Arrays and Pointers

```
int a[20], *p;

p = & a[5];
*p = 0;  /* equivalent to a[5] = 0;  */

for (p=&a[0]; p<&a[20]; p++) *p = 0;
```

In older compilers the above is more efficient than:

```
int a[20], i;

for (i=0; i<20; i++) a[i]=0;
```

Pointer Arithmetic

An integer may be added to a pointer. It returns a pointer to the adjacent location of the appropriate size.

If `p` points to a 32 bit integer on a byte addressed machine, then `p+1` or `1+p` pints to the adjacent 32 bit integer. In machine code terms this typically involves adding 4 to the byte address held in `p`.

C	BCPL
`int a[20];`	`LET a = VEC 19`
`int *p;`	`LET p = ?`
`a[1]`	`a!1`
`*(a+1)`	`!(a+1)`
`*(1+a)`	`!(1+a)`
`1[a]`	`1!a`
`&a[1]`	`@ a!1`
`& *(a+1)`	`@ !(a+1)`
`a+1`	`a+1`

Strings

C character strings are zero terminated char vectors. BCPL strings are byte vectors with the length held in byte zero.

C	BCPL
`char s[5]="ABCD";`	`LET s = "ABCD"`
`s[0] == 'A'`	`s%0 = 4`
`s[1] == 'B'`	`s%1 = 'A'`
`s[2] == 'C'`	`s%2 = 'B'`
`s[3] == 'D'`	`s%3 = 'C'`
`s[4] == 0`	`s%4 = 'D'`

String Copying

C:

```
void copystring(char *from, char *to) {
  while (*from) *to++ = *from++;
  *to=0;
}
```

BCPL:

```
LET copystring(from, to) BE
  FOR i = 0 TO from%0 DO to%i := from%i
```

Space Allocation

C:

```
int *p = (int *)malloc(sizeof(int[100]));
if(p==0) .... /* no space  */
...
free(p);
```

BCPL:

```
LET p = getvec(99)
IF p=0 DO  ... // no space
...
freevec(p)
```

Structures

```
#include <stdio.h>

struct Inode {
  int val;
  struct Inode* next;
} *p, *q;

struct Inode *mk(int x, struct Inode *rest) {
  struct Inode *res =
   (struct Inode*)malloc(sizeof(struct Inode));
  res->val = x;
  res->next = rest;
  return res;
}

int main(int argc, char** argv) {
  p = mk(13, 0);
  q = mk(541, p);
  printf("%d\n", q->next->val);
  return 0;
}
```

Structures

```
#include <stdio.h>

typedef struct Inode {
  int val;
  struct Inode *next;
} *Ilist;

Ilist mk(int x, Ilist rest) {
  Ilist res = (Ilist)malloc(sizeof(struct Inode));
  res->val = x;
  res->next = rest;
  return res;
}

int main(int argc, char** argv) {
  Ilist p = mk(13, 0);
  Ilist q = mk(541, p);
  printf("%d\n", q->next->val);
  return 0;
}
```

Structures

```
#include <stdio.h>

typedef struct Inode Inode;

struct Inode {
  int val;
  Inode *next;
};

Inode *mk(int x, Inode *rest) {
  Inode *res = (Inode *)malloc(sizeof(Inode));
  res->val = x;
  res->next = rest;
  return res;
}

int main(int argc, char** argv) {
  Inode *p = mk(13, 0);
  Inode *q = mk(541, p);
  printf("%d\n", q->next->val);
  return 0;
}
```

Field Selection

The field selection operator is dor (`.`) as in Java. The arrow notation is a shortcut for indirection followed by field selection.

`e -> name` means `(* e) . name`.

Unions

Sometimes it is useful to have a variable whose value is one type at one moment and another type later. This is possible using unions.

Syntactically a union declaration is just like a structure declaration, only the `struct` is replaced by `union`.

It causes all the fields to be overlaid at the same position, rather than being given distinct locations.

Unions

```
union {
  float f;
  int i;
} var;

var.f = 23.5
printf("val = %f\n", var.f);

var.i = 5
printf("val = %d\n", var.i);
```

Example Problem

Construct a tree representation suitable for the expressions like `1+2*3`, and define a function `eval` to evaluate it.

The required structure is something like:

```
        ADD
       /   \
      /     \
   NUM       MUL
    1       /   \
           /     \
        NUM       NUM
         2         3
```

ML Solution

```
datatype E = NUM of int
           | POS of E
           | NEG of E
           | MUL of E * E
           | DIV of E * E
           | ADD of E * E
           | SUB of E * E;

fun eval(NUM k)      = k
  | eval(POS e)      = eval e
  | eval(NEG e)      = ~ (eval e)
  | eval(MUL(x,y)) = eval x  *  eval y
  | eval(DIV(x,y)) = eval x div eval y
  | eval(ADD(x,y)) = eval x  +  eval y
  | eval(SUB(x,y)) = eval x  -  eval y;

eval( ADD(NUM 1,
          MUL(NUM 2, NUM 3)));
```

BCPL Solution

```
GET "libhdr"

MANIFEST {
Op=0; Rand1=1; Rand2=2
NUM=1; POS=2; NEG=3; MUL=4; DIV=5; ADD=6; SUB=7
}

LET eval(e) = VALOF SWITCHON Op!e INTO
{ CASE NUM: RESULTIS Rand1!e
  CASE POS:  RESULTIS + eval(Rand1!e)
  CASE NEG:  RESULTIS - eval(Rand1!e)
  CASE MUL:  RESULTIS eval(Rand1!e) * eval(Rand2!e)
  CASE DIV:  RESULTIS eval(Rand1!e) / eval(Rand2!e)
  CASE ADD:  RESULTIS eval(Rand1!e) + eval(Rand2!e)
  CASE SUB:  RESULTIS eval(Rand1!e) - eval(Rand2!e)
}
```

BCPL Solution

```
LET mk1(op, a) = VALOF
{ LET r = getvec(1)
  Op!r, Rand1!r := op, a
  RESULTIS r
}

LET mk2(op, a, b) = VALOF
{ LET r = getvec(2)
  Op!r, Rand1!r, Rand2!r := op, a, b
  RESULTIS r
}

LET start() = VALOF
{ LET exp = mk2(ADD, mk1(NUM, 1),
                     mk2(MUL, mk1(NUM,2), mk1(NUM,3)))
  writef("eval(exp) = %n*n", eval(exp))
  RESULTIS 0
}
```

Union Example

```
#include <stdio.h>

enum Ops {NUM, POS, NEG, MUL, DIV, ADD, SUB};

typedef struct Num Num;
typedef struct Monad Monad;
typedef struct Dyad Dyad;

typedef union Expr {
  Num *N;
  Monad *M;
  Dyad  *D;
} Expr;

struct Num {
  int op;
  int val;
};

struct Monad {
  int op;
  Expr rand;
};

struct Dyad {
  int op;
  Expr left;
  Expr right;
};
```

Union Example

```
Expr mknum(int n) {
  Expr res;
  Num *e = (Num *) malloc(sizeof(Num));
  e->op = NUM;
  e->val = n;
  res.N = e;
  return res;
}

Expr mk1(int operator, Expr a) {
  Expr res;
  Monad *e = (Monad *) malloc(sizeof(Monad));
  e->op = operator;
  e->rand = a;
  res.M = e;
  return res;
}

Expr mk2(int operator, Expr a, Expr b) {
  Expr res;
  Dyad *e = (Dyad *) malloc(sizeof(Dyad));
  e->op = operator;
  e->left = a;
  e->right = b;
  res.D = e;
  return res;
}
```

Union Example

```
int eval(Expr e) {
  switch(e.M->op)
  { default:  return 0;

    case NUM: return e.N->val;
    case POS: return eval(e.M->rand);
    case NEG: return - eval(e.M->rand);
    case MUL: return eval(e.D->left) * eval(e.D->right);
    case DIV: return eval(e.D->left) / eval(e.D->right);
    case ADD: return eval(e.D->left) + eval(e.D->right);
    case SUB: return eval(e.D->left) - eval(e.D->right);
  }
  return 0;
}

int main() {
  Expr exp = mk2(ADD,
                 mknum(1),
                 mk2(MUL, mknum(2), mknum(3)));
  printf("eval(exp) => %d\n", eval(exp));
}
```