

Exercise 1 (Procedure Stack):

(1 Points)

Which of the following procedure stacks could be result of the execution of an EPL-programm? Why?

- a) $p_1 = 13 : 3 : 9 : 1 : 4 : 3 : 2 : 2 : 4 : 5 : 5 : 15 : 1 : 3 : 2 : 12 : 0 : 0 : 0 : 17 : 3$
- b) $p_2 = 13 : 3 : 9 : 1 : 4 : 3 : 2 : 2 : 5 : 4 : 5 : 15 : 1 : 3 : 2 : 12 : 0 : 0 : 0 : 17 : 3$
- c) $p_3 = 13 : 3 : 9 : 1 : 4 : 3 : 2 : 2 : 8 : 4 : 5 : 15 : 1 : 3 : 2 : 12 : 0 : 0 : 0 : 17 : 3$

Lösung: _____

- a) No valid procedure stack as 4 : 5 have to be sl, dl. 4 would then point to the last local variable of the procedure.
- b) Valid.
- c) The contained 8 is a sl and points to the I/O-frame, but is not the sl of the MAIN-frame.

Exercise 2 (Abstract machine execution):

(2 Points)

Consider the following intermediate code:

```

⋮
7:LOAD(1, 2);    (dif, off)
8:ADD;
9:RET;
⋮
26:CALL(38, 1, 3);(ca, dif, loc)

```

Give the next four states of the abstract machine starting in:

$$(ca, d, p) := (7, -3, 9 : 4 : 26 : 3 : 7 : 4 : 3 : 36 : 5 : 10 : 4 : 40 : 1 : 2 : \dots)$$

Recall that the procedure stack has the form:

sl	dl	ra	v ₁	...	v _n	...
----	----	----	----------------	-----	----------------	-----

and the *base*-function is defined as:

$$\begin{aligned}
 base(p, 0) &:= 1 \\
 base(p, dif + 1) &:= base(p, dif) + p.base(p, dif)
 \end{aligned}$$

Lösung:

$p = 9 : 4 : 26 : 3 : 7 : 4 : 3 : 36 : 5 : 10 : 4 : 40 : 1 : 2 : \dots$ and $p' = 4 : 3 : 36 : 5 : 10 : 4 : 40 : 1 : 2 : \dots$

7 : $\llbracket \text{LOAD}(1, 2); \rrbracket$ $(7, -3, p) \rightarrow (8, -3 : (p.14), p)$
 8 : $\llbracket \text{ADD}; \rrbracket$ $(8, -3 : 2, p) \rightarrow (9, -1, p)$
 9 : $\llbracket \text{RET}; \rrbracket$ $(9, -1, p) \rightarrow 26, -1, p'$

26 : $\llbracket \text{CALL}(38, 1, 3); \rrbracket$ $(26, -1, p') \rightarrow (38, -1, 10 : 5 : 27 : 0 : 0 : 0 : p')$

Exercise 3 (Translation function):

(3 Points)

In addition to `while`-loops we want to have `for`-loops with implicit declaration of the counter variable in our example programming language:

`for (var l := A ; B ; C1) C2`

- Extend the translation function *ct* accordingly.
- Generate intermediate code for

`for (var x := 0; x < 10; x := x + 1) P();`

without parameters for the `CALL` instruction generated for `P()`.

Lösung:

- We assume that the variable *X* is declared. It could be done where the current frame is declared. So we already have *update(var l, st, l)*, where *st* is the symbol table and *l* is the current level.

$ct(\text{for } (var\ X := A ; B ; C_1) C_2) = ct(X := A, st, a, l)$
 $= bt(B, st, a_1, l)$
 $= a_2 : JFALSE(a_4 + 1)$
 $= ct(C_2; C_1, st, a_3, l)$
 $= a_4 : JMP(a_1)$

- We assume the following parameters: $st(x) = (var, l, off)$ and $st(P) = (proc, ca, l', loc)$.

```
1: LIT (0);
2: STORE (0, off);
3: LOAD (0, off);
4: LIT (10);
5: LT;
6: JFALSE (13);
7: CALL (ca, l - l', loc);
8: LOAD (0, off);
9: LIT (1);
10: ADD;
11: STORE (0, off);
12: JMP (3);
```

Exercise 4 (Code generation):**(4 Points)**

Implement `generator.Generator.translateWHILE(AST)` which given an abstract syntax tree returns Jasmin Code in a string. *Hint: It is a good approach to write methods for every language construct and call them recursively. Once you get the idea, it is actually less effort than you might think!*

To test your implementation you can write code in *WHILE* run it through our compiler, save the output and use Jasmin to build executable Java class files. The latter you can execute and observe their behaviour.

This exercise concludes the implementation of our "While to Jasmin" compiler!

Lösung:
