RL1 - Exercises and solution sketches

NOTE: We sketch here the solutions for the exercises of lecture RL1 in a brief manner. Note that a proper solution would require more detailed descriptions, explanations, and in some cases examples. Some of the exercises may have more than one solution, and we just show one of them.

Selected official solutions to exercises from HMU can also be found here: `http://infolab.stanford.edu/~ullman/ialcsols/sols.html`.

Exercise RL1.1. Building DFAs Solve exercise HMU 2.2.4.

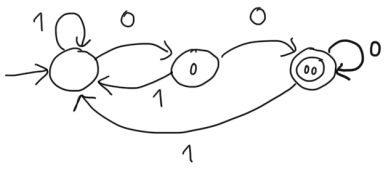
Exercise RL1.1. Building DFAs - solution In the below solutions we use state names that indicate the role of the state. For example:

- in (a) the name of the state indicates the sequence of 0's seen so far.
- in (b) the name of the state indicates how many 0's we have seen so far.
- in (c) the name of the state indicates the substring of 011 seen so far.

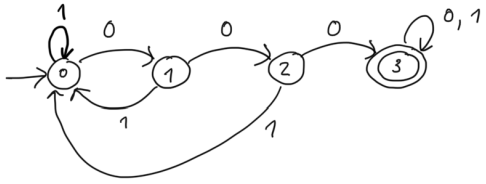
Finding good names fo the states can help in the design of DFA.
Note also the two kinds of final states we have in the solution:

- in (a) we have to leave the final state if we encounter a 1 (we reset the problem) since we need to finish with 00.
- in (b) and (c), instead, once we find the substring of interest we stay in the final state no matter what follows.

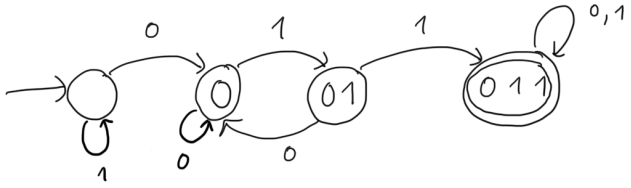
2.2.4 (a)



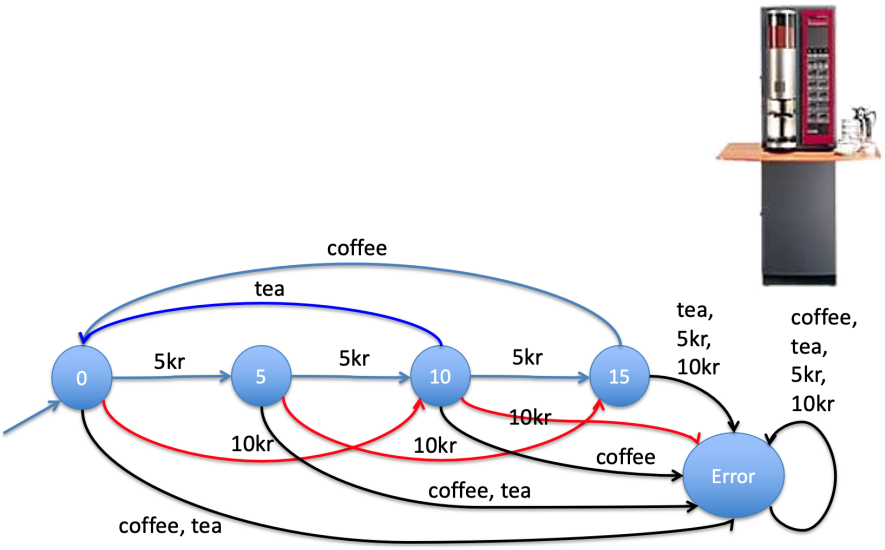
2.2.4 (b)



2.2.4 (c)



Exercise RL1.2. Vending Machine in F# Implement the below DFA of a vending machine in F#.

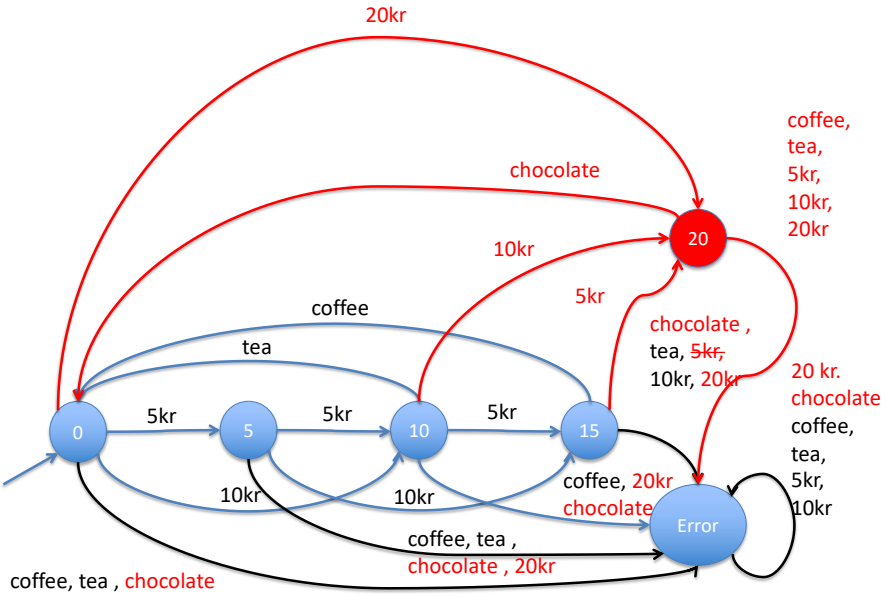


Exercise RL1.2. Vending Machine in F# - solution One possibility is to encode the automaton / vending machine as a follows:

```
// States are implemented as union types
type State = S0 | S5 | S10 | S15 | ERROR

// Input symbols are implemented as union types
type Symbol = KR5 | KR10 | TEA | COFFEE

// Transitions are implemented as a function
let nextState state symbol =
    match state , symbol with
    // Transitions from state S0
    | S0 , KR5    -> S5
    | S0 , KR10   -> S10
    | S0 , TEA    -> ERROR
    | S0 , COFFEE -> ERROR
    // Transitions from state S5
    | S5 , KR5    -> S10
    | S5 , KR10   -> S15
    | S5 , TEA    -> ERROR
    | S5 , COFFEE -> ERROR
    // Transitions from state S10
    | S10 , KR5    -> S15
    | S10 , KR10   -> ERROR
    | S10 , TEA    -> S0
    | S10 , COFFEE -> ERROR
    // Transitions from state S15
    | S15 , KR5    -> ERROR
    | S15 , KR10   -> ERROR
    | S15 , TEA    -> ERROR
    | S15 , COFFEE -> S0
      // Transitions from state ERROR
    | ERROR , KR5    -> ERROR
    | ERROR , KR10   -> ERROR
    | ERROR , TEA    -> ERROR
    | ERROR , COFFEE -> ERROR

// Running machine on a sequence of input symbols can be implemented recursively:
let rec run state input =
    match input with
    | [] -> state
    | symbol::more -> run (nextState state symbol) more

printfn "Running...\nFinished at state %A" (run S0 (KR5::KR10::COFFEE::[]))
```

Exercise RL1.3. Extension of the Vending Machine Extend the vending machine of the previous exercise to offer chocolate at a price of 20kr and to accept 20kr coins as well.

Exercise RL1.3. Extension of the Vending Machine - solution In the picture the main changes with respect to the original vending machine are highlighted in red.

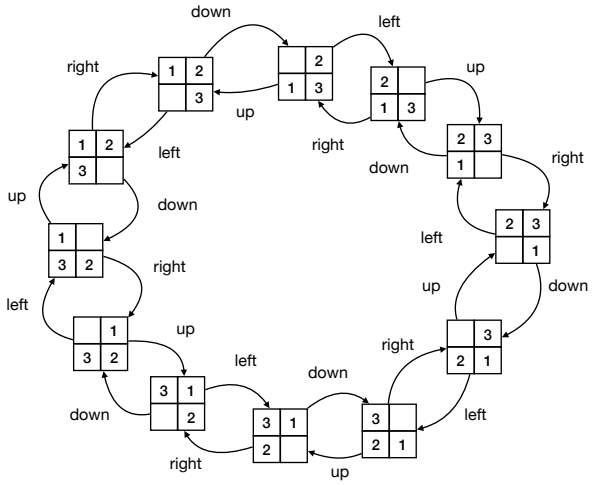


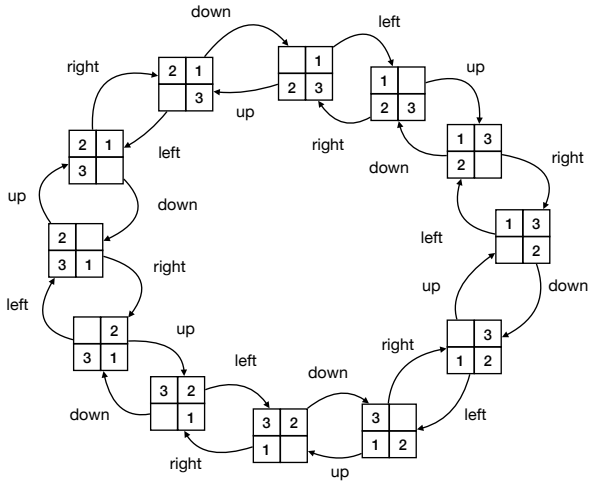
Exercise RL1.4. 2x2 puzzle Model the 2×2 puzzle, the simplest form of the $N \times N$ puzzle (`https://en.wikipedia.org/wiki/15_puzzle`) with a DFA. Which sequences of transitions help you solve the puzzle?

Exercise RL1.4. 2x2 puzzle - solution The following DFA models the entire puzzle. For simplicity we use the compact representation with an implicit error state that we do not depict (missing transitions all lead to that state).

The states are the possible placement of the tiles (1, 2 and 3) and the hole (denoted with a blank space). The actions are up, down, left, right and correspond to moving a tile to the hole. Starting in any state you can solve the puzzle by reaching the state (1,2,3,blank).

Note that there are two disconnected parts in the automaton. The one above is the actual one that you can play regularly. The one below is one that you could obtain by physically extracting a tile from the board and replacing it by an adjacent one. In the new board obtained there is no way to solve the puzzle.

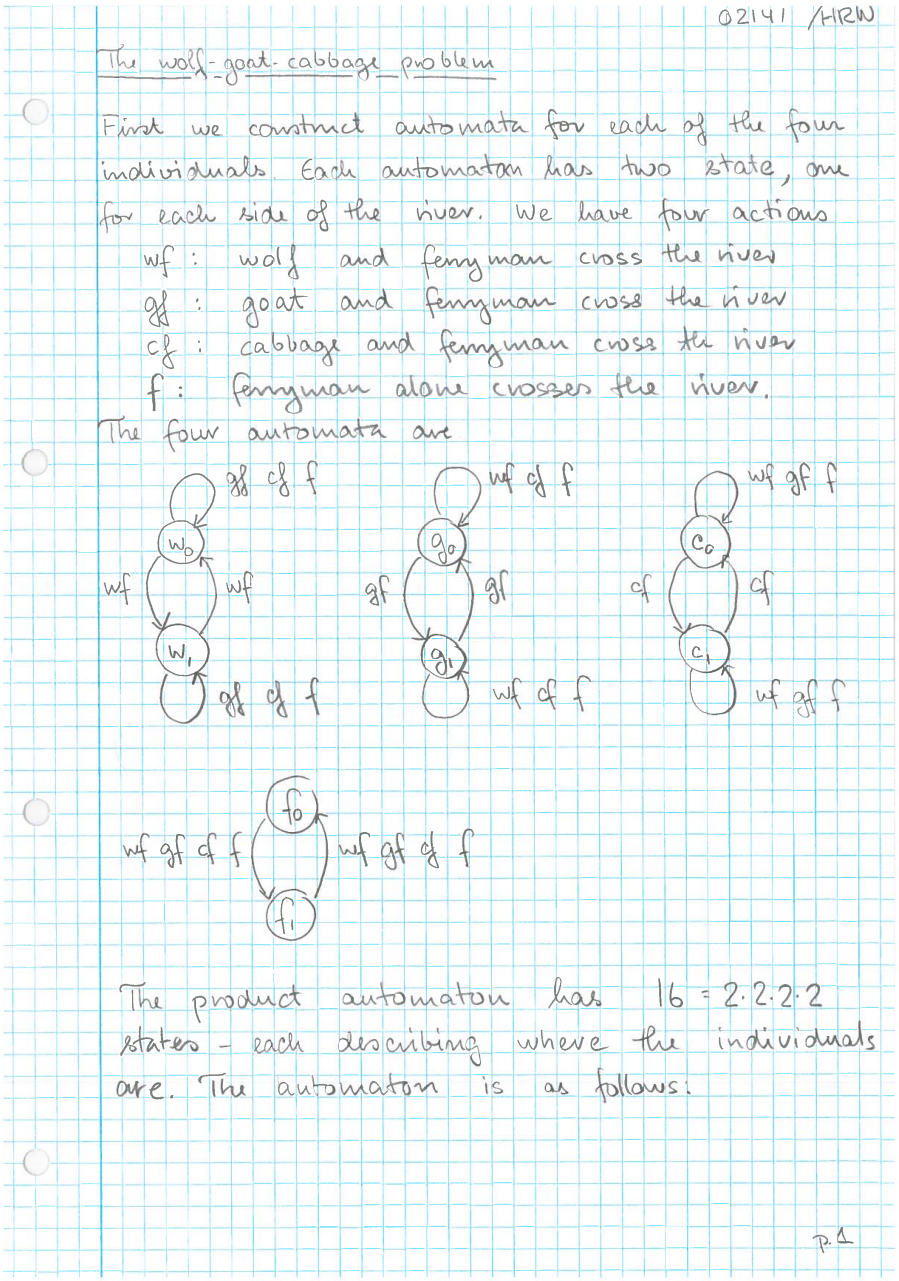


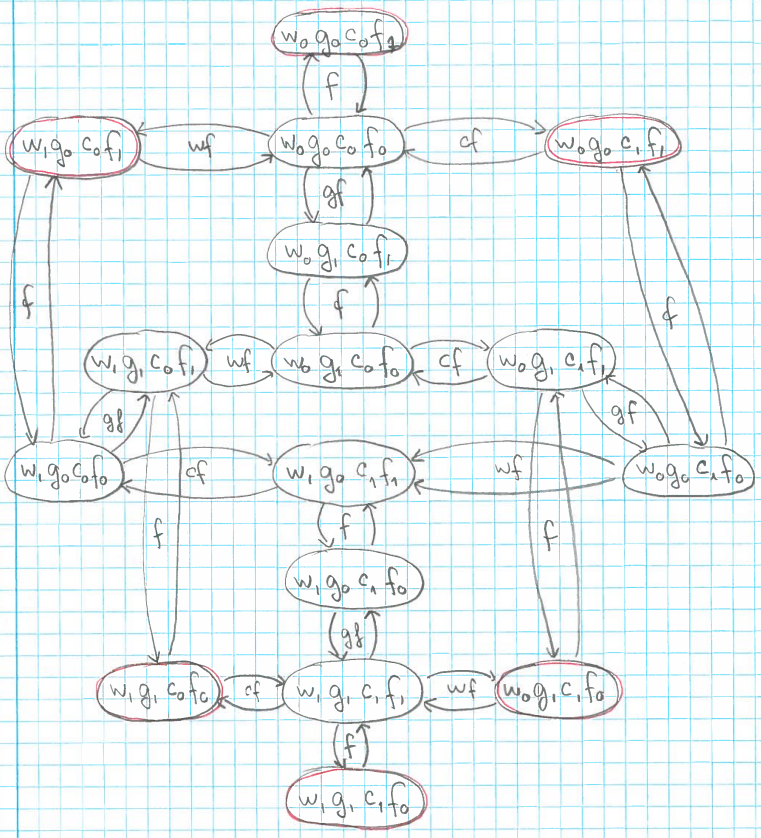


Exercise RL1.5. Wolf-goat-cabbage Model the classical river-crossing puzzle "wolf, goat and cabbage" (`https://en.wikipedia.org/wiki/Wolf,_goat_and_cabbage_problem`) with DFA.

- Model each of the four individuals (ferrymand, goat, cabbage and wolf) as a separate individual.
- Model the entire puzzle as the DFA that results from the product (intersection) of all 4 DFA.
- Identify the "bad" states of the puzzle (where an individual can eat another one)
- Identify a sequence of actions (a string) that solves the puzzle (i.e. everyone safely crosses the river).

Exercise RL1.5. Wolf-goat-cabbage - solution





The dangerous states in the automata are marked with red circles: the wolf and the goat on the same side (without ferryman) or goat and cabbage on the same side (without ferryman).

The following is a safe sequence of boat trips (from $\langle w_0 g_0 c_0 f_0 \rangle$ to $\langle w_1 g_1 c_1 f_1 \rangle$):

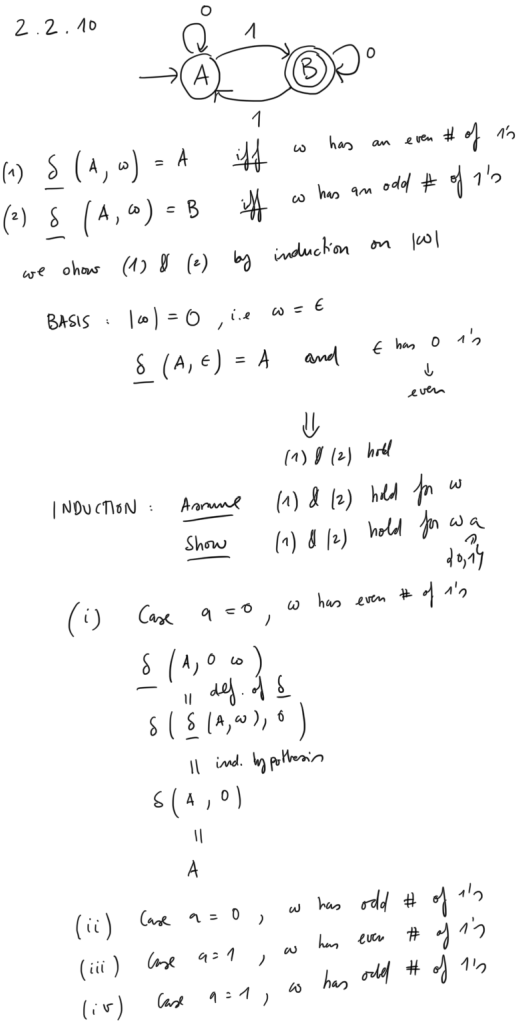
gf - f - wf - gf - cf - f - gf

P.2

Exercise RL1.6. Understanding DFAs Solve exercise 2.2.10.

Exercise RL1.6. Building DFAs - solution In the below proof, we will prove two properties (1) and (2) at the same time. The proof is by induction on size of w (equivalently, by induction on the structure of w). The inductive case is when we have a string wa. It requires us to inspect four cases. In the sketch below we only develop one.

A similar proof can be found at `http://infolab.stanford.edu/~ullman/ialcsols/sol2.html`. There, the proof only uses statement (1), since (2) holds iff (1) does not hold.



Cases (ii), (iii) and (iv) above are not presented in full detail for brevity. They are similar to (i).

Exercise RL1.7. Proving properties of transition functions Solve exercise HMU 2.2.2.

Exercise RL1.7. Proving properties of transition functions - solution

02141 /HIRW.

Exercise HMU 2.2.2.

We have to prove

$$\hat{\delta}(q_1, xy) = \hat{\delta}(\hat{\delta}(q_1, x), y) \qquad (*)$$

where $\hat{\delta}$ is defined by

$$\hat{\delta}(q_1, \varepsilon) = q_1 \qquad (1)$$

$$\hat{\delta}(q_1, za) = \delta(\hat{\delta}(q_1, z), a) \qquad (2)$$

The proof is by induction on $|y|$.

Base case: Then $|y| = 0$ so $y = \varepsilon$. We calculate:

$\hat{\delta}(q_1, x\varepsilon) = \hat{\delta}(q_1, x)$ as $x\varepsilon = x$

$\hat{\delta}(\hat{\delta}(q_1, x), \varepsilon) = \hat{\delta}(q_1, x)$ using (1)

so (*) follows.

Inductive case: The induction hypothesis gives

$$\hat{\delta}(q_1, xz) = \hat{\delta}(\hat{\delta}(q_1, x), z) \text{ if } |z| \leq k \qquad (IH)$$

Now assume $|y| = k+1$; then $y = za$ where $|z| = k$.

We calculate

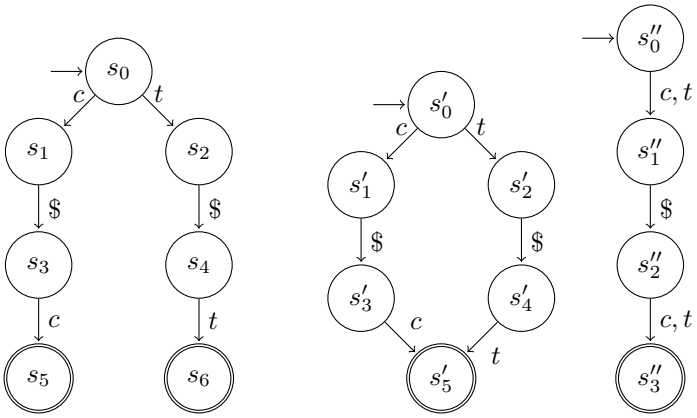
$$\hat{\delta}(q_1, xza) = \delta(\underline{\hat{\delta}(q_1, xz)}, a) \quad \text{using (2)}$$

$$= \delta(\underline{\hat{\delta}(\hat{\delta}(q_1, x), z)}, a) \quad \text{using (IH)}$$

$$\hat{\delta}(\underline{\hat{\delta}(q_1, x)}, za) = \delta(\hat{\delta}(\underline{\hat{\delta}(q_1, x)}, z), a) \quad \text{using (2)}$$

so (*) follows.

Exercise RL1.8. Testing equivalence of DFA A naïve teacher provides the following exercise "construct a DFA for a coffee machine that only accepts the following behaviour: choose coffee (c) or tea (t), pay one dollar (\$), and confirm your beverage choice". Students Alice, Bob and Charlie come out with the below DFA as possible solutions. Are those DFA language-equivalent? Find out by applying the algorithm seen in class for testing equivalence of DFA.



Exercise RL1.8. Testing equivalence of DFA - solution The first two DFA are equivalent. Running the algorithm we will discover the following groups of equivalent states:

- $\{s_0, s'_0\}$
- $\{s_1, s'_1\}$
- $\{s_2, s'_2\}$
- $\{s_3, s'_3\}$
- $\{s_4, s'_4\}$
- $\{s_5, s_6, s'_5\}$

Instead, the third one is not equivalent to any of the two first. It is easy to see that we will spot this as soon as we see that s''_2 can easily issue a challenge (reach an accepting state with either c or t) to $\{s_3, s_4, s'_3, s'_4\}$ to which any of them can reply.

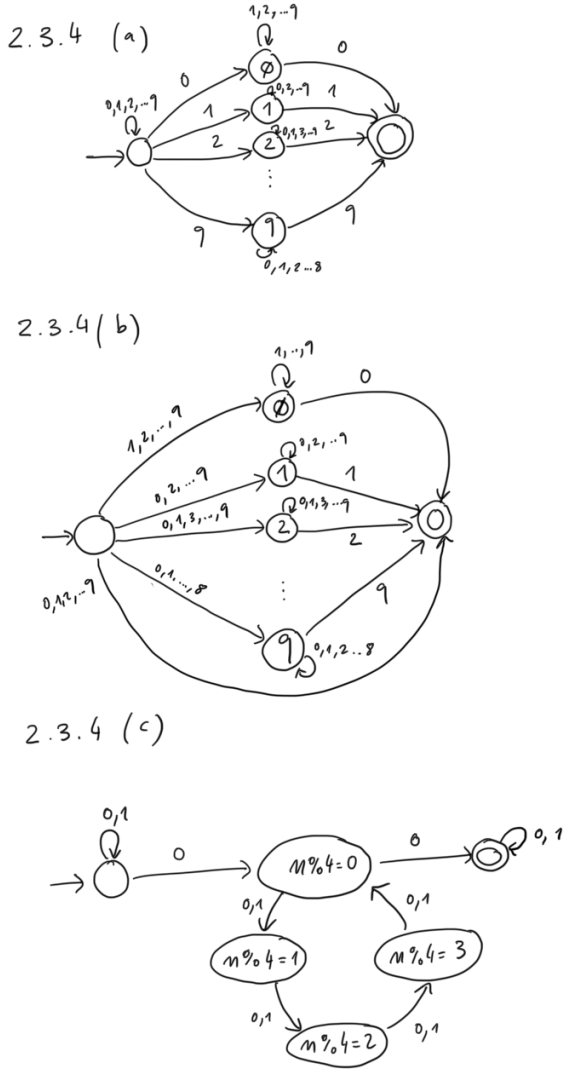
Hungry of more exercises? Check the HMU book (exercises 2.2.X). Recall that Selected official solutions to exercises from HMU can also be found here: `http://infolab.stanford.edu/~ullman/ialcsols/sols.html`.

RL2 - Exercises and solution sketches

NOTE: We sketch here the solutions for the exercises of lecture RL2 in a brief manner. Note that a proper solution would require more detailed descriptions, explanations, and in some cases examples. Some of the exercises may have more than one solution, and we just show one of them.

Exercise RL2.1. Building and understanding NFAs Solve exercise HMU 2.3.4.

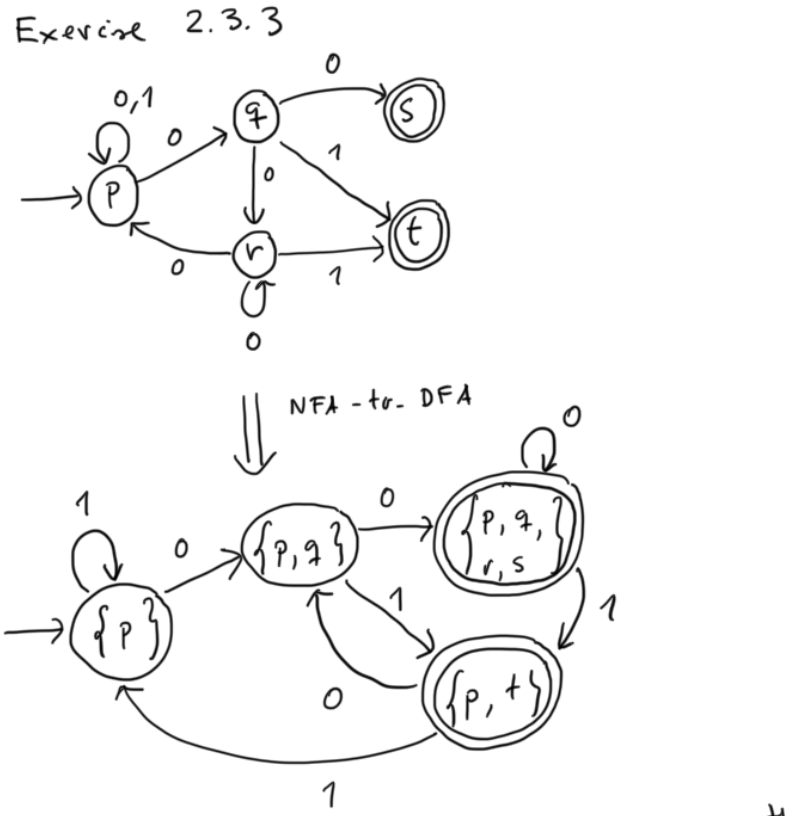
Exercise RL2.1. Building and understanding NFAs - solution



In (a) and (b) the name of the middle states (0,1,..9) indicates the "guess" of the digit. In (c) the name of the states in the loop indicate how many symbols between the two 0s we have observed (modulo 4).

Exercise RL2.2. NFAs to DFAs Solve exercise HMU 2.3.3.

Exercise RL2.2. NFAs to DFAs - solution



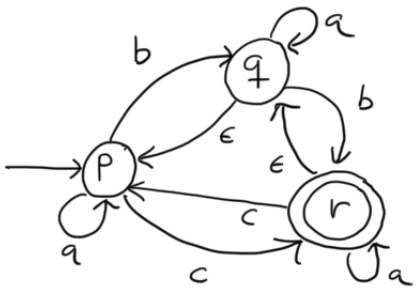
Strings of 0's and 1's that finish with

00 or 01

Exercise RL2.3. On ϵ-NFAs Solve exercises HMU 2.5.1 and 2.5.3(b). In 2.5.1(b) consider strings of length 2 or less (instead 3 or less as the book says).

Exercise RL2.3. On ϵ-NFAs - solution

Exercise 2.5.1



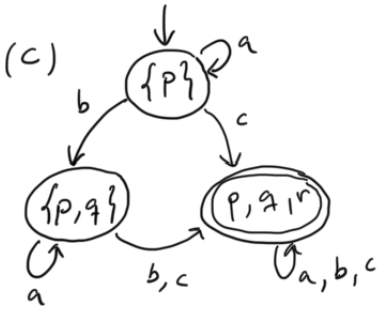
(a)

	ECLOSE
p	{p}
q	{q, p}
r	{r, q, p}

(b)

w	accept?
ε	×
a	×
b	×
c	✓
a a	×
a b	×
a c	✓
b a	×
b b	✓
b c	✓
c a	✓
c b	✓
c c	✓

(c)

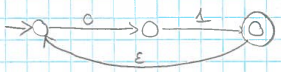


02141 /HRW

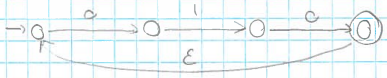
Exercise HMU 2.5.3 (b)

Design an ε-NFA for the set of strings consisting of either 01 repeated one or more times or 010 repeated one or more times.

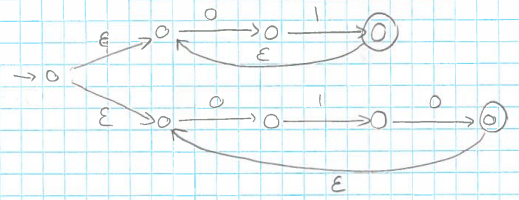
The following ε-NFA will accept strings with 01 repeated one or more times:



The following ε-NFA will accept strings with 010 repeated one or more times:



For the combined language we therefore take



Exercise RL2.4. Correctness of the subset construction In class we have seen the proof of correctness of the subset construction, i.e. that the language of an NFA $N = (Q, \Sigma, \delta_N, q_0, F)$ is the same as the language of the DFA $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ obtained by the subset construction. The key property prove is $\forall q \in Q, w \in \Sigma.\delta_N^*(q, w) = \delta_D^*(\{q\}, w)$. Prove this again on your own. HINT: Use induction on the length of w or alternatively on the structure of w.

Exercise RL2.4. Correctness of the subset construction - solution The correctness of the subset construction is formally stated in Theorem 2.11 of HMU. The book provides a proof (page 63).

Hungry of more exercises? Check the HMU book (exercises 2.2.X). Recall that Selected official solutions to exercises from HMU can also be found here: `http://infolab.stanford.edu/~ullman/ialcsols/sols.html`.

RL3 - Exercises and solution sketches

NOTE: We sketch here the solutions for the exercises of lecture RL3 in a brief manner. Note that a proper solution would require more detailed descriptions, explanations, and in some cases examples. Some of the exercises may have more than one solution, and we just show one of them.

Exercise RL3.1. Writing and understanding regular expressions Solve exercises HMU 3.1.1 and HMU 3.1.4.

3.1.4 Exercises for Section 3.1

Exercise 3.1.1: Write regular expressions for the following languages:

* a) The set of strings over alphabet $\{a, b, c\}$ containing at least one a and at least one b.

b) The set of strings of 0's and 1's whose tenth symbol from the right end is 1.

c) The set of strings of 0's and 1's with at most one pair of consecutive 1's.

! **Exercise 3.1.4:** Give English descriptions of the languages of the following regular expressions:

* a) $(\mathbf{1} + \epsilon)(\mathbf{00^*1})^*\mathbf{0^*}$.

b) $(\mathbf{0^*1^*})^*\mathbf{000}(\mathbf{0} + \mathbf{1})^*$.

c) $(\mathbf{0} + \mathbf{10})^*\mathbf{1^*}$.

Exercise RL3.2. From DFA to regular expressions Solve exercise HMU 3.2.3.

Exercise 3.2.3: Convert the following DFA to a regular expression, using the state-elimination technique of Section 3.2.2.

	0	1
$\rightarrow *p$	s	p
q	p	s
r	r	q
s	q	r

Exercise RL3.3. From regular expressions to ϵ-NFAs Solve exercises HMU 3.2.4(a,b), HMU 3.2.5, and HMU 3.2.7.

NOTE: For 3.2.5: you can try several approaches (and compare the results): (1) construct the ϵ-NFA with the algorithm, and then remove the ϵ using the conversion of ϵ-NFA to DFA, (2) construct the ϵ-NFA with the algorithm, and then remove the ϵ using the simplification rules of HMU 327, (3) constructing the finite automaton directly.

Exercise 3.2.4: Convert the following regular expressions to NFA's with ϵ-transitions.

* a) $\mathbf{01}^*$.

b) $(\mathbf{0}+\mathbf{1})\mathbf{01}$.

c) $\mathbf{00}(\mathbf{0}+\mathbf{1})^*$.

Exercise 3.2.5: Eliminate ϵ-transitions from your ϵ-NFA's of Exercise 3.2.4. A solution to part (a) appears in the book's Web pages.

!! **Exercise 3.2.7:** There are some simplifications to the constructions of Theorem 3.7, where we converted a regular expression to an ϵ-NFA. Here are three:

1. For the union operator, instead of creating new start and accepting states, merge the two start states into one state with all the transitions of both start states. Likewise, merge the two accepting states, having all transitions to either go to the merged state instead.
2. For the concatenation operator, merge the accepting state of the first automaton with the start state of the second.
3. For the closure operator, simply add ϵ-transitions from the accepting state to the start state and vice-versa.

Each of these simplifications, by themselves, still yield a correct construction; that is, the resulting ϵ-NFA for any regular expression accepts the language of the expression. Which subsets of changes (1), (2), and (3) may be made to the construction together, while still yielding a correct automaton for every regular expression?

Exercise RL3.4. Algebraic laws Solve exercise HMU 3.4.2.

! **Exercise 3.4.2:** Prove or disprove each of the following statements about regular expressions.

* a) $(R + S)^* = R^* + S^*$.

b) $(RS + R)^*R = R(SR + R)^*$.

* c) $(RS + R)^*RS = (RR^*S)^*$.

d) $(R + S)^*S = (R^*S)^*$.

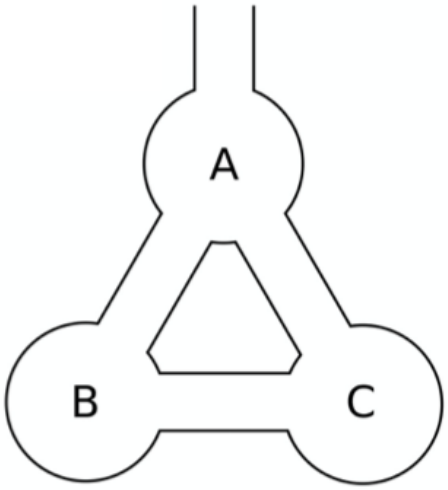
e) $S(RS + S)^*R = RR^*S(RR^*S)^*$.

Exercise RL3.5. Reversing a language Let $w \in \Sigma^*$ be a string. We write w^R for the reversal of w. For example $(011)^R$ is 110. Let L be a language over Σ. We denote by L^R the reversal of L, i.e. L^R is like L where all words are reversed.

(a) Provide a formal definition for the reversal function $\cdot^R : \Sigma^* \to \Sigma^*$ and the function $\cdot^R : \mathcal{P}(\Sigma^*) \to \mathcal{P}(\Sigma^*)$.

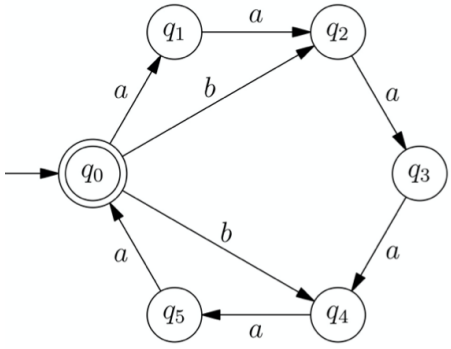
(b) Let L be a regular language. Show that L^R is a regular language.

Exercise RL3.6. Museum tours Supose we have a museum with 3 rooms: A, B and C as in the picture below.



Give a regular expression describing (non-empty) paths through the museum. Every path starts and ends in A. For example, $ABCABABCA$ is a valid path, but $ABBA$ is not.

Exercise RL3.7. RegEx for a automaton Supose we have the below automaton.



Construct a (preferably short) regular expression E that has the same language as the automaton.

Exercise RL3.1. Writing and understanding regular expressions - solution

3.1.1 (a) $(a+b+c)^* a (a+b+c)^* b (a+b+c)^* + (a+b+c)^* b (a+b+c)^* a (a+b+c)^*$

(b) $(0+1)^* 1 (0+1)^9$

(c) $(0+10)^* (11+1+\epsilon) (0+01)^*$

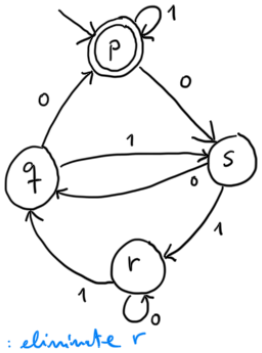
3.1.4 (a) "Strings of 0s and 1s without 2 consecutive 1's"

3.1.4 (b) "Strings of 0s and 1s with 000 as a substring"

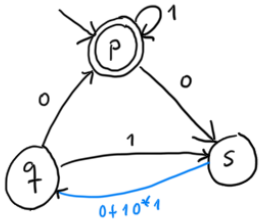
3.1.4 (c) "Strings of 0s and 1s where every 1 is followed by 0 except the last consecutive 1s"

Exercise RL3.2. From DFA to regular expressions - solution

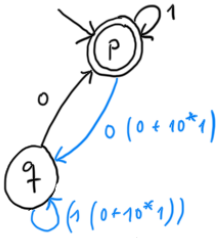
3.2.3



STEP 1 : eliminate r



STEP 2 : eliminate s

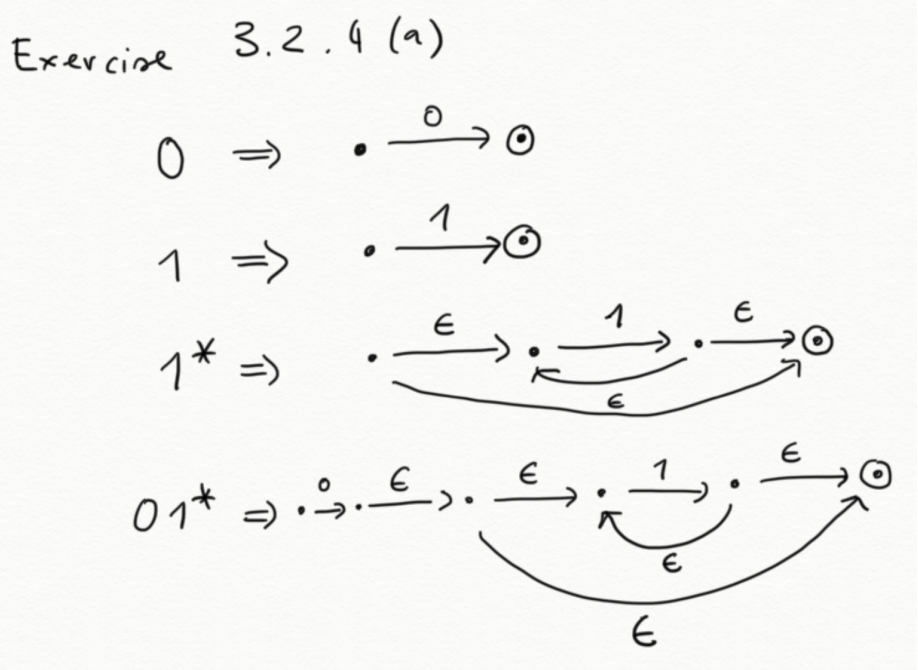


STEP 3 : eliminate q

STEP 4 : After elimination we get

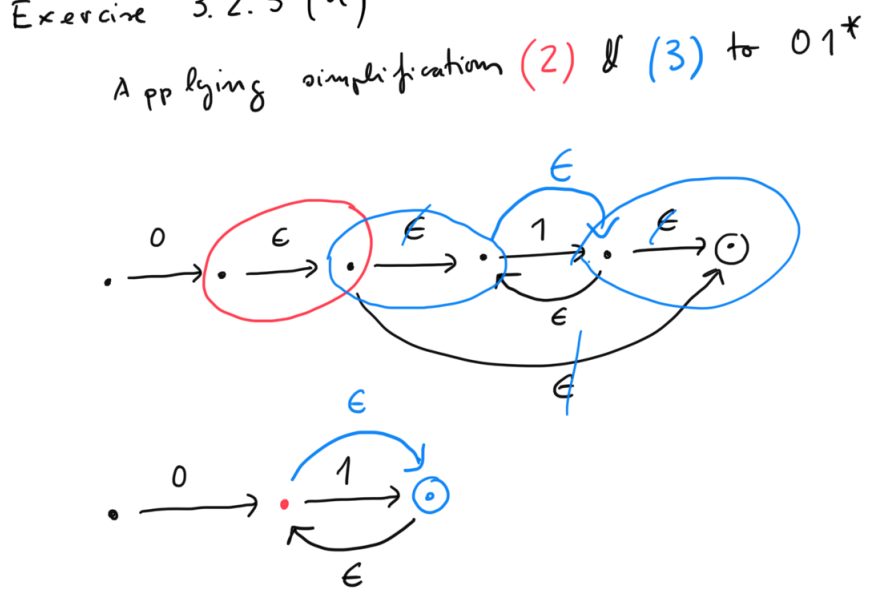
$$(1 + (0\ (0+10^*1)\ (1\ (0+10^*1))^*\ 0))^*$$

Exercise RL3.3. From regular expressions to ϵ-NFAs - solution For 3.2.4 we provide the solution for (a) only:



The solutions of 3.2.4 (b) and (c) can be obtained by applying the same method. You can easily check your final result with one of the recomended online tools for translating regular expressions to ϵ-NFA (e.g. `https://ivanzuzak.info/noam/webapps/fsm2regex/`).

For 3.2.5 we provide the solution for (a) only, applying some of the simplification rules of 3.2.7



This solution shows that the method in 3.2.7 cannot get rid of all ϵ transitions, contrary to the elimination algorithm based on ϵ-closure (as used in the ϵ-NFA to DFA conversion).

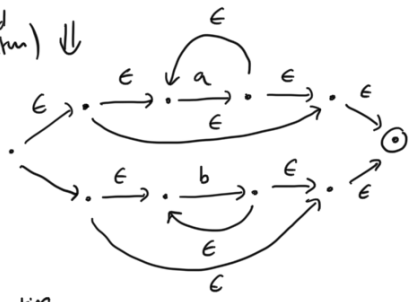
For 3.2.7 we provide an example of 2 simplification rules that do not work well together. Can you guess others?

Exercise 3.2.7

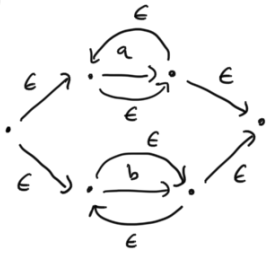
Example where simplification (1) & (3) do not work well

$a^* + b^*$

(standard algorithm) ⇓



simplification (3) ⇓



simplification (2) ⇓

this is not $a^* + b^*$!

it is $(a+b)^*$

Exercise RL3.4. Algebraic laws - solution

3.4.2

(a) $(R+S)^* \neq R^* + S^*$

$(R+S)^* \not\subseteq R^* + S^*$

↑

Let $R = a$, $S = b$ then

$ab \in (R+S)^*$ but $ab \notin R^* + S^*$

We can show $(R+S)^* \supseteq R^* + S^*$

$\|$ idem. of $+$ $\|$

$(R+S)^* + (R+S)^*$

$\cup$ $\cup$

R S

(b) $(RS+R)^* R = R(SR+R)^*$

PROOF by induction on n of

$\forall n \in \mathbb{N}at : (RS+R)^n R = R(SR+R)^n$

BASIS: $n = 0$

$(RS+R)^0 R$ $\quad$ $R(SR+R)^0$

$\|$ $\quad$ $\|$

$\epsilon R = R = R\epsilon$

INDUCTIVE STEP: Assume $(RS+R)^n R = R(SR+R)^n$

Show $(RS+R)^{n+1} R = R(SR+R)^{n+1}$

$(RS+R)^{n+1} R$ $\quad$ $R(SR+R)^{n+1}$

$\|$ def. of $^{n+1}$ $\quad$ $\|$

$(RS+R)(RS+R)^n R$ $\quad$ $R(SR+R)(SR+R)^n$

$\|$ ind. $\quad$ // distr.

$(RS+R)R(SR+R)^n$

$\|$ distr.

$(RSR+RR)(SR+R)^n$

PROOF 2

$$(RS+R)^* R = R\,(SR+R)^*$$

$$\text{II distr.} \qquad \text{II distr}$$

$$(R\,(S+\epsilon))^* R = R\,((S+\epsilon)R)^*$$

$$(AB)^* A = A(BA)^*$$

$$(AB)^\circ A \stackrel{?}{=} A(BA)^\circ$$

$$\text{II} \qquad \text{II}$$

$$\epsilon A = A = A\epsilon$$

Assume $(AB)^m A = A(BA)^m$

Show $(AB)^{m+1} A = A(BA)^{m+1}$

$$\text{II} \qquad \text{II}$$

$$AB(AB^m)A \cdot ABA(BA)^m$$

$$\text{II}$$

$$AB\,A(BA)^m$$

(c) $(RS+R)^* RS \neq (RR^* S)^*$

Let $R = \{a\}$, $S = \{b\}$

then $\epsilon \notin (RS+R)^* RS$

but $\epsilon \in (RR^* S)^*$

(d) $(R+S)^* S \neq (R^* S)^*$

Let $R = \{a\}$, $S = \{b\}$

then $\epsilon \notin (R+S)^* S$

but $\epsilon \in (R^* S)^*$

(e) $S(RS+S)^* R \neq RR^* S(RR^* S)^*$

Let $R = \{a\}$, $S = \{b\}$

then $ba \in S(RS+S)^* R$

but $ba \notin RR^* S(RR^* S)^*$

Exercise RL3.5. Reversing a language - solution

(a) The reversal function $\cdot^R : \Sigma^* \to \Sigma^*$ for strings can be defined as inductively:

$$\begin{aligned} \epsilon^R &= \epsilon \\ (wa)^R &= a(w^R) \end{aligned}$$

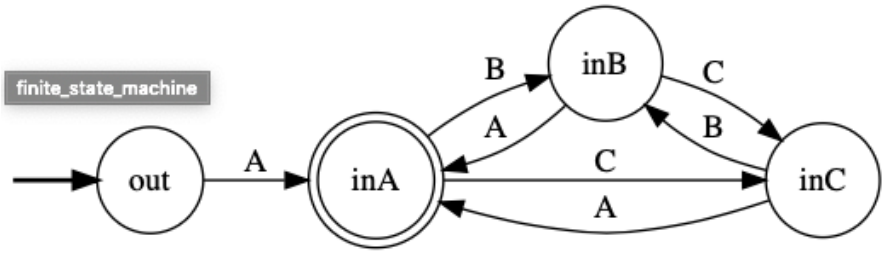
and the reversal function for languages $\cdot^R : \mathcal{P}(\Sigma^*) \to \mathcal{P}(\Sigma^*)$ can be defined simply as

$$L^R = \{w^R \mid w \in L\}$$

(b) This is formally stated in Theorem 4.11 of HMU, which includes a formal proof. The proof uses regular expressions and structural induction on regular expressions.

Exercise RL3.6. Museum tours - solution One way to solve this exercise is to see the museum map as an automaton, where every state represents the room in which we are (including the entrance as initial state) and every transition corresponds to entering a room. It is then easy to obtain a regular expression just by applying the transformation algorithm.

Automaton:



Regular expression:

$$A + A(CA + (B + CB)(CB)^*(A + CA))^*(\epsilon + CA + (B + CB)(CB) * (A + CA))$$

Exercise RL3.7. RegEx for a automaton - solution The safe way of solving the exercise and having the certainty that the obtained expression is equivalent to the automaton is to apply the transformation algorithm seen in class.

Alternatively, we can observe the 4 ways of forming a simple loop and then add the Kleen closure:

$$(aaaaaa + baaaa + baa)^*$$

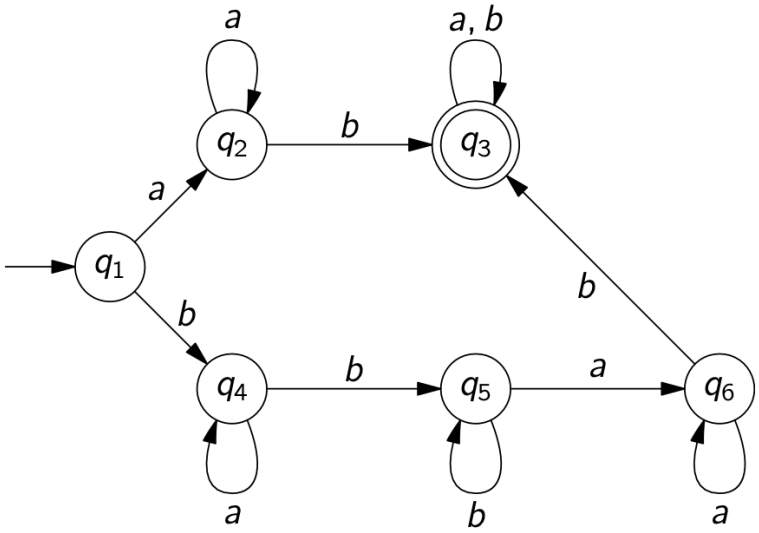
To prove that it is equivalent to the automaton we can transform it into a DFA and use the equivalence testing algorithm.

General remarks:

- We recommend that you first read all tasks during class. Make sure that you have a rough approach for every task in mind before you start working on the details. Ask for help if a task is unclear such that you do not get stuck at home.
- The difficulty of most exercises is comparable to typical exam tasks.
- During the exercise class, we will focus on the theoretical exercises; we will not answer questions about the mandatory assignment.

1 Minimization of DFAs

Use the partition (or block) refinement algorithm seen in class to construct a minimal DFA that accepts the same language es the DFA below.



2 The Myhill-Nerode Relation

For each of the following languages L_i, determine the set of L_i-equivalence classes. If that set is finite, construct a minimal DFA that accepts L_i.

(a) $L_1 = \{w \in \{a,b\}^* \mid aba$ or $abba$ appears in $w\}$.

(b) $L_2 = \{1^n\, 0\, 1^{2n} \mid n \geq 0\}$.

3 The Pumping Lemma

Use the Pumping Lemma to prove that the following languages are not regular.

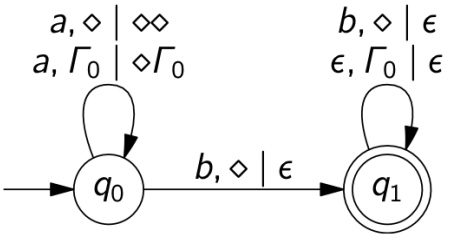
(a) $L_1 = \{a^k b^m c^{k+m} \mid k, m \geq 0\}$ over the alphabet $\Sigma = \{a, b, c\}$.

(b) $L_2 = \{1^k 0 1^k \mid k \geq 0\}$ over the alphabet $\Sigma = \{0, 1\}$.

(c) $L_3 = \{a^p \mid \text{p is a prime number}\}$ over the alphabet $\Sigma = \{a\}$.

4 Languages of Pushdown Automata

Consider the following PDA P:



(a) Show that P accepts the word $aabb$.

Hint: construct a sequence of IDs that starts with configuration $(q_0, aabb, \Gamma_0)$ and ends with $(q_1, \varepsilon, \varepsilon)$.

(b) What is the language $L(P)$ if we accept strings whenever we end up in a final state?

(c) What is the language $N(P)$ if we accept strings whenever we end up with an empty stack?

5 Constructing Pushdown Automata

(a) Construct a PDA P such that $L(P) = \{a^n b^{n+3k} \mid n, k \geq 0\}$.

(b) Construct a PDA P such that $N(P) = L(G)$ (i.e. P accepts with an empty stack), where G is the following context-free grammar:

$$S \;\rightarrow\; aSbS \mid C \qquad\qquad C \;\rightarrow\; cC \mid c$$

Hint: Use the construction considered during class.

CFL1 - Solution Sketches

February 21, 2023

We sketch here the solutions for the exercises of lecture CFL1 in a brief manner. Note that a proper solution would require more detailed descriptions, explanations, and in some cases examples. Some of the exercises may have more than one solution, and we just show one of them. When the exercise requests a context-free grammar, we provide one in a very compact way (as done in the slides) where we just provide the productions, with the implicit assumption that non-terminals start with capital letters, and that the initial symbol is the on corresponding to the first production.

Exercise 1.1.(a)

$$S \quad \rightarrow \quad \epsilon \mid 0S$$

To actually convince us that the above grammar G is a correct solution we need to prove that its language $L(G)$ coincides with $L = \{0^n \mid n \geq 0\}$. We can do this by proving that for every word w (1) $S \in L(G)$ implies $w \in L$ and (2) $w \in L$ implies $w \in L(G)$.

We show (1) by induction and using the bottom-up definition of a language (recursive inference). The base case is the production $S \rightarrow \epsilon$, from which we have that $\epsilon \in L(G)$. Trivially ϵ belongs also to L.

For the inductive step, we consider the production $S \rightarrow 0S$, from which we have that if a word $w' \in L(G)$ then $0w' \in L(G)$. Using the induction hypothesis $w' \in L(G)$ implies $w' \in L$. All we need to prove is that if $w' \in L$ then $0w' \in L$ which is trivial.

To show (2) we can proceed by induction on the length of $w = 0^n$. As base case take $n = 0$. In this case $w = \epsilon$. By using production $S \rightarrow \epsilon$ we know that $\epsilon \in L(G)$.

For the inductive step, we need to show that $w = 0^{n+1} \in L(G)$. By the inductive hypothesis we know that $0^n \in L(G)$. We can the use production $S \rightarrow 0S$ to conclude that $00^n = 0^{n+1}$ belongs to $L(G)$.

Exercise 1.1.(b)

$$S \rightarrow 0 \mid 0S$$

Exercise 1.1.(c)

$$\begin{array}{lcl} S & \rightarrow & AB \\ A & \rightarrow & 0 \mid 0A \\ B & \rightarrow & 1 \mid 1B \end{array}$$

Exercise 1.1.(d)

$$S \rightarrow 01 \mid 0S1$$

We can prove correctness of this solution by showing that the language $L = \{0^n 1^n \mid n \geq 1\}$ and the language L_S defined by S in our CFG is the same.

To do this we will prove two statements:

(1) For any word w, if $w \in L$ then $w \in L_S$

(2) For any word w, if $w \in L_S$ then $w \in L_1$

Proving (1) To prove (1) we observe that all words in S are of the form $0^n 1^n$. We can then prove the statement by induction on n.

Base case: $n = 1$. We have then that $w = 0^1 1^1 = 01$. We can show that $01 \in L_S$ with the simple derivation $S \Rightarrow 01$ obtained by applying the production $S \rightarrow 01$.

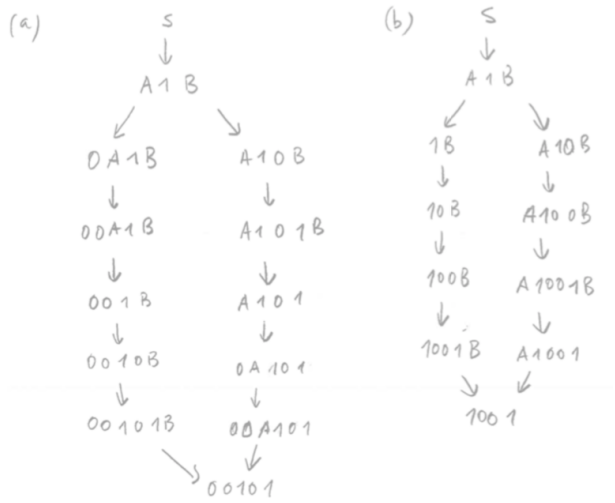
Inductive case: We will assume that (1) holds for words of the form $0^n 1^n$ and we will show that it then holds for words of the form $w = 0^{n+1} 1^{n+1}$. It is trivial to see that w can be rewritten as follows $w = 0^{n+1} 1^{n+1} = 00^n 1^n 1$. Now, by applying the induction hypothesis we know that $0^n 1^n \in L_S$. We can then use production $S \rightarrow 0S1$ and inference, to conclude that $00^n 1^n 1 \in L_S$.

To prove (2) we can apply induction on the length of the derivation or inference used to show that w is in L_S.

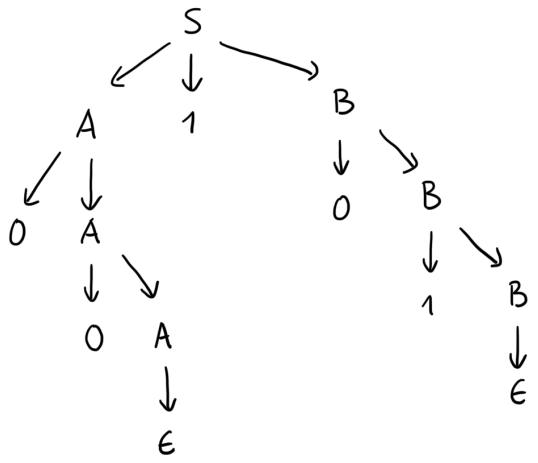
Base case: The only base case for inferring that $w \in L_S$ is when w has been obtained with the production $S \rightarrow 01$. In this case $w = 01$, which is clearly in L_S (in particular, for $n = 1$).

Inductive case. Assume that (2) holds for any word w. We have show that for any word w' that we can infer to be in the language with just one production, then w' is in L. The only inference step is provided by using production $S \rightarrow 0S1$. This yields $w' = 0w1$. By the induction hypothesis, we know that $w = 0^n 1^n$ for some number $n \geq 1$. It is trivial to see that we can rewrite w' as follows: $w' = 00^n 1^n 1 = 0^{n+1} 1^{n+1}$. Hence, $w' \in L$.

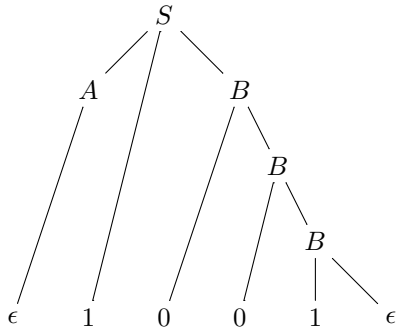
Exercise 1.2. For simplicity, the left- and right-most derivations are drawn in the same picture below.


(a)
S
A 1 B
0 A 1 B
A 1 0 B
0 0 A 1 B
A 1 0 1 B
0 0 1 B
A 1 0 1
0 0 1 0 B
0 A 1 0 1
0 0 1 0 1 B
0 0 A 1 0 1
0 0 1 0 1
(b)
S
A 1 B
1 B
A 10 B
10 B
A 10 0 B
100 B
A 100 1 B
1001 B
A 1001
1001

Exercise 1.3.(a)



Exercise 1.3.(b)



Exercise 1.4.(b)

$$S \rightarrow \epsilon \mid SS \mid (S)$$

Exercise 1.4.(c)

$$S \rightarrow 010 \mid 0S0$$

Exercise 1.4.(d)

$$\begin{aligned} S &\rightarrow 0S2 \mid A \\ A &\rightarrow \epsilon \mid 1A \end{aligned}$$

Exercise 1.4.(e)

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow \epsilon \mid 0A1 \\ B &\rightarrow \epsilon \mid 1B \end{aligned}$$

Exercise 1.4.(f)

$$S \rightarrow 011 \mid 0S11$$

Exercise 1.5. We could start with simple lists where symbols are just put one after the other, as in "010101010101". For lists of such format without separators the grammar is simple:

$$\begin{array}{lll} S & \rightarrow & \epsilon \mid AS \\ A & \rightarrow & 0 \mid 1 \end{array}$$

If we want to admit separators (for example, "," as in "0,1,0,1,0,1,0,1,0,1,0,1") we need a different grammar:

$$\begin{array}{lll} S & \rightarrow & \epsilon \mid N \\ N & \rightarrow & A \mid A, N \\ A & \rightarrow & 0 \mid 1 \end{array}$$

intuitively, N represents non-empty lists, and S possibly empty lists.

Exercise 1.6. In this solution we want to represent a simple tree with just one leaf 0 as the string 0. Composing two such leafs in a binary tree will be represented as $[0 \bullet 0]$. A complete binary tree of depth 2 would look like this $[[0 \bullet 0] \bullet [0 \bullet 0]]$. The following grammar accomplished this goal:

$$\begin{array}{lll} T & \rightarrow & \epsilon \qquad\qquad\qquad \text{(an empty tree)} \\ & \mid & A \qquad\qquad\qquad \text{(a leaf)} \\ & \mid & [T \bullet T] \qquad \text{(a root } \bullet \text{ with two subtrees)} \\ A & \rightarrow & 0 \mid 1 \end{array}$$

Exercise 1.7.a Yes. Any mapping can be write down as a list of (key,value) "pairs" in the format "key $\mapsto$ value".

Exercise 1.7.b It depends on the semantic interpretation of the words. For example, consider the following word:

$$0 \mapsto a, 0 \mapsto b$$

There are several ways of interpreting it as a map: as an ill-formed one (because the value of key 0 is not well-defined), as the map $0 \mapsto b$ (if we interpret the list as updates on a map, overwriting previous maps), or as the map $0 \mapsto a$ (if we discard overwriting already defined pairs of key/value).

Exercise 1.7.c It depends on the semantic interpretation of the words, but for a reasonable interpretation the answer is yes. Here are some examples

$$0 \mapsto a, 1 \mapsto b$$

and

$$1 \mapsto b, 0 \mapsto a$$

denote the same map (the order of single maps is irrelevant).

Also $0 \mapsto a$, and $0 \mapsto a, \mathsf{nil}$, and $0 \mapsto a, \mathsf{nil}, \mathsf{nil}$, etc. denote the same map (since nil is irrelevant).

Exercise 1.8. We actually saw a very similar grammar in the RL part of the course:

$$E \quad \rightarrow \quad e \mid \emptyset \mid 0 \mid 1 \mid E + E \mid EE \mid E* \mid (E)$$

Exercise 1.9. We define a function cfg that, given a regular expression E (i.e. a word recognised by the grammar of Exercise 1.8), and a non-terminal symbol S provides a set of productions such such that the language defined by S is the same as the language of E. The function is defined by induction on the structure of E:

$$\begin{array}{rcl}
\mathsf{cfg}(e, S) & = & \{S \rightarrow \epsilon\} \\
\mathsf{cfg}(\emptyset, S) & = & \emptyset \\
\mathsf{cfg}(0, S) & = & \{S \rightarrow 0\} \\
\mathsf{cfg}(1, S) & = & \{S \rightarrow 1\} \\
\mathsf{cfg}(E + E', S) & = & \{S \rightarrow S_E \ , S \rightarrow \ S_{E'}\} \ \cup \ \mathsf{cfg}(E, S_E) \cup \mathsf{cfg}(E', S_{E'}) \\
\mathsf{cfg}(EE', S) & = & \{S \rightarrow S_E S_{E'}\} \ \cup \ \mathsf{cfg}(E, S_E) \ \cup \ \mathsf{cfg}(E', S_{E'}) \\
\mathsf{cfg}(E*, S) & = & \{S \rightarrow \epsilon, S \rightarrow S_E S\} \ \cup \ \mathsf{cfg}(E, S_E) \\
\mathsf{cfg}((E), S) & = & \{S \rightarrow S_E\} \ \cup \ \mathsf{cfg}(E, S_E)
\end{array}$$

NOTE: The set of terminal and non-terminal symbols and the initial symbols are implicit and can be reconstructed from the resulting productions and from the invocation of the function.

NOTE: If an expression E appears multiple times as a subexpression, the final result will not contain multiple times the same production, as those are combined with set union.

The correctness of the solution can be shown by structural induction, with base cases e, $\emptyset$, 0 and 1 and inductive cases for the rest of the equations above.

CFL2 - Solution Sketches

February 23, 2023

We sketch here the solutions for the exercises of lecture CFL2 in a brief manner. Note that a proper solution would require more detailed descriptions, explanations, and in some cases examples. Some of the exercises may have more than one solution, and we just show one of them. When the exercise requests a context-free grammar, we provide one in a very compact way (as done in the slides) where we just provide the productions, with the implicit assumption that non-terminals start with capital letters, and that the initial symbol is the on corresponding to the first production.

Exercise 2.1

$$
\begin{array}{rcl}
\texttt{EMAIL} & \rightarrow & \texttt{<email> FROM TOP BCCN SUBJECT BODY </email>} \\
\texttt{FROM} & \rightarrow & \texttt{<from>\#PCDATA</from>} \\
\texttt{TOP} & \rightarrow & \texttt{TO} \mid \texttt{TO TOP} \\
\texttt{TO} & \rightarrow & \texttt{<to>\#PCDATA</to>} \\
\texttt{BCCN} & \rightarrow & \epsilon \mid \texttt{BCC BCCN} \\
\texttt{BCC} & \rightarrow & \texttt{<bcc>\#PCDATA</bcc>} \\
\texttt{SUBJECT} & \rightarrow & \texttt{<subject>\#PCDATA</subject>} \\
\texttt{BODY} & \rightarrow & \texttt{<body>\#PCDATA</body>}
\end{array}
$$

Exercise 2.2

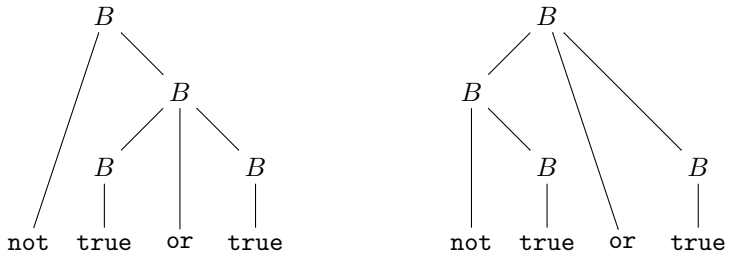
$$
\begin{array}{rcl}
\texttt{Obj} & \rightarrow & \{\} \mid \{\texttt{Pairs}\} \\
\texttt{Pairs} & \rightarrow & \texttt{Pair} \mid \texttt{Pair}, \texttt{Pairs} \\
\texttt{Pair} & \rightarrow & \texttt{String} : \texttt{Value} \\
\texttt{Array} & \rightarrow & [] \mid [\texttt{Values}] \\
\texttt{Values} & \rightarrow & \texttt{Value} \mid \texttt{Value}, \texttt{Values}
\end{array}
$$

Exercise 2.3 The are two sources of ambiguity: (1) precedence between `not` and `or` and (2) associativity of the binary operator `or`.

An example of (1) can be observed in string

```
not true or true
```

for which we can give two parse trees:



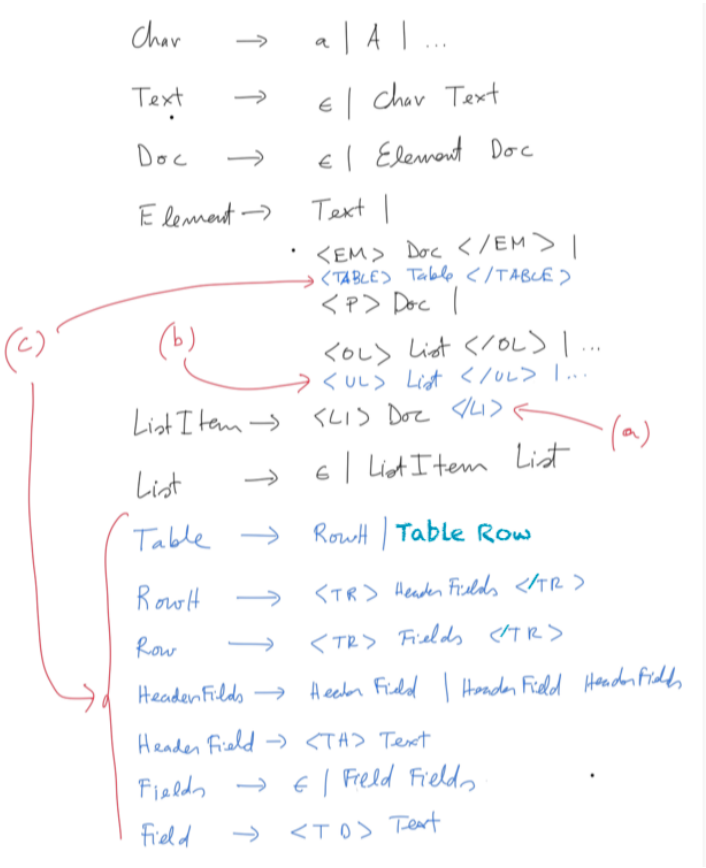
We begin solving (1) by giving precedence to `not` over `or` using the stratification technique:

$$\begin{array}{lcl} B_0 & \rightarrow & B_0 \texttt{ or } B_0 \mid B_1 \\ B_1 & \rightarrow & \texttt{true} \mid \texttt{not } B_1 \mid (B_0) \end{array}$$

Next we address (2) by choosing left-associativity and transforming the grammar as follows:

$$\begin{array}{lcl} B_0 & \rightarrow & B_0 \texttt{ or } B_1 \mid B1 \\ B_1 & \rightarrow & \texttt{true} \mid \texttt{not } B_1 \mid (B_0) \end{array}$$

Exercise 2.4



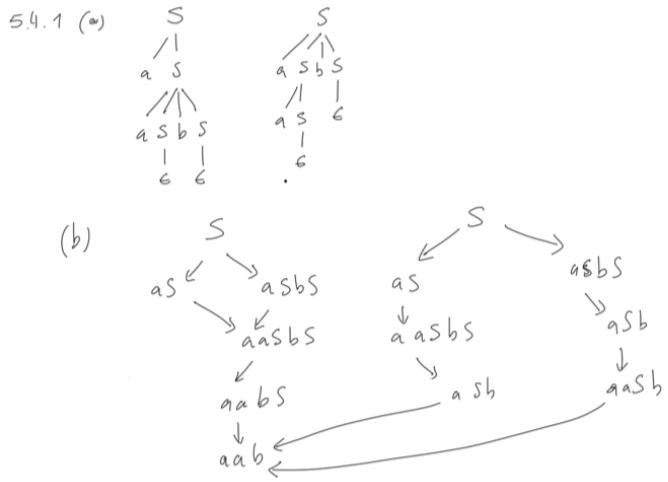
Exercise 2.5

```
COURSESOPECS  →  <COURSESPECS> COURSES </COURSESPECS>
      COURSES  →  COURSE | COURSE COURSES
       COURSE  →  <COURSE> CNAME PROFSTUDENTS TAP </COURSE>
        CNAME  →  <CNAME> #PCDATA </CNAME>
         PROF  →  <PROF> #PCDATA </PROF>
     STUDENTS  →  ϵ | STUDENT STUDENTS
      STUDENT  →  <STUDENT> #PCDATA </STUDENT>
          TAP  →  ϵ | TA
           TA  →  <TA> #PCDATA </TA>
```

Exercise 2.6 / 5.4.1 (d) The ambiguity is similar to the *dangling else* problem. Think of a being `if...then...` and b being `else`. The ambiguity means that in general the grammar does not tell us to which `if` an `else` belongs, e.g. in `aab` the `b` could refer either to the first or the second `a`.

A common convention in programming languages is that each `else` belongs to the closest syntactically possible `if`. E.g. in `aaabb`, the first `b` belongs to the third `a`, the second `b` belongs to the second `a`, the first `a` has no `b`, i.e. as writing `aaabb` in a programming language with braces for structuring.

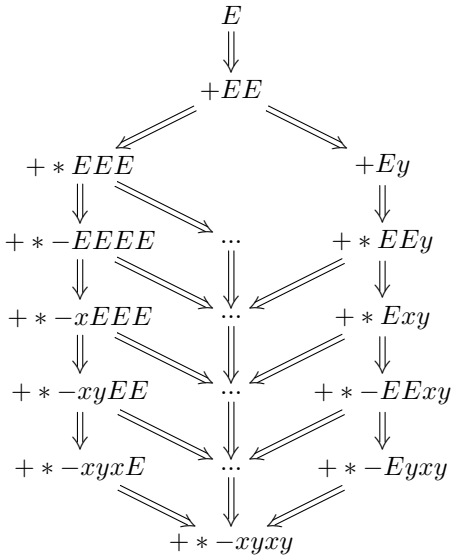
Exercise 2.6 / 5.4.1 (a)–(b)



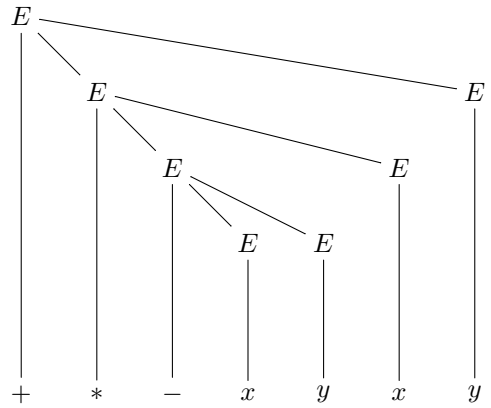
Exercise 2.6 (d) / 5.4.3 Idea: the production $S \to aSbS$ allows more than we want: the first S could be replaced with a string that has more a's then b's, then we could attach the b of this production also to an a produced by S, and that is exactly the ambiguity we want to avoid. Therefore, we want to restrict this first S to "well-balanced" strings, represented by the new symbol S_0:

$$\begin{array}{rcl} S & \to & aS \mid aS_0bS \mid \epsilon \\ S_0 & \to & aS_0bS_0 \mid \epsilon \end{array}$$

Exercise 2.7 (a) The derivation tree would be as follows, where, for brevity, we just draw the left-most and right-most derivation branches, and leave the rest underspecified with ...:

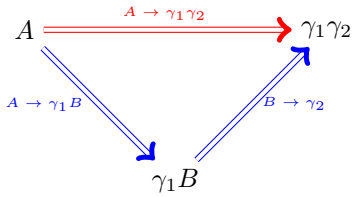


(b) The unique parse tree for $+*-xyxy$ is



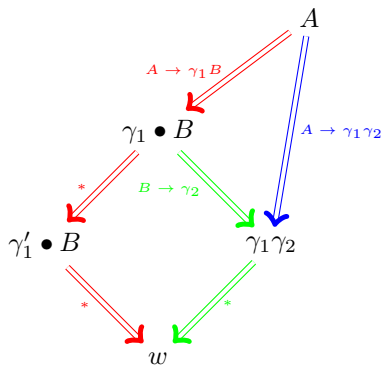
(c) The grammar is unambigous. To see this observe that for any string w generated by the grammar, and each possible initial terminal symbol of w, w can only be obtained by one production. This means that the left-derivation of E and any string derived by E is always unique, and corresponds hence to a unique parsing tree.

Exercise 2.8 To see that any word in $L(G)$ is also in $L(G')$ we observe that any derivation in G can be mimicked by G'. More specifically, any word generated by A in G can also be derived in G':



where $\Rightarrow$ is the challenge by G, and $\Rightarrow$ is how G mimicks the challenge.

The opposite direction (any word in $L(G')$ is also in $L(G)$) is a bit more tricky, but we can also observe that any derivation from A in G can be mimicked by G:



The left derivation in G (red) can be of course mimicked by the right derivation in G (green), which in turn can be mimicked by G' (blue).

CFL3 - Solution Sketches

February 28, 2023

We sketch here the solutions for the exercises of lecture CFL3 in a brief manner. Note that a proper solution would require more detailed descriptions, explanations, and in some cases examples. Some of the exercises may have more than one solution, and we just show one of them.

Exercise 3.1.(a) A functional datatype for Boolean expressions could look like this

```
type B = True
       | Or of (B * B)
       | Not of B
```

The expression `not ( true or true )` would be represented by the functional expression as follows

```
Not(Or(True,True))
```

Exercise 3.1.(b) A possible set of classes for storing Boolean expressions is sketched below:

```
public abstract class BoolExpr {};
public class TrueExpr extends BoolExpr {
    public TrueExpr(void) {};
}
public class OrExpr extends BoolExpr {
    private BoolExpr lhs;
    private BoolExpr rhs;
    public OrExpr(OrExpr x, OrExpr y) {
        lhs = x;
        rhs = y;
    }
}
public class NotExpr extends BoolExpr {
    private BoolExpr b;
    public NotExpr(BoolExpr x) {
        b = x;
    }
}
```

The expression `not ( true or true )` would be represented by the object:

```
BoolExpr b = new NotExpr(new OrExpr(new TrueExpr(),new TrueExpr()));
```

Exercise 3.3.(1) In the below solution we present just the part of the code that has been modified

```
// We now have 2 levels of expressions
%type <expr> expression0
%type <expr> expression1

// we start with level 0
start: expression0 EOF             { $1 }

// Level 0 less priority for exponentiation
expression0:
  | expression0 TIMES expression0   { TimesExpr($1,$3) }
  | expression0 DIV expression0     { DivExpr($1,$3) }
  | expression0 PLUS expression0    { PlusExpr($1,$3) }
  | expression0 MINUS expression0   { MinusExpr($1,$3) }
  | expression0 POW expression0     { PowExpr($1,$3) }
  | expression1                     { $1 }

// Level 1: more priority for unary minus
expression1:
  | PLUS expression1                { UPlusExpr($2) }
  | MINUS expression1               { UMinusExpr($2) }
  | NUM                             { Num($1) }
  | LPAR expression0 RPAR           { $2 }
```

Exercise 3.3.(2)

```
// These lines are now removed / commented-out
//%left PLUS MINUS
//%left TIMES DIV
//%right POW

// We now have 4 levels for expressions
%type <expr> expression0
%type <expr> expression1
%type <expr> expression2
%type <expr> expression3

// we start with level 0
start: expression0 EOF             { $1 }


// Now we have one level for each of the operator precedence levels (see GCL rules)
// Note how we deal with associativity by allowing recursion on one side only
expression0:
  | expression0 PLUS expression1    { PlusExpr($1,$3) }
  | expression0 MINUS expression1   { MinusExpr($1,$3) }
  | expression1                     { $1 }

expression1:
  | expression1 TIMES expression2   { TimesExpr($1,$3) }
  | expression1 DIV expression2     { DivExpr($1,$3) }
  | expression2                     { $1 }

expression2:
  | expression3 POW expression2     { PowExpr($1,$3) }
  | expression3                     { $1 }

expression3:
  | PLUS expression3               { UPlusExpr($2) }
  | MINUS expression3              { UMinusExpr($2) }
  | NUM                            { Num($1) }
  | LPAR expression0 RPAR          { $2 }
```

Solution

General remarks:

- Solutions will be published a day before the next class; feel free to ask questions during class after comparing the solution to yours.
- We strongly encourage you to ask for help and to request feedback about your solutions during the exercise classes.
- The difficulty of most exercises is comparable to typical exam tasks.

1 Greatest Common Divisor

1.1 Programming with Program Graphs

Specify a program graph **PG** modelling a function that computes the greatest common divisor of two numbers (given by variables n and m) and stores the result in a variable called `out`.

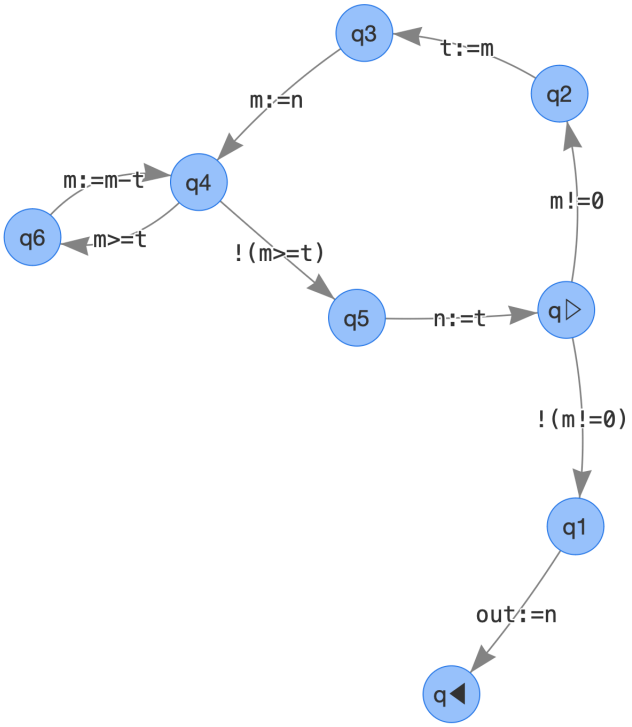
You may only use simple assignments, for example `x := x + y`, and simple Boolean tests, for example `x > 0`, as actions.

Hint: You may use the following pseudo-code for computing the greatest common divisor as inspiration. However, notice that you are not allowed to use the remainder ($a \bmod b$) in an action; you may want to write a program graph for the remainder first.

```
function gcd(a, b)
    while b != 0
        t := b
        b := a mod b
        a := t
    return a
```

Solution

A possible program graph is depicted below. Note that we implement the modulo operation in nodes q_4 and q_6 using a loop.



1.2 Semantic Functions

Define the semantic function $\mathcal{S}$ for all actions that you used in task 1.1.

Solution

$$\mathcal{S}[\![\texttt{m} \neq \texttt{0}]\!](\sigma) = \begin{cases} \sigma, & \text{if } \sigma(m) \neq 0 \\ \text{undefined}, & \text{if } \sigma(m) = 0 \end{cases}$$

$$\mathcal{S}[\![\neg(\texttt{m} \neq \texttt{0})]\!](\sigma) = \begin{cases} \sigma, & \text{if } \sigma(m) = 0 \\ \text{undefined}, & \text{if } \sigma(m) \neq 0 \end{cases}$$

$$\mathcal{S}[\![\texttt{m} \geq \texttt{t}]\!](\sigma) = \begin{cases} \sigma, & \text{if } \sigma(m) \geq \sigma(t) \\ \text{undefined}, & \text{if } \sigma(m) < \sigma(t) \end{cases}$$

$$\mathcal{S}[\![\neg(\texttt{m} \geq \texttt{t})]\!](\sigma) = \begin{cases} \sigma, & \text{if } \sigma(m) < \sigma(t) \\ \text{undefined}, & \text{if } \sigma(m) \geq \sigma(t) \end{cases}$$

$$\mathcal{S}[\![\texttt{t := m}]\!](\sigma) = \sigma[t \mapsto \sigma(m)]$$

$$\mathcal{S}[\![\texttt{m := n}]\!](\sigma) = \sigma[m \mapsto \sigma(n)]$$

$$\mathcal{S}[\![\texttt{m := m-t}]\!](\sigma) = \sigma[m \mapsto \sigma(m) - \sigma(t)]$$

$$\mathcal{S}[\![\texttt{n := t}]\!](\sigma) = \sigma[t \mapsto \sigma(t)]$$

$$\mathcal{S}[\![\texttt{out := n}]\!](\sigma) = \sigma[out \mapsto \sigma(n)]$$

1.3 Operational Semantics

Test whether your program graph indeed computes the greatest common divisor by computing a complete execution sequence

$$\langle q_{\triangleright}; \sigma \rangle \quad \Longrightarrow^* \langle q_{\blacktriangleleft}; \sigma' \rangle$$

with $\sigma(n) = 24$ and $\sigma(m) = 8$. What is the sequence ω of executed actions? What is σ'?

Solution

A detailed execution sequence is given by the table below. The sequence of executed actions ω is shown in the leftmost column (read from top to bottom). The final memory is given in the last row: $\sigma(m) = 0$, $\sigma(n) = 8$, $\sigma(t) = 8$ and $\sigma(out) = 8$.

Action	Node	m	n	t	out
	q▷	8	24	0	0
`m!=0` ⟹	q2	8	24	0	0
`t:=m` ⟹	q3	8	24	8	0
`m:=n` ⟹	q4	24	24	8	0
`m>=t` ⟹	q6	24	24	8	0
`m:=m-t` ⟹	q4	16	24	8	0
`m>=t` ⟹	q6	16	24	8	0
`m:=m-t` ⟹	q4	8	24	8	0
`m>=t` ⟹	q6	8	24	8	0
`m:=m-t` ⟹	q4	0	24	8	0
`!(m>=t)` ⟹	q5	0	24	8	0
`n:=t` ⟹	q▷	0	8	8	0
`!(m!=0)` ⟹	q1	0	8	8	0
`out:=n` ⟹	q◀	0	8	8	8

1.4 Properties of Program Graphs

1. Does your program graph and its semantic function constitute a deterministic system? Justify your answer.
2. Does your program graph and its semantic function constitute an evolving system? Justify your answer.

Solution

Yes, the program graph in the solution of 1.1 together with the semantic function in the solution of 1.2 constitutes both a deterministic and an evolving system. To check these properties, we use the sufficient criteria presented in the lecture (Propositions 1.22 and 1.25 in the FM book).

Deterministic system. By Proposition 1.22, it suffices to show that

$$\forall (q, \alpha_1, q_1), (q, \alpha_2, q_2) \in \mathbf{E}\colon \quad (\alpha_1, q_1) \neq (\alpha_2, q_2) \text{ implies } \mathsf{dom}(\mathcal{S}[\![\alpha_1]\!]) \cap \mathsf{dom}(\mathcal{S}[\![\alpha_2]\!]) = \emptyset.$$

Since the above criterion immediately holds for nodes with at most one outgoing edge, it suffices to consider the edges of two nodes: $q_\triangleright$ and q_4.

- There are two edges with source node $q_\triangleright$: $(q_\triangleright, \mathtt{m} \neq \mathtt{0}, q_2)$ and $(q_\triangleright, \neg(\mathtt{m} \neq \mathtt{0}), q_1)$. Computing the domains of the semantic functions for the actions of these actions yields:
  - $\mathsf{dom}(\mathcal{S}[\![\mathtt{m} \neq \mathtt{0}]\!]) = \{\sigma \mid \sigma(m) \neq 0\}$ and
  - $\mathsf{dom}(\mathcal{S}[\![\neg(\mathtt{m} \neq \mathtt{0})]\!]) = \{\sigma \mid \sigma(m) = 0\}$

  Clearly, $\mathsf{dom}(\mathcal{S}[\![\mathtt{m} \neq \mathtt{0}]\!]) \cap \mathsf{dom}(\mathcal{S}[\![\neg(\mathtt{m} \neq \mathtt{0})]\!]) = \emptyset$.
- There are two edges with source node q_4: $(q_4, \mathtt{m} \geq \mathtt{t}, q_6)$ and $(q_4, \neg(\mathtt{m} \geq \mathtt{t}), q_5)$. Computing the domains of the semantic functions for the actions of these actions yields:
  - $\mathsf{dom}(\mathcal{S}[\![\mathtt{m} \geq \mathtt{t}]\!]) = \{\sigma \mid \sigma(m) \neq \sigma(t)\}$ and
  - $\mathsf{dom}(\mathcal{S}[\![\neg(\mathtt{m} \geq \mathtt{t})]\!]) = \{\sigma \mid \sigma(m) = \sigma(t)\}$

  Clearly, $\mathsf{dom}(\mathcal{S}[\![\mathtt{m} \geq \mathtt{t}]\!]) \cap \mathsf{dom}(\mathcal{S}[\![\neg(\mathtt{m} \geq \mathtt{t})]\!]) = \emptyset$.

Hence, the system is deterministic.

Evolving system. By Proposition 1.25, it suffices to show that

$$\forall q \in \mathbf{Q} \setminus \{q_{\blacktriangleleft}\}\ \forall \sigma \in \mathbf{Mem}\ \exists (q, \alpha, q') \in \mathbf{E} \colon \sigma \in \mathsf{dom}(\mathcal{S}[\![\alpha]\!]).$$

We check the above criterion by computing the union of the domains of the actions of all outgoing edges for every node except the final one and checking whether this union equals the set of all memories. The following table summarizes these computations.

$q \in \mathbf{Q} \setminus \{q_{\blacktriangleleft}\}$	$\bigcup_{(q,\alpha,q') \in \mathbf{E}} \mathsf{dom}(\mathcal{S}[\![\alpha]\!])$	$= \mathbf{Mem}$?
$q_{\triangleright}$	$\{\sigma \mid \sigma(m) \neq 0\} \cup \{\sigma \mid \sigma(m) = 0\}$	✓
q_1	**Mem**	✓
q_2	**Mem**	✓
q_3	**Mem**	✓
q_4	$\{\sigma \mid \sigma(m) \neq \sigma(t)\} \cup \{\sigma \mid \sigma(m) = \sigma(t)\}$	✓
q_5	**Mem**	✓
q_6	**Mem**	✓

Since we obtain the full set of memories for every node except the final one, we conclude that, by Proposition 1.25, the system is evolving.

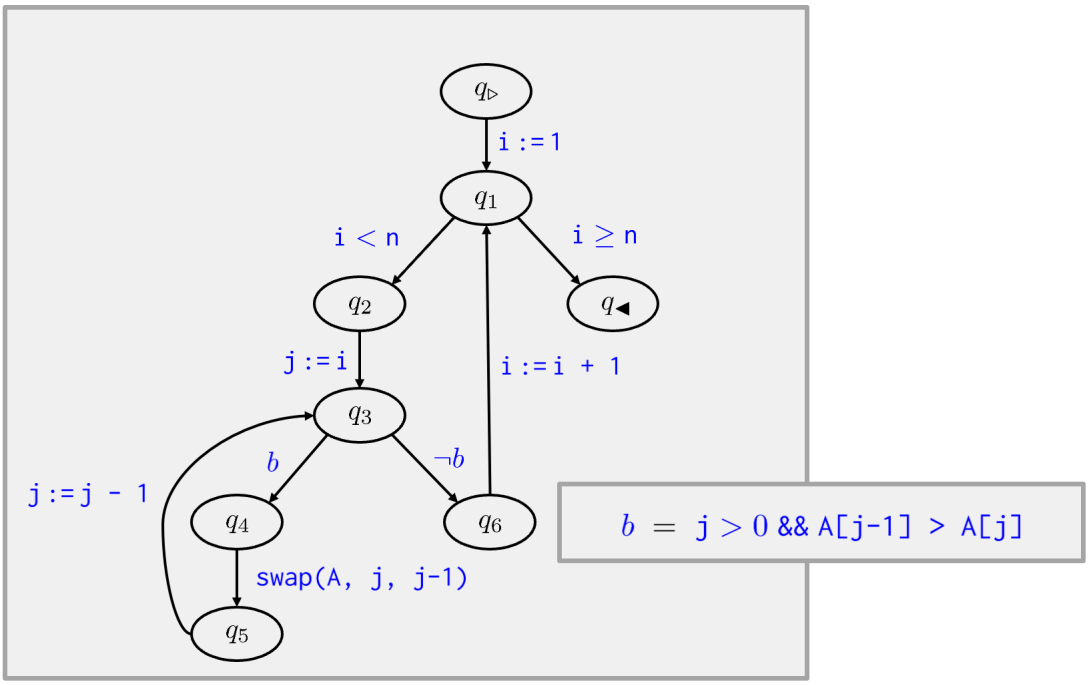
2 Semantics of Array Operations

Recall the program graph **PG** and most of its semantic function modelling insertion sort shown further below. Test whether **PG** indeed sorts a given array by computing a complete execution sequence

$$\langle q_{\triangleright}; \sigma \rangle \quad \Longrightarrow^{*} \langle q_{\blacktriangleleft}; \sigma' \rangle$$

for the initial memory displayed further below.

What is the sequence ω of executed actions? What is σ'?



$$\mathcal{S}[\![\texttt{j > 0 \&\& A[j-1] > A[j]}]\!](\sigma) = \begin{cases} \sigma, & \text{if } \sigma(\texttt{j}) > 0 \\ & \quad \text{and } \texttt{A}[\sigma(\texttt{j-1})] \in \text{dom}(\sigma) \\ & \quad \text{and } \texttt{A}[\sigma(\texttt{j})] \in \text{dom}(\sigma) \\ & \quad \text{and } \sigma(\texttt{A}[\sigma(\texttt{j-1})]) > \sigma(\texttt{A}[\sigma(\texttt{j})]) \\ \text{undefined}, & \text{otherwise} \end{cases}$$

$$\mathcal{S}[\![\texttt{swap(A, j, j-1)}]\!](\sigma) = \begin{cases} \sigma[\texttt{A[j]} \mapsto u][\texttt{A[j-1]} \mapsto v], & \text{if } \exists \texttt{z}: \ \sigma(\texttt{j}) = \texttt{z} \\ & \quad \text{and } \texttt{A[z-1]} \in \text{dom}(\sigma) \\ & \quad \text{and } \texttt{A[z]} \in \text{dom}(\sigma) \\ & \quad \text{and } u = \sigma(\texttt{A[z-1]}) \\ & \quad \text{and } v = \sigma(\texttt{A[z]}) \\ \text{undefined}, & \text{otherwise} \end{cases}$$

v	σ(v)
n	4
i	1
j	2
A[0]	4
A[1]	2
A[2]	17
A[3]	9

Solution

We use the same order for variables and array elements as in the table illustrating the initial memory. That is, we write σ as $(4, 1, 2, 4, 2, 17, 9)$.

$$
\begin{aligned}
&\langle q_{\triangleright}; (4,1,2,4,2,17,9)\rangle \\
\xRightarrow{\texttt{i := 1}} &\langle q_1; (4,1,2,4,2,17,9)\rangle \\
\xRightarrow{\texttt{i<n}} &\langle q_2; (4,1,2,4,2,17,9)\rangle \\
\xRightarrow{\texttt{j := i}} &\langle q_3; (4,1,1,4,2,17,9)\rangle \\
\xRightarrow{b} &\langle q_4; (4,1,1,4,2,17,9)\rangle \\
\xRightarrow{\texttt{swap(A, j, j-1)}} &\langle q_5; (4,1,1,2,4,17,9)\rangle \\
\xRightarrow{\texttt{j := j-1}} &\langle q_3; (4,1,0,2,4,17,9)\rangle \\
\xRightarrow{\neg b} &\langle q_6; (4,1,0,2,4,17,9)\rangle \\
\xRightarrow{\texttt{i := i+1}} &\langle q_1; (4,2,0,2,4,17,9)\rangle \\
\xRightarrow{\texttt{i<n}} &\langle q_2; (4,2,0,2,4,17,9)\rangle \\
\xRightarrow{\texttt{j := i}} &\langle q_3; (4,2,2,2,4,17,9)\rangle \\
\xRightarrow{\neg b} &\langle q_6; (4,2,2,2,4,17,9)\rangle \\
\xRightarrow{\texttt{i := i+1}} &\langle q_1; (4,3,2,2,4,17,9)\rangle \\
\xRightarrow{\texttt{i<n}} &\langle q_2; (4,3,2,2,4,17,9)\rangle \\
\xRightarrow{\texttt{j := i}} &\langle q_3; (4,3,3,2,4,17,9)\rangle \\
\xRightarrow{b} &\langle q_4; (4,3,3,2,4,17,9)\rangle \\
\xRightarrow{\texttt{swap(A, j, j-1)}} &\langle q_5; (4,3,3,2,4,9,17)\rangle \\
\xRightarrow{\texttt{j := j-1}} &\langle q_3; (4,3,2,2,4,9,17)\rangle \\
\xRightarrow{\neg b} &\langle q_6; (4,3,2,2,4,9,17)\rangle \\
\xRightarrow{\texttt{i := i+1}} &\langle q_1; (4,4,2,2,4,9,17)\rangle \\
\xRightarrow{\texttt{i>=n}} &\langle q_{\blacktriangleleft}; (4,1,2,4,2,17,9)\rangle
\end{aligned}
$$

3 Reasoning about Program Graphs

Recall the program graph **PG** that computes the factorial of x displayed below. Now that we have established precise semantics of program graphs based on execution sequences, we can check whether **PG** indeed computes the factorial.

Give a formal proof that, for every complete execution sequence

$$\langle q_{\triangleright}; \sigma \rangle \quad \Longrightarrow^* \langle q_{\blacktriangleleft}; \sigma' \rangle$$

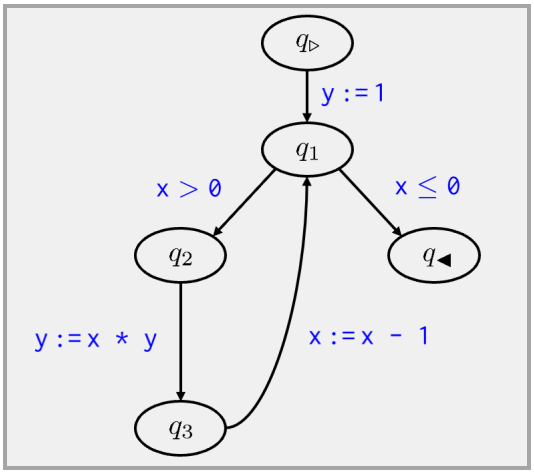
of **PG**, it holds that $\sigma'(\mathsf{y}) = \sigma(\mathsf{x})!$, that is, y evaluated in the final memory stores the factorial of x evaluated in the initial memory.

Hint: You may first want to show that, for all natural numbers n and all $0 < m \leq n$, we have

$$\langle\, q_1; \sigma[x \mapsto m][y \mapsto \prod_{i=m+1}^{n} i] \,\rangle \quad \Longrightarrow^* \langle\, q_1; \sigma[x \mapsto m-1][y \mapsto \prod_{i=m}^{n} i] \,\rangle,$$

where $\prod_{i=n+1}^{n} i = 1$, $\prod_{i=n}^{n} i = n$, and $\prod_{i=m}^{n} i = m * \prod_{i=m+1}^{n} i$. In particular, notice that $n! = \prod_{i=1}^{n} i$.

Moreover, you may assume that $n! = 1$ if $n \leq 0$.



$$\mathcal{S}[\![\,\texttt{y := 1}\,]\!](\sigma) \;=\; \sigma[\mathsf{y} \mapsto 1]$$

$$\mathcal{S}[\![\,\texttt{x > 0}\,]\!](\sigma) \;=\; \begin{cases} \sigma, & \text{if } \sigma(\mathsf{x}) > 0 \\ \text{undefined}, & \text{if } \sigma(\mathsf{x}) \leq 0 \end{cases}$$

$$\mathcal{S}[\![\,\texttt{x} \leq \texttt{0}\,]\!](\sigma) \;=\; \begin{cases} \sigma, & \text{if } \sigma(\mathsf{x}) \leq 0 \\ \text{undefined}, & \text{if } \sigma(\mathsf{x}) > 0 \end{cases}$$

$$\mathcal{S}[\![\,\texttt{x := x - 1}\,]\!](\sigma) \;=\; \sigma[\mathsf{x} \mapsto \sigma(\mathsf{x}) - 1]$$

$$\mathcal{S}[\![\,\texttt{y := x * y}\,]\!](\sigma) \;=\; \sigma[\mathsf{y} \mapsto \sigma(\mathsf{x}) * \sigma(\mathsf{y})]$$

Solution

We have to show that, for every complete execution sequence

$$\langle q_{\triangleright}; \sigma\rangle \quad \Longrightarrow^* \langle q_{\blacktriangleleft}; \sigma'\rangle$$

of **PG**, it holds that $\sigma'(\mathsf{y}) = \sigma(\mathsf{x})!$.

Let $\langle q_{\triangleright}; \sigma\rangle \quad \Longrightarrow^* \langle q_{\blacktriangleleft}; \sigma'\rangle$ be an arbitrary complete execution sequence for some initial memory σ with $\sigma(\mathsf{x}) = n$, where n is some arbitrary, but fixed, integer constant. Hence, $\sigma = \sigma[\mathsf{x} \mapsto n]$; we write the latter to highlight that we know the initial value of x. We consider two cases: $n \leq 0$ and $n > 0$.

Case 1. Assume $n \leq 0$. Then the only (since we consider a deterministic system) possible complete execution sequence is

$$\langle q_{\triangleright}; \sigma[\mathsf{x} \mapsto n]\rangle \overset{\mathtt{y := 1}}{\Longrightarrow} \langle q_1; \sigma[\mathsf{x} \mapsto n][y \mapsto 1]\rangle \overset{\mathtt{x \leq 0}}{\Longrightarrow} \langle q_{\blacktriangleleft}; \sigma[\mathsf{x} \mapsto n][y \mapsto 1]\rangle.$$

For the final memory $\sigma' = \sigma[\mathsf{x} \mapsto n][y \mapsto 1]$, we have

$$\sigma'(y) = 1 = n! = \sigma(\mathsf{x})!$$

since $n \leq 0$ (by assumption), $n! = 1$ (see hint), and $\sigma(\mathsf{x}) = n$ (by assumption).

Case 2. Assume $n > 0$. We will show that the only (since our system is deterministic) complete execution sequence always satisfies

$$\begin{aligned}
& \langle q_{\triangleright}; \sigma[\mathsf{x} \mapsto n]\rangle \\
\overset{\mathtt{y := 1}}{\Longrightarrow} \quad & \langle q_1; \sigma[\mathsf{x} \mapsto n][y \mapsto \prod_{i=n+1}^{n} i]\rangle && (\textstyle\prod_{i=n+1}^{n} i = 1) \\
\Longrightarrow^* \quad & \langle q_1; \sigma[\mathsf{x} \mapsto 0][y \mapsto \prod_{i=1}^{n} i]\rangle \\
\overset{\mathtt{x \leq 0}}{\Longrightarrow} \quad & \langle q_{\blacktriangleleft}; \sigma[\mathsf{x} \mapsto 0][y \mapsto \prod_{i=1}^{n} i]\rangle.
\end{aligned}$$

It then follows immediately that

$$\sigma'(y) \;=\; \sigma[\mathsf{x} \mapsto 0][y \mapsto \prod_{i=1}^{n} i](y) \;=\; \prod_{i=1}^{n} i \;=\; n! \;=\; \sigma(\mathsf{x})!.$$

The first and the last execution step (with action `y := 1` respectively `x < 0`) is immediate by definition. It remains to show that, for all $n > 0$, we have

$$\langle q_1; \sigma[\mathsf{x} \mapsto n][y \mapsto \prod_{i=n+1}^{n} i]\rangle \quad \Longrightarrow^* \quad \langle q_1; \sigma[\mathsf{x} \mapsto 0][y \mapsto \prod_{i=1}^{n} i]\rangle.$$

Let $n > 0$ be an arbitrary, but fixed, natural number. We show that the following more general statement holds for all $0 \leq m \leq n$:

$$P(m): \quad \langle q_1; \sigma[\mathsf{x} \mapsto m][y \mapsto \prod_{i=m+1}^{n} i]\rangle \quad \Longrightarrow^* \quad \langle q_1; \sigma[\mathsf{x} \mapsto 0][y \mapsto \prod_{i=1}^{n} i]\rangle.$$

By complete induction on m.

Induction base. For $P(0)$, there is nothing to show since our initial configuration is already of the form

$$\langle q_1; \sigma[\mathsf{x} \mapsto 0][y \mapsto \prod_{i=0+1}^{n} i]\rangle.$$

Induction hypothesis. Assume that $P(m)$ holds for an arbitrary, but fixed, natural number m with $0 \leq m < n$.

Induction step. We show that $P(m+1)$ holds, where $0 \leq m < n$. To this end, we construct the only possible (due to determinism) execution sequence starting with the given configuration as follows:

$$\begin{aligned}
& \langle q_1; \sigma[\mathsf{x} \mapsto m+1][y \mapsto \prod_{i=m+1+1}^{n} i]\rangle \\
\overset{\mathsf{x>0}}{\Longrightarrow} \quad & \langle q_2; \sigma[\mathsf{x} \mapsto m+1][y \mapsto \prod_{i=m+1+1}^{n} i]\rangle \\
\overset{\mathsf{y := x * y}}{\Longrightarrow} \quad & \langle q_3; \sigma[\mathsf{x} \mapsto m+1][y \mapsto \prod_{i=m+1}^{n} i]\rangle \\
& \qquad\qquad ((\textstyle\prod_{i=m+1+1}^{n} i) * (m+1) = \textstyle\prod_{i=m+1}^{n} i) \\
\overset{\mathsf{x := x-1}}{\Longrightarrow} \quad & \langle q_1; \sigma[\mathsf{x} \mapsto m][y \mapsto \prod_{i=m+1}^{n} i]\rangle \\
\Longrightarrow^* \quad & \langle q_1; \sigma[\mathsf{x} \mapsto 0][y \mapsto \prod_{i=1}^{n} i]\rangle. \qquad \text{(by I.H.)}
\end{aligned}$$

Hence, $P(m+1)$ holds. By the principle of complete induction, $P(m)$ holds for all $m \leq n$. Since n is an arbitrary natural number, this completes the proof. □

4 More on Semantics and Memories

Let A and B be arrays corresponding to two vectors of length n and m, respectively. Construct program graphs and define the semantic function for the involved actions for the following two operations:

1. the *inner* product, that is, the number defined by

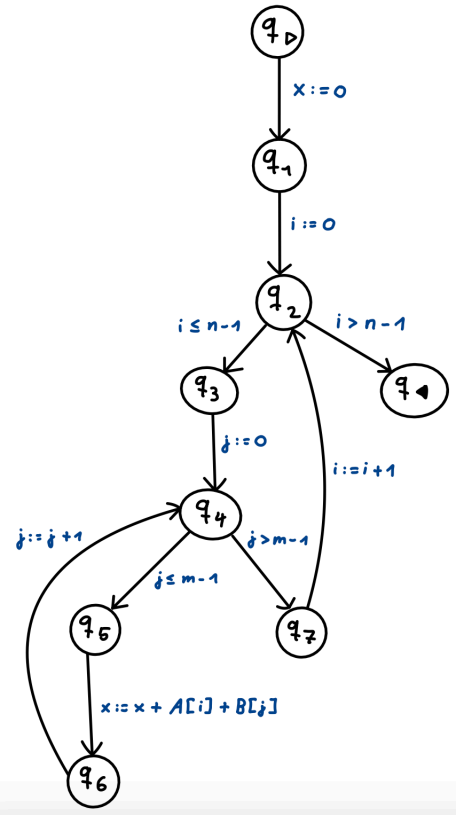
$$\sum_{i=0}^{n-1} \sum_{j=0}^{m-1} \mathtt{A}[i] \cdot \mathtt{B}[j]; \text{ and}$$

2. the *outer* product, that is, the $n \times m$ matrix C given by

$$\mathtt{C}[i,j] \;=\; \mathtt{A}[i] \cdot \mathtt{B}[j]$$

Hint: To define the semantic function of actions for the outer product, you also have to change the structure of memories, since memories now need to support matrices in addition to variables and arrays.

Solution 4.1



Act is the set of all actions (in blue) in the above program graph. The set of variables is **Var** $= \{\mathtt{x}, \mathtt{i}, \mathtt{j}, \mathtt{n}, \mathtt{m}\}$; the set of arrays is **Arr** $= \{A, B\}$. The set of memories is given by

$$\textbf{Mem} = (\textbf{Var} \cup \{Z[k] \mid Z \in \textbf{Arr}, 0 \leq k < \mathtt{length}(Z)\}) \ \rightarrow \ \textbf{Int}.$$

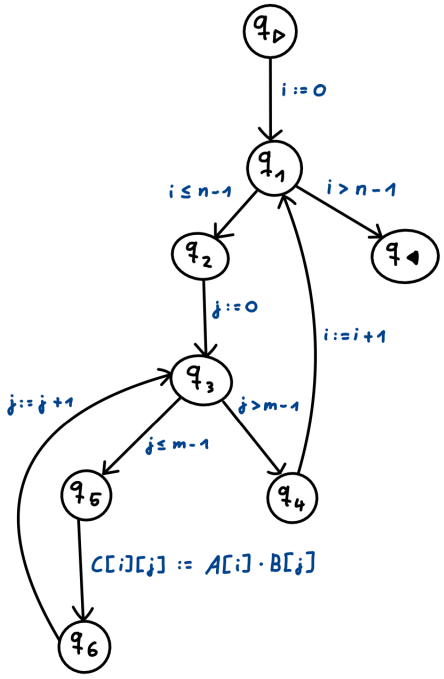
The semantics function

$$\mathcal{S}[\![.]\!] \colon \textbf{Act} \rightarrow (\textbf{Mem} \rightharpoonup \textbf{Mem})$$

is defined by the following table:

$\alpha \in$ **Act**	$\mathcal{S}[\![\alpha]\!](\sigma)$
`x := 0`	$\sigma[\mathtt{x} \mapsto 0]$
`i := 0`	$\sigma[\mathtt{i} \mapsto 0]$
`j := 0`	$\sigma[\mathtt{j} \mapsto 0]$
`i := i+1`	$\sigma[\mathtt{i} \mapsto \sigma(\mathtt{i}) + 1]$
`j := j+1`	$\sigma[\mathtt{j} \mapsto \sigma(\mathtt{j}) + 1]$
`i` $\leq$ `n-1`	$\begin{cases} \sigma, & \text{if } \sigma(\mathtt{i}) \leq \sigma(\mathtt{n}) - 1 \\ \text{undefined}, & \text{otherwise} \end{cases}$
`i` $>$ `n-1`	$\begin{cases} \sigma, & \text{if } \sigma(\mathtt{i}) > \sigma(\mathtt{n}) - 1 \\ \text{undefined}, & \text{otherwise} \end{cases}$
`j` $>$ `m-1`	$\begin{cases} \sigma, & \text{if } \sigma(\mathtt{j}) > \sigma(\mathtt{m}) - 1 \\ \text{undefined}, & \text{otherwise} \end{cases}$
`j` $\leq$ `m-1`	$\begin{cases} \sigma, & \text{if } \sigma(\mathtt{j}) \leq \sigma(\mathtt{m}) - 1 \\ \text{undefined}, & \text{otherwise} \end{cases}$
`x := x+A[i]+B[j]`	$\begin{cases} \sigma[\mathtt{x} \mapsto w], & \text{if } u = \sigma(\mathtt{i}), v = \sigma(\mathtt{j}), \\ & \quad \{A[u], B[v]\} \subseteq \mathsf{dom}(\sigma), \\ & \quad w = \sigma(\mathtt{x}) + \sigma(A[u]) + \sigma(B[v]) \\ \text{undefined}, & \text{otherwise} \end{cases}$

Solution 4.2



Act is the set of all actions (in blue) in the above program graph. The set of variables is **Var** $= \{\mathtt{i}, \mathtt{j}, \mathtt{n}, \mathtt{m}\}$; the set of arrays is **Arr** $= \{A, B\}$.

Additionally, the set of matrices is **Mat** $= \{C\}$.

We thus need to adapt the definition of the set of memories:

$$\begin{aligned}\textbf{Mem} \;= \big(&\textbf{Var} \cup \{Z[k] \mid Z \in \textbf{Arr}, 0 \leq k < \mathtt{length}(Z)\} \\ &\cup \{Z[k][\ell] \mid Z \in \textbf{Mat}, 0 \leq k < \mathtt{rows}(Z), 0 \leq k < \mathtt{cols}(Z)\}\big) \;\to\; \textbf{Int}.\end{aligned}$$

The semantics function

$$\mathcal{S}[\![.]\!] \colon \textbf{Act} \to (\textbf{Mem} \rightharpoonup \textbf{Mem})$$

is defined as in the table for task 4.1 except that we additionally define

$$\mathcal{S}[\![\texttt{C[i][j] := A[i] * B[j]}]\!](\sigma) = \begin{cases} \sigma[\mathtt{C}[u][v] \mapsto w], & \text{if } I = \sigma(\mathtt{i}), J = \sigma(\mathtt{j}), \\ & \quad \{A[I], B[j], C[I][J]\} \subseteq \mathsf{dom}(\sigma), \\ & \quad w = \sigma(A[I]) + \sigma(B[J]) \\ \text{undefined}, & \text{otherwise} \end{cases}$$

Solution

General remarks:

- We recommend that you first read all tasks during class. Make sure that you have a rough approach for every task in mind before you start working on the details. Ask for help if a task is unclear such that you do not get stuck at home.
- Solutions will be published a day before the next class; feel free to ask questions during next class after comparing the solution to yours.
- The difficulty of most exercises is comparable to typical exam tasks.

1 From Guarded Commands to Program Graphs

1. Write a program in the guarded command language that computes the greatest common divisor $\gcd(n, m)$ of two numbers n and m, which are initially stored in variables of the same name.
2. Construct an abstract syntax tree (AST) corresponding to your program.
3. Apply the algorithm from the lecture to construct the program graph of your program.
4. Compare the program graph constructed in (3) to the program graph that you constructed manually in the previous exercise (Ex. 01.1.1). Are there differences in the structure of the two program graph?
5. Compare the semantic function you defined in the previous exercise (Ex. 01.1.2) to the semantic function obtained for your program according to the recursive definition from the lecture (Chapter 2.3 in the FM book).

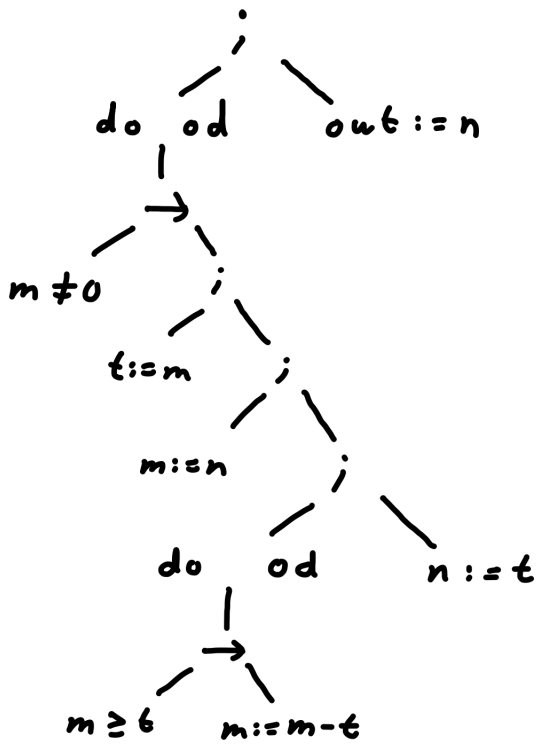
Solution

1.

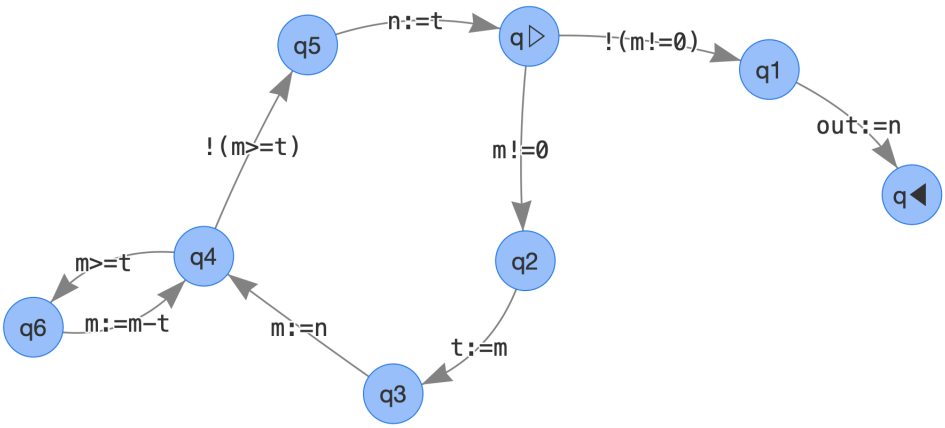
```
do m != 0 -> t := m;
             m := n;
             do m >= t -> m := m - t od;
             n := t
od;
out := n
```

Note: we write `m != 0` instead of $\neg(m = 0)$, `m >= t` instead of $m \geq t$, and `!b` instead of $\neg b$.

2. Notice that there are different versions of correct ASTs (we may rotate around inner nodes). One possible solution is as follows:



3. We need to compute **edges**$(q_{\triangleright} \rightsquigarrow q_{\blacktriangleleft})[\![\mathtt{C}]\!]$, where `C` is the Guarded Command program from above. The resulting program graph is as follows:



4 and 5. For the solution, there is no difference, since we already applied the algorithm last time. You may want to go back to your previous attempt and check whether the program graph and the semantics resulting from the formal definitions is the same as the ones you constructed based on your intuitive understanding.

2 More Expressions

1. Our arithmetic expressions a and Boolean expressions b are quite limited. Extend the syntax and semantics of expressions to also incorporate `false`, $a_1 \neq a_2$, $b_1 \vee b_2$, $a_1 \leq a_2$, $b_1 || b_2$, where $b_1 || b_2$ is a short-circuiting version of disjunction that does not evaluate b_2 if b_1 evaluates to `true`.

2. Do any of the added expressions increase the expressive power of the Boolean expressions or could we have managed without them?

Solution

1. We first extend the syntax by extending the BNF grammar defining b:

$$
\begin{array}{ll}
b ::= \ldots & \text{(all previous definitions from the book)} \\
\quad \mid \texttt{false} & \\
\quad \mid a_1 \neq a_2 & \\
\quad \mid b_1 \vee b_2 & \\
\quad \mid a_1 \leq a_2 & \\
\quad \mid b_1 \,||\, b_2 &
\end{array}
$$

Furthermore, we need to update the evaluation function $\mathcal{B}[\![b]\!]$ such that it incorporates the added atoms and composite expressions:

b	$\mathcal{B}[\![b]\!](\sigma)$	
$\texttt{false}$	$\texttt{false}$	
$a_1 \neq a_2$	$\texttt{true}$,	if $z_1 = \mathcal{A}[\![a_1]\!](\sigma), z_2 = \mathcal{A}[\![a_2]\!](\sigma), z_1 \neq z_2$
	$\texttt{false}$,	if $z_1 = \mathcal{A}[\![a_1]\!](\sigma), z_2 = \mathcal{A}[\![a_2]\!](\sigma), z_1 = z_2$
	$\texttt{undefined}$,	if $\mathcal{A}[\![a_1]\!](\sigma)$ or $\mathcal{A}[\![a_2]\!](\sigma)$ undefined
$b_1 \vee b_2$	$\texttt{true}$,	if $\mathcal{B}[\![b_1]\!](\sigma) = \texttt{true}$ or $\mathcal{B}[\![b_2]\!](\sigma) = \texttt{true}$
	$\texttt{false}$,	if $\mathcal{B}[\![b_1]\!](\sigma) = \texttt{false}$ and $\mathcal{B}[\![b_2]\!](\sigma) = \texttt{false}$
	$\texttt{undefined}$,	otherwise
$a_1 \leq a_2$	$\texttt{true}$,	if $z_1 = \mathcal{A}[\![a_1]\!](\sigma), z_2 = \mathcal{A}[\![a_2]\!](\sigma), z_1 \leq z_2$
	$\texttt{false}$,	if $z_1 = \mathcal{A}[\![a_1]\!](\sigma), z_2 = \mathcal{A}[\![a_2]\!](\sigma), z_1 > z_2$
	$\texttt{undefined}$,	otherwise
$b_1 \,\|\|\, b_2$	$\texttt{true}$,	if $\mathcal{B}[\![b_1]\!](\sigma) = \texttt{true}$
	$\mathcal{B}[\![b_2]\!](\sigma)$,	if $\mathcal{B}[\![b_1]\!](\sigma) = \texttt{false}$
	$\texttt{undefined}$,	otherwise

2. We could have managed without them because each of the added Boolean expressions can be defined as syntactic sugar in terms of the existing ones:

- $\texttt{false} \triangleq \neg\texttt{true}$
- $a_1 \neq a_2 \triangleq \neg(a_1 = a_2)$
- $b_1 \vee b_2 \triangleq \neg(\neg b_1 \wedge \neg b_2)$
- $a_1 \leq a_2 \triangleq \neg(a_1 > a_2)$
- $b_1 \,||\, b_2 \triangleq \neg(\neg b_1 \,||\, \neg b_2)$

3 Even More Expressions – Arrays

Incorporate operations for arrays into the Guarded Command Language. That is, extend the syntax and semantics of the Guarded Command language such that it supports assignment commands `A[a1] := a2` for writing to an array element and arithmetic expressions `A[a]` for reading the value of an array element.

Solution

Note: Arrays have not been covered before FM lecture 3.

We first extend the syntax:

$$\texttt{a} ::= \ldots \mid \texttt{A[a]}$$
$$\texttt{C} ::= \ldots \mid \texttt{A[a1] := a2}$$

To define the semantics function $\mathcal{S}[\![\texttt{A[a1] := a2}]\!]$ and the evaluation function $\mathcal{A}[\![\texttt{A[a]}]\!]$, we first incorporate arrays into our definition of memories. To this end, let **Arr** denote the set of arrays, and $\mathsf{size}(A)$ denote the size of array $A \in \textbf{Arr}$. The set of memories is then given by:

$$\textbf{Mem} \;=\; \{\sigma \mid \sigma \colon \textbf{Var} \cup \{A[i] \mid A \in \textbf{Arr}, 0 \leq i < \mathsf{size}(A) \to \textbf{Int}\}\}$$

$$\mathcal{S}[\![\texttt{A[a1] := a2}]\!](\sigma) \;=\; \begin{cases} \sigma[\texttt{A[u]} \mapsto v], & \text{if } u = \mathcal{A}[\![\texttt{a1}]\!](\sigma) \text{ defined}, \\ & \quad v = \mathcal{A}[\![\texttt{a2}]\!](\sigma) \text{ defined}, \\ & \quad \texttt{A[u]} \in \mathsf{dom}(\sigma) \\ \text{undefined}, & \text{otherwise} \end{cases}$$

$$\mathcal{A}[\![\texttt{A[a]}]\!](\sigma) \;=\; \begin{cases} \sigma(A[u]), & \text{if } u = \mathcal{A}[\![\texttt{a}]\!](\sigma) \text{ defined}, \texttt{A[u]} \in \mathsf{dom}(\sigma) \\ \text{undefined}, & \text{otherwise} \end{cases}$$

4 The Coincidence Property

If we want to define one set of memories for all Guarded Commands, then we need to choose an infinite set **Var** of variables. However, so far we always assumed that **Var** is the finite set of variables that actually appear in a program. The goal of this task is to check that the presence or absence of additional variables does not matter, at least not when evaluating arithmetic expressions.

1. Define a recursive function $\mathtt{FV}[\![a]\!]$, that takes an arithmetic expression a and returns the set of all variables that appear in a.

2. Let a be an arbitrary arithmetic expression as admitted by the syntax of the Guarded Command Language. Moreover, let σ and σ' be memories with $\mathsf{dom}(\sigma) = \mathtt{FV}[\![a]\!] \subseteq \mathsf{dom}(\sigma')$ such that, for all $x \in \mathtt{FV}[\![a]\!]$, we have $\sigma(x) = \sigma'(x)$. Prove that $\mathcal{A}[\![a]\!](\sigma) = \mathcal{A}[\![a]\!](\sigma')$.

Solution

1.

a	$\mathsf{FV}[\![a]\!]$
n	$\emptyset$
x	$\{x\}$
$a_1 \oplus a_2$	$\mathsf{FV}[\![a_1]\!] \cup \mathsf{FV}[\![a_2]\!]$, where $\oplus \in \{+, -, *, /, \hat{}\}$

2. Let a be an arbitrary arithmetic expression as admitted by the syntax of the Guarded Command Language. Moreover, let σ and σ' be memories with $\mathsf{dom}(\sigma) = \mathsf{FV}[\![a]\!] \subseteq \mathsf{dom}(\sigma')$ such that, for all $x \in \mathsf{FV}[\![a]\!]$, we have $\sigma(x) = \sigma'(x)$.

We show

$$P(a): \qquad \mathcal{A}[\![a]\!](\sigma) = \mathcal{A}[\![a]\!](\sigma')$$

by induction on the structure of arithmetic expressions a.

I.B. We need to show the base cases $P(n)$, where n is an integer constant, and $P(x)$, where x is a program variable.

For $P(n)$, we have

$$\mathcal{A}[\![n]\!](\sigma) \;=\; n \;=\; \mathcal{A}[\![n]\!](\sigma').$$

For $P(x)$, we have

$$\begin{aligned} & \mathcal{A}[\![x]\!](\sigma) \\ &= \sigma(x) \\ &= \sigma'(x) && \text{(by assumption)} \\ &= \mathcal{A}[\![x]\!](\sigma'). \end{aligned}$$

I.H. Assume $P(a_1)$ and $P(a_2)$ hold for arbitrary, but fixed arithmetic expressions a_1 and a_2.

I.S. We need to show that $P(a_1 \oplus a_2)$ holds, where $\oplus \in \{+, -, *, /, \hat{}\}$. To this end, we distinguish two cases. First, assume $\mathcal{A}[\![a_1]\!](\sigma)$ and $\mathcal{A}[\![a_2]\!](\sigma)$ are defined. Then, we have

$$\begin{aligned} & \mathcal{A}[\![a_1 \oplus a_2]\!](\sigma) \\ &= \mathcal{A}[\![a_1]\!](\sigma) \;\oplus\; \mathcal{A}[\![a_2]\!](\sigma) && (\mathcal{A}[\![a_1]\!](\sigma) \text{ and } \mathcal{A}[\![a_2]\!](\sigma) \text{ are defined}) \\ &= \mathcal{A}[\![a_1]\!](\sigma') \;\oplus\; \mathcal{A}[\![a_2]\!](\sigma') && \text{(by I.H.)} \\ &= \mathcal{A}[\![a_1 \oplus a_2]\!](\sigma'). \end{aligned}$$

Second, assume $\mathcal{A}[\![a_1]\!](\sigma)$ or $\mathcal{A}[\![a_2]\!](\sigma)$ is undefined. Without loss of generality, let's say $\mathcal{A}[\![a_1]\!](\sigma)$ is undefined. Then, we have

$$
\begin{aligned}
& \mathcal{A}[\![a_1 \oplus a_2]\!](\sigma) \\
&= \text{undefined} && (\mathcal{A}[\![a_1]\!](\sigma) \text{ undefined}) \\
&= \text{undefined} && (\mathcal{A}[\![a_1]\!](\sigma') \text{ undefined by I.H.}) \\
&= \mathcal{A}[\![a_1 \oplus a_2]\!](\sigma').
\end{aligned}
$$

□

General remarks:

- We recommend that you first read all tasks during class. Ask for help if a task is unclear such that you do not get stuck at home.
- The difficulty of most exercises is comparable to typical exam tasks.
- We use stars ranging from ★ (lowest) to ★★★★★ (highest) to indicate the relative difficulty of a task.

1 Constructing Program Proofs (★)

Use Floyd-Hoare Logic to construct program proofs for the following triples or explain why this is not possible.

(a) $\{\, 0 \leq \mathsf{x} \,\}\ x := \mathsf{x} + 1\,;\ x := \mathsf{x} + 1\ \{\, 2 \leq \mathsf{x} \,\}$

(b) $\{\, 0 \leq \mathsf{x} \,\}\ x := 3 * \mathsf{x} + 1\,;\ x := \mathsf{x} + 1\ \{\, 3 \leq \mathsf{x} \,\}$

(c) $\{\, \mathsf{x} < 2 \,\}\ y := \mathsf{x} + 5\,;\ x := 2 * \mathsf{x}\ \{\, \mathsf{x} < \mathsf{y} \,\}$

2 Checking Program Proofs (★★)

Explain all errors in the program proof below. Moreover, determine for every contract assigned to a sub-command whether it is valid or not.

$$
\begin{array}{l}
\{\, \mathsf{x} = 0 \,\} \\
\{\, \mathsf{x} = 0 \ \wedge\ \mathsf{y} = 6 \,\} \\
\mathsf{x} := \mathsf{x} + 2\,; \\
\{\, \mathsf{x} = 2 \ \wedge\ \mathsf{y} = 6 \,\} \\
\{\, \mathsf{x} + \mathsf{y} = 8 \,\} \\
\mathsf{y} := \mathsf{x} + \mathsf{y} \\
\{\, \mathsf{y} = 8 \,\}
\end{array}
$$

3 A Simplified Assignment Rule (★★)

Show the following useful specialised version of the assignment rule:
If x appears neither in P nor in a, then $\models \{\, P \,\}\ x := \mathsf{a}\ \{\, P \wedge \mathsf{x} = \mathsf{a} \,\}$.

4 Minimum and Maximum (★★★)

Use Floyd-Hoare Logic to construct a program proof for the following triple:

$$\{\, \mathsf{x} = \underline{\mathsf{x}} \;\wedge\; \mathsf{y} = \underline{\mathsf{y}} \,\}$$

```
if
        x < y →
                skip
  [ ]
        x ≥ y →
                z := x ;
                x := y ;
                y := z
fi
```

$$\{\, \mathsf{x} = \min(\underline{\mathsf{x}}, \underline{\mathsf{y}}) \;\wedge\; \mathsf{y} = \max(\underline{\mathsf{x}}, \underline{\mathsf{y}}) \,\}$$

5 The Goldbach Conjecture (★★★★)

The mathematician Goldbach conjectured that every even natural number $n > 2$ can be written as the sum of two prime numbers p and q - these two numbers are called a Goldbach partition of n.

(a) Write a predicate $\mathsf{prime}(x)$ which is satisfied iff x is a prime number; you may not introduce new functions.

(b) Give a contract $\{\, P \,\}$ _ $\{\, Q \,\}$ specifying programs that compute a Goldbach partition of the value given by program variable n.

(c) Assume we manage to implement a program C that complies with your contract. Would this prove Goldbach's conjecture? Justify your answer.

6 An Alternative Assignment Rule (★★★★★)

Prove that the following alternative rule for assignments is sound:

$$\models \; \{\, P[\mathsf{a} \,/\, \mathsf{x}] \,\}\; \mathsf{x} := \mathsf{a} \;\{\, P \,\}$$

General remarks:

- We recommend that you first read all tasks during class. Ask for help if a task is unclear such that you do not get stuck at home.
- The difficulty of most exercises is comparable to typical exam tasks.
- We use stars ranging from ★ (lowest) to ★★★★★ (highest) to indicate the relative difficulty of a task.

1 Constructing Program Proofs (★)

Use Floyd-Hoare Logic to construct program proofs for the following triples or explain why this is not possible.

(a) $\{\, 0 \leq \mathsf{x} \,\}\ x := \mathsf{x} + 1\,;\ x := \mathsf{x} + 1\ \{\, 2 \leq \mathsf{x} \,\}$

(b) $\{\, 0 \leq \mathsf{x} \,\}\ x := 3 * \mathsf{x} + 1\,;\ x := \mathsf{x} + 1\ \{\, 3 \leq \mathsf{x} \,\}$

(c) $\{\, \mathsf{x} < 2 \,\}\ y := \mathsf{x} + 5\,;\ x := 2 * \mathsf{x}\ \{\, \mathsf{x} < \mathsf{y} \,\}$

2 Checking Program Proofs (★★)

Explain all errors in the program proof below. Moreover, determine for every contract assigned to a sub-command whether it is valid or not.

$$
\begin{array}{l}
\{\, \mathsf{x} = 0 \,\} \\
\{\, \mathsf{x} = 0 \ \wedge\ \mathsf{y} = 6 \,\} \\
\mathsf{x} := \mathsf{x} + 2\,; \\
\{\, \mathsf{x} = 2 \ \wedge\ \mathsf{y} = 6 \,\} \\
\{\, \mathsf{x} + \mathsf{y} = 8 \,\} \\
\mathsf{y} := \mathsf{x} + \mathsf{y} \\
\{\, \mathsf{y} = 8 \,\}
\end{array}
$$

3 A Simplified Assignment Rule (★★)

Show the following useful specialised version of the assignment rule:
If x appears neither in P nor in a, then $\models \{\, P \,\}\ x := \mathsf{a}\ \{\, P \wedge \mathsf{x} = \mathsf{a} \,\}$.

4 Minimum and Maximum (★★★)

Use Floyd-Hoare Logic to construct a program proof for the following triple:

$$\{\, \mathsf{x} = \underline{\mathsf{x}} \;\wedge\; \mathsf{y} = \underline{\mathsf{y}} \,\}$$

```
if
        x < y →
                skip
  [ ]
        x ≥ y →
                z := x ;
                x := y ;
                y := z
fi
```

$$\{\, \mathsf{x} = \min(\underline{\mathsf{x}}, \underline{\mathsf{y}}) \;\wedge\; \mathsf{y} = \max(\underline{\mathsf{x}}, \underline{\mathsf{y}}) \,\}$$

5 The Goldbach Conjecture (★★★★)

The mathematician Goldbach conjectured that every even natural number $n > 2$ can be written as the sum of two prime numbers p and q - these two numbers are called a Goldbach partition of n.

(a) Write a predicate $\mathsf{prime}(x)$ which is satisfied iff x is a prime number; you may not introduce new functions.

(b) Give a contract $\{\, P \,\}$ _ $\{\, Q \,\}$ specifying programs that compute a Goldbach partition of the value given by program variable n.

(c) Assume we manage to implement a program C that complies with your contract. Would this prove Goldbach's conjecture? Justify your answer.

6 An Alternative Assignment Rule (★★★★★)

Prove that the following alternative rule for assignments is sound:

$$\models \; \{\, P[\mathsf{a} \,/\, \mathsf{x}] \,\}\; \mathsf{x} := \mathsf{a} \;\{\, P \,\}$$

General remarks:

- We recommend that you first read all tasks during class. Ask for help if a task is unclear such that you do not get stuck at home.
- The difficulty of most exercises is comparable to typical exam tasks.
- We use stars ranging from ★ (lowest) to ★★★★★ (highest) to indicate the relative difficulty of a task.

1 Verifying a Loop (★)

In class, we proposed the invariant

$$I \quad = \quad \underline{x} = z * y + x \wedge x \geq 0 \wedge y > 0 \wedge y = \underline{y}$$

for the triple

$$\{x \geq 0 \wedge y > 0 \wedge x = \underline{x} \wedge y = \underline{y}\}$$
$$z := 0;$$
do
$$\quad y \leq x \rightarrow$$
$$\quad\quad x := x - y;$$
$$\quad\quad z := z + 1$$
od
$$\{\exists \underline{z}\colon \underline{x} = \underline{z} * \underline{y} + x \wedge 0 \leq x \wedge x < \underline{y}\}$$

Use Floyd-Hoare logic and the above invariant I to complete the above triple to a correct program proof (where the triple is assigned to the whole program).

2 Verifying Loops (★★)

Consider the following Floyd-Hoare triple:

$$\{n \geq 0\}$$
$$z := 0;$$
$$x := 1;$$
do
$$\quad x \leq n \rightarrow$$
$$\quad\quad z := z + x;$$
$$\quad\quad x := x + 1$$
od
$$\{z = n * (n+1)/2\}$$

(a) Propose an invariant for the loop that is sufficient to prove the given triple.

(b) Use Floyd-Hoare logic to show that your proposal from (a) is indeed an invariant.

3 Verifying More Loops (★★★)

Use Floyd-Hoare logic to show that the program C below complies with the contract $\{y \geq 0 \land y = \underline{y}\}$ C $\{z = x \hat{} \underline{y}\}$. That is, construct a correct program proof, where C is assigned the triple $\{y \geq 0 \land y = \underline{y}\}$ C $\{z = x \hat{} \underline{y}\}$.

$\{y \geq 0 \land y = \underline{y}\}$
z := 1;
do
  $\neg(y = 0) \rightarrow$
    y := y − 1;
    z := z ∗ x
od
$\{z = x \hat{} \underline{y}\}$

Hints:

- Try to find an invariant first, similarly to the previous task.
- You can provide the whole program proof as an annotated program.

1 Sign Analysis via Abstract Semantics (★)

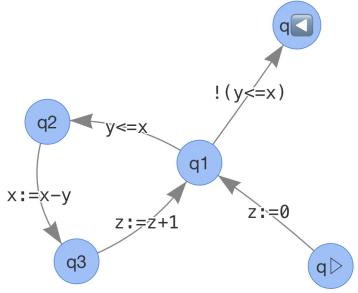
Recall from today's class the abstract semantics computing with signs. Use that abstract semantics to determine all abstract complete execution sequences of the following program, where the set of abstract initial memories is

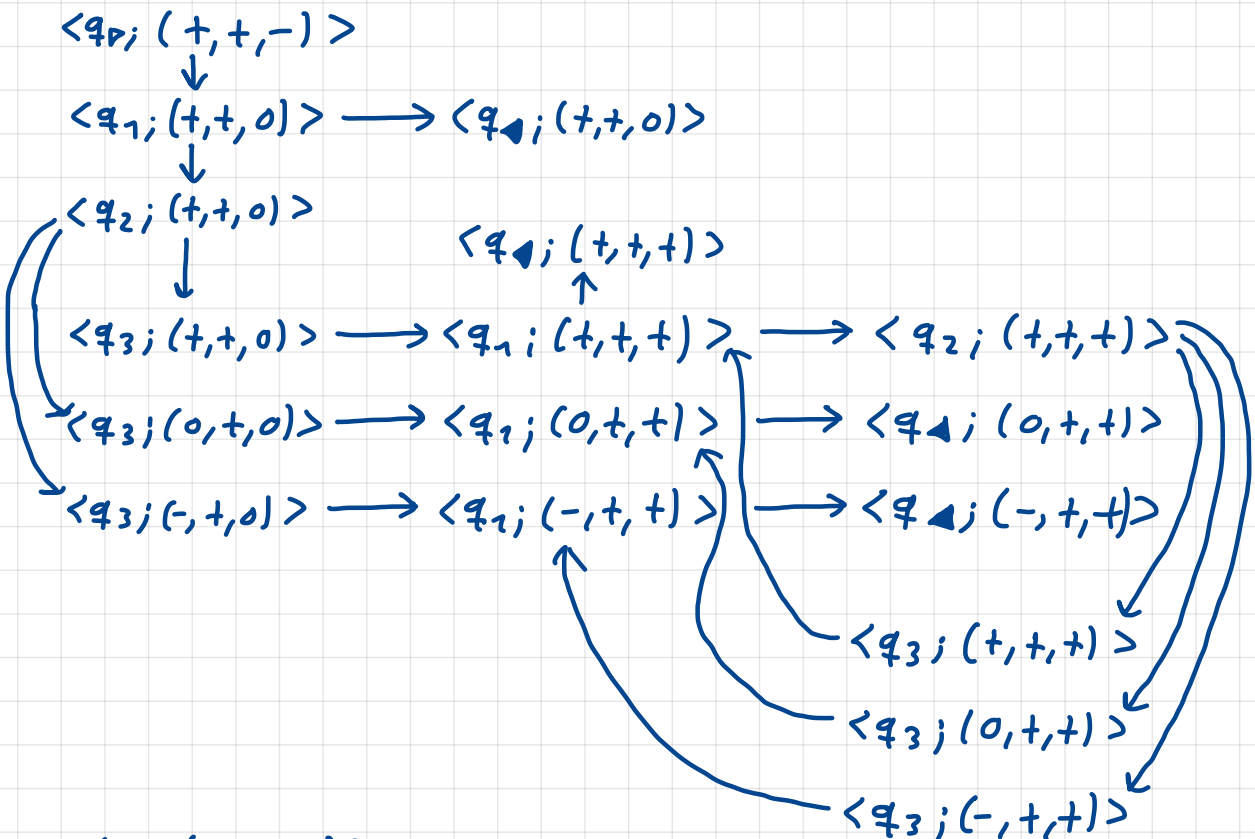
$$\mathbf{Mem}_\flat = \{\hat{\sigma}[x \mapsto +][y \mapsto +][z \mapsto -], \hat{\sigma}[x \mapsto -][y \mapsto -][z \mapsto +]\}.$$

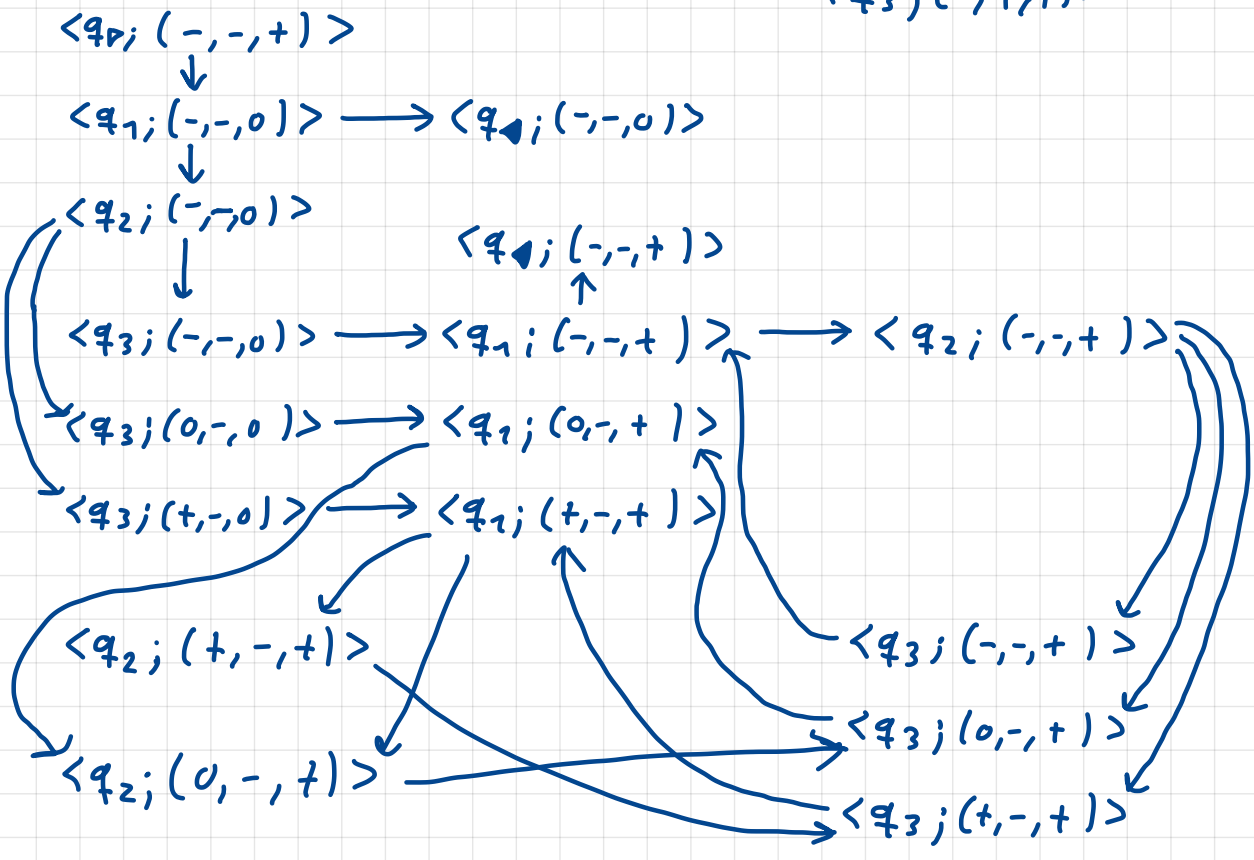
(+,+,-) (-,-,+)

```
z := 0;
do
    y ≤ x →
        x := x − y;
        z := z + 1
od
```

We denote by (a,b,c) the memory $\hat{\sigma}[x \mapsto a][y \mapsto b][z \mapsto c]$.







2 Abstract Evaluation of Boolean Expressions (★★)

(a) Complete the formal definition of the abstract evaluation function $\widehat{\mathcal{B}}$ for sign analysis.

Hint: [FM, p. 53] sketches what should be supported.

b	$\hat{B}[\![b]\!](\hat{\sigma})$
true	$\{tt\}$
a_1 op a_2 / $>, \geq, =$	$\bigcup_{s_1 \in \hat{A}[\![a_1]\!](\hat{\sigma}),\, s_2 \in \hat{A}[\![a_2]\!](\hat{\sigma})} s_1 \;\widehat{op}\; s_2$
$b_1 \wedge b_2$	$\bigcup_{t_1 \in \hat{B}[\![b_1]\!](\hat{\sigma}),\, t_2 \in \hat{B}[\![b_2]\!](\hat{\sigma})} t_1 \;\hat{\wedge}\; t_2$
b_1 && b_2	$(\hat{B}[\![b_1]\!](\hat{\sigma}) \cap \{ff\}) \cup \hat{B}[\![b_1 \wedge b_2]\!](\hat{\sigma})$
$\neg b$	$\bigcup_{t \in \hat{B}[\![b]\!](\hat{\sigma})} \{tt, ff\} \setminus \{t\}$

$s_1 > s_2$	$-$	0	$+$
$-$	$\{tt, ff\}$	$\{ff\}$	$\{ff\}$
0	$\{tt\}$	$\{ff\}$	$\{ff\}$
$+$	$\{tt\}$	$\{tt\}$	$\{tt, ff\}$

$s_1 \geq s_2$	$-$	0	$+$
$-$	$\{tt, ff\}$	$\{ff\}$	$\{ff\}$
0	$\{tt\}$	$\{tt\}$	$\{ff\}$
$+$	$\{tt\}$	$\{tt\}$	$\{tt, ff\}$

$s_1 = s_2$	$-$	0	$+$
$-$	$\{tt, ff\}$	$\{ff\}$	$\{ff\}$
0	$\{ff\}$	$\{tt\}$	$\{ff\}$
$+$	$\{ff\}$	$\{ff\}$	$\{tt, ff\}$

$t_1 \;\hat{\wedge}\; t_2$	ff	tt
ff	$\{ff\}$	$\{ff\}$
tt	$\{ff\}$	$\{tt\}$

(b) Show that the actual truth value of every evaluation of boolean expression b is contained in the truth values of its abstract evaluation function. That is, show that for every boolean expression b and every memory σ,

$$\mathcal{B}[\![\mathrm{b}]\!](\sigma) \in \widehat{\mathcal{B}}[\![\mathrm{b}]\!](\eta(\sigma)) \quad \text{whenever} \quad \mathcal{B}[\![\mathrm{b}]\!](\sigma) \text{ is defined.}$$

Hints:

- Recall that $\eta(\sigma)$ is the extraction function of our sign analysis.
- Proceed by structural induction on b.
- Recall that we know the following result for arithmetic expressions:

$$\textbf{sign}(\mathcal{A}[\![a]\!](\sigma)) \in \widehat{\mathcal{A}}[\![a]\!](\eta(\sigma)) \quad \text{whenever} \quad \mathcal{A}[\![a]\!](\sigma) \text{ is defined}$$

Proof by structural induction on the syntax of b.

I.B.

b = true:

$\hat{B}[\![\text{true}]\!](\eta(\sigma)) = \{tt\} \ni B[\![\text{true}]\!](\sigma)$.

b = a_1 op a_2 (where op $\in \{>, =, \geq\}$)

$$\hat{B}[\![a_1 \text{ op } a_2]\!](\eta(\sigma)) = \bigcup_{s_1 \in \hat{A}[\![a_1]\!](\eta(\sigma)),\, s_2 \in \hat{A}[\![a_2]\!](\eta(\sigma))} s_1 \,\tilde{op}\, s_2$$

$$\overset{\text{hint}}{\supseteq} \text{sign}(A[\![a_1]\!](\sigma)) \,\tilde{op}\, \text{sign}(A[\![a_2]\!](\sigma))$$

$$\overset{\text{def. of tables}}{\supseteq} \{A[\![a_1]\!](\sigma) \text{ op } A[\![a_2]\!](\sigma)\}$$

$$\ni B[\![a_1 \text{ op } a_2]\!](\sigma).$$

I.H. For arbitrary, but fixed b_1, b_2, we have

$B[\![b]\!](\sigma) \in \hat{B}[\![b]\!](\eta(\sigma))$ whenever $B[\![b]\!](\sigma)$ is defined.

I.S.

b = $\neg b_1$

$$\hat{B}[\![\neg b_1]\!](\eta(\sigma)) = \bigcup_{t \in \hat{B}[\![b]\!](\eta(\sigma))} \{tt, ff\} \setminus \{t\}$$

$$\overset{\text{I.H.}}{\supseteq} \{tt, ff\} \setminus \{B[\![b]\!](\sigma)\}$$

$$\ni B[\![\neg b]\!](\sigma).$$

$b = b_1 \wedge b_2$

$$\hat{B}[\![b_1 \wedge b_2]\!](\eta(\sigma))$$

$$= \bigcup_{t_1 \in \hat{B}[\![b_1]\!](\eta(\sigma)),\ t_2 \in \hat{B}[\![b_2]\!](\eta(\sigma))} t_1 \ \hat{\wedge}\ t_2$$

I.H.

$$\supseteq \quad B[\![b_1]\!](\sigma) \ \hat{\wedge}\ B[\![b_2]\!](\sigma)$$

construction of $\hat{\wedge}$

$$\ni \quad B[\![b_1 \wedge b_2]\!](\sigma).$$

$b = b_1 \,\&\&\, b_2$

$$\hat{B}[\![b_1 \,\&\&\, b_2]\!](\eta(\sigma))$$

$$= \left(\hat{B}[\![b_1]\!](\eta(\sigma)) \cap \{ff\}\right) \cup \hat{B}[\![b_1 \wedge b_2]\!](\eta(\sigma))$$

I.H.

$$\supseteq \left(\{B[\![b_1]\!](\sigma)\} \cap \{ff\}\right) \cup \hat{B}[\![b_1 \wedge b_2]\!](\eta(\sigma))$$

analogous to previous case

$$\supseteq \left(\{B[\![b_1]\!](\sigma)\} \cap \{ff\}\right) \cup B[\![b_1 \wedge b_2]\!](\sigma)$$

$$\ni B[\![b_1 \,\&\&\, b_2]\!](\sigma).$$

□

3 Parity Analysis (★★★)

In this task we will design a new program analysis that will determine, for each variable, the possible parities (i.e. *even* or *odd*) reachable at every program node of a given program graph. To this end:

(a) Define the finite set $\widehat{\textbf{Mem}}$ of abstract memories used for the new analysis.

(b) Define a new extraction function η mapping every concrete memory to an abstract memory.

(c) Define the abstract semantics $\widehat{\mathcal{S}}$ for the parity analysis, where the considered actions are assignments $x := a$ and guards $\mathtt{b}$.

Hint: You may also have to define abstract evaluation functions for arithmetic and boolean expressions. Soundness proofs are not required.

(d) Assume the set of initial abstract memories is given by

$$\textbf{Mem}_{\triangleright} = \{\hat{\sigma}[x \mapsto even][y \mapsto odd][z \mapsto odd]\}.$$

Use your abstract semantics to compute all abstract complete execution sequences for the following program:

```
z := 1;
do
    ¬(y = 0) →
        y := y − 1;
        z := z ∗ x
od
```

a) Let $Parity = \{even, odd\}$.

$\widehat{Mem} = Var \to Parity$

b) $\eta : Mem \to \widehat{Mem}$

is given by (for $x \in Var$)

$\eta(\sigma(x)) = parity(\sigma(x))$,

where

$$parity(n) = \begin{cases} even & \text{if } n \text{ even} \\ odd & \text{otherwise,} \end{cases}$$

c) $\hat{S}[\![b]\!](M) = \{\hat{\sigma} \mid \hat{\sigma} \in M \text{ and } tt \in \hat{B}[\![b]\!](\hat{\sigma})\}$

$\hat{S}[\![x := a]\!](M) = \{\hat{\sigma}[x \mapsto s] \mid \hat{\sigma} \in M \text{ and } s \in \hat{A}[\![a]\!](\hat{\sigma})\}$

$\hat{B}$ is defined as in exercise 2 except for the tables:

$s_1 > s_2$	even	odd
even	$\{tt, ff\}$	$\{tt, ff\}$
odd	$\{tt, ff\}$	$\{tt, ff\}$

$s_1 \geq s_2$	even	odd
even	$\{tt, ff\}$	$\{tt, ff\}$
odd	$\{tt, ff\}$	$\{tt, ff\}$

$s_1 = s_2$	even	odd
even	$\{tt, ff\}$	$\{ff\}$
odd	$\{ff\}$	$\{tt, ff\}$

$\hat{A}[\![a]\!](\hat{\sigma})$ is given by:

a	$\hat{A}[\![a]\!](\hat{\sigma})$
z	$\{parity(z)\}$
x	$\{\hat{\sigma}(x)\}$
a_1 op a_2 | $+, -, *, /$	$\bigcup$ $p_1 \in \hat{A}[\![a_1]\!](\hat{\sigma})$, $p_2 \in \hat{A}[\![a_2]\!](\hat{\sigma})$	p_1 $\hat{op}$ p_2 | given by tables below

$p_1 \hat{+} p_2$	even	odd
even	{even}	{odd}
odd	{odd}	{even}

$p_1 \hat{-} p_2$	even	odd
even	{even}	{odd}
odd	{odd}	{even}

$p_1 \hat{*} p_2$	even	odd
even	{even}	{even}
odd	{even}	{odd}

$p_1 \hat{/} p_2$	even	odd
even	{even}	{even, odd}
odd	{even, odd}	{even, odd}

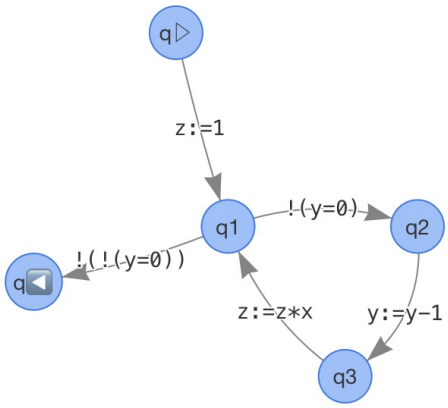
(d) Assume the set of initial abstract memories is given by

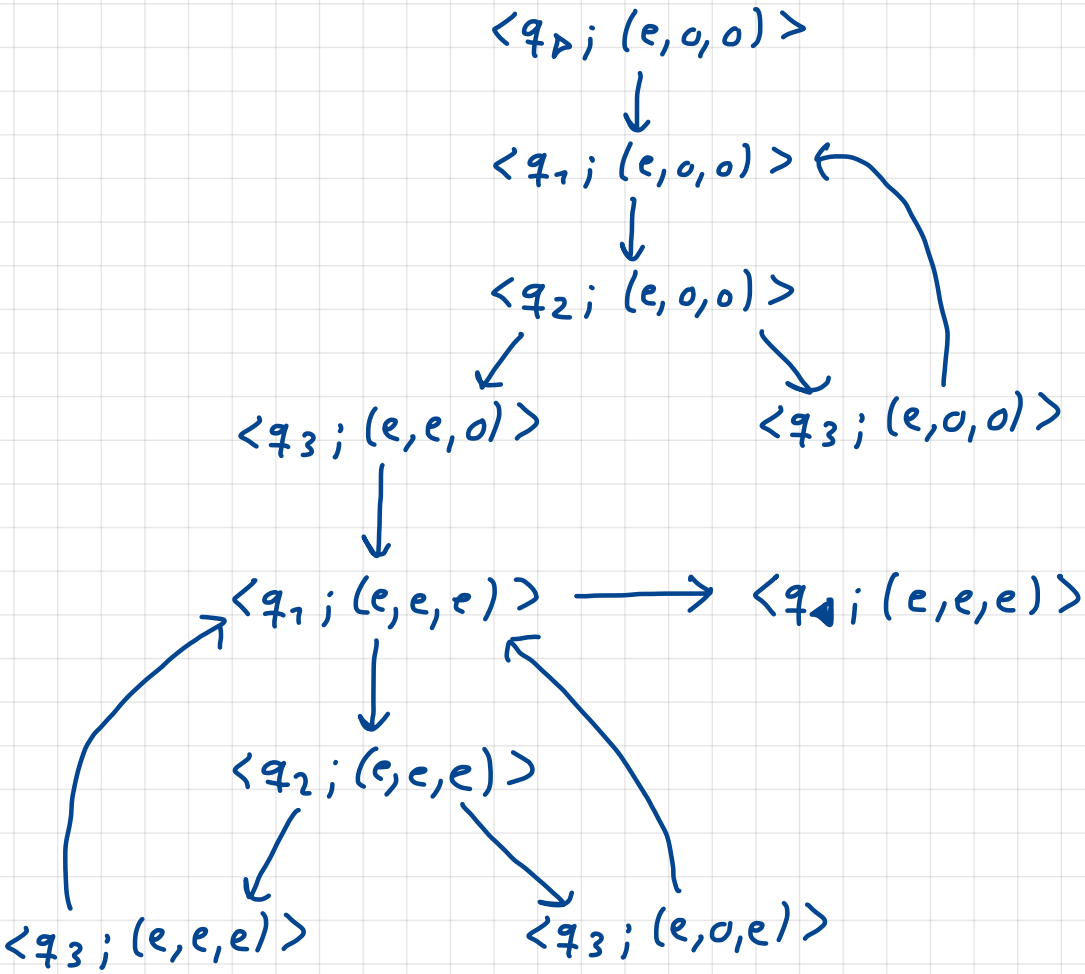
$$\mathbf{Mem}_{\triangleright} = \{\hat{\sigma}[x \mapsto even][y \mapsto odd][z \mapsto odd]\}.$$

denoted by (e,o,o).

Use your abstract semantics to compute all abstract complete execution sequences for the following program:

```
z := 1;
do
    ¬(y = 0) →
        y := y − 1;
        z := z ∗ x
od
```





4 Soundness (★★★★)

In class (and in [FM, Def. 4.23]), we have defined that an abstract semantic function $\widehat{\mathcal{S}}$ is *sound* with respect to a semantics function $\mathcal{S}$ and an extraction function η iff for all actions α, memories σ, σ' and sets of memories M:

$$\text{if } \sigma' = \mathcal{S}[\![\alpha]\!](\sigma) \quad \text{and} \quad \eta(\sigma) \in M \qquad \text{then} \qquad \eta(\sigma') \in \widehat{\mathcal{S}}[\![\alpha]\!](M).$$

Alternatively, we could say that $\widehat{\mathcal{S}}$ is *sound* iff

$$\{\eta(\mathcal{S}[\![\alpha]\!](\sigma))\} \quad \subseteq \quad \widehat{\mathcal{S}}[\![\alpha]\!](\{\eta(\sigma)\}),$$

where $\{\eta(\mathcal{S}[\![\alpha]\!](\sigma))\} = \emptyset$ if $\mathcal{S}[\![\alpha]\!](\sigma)$ is undefined.

Determine whether these two definitions are equivalent or not.

Claim: Yes, the two definitions are equivalent.

Proof:

"⇒" Let $\sigma' = \mathcal{S}[\![\alpha]\!](\sigma)$.

By assumption, $\eta(\sigma') \in \hat{\mathcal{S}}[\![\alpha]\!](M)$ for $M = \{\eta(\sigma)\}$.

By definition of σ' and M,

$$\{\eta(\underbrace{\mathcal{S}[\![\alpha]\!](\sigma)}_{=\sigma'})\} \subseteq \hat{\mathcal{S}}[\![\alpha]\!](\underbrace{\{\eta(\sigma)\}}_{=M}).$$

"⇐" Assume $\{\eta(\mathcal{S}[\![\alpha]\!](\sigma))\} \subseteq \hat{\mathcal{S}}[\![\alpha]\!](\{\eta(\sigma)\})$.

Moreover, let $\sigma' = \mathcal{S}[\![\alpha]\!](\sigma)$ and $\eta(\sigma) \in M$;

otherwise, there is nothing to show.

Then $\{\eta(\sigma)\} \subseteq M$.

By monotonicity of $\hat{\mathcal{S}}[\![\alpha]\!]$ and our assumption,

$$\{\eta(\underbrace{\mathcal{S}[\![\alpha]\!](\sigma)}_{=\sigma'})\} \subseteq \hat{\mathcal{S}}[\![\alpha]\!](\{\eta(\sigma)\}) \subseteq \hat{\mathcal{S}}[\![\alpha]\!](M).$$

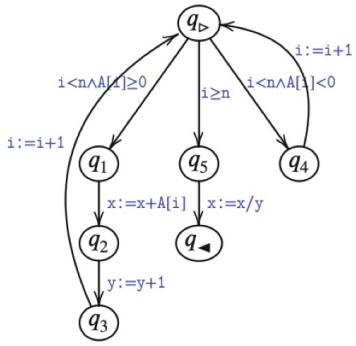
Hence, $\eta(\sigma') \in \hat{\mathcal{S}}[\![\alpha]\!](M)$. □

1 Constraint Solutions (★)

Consider the set of initial abstract memories $\mathbf{Mem}_{\triangleright}$ given by

$$\mathbf{Mem}_{\triangleright} = \{\hat{\sigma}[x \mapsto 0][y \mapsto 0][i \mapsto 0][n \mapsto +][A \mapsto \{+\}]\}.$$

Propose an analysis assignment **A** that is solution for the constraints obtained from the program graph below. Justify your answer.



$$\begin{aligned}
\hat{S}[\![\mathtt{i} < \mathtt{n} \wedge \mathtt{A[i]} \geq 0]\!](\mathbf{A}(q_{\triangleright})) &\subseteq \mathbf{A}(q_1)\\
\hat{S}[\![\mathtt{i} < \mathtt{n} \wedge \mathtt{A[i]} < 0]\!](\mathbf{A}(q_{\triangleright})) &\subseteq \mathbf{A}(q_4)\\
\hat{S}[\![\mathtt{i} \geq \mathtt{n}]\!](\mathbf{A}(q_{\triangleright})) &\subseteq \mathbf{A}(q_5)\\
\hat{S}[\![\mathtt{x} := \mathtt{x} + \mathtt{A[i]}]\!](\mathbf{A}(q_1)) &\subseteq \mathbf{A}(q_2)\\
\hat{S}[\![\mathtt{y} := \mathtt{y} + 1]\!](\mathbf{A}(q_2)) &\subseteq \mathbf{A}(q_3)\\
\hat{S}[\![\mathtt{i} := \mathtt{i} + 1]\!](\mathbf{A}(q_3)) &\subseteq \mathbf{A}(q_{\triangleright})\\
\hat{S}[\![\mathtt{i} := \mathtt{i} + 1]\!](\mathbf{A}(q_4)) &\subseteq \mathbf{A}(q_{\triangleright})\\
\hat{S}[\![\mathtt{x} := \mathtt{x}/\mathtt{y}]\!](\mathbf{A}(q_5)) &\subseteq \mathbf{A}(q_{\blacktriangleleft})
\end{aligned}$$

Trivial solution: $A(q) = \widehat{Mem}$ for all $q \in Q$.

More useful solution (can e.g. be computed via chaotic iteration):

We write $(0,0,0,+,\{+\})$ instead of $\hat{\sigma}[x \mapsto 0][y \mapsto 0][i \mapsto 0][n \mapsto +][A \mapsto \{+\}]$

q	$A(q)$
$q_{\triangleright}$	$\{(0,0,0,+,\{+\}), (+,+,+,+,\{+\})\}$
q_1	$\{(0,0,0,+,\{+\}, (+,+,+,+,\{+\})\}$
q_2	$\{(+,0,0,+,\{+\}), (+,+,+,+,\{+\})\}$
q_3	$\{(+,+,0,+,\{+\}), (+,+,+,+,\{+\})\}$
q_4	$\emptyset$
q_5	$\{(+,+,+,+,\{+\})\}$
$q_{\blacktriangleleft}$	$\{(+,+,+,+,\{+\}), (0,+,+,+,\{+\})\}$

This solution satisfies all constraints in the task.

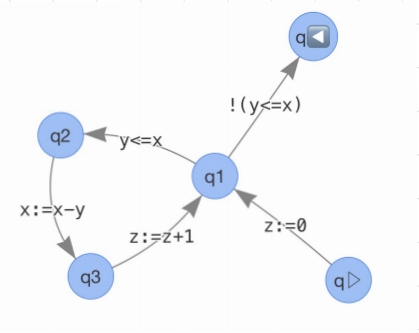
2 Sign Analysis via Constraints (★★)

Use the constraint solving approach to perform a sign analysis for the program and the set of initial memories below. To this end,

1. Give the system of constraints.
2. Compute a solution using the chaotic iteration algorithm.

$\mathbf{Mem}_{\triangleright} = \{\hat{\sigma}[x \mapsto +][y \mapsto +][z \mapsto -], \hat{\sigma}[x \mapsto -][y \mapsto -][z \mapsto +]\}$.

```
z := 0;
do
    y ≤ x →
        x := x − y; z := z + 1
od
```



1) $\widehat{Mem}_{\triangleright} \subseteq A(q_{\triangleright})$

$\hat{S}[\![z := 0]\!](A(q_{\triangleright})) \subseteq A(q_1)$

$\hat{S}[\![y \leq x]\!](A(q_1)) \subseteq A(q_2)$

$\hat{S}[\![\neg(y \leq x)]\!](A(q_1)) \subseteq A(q_4)$

$\hat{S}[\![x := x - y]\!](A(q_2)) \subseteq A(q_3)$

$\hat{S}[\![z := z + 1]\!](A(q_3)) \subseteq A(q_1)$

2) We write $(+,+,-]$ instead of $\hat{\sigma}[x \mapsto +][y \mapsto +][z \mapsto -]$.

Initialization:

q	$A(q)$
$q_\triangleright$	$\{(+,+,-),(-,-,+)\}$
q_1	$\varnothing$
q_2	$\varnothing$
q_3	$\varnothing$
$q_\triangleleft$	$\varnothing$

Step 1: pick edge $(q_\triangleright, z:=0, q_1)$.

$$\hat{S}[\![z:=0]\!](A(q_\triangleright)) = \{(+,+,0),(-,-,0)\}$$

$$\not\subseteq \varnothing = A(q_1)$$

q	$A(q)$
$q_\triangleright$	$\{(+,+,-),(-,-,+)\}$
q_1	$\{(+,+,0),(-,-,0)\}$
q_2	$\varnothing$
q_3	$\varnothing$
$q_\triangleleft$	$\varnothing$

Step 2: pick edge $(q_1, y \le x, q_2)$.

$\hat{S}[\![y \le x]\!](A(q_1))$

$= \{(+,+,0), (-,-,0)\}$

$\not\subseteq \varnothing = A(q_2)$

q	$A(q)$
$q_\triangleright$	$\{(+,+,-), (-,-,+)\}$
q_1	$\{(+,+,0), (-,-,0)\}$
q_2	$\{(+,+,0), (-,-,0)\}$
q_3	$\varnothing$
$q_\blacktriangleleft$	$\varnothing$

Step 3: pick edge $(q_2, x := x-y, q_3)$.

$\hat{S}[\![x := x-y]\!](A(q_2))$

$= \{(-,+,0), (0,+,0),$
$(+,+,0), (-,-,0),$
$(0,-,0), (+,-,0)\}$

$\not\subseteq \varnothing = A(q_3)$

q	$A(q)$
$q_\triangleright$	$\{(+,+,-), (-,-,+)\}$
q_1	$\{(+,+,0), (-,-,0)\}$
q_2	$\{(+,+,0), (-,-,0)\}$
q_3	$\{(-,+,0), (0,+,0), (+,+,0), (-,-,0),$ $(0,-,0) (+,-,0)\}$
$q_\blacktriangleleft$	$\varnothing$

Step 4: pick edge $(q_3, z := z+1, q_1)$.

$\hat{S}[\![z := z+1]\!](A(q_3))$

$= \{(-,+,+), (0,+,+),$
$(+,+,+), (-,-,+),$
$(0,-,+), (+,-,+)\}$

$\not\subseteq \{(+,+,0), (-,-,0)\}$

$= A(q_1)$

q	$A(q)$
$q_\triangleright$	$\{(+,+,-), (-,-,+)\}$
q_1	$\{(+,+,0), (-,-,0),$ $(-,+,+), (0,+,+), (+,+,+), (-,-,+),$ $(0,-,+) (+,-,+)\}$
q_2	$\{(+,+,0), (-,-,0)\}$
q_3	$\{(-,+,0), (0,+,0), (+,+,0), (-,-,0),$ $(0,-,0) (+,-,0)\}$
$q_\blacktriangleleft$	$\varnothing$

Step 5: pick edge $(q_1, y \le x, q_2)$.

$\hat{S}[\![y \le x]\!](A(q_1))$

$= \{(+,+,0), (-,-,0),$
$(+,+,+), (-,-,+),$
$(0,-,+), (+,-,+)\}$

$\notin \{(+,+,0), (-,-,0)\}$

$= A(q_2)$

q	$A(q)$
$q_\triangleright$	$\{(+,+,-), (-,-,+)\}$
q_1	$\{(+,+,0), (-,-,0),$ $(-,+,+), (0,+,+), (+,+,+), (-,-,+),$ $(0,-,+) (+,-,+)\}$
q_2	$\{(+,+,0), (-,-,0), (+,+,+),$ $(-,-,+), (0,-,+), (+,-,+)\}$
q_3	$\{(-,+,0), (0,+,0), (+,+,0), (-,-,0),$ $(0,-,0) (+,-,0)\}$
$q_\blacktriangleleft$	$\emptyset$

Step 6: pick edge $(q_2, x := x-y, q_3)$.

$\hat{S}[\![x := x-y]\!](A(q_2))$

$= \{(+,+,0), (0,+,0), (-,+,0),$
$(+,-,0), (0,-,0), (-,-,0),$
$(+,+,+), (0,+,+), (-,+,+),$
$(+,-,+), (0,-,+), (-,-,+)\}$

$\notin A(q_3)$

q	$A(q)$
$q_\triangleright$	$\{(+,+,-), (-,-,+)\}$
q_1	$\{(+,+,0), (-,-,0),$ $(-,+,+), (0,+,+), (+,+,+), (-,-,+),$ $(0,-,+) (+,-,+)\}$
q_2	$\{(+,+,0), (-,-,0), (+,+,+),$ $(-,-,+), (0,-,+), (+,-,+)\}$
q_3	$\{(+,+,0), (0,+,0), (-,+,0),$ $(+,-,0), (0,-,0), (-,-,0),$ $(+,+,+), (0,+,+), (-,+,+),$ $(+,-,+), (0,-,+), (-,-,+)\}$
$q_\blacktriangleleft$	$\emptyset$

Step 7: pick edge $(q_1, \neg(y \le x), q_\blacktriangleleft)$.

$\hat{S}[\![\neg(y \le x)]\!](A(q_1))$

$= \{(+,+,0), (-,-,0), (-,+,+), (0,+,+), (+,+,+), (-,-,+)\}$

$\not\subseteq \emptyset = A(q_\blacktriangleleft)$

q	$A(q)$
$q_\blacktriangleright$	$\{(+,+,-), (-,-,+)\}$
q_1	$\{(+,+,0), (-,-,0), (-,+,+), (0,+,+), (+,+,+), (-,-,+), (0,-,+), (+,-,+)\}$
q_2	$\{(+,+,0), (-,-,0), (+,+,+), (-,-,+), (0,-,+), (+,-,+)\}$
q_3	$\{(+,+,0), (0,+,0), (-,+,0), (+,-,0), (0,-,0), (-,-,0), (+,+,+), (0,+,+), (-,+,+), (+,-,+), (0,-,+), (-,-,+)\}$
$q_\blacktriangleleft$	$\{(+,+,0), (-,-,0), (-,+,+), (0,+,+), (+,+,+), (-,-,+)\}$

$\rightsquigarrow$ stabilized, A is a solution.

3 Monotone Analysis Functions (★★★)

Argue why the analysis functions $\hat{S}$ in Section 4.3 of [FM] are monotone.

Note that each of the analysis functions in [FM] is of the form

$$\hat{S}[\![\alpha]\!](M) = \{\, \hat{\sigma}' \mid \hat{\sigma} \in M \text{ and } P_\alpha(\hat{\sigma}, \hat{\sigma}') \,\}$$

where $P_\alpha(\hat{\sigma}, \hat{\sigma}')$ is a predicate that determines whether $\hat{\sigma}'$ should be amongst the abstract memories obtained from running α on $\hat{\sigma}$.

Now, assume $M \subseteq N$.

Then

$$\begin{aligned}
\hat{S}[\![\alpha]\!](M) &= \{\, \hat{\sigma}' \mid \hat{\sigma} \in M \text{ and } P_\alpha(\hat{\sigma}, \hat{\sigma}') \,\} \\
&\subseteq \{\, \hat{\sigma}' \mid \hat{\sigma} \in N \text{ and } P_\alpha(\hat{\sigma}, \hat{\sigma}') \,\} \\
&= \hat{S}[\![\alpha]\!](N).
\end{aligned}$$

Hence, $\hat{S}[\![\alpha]\!]$ is monotone.

FM5 - Exercises and solutions

Exercise 5.a. Actual flows As part of the mandatory assignment you will have to compute the actual flows of a given guarded command program c. In this exercise you are asked to formally define a function actualFlows that, given a guarded command c provides the actual information flows of c according to the notion of information flows seen in the course. You should define the function recursively on the syntactic structure of guarded commands.

We will denote an actual flow between x and y with $x \rightarrow_a y$. A first attempt on definining actualFlows could look like this:

$$\begin{array}{rcl}
\mathsf{actualFlows}(\texttt{skip}) & = & \emptyset \\
\mathsf{actualFlows}(x{:=}e) & = & \{x\} \overset{\rightarrow}{_a} fv(e) \\
\mathsf{actualFlows}(A[e_1]{:=}e_2) & = & (fv(e_1) \cup fv(e_2)) \overset{\rightarrow}{_a} A \\
\mathsf{actualFlows}(c_1;c_2) & = & \mathsf{actualFlows}(c_1) \cup \mathsf{actualFlows}(c_2) \\
\mathsf{actualFlows}(\texttt{if } gc \texttt{ fi}) & = & \mathsf{actualFlows}(gc) \\
\mathsf{actualFlows}(\texttt{do } gc \texttt{ od}) & = & \mathsf{actualFlows}(gc) \\
\mathsf{actualFlows}(b \texttt{ -> } c) & = & \mathsf{actualFlows}(c) \\
\mathsf{actualFlows}(gc_1 [] gc_2) & = & \mathsf{actualFlows}(gc_1) \cup \mathsf{actualFlows}(gc_2)
\end{array}$$

where $\overset{\rightarrow}{_a}$ is an operation similar to the flow relation of Definition 5.1, which allows us to define flows in a very compact way, e.g $\{a, b\} \overset{\rightarrow}{_a} \{c, d\} = \{a \rightarrow_a c, a \rightarrow_a d, b \rightarrow_a c, b \rightarrow_a d\}$. Formally

$$X \overset{\rightarrow}{_a} Y = \{x \rightarrow_a y \mid x \in X, y \in Y\}$$

There are, however, two issues with this solution that make it wrong. Can spot them? (answer in next page)

The issues of the wrong solution above are related to the introduction of explicit flows in guarded commands. Indeed, in b `->` c we have an implicit flow from the variables in b to variables being updated in c. Similarly, in $gc_1 [] gc_2$, the deterministic semantics will result in implicit flows from the (top-level) guards in gc_1 into gc_2.

Therefore, akin to the security analysis function of Definition 5.12, we need to make our function dependent on an additional parameter: The set of (implicit) variable dependencies. The new definition of our function is now:

$$
\begin{array}{rcl}
\mathsf{actualFlows}(c) & = & \mathsf{actualFlows}(c, \emptyset) \\
 & & \\
\mathsf{actualFlows}(\texttt{skip}, X) & = & \emptyset \\
\mathsf{actualFlows}(x{:=}e, X) & = & (fv(e) \cup X) \vec{a} \{x\} \\
\mathsf{actualFlows}(A[e_1]{:=}e_2, X) & = & (fv(e_1) \cup fv(e_2) \cup X) \vec{a} \{A\} \\
\mathsf{actualFlows}(c_1;c_2, X) & = & \mathsf{actualFlows}(c_1, X) \cup \mathsf{actualFlows}(c_2, X) \\
\mathsf{actualFlows}(\texttt{if } gc \texttt{ fi}, X) & = & \mathsf{actualFlows}(gc, X) \\
\mathsf{actualFlows}(\texttt{do } gc \texttt{ od}, X) & = & \mathsf{actualFlows}(gc, X) \\
\mathsf{actualFlows}(b \texttt{ -> } c, X) & = & \mathsf{actualFlows}(c, X \cup fv(b)) \\
\mathsf{actualFlows}(gc_1 [] gc_2, X) & = & \mathsf{actualFlows}(gc_1, X) \cup \mathsf{actualFlows}(gc_2, X \cup \mathsf{implicitDeps}(gc_1))
\end{array}
$$

where the auxiliary function implicitDeps is defined as:

$$
\begin{array}{rcl}
\mathsf{implicitDeps}(b \texttt{ -> } C) & = & fv(b) \\
\mathsf{implicitDeps}(gc_1 [] gc_2) & = & \mathsf{implicitDeps}(gc_1) \cup \mathsf{implicitDeps}(gc_2)
\end{array}
$$

Exercise 5.b [TEASER] Would you be able to compute the actual flows by using a program graph instead of the AST of a program?

Exercise 5.c. Allowed flows As part of the mandatory assignment you will have to compute the allowed flows for a given security policy, defined by a security lattice $\langle L, \sqsubseteq \rangle$ and a security classification C. In this exercise you are asked to formally define a function allowedFlows that, given by a security lattice $\langle L, \sqsubseteq \rangle$ and a security classification C, provides the set of flows allowed by the policy.

HINT: have a look at the first page of chapter 5.4.

Here is a possible solution:

$$\mathsf{allowedFlows}(\langle L, \sqsubseteq \rangle, C) = \{x_1 \to x_2 \mid C(x_1) \sqsubseteq C(x_2)\}$$

Exercise 5.d. Allowed flows (again) In the exercise above, we were assuming that we are given a security lattice $\langle L, \sqsubseteq \rangle$ but in the mandatory assignment we are given the lattice as a DAG $\langle L, \rightarrow \rangle$. How can you formally define the function allowedFlows that, given by a security lattice as a DAG $\langle L, \rightarrow \rangle$?

HINT: have a look at the first page of chapter 5.4.

On possible solution would be to formally define $\langle L, \sqsubseteq \rangle$ as suggested in the book (i.e. as the reflexive transitive closure of $\rightarrow$) and then re-use our previous solution:

$$\mathsf{allowedFlows}(\langle L, \sqsubseteq \rangle, C) = \{x1 \rightarrow x2 \mid C(x_1) \sqsubseteq C(x_2)\}$$

Defining $\sqsubseteq$ as the reflexive transitive closure of $\rightarrow$ can be done by defining it as the smallest relation on L such that:

$$l_1 \sqsubseteq l_1$$

and

$$l_1 \sqsubseteq l_2 \text{ and } l_2 \sqsubseteq l_3 \text{ implies } l_1 \sqsubseteq l_3$$

Exercise 5.e. Allowed flows (again) If we look into actual code, the security lattice/DAG is given as a list of of flows $F = [l_1 \to l_2, l_3 \to l_4, ...]$ between security domains. How can you formally define the function allowedFlows given by a security lattice/DAG as a list of flows F?

Exercise 5.f Flow violations The final step in the mandatory assignment task is to determine the flow violations. Formally define a function violations that given a set of actual flows A and a set of allowed flows B returns the set of flows that are not allowed.

This is simple defined as follows:

$$\mathsf{violations}(A, B) = A \setminus B$$

FM Chapter 5 – Language Based Security – Exercise 5.10

Lemma If $\langle q;\sigma\rangle \Longrightarrow_1^* \langle q';\sigma'\rangle$ then we also have $\langle q;\sigma\rangle \Longrightarrow_0^* \langle q';\sigma'\rangle$.

Proof. By induction on the size of the trace $\langle q;\sigma\rangle \Longrightarrow_1^* \langle q';\sigma'\rangle$.

- **Base case**: Holds vacuously, as there is no successor of $\langle q;\sigma\rangle$.
- **Induction Hypothesis**: Assume that if $\langle q;\sigma\rangle \Longrightarrow_1^n \langle q';\sigma'\rangle$ then we also have $\langle q;\sigma\rangle \Longrightarrow_0^n \langle q';\sigma'\rangle$.
- **Induction Step**: We need to show that if $\langle q;\sigma\rangle \Longrightarrow_1^{n+1} \langle q'';\sigma''\rangle$ then we can construct $\langle q;\sigma\rangle \Longrightarrow_0^{n+1} \langle q'';\sigma''\rangle$.

  With this we have that $\langle q;\sigma\rangle \Longrightarrow_1^n \langle q';\sigma'\rangle \overset{act}{\Longrightarrow}_1 \langle q'';\sigma''\rangle$. First notice that by IH on $\langle q;\sigma\rangle \Longrightarrow_1^n \langle q';\sigma'\rangle$ we get $\langle q;\sigma\rangle \Longrightarrow_0^n \langle q';\sigma'\rangle$.

  It now remains to show that if $\langle q';\sigma'\rangle \overset{act}{\Longrightarrow}_1 \langle q'';\sigma''\rangle$ then $\langle q';\sigma'\rangle \overset{act}{\Longrightarrow}_0 \langle q'';\sigma''\rangle$. We now have three cases for act:

  - Case $act = \texttt{skip}$. Because $\langle q';\sigma'\rangle \overset{\texttt{skip}}{\Longrightarrow}_1 \langle q'';\sigma''\rangle$, then we must have that $\mathcal{S}_1[\![\texttt{skip}]\!](\sigma') = \sigma'$, so $\sigma' = \sigma''$. Furthermore $\mathcal{S}_0[\![\texttt{skip}]\!](\sigma') = \sigma' = \sigma''$, and we now get that $\langle q';\sigma'\rangle \overset{\texttt{skip}}{\Longrightarrow}_0 \langle q'';\sigma''\rangle$.
  - Case $act = x := a\{X\}$. Because $\langle q';\sigma'\rangle \overset{x:=a\{X\}}{\Longrightarrow}_1 \langle q'';\sigma''\rangle$, then we must have that $\mathcal{A}[\![a]\!](\sigma')$ is defined and that $X \cup \mathsf{fv}(a) \rightrightarrows \{X\}$, which gives that $\mathcal{S}_1[\![x := a\{X\}]\!](\sigma') = \sigma'[x \mapsto \mathcal{A}[\![a]\!](\sigma')]$. So $\sigma'' = \sigma'[x \mapsto \mathcal{A}[\![a]\!](\sigma')]$.
    Because $\mathcal{A}[\![a]\!](\sigma')$ is defined and $0 = 0$, then we also have that $\mathcal{S}_0[\![x := a\{X\}]\!](\sigma') = \sigma''$, and we now get that $\langle q';\sigma'\rangle \overset{x:=a\{X\}}{\Longrightarrow}_0 \langle q'';\sigma''\rangle$.
  - Case $act = b$. Because $\langle q';\sigma'\rangle \overset{act}{\Longrightarrow}_1 \langle q'';\sigma''\rangle$ then we must have that $\mathcal{B}[\![b]\!](\sigma')$ is defined and holds, and we get that $\mathcal{S}_1[\![b]\!](\sigma') = \sigma'$. So $\sigma'' = \sigma'$.
    Because $\mathcal{B}[\![b]\!](\sigma')$ is defined and holds, then we get that $\mathcal{S}_0[\![b]\!](\sigma') = \sigma''$, and we now get that $\langle q';\sigma'\rangle \overset{b}{\Rightarrow}_0 \langle q'';\sigma''\rangle$. □

The program in Try It Out 5.9 with $x \not\to y$ would be allowed to progress by the reference-monitor semantics $\Longrightarrow_0$, but is halted by the reference-monitor semantics $\Longrightarrow_1$. This will therefore give us that $\langle q;\sigma\rangle \Longrightarrow_0^* \langle q';\sigma'\rangle$ but where $\langle q;\sigma\rangle \not\Longrightarrow_1^* \langle q';\sigma'\rangle$.

FM5 - Exercises and solutions

Exercise 5.a. Actual flows As part of the mandatory assignment you will have to compute the actual flows of a given guarded command program c. In this exercise you are asked to formally define a function actualFlows that, given a guarded command c provides the actual information flows of c according to the notion of information flows seen in the course. You should define the function recursively on the syntactic structure of guarded commands.

We will denote an actual flow between x and y with $x \rightarrow_a y$. A first attempt on definining actualFlows could look like this:

$$\begin{array}{rcl}
\mathsf{actualFlows}(\texttt{skip}) & = & \emptyset \\
\mathsf{actualFlows}(x{:=}e) & = & \{x\} \overset{\rightarrow}{_a} fv(e) \\
\mathsf{actualFlows}(A[e_1]{:=}e_2) & = & (fv(e_1) \cup fv(e_2)) \overset{\rightarrow}{_a} A \\
\mathsf{actualFlows}(c_1;c_2) & = & \mathsf{actualFlows}(c_1) \cup \mathsf{actualFlows}(c_2) \\
\mathsf{actualFlows}(\texttt{if } gc \texttt{ fi}) & = & \mathsf{actualFlows}(gc) \\
\mathsf{actualFlows}(\texttt{do } gc \texttt{ od}) & = & \mathsf{actualFlows}(gc) \\
\mathsf{actualFlows}(b \texttt{ -> } c) & = & \mathsf{actualFlows}(c) \\
\mathsf{actualFlows}(gc_1 [] gc_2) & = & \mathsf{actualFlows}(gc_1) \cup \mathsf{actualFlows}(gc_2)
\end{array}$$

where $\overset{\rightarrow}{_a}$ is an operation similar to the flow relation of Definition 5.1, which allows us to define flows in a very compact way, e.g $\{a, b\} \overset{\rightarrow}{_a} \{c, d\} = \{a \rightarrow_a c, a \rightarrow_a d, b \rightarrow_a c, b \rightarrow_a d\}$. Formally

$$X \overset{\rightarrow}{_a} Y = \{x \rightarrow_a y \mid x \in X, y \in Y\}$$

There are, however, two issues with this solution that make it wrong. Can spot them? (answer in next page)

The issues of the wrong solution above are related to the introduction of explicit flows in guarded commands. Indeed, in b `->` c we have an implicit flow from the variables in b to variables being updated in c. Similarly, in $gc_1 [] gc_2$, the deterministic semantics will result in implicit flows from the (top-level) guards in gc_1 into gc_2.

Therefore, akin to the security analysis function of Definition 5.12, we need to make our function dependent on an additional parameter: The set of (implicit) variable dependencies. The new definition of our function is now:

$$
\begin{array}{rcl}
\mathsf{actualFlows}(c) & = & \mathsf{actualFlows}(c, \emptyset) \\
 & & \\
\mathsf{actualFlows}(\texttt{skip}, X) & = & \emptyset \\
\mathsf{actualFlows}(x{:=}e, X) & = & (fv(e) \cup X) \vec{a} \{x\} \\
\mathsf{actualFlows}(A[e_1]{:=}e_2, X) & = & (fv(e_1) \cup fv(e_2) \cup X) \vec{a} \{A\} \\
\mathsf{actualFlows}(c_1;c_2, X) & = & \mathsf{actualFlows}(c_1, X) \cup \mathsf{actualFlows}(c_2, X) \\
\mathsf{actualFlows}(\texttt{if } gc \texttt{ fi}, X) & = & \mathsf{actualFlows}(gc, X) \\
\mathsf{actualFlows}(\texttt{do } gc \texttt{ od}, X) & = & \mathsf{actualFlows}(gc, X) \\
\mathsf{actualFlows}(b \texttt{ -> } c, X) & = & \mathsf{actualFlows}(c, X \cup fv(b)) \\
\mathsf{actualFlows}(gc_1 [] gc_2, X) & = & \mathsf{actualFlows}(gc_1, X) \cup \mathsf{actualFlows}(gc_2, X \cup \mathsf{implicitDeps}(gc_1))
\end{array}
$$

where the auxiliary function implicitDeps is defined as:

$$
\begin{array}{rcl}
\mathsf{implicitDeps}(b \texttt{ -> } C) & = & fv(b) \\
\mathsf{implicitDeps}(gc_1 [] gc_2) & = & \mathsf{implicitDeps}(gc_1) \cup \mathsf{implicitDeps}(gc_2)
\end{array}
$$

Exercise 5.b [TEASER] Would you be able to compute the actual flows by using a program graph instead of the AST of a program?

Exercise 5.c. Allowed flows As part of the mandatory assignment you will have to compute the allowed flows for a given security policy, defined by a security lattice $\langle L, \sqsubseteq \rangle$ and a security classification C. In this exercise you are asked to formally define a function allowedFlows that, given by a security lattice $\langle L, \sqsubseteq \rangle$ and a security classification C, provides the set of flows allowed by the policy.

HINT: have a look at the first page of chapter 5.4.

Here is a possible solution:

$$\mathsf{allowedFlows}(\langle L, \sqsubseteq \rangle, C) = \{x_1 \to x_2 \mid C(x_1) \sqsubseteq C(x_2)\}$$

Exercise 5.d. Allowed flows (again) In the exercise above, we were assuming that we are given a security lattice $\langle L, \sqsubseteq \rangle$ but in the mandatory assignment we are given the lattice as a DAG $\langle L, \rightarrow \rangle$. How can you formally define the function allowedFlows that, given by a security lattice as a DAG $\langle L, \rightarrow \rangle$?

HINT: have a look at the first page of chapter 5.4.

On possible solution would be to formally define $\langle L, \sqsubseteq \rangle$ as suggested in the book (i.e. as the reflexive transitive closure of $\rightarrow$) and then re-use our previous solution:

$$\mathsf{allowedFlows}(\langle L, \sqsubseteq \rangle, C) = \{x1 \rightarrow x2 \mid C(x_1) \sqsubseteq C(x_2)\}$$

Defining $\sqsubseteq$ as the reflexive transitive closure of $\rightarrow$ can be done by defining it as the smallest relation on L such that:

$$l_1 \sqsubseteq l_1$$

and

$$l_1 \sqsubseteq l_2 \text{ and } l_2 \sqsubseteq l_3 \text{ implies } l_1 \sqsubseteq l_3$$

Exercise 5.e. Allowed flows (again) If we look into actual code, the security lattice/DAG is given as a list of of flows $F = [l_1 \to l_2, l_3 \to l_4, ...]$ between security domains. How can you formally define the function allowedFlows given by a security lattice/DAG as a list of flows F?

Exercise 5.f Flow violations The final step in the mandatory assignment task is to determine the flow violations. Formally define a function violations that given a set of actual flows A and a set of allowed flows B returns the set of flows that are not allowed.

This is simple defined as follows:

$$\mathsf{violations}(A, B) = A \setminus B$$