BCPL

An implementation is available on thor. If you would like to try it, look at `README.thor` in directory:

`/group/clteach/mr/BCPL`

It is also available via my Home Page:

`http://www.cl.cam.ac.uk/ mr`

BCPL History

- It was designed and implemented in early 1967 when I was at at MIT.
- It is a very cut down version of CPL(1962-68) that is easy to implement.
- It was used extensively for systems research, particularly for the development of the Tripos Operating System and early developments of the Cambridge Ring.
- It is still used as a testbed for compiler research.

BCPL Summary

- It was designed as a systems programming language suitable for writing compilers and operating systems.
- It is typeless. All values are the same size (typically 32 bits).
- Values can be used to represent characters, integers, truth values, and pointers to data or code.
- Any operation is allowed on any value even though the result is sometimes meaningless.

BCPL Summary

- Pointers can be used to represent:
  - character strings,
  - vectors,
  - structures,
  - functions and
  - program labels.
- Memory was modelled as a collection of equal sized words, addressed by consecutive integers. This allows machine independent pointer arithmetic.

BCPL Example

```
GET "libhdr"

LET f(n) = n=0 -> 1, n*f(n-1)

LET start() = VALOF
{ FOR i = 1 TO 9 DO
    writef("f(%n) = %i5*n", i, f(i))
  RESULTIS 0
}
```

BCPL Functions

- Function arguments are called by value and stored in consecutive memory locations.
- Functions are variadic (allow a variable number of arguments, as in `writef`).
- Functions may be passed as arguments, returned as results, or assigned.
- Although function definitions may be nested, they may not contain variables referring to either arguments or locals of enclosing functions.
- Functions can be be correctly represented by just their entry addresses.
- Separate compilation was allowed using the global vector for inter module referencing.

BCPL Example

```
GET "libhdr"

GLOBAL { count:200; all:201  }

LET try(ld, row, rd) BE
  TEST row=all
  THEN count := count + 1
  ELSE { LET poss = all & ~(ld | row | rd)
         UNTIL poss=0 DO
         { LET p = poss & -poss
           poss := poss - p
           try(ld+p << 1, row+p, rd+p >> 1)
         }
       }

LET start() = VALOF
{ all := 1

  FOR i = 1 TO 12 DO
  { count := 0
    try(0, 0, 0)
    writef("%i2-queens solutions %i5*n",
            i,                   count)
    all := 2*all + 1
  }

  RESULTIS 0
}
```

BCPL Example

```
GET "libhdr"

LET coins(sum) = c(sum, (TABLE 50, 20, 10, 5, 2, 1))

AND c(sum, t) = sum<0 -> 0,
                sum=0 | !t=1 -> 1,
                c(sum, t+1) + c(sum-!t, t)

LET start() = VALOF
{  writes("Coins problem*n")

   t(0)
   t(1)
   t(2)
   t(5)
   t(21)
   t(100)
   t(200)
   RESULTIS 0
}

AND t(n) BE
  writef("Sum = %i3  number of ways = %i6*n",
                 n,                 coins(n))
```

BCPL Memory

Address	Contents
1000:	13
1001:	2000
1002:	0
...	
2000:	61
2001:	0
2002:	24

The expression: `1000!1!2` evaluates as follows:

```
     1000 ! 1 ! 2
=   (!(1000+1)) ! 2
=   (!(1001)) ! 2
=     2000 ! 2
=     !(2000+2)
=     !2002
=     24
```

BCPL Streams

```
LET next(s) = (s!0)(s)

LET out(s, x) = (s!1)(s, x)

LET close(s) = (s!2)(s)
```

The functions `s!0`, `s!1`, `s!2` are analogous to methods of the object `s` in an object oriented language.

The functions defined above were used by Christopher Strachey in the operating system OS/6 in 1970.

BCPL Objects

In an object oriented language, classes have fields and methods.

Often the fields hold different values for different instances of a class, but the methods are usually the same for each instance.

This suggests an implementation in which the methods are held in a vector (of method functions) which is referenced by each instance of the class.

The fields vector must be accessible to each method and so is normally passed as a (hidden) extra argument in a method call.

BCPL Objects

The following scheme works well for BCPL.

In the scope of the declaration:

```
MANIFEST { F=0; G=1; H=2 }
```

The expression

```
                G#(obj, 13, 63)
```

means:

```
          (obj!0!G)(obj, 13, 63)
           (fns!1) (obj, 13, 63)
           methodG (obj, 13, 63)


obj -> 0:  fns -------------> 0: methodF
       1: FieldA              1: methodG
       2: FieldB              2: methodH
            ...
       n: FieldZ
```

BCPL Coroutines

Coroutines are somewhat like threads (or processes).

- They simulate parallel execution.
- They share the same address space.
- But they only giveup control voluntarily (no pre-emption).

Each coroutine needs its own runtime stack.

When control passes from one BCPL coroutine to another, a value is passed.

This value looks like an argument in the coroutine than is suspending itself, and like a result in the coroutine that is resuming control.

BCPL Coroutines

The implementation is as follows:

```
  --------------------->------------------
 |                                        |
 ----------------------------------------------------------
|p  parent nxt  f  size  c  | .. stack .. suspended frame ..
 ----------------------------------------------------------
 |    |     |   |   |    |
 |    |     |   |   |    System work variable
 |    |     |   |   Size of the coroutine stack
 |    |     |   Coroutine main function
 |    |     Next coroutine
 |    The parent coroutine
 The save stack frame pointer
```

```
cptr := createco(f, size)   -- Create a coroutine.
deleteco(cptr)              -- Delete a coroutine.
res := callco(cptr, val)    -- Call another coroutine (cptr).
res := cowait(val)          -- Transfer control to the parent.
res := resumeco(cptr, val)  -- Suspend the current coroutine
                               and transfer to another.
```

The body of a coroutine behaves like:

```
   c := f(cowait(c))    // Repeat for ever
   REPEAT
```

Coroutines Example

```
GET "libhdr"

MANIFEST { upb = 4000 }

LET prime1() = VALOF

{ LET isprime = getvec(upb)
  FOR i = 2 TO upb DO
    isprime!i := TRUE // Until disproved
  FOR p = 2 TO upb IF isprime!p DO
  { LET i = p*p
    UNTIL i>upb DO
    { isprime!i := FALSE; i := i + p }
    cowait(p)
  }
  freevec(isprime)
  RESULTIS 0
}

AND prime2() = VALOF
{ FOR n = 2 TO upb DO
  { LET a, b = 2, 1
    FOR i = 1 TO n DO
    { LET c = (a+b) REM n; a := b; b := c }
    IF a=1 DO cowait(n)
  }
  RESULTIS 0
}
```

Coroutines Example

```
LET start() = VALOF
{ LET P1 = createco(prime1, 100)
  LET P2 = createco(prime2, 100)
  LET n1, n2, min = 0, 0, 0
  { IF n1=min DO n1 := callco(P1)
    IF n2=min DO n2 := callco(P2)
    min := n1<n2 -> n1, n2
    UNLESS n1=n2 DO
      writef(" %i4 from P%c*n", min, n1<n2 -> '1', '2')
  } REPEATUNTIL min=0
  deleteco(P1)
  deleteco(P2)
  RESULTIS 0
}
```