Wertzuweisung

Für die Wertzuweisung $x \leftarrow y + 1$ ergibt sich mit Adressumgebung $\rho = \{x \mapsto 5, y \mapsto 7\}$:

$\text{code}_W^\rho\ (x \leftarrow y + 1)$	Regel für Wertzuweisung
$= \text{code}_W^\rho\ (y + 1), \text{code}_A^\rho\ x,$ **store**	Regel für binäre Op.
$= \text{code}_W^\rho\ y, \text{code}_W^\rho\ 1,$ **add**, $\text{code}_A^\rho\ x,$ **store**	Variable rechts
$= \text{code}_A^\rho\ y,$ **load**, $\text{code}_W^\rho\ 1,$ **add**, $\text{code}_A^\rho\ x,$ **store**	Konstante
$= \text{code}_A^\rho\ y,$ **load**, **loadc** 1, **add**, $\text{code}_A^\rho\ x,$ **store**	Variable links
$= \text{code}_A^\rho\ y,$ **load**, **loadc** 1, **add**, **loadc** $\rho(x),$ **store**	Adresse berechnen
$=$ **loadc** $\rho(y)$, **load**, **loadc** 1, **add**, **loadc** $\rho(x)$, **store**	Adressumgebung ρ
$=$ **loadc** 7, **load**, **loadc** 1, **add**, **loadc** 5, **store**	

Wertzuweisung

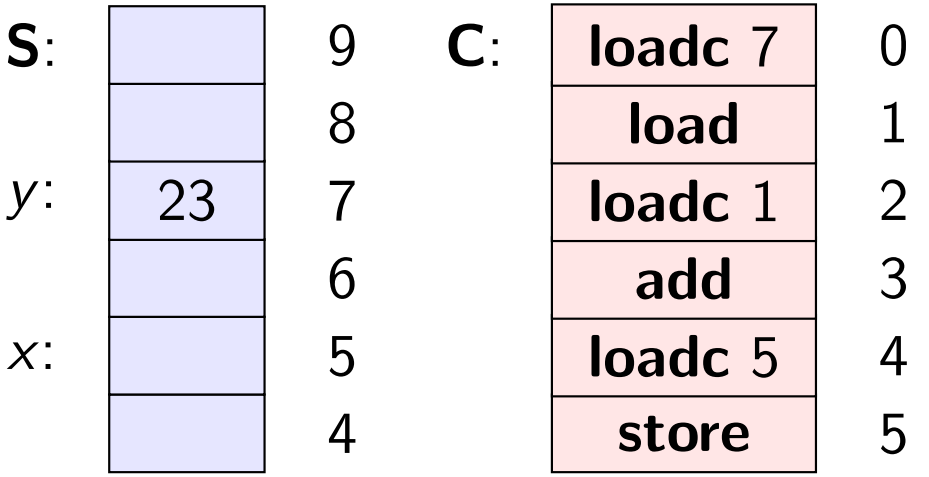
Kurz: Mit Adressumgebung $\rho = \{x \mapsto 5,\ y \mapsto 7\}$ ergibt sich für die Wertzuweisung $x \leftarrow y + 1$:

loadc 7, **load**, **loadc** 1, **add**, **loadc** 5, **store**

Ermittle Wert der rechten Seite.

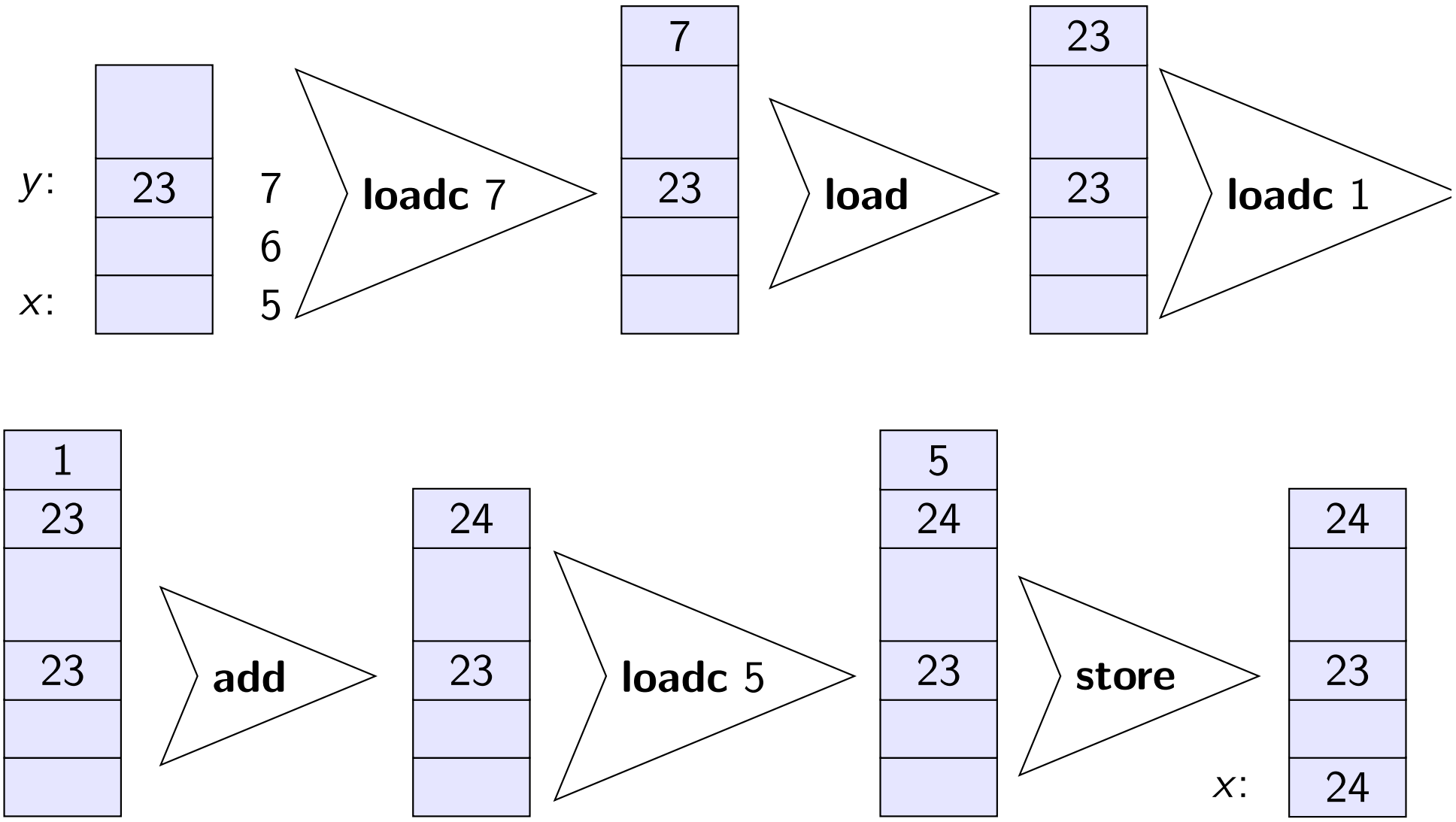
Berechne Adresse der linken Seite.

Führe Wertzuweisung aus.



Übersetzung der Wertzuweisung $x \leftarrow y + 1$.

Abarbeitung der Befehlsfolge für $x \leftarrow y + 1$



Wertzuweisung

Eine Wertzuweisung $a \leftarrow (b + (b \cdot c))$ wird mit Adressumgebung $\rho = \{a \mapsto 5,\ b \mapsto 6,\ c \mapsto 7\}$ übersetzt zu:

$\text{code}_W^\rho\ (a \leftarrow (b + (b \cdot c)))$

$= \text{code}_W^\rho\ (b + (b \cdot c)),\ \text{code}_A^\rho\ a,\ \textbf{store}$

$= \text{code}_W^\rho\ b,\ \text{code}_W^\rho\ (b \cdot c),\ \textbf{add},\ \text{code}_A^\rho\ a,\ \textbf{store}$

$= \textbf{loadc}\ 6,\ \textbf{load},\ \text{code}_W^\rho\ (b \cdot c),\ \textbf{add},\ \text{code}_A^\rho\ a,\ \textbf{store}$

$= \textbf{loadc}\ 6,\ \textbf{load},\ \text{code}_W^\rho\ b,\ \text{code}_W^\rho\ c,\ \textbf{mul},\ \textbf{add},\ \text{code}_A^\rho\ a,\ \textbf{store}$

$= \textbf{loadc}\ 6,\ \textbf{load},\ \textbf{loadc}\ 6,\ \textbf{load},\ \text{code}_W^\rho\ c,\ \textbf{mul},\ \textbf{add},\ \text{code}_A^\rho\ a,\ \textbf{store}$

$= \textbf{loadc}\ 6,\ \textbf{load},\ \textbf{loadc}\ 6,\ \textbf{load},\ \textbf{loadc}\ 7,\ \textbf{load},\ \textbf{mul},\ \textbf{add},\ \textbf{loadc}\ 5,\ \textbf{store}$

Optimierung: Befehle für häufige Befehlsfolgen

Routineaufgaben wiederholen sich und führen zu ähnlichen Befehlsfolgen.

Oft wird ein Wert von einer konstanten (zur Übersetzungszeit bekannten) Adresse geladen bzw. an diese Adresse geschrieben.

Als Optimierung können wir dafür Spezialbefehle einführen:

loada $q \equiv$ **loadc** q, **load**
storea $q \equiv$ **loadc** q, **store**

Diese Spezialbefehle erhöhen die Effizienz:

- der erzeugte Code wird kürzer;
- die Implementierung oft effizienter, z.B. kann **storea** q

statt durch $\overbrace{SP{+}{+};\ S[SP] \leftarrow q;}^{\textbf{loadc } q}\ \overbrace{S[S[SP]] \leftarrow S[SP-1];\ SP--;}^{\textbf{store}}$

durch $S[q] \leftarrow S[SP];$ implementiert werden.

Wertzuweisung

Eine Wertzuweisung $a \leftarrow (b + (b \cdot c))$ wird mit
Adressumgebung $\rho = \{a \mapsto 5,\ b \mapsto 6,\ c \mapsto 7\}$ übersetzt zu:

$\text{code}^{\rho}_{W}\ (a \leftarrow (b + (b \cdot c)))$

$= \text{code}^{\rho}_{W}\ (b + (b \cdot c)),\ \text{code}^{\rho}_{A}\ a,\ \textbf{store}$

$= \text{code}^{\rho}_{W}\ b,\ \text{code}^{\rho}_{W}\ (b \cdot c),\ \textbf{add},\ \text{code}^{\rho}_{A}\ a,\ \textbf{store}$

$= \textbf{loadc}\ 6,\ \textbf{load},\ \text{code}^{\rho}_{W}\ (b \cdot c),\ \textbf{add},\ \text{code}^{\rho}_{A}\ a,\ \textbf{store}$

$= \textbf{loadc}\ 6,\ \textbf{load},\ \text{code}^{\rho}_{W}\ b,\ \text{code}^{\rho}_{W}\ c,\ \textbf{mul},\ \textbf{add},\ \text{code}^{\rho}_{A}\ a,\ \textbf{store}$

$= \textbf{loadc}\ 6,\ \textbf{load},\ \textbf{loadc}\ 6,\ \textbf{load},\ \text{code}^{\rho}_{W}\ c,\ \textbf{mul},\ \textbf{add},\ \text{code}^{\rho}_{A}\ a,\ \textbf{store}$

$= \underbrace{\textbf{loadc}\ 6,\ \textbf{load}}_{\textbf{loada}\ 6},\ \underbrace{\textbf{loadc}\ 6,\ \textbf{load}}_{\textbf{loada}\ 6},\ \underbrace{\textbf{loadc}\ 7,\ \textbf{load}}_{\textbf{loada}\ 7},\ \textbf{mul},\ \textbf{add},\ \underbrace{\textbf{loadc}\ 5,\ \textbf{store}}_{\textbf{storea}\ 5}$

6 Befehle statt 10 Befehlen

Verallgemeinerte Lade- und Speicheroperationen

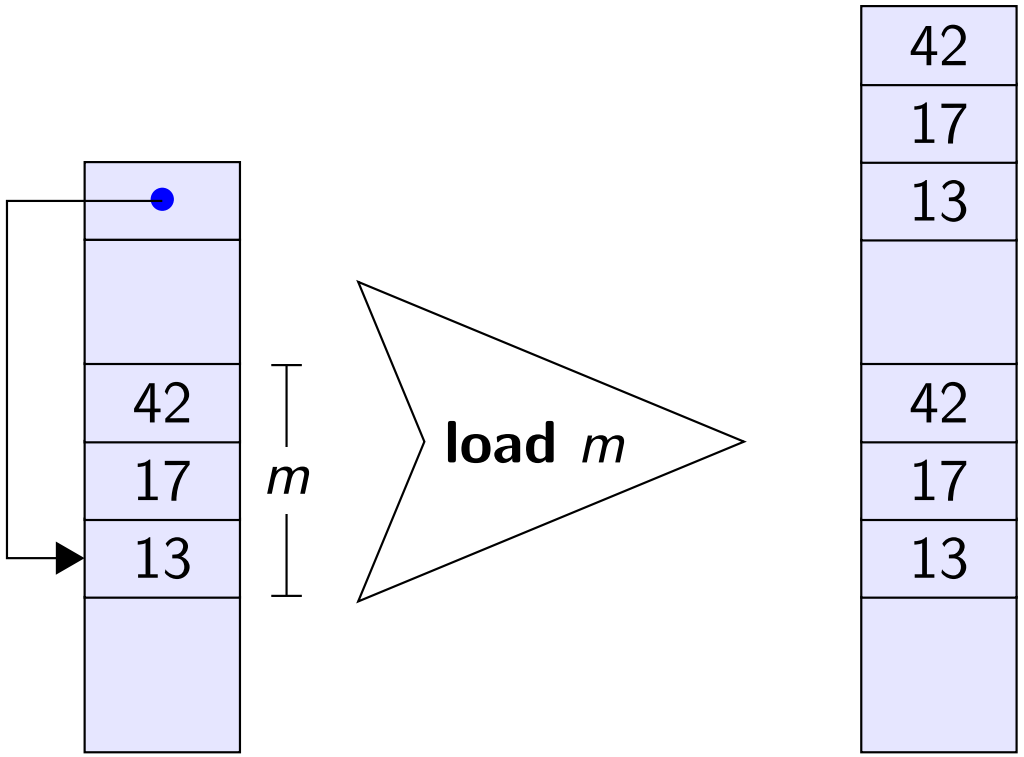
Später werden wir Daten bewegen, die größer sind als eine Speicherzelle.

Daher verallgemeinern wir die Befehle **load** und **store** zu Befehlen **load** *m* bzw. **store** *m* für nicht-negative Werte *m*.

Der Befehl **load** *m* legt den Inhalt von *m* aufeinanderfolgenden Zellen auf den Keller – ab der Adresse, die oben auf dem Keller liegt.
Der Befehl **load** 1 entspricht unserem bisherigen Befehl **load**.

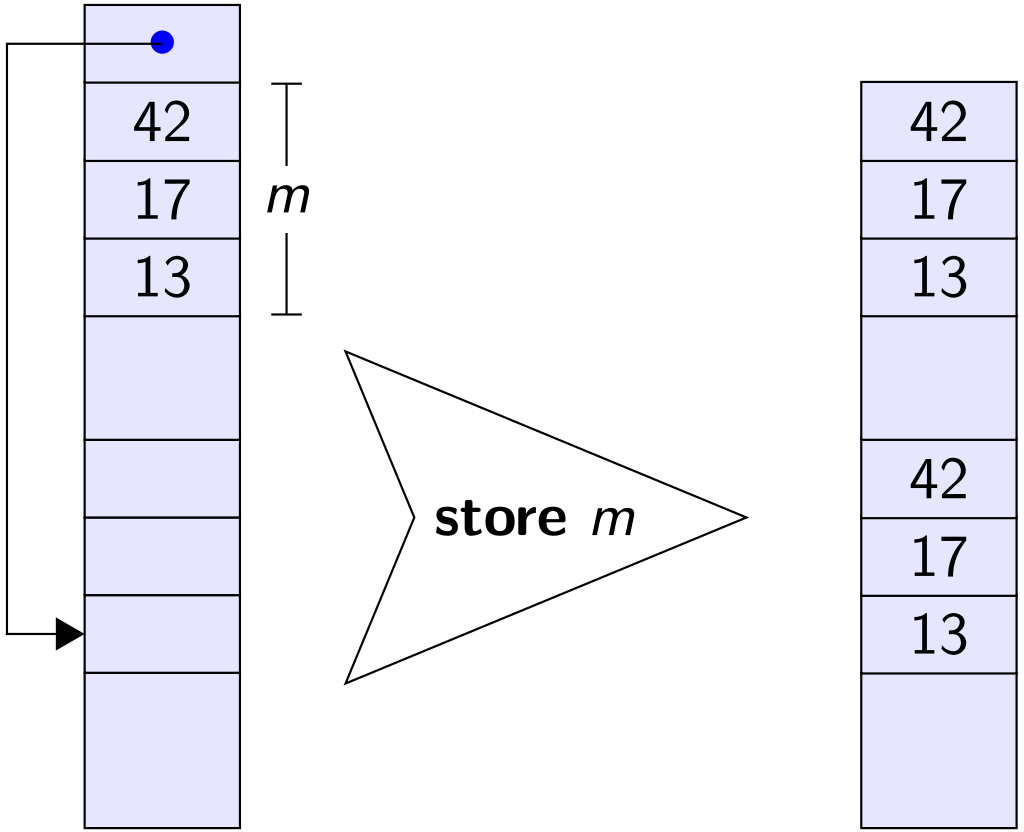
Natürlich können wir wieder abkürzen:
loada *q m* $\equiv$ **loadc** *q*, **load** *m*
storea *q m* $\equiv$ **loadc** *q*, **store** *m*

Laden aufeinanderfolgender Speicherzellen



for $i \leftarrow m-1;\ i \geq 0;\ i{-}{-}$ **do**
$S[SP+i] \leftarrow S[S[SP]+i];$
$SP \leftarrow SP+m-1;$
Der Befehl **load** m.

Speichern in aufeinanderfolgenden Speicherzellen



for $i \leftarrow 0;\ i < m;\ i{+}{+}$ **do**
$S[S[SP]{+}i] \leftarrow S[SP{-}m{+}i];$
$SP - -;$

Der Befehl **store** m.

Anweisung

In C gilt: Ist e ein Ausdruck, dann ist $e;$ eine **Anweisung**.

Eine Anweisung liefert keinen Wert zurück.
Folglich muss der Kellerzeiger SP vor und nach einer Anweisung den gleichen Wert haben.

Deshalb entfernen wir das Ergebnis des Ausdrucks e.
Ist $|e|$ die Größe (des Typs) von e, so erzeugen wir:

$$\text{code}^{\rho}(e;) = \text{code}^{\rho}_{W}\ e,\ \textbf{alloc}\ -|e|$$

Anweisungsfolge

Der Code für eine **Folge von Anweisungen** entsteht durch Konkatenation der Codesequenzen der einzelnen Anweisungen:

$\text{code}^{\rho}(s\ ss) = \text{code}^{\rho}\ s, \text{code}^{\rho}\ ss$ // s Anweisung, ss Anweisungsfolge

$\text{code}^{\rho}(\varepsilon) = \varepsilon$ // ε leere Folge von Anweisungen

Sprungbefehle

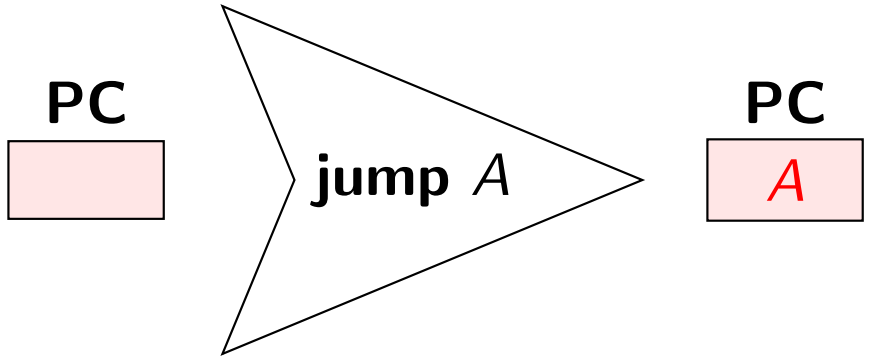
Als nächstes übersetzen wir die **binäre Fallunterscheidung** (**Alternative**). Wir geben ein Übersetzungsschema an für die **if**-Anweisung:

if e **then** s_1 **else** s_2

wobei e ein Ausdruck und jedes s_i eine Anweisung oder eine zu einem Block zusammengefasste Anweisungsfolge ist.

Um von der sequentiellen Programmausführung abzuweichen, benötigen wir Sprungbefehle.

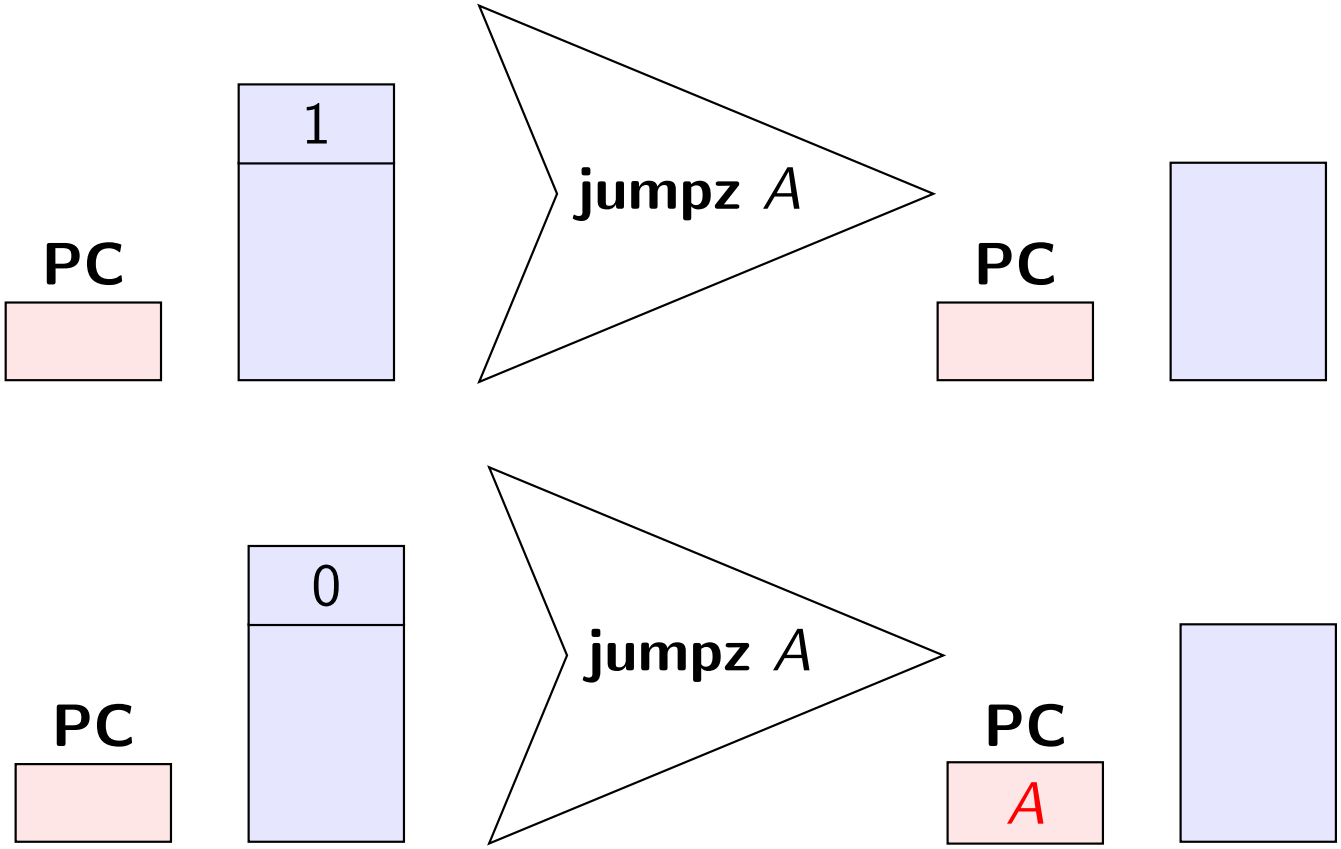
Ein **unbedingter Sprung** setzt die Programmausführung an anderer Stelle fort, und zwar immer.



$PC \leftarrow A;$

Sprungbefehle

Ein **bedingter Sprung** weicht nur dann von der sequentiellen Programmausführung ab, wenn eine bestimmte Bedingung zutrifft.
Als Bedingung testen wir, ob das oberste Kellerelement 0 (**false**) ist.



if $S[SP] = 0$ **then** $PC \leftarrow A$; $SP{-}{-}$;

Marken

Für die Übersetzung von bedingten Anweisungen und Schleifen benötigen wir ein neues Sprachmittel:

Wir markieren Befehle oder das Ende des Schemas durch symbolische **Marken** (*label*), die wir in Sprungbefehlen als Ziele verwenden.

Eine Marke steht für die –zu diesem Zeitpunkt evtl. noch unbekannte– Adresse des Befehls, der diese Marke trägt.

Im nächsten Durchlauf können die Marken durch absolute Codeadressen ersetzt werden.

Binäre Fallunterscheidung

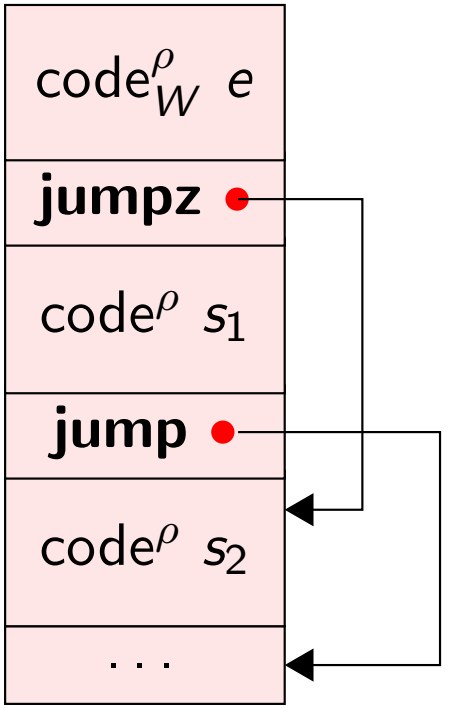
Betrachten wir eine **binäre Fallunterscheidung**: **if** e **then** s_1 **else** s_2

Wir legen Codesequenzen für e, s_1 und s_2 in den Programmspeicher.

Dazwischen fügen wir Sprünge für den korrekten Programmablauf ein:

Wird die Bedingung e zur Laufzeit zu **false** (0) ausgewertet, springen wir an den Anfang des **else**-Teils s_2.

Sonst wird der Code für s_1 ausgeführt. Dahinter fügen wir einen unbedingten Sprung hinter die Fallunterscheidung ein, damit der Code für s_2 nicht ausgeführt wird.



Code für die Alternative.

code$^{\rho}$(**if** e **then** s_1 **else** s_2) = code$^{\rho}_{W}$ e, **jumpz** A, code$^{\rho}$ s_1, **jump** B,
A: code$^{\rho}$ s_2, B: . . .

Binäre Fallunterscheidung

Beispiel: Sei $\rho = \{x \mapsto 4,\ y \mapsto 7\}$.

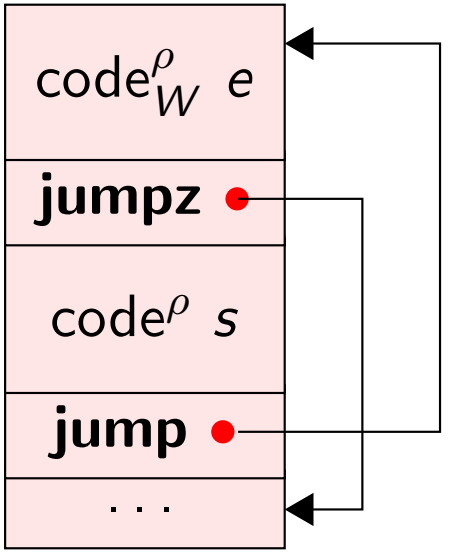
code^{ρ} (**if** $x > y$ **then** $x \leftarrow x - y;$ **else** $y \leftarrow y - x;$) =

loada 4, **loada** 7, **gr**, **jumpz** A,
loada 4, **loada** 7, **sub**, **storea** 4, **alloc** -1, **jump** B,
A: **loada** 7, **loada** 4, **sub**, **storea** 7, **alloc** -1, B: ...

Wiederholungsanweisungen (Schleifen)

Betrachten wir eine **while-Schleife**: **while** e **do** s

Hinter den Code für die Bedingung e fügen wir einen bedingten Sprung ein, der die Schleife verlässt.
Am Ende des Codes für den Rumpf wird ein unbedingter Sprung zurück an den Anfang der Schleife eingefügt
– vor den Code der Bedingung.



Code für die while-Schleife.

code^{ρ}(**while** e **do** s) = A: $\text{code}^{\rho}_{W}\ e$, **jumpz** B, $\text{code}^{\rho}\ s$, **jump** A, B: . . .

Wiederholungsanweisungen (Schleifen)

Beispiel: Sei $\rho = \{a \mapsto 7,\ b \mapsto 8,\ c \mapsto 9\}$.

code^{ρ} (**while** $a > 0$ **do** $\{c \leftarrow c + 1;\ a \leftarrow a - b;\}$) =

A: **loada** 7, **loadc** 0, **gr**, **jumpz** B,
loada 9, **loadc** 1, **add**, **storea** 9, **alloc** -1,
loada 7, **loada** 8, **sub**, **storea** 7, **alloc** -1, **jump** A, B: ...

Im Rumpf einer Schleife sind **break**- oder **continue**-Anweisungen erlaubt:

- Durch ein **break** wird die Schleife durch einen unbedingten Sprung verlassen.
- Ein **continue** beendet nur den aktuellen Schleifendurchlauf und springt an das Ende des Schleifenrumpfs.

Die Übersetzungsfunktion muss auch die Sprungziele für **break** bzw. **continue** erzeugen und verwalten.

Speicherüberlauf

Bisher haben wir unbefangen den Wert des Registers SP erhöht, um neue Speicherzellen des Datenspeichers zu nutzen.

Um einen **Kellerüberlauf** (*stack overflow*) zu vermeiden, hätten wir vor jeder Erhöhung prüfen müssen, ob die maximale Größe des Datenspeichers schon erreicht ist.

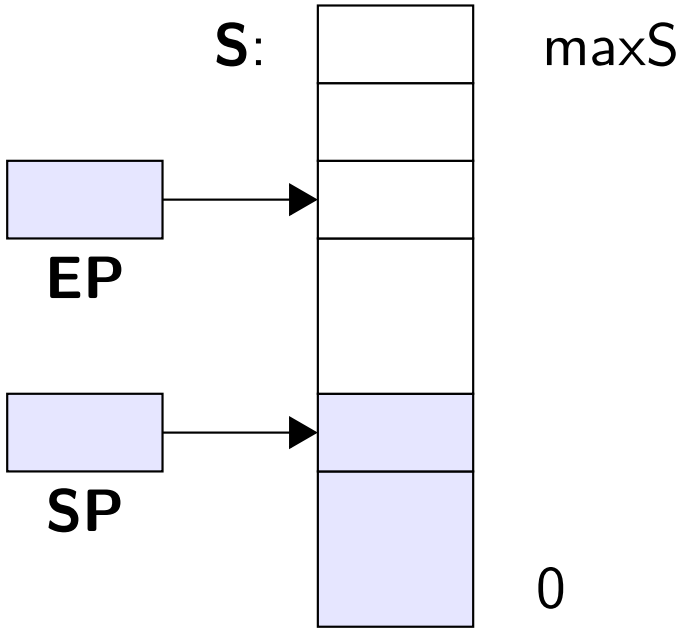
Für eine gegebene Anweisungsfolge lässt sich schon zur Übersetzungszeit der maximal nötige Speicherplatz für Zwischenergebnisse berechnen, und damit die oberste Kellerzelle, auf die SP bei deren Auswertung zeigen kann.

Wir ermitteln daraus die Adresse der obersten Kellerzelle, auf die SP bei der Auswertung von Ausdrücken im Anweisungsteil des aktuellen Funktionsaufrufs zeigen kann und sparen damit die vielen Vergleiche bei der Vergrößerung des Kellers ein.

Schrankenzeiger

Für das Hauptprogramm und jeden Funktionsaufruf prüfen wir dann nur, ob diese Adresse kleiner gleich der oberen Grenze des Kellers ist.

Für spätere Zwecke speichern wir diese Adresse im **Schrankenzeiger**, einem Register **EP**.



Speicherplatzbedarf für Zwischenergebnisse

Maximale Anzahl der Speicherzellen zur Auswertung eines **Ausdrucks** e (Alle Operanden und Ergebnisse mögen in je eine Speicherzelle passen)

$$cn(e) = \begin{cases} 1 & \text{, falls } e \text{ Variable oder Konstante} \\ cn(e_1) & \text{, falls } e \equiv op_{unär}\ e_1 \\ \max\{cn(e_1),\ 1 + cn(e_2)\} & \text{, falls } e \equiv e_1\ op_{binär}\ e_2 \end{cases}$$

Denn bei unären Operationen wird nur das Ergebnis von e_1 durch das Ergebnis von e ersetzt.
Bei binären Operationen muss man bei der Berechnung von e_2 zusätzlich das Ergebnis von e_1 speichern.

Für jede kommutative binäre Operation mit $cn(e_2) > cn(e_1)$ kann man einen Speicherplatz sparen, indem man die Operanden vertauscht.

Funktionen

Die **Deklaration** einer Funktion besteht aus:

- einem **Namen**, unter dem die Funktion aufrufbar ist,
- einer **Ein-/Ausgabeschnittstelle**: der Spezifikation (Namen und Typen) der formalen Parameter und der Rückgabewerte,
- einem **Rumpf** (*body*), der aus (lokalen) Deklarationen und Anweisungen besteht.

Funktionen werden **aufgerufen** (aktiviert), wenn ein Vorkommen ihres Namens im Anweisungsteil einer Funktion abgearbeitet wird.

Eine Funktion kann weitere Funktionen aufrufen (auch sich selbst).

Hat eine aufgerufene Funktion g ihren Anweisungsteil abgearbeitet, so wird sie **verlassen** und die Funktion f, die g aktiviert hat, fährt in der Ausführung hinter dem Aufruf fort.

Funktionen: Aufrufbaum

Die Funktionsaufrufe während der Ausführung eines Programms bilden einen geordneten Baum, den **Aufrufbaum** des Programmlaufs.

Die Wurzel des Aufrufbaums ist markiert mit dem Namen der Funktion *main*, mit deren Aufruf die Programmausführung startet.

Jeder Knoten im Aufrufbaum ist markiert mit einem Funktionsnamen f,
sein direkter Vorgänger mit dem Namen der Funktion, die diesen Aufruf von f ausgeführt hat;
seine direkten Nachfolger bilden die Liste der Funktionen, die f aufgerufen hat – geordnet in der Reihenfolge ihrer Aufrufe.

Eine Markierung f kann mehrfach im Aufrufbaum auftreten;
sie tritt so oft auf, wie f aufgerufen wird.

Funktionen: Inkarnation

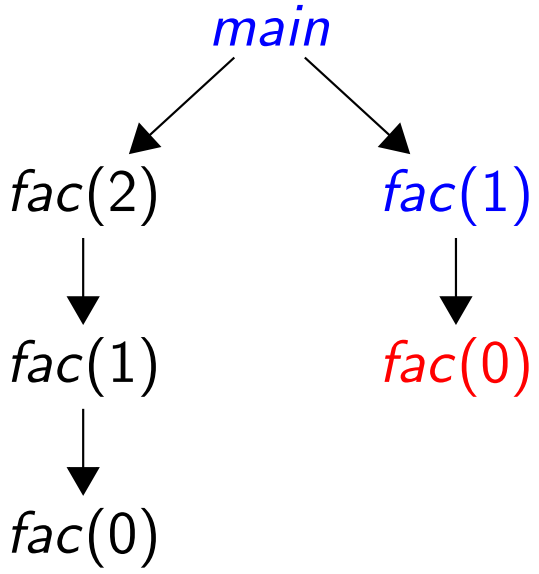
Jedes Vorkommen von f im Aufrufbaum heißt **Inkarnation** von f; der Weg von der Wurzel des Aufrufbaums bis zu diesem Knoten (**Inkarnationsweg**) charakterisiert diese Inkarnation.

Betrachten wir den Zustand der Programmausführung, wenn eine bestimmte Inkarnation von f aktiv ist:
Alle Vorfahren dieser Inkarnation, also alle Knoten auf dem Inkarnationsweg, sind bereits aufgerufen, aber noch nicht verlassen worden; sie sind zu diesem Zeitpunkt **lebendig**.

Funktionen

```
int fac (int n) {
    if n ≤ 0 then return 1;
    else return n · fac(n−1);
}

int main() {
    int n;
    n ← 2;
    return fac(n) + fac(n−1);
}
```



Der Aufrufbaum.

Zum Zeitpunkt des fünften Aufrufs von *fac* sind die Funktion *main* und zwei Inkarnationen von *fac* lebendig.

Abhängig vom Wert der globalen Variablen und der aktuellen Parameter gibt es evtl. zu einem Programm

- mehr als einen Aufrufbaum bzw. sogar
- unendlich große Aufrufbäume.

Funktionen: Auswertungsstrategie

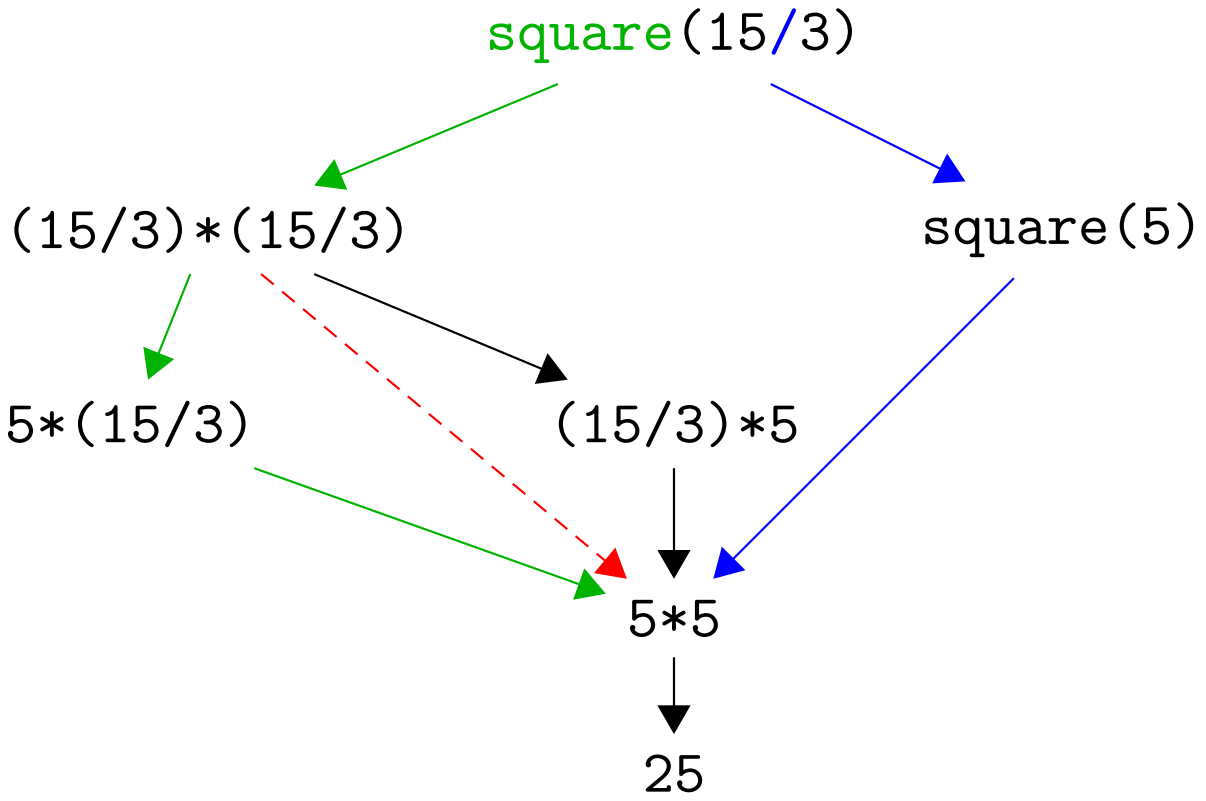
Die Auswertung eines Ausdrucks *e* geschieht durch **Termersetzung**, indem folgende zwei Schritte wiederholt werden, bis kein Ersetzungsschritt mehr möglich ist:

1. Man sucht einen Teilausdruck (**Redex**, *reducible expression*) in *e*, der mit der linken Seite einer Funktionsdefinition übereinstimmt, wobei Variablen der linken Seite durch geeignete Ausdrücke ersetzt werden.
2. Der Redex wird durch die rechte Seite der Funktionsdefinition ersetzt, wobei die Variablen auf der rechten Seite durch die gleichen Ausdrücke ersetzt werden.

Eine **Auswertungsstrategie** ist ein Algorithmus zur Auswahl des nächsten Redex.

Funktionen: Auswertungsstrategie

Wir betrachten Auswertungsreihenfolgen am Beispiel einer Funktion `square(x) = x*x` und dem Ausdruck `square(15/3)`.



Auswertungsstrategien

Funktionen: Auswertungsstrategie

Die **strikte Auswertungsstrategie** (*eager evaluation*) wählt einen der im Ausdruck am weitesten innen stehenden Redexe.
Gibt es mehrere, wird von diesen der am weitesten links stehende Redex ausgewählt (*leftmost innermost evaluation*).

Die **nicht-strikte Auswertungsstrategie** (*lazy evaluation*, **Bedarfsauswertung**, **verzögerte Auswertung**) wählt einen der im Ausdruck am weitesten außen stehenden Redexe.
Gibt es mehrere, wird von diesen der am weitesten links stehende Redex gewählt (*leftmost outermost evaluation*).
Die Argumente der Funktion sind dann i.Allg. noch nicht ausgewertet.

Funktionen: Vor- und Nachteile der Auswertungsstrategien

- Nicht-strikte Auswertung wertet nur Teilausdrücke aus, deren Wert zum Endergebnis beiträgt;

  Strikte Auswertung wertet evtl. weitere Teilausdrücke aus.
  Kritisch, wenn Auswertung teuer, fehlerhaft oder nicht terminiert.

  Der Ausdruck `g(f(0))` mit den Funktionen `f(x) = f(x+1)` (terminiert nicht!) und `g(y) = 5` (konstant) terminiert bei nicht-strikter Auswertung; bei strikter Auswertung aber nicht.
- Nicht-strikte Auswertung berechnet manchmal einen Wert mehrfach (z.B. `15/3`), obwohl dies bei strikter Auswertung nicht nötig ist.
  Diesen Nachteil vermeidet die **Bedarfsauswertung mit gemeinsamer Nutzung** (*sharing*), die als gleich erkannte Teilausdrücke nur einmal auswertet.

Alle terminierenden Auswertungsstrategien liefern das gleiche Ergebnis.
Terminiert eine Strategie, so terminiert auch die nicht-strikte Auswertung (aber evtl. nicht die strikte).

Funktionen: Parameterübergabe

Der Programmierer erkennt die Auswertungsstrategie oft nur indirekt über die Verfahren zur **Parameterübergabe**.

Der Parameterübergabemechanismus einer Programmiersprache regelt,

- in welcher Form die aktuellen Parameter übergeben und
- zu welchem Zeitpunkt diese ausgewertet werden.

Wir unterscheiden vier Möglichkeiten:
Strikte Auswertungsstrategien:

- **Adressaufruf** / **Call-By-Reference** (z.B. Pascal, C++)
- **Wertaufruf** / **Call-By-Value** (*applicative order evaluation*)
  (z.B. Algol 60, Pascal, C)

Nicht-strikte Auswertungsstrategien:

- **Formelaufruf** / **Call-By-Name** (*normal order evaluation*)
  (z.B. Algol 60)
- **Formelaufruf mit Ergebnisteilhabe** / **Call-By-Need**
  (z.B. Haskell, Scala)

Parameterübergabe: Adressaufruf / Call-By-Reference

Es wird stets eine Adresse übergeben; dort steht der Wert des Parameters.

Vorteile: Einfache Übergabe von Datenverbunden und Datenfeldern.
An die Adresse kann –als Nebenwirkung– ein (weiteres) Ergebnis an die aufrufende Funktion zurückgeliefert werden.

Nachteil: Ausdrücke (insbesondere Konstante) können für diesen Parameter nicht benutzt werden.

Parameterübergabe: Wertaufruf / Call-By-Value

Die Parameter werden zuerst ausgewertet und ihre Werte an die Funktion übergeben.

Vorteil: Die Parameter werden nur einmal ausgewertet;
außer der Auswertung entsteht kein zusätzlicher Aufwand.

Nachteil: Jeder Parameter wird ausgewertet, auch wenn sein Wert für die Auswertung des Funktionsrumpfs gar nicht benötigt wird.
Das ist kritisch, wenn die Auswertung des Parameters sehr teuer ist, zu einem Laufzeitfehler führt oder nicht terminiert.

Call-By-Value hat daher schlechtere Terminierungseigenschaften als Call-By-Name.

Parameterübergabe: Formelaufruf / Call-By-Name

Es wird mit der Auswertung des Rumpfes der Funktion begonnen;
wird der Wert eines Parameters benötigt, so wird der zugehörige Ausdruck ausgewertet.

Vorteil: Ein Parameter wird nur ausgewertet, wenn sein Wert tatsächlich benötigt wird.

Nachteil: Wird der Wert eines Parameters mehrmals benötigt, wird er mehrfach ausgewertet.

Parameterübergabe: Formelaufruf mit Ergebnisteilhabe / Call-By-Need

Ein Parameter wird nur ausgewertet, wenn sein Wert benötigt wird, und dann nur einmal. Der erste Zugriff erzwingt die Auswertung des Parameters; alle weiteren Zugriffe nutzen den bereits berechneten Wert.

Vorteil: Call-By-Need vereint das gute Terminierungsverhalten von Call-By-Name mit der Effizienz von Call-By-Value.

Nachteil: Verwaltungsaufwand:
Wie ist erkennbar, ob ein Parameter bereits ausgewertet wurde?
Benötigte nicht-lokale Werte müssen bei verzögerter Auswertung gespeichert werden.

Namen: definierende und angewandte Vorkommen

In einem Programm kann ein **Name** mehrmals vorkommen, sogar mehrfach deklariert werden.

Wie finden wir zu einem Namen die richtige Definition?

Wird ein Name in einer Deklaration definiert oder in einer formalen Parameterliste spezifiziert, handelt es sich
um ein **definierendes Vorkommen** (*binding occurrence*),
sonst um ein **angewandtes Vorkommen** (*applied occurrence*).

Funktionen: lokale (gebundene) Namen

Namen, die als formale Parameter oder durch lokale Deklarationen eingeführt wurden, nennen wir **lokal** oder **gebunden**.

Beim Aufruf einer Funktion wird für jeden lokalen Namen eine neue **Inkarnation** erzeugt. Dazu wird Platz für Variablen bereitgestellt.
Die formalen Parameter werden mit den aktuellen Parametern initialisiert.

Die **Lebensdauer** der lokalen Variablen ist die der Funktionsinkarnation.
Ihr Platz kann daher bei Verlassen der Funktion wieder freigegeben werden.
Dies kann durch eine kellerartige Speicherverwaltung geschehen.

Derselbe Speicherbereich, der bei Betreten der Funktion für die formalen Parameter, die lokalen Variablen und für anfallende Zwischenergebnisse bereitgestellt wurde, wird bei Verlassen der Funktion wieder freigegeben.

Funktionen: globale (freie) Namen

Nicht ganz so einfach ist die Behandlung von angewandt auftretenden Namen, die nicht lokal sind. Wir bezeichnen diese Namen als **global** oder **frei** – relativ zur betrachteten Funktion.

Die **Sichtbarkeits**- und/oder **Gültigkeitsregeln** der Programmiersprache legen fest, wie das zu einem angewandten Vorkommen eines Namens gehörende definierende Vorkommen gefunden wird.

Die umgekehrte, aber äquivalente Sicht geht von einem definierenden Vorkommen eines Namens aus und legt fest, in welchem Programmstück alle angewandten Vorkommen des Namens sich auf dieses definierende Vorkommen beziehen.

Funktionen: Sichtbarkeitsregel

Sichtbarkeitsregel in Algol-ähnlichen Sprachen:

Ein definierendes Vorkommen eines Namens ist sichtbar
in der Programmeinheit, in der die Definition steht,
ohne innere Programmeinheiten, die denselben Namen neu definieren.

Dabei steht „Programmeinheit“ für eine Funktion oder einen **Block**.

Suchen wir zu einem angewandten Vorkommen das zugehörige definierende Vorkommen, so

- beginnen wir mit der Suche in der Programmeinheit, in der das angewandte Vorkommen steht,
- setzen ggf. die Suche in der direkt umfassenden Programmeinheit fort.
- Findet sich in allen umfassenden Programmeinheiten kein definierendes Vorkommen, so liegt ein Programmierfehler vor.

Funktionen: statische und dynamische Bindung

Durch diese Sichtbarkeitsregel erhält man die **statische Bindung** (*static scoping*), d.h. globale Namen werden definierenden Vorkommen in textlich umgebenden Programmeinheiten zugeordnet.
Diese Zuordnung beruht nur auf dem Programmtext und ist daher statisch.
Jede Benutzung eines globalen Namens zur Ausführungszeit betrifft eine Inkarnation des statisch zugeordneten definierenden Vorkommens.

Bei **dynamischer Bindung** (*dynamic scoping*) betrifft Zugriff auf einen globalen Namen die zuletzt angelegte Inkarnation dieses Namens – unabhängig davon, in welcher Funktion er definierend auftrat.

Statische Bindung wird von allen Algol-ähnlichen Sprachen und modernen funktionalen Sprachen wie Haskell oder OCaml vorgeschrieben, während ältere Lisp-Dialekte dynamische Bindung benutzen.

Vergleich: statische und dynamische Bindung

```
int x ← 1;

void q() {
    printf("%d", x);
}

int main() {
    int x ← 2;
    q();
}
```

Bei statischer Bindung bezieht sich das angewandte Vorkommen der Variablen x in der Funktion q auf die globale Variable x und liefert deshalb den Wert 1.

Das dynamisch zuletzt angelegte Vorkommen einer Variablen x vor dem Aufruf der Funktion q in der Funktion *main* ist dagegen die zu *main* lokale Variable x. Bei dynamischer Bindung liefert das angewandte Vorkommen von x in q deshalb den Wert 2.

keine geschachtelten Funktionsdefinitionen in C

Die Programmiersprache ANSI-C verbietet –im Gegensatz zu Pascal– geschachtelte Funktionsdefinitionen.

Diese Entwurfsentscheidung vereinfacht die Verwaltung der Sichtbarkeitsbereiche erheblich; der **statische Vorgänger** einer Funktion ist hier immer das Hauptprogramm.

Für ANSI-C genügt es, zwei Arten von Variablen zu unterscheiden:

- **globale Variable** (außerhalb der Funktionsdefinitionen deklariert) und
- **lokale Variable** (lokal zu einzelnen Funktionen definiert).

Speicherorganisation für Funktionen

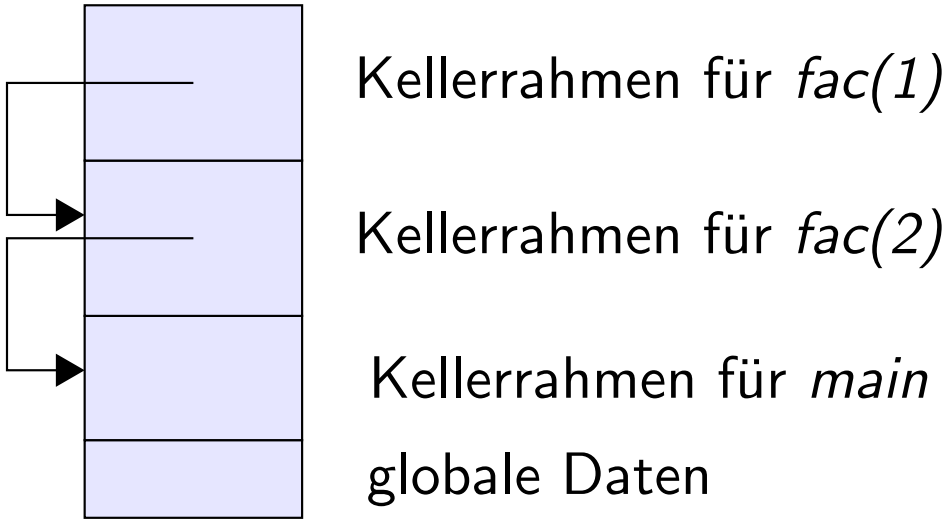
Eine einfache Implementierung ersetzt jeden Funktionsaufruf durch den passenden Code für den Funktionsrumpf (*open subroutine*).

Ein raffinierteres Vorgehen erzeugt nur *einmal* Code für eine Funktion (*closed subroutine*).
Dann benutzen zwar alle Inkarnationen einer Funktion denselben Code; jede Inkarnation hat aber i.d.R. unterschiedliche Daten
(auch wenn Anzahl und Typen der Daten für jede Inkarnation gleich sind) und benötigt daher eigenen Datenspeicherbereich.
Diese Variante ermöglicht auch rekursive Funktionsaufrufe.

Dieser Datenbereich einer Inkarnation wird im Keller angelegt und heißt **Kellerrahmen** (*stack frame*).

Speicherorganisation für Funktionen: Laufzeitkeller

Der **Laufzeitkeller**, die Speicherorganisation für Funktionen, enthält für jede aktuell lebende Inkarnation einen Speicherbereich, und zwar in der gleichen Reihenfolge, in der sie auf dem Inkarnationsweg auftreten.

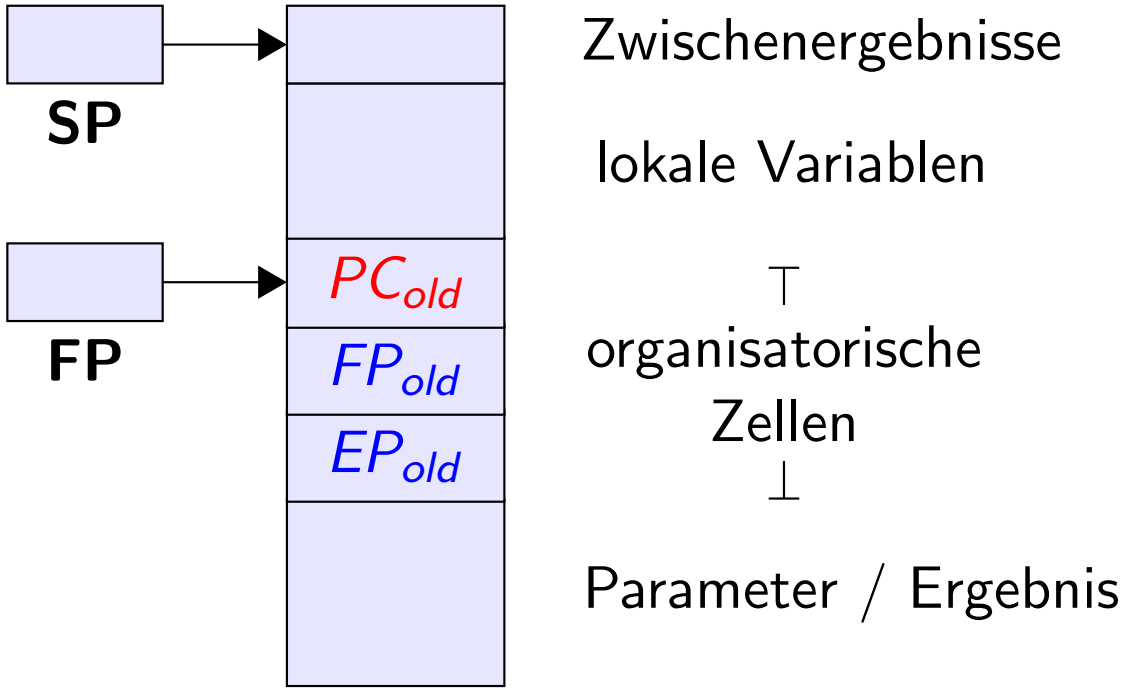


Der Laufzeitkeller der C-Maschine.

Um eine Funktion effizient verlassen zu können, sind die Inkarnationen so verkettet, dass jede auf ihren **dynamischen Vorgänger** zeigt, also auf die Inkarnation, von der sie durch einen Aufruf angelegt wurde.

Rahmenzeiger zur Adressierung von Parametern und lokalen Variablen

Ein Register, der **Rahmenzeiger FP** (*frame pointer*), zeigt stets auf eine bestimmte Stelle des aktuellen Kellerrahmens und wird zur Adressierung von Parametern und lokalen Variablen benutzt.



Der Kellerrahmen einer Funktionsinkarnation.

Funktionen: organisatorische Zellen

Der Kellerrahmen enthält auch Platz für die Register, deren Inhalt bei Betreten einer Funktion gerettet wird, weil er bei Verlassen der Funktion wiederhergestellt werden muss.
Diese Zellen heißen **organisatorische Zellen**, weil sie eine korrekte und effiziente Organisation von Funktionsanfang und -ende ermöglichen.

In der C-Maschine sind dies die drei Register PC, FP und EP.

- Der gerettete Inhalt des PC ist die **Rücksprungadresse**, an der die Berechnung nach Beendigung des Funktionsaufrufs fortfahren soll.
- Der gerettete Inhalt des FP verweist auf die Daten der aufrufenden Funktion, genauer auf den Kellerrahmen des dynamischen Vorgängers des aktuellen Funktionsaufrufs.
- Vorsorglich retten wir auch den Inhalt des EP.

Funktionen: lokale Variablen

Oberhalb der organisatorischen Zellen beginnt der Datenbereich der aktuellen Funktionsinkarnation.
Hier legen wir konsekutiv die lokalen Variablen der aktuellen Funktion an.
Dadurch können wir auf diese Variablen mit **festen Relativadressen** relativ zum Rahmenzeiger FP zugreifen.
Zeigt der Rahmenzeiger auf die oberste organisatorische Zelle, können für die lokalen Variablen Relativadressen ab 1 vergeben werden.

Oberhalb des Datenbereichs legen wir den **lokalen Keller**, das ist der Keller, der uns bei der Auswertung von Ausdrücken begegnet ist.
Seine maximale Länge ist statisch bestimmbar.

Konzeptuell müssen noch zwei Dinge im Kellerrahmen untergebracht werden: die aktuellen Parameter und der Rückgabewert der Funktion.