



# OCCAM

A software archaeology presentation

Andrei Cojocaru

Nicolae Pavel

MOC2

# IN THE BEGINNING THERE WAS THE TRANSPUTER

- A general purpose microprocessor built specifically for parallel computing
- First released in 1984, shut down around 1989
- Was used for image processing, data acquisition, virtual reality, and even in space





# THE LEGACY: OCCAM

- An imperative programming language developed by INMOS to match their transputer's capabilities
- Built in parallelism and message passing between independent processes
- Pretty low level, comparable to early C

# ARCHAEOLOGY: A WHINE

- One modern-ish version of Occam, the Kent Retargetable occam Compiler
- OS X build requires installing an old version of Apple's GCC, compiling an old version of LLVM with it, and only then compiling the occam compiler
- Linux build only works out of the box with an Ubuntu from 2011
- Windows... XP... is supported



# HELLO WORLD

```
#INCLUDE "course.module"  
PROC hello (CHAN BYTE out!)  
  out.string ("Hello, world!*n", 0, out!)  
:
```

- Keywords are CAPITALIZED
- Strings are byte arrays
- Statements are **processes**
- out! is a communication **channel** between processes, carrying BYTEs
- the ! signifies sending, ? is receiving
- indentation is 2 spaces **exactly**



# SEQ AND PAR

```
#INCLUDE "course.module"
PROC sequential (CHAN BYTE
out!)
  INT x, y, z:
  SEQ
    x := 2
    y := 3
    z := 4
  :
```

```
#INCLUDE "course.module"
PROC parallel (CHAN BYTE
out!)
  INT x, y, z:
  PAR
    x := 2
    y := 3
    z := 4
  :
```

```
#INCLUDE "course.module"
PROC parallel (CHAN BYTE
out!)
  INT x, y, z:
  PAR
    x := 2
    x := 3
    z := 4
  :
```

- Types: INT, BYTE, REAL32, REAL64
- Sequentiality is explicit with the SEQ **constructor**
- Replacing it with PAR executes all **processes** in parallel
- Writing to the same variable in parallel is impossible, the right box is a compile error!

# CONSTRUCTORS

- IF statements

IF

```
x = y
  foo (x)
y = 0
  bar (x)
TRUE
  SKIP
```

```
if (x == y) {
    foo (x);
} else if (y == 0) {
    bar (x);
} else {
    /* nothing! */
}
```

IF

```
FALSE
  SKIP
```

```
if (0) {
    /* do nothing! */
} else {
    /* generate error */
    *(int *)0 = 0;
}
```

# CONSTRUCTORS

- SWITCH/CASE

```
CASE array[i]
  'a','b','c','d','e'
    ch := array[i]
  'f','g'
    ch := 'z'
ELSE
  ch := #00
```

```
switch (array[i]) {
  case 'a': case 'b': case 'c': case 'd':
  case 'e':
    ch = array[i];
    break;
  case 'f': case 'g':
    ch = 'z';
    break;
  default:
    ch = 0;
    break;
}
```

- LOOPS

```
WHILE (NOT end.of.file)
  ... process ...
```

```
while (!end_of_file) {
  ... process ...
}
```

```
SEQ i = 0 FOR count
  P (i)
```

```
for (i = 0; i < count; i++) {
  P (i);
}
```



# FUNCTIONS

```
IF
INT FUNCTION foo (VAL INT v)
  INT r:
  VALOF
    r := (v * 10)
  RESULT r
:
```

```
if (x == y) {
int foo (int v)
{
  int r;
  r = (v * 10);
  return r;
}
```

```
PROC foo (VAL INT x, REAL64 r)
  ... process ...
:
```

```
if (0) {
void foo (int x, double *r)
{
  ... process ...
}
```

# CHANNELS

```
#INCLUDE "course.module"
PROC sender (CHAN INT write!)
  INT seed:
  SEQ
    seed := 5000
  WHILE TRUE
    INT x:
    SEQ
      x, seed := random(256, seed)
      write ! x
:
```

```
PROC receiver (CHAN INT
  read?, CHAN BYTE out!)
  INT val:
  WHILE TRUE
    INT val:
    SEQ
      read ? val
      out.int(val, 0, out)
      out.string("*n", 0, out)
:
```

```
PROC mainisnotspecial
  (CHAN BYTE out!)
  CHAN INT comms:
  PAR
    sender(comms)
    receiver(comms, out)
:
```

- A **channel** is a pipe that allows one way communication between two processes
- sender writes random INT to a channel of... INT
- receiver reads INT from its read channel and writes their textual representation to the out channel
- There is no special name for the main procedure of a program, the last defined is executed first
- mainisnotspecial defines the channel that will pass data between sender and receiver, then runs both in parallel

# ALTING

```
-- either find the channel 'in'
available
-- and output to 'out' or output 0 to
'out'
```

```
PRI ALT
  in ? v
    out ! v
  SKIP
  out ! 0
```

-- this is polling which can be used to terminate a loop on an input from a channel

```
ALT
  count1 < 100 & c1 ? data
  SEQ
    count1 := count1 + 1
  merged ! data
  count2 < 100 & c2 ? data
  SEQ
    count2 := count2 + 1
  merged ! data
-- continued
```

```
status ? request
SEQ
  out ! count1
  out ! count2
```

-- read from c1 or c2 and pass to merged channel  
-- on request to status channel will output count to out

- One of the most useful features of the language, a *successful* **alternative**
- Allows a process to wait for multiple events, but only engage in one of them
- From an abstract view it's similar to POSIX select() function
- Can select either arbitrary (**ALT**) or based on priority (**PRI ALT**) from available events
- ... except all existing implementations treat ALT like a PRI ALT and favor the first listed alternatives



# FUN AND TRICKS

```
x := 3 + 5 * 2  
x := 3 + (5 * 2)
```

```
x, y := y, x
```

```
PROC testpass(INT x, VAL INT y)  
  SEQ  
    x := 3  
  :
```

- Occam has **NO** operator precedence! You have to use parantheses for any kind of complex expression
- It supports multiple assignments which are done in parallel. The middle box will swap the values of x and y with no explicit intermediary needed
- Parameters are passed by reference by default, specify VAL TYPE for passing by value
- In the right box, whatever is passed as x will stay 3 after the function call, while an assignment to y will be rejected at compile time
- Incidentally, a constant (declared with VAL xTYPE IS value) cannot be passed as a parameter except if it's declared VAL