Introduction to Program Synthesis - Exercise Sheet 3 Computational Graphs and Call Stacks

Deadline: 02-07-2025 (Anywhere on Earth)

In these exercises, you will set up data structures which allow you to represent computer programs. Remember to keep your code modular and easily expandable, to re-use it in future exercises.

Exercise 1: Computational Graphs

Extend your code from exercise sheet 1 to support computational graphs. Be aware that, unlike trees, computational graphs may have more than one connected component, but may not contain loops, as per the definition given to you in the slides (Chapter 2.2, Slide 10).

(1) Research and implement an algorithm that checks and ensures the acyclic property of your graph when inserting nodes.

(2) Use your graph implementation to calculate the following equation, given two integers x and y:

$$x^2 + x \cdot y - 4.$$

Run your program for a few test values to ensure it is working correctly.

(3) Implement a second graph that calculates the same equation from 1.2, but ensures that $x \geq y$. In case of $y > x$, it first swaps the values of the two variables before calculating the equation.

While computational graphs allow you to model a program's control flow, they are yet another abstraction of what really happens inside of a processor. **Call stacks** are another step closer to the instruction-based designs implemented in modern processing units.

Exercise 2: Call Stacks

In a new python script, implement a data structure for a call stack, as introduced to you in chapter 2 of the course material. It must support the basic functions `push`, `pop`, and `peek`. Use your call stack implementation to run the following program:

$x \leftarrow 3$
$y \leftarrow 5$
$z \leftarrow x \cdot y$
if $z \leq 20$ **then**
    print("Hello")
else
    print("World")
end if

Exercise 3: Dynamic Program Flow

The Call Stack data structure also allows your function calls to dynamically affect the flow of your program. If needed, extend your code such that your functions are able to `push` to

the call stack, and use it to implement the following program:

```
x ← 5
while x > 0 do
    print(x)
    if x  mod 2 = 0 then
        print("Even")
    else
        print("Odd")
    end if
    x ← x − 1
end while
```

Consider whether your call stack would look differently if you used a for-loop in place of the while-loop.