Lernziele:

Das primäre Ziel ist weiterhin, den Workflow

"Bearbeiten (und speichern) -> Übersetzen -> Ausführen"

zu üben (sofern beim Übersetzen keine Fehler aufgetreten sind).

Das sekundäre Ziel ist die Sensibilisierung für folgende Konzepte:
- Lebenszeit von lokalen und globalen Variablen.
- Parameterübergabe nach dem Prinzip Call-by-Value.
- Bedeutung des Stacks für Funktionsaufrufe und lokale Variablen.

Wie in den vorherigen Übungen werden außerdem neue Features der ABC-Programmiersprache anhand von Beispielen eingeführt.

Schritt 1

Verwende die Shell, um die Dateien zur Vorlesung in Unterverzeichnissen zu organisieren. Verwende beispielsweise im Heimatverzeichnis die Unterverzeichnisse **hpc0/session1/**, **hpc0/session2/**, **hpc0/session3/**.

Schritt 2 (Lebenszeit einer Variable)

Variablen haben nicht nur einen Scope (an welcher Stelle kann auf eine Variable zugegriffen werden), sondern auch eine sogenannte Lebensdauer (wann kann auf eine Variable zugegriffen werden):
- Eine globale Variable existiert während der gesamten Laufzeit eines Programms, d.h. vom Moment des Programmstarts bis zum Programmende.
- Eine lokale Variable existiert nur während der Ausführung eines Funktionsaufrufs:
  - Die Lebensdauer beginnt mit dem Betreten der Funktion.
  - Die Lebensdauer endet mit dem Verlassen der Funktion.

```
1 @ <stdio.hdr>
2
3 global a: int = 42;
4
5 fn foo()
6 {
7     local b: int;
8     printf("foo entered: b = %d\n", b);
9     b = a;
10    printf("foo leaving: b = %d\n\n", b);
11 }
12
13 fn main()
14 {
15    // printf("calling foo:\n");
16    foo();
17
18    // printf("calling foo:\n");
19    foo();
20
21 }
```

Aufgaben

1. Übersetzt das rechts abgebildete Programm **ex3.abc** und führt es aus. Auf meinem Rechner erhalte ich folgende Ausgabe:

```
MCL:tmp lehn$ ./a.out
foo entered: b = 1
foo leaving: b = 42

foo entered: b = 42
foo leaving: b = 42
```

   Bei euch kann diese anders aussehen, denn wie wir sehen werden, haben wir es hier mit "undefiniertem Laufzeitverhalten" zu tun.

2. Wenn man nun genau den Kontrollfluss des Programms und dessen Ausgabe nachvollzieht, dann scheint dies der obigen Aussage über die Lebenszeit lokaler Variablen zu widersprechen. Geht das Programm zeilenweise durch und macht euch klar, in welchem Augenblick welche Ausgabe erzeugt wird.

   Hier ein paar Stellen, die man dabei ganz genau beobachten sollte:
   (A) In Zeile 8 wird der Wert von Variable **b** ausgegeben, bevor diese initialisiert wurde. Welcher Wert beim ersten Funktionsaufruf von **foo** ausgegeben wird (bei mir 1), ist also nicht definiert.
   (B) In Zeile 9 wird der Wert von **b** mit dem Wert von **a** (also mit 42) überschrieben.
   (C) Beim zweiten Funktionsaufruf wird in Zeile 8 wieder der Wert von **b** ausgegeben, bevor die Variable initialisiert wurde. Bei mir wird jetzt der Wert 42 ausgegeben. Das ist der Wert, den die Variable am Ende des letzten Funktionsaufrufs hatte. Es scheint also, dass die lokale Variable **b** den letzten Funktionsaufruf "überlebt" hat.
3. Entfernt die Kommentare in Zeile 15. Übersetzt das Programm nochmals und führt es aus. Es werden dann zwei zusätzliche Zeilen ausgegeben. Insbesondere sollte sich aber die Ausgabe von Funktion foo geändert haben. Ich erhalte zum Beispiel:

```
MCL:tmp lehn$ ./a.out
calling foo:
foo entered: b = 0
foo leaving: b = 42

calling foo:
foo entered: b = 0
foo leaving: b = 42
```

   Hier kann man also beobachten, dass die Variable b den letzten Funktionsaufruf doch nicht überlebt hat, sondern beim zweiten Funktionsaufruf neu angelegt worden ist. Es scheint aber so, dass diese neu angelegte Variable jetzt auch mit Null initialisiert wurde.
4. Übersetzt das Programm nochmals, diesmal aber mit aktivierter Optimierung. Zum Beispiel mit:

```
MCL:tmp lehn$ abc -O1 ex3.abc
```

   Bei mir erhalte ich dann folgende Ausgabe:

```
calling foo:
foo entered: b = 73896
foo leaving: b = 42

calling foo:
foo entered: b = 73896
foo leaving: b = 42
```

   Jetzt scheint es, dass die Variable **b** jedes Mal mit dem Wert 73896 initialisiert wird. Kommentiert man die **printf**-Anweisungen in Zeile 15 und Zeile 18 aus, dann erhalte ich bei mir aber:

```
MCL:tmp lehn$ abc -O1 ex3.abc
MCL:tmp lehn$ ./a.out
foo entered: b = -1148205632
foo leaving: b = 42

foo entered: b = 73896
foo leaving: b = 42
```

   Fazit:
   a. Lokale Variablen überleben einen Funktionsaufruf nicht.
   b. Lokale Variablen werden bei jedem Funktionsaufruf neu angelegt und haben ohne explizite Initialisierung einen undefinierten Wert.
   c. Gibt man einen undefinierten Wert aus, dann kann die Ausgabe abhängen vom Kalendertag, der Uhrzeit, dem persönlichen Empfinden und dem Wetter.

Schritt 3 (Call by Value)

Übergibt man einer Funktion ein Argument, dann bekommt die Funktion eine Kopie des Arguments. Nach dem Funktionsaufruf ist das übergebene Argument also stets unverändert, auch dann, wenn die aufgerufene Funktion das erhaltene Argument überschreibt.

```
1 @ <stdio.hdr>
2
3 fn foo(n: int)
4 {
5     printf("foo entered: n = %d\n", n);
6     n = 42;
7     printf("foo leaving: n = %d\n", n);
8 }
9
10 fn main()
11 {
12     local n: int = 123;
13
14     printf("main: n = %d\n", n);
15     printf("calling foo:\n");
16     foo(n);
17     printf("returned from foo\n");
18     printf("main: n = %d\n", n);
19 }
```

Aufgabe:

1. Übersetzt das Programm und führt es aus. Geht das Programm wieder zeilenweise durch, um nachzuvollziehen, in welcher Zeile welche Ausgabe erzeugt wird. Bestätigt damit die Behauptung, dass eine Funktion nur eine Kopie (und nicht das Original) des Arguments erhält.
2. Um klar zu machen, dass der Aufrufer (Caller) einer Funktion das Argument anders nennen kann als der Aufgerufene (Callee), macht folgende Änderung: Nennt in Funktion main die lokale Variable um in x.

Schritt 4 (Verwendung eines Stacks bei Funktionsaufrufen)

Ein Stack (Stapel) ist eine Datenstruktur, auf die man zwei Operationen anwenden kann:

- Mit einer Push-Operation kann man Daten auf den Stack legen.
- Mit einer Pop-Operation kann man Daten (von oben) wieder entfernen.

Diese Datenstruktur wird verwendet, um die Lebenszeit von lokalen Variablen zu verwalten. Wir werden später sehen, dass diese Datenstruktur auch dazu verwendet wird, um festzuhalten, an welche Stelle des Programms eine Funktion zurückkehren soll.

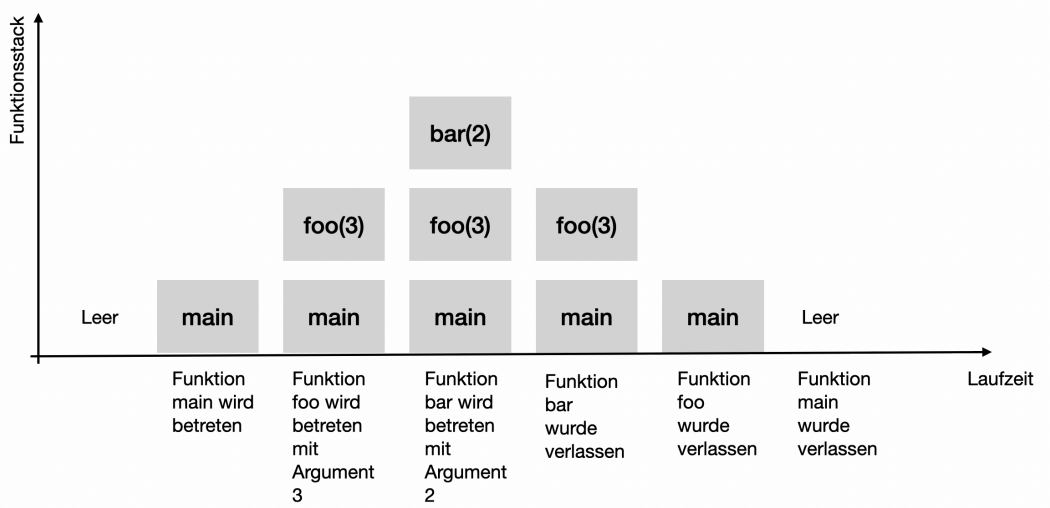
Code für das Verwenden dieser Datenstruktur wird vom Compiler erzeugt und ist damit im ausführbaren Programm enthalten. In den Bottom-Up-Sessions dieser Veranstaltungen werden wir diesen Code selbst in Assembler schreiben und damit genauer verstehen, was der Compiler für uns erzeugt. Hier geht es darum, nachzuvollziehen, was dieser Code macht. Dazu verwenden wir das rechts dargestellte Beispielprogramm.

```
1 @ <stdio.hdr>
2
3 fn bar(n: int)
4 {
5     printf("bar entered: n = %d\n", n);
6 }
7
8 fn foo(n: int)
9 {
10     printf("foo entered: n = %d\n", n);
11     bar(n - 1);
12     printf("foo leaving: n = %d\n", n);
13 }
14
15 fn main()
16 {
17     foo(3);
18 }
```

Aufgabe:

1. Übersetzt das Programm und führt es aus. Geht das Programm wieder zeilenweise durch, um nachzuvollziehen, in welcher Zeile welche Ausgabe erzeugt wird. Macht euch dabei zum Beispiel klar, dass die lokale Variable **n** in Funktion **foo** auch nach dem Aufruf von Funktion **bar** noch existiert und einen unveränderten Wert hat. Macht euch ebenfalls klar, dass Funktion **bar** ebenfalls eine Variable **n** hat, dass dies aber nicht dieselbe Variable wie die von Funktion **foo** ist.

2. In folgender Abbildung wird veranschaulicht, wie ein Stack zur Verwaltung der Funktionsaufrufe verwendet wird:



Bei jedem Funktionsaufruf wird eine Box angelegt mit den relevanten Daten. Diese relevanten Daten umfassen insbesondere die lokalen Variablen (einschließlich der Funktionsargumente) und, wie oben erwähnt, die Rücksprungadresse. Diese Boxen werden gestapelt:

(A) Beim Programmstart ist der Stapel zunächst leer.
(B) Wenn die Funktion main betreten wird, wird eine Box angelegt und auf den Stapel gelegt.
(C) Wenn in main die Funktion foo mit dem Argument 3 aufgerufen wird, wird eine neue Box angelegt und auf den Stapel gelegt.
(D) Wenn in foo die Funktion bar mit dem Argument 2 aufgerufen wird, wird eine neue Box angelegt und auf den Stapel gelegt.
(E) Wenn bar verlassen wird, wird die oberste Box wieder entfernt. Wenn die Programmausführung in Funktion foo fortgesetzt wird, enthält die oberste Box also den Zustand vor dem Funktionsaufruf.
(F) Und so weiter.

Mit solchen Schaubildern kann man natürlich auch die Verwendung des Stacks bei rekursiven Funktionsaufrufen veranschaulichen. Macht dies für das rechts abgebildete Programm. In den Boxen sollen dabei auch die Werte der lokalen Variablen eingetragen werden. Unten ist ein nicht-fertiges Diagramm abgebildet, das diese für die ersten paar Funktionsaufrufe zeigt.

```
1 @ <stdio.hdr>
2
3 fn foo(n: int): int
4 {
5     local res: int = n;
6
7     if (n > 1) {
8         res += foo(n - 1);
9     }
10    return res;
11 }
12
13 fn main()
14 {
15    printf("foo(%d) = %d\n", 3, foo(3));
16 }
```