KONZEPTE SYSTEMNAHER PROGRAMMIERUNG

Technische Hochschule Mittelhessen

Andre Rein

– Kontrollstrukturen –

KONTROLLSTRUKTUREN IN NINJA

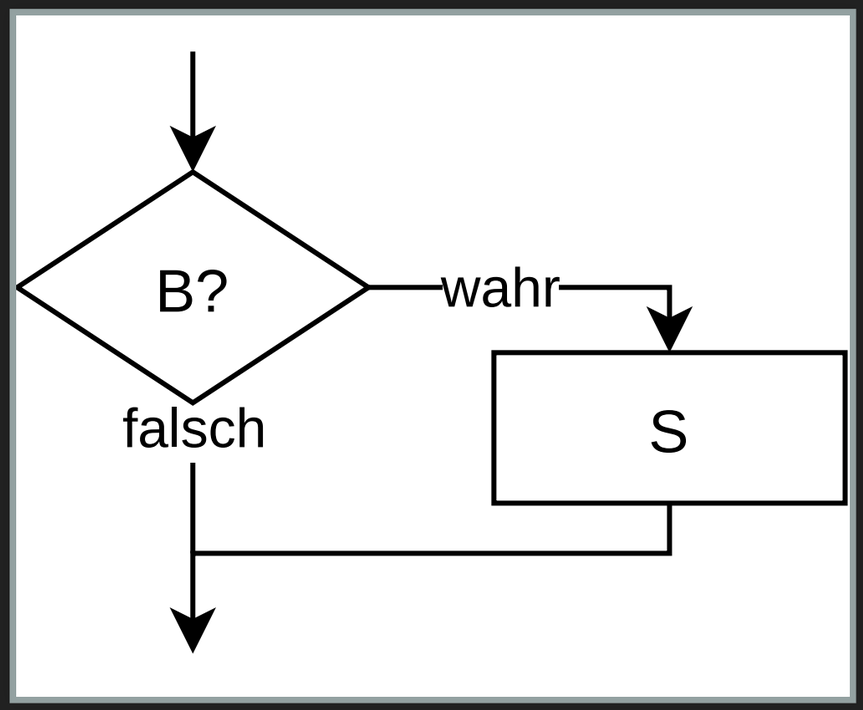
- Unsere Programmabläufe erfolgen bisher immer **strikt sequentiell**:
  - Der Programmzähler (**PC**) wird nach der Ausführung einer Instruktion **immer** um `1` erhöht
- Kontrollstrukturen wie Verzweigungen und Schleifen erfordern aber andere Programmabläufe
  - Es muss also möglich sein, den PC anderweitig anzupassen

KONTROLLSTRUKTUREN IN NINJA: ÜBERSICHT

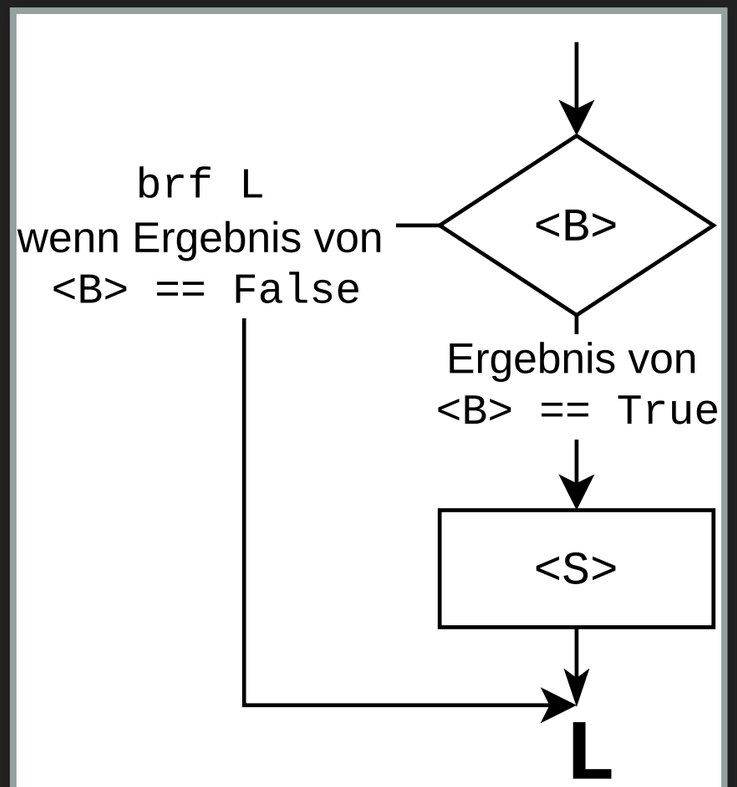
1. **Einarmiges `if`** → führe das Statement `S` aus, **wenn** die Bedingung `B` *wahr* ist!
   - **Beispiel:** `if (B) S`
2. **Zweiarmiges `if`** → führe das Statement S_1 aus, **wenn** die Bedingung `B` *wahr* ist, andernfalls führe das Statement S_2 aus!
   - **Beispiel:** `if (B)` S_1 `else` S_2
3. **`while`-Schleife** → führe das Statement `S` aus, **wenn** und **solange** die Bedingung `B` *wahr* ist! (*kopfgesteuert*)
   - **Beispiel:** `while(B) S`
4. **`do`-Schleife** → führe das Statement `S` aus, **solange** die Bedingung `B` *wahr* ist! (*fußgesteuert*)
   - **Beispiel:** `do S while(B)`

EINARMIGES `if` IN NINJA

- `B` ist z.B. (`a==b && c>25`), d.h. `B` kann ein komplexer Ausdruck sein!



EINARMIGES `if` IN NINJA



- **Notation**: `<B>` ist die Übersetzung des vollständigen Ausdrucks `B` in Ninja Assembler
  - Anmerkung: Nach der Ausführung liegt ein `True` oder `False` auf unserem Stack
- **Notation**: `<S>` ist die Übersetzung aller Anweisungen von `S` in Ninja Assembler
- `L`: Ein sog. Label, d.h. eine Position im Assembler Code, die angesprungen werden kann
- `brf`: Instruktion "**Branch on False**" → Führt Sprung zum Label `L` aus, falls `Ergebnis` von `<B> == False`

```
<B>     // hinterlässt ein Ergebnis True / False auf dem Stack
brf L   // Springt zum sog. Label "L:" wenn Ergebnis == False
<S>     // Anweisungen von S (werden ausgeführt falls Ergebnis == True)
L:      // Label: Generelle Anweisungen
...     // (werden immer ausgeführt)
```

SPRUNGINSTRUKTIONEN

- `brf <n>` → `... value -> ...` — Springe, wenn `value == False`
- `brt <n>` → `... value -> ...` — Springe, wenn `value == True`
- `jump <n>` → `... -> ...` — Springe (ohne Bedingung)!

Die Ausführung von Sprüngen bedeutet, dass der Programmzähler `PC` auf den Wert `<n>` gesetzt wird!

SPRUNGINSTRUKTIONEN (AUSFÜHRUNGSSEQUENZ)

```
while (!halt) {
  IR = program_memory[PC];
  PC = PC + 1;
  execute(IR)
}
```

Eine Ausführung **ohne Sprünge** bedeutet, dass der Programmzähler (`PC`) **vor** der eigentlichen Ausführung `execute(IR)` um `1` inkrementiert wird!

SPRUNGINSTRUKTIONEN: BEISPIEL `jmp 28`

```
while (!halt) {
  IR = program_memory[PC];
  PC = PC + 1;
  execute(IR)
}
```

	Normaler Ablauf		Ablauf mit `jmp 28`
PC = 7	`IR=program_memory[7]`	PC = 7	`IR=program_memory[7]`
PC = 7	`PC = 7 + 1;`	PC = 7	`PC = 7 + 1;`
PC = 8	`execute(IR)`	PC = 8	`execute(IR) // jmp 28`
PC = 8	`IR=program_memory[8]`	PC = 28	`IR=program_memory[28]`
PC = 8	`PC = 8 + 1;`	PC = 28	`PC = 28 + 1;`
PC = 9	`execute(IR)`	PC = 29	`execute(IR)`

Beachten Sie **wann** der PC erhöht wird → Dies darf **nicht nach** der Ausführung (`execute`) erfolgen! (*Häufiger Fehler der zu Fehlverhalten der VM bei Sprüngen führt!*)

ANMERKUNG ZU LABELS

In Ninja Assembler werden Labels mit eine Doppelpunkt gekennzeichnet und markieren eine Adresse im Code. Der Compiler sorgt dafür, dass die Adresse des Labels der Instruktion entspricht, die nach dem definierten Label kommt.

Beispiel: Definition und Sprung zum Label `L1`.

```
// Ninja Assembler
Adresse |  Assembler Code
--------+---------------------------------
[4]     |      ...
[5]     |      pushc 56  // adresse  [5]
        | L1:            // adresse  -.
        |                //          V
[6]     |      add       // adresse  [6]<-.
[7]     |      ...       //              |
[8]     |      ...       //              .
[9]     |      jmp L1    //---------------
[10]    |      ...
[11]    |      ...
```

BOOLESCHE WERTE UND VERGLEICHE

- In C:
  - `False` entspricht dem Wert `0`
  - `True` entspricht einem Wert `!=0` → `-100`, `42`, `1`, `-1`, `...` alles `True`
- In Ninja:
  - `False` entspricht dem Wert `0`
  - `True` entspricht dem Wert `1`
    - Ein Wert ungleich `0` oder `1` stellt einen ungültigen Booleschen Wert in Ninja dar!

BOOLESCHE WERTE UND VERGLEICHE

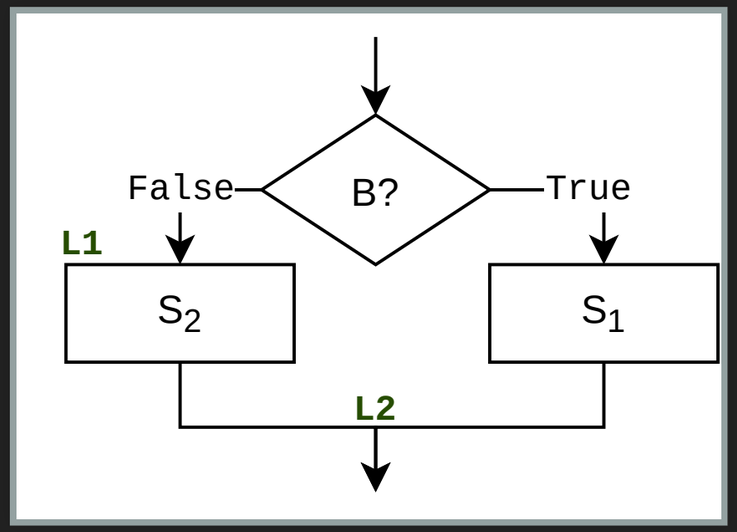
- `a==b`: `eq` → `... a b -> ... (a==b)` — `(a==b)` entspricht `True` (1) oder `False` (0)
- `a!=b`: `ne` → `... a b -> ... (a!=b)` — `(a!=b)` entspricht `True` (1) oder `False` (0)
- `a <b`: `lt` → `... a b -> ... (a<b)` — `(a<b)` entspricht `True` (1) oder `False` (0)
- `a<=b`: `le` → `... a b -> ... (a<=b)` — `(a<=b)` entspricht `True` (1) oder `False` (0)
- `a >b`: `gt` → `... a b -> ... (a>b)` — `(a>b)` entspricht `True` (1) oder `False` (0)
- `a>=b`: `ge` → `... a b -> ... (a>=b)` — `(a>=b)` entspricht `True` (1) oder `False` (0)

Wir können nun Boolesche Vergleiche ausführen, deren Ergebnis ein Boolescher Wert `True` (`1`) oder `False` (`0`) zugeordnet wird. Weiterhin können wir auch Sprünge definieren und ausführen. Jetzt können wir uns dem Zweiarmigen `if` zuwenden.

ZWEIARMIGES-`if`

- `B` ist z.B. (`a==b && c>25`), d.h. `B` kann ein komplexer Ausdruck sein!

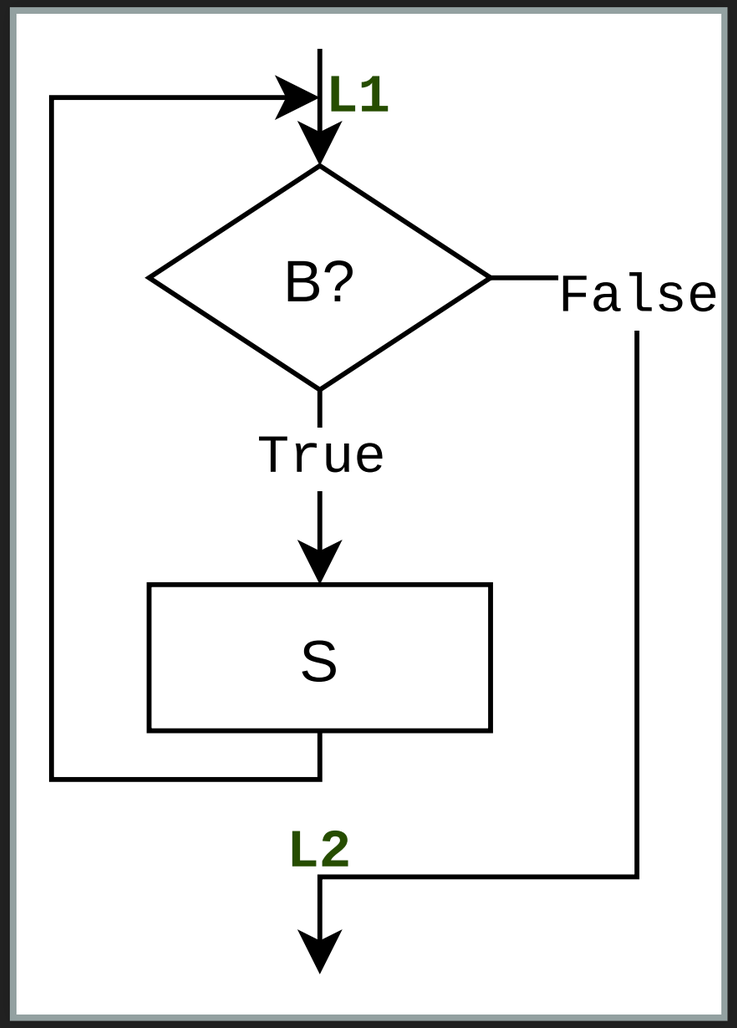
`if (B) S1 else S2;`




```
    ...
    <B>      // hinterlässt einen Booleschen Wert True/False auf dem Stack
    brf L1   // Springt zum Label "L1" -> <S2> wenn Ergebnis == False
    <S1>
    jmp L2  // Springt zum Label "L2" nach Abarbeitung von <S1>
L1:
    <S2>
L2:
    ...
    ...
```

`while`-SCHLEIFE

`while (B) S;`



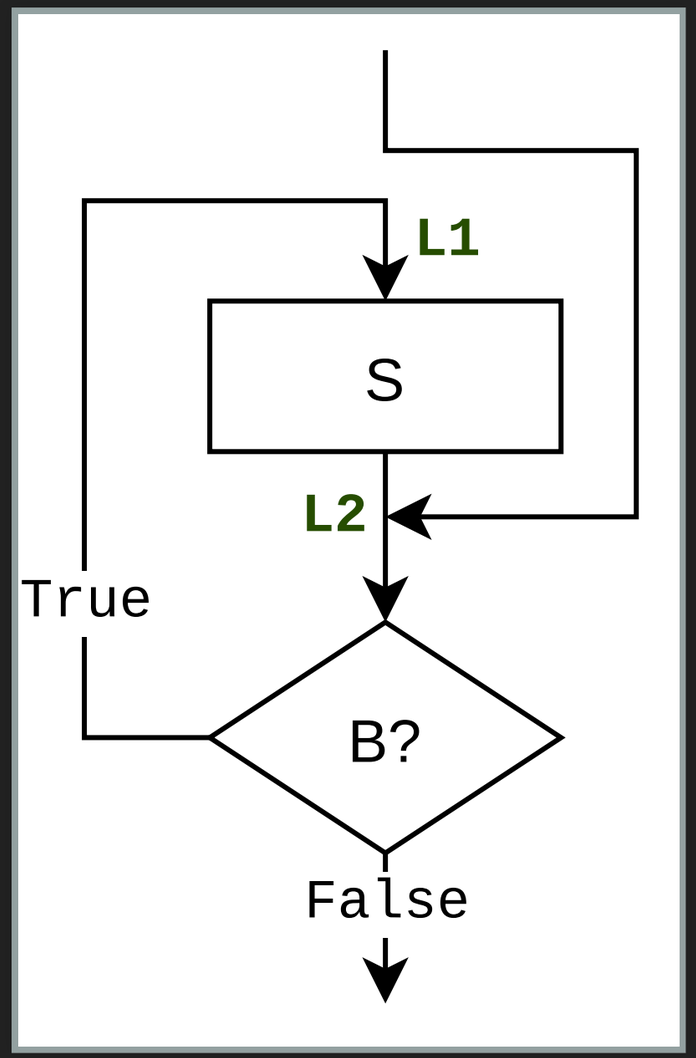
Alternative 1:

```
L1:
    <B>      // hinterlässt einen Booleschen Wert True/False auf dem Stack
    brf L2   // Springt zum Label "L2" wenn Ergebnis == False
    <S>
    jmp L1  // Springt zum Label "L1" nach Abarbeitung von <S>
L2:
    ...
    ...
```

	In diesem Fall benötigen wir 2 Sprünge in **jedem Schleifendurchlauf**. Da die Ausführung jeder Instruktion Rechenzeit kostet, ist diese Lösung nicht optimal!

`while`-SCHLEIFE

`while (B) S;`



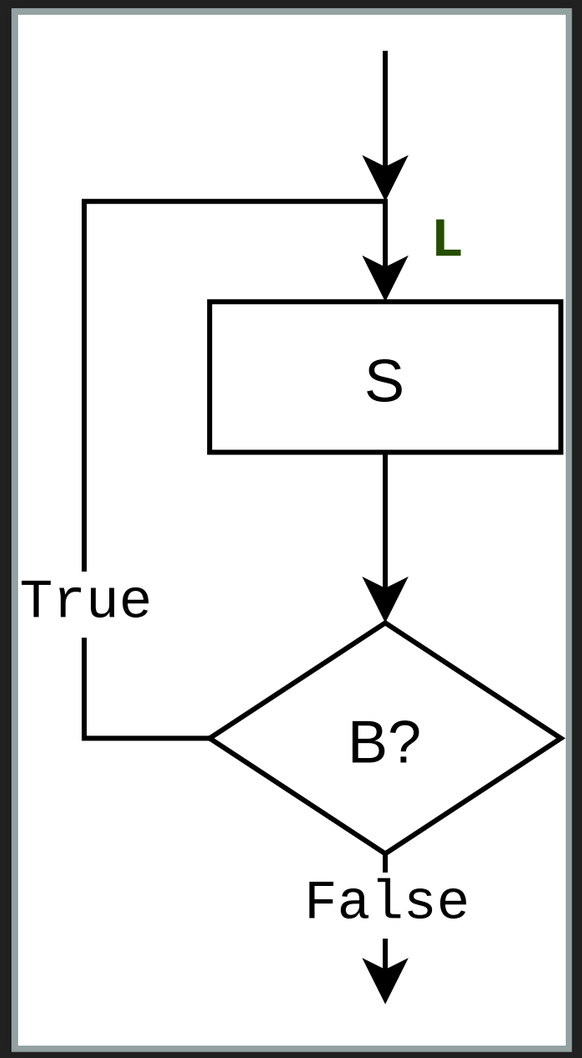
Alternative 2:

```
    jmp L2    // Initialer Sprung zu L2, d.h. Auswertung Boolescher Verlgeich
L1:
    <S>
L2:
    <B>      // hinterlässt einen Booleschen Wert True/False auf dem Stack
    brt L1   // Springt zum Schleifenanfang, falls True!
    ...      // Andernfalls (False) weiter mit nächster Instruktion
    ...
```

	In diesem Fall wird nur **ein** Sprung in **jedem Schleifendurchlauf** benötigt (nämlich `brt L1;`). Dies spart pro Schleifendurchlauf die Ausführung einer zusätzlichen Instruktion!

do-SCHLEIFE

`do S while (B);`




```
L:
    <S>
    <B>      // hinterlässt einen Booleschen Wert True/False auf dem Stack
    brt L    // Springt zum Schleifenanfang, falls True!
    ...      // Andernfalls (False) weiter mit nächster Instruktion
    ...
```

BOOLESCHE AUSDRÜCKE: VOLLSTÄNDIGE AUSWERTUNG

Mit unseren gegebenen Mitteln können wir auch bereits komplexere Boolesche Ausdrücke, wie z.B. `(i == 25 && a < b)` auswerten. Hierbei gibt es zwei Varianten:

Vollständige Auswertung B_1 && B_2

```
<B1>   // Werte
<B2>
and
```

Vollständige Auswertung B_1 || B_2

```
<B1>
<B2>
or
```

Nachteil `if (x !=0 && y/x < 5) {...}` ist nicht auswertbar falls `x = 0`, da hier für B_2 also (`y/x < 5`) eine Division durch `0` ausgelöst wird!

BOOLESCHE AUSDRÜCKE: KURZSCHLUSSAUSWERTUNG

Kurzschlussauswertung komplexer Boolescher Ausdrücke, wie z.B.
`(x !=0 && y/x < 5)`.

Kurzschlussauswertung: `AND` — B_1 && B_2

```
    ...
    <B1>
    brf L1   // Wenn B1 == False dann wird Ausdruck insgesamt False
    <B2>     // andernfalls Werte B2 aus und lege Ergebnis auf Stack
    jmp L2
L1:
    pushc 0  // False
L2:
    ...
```

B_1	B_2	B_1 && B_2
0	0	0
0	1	0
1	0	0
1	1	1

`if (x !=0 && y/x < 5) {...}` ist nun auswertbar, der zweite Boolesche Vergleich (`y/x < 5`) wird nicht mehr evaluiert.

BOOLESCHE AUSDRÜCKE: KURZSCHLUSSAUSWERTUNG

Kurzschlussauswertung komplexer Boolescher Ausdrücke, wie z.B. `(i == 25 || a < b)`.

Kurzschlussauswertung: `OR` — B_1 `||` B_2

```
    ...
    <B1>
    brt L1   // Wenn B1 == True dann wird Ausdruck insgesamt True
    <B2>     // andernfalls Werte B2 aus und lege Ergebnis auf Stack
    jmp L2
L1:
    pushc 1  // True
L2:
    ...
```

B_1	B_2	B_1 `||` B_2
0	0	0
0	1	1
1	0	1
1	1	1