Mandatory 2 questions?

```
datatype direction = …

datatype grid = Size of int * int;

datatype exp =      Identifier of string  …

datatype boolexp = BiggerThan of exp * exp
                 | LessThan of exp * exp …

datatype stmt =   Move of direction * exp …

datatype robot = Robot of vardecl list * start * stmt list …
datatype program = Program of grid * robot;

fun evalExp(BiggerThan(e1, e2), decls) = if evalExp(e1, decls) > evalExp(e2, decls) then 1 else 0 …
            exception OutOfBounds;

fun interpret (Program(Size(x, y), Robot(decls, Start(xpos, ypos), Move(direction, exp) :: stmtlst))) = …
```

- Two functions needs to call each other
  - Define with and
- boolexp/aritexp not of type exp?
  - Wrap it in an exp
- If x and y
  - Keyword andalso

Problem 1 –Allocation

- Exercise 7.1 in Mitchell: Debug a program that (tries to) calculate the absolute value of an integer
  - This exercise is about C specifics!
  - The moral of this story is: know the semantics of your language before you try to program (things of importance) in it.

7.1 Activation Records for In-Line Blocks

You are helping a friend debug a C program. The debugger gdb, for example, lets you set breakpoints in the program, so the program stops at certain points. When the program stops at a breakpoint, you can examine the values of variables. If you want, you can compile the programs given in this problem and run them under a debugger yourself. However, you should be able to figure out the answers to the questions by thinking about how activation records work.

(a) Your friend comes to you with the following program, which is supposed to calculate the absolute value of a number given by the user:

```
1:  int main( )
2:  {
3:          int num, absVal;
4:
5:          printf("Absolute Value\n");
6:          printf("Please enter a number:");
7:          scanf("%d",&num);
8:
9:          if (num <= 0)
10:         {
11:                 int absVal = num;
12:         }
13:         else
14:         {
15:                 int absVal = -num;
16:         }
17:
18:     printf("The absolute value of %d is %d.\n \n",num,absVal);
19:
20:     return 0;
21: }
```

Why does the program not work?

- C does not require that variables are initialized (so the program compiles and runs)
- But: the inner absval declarations shadow the outer

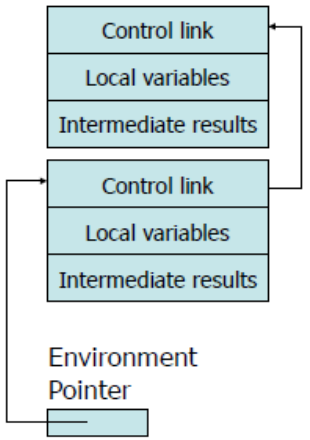
(b) Your explanation must not have been that good, because your friend does not believe you. Your friend brings you another program:

```
1:   int main( )
2:   {
3:           int num;
4:
5:           printf("Absolute Value\n");
6:           printf("Please enter a number:");
7:           scanf("%d",&num);
8:
9:           if (num >= 0)
10:          {
11:                  int absVal = num;
12:          }
13:          else
14:          {
15:                  int absVal = -num;
16:          }
17:
18:          {
19:          int absVal;
20:          printf("The absolute value of %d is %d.\n\n",num, absVal);
21:          }
22:  }
```

This program works. Explain why.

Might not work - depends on compiler's implementation!

- The behavior of such programs is left unspecified in C
- What *might* happen is that inner absVal will be written to the same location as the (uninitialized) outer absVal
  - The value *may* thus be correct at the end of the program



(c) Imagine that line 17 of the program in part (b) was split into three lines:

```
17a:    {
17b:    ...
17c:    }
```

Write a single line of code to replace the ... that would guarantee this program would NEVER be right. You may not declare functions or use the word absVal in this line of code.

For instance: { int arbitraryValue = -1; }

(d) Explain why the change you made in part (c) breaks the program in part (b).

This will allocate the arbitraryValue in the same place in memory as absVal.
-1 is never a valid absolute value.

Problem 2 – Static and Dynamic scope

- Static scope
  - global refers to declaration in closest enclosing block
- Dynamic scope
  - global refers to most recent activation record

7.8 Static and Dynamic Scope

Consider the following program fragment, written both in ML and in pseudo-C:

```
1  let x = 2 in                    int x = 2; {
2    let val fun f(y) = x + y in     int f (int y) { return x + y; } {
3      let val x = 7 in                int x = 7; {
4        x +                             x +
5            f(x)                            f(x);
6      end                             }
7    end                             }
8  end;                            }
```

The C version would be legal in a version of C with nested functions.

(a) Under static scoping, what is the value of x + f(x) in this code? During the execution of this code, the value of x is needed three different times (on lines 2, 4, and 5). For each line where x is used, state what numeric value is used when the value of x is requested and explain why these are the appropriate values under static scoping.

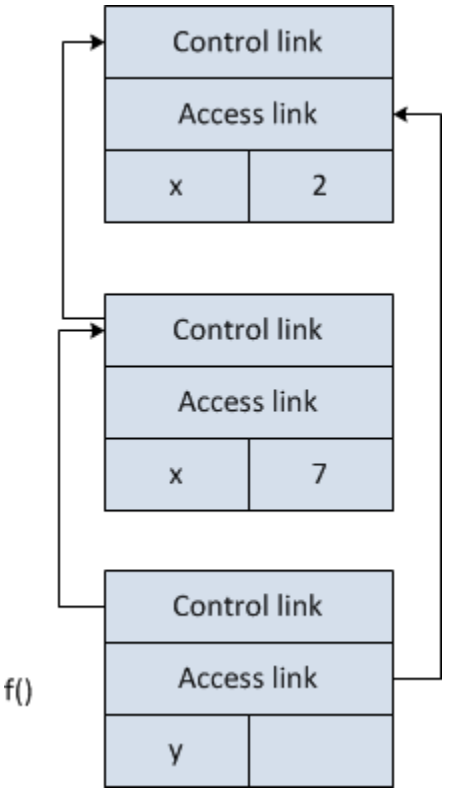
(b) Under dynamic scoping, what is the value of x + f(x) in this code? For each line in which x is used, state which value is used for x and explain why these are the appropriate values under dynamic scoping.

(a) x+f(x)= 7 + f(7) = 7 + (2+7) = 16

line	2	4	5
x	2	7	7

(b) x+f(x)= 7 + f(7) = 7 + (7 + 7) = 21

line	2	4	5
x	7	7	7



Problem 3 – Scope and Lifetime

- Scope
  - Region of program text where declaration is visible
- Lifetime
  - Period of time when location is allocated

```
{                           --block 1
  int i, j, k;              --1
  ...
  {                         --block 2
    int i, k;               --2
    ...
    {                       --block 3
      int j;                --3
      ...
    }                       --end block 3
    ...
    {                       --block 4
      int i, l;             --4
      ...
    }                       --end block 4
  }                         --end block 2
  ...
  {                         --block 5
   int a, b, c, d;          --5
   ...
  }                         --end block 5
}                           --end block 1
```

	Scope	Life-time
'i' in block 1	block 1 minus block 2 (and thereby also minus block 3 and 4, as these are defined within block 2), and block 5 as this does not define the name 'i'	block 1
'j' in block 1	block 1, block 2 (minus block 3), block 4 and 5	block 1
'k' in block 1	block 1 minus block 2 (and thereby also block 3 and 4), and block 5	block 1
'i' in block 2	block 2 including block 3, but not block 4	block 2
'k' in block 2	block 2 (including block 3 and 4)	block 2
'j' in block 3	block 3	block 3
'i' in block 4	block 4	block 4
'l' in block 4	block 4	block 4
'a' in block 5	block 5	block 5
'b' in block 5	block 5	block 5
'c' in block 5	block 5	block 5
'd' in block 5	block 5	block 5

Describe scope and life-time of the various variables wrt to the different blocks. Use the names of the blocks and the numbers of interesting lines.

Problem 4 – Static Scope

```
1  {
2      int i =1, j=2, k=3;
3      alpha()
4      {
5          int i=4, l=5;
6          i = k+l;
7                  -- **
8          beta();
9      };
10
11     beta()
12     {
13         int k =6;
14         i = j + k;
15                 -- *
16         alpha();
17     };
18
19     main()
20     {
21         beta();
22     }
23 }
```

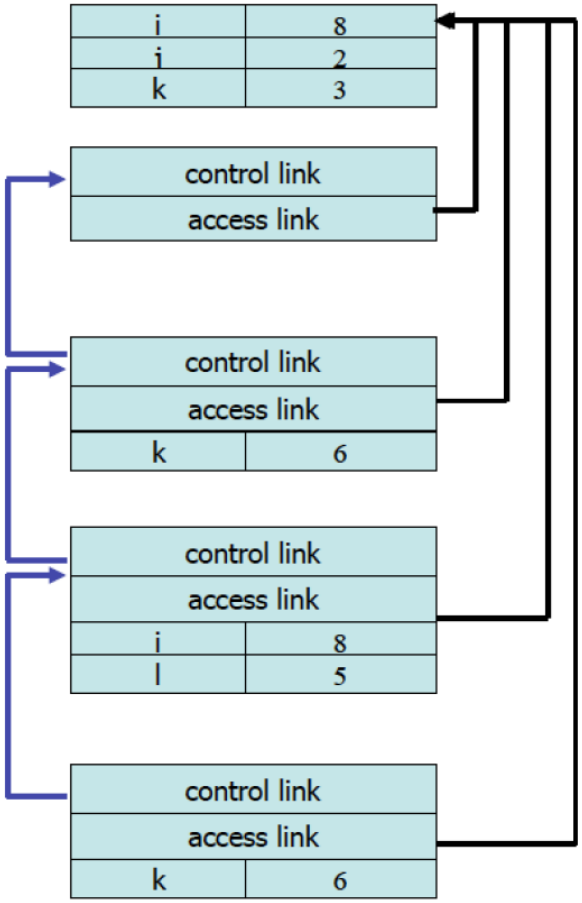
program

main

1) * beta

2) ** alpha

3) * beta



Execution starts by executing main().

Draw the run-time stack at three points of execution:
1) first time execution reaches the line marked with *,
2) when execution reaches the line marked with **, and
3) second time execution reaches the line marked with *

Show access and control links, and values of variables. Assume that the language is statically scoped.

As execution starts with execution of main, the control link of the main activation record is of no significance.

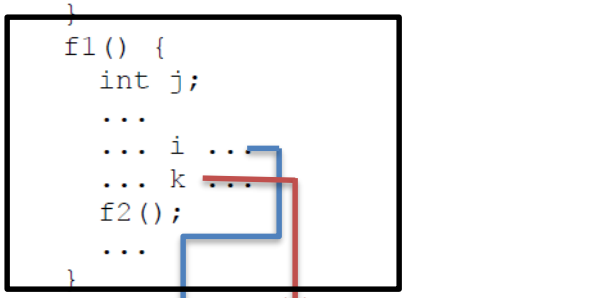
Problem 5 – Dynamic Scope

```
{
  f2() {
    int i;
    ...
    ... i ...
    ... j ...
    ...
  }
  f1() {
    int j;
    ...
    ... i ...
    ... k ...
    f2();
    ...
  }
  main() {
    int i, j, k;
    k = 0;
    i = 5;
    j = 7;
    f1();
    f2();
    ...
  }
}
```

Assume that this is written in a language with dynamic scoping. What will happen at the two calls f1() and f2(): to which declarations will the applications of i, j, k within f1 and f2 be bound?

Problem 5 – Dynamic Scope

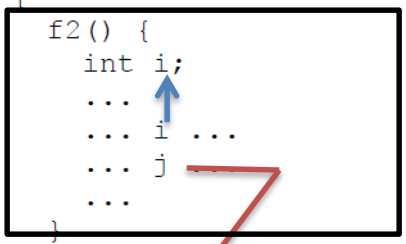
```
{
  f2() {
    int i;
    ...
    ... i ...
    ... j ...
    ...
  }
  f1() {
    int j;
    ...
    ... i ...
    ... k ...
    f2();
    ...
  }
  main() {
    int i, j, k;
    k = 0;
    i = 5;
    j = 7;
    f1();
    f2();
    ...
  }
}
```



Assume that this is written in a language with dynamic scoping. What will happen at the two calls f1() and f2(): to which declarations will the applications of i, j, k within f1 and f2 be bound?

f1()in main: 'i' is bound to 'i' in main, 'k' is bound to 'k' in main
f2()in f1: 'i' is bound to 'i' in f2, 'j' is bound to 'j' in f1
f2()in main: 'i' is bound to 'i' in f2, 'j' is bound to 'j' in main

Problem 5 – Dynamic Scope

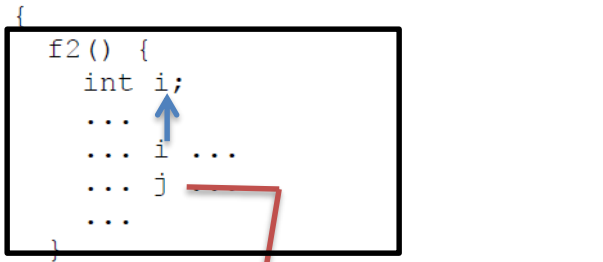


```
{
  f2() {
    int i;
    ...
    ... i ...
    ... j ...
    ...
  }
  f1() {
    int j;
    ...
    ... i ...
    ... k ...
    f2();
    ...
  }
  main() {
    int i, j, k;
    k = 0;
    i = 5;
    j = 7;
    f1();
    f2();
    ...
  }
}
```

Assume that this is written in a language with dynamic scoping. What will happen at the two calls f1() and f2(): to which declarations will the applications of i, j, k within f1 and f2 be bound?

f1()in main: 'i' is bound to 'i' in main, 'k' is bound to 'k' in main
f2()in f1: 'i' is bound to 'i' in f2, 'j' is bound to 'j' in f1
f2()in main: 'i' is bound to 'i' in f2, 'j' is bound to 'j' in main

Problem 5 – Dynamic Scope



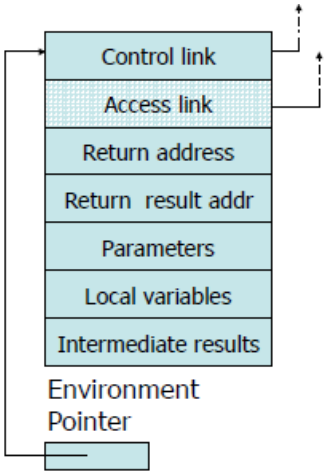
```
{
  f2() {
    int i;
    ...
    ... i ...
    ... j ...
    ...
  }
  f1() {
    int j;
    ...
    ... i ...
    ... k ...
    f2();
    ...
  }
  main() {
    int i, j, k;
    k = 0;
    i = 5;
    j = 7;
    f1();
    f2();
    ...
  }
}
```

Assume that this is written in a language with dynamic scoping. What will happen at the two calls f1() and f2(): to which declarations will the applications of i, j, k within f1 and f2 be bound?

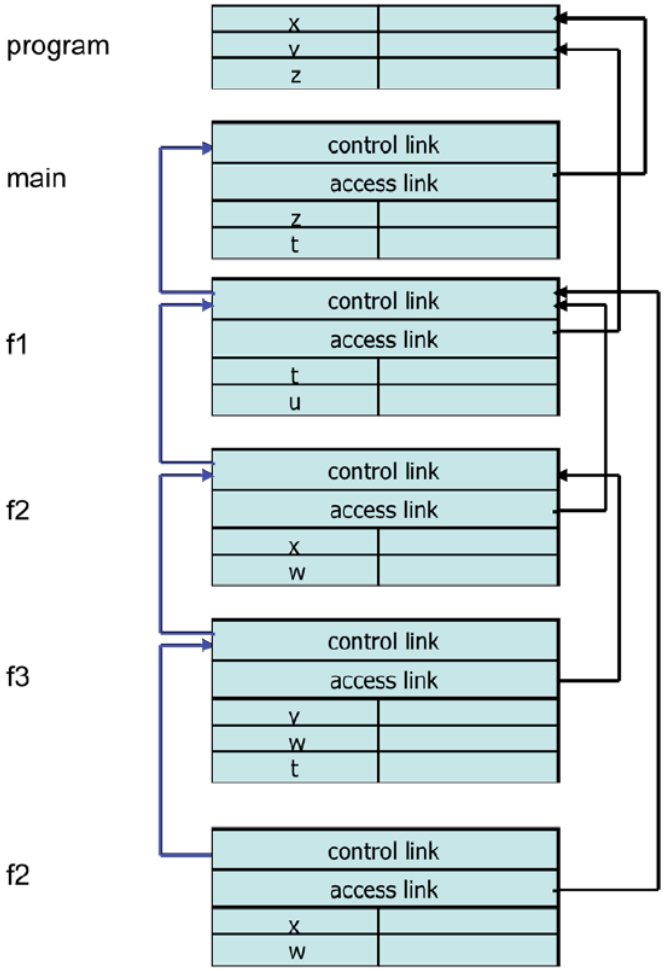
f1()in main: 'i' is bound to 'i' in main, 'k' is bound to 'k' in main
f2()in f1: 'i' is bound to 'i' in f2, 'j' is bound to 'j' in f1
f2()in main: 'i' is bound to 'i' in f2, 'j' is bound to 'j' in main

Problem 6 – Static Scope

```
This is an example in a language with static scoping.

{
  int x, y, z;
  f1(){
    int t, u;
    f2(){
      int x, w;
      f3(){
        int y, w, t;
        ... ;
        f2();
        ...
      };
      x = y + t + w + z;
      f3();
    }
    ...;
    f2();
    ...
  }
  main(){
    int z, t;
    ...
    f1();
    ...
  }
}

Show the run-time stack with both control and access links for the
following call sequence: main; f1(); f2(); f3(); f2().

Explain what happens with variable bindings when executing 'x = y + t +
w + z' in the latest call of f2. Look especially at y, t and z.
```



- Control link
  - Link to activation record of previous (calling) block
- Access link (static link)
  - Link to activation record of closest enclosing block in program text
- Difference
  - Control link depends on dynamic behavior of program
  - Access link depends on static form of program text



y is from "program", t is from "f1", w is from “f2”, z is from "program"

Problem 7 – Parameter passing

Consider the example below. Discuss call by reference and call by value-result for swap(a[i], a[j]). What happens if i=j?

```
swap(int x, int y) {
  x = x + y;
  y = x - y;
  x = x - y;
}
```

In call by value-result, the actual parameter supplied by the caller is copied into the callee's formal parameter; the function is run; and the (possibly modified) formal parameter is then copied back to the caller

– c2 wiki

As an example we assume a(i)=1 and a(j)=2. In the following table x and y are only used in the call-by-result case, while for call by reference the addresses are used. For i=j we just use i and thereby a(i)=1.

by reference, not(i=j)	**by value-result, not(i=j)**
a(i) = a(i) + a(j) = 1 + 2 = 3 a(j) = a(i) - a(j) = 3 - 2 = 1 a(i) = a(i) - a(j) = 3 - 1 = 2	x = a(i) = 1 y = a(j) = 2 x = x + y = 1 + 2 = 3 y = x - y = 3 - 2 = 1 x = x - y = 3 - 1 = 2 a(i) = x = 2 a(j) = y = 1
by reference, i=j	**by value-result, i=j**
a(i) = a(i) + a(i) = 1 + 1 = 2 a(j) = a(i) - a(i) = 2 - 2 = 0 a(i) = a(i) - a(i) = 0 - 0 = 0	x = a(i) = 1 y = a(i) = 1 x = x + y = 1 + 1 = 2 y = x - y = 2 - 1 = 1 x = x - y = 2 - 1 = 1 a(i) = x = 1

Problem 8 – Parameter passing

- Exercise 7.4 a) in Mitchell: Call by value/ call by reference

7.4 Parameter Passing

Consider the following procedure, written in an Algol/Pascal-like notation:

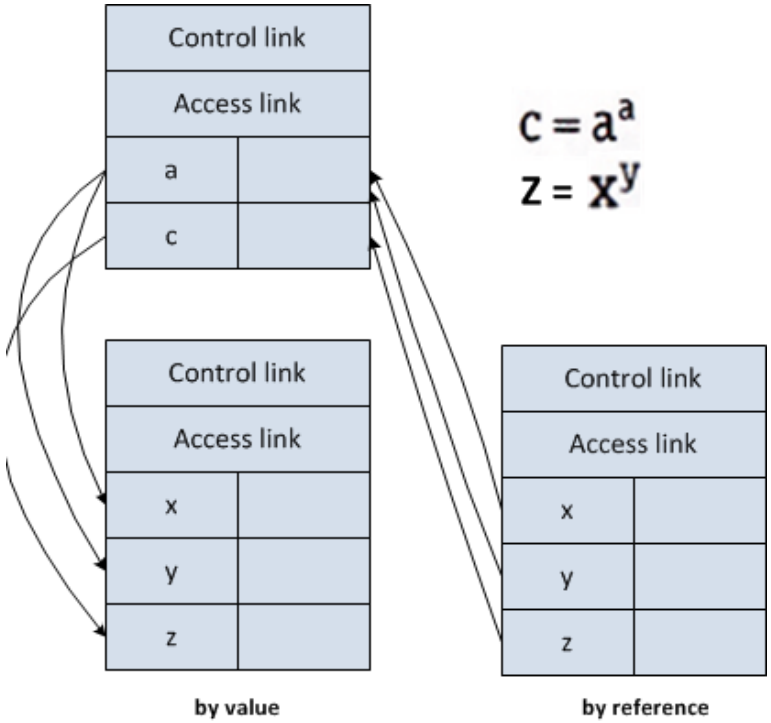
```
proc power(x, y, z : int)
begin
    z := 1
      while y > 0 do
          z := z*x
          y := y-1
      end
end
```

The code that makes up the body of power is intended to calculate x^y and place the result in z. However, depending on the actual parameters, power may not behave correctly for certain combinations of parameter-passing methods. For simplicity, we only consider call-by-value and call-by-reference.

(a) Assume that a and c are assignable integer variables with distinct L-values. Which parameter-passing methods make $c = a^a$ *after* a call power(a, a, c). You may assume that the R-values of a and c are nonnegative integers.

Problem 8 – Parameter passing

```
proc power(x, y, z : int)
begin
      z := 1
         while y > 0 do
              z := z*x
              y := y-1
         end
end
```



The following cases work:

- 1: power(x by value, y by value, z by ref)
- 2: power(x by value, y by ref, z by ref)
- 3: power(x by ref, y by value, z by ref)

Why?

- Obviously z needs to be by ref, in order to return a value from the function
- 1: local x and y; harmless
- 2: since x is by value, changing the original value of a through its reference in y is harmless, and thus this is OK
- 3: x is never assigned to in the function, so passing this by ref is OK, as long as y is by value
- Having all three by ref will clearly *not* work (decrementing y would also change x)

Problem 9 – Parameter passing

- Exercise 7.7 in Mitchell

7.7 Parameter-Passing Comparison

For the following Algol-like program, write the number printed by running the program under each of the listed parameter passing mechanisms. Pass-by-value-result, also sometimes called copy-in/copy-out, is explained in problem 6:

```
begin
    integer i;

    procedure pass ( x, y );
        integer x, y; // types of the formal parameters
        begin
            x := x + 1;
            y := x + 1;
            x := y;
            i := i + 1
        end

    i := 1;
    pass (i, i);
    print   i
end
```

(a) pass-by-value

(b) pass-by-reference

(c) pass-by-value-result

Problem 9 – Parameter passing

```
i := 1;
pass (i, i);
print   i
```

	by-value	by-reference	by-value-result
x:= x + 1;	x=2	i=2	x=2
y:= x + 1;	y=3	i=3	y=3
x:= y;	x=3	i=3	x=3
i:= i + 1	i=2 2 is printed	i=4 4 is printed	i=2 At exit i is assigned the value 3 from both x and y, so 3 is printed

For the by-value-result case it is not specified in which order the local x and y are assigned back to i, but in this special case this does not matter, since x and y have the same value (3).