Possible Answers for Midterm Exam 2 (may 24, 2017)

Compilers course

Masters in Informatics and Computing Engineering (MIEIC), 3rd Year

João M. P. Cardoso

Dep. de Engenharia Informática
Faculdade de Engenharia (FEUP), Universidade do Porto,
Porto, Portugal
Email:jmpc@acm.org

1

Example:

```
for(int i=0; i<N; i++) {
    A[i] = A[i]*c+d;
}
```

Data types for variables and storage Information:

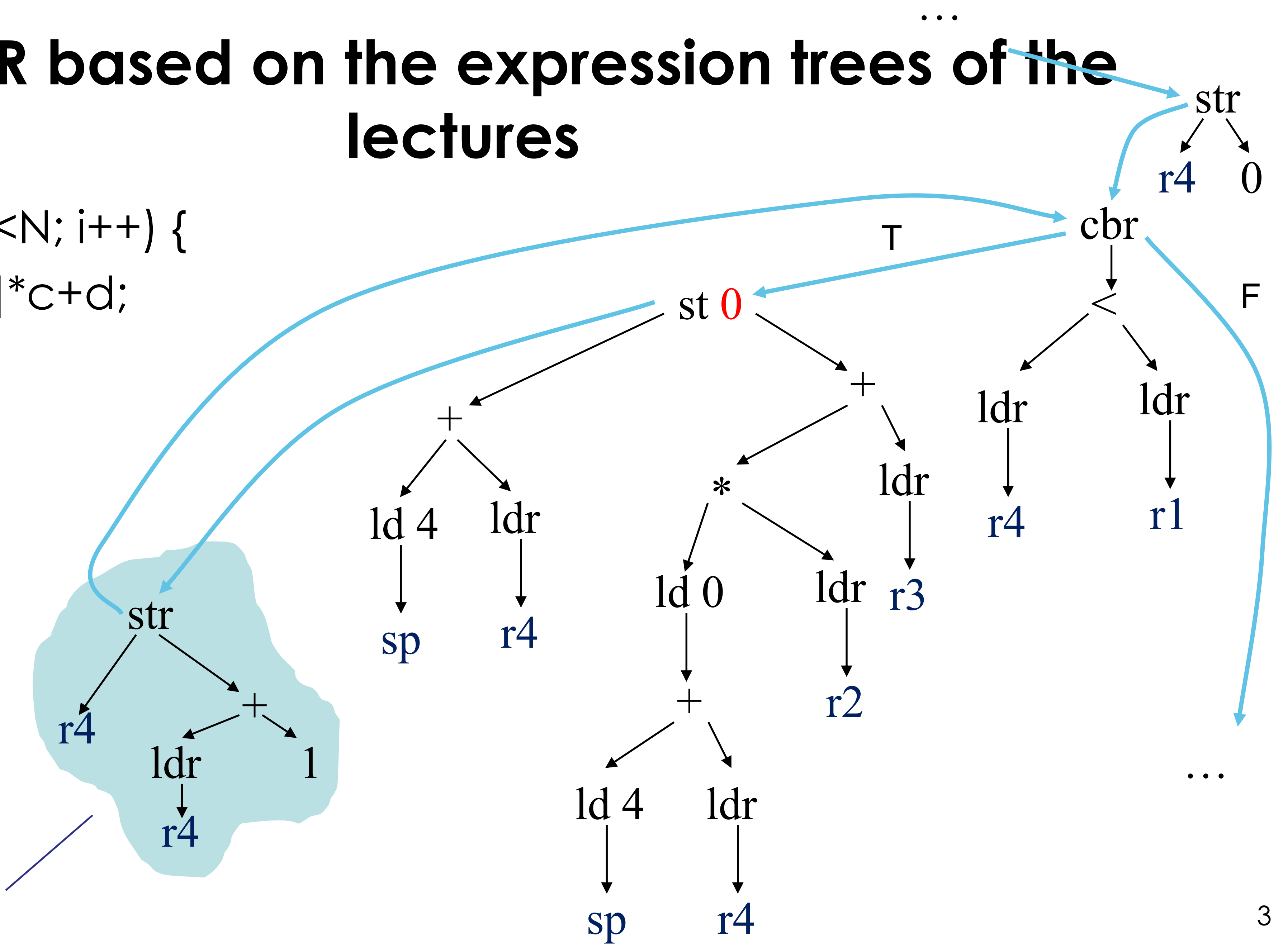
Variable	Type	Storage
A	int8 A[N] (array of 8-bit integers)	Base address of the array stored in the stack, position SP + 4
c	int c (32-bit scalar variable)	Variable stored in register r2
d	int d (32-bit scalar variable)	Variable stored in register r3
i	int i (32-bit scalar variable)	Variable stored in register r4
N	int N (32-bit scalar variable)	Variable stored in register r1

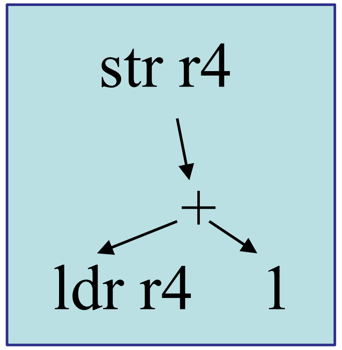
1a) LLIR based on the expression trees of the lectures

```
for(int i=0; i<N; i++) {
    A[i] = A[i]*c+d;
}
```

Notes: nodes "ldr" and "str" have been introduced to identify loads (reads) and stores (writes) from/to registers.

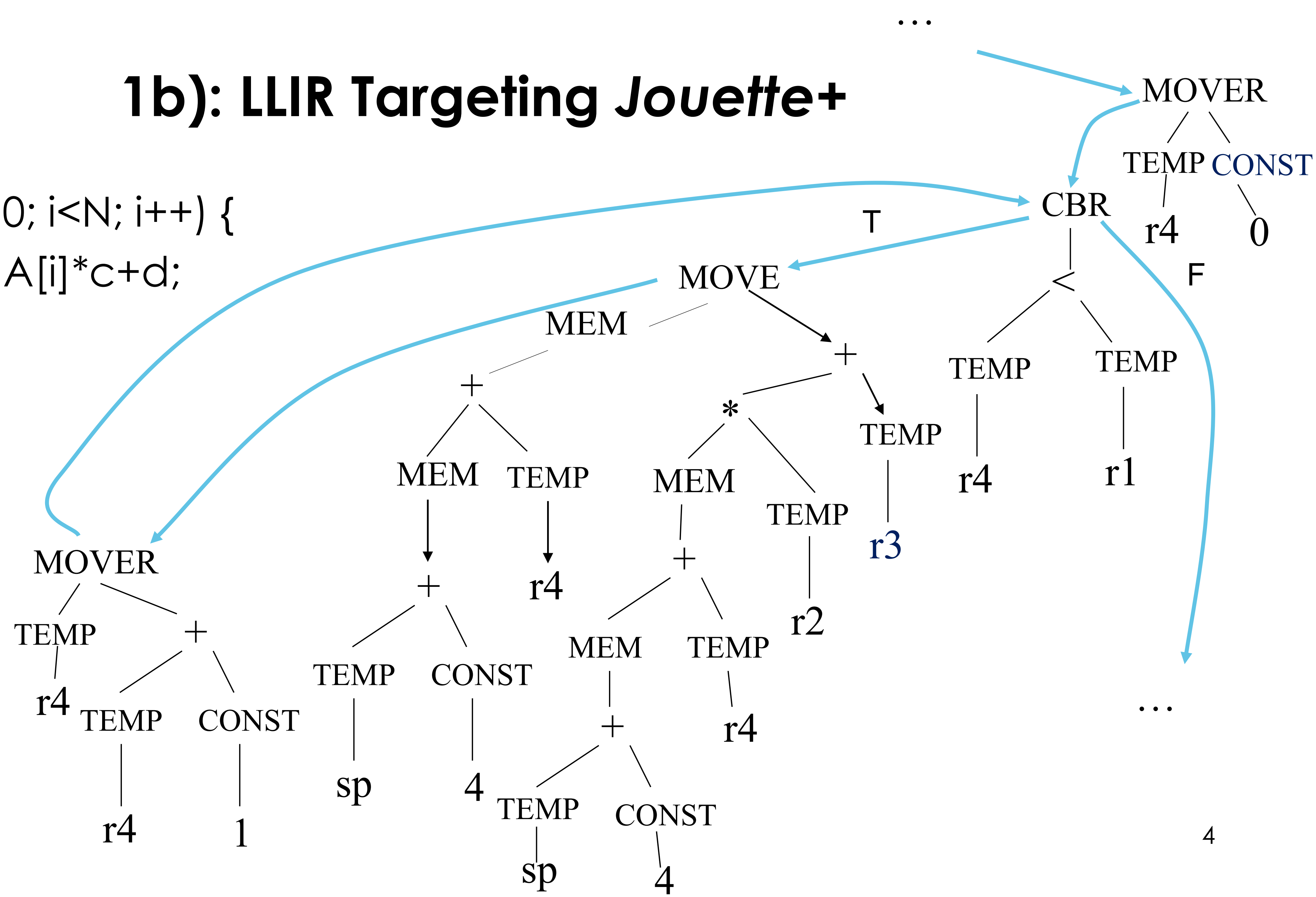
Other possible represent ation





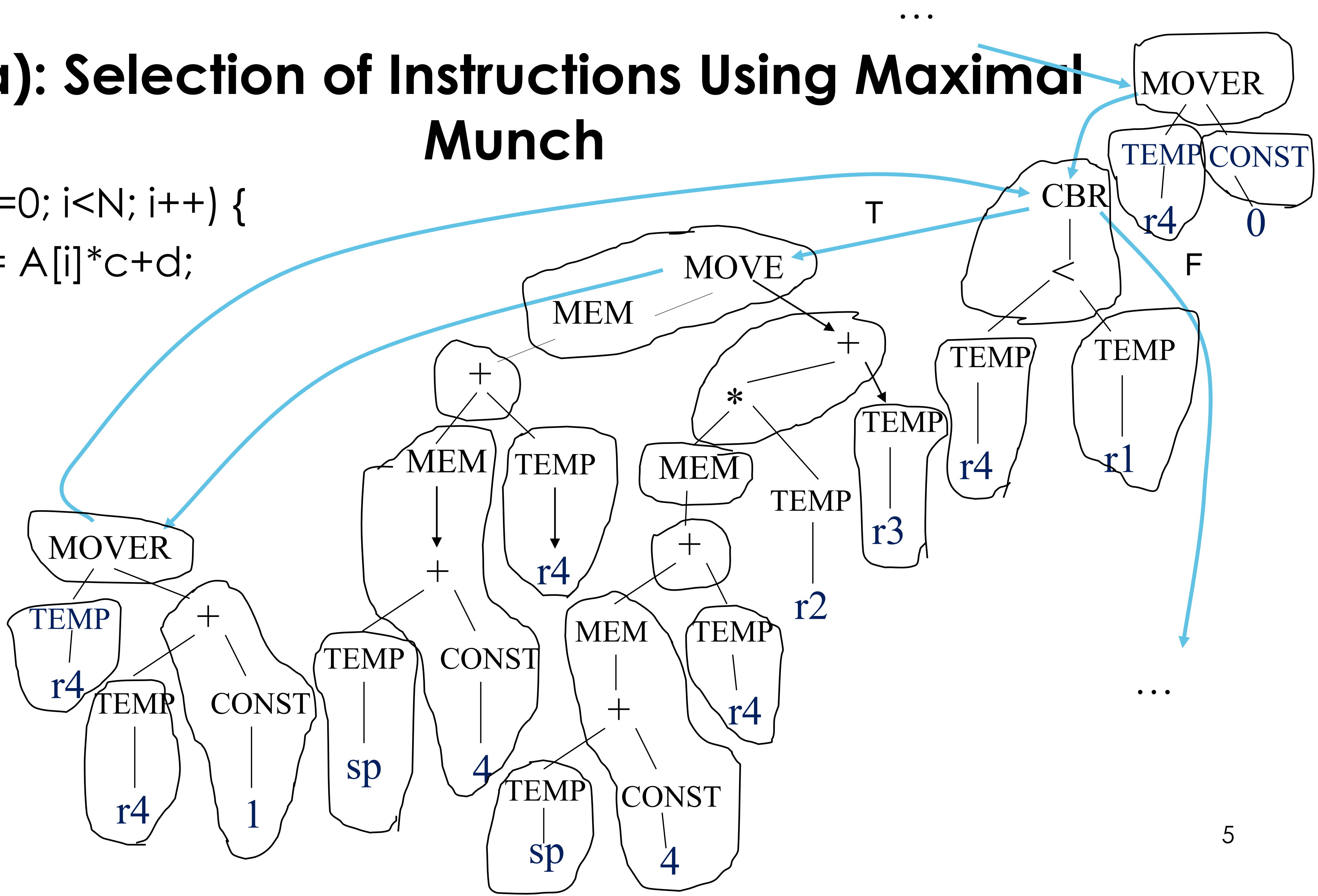
1b): LLIR Targeting *Jouette+*

```
for(int i=0; i<N; i++) {
    A[i] = A[i]*c+d;
}
```



2a): Selection of Instructions Using Maximal Munch

```
for(int i=0; i<N; i++) {
   A[i] = A[i]*c+d;
}
```



2b) Instructions Generated according to Maximal Munch

```
ADDI:                        rx ← r0 + 0
MOVER:                       r4 ← rx
BLT:      Loop:              blt r4, r1, LoopBody
GOTO:                        goto End
LOAD:     LoopBody:          rf ← M[SP + 4]
ADD:                         re ← rf + r4
LOAD:                        rd ← M[re + 0]
MADD:                        rb ← rd * r2 + r3
LOAD:                        rc ← M[SP + 4]
ADD:                         ra ← rc + r4
STORE:                       M[ra+0] ← rb
ADDI:                        rc ← r4 + 1
MOVER:                       r4 ← rc
GOTO:                        goto Loop
          End:
```

Notes:
rx, rf, re, rd, rc, ra represent virtual registers and another possibility is to identify them as r5, r6,.., r10

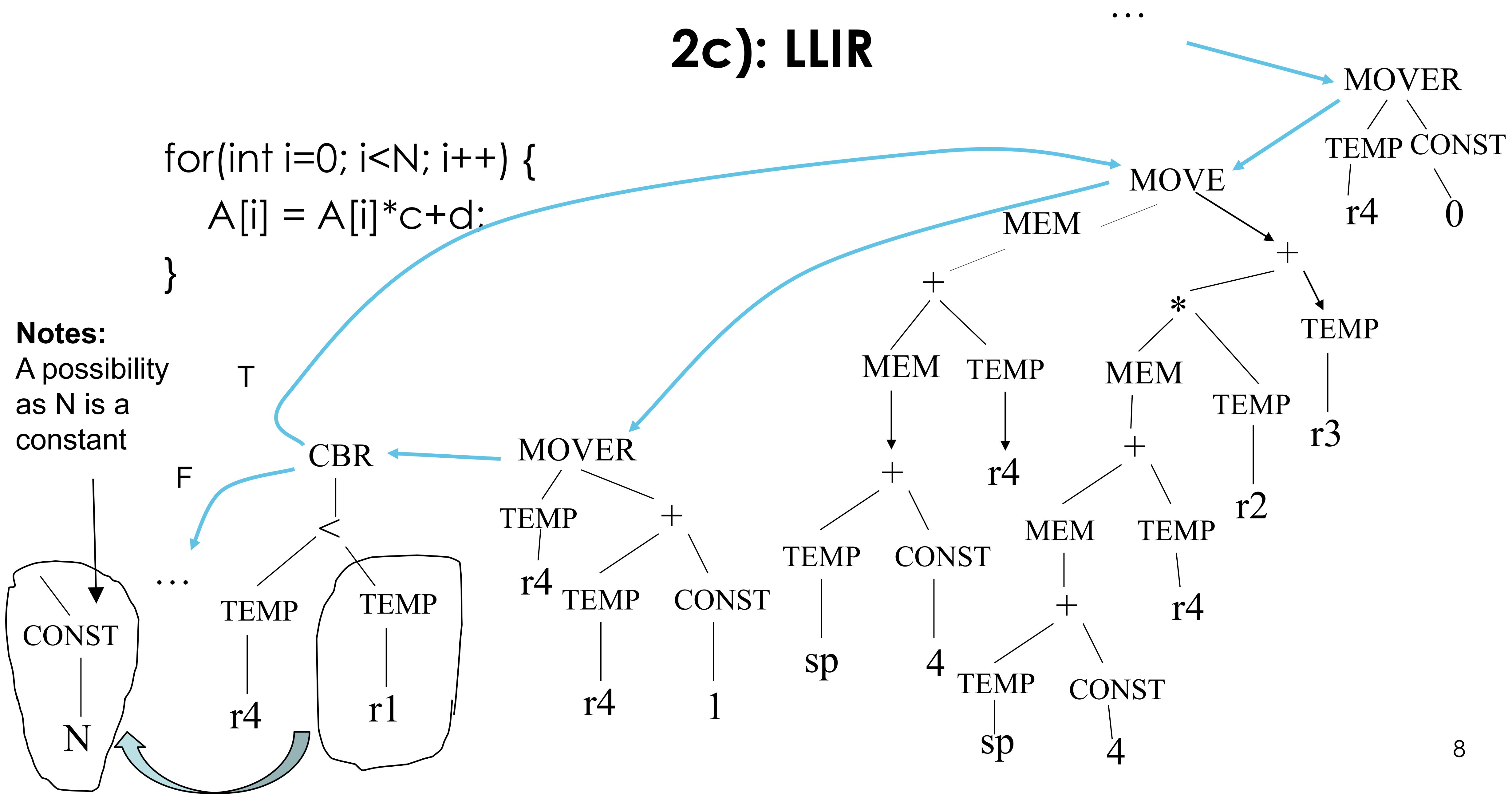
2c) LLIR

- Draw an LLIR for the code example which allows to generate more efficient code than the LLIR of 1b), having as the basis the expression trees but considering the *Jouette* + (instruction set presented in annex) as the target, when "N" represents a constant with value 1,000.

2c): LLIR

```
for(int i=0; i<N; i++) {
    A[i] = A[i]*c+d;
}
```

Notes:
A possibility as N is a constant



```
// Live-in={d}

// N and M represent integer constants

int A[N];
i=0;
max = MIN_INT;
loop1: if(i>=N) goto end1;
A[i] = read();
if(A[i] > max) max = A[i];
i = i+1;
goto loop1;
end1:

s = min(M,N);
int B[s];
i=0;
loop2: if(i>=s) goto end2;
B[i] = A[i]/max;
i = i+1;
goto loop2;
end2:

int C[s];
i=0;
loop3: if(i>=s) goto end3;
C[i] = B[i]*d;
i = i+1;
goto loop3;
end3:
// array C is the only array used after this point
```

3

3a)

- **Describe how to solve this problem considering the use of “liveness analysis”;**

- We can use “liveness analysis” to determine the liveness of the local array variables. Each write to an array is a “def” and each read a “use”.
- From the liveness analysis we can build an interference graph (IG) for the array variables.
- Then we can iteratively merge pairs of nodes not connected each other. The selection of the ordering of how two nodes are merged can be based on the size of the arrays (e.g., it is more useful to merge nodes corresponding to large arrays).
- When there are not more nodes to merge, we can select one stack region for each node in the resultant graph. For each node, the stack space needed will be the maximum space needed by the arrays associated to that node.

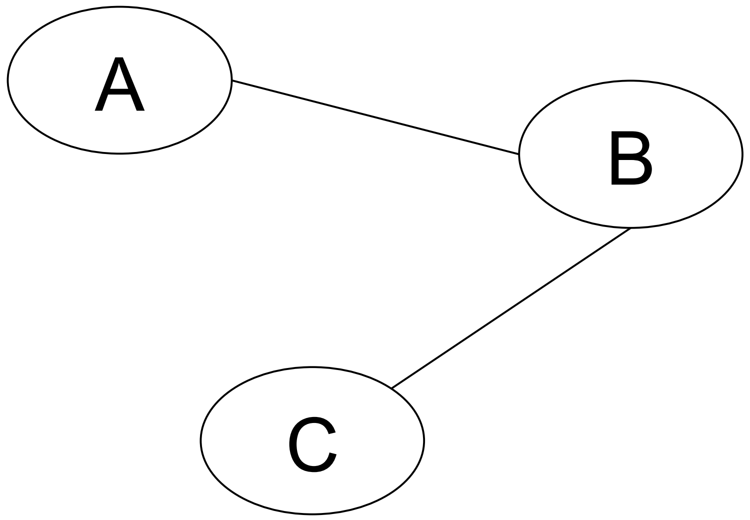
3b) Use and Defs for Arrays

```
// Live-in={d}
// N and M represent integer constants
int A[N];
i=0;
max = MIN_INT;
loop1: if(i>=N) goto end1;
A[i] = read();                    def={A}, use={}
if(A[i] > max) max = A[i];        def={}, use={A}
i = i+1;
goto loop1;
end1:

s = min(M,N);
int B[s];
i=0;
loop2: if(i>=s) goto end2;
B[i] = A[i]/max;                  def={B}, use={A}
i = i+1;
goto loop2;
end2:

int C[s];
i=0;
loop3: if(i>=s) goto end3;
C[i] = B[i]*d;                    def={C}, use={B}
i = i+1;
goto loop3;
end3:
// array C is the only array used after this point
```

- All lines with **def={}, use={}** except the ones on the left with def and use sets
- Is it needed to mark as def the declaration of the arrays?

3c)

- Interference Graph (IG) for array variables:



3d:

- **Describe how graph coloring can be used and show the result of applying graph coloring to the graph of interferences.**

- We can use graph coloring considering k=3 (number of nodes) and try to use as fewer as possible colors when coloring. In this case, we can simplify by the order A→B →C and then color by C→R1, B→R2 and A→R1. This indicates that arrays A and C can use the same stack region (necessary to store N ints).
- Note, however, that this was lucky as the coloring algorithm does not try to assign the same color to nodes not connected (this is a different problem).
- One possibility is to apply graph coloring considering K = number of nodes, simplify the graph as usual, and in the coloring stage try to color a node with the same color of other node not connected to it (and here we can think about possible approaches to select nodes).
- Another possibility is to use the approach described in 3a). Then, the coloring resumes to color each node in the IG with a different color. In the example of 3c), we can merge all the IG nodes without edges among them and then color the resulting IG with the minimum number of colors we can. The merged nodes of the IG will imply that a stack region given by the maximum stack space used by the respective arrays can be used for all those arrays. For instance, nodes A and C can be merged and then we can assign a different color to each one of the 2 resultant nodes.