Input/Output **12**

3.1 Overview 12

3.2 Streams 12

3.3 How to... 13

3.3.1 Make output appear on standard output 13

3.3.2 Input a string 13

A Syntax **14**

Chapter 1

Introduction

1.1 Overview

Hume is a programming language based on concurrent finite state machines with transitions defined through pattern matching and recursive functions. A formal definition of Hume may be found in [**?**].

This document describes how to write and run programs in Hume 0.0, a prototype implementation of a substantial Hume subset. Hume 0.0 includes:

- imprecise integer and float types
- bool, char and string types
- tuple, vector and list types
- conditional and case expressions
- local definitions
- recursive function definitions
- constant definitions
- type synonyms
- unions
- exceptions
- pattern matching boxes with unfair and fair matching
- wiring with input initialisation and output tracing
- streams, connected to files and standard input and output
- I/O and comparison overloaded for arbitrary sized structures

Hume 0.0 lacks:

- precise integer and float types
- nat, unicode, word, fixed and exact types

- type cast and coercion
- type conversion functions
- ports
- time
- modules
- output initialisation and input tracing

The Hume 0.0 implementation:

- offers stepped or continuous mode execution
- offers output tracing in both modes
- displays box and wire states in stepped mode
- runs each box, to completion, with round robin scheduling
- lacks polymorphic type checking
- lacks exception consistency checking

1.1.1 Acquiring Hume

Hume can be acquired via: `www-fp.dcs.st-and.ac.uk/hume`.

DREADFUL WARNING...

Hume is very much under development. It is likely that this first implementation is full of bugs and unexpected behaviours. Please report:

- parser problems to Kevin Hammond - `kh@dcs.st-and.ac.uk`;
- run-time problems to Greg Michaelson - `greg@cee.hw.ac.uk`.

Chapter 2

Running Hume programs

2.1 Execution Model

A Hume program consists of one or more *boxes* with *inputs* and *outputs*. Boxes are generalised finite state machines. A box's transitions are defined by *patterns* on the inputs. Boxes are connected to each other, and to *streams* and *ports* by links established by *wiring*.

When a program is running each box repeatedly tries to match one of its pattern against its inputs. When a match succeeds, the expression associated with the pattern generates the values for the box's outputs. At the end of each execution cycle a box may be in one of three states:

- *runnable*: the box completed successfully;
- *matchfail*: the box lacked appropriate inputs;
- *blockedout*: the box completed but blocked as some outputs from the previous cycle were not consumed. On the next cycle, the box will again attempt to establish its outputs.

Matching may be *unfair* or *fair*. For unfair matching, on each cycle the matching starts with the box's first pattern. For fair matching, on each cyclethe matching starts with the pattern after that which succeeded on the previous cycle.

2.2 Runing Hume

By convention, Hume programs end with `.hume`.

To run a Hume program in *continuous mode*;

```
% hume program.hume
Hume 0.0
RUNNING
...
```

To run a Hume program in *stepped mode*:

```
% hume -t program.hume
Hume 0.0
STEPPING
ugly print traces: ...
```

```
outputs: ...
states: ...
std_out: ...
NEXT> return
...
```

All boxes run for one cycle.

ugly print displays the program in a strange mix of Hume syntax and internal representation.

`traces:` displays all outputs followed by `trace` in the corresponding wiring.

`outputs:` displays all unconsumed outputs.

`states:` displays the state of each box.

`std_out:` displays any output for standard output;

Pressing *return* after `NEXT>` initiates the next cycle.

2.3 Halting execution

`Control C` halts execution and returns to the host system.

2.4 Example 1: Counter

The following program generates a sequence of ascending integers on the standard output, starting with `0`.

Note that line numbers are not part of Hume but are used in subsequent discussion.

```
 1 -- counter - inc.hume

 2 program

 3 stream output to "std\_out";

 4 box inc
 5 in (n::int 64)
 6 out (n'::int 64,shown::(int 64,char))
 7 match
 8  x -> (x+1,(x,'\n'));

 9 wire inc
10  (inc.n' initially 0)
11  (inc.n,output);
```

1: Comments begin with `--`.

2: Programs start with `program` or `module`.

3: The stream `output` is associated with standard output, named through the string `"std_out"`.

4: The box is named `inc`.

5: The box has one input called `n`, a 64 bit integer.

6: The box has two outputs. `n'` is a 64 bit integer. `shown` is a tuple of a 64 bit integer and a character.

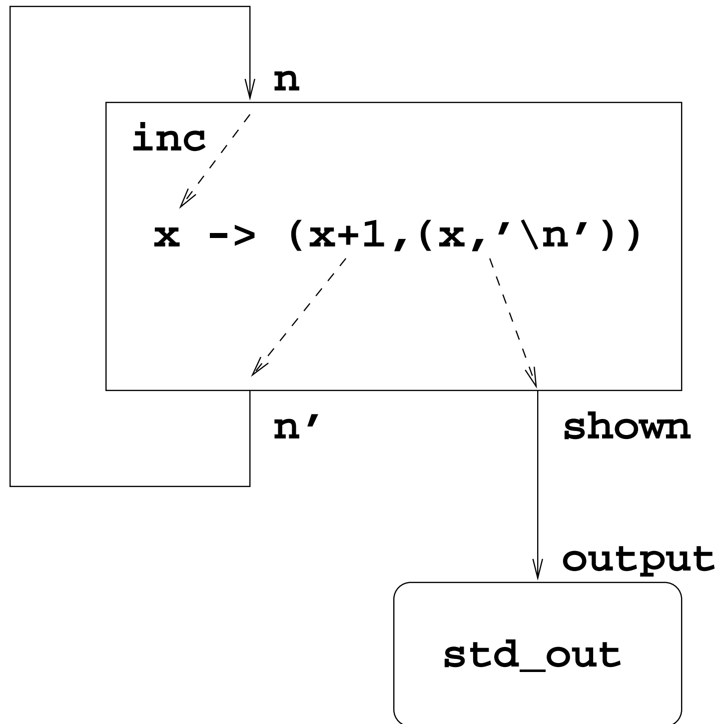

Figure 2.1: Counter - inc.hume

`7-8`: If there is a value on input `n`, then the pattern `x` will set a new local variable called `x` to that value and evaluate the right hand side. Output `n'` will be set to `x+1`. Output `shown` will be set to `(x+1,'\n')` i.e. `x+1` followed by a newline character.

`9`: Box `inc` is now wired by position.

`10`: `inc`'s input `n` is wired implicitly to `inc`'s input `n'`. `n` is initialised to `0`.

`11`: `inc`'s output `n'` is wired implicitly to `inc`'s input `n`. `inc`'s output `shown` is wired to `output` and hence to `"std_out"`.

The program is illustrated in Figure 2.1.

Thus, on each step, this program:

- sends `x+1` round the wire loop from `n'` back to `n`
- displays the old value of `n` i.e. `c`, followed by a newline

Running this example in continuous mode:

```
% hume inc.hume
Hume 0.0
RUNNING
0
1
2
3
...
```

Running this example in stepped mode:

```
% hume -t inc.hume
HUME 0.0
...
STEPPING
outputs: inc.n':1
status: inc: RUNNABLE
std_out: 0

NEXT>
outputs: inc.n':2
status: inc: RUNNABLE
std_out: 1

NEXT>
outputs: inc.n':3
status: inc: RUNNABLE
std_out: 2

NEXT>
...
```

At each stage:

- `n'` is set to one more than `n`'s value;
- matching succeeds so `inc` is runnable;
- `n`'s value appears on the standard output.

2.5 Example 2: Square and double

The following program extends the counter program to generate a sequence of squares and doubles of ascending integers. A new box `sqdouble` is added to:

- accept an integer from `inc` every two cycles;
- on the first cycle, print the integer's square;
- on the second cycle, print the integer's double;
- keep track as to whether it is currently squaring or doubling.

```
 1 -- square and double

 2 program

 3 stream output to "std_out";

 4 union STATE = SQUARING | DOUBLING;

 5 box inc
 6 in (n::int 64)
 7 out (n'::int 64,shown::int 64)
 8 match
```

```
 9 n -> (n+1,n);

10 wire inc (inc.n' initially 0) (inc.n,sqdouble.n);

11 box sqdouble
12 in (n::int 64,oldn::int 64,s::STATE)
13 out (n'::(int 64,char),oldn'::int 64,s'::STATE)
14 match
15  (x,*,SQUARING) -> ((x*x,'\n'),x,DOUBLING) |
16  (*,x,DOUBLING) -> ((2*x,'\n'),*,SQUARING);

17 wire sqdouble
18  (inc.shown,sqdouble.oldn',sqdouble.s' initially SQUARING)
19 (output,sqdouble.oldn,sqdouble.s);
```

4 introduces a concrete data type `STATE` with values `SQUARING` and `DOUBLING`.

10 wires `inc.shown` to the new box `sqdouble`'s input `n`.

`11-16` introduce the new box `sqdouble`. It has 3 inputs and 3 outputs. Looking at the wiring in `18-19`: the link from`oldn` to oldn' circulates the last value from `inc`; the link from `s` to `s'` circulates local state information. Initially, `s` is set to `SQUARING`.

`15-16` define `sqdouble`'s transitions. For an input `n` from `inc.shown` and with `s` indicating the `SQUARING` internal state, `oldn` is ignored by the `*`, `n` is matched to `x` which is squared and output, `n` is circulated as `oldn'` back to `oldn`, and `s'` is circulated as `DOUBLING` back to `s`.

If `s` is `DOUBLING`, the `n` is ignored, `oldn` is matched to `x` which is doubled and output, no output is generated for `oldn'` by the `*` and `s'` is circulated as `SQUARING` back to `s`.

The effect is that for each integer that `inc` generates, `doublesq` will generate its square on the first cycle and its double on the second cycle. After the first cycle, `inc` will generate a new integer but will block on output until `sqdouble` has completed the second cycle and consumes it.

The program is illustrated in Figure 2.2.

Running the program in continuous mode:

```
%hume sqdouble.hume
HUME 0.0
RUNNING
0
0
1
2
4
4
9
6
...
```

Running the program in stepped mode:

```
% hume -t sqdouble.hume
HUME 0.0
...
STEPPING
outputs: inc.n':1, inc.shown:0
```

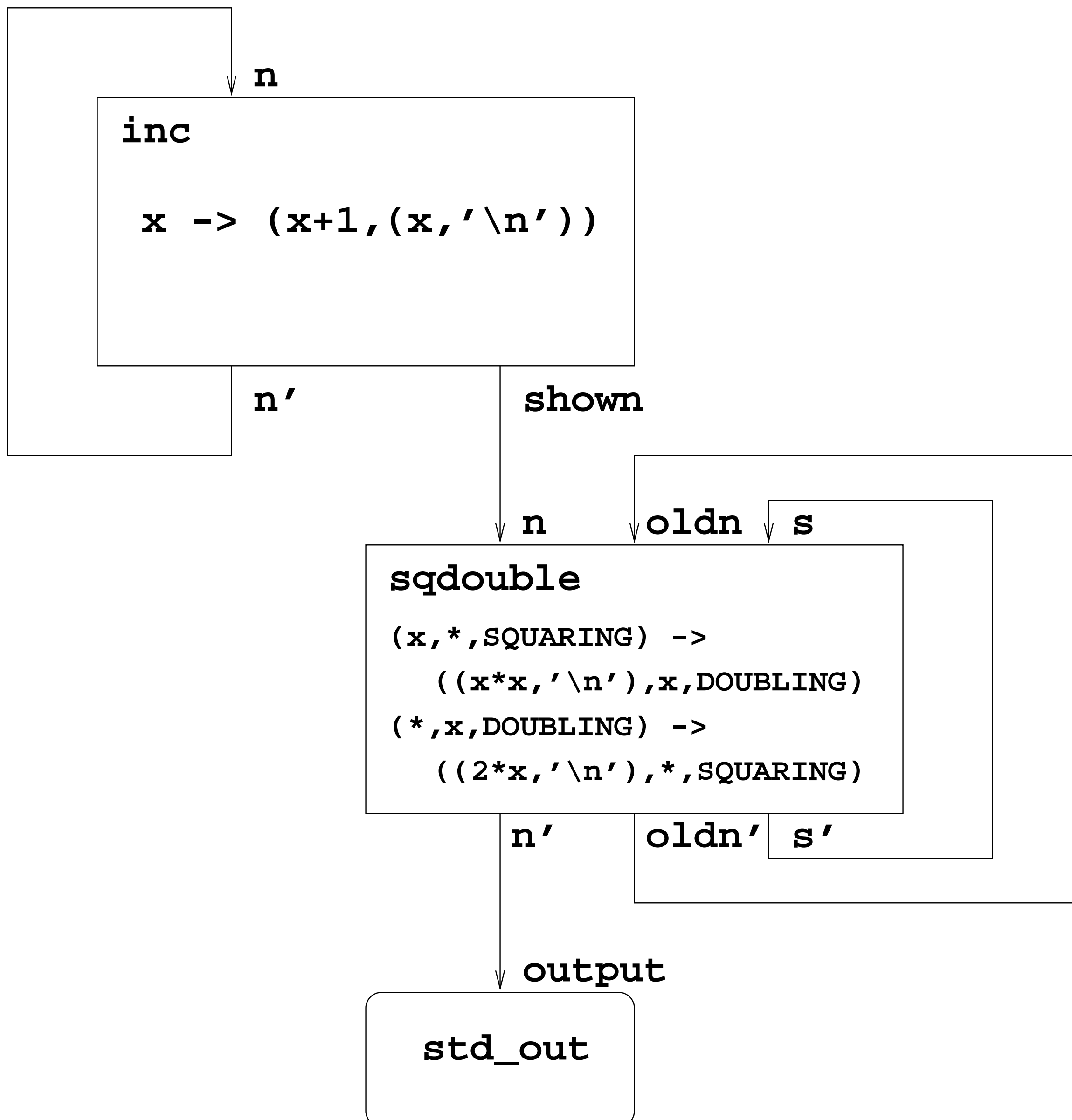

Figure 2.2: Square and double - sqdouble.hume

```
status: sqdouble: MATCHFAIL; inc: RUNNABLE
std_out:
NEXT>

outputs: sqdouble.oldn':0, sqdouble.s':DOUBLING, inc.n':2, inc.shown:1
status: sqdouble: RUNNABLE; inc: RUNNABLE
std_out: 0

NEXT>
outputs: inc.shown:1, sqdouble.s':SQUARING
status: sqdouble: RUNNABLE; inc.shown: BLOCKEDOUT
std_out: 0

NEXT>
outputs: sqdouble.oldn':1, sqdouble.s':DOUBLING, inc.n':3, inc.shown:2
status: sqdouble: RUNNABLE; inc: RUNNABLE
std_out: 1

NEXT>
```

At the end of the first cycle:

- `inc`'s `n'` is `1` and it's `shown` is 0;
- `sqdouble` could not match any of it's patterns.

At the end of the second cycle:

- `sqdouble` has output the square of `0` from `inc`'s `shown` via it's own `n` i.e. `0`;
- `inc`'s `n'` is 2 and it's `shown` is 1;
- `sqdouble`'s `oldn'` is `0` and it's `s'` is `DOUBLING`.

At the end of the third cycle:

- `sqdouble` has ouput the double of `0` from it's `oldn` via it's own `oldn'` i.e. `0`;
- `sqdouble`'s `s'` is back to `SQUARING`;
- `inc`'s `shown` is `1` but this has been ignored by `sqdouble`. Thus `inc` is blocked on output.

At the end of the fourth cycle:

- `sqdouble` has output the square of `1` from it's `n` via `inc`'s `shown` i.e. `1`;
- `inc`'s `n'` is `3` and its `shown` is `2`;
- `sqdouble`'s `s'` indicates the `DOUBLING`internal state for it's `oldn'` which is now `1`.

Thus `sqdouble` runs twice as often as `inc`.

2.6 Example 3: Square and double with fair matching

For this example, the same effect can be achieved through *fair matching* without the `SQUARING/DOUBLING` feedback loop:

```
 1 -- square and double 2
...
10 wire inc (inc.n' initially 0) (inc.n,sqdouble2.n);

11 box sqdouble2
12 in (n::int 64,oldn::int 64)
13 out (n'::(int 64,char),oldn'::int 64)
14 fair
15  (x,*) -> ((x*x,'\n'),x) |
16  (*,x) -> ((2*x,'\n'),*);

17 wire sqdouble2
18  (inc.shown,sqdouble2.oldn')
19  (output,sqdouble2.oldn);
```

We have dropped:

- the `s` input and `s'` output at `12-13`;
- the corresponding matches for `DOUBLING` and `SQUARING` at `15-16`;
- the corresponding wiring for `s` and `s'` at `18-19`.

and replaced `match` with `fair` at `14`.

Now, if the match at `15` succeeds on one cycle then the alternate pattern at `16` will be tried first on the next cycle, and vice versa. Here, the behaviour is the same as for `sqdouble` above.

Chapter 3

Input/Output

3.1 Overview

In Hume, boxes communicate with the outside world over links to *streams*, associated with files, and to *ports*, associated with devices.

As with all Hume wiring, an individual stream or port can only be linked to one box input or output.

3.2 Streams

Streams are uni-directional and may be used for either input or output. Streams are declared by:

> input: `stream` *streamid* `from` *string*
> output: `stream` *streamid* `to` *string*

where:

> *streamid* = Hume *identifier;*
> *string* = file path within "s.

Standard input is `from "std_in"`.

Standard output is `to "std_out"`.

Streams are linked to inputs and outputs by mentioning their names in the appropriate positions in wiring.

Streams accept/deliver character sequences from/to sources/destinations. Such sequences are terminated with the character `'\0'`.

For input, streams should only be linked to inputs of determinate size type i.e. character, integer, float and bool, and arbitrarily nested tuples and vectors of these base types. The source character sequences will be automatically coerced to values of such types.

For output, streams may be linked to outputs of string and list type, as well as determinate size types. Link values will be automatically coerced to character sequences.

In both cases a canonical, flat, textual representation is used:

- integer/float/bool - text representation preceded by white space

- tuple/vector - unbracketed, white space separated element representation
- character - single un-quoted character
- string - output only - un-quoted text representation
- list - output only - unbracketed, white space separated element representation

For input, at the end of stream, the character `'\0'` is returned repeatedly.

For output, a stream is closed by the character `'\0'`.

In both cases, the system ought to shut any associated files...

3.3 How to...

3.3.1 Make output appear on standard output

Output to standard output must be forced by a newline i.e. `'\n'`. This can be effected by changing:

- the type of the output from *type* to (*type*,`char`);
- the corresponding expression from *exp* to (*exp*,`'\n'`).

Alas, this precludes same-line prompt/response sequences...

3.3.2 Input a string

A string is of indeterminate size and delimitatation, and so cannot be input automatically. Instead, a vector of appropriate size should be used to input a fixed width string:

```
vector 1 .. size of char
```

Appendix A

Syntax

This appendix gives a BNF definition of the Hume syntax. The meta-syntax is conventional. Terminals are enclosed in double quotes " ... ". Non-terminals are enclosed in angle brackets < ... >. Vertical bars | are used to indicate alternatives. Constructs enclosed in brackets [...] are optional. Parentheses (...) are used to indicate grouping. Ellipses (...) indicate obvious repetitions. An asterisk (*) indicates zero or more repetitions of the previous element, and a plus (+) indicates one or more repetitions.

Programs and modules

```
<program> ::=
        "program" <decls>

<module> ::=
        "module" <modid> "where" <decls>
```

Declaration Language

```
<decls> ::=
        <decl1> ";" ... ";" <decln>     n >= 1

<decl> :: =
          "import" <idlist>
        | "export" <idlist>
        | "exception" <exnid> <exprtype>
        | "union" <typeid> <varids> "=" <constrs>
        | "type" <typeid> <varids> "=" <type>
        | "constant" <varid> "=" <cexpr>
        | "stream" <iodes>
        | "port" <iodes>
        | <boxdecl>
        | <wiredecl>
        | <fundecl>

<constrs> ::=
            <conid1> <type11> ... <type1n>      m > 0, n >= 0
        "|" ...
        "|" <conidm> <typem1> ... <typemn>

<iodes> ::=
        ( <strid> | <portid> ) ( "from" | "to" ) <string>
        [ "timeout" <cexpr> ]

<fundecl> ::=
          <varid> "::" <type>
        | <varid> <args> "=" <expr>
        | <patt1> <op> <patt2> "=" <expr>

<args> ::=
        <patt1> ... <pattn>                    n >= 0

<fundecls> ::=
        <fundecl1> ";" ... ";" <fundecln>      n >= 1
```

Types

```
<type> ::=
          <exprtype>
        | "stream" <exprtype>           stream type
        | "port" <exprtype>             port type
        | "time"                        time type
        | "bandwidth"                   bandwidth type

<exprtype> ::=
          <basetype>                    base type
        | "vector" <range> "of" <type>  vector
        | "()"                          empty tuple
        | "(" <type> "," <types> ")"    tuple
        | "[" <types> "]"               list
        | <typeid> <type1> ... <typen>  discr. union, n >= 0
        | <type> "->" <type>            function type
        | "view" <type>                 view as type
        | "(" <exprtype> ")"            grouping

<types> ::=
        <type1> "," ... "," <typen>     n >= 0

<basetype> ::=
          "int" <precision>
        | "nat" <precision>
        | "bool"
        | "char"
        | "unicode"
        | "string" [ <intconst> ]
        | "word" <precision>
        | "float" <precision>
        | "fixed" <precision>
              [ @ ( "2" | "10" | "16" ) [ "**" <intconst> ] ]
        | "exact"

<precision> ::=
        "1" | ... | "64"

<range> ::=
        <intconst1> ".." <intconst2>
```

Expression Language

```
<expr> ::=
          <constant>
        | <varid>                        variable/constant
        | <expr1> <op> <expr2>           binary operator
        | <varid> <expr1> ... <exprn>    function appl., n >= 1
        | <conid> <expr1> ... <exprn>    constructor appl., n >= 0
        | "[" <exprs> "]"                list
        | "()"                           empty tuple
        | "(" <expr> "," <exprs> ")"     tuple
        | "<" <exprs> ">"                vector
        | "case" <expr> "of" <matches>   case expression
        | "if" <expr1> "then" <expr2>    conditional
                       "else" <expr3>
        | "let" <fundecls> "in" <expr>   local definition
        | <expr> "::" <exprtype>         type cast/view
        | <expr> "as" <exprtype>         type coercion
        | "raise" <exnid> <expr>         raise an exception
        | <expr> "within" <cexpr>        timeout
        | "(" <expr> ")"                 grouping

<cexpr> ::= <expr>                       constant expression

<exprs> ::=
        <expr0> "," ... "," <exprn>      n >= 0

<matches> ::=
        <match1> "|" ... "|" <matchn>    n >= 1

<match> ::=
       <patt> "->" <expr>
```

Constants

```
<constant> ::=
          <intconst>
        | <floatconst>
        | <boolconst>
        | <charconst>
        | <stringconst>
        | <wordconst>
        | <timeconst>
```

Patterns

```
<patt> :: =
          <constant>
        | <varid>                        variable
        | <conid>                        nullary constructor
        | "_"                            wildcard
        | "[" <patts> "]"                list pattern
```

```
        | "<" <patts> ">"              vector pattern
        | "()"                         empty tuple pattern
        | "(" <patt> "," <patts> ")"   tuple pattern
        | <conid> <patt1> ... <pattn>  discr. pattern, n >= 1
        | "(" <patt> ")"               grouping

<patts> ::=
        <patt0> "," ... "," <pattn>    n >= 0
```

Coordination language

```
<boxdecl> ::= <prelude> <body>

<prelude> ::=
        "box" <boxid>
        "in"  <inoutlist>
        "out" <inoutlist>
        [ "handles" <exnidlist> ]

<inoutlist> ::=
        <inout1> "," ... "," <inoutn>          n >= 1

<inout> ::=
        <varid> "::" <exprtype>
```

Boxes

```
<body> ::=
        "match"
        <boxmatches>
        [ "timeout" <cexpr> ]
        [ "handle" <handlers> ]

<handlers> ::=
        <handler1> "|" ... "|" <handlern>      n >= 1

<boxmatches> ::=
        <matches>
```

```
<handler> ::=
        <hpatt> "->" <cexpr>

<hpatt> ::=
        <exnid> <patt1> ... <pattn>             n >= 1
```

Replication

```
<replication> ::=
"replicate" <boxid> "as" <boxid> [ "*" <intconst> ]
```

Wiring

```
<wiredecl> ::=
        "wire" <boxid> <sources> <dests>

<sources>/<dests> ::=
        "(" <link1> "," ... "," <linkn> ")"     n >= 0

<link> ::=
          <connection>
        | <strid>
        | <portid>

<connection> ::= <boxid> "." <varid>
```

Identifiers

```
<id>     ::= [ <modid> "." ] <localid>

<idlist> ::= <id1> "," ... "," <idn>            n >= 1

<varids> ::= <varid1> ... <varidn>              n >= 0

<exnidlist> ::= <exnid1> "," ... "," <exnidn>   n >= 1

<boxid>  ::= <id>
<modid>  ::= <id>
<exnid>  ::= <id>
<conid>  ::= <id>
<typeid> ::= <id>
<varid>  ::= <id>
<strid>  ::= <id>
<portid>  ::= <id>
```

Lexical Syntax

```
<localid> ::= <letter> ( <letter> | <digit> ) *

<op> ::= ( "+" | "-" | "*" | "/" ... ) *

<intconst>    ::= <digit> +
<floatconst>  ::= <intconst> "." <intconst> [ "e" <intconst> ]
<boolconst>   ::= "true" | "false"
<charconst>   ::= "'" <char> "'"
<stringconst> ::= """ <char> * """
<wordconst>   ::= "0x" <hexdigit> +
<timeconst>   ::= <intconst> <timedes>
<timedes>     ::= "ps" | "ns" | "us" | "ms" | "s" | "min"
<char>        ::= "A" | ... | "Z" | " " | "\t" | "\n" | "\\" |
                  "0x" <hexdigit> +
```

Hume Functions and Operators

Int

```
+, -, *, div :: Int -> Int -> Int
=, /=, <=, <, >, >= :: Int -> Int -> Bool
```

Nat

```
+, -, *, div :: Int -> Int -> Int
=, /=, <=, <, >, >= :: Int -> Int -> Bool
```

Word

```
+, - :: Word -> Nat -> Word
=, /=, <=, <, >, >= :: Word -> Word -> Bool
```

Vectors

```
range :: vector
```

Tuples

```
=, /=, <=, <, >, >= :: (a1, .., an) -> (a1, ..., an) -> Bool
```

Lists

```
++ :: [a] -> [a] -> [a]
member :: [a] -> a -> Bool
=, /=, <=, <, >, >= :: [a] -> [a] -> Bool
```