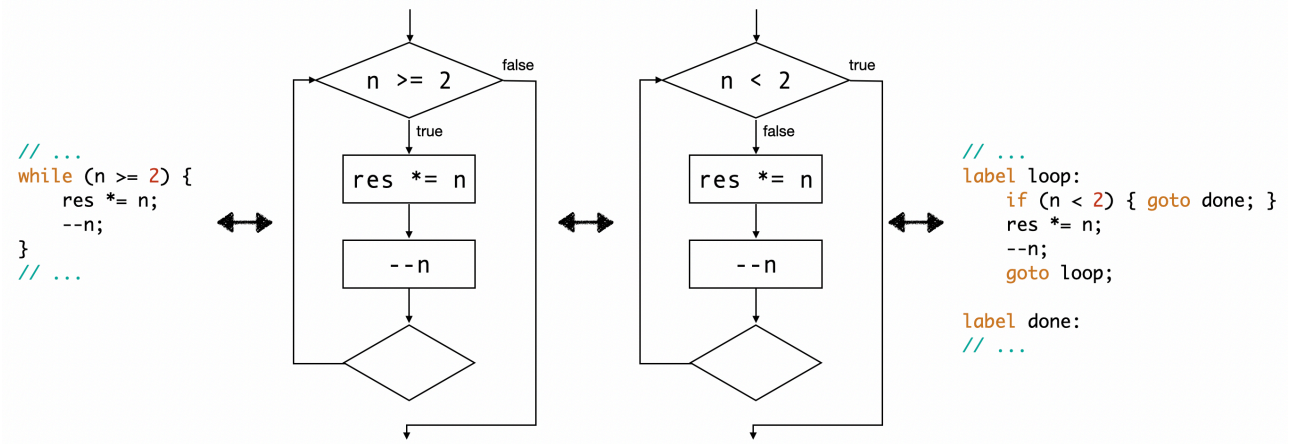
Lernziele:
- Technisches Verständnis für den Kontrollfluss/Programmablauf
- Umgang mit Arrays
- Technisches Verständnis für die Verwendung eines Stacks bei lokalen Variablen

Schritt 1: Beim Top-Down-Ansatz geht es jetzt wirklich "Down"
In der Vorlesung wurde gezeigt, wie Flow-Charts verwendet werden können, um den Programmablauf innerhalb einer Funktion zu beschreiben. Dabei wurden folgende Konventionen verwendet:
- Bedingte Sprünge werden mit einer Raute dargestellt, in der die Bedingung dargestellt wird.
- Unbedingte Sprünge werden mit einer leeren Raute dargestellt.
- Alle anderen Anweisungen werden mit einem Rechteck dargestellt, das die Anweisung enthält.
- Alle Routen und Rechtecke sind vertikal angeordnet. Damit wird ausgedrückt, dass im zugehörigen Maschinen-Code die Anweisungen nacheinander im Speicher liegen, was die sequenzielle Ausführung der Anweisungen repräsentiert. Mit Pfeilen wird visualisiert, in welche Reihenfolge die Anweisungen ausgeführt werden. Die Ausführung beginnt dabei immer oben.

Um besser zu verstehen, wie Kontrollstrukturen auf primitive Maschinenbefehle abgebildet werden können, wurde zudem beschrieben, wie eine While-Schleife mit Goto-Anweisungen realisiert werden kann:



Vergleicht man den Flow-Chart mit dem Code, der Goto-Anweisungen enthält, erkennt man, dass folgende Regeln gelten:
- Eine leere Route wird mit einer Goto-Anweisung realisiert.
- Eine Route mit Bedingung wird mit einem Konstrukt der Form "if (…) { goto … }" realisiert, das informell als "Bedingtes Goto" bezeichnet wird.

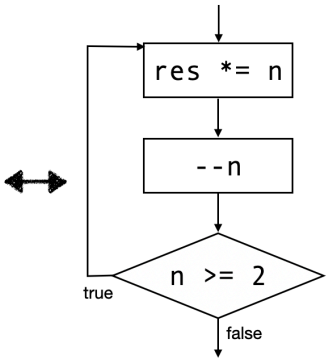
Aufgaben:
1. Regeln, um Kontrollstrukturen mit Goto-Anweisungen ausdrücken zu können, sollt ihr selbst herleiten, indem ihr verschiedene Beispiele betrachtet. Als erstes Beispiel sollt ihr dafür folgendes Programm (File: **if.abc**) verwenden:

```
@ <stdio.hdr>

fn main()
{
    local ch: int = getchar();

    if (ch == 'A') {
        printf("I got an A!\n");
    } else {
        printf("Not an A :(\n");
    }
}
```

Testet das Programm zunächst mit verschiedenen Eingaben. Beschreibt das Programm dann mit einem Flow-Chart. Schreibt den Flow-Chart dann so um, dass Goto-Anweisungen verwendet werden können, und realisiert dann natürlich diesen Flow-Chart in einem Programm (File: **if_goto.abc**).

2. Eine Do-While-Schleife kann mit folgendem Flow-Chart beschrieben werden:

```
// ...
do {
    res *= n;
    --n;
} while (n >= 2);
// ...
```



Folgendes Programm (File: **do_while.abc**) liest solange ein Zeichen ein und gibt es aus, bis es einen Punkt '.' erhält:

```
@ <stdio.hdr>

fn main()
{
    local ch: int;

    do {
        ch = getchar();
        printf("got '%c' (ASCII %d)\n", ch, ch);
    } while (ch != '.');
}
```

Testet das Programm und bestätigt, dass das letzte Zeichen (das Zeichen '.') ebenfalls mit ausgegeben wird. Beschreibt das Programm zunächst wieder mit einem Flow-Chart und schreibt es dann so um (File: **do_while_goto.abc**), dass Goto-Anweisungen verwendet werden.

3. Wiederholt für folgendes Programm (File: for.abc) das gesamte Vorgehen, d.h. Testen, Flow-Chart und in ein Programm (File: **for_goto.abc**) umschreiben, das Goto-Anweisungen verwendet:

```
@ <stdio.hdr>

fn main()
{
    local n: int = 5;
    local res: int = 1;

    for (local i: int = 1; i <= n; ++i) {
        res *= i;
    }
    printf("res = %d\n", res);
}
```

Schritt 2: Stack und undefiniertes Verhalten

In der Vorlesung wurde ein Array verwendet, um einen einfachen Stack zu implementieren, der maximal 5 Integer-Werte speichern kann. Den Programm-Code (File: **stack.abc**) findet ihr am Ende der Aufgabe.

Aufgaben:

1. Ändert die Anweisungen in **main** so um, dass zunächst zwei Push-Operationen und dann drei Pop-Operationen ausgeführt werden. Bestätigt, dass sich das Programm übersetzen lässt, es jedoch bei der Ausführung zu einem Laufzeitfehler kommt.
2. Ändert die Anweisungen in **main** so um, dass zunächst sechs Push-Operationen und dann drei Pop-Operationen ausgeführt werden. Bestätigt wieder, dass der Fehler erst zur Laufzeit erkannt wird.
3. Kommentiert in den Funktionen **push** und **pop** jeweils die Assertions und wiederholt die vorherigen zwei Teilaufgaben. Ob es zu einem Fehler kommt oder das Programm scheinbar korrekt ausgeführt wird, kann nicht vorhergesagt werden. Höchstwahrscheinlich kommt es jedoch zu einem Fehler, wenn ihr die Anweisungen in **main** so ändert, dass eine Pop-Operation und dann eine Push-Operation ausgeführt wird.
4. Entfernt die Kommentare für die Assertions in **push** und **pop**.

```
@ <stdio.hdr>

global
    stack: array[5] of int,
    stackSize: int = 0;

fn push(val: int)
{
    assert(stackSize < sizeof(stack) / sizeof(stack[0]));
    stack[stackSize++] = val;
}

fn pop(): int
{
    assert(stackSize > 0);
    return stack[--stackSize];
}

fn main()
{
    push(3);
    push(4);
    pop();
}
```

Schritt 3: Zugriff auf die Stack-Spitze

Das Programm soll nun so ergänzt werden, dass mit einer Funktion **top** der Wert des obersten Stack-Elements ausgelesen werden kann, ohne es zu entfernen. In **main** kann die Funktion dann beispielsweise so verwendet werden:

```
fn main()
{
    push(3);
    printf("top = %d\n", top());
    push(4);
    printf("top = %d\n", top());
    pop();
    printf("top = %d\n", top());
}
```

Dies sollte folgende Ausgabe erzeugen:

```
lehn$ ./a.out
top = 3
top = 4
top = 3
```

Schritt 4 (Darstellung des Stack-Inhaltes)

Um den gesamten Stack-Inhalt zu sehen, kann folgende Funktion **info** verwendet werden:

```
fn info()
{
    printf("stack: ");
    for (local i: int = 0; i < stackSize; ++i) {
        printf("[%d] ", stack[i]);
    }
    printf("\n");
}
```

Ändert die Anweisungen in main so ab, dass nach jeder Push-/Pop-Operation das oberste Element und der gesamte Stack-Inhalt angezeigt wird und beispielsweise folgende Ausgabe erzeugt wird:

```
lehn$ ./a.out
top = 3
stack: [3]
top = 4
stack: [3] [4]
top = 3
stack: [3]
```

Schritt 5 (Darstellung der freien Array-Elemente)
Erweitert die Funktion **info** so, dass auch die unbenutzten/verfügbaren Array-Elemente ausgegeben werden. Dabei sollen wie zuvor die Array-Elemente ausgegeben werden, die auf dem Stack liegen. Dann sollen nach einem senkrechten Strich die noch verfügbaren Array-Elemente ausgegeben werden.

Beispielsweise sollten die Operationen **push(3)**, **push(4)** und dann **pop()** zu folgender Ausgabe führen:

```
MCL:tmp lehn$ ./a.out
top = 3
stack: [3] | [0] [0] [0] [0]
top = 4
stack: [3] [4] | [0] [0] [0]
top = 3
stack: [3] | [4] [0] [0] [0]
```

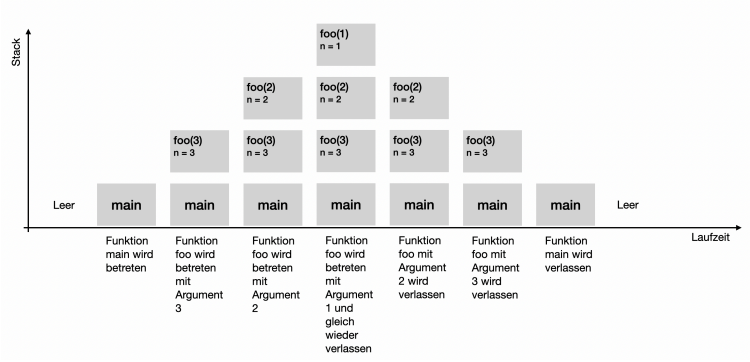
Durch diese Ausgabe wird deutlich, dass bei einer Pop-Operation ein Array-Inhalt nicht "gelöscht" oder überschrieben wird, sondern nur der Stack-Zeiger dekrementiert wird.

Beachtet dabei folgendes: Im Programm ist **stack** als ein Array mit Dimension 5 definiert. Sollte diese Dimension geändert werden (zum Beispiel auf 10), dann sollte es nicht notwendig sein, die Funktion **info** zu ändern. Die Funktion **info** soll sich automatisch der neuen Dimension des **stack**-Arrays anpassen und trotzdem funktional bleiben.

Schritt 6: Verwendung des Stack zur Paramterübergabe
Blatt 3 behandelte die Lebenszeit von lokalen Variablen. Dabei wurde erklärt, wie der Stack verwendet wird, um diese Lebenszeit zu verwalten:
- Wird eine Funktion aufgerufen, werden auf dem Stack alle lokalen Variablen für diesen Funktionsaufruf neu angelegt.
- Wird die Funktion verlassen, werden die lokalen Variablen vom Stack entfernt.

Betrachten wir dazu folgendes Beispiel, bei dem die Funktion **foo** nur eine lokale Variable, nämlich den Funktionsparameter **n**, besitzt:

```
@ <stdio.hdr>

fn foo(n: int)
{
    printf("entered foo with n = %d\n", n);
    if (n > 1) {
        foo(n - 1);
    }
    printf("leaving foo with n = %d\n", n);
}

fn main()
{
    foo(3);
}
```



Wie lokale Variablen bei einem Funktionsaufruf auf dem Stack angelegt und beim Verlassen freigegeben werden, soll anhand dieses Beispiels mit unserem eigenen Stack nachgebildet werden. Führen Sie dazu folgende Änderungen durch:
- Aus einem Funktionsaufruf wie "foo(3)" werden zwei Anweisungen:
```
push(3); // Parameter auf den Stack legen
foo();   // und dann die Funktion aufrufen
```
- Die Deklaration der Funktion **foo** ist jetzt also parameterlos. Anstatt auf **n** wird nun auf das oberste Element des Stacks mit Hilfe von **top()** zugegriffen.
- Bevor die Funktion **foo** verlassen wird, wird der Parameter vom Stack mit einer Pop-Operation entfernt.
- Verwendet die Funktion **info**, um den Stack beim Betreten und Verlassen der Funktion **foo** darzustellen.