Programming Languages
Message-Passing Concurrency

Dr Russ Ross

Dixie State University—Computer and Information Technologies

Fall 2014
The message-passing concurrent model

Reading: CTM §5.1
Extending the declarative model

- The declarative concurrent model cannot have observable nondeterminism. This makes client/server programs impossible, except in the limited case where the server always knows which client its next message will come from.

- We extend the model with asynchronous communication channels to give the *message-passing concurrent model*. Many clients can send messages to a single channel, which the server can read.

- The order in which the messages are added can affect the outcome, so different results are possible with the same inputs. The message-passing model is nondeterministic and therefore not declarative.

- All messages sent to a given *port* become part of a single, readonly stream.
The kernel language is extended with two new instructions:

\[ \langle s \rangle ::= \]

- `skip`
- `\langle s\rangle_1 \langle s\rangle_2`
- `local \langle x \rangle \text{ in } \langle s \rangle \text{ end}`
- `\langle x \rangle_1 = \langle x \rangle_2`
- `\langle x \rangle = \langle v \rangle`
- `if \langle x \rangle \text{ then } \langle s\rangle_1 \text{ else } \langle s\rangle_2 \text{ end}`
- `case \langle x \rangle \text{ of } \langle \text{pattern} \rangle \text{ then } \langle s\rangle_1 \text{ else } \langle s\rangle_2 \text{ end}`
- `\{ \langle x \rangle, \langle y \rangle_1, \ldots, \langle y \rangle_n \}`
- `thread \langle s \rangle \text{ end}`
- `\{ \text{NewName } \langle x \rangle \}`
- `\langle y \rangle = !! \langle x \rangle`
- `try \langle s\rangle_1 \text{ catch } \langle x \rangle \text{ then } \langle s\rangle_2 \text{ end}`
- `raise \langle x \rangle \text{ end}`
- `\{ \text{NewPort } \langle y \rangle \langle x \rangle \}`
- `\{ \text{Send } \langle x \rangle \langle y \rangle \}`
Ports

A port is an ADT with two operations:

- \{\text{NewPort } S \ P\}\: create a new port with entry point \(P\) and stream \(S\).
- \{\text{Send } P \ X\}\: append \(X\) to the stream corresponding to the entry point \(P\).

Sends from the same thread appear on the stream in order, so the port is an asynchronous FIFO channel:

```plaintext
declare S P in
{NewPort S P}
{Browse S}
{Send P a}
{Send P b}
```

The end of the stream is a read-only variable; the only way to extend it is with \{\text{Send}\}.
Port objects

**Reading:** CTM §5.2
A port object is a combination of one or more ports and a stream object. Unlike stream objects, port objects can handle many-to-one communications.

Variations on port objects go by the names “agents”, Erlang “processes”, and “active objects” all mean roughly the same thing.

Port objects can create new port objects, and they can send references to port objects in messages, so the graph of port streams can evolve as the program executes.

When modeling distributed systems, a port objects models a single OS process or a single computer node. A distributed algorithm is one that operates between port objects.
A port object has the following structure:

```
declare P1 P2 ... Pn in
local S1 S2 ... Sn in
    {NewPort S1 P1}
    {NewPort S2 P2}
    ...
    {NewPort Sn Pn}
thread {RP S1 S2 ... Sn} end
end
```

The recursive procedure $\text{RP}$ reads the port streams and performs some action based on each message received.

Sending a message to the port object just means sending a message to one of the ports.
Port objects

This port object displays all the messages it receives:

```
declare P in
local S in
    {NewPort S P}
    thread {ForAll S Browse} end
end
```

Note that `Browse` is a one-argument procedure. Rewritten using the `for` loop syntax:

```
declare P in
local S in
    {NewPort S P}
    thread for M in S do {Browse M} end end
end
```

Doing `{Send P hi}` will eventually display `hi`. This can come from any thread. In stream objects, the next message in the stream always comes from a known thread.
The **NewPortObject** abstraction

An abstraction simplifies the creation of port objects. For port objects with a single port, we need an initial state, and a state transition function:

```plaintext
fun {NewPortObject Init Fun}
Sin Sout in
  thread {FoldL Sin Fun Init Sout} end
{NewPort Sin}
end
```

This uses `FoldL` to implement the accumulator loop. When the input stream terminates, `Sout` holds the final state. For port objects with no internal state, it is even simpler:

```plaintext
fun {NewPortObject2 Proc}
Sin in
  thread for Msg in Sin do {Proc Msg} end end
{NewPort Sin}
end
```
Players tossing a ball

Imaging three players standing in a circle, tossing a ball among themselves. Each time a player catches the ball, she randomly picks one of the other two and tosses it to her.

We can model this with three port objects, each of which has a reference to the other two. When a player receives the ball, it immediately sends it to another.
Players tossing a ball

The player:

```fun {Player Others}
  {NewPortObject2
    proc {\$ Msg}
      case Msg of ball then
        Ran={OS.rand} mod {Width Others} + 1
        in
          {Send Others.Ran ball}
        end
      end
  end
end```

Others is a tuple containing the other players. We create three players and start it by tossing a single ball:

P1={Player others (P2 P3)}
P2={Player others (P1 P3)}
P3={Player others (P1 P2)}

{Send P1 ball}
Simple message protocols

**Reading**: CTM §5.3
In RMI, the client sends a request to the server and then waits for the server to send back a reply. We ignore details about how data structures are passed from one address space to another.

The server:

```
proc {ServerProc Msg} 
  case Msg 
    of calc(X Y) then 
      Y=X*X+2.0*X+2.0 
    end 
  end

Server={NewPortObject2 ServerProc}
```

This server has no internal state. It does a calculation and returns the result by binding a value to $Y$, which is the second argument of the `calc` request message.
The client

The client sends a pair of requests, one after the other:

```
proc {ClientProc Msg}
  case Msg
    of work(Y) then Y1 Y2 in
      {Send Server calc(10.0 Y1)}
      {Wait Y1}
      {Send Server calc(20.0 Y2)}
      {Wait Y2}
      Y=Y1+Y2
  end
end
Client={NewPortObject2 ClientProc}
{Browse {Send Client work(\$)}}
```

Note the use of the nesting marker “\$”. Recall that the last line is equivalent to:

```
local X in {Send Client work(X)} {Browse X} end
```
Notes

- The client references the server directly, but the server definition knows nothing about the client.

- The server gets an indirect reference to the client through its argument $\gamma$. The client waits for an answer (for $\gamma$ to be bound) before continuing.

- All messages are executed sequentially by the server. This is usually the simplest way to implement RMI.

- In distributed systems, the client and server may not be implemented in the same OS process or even using the same language, e.g., using CORBA or XMLRPC. The techniques we will explore still apply in this more general case.
Asynchronous RMI

With asynchronous RMI, the client continues execution immediately after sending the request, and is notified when the response arrives. Multiple requests can be outstanding at the same time. The client:

```
proc {ClientProc Msg}
  case Msg
    of work(?Y) then Y1 Y2 in
      {Send Server calc(10.0 Y1)}
      {Send Server calc(20.0 Y2)}
      Y=Y1+Y2
  end
end

Client={NewPortObject2 ClientProc}
{Browse {Send Client work($)}}
```

The message sends overlap, and the client waits for both replies ($Y_1$ and $Y_2$) before doing the addition.

The server is the same sequential process as with standard RMI.
Sometimes the server needs to contact the client in order to fulfill the request. Here the server contacts the client to get an extra parameter:

```
proc {ServerProc Msg}
    case Msg
    of calc(X ?Y Client) then X1 D in
        {Send Client delta(D)}
        X1=X+D
        Y=X1*X1+2.0*X1+2.0
    end
end
```

The server knows where to find the client because a reference is included in the request message.

X+D implies {Wait D}, so we omit it.
The client

The client accepts two kinds of messages: one to start a unit of work, and another to answer a callback from the server:

```plaintext
proc {ClientProc Msg}
    case Msg
        of work(?Z) then
            Y in
                {Send Server calc(10.0 Y Client)}
            Z=Y+100.0
        [] delta(?D) then
            D=1.0
    end
end
```

Client={NewPortObject2 ClientProc}
{Browse {Send Client work($)}}

Problem: this deadlocks during the {Send Client work(Z)} call.

Why?
Breaking the deadlock

The client must not wait for the reply. It must continue immediately after making its call, so that it is ready to accept the callback request. When the reply does come, it must still be handled:

```plaintext
proc {ClientProc Msg}
  case Msg
    of work(?Z) then Y in
      {Send Server calc(10.0 Y Client)}
    thread Z=Y+100.0 end
    [] delta(?D) then
      D=1.0
    end
  end
end
Client={NewPortObject2 ClientProc}
{Browse {Send Client work($)}}
```

When the \{Send Client work(Z)\} call completes, \(Z\) will usually not be bound yet. Normally this is okay, because the operation that needs \(Z\) will block until it is available. Or \{Wait Z\} is always an option.
If creating a new thread per request is not an option, the client can pass a *continuation* to the server. When the server is done, it passes the continuation back to the client, and the client uses it to resume. The client never waits, so there is no deadlock.

The server:

```plaintext
proc {ServerProc Msg} 
    case Msg 
        of calc(X Client Cont) then X1 D Y in 
            {Send Client delta(D)}
            X1=X+D
            Y=X1*X1+2.0*X1+2.0
            {Send Client Cont#Y}
        end
    end
Server={NewPortObject2 ServerProc}
```

When it is finished, the server sends Cont#Y back to the client to signal it to resume. Note that including Y is necessary.
The client

The client now understands three message types:

```plaintext
proc {ClientProc Msg}
  case Msg
  of work(?Z) then
    {Send Server calc(10.0 Y Client cont(Z))}
  [] cont(Z)#Y then
    Z=Y+100.0
  [] delta(?D) then
    D=1.0
  end
end
Client={NewPortObject2 ClientProc}
{Browse {Send Client work($)}}
```

When the `cont(Z)#Y` message is received, the client continues where it left off.

As before, `Z` will probably not be bound by the time the `Send` to the client returns.
A powerful generalization it to pass a procedure instead of a record. The server is unchanged, but the client becomes:

```plaintext
proc {ClientProc Msg}
    case Msg
        of work(?Z) then
            C=proc {$ Y} Z=Y+100.0 end
        in
            {Send Server calc(10.0 Y Client cont(C))}
    [] cont(C)#Y then
        {C Y}
    [] delta(?D) then
        D=1.0
end
end
Client={NewPortObject2 ClientProc}
{Browse {Send Client work($)}}
```

The continuation contains the work that remains to be done. It is a self-contained procedure value (a closure) that can be run anywhere.
Error reporting

What if an error occurs on the server? What if there is a network problem? What if the server is no longer running? For the first case, the server can signal an error explicitly:

```
proc {ServerProc Msg}
    case Msg
        of sqrt(X Y E) then
            try
                Y={Sqrt X}
                E=normal
            catch Exc then
                E=exception(Exc)
            end
        end
    end
end
```

Server={NewPortObject2 ServerProc}

The server passes back an extra argument that indicates whether or not the operation succeeded.
Error reporting

A synchronous client can detect the error and repeat the exception:

```
{Send Server sqrt(X Y E)}
case E of exception(Exc) then raise Exc end end
```

The `case` statement blocks until $E$ is bound, at which point an exception has occurred and been detected, or $Y$ is bound to a normal result. In an asynchronous protocol, the client must check $E$ before trying to use $Y$.

This assumes that the server can catch the exception and pass it back. If the network is down or the server fails in other ways, other techniques are necessary to detect and recover from the failure. This is a harder problem, and the subject of CS 3410: Distributed Systems.
These techniques can be mixed and matched. For example, multiple asynchronous RMIs where each RMI does a callback.

The server:

```plaintext
proc {ServerProc Msg} 
case Msg 
of calc(X ?Y Client) then X1 D in 
  {Send Client delta(D)} 
  thread 
    X1=X+D 
    Y=X1*X1+2.0*X1+2.0 
  end 
end 
end
```
The client

And the client:

```
proc {ClientProc Msg}
  case Msg
  of
    work(?) then Y1 Y2 in
      {Send Server calc(10.0 Y1 Client)}
      {Send Server calc(20.0 Y2 Client)}
      thread Y=Y1+Y2 end
    [] delta(?) then
      D=1.0
    end
  end
```

What would happen if the server did not create a thread for doing the work after the callback?

Hint: consider how the message diagram for `{Send Client work(Y)}` would change...
Double callbacks

What if the client calls the server, which issues a callback to the client, which in turn must contact the server? To handle this, both client and server must avoid blocking.

The server:

```prolog
proc {ServerProc Msg}
  case Msg
    of calc(X ?Y Client) then X1 D in
        {Send Client delta(D)}
        thread
            X1 = X + D
            Y = X1 * X1 + 2.0 * X1 + 2.0
        end
    [] serverdelta(?S) then
        S = 0.01
    end
  end
```

Dr Russ Ross  (Dixie State University)
The client

And the client:

```plaintext
proc {ClientProc Msg}
  case Msg
    of work(Z) then Y in
      {Send ServerProc calc(10.0 Y Client)}
      thread Z=Y+100.0 end
    [] delta(?D) then S in
      {Send Server serverdelta(S)}
      thread D=1.0+S end
  end
end
```

The \( D=1.0+S \) statement does not need to be in a thread in this particular case, but it would be necessary if the server needed to do an additional callback.

If you are going to go to the trouble of making it asynchronous, you might as well go all the way and accommodate future changes.
Program design for concurrency

**Reading:** CTM §5.4
To design a concurrent application, first model it a set of concurrent activities that interact in well-defined ways.

- Each activity is modeled by one concurrent component, sometimes called an *agent*

- Agents can be *reactive*, meaning they have no internal state and respond to each request independently

- Agents can also have internal state, and vary their behavior based on the chain of requests

- In component-based programming, agents are usually simple entities with little intelligence built in

- In artificial intelligence, agents are usually considered as doing some kind of reasoning
Concurrent components

In our model, a *concurrent component* is a procedure with inputs and outputs.

- When run, it creates a *component instance*, which is a port object
- The inputs are ports read by the component
- The outputs are ports to which the component sends
- Subcomponents can be composed in the procedure, hiding some inputs and outputs and only exposing the external interface
- Components interact only through *wires*—the set of inputs and outputs
- Dataflow variables can be used as *one-shot wires*, useful for values that do not change or one-time messages (like acknowledgements)
Basic operations

There are four basic operations in component-based programming:

1. Instantiation: creating an instance of a component. Usually independent of other instances, but possibly with a shared dependency.

2. Composition: building a new component out of other components. The subcomponents are normally independent and concurrent.

3. Linking: combining instances by connecting inputs and outputs together. Links can be one-shot or many-shot; inputs can be connected to one or many outputs; outputs can be connected to one or many inputs, etc.

4. Restriction: restricting the visibility of inputs or outputs to within a compound component. Wires used inside the component may not be available outside the component.
Example

Recall the digital logic circuit simulation, written here as a component:

```
proc {Latch C DI ?DO}
  X Y Z F
in
  {DelayG DO F}
  {AndG F C X}
  {NotG C Z}
  {AndG Z DI Y}
  {OrG X Y DO}
end
```

The latch component has five subcomponents, linked together by inputs and outputs. For example, the output $X$ of the first $\text{And}$ gate is given an input to the $\text{Or}$ gate. Only the wires $C$, $DI$, and $DO$ are visible to the outside of the latch. The wires $X$, $Y$, $Z$, and $F$ are restricted to the inside of the component.
Design methodology

Designing concurrent programs is harder than designing sequential programs. Following some rules can help:

- **Specification**: write down what it is supposed to do, as precisely as possible

- **Components**: List all the different concurrent activities and make each one a component. Draw a block diagram showing how they interact.

- **Message protocols**: Design the message protocols between components. Draw the component diagram with all message protocols.

- **State diagrams**: Write down the state diagram for each component. Verify that all appropriate messages are received and sent with the right conditions and actions.
Message-passing concurrency is closely related to declarative concurrency, so many components can be programmed as list operations:

\[
AL = \{ \text{Map PL fun } \{\$ P\} \text{ Ans in } \{\text{Send P query(\text{foo Ans})}\} \text{ Ans end} \}
\]

Here \text{Map} broadcasts queries to \text{PL}, a list of ports, and collects their replies in a list. It sends the message \text{query(\text{foo Ans})} to each one, and they bind the answer to \text{Ans}. \text{Map} collects the answers into a list:

\[
AL = \{ \text{Map PL fun } \{\$ P\} \{\text{Send P query(\text{foo $})}\} \text{ end} \}
\]

With the nesting operator, we do not even have to refer to \text{Ans} directly. We can start computing on \text{AL} immediately; the calculation will automatically wait if needed.
Lift control system

Reading: CTM §5.5
Lift control system

We will model a lift (elevator) control system of a single building:

- There are a fixed number of lifts, each with a controller
- There are a fixed number of floors to travel between
- Users come and go; each presses the call button to summon the lift to a floor, and then “calls” the lift to his target floor
- Each lift has a schedule, which is a list of floors it will visit in order
- The lift uses a simple, FCFS (first-come, first-served) schedule, which means it always travels directly to the next floor on its schedule
- When the lift arrives at a scheduled floor, the doors open and stay open for a fixed amount of time before closing. The lift takes a fixed amount of time to travel from one floor to the next.
Lift control system

There are four kinds of components:

1. Floors
2. Lifts
3. Controllers (to handle lift motion)
4. Timers (to handle the real-time aspect of the system)

Note that users are not components: they are the clients of the system, and interact directly with the input ports of the components.

Our controllers are dumb: FCFS scheduling means the lift will travel right past a waiting user to get to its next scheduled floor.
State transition diagrams

Port objects can often be modeled with *state transition diagrams*, which are finite state automata. The object is always in one well-defined state. In response to input, it can take some action and transition to another state.

Our states may be parameterized by a variable, so they are not technically finite, but the model still works.

Messages are sent to a port, or values are bound to dataflow variables. Dataflow variables act as “one-shot wires”, which are a handy way to send acknowledgements.

To model time delays, we use a timer protocol: the caller `Pid` sends the message `starttimer(N Pid)` to a timer agent to request a delay of `N` ms. The caller then continues, and when the time is up, the timer agent sends a message `stoptimer` back to the caller.
We will consider each component separately. We will define functions to create each one:

- \{\text{Floor Num Init Lifts}\} \text{ returns a floor } Fid \text{ with number Num, initial state Init, and lifts Lifts.}
- \{\text{Lift Num Init Cid Floors}\} \text{ returns a lift } Lid \text{ with number Num, initial state Init, controller Cid, and floors Floors.}
- \{\text{Controller Init}\} \text{ returns a controller Cid.}
The timer

The timer is simple:

```plaintext
fun {Timer}
{NewPortObject2
   proc {$ Msg}
      case Msg of starttimer(T Pid) then
         thread {Delay T} {Send Pid stoptimer} end
      end
   end
end
```
The controller

The controller is the simplest. It has two states: motor stopped and motor running. The first part implements the running state:

```
fun {Controller Init}
  Tid={Timer}
  Cid={NewPortObject Init
       fun {state(Motor F Lid) Msg}
         case Motor
           of running then
             case Msg
               of stoptimer then
                 {Send Lid 'at'(F)}
                 state(stopped F Lid)
             end
           end
         end
   end
```

When running, it only responds to a stoptimer message, indicating that it has arrived at the next floor. It notifies the lift of its new floor, and transitions to the stopped state.
The controller

When stopped it responds to `step(Dest)` messages, which request that it move one floor closer to `Dest`. It is either there already, or it starts the motor and waits 5 seconds for the lift to get to the next floor.

```plaintext
[] stopped then
  case Msg
    of step(Dest) then
      if F==Dest then
        state(stopped F Lid)
      elseif F<Dest then
        {Send Tid starttimer(5000 Cid)}
        state(running F+1 Lid)
      else % F>Dest
        {Send Tid starttimer(5000 Cid)}
        state(running F-1 Lid)
    end
  end
end
```

The floor

Floors have three states; the first is when no lift has been called:

fun {Floor Num Init Lifts}
  Tid={Timer}
  Fid={NewPortObject Init
  fun {$ state(Called) Msg}
    case Called
    of notcalled then Lran in
      case Msg
      of arrive(Ack) then
        {Browse 'Lift at floor '#Num#': open doors'}
        {Send Tid starttimer(5000 Fid)}
        state(doorsopen(Ack))
    [] call then
      {Browse 'Floor '#Num#' calls a lift!'}
    Lran=Lifts.(1+{OS.rand} mod {Width Lifts})
    {Send Lran call(Num)}
    state(called)
  end
end

call comes from a user, arrive(Ack) from a lift.
Next is the called state, waiting for a lift to arrive. This state is reached after sending a `call(Num)` message to a specific lift:

```plaintext
[] called then
  case Msg
    of arrive(Ack) then
      {Browse 'Lift at floor '#Num#': open doors'}
      {Send Tid starttimer(5000 Fid)}
      state(doorsopen(Ack))
  [] call then
    state(called)
end
```

The lift announces its arrival (called or not) with an `arrive(Ack)` message. If other users press the call button, no new lift is summoned.

The lift waits for `Ack` to be bound before it can close the doors and move on, so this is stored in the state.
The floor

Finally, when the floor has a lift with its doors open:

```plaintext
[] doorsopen(Ack) then
  case Msg
  of stoptimer then
    {Browse 'Lift at floor '#Num#': close doors'}
    Ack=unit
    state(notcalled)
  [] arrive(A) then
    A=Ack
    state(doorsopen(Ack))
  [] call then
    state(doorsopen(Ack))
  end
end
```

After the wait, it signals the lift to close its doors and leave. Additional lifts that arrive are tied to the same (future) signal, and incoming calls are effectively ignored (the user can jump in an open lift).
The lift

The lift maintains a queue of floors it is to visit. To add a new floor:

```
fun {ScheduleLast L N}
  if L=nil andthen {List.last L}==N then L
  else {Append L [N]} end
end
```

The lift has four states, but we implement it by first considering messages. It can receive two port messages:

1. call(N): either a floor has summoned the lift, or a user on the lift has requested a destination floor.

2. ’at’(NewPos): the controller signals that the lift has arrived at a new floor (real lifts would get such a signal from a sensor).

In addition, it send an Ack variable to a floor when it arrives, and waits for it to be bound to signal that it can leave the floor.
The lift

In response to a call(N) message:

```
fun {Lift Num Init Cid Floors}
  {NewPortObject Init
   fun {$ state(Pos Sched Moving) Msg}
     case Msg
     of call(N) then
       {Browse 'Lift '#Num#' needed at floor '#N'}
       if N==Pos andthen {Not Moving} then
         {Wait {Send Floors.Pos arrive($)}}
         state(Pos Sched false)
       else Sched2 in
         Sched2={ScheduleLast Sched N}
       end
       if {Not Moving} then {Send Cid step(N)} end
       state(Pos Sched2 true)
   end
  }
end
```

It is either stopped at that floor already (and must wait for the doors to close), or it must add the floor to the schedule. If it was sitting idle, it must also ask the controller to start moving.
The lift

At the next floor, either make a scheduled stop or keep moving:

```pseudocode
[] 'at'(NewPos) then
{Browse 'Lift '#Num#' at floor '#NewPos'}
case Sched of S|Sched2 then
  if NewPos==S then
    if Sched2==nil then
      state(NewPos nil false)
    else
      {Send Cid step(Sched2.1)}
      state(NewPos Sched2 true)
    end
  else
    {Send Cid step(S)}
    state(NewPos Sched Moving)
  end
end
end
```
The building

A compound building component combines all the pieces:

```plaintext
proc {Building FN LN ?Floors ?Lifts}
    Lifts={MakeTuple lifts LN}
    for I in 1..LN do Cid in
        Cid={Controller state(stopped 1 Lifts.I)}
        Lifts.I={Lift I state(1 nil false) Cid Floors}
    end
    Floors={MakeTuple floors FN}
    for I in 1..FN do
        Floors.I={Floor I state(notcalled) Lifts}
    end
end
```
We can use it for a building with 10 floors and 2 lifts:

```plaintext
declare F L in
{Building 10 2 F L}
{Send F.9 call}
{Send F.10 call}

{Delay 500}
{Send L.1 call(4)}
{Send L.2 call(5)}
```

Lifts are called to floors 9 and 10, and then lift 1 is called to floor 4, and lift 2 to floor 5.

If the first two calls happen to go to different lifts, then this simulates two users going from floors 9 and 10 to 4 and 5 (or 5 and 4).
Improvements to the lift control system

The controller only ever communicates with its lift. We can combine the controller and lift into a compound component, called a lift shaft:

```hs
fun { LiftShaft I state(F S M) Floors }
   Cid={Controller state(stopped F Lid)}
   Lid={Lift I state(F S M) Cid Floors}
in Lid end
```

Then Building can be simplified:

```hs
proc { Building FN LN ?Floors ?Lifts }
   Lifts={MakeTuple lifts LN}
   for I in 1..LN do Cid in
      Lifts.I={LiftShaft I state(1 nil false) Floors}
   end
   Floors={MakeTuple floors FN}
   for I in 1..FN do
      Floors.I={Floor I state(notcalled) Lifts}
   end
end
```
The Erlang language

Reading: CTM §5.7