1 4.1

```
proc double(x: integer): integer;
begin
    return x + x
end;

proc apply3(proc f(x:integer): integer): integer;
begin
    return f(3)
end;

begin
    print_num(apply3(double));
    newline()
end.
```

`MODULE main 0 0`	
`IMPORT Lib 0`	
`ENDHDR`	

`PROC _double 0 0 0`	Declares _double as a procedure that uses 0 words for local variables
`LDLW 16`	Get x
`LDLW 16`	Get x
`PLUS`	Calculate x+x
`RETURNW`	Return x+x

`PROC _apply3 0 0 0`	Like the previous PROC
`CONST 3`	First argument of f
`LDLW 16`	Static link of f
`LDLW 20`	Address of f
`PCALLW 1`	Call f with one argument
`RETURNW`	Return f(3)
`PROC _main 0 0 0`	Main procedure
`GLOBAL _double`	First argument of apply3
`CONST 0`	Static link of previous procedure
`CONST 0`	Static link of procedure we now call
`GLOBAL _apply3`	Address of function we now call
`PCALLW 2`	2, as the static link passed counts as a parameter
`CONST 0`	A static link
`GLOBAL _print_num`	Address of _print_num
`PCALLW 1`	Call print_num
`CONST 0`	A static link
`GLOBAL _newline`	Address of _newline
`PCALLW 0`	Call _newline

2 4.2

2.1 (i)

```
proc flip⁰(x¹: integer): integer;
    proc flop¹(y²: integer): integer;
    begin
        if y² = 0 then return 1 else return flip⁰(y²−1) + x¹ end
    end;
begin
    if x¹ = 0 then return 1 else return 2 * flop¹(x¹−1) end
end;
```

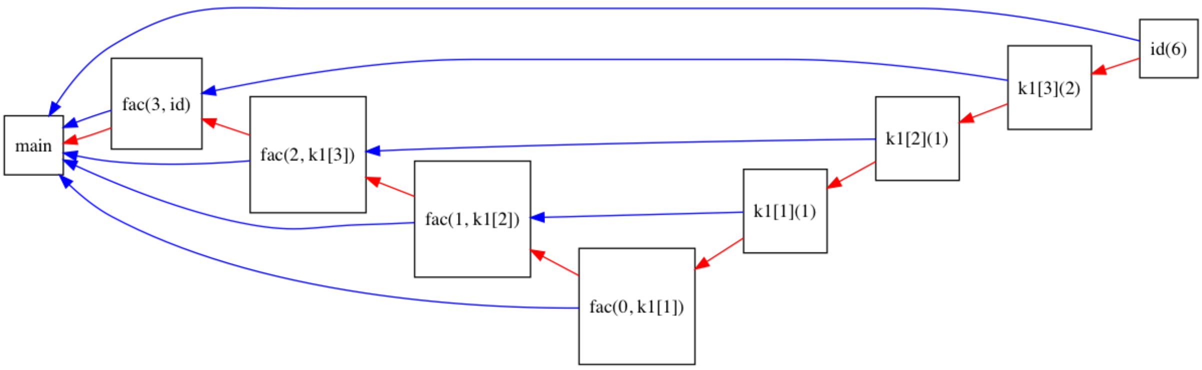
2.2 (ii)

Let `L1` be the position in the program right after the call of flip in main, `L2` be the position in the program right after the call of flip in flop, and `L3` be the position in the program right after the call of flop. Let `@x` be the position of a procedure x.

Frame	Adress	Contents	Meaning
main			
	b - 0	?	Arbitrary static link for main
	b - 4	@main	Procedure adress
	b - 8	?	Arbitrary return adress for main
	b - 12	?	Arbitrary dynamic link for main
flip			
	b - 16	4	Argument
	b - 20	b - 12	Static link
	b - 24	@flip	Procedure adress
	b - 28	L1	Return adress
	b - 32	b - 12	Dynamic link
flop			
	b - 36	3	Argument
	b - 40	b - 32	Static link
	b - 44	@flop	Procedure adress
	b - 48	L3	Return adress
	b - 52	b - 32	Dynamic link
flip			
	b - 56	2	Argument
	b - 60	b - 12	Static link
	b - 64	@flip	Procedure adress
	b - 68	L2	Return adress
	b - 72	b - 52	Dynamic link
flop			
	b - 76	1	Argument
	b - 80	b - 72	Static link
	b - 84	@flop	Procedure adress
	b - 88	L3	Return adress
	b - 92	b - 72	Dynamic link
flip			
	b - 96	0	Argument
	b - 100	b - 12	Static link
	b - 104	@flip	Procedure adress
	b - 108	L2	Return adress
	b - 112	b - 92	Dynamic link

3 4.3

Blue arrows represent static links, red ones represent dynamic links. `k1[i]` represents `k1` called in a context where `n` is `i`. Note that the return value (6) is propagated down the chain of red links, from id(6)



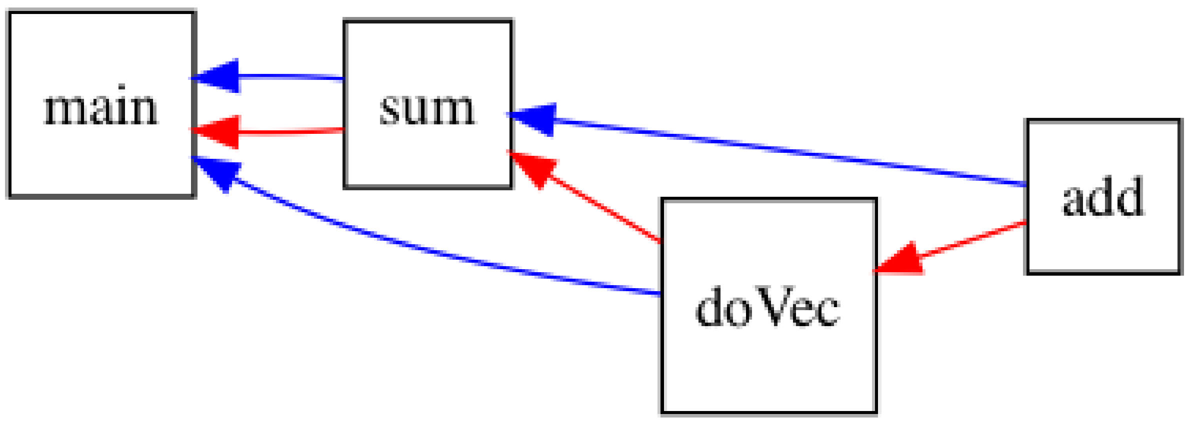
4 4.4

4.1 (a)

Let L1 be the position in code right after the call of f, L2 the position right after the call of doVec, and L3 the position right after our call of sum. Let @f be the address of procedure f.

Frame	Adress	Contents	Meaning
main			
	b - 0	?	Arbitrary static link for main
	b - 4	@main	Procedure adress
	b - 8	?	Arbitrary return adress for main
	b - 12	?	Arbitrary dynamic link for main
	b - 52	Values of a	Local array a
	b - 56	Value of i	Local integer i in main
sum			
	b - 60	b - 52	Argument v
	b - 64	b - 12	Static link
	b - 68	@sum	Procedure adress
	b - 72	L3	Return adress
	b - 76	b - 12	Dynamic link
	b - 80	Value of s	Local integer s
doVec			
	b - 84	b - 52	Argument v
	b - 88	@add	Address of argument add
	b - 92	b - 76	Static link of argument add
	b - 96	b - 12	Static link
	b - 100	@doVec	Procedure adress
	b - 104	L2	Return adress
	b - 108	b - 76	Dynamic link
	b - 112	Value of i	Local integer i in doVec
add			
	b - 116	v[i]	Argument
	b - 120	b - 76	Static link
	b - 124	@add	Procedure adress
	b - 128	L1	Return adress
	b - 132	b - 108	Dynamic link

Blue arrows represent static links, red ones represent dynamic links.



4.2 (b)

4.2.1 (i)

Note that at this point we have the stack from (a) up to `b - 112`, and that the frame pointer is currently `b - 108`.

`LDLW 24`	Get v
`LDLW -4`	Get i
`OFFSET`	Get address of v[i]
`LOADW`	Get v[i]
`LDLW 16`	Get static link of f
`LDLW 20`	Get address of f
`PCALLW 1`	Call f

4.3 (ii)

Note that at this point we have the entire stack from (a), and the frame pointer is currently `b - 132`.

`LDLW 12`	Follow static link once
`LDNW -4`	Get s
`LDLW 16`	Get x
`BINOP PLUS`	Calculate s + x
`LDLW 12`	Follow static link once
`STNW -4`	Set s to s + x

4.3.1 (iii)

Note that at this point we have the stack from (a) up to `b - 80`, and that the frame pointer is currently `b - 76`.

`LDLW 16`	Get v
`GLOBAL _add`	Get address of procedure add
`LOCAL 0`	Get static link for procedure add
`LOCAL 0`	Get static link
`PCALLW 3`	Call procedure with 3 arguments (closure counts as 2)

4.4 (c)

In this case, several things would need to change:

- We must arrange for `v` to be copied to the stack as an argument when calling `sum` or `doVec`.
- We must arrange for `v` to be properly aligned to word boundaries.
- When calling these functions, the number of words used for arguments must take into account the lengths of `v`, and its alignment.
- When using local variables in these functions, we must take into account the lengths and alignements of arrays when accessing parameters held deeper in the stack than them.
- When using elements in these vectors, we must use one less indirection.

One case when it might be faster to copy the entire array is when dealing with small arrays placed in distant spots in memory, that are used very many times. In such cases, the importance of locality of reference dominates, and the one time cost of copying the arrays is insignificant. If our language has only unaliased arrays, like PicoPascal or Fortran, we can introduce an optimisation that makes code that passes by reference just as fast:

- Identify often used, small arrays.
- Start simulating the array with registers once we reach an area when the array is often used.
- Upon exiting this area, write the values in the registers to the array.

5 4.4

5.1 (a)

Any implementation must take into account both size and alignement, as both are needed to determine how much memory to allocate for a particular variable, and are used to determine the size, layout and alignment of record and array types. One example of a pair data types that have the same size but different alignments are:

```
type rec1 = record a, b: integer
type rec2 = record a: longint
```

(where longint is a 64 bit integer). Both of these have a size of $64 = 32 + 32$, yet `rec1` is aligned to 4 bytes, whereas `rec2` is aligned to 8.

5.2 (b)

Assume `char` is 1 byte long, and aligned to 1 byte, and `integer` is 4 bytes long, and aligned to 4 bytes. `rec` would then be 8 bytes long:

- The first two bytes would be used to store `c1` and `c2`.
- The next two bytes have arbitrary values, and are used to align `n`.
- The next four bytes contain `n`

`rec` is also aligned to 4 bytes.

5.3 (c)

Stack layout when in f:

Frame	Adress	Contents	Meaning
g			
	b - 0	?	Static link
	b - 4	@g	Procedure adress
	b - 8	?	Return adress
	b - 12	?	Dynamic link
	b - 20	?	s
f			
	b - 24	b - 20	Argument r
	b - 28	b - 12	Static link
	b - 32	@g	Procedure adress
	b - 36	Position after f in g	Return adress
	b - 40	b - 12	Dynamic link

Instructions used:

```
LDLW n    := dereference fp+n and push it on the stack
LDNW n    := dereference (address on the top of stack)+n and replace the value on
             the top of the stack with it
STNW n    := pop the top of the stack and add n to get an address; pop the top of
             the stack again and assign this value to that address
CONST n   := push n to the top of the stack
ADD       := add the top 2 elements on the stack and replace them with the sum
OFFSET    := synonim for ADD for which one of the operands is an address
PCALL n   := call a procedure with n arguments
LOCAL n   := push (fp+n) to the top of the stack
```

Postfix code for `r.n := r.n + 1`:

`LDLW 16`	Get address of r
`LDNW 4`	Get value of r.n
`CONST 1`	
`ADD`	
`LDLW 16`	Get address of r
`STNW 4`	Store r.n+1 into r.n

Postfix code for `f(s)`:

`LOCAL 0`	
`OFFSET -8`	Get address of s
`LOCAL 0`	Get static link
`GLOBAL _f`	Get adress of f
`PCALL 1`	Call f

5.4 (d)

Postfix code for `r.n := r.n + 1`:

`LDLW 16`	Get address of r
`LDNW 4`	Get value of r.n
`CONST 1`	
`ADD`	
`LDLW 16`	Get address of r
`STNW 4`	Store r.n+1 into r.n

Postfix code for `f(s)`:

`LDLW -4`	Get address of s
`LOCAL 0`	Get static link
`GLOBAL _f`	Get adress of f
`PCALL 1`	Call f

5.5 (e)

If we suppose that our Java-like language happens to have Pascal-like syntax, but Java semantics, then the following code shows the difference:

```
proc swap(x, y : rec);
    var tmp: rec;
begin
    tmp := x;
    x := y;
    y := tmp;
end;

var r, w: rec;
begin
    r.x := 0; w.x := 1;
```

```
    swap(r, w);
    print(r.x)
end.
```

If we have pass-by-reference semantics, then the output should be 1. If we have pass-by-value semantics, then the output should be 0.