xy5}

{7}

20

Multiple loops

• It is possible for two loops to share the same header

• This example has two back edges, (5,3) and (7,3)

• In many cases these two natural loops arise from one sourcecode loop

February 20

p3

p4

p5

p7

p1 p2

p6

{1} {2}

{3,4,5,6,7}

{4,5,6,7}

{6,7} {5}

{7} Back edges

21

M1a

Solution 2:

**\b0\* ( [1-24-9] \d\* | 3 ( [0-13-9]? | \d{2,} ) ) \b** any easier way?

\b - Start of string

0\* - Optional many preceding 0s

[1-24-9]\d\* - Any number that doesn’t start with 3

| - Or

3 - A 3

([0-13-9]?) that isn’t followed by just a 2 (removing 32)

| - Or

\d{2,} - Followed by at least 2 digits

\b - End of string

Solution 3: [1-9][0-9]{3,} | [124-9][0-9] | [1-9][013-9]

Solution 4: [0-9]\* ( (3 | [0-1 3-9]) | ([0-2 4-9] [0-9]) )

this matches zero though, and doesn’t match single digit numbers

Official answer is **0\* ( [1-9] \d{2,}? | [1-2]\d | 3[0-13-9] | [4-9]\d )**

0\* - Optional many preceding 0

[1-9] \d{2,}? - Any single digit or triple or more number

[1-2] \d - Anything from 10-29

3[0-13-9] - 30-39 excluding 32

[4-9]\d - Anything from 40-99

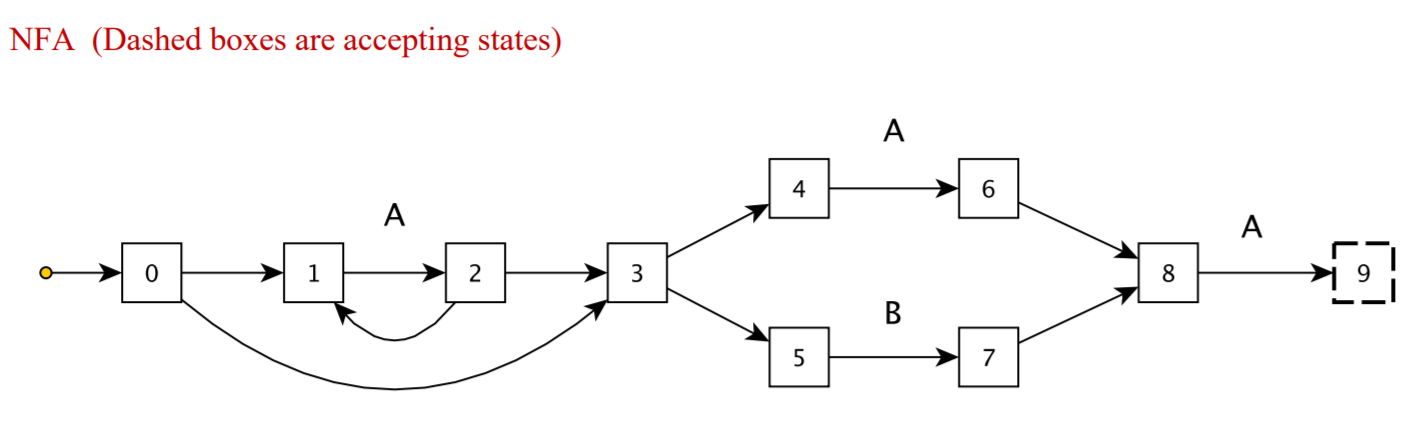
Solution 5: **([0-24-9][0-9]\*) | (3[0-13-9][0-9]\*) | (32[0-9]+) | 3**

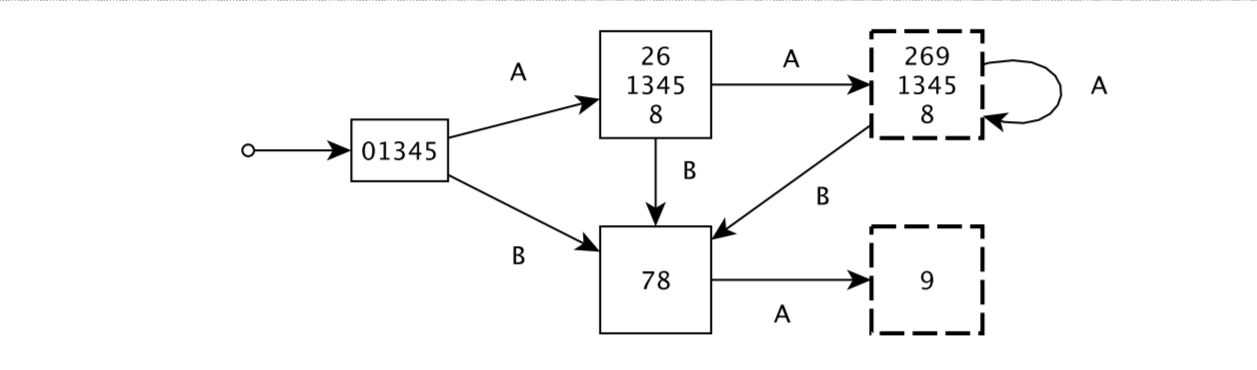
* First part matches all non-3-starting numbers
* Second part matches 3 starting n-digit numbers avoiding 32
* Third part matches 32 starting numbers with any number of digits >= 3
* Last part matches 3 on its own

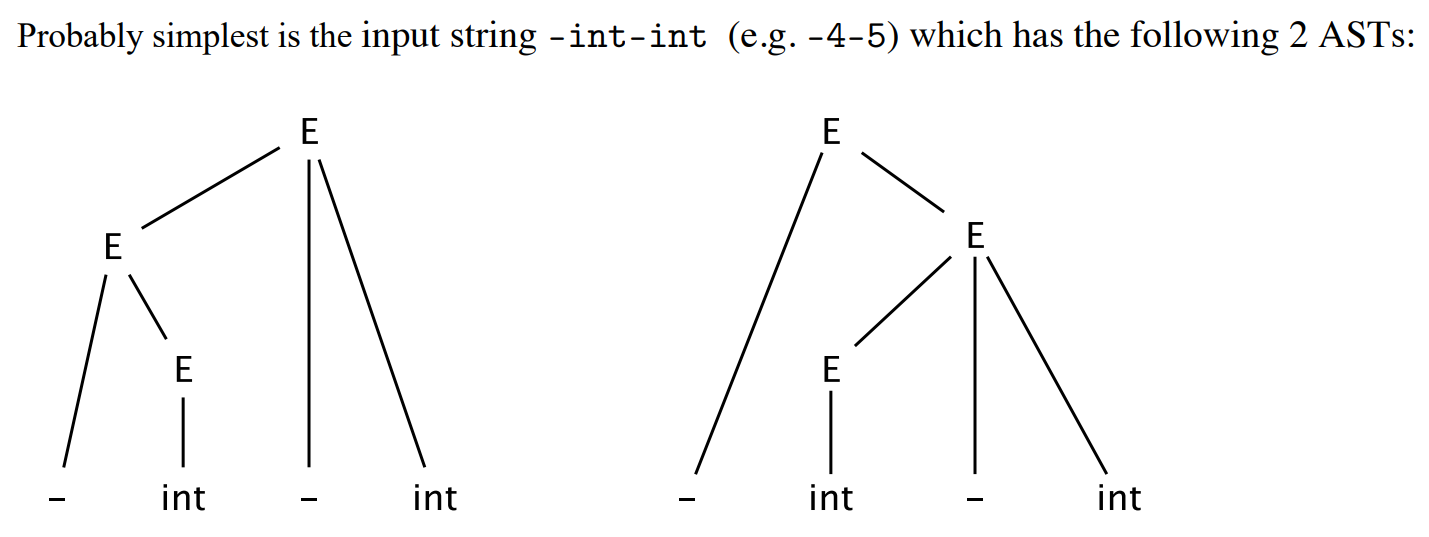
Useful:

[https://regex101.com/](https://regex1101.com/)

1b

i) 

ii)

1ci) 

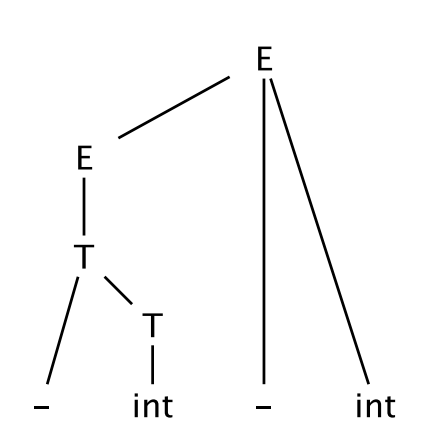
Can bracket as -(int - int) or (-int) - int

cii)

Use a different non-terminal for each precedence level. Lowest precedence always comes first.

E -> E - int | T

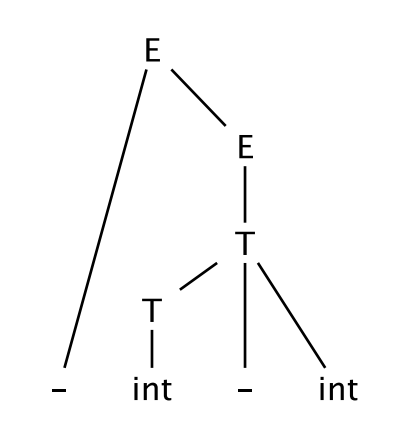
T -> - T | int



ciii)

E -> - E | T

T -> T - int | int



1d i)

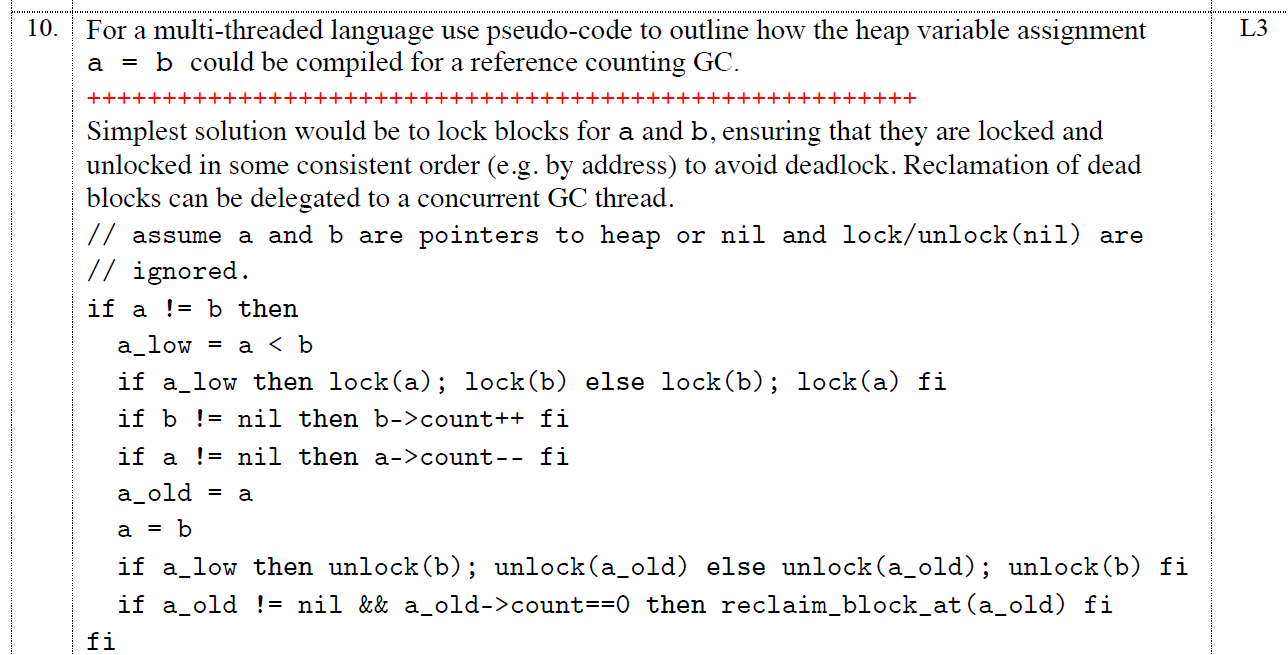
After line 2: two objects in the heap with stack references from a to one and b to the other.

After line 3: two objects in the heap with stack references from a to one and b to the other, and .next references from each heap object to the other heap object in a cycle.

After 4: two objects in the heap with no references to them from the stack but the cyclic references from each to the other.

Without explicit support for detecting cycles, a referencing counting GC would not be able to reclaim the two objects that were referenced by a and b, since they both reference each other. However, both should be reclaimable as they are not accessible from any variable in the program.

1d ii)



2a)

i) ReachIn(S12) = {S1,S4, S5, S6, S8, S9, S10, S12, S16} **??(Not correct as we only consider local variables)**

If we assume that dereferences don't count as definitions, then ReachIn(S12) = {S1, S8, S9, S10, S12}. The semantics if dereferences do count is quite ambiguous: for example, what would happen to S6 \*r = 300 when S9 r = q?

ii) RelevantReachIn(S12) = {S9}.

iii) Dominated by S11 = {S11, S12, S13, S14, S15, S16, S17}

iv) Dominated by S16 = {S16, S17}

v)

--------

In Ex5, where PointsIn is defined, I think it implies that the id element in (var, id) must always be the line where the malloc originally took place. Considering different iterations, this leads to something like (please check!):

PointsIn(S10) = {(p, S1), (p, S2), (r, S1), (r, S2), (q, S1), (q, S2)}

2b)

i)

Spill q, p and n. i is used most often and is modified on every iteration so it would be more expensive if we were to spill it. r is written to most out of the remaining ones and has more accesses in the loop. q and p are used less often than i and r, and n is never modified so a read every iteration is less expensive than a read and a write like in the case of i and r.

----

Alternative:

Spill n - not accessed inside loop itself

Spill p - not written to for i > 0

Spill r - because r = q is the first line in the loop, so no point keeping original r value in a register

Note on i - we remove it in (ii):

Can maybe spill i - not written to inside loop body, and not even accessed if we extract the “if” statement

ii) i is not actually used in any of the calculations in the loop. We can just have a loop with n only and decrement n until we get to 0 to get the same effect. Even though n is being modified, it doesn’t matter as it isn’t used anywhere else. We can copy lines S9, S10, S12 to outside the loop as these lines have to happen initially.

2c)H

void testFunction(){

Scanner scanner = new Scanner(System.in);

int x = scanner.nextInt();

int a = ...

int b = ...

if (x < 100){

// use a in calculations

}else{

// use b in calculations

}

}

The conditional is dependent on user input. Do we need to spill a or b? Impossible to know which option is more likely to be picked without actually running the program.

2d)

1. Return the pointer from the function
2. Assign pointer to a global variable/ data structure that is outside the function scope
3. Pass to another function as a parameter
4. Return a pointer to a data structure that contains the pointer
5. Write at an address specified by a pointer parameter
6. Use pointer manipulation to overwrite a local variable up in the program stack
7. use black magic
8. Assign to class member variable