Compilerbau CS1019

Th. Letschert

TH Mittelhessen Gießen

University of Applied Sciences

Prozeduren und Laufzeit-Organisation

- Prozeduren, Funktionen, Methoden
- Laufzeit-Organisation: Konstanten, Variablen, Speicherklassen
- Laufzeit-Organisation: Prozeduren
- Beispiel: Minisprache mit Prozeduren

Prozeduren, Funktionen, Methoden

Prozeduren, Funktionen, Methoden

Funktionen / Prozeduren / Methoden

- Eine **Funktion** ist ein parametrisierter Codeblock, der einen Wert liefert
- Eine **reine** Funktion / Funktion **ohne Seiteneffekt** ist eine Funktion, die
  - nur einen Wert liefert und
  - ansonsten keine Effekte hat – also beispielsweise keine nicht-lokalen Variablen verändert
- Eine **Prozedur** ist ein parametrisierter Codeblock, der keinen Wert liefert
  - Prozeduren ohne externe Effekte sind sinnlos
- Eine **Methode** ist ein parametrisierter Codeblock, der im Kontext eines Objektes ausgeführt wird. (Das Objekt, auf das *this* zeigt.)

Parameter und Argumente

- Ein **Parameter** ist ein Platzhalter für wechselnde **Argumente**
- Beim **Funktionsaufruf** werden die Parameter durch die **Argumente** ersetzt und der Codeblock ausgeführt

Funktionen höherer Ordnung

- Eine Funktion / Prozedur ist von höherer Ordnung, wenn sie Funktionen / Prozeduren als Argument und / oder als Ergebnis hat

8.4. Method Declarations

Aus „Java Language Specification“
https://docs.oracle.com/javase/specs/jls/se8/html/jls-8.html#jls-8.4

A *method* declares executable code that can be invoked, passing a fixed number of values as arguments.

Funktionsdefinition / Funktionsausdruck

Funktion durch Funktionsdefinition erzeugen

In einfachen Sprachen können Funktionen nur durch Funktions-Definitionen im Programm eingeführt werden.

Funktionsausdruck: Funktion durch Ausdrucksauswertung mit Funktionswert

In neueren Sprachen ist es (nach ca 25 Jahren wieder) üblich (geworden), dass Funktionen nicht nur durch Definitionen eingeführt werden können:

Es gibt auch Ausdrücke deren Wert eine Funktion ist

Ausdrücke deren Wert eine Funktion ist, werden oft *Closure* oder *Lambda-Ausdrücke* oder *anonyme Funktionen* genannt

```
static Integer fun(Integer x, Integer y, BiFunction<Integer, Integer, Integer> f) {
      return f.apply(x,y);
}

public static void main(String[] args) {
      System.out.println(
            fun(1, 2,
                (x,y) -> x+y
            )
      );
}
```

15.27. Lambda Expressions

A lambda expression is like a method: it provides a list of formal parameters and a body - an expression or block - expressed in terms of those parameters.

LambdaExpression:
LambdaParameters -> *LambdaBody*

Aus „Java Language Specification“
https://docs.oracle.com/javase/specs/jls/se8/html/jls-15.html#jls-15.27

Parameterübergabe

Wertübergabe / *call by value*

Die Argument-Ausdrücke werden ausgewertet und dann die berechneten Werte an die Funktion / Prozedur übergeben.

Übliche effiziente Form der Parameterübergabe

Referenzübergabe / *call by reference*

Die Argument-Ausdrücke werden zu Adressen ausgewertet und dann die berechneten Adressen an die Funktion / Prozedur übergeben.

Funktioniert nur, wenn die Argumentausdrücke sich zu Adressen auswerten lassen (z.B. Variablen, Array-Positionen)

Call by need

Argument-Ausdrücke werden ausgewertet, wenn in der Funktion / Prozedur zum ersten Mal auf sie zugegriffen wird.

Nicht genutzte Argumente werden nicht ausgewertet, genutzte Argumente werden einmal ausgewertet

Namensübergabe / *call by name*

Argument-Ausdrücke werden bei jedem Zugriff in der Funktion / Prozedur aufs Neue ausgewertet.

Sinn: Änderungen der Funktion von Variablen in den Ausdrücken werden wirksam

Parameterübergabe

Namensübergabe Beispiel:

```
object CallByName_Main extends App {
  var x = 5
  def sum(exp: =>Int, from: Int, to: Int) : Int = {
    var s = 0
    var i = from
    while (i < to) {
      s = s + exp
      i = i+1
      x = x + 1
    }
    s
  }
  println( sum( (x*x + 2*x+3), x, x+3) )

  println(5*5 + 2*5 +3 +
          6*6 + 2*6 +3 +
          7*7 + 2*7 +3)
}
```

Der Wert von exp *hängt vom Wert von* x *ab und ändert sich darum in der Schleife.*

```
155
155
```

s *im ersten, zweiten und dritten Schleifendurchlauf in sum.*

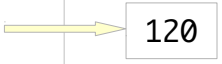
Parameterübergabe

Referenzübergabe Beispiel (C++):

```
#include <iostream>
using namespace std;

void f(int x, int &r) {
     if (x == 0){
          r = 1;
     } else {
          f(x-1, r);
          r = r * x;
     }
}

int main() {
     int res;
     f(5, res);
     cout << res << endl;
     return 0;
}
```



Fakultätsprozedur:
Mit Referenzparametern können Funktionen durch Prozeduren emuliert werden.

Übersetzung von Prozeduren, Funktionen, Methoden

Thema 1: Statische Namensauflösung und Gültigkeitsbereiche

Prozeduren erlauben üblicherweise die Definition von Parametern und lokalen Variablen

Eine Prozedur ist darum ein auch ein Gültigkeitsbereich.

Bei der Übersetzung einer Sprache mit Prozeduren muss der Compiler mit Gültigkeitsbereichen und Namensbindungen umgehen können.

Problem: die Namensbindung kann i.A. nicht komplett zur Compilezeit behandelt werden, da der Compiler nicht vorhersehen kann wo der Speicherplatz, der zu einem Namen gehört, sich zur Laufzeit befindet.

Thema 2: Laufzeitorganisation und dynamische Adressberechnungen

Prozeduren (Funktionen / Methoden) werden zur Laufzeit aktiviert und wieder verlassen.

Dazu muss Code generiert werden für:

- **eventuell notwendige Restarbeiten der Namensbindung**
- **die Speicherbeschaffung für die aufgerufene Funktion**
- **die Parameterübergabe**
- **den Sprung zur Funktion**
- **die Rückkehr zum Aufrufer**

sowie

- **die Berechnung der Adressen von Variablen / Parametern und Konstanten**

Prozeduren und Namensbindungen

Prozeduren ~ Gültigkeitsbereiche

Prozeduren erlauben üblicherweise die Definition von Parametern und lokalen Variablen

Eine Prozedur ist darum ein auch ein Gültigkeitsbereich – und wird hier als Repräsentant eines Gültigkeitsbereichs behandelt.

Andere Gültigkeitsbereiche: Blöcke

Arten von Prozeduren in Bezug auf Gültigkeitsbereiche:

Verschachtelt oder nicht verschachtelt

Ist es erlaubt eine Prozedur in einer anderen zu definieren?

Problem verschachtelter Prozeduren: Welches ist die gültige Definition eines Namens?

Rekursiv oder nicht rekursiv

Darf eine Prozedur sich selbst (direkt oder indirekt) aufrufen

Problem rekursiver Prozeduren: Eine Name kann gleichzeitig mehrere gültige Definitionen haben: Welche der gültigen Definitionen ist die aktuell gemeinte?

Prozeduren und Gültigkeitsbereiche

Flache Prozeduren / Funktionen

In Sprachen mit einem „flachen“ Prozedur-Konzept (flachen Gültigkeitsbereichen) darf eine Prozedur (ein Gültigkeitsbereich) nur auf der globalen Ebene definiert werden.

Beispiel: C ist eine Sprache mit flachem Prozedur-Konzept

Das Management der Umgebungen wird dadurch besonders einfach:

Ein Name ist entweder

- **global, oder**
- **lokal in der Prozedur definiert, die ihn verwendet.**

Die Suche nach der richtigen Definition eines Namens ist dadurch einfach:

Er ist

- **entweder in der Prozedur definiert in der er verwendet wird**
- **oder er ist global definiert**
- **oder er ist undefiniert**

Prozeduren und Gültigkeitsbereiche

Verschachtelte Prozeduren / Funktionen

In Sprachen mit einem „verschachtelten“ Prozedur-Konzept darf eine Prozedur andere Prozedur-Definitionen enthalten.

Java ist eine Sprache mit geschachteltem Prozedur-Konzept:
In Methoden und Klasse dürfen andere Klassen (und damit Methoden) definiert werden.

Scala und JavaScript weitere Beispiele für Sprachen mit verschachtelten Prozeduren

```
function f() {
  var x = 40;
  return function() {
     return x+2;
  }
}
```

Beispiel: Verschachtelung in JavaScript

Rekursive Prozeduren

Die Identifikation der Namen

während der Übersetzung in einer solchen Sprache ist noch etwas anspruchsvoller:

Wie bei Prozeduren ohne Rekursion

- **legt der Compiler für jede Prozedur (jeden Gültigkeitsbereich) eine Tabelle an**
- **und kann alle Namen über diese Tabellen identifizieren**

Bei der Identifikation der Namen sind allerdings komplexere Regeln der Sichtbarkeit zu beachten

- **ein Name kann mehrfach definiert sein**
- **Erst zur Laufzeit ist feststellbar, welche Definition die gemeinte ist**

Beispiel:

```c
#include <stdio.h>
#include <stdlib.h>

int f(int x) {
  if (x==0) return 1;
  else return f(x-1)*x;
}

int main(void) {
  printf("%d\n", f(10));
}
```

x ist einmal definiert. Die Definition ist leicht zu finden und die Identifikation kann zur Compilezeit stattfinden.

Da f rekursiv ist, können zur Laufzeit beliebig viele Instanzen von f und damit von x existieren.

Welche davon aktuell gemeint ist, kann erst zur Laufzeit bestimmt werden.

Übersetzung von Funktionen und Prozeduren: Themenstellung

Aktionen zur Laufzeit

- **Ausführung des Funktions- / Prozedur-Körpers**
- **Aufruf mit Parameterübergabe**
- **Rückkehr zur Aufrufstelle**

Problemstellungen /Entscheidungen

- **Daten: Speicherort und Zugriff**
  - auf lokale Variablen und Parameter
  - auf globale Variablen und Parameter
- **Rückkehr-Adresse: Speicherort und Zugriff**
- **Mechanismus der Parameterübergabe**
- **Mechanismus der Rückgabe des Ergebnisses**
- **Format der Datenstruktur die eine Funktion / Prozedur zur Laufzeit repräsentiert**

Prozeduren, Funktionen, Methoden

Identifikation

Identifikation von Namen zur Übersetzungszeit

- ordnet Anwendungen von Namen ihre Definition zu
- verwendet Datenstrukturen im Compiler: Symboltabellen / Umgebungen

Identifikation von Adressen (die zu Namen gehören) zur Laufzeit

- ordnet Anwendungen von Namen deren aktuelle Adresse zu
- verwendet Datenstrukturen zur Laufzeit

Variablen, Prozeduren etc. müssen eine Laufzeit-Existenz haben:

- eine Adresse mit
- einen Speicherbereich mit Daten / Code an dieser Adresse

Die Adresse der Laufzeit-Daten

muss im generierten Code zur Verfügung stehen

entweder

- als Konstante, oder
- als Codesequenz die die Adresse berechnet

Laufzeit-Organisation

Speicherorganisation / Laufzeitorganisation

Problemstellung

- Der Compiler tut was er kann, aber vieles kann erst zur Laufzeit getan werden.
  Dazu Statt zu tun, muss der Compiler dann Code erzeugen, der tun wird.
- Im Quellprogramm werden Konstante, Variablen, Prozeduren definiert
  Diese müssen zur Laufzeit
  - existieren und
  - effizient über ihre Adresse (statt ihren Namen) angesprochen werden
- Manche Adressen
  - können vom Compiler bestimmt werden,
  - andere sind erst zur Laufzeit bekannt.
- Vor der Codegenerierung wird die Transformation von Namen in Adressen vorbereitet
  - Name ~> Adresse, oder
  - Name ~> Codesequenz, die die Adresse zur Laufzeit berechnet.

  Dazu ist es notwendig zu wissen / entscheiden, wir die Daten zur Laufzeit im Speicher liegen

Speicherorganisation / Laufzeitorganisation

Beispiel

```
object Example {

  var g: Int = 1

  def f(x: Int): Unit =
    if (x == 0) g
    else {
      g = g * x
      f(x - 1)
    }

  def main(args: Array[String]): Unit = {
    f(5)
    println(g)
  }

}
```

f g 1 5 **liegen immer an der gleichen Stelle**

x **hat „wechselnde Positionen“ die erst zur Laufzeit berechnet werden können**

Laufzeitorganisation

Aktionen

- **Konzeption**
  - Speicherlayout festlegen
  - statische und dynamische Berechnungen unterscheiden
- **Compiler-Aktionen (Teil der Kontextanalyse)**

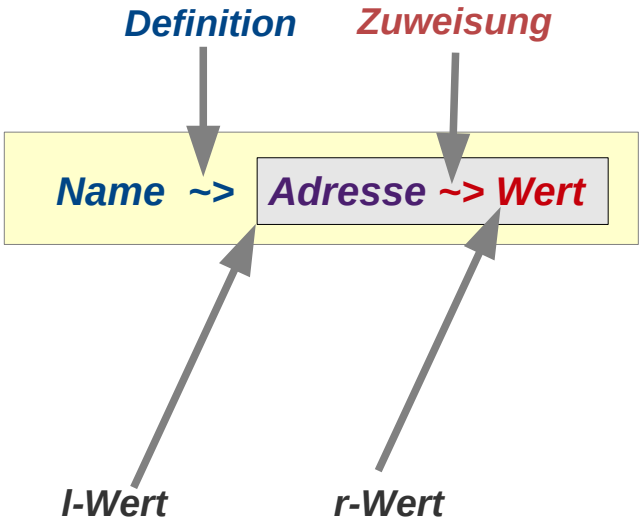
  Vorbereitung der Codegenerierung
  - durch statisches (= zur Compilezeit) Berechnen von Adressen
  - durch Vorbereiten der dynamischen (= zur Laufzeit) Berechnung von Adressen

Variablen

Variablendefinition

Variablen werden mit einer (Variablen-) Definition im Programm eingeführt

Bei der Definition wird dem Namen ein Speicherplatz / eine Adresse zugeordnet



Zuweisung

Bei einer Zuweisung wird der Inhalt des Speicherplatzes modifiziert, der der Variablen zugeordnet ist.

Zugriff

Bei einer Zuweisung an eine Variable wird die Adresse gebraucht.

Taucht die Variable in einem Ausdruck auf, dann interessiert man sich für den Wert an der Adresse.

Man unterscheidet darum:

- l-Wert : die Adresse / linksseitiger Wert der Variablen
- r-Wert : der Wert an der Adresse / rechtsseitiger Wert der Variablen

Variablen

Modellierung von Variablen

Zwei Abbildungen speichern den aktuellen Zustand des (laufenden) Programms

- **Umgebung / *Environment*:**
  Bezeichner ~> Variable (Adresse / Zelle)
- **Speicher: Adresse / Zelle ~> Wert**

Name ~> Adresse

Umgebung: Speichert den Effekt von Definitionen

Sinn der zwei-stufigen Abbildung

- **Eine Variable ist mit der Folge ihrer wechselnden Werte assoziiert**
- **Vorteil für Compiler:**
  - **Speicherverwaltung wird einfacher**
  - **Erzeugter Code ist einfacher und schneller**
- **Vorteil für Programmierer**
  - **Sicherheit: statisches Typkonzept auch bei Variablen möglich: fester Typ trotz wechselnder Werte**
  - **Programme sind verstehbarer**
    Die Variable *t* enthält immer die (wechselnde) Temperatur ...

Adresse ~> Wert

Speicher: Speichert den Effekt von Zuweisungen

Variablen

Speicherklasse

Die Speicherklasse einer Variablen bezieht sich

- den Ort an dem zur Laufzeit ihre Werte gespeichert sind und
- die Lebensdauer der Variablen / des Speicherorts der Variablen

Beispiel Java

In Java unterscheidet man

- **Klassenvariable**
  Speicherort: Klassenobjekt im Heap,
  Lebensdauer: unbegrenzt / bestimmt vom *Garbage-Collector*
- **Objekt-** (oder Instanz-) **Variable**
  Speicherort: Objekt im Heap,
  Lebensdauer: unbegrenzt / bestimmt vom *Garbage-Collector*
- **Lokale Variable**
  Speicherort: Rahmen (Stack-Frame),
  Lebensdauer: entspricht dem des definierenden Gültigkeitsbereichs (z.B. Methode)
- **Parameter**
  Speicherort: Rahmen (Stack-Frame),
  Lebensdauer: entspricht dem der definierenden Methode-Instanz

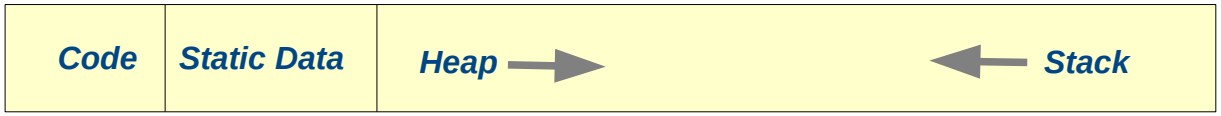
siehe bei Bedarf: https://homepages.thm.de/~hg51/Veranstaltungen/OOP_WS13_14/Folien/oop-10.pdf

Variablen

Speicherklasse / Beispiel C

In C unterscheidet man

- **Globale Variablen**
  Speicherort: Statischer Datenbereich,
  Lebensdauer: unbegrenzt
- **Lokale Variablen**
  Speicherort:
  - normalerweise
    Speicherort: Rahmen (Stack-Frame),
    Lebensdauer: entspricht dem des definierenden Gültigkeitsbereichs (z.B. Methode)
  - falls mit static deklariert
    Speicherort: Statischer Datenbereich,
    Lebensdauer: unbegrenzt
- **Parameter**
  Speicherort: Rahmen (Stack-Frame),
  Lebensdauer: entspricht dem der definierenden Funktion



Konstante

Eine Konstante

ist ein Name dem in einer Definition ein fester Wert als Bedeutung zugeordnet wird.

Zwei Varianten von Konstanten:

- **statische Konstante: Ein Name der einen fixen statischen (einmal zur Compilezeit berechneten) Wert bezeichnet**
- **dynamische Konstante: Name der einen fixen dynamischen (einmal zur Laufzeit berechneten) Wert bezeichnet**

Eine Variable

ist ein Name dem in einer Definition ein Speicherplatz (und oft auch ein Typ) als Bedeutung zugeordnet wird.

Der Speicherplatz kann wechselnde Werte haben.

Zuweisungen ändern den Wert

Variable vs Konstante

Nicht alles, was (als Konstante) definierbar ist, kann auch der Wert einer Variablen sein. In Java können Klassen beispielsweise nur als statische Konstanten definiert werden. Variablen, deren Wert eine Klasse ist, sind nicht erlaubt.

In Java können umgekehrt Objekte und primitive Werte nicht als statische Konstanten definiert werden: Int, Bool, … -Werte sind bestenfalls dynamische Konstanten (wenn sie als *final* deklariert werden)

Laufzeitorganisation

Laufzeitorganisation: Welche Datenstrukturen repräsentieren ein Programm zur Laufzeit

Zur Laufzeit benötigte Daten

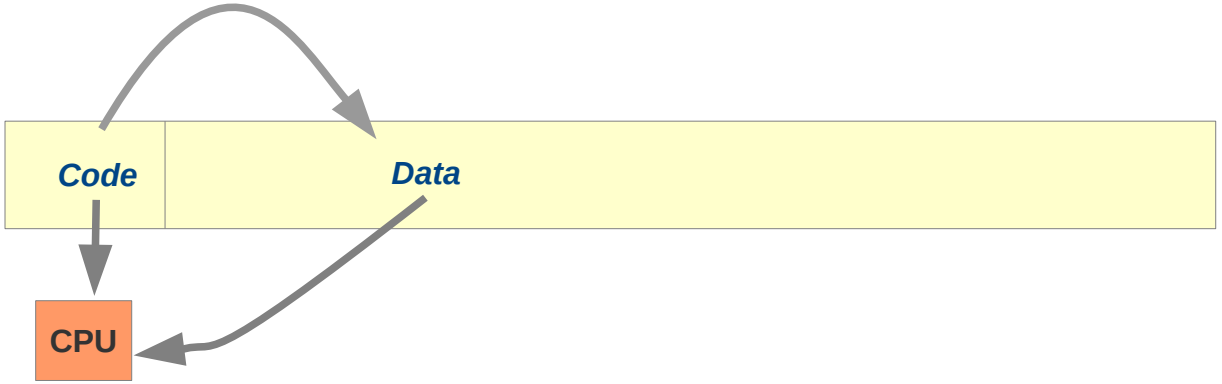
- **Code**

  Der gesamte auszuführende Maschinencode wird zur Laufzeit benötigt,

- **Daten**

  Alle Daten, die der vom Compiler generierte Code (via CPU) erzeugt und / oder verwendet

Für diese Daten muss Speicherplatz im Speicher der Maschine bereit gestellt werden.



Laufzeitorganisation

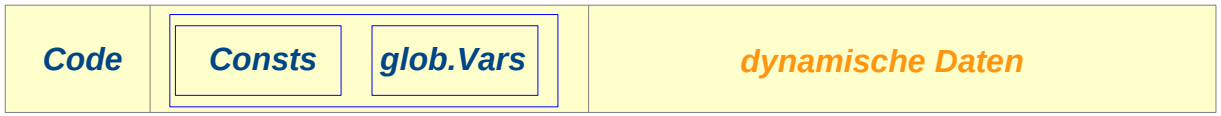
Statische Daten

Daten deren Ort und eventuell auch Wert zur Übersetzungszeit bekannt ist

- **statische Daten I: Konstanten / fixe Werte**
  Daten, die der Compiler zur Übersetzungszeit generieren kann,
  und auf die zur Laufzeit nur lesend zugegriffen wird.
  z.B. String-Literale, oder definierte echte Konstanten
  für diese Daten kann der Compiler
  - Ort (Speicherplatz) und
  - Wert

  festlegen
- **statische Daten II: Variablen mit fixem Speicherplatz**
  Alle Daten, die der vom Compiler generierte Code erzeugt und/oder verwendet,
  deren Speicherplatz aber währende der Laufzeit nicht verändert wird.
  Beispiel globale Variablen
  Für diese Daten kann der Compiler
  - den Ort (Speicherplatz)

  festgelegen,



statische Daten

Laufzeitorganisation

Dynamische Daten

Daten die von generiertem Code manipuliert werden, deren Wert und Ort zur Übersetzungszeit unbekannt ist

dynamische Daten I: Stack

- **Daten die zu Methoden / Funktionen / Prozeduren gehören.**
  Ihr Speicherplatz wird mit dem Aufruf der Methode angefordert und mit deren Ende freigegeben
- **Daten die bei der Ausdrucksauswertung anfallen**
  (vom Compiler angelegte temporäre Variablen)

dynamische Daten II: Heap

Daten die mit Anweisungen des (vom Compiler generierten) Programmcodes erzeugt werden (Speicherallokation / new)

und durch Anweisungen (im Programmcode / Garbage-Collector) frei gegeben werden.

Organisation der Adressvergabe

Die vom Compiler berechnete Adresse wird in den AST eingetragen

Typischerweise als weiterer Bestandteile der Symbole

Beispiel:

```
object ProgSymbols {

  // all entities defined in a program (i.e. names with a compile time value) are symbols
  sealed abstract class ProgSymbol {
    val name: String
  }

  ...

  // Variables
  case class Variable(
     override val name: String,
     var staticType: Option[StaticType] = None, // static type; will be set by Typifier
     var runTimeLoc: Option[RTLocInfo] = None   // statically known aspects of the runtime location;
                                                // will be set by RuntimeLocator
  ) extends ProgSymbol

  ...

}
```

Laufzeit-Organisiation: Prozeduren

Laufzeitorganisation: Prozeduren

Daten-Klassen

Statische Daten

- Konstanten, globale Variablen, in sehr vielen Sprachen auch Prozeduren / Funktionen
  haben einen festen Speicherplatz
- deren Adresse kann statisch (vom Compiler) bestimmt und im Code direkt verwendet werden

Dynamische Daten

Prozedur- / Funktionslokale Daten (call)

- Parameter und lokale Variablen
  können (in Sprachen mit Rekursion) keinen festen Speicherplatz haben (beliebig viele Instanzen der gleichen Prozedur)
- haben eine Adresse die zur Laufzeit berechnet werden muss
- werden dynamisch allokiert (Aufruf) und wieder freigegeben (Rückkehr)

Dynamisch vom Programm allokierte Daten (new)

- Werden im Heap allokiert
- haben eine Lebensdauer, die nicht an die Codestruktur gebunden ist
  meist Explizite Allokation und implizite (*Garbage Collector*) Freigabe

Temporäre Daten

- Daten die zur Laufzeit zur Speicherung von Zwischenwerten benötigt werden

Laufzeitorganisation

Dynamische Daten

Daten die von generiertem Code manipuliert werden, deren Wert und Ort zur Übersetzungszeit unbekannt ist

Dynamische Daten I: Stack

- **Daten die zu Methoden / Funktionen / Prozeduren gehören.**

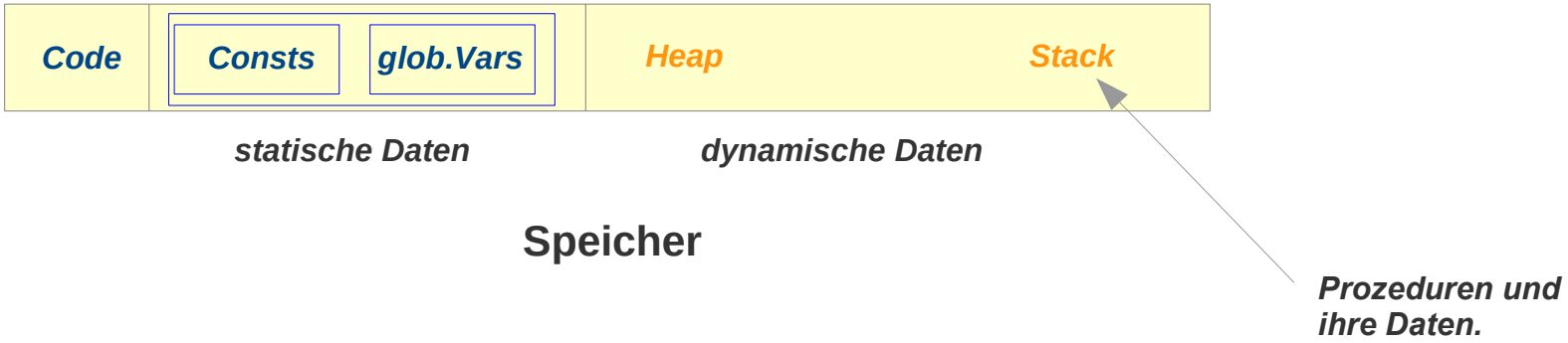
  Ihr Speicherplatz wird mit dem Aufruf der Methode angefordert und mit deren Ende freigegeben
- **Daten die bei der Ausdrucksauswertung anfallen**

  (vom Compiler angelegte temporäre Variablen)

Dynamische Daten II: Heap

Daten die mit Anweisungen des (vom Compiler generierten) Programmcodes erzeugt werden (Speicherallokation / new)

und durch Anweisungen (dispose) oder den Garbage-Collector frei gegeben werden.



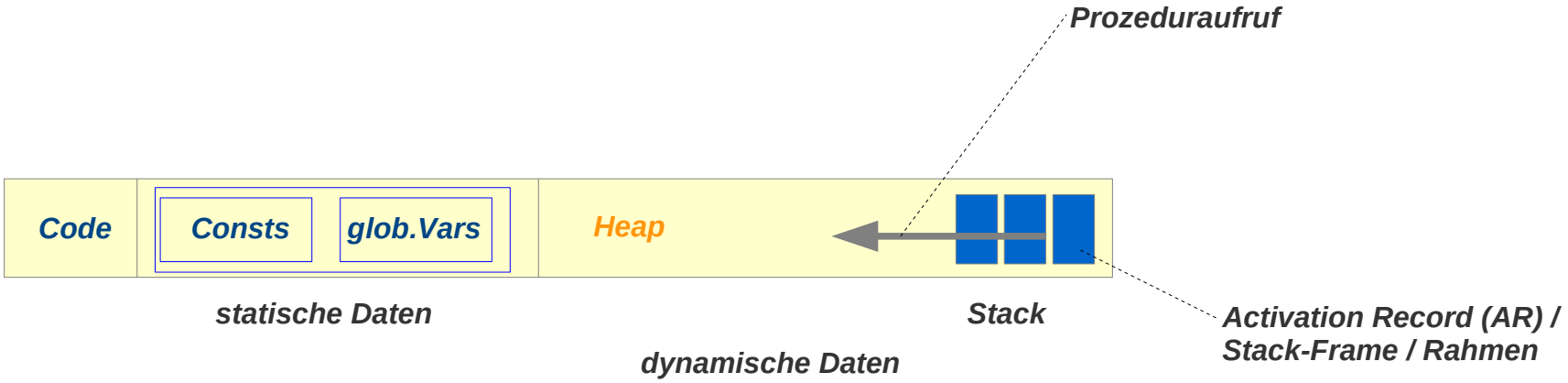
Laufzeitorganisation: Stack und Activation-Records

Stack

Der Stack enthält alle Speicherplätze / Daten die zu Funktionen / Prozeduren gehören
Er ist eingeteilt in:

Activation Record (AR) auch Stack-Frame / Rahmen / Laufzeit-Rahmen / ..
Für jede aktive Prozedur- / Funktions-Instanz existiert ein AR,
der alle Daten enthält die (zur Laufzeit) für diese Instanz benötigt werden:

- **lokale Variablen**
- **Parameter**
- **eventuell temporäre Daten**
- **Verwaltungsinformationen (z.B. Rückkehradresse)**
- **...**

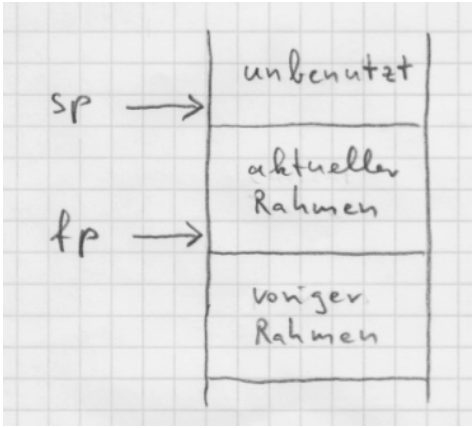


Activation-Record (AR) / Rahmen

Datenstruktur mit den

- **Laufzeit-Informationen einer**
- **aktiven Prozedur (Prozedur-Aufruf in Ausführung, jede Prozedur-Aktivierung)**

Die ARs alle Prozeduren in Ausführung bilden den (Laufzeit-) Stack
(der nicht bei allen Programmiersprachen tatsächlich ein Stapel ist)



Laufzeit-Stack mit Activation-Records (Rahmen) in der Ninja-VM

vergl. Konzepte Systemnaher Programmierung
Prof. Dr. Hellwig Geisse
https://homepages.thm.de/~hg53/ksp-ws1617/ksp.pdf

Activation-Record (AR) / Rahmen

Generischer Inhalt:

- **Lokale Daten**

Parameter	aktuelle Argumente
lokale Variablen	aktuelle Variablen-Werte
Return-Wert	Platz für das Ergebnis einer aufgerufenen (anderen) Funktion

- **Verwaltungsinformationen**

Return-Adresse	Rücksprung-Adresse, Adresse der Anweisung hinter dem Aufruf
dynamischer Link	Zeiger auf der AR der aufrufenden Prozedur / Funktion
statischer Link	Zeiger auf den AR der definierenden Prozedur / Funktion Hier finden sich die globalen Variablen Nur notwendig bei Sprachen mit geschachtelten Gültigkeitsbereichen

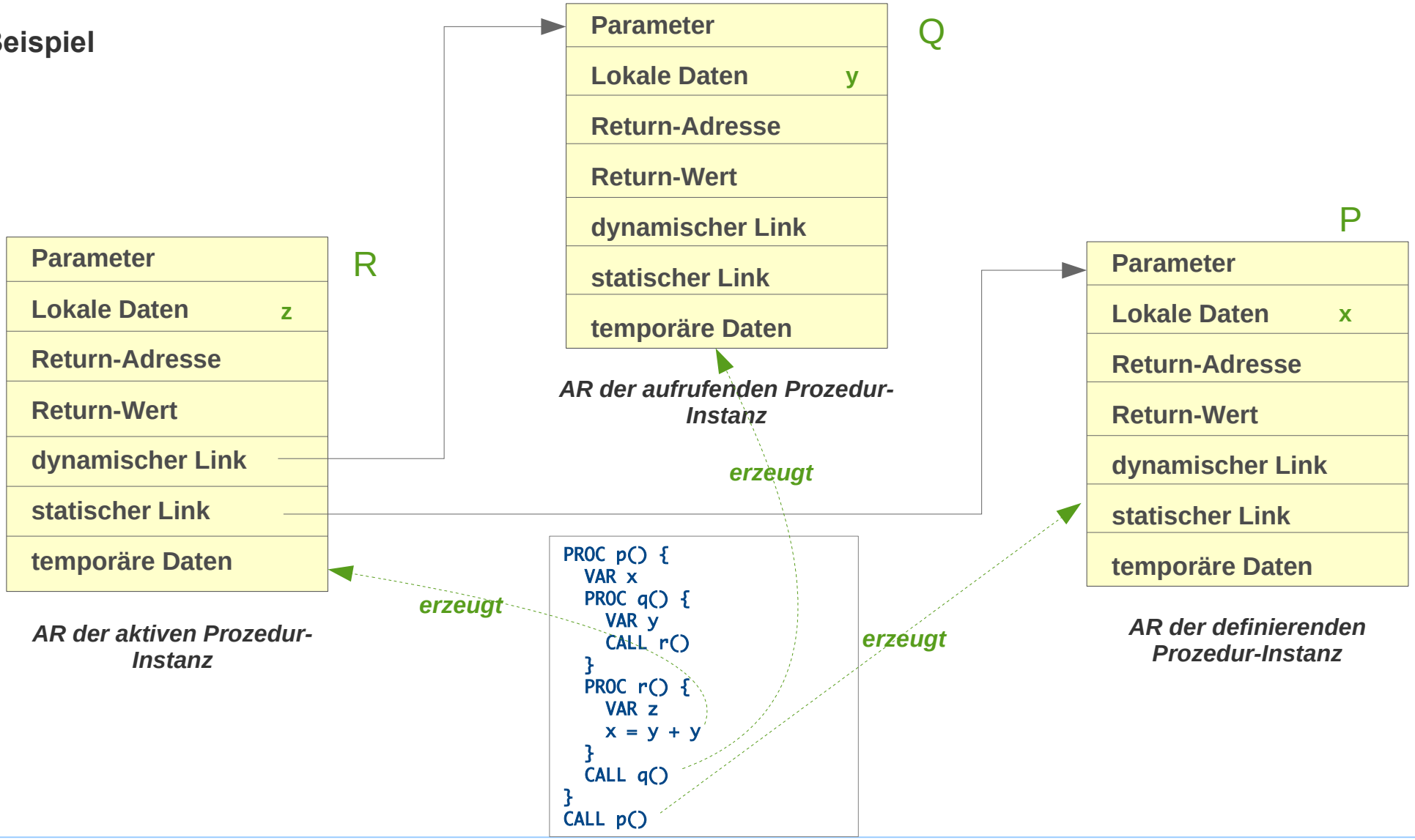
- **Hilfsinformationen**

Register-Bereich	Inhalt von Registern die bis zur Rückkehr an die Aufrufstelle gespeichert werden
temporäre Daten	Hilfsvariablen die für die Ausdrucksauswertung gebraucht werden

Activation-Record (AR) / Rahmen

Generischer Inhalt

Beispiel



Die Begriffe AR, Activation-Record, Frame, und Rahmen werden hier als Synonyme behandelt.

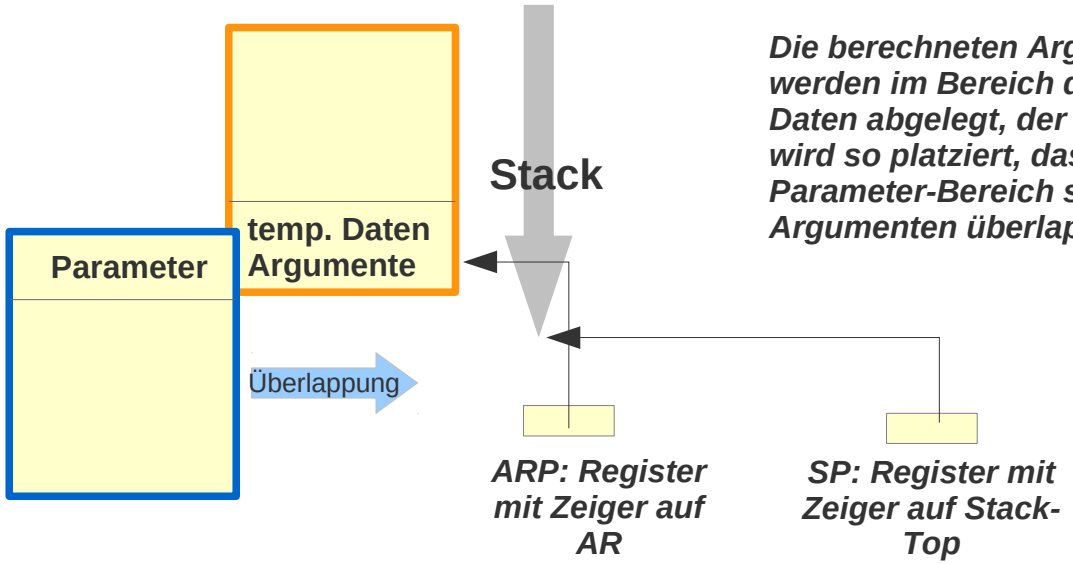
Activation-Record (AR) / Rahmen

Konkreter Inhalt: Trickreich optimiert

ARs und ihre Verwaltung sind ganz zentral für die Effizienz einer Programmausführung

Man versucht sie darum trickreich

- **möglichst kompakt anzulegen**
  enthält nur die notwendigen Informationen
  eventuell mit Überlappungen: z.B. die temporären Daten des einen AR überlappen sich mit den Parametern des anderen AR
- **und mit möglichst wenig Aufwand zu verwalten**
  wichtige / häufig gebrachte Daten in Registern statt im Speicher

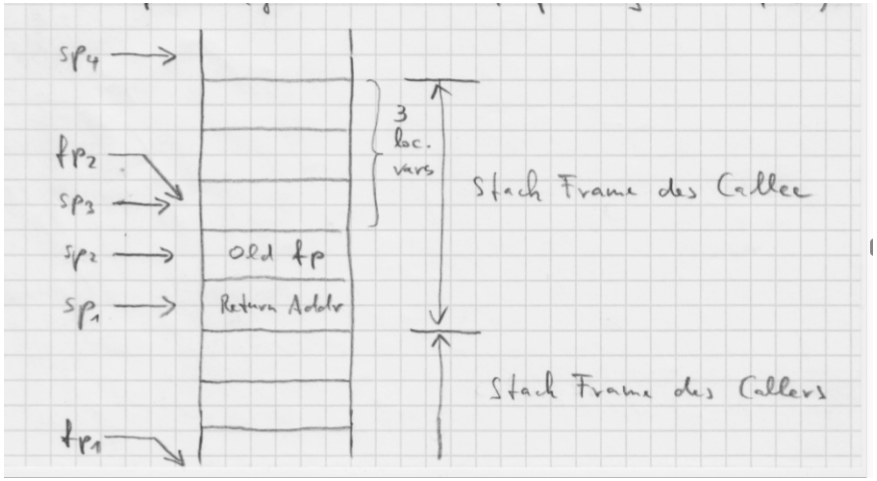


Die berechneten Argumente werden im Bereich der temporären Daten abgelegt, der nächste AR wird so platziert, dass sein Parameter-Bereich sich mit den Argumenten überlappt.

Activation-Record (AR) / Rahmen

Konkreter Inhalt: Trickreich optimiert

Beispiel



Layout eines Activation-Records in der Ninja-VM („Frame-Layout“)

Format für flache Sprachen: verschachtelte Gültigkeitsbereiche können nicht unterstützt werden.

vergl. Konzepte Systemnaher Programmierung
Prof. Dr. Hellwig Geisse
https://homepages.thm.de/~hg53/ksp-ws1617/ksp.pdf

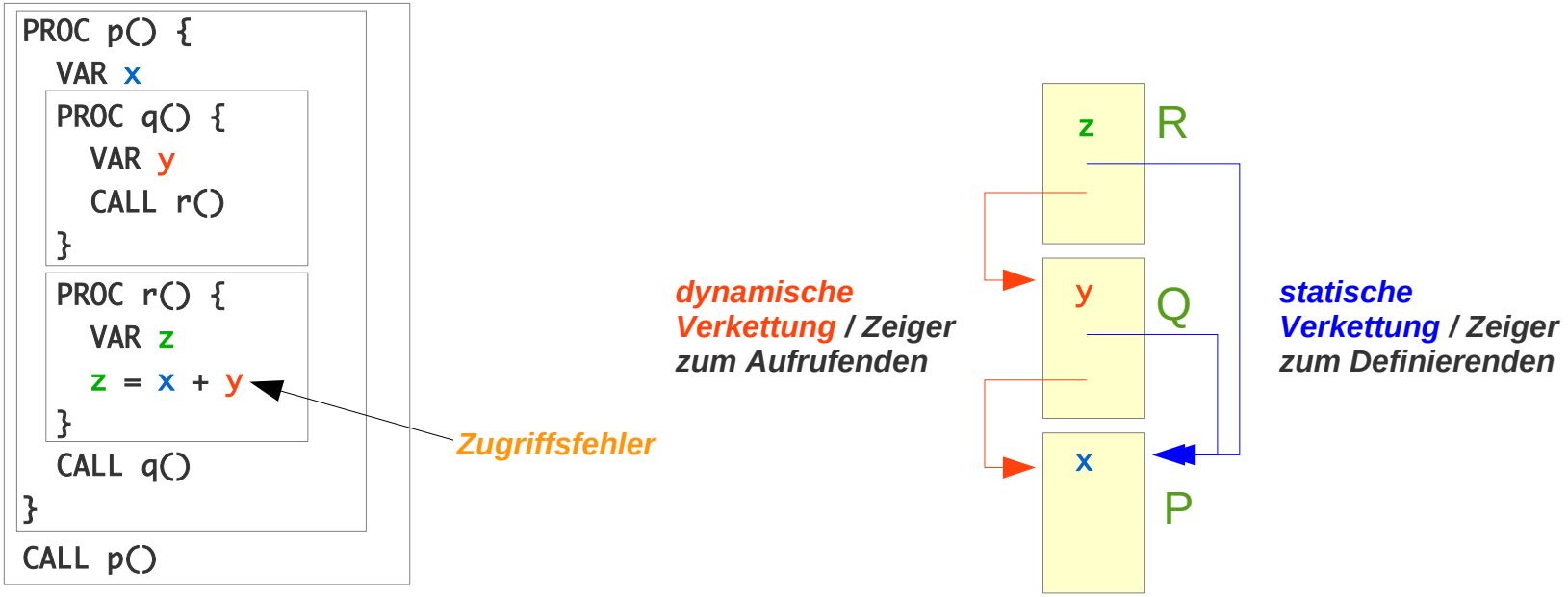
Dynamische Daten: Lokal oder in der statischen Verkettung

Lokale Variablen und Parameter

- sind im aktuellen *Activation Record* (AR) zu finden
- können mit einer Distanzadresse (z.B. relativ zum Anfang des AR) adressiert werden

Nicht lokale Variablen und Parameter

- sind in einem der AR entlang der statischen Verkettung
  (im definierenden / umfassenden Gültigkeitsbereich oder dessen definierenden / umfassenden ...)



Dynamische Daten: Lokal oder in der statischen Verkettung

Adresse Lokaler Variablen und Parameter

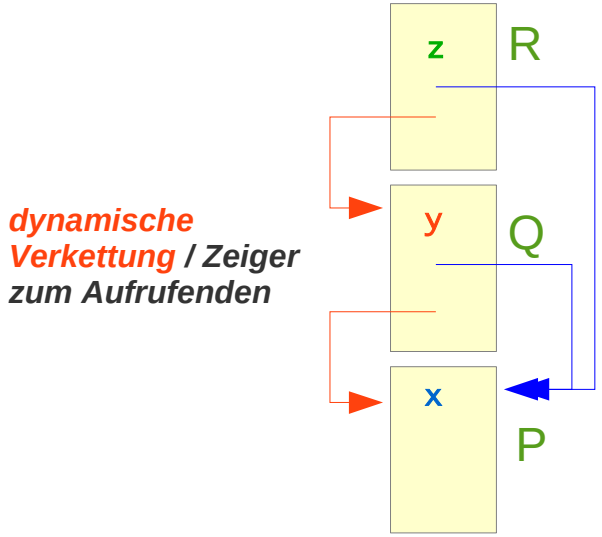
– Adresse aktueller AR + Distanzadresse

Adresse Nicht lokaler Variablen und Parameter

– Adresse definierender AR + Distanzadresse

```
PROC p() {
  VAR x
  PROC q() {
    VAR y
    CALL r()
  }
  PROC r() {
    VAR z
    z = x + y
  }
  CALL q()
}
CALL p()
```

Zugriffsfehler



dynamische Verkettung / Zeiger zum Aufrufenden

statische Verkettung / Zeiger zum Definierenden

Laufzeitorganisation: Prozeduren

Dynamische Daten: Lokal oder in der statischen Verkettung

Beobachtung:

Pro Verschachtelungstiefe kann es nur genau einen aktiven Gültigkeitsbereich geben

Denn:

- **Greift eine Prozedur auf einen globalen Namen zu**
- **und ist dieser Name in einer rekursiven Prozedur definiert, von der mehrere Instanzen erzeugt wurden, dann gibt es mehrere Instanzen eines globaleren Gültigkeitsbereichs.**
- **Aber: Nur auf Bindungen in der zuletzt aktivierten Instanz kann zugegriffen werden.**

Beispiel:

```
object Nesting extends App {
  def f(x: Int) : Unit = {
    def g: Unit = {
      // Ausgabe des x von genau einer von eventuell vielen
      // f-Instanzen mit einem x: der zuletzt aktivierten
      println(s"x = $x")
    }

    if (x > 0) f(x-1) else g
  }

  f(5)
}
```

```
x = 0
```

x ist global in g; es gibt in jeder Instanz von f ein x, gemeint in g ist „das x im letzten f“

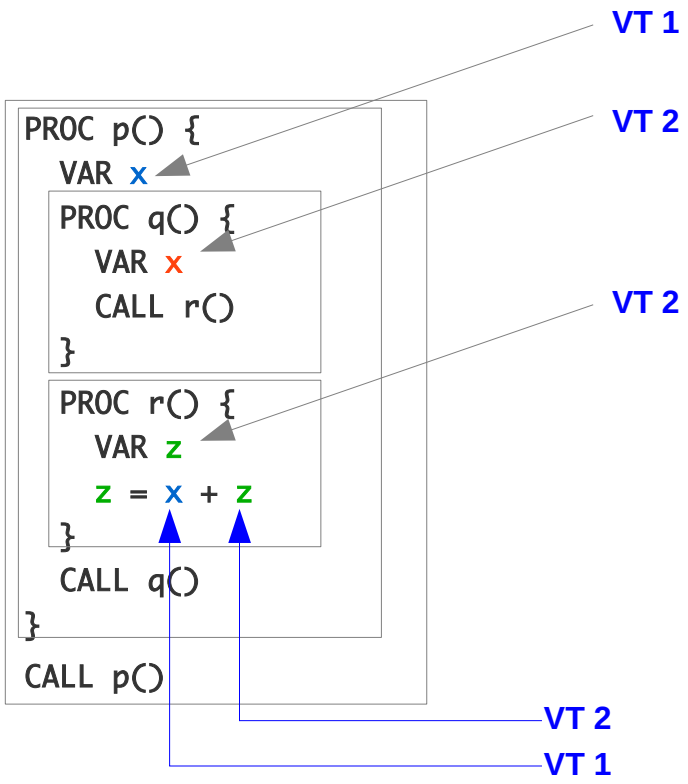
Adressierung Nicht-lokale Daten

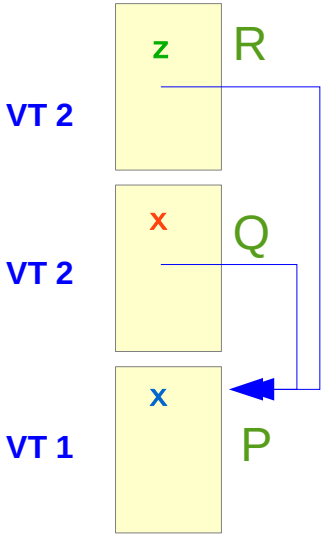
Schlussfolgerung:

Jeder Name kann über die Verschachtelungstiefe adressiert werden.

Ein Name (einer Variablen) kann darum repräsentiert werden durch drei numerische Werte:

- **Verschachtelungstiefe**
- **Adresse des aktiven Rahmens dieser Verschachtelungstiefe**
- **Adresse des Wertes im Rahmen**





im Code von R:
x ist in VT 1, hier ist VT 2, also ist x 1-en Schritt in der statischen Verkettung entfernt:
2-1 = 1

Laufzeitorganisation: Prozeduren

Adressierung Nicht-lokale Daten

Jeder Name kann repräsentiert werden durch drei numerische Werte (zwei davon statisch, einer dynamisch):

Statische Werte:

- **Verschachtelungstiefe**
- **Adresse des Wertes im Rahmen**

Dynamischer Wert:

- **Adresse des aktiven Rahmens dieser Verschachtelungstiefe**

Übersetzungszeit: Jeder definierte Name kann zur Übersetzungszeit transformiert werden in ein Paar

(Verschachtelungstiefe, Distanzadresse)

Laufzeit: Zur Laufzeit muss existieren:

- **Der Stapel der ARs aller betretenen aber noch nicht verlassenen Gültigkeitsbereiche**
- **Eine Abbildung**
  Verschachtelungstiefe ~> Adresse des ARs dieser Verschachtelungstiefe

Adressierung Nicht-lokaler Daten

Frontend

Um dynamische Daten zur Laufzeit adressieren zu können,
muss das Frontend jedem Namen

- **Distanzadresse und**
- **Verschachtelungstiefe**

zuordnen

Backend

Das Backend muss aus diesen Informationen den passenden Code zum Zugriff generieren
und dazu stets die Adressen aller aktiven ARs (Rahmen) zur Verfügung haben

Adressierung Nicht-lokaler Daten

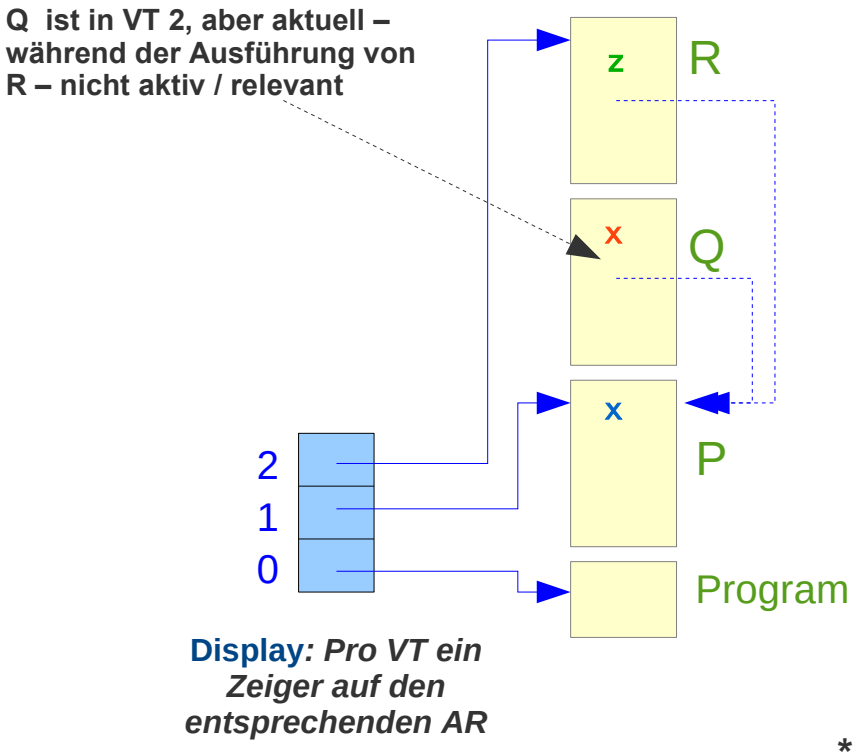
Display-Technik*

Einen von vielen Möglichkeiten, den Zugriff auf dynamische Daten zu organisieren ist die Display-Technik:

Display: Tabelle mit Zeigern auf die ARs pro Verschachtelungstiefe gibt es genau einen aktiven AR

Das Display kann in Registern oder in den ARs selbst gespeichert werden.

```
PROGRAM
  PROC p() {
    VAR x
    PROC q() {
      VAR x
      CALL r()
    }
    PROC r() {
      VAR z
      z = x + z
    }
    CALL q()
  }
BEGIN
  CALL p()
END
```



Display: *Pro VT ein Zeiger auf den entsprechenden AR*

im Code von R:

x ist in VT 1.
Dort hat es die Distanzadresse 0.
Zugriff auf x insgesamt:

Display[1] + 0

z ist in VT 2.
Dort hat es die Distanzadresse 0.
Zugriff auf z insgesamt:

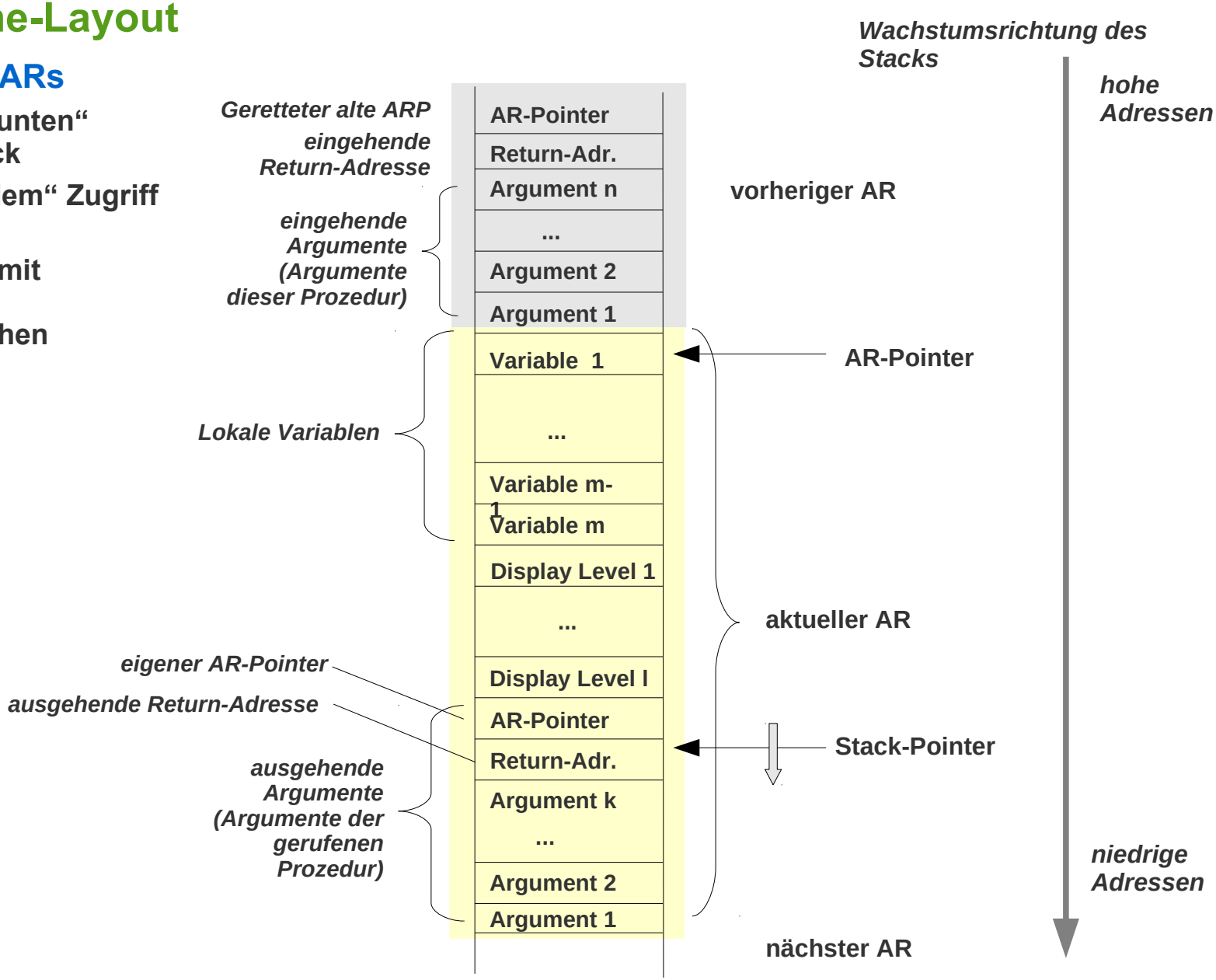
Display[2] + 0

* von E.W. Dijkstra für Implementierung von Algol 60 vorgeschlagen

Struktur AR / Frame-Layout

Beispiel: Displays in ARs

- mit einem nach „unten“ wachsenden Stack
- mit „überlappendem“ Zugriff auf Argumente
- für eine Sprache mit verschachtelten Gültigkeitsbereichen



Laufzeitorganisation: Prozeduren

Aufruf-Kontrakt

„Vertrag“ zwischen Rufer und Gerufenem / Beispiel

Der Vertrag hängt ab vom Format der ARs und kann natürlich auch in anderer Art gestaltet werden.

	Rufer	Gerufener
Aufruf	Push Display-Pointer Push AR-Pointer Push Argumente Push Return-Adresse Sprung zum Gerufenem	ARTemp <= Stack-Pointer (SP) Push lokale Variablen DPTemp <= Stack-Pointer Push Display (statische Verkettung) Display-Pointer <= DPTemp AR-Pointer <= ARTemp führe Prozedur-Code aus
Rückkehr	Pop Argumente DP <= Pop Display-Pointer ARP <= Pop AR-Pointer	Pop Display Pop Lokale Variablen Pop Return-Adresse Springe zu Return-Adresse

Push und Pop-Operationen bewegen den Stack-Pointer (SP)