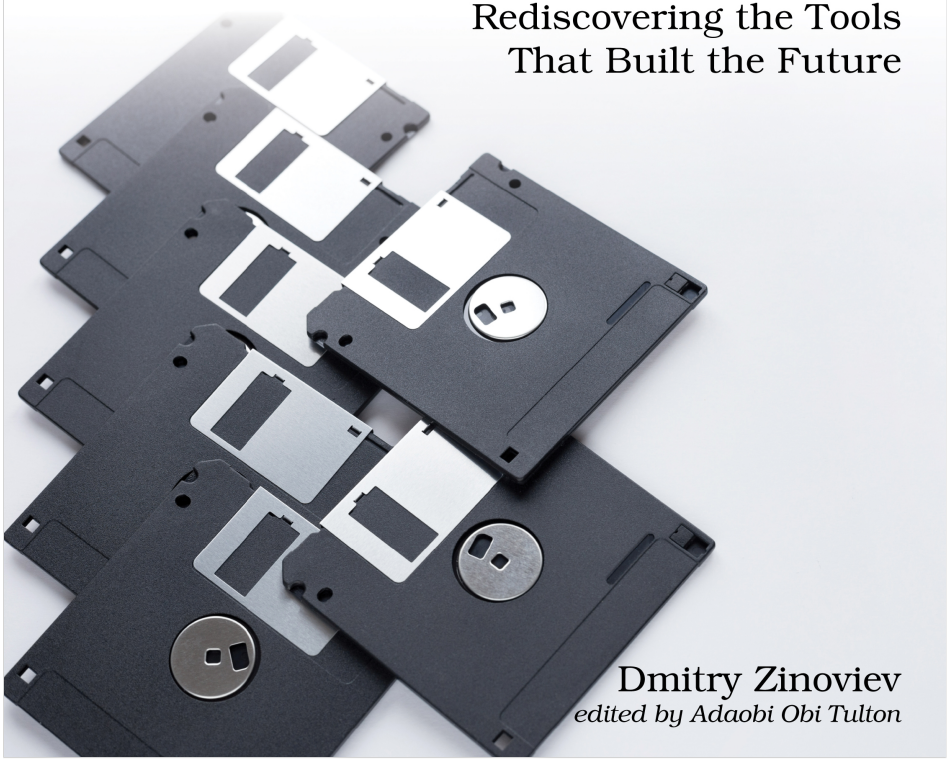
The Pragmatic Programmers

# Seven Obscure Languages in Seven Weeks

## Rediscovering the Tools That Built the Future



Dmitry Zinoviev
*edited by Adaobi Obi Tulton*

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

## Greeting in Occam and KRoC

In this chapter, you'll primarily use KRoC—the Kent Retargetable occam Compiler (note the official spelling!) from the University of Kent. The KRoC implements Occam-π, a modern version of Occam 2.5 infused with some elements of π-calculus.

### π-calculus

π-calculus is a theoretical model for concurrent computing, developed by Robin Milner in 1992 as an extension of his work on the calculus of communicating systems (CCS). π-calculus provides a framework for understanding and analyzing the behaviors of concurrent systems, where multiple processes operate simultaneously and interact. A key feature of π-calculus is its ability to describe dynamic topology, meaning that the connections between processes can change over time. This flexibility is achieved through the concept of "channel mobility," where the names of communication channels can be passed between processes, allowing for flexible and evolving communication structures. π-calculus is highly abstract and mathematical, but it has significantly influenced computer science and theoretical informatics, particularly in designing and analyzing distributed systems, communication protocols, and mobile computing.

A significant difference exists between the highly optimized, industrial-grade KRoC and the "classical" Occam of David May and Tony Hoare. Out of respect for its originators, this chapter will guide you through classical Occam, providing specific notes to address the disparities between the two dialects as necessary.

So here's the standard greeting in Occam, illustrated in two ways. (This code doesn't run with KRoC.) One immediate observation is that comments begin with two dashes and continue until the end of the line.

occam/hello-classical.occ

```
-- Display "Hello, world!"
CHAN([BYTE]BYTE) output:
output ! "Hello,*sworld!*n"
```

The peculiar markers *s and *n denote a space and a line break, respectively. You can substitute *s with a literal space, but *n cannot be replaced with the familiar \n.

Next, KRoC employs a preprocessor akin to CPP, the C/C++ preprocessor, except the directives are composed in uppercase letters. The directive #INCLUDE "course.module" includes the file course.module verbatim. The file contains the KRoC standard library, a feature absent in classical Occam.

occam/hello.occ

```
-- Display "Hello, world!"
```

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

Lastly, be mindful of the colon at the end of the second example. Usually placed on a separate line, it signifies the end of a definition (in this case, the end of the procedure).

You'll learn about other features used in these examples in the subsequent sections.

## Studying Variables and Data Types

In the spirit of Occam's razor and in the spirit of being essentially a glorified assembly language, Occam offers a limited set of numerical data types. Three primary types are mandatory, with some allowing further specifications:

- `INT`. Required, equal in size to the computer's machine word; `INT16`, `INT32`, `INT64` are optional extensions.
- `BYTE`. Required, comprising eight bits; also represents characters.
- `BOOL`. Required, can be `TRUE` or `FALSE`.
- `REAL32`, `REAL64`. Both are optional extensions.

**Simple Data Types**

Occam's palette of simple data types, especially the absence of required floating point data types, resembles one of Forth, explained on page ?. It almost feels like all assembly-like languages are the same, no matter how disguised.

Other simple data types denote Occam-specific objects, such as timers and channels. If you need an equivalent of a C structure of a C++ class, use declarations `RECORD` in combination with `DATA TYPE`. In the example that follows (highlighted), the former defines a new data structure with two `REAL32` fields, `x` and `y`. The latter incorporates the data structure into Occam's type system. Consequently, you can generate variables of this data type and access their fields through square bracket notation.

occam/testrecord.occ

```
PROC test.record ()
  DATA TYPE xy.point
    RECORD
      REAL32 x :
      REAL32 y :
  :
```

```
  -- In "classical" Occam:
  -- RECORD xy.point IS (REAL32, REAL32):
  -- Declare a 2D point
  xy.point center :

  SEQ
    -- Initialize the point coordinates
    center := [2.0 (REAL32), 3.0 (REAL32)]
    -- Translate the point by 1,-1
    center[x] := center[x] + 1.0
    center[y] := center[y] - 1.0
:
```

**Records**

Occam RECORDs loosely correspond to the C language structs or to the attributes (but not methods) of the C++ language class. The declaration DATA TYPE loosely corresponds to the C language operator typedef.

Occam doesn't support implicit type conversion. You must explicitly convert a value to a matching type using either of two methods: by prefacing the new type name before the value (without parentheses, known as pre-casting) or after the value (in parentheses, referred to as post-casting).

```
INT big.dog: -- An integer variable
BYTE am.potat: -- A byte-sized variable
SEQ -- Disregard!
  big.dog := (INT TRUE) + '0' (INT)
  am.potat := BYTE big.dog
```

Take note of a few things: a variable declaration ends with a colon; a block of code (SEQ, Sequential Processes, on page ?) is indented by two spaces; pre-casting can be used with constants and variables, whereas post-casting only applies to constants; if an operand of an arithmetic expression is a pre-cast, it must be parenthesized.

From the previous code example, you've been subtly introduced to identifiers. Occam identifiers are case-sensitive. They must begin with a letter and can include only letters, digits, and periods. All variables used in a process need to be declared in the specification section; their scope remains local to the specified process and all processes depending on it.

Processes cannot share variables, as two processes may be executed by separate transputers with no shared memory! The sole method to interchange values between processes is via channels.

A declaration doesn't initialize the variable. Hence, you must initialize each variable prior to its first usage.

In addition to variables, Occam also supports constants. The combination of keywords VAL (or CONST in some dialects) and IS defines a constant. If the constant's data type can be inferred, you don't have to declare it. In classical Occam, several variables of the same data type can be declared on the same line, separated by commas.

```
VAL INT year IS 365:
VAL leap.year IS year + 1:
VAL ESC IS 27 (BYTE):
```