Technische Universität Braunschweig

Institute of Operating Systems and Computer Networks
Reliable System Software

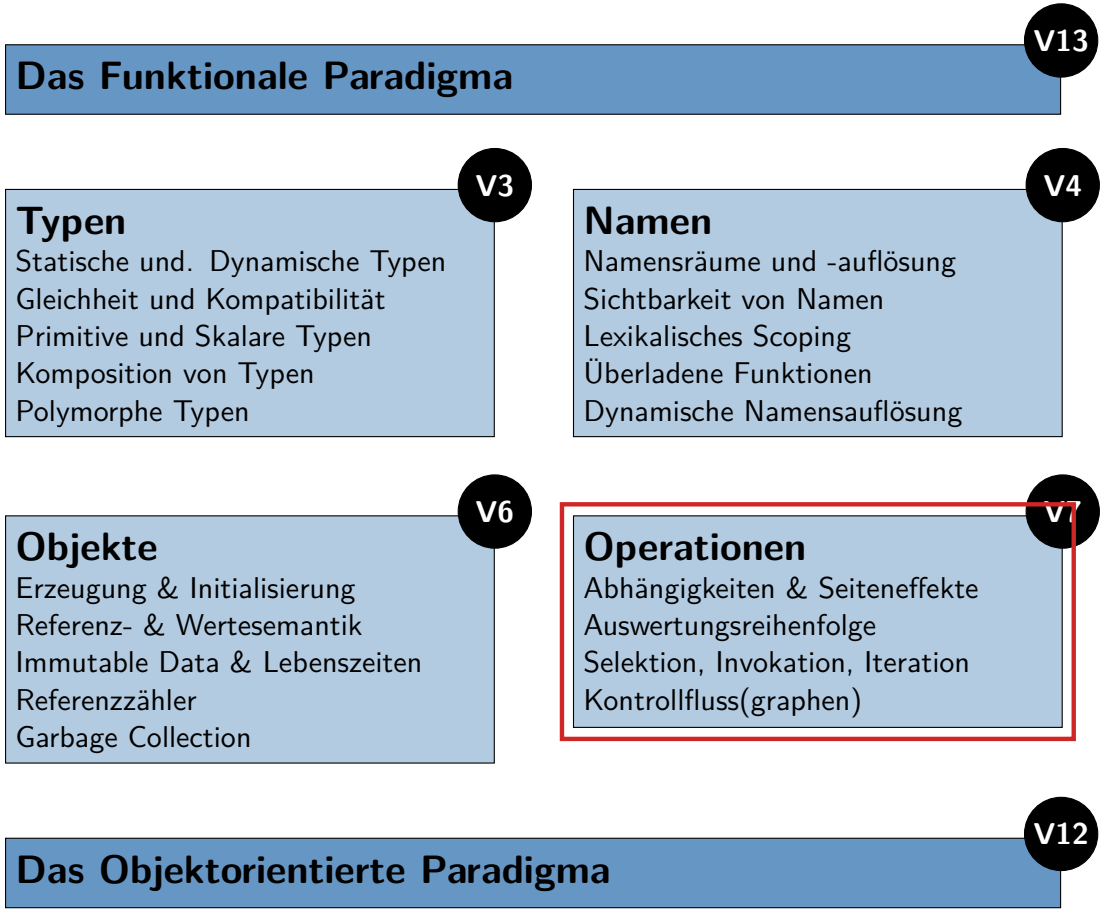
Programmiersprachen und Übersetzer

07 - Operationen

Christian Dietrich

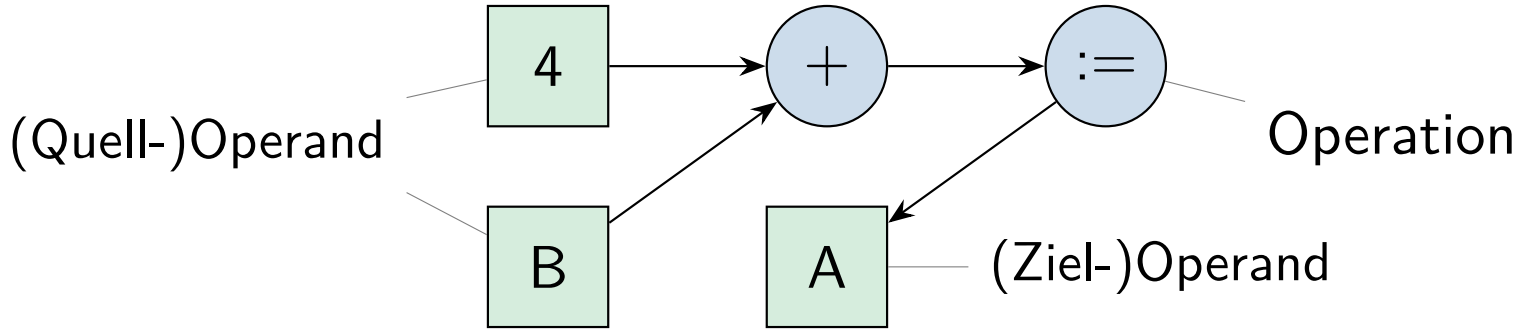
Sommersemester 2024

Einordnung in die Vorlesung: Operationen



- Operationen kombinieren Daten und manipulieren Objekte.
  - Operationen hängen implizit und/oder explizit voneinander ab.
  - Abbildung des virtuellen Operationsstrom auf die reale sequentielle Maschine

Was ist eine ”Operation”?



- **Erinnerung 1. Vorlesung**: Maschinenmodell der virtuellen Maschine:
  - Speicher/Objekte: Wie kann man Informationen ablegen und wieder abrufen?
  - Befehle/Operationen: Wie kann man Informationen miteinander kombinieren?
- Operationen sind Aktiv $\leftrightarrow$ Objekte sind Passiv
  - Quelloperanden: Eingabeobjekte, die von der Operation gelesen werden
  - Abbildungsvorschrift: Kombinationsregel für die gelesenen Daten
  - Zieloperanden: Ausgabeobjekt nimmt das Rechenergebnis auf

```
Operation: "+"
  Quelloperand: LHS (Typ: 32-Bit Integer), RHS (Typ: 32-Bit Integer)
  Zieloperand:  ret (Typ: 32-Bit Integer)
  Abbildungsvorschrift:
     L   = eval(LHS) // Linke Seite auswerten
     R   = eval(RHS) // Rechte Seite auswerten
     ret = modulo(add(L, R), 32)
```

Arithmetische, bitweise und logische Operationen

`+ - * / %` `~ & | ^ << >>` `< <= == => >`

- Notationen in unterschiedlichen Sprachen
  - Präfix: `op A B` oder `op(A,B)` oder `(op A B)` — Lisp, Scheme
  - Infix: `A op B` — C, C++, Java, …
  - Postfix: `A B op` — Forth, PostFix
- Vorrang (Precedence): Bindet Multiplikation stärker als Addition?

  `1*2+3*4` ⟶ `(1*2)+(3*4)` (14) ∨ `1*(2+3)*4` (20)
- Assoziativität: Wird links oder rechts geklammert?

  `1-2-3-4` ⟶ `(((1-2)-3)-4)` (-8) ∨ `1-(2-(3-4))` (-2)

Arithmetische, bitweise und logische Operationen

`+ - * / %  ~ & | ^ << >>  < <= == => >`

Notation, Vorrang und Assoziativität wird vom Parser aufgelöst.

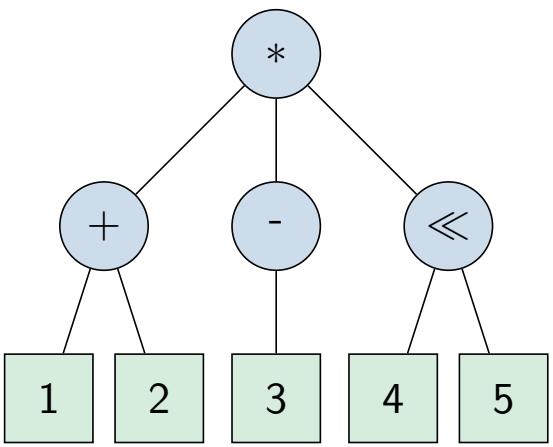
- Notationen in unterschiedlichen Sprachen
  - Präfix: `op A B` oder `op(A,B)` oder `(op A B)` — Lisp, Scheme
  - Infix: `A op B` — C, C++, Java, …
  - Postfix: `A B op` — Forth, PostFix
- Vorrang (Precedence): Bindet Multiplikation stärker als Addition?
  `1*2+3*4` ⟶ `(1*2)+(3*4)` (14) ∨ `1*(2+3)*4` (20)
- Assoziativität: Wird links oder rechts geklammert?
  `1-2-3-4` ⟶ `(((1-2)-3)-4)` (-8) ∨ `1-(2-(3-4))` (-2)
- **Wichtige** Eigenschaften dieser Operationen
  - Quelloperanden werden alle ausgewertet, bevor die Operation ausgeführt wird.
  - Keine Seiteneffekte und genau ein temporäres Objekt als Zieloperand

Auswertungsreihenfolge von einfachen Operationen

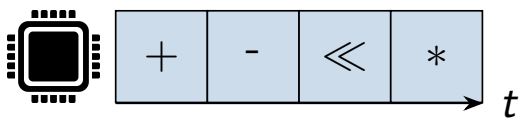
- Semantische Lücke zwischen virtueller und realer Maschine
  - Operationen auf Sprachebene sind als (expliziter oder impliziter) Baum notiert
  - Alle realen Maschinen sind sequentielle Maschinen

Auswertungsreihenfolge

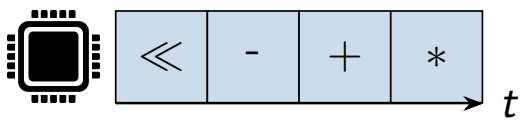
Abbildung der Operationen des ASTs auf eine lineare Befehlssequenz



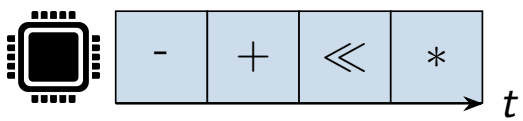
- „links nach rechts“



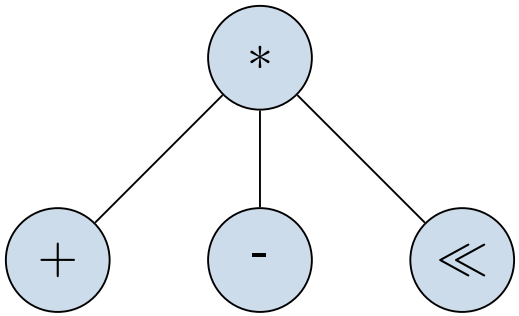
- „rechts nach links“

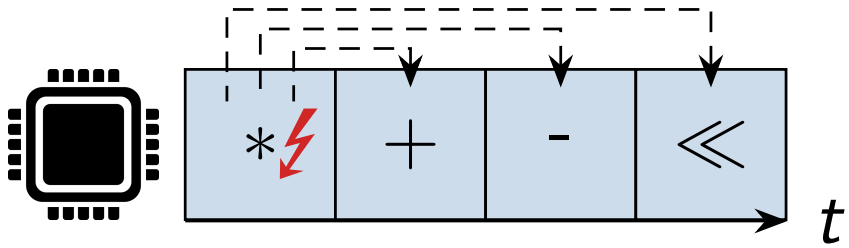


- „beliebiges Kind zuerst“:



Abhängigkeiten zwischen Operationen





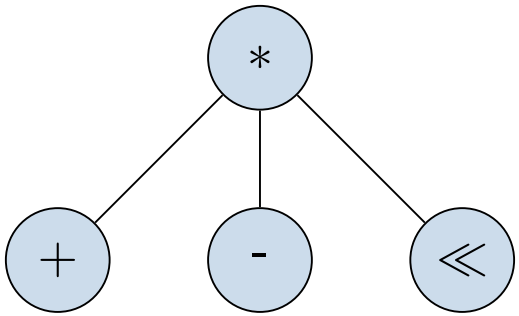
- Abhängigkeit: Eine Operationen hängt vom Ergebnis einer anderen ab.
  - Abhängigkeiten müssen zuerst ausgewertet werden.
  - Bei der Auswertungsreihenfolge: keine Abhängigkeiten in die Zukunft
  - AST definiert hierarchische Abhängigkeiten zwischen Eltern und Kind

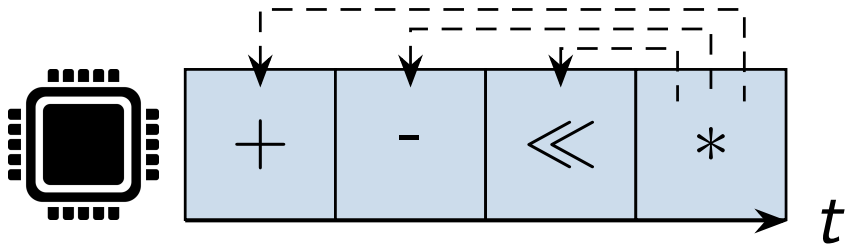
- **Freiheiten** in der Auswertungsreihenfolge
  - Haben Operationen keine Abhängigkeit, ist ihre Reihenfolge **beliebig**.

  ⇒ Übersetzer kann Reihenfolge frei wählen, um Ausführung zu optimieren
  z.B: Weniger Speicherlatenz durch Einschieben von unabhängigen Operationen

Viele Sprachkonstrukte und -regeln erlauben es uns, die **Auswertungsreihenfolge zu formen**, um die Abhängigkeiten explizit zu machen.

Abhängigkeiten zwischen Operationen





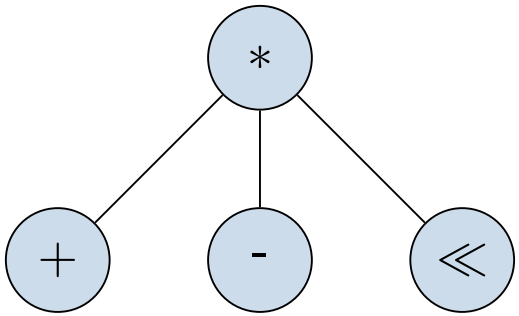
- Abhängigkeit: Eine Operationen hängt vom Ergebnis einer anderen ab.
  - Abhängigkeiten müssen zuerst ausgewertet werden.
  - Bei der Auswertungsreihenfolge: keine Abhängigkeiten in die Zukunft
  - AST definiert hierarchische Abhängigkeiten zwischen Eltern und Kind

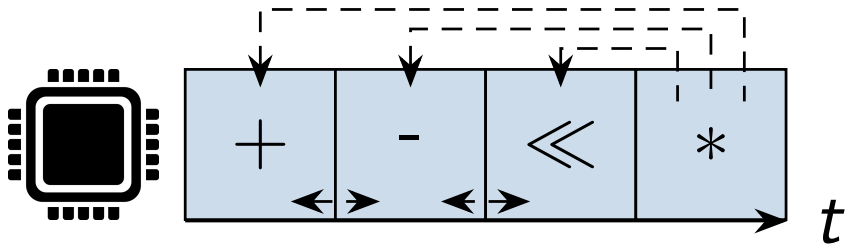
- **Freiheiten** in der Auswertungsreihenfolge
  - Haben Operationen keine Abhängigkeit, ist ihre Reihenfolge **beliebig**.

  $\Rightarrow$ Übersetzer kann Reihenfolge frei wählen, um Ausführung zu optimieren
  z.B: Weniger Speicherlatenz durch Einschieben von unabhängigen Operationen

Viele Sprachkonstrukte und -regeln erlauben es uns, die **Auswertungsreihenfolge zu formen**, um die Abhängigkeiten explizit zu machen.

Abhängigkeiten zwischen Operationen





- Abhängigkeit: Eine Operationen hängt vom Ergebnis einer anderen ab.
  - Abhängigkeiten müssen zuerst ausgewertet werden.
  - Bei der Auswertungsreihenfolge: keine Abhängigkeiten in die Zukunft
  - AST definiert hierarchische Abhängigkeiten zwischen Eltern und Kind

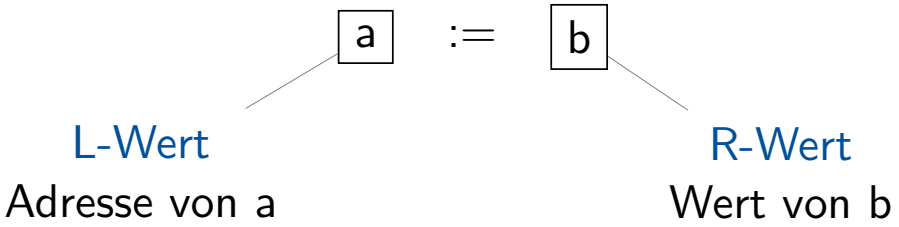
- **Freiheiten** in der Auswertungsreihenfolge
  - Haben Operationen keine Abhängigkeit, ist ihre Reihenfolge **beliebig**.

  $\Rightarrow$ Übersetzer kann Reihenfolge frei wählen, um Ausführung zu optimieren
  z.B: Weniger Speicherlatenz durch Einschieben von unabhängigen Operationen

Viele Sprachkonstrukte und -regeln erlauben es uns, die **Auswertungsreihenfolge zu formen**, um die Abhängigkeiten explizit zu machen.

Die Zuweisungsoperation (`:=`)

- Auswertung der rechten Seite → Ergebnis in der linken Seite speichern
  - Linke und rechte Seite werden nicht symmetrisch übersetzt!
  - R-Wert: Auf der rechten Seite berechnen wir das Ergebnis eines Ausdrucks.
  - L-Wert: Auf der linken Seite muss berechnet werden, wohin das Ergebnis soll.



- Nicht jede Operation hat einen L-Wert.
  - Konstanten (`1` , `"abc"`) haben keinen L-Wert
  - Temporäre Zwischenergebnisse haben keinen L-Wert

```
1 := 2
```

```
a+b := 2
```

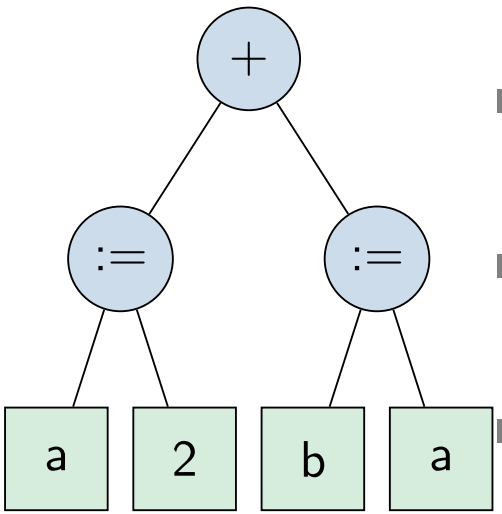
- Unterschied bei L-Werten zwischen Werte- und Referenzmodell
  - Referenzmodell: Alles, was eine Referenz speichern kann, ist ein L-Wert
    Variablen, Array-Elemente und Felder
  - Wertemodell: Alles, was ein Objekt speichern kann, ist ein L-Wert
    zusätzlich: Dereferenzierung: `*(ptr)`

Seiteneffekte und Idempotenz

Die Zuweisungsoperation hat den gewollten Seiteneffekt, den Zustand der virtuellen Maschine zu verändern.

- Seiteneffekte: Wirkung über ein temporäres Zielobjekt hinaus
  - Operationen sind nicht mehr idempotent `i := i + 1`
  - Nur beobachtbare Seiteneffekte sind ein Problem (seiteneffektfreie Funktion)
  - Strikt funktionale Sprachen (Haskell): Verbot jeglicher Seiteneffekte

- Wer sich auf Seiteneffekte verlässt, erzeugt eine implizite Abhängigkeit.



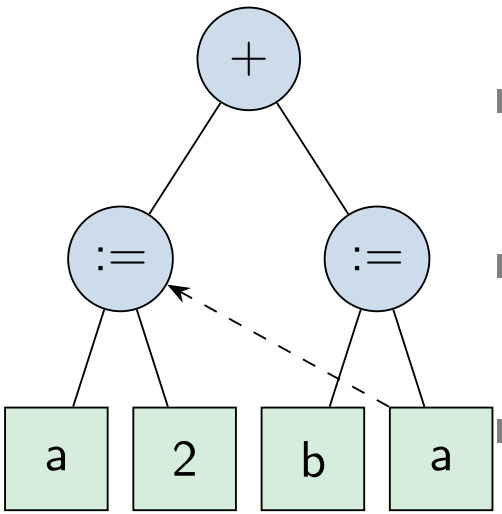
  - Das Ergebnis von `(a := 2) + (b := a)` hängt von der Auswertungsreihenfolge ab
  - Ohne strikt definierte Auswertungsreihenfolge ist das Ergebnis **undefiniert**!
  - Sprachregeln und -konstrukte **vermeiden** implizite Abhängigkeiten oder **machen sie explizit**.

Seiteneffekte und Idempotenz

Die Zuweisungsoperation hat den gewollten Seiteneffekt, den Zustand der virtuellen Maschine zu verändern.

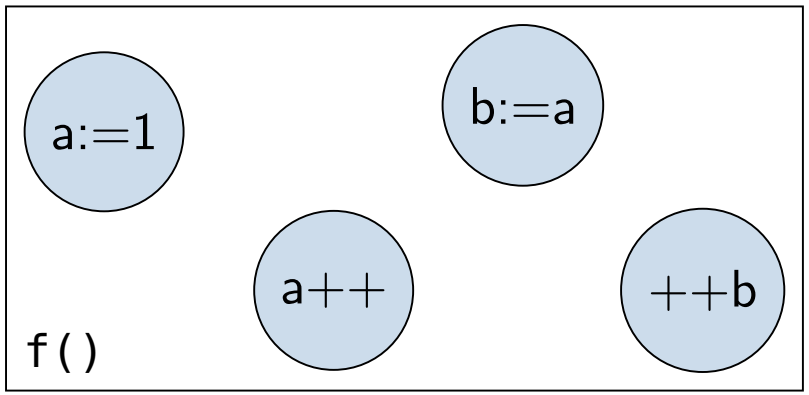
- Seiteneffekte: Wirkung über ein temporäres Zielobjekt hinaus
  - Operationen sind nicht mehr idempotent `i := i + 1`
  - Nur beobachtbare Seiteneffekte sind ein Problem (seiteneffektfreie Funktion)
  - Strikt funktionale Sprachen (Haskell): Verbot jeglicher Seiteneffekte

- Wer sich auf Seiteneffekte verlässt, erzeugt eine implizite Abhängigkeit.



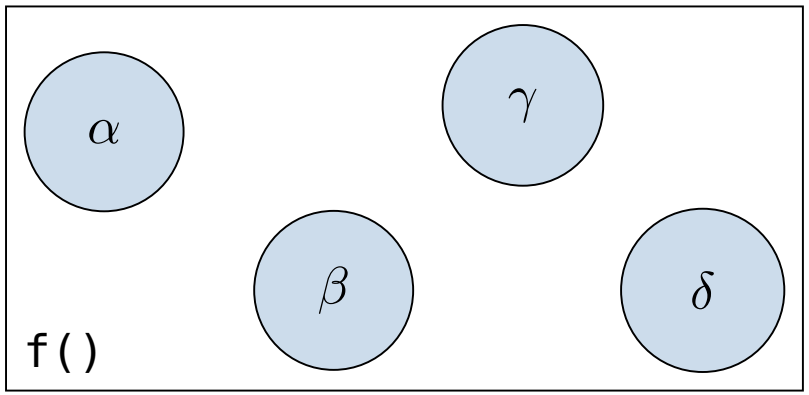
- Das Ergebnis von `(a := 2) + (b := a)` hängt von der Auswertungsreihenfolge ab
- Ohne strikt definierte Auswertungsreihenfolge ist das Ergebnis **undefiniert**!
- Sprachregeln und -konstrukte **vermeiden** implizite Abhängigkeiten oder **machen sie explizit**.

Umgang mit impliziten Abhängigkeiten



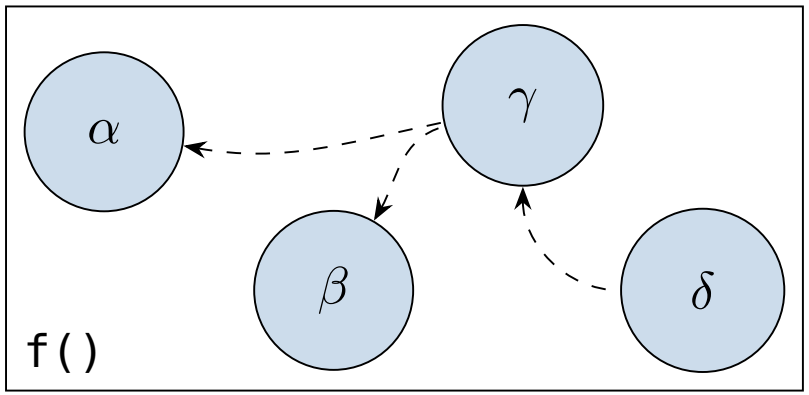
- **Ausgangspunkt** und geistiges Modell für die folgenden Folien
  - **Gegeben**: Operationen (mit und ohne) Seiteneffekt $(\alpha, \beta, \gamma, \ldots)$
  - Die Operationen hängen implizit voneinander ab (gestrichelt)
  - Sprachmittel geben Reihenfolge und Vorrang explizit vor (durchgezogen)

Umgang mit impliziten Abhängigkeiten



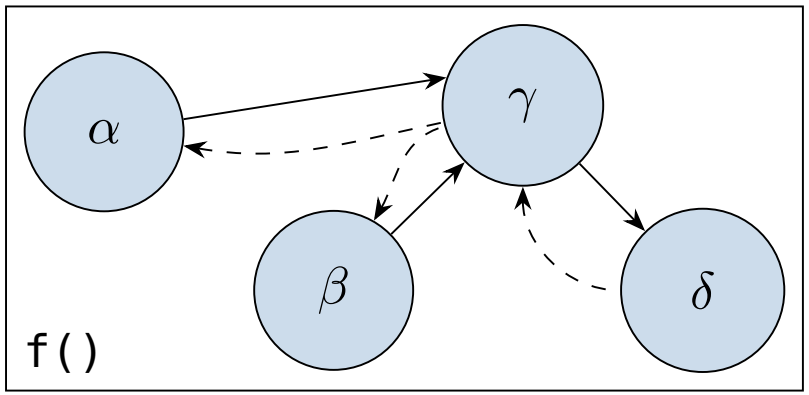
- **Ausgangspunkt** und geistiges Modell für die folgenden Folien
  - **Gegeben**: Operationen (mit und ohne) Seiteneffekt $(\alpha, \beta, \gamma, \ldots)$
  - Die Operationen hängen implizit voneinander ab (gestrichelt)
  - Sprachmittel geben Reihenfolge und Vorrang explizit vor (durchgezogen)

Umgang mit impliziten Abhängigkeiten



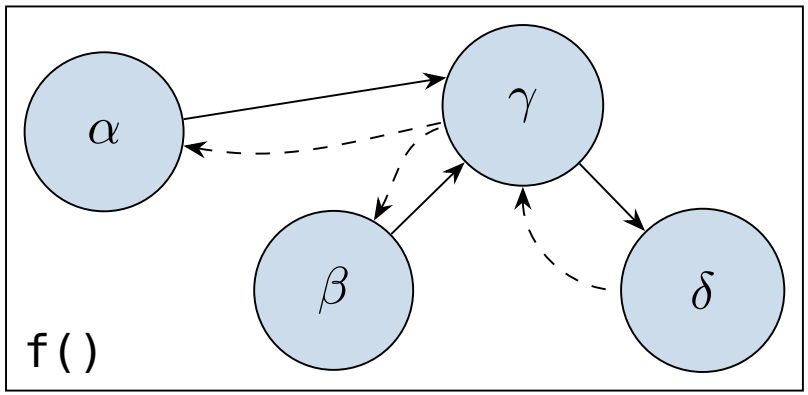
- **Ausgangspunkt** und geistiges Modell für die folgenden Folien
  - **Gegeben**: Operationen (mit und ohne) Seiteneffekt $(\alpha, \beta, \gamma, \ldots)$
  - Die Operationen hängen implizit voneinander ab (gestrichelt)
  - Sprachmittel geben Reihenfolge und Vorrang explizit vor (durchgezogen)

Umgang mit impliziten Abhängigkeiten

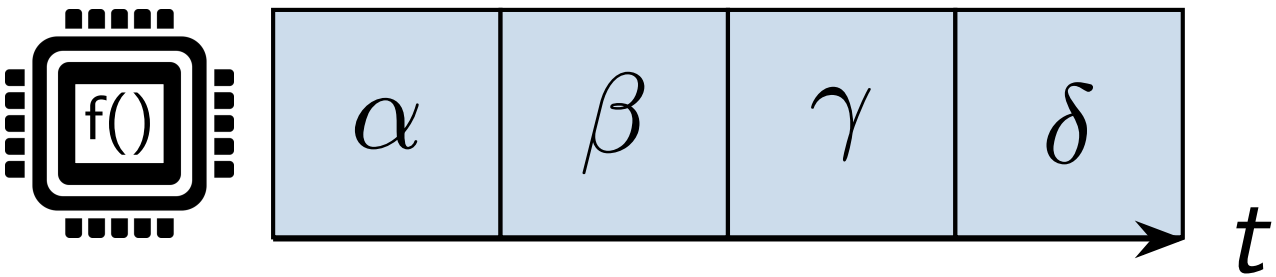


- **Ausgangspunkt** und geistiges Modell für die folgenden Folien
  - **Gegeben**: Operationen (mit und ohne) Seiteneffekt $(\alpha, \beta, \gamma, \ldots)$
  - Die Operationen hängen implizit voneinander ab (gestrichelt)
  - Sprachmittel geben Reihenfolge und Vorrang explizit vor (durchgezogen)

Umgang mit impliziten Abhängigkeiten



- **Ausgangspunkt** und geistiges Modell für die folgenden Folien
  - **Gegeben**: Operationen (mit und ohne) Seiteneffekt $(\alpha, \beta, \gamma, \ldots)$
  - Die Operationen hängen implizit voneinander ab (gestrichelt)
  - Sprachmittel geben Reihenfolge und Vorrang explizit vor (durchgezogen)
  - **Ziel**: Eine wohldefinierte Auswertungsreihenfolge, welche die Abhängigkeiten erfüllt



Sequenzierung: Statements

- Sequenzierung: Vorgabe einer linearen, strikten Reihenfolge
  - Der Sequenzoperator (`;`) führt seine Kinder hintereinander aus.
  - Die Operanden des Sequenzoperators heißen Statements.
  - Ergebnisse werden verworfen oder das erste/letzte wird zurückgegeben.
- Statements sind das grundlegende Sprachmittel in imperativen Sprachen.


```c
int a, b;
void foo() {
  a= 1; a++; b= (a+=4, a*3);
} // b == 18
```

C

```ruby
a = (1+2 ; begin
       x = 1 + 2
       x * 3
     end + 1) # a == 10
```

Ruby

C/C++

- Sequenzierung: Komma, Semikolon
- $\alpha;\beta$ hat kein Ergebnis
- α,β liefert letztes Teilergebnis (β)

Ruby

- Sequenzierung: `'\n'`, Semikolon
- Letztes Statement liefert den Rückgabewert

Unterscheidung zwischen Ausdrücken und Statements

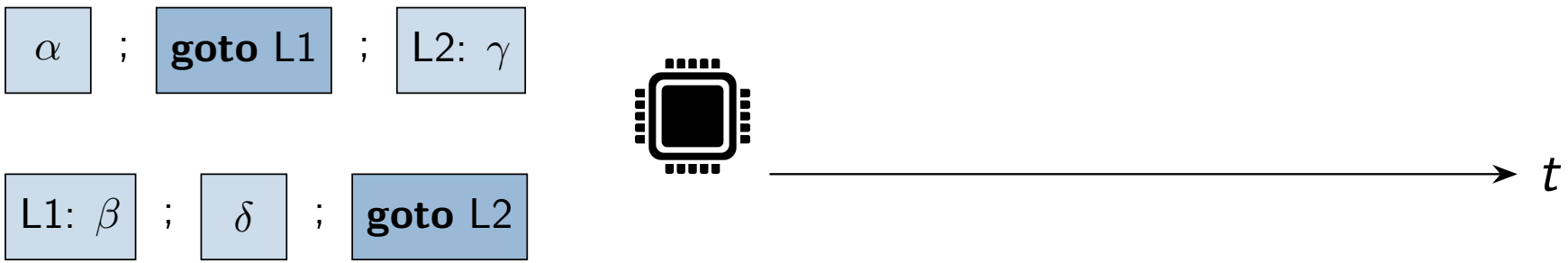
Die meisten Sprachen treffen diese Unterscheidung.

- Ausdrücke/Expressions
  - Haben einen Rückgabewert
  - Haben einen Rückgabetypen (bei statischer Typisierung)
  - Können Operanden anderer Expressions sein
- Statements
  - Haben keinen Rückgabewert und daher auch keinen Typen
  - Dienen als Elemente der Sequenzierung und anderer Programmstrukturen
  - Entsprechen (in etwa) einer Zeile Code
  - Werden in Code-Blöcken bzw. Compound-Statements zusammengefasst
- Manche Operationen haben eine Statement- und eine Ausdrucks-Form.

  C: `if (cond) { then_block } else { else_block}` `cond ? then_expr : else_expr`

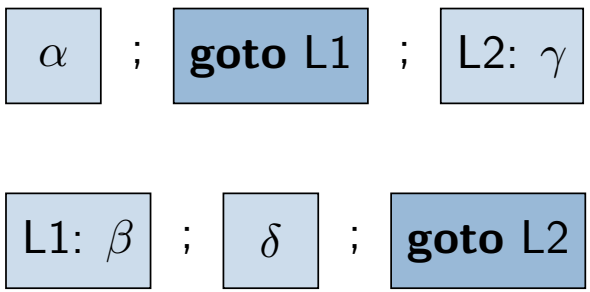
Sequenzierung: Goto

- Ein unbedingter Sprung verbindet syntaktisch entfernte Statements.
  - Sprungmarke: gibt einer Operation einen Namen
  - Sprung: Auswertung wird an dieser Stelle abgebrochen und an der genannten Sprungmarke fortgesetzt.



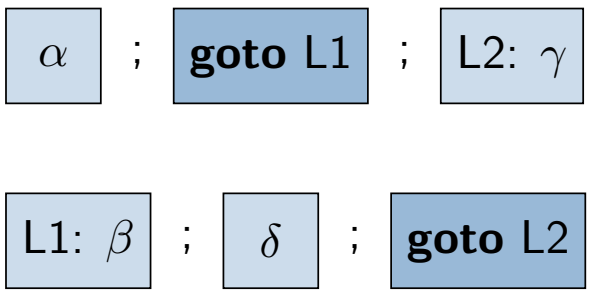
Sequenzierung: Goto

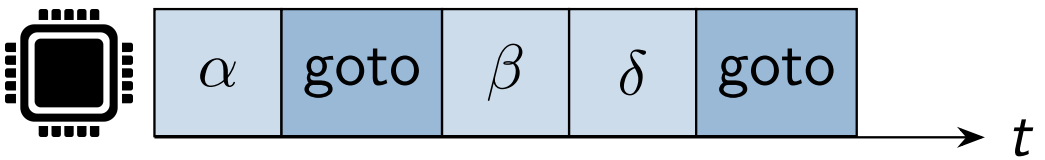
- Ein unbedingter Sprung verbindet syntaktisch entfernte Statements.
  - Sprungmarke: gibt einer Operation einen Namen
  - Sprung: Auswertung wird an dieser Stelle abgebrochen und an der genannten Sprungmarke fortgesetzt.



Sequenzierung: Goto

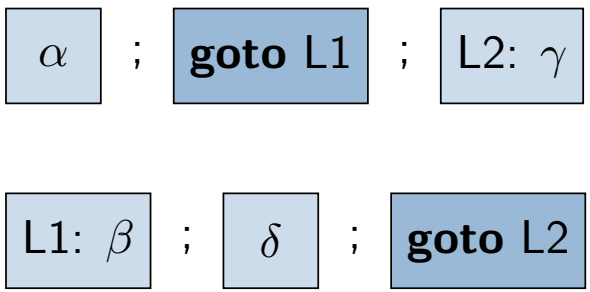
- Ein unbedingter Sprung verbindet syntaktisch entfernte Statements.
  - Sprungmarke: gibt einer Operation einen Namen
  - Sprung: Auswertung wird an dieser Stelle abgebrochen und an der genannten Sprungmarke fortgesetzt.

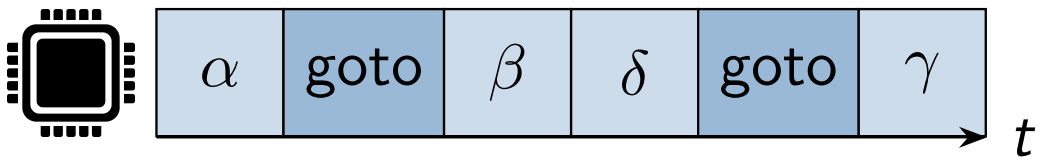




Sequenzierung: Goto

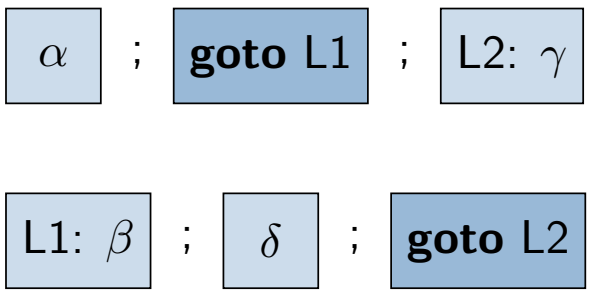
- Ein unbedingter Sprung verbindet syntaktisch entfernte Statements.
  - Sprungmarke: gibt einer Operation einen Namen
  - Sprung: Auswertung wird an dieser Stelle abgebrochen und an der genannten Sprungmarke fortgesetzt.

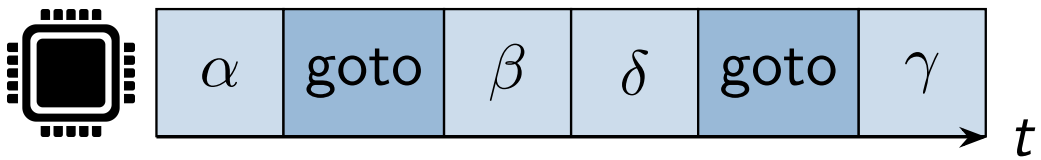




Sequenzierung: Goto

- Ein unbedingter Sprung verbindet syntaktisch entfernte Statements.
  - Sprungmarke: gibt einer Operation einen Namen
  - Sprung: Auswertung wird an dieser Stelle abgebrochen und an der genannten Sprungmarke fortgesetzt.





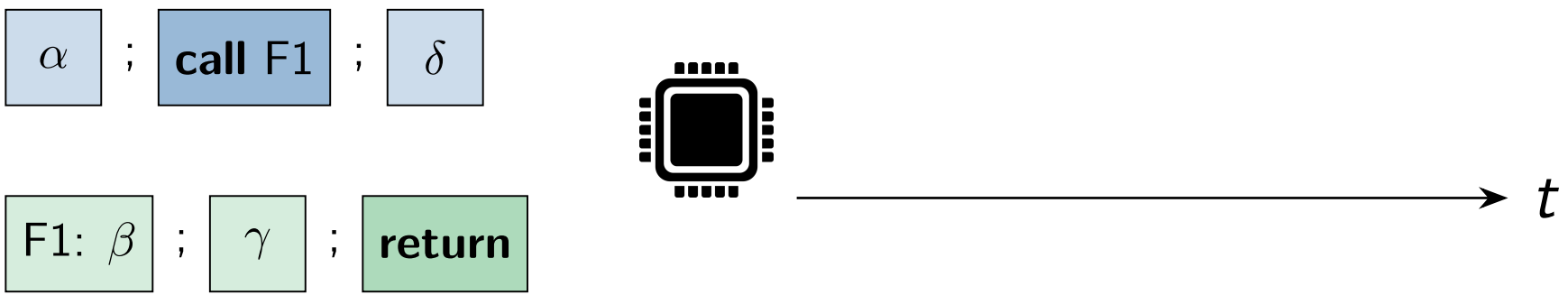
- Nachfolgeoperation einer Operation wird explizit und statisch benannt
  - Sprünge erlauben es, Sequenzen zusammenzustückeln
  - (Endlos laufende) Schleifen sind nun möglich *Rückwärtssprünge*

Definition: Kontrollfluss *(vgl. mit Kontrollwerk der CPU)*

Ein Kontrollfluss ist eine **konkrete** Sequenz von Operationen, die hintereinander ausgeführt werden.

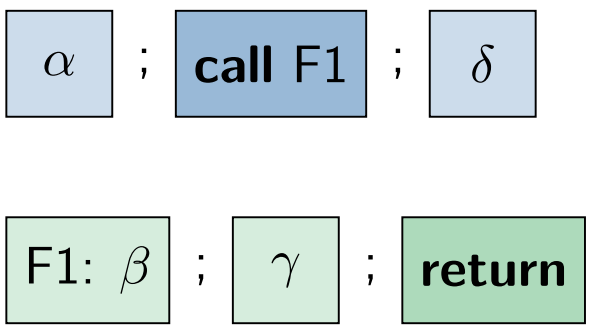
Invokation: Funktionsaufrufe

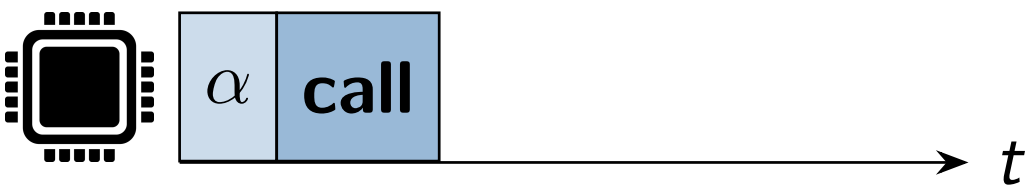
- Unbedingte Sprünge und Funktionsaufrufe sind sehr ähnlich.
  - Der Kontrollfluss verlässt die aktuelle Operationssequenz.
  - Die Auswertung wird an einer entfernten benannten Operation fortgesetzt.
- **Aber**: Funktionsaufrufe haben zwei grundlegende Erweiterungen
  - Rücksprung: Wir können von verschiedenen Stellen angesprungen werden und zur jeweiligen verlassenen Operationssequenz zurückkehren.
  - Argumente: Wir können Daten an den Kontrollfluss heften.



Invokation: Funktionsaufrufe

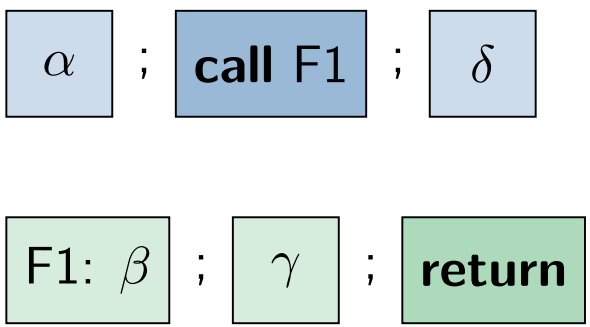
- Unbedingte Sprünge und Funktionsaufrufe sind sehr ähnlich.
  - Der Kontrollfluss verlässt die aktuelle Operationssequenz.
  - Die Auswertung wird an einer entfernten benannten Operation fortgesetzt.
- **Aber**: Funktionsaufrufe haben zwei grundlegende Erweiterungen
  - Rücksprung: Wir können von verschiedenen Stellen angesprungen werden und zur jeweiligen verlassenen Operationssequenz zurückkehren.
  - Argumente: Wir können Daten an den Kontrollfluss heften.

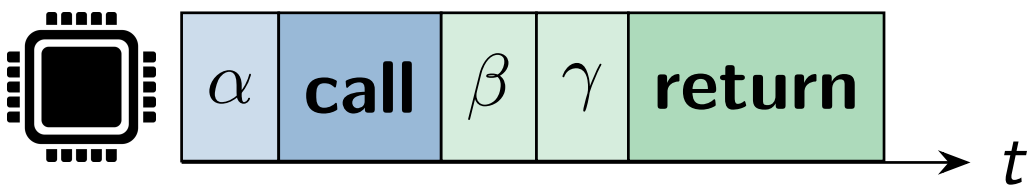




Invokation: Funktionsaufrufe

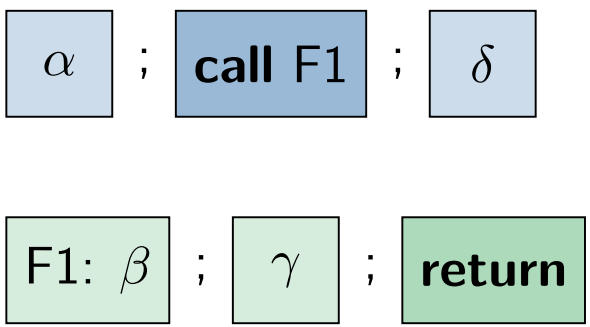
- Unbedingte Sprünge und Funktionsaufrufe sind sehr ähnlich.
  - Der Kontrollfluss verlässt die aktuelle Operationssequenz.
  - Die Auswertung wird an einer entfernten benannten Operation fortgesetzt.
- **Aber**: Funktionsaufrufe haben zwei grundlegende Erweiterungen
  - Rücksprung: Wir können von verschiedenen Stellen angesprungen werden und zur jeweiligen verlassenen Operationssequenz zurückkehren.
  - Argumente: Wir können Daten an den Kontrollfluss heften.

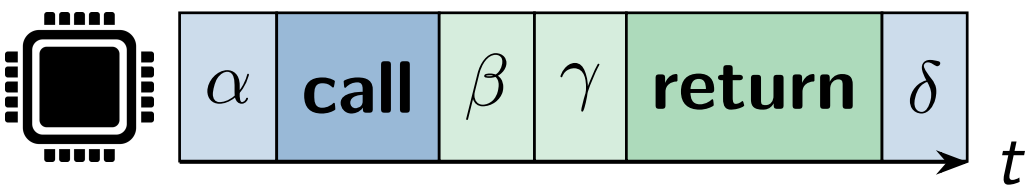




Invokation: Funktionsaufrufe

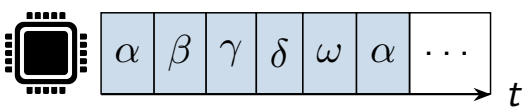
- Unbedingte Sprünge und Funktionsaufrufe sind sehr ähnlich.
  - Der Kontrollfluss verlässt die aktuelle Operationssequenz.
  - Die Auswertung wird an einer entfernten benannten Operation fortgesetzt.
- **Aber**: Funktionsaufrufe haben zwei grundlegende Erweiterungen
  - Rücksprung: Wir können von verschiedenen Stellen angesprungen werden und zur jeweiligen verlassenen Operationssequenz zurückkehren.
  - Argumente: Wir können Daten an den Kontrollfluss heften.





Vergleich: Sprünge vs. Invokation

```
F1: α;
    goto L1;
L2: γ;
    call F2;
    ω;
    goto F1;
L1: β;
    goto L2;
F2: δ;
    return;
```



- Übersichtlichkeit von Code
  - Sprungmarken, Sprünge und der Aufruf von Labels werden schnell unübersichtlich.
  - Dijkstra, 1968: „GOTO considered harmful“
  - Manche argumentieren, dass goto einen Anwendungsfall bei der Fehlerbehandlung hat (z.B. Linus Torvalds).

⇒ Funktionen mit Blockstruktur, Sprünge nur selten:

```
void F1() {
  α; β ; F2() ; ω
}
```

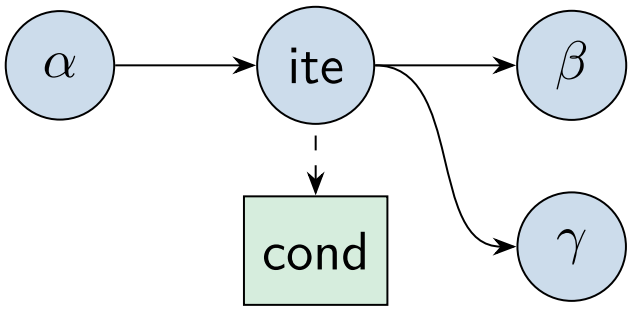
```
void F2() {
  γ;
}
```

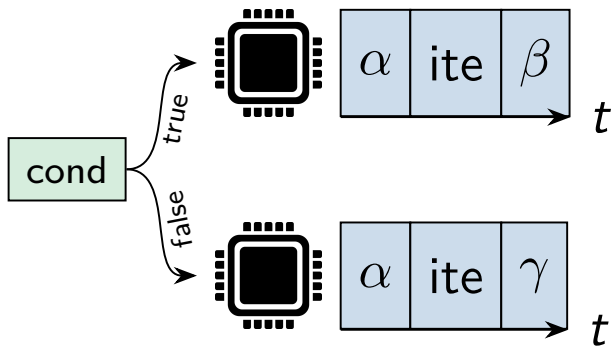
Selektion: Bedingte Ausführung

Sequenzoperator, Sprüngen und Funktionsaufrufe

Wir können nur einen **linearen** Kontrollfluss vorschreiben; nur endlose Schleifen/Rekursionen sind möglich.

- **Wichtiger** als Operationen zu reihen, ist es **Operationen auszuwählen**.
  - if-then-else–Operation (`ite`) wählt eine der Nachfolgeoperationen aus
  - Auswahl der nächsten Operation erfolgt anhand des **Programmzustandes**

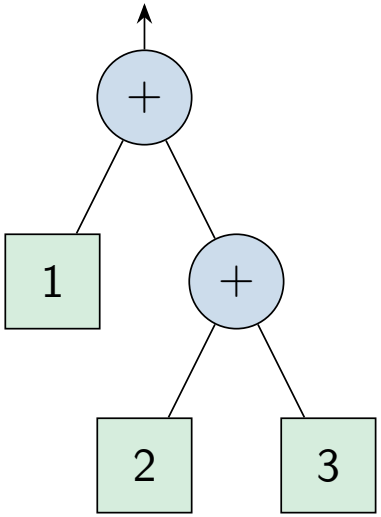




  - Mit Bedingungen gibt es plötzlich mehrere Kontrollflüsse durch das Programm
  - Variationen von Bedingungen: Wie viele Nachfolger? Wie wird ausgewählt?

Die Selektion-Operation ist eine seltsame Operation

Normale Operationen

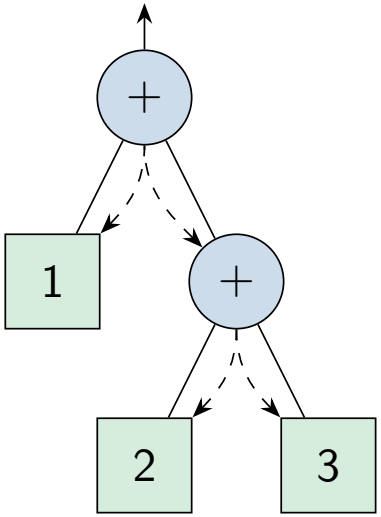


- Operation hängt von **allen** Kinder ab
- Kinder werden zuerst ausgeführt
- Reihenfolge der Kinder nicht zwingend

McCarthy (Erfinder von Lisp), 1960: „[...] but the notion of conditional expression is believed to be new, and the use of conditional expressions permits functions to be defined recursively in a new and convenient way.“

Die Selektion-Operation ist eine seltsame Operation

Normale Operationen

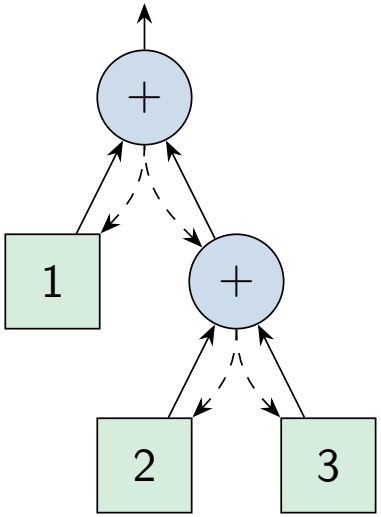


- Operation hängt von **allen** Kinder ab
- Kinder werden zuerst ausgeführt
- Reihenfolge der Kinder nicht zwingend

McCarthy (Erfinder von Lisp), 1960: „[...] but the notion of conditional expression is believed to be new, and the use of conditional expressions permits functions to be defined recursively in a new and convenient way.“

Die Selektion-Operation ist eine seltsame Operation

Normale Operationen

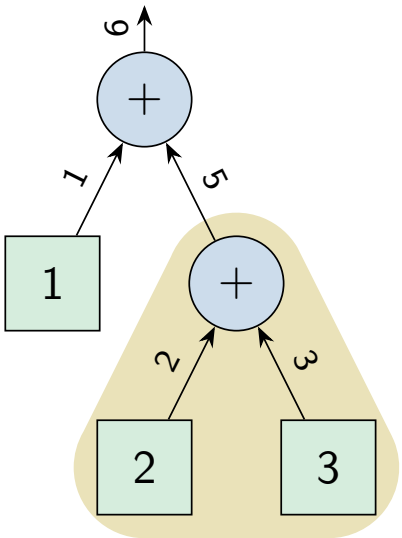


- Operation hängt von **allen** Kinder ab
- Kinder werden zuerst ausgeführt
- Reihenfolge der Kinder nicht zwingend

McCarthy (Erfinder von Lisp), 1960: „[...] but the notion of conditional expression is believed to be new, and the use of conditional expressions permits functions to be defined recursively in a new and convenient way.“

Die Selektion-Operation ist eine seltsame Operation

Normale Operationen

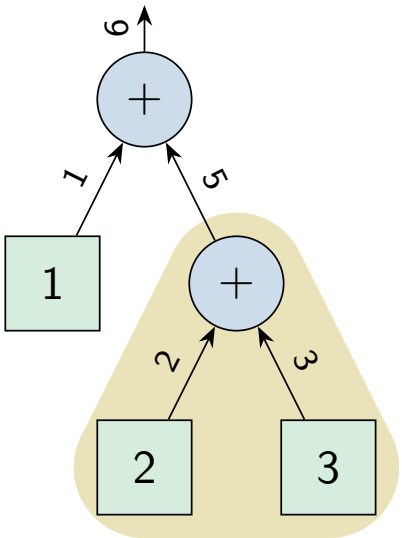


- Operation hängt von **allen** Kinder ab
- Kinder werden zuerst ausgeführt
- Reihenfolge der Kinder nicht zwingend

McCarthy (Erfinder von Lisp), 1960: „[...] but the notion of conditional expression is believed to be new, and the use of conditional expressions permits functions to be defined recursively in a new and convenient way.“

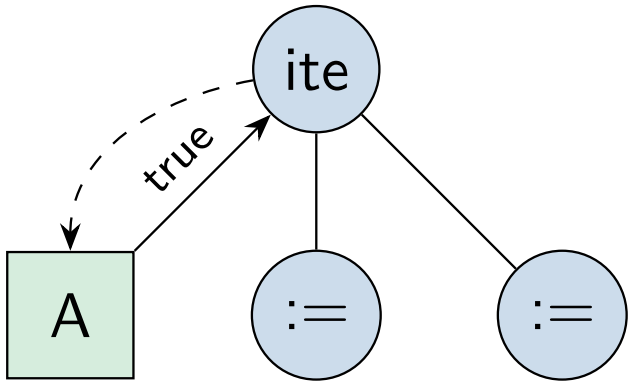
Die Selektion-Operation ist eine seltsame Operation

Normale Operationen



- Operation hängt von **allen** Kinder ab
- Kinder werden zuerst ausgeführt
- Reihenfolge der Kinder nicht zwingend

Selektion



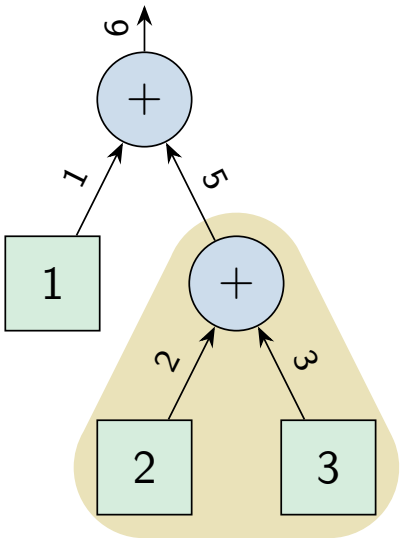
- Hängt **immer** von der Bedingung ab
- Nur ein anderes Kind wird ausgeführt
- Auswertung wird Daten-sensitiv

$\Rightarrow$ hat programmierbare Abhängigkeit

McCarthy (Erfinder von Lisp), 1960: „[...] but the notion of conditional expression is believed to be new, and the use of conditional expressions permits functions to be defined recursively in a new and convenient way.“

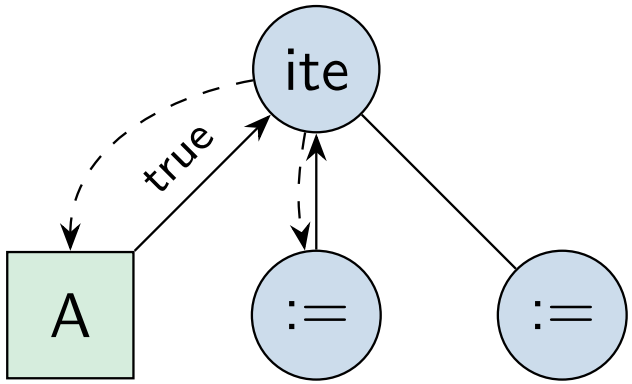
Die Selektion-Operation ist eine seltsame Operation

Normale Operationen



- Operation hängt von **allen** Kinder ab
- Kinder werden zuerst ausgeführt
- Reihenfolge der Kinder nicht zwingend

Selektion



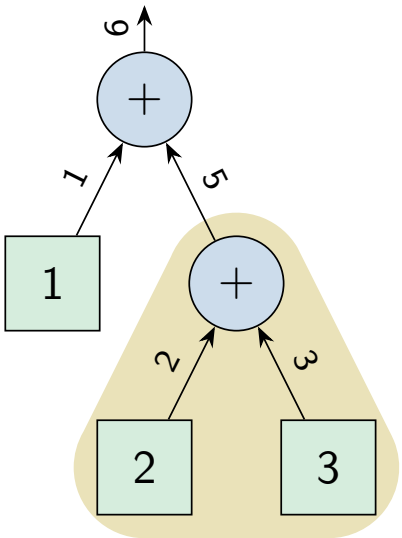
- Hängt **immer** von der Bedingung ab
- Nur ein anderes Kind wird ausgeführt
- Auswertung wird Daten-sensitiv

$\Rightarrow$ hat programmierbare Abhängigkeit

McCarthy (Erfinder von Lisp), 1960: „[...] but the notion of conditional expression is believed to be new, and the use of conditional expressions permits functions to be defined recursively in a new and convenient way.“

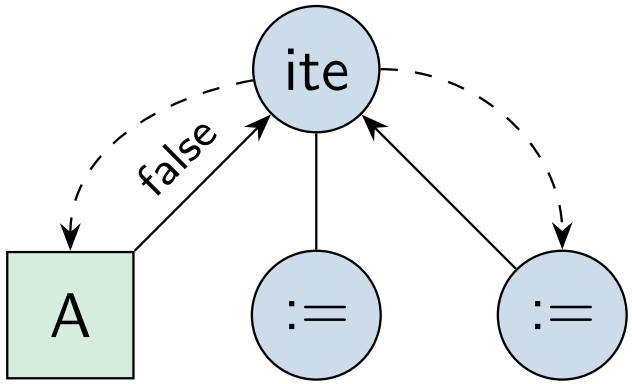
Die Selektion-Operation ist eine seltsame Operation

Normale Operationen



- Operation hängt von **allen** Kinder ab
- Kinder werden zuerst ausgeführt
- Reihenfolge der Kinder nicht zwingend

Selektion



- Hängt **immer** von der Bedingung ab
- Nur ein anderes Kind wird ausgeführt
- Auswertung wird Daten-sensitiv

$\Rightarrow$ hat programmierbare Abhängigkeit

McCarthy (Erfinder von Lisp), 1960: „[...] but the notion of conditional expression is believed to be new, and the use of conditional expressions permits functions to be defined recursively in a new and convenient way.“

Verschiedene Formen der Selektion

if–then–else

```c
if (cond) { /* then-block */ }
else      { /* else-block */ }
```
C

- Weitere Selektionen im else-block
- Manchmal auch Postfix-Notation:
  `expr if cond` (Ruby)

Short-Circuit Expression

```c
expr1 && expr2 && expr3;
expr4 || expr5 || expr6;
```
C

- Frühzeitiger Auswertungs-Abbruch sobald der Wahrheitswert feststeht.
- `!expr1` $\Rightarrow$ Übersprung von expr2/3

if-elseif-else Kaskade

```lisp
(cond
  (cond1 expr1)
  (cond2 expr2)
  (cond3 expr3))
```
Lisp

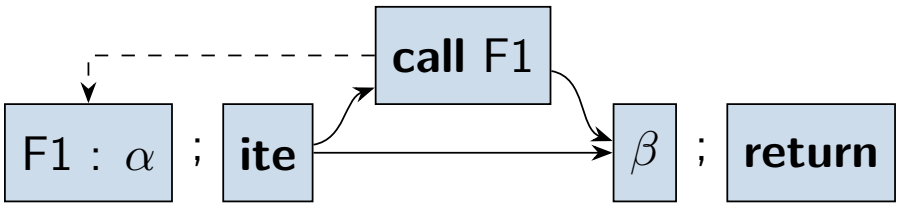
- Lineares Durchlaufen der Paare
- Abbruch bei der ersten erfolgreichen Bedingung

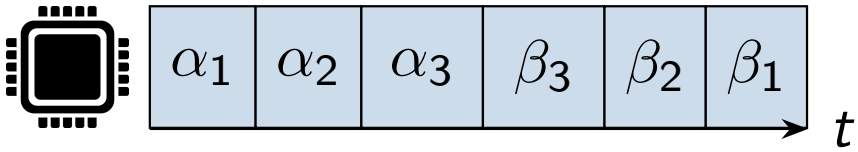
Switch/Case

```rust
match number {
    1     => println!("One"),
    2|3|5 => println!("Two"),
    7..=9 => println!("Range"),
    _     => println!("Def"),
}
```
Rust

- Verzweigungsauswahl am Wert
- Wertebereiche, Default-Verzweigung
- Effizient durch Sprungtabellen

Rekursion: Invokation und Selektion

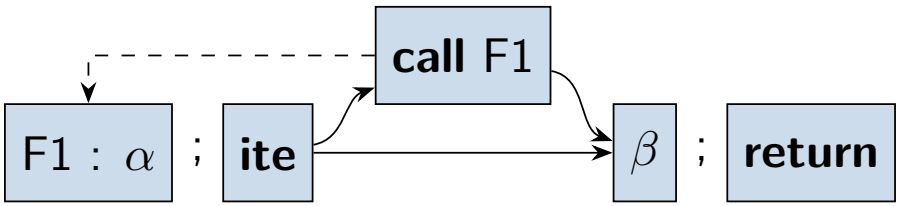


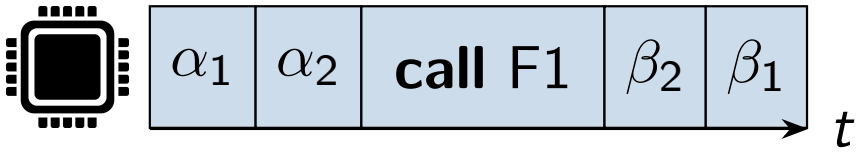


- Rekursion ist das Zusammenspiel von Invokation und Selektion.
  - `call` erlaubt uns, Operationen mehrfach auszuführen.
  - `ite` erlaubt uns, die Invokation irgendwann auszulassen.
  - Selbe Operation mit unterschiedlichen Daten auswerten (α_1, α_2,...)
- Warum macht das überhaupt Sinn?
  - `call` formt nicht nur den Kontrollfluss, sondern auch den Datenfluss!
  - Argumente „fließen“ in die Call-Operation, Rückgabewerte hinaus
  - Datenfluss trotz seiteneffektfreier Operationen

$\Rightarrow$ `call F1` ist eine Komplexoperation (= zusammengesetzte Operation)

Rekursion: Invokation und Selektion



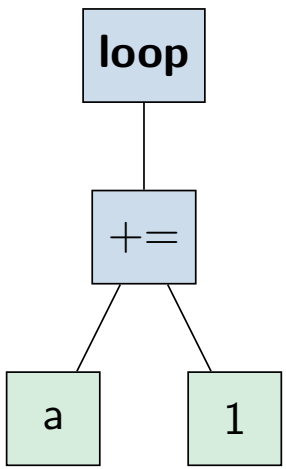


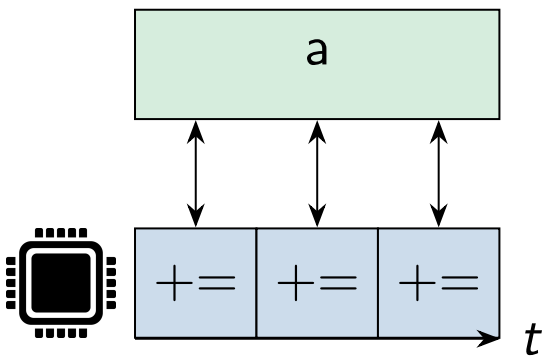
- Rekursion ist das Zusammenspiel von Invokation und Selektion.
  - `call` erlaubt uns, Operationen mehrfach auszuführen.
  - `ite` erlaubt uns, die Invokation irgendwann auszulassen.
  - Selbe Operation mit unterschiedlichen Daten auswerten (α_1, α_2,...)
- Warum macht das überhaupt Sinn?
  - `call` formt nicht nur den Kontrollfluss, sondern auch den Datenfluss!
  - Argumente „fließen“ in die Call-Operation, Rückgabewerte hinaus
  - Datenfluss trotz seiteneffektfreier Operationen

$\Rightarrow$ `call F1` ist eine Komplexoperation (= zusammengesetzte Operation)

Iteration: Dasselbe, immer wieder tun

- Operationen in Schleifen werden wiederholt ausgeführt.
  - Nur Kontrollfluss-, keine Datenflusseigenschaften (⇔ Invokation)
  - Selbe Operation, selbe Operanden, immer wieder auswerten.
  - Das macht **nur Sinn**, wenn die Operationen **Seiteneffekte** haben

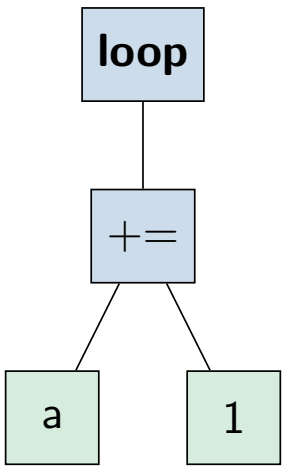


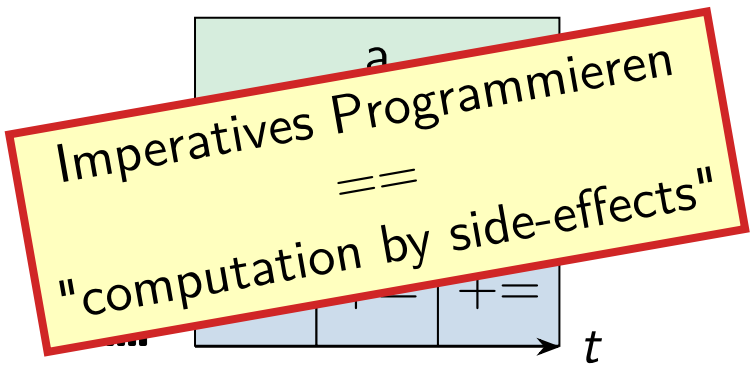


- Wie oft wird der Schleifenkörper (loop body) ausgewertet?
  - Logically-Controlled: Wiederholung bis die Abbruchbedingung fehlschlägt
  - Enumeration-Controlled: Wiederholung für jedes Element einer Aufzählung

Iteration: Dasselbe, immer wieder tun

- Operationen in Schleifen werden wiederholt ausgeführt.
  - Nur Kontrollfluss-, keine Datenflusseigenschaften (⇔ Invokation)
  - Selbe Operation, selbe Operanden, immer wieder auswerten.
  - Das macht **nur Sinn**, wenn die Operationen **Seiteneffekte** haben





- Wie oft wird der Schleifenkörper (loop body) ausgewertet?
  - Logically-Controlled: Wiederholung bis die Abbruchbedingung fehlschlägt
  - Enumeration-Controlled: Wiederholung für jedes Element einer Aufzählung

Logically-Controlled Loops

Test am Anfang

```c
while (cond) {
  stmt1;
  stmt2
}
```

Test am Ende

```c
do {
  stmt1;
  stmt2
} while (cond)
```

- Logisch kontrollierte Schleifen unterscheiden sich kaum
  - Liefert der Test ein negatives Ergebnis, wird die Schleife abgebrochen.
  - Zeitpunkt des Tests: **Vor oder nach** jedem Durchlauf?
- Komplexere Schleifentypen können in einfachere umgewandelt werden.

```c
for (init; cond; next) {
  stmt1;
  stmt2;
}
```

$\Rightarrow$

```c
init;
while(cond) {
  stmt1; stmt2;
  next;
}
```

- Spezialisierte `goto`-Befehle für Schleifen
  - `break`: Sprung hinter die Schleife
  - `continue`: Sprung ans Endes des Schleifenkörpers

Enumeration-Controlled Loops

```vector<int> nums;

for (int elem : nums){
  stmt1; stmt2;
}
```
C++

```numbers = [23, 42, 65]

for elem in numbers:
   stmt1
   stmt2
```
Python

```! [1, 3, 5, 7, 9]
do i = 1, 10, 2
  stmt1;
  stmt2;
end
```
Fortran

- Aufzählungsschleifen iterieren über eine Aufzählung.
  - Aufzählung liefert eine Sequenz von Objekten
  - Für jedes Sequenz-Element wird der Schleifenkörper einmal ausgeführt
  - Aktuelles Element wird an eine lokale Variable **gebunden**

`enumerate(tree)` $\rightarrow$ `node, node, node, node,…`

- Bei der Aufzählung muss man sich für eine Ordnung entscheiden
  - Vorgegeben bei Sequenzen/Listen und Arrays
  - Beliebig bei ungeordneten Mengen (häufig: internen Datenstruktur)
  - Pre-, Post-, oder In-Order bei Bäumen

Iteratoren und Generatoren

- Aufzählungen können als eigenständige Objekte existieren.
  - Iterator: Ein Objekt, auf dem man `next(obj)` ausführen kann.
  - Generator: Eine pausierte Funktionsinstanz (Continuation)

Iterator

```python
class FancyList(list):
  def __iter__(self):
    return PairIterator(self)

class PairIterator:
  def __init__(self, seq):
    self.seq = seq[:]

  def __next__(self):
    if not self.seq:
       raise StopIteration
        ret = self.seq[0:2]
    del self.seq[0:2]
    return ret

for p in FancyList([1,2,3,4]):
  print(p)
```

Python3

Generatoren

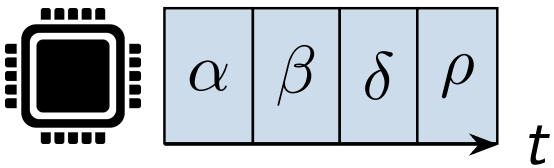
```python
def pairs(seq):
  i = 0
  while i + 1 < len(seq):
    yield seq[i:i+2]
    i += 2

for pair in pairs([1,2,3,4]):
  print(pair)
```

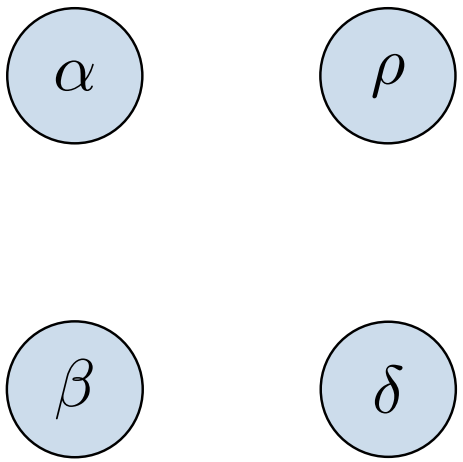
Python

- `pairs()` erzeugt Funktionsinstanz
- Bei jedem Schleifendurchgang:
  - **Fortsetzung** bis `yield`
  - Generator liefert einen Wert
  - Funktion wird wieder pausiert

Kontrollflüsse am Beispiel

Erinnerung: – Ein Kontrollfluss ist eine Sequenz von Operationen.
– Mit Sprachkonstrukten manipulieren wir den Kontrollfluss.

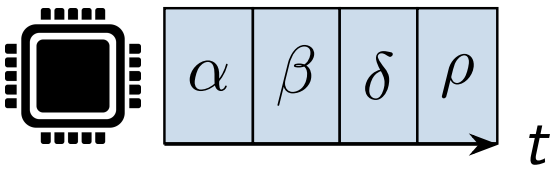


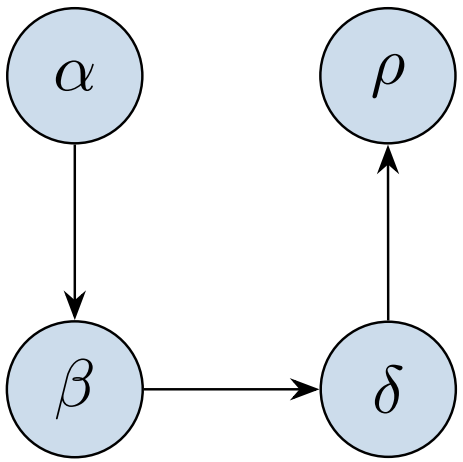


Intuition: Im Kontrollflussgraphen sind alle potentiellen Kontrollflüsse.

Kontrollflüsse am Beispiel

Erinnerung: – Ein Kontrollfluss ist eine Sequenz von Operationen.
– Mit Sprachkonstrukten manipulieren wir den Kontrollfluss.

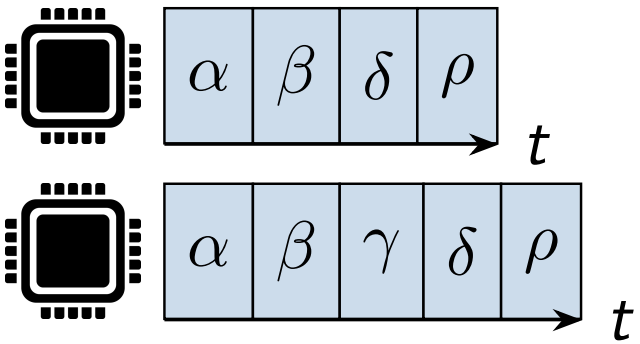


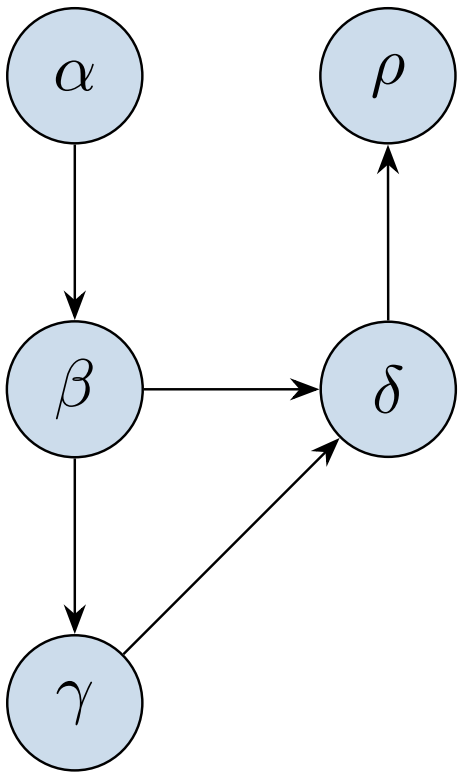


Intuition: Im Kontrollflussgraphen sind alle potentiellen Kontrollflüsse.

Kontrollflüsse am Beispiel

Erinnerung: – Ein Kontrollfluss ist eine Sequenz von Operationen.
– Mit Sprachkonstrukten manipulieren wir den Kontrollfluss.

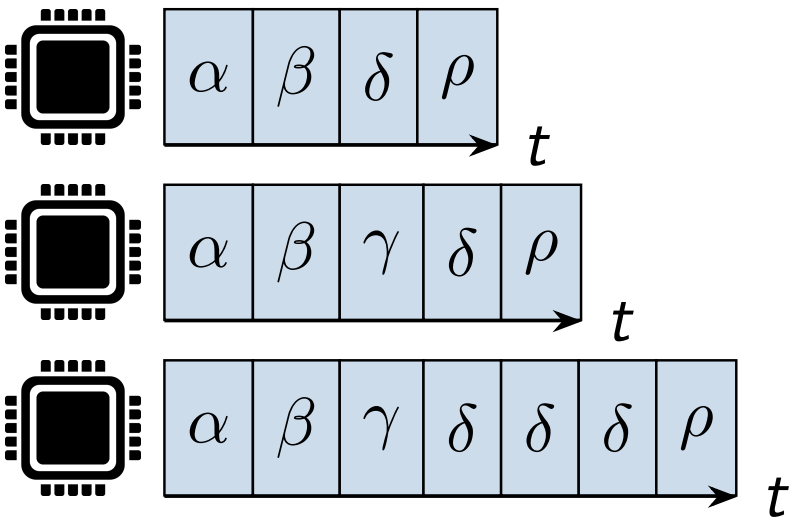


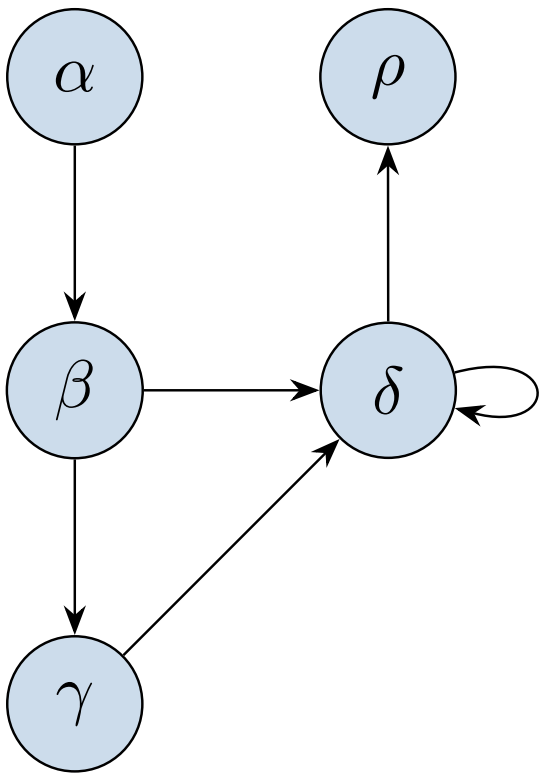


Intuition: Im Kontrollflussgraphen sind alle potentiellen Kontrollflüsse.

Kontrollflüsse am Beispiel

Erinnerung: – Ein Kontrollfluss ist eine Sequenz von Operationen.
– Mit Sprachkonstrukten manipulieren wir den Kontrollfluss.

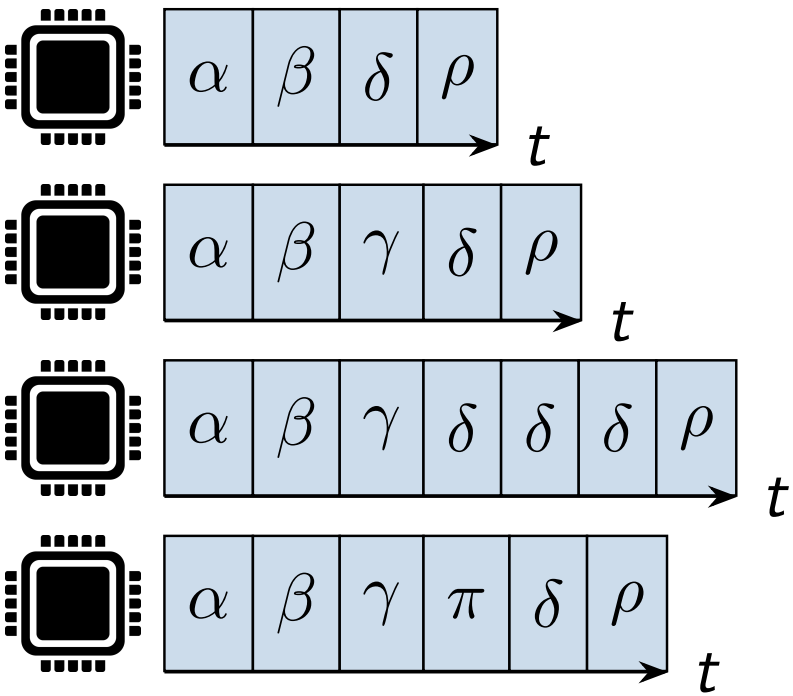


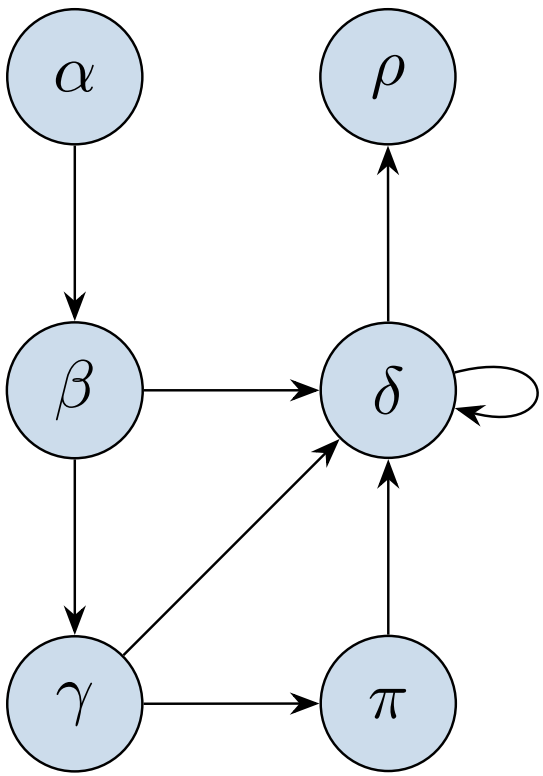


Intuition: Im Kontrollflussgraphen sind alle potentiellen Kontrollflüsse.

Kontrollflüsse am Beispiel

Erinnerung: – Ein Kontrollfluss ist eine Sequenz von Operationen.
– Mit Sprachkonstrukten manipulieren wir den Kontrollfluss.

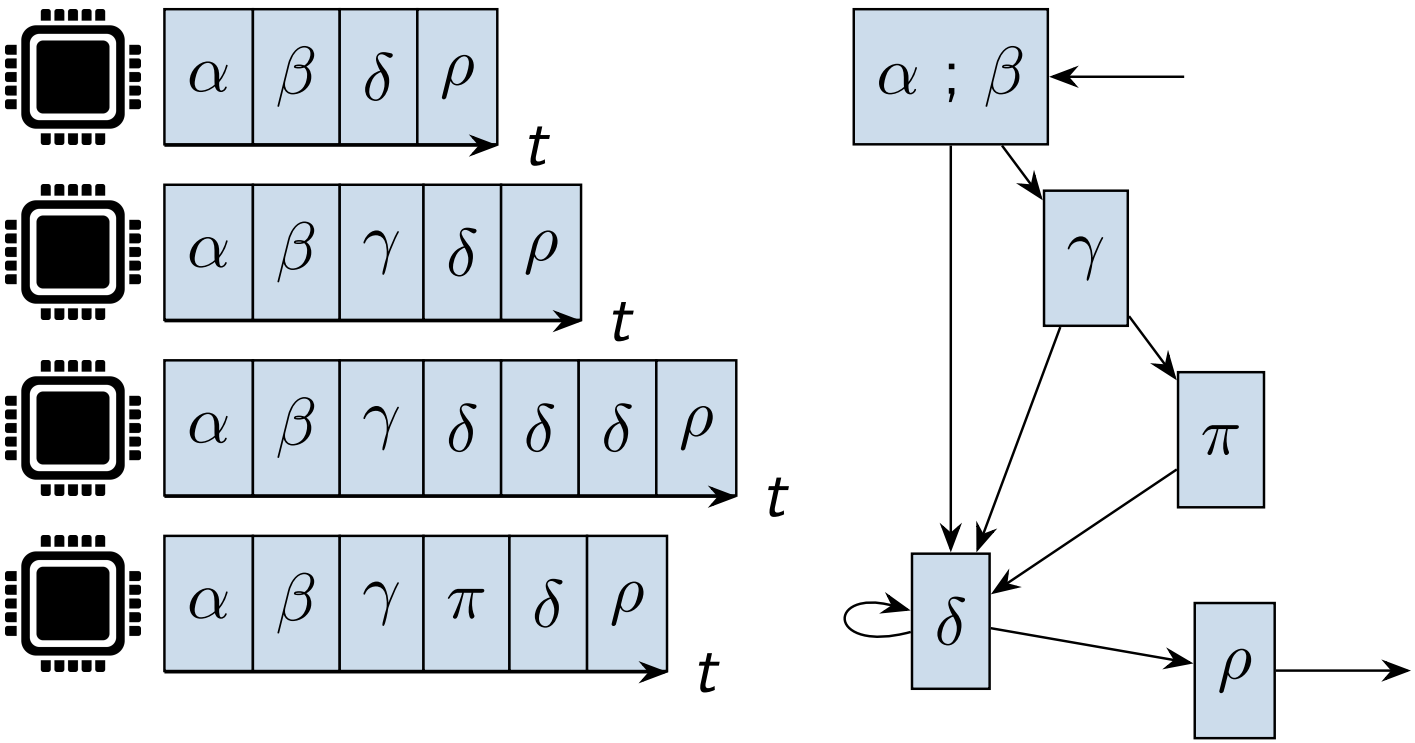




Intuition: Im Kontrollflussgraphen sind alle potentiellen Kontrollflüsse.

Kontrollflüsse am Beispiel

Erinnerung: – Ein Kontrollfluss ist eine Sequenz von Operationen.
– Mit Sprachkonstrukten manipulieren wir den Kontrollfluss.



Intuition: Im Kontrollflussgraphen sind alle potentiellen Kontrollflüsse.

Der Kontrollflussgraph

Definition: Kontrollflussgraph (CFG)

Der CFG ist ein gerichteter Graph mit Basisblöcke als Knoten; Kanten entsprechen den möglichen Übergängen zwischen den Blöcken.

Definition: Basisblock (BB)

Ein Basisblock ist eine Sequenz von Operationen.
Die Sequenz **immer** komplett abgearbeitet.

- Fakten für Basisblöcke, die aus den Definitionen folgen
  - Nur die erste Instruktion eines BB kann angesprungen werden
  - Nur die letzte Instruktion eines BB kann ein Sprung sein
  - Eine einzelne Operation ist bereits ein minimaler Basisblock
- Fakten über den CFG, die aus den Definitionen folgen
  - Können zwei BBs hintereinander ausgeführt werden, existiert eine CFG-Kante
  - **Aber:** Nicht jeder Pfad durch den CFG ist ein valider Kontrolfluss

Kontrollflussgraphen im Übersetzungsprozess

- Übersetzer: AST → CFG
  - CFG ist zentrale Datenstruktur
  - Codeerzeugung: if,… → `goto`
  - Nur Sprünge (und „Durchfallen“)
  - Bereits nahe an realen Maschinen

```
.BB0:  mov EAX, 1
       cmp ESI, 31
       jl .BB3
       mov EAX, 3
       cmp EDX, 14
       jl .BB3
       mov EAX, 13
.BB3:  add EAX, EAX
       inc EDI
       jne .BB3
       ret
```

Kontrollflussgraphen im Übersetzungsprozess

- Übersetzer: AST → CFG
  - CFG ist zentrale Datenstruktur
  - Codeerzeugung: if,… → `goto`
  - Nur Sprünge (und „Durchfallen“)
  - Bereits nahe an realen Maschinen
- CFG-Struktur im Maschinencode

```
.BB0:  mov EAX, 1
       cmp ESI, 31
       jl .BB3
       mov EAX, 3
       cmp EDX, 14
       jl .BB3
       mov EAX, 13
.BB3:  add EAX, EAX
       inc EDI
       jne .BB3
       ret
```

Kontrollflussgraphen im Übersetzungsprozess

- Übersetzer: AST → CFG
  - CFG ist zentrale Datenstruktur
  - Codeerzeugung: if,… → `goto`
  - Nur Sprünge (und „Durchfallen“)
  - Bereits nahe an realen Maschinen
- CFG-Struktur im Maschinencode
  - BB endet nach Sprüngen

```
.BB0:  mov EAX, 1
       cmp ESI, 31
       jl .BB3
-------------------
       mov EAX, 3
       cmp EDX, 14
       jl .BB3
-------------------
       mov EAX, 13
.BB3:  add EAX, EAX
       inc EDI
       jne .BB3
-------------------
       ret
```

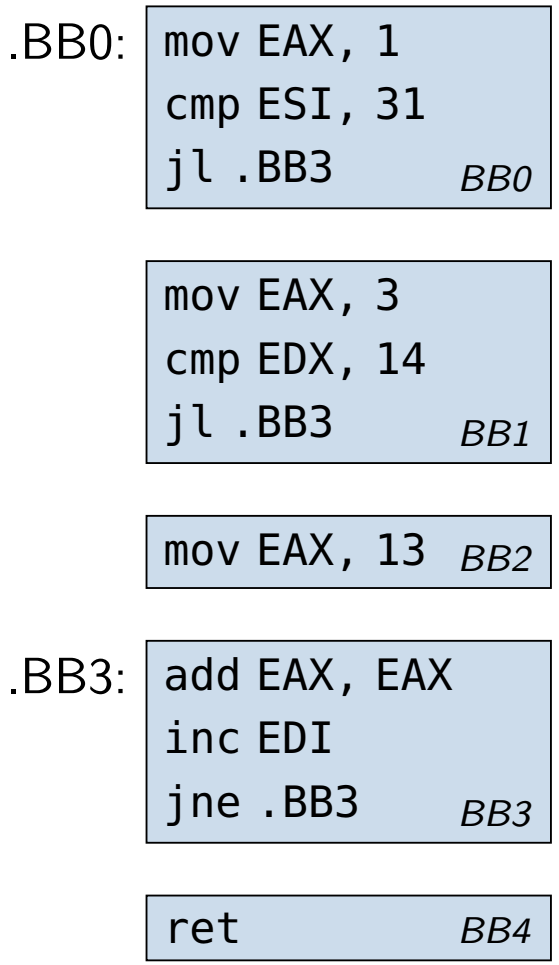
Kontrollflussgraphen im Übersetzungsprozess

- Übersetzer: AST → CFG
  - CFG ist zentrale Datenstruktur
  - Codeerzeugung: if,… → `goto`
  - Nur Sprünge (und „Durchfallen“)
  - Bereits nahe an realen Maschinen
- CFG-Struktur im Maschinencode
  - BB endet nach Sprüngen
  - BB endet vor Sprungmarken

```
.BB0:  mov EAX, 1
       cmp ESI, 31
       jl .BB3
-------------------
       mov EAX, 3
       cmp EDX, 14
       jl .BB3
-------------------
       mov EAX, 13
-------------------
.BB3:  add EAX, EAX
       inc EDI
       jne .BB3
-------------------
       ret
```

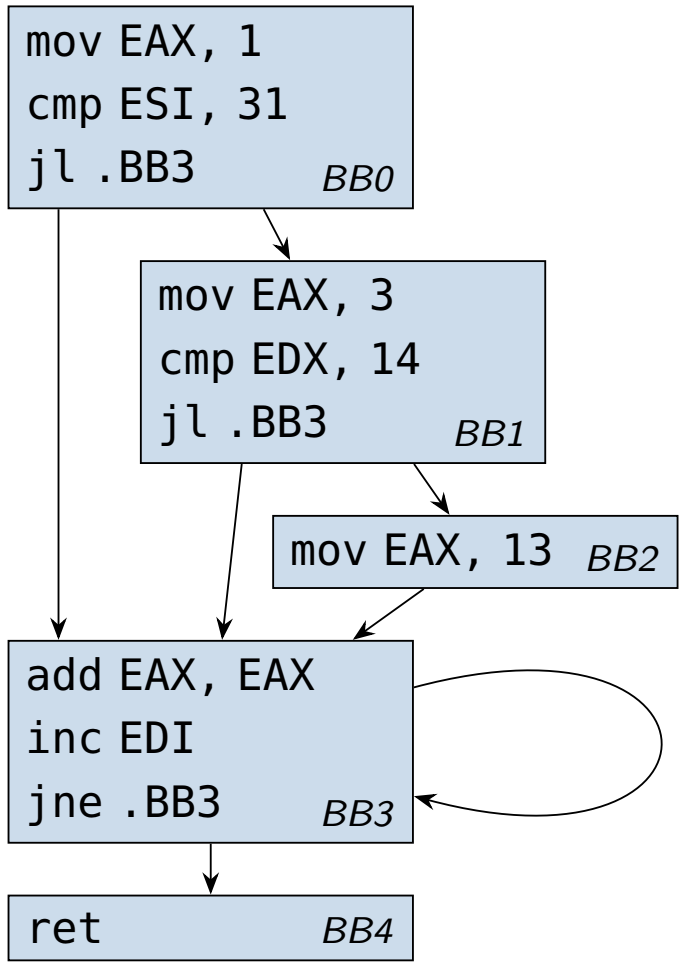
Kontrollflussgraphen im Übersetzungsprozess

- Übersetzer: AST → CFG
  - CFG ist zentrale Datenstruktur
  - Codeerzeugung: if,… → `goto`
  - Nur Sprünge (und „Durchfallen“)
  - Bereits nahe an realen Maschinen
- CFG-Struktur im Maschinencode
  - BB endet nach Sprüngen
  - BB endet vor Sprungmarken
  - Maximale Basisblöcke



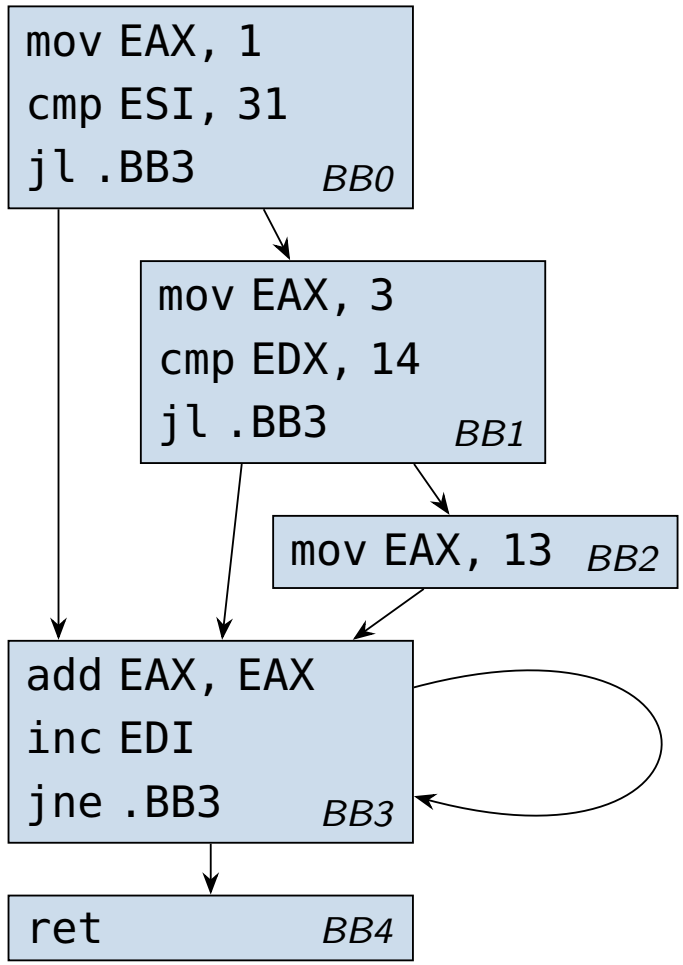
Kontrollflussgraphen im Übersetzungsprozess

- Übersetzer: AST → CFG
  - CFG ist zentrale Datenstruktur
  - Codeerzeugung: if,… → `goto`
  - Nur Sprünge (und „Durchfallen“)
  - Bereits nahe an realen Maschinen
- CFG-Struktur im Maschinencode
  - BB endet nach Sprüngen
  - BB endet vor Sprungmarken
  - Maximale Basisblöcke
  - Kanten anhand von Sprüngen und beim Durchfallen



Kontrollflussgraphen im Übersetzungsprozess

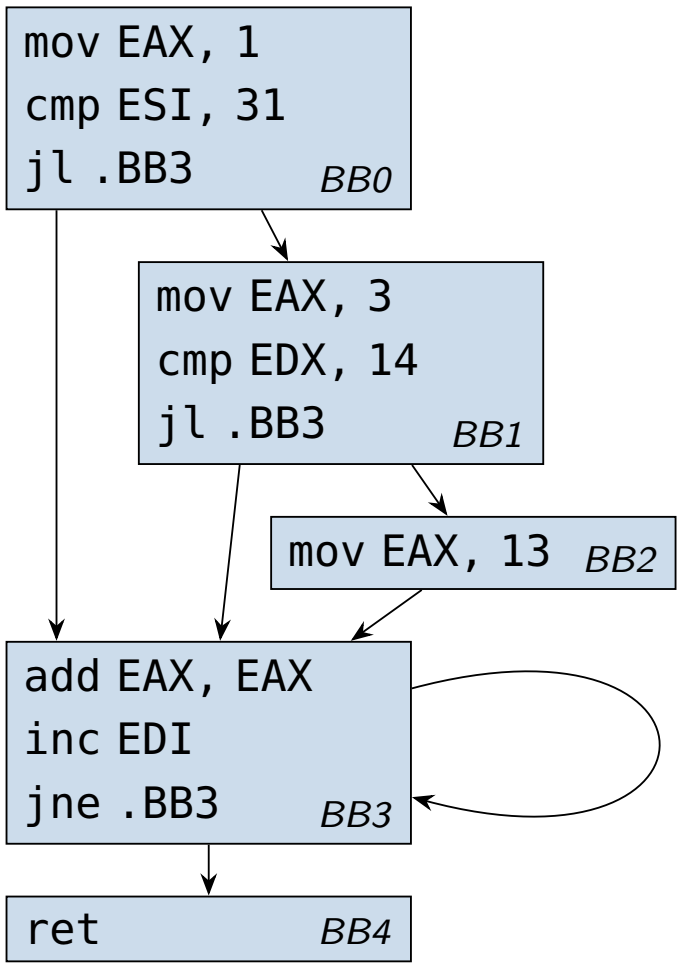
- Übersetzer: AST → CFG
  - CFG ist zentrale Datenstruktur
  - Codeerzeugung: if,… → `goto`
  - Nur Sprünge (und „Durchfallen“)
  - Bereits nahe an realen Maschinen
- CFG-Struktur im Maschinencode
  - BB endet nach Sprüngen
  - BB endet vor Sprungmarken
  - Maximale Basisblöcke
  - Kanten anhand von Sprüngen und beim Durchfallen
- Vorteile des CFG vs. AST
  - Flachgeklopfte Hierarchie
  - Nähe zum Maschinenmodell
  - Optimierung durch Graphalgorithmen



Kontrollflussgraphen im Übersetzungsprozess

```c
int foo(int c, int a, int b) {
    int i = 0;
    i += c;
    if (a > 30) {
        i += 2;
        if (b > 13) {
            i += 10;
        }
    }
    do {
        i *= 2;
    } while(c--);
    return i;
}
```

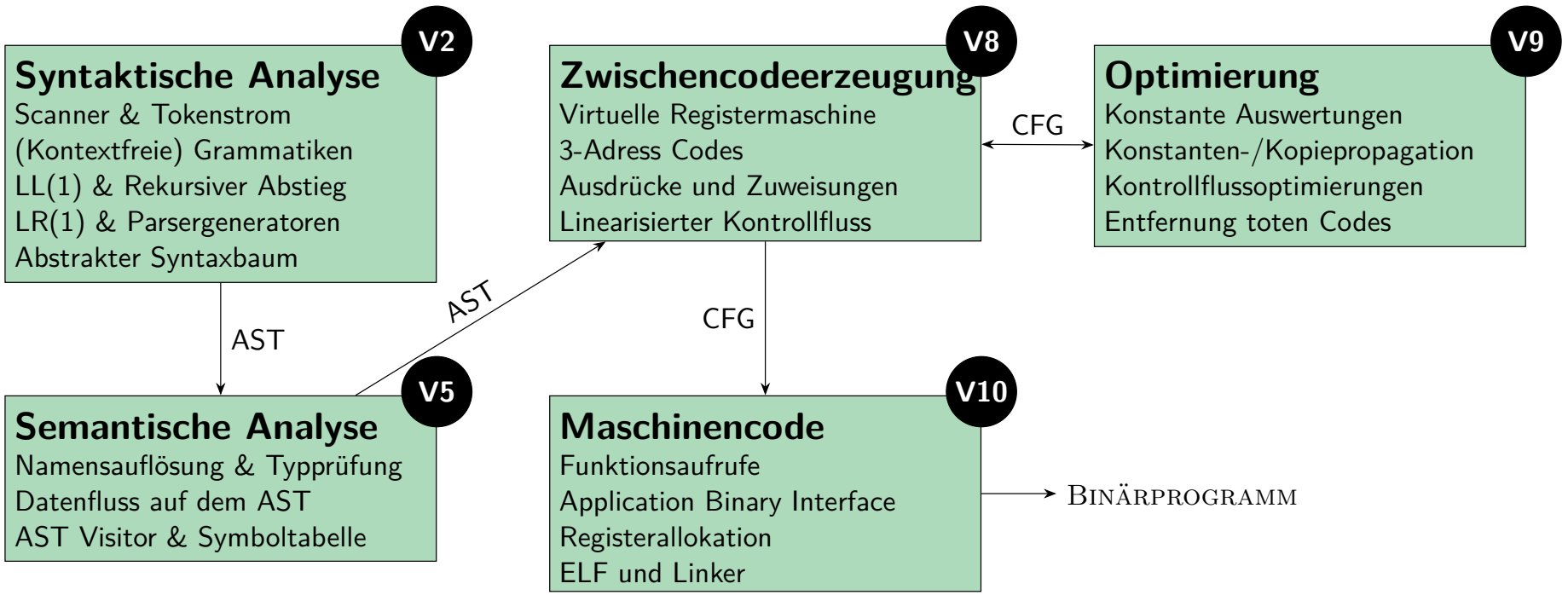
C



Nächste Vorlesung:

Codeerzeugung (AST→CFG)

Einordnung des CFGs in die Vorlesung



Kontrollflussgraph auf mehreren Ebenen

Frage: Beendet ein `call` den aktuellen Basisblock?

Kontrollflussgraph auf mehreren Ebenen

Frage: Beendet ein `call` den aktuellen Basisblock?

Nein: Calls sind Komplexbefehle

⇓

Funktionslokaler CFG

- Nur Sprünge teilen einen BB
- CFG überdeckt nur eine Funktion
- Anwendung: lokalen Optimierungen

⇒ Der gängige CFG

foo:

```
mov EAX, 1
call bar
inc EAX
ret          BB0
```

bar:

```
inc EAX
ret          BB10
```

Kontrollflussgraph auf mehreren Ebenen

Frage: Beendet ein `call` den aktuellen Basisblock?

Nein: Calls sind Komplexbefehle

⇓

Funktionslokaler CFG

- Nur Sprünge teilen einen BB
- CFG überdeckt nur eine Funktion
- Anwendung: lokalen Optimierungen

⇒ Der gängige CFG

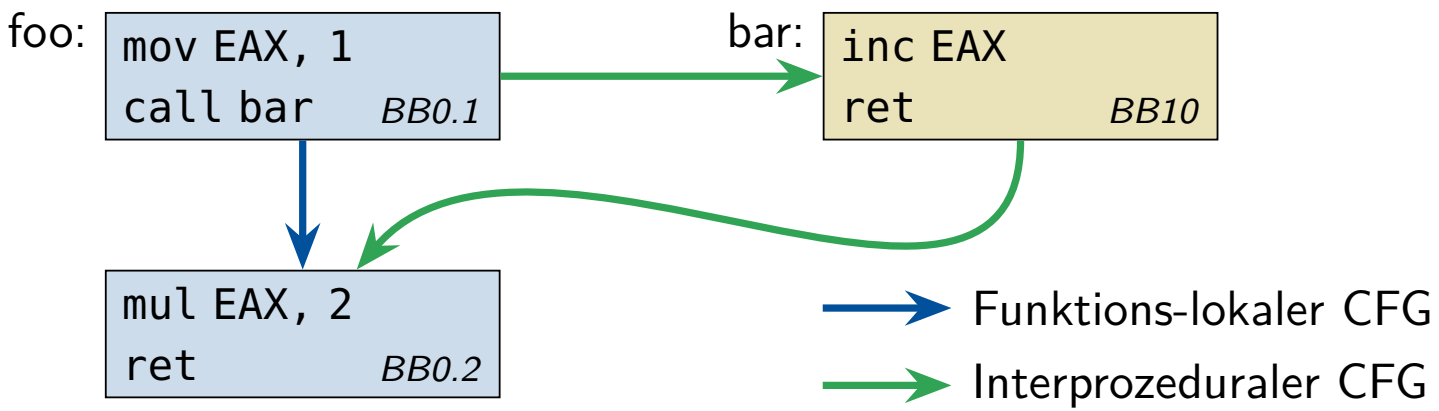
Ja: Ein Call ist ein Jump

⇓

Interprozeduraler CFG

- Kante: `call`→Aufgerufene Funktion
- Alle Blöcke aus allen Funktionen
- Whole-Program Analysis

⇒ Der selten verwandte CFG



Kontrollflussgraph auf mehreren Ebenen

Frage: Beendet ein `call` den aktuellen Basisblock?

Nein: Calls sind Komplexbefehle

⇓

Funktionslokaler CFG

- Nur Sprünge teilen einen BB
- CFG überdeckt nur eine Funktion
- Anwendung: lokalen Optimierungen

⇒ Der gängige CFG

Ja: Ein Call ist ein Jump

⇓

Interprozeduraler CFG

- Kante: `call`→Aufgerufene Funktion
- Alle Blöcke aus allen Funktionen
- Whole-Program Analysis

⇒ Der selten verwandte CFG

foo:

```
mov EAX, 1
call bar
inc EAX
ret                BB0
```

bar:

```
inc EAX
ret                BB10
```

Zusammenfassung: Operationen

- Operationen lesen Objekte, verarbeiten Daten und liefern Ergebnisse.
  - Abhängigkeiten zu anderen Operationen müssen zuerst ausgewertet werden.
  - Seiteneffekte: Operation verändert ein Objekt oder eine Variable.
  - Auswertungsreihenfolge bildet Operationen auf lineare Operationssequenz ab
- Sprachkonstrukte formen die möglichen Kontrollflüsse
  - Sequenzierung Operationen werden direkt hintereinander ausgeführt.
  - Invokation Einschub einer Funktionsausführung, Komplexbefehl
  - Selektion Operationen durch dynamische Entscheidungen auslassen
  - Iteration Operationen wegen ihrer Seiteneffekte mehrfach ausführen
- Kontrollflussgraphen überdecken die möglichen Kontrollflüsse.
  - Basisblöcke sind ununterbrochene Operationssequenzen.
  - **Ausblick**: Ergebnis der Codeerzeugung und Grundlage für die Optimierung