



Technische
Universität
Braunschweig



Programmiersprachen und Übersetzer

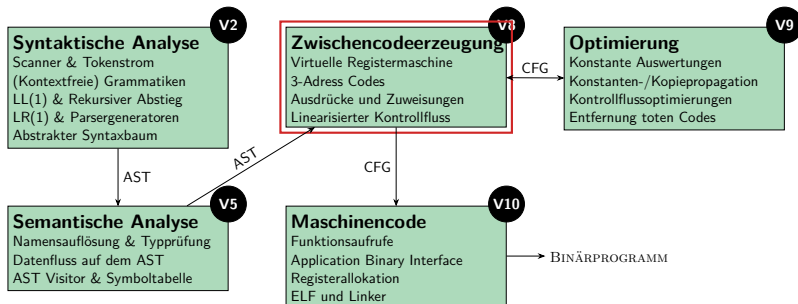
08 - Zwischencodeerzeugung

Christian Dietrich

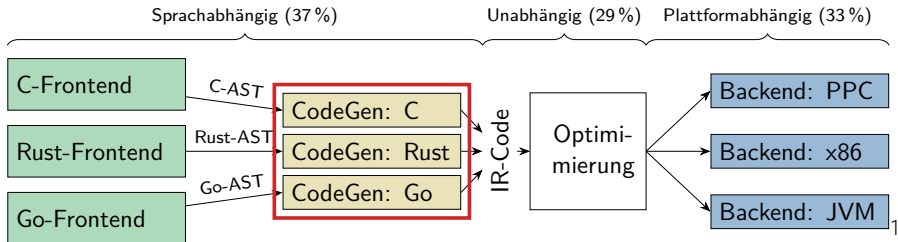
Sommersemester 2024



Einordnung in die Vorlesung: Zwischencodeerzeugung

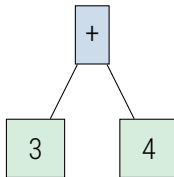


- **Zwischencodeerzeugung:** Linearisierung und Übersetzung des ASTs
 - **Zielarchitektur:** virtuelle Maschinen mit wohldefiniertem Funktionsumfang
 - Konkret genug für Assemblererzeugung, abstrakt genug für Optimierungen
 - Wiederverwendung des Middle- und Backend für **mehrere Frontend-Sprachen**



- Die Konstruktion eines Übersetzers ist aufwendig und schwierig!
 - Assembler im Parser erzeugen ist (manchmal) möglich (One-Pass Compiler)
 - Wiederverwendung von Komponenten erleichtert das portieren (~60 % Re-Use)
- **Zwischenrepräsentation** erhält die Programmsemantik
 - AST ist bereits eine **Intermediate Representation** (IR-Form)
 - Der klassische IR-Code: CFG mit Instruktionen für eine sequentielle Maschine

¹Zahlen für Clang (C/C++ Frontend, 700k) und LLVM 9.0 (18 Backends, 1,4M)



Stack-Maschine

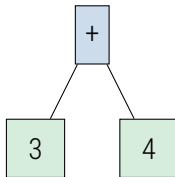
- Berechnung findet auf Stack statt
- Operanden gehen über den Stack
- Datenfluss ist implizit
- Unendlich großer Stack

```
push 3
push 4
add      // = push(pop() + pop())
```

Register-Maschine

- Operationen manipulieren Register
- Operanden gehen durch „Register“
- Register machen Datenfluss sichtbar
- Beliebig viele Register

```
r0 = 3
r1 = 4
r3 = add r1, r2
```



Stack-Maschine

- Berechnung findet auf Stack statt
- Operanden gehen über den Stack
- Datenfluss ist implizit
- Unendlich großer Stack

```
// code(Op1) -> 1 Element am Stack  
// code(Op2) -> 1 Element am Stack  
add      // = push(pop() + pop())
```

Register-Maschine

- Operationen manipulieren Register
- Operanden gehen durch „Register“
- Register machen Datenfluss sichtbar
- Beliebig viele Register

```
// r1 = code(Op1)  
// r2 = code(Op2)  
r3 = add r1, r2
```

⇒ Codeerzeugung für Stack- und Register-Maschinen sieht ähnlich aus.

Vorneweg: Die Spezifikation einer virtuellen Maschine ist schwierig.

- Wahl des Abstraktionsniveaus: Endlicher oder **unendlicher Registersatz**?
 - Abstraktere Semantik $\Rightarrow$ IR-Code ist näher an Programmstruktur
 - Einfachere Codeerzeugung und Optimierung, komplexeres Backend
 - Ausführungsmodell: **Register-Maschine** oder Stack-Maschine?
 - Stack-Maschinen haben besonders kompakte Programmdarstellung
 - Register-Maschinen sind näher an real existierenden Prozessoren
 - Verbraucher der IR-Form: Interpreter oder **Übersetzer-Backend**?
 - Übersetzer brauchen eine gut analysierbare und veränderbare IR-Form
 - Bei Interpretern soll die Ausführung möglichst schnell starten
- $\Rightarrow$ Wir wählen eine Register-Maschine mit unendlich vielen Registern

➤ 3-Adress-Code (Quadrupel-Notation)

- Programm wird als Folge von Quadrupeln dargestellt
 - Quadrupel besteht aus einem Operator und **maximal** drei Operanden

$$x := op\ y, z$$

- Operanden (x, y, z) können sein:
 - Variablen/temporäre Register halten ein Maschinenwort
 - Literale als konstante Operanden
 - Sprungmarke adressieren andere Operationen
- Nur (un-)bedingte Sprünge, keine Schleifen oder Bedingungen
 - Sprachkonstrukte werden mit **goto/ifgoto** linearisiert
 - 3-Adress-Code passt perfekt in Basisblock/CFG Struktur

Beispiel: `if ((*a) <= (4 + b)) ...`

```
t1 := Load *a
```

```
t2 := Add 4, b
```

```
t3 := LessEqual t1, t2
```

```
IfGoto t3, .BB1, .BB2
```

Für Vorlesung und Übung entwickeln wir einen Zwischencode auf Basis einer Register-Maschine mit unendlich vielen Registern. Dazu schauen wir uns Semantik und Notation an.

■ Operanden können sein:

- (C) Konstante Ganzzahlen: 3, 4
- (L) Statische Label: .BB2
- (V) Variablen: a, t1

- **Variablen** sind funktionslokal
- Beliebig viele temporäre Variablen
- Alle Werte sind 32-Bit breit
- Keine Typisierung

■ Einfache Operationen (6 Befehle)

- **Assign**: Einer Variable einen Wert zuweisen
- 4 arithmetische Operationen: Add, Sub, Mul, Div
- 1 logische Operation: **LessEqual**
Falsch: „0“, Wahr: „!= 0“

V := Assign CV
V := Add CV, CV
V := LessEqual CV, CV

➤ Für PSÜ: Ein beispielhafter Zwischencode (2)

■ Speicheroperationen (3 Befehle)

- **Ref:** Erzeugen einer Referenz
- **Load:** Lesen des adressierten Speicherwortes
- **Store:** Schreiben eines Wertes an ein Speicherstelle

$V := \text{Ref } V$
 $V := \text{Load } *V$
 $*V := \text{Store } CV$

■ Speichermanagement (3 Befehle)

- **StackAlloc:** Speicherallokation im Call-Frame
- **HeapAlloc:** Speicherallokation dem Heap
- **HeapFree:** Freigabe eines Heap-Objekts

$V := \text{StackAlloc } C$
 $V := \text{HeapAlloc } C$
 $\text{HeapFree } V$

■ Funktionslokale Kontrollflüsse (2 Befehle)

- **Goto:** Sprung an das gegebene Label
- **IfGoto:** Bedingter Sprung

$\text{Goto } L$
 $\text{IfGoto } CV, L, L$

■ Interprozedurale Kontrollflüsse (2 Befehle)

- **Call:** Funktionsaufruf mit beliebig vielen Argumenten
- **Return:** Rückkehr aus einer Funktionsinstanz

$\text{Call } F, [CV]^*$
 $\text{Return } CV$

➤ Für PSÜ: Ein Beispielhafter Zwischencode (3)

■ Basisblöcke

- Geordnete Liste von Operationen
 - Jeder Basisblock hat ein (funktionslokal) eindeutiges Label .BB0,...
 - Der letzte Befehl muss ein **Goto**, **IfGoto**, oder **Return** sein.
- ⇒ Maximale Basisblöcke

■ Funktionen

- Ungeordnete Menge von Basisblöcken mit einem Eingangsblock
- Funktionen haben ein Label, können aber nicht direkt angesprungen werden
- Beliebig viele lokale Variablen und temporäre Register
- Beliebig viele Parameter, die als Variablen zugänglich sind

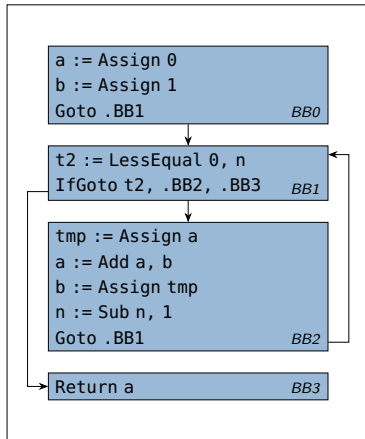
```
class BasicBlock:  
    def __init__(self, label):  
        self.label = label  
        self.instructions = []
```

```
class Function:  
    def __init__(self, label):  
        self.label = label  
        self.parameters = []  
        self.variables = []  
        self.entry_block = None  
        self.basic_blocks = []
```



Iterativer Fibonacci als Zwischencode

```
fib_iter(n : int) : int
```



```
parameters = [n]
variables = [a, b, t2, tmp]
basic_blocks = {BB0, BB1, BB2, BB3}
entry_block = BB0
```

Beobachtungen

- BB0 initialisiert die Startwerte
- BB1/2 sind eine while-Schleife
- Goto .BB1 spaltet BB0/1

```
func fib_iter(n : int) : int {
  var a : int;
  var b : int;
  a := 0;
  b := 1;
  while (n >= 0) {
    var tmp : int;
    tmp := a;
    a := a + b;
    b := tmp;
    n := n - 1;
  }
  return a;
}
```

➤ Notwendige Operationen auf CFG und Basisblöcken

Schlachtplan: Wir traversieren den AST einer Funktion und erzeugen dabei einen CFG mit Basisblöcken voller IR-Operationen.

■ Erstelle ein Funktionsobjekt

`Programm.create_function :: string ↦ Function`

- Funktionen haben den Namen als Label (`func.label`).
- Parameter und Variablen werden beim Traversieren erstellt.

■ Erstelle einen Basisblock

`Function.create_block :: () ↦ BasicBlock`

■ Erstelle einen Parameter

`Function.create_parameter :: name ↦ Variable`

■ Erstelle eine Variable

`Function.create_variable :: ↦ Variable`

- Variablen und Parameter werden im Funktionsscope verwaltet.
- Der erste erstellte Basisblock wird automatisch zum Eingangsblock.

■ Hänge eine Operation an

`BasicBlock.append :: Operation, Operands* ↦ ()`

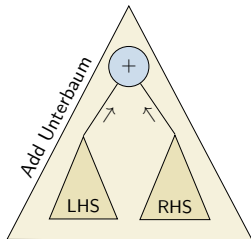
- Das Label für den Basisblock wird automatisch von der Funktion vergeben.
- Ein `Goto/IfGoto` schließt den Basisblock ab.

Erinnerung: Ausdrücke sind AST-Unterbäume, die Werte berechnen.

- Jeder AST-Knotentyp erzeugt anderen Code.
 - Dynamischer Dispatch anhand des Knotentyps $\Rightarrow$ Visitor-Pattern
 - Das Ergebnis des Ausdrucks wird in einer temporäre Variable gespeichert.

Wir werden beliebig viele virtuelle Register erzeugen. Die Abbildung auf reale Register ist Aufgabe der Maschinencodeerzeugung

- Wer entscheidet, wo das Ergebnis des Ausdrucks gespeichert wird?
 - **Bottom-Up:** Codegenerierung des Unterbaums erzeugt die Zielvariable
`rvalue_Add :: Add  $\mapsto$  Variable`
 - **Top-Down:** Codegenerierung bekommt Zielvariable als Parameter
`rvalue_Add :: Add, Variable  $\mapsto$  ()`
 - Beide Strategien weitgehend äquivalent und Mischungen sind möglich.



```
class CodeGen:
```

```
...
```

```
def rvalue_Add(self, expr):  
    l = self.rvalue(expr.LHS) # <- dynamic dispatch  
    r = self.rvalue(expr.RHS) # <-  
    ret = self.current_function.create_variable()  
    self.current_block.append(Add, ret, l, r)  
    return ret
```

- Codegenerierung delegiert das Berechnen der Unterbäume
 - Alle Visit-Funktionen für Ausdrücke liefern Ergebnis als Rückgabewert
 - Getrennte Visitor-Methoden für R- und L-Werte
- Explizite Traversierung bei der Codeerzeugung
 - Wir verwenden absichtlich keine automatische Traversierung des Baums
 - Generierungsfunktionen verwenden Referenz auf aktuelle Funktion/Block
 - Sie haben als **Seiteneffekt** das **Emittieren** einzelner Instruktionen



Code-Erzeugung für Funktionsaufruf und Zuweisung

■ Funktionsaufrufe

```
def rvalue_CallExpr(self, call_expr):  
    args = [self.rvalue(arg) for arg in call_expr.arguments]  
    callee = call_expr.callee.decl.ir_obj  
    ret = self.current_function.create_variable()  
    self.current_block.append(Call, ret, callee, *args)  
    return ret
```

- Wir entscheiden über eine Auswertungsreihenfolge der Argumente
- Auffinden der IR-Funktion über Querverbindung zur AST-Deklaration

■ Zuweisung braucht den L-Wert der linken Seite

- Spezieller Visitor für die linken Seiten liefert Referenz `lvalue_Variable()`
- Funktioniert immer: Emittieren eines **Store**-Befehls
- **Optimierung**: Sonderfall für Variablen auf der linken Seite macht IR kompakter

```
t0 := Add 2, 3  
t1 := Reference var  
*t1 := Store t0
```

```
t0 := Add 2, 3  
var := Assign t0
```

- **Sequenzierung**: Sequenzielle Codegenerierung für alle Statements

```
def visit_Block(self, block):  
    for stmt in block.statements:  
        self.visit(stmt)
```

- Unterfunktionen hängen Operationen *implizit* an den `current_block` .
- Invariante: Nach jedem Statement kommt der Kontrollfluss wieder zusammen.

- **Sequenzierung**: Sequenzielle Codegenerierung für alle Statements

```
def visit_Block(self, block):  
    for stmt in block.statements:  
        self.visit(stmt)
```

- Unterfunktionen hängen Operationen *implizit* an den `current_block`.
- Invariante: Nach jedem Statement kommt der Kontrollfluss wieder zusammen.

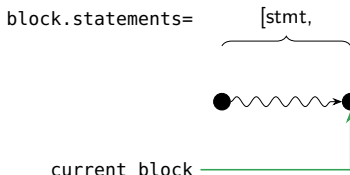
`block.statements=`



- **Sequenzierung**: Sequenzielle Codegenerierung für alle Statements

```
def visit_Block(self, block):  
    for stmt in block.statements:  
        self.visit(stmt)
```

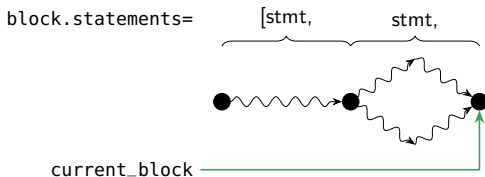
- Unterfunktionen hängen Operationen *implizit* an den `current_block`.
- Invariante: Nach jedem Statement kommt der Kontrollfluss wieder zusammen.



- **Sequenzierung**: Sequenzielle Codegenerierung für alle Statements

```
def visit_Block(self, block):  
    for stmt in block.statements:  
        self.visit(stmt)
```

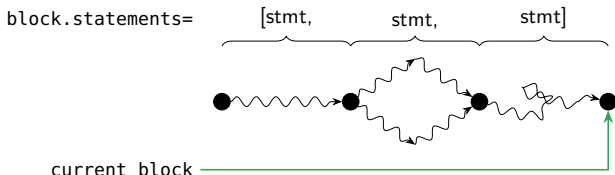
- Unterfunktionen hängen Operationen *implizit* an den `current_block`.
- Invariante: Nach jedem Statement kommt der Kontrollfluss wieder zusammen.



- **Sequenzierung:** Sequenzielle Codegenerierung für alle Statements

```
def visit_Block(self, block):  
    for stmt in block.statements:  
        self.visit(stmt)
```

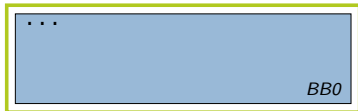
- Unterfunktionen hängen Operationen *implizit* an den `current_block`.
- Invariante: Nach jedem Statement kommt der Kontrollfluss wieder zusammen.



- Die Statements etablieren die komplexeren Kontrollflüsse.
 - Anlegen neuer Basisblöcke und Verbindung über `Goto` und `IfGoto`
 - **Wichtig:** Ein `visit()`-Aufruf kann den `current_block`-Zeiger umbiegen



Kontrollfluss: Selektion



current_block



```
...  
cond := Assign ...
```

BB0

Ablauf der Codeerzeugung:

- Bedingung generieren
`self.rvalue(ifStmt.cond)`



```
...  
cond := Assign ...
```

BB0

BB1

BB2

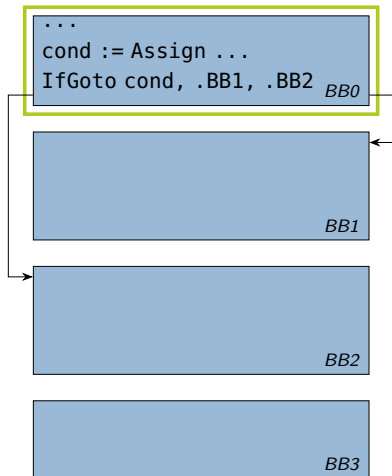
BB3

Ablauf der Codeerzeugung:

- ▶ Bedingung generieren
`self.rvalue(ifStmt.cond)`
- ▶ Blöcke für Then-/Else-Teil, sowie den Sequenzierungsblock erstellen



Kontrollfluss: Selektion

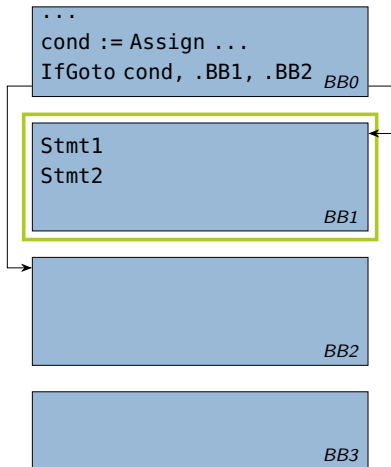


Ablauf der Codeerzeugung:

- ▶ Bedingung generieren
`self.rvalue(ifStmt.cond)`
- ▶ Blöcke für Then-/Else-Teil, sowie den Sequenzierungsblock erstellen
- ▶ Bedingte Kontrollflussverzeugung



Kontrollfluss: Selektion

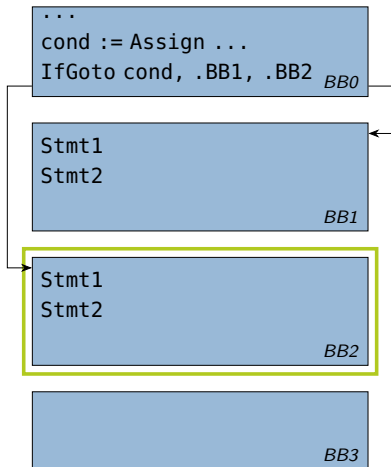


Ablauf der Codeerzeugung:

- ▶ Bedingung generieren
`self.rvalue(ifStmt.cond)`
- ▶ Blöcke für Then-/Else-Teil, sowie den Sequenzierungsblock erstellen
- ▶ Bedingte Kontrollflussverzeigung
- ▶ Then-Block generieren
`self.visit(ifStmt.then_block)`

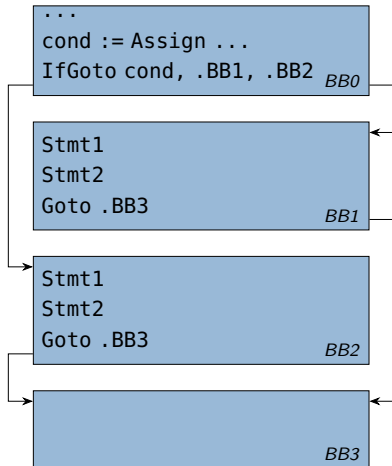


Kontrollfluss: Selektion



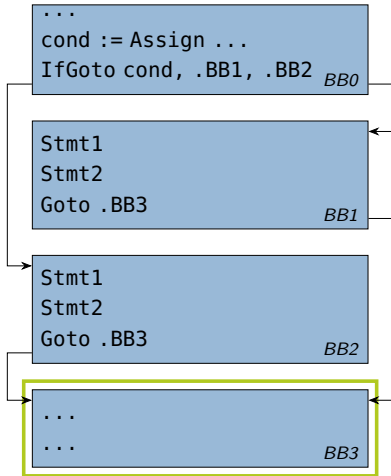
Ablauf der Codeerzeugung:

- ▶ Bedingung generieren
`self.rvalue(ifStmt.cond)`
- ▶ Blöcke für Then-/Else-Teil, sowie den Sequenzierungsblock erstellen
- ▶ Bedingte Kontrollflussverzeugung
- ▶ Then-Block generieren
`self.visit(ifStmt.then_block)`
- ▶ Else-Block generieren
`self.visit(ifStmt.else_block)`



Ablauf der Codeerzeugung:

- ▶ Bedingung generieren
`self.rvalue(ifStmt.cond)`
- ▶ Blöcke für Then-/Else-Teil, sowie den Sequenzierungsblock erstellen
- ▶ Bedingte Kontrollflussverzeigung
- ▶ Then-Block generieren
`self.visit(ifStmt.then_block)`
- ▶ Else-Block generieren
`self.visit(ifStmt.else_block)`
- ▶ Kontrollfluss wieder zusammenführen

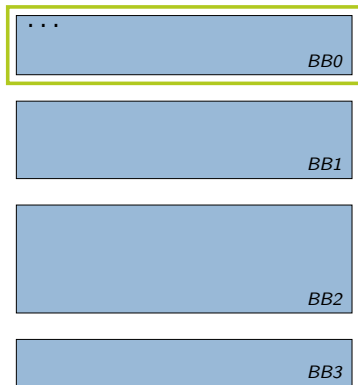


Ablauf der Codeerzeugung:

- ▶ Bedingung generieren
`self.rvalue(ifStmt.cond)`
- ▶ Blöcke für Then-/Else-Teil, sowie den Sequenzierungsblock erstellen
- ▶ Bedingte Kontrollflussverzeugung
- ▶ Then-Block generieren
`self.visit(ifStmt.then_block)`
- ▶ Else-Block generieren
`self.visit(ifStmt.else_block)`
- ▶ Kontrollfluss wieder zusammenführen
- ▶ `current_block`-Invariante wiederherstellen

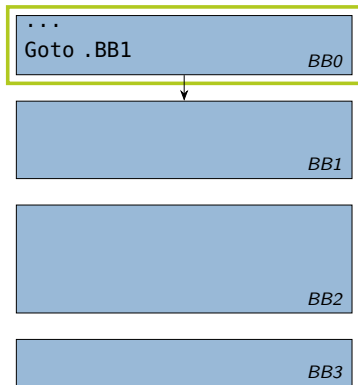


Codeerzeugung (while-Schleife):



Codeerzeugung (while-Schleife):

- ▶ Blöcke erstellen
 - ▶ BB1: Loop-Header Block
 - ▶ BB2: Loop-Body Block
 - ▶ BB3: Sequenzierungsblock

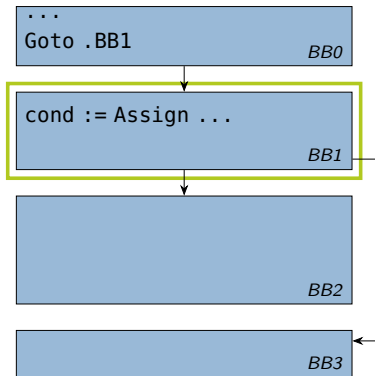


Codeerzeugung (while-Schleife):

- ▶ Blöcke erstellen
 - ▶ BB1: Loop-Header Block
 - ▶ BB2: Loop-Body Block
 - ▶ BB3: Sequenzierungsblock
- ▶ Eintritt in die Schleife



Kontrollfluss: Iteration

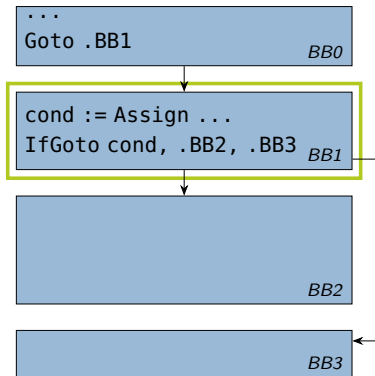


Codeerzeugung (while-Schleife):

- ▶ Blöcke erstellen
 - ▶ BB1: Loop-Header Block
 - ▶ BB2: Loop-Body Block
 - ▶ BB3: Sequenzierungsblock
- ▶ Eintritt in die Schleife
- ▶ Bedingung generieren
`self.rvalue(whileStmt.cond)`



Kontrollfluss: Iteration

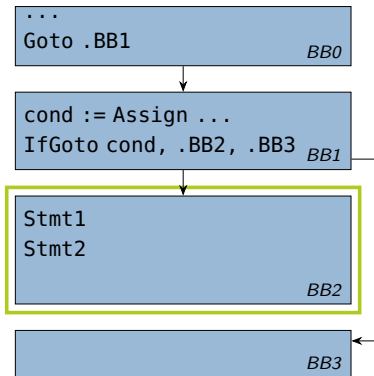


Codeerzeugung (while-Schleife):

- ▶ Blöcke erstellen
 - ▶ BB1: Loop-Header Block
 - ▶ BB2: Loop-Body Block
 - ▶ BB3: Sequenzierungsblock
- ▶ Eintritt in die Schleife
- ▶ Bedingung generieren
`self.rvalue(whileStmt.cond)`
- ▶ Bedingte Kontrollflussverzweigung



Kontrollfluss: Iteration

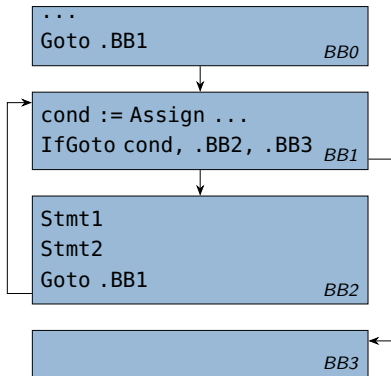


Codeerzeugung (while-Schleife):

- ▶ Blöcke erstellen
 - ▶ BB1: Loop-Header Block
 - ▶ BB2: Loop-Body Block
 - ▶ BB3: Sequenzierungsblock
- ▶ Eintritt in die Schleife
- ▶ Bedingung generieren
`self.rvalue(whileStmt.cond)`
- ▶ Bedingte Kontrollflussverzweigung
- ▶ Loop-Body generieren
`self.visit(whileStmt.body)`



Kontrollfluss: Iteration

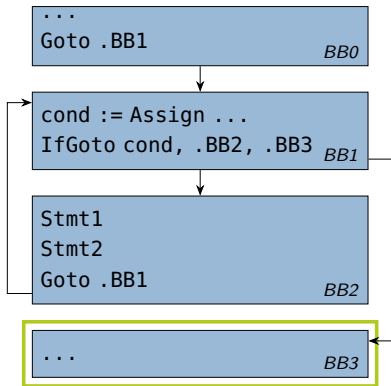


Codeerzeugung (while-Schleife):

- ▶ Blöcke erstellen
 - ▶ BB1: Loop-Header Block
 - ▶ BB2: Loop-Body Block
 - ▶ BB3: Sequenzierungsblock
- ▶ Eintritt in die Schleife
- ▶ Bedingung generieren
`self.rvalue(whileStmt.cond)`
- ▶ Bedingte Kontrollflussverzweigung
- ▶ Loop-Body generieren
`self.visit(whileStmt.body)`
- ▶ Rücksprungkante



Kontrollfluss: Iteration



Codeerzeugung (while-Schleife):

- ▶ Blöcke erstellen
 - ▶ BB1: Loop-Header Block
 - ▶ BB2: Loop-Body Block
 - ▶ BB3: Sequenzierungsblock
- ▶ Eintritt in die Schleife
- ▶ Bedingung generieren
`self.rvalue(whileStmt.cond)`
- ▶ Bedingte Kontrollflussverzweigung
- ▶ Loop-Body generieren
`self.visit(whileStmt.body)`
- ▶ Rücksprungkante
- ▶ `current_block`-Invariante wiederherstellen

- Die gezeigte Codeerzeugung generiert **ineffizienten Code**.
 - Fehlender Else-Teil führt zu einem leeren Else-Block und doppeltem **Goto**
 - Bottom-Up Auswertung führt zu unnötigem **Assign** bei der Zuweisung
 - Keine Wiederverwendung von temporäre Variablen
 - **Das ist Absicht!** Es soll die Codeerzeugung vereinfachen.
 - Optimierte Codeerzeugung braucht viele Sonderfälle
 - Komplexität erhöht die Fehleranfälligkeit
 - Wir verlagern die Arbeit in die Optimierungsphase.
 - **Copy-Propagation** wird unnötige Zuweisungen tilgen
 - **Constant-Folding** ersetzt Auswertungen mit konstanten Operanden
 - **Dead-Code Elimination**: Sprünge auf Sprünge auflösen, tote Blöcke löschen
- ⇒ Weiterer Vorteil durch diese **Trennung der Belange**:
Optimierungen werden ebenfalls auf den Code des Benutzers angewendet

Eine valide Entscheidung für Übersetzer, aber nicht für Interpreter.

Intentionale Unvollständigkeit der Vorlesung

Der **Grundstock der Codeerzeugung** ist gelegt und kann beliebig erweitert werden. Um Ihnen einen Einblick zu geben, ohne Sie mit endlosen Erzeugungsregeln zu langweilen, werfe ich nur ein **Schlaglicht auf Records**.

- Records haben mehrere benannte und typisierte Felder.
 - Ein Record-Objekt kann größer sein als ein einzelnes Maschinenwort.
 - **Aber:** Unsere IR-Maschine kann nur 32-Bit Zahlen speichern und verarbeiten.
- Wir müssen eine Übersetzung des Sprachkonstrukts vornehmen:
 - Die Record-Objekte müssen in den Speicher gelegt werden
 - Berechnung der relativen und absoluten Feldadressen
 - Zugriffe erfolgen über **Load/Store**



- **Erinnerung (Record):** Geordnete Sequenz von Paaren (Name, Typ)

```
type(struct foo) = = record(("a", int), ("b", foo_t), ("c", int))
```

- Um dessen Objekte im Speicher abzulegen, brauchen wir ein **Datenlayout**

```
layout(struct foo) =
```

- Übersetzer leitet das Layout ab

Name	Typ
a	int
b	foo_t
c	int



- **Erinnerung (Record):** Geordnete Sequenz von Paaren (Name, Typ)

```
type(struct foo) = = record(("a", int), ("b", foo_t), ("c", int))
```

- Um dessen Objekte im Speicher abzulegen, brauchen wir ein **Datenlayout**

```
layout(struct foo) =
```

Name	Typ	Länge
a	int	4
b	foo_t	5
c	int	4

- Übersetzer leitet das Layout ab
 - Jeder enthaltende Typ belegt eine gewisse **Anzahl an Bytes** im Speicher.



- **Erinnerung (Record):** Geordnete Sequenz von Paaren (Name, Typ)

```
type(struct foo) = = record(("a", int), ("b", foo_t), ("c", int))
```

- Um dessen Objekte im Speicher abzulegen, brauchen wir ein **Datenlayout**

```
layout(struct foo) =
```

Name	Offset	Typ	Länge
a	0	int	4
b	4	foo_t	5
c		int	4

- Übersetzer leitet das Layout ab
 - Jeder enthaltene Typ belegt eine gewisse **Anzahl an Bytes** im Speicher.
 - Berechnung eines **Feld-Offsets**



- **Erinnerung (Record):** Geordnete Sequenz von Paaren (Name, Typ)

```
type(struct foo) = = record(("a", int), ("b", foo_t), ("c", int))
```

- Um dessen Objekte im Speicher abzulegen, brauchen wir ein **Datenlayout**

```
layout(struct foo) =
```

Name	Offset	Typ	Länge
a	0	int	4
b	4	foo_t	5
-	9		3
c	12	int	4

- Übersetzer leitet das Layout ab
 - Jeder enthaltende Typ belegt eine gewisse **Anzahl an Bytes** im Speicher.
 - Berechnung eines **Feld-Offsets**
 - Zusätzliche **Alignment**-Bedingungen durch das Einfügen von Lücken

- **Erinnerung (Record):** Geordnete Sequenz von Paaren (Name, Typ)

```
type(struct foo) = = record(("a", int), ("b", foo_t), ("c", int))
```

- Um dessen Objekte im Speicher abzulegen, brauchen wir ein **Datenlayout**

```
layout(struct foo) =
```

Name	Offset	Typ	Länge
a	0	int	4
b	4	foo_t	5
-	9		3
c	12	int	4

```
sizeof(struct foo) = 16
```

- Übersetzer leitet das Layout ab
 - Jeder enthaltende Typ belegt eine gewisse **Anzahl an Bytes** im Speicher.
 - Berechnung eines **Feld-Offsets**
 - Zusätzliche **Alignment**-Bedingungen durch das Einfügen von Lücken
 - Es ergibt sich die Größe **dieses Typs**.



Record-Typen: Datenlayout

- **Erinnerung (Record):** Geordnete Sequenz von Paaren (Name, Typ)

```
type(struct foo) = = record(("a", int), ("b", foo_t), ("c", int))
```

- Um dessen Objekte im Speicher abzulegen, brauchen wir ein **Datenlayout**

```
layout(struct foo) =
```

Name	Offset	Typ	Länge
TAG	0	type_t	4
a	4	int	4
b	8	foo_t	5
-	13		3
c	16	int	4

```
sizeof(struct foo) = 20
```

- Übersetzer leitet das Layout ab
 - Jeder enthaltende Typ belegt eine gewisse **Anzahl an Bytes** im Speicher.
 - Berechnung eines **Feld-Offsets**
 - Zusätzliche **Alignment**-Bedingungen durch das Einfügen von Lücken
 - Es ergibt sich die Größe **dieses Typs**.
 - Mit dynamische Typen: Tag-Feld

- **Erinnerung (Record):** Geordnete Sequenz von Paaren (Name, Typ)

```
type(struct foo) = = record(("a", int), ("b", foo_t), ("c", int))
```

- Um dessen Objekte im Speicher abzulegen, brauchen wir ein **Datenlayout**

```
layout(struct foo) =
```

Name	Offset	Typ	Länge
TAG	0	type_t	4
a	4	int	4
b	8	foo_t	5
-	13		3
c	16	int	4

```
sizeof(struct foo) = 20
```

- Übersetzer leitet das Layout ab
 - Jeder enthaltene Typ belegt eine gewisse **Anzahl an Bytes** im Speicher.
 - Berechnung eines **Feld-Offsets**
 - Zusätzliche **Alignment**-Bedingungen durch das Einfügen von Lücken
 - Es ergibt sich die Größe **dieses Typs**.
 - Mit dynamische Typen: Tag-Feld
- Verwendung des Layout-Deskriptors
 - Codeerzeugung
 - Debug-Informationen
 - Dynamische Introspektion

- Codeerzeugung muss an mehreren Stellen angepasst werden
 - Variablendeklarationen, Feldzugriffe, Referenzerzeugung
 - Parameterübergabe, Rückgabewerte
 - Wir streben (beispielhaft) ein Wertemodell für Variablen an
- Definition einer lokalen Variable mit Record-Typ
 - Unsere IR-Maschine kann nur 32-Bit Variablen $\Rightarrow$ IR-Variable hält Pointer
 - Allokation am Stack mittels `StackAlloc` und `sizeof(Type)`

```
def visit_VarDecl(self, decl):  
    bb = self.current_block  
    cf = self.current_function  
  
    # IR-Variable  
    var = cf.create_variable(decl.name)  
    decl.ir_obj = var  
    # ggf. Objekt am Stack anlegen  
    size = self.sizeof(decl.Type)  
    if size > 4:  
        bb.append(StackAlloc, var, size)  
        if self.dynamic_types:  
            bb.append(Store, var,  
                    decl.Type.tag)
```

```
var obj : struct foo;
```



```
obj := StackAlloc 20  
*obj := Store 12345 // Typ-Tag
```

Alternative: Größere Objekte auf den Heap (`HeapAlloc`, `HeapFree`)

- Für die Zuweisung brauchen wir **R- und L-Wert** des Record-Objekts.
 - **Problem:** Nur Referenz, nicht aber Objekt, passt nicht in eine IR-Variable
 - Unsere einfache IR-Sprache erfordert komplexere Codeerzeugung
- ⇒ Wir verwenden die Referenz sowohl als R- als auch als L- Wert

```
def lvalue_Identifier(self, expr):  
    ...  
    if isinstance(expr.Type, RecordType):  
        return expr.decl.ir_obj
```

```
def rvalue_Identifier(self, expr):  
    ...  
    if isinstance(expr.Type, RecordType):  
        return expr.decl.ir_obj
```

- **Zuweisungs-Operation** für größere Objekte (> 4 Bytes)
 - Am Zuweisungsknoten kommen 2 Zeiger für Quelle, als auch für Ziel an
 - Verwendung der `memcpy()`-Hilfsfunktion mit konstanter Größe

```
var x : struct foo;  
var y : struct foo;  
x := y;
```

⇒

```
x := StackAlloc 16  
y := StackAlloc 16  
t1 := Call memcpy, x, y, 16
```

Erinnerung: Wir wollen Wertemodell für Variablen in der Sprache.

- Wir müssen Record-Objekte in Funktionen und wieder herausbekommen.
 - **Problem:** Wir können nur 32-Bit Wörter übergeben.
 - **Call-by-Value:** Der Aufgerufene soll eine Kopie des Objekts bekommen.

```
func f(obj : struct foo) : struct foo {  
    obj.c := 33;  
    return obj.c;  
}
```

- **Lösung:** Wir führen ein **Aufrufkonvention** für größere Objekte ein.
 - Für **Parameter:** Der Aufrufende gibt Zeiger auf Originalobjekt mit.
Der Aufgerufene legt sich eine Kopie als lokales Objekt an.
 - Für **Rückgabe:** Aufrufer gibt Zieladresse als zusätzlichen Parameter mit
Ausschleusen des Rückgabewerts durch `memcpy()`

Aufrufstelle

```
var src : struct foo;  
var dst : struct foo;  
dst := f(src);
```



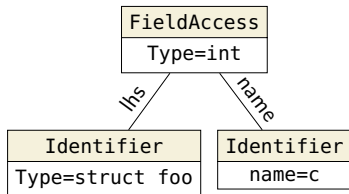
```
src := StackAlloc 16  
dst := StackAlloc 16  
t0  := Call f, dst, src
```

Funktion

```
func f(obj : struct foo) : struct foo {  
    return obj;  
}
```



```
<Function: f> {  
    parameters = [__ret, obj]  
    variables = [__obj]  
    <BB: BB0> {  
        // Prologue  
        __obj := StackAlloc 16  
        t0  := Call memcpy, __obj, obj, 16  
        // Function Return  
        t1  := Call memcpy, __ret, __obj, 16  
        Return __ret  
    }  
}
```



L-Wert

`obj.c := 100;`



`t0 := Add obj, 16
*t0 := Store 100`

R-Wert

`y := obj.c;`



`t0 := Add obj, 16
t1 := Load *t0
y := Assign t1`

■ Ableitung der Feld-Adresse aus Objekt-Adresse und Feld-Offset

L-Wert

- L-Wert von `lhs` ist Basisadresse
- Offset **statisch** aus Layout bestimmen
- Feldadresse ist Basis+Offset

R-Wert

- Dereferenzierung des L-Werts
- Re-Use: `lvalue_FieldAccess()`

```

def lvalue_FieldAccess(self, expr):
    # Basisadresse des Objekts
    lhs_ptr = self.lvalue(expr.lhs)
    # Konstanter Offset des Felds
    layout = expr.Type.layout
    offset = layout[expr.name].offset
    # Adressberechnung
    ret = self.current_function\
        .create_variable\
        self.current_block\
        .append(Add, ret, lhs_ptr, offset)
    return ret
  
```



- Records und Wertemodell haben nicht gut in unser IR-Modell gepasst.
 - Variablen sind nur 32-Bit
 - Abbildung des Wertemodell auf Referenzen

- Codeerzeugung verhält sich abhängig vom Typ völlig anders
 - Zuweisungsoperation bekommt Referenz auf Record-Objekte als R-Wert
 - Besondere Aufrufkonvention für Record-Objekte
 - Zeiger auf Record-Objekte sind kein Problem (`&struct foo`)

- Notwendigkeit einer Bibliothek mit Laufzeitfunktionen
 - Übersetzer verlässt sich auf die Existenz von Bibliotheksfunktionen (`memcpy()`)
 - Erweiterung des IR-Codes durch Funktionen (`call`: Komplexbefehl)



- Zwischencode für eine **sequentielle virtuelle Maschine**
 - Stack-Maschine oder Register-Maschine mit unendlich vielen Registern
 - 3-Adress-Code bzw. Quadrupel-Notation
- **CFG-Datenstruktur** nimmt Code während der Erzeugung auf
 - Erzeugen neuer Funktionen, Basisblöcke und lokaler Variablen
 - Anhängen einer Operation an den aktuellen Basisblock
- Unterschiedliche Codeerzeugung für L-/R-Werte
 - **L-Wert**: Visitor erzeugt eine temporäre Variable, die die Referenz enthält
 - **R-Wert**: Zurückgegebene Variable enthält berechneten Wert
- **Kontrollstrukturen** (**if**, **while**) werden linearisiert
 - Erzeugen neuer Basisblöcke und Abbildung des Kontrollflusses auf (**If**)**Goto**
 - Trennung der Belange zwischen Codeerzeugung und Optimierer
- Fallbeispiel: Records als **komplexeres Sprachkonstrukt**