FPV Tutorübung

Woche 1

Implications, Assertions and Conditions

Manuel Lerchner

20.04.2023

Organisatorisches

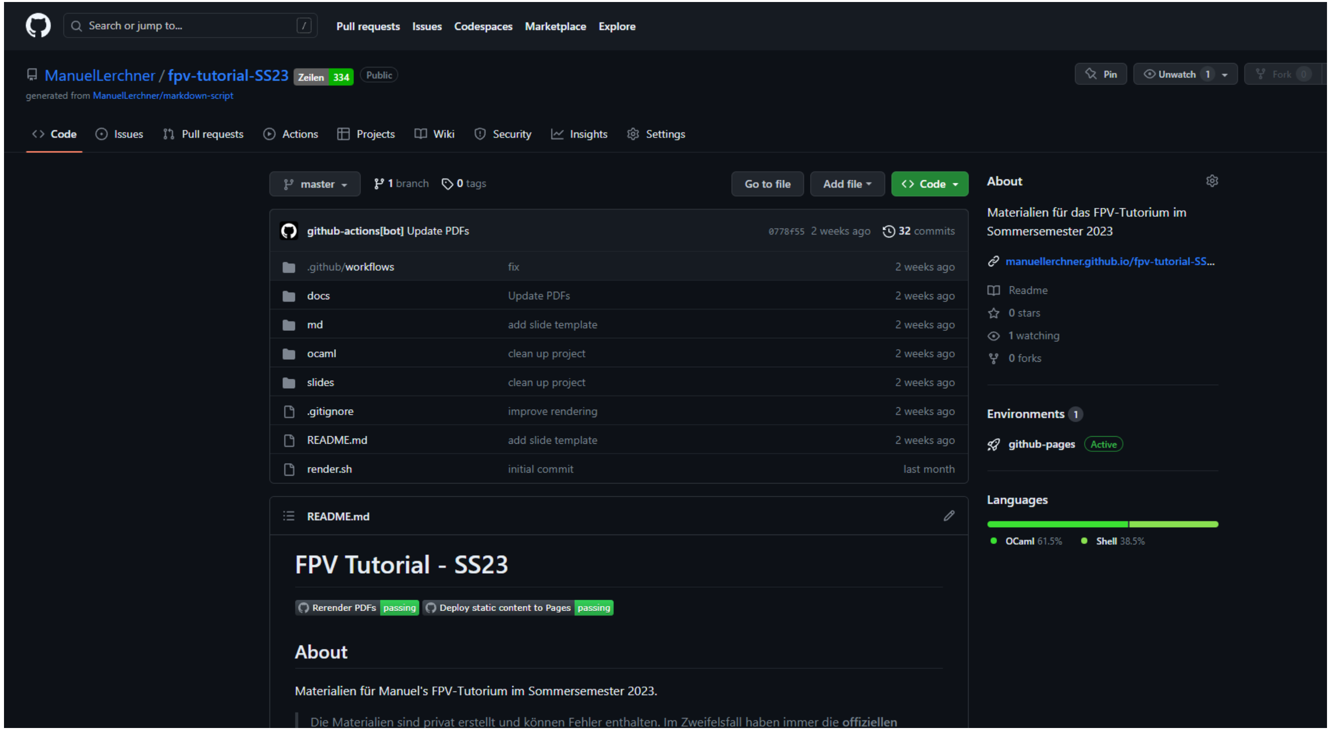
Grade Bonus

- Successful participation ($\geq 70\%$) in quizzes and programming tasks will lead to a bonus of 0.3 in the final exam, provided that you passed the exam.
- Programming homework and quizzes are to be submitted individually.
- Discussing solutions before the end of the week is considered plagiarism.
- Plagiarism will not be tolerated and will (at the very least) lead to exclusion from the bonus system

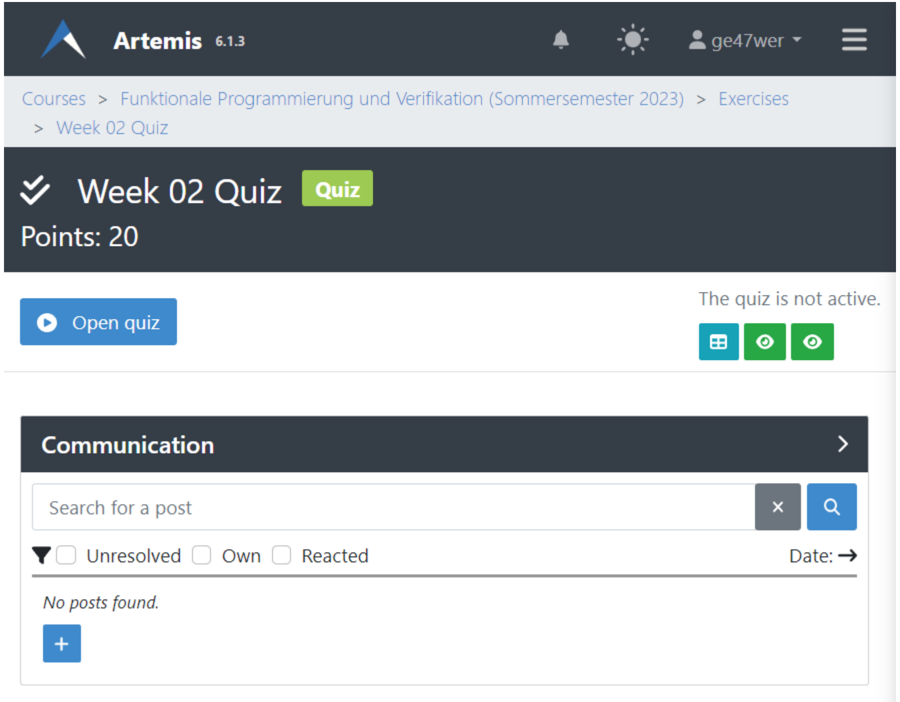
Changes

- Manual correction of homework not possible. However, non-programming exercises remain crucial for the exam
- 20% of the exam will be Single-Choice
- To receive points in the exam, your code needs to compile
- We currently anticipate an in-person exam using Artemis

Materialien



Quiz

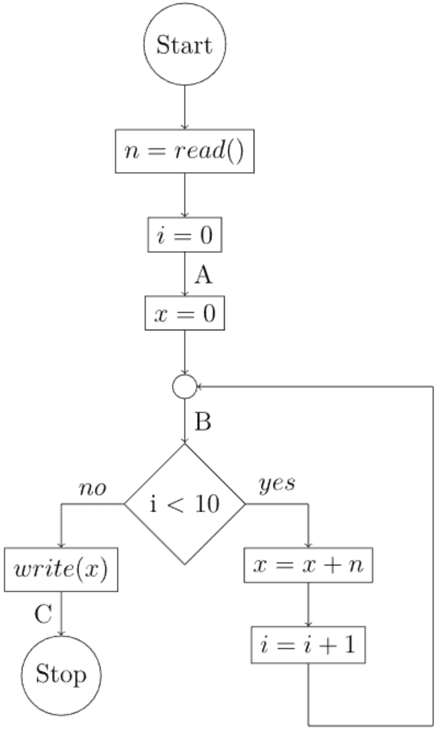


Passwort:

T01: Recap Implications

1. $x = 1 \implies 0 < x$
2. $x < 6 \implies x = 3$
3. $x > 0 \implies x \geq 0$
4. $x = -2 \implies x < -1 \lor x > 1$
5. $x = 0 \lor x = 7 \implies 4 \neq x$
6. $x = 1 \implies x \leq 3 \land y > 0$
7. $x < 8 \land y = x \implies y \neq 12$
8. $x = 1 \lor y = 1 \implies x > 0$
9. $x \neq 5 \implies false$
10. $true \implies x \neq y$
11. $false \implies x = 1$
12. $x \geq 1 \implies 2x + 3 = 5$
13. $A \land x = y \implies A$
14. $B \implies A \lor B$
15. $A \implies (B \implies A)$
16. $(A \implies B) \implies A$

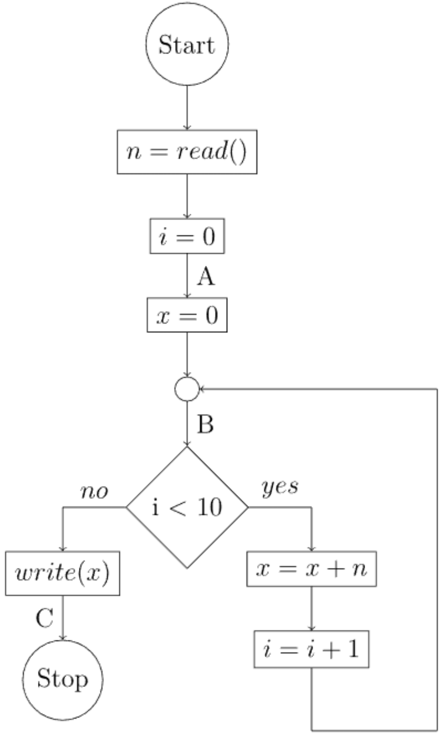
T02: Assertions



1. Which of the following assertions hold at point A?
   - a) $i \geq 0$
   - b) $x = 0$
   - c) $i \leq 10 \land x \neq 0$
   - d) $true$
   - e) $i = 0$
   - f) $x = i$
2. Which of the following assertions hold at point B?
   - a) $x = 0 \land i = 0$
   - b) $x = i$
   - c) $i < x$
   - d) $0 \leq i \leq 10$
   - e) $i \geq 0 \land x \geq 0$
   - f) $n = 1 \implies x = i$
3. Which of the following assertions hold at point C?
   - a) $i \geq 0$
   - b) $i = 10$
   - c) $i > 0$
   - d) $x \neq n$
   - e) $x = 10n$
   - f) $x = i * n \land i = 10$

T03: The Strong and the Weak

3. Which of the following assertions hold at point C?
   - a) $i \geq 0$ ✓
   - b) $i = 10$ ✓
   - c) $i > 0$ ✓
   - d) $x \neq n$ ✗
   - e) $x = 10n$ ✓
   - f) $x = i * n \land i = 10$ ✓



Again consider the assertions that hold at point C of assignment 2. Discuss the following questions:

1. When annotating the control flow graph, can you say that one of the given assertions is "better" than the others?
2. Can you arrange the given assertions in a meaningful order?
3. How can you define a *stronger than* relation formally?
4. How do $true$ and $false$ fit in and what is their meaning as an assertion?
5. What are the strongest assertions that still hold at A, B and C?

T04: Strongest Postconditions 1

1. $true$ → $\boxed{x = 5}$ → B

2. $x = 0 \land y = 2$ → $\boxed{x = 18}$ → B

3. $x \geq 7 \land y = -2$ → $\boxed{x = y + x}$ → B

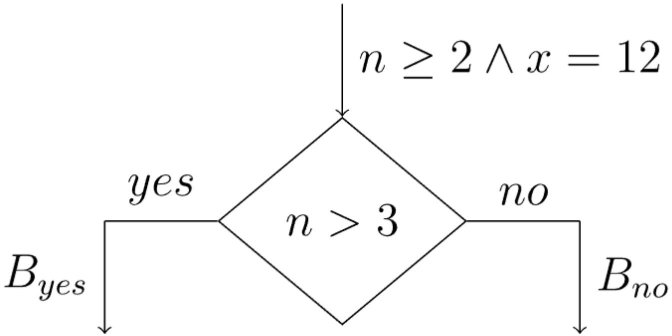
4. $x = 3 * i \land i \geq 0$ → $\boxed{i = i + 1}$ → B

5. $y = 2 + x$ → $\boxed{x = read()}$ → B

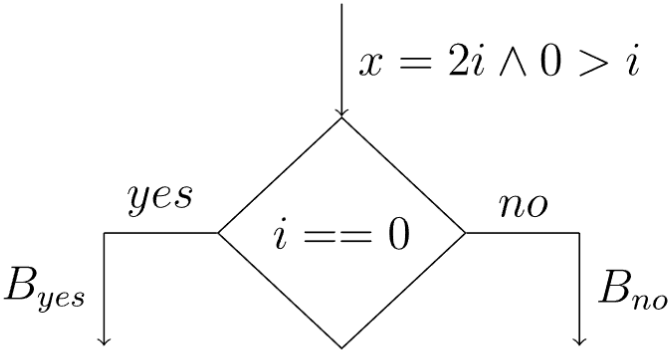
6. $i = i + 1$ → $\boxed{i = 0}$ → B

T04: Strongest Postconditions 2

7.

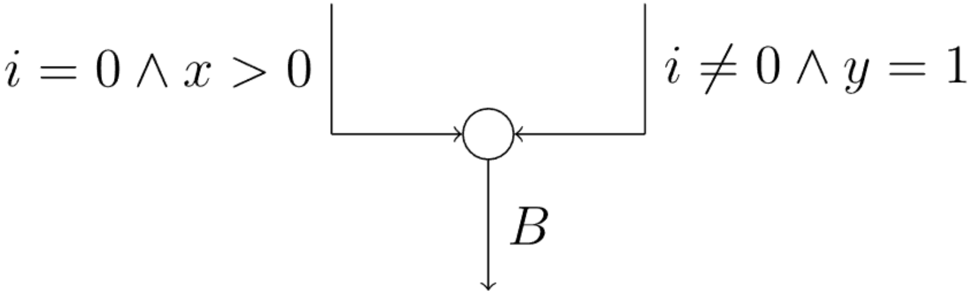


8.



T04: Strongest Postconditions 3

9.



FPV Tutorübung

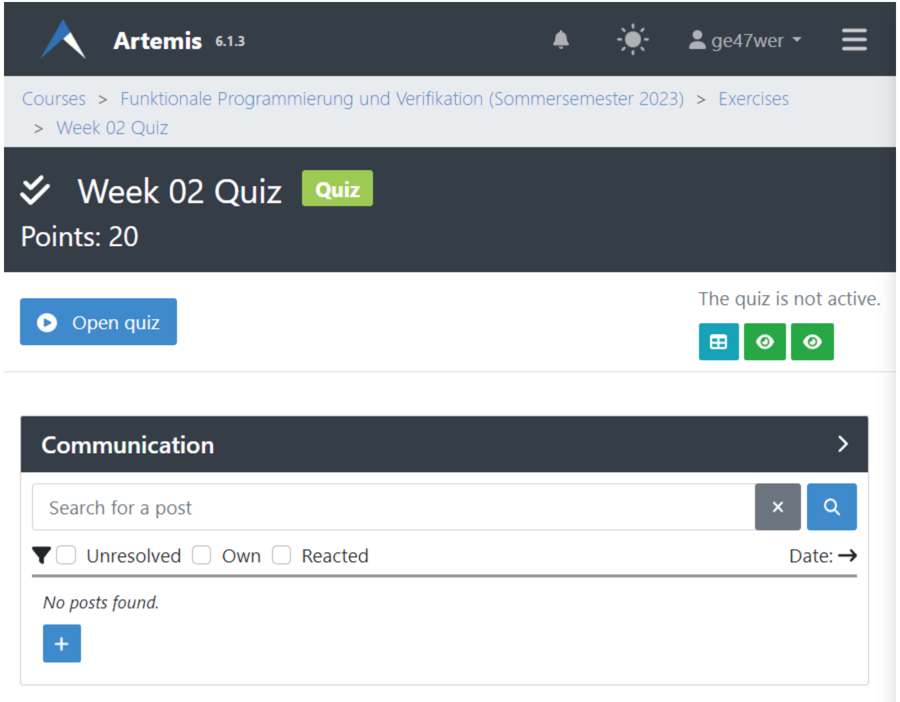
Woche 2

Preconditions, Postconditions and Local Consistency

Manuel Lerchner

03.05.2023

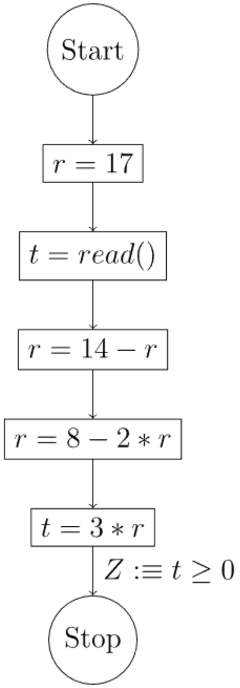
Quiz



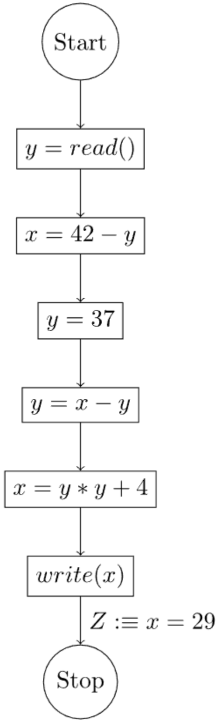
Passwort:

T01: From Post- to Preconditions

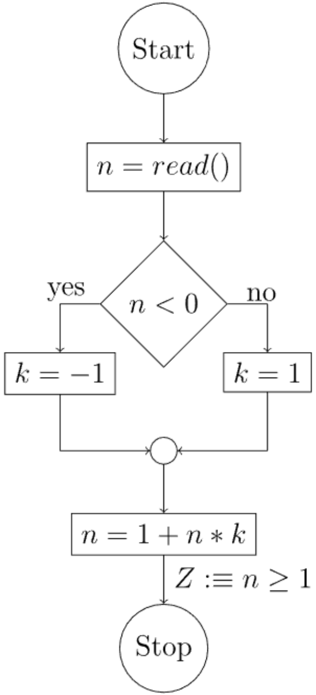
1.



2.



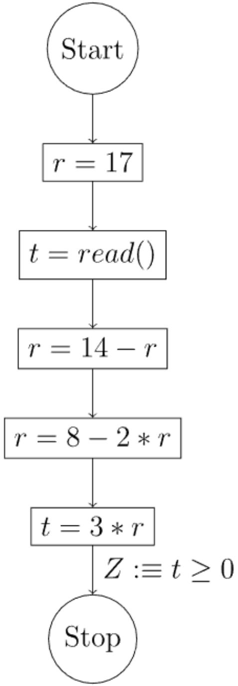
3.



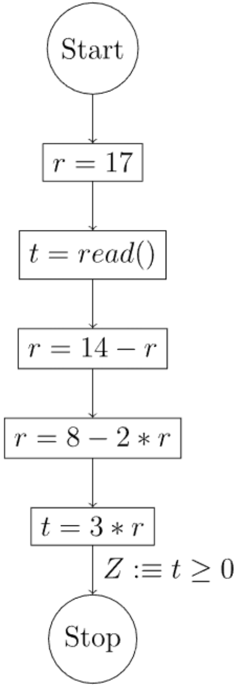
1. For each of these graphs show whether the assertion Z holds...
   (a) ...using strongest postconditions and
   (b) ...using weakest preconditions.
2. Discuss advantages and disadvantages of either approach.

T01: From Post- to Preconditions 1

Post-Condition:

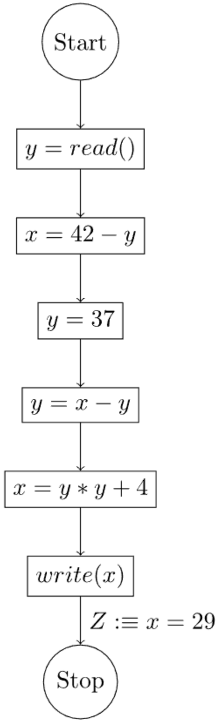


Pre-Condition:

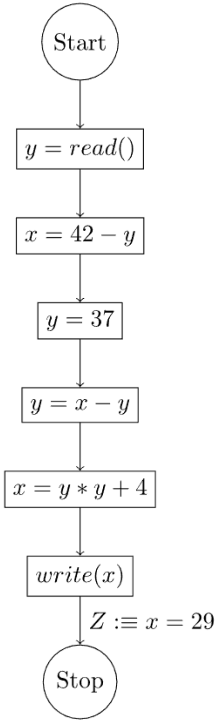


T01: From Post- to Preconditions 2

Post-Condition:

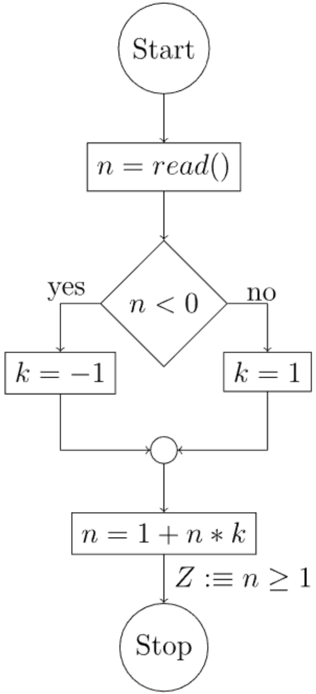


Pre-Condition:

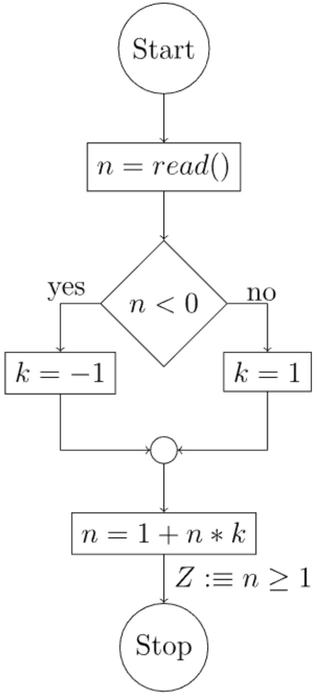


T01: From Post- to Preconditions 3

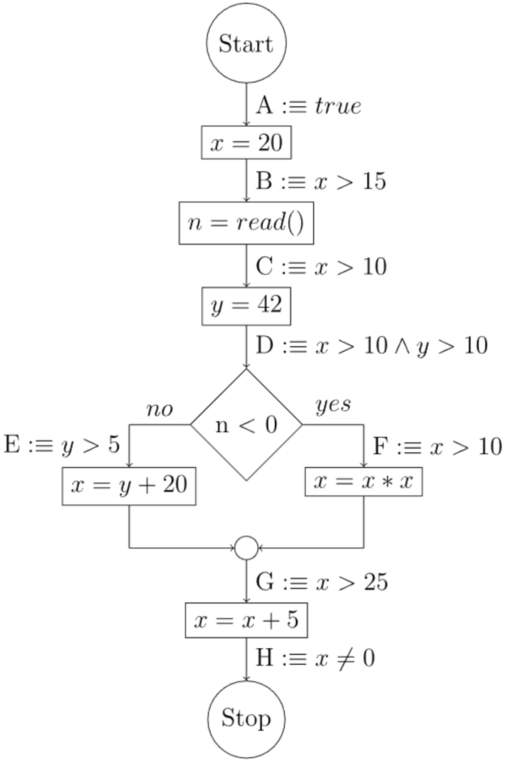
Post-Condition:



Pre-Condition:

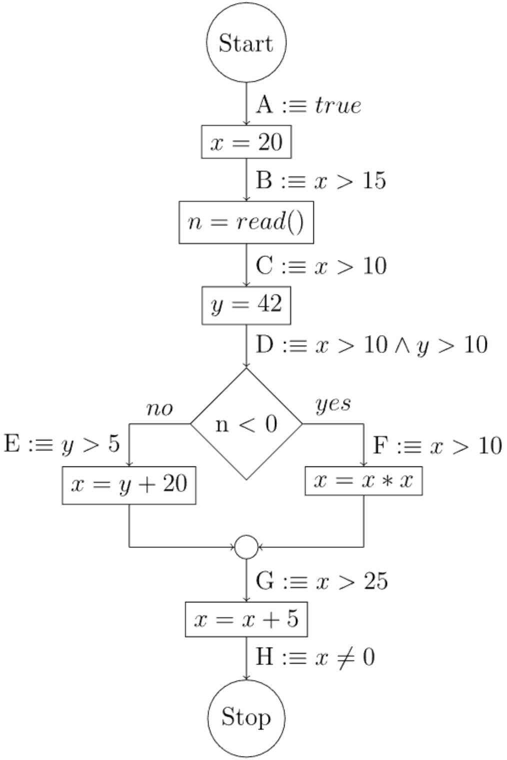


T02: Local Consistency

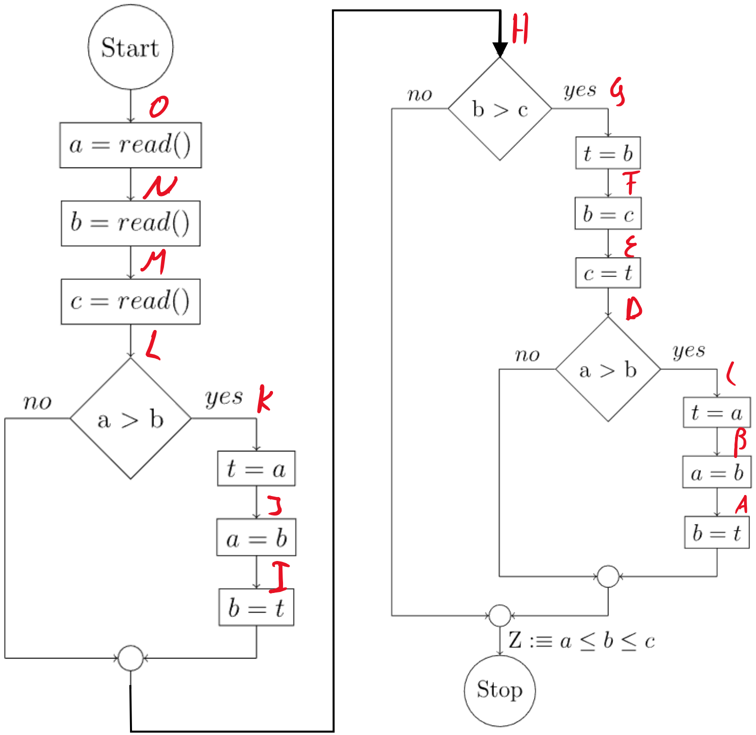


Check whether the annotated assertions prove that the program computes an $x \neq 0$ and discuss why this is the case.

T02: Local Consistency (Extra Space)

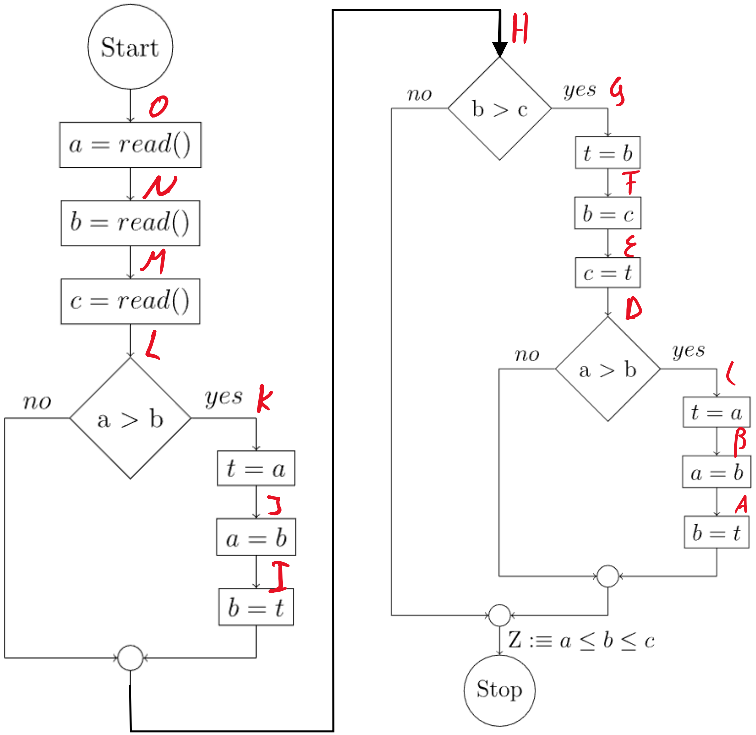


T03: Trouble Sort



1. Annotate each program point in the following control flow diagram with a suitable assertion, then show that your annotations are locally consistent and prove that Z holds at the given program point.
2. Discuss the drawbacks of annotating each program point with an assertion before applying weakest preconditions, and discuss how you could optimize the approach to proving that Z holds.

T03: Trouble Sort (Extra Space)



FPV Tutorübung

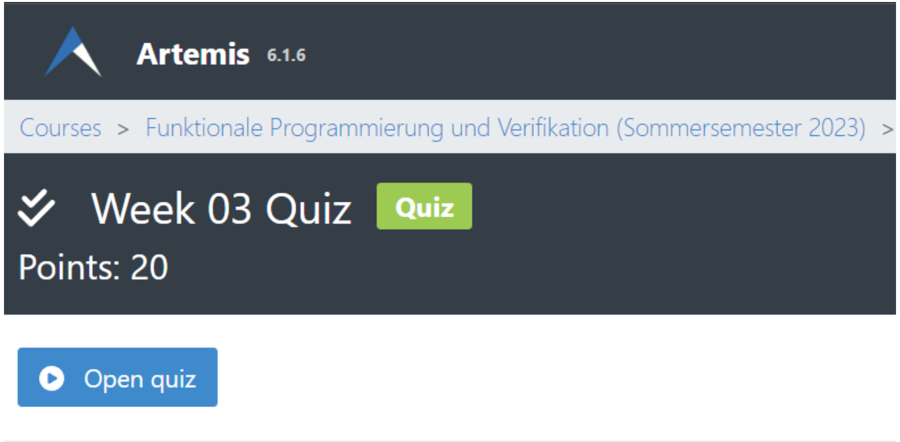
Woche 3

MiniJava 2.0, Loop Invariants

Manuel Lerchner

09.05.2023

Quiz



Passwort:

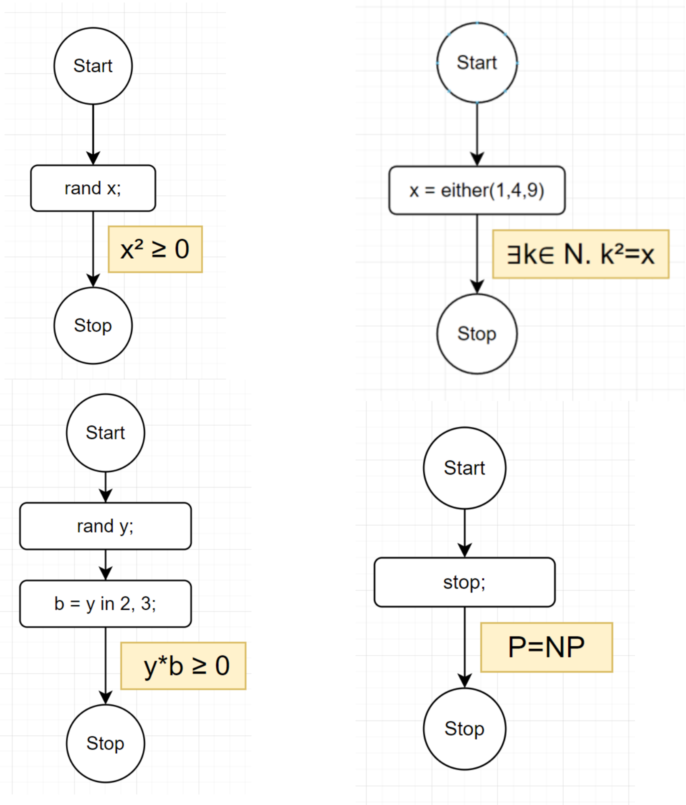
T01: MiniJava 2.0

In the lecture, the weakest precondition operator has been defined for all statements of MiniJava. In this assignment, we consider an extension of the MiniJava language, which provides four new statements:

1. `rand x`:
   Assigns a random value to variable x,
2. `x = either` $e_0, \dots, e_k$:
   Assigns one of the values of the expressions $e_0, \dots, e_k$ to variable `x` non-deterministically,
3. `x = e in a, b`:
   Assigns the value 1 to variable `x`, if the value of expression `e` is in the range $[a, b]$ and 0 if `e` is not in the range or the range is empty $(a > b)$,
4. `stop`:
   Immediately stops the program.

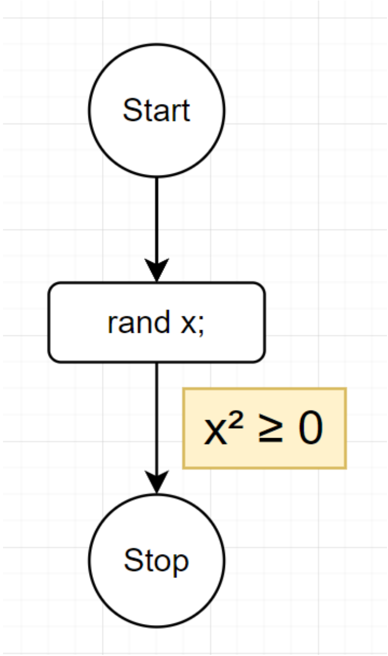
Define the weakest precondition operator $\mathbf{WP}[\![\dots]\!](B)$ for each of these statements. (In terms of B)

Beispiele zum Testen:



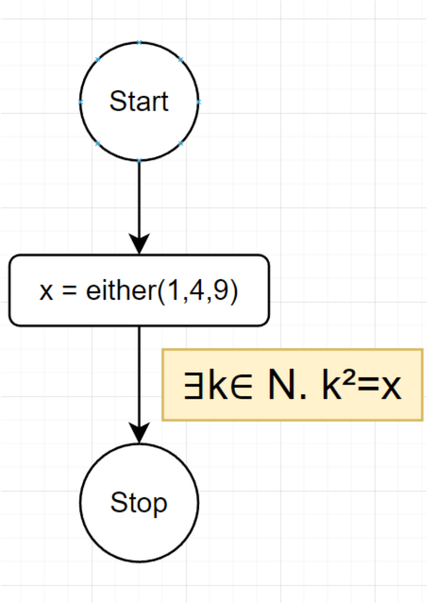
T01: MiniJava 2.0

WP[rand x;](B) =



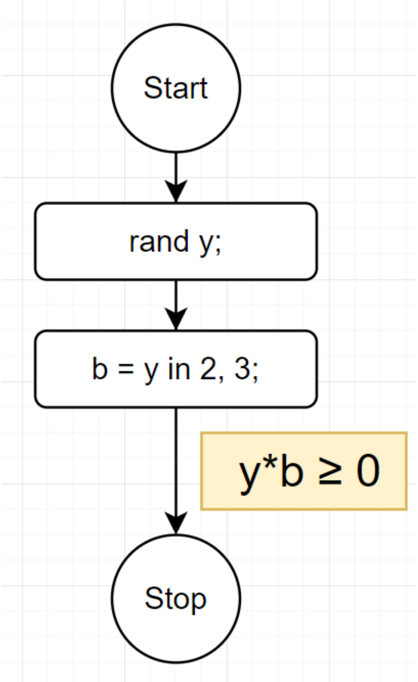
T01: MiniJava 2.0

$WP[x = \text{either } e_0, e_1 \ldots e_k](B) =$



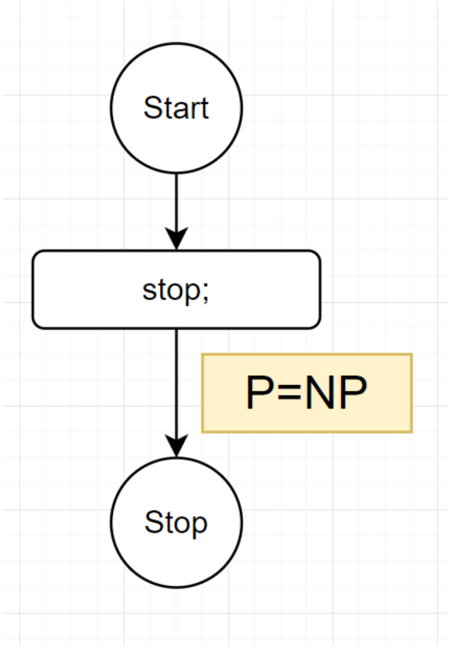
T01: MiniJava 2.0

WP[x e in a, b](B) =



T01: MiniJava 2.0

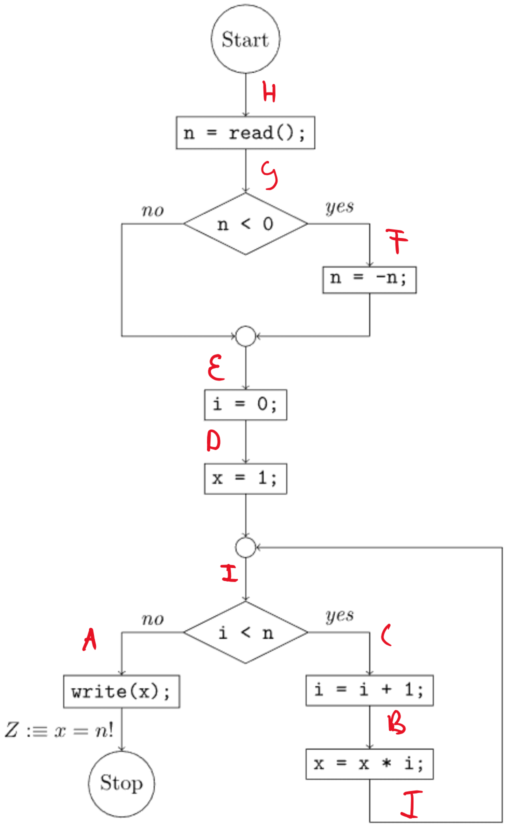
WP[stop](B) =



T02: Loop Invariants

1. Discuss the problem that arises when computing weakest preconditions to prove Z.
2. How can you use weakest preconditions to prove Z anyway?
3. Try proving Z using the the loop invariants $x \geq 0$ and $i = 0 \wedge x = 1 \wedge n = 0$ **at the end of the loop body** and in particular discuss these questions:
   - a) How has a useful loop invariant be related to Z?
   - b) What happens if the loop invariant is chosen too strong?
   - c) What happens if the loop invariant is chosen too weak?
   - d) Can you give a meaningful lower and upper bound for useful loop invariants?
4. Retry proving Z using the loop invariant $x = i!$ (again at the end of the loop body) and improve this invariant until the proof succeeds.

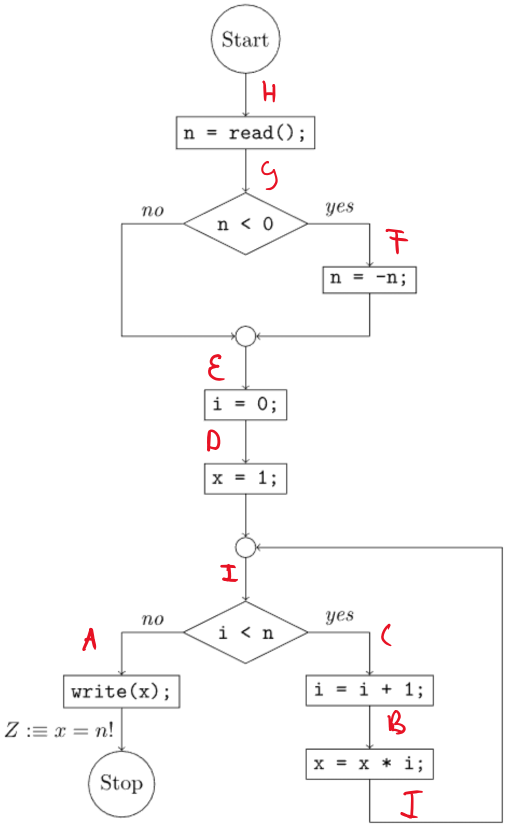
A program computes the factorial of its input:



T02: Loop Invariants 1

3. Try proving Z using the the loop invariants $x \geq 0$ and $i = 0 \wedge x = 1 \wedge n = 0$ **at the end of the loop body** and in particular discuss these questions:

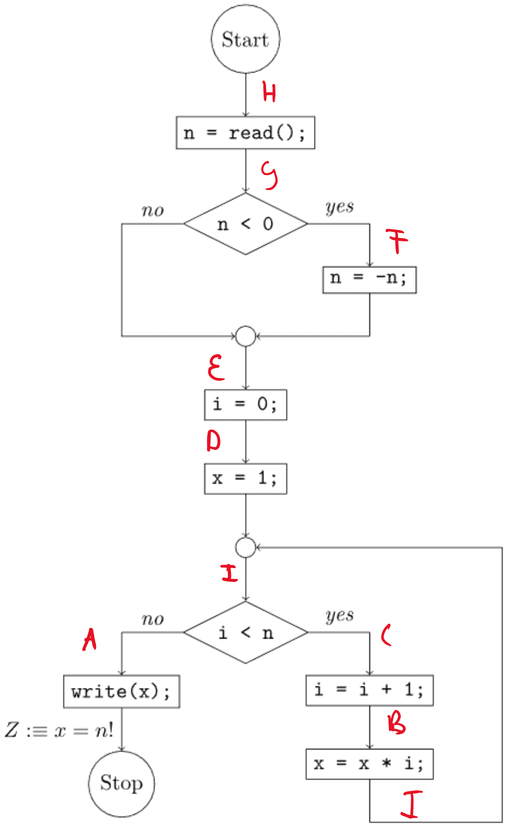
A program computes the factorial of its input:



T02: Loop Invariants 2

3. Try proving Z using the the loop invariants $x \geq 0$ and $i = 0 \land x = 1 \land n = 0$ **at the end of the loop body** and in particular discuss these questions:

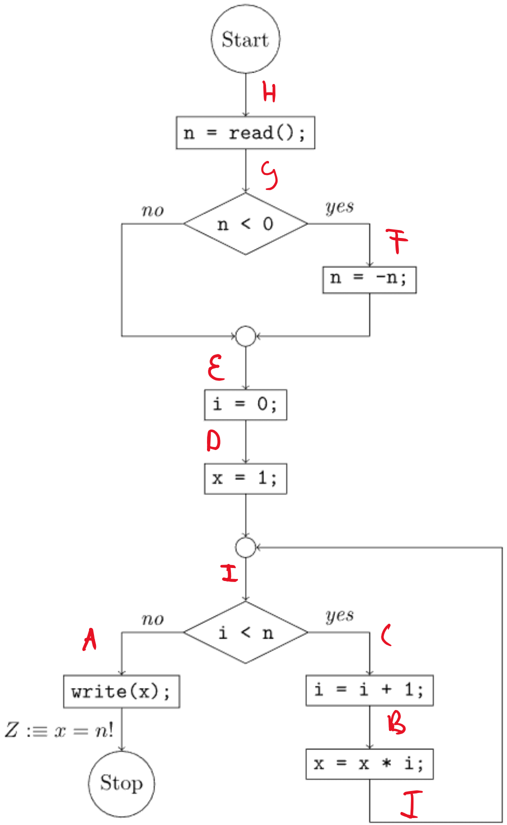
A program computes the factorial of its input:



T02: Loop Invariants 3

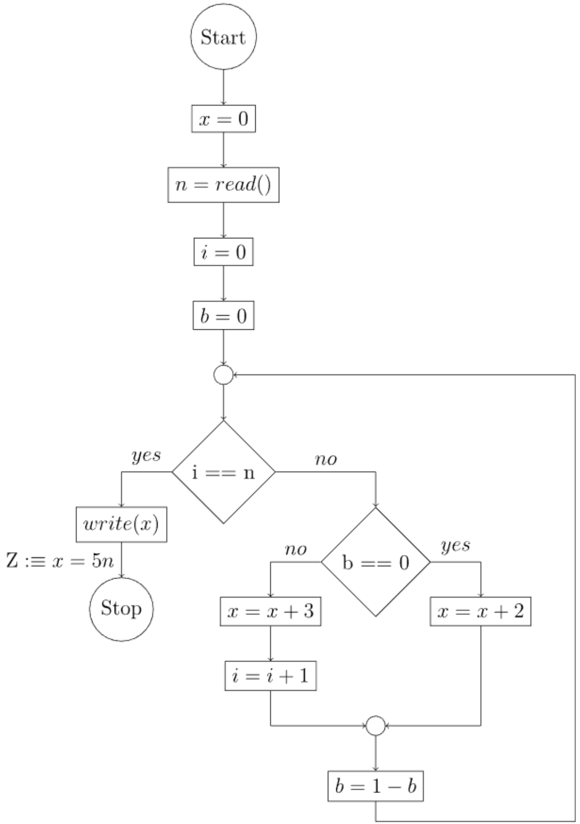
4. Retry proving Z using the loop invariant $x = i!$ (again at the end of the loop body) and improve this invariant until the proof succeeds.

A program computes the factorial of its input:



T03: Two b, or Not Two b

Prove Z using weakest preconditions.

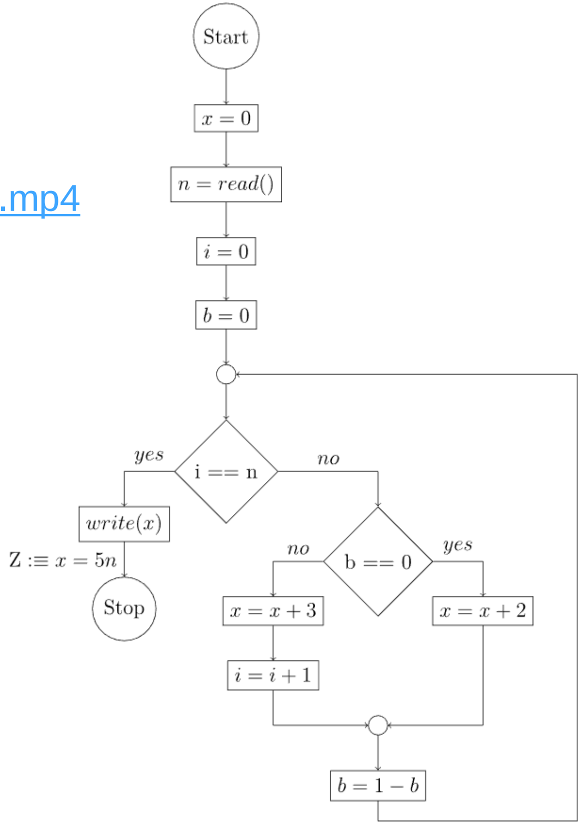


T03: Two b, or Not Two b

Tipps zum finden von Loop Invarianten:
https://ttt.in.tum.de/recordings/Info2_2017_11_24-1/Info2_2017_11_24-1.mp4

Beispieltrace: n=3

Variable \ Schleifendurchgang	0	1	2	3	4	5	6
x	**0**	**2**	**5**	**7**	**10**	**12**	**15**
i	**0**	**0**	**1**	**1**	**2**	**2**	**3**
b	**0**	**1**	**0**	**1**	**0**	**1**	**0**



Tipps für Loop Invarianten

https://ttt.in.tum.de/recordings/Info2_2017_11_24-1/Info2_2017_11_24-1.mp4

Tipp 1

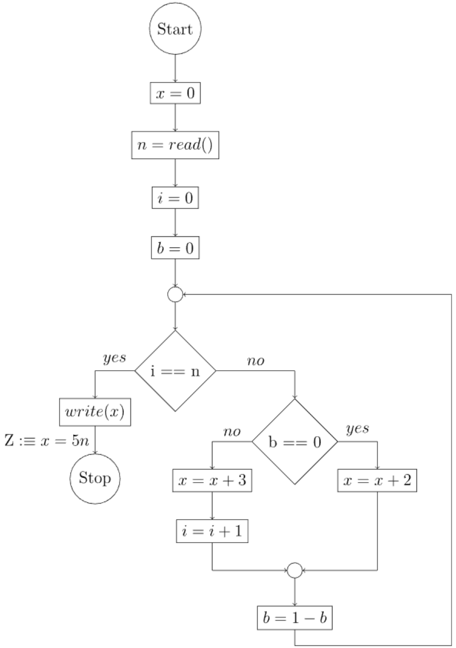
Wir benötigen eine Aussage über den Wert der Variablen, über die wir etwas beweisen wollen (x) in der Schleifeninvariante. Die Aussage muss dabei mindestens so präzise ($\neq, \geq, \leq, =$) sein, wie die Aussage, die wir beweisen wollen.

Tipp 2

Variablen, die an der Berechnung von x beteiligt sind **und** Werte von einer Schleifeniteration in die nächste transportieren ("loop-carried"), müssen in die Schleifeninvariante aufgenommen werden.

Tipp 3

Die Schleife zu verstehen ist unerlässlich. Eine Tabelle für einige Schleifendurchläufe kann helfen die Zusammenhänge der Variablen (insbesondere mit dem Schleifenzähler i) aufzudecken. Oft lassen sich mit einer Tabelle, in der man die einzelnen Berechnungsschritte notiert, diese Zusammenhänge deutlich leichter erkennen, als mit einer Tabelle, die nur konkrete Werte enthält.



$$I :\equiv x = 5i + 2b \wedge b \in \{0, 1\} \wedge (i = n \implies b = 0)$$

FPV Tutorübung

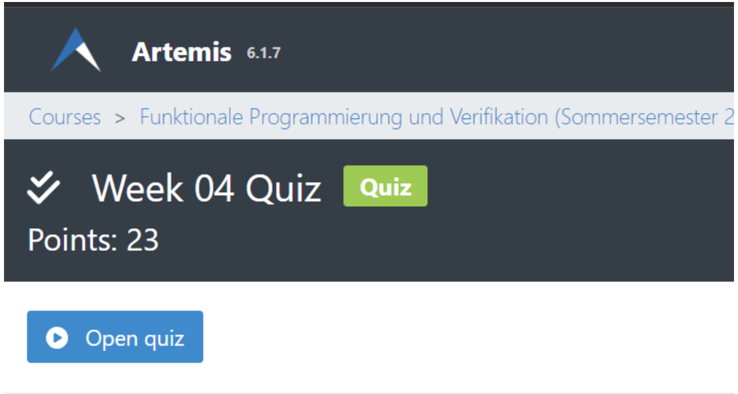
Woche 4

Loop Invariants and Termination proofs

Manuel Lerchner

15.05.2023

Quiz

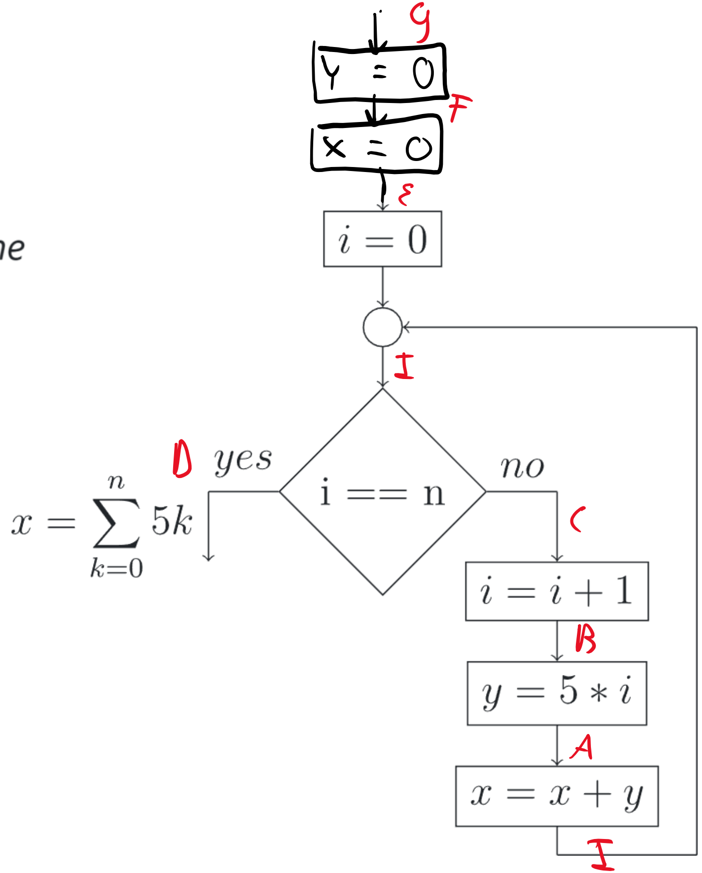


Passwort:

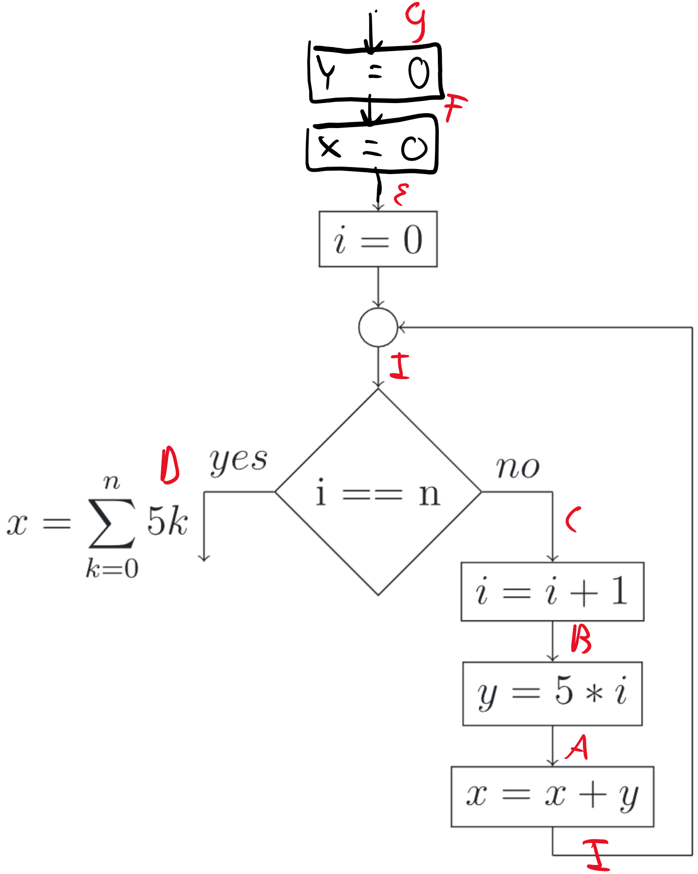
T01: Loop Invariants

Find a suitable loop invariant and prove it locally consistent.

Note: We follow the standard practice that the empty sum, where the number of terms is zero, is 0*, e.g.:* $\sum_{k=0}^{-1}(\ldots) = 0$.



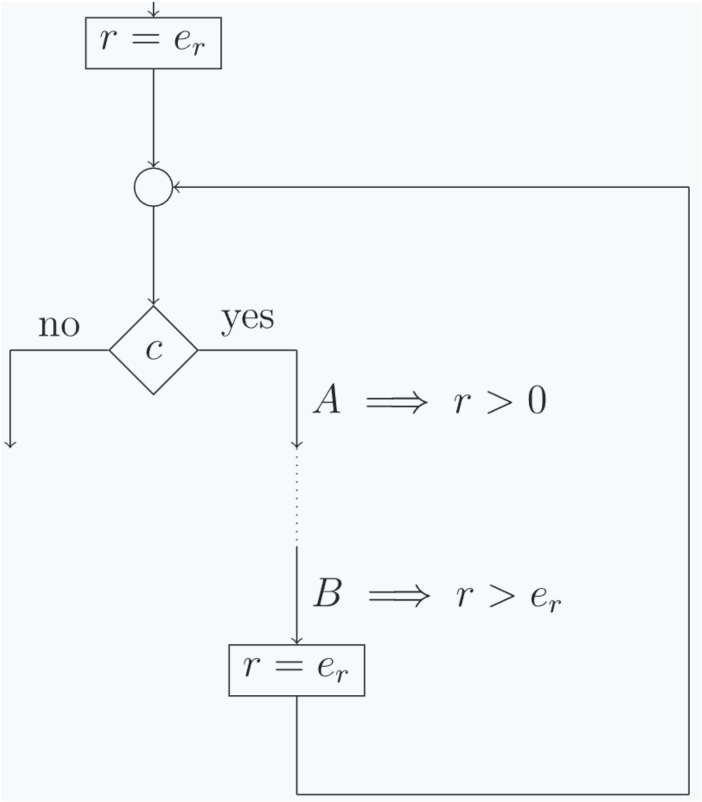
T01: Loop Invariants



T02: Termination

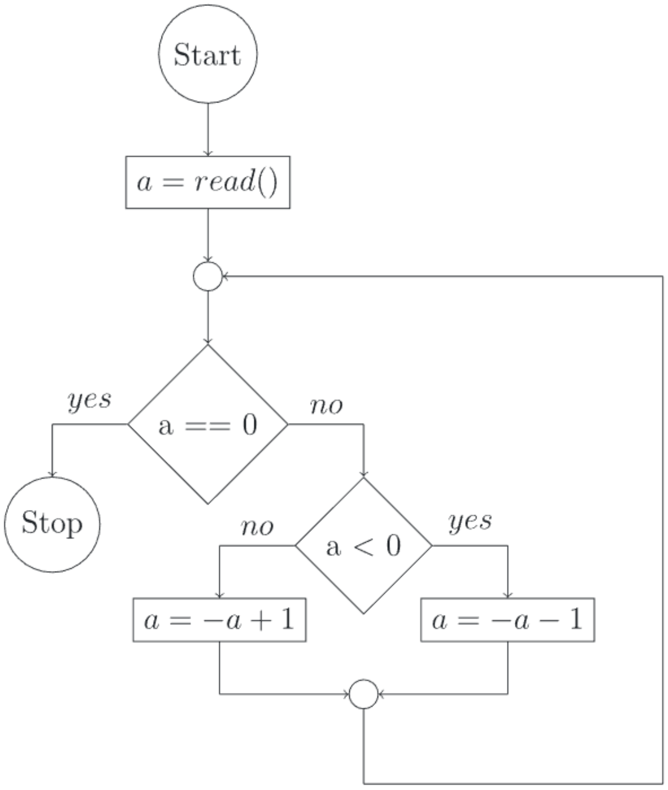
In the lecture, you have learned how to prove termination of a MiniJava program. Discuss these questions:

1. How can you decide whether a termination proof is required at all?
2. What is the basic idea of the termination proof?
3. How is the program to be modified?
4. What has to be proven?
5. How is the loop invariant influenced?



T03: A Wavy Approach

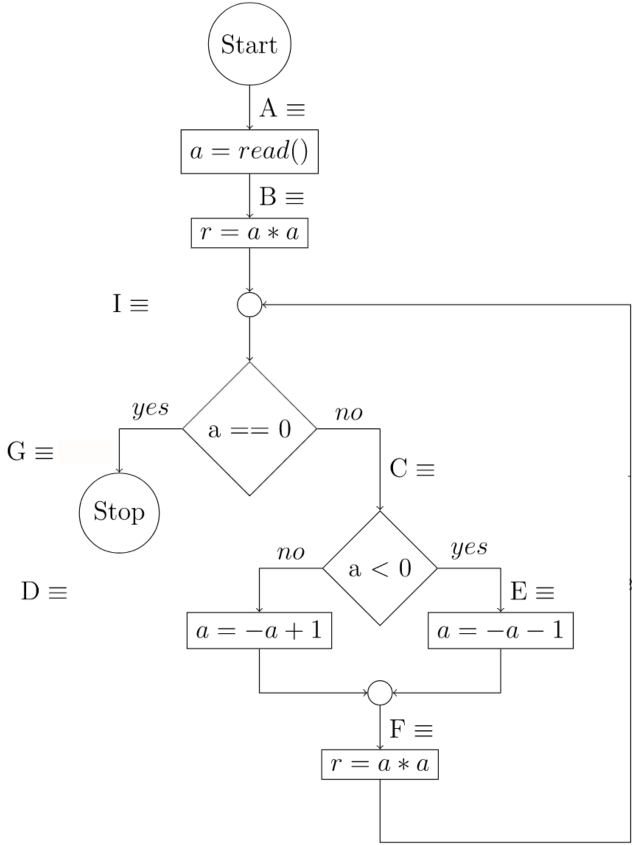
Prove termination of the following program:



Todos:

1. Schleife verstehen
2. Variable r definieren / finden
   - r ≥ 0 in jedem Durchgang
   - r wird strikt kleiner
3. Neue Variable und Assertions einfügen
   - Am Ende „true“ Assertion!
4. Local-Consistency zeigen

T03: A Wavy Approach



Tipps für Loop Invarianten

https://ttt.in.tum.de/recordings/Info2_2017_11_24-1/Info2_2017_11_24-1.mp4

Tipp 1

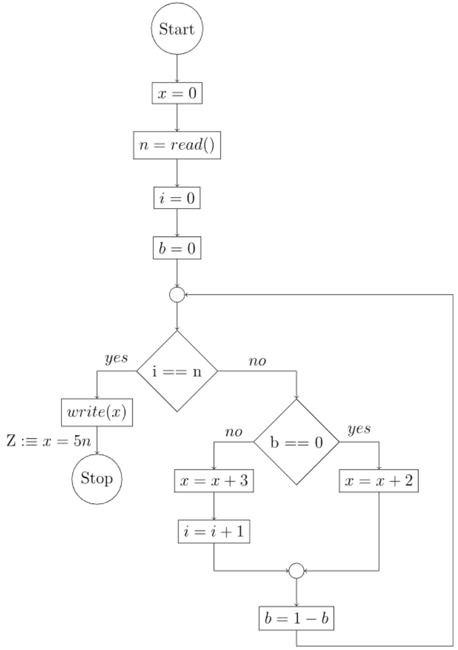
Wir benötigen eine Aussage über den Wert der Variablen, über die wir etwas beweisen wollen (x) in der Schleifeninvariante. Die Aussage muss dabei mindestens so präzise ($\neq, \geq, \leq, =$) sein, wie die Aussage, die wir beweisen wollen.

Tipp 2

Variablen, die an der Berechnung von x beteiligt sind **und** Werte von einer Schleifeniteration in die nächste transportieren ("loop-carried"), müssen in die Schleifeninvariante aufgenommen werden.

Tipp 3

Die Schleife zu verstehen ist unerlässlich. Eine Tabelle für einige Schleifendurchläufe kann helfen die Zusammenhänge der Variablen (insbesondere mit dem Schleifenzähler i) aufzudecken. Oft lassen sich mit einer Tabelle, in der man die einzelnen Berechnungsschritte notiert, diese Zusammenhänge deutlich leichter erkennen, als mit einer Tabelle, die nur konkrete Werte enthält.



$$I :\equiv x = 5i + 2b \wedge b \in \{0,1\} \wedge (i = n \implies b = 0)$$

FPV Tutorübung

Woche 5

Ocaml

Manuel Lerchner

22.05.2023

T01: Expressions

So far, you learned about the following types of expressions:

- Constants
- Variables
- Unary operators
- Binary operators
- Tuples
- Records
- Lists
- If-then-else
- Pattern matching
- Function definition
- Function application
- Variable binding

1. For each of the aforementioned types of expressions, give the general structure and two concrete examples with different subexpressions.

T01: Expressions

- Constants:
- Variables:
- Unary Operator:
- Binary Operator:
- Tuples:

T01: Expressions

- Records (definition):
- Records (access):
- Lists:
- if-then-else:

T01: Expressions

- Pattern Matching:
- Function Definition :
- Function Application :
- Variable Binding:

T01: Expressions

2. For the following expressions, list all contained subexpressions and give their corresponding types. Then evaluate the expressions:

```
(* a *) let a = fun x y -> x + 2 in a 3 8 :: []
```

```
(* a *) let a = fun x y -> x + 2 in a 3 8 :: []

(* b *) ((fun x -> x::[]) (9 - 5), true, ('a', 7))
```

T01: Expressions

```
(* a *) let a = fun x y -> x + 2 in a 3 8 :: []
```

T01: Expressions

```
(* b *) ((fun x -> x::[]) (9 - 5), true, ('a', 7))
```

T02: What‘s the Point

Using what you learned about tuple types in the lecture, implement functionality for computing with three-dimensional vectors.

1. **Define a suitable data type for your point.** 0 of 1 tests passing
   The type `vector3` should be a tuple of 3 float values.
2. **Define three points** 0 of 1 tests passing
   The points `p1`, `p2` and `p3` should all be different, but their exact values don't matter. Use them, along with other points, to test your functions.
3. **string_of_vector3** 0 of 1 tests passing
   Implement a function `string_of_vector3 : vector3 -> string` to convert a vector into a human-readable representation.
   For example, the string for the zero vector should be: `(0.,0.,0.)`.
   Hint: use `string_of_float` to convert components.
4. **vector3_add** 0 of 1 tests passing
   Write a function `vector3_add : vector3 -> vector3 -> vector3` that adds two vectors component-wise.
5. **vector3_max** 0 of 1 tests passing
   Write a function `vector3_max : vector3 -> vector3 -> vector3` that returns the larger argument vector (the vector with the greater magnitude).
6. **combine** 0 of 1 tests passing
   Write a function `combine : vector3 -> vector3 -> vector3 -> string` that adds its first argument to the larger of the other two arguments and returns the result as a string.

T03: Student Database

In this assignment, you have to manage the students of a university.

1. **Type** No results

   First you need to define some types.

   - Define a data type for a `student`.

     A student should be represented as a record of the students `first_name`, `last_name`, identification number `id`, number of the current `semester` as well as the list of `grades` received in different courses.

     The grades should be a pair of the course number and the grade value, a floating point number.
   - To actually manage student you need a `database` which shall be represented as a list of students.

2. **insert** No results

   Write a function `insert : student -> database -> database` that inserts a student into the database.

3. **find_by_id** No results

   Write a function `find_by_id : int -> database -> student list` that returns a list with the (first) student with the given id (either a single student or an empty list, if no such student exists).

4. **find_by_last_name** No results

   Implement a function `find_by_last_name : string -> database -> student list` to find all students with a given last name.

FPV Tutorübung

Woche 6

Ocaml: List-Module, Binary Search Trees

Manuel Lerchner

31.05.2023

T01: Explicit Type Annotation

In OCaml, types are inferred automatically, so there is no need to write them down explicitly. However, types can be annotated by the programmer. Discuss:

1. In the following expression, annotate the types of all subexpressions:

```
let f = fun x y -> x, [y]
```

2. When can explicitly annotated types be helpful?

T02: The List Module

Check the documentation of the OCaml `List` module here and find out what the following functions do. Then implement them yourself. Make sure your implementations have the same type. In cases where the standard functions throw exceptions, you may just `failwith "invalid"`.

1. **hd** 0 of 1 tests passing
   Implement the function `hd`
2. **tl** 0 of 1 tests passing
   Implement the function `tl`
3. **length** 0 of 1 tests passing
   Implement the function `length`
4. **append** 0 of 1 tests passing
   Implement the function `append`
5. **rev** 0 of 1 tests passing
   Implement the function `rev`
6. **nth** 0 of 1 tests passing
   Implement the function `nth`

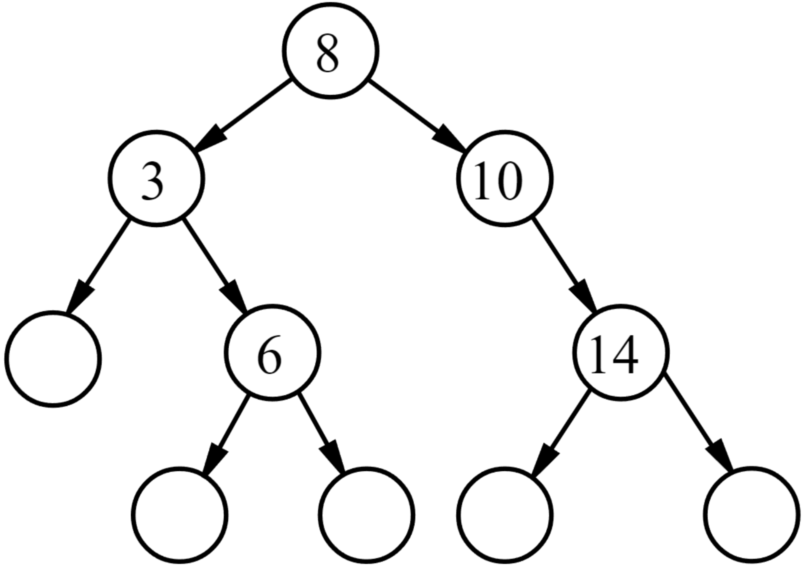
T03: Binary Search Tree 1

In this assignment, a collection to organize integers shall be implemented via binary search trees.

1. Define a suitable data type `tree` for binary trees that store integers. Each node in the binary tree should either be
   - an inner node which stores a value of type `int` and has a left and a right child of type `tree`, or
   - a leaf node and contain no value.

Since you are free to define your `tree` type however you wish (the type `tree` is said to be *abstract*), we need to define functions for creating trees.

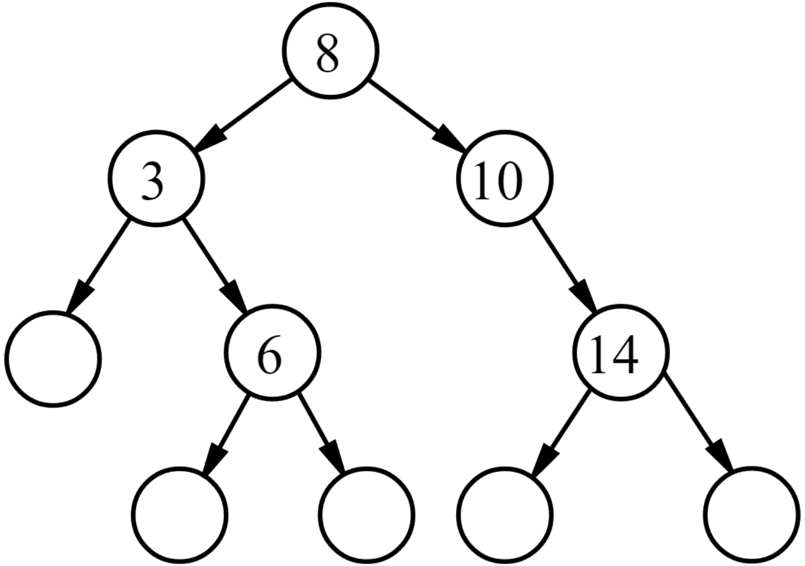
2. Define functions `node` and `leaf`, which should create inner nodes and leaves, respectively:
   - `node v l r` should create an inner node with the value `v`, left child `l` and right child `r`,
   - `leaf ()` should create an leaf node of your `tree` type.



T03: Binary Search Tree 2

Similarly, we need a function that allows us to inspect the structure of your tree, specifically to access the children of a node. The OCaml `Option` type (API documentation for Option) allows us to cleanly distinguish between the presence of absence of a value. Here, we will use it to distinguish between inner nodes, which have children, and leaf nodes, which do not. Using the `Option` type and returning `None` instead of raising an exception is (often) good functional programming style!

3. Define the function `inspect`, which allows us to access the children of a node:
    - `inspect n`, where `n` is an inner node of your `tree` type with the value `v` and children `l` and `r`, should return `Some (v, l, r)`. Here, `Some` is used to indicate the *presence* of a value and children.
    - `inspect l`, where `l` is a leaf of your `tree` type (with no children), should return `None`. Here, `None` is used to indicate the *absence* of a value and children.

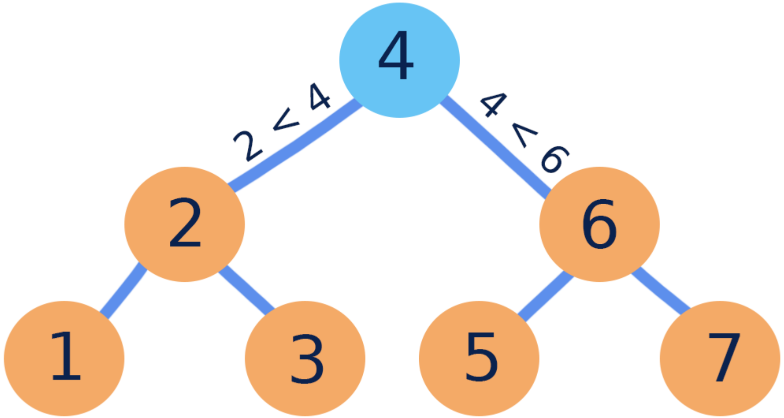


T03: Binary Search Tree 3

4. Define a binary tree `t1` which contains the values $8, 12, 42, 1, 6, 9, 8$. To construct the tree, start with an empty tree, then insert the given values in order.

T03: Binary Search Tree 4

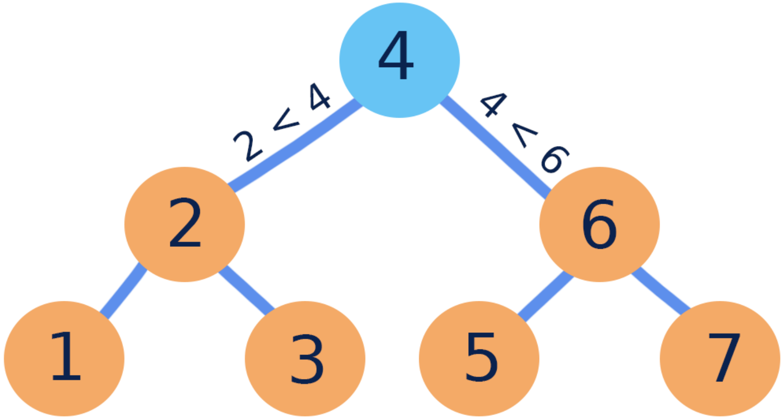
5. Implement a function `to_list : tree -> int list` that returns an ordered list of all values in the tree.



In Order Traversal: [1 2 3 4 5 6 7]

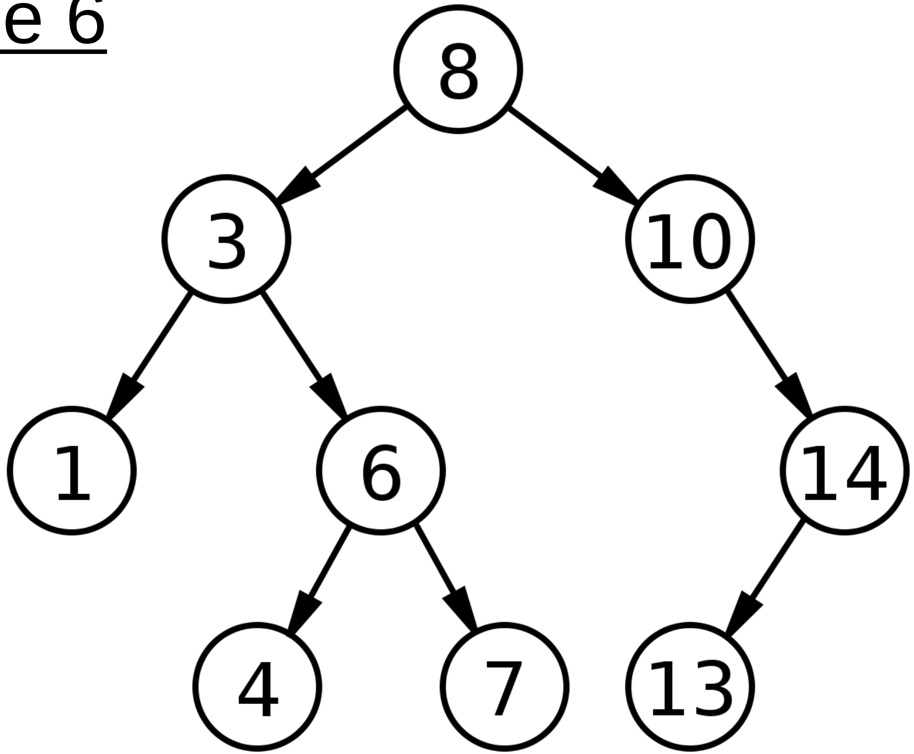
T03: Binary Search Tree 5

6. Implement a function `insert : int -> tree -> tree` which inserts a value into the tree. If the value exists already, the tree is not modified.



T03: Binary Search Tree 6

7. Implement a function `remove : int -> tree -> tree` to remove a value (if it exists) from the tree. Upon removing a value from an inner node:
   - if either child of the inner node is empty, replace the entire inner node with the other child, otherwise
   - if neither child of the inner node is empty, the value should be replaced with the largest value from the *left* subtree.



FPV Tutorübung

Woche 7

OCaml: List-Module 2, Mappings, Operator Functions

Manuel Lerchner

08.06.2023

T01: List Module Part 2

- Use functions from the List-Module!

Implement the following functions without defining any recursive functions yourself:

1. **float_list** 0 von 1 Tests bestanden
   Implement the function `float_list : int list -> float list` that converts all ints in the list to floats.
2. **to_string** 0 von 1 Tests bestanden
   Implement the function `to_string : int list -> string` that builds a string representation of the given list. E.g.: `"[0;42;123;420;1;]"`
3. **part_even** 0 von 1 Tests bestanden
   Implement the function `part_even : int list -> int list` that partitions all even values to the front of the list.
4. **squaresum** 0 von 1 Tests bestanden
   Implement the function `squaresum : int list -> int` that computes $\sum_{i=1}^{n} x_i^2$ for a list $[x_1, \ldots, x_n]$.

T01: List Module Part 2

- Selected Functions from the List-Module
  - `List.map ('a -> 'b) -> 'a list -> 'b list`
  - `List.fold_left ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`
  - `List.find_opt ('a -> bool) -> 'a list -> 'a option`
  - `List.filter ('a -> bool) -> 'a list -> 'a list`

T01: List Module Part 2

- `List.map ('a -> 'b) -> 'a list -> 'b list`

1	2	-3	4	5	8

T01: List Module Part 2

- `List.fold_left` `('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`

1	2	-3	4	5	8

T01: List Module Part 2

- `List.find_opt` `('a -> bool) -> 'a list -> 'a option`

1	2	-3	4	5	8

T01: List Module Part 2

- `List.filter` `('a -> bool) -> 'a list -> 'a list`

1	2	-3	4	5	8

T02: Mappings

Idea: Create a Dictionary-Datastructure

```
age_dictionary = {
    "John": 25,
    "Mary": 20,
    "Tom": 30
}
```

1. Implement these functions to work with mappings based on associative lists:
   1. **is_empty** 0 of 1 tests passing
      `is_empty : ('k * 'v) list -> bool`
   2. **get** 0 of 1 tests passing
      `get : 'k -> ('k * 'v) list -> 'v option`

      If the key is mapped to multiple values, return the first such value
   3. **put** 0 of 1 tests passing
      `put : 'k -> 'v -> ('k * 'v) list -> ('k * 'v) list`

      If the key is already mapped to one or more values, remove those pairs first
   4. **contains_key** 0 of 1 tests passing
      `contains_key : 'k -> ('k * 'v) list -> bool`
   5. **remove** 0 of 1 tests passing
      `remove : 'k -> ('k * 'v) list -> ('k * 'v) list`

      If the key is mapped to multiple values, remove all such values
   6. **keys** 0 of 1 tests passing
      `keys : ('k * 'v) list -> 'k list`
   7. **values** 0 of 1 tests passing
      `values : ('k * 'v) list -> 'v list`

T02: Mappings

- How to store dictionaries?
  - Association Lists
  - Functional mapping

```
assoc_list = [
    ("John", 25),
    ("Mary", 20),
    ("Tom", 30)
]
```

```
func_map = fun x-> { 25 wenn x = "John"
                   { 20 wenn x = “Mary"
                   { 30 wenn x = “Tom"
                   { expr sonst
```

T02: Functional Mappings

- Every layer saves **exactly** one datapoint
  - If the parameter matches the datapoint -> return its value
  - Else delegate to sub-function

```
func_map = fun x->
    25 wenn x = "John"
    20 wenn x = “Mary"
    30 wenn x = “Tom"
    expr sonst
    x sonst
    x sonst
```


*Green brackets mean function evaluation with parameter x

T03: Operator Functions

In OCaml, infix notation of operators is just syntactic sugar for a call to the corresponding function. For example, the binary addition `+` merely calls the function `(+) : int -> int -> int`.

1. Discuss why this is a very useful feature.

Note: This is a tutorial exercise, you do not need to submit anything for this exercise.

FPV Tutorübung

Woche 8

OCaml: Tail Recursion, Lazy Lists, Partial Application

Manuel Lerchner

14.06.2023

T01: Tail Recursion 1

a)

```
let rec f a = match a with
  | [] -> a
  | x::xs -> (x + 1)::f xs
```

b)

```
let rec g a b =
  if a = b then 0
  else if a < b then g (a + 1) b
  else g (a - 1) b
```

c)

```
let rec h a b c =
  if b then h a (not b) (c * 2)
  else if c > 1000 then a
  else h (a + 2) (not b) c * 2
```

d)

```
let rec i a = function
  | [] -> a
  | x::xs -> i (i (x,x) [x]) xs
```

1. Decide which of the following functions are implemented tail recursively:

T01: Tail Recursion 2

2. Write tail recursive versions of the following functions (without changing their types). In addition to the definition from the lecture, all functions must use constant stack space ($\mathcal{O}(1)$ in the size of its input). In particular, all the helper functions used need to be tail-recursive and use constant stack space too! If you use a library function, check that the documentation (e.g. for List) marks it as tail-recursive, or when in doubt implement a tail-recursive version yourself!

- Tipp: Use accumulator variables and helper functions

a)

```
let rec fac n =
  if n = 0 then 1
  else n * fac (n - 1)
```

b)

```
let rec remove a = function
  | [] -> []
  | x::xs -> if x = a then remove a xs else x::remove a xs
```

c)

```
let rec partition p l = match l with
  | [] -> [],[]
  | x::xs ->
    let a,b = partition p xs in
    if p x then x::a,b else a,x::b
```

T02: Lazy List Idea

```python
# Classical: Calculate all values at once and return them as a list
def powers_of_two(n):
    return [2 ** i for i in range(n)]


my_powers = powers_of_two(10)

print(my_powers)
# [1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

```python
# Generator: Calculate the values on demand, one by one
def powers_of_two_generator(i):
    while True:
        yield 2 ** i
        i += 1


generator = powers_of_two_generator(0)


for i in range(10):
    print(next(generator))

# 1
# 2
# 4
# 8
# 16
# 32
# 64
# 128
# 256
# 512
# ... and so on
```

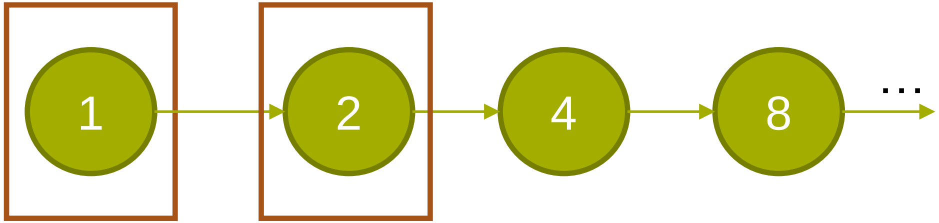
T02: Lazy List

Infinite data structures (e.g. lists) can be realized using the concept of **lazy evaluation**. Instead of constructing the entire data structure immediately, we only construct a small part and keep us a means to construct more on demand.

```
type 'a llist = Cons of 'a * (unit -> 'a
llist)
```

```
int -> int llist
let rec powers_of_2 i =
  Cons (pow 2 i, fun () → powers_of_2 (i + 1))
```

1, fun ()->**n**   **n**



T02: Lazy List

1. **lnat** 0 von 1 Tests bestanden
   Implement the function `lnat : int -> int llist` that constructs the list of all natural numbers starting at the given argument.
2. **lfib** 0 von 1 Tests bestanden
   Implement the function `lfib : unit -> int llist` that constructs a list containing the Fibonacci sequence.
3. **ltake** 0 von 1 Tests bestanden
   Implement the function `ltake : int -> 'a llist -> 'a list` that returns the first n elements of the list.
4. **lfilter** 0 von 1 Tests bestanden
   Implement the function `lfilter : ('a -> bool) -> 'a llist -> 'a llist` to filter those elements from the list that do not satisfy the given predicate.

```
type 'a llist = Cons of 'a * (unit -> 'a llist)
```

```
int -> int llist
let rec powers_of_2 i =
  Cons (pow 2 i, fun () → powers_of_2 (i + 1))
```

T02: Lazy List

```
type llist<T> = [T, () => llist<T>];

function fibonaci_generator(): llist<number> {
  function fib_step(a: number, b: number): llist<number> {
    return [a, () => fib_step(b, a + b)];
  }

  return fib_step(0, 1);
}

let fibonacci_numbers = fibonaci_generator();

for (let i = 0; i < 10; i++) {
  let [value, next_generator] = fibonacci_numbers;

  console.log(value);

  fibonacci_numbers = next_generator();
}

// [0 1]
//  0 [1 1]
//  0  1 [1 2]
//  0  1  1 [2 3]
//  0  1  1  2 [ ... ]

```

```
type 'a llist = Cons of 'a * (unit -> 'a
llist)
```

T03: Partial Application

Types of (apparently) n-ary functions are denoted as `arg_1 -> ... -> arg_n -> ret` in OCaml.

1. Discuss, why this notation is indeed meaningful.

2. Give the types of these expressions and discuss to what they evaluate:

```
let a (* : todo *) = (fun a b -> (+) b)

let b (* : todo *) = (fun a b -> List.fold_left b 1 (List.map ( * ) a))

let c (* : todo *) = (fun a b c -> c (a + b)) 3

let d (* : todo *) = (fun a b c -> b (c a) :: [a]) "x"

let e (* : todo *) = (let x = List.map in x (<))
```

T03: Partial Application

Types of (apparently) n-ary functions are denoted as `arg_1 -> ... -> arg_n -> ret` in OCaml.

1. Discuss, why this notation is indeed meaningful.
2. Give the types of these expressions and discuss to what they evaluate:

```
let a (* : todo *) = (fun a b -> (+) b)

let b (* : todo *) = (fun a b -> List.fold_left b 1 (List.map ( * ) a))

let c (* : todo *) = (fun a b c -> c (a + b)) 3

let d (* : todo *) = (fun a b c -> b (c a) :: [a]) "x"

let e (* : todo *) = (let x = List.map in x (<))
```

T03: Partial Application

```
let a (* : todo *) = (fun a b -> (+) b)
```

T03: Partial Application

```
let b (* : todo *) = (fun a b -> List.fold_left b 1 (List.map ( * ) a))
```

T03: Partial Application

```
let c (* : todo *) = (fun a b c -> c (a + b)) 3
```