Adam Sampson
atsl@kent.ac.uk

University of Kent
http://www.cs.kent.ac.uk/
Introduction
What’s this workshop about?

- occam-π for occam 2.1 users (i.e. what happens after CO516)
- Explain the new features
- Give you a chance to try them out
- Please stop me if anything isn’t clear!
Generalities

- occam-π aims to be a modern language which maintains the occam spirit
- (Mostly) backwards-compatible with occam 2.1
- A work in progress; you will find broken stuff, missing documentation, etc.
- … and things will change in the future…
- … but we have used it to build big distributed applications already
Resources

- Most things are linked from: http://occam-pi.org/
- No reference manual for occam-π yet – use the occam 2.1 manual and the OEPs
- The Systems Group Wiki:
  https://www.cs.kent.ac.uk/research/groups/sys/wiki/
  Includes checklist, occam-π style guide, OccamDoc spec, ...
- Library reference (incomplete):
  http://occam-pi.org/occamdoc/
What’s on the menu?

- Syntax changes
- Mobile data
- Mobile channel types
- Sharing channels
- Forking
- Barriers
- Extended rendezvous
- Useful libraries
Implementations

- KRoC is our occam-\(\pi\) suite for x86 systems
- KRoC has some oddities – particularly the compiler
- Make sure you’re using the latest KRoC release
- If that doesn’t work, try the latest SVN version
- If you find a bug, please report it:
  kroc-bugs@kent.ac.uk
- For occam-\(\pi\) on other platforms, see the Transterpreter:
  http://www.transterpreter.org/
Syntax changes
Syntax changes

- Mostly trivial – but they make the code clearer
- I’ll describe the common ones
- There are a few more, but you’re unlikely to need them; see the OEPs for more details
PROC head (CHAN OF INT in, out)
  INT x:
  SEQ
    in ? x
    out ! x
    black.hole (in)

:
PROC head (CHAN INT in?, out!)
  INT x:
  SEQ
    in ? x
    out ! x
  black.hole (in?)

CHAN OF → CHAN

Channel direction specifiers

Use direction specifiers wherever possible – they help catch errors sooner
In occam 2.1, you had to do this:

```plaintext
INT foo:
SEQ
  foo := 42
...
```

In occam-π, you can say:

```plaintext
INITIAL INT foo IS 42:
...
```

This is treated as a kind of abbreviation – often useful to replace an abbreviation when you want to change the variable later, e.g.

```plaintext
VAL INT x IS y: → INITIAL INT x IS y:
```
Array constructors

VAL []INT squares IS [i = 1 FOR 10 | i * i]:

- Like Haskell’s array comprehensions
- Easy way of generating an array constant
- Left hand side is a replicator
- Right hand side is an expression
- Size must be determinable at compile time (currently)
In occam, PROC parameters are passed by reference, unless you say VAL

... so PROCs can return results in variables they’re given

In occam-π, say RESULT (in the same way as VAL) to mean “this parameter is only used to return a result”

Helps the definedness checker

PROC get.random (VAL INT range, INT seed, RESULT INT number) ... :
Mobile data
[1000000]BYTE buf:
SEQ

...  
c ! buf

- Classical occam has no concept of reference types (like C pointers, or Java references)
- Doing the above will copy 1,000,000 bytes of data
[80] BYTE buf:
SEQ
  read.http.request (socket, buf)

► Classical occam has no way to dynamically allocate memory
► How can we tell at compile time how big the buffer should be?
► We want to choose the size at runtime
MOBILE []BYTE buf:
SEQ
get.request.size (socket, size)
buf := MOBILE [size]BYTE
read.http.request (socket, buf)
c ! buf

- MOBILE []BYTE indicates a mobile reference type – an array of BYTES of unknown size
- MOBILE operator allocates a new mobile with the given size
- Output only sends the reference
CHAN MOBILE []BYTE c:
PAR
SEQ
...
  c ! a.mobile
...
SEQ
...
  c ? b.mobile
...

► ...isn’t that terribly unsafe?

► In most languages, having references (pointers) leads to aliasing, where several names can refer to the same object...
In occam-π, only one name can ever refer to the same object.

...so when you communicate or assign a mobile reference somewhere else, then you lose it – it becomes undefined.

The compiler will check that you aren’t trying to operate upon an undefined value.

Don’t ignore the warnings!
INITIAL MOBILE [ ]BYTE ma IS MOBILE [123]BYTE:
MOBILE [ ]BYTE mb:

► Initially, ma is defined; mb is undefined
► Let’s do:

    mb := ma

► Now ma is undefined; mb is defined
► ... and mb refers to the array that ma used to refer to
 INITIAL MOBILE []BYTE ma IS MOBILE [123]BYTE:
 MOBILE []BYTE mb:

► (Again:) Initially, ma is defined; mb is undefined
► Let’s do:

CHAN MOBILE []BYTE c:
   c ! ma

► Now both ma and mb are undefined
You can explicitly duplicate a mobile using the CLONE operator.

(Again:) Initially, ma is defined; mb is undefined.

If we do:

mb := CLONE ma

Now both ma and mb are defined.

mb is a new mobile, with a copy of ma’s contents.

You can say c ! CLONE ma too.
Using mobiles

INITIAL MOBILE []BYTE ma IS MOBILE [123]BYTE:
MOBILE []BYTE mb:

- Mobile data types can be used just like their regular counterparts

\[
\text{ma}[42] := 'x'
\]
SEQ \(i = 0\) FOR SIZE ma
\[\text{out ! ma}[i]\]
PROC foo ([]BYTE bs) ... :
foo (ma)
VAL []BYTE bs IS ma:

- Note that ma := mb means different things for normal and mobile arrays, though
Nearly any occam *data type* can be made mobile:

```occam
MOBILE INT x:
MOBILE []INT xs:
DATA TYPE MY.RECORD
  RECORD
    INT x:
  :
MOBILE MY.RECORD r:
```

A multidimensional array is just a mobile version of a regular array – it is not an array of mobile arrays:

```occam
MOBILE [][]INT xss:
```
Nested mobiles

- We often use `MOBILE [ ] BYTE` to represent a string of arbitrary length
- It’s quite often useful to have an array of strings, all of which can be different lengths
- You can do this with `MOBILE [ ] MOBILE [ ] BYTE`
- ...i.e. a mobile array of mobile arrays of bytes
- **However**: the existing compiler has **very limited** support for nested mobiles – the above type is one of two that work
- You also can’t have a non-mobile containing a mobile...
Two ways of doing mobile records

DATA TYPE MY.RECORD.1
  RECORD
    INT x:
  :
  MOBILE MY.RECORD.1 r:

► ...means nearly as the same as ...

DATA TYPE MY.RECORD.2
  MOBILE RECORD
    INT x:
  :
  MY.RECORD.2 r:

► ...but the compiler knows that MY.RECORD.2 instances will always be mobile
Mobile data summary

- Mobiles are safe references
- Assignment and communication with reference semantics
- Only one process may hold a given mobile
Exercise 1

- Please download:  
  http://occam-pi.org/picourse/q1.occ

- Fill in the ...s

- Try using the mobiles *before* you’ve allocated them, and look at the error messages
Mobile channel types
In occam 2 programs, channels are fixed in place at compile time

... but what if we want to reconnect the process network at runtime?

For example, if we’re building a graphical process network editor...

... or a highly-dynamic biological simulation...

... or or or...

Let’s use our new mobility mechanism!
CHAN TYPE GRAPHICS.CT

MOBILE RECORD

CHAN REQUEST req?:
CHAN RESPONSE resp!:

A channel type is a bundle of one or more related channels

...for example, the set of channels connecting a client and a server

Note this has to be a CHAN TYPE, else you can’t put channels in it

Channel direction specifiers are mandatory
CHAN TYPE GRAPHICS.CT

MOBILE RECORD

CHAN REQUEST req?:
CHAN RESPONSE resp!:

When you create one, you get its two ends:

GRAPHICS.CT! client:
GRAPHICS.CT? server:
SEQ

\texttt{client, server := MOBILE GRAPHICS.CT}

We call them \textit{client} and \textit{server} ends by convention

The direction specifiers in the record are from the server end’s point of view
Argh, the specifiers!

- ! and ? are used in the type of channel type end variables too:
  
  GRAPHICS.CT! client:  
  GRAPHICS.CT? server:  

- Mnemonic: in *client-server* communication, the client always *sends* first  

- ...so the client end gets the specifier that means *send*
Using channel types

CHAN TYPE GRAPHICS.CT
  MOBILE RECORD
    CHAN REQUEST req?:
    CHAN RESPONSE resp!:
  GRAPHICS.CT! client:

- Channel types are a special kind of mobile record (that can only contain channels)
- To get at the channels inside them, use []:
  client[req] ! want.raster; 640; 480
  client[resp] ? raster; r
  CHAN REQUEST c! IS client[req]!:
And the point of this is...
Earlier I said that you can’t have a non-mobile object containing a mobile one...

...so you can’t have a regular array of ends:

\[
[4]\text{GRAPHICS.CT! clients:}
\]

But you can have a mobile array of ends. Remember it has to be allocated!

\[
\text{INITIAL MOBILE } [\ ]\text{GRAPHICS.CT! clients IS MOBILE } [4]\text{GRAPHICS.CT!:}
\]

\[
\text{clients}[0] := \text{client}
\]

(This is the other working nested mobile type that I mentioned earlier.)
Mobile channels summary

- Channel types are bundles of channels
- Allocating a channel type gives you a client end and a server end
- Channel type ends are mobile records containing channel ends
- Channel ends inside channel type ends can be used like regular channels
- If you want an array of ends, use a mobile array
- Channel types work well with the client/server design rule – but can be used in other ways too ("peer-to-peer")
Please download:
http://occam-pi.org/picourse/q2.occ

Run it and see what it does

It currently uses two channels to connect the client and server

Modify it to use a channel type:

- Add a **CHAN TYPE** declaration with two channels
- **server** and **client** should take a channel type end as a parameter, rather than a pair of channels
- **q2** will need to declare and create the channel type ends
Sharing channels
In occam 2, channels are one-to-one – as are channel types, by default.

occam-π also allows:
- any-to-one
- one-to-any
- any-to-any

We do this by declaring channel type ends as shared, using the `SHARED` keyword.
CHAN TYPE MY.CT ... :

MY.CT! normal.client:
MY.CT? normal.server:

**SHARED** MY.CT! shared.client:
**SHARED** MY.CT? shared.server:

- These are still allocated by saying:

  normal.client, shared.server :=
  MOBILE MY.CT

  (etc.)
One-to-one:
MY.CT! client:
MY.CT? server:
client, server := MOBILE MY.CT

One-to-any:
MY.CT! client:
SHARED MY.CT? server:
client, server := MOBILE MY.CT
► Any-to-one:

```
SHARED MY.CT! client:
MY.CT? server:
client, server := MOBILE MY.CT
```

► Any-to-any:

```
SHARED MY.CT! client:
SHARED MY.CT? server:
client, server := MOBILE MY.CT
```
CHAN TYPE MY.CT ... :
SHARED MY.CT! shared.client:
SHARED MY.CT? shared.server:

► When using a shared channel end, you must claim it first using a CLAIM block:

... CLAIM shared.client
    shared.client[c] ! something
...
CLAIM shared.server
    shared.server[c] ? something
...

occam 1.04159... – p.45/12
While a channel type end is claimed, nothing else can be using it – so this preserves the no-aliasing safety guarantee

And since we have this guarantee...

...communicating or assigning away a shared end does *not* cause you to lose it

Don’t claim an end for longer than you need it, because you’ll block others trying to get at it!
All this messing around with channel types is a bit awkward if you just want one shared channel...

...so there’s a shorthand:

```
SHARED! CHAN INT c:
PAR

CLAIM c!
  c ! 42

  c ? x
```

The compiler will turn this into an *anonymous channel type* automatically

**Direction specifier** indicates direction of communication
Declaring shared channels

**SHARED!** CHAN INT c:

- **SHARED** and direction specifier says what sort of channel it is:
  - Nothing means it's an ordinary channel
  - **SHARED!** means any-to-one
  - **SHARED?** means one-to-any
  - Just **SHARED** means any-to-any
Using shared channels

\[
\text{SHARED! CHAN INT } c:\n\]

- You can pass the ends as an argument to \text{PROC}s:
  
  \[
  \text{PROC reader (CHAN INT in?) \ldots : reader (c?)}
  \]
  
  \[
  \text{PROC writer (SHARED CHAN INT out!) \ldots : writer (c!)}
  \]

- The \text{PROC}s only need to care about the end they can see

- \text{reader} can just treat it like a regular channel

- \text{writer} needs to know it’s shared, and must \text{CLAIM} the channel before writing

- No direction specifiers on \text{SHARED} in args

occam 1.04159\ldots – p.49/124
One use for shared channels is error reporting – having lots of processes able to print to the screen

In occam-$\pi$, you can declare the top-level channels as $\text{SHARED}$ if you like:

```
PROC q7 (CHAN BYTE in?, out!,
         \text{SHARED} \text{CHAN BYTE} \text{err}!)
```

\ldots

\ldots and then just give $\text{err}!$ to everything that needs to be able to print error messages
Channels and channel types can be one-to-one, one-to-any, any-to-one or any-to-any – just say **SHARED**

**CLAIM** shared ends when you need them

... but *only* when you need them!

You can declare shared channels directly if you only need one
Mobility patterns
Registration: problem

- Similar to the OO observer pattern
- You’ve got a fixed server and a variable number of clients
- The server needs to be able to talk to all of the clients
- Clients can start up and shut down at any time
Have an any-to-one shared channel that new clients can write to

When a client starts up, it creates a one-to-one channel, and sends the server end to the server using the shared channel

The client can then communicate with the server along the newly-set-up private channel

... and the server can attend across all the private channels it has, waiting for requests from clients

When a client exits, it uses its private channel to send an “I’m done now” message, and the server disconnects the private channel
Channel types are often used for temporary connections to a long-lived server.

Client ends are obtained from the server somehow.

When the client is done with its client end, it should return it to the server for future reuse.

This can be done using one of the channels in the channel bundle!
CHAN TYPE GRAPHICS.CT:
CHAN TYPE GRAPHICS.CT
    MOBILE RECORD
    ...  request, response channels, etc.
    CHAN GRAPHICS.CT! shutdown?:

: GRAPHICS.CT! client:
...  get client
...  do stuff
client[shutdown] ! client

- Note forward declaration (or could say REC CHAN TYPE)
- Alternatively, shutdown could be a variant in the request protocol, rather than a separate channel: shutdown; GRAPHICS.CT!
Please download:
http://occam-pi.org/picourse/q3.occ

This is a (relatively) simple client-server program using the “Registration” pattern

Clients ask a server to roll dice for them

Note: shared channel types, shared regular channel (register), shared top-level channel (out)

Fill in the ...s in server

(You don’t need to write a lot of code – this one’s more about understanding the rest of the program)
Exercise 3 extended

► If you’re bored...
► Think how to make this use the “Snap-back” pattern too
► ... and how to make it *not* deadlock once finished
Part 2
More simple stuff
In a PROC or FUNCTION header, you can now say:

```
INLINE PROC foo (args)
```

When compiling the program, rather than compiling a call to `foo`, the compiler will just *insert* the compiled version of `foo`

No call overhead – but bigger code; trade-off against cache effects

Only use it for small PROCs
In occam 2, things do not come into scope until “after the colon” – so you can’t write a recursive PROC.

In occam-π, you can say:

```
REC PROC foo (args)
 ... foo (v)
 :
```

i.e. saying REC PROC rather than PROC makes the PROC immediately available to call inside itself.

You can go parallel with yourself recursively!
Recursive channel types

► You can use `REC` to refer to a channel type inside itself too:

```
REC CHAN TYPE FOO
  MOBILE RECORD
  CHAN FOO! return?:
```

► If you want mutually recursive channel types (or protocol definitions, etc.), you can do a *forward declaration*:

```
CHAN TYPE FOO:
(i.e. “there is a channel type called FOO that I’ll describe later”)
```
Replicator steps

- occam 2 replicators always count upwards by ones:
  \[ \text{SEQ } i = 0 \text{ FOR } 5 \]
  counts 0, 1, 2, 3, 4

- occam-π lets you specify a step size too:
  \[ \text{SEQ } i = 0 \text{ FOR } 5 \text{ STEP } 10 \]
  counts 0, 10, 20, 30, 40

- Negative steps are allowed

- Note that the \text{FOR} value is the number of steps, not the final value
sometimes you want to say “if both process A and process B are ready to run, then you should run process A first”

useful for managing latency (e.g. making user interface processes run at a high priority)

in occam 2, you had to use the PRI PAR construct to specify process priority
In occam-\(\pi\), you can explicitly fetch and adjust the priority using two new builtins:

\[
x := \text{GETPRI} \\
\text{SETPRI} (x + 5) \quad -- \text{decrease priority}
\]

- Priorities are integers from 0 (high) to 31 (low)
- Priorities are *advisory* – don’t rely on them!
Forking
In occam 2, PAR blocks have to have a fixed number of processes at compile time

Either a regular PAR with several processes inside it

... or a replicated PAR where the replicator count is a constant

In occam-π, a replicated PAR can have a dynamic replicator count:

```plaintext
INT x:
SEQ
   read.from.user (x)
PAR i = 0 FOR x
...
```
More dynamic parallelism

- ...but this assumes that you know the replicator count at the start of the PAR
- Suppose we’re writing a webserver – we don’t know in advance how many connections we’ll have
- We want to be able to spawn new processes as appropriate
- ...which is actually how concurrency works in most other languages
occam-$\pi$ introduces two new keywords – **FORKING** and **FORK**

- Inside a **FORKING** block, you can use **FORK** at any time to spawn a new process
- When the **FORKING** block exits, it’ll wait for all the spawned processes to finish
Forking example

- Spawning worker processes for incoming requests

CHAN REQUEST in?:
...
FORKING
  REQUEST r:
  WHILE TRUE
  SEQ
    in ? r
  FORK request.handler (r)
FORK request.handler (r)

- Currently **FORK** must be followed by a single **PROC** call
- All the arguments to the **PROC** must be *things you could communicate across a channel*:
  - Passed by value (i.e. **VAL**)
  - Shared
  - Mobile – in which case they are transferred to the new process
PAR replicator counts can now be dynamic

FORKING and FORK let you spawn arbitrary numbers of processes at runtime

FORK PROC arguments have communication semantics
Please download:
http://occam-pi.org/picourse/q4.occ

Modify the top-level process as suggested

When you quit the loop, note how the program doesn’t exit until all the FORKed processes are complete
Extended rendezvous
This is a bit of an oddity – but it’s very useful in some situations.

Normally, when you do a channel communication:

\[
\text{c ! x} \quad \parallel \quad \text{c ? x}
\]

whichever of the two processes gets there first waits for the other one,

they communicate,

and both are immediately able to run again
If, instead, we use the extended input operator... 

c ?? x
do.stuff ()

Does an input from channel c into x as usual

... but the process given executes while the writing process is still blocked

This means the writing process can’t continue until do.stuff () has finished running

(We call do.stuff () the rendezvous process)
We’ve thought of a couple of uses... Suppose you’ve got this network:

sender -> receiver

But it’s not working! What’s getting sent across that channel?
What you need is a “tap” process. Like this:

- But you don’t want to change the behaviour of the system when you add the tap.
So you write it like this:

```occam
PROC tap (CHAN INT in?, out!, tap!)
  INT x:
  WHILE TRUE
    in ?? x
    PAR
      out ! x
      tap ! x
  :
```

From the point of view of the sender and receiver processes, this just looks like a regular channel.
Another use

- Providing a channel interface to external hardware or software

```occam
PROC driver (CHAN FOO in?)
  FOO f:
  WHILE TRUE
    in ?? f
    SEQ
      send.req (f)
      wait.complete ()
  :.

- pony uses this to implement network channels
```
This works inside \texttt{ALT} as well – although the syntax is rather odd:

\begin{verbatim}
ALT
  c ? x
    handle.c (x)
  d ?? x
    while.blocked ()
    handle.d (x)
\end{verbatim}

Two processes after the extended input guard

The \textit{rendezvous process} is the first one

The \textit{regular guarded process} is the second one
Extended rendezvous summary

- Extended input lets you execute code while the sending process is blocked
- Useful for tap processes
- Useful for channel interfaces to other code/devices
- Probably useful for other things too? Let me know!
Barriers
Motivation

🔹 Not only do occam channels let us communicate data, they also have the effect of *synchronising* two processes
🔹 Neither the sender nor the receiver can proceed until the communication can complete
🔹 What if we want to synchronise *more than two* processes?
🔹 We use a barrier
A barrier has a number of processes enrolled upon it. When a process synchronises on the barrier, it blocks until all the enrolled processes are trying to synchronise. . .

. . . at which point they all proceed

This is equivalent to a CSP event
A process can *resign* from a barrier

Resignation is the opposite of enrollment: once you’ve resigned, all the other processes synchronise without waiting for you

It’s sometimes useful to resign temporarily
There’s a new **BARRIER** data type:

```
BARRIER  b:
```

By default, only the current process is enrolled

When using **PAR**, you can say **ENROLL** to enroll all the parallel processes on a barrier:

```
PAR  ENROLL  b
  foo  (b)
  bar  (b)
  baz  (b)
```

To synchronise, you use **SYNC**:

```
SYNC  b
```
To resign from a barrier temporarily, there’s a RESIGN block:

```
RESIGN b
... code
```

Inside the block, you cannot use `b` at all.

The compiler makes sure that you can’t `SYNC` on a barrier unless you’re enrolled upon it.
Suppose you’ve said:

```
PAR i = 0 FOR 100 ENROLL b
  worker (i, b)
```

If one worker exits, does this stop the others from synchronising on \( b \)?

No – when a process in a \texttt{PAR . . . ENROLL} block exits, it is \textit{automatically resigned} from the barrier.
You can enroll processes upon multiple barriers within the same \texttt{PAR} construct:

\begin{verbatim}
BARRIER long, short:
PAR ENROLL long, short
PAR
  long.timer (long)
  short.timer (short)
BARRIER internal:
PAR ENROLL long, short, internal:
  process.a (long, short, internal)
  process.b (long, short, internal)
\end{verbatim}
One use for barriers is to implement *phased access*

Suppose you have some shared resource that several processes have access to, but cannot be used safely in parallel

You could use semaphores, but they don’t guarantee fairness

You really want the processes to take turns
Phase Two

- Give all the processes a barrier to synchronise on.
- Divide your work up into *phases* – in phase 1, one process uses the resource; in phase 2, another does; and so forth.
- At the end of each phase, everyone syncs on the barrier.
A particularly useful instance of this pattern:

- Lots of processes share an array; each needs to update its cell, and examine some of the others
- You can read safely in parallel, but can’t mix reads and writes
- Have two phases
  - Phase 1: everybody reads the array
  - Phase 2: everybody updates only their cell
- You can implement a cellular automaton this way
Lazy Phases

- To make this more efficient, use resignation
- When a cell isn’t changing, have it resign from the barrier and go to sleep
- When propagating changes around, wake up any sleeping cells
- This means you only recalculate the areas that are changing
- For more details, see CPA2005 paper!
How do you pass a barrier to a forked process?

(since normal barriers can’t be communicated)

You need a mobile barrier

These have distinctly odd semantics
Using mobile barriers

► Like any MOBILE, you must allocate it before use:

INITIAL MOBILE BARRIER mb IS MOBILE BARRIER:

► If you hold a MOBILE BARRIER, you’re enrolled on it

► When you lose a reference to a barrier (if it goes out of scope, or you assign over it), you resign from it
Cloning mobile barriers

- When you \texttt{CLONE} one, you get \textit{another reference to the same barrier}

\[
mc := \text{CLONE } mb
\]

- Now \texttt{mc} is an alias for \texttt{mb} – this is bad!

- Imagine you had a process that took two barrier arguments; you could now give it the same barrier twice (which \texttt{occam-\pi} normally wouldn’t let you do)

- Generally you \textit{only} use this when you’re \texttt{FORKing} a process off

\[
\text{FORK worker (CLONE mb)}
\]
Barriers summary

- Generalise channel synchronisation to any number of processes
- Can use phases to control access to shared resources
- Resignation allows processes to sleep while they’re not interested
- We’re still finding uses for barriers!
Exercise 5

► Please download:
http://occam-pi.org/picourse/q5.occ

► Compile and run it – note how the rowers get out of sync fairly quickly

► Make it use barriers so they all row together
User-defined operators
DATA TYPE COORD
RECORD
    REAL32 x, y:
:

- Suppose you’ve defined a coordinate data type
- In occam 2, saying $x + y$ would produce an error
User-defined operators

DATA TYPE COORD
RECORD
    REAL32 x, y:

In occam-π, you can define what the + operator means for that data type:

COORD FUNCTION "+" (VAL COORD a, b) IS
    [a[x] + b[x], a[y] + b[y]]:

This is a user-defined operator
COORD FUNCTION "+" (VAL COORD a, b) IS
[a[x] + b[x], a[y] + b[y]]:

- Like a normal function definition, but with the operator in quotes in place of the function name
- Any operator works:
  + - / * PLUS MINUS \ / \\ ...
- Dyadic operators (as above) have two args; monadic operators have one
- You can define multiple "+" operators for different types...
- ...even regular occam types (like INT or [4]BOOL)!
- There’s clearly some deep magic going on here
This is overloading on argument types – like in C++ or Java

```
FOO FUNCTION "+" (VAL FOO a, b) IS ... :
BAR FUNCTION "+" (VAL BAR a, b) IS ... :
BAZ FUNCTION "+" (VAL BAZ a, b) IS ... :
```

When you use an operator, the compiler will look at the types of the arguments to decide which version to use

Later definitions override earlier ones

You can’t do the same with regular PROC or FUNCTION arguments (at least yet)
Many people think operator overloading is a bad idea.

What looks like a simple operation might actually be doing some big expensive calculation.

It’s easy to be deliberately perverse:

```occam
INT FUNCTION "+" (VAL INT a, b) IS a - b:

ASSERT ((4 + 3) = 1)
```

And why are 4 and 3 there INTs and not, say, BYTES? There are special rules for literals and UDOs...

Tread very carefully when using this!
Please download:
http://occam-pi.org/picourse/q6.occ

Implement + and * for COMPLEX

Remember * as a string literal is "**"
Protocol inheritance
In OO design, objects have interfaces with methods.

When we want to add new functionality, we extend the interface with more methods.

In process-oriented design, process communicate using protocols.

To add new functionality, we extend existing protocols with new messages.
PROTOCOL A
CASE
  foo; INT
  bar
:
PROTOCOL B EXTENDS A
CASE
  baz
:

- The B protocol now has foo, bar and baz variants
- (KRoC limitation: A and B must be declared in the same source file)
PROTOCOL B EXTENDS A ... :
PROC sends.a (CHAN A out!) ... :
PROC reads.b (CHAN B in?) ... :
CHAN B c:
PAR
    sends.a (c!)
    reads.b (c?)

- A process that outputs using $A$ can be connected to a channel carrying $B$
- No need to change $sends.a$ when we extend the $A$ protocol
You can extend *multiple* protocols:

```plaintext
PROTOCOL MANY EXTENDS ONE, TWO:
```

- Doing this means you pick up all the variants from `ONE` and `TWO`
- Variants with the same *name* must have the same *structure*
- ...but might not necessarily have the same *meaning*!
- This is isomorphic to OO multiple inheritance – which is generally considered a really bad idea; be cautious
Protocol inheritance summary

- You can extend an existing protocol with new variants
- Processes writing to channels of the old protocol can write to channels of the extended protocol
Exercise 7

► Please download:
  http://occam-pi.org/picourse/q7.occ

► We have some clients and an FM/MW radio

► ...but we’ve just bought a shiny new radio with DAB

► Make the radio support DAB via a new protocol that extends TUNER

► ...without changing the client code
Writing real programs
- So now you know the language.
- What else do you need for a real occam-π application?
- Libraries!
- I’ll go through some of the useful ones.
- It’s a mess – we’ll tidy it up in the near future.
Using libraries

- Include the appropriate headers and USE the .lib:
  
  #INCLUDE "consts.inc"
  #USE "course.lib"

- Link with -llibname (and other libraries as required)

  kroc my.occ -lcourse

- This should all be in the OccamDoc...
- occam 2 programs had `hostio`, `hostsp`, etc.
- These days we don’t normally use those – mostly because everybody’s used to using the `course` library...

- `out.int` etc. are in the `course` library
- `filelib` contains various POSIX bindings
  - In particular, `file.get.options`, a `getopt`-style option parser; please use it instead of `ask.int` when getting parameters
- **socklib** has most of the standard POSIX networking stuff
- The occam web server’s built on this
- For transparently-networked occam-π applications, there’s pony: network channels that behave like regular occam channels
- See Mario’s thesis
- \texttt{sdldraster} provides trivial 2D bitmap graphics
- Adam’s got a 2D vector graphics package, and audio output bindings
- Damian’s OpenGL bindings do accelerated 3D
- Carl’s video library handles various media types and video IO
Doing your own bindings

- There are several ways of binding to C code from occam-π
- The “old” FFI interface – simple, a bit awkward to use
- Damian’s SWIG patches – automatically generate bindings from C headers
- CIF – occam-like concurrency and channel communications in C
- Plenty of examples around if you’re interested
- Standard for inline documentation – like JavaDoc
- See the Wiki for the syntax; it’s pretty obvious
  ```occam
  --* Launch the nuclear missiles
  PROC launch.missiles () ... :
  ```
- The `occamdoc` program converts these to HTML (via XML and XSLT, so other formats also doable)
  ```bash
  occamdoc -d outputdir *.occ *.inc
  ```
- Some libraries have OccamDoc markup already
Please download:
http://occam-pi.org/picourse/q8.occ

Draw some pretty graphics!

For example, “munching squares”:

clear the screen
for each T from 0 .. (width - 1)
  for each X from 0 .. (width - 1)
    plot the point (X, X xor T)
draw the screen
That’s all, folks!