# Compilers

Spring 2023

## *Register Allocation*

## *Sample Exercises and Solutions*

Prof. Pedro C. Diniz

Faculdade de Engenharia da Universidade do Porto
Departamento de Engenharia Informática
pedrodiniz@fe.up.pt

**Problem 1**

Consider the three-address code below:

```
 1:     i = 0;
 2:     a = p1;
 3:     b = i * 4;
 4:     c = a + b;
 5: L1: if (i > 100) goto L2
 6:     c = a + 1;
 7:     i = i + 1;
 8:     b = i * 4;
 9:     e = p1;
10:     if (c <= p1) goto L3
11:     c = e - b;
12:     a = e;
13:     goto L4
14: L3: d = p1;
15:     c = d + b;
16:     a = d;
17: L4: if (i <= 100) goto L1
18: L2: return
```

a) Determine the interference graph assuming a given variable is live at the end of a specific instruction if after the use in that instruction the value is still used elsewhere in the code. You should assume the variable `p1` is defined upon entry of the code as it corresponds to a parameter of the procedure and is used after the return instruction.
b) Determine the minimum number of registers needed (without spilling, of course).
c) Assuming you were short of one register, which register(s) would you spill and at which points in the program would you insert the spill code? Explain the rationale of your choice.
d) Now redo the register allocation using the top-down method in which variables used inside a loop are weighted more that variables outside the loop.

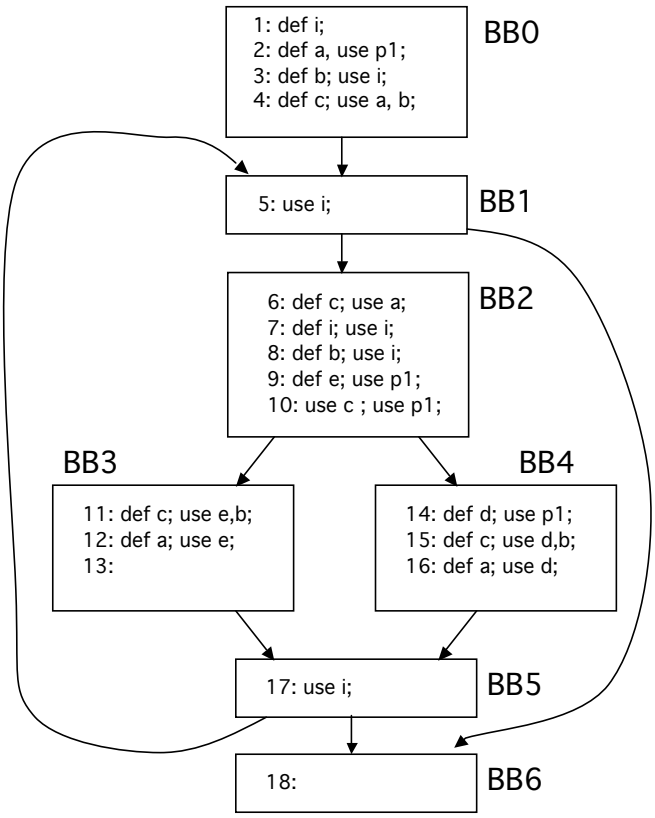
## Solution:

a) We have drawn the interference graph between the webs as shown below where we use the interference notion as dictated by the liveness of a given variable after the execution of a given instruction. In this context we say a variable is live in a specific instruction (denoted by the line number) if that variables participates in that instruction. If a variable is not used after a specific instruction it is considered dead after that instruction. We consider that the parameter p1 is defined outside and is still needed after the code of the procedure executes. It is thus always live.

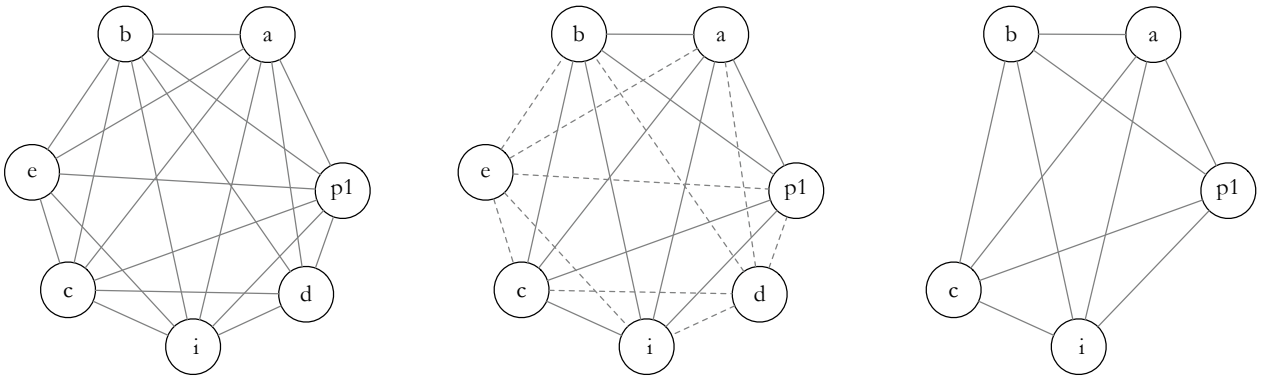
   As such as have the following live ranges (denoted by the line numbers) where each variable is live:

```
a:   { 2, 3, 4, 5, 6 } + { 12, 13,16, 17 }
b:   { 3, 4 } + { 8, 9, 10, 11 } + { 14, 15 }
c:   { 4 } + { 6, 7, 8, 9, 10 } + { 11 } + { 15 }
d:   {14, 15, 16 }
e:   { 9, 10, 11, 12 }
i:   { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17 }
p1:  { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17 }
```

   The figure below illustrates the Control-Flow-Graph (CFG) with the basic blocks corresponding to this code that helps you determine which value of which variables flow to which uses.




b) Using the interference webs for computing the interference graph one would get the graph below. To color this graph, one needs 6 colors. Notice that there is a clique of size 6 (a complete set of 6 nodes where all nodes in this set are connected to the other nodes in the same set). This means that we need at least 6 colors. The 6-clique in this example is composed by the nodes {a,b,c,e,i,p1}. The node {d} can be colored with the same color as node {e} as they have no conflict or interference.




c) Removing the nodes {e} and {d} which corresponds to the variable with the shortest live ranges would leave us with a clique of size 5 thus requiring 5 registers (see interference graph on the right).

d) Using the top-down allocation algorithm we must first compute the COST(V,B) functions for each variables V and basic block B. For the example above, we have the following matrix of cost values:

| | **BB0** | **BB1** | **BB2** | **BB3** | **BB4** | **BB5** | **BB6** |
|---|---|---|---|---|---|---|---|
| i | 2 | 1 | 3 | 0 | 0 | 1 | 0 |
| p1 | 1 | 0 | 2 | 0 | 1 | 0 | 0 |
| a | 2 | 0 | 1 | 1 | 1 | 0 | 0 |
| b | 2 | 0 | 1 | 1 | 1 | 0 | 0 |
| c | 1 | 0 | 2 | 1 | 1 | 0 | 0 |
| d | 0 | 0 | 0 | 0 | 3 | 0 | 0 |
| e | 0 | 0 | 1 | 2 | 0 | 0 | 0 |

In addition, we now have to compute the nesting depth of each basic block as shown in the table below.

| | **BB0** | **BB1** | **BB2** | **BB3** | **BB4** | **BB5** | **BB6** |
|---|---|---|---|---|---|---|---|
| **Depth** | 0 | 1 | 1 | 1 | 1 | 1 | 0 |

Given these two tables we now compute the total cost associated with a given variable in this algorithm as TotCost(V) = $\sum$ cost(V, B)*freq(B) where freq(B) of the basic block B is $10^{depth(B)}$.

| **V** | **TotCost(V)** | **TotCost(V)** |
|---|---|---|
| i | 2 + 1*10 + 3*10 + 1*10 | 52 |
| p1 | 1 + 2*10+ 1*10 | 31 |
| a | 2 + 1*10+ 1*10 + 1 *10 | 32 |
| b | 2 + 1*10+ 1*10 + 1 *10 | 32 |
| c | 1 + 2*10+ 1*10 + 1 *10 | 41 |
| d | 0 + 3*10 | 30 |
| e | 0+1*10 + 2*10 | 30 |

As such the variables are assigned register in the following priority: {i, c, a, b, p1, d, e}

**Problem 2**

Consider the following 3-address format representation of a computation. Here we have used the Stack Pointer (`$sp`) register to load local variables into temporary variables introduced in the intermediate code generation process. We have also used compile-time offsets to load the values of several local variables as they are located and fixed locations in the current procedure Activation Record (AR).

```
01:    t1 = k * 8
02:    t2 = $sp + offset_A
03:    t3 = t1 + t2
04:    t4 = *t3
05:    t5 = t4 * x
06:    t6 = $sp + offset_B
07:    t7 = t6 * 8
08:    t8 = *t7
09:    t9 = t5 + t8
10:    *t7 = t9
11:    k = k + 1
```

## Questions:

(a) Using the bottom-up register allocator described in class assign the various temporary variables and variables to actual registers. For the purpose of this section, assume you only have 4 physical registers. At each point when choosing to reuse a register indicate why do you pick each one.

(b) Use the graph-coloring based algorithm for doing register allocation instead. In this section we are to explore the use of interference webs for different definitions of interference as mentioned in class.

a. In the first web use the definition that two variables interfere if there is at least one instruction in which they participate. Derive the interference web between variables using this definition.
b. In the second definition there is no interference if the two webs either do not intersect at all or if they do intersect at a single instruction the web that ends at that instruction participates as the argument of the instruction and the web that begins at that instruction corresponds to the destination value of the instruction.
c. For both definitions determine the minimum number of required registers.
d. Using the graph-coloring heuristic described in class determine the number of required registers for each of the two interference definitions. Why do they differ, or why not?

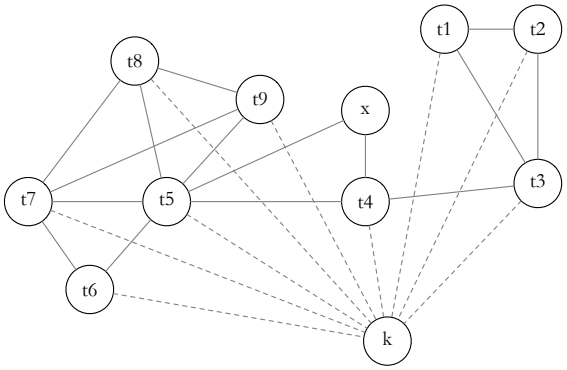
## Solution:

(a) Below is a description on how to translate the code in this question using 4 physical registers and using the local allocation algorithm described in class. For each variable we have loaded its value into a register via the indirection using the `$sp` and a specific offset in the AR. At specific points in the execution we indicate the current assignment of registers to program variables.

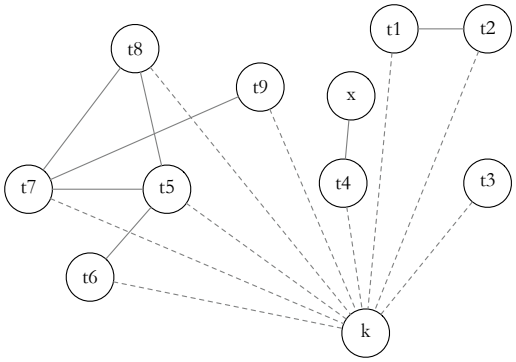
```
01:    r0 = $sp + offset_K //: r0 ← k; r1 ← empty;  r2 ← empty;  r3 ← empty;
02:    r1 = r0 * 8         //: r0 ← k; r1 ← t1;     r2 ← empty;  r3 ← empty;
03:    r2 = $sp + offset_A //: r0 ← k; r1 ← t1;     r2 ← t2;     r3 ← empty;
04:    r3 = r1 + r2        //: r0 ← k; r1 ← t1;     r2 ← t2;     r3 ← t3;
05:    r1 = (r3)           //: r0 ← k; r1 ← t4;     r2 ← t2;     r3 ← t3;
06:    r2 = $sp + offset_X //: r0 ← k; r1 ← t4;     r2 ← x;      r3 ← t3;
07:    r1 = r1 * r2        //: r0 ← k; r1 ← t5;     r2 ← x;      r3 ← t3;
08:    r2 = $sp + offset_B //: r0 ← k; r1 ← t5;     r2 ← t6;     r3 ←t7;
09:    r3 = r2 * 8         //: r0 ← k; r1 ← t5;     r2 ← t6;     r3 ← t7;
```

```
10:    r2 = *r3            //: r0 ← k; r1 ← t5; r2 ← t8; r3 ← t7;
11:    r1 = r2 + r1        //: r0 ← k; r1 ← t9; r2 ← t8; r3 ← t7;
12:    *r3 = r1            //: r0 ← k; r1 ← t9; r2 ← t8; r3 ← t7;
13:    r0 = r0 + 1         //: r0 ← k; r1 ← t9; r2 ← t8; r3 ← t7;
```

(b) We present the interference table instead as it is easier to read. On the left we have the interference graph for the simpler notion of interference whereas on the right we have for the notion of interference considering the use of the value in each register when the RHS is evaluated.



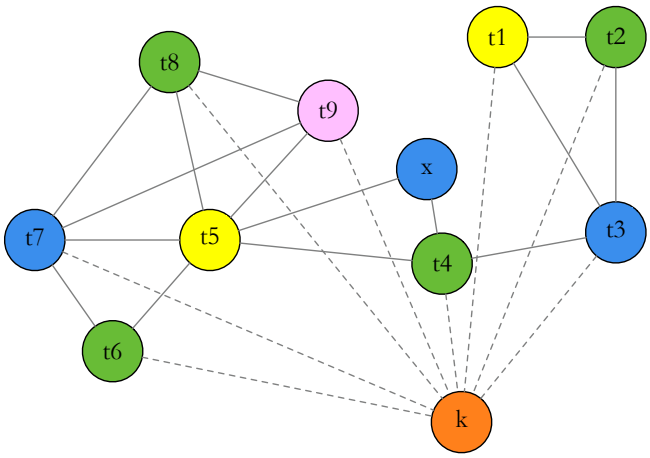

a. Interference Graph




b. Interference Graph

(c) In the first case we need 5 registers as there is a clique of size 5. In the second interference graph (middle) we have a clique of size 4, with t5, t7, t8, t9 and k and thus only need 4 registers.

(d) Using the graph-coloring heuristic algorithm (for the first notion of interference) and using 5 registers during the first phase we push the nodes for all nodes (in lexicographical order) onto the stack leaving only the nodes corresponding to x, t5 and k. We assign r0 to k and r1 to t5. Then popping each of the nodes off the stack we can assign r1 to t1, r2 to t2, t4, t6 and t8, r3 to x, t3 abd t7 and r4 to t9, yielding the colored graph below.




c. Colored Interference Graph

## Problem 3

Consider the code fragment depicted below.

```
01:       i = 0
02:       a = 1
03:       b = i * 4
04:       c = a + b
05: L1:   if i > n goto L2
06:       c = a + 1
07:       i = i + 1
08:       b = i * 4
09:       if c <= p goto L3
10:       e = 1
11:       c = e - b
12:       a = e
13:       goto L4
14: L3:   d = 2
15:       c = d + b
16:       a = d
17: L4:   goto L1
18: L2:   return
```

For this code determine the following:

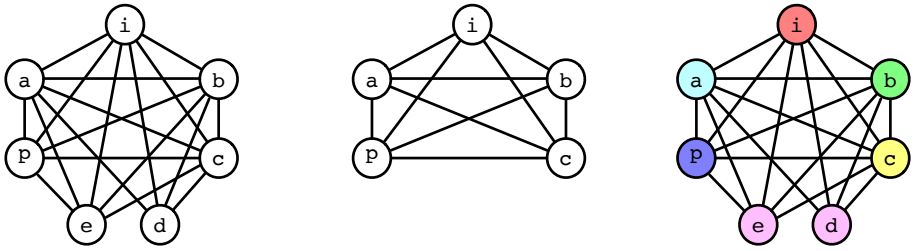
a) The live ranges for the variables `i`, `a`, `b`, `c`, `d`, `e`, `p`.
b) The interference graph using the simplest definition of interference where the webs include the live range in terms of the line numbers where an access to a variable (either a read or a write operation) occurs.
c) Determine the minimum number of registers needed (without spilling, of course) using the graph coloring algorithm described in class.
d) Assuming that you were short of one register, which register(s) would you spill and at which points in the program would you insert the spill code? Explain.

### Solution:

a) The basic observation is that one variable is live at a specific program point p if its value is still going to possibly be used in the future. The best way to figure this out is to begin by constructing the Control-Flow Graph (CFG) for this code. For instance, variable `i` is live at program point `11` as there is a possibility that the value defined for this variable at point `7` is going to be used along the path that traverses the basic block that contains the instructions `10` through 13 which then jumps to line `5` which reads this `i` variable to evaluate the test `(i>n)`. Tracing the definitions and uses for each variable we can arrive at the following live ranges where the number indication the source program line numbers:

```
i: {1–17}
p: {1-17}
a: {2-6,12,13,16,17}
b: {3,4},{8,9,10,11},{8,9,14,15}
c: {4},{6,7,8,9,11,15}
d: {14,15,16}
e: {10,11,12}
```

b) The variables {`i`, `a`, `p`} interfere will all the other variables as depicted in the interference graph is as shown below (left).




c) Using the algorithm studied in class for graph coloring for N = 6 given that that is the highest degree in the graph it is highly likely that for N = 6 we will be able to find such coloring of the interference graph. In this case only nodes `d` and `e` go into the stack as they have degree 5 less than 6. Then we push all the remaining nodes on the stack again, this time all of them have degree less than 6. We then pop the nodes one by one and assign colors to the nodes `i`, `a`, `b`, `c`, `p` thus coloring them with 5 colors since these 5 nodes form a 5-clique. Lastly, we pop the last two nodes `e`, `d`, off the stack and give them the same color.

d) If you had one less register say 5 registers, you could leave the node with fewer references since all the variables have long live ranges that interfere with any other variable. So, in this case I would pick `p`.

**Problem 4:** Consider the following three-address format representation of a computation using scalars and arrays A and B. Assume for the purposes of this exercise that no additional registers are needed to access (either writing or reading) the value of an array. As such the access to A[t1] requires no additional registers in addition to the one carrying the value of t1.

```
01:    t1 = i
02:    t2 = A[t1]
03:    t3 = i + 1
04:    t4 = A[t3]
05:    t5 = t2 + t4
06:    t6 = t5 / 2
07:    t7 = i
08:    B[t7] = t6
09:    i = i + 1
```

**Questions:**

(a) Using the bottom-up register allocator described in class assign the various temporary variables and variables to actual registers. For the purpose of this section, assume you have 3 physical registers and that the scalar `i` is already loaded into register r0. At each point when choosing to reuse a register indicate why do you pick each one. Also you should try to reuse registers in copy operations such as t1 = i in line 01 thus not consuming any more registers but simply propagating the use of r0 in this case.

(b) Use the graph-coloring based algorithm for doing register allocation instead. In this section we are to explore the use of interference webs for different definitions of interference as mentioned in class.

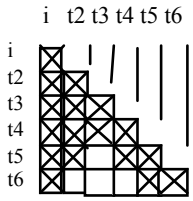
e. In the first web use the definition that two variables interfere if there is at least one instruction in which they participate. Derive the interference web between the variables in this code using this definition.
f. In the second definition there is no interference if the two webs either do not intersect at all or if they do intersect at a single instruction the web that ends at that instruction participates as the argument of the instruction and the web that begins at that instruction corresponds to the destination value of the instruction.
g. Using the graph-coloring heuristic described in class with N=4 and determine the coloring and hence register allocation for both interference definitions. Why do they differ, or why not?

## Solution:

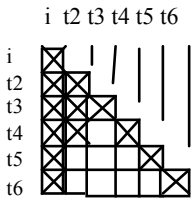
(a) Below is a description on how to translate the code in this question using 3 physical registers and using the local allocation algorithm described in class. At specific points in the execution we indicate the current assignment of registers to program variables. Note that we could have eliminated t1 as the first instruction makes a copy of the value of i into t1 (and the same is true for t7 and i). For this reason in the assignment below the pair t1,i and t7,i appear as a single tuple.

```
01:   r0 = r0          // r0 ← t1, i;    r1 ← Ø;     r2 ← Ø;
02:   r1 = A[r0]       // r0 ← t1, i;    r1 ← t2;    r2 ← Ø;
03:   r2 = r0 + 1      // r0 ← t1, i;    r1 ← t2;    r2 ← t3;
04:   r2 = A[r2]       // r0 ← t1, i;    r1 ← t2;    r2 ← t4;
05:   r2 = r1 + r0     // r0 ← t1, i;    r1 ← t2;    r2 ← t5;
06:   r0 = r2 / 2      // r0 ← t6;       r1 ← t2;    r2 ← t5;
07:   r1 = i           // r0 ← t6;       r1 ← t7,i;  r2 ← t5;
08:   B[r1] = r0       // r0 ← t6;       r1 ← t7,i;  r2 ← t5;
09:   r1 = r1 + 1      // r0 ← t6;       r1 ← t7,i;  r2 ← t5;
```

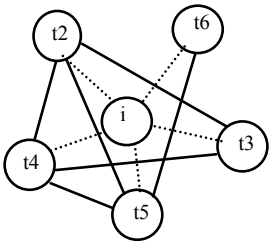
(e) We present the interference table instead as it is easier to read. On the left we have the interference table for the simpler notion of interference whereas on the right we have for the notion of interference considering the use of the value in each register when the RHS is evaluated. In this example we have ignored the variables t1 and t7 as they can be eliminated by copy-propagation - an optimization that removes redundant variables by observing when the identities {a = b} hold through the execution of a code section. This greatly simplifies the register allocation.



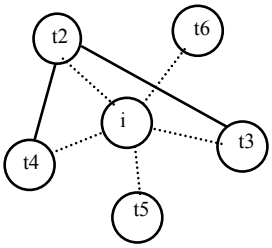

a. Interference Table




b. Interference Table



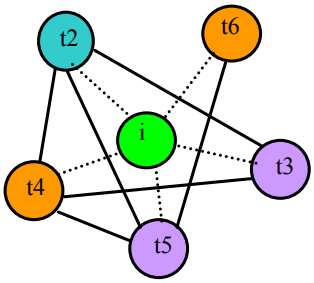

a. Interference Graph

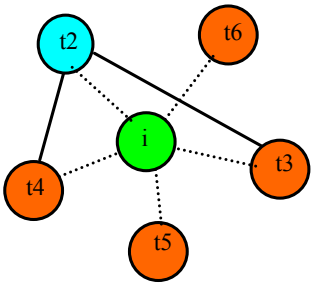



b. Interference Graph

(f) In the first case we need 4 registers as the maximum number of conflicts in any row of the table is 4 (for t2) whereas in the second case we only need 3 registers. There are several cliques of size 3 in the second case and at least one clique of size 4 in the first case (involving i).

(g) Using the graph-coloring heuristic algorithm (for the first notion of interference) and using 4 registers or N=4 in the algorithm, we fist push onto the stack the nodes in the order {t6, t5, t4, t3, t2, i}. We then pop the node corresponding to i and assign it the color c0. Next, we color t2 with color c1, color t3 with color c2 and t4 with color c3. When we pop t5 we can color it with the same color as t3, i.e., c2 and when we pop t6 we can use either c3 or c1. The figures below depict these colorings on the left for the first interference graph and on the right for the second interference graph, which uses only 3 color. Notice that for the graph on the left there is a clique of size 4 so using four colors (registers) is the best we can do and is in fact optimal.



**Problem 5:** Consider the following three-address format representation of a computation using scalars and arrays A and B and a procedure `F(int a, int b)`. Assume for the purposes of this exercise that no additional registers are needed to access (either writing or reading) the value of an array. As such the access to A[t1] requires no additional registers in addition to the one carrying the value of t1.

```
01:   t1 = i
02:   t2 = A[t1]
03:   t3 = i + 1
04:   t4 = A[t3]
05:   t5 = t2 + t4
06:   t6 = t5 / 2
07:   t7 = i
08:   B[t7] = t6
09:   putparam i
10:   putparam 1
11:   call F,2
12:   t8 = i
13:   t9 = t8 + 2
```

a) Apply copy propagation and dead code elimination to the basic block immediately preceding the function invocation.
b) Use the bottom-up register allocation algorithm for 3 registers under the assumption that variable `i` is live outside this basic block and after the call to F. In this section you can simply ignore the instructions on lines `09` through `11`. In this part of the code load the value of the scalar variables using the offset of the scalar from the Frame-Pointer (`$fp`) register. So, you need to make some changes to the intermediated code.
c) Passing the arguments to the procedure F could be done via registers. This would come at the cost of additional registers unless the values required by a given procedure would already reside in registers. For this particular case explain which `putparam` instruction could be eliminated using arguments for the procedure that are already in registers.

**Solution:**

a) The statements on lines 01, 07 and 12 are dead once we propagate the corresponding assignments to the instructions on lines 02, 08 and 13.

```
01:   t1 = i
02:   t2 = A[i]
03:   t3 = i + 1
04:   t4 = A[t3]
05:   t5 = t2 + t4
06:   t6 = t5 / 2
07:   t7 = i
08:   B[i] = t6
09:   putparam i
10:   putparam 1
11:   call F,2
12:   t8 = i
13:   t9 = t8 + 2
```

b) The code below illustrates the resulting code under the assumption that the variable i is initially stored at a negative offset 16 of the Frame-Pointer ($fp) register.

```
01:    r0 = $fp – 16
02:    r0 = *r0            // loads the value of variable i into r0
02:    r1 = A[r0]          // r0 ← i,t1 r1 ← t2,   r2 ← Ø
03:    r2 = r0 + 1         // r0 ← i,t1 r1 ← t2,   r2 ← t3
04:    r2 = A[r2]          // r0 ← i,t1 r1 ← t2,   r2 ← t4
05:    r2 = r1 + r2        // r0 ← i,t1 r1 ← t2,   r2 ← t5
06:    r0 = r2 / 2         // r0 ← t6   r1 ← t2,   r2 ← t5
07:    r1 = $fp - 16       // r0 ← t6   r1 ← i,t7  r2 ← t5
08:    r1 = *r1            // r0 ← t6   r1 ← i,t7  r2 ← t5
08:    B[r1] = r0          // r0 ← t6   r1 ← i,t7  r2 ← t5
09:    putparam r1
10:    putparam 1
11:    call F,2
12:    r0 = r1             // r0 ← t8   r1 ← i,t7   r2 ← t5
13:    r2 = r1 + 2         // r0 ← t8   r1 ← i,t7   r2 ← t9
```

c) For this particular case the argument value i is already in register r1, so we could eliminate the first putparam instruction provided now that when doing the register allocation for F we would start with the assumption that the first parameter is already in r1.

**Problem 6:**
The code shown on the left is for a procedure in C where `p` and `n` are arguments to the function.

```
1 func:  getparam p
2        getparam n
3        i = 0
4        a = 1
5        b = i * 4
6        c = a + b
7 L1:    if i > n goto L2
8        c = a + 1
9        i = i + 1
10       b = i * 4
11       if c <= p goto L3
12       e = 1
13       c = e - b
14       a = e
15       goto L4
16 L3:   d = 2
17       c = d + b
18       a = d
19 L4:   goto L1
20 L2:   return
```

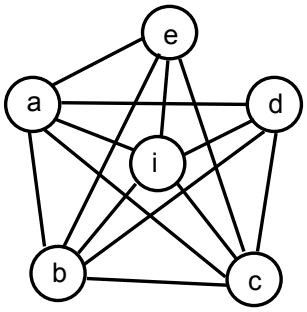
For this code determine the following:

a) The live ranges for the variables i, a, b, c, d, e.
b) The interference graph using the simplest definition of interference where the webs include the live range in terms of the line numbers where an access to a variable (either a read or a write operation) occurs. Assume that the parameters `n` and `p` are not in registers.
c) Determine which variables should be put in registers using the top-down register allocation algorithm described in class given that you only had 4 registers. Explain.

**Solution:**

a) Variables i is live at all instructions in this procedures and thus its live ranges are {3,…, 20}. Variable `a` has two live ranges r1 = {4,5,6,7,8,9,10,12,12,13,14} and r2 = {18, 19, 7, 8} in this second range he value defined in line 18 is used in instruction 8 via the goto statement on line 19. For variable `b` we have three live ranges, r = {5,6}, r2 = {10,11,12,13} and r3 = {10,11,16,17}. For variable `c` we have 4 live ranges r1 = {6}, r2 = {8,9,10,11}, r3 = {13} and r4 = {17}. For variable `d` we have a single live range r1 = {16,17,18} and for `e` we have r1 = {12,13,14}.

b) The interference graph using the simplest definition of interference where the webs include the live range in terms of the line numbers where an access to a variable (either a read or a write operation) occurs. Assume that the parameters `n` and `p` are not in registers. Using the information from a) the interference graph is as shown below. The only two variables that do not interfere (using this definition) with each other are variables `d` and `e`.




c) Determine which variables should be put in registers using the top-down register allocation algorithm described in class given that you only had 5 registers. Explain. We need to determine the loop structure of the code so that we can weigh the occurrences of each of the variables. By inspection of the code we can see that there is a loop that contains all statements from lines 7 through 19. Inside the loop there is an if-then-else construct so all the accesses inside this loop with be weighted with weight 10 as there is a single nested loop here.

As such the table of weights for this code is as follows:

| | **Variable** | | | | | |
|---|---|---|---|---|---|---|
| | `i` | `a` | `b` | `C` | `d` | `e` |
| Breakdown | 4 x 10 + 2 | 30 x 10 + 2 | 3 x 10 + 2 | 4 x 10 + 1 | 3 x 10 + 0 | 3 x 10 + 0 |
| Weight | 42 | 32 | 32 | 41 | 30 | 30 |

We can see that preferentially we should put in registers the variables `i` and `c` followed by the variables `a` and `b` and finally the variables `d` and `e`. Note that because variables `d` and `e` do not interference (see b) above they need only one register and thus I would be able to allocate all of them to the 5 registers provided.

## Problem 7. Control-Flow Analysis and Register Allocation

Consider the three-address code below for a procedure with input/output arguments p0 and p1 and using several temporary variables, named `t0` through `t5`.

```
01:      t5 = 0
02:      t0 = p1
03:      t2 = 0
04:      t1 = p2
05:      if (t1 > 0) goto L1
06:      t5 = t0
07:      t0 = 1
08:      t2 = 0
09: L1:  t1 = t0
10:      t4 = t0
11:      if (t4 > t5) goto L2
12:      t4 = t1 + 1
13:      t2 = t2 + 1
14:      goto L1
15: L2:  t0 = p1
16:      t3 = t0
17:      t1 = t2 + t3
18:      ret t1
```

## Questions:

For this code determine the following:

a. Basic blocks and the corresponding control-flow graph (CFG) indicating for each basic block the corresponding line numbers of the code above.

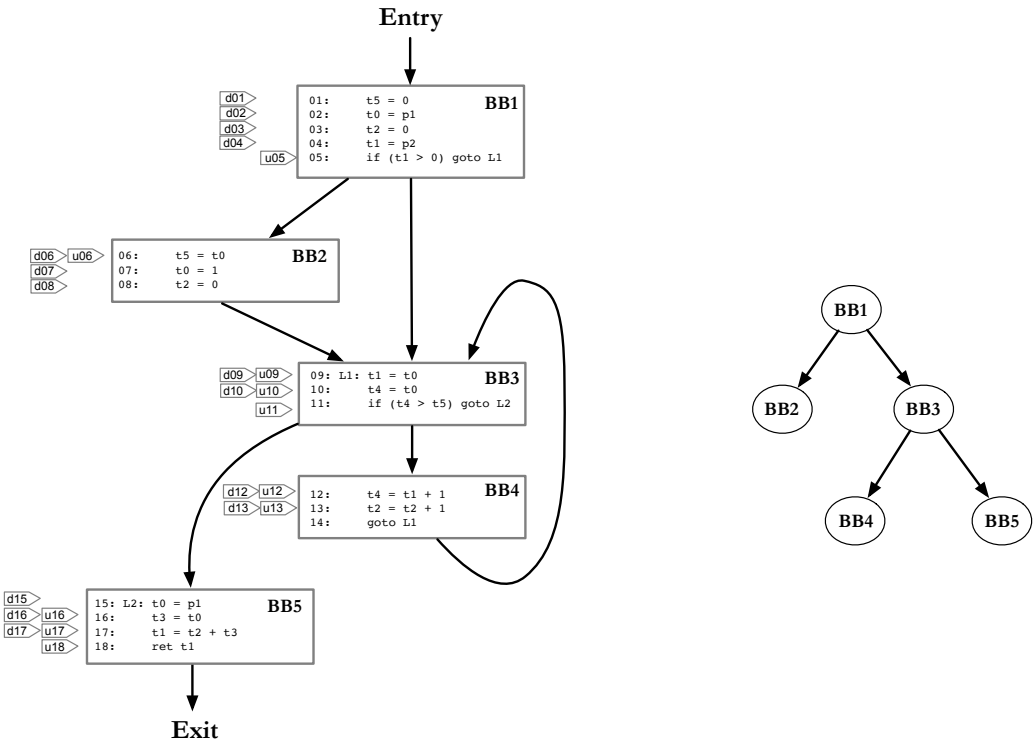
b. Dominator tree and the natural loops in this code (if any) along with the corresponding back edge(s).

c. Determine the live ranges for the variables `t0`, `t1`, `t2`, `t3`, `t4` and `t5`, the corresponding webs and interference graph, for the refined notion of interference discussed in class. Assume that you do not need registers for the parameters p0 and p1 and assume that on exit of the last basic block (the one ending with the return instruction) `t2` is live but the remainder temporaries are dead on exit of the procedure. In this analysis you should ignore the parameter variables `p0` and `p1`.

d. Can you color the resulting interference graphs with 4 colors? Why not? Suggest a modification to the code, possibly using source-level code transformations, that allows for the coloring with 4 colors. Present a coloring assignment for 4 colors.

**Solution:**

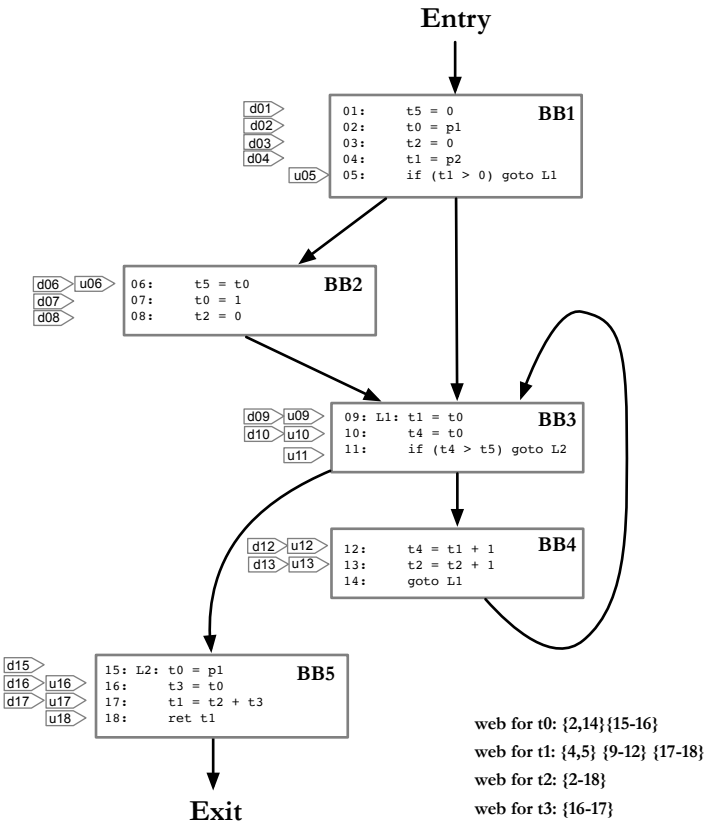
a) and b) The CFG and the corresponding dominator tree are as shown below. The only back edge, whose "head" dominates its "tail" is edge (3,4) and the corresponding natural loop is composed of the basic blocks 3 and 4.



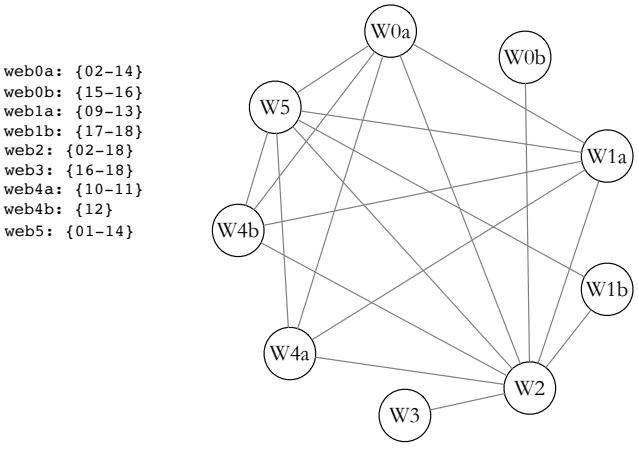

Control-Flow Graph Dominator Tree

This figure also indicates the definition points and use points so that DU-chains, and thus webs, can more easily derived. For instance, variable `t5` is defined at instruction `01`, where we have a designation of `d01`. The particular value is used in instruction at line 11, thus making use of the use `u11`.

c) The live ranges and the corresponding webs are shown below.




The figure below depicts the interference graph for the temporaries `t0` through `t0` using the refined notion of interference as discussed in class. As it can be seen there is a clique of size 5 (W0a, W1a, W5, W4b and W2), so clearly it cannot be colored using 4 colors.

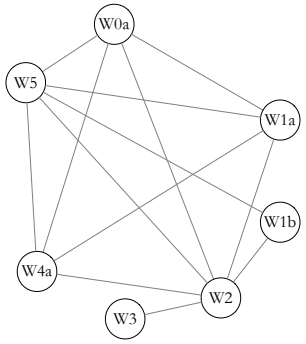


Using symbolic forward-substitution, and dead code elimination, we can eliminate two webs (left graph) which is still not colorable using 4 colors, since it contains a 5-clique. On the picture on the right, we have the interference graph that reflects a symbolic forward-substitution of `t4` and thus results in a 4-colorable graph.

```
web0a: {02-14}

web1a: {09-13}
web1b: {17-18}
web2: {02-18}
web3: {16-18}
web4a: {10-11}

web5: {01-14}
```




```
web0a: {02-14}

web1a: {09-13}
web1b: {17-18}
web2: {02-18}
web3: {16-18}

web5: {01-14}
```