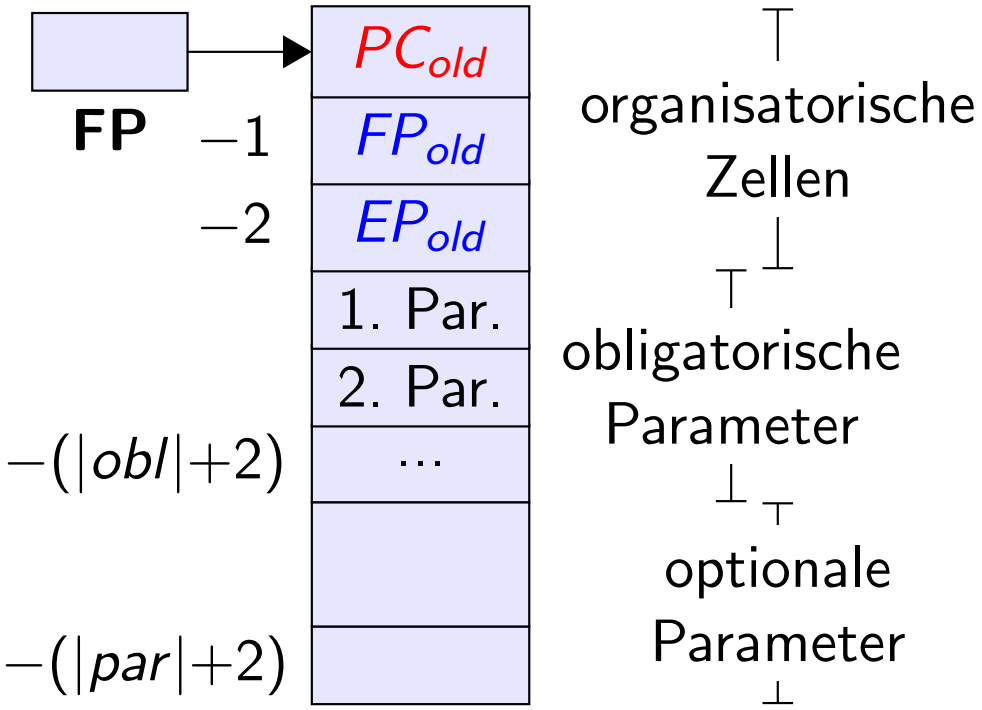
Funktionen: optionale aktuelle Parameter

Mit den **aktuellen Parametern** haben wir ein Problem:
Die Sprache C erlaubt Funktionen mit **variablen Parameterlisten**,
z.B. ist bei der Funktion *printf* nur der erste Parameter **obligatorisch**,
alle weiteren aktuellen Parameter sind **optional**,
ihre Anzahl kann also von Funktionsaufruf zu Aufruf variieren.

Um trotzdem feste Relativadressen für lokale Variablen vergeben zu können, nutzen wir einen Trick:
die aktuellen Parameter werden *unterhalb* der organisatorischen Zellen und *in umgekehrter Reihenfolge* auf den Keller gelegt!
Der erste Parameter liegt also oberhalb des zweiten usw.

Funktionen: Parameteranordnung



Parameter im Kellerrahmen.

Als Relativadressen stehen die negativen Zahlen ab -3 zur Verfügung. Hat der Typ des ersten Parameters z.B. die Größe $|t|$, bekommt er die Relativadresse $-(|t|+2)$.

Funktionen: Rückgabewert

Liefert die Funktion einen **Rückgabewert** zurück, sollten wir dafür einen Standardplatz vorsehen, auf den sich mit einer festen Relativadresse relativ zum FP zugreifen lässt.

Wir könnten dazu einen Extrabereich unterhalb der organisatorischen Zellen wählen.

Da der Platz für die aktuellen Parameter nicht mehr benötigt wird, werden wir diesen Bereich zum Ablegen des Rückgabewerts wiederverwenden.

Übersetzung von lokalen und globalen Variablen

Wir haben deklarierten globalen Variablen bereits Adressen zugeordnet.
Dabei griffen wir über den Kellerzeiger SP auf Variablen zu.

Bei Parametern und lokalen Variablen wollen wir auf Instanzen im aktuellen Funktionsaufruf zugreifen.
Die Adressierung erfolgt daher relativ zum Rahmenzeiger FP.

Die Adressumgebung ρ soll jetzt außer der Relativadresse auch die Information verwalten, ob der Name global oder lokal ist:

$$\rho : \mathit{Names} \to \{G, L\} \times \mathbb{Z},$$

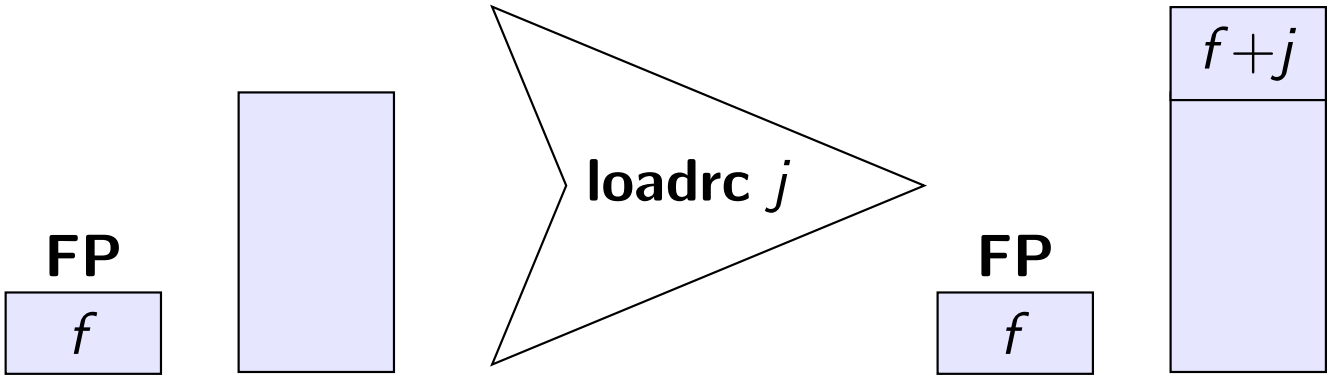
die Etiketten G und L bezeichnen globale bzw. lokale Gültigkeit.

Um Zugriffe auch auf lokale Variablen oder Parameter zu ermöglichen, wird die Übersetzungsfunktion code_A verallgemeinert:

$$\text{code}_A^\rho\ x = \begin{cases} \textbf{loadc}\ j & \text{, falls } \rho(x) = (G, j) \\ \textbf{loadrc}\ j & \text{, falls } \rho(x) = (L, j) \end{cases}$$

Übersetzung von lokalen und globalen Variablen

Die neuen Befehle **loadrc** j laden den Wert $FP+j$ oben auf den Keller.



$SP++; \quad S[SP] \leftarrow FP + j;$

Der Befehl **loadrc** j.

Natürlich können wir wieder Spezialbefehle einführen:

loadr $j \equiv$ **loadrc** j, **load**
storer $j \equiv$ **loadrc** j, **store**
wobei wir die Befehle zu **loadr** j m und **storer** j m erweitern, falls m Speicherzellen bewegt werden sollen.

Übersetzung von lokalen und globalen Variablen

Mit dieser Änderung können wir die bisherige Codeerzeugung auch auf die Rümpfe von Funktionen anwenden.

Ein Name kann auch eine Funktion bezeichnen.
Auch Funktionsnamen wollen wir eine Adresse zuordnen, nämlich die Anfangsadresse ihres Codes im *Programmspeicher* C.
Diese Anfangsadresse des Codes einer Funktion f beschreiben wir durch die Marke $\_f$.

Wir benötigen noch eine systematische Methode zur **Berechnung der Adressumgebung** ρ.
Diese Methode muss dafür sorgen, dass an jeder Anwendungsstelle die Adressumgebung genau die dort sichtbaren Namen mit der jeweils aktuellen Information zeigt.

Berechnung der Adressumgebung

Für eine Variablendeklaration $t\ x;$ definieren wir Funktionen, die für ein Paar (ρ, n) aus Adressumgebung ρ und erster freier Relativadresse n eine neue Adressumgebung und die nächste freie Relativadresse liefern.

Für eine Deklaration

- einer globalen Variablen:
$$\textbf{elab\_global}(\rho, n)\ (t\ x) = (\rho \oplus \{x \mapsto (G, n)\},\ n + |t|)$$
- einer lokalen Variablen:
$$\textbf{elab\_local}(\rho, n)\ (t\ x) = (\rho \oplus \{x \mapsto (L, n)\},\ n + |t|)$$
- eines formalen Parameters:
$$\textbf{elab\_formal}(\rho, z)\ (t\ x) = (\rho \oplus \{x \mapsto (L, z - |t|)\},\ z - |t|)$$

$\rho \oplus \{x \mapsto a\}$ bezeichnet die (partielle) Funktion, die für das Argument x den Eintrag a zu ρ hinzufügt bzw. einen alten Eintrag für x mit dem neuen Wert a überschreibt.

Berechnung der Adressumgebung

Bei der Funktion **elab_formal** erhält jeder weitere Parameter eine *kleinere* Adresse. Hier übergeben wir die unterste bisher belegte Adresse z im Kellerrahmen anstelle der ersten freien Relativadresse.

Mehrere Deklarationen arbeiten wir durch wiederholte Anwendung dieser Funktionen ab (formal: **elab_globals** etc.).

Berechnung der Adressumgebung

Sei f eine Funktion ohne Rückgabewert mit
einer Spezifikation sp formaler Parameter und
einer Deklaration ll lokaler Variablen.
Sei $\_f$ die Anfangsadresse des Codes von f.

Aus einer Adressumgebung ρ für globale Namen erhalten wir dann die Adressumgebung ρ_f für die Funktion f:

$$\begin{aligned}\rho_f = \quad & \textbf{let } \rho' = \rho \oplus \{f \mapsto (G, \_f)\} \\ & \textbf{in let } (\rho'', \_) = \textbf{elab\_formals}(\rho', -2)\ sp \\ & \textbf{in let } (\rho''', \_) = \textbf{elab\_locals}(\rho'', 1)\ ll \\ & \textbf{in } \rho'''\end{aligned}$$

Berechnung der Adresse des Rückgabewertes

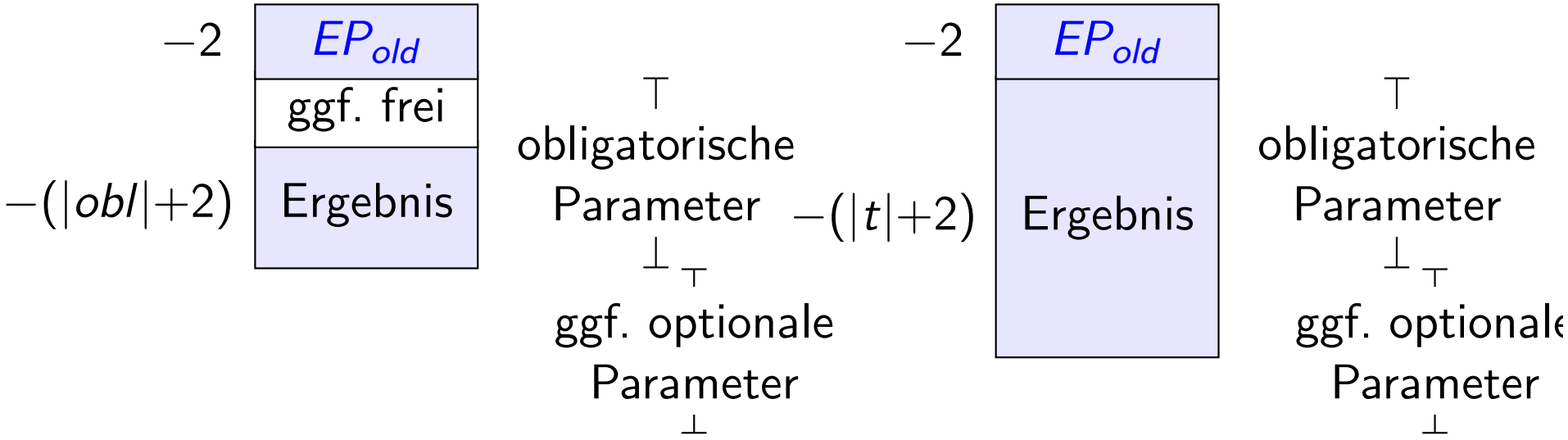
Hat die Funktion f einen Rückgabewert, verwalten wir dessen Relativadresse unter dem lokalen Namen *ret* ebenfalls in der Adressumgebung ρ_f für f.

Sei t der Typ des Rückgabewerts und
$|obl|$ der Platzbedarf der obligatorischen Parameter.

- Ist $|t| \leq |obl|$, können wir den Rückgabewert *am unteren Rand* des Blocks für die obligatorischen Parameter ablegen,
  also ab Relativadresse $-(|obl|+2)$.
- Ist $|t| > |obl|$, benötigen wir für die Rückgabe evtl. einen größeren Block als für die Parameter.
  Dann legen wir den Rückgabewert ab Adresse $-(|t|+2)$ ab.

Wir ergänzen dann unsere Definition von ρ_f um die Bindung
$ret \mapsto (L, -(max(|obl|, |t|)+2))$.

Platzierung des Rückgabewerts



Rückgabewert mit $|t| \leq |obl|$.

Rückgabewert mit $|t| > |obl|$.

Beispielprogramm

Betrachten wir unser Beispielprogramm:

```
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);
}
```

Berechnung der Adressumgebung

Unser Beispielprogramm hat folgende Adressumgebungen:

- Da wir keine globale Variablen haben,
  ist die Adressumgebung vor der Deklaration der Funktion *fac*: $\rho_0 = \emptyset$
- Wir erhalten die Adressumgebung ρ_{fac},
  indem wir zuerst die Bindung $fac \mapsto (G, \_fac)$ zu ρ_0 hinzufügen.
  Der formale Parameter n bekommt die Bindung $n \mapsto (L, -3)$.
  Die Relativadresse des Rückgabewerts ist -3.
  $\rho_{fac} = \{fac \mapsto (G, \_fac), n \mapsto (L, -3), ret \mapsto (L, -3)\}$
- Vor der Definition der Funktion *main* ist der Name der Funktion *fac* bereits bekannt: $\rho_1 = \{fac \mapsto (G, \_fac)\}$
- Die Funktion *main* hat keine Parameter, aber einen Rückgabewert der Größe 1. Deshalb erhält dieser die Relativadresse -3.
  ρ_{main} enthält eine Bindung für die lokale Variable n sowie eine Bindung für den globalen Funktionsnamen *main*.
  $\rho_{main} =$
  $\{fac \mapsto (G, \_fac), main \mapsto (G, \_main), n \mapsto (L, 1), ret \mapsto (L, -3)\}$

Betreten von Funktionen

Betrachten wir den **Aufruf** einer Funktion.

Sei f die gegenwärtig aktive Funktion.
Der Kellerrahmen von f ist daher der oberste auf dem Keller.
Jetzt rufe die Funktion f eine Funktion g auf.

Wir wollen für den Aufruf von g eine Befehlsfolge erzeugen, die den Aufruf auswertet und den Rückgabewert oben auf dem Keller hinterlässt.

Betreten von Funktionen

Folgende Aktionen müssen bei **Aufruf der Funktion** g ausgeführt werden:

- B1 Platz für das Funktionsergebnis reservieren, falls der Platz für die obligatorischen Parameter nicht ausreicht
- B2 Werte der aktuellen Parameter ermitteln und auf den Keller schreiben
- B3 Alte Werte der Register EP und FP auf den Keller retten
- B4 Anfangsadresse der Funktion g berechnen
- B5 Rücksprungadresse ermitteln und auf den Keller retten
- B6 Register FP auf den aktuellen Kellerrahmen setzen
- B7 Anfangsadresse von g anspringen

- B8 Platz für lokale Variablen von g reservieren
- B9 Register EP auf den Wert für die neue Inkarnation setzen

Danach können die Anweisungen des Rumpfes von g ausgeführt werden.

Betreten von Funktionen

Die Codeerzeugung muss diese Aktionen nun auf den **Aufrufer** (*caller*) f und den **Aufgerufenen** (*callee*) g verteilen.

So kann z.B. nur die

- aufrufende Funktion f die Werte der aktuellen Parameter ermitteln, und nur die
- aufgerufene Funktion g kennt den Platzbedarf für ihre lokalen Variablen.

In unserer Liste verläuft die Grenze zwischen dem Code der aufrufenden und dem Code der aufgerufenen Funktion zwischen Punkt (B7) und (B8).

Verlassen von Funktionen

Eine Funktion wird verlassen nach Bearbeitung

- einer C-Anweisung **return** oder
- der textuell letzten Anweisung der Funktion.

Beim **Verlassen einer Funktion** sind folgende Aufgaben auszuführen:

V1 Rückgabewert –falls vorhanden abspeichern
V2 Register EP und FP wiederherstellen
V3 Keller aufräumen: Speicher oberhalb des Rückgabewerts freigeben
V4 zur Rückkehradresse des Aufrufers f zurückspringen.

V5 Keller aufräumen: Rückgabewert über Bereich der optionalen Parameter verschieben

(V1) bis (V4) können von der aufgerufenen Funktion g ausgeführt werden.

(V5) muss von der aufrufenden Funktion f ausgeführt werden, da sich die Zahl der optionalen Parameter von Aufruf zu Aufruf unterscheiden kann. (V5) wird daher unmittelbar nach dem Rücksprung zur aufrufenden Funktion f erledigt.

Betreten und Verlassen von Funktionen

Insgesamt erhalten wir folgendes Schema:

Code für Funktionsdefinition von f:

$$\underbrace{\ldots}_{\text{Code von } f_{\text{vor Aufruf}}}, B_1, \ldots, B_7, V_5, \underbrace{\ldots}_{\text{Code von } f_{\text{nach Aufruf}}}$$

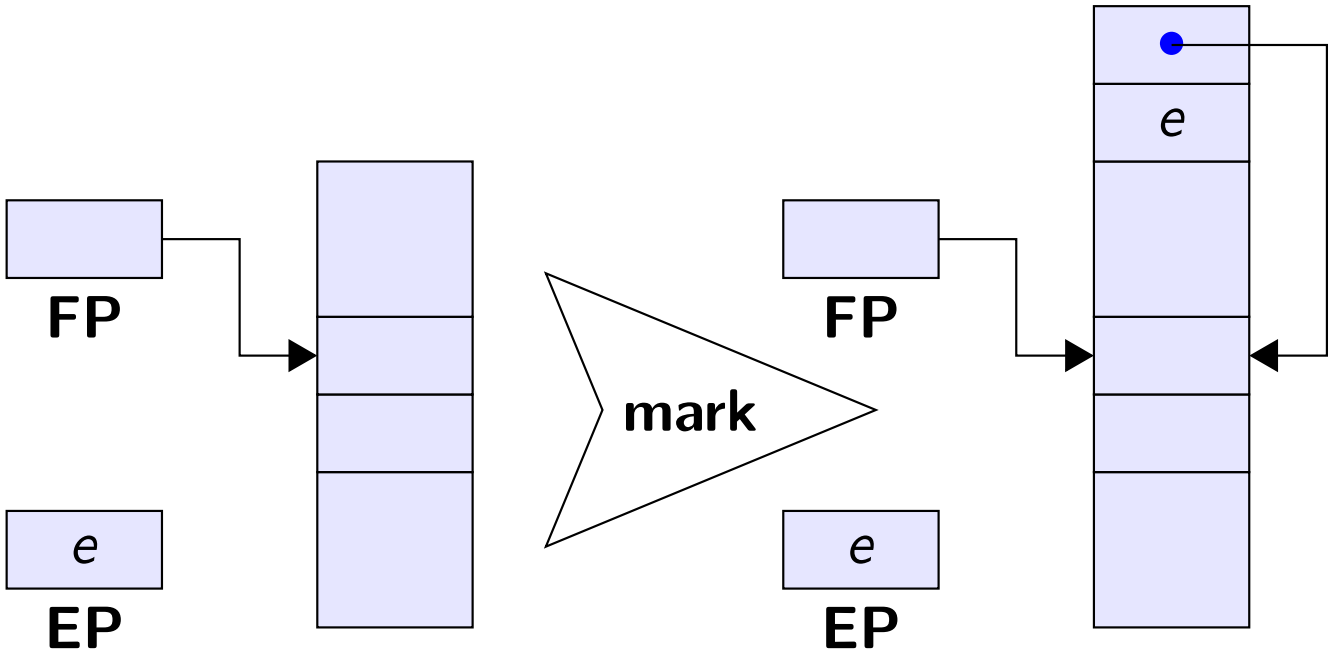
Code für Funktionsdefinition von g:

$$B_8, B_9, \underbrace{\ldots}_{\text{Code für Rumpf von } g}, V_1, \ldots, V_4$$

Übersetzung eines Funktionsaufrufs

Falls der Platz der obligatorischen Parameter für den Rückgabewert nicht ausreicht (B1), reservieren wir zusätzlichen Platz mit dem Befehl **alloc** q.

Um die Register EP und FP zu retten (B3), benutzen wir den neuen Befehl **mark**, der den Inhalt dieser Register oben auf den Keller legt.

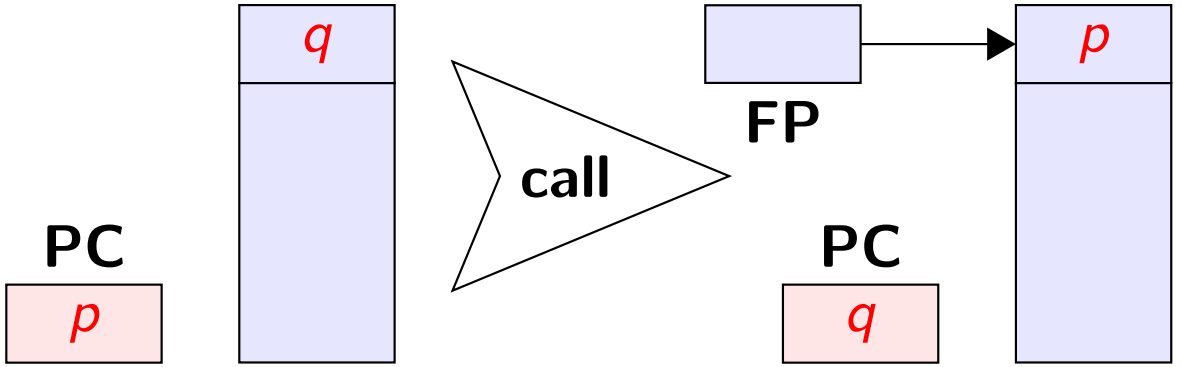


$S[SP+1] \leftarrow EP; \quad S[SP+2] \leftarrow FP; \quad SP \leftarrow SP+2;$

Der Befehl **mark**.

Übersetzung eines Funktionsaufrufs

Um die aufgerufene Funktion anzuspringen (B7), müssen wir die Anfangsadresse von g, die oben auf dem Keller steht, in das Register PC laden. Dazu benötigen wir den neuen Befehl **call**, der als letzter Befehl vor Betreten von g im Aufrufer f steht.
Bei Ausführung dieses Befehls enthält PC gerade die Rückkehradresse!
Daher kann **call** die Aufgabe (B5) miterledigen, indem er die Zieladresse oben auf dem Keller durch den aktuellen PC ersetzt.
Außerdem wird das Register FP auf den aktuellen Wert (die oberste belegte Kellerzelle) gesetzt (B6).



$FP \leftarrow SP$; **var** $tmp \leftarrow PC$; $PC \leftarrow S[SP]$; $S[SP] \leftarrow tmp$;

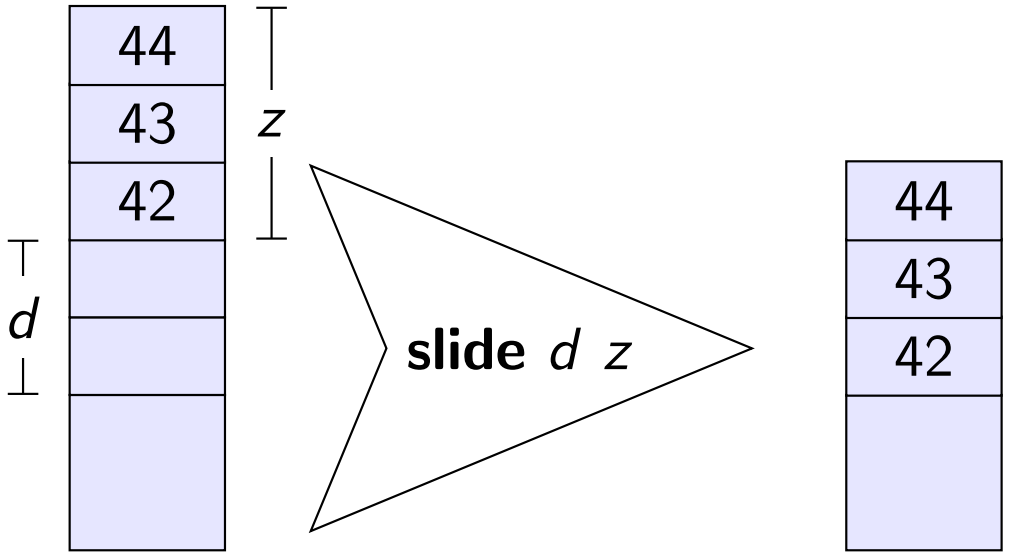
Der Befehl **call**.

Übersetzung eines Funktionsaufrufs

Liegt der Rückgabewert nicht am unteren Rand des Kellerrahmens, weil optionale Parameter vorhanden sind, so verschieben wir ihn (nach dem Rücksprung) mit dem Befehl **slide** entsprechend nach unten (V5).

Hat eine Funktion keine optionalen Parameter ($|obl| = |par|$), so ist das Argument d des **slide**-Befehls immer 0, so dass ein Optimierer diesen Befehl streichen kann.

Übersetzung eines Funktionsaufrufs



if $d > 0$ **then**
  if $z = 0$ **then** $SP \leftarrow SP - d$;
  else {
    $SP \leftarrow SP - d - z$;
    for $i \leftarrow 0$; $i < z$; $i{+}{+}$ **do** {
      $SP{+}{+}$; $S[SP] \leftarrow S[SP + d]$;
    }
  }

Der Befehl **slide** d z.

Übersetzung eines Funktionsaufrufs

Übersetzungsschema für einen Funktionsaufruf $e \equiv g(e_1, \ldots, e_n)$:

Sei t der Ergebnistyp von g,
$|obl|$ der Platzbedarf für die obligatorischen Parameter und
$|par| \geq |obl|$ der Platzbedarf für alle Parameter. Dann:

$$\text{code}_W^\rho\ g(e_1, \ldots, e_n) =$$
$$\quad \textbf{alloc}\ q,\ \text{code}_W^\rho\ e_n, \ldots, \text{code}_W^\rho\ e_1,\ \textbf{mark}, \text{code}_W^\rho\ g, \textbf{call}, \textbf{slide}\ d\ |t|$$

mit
$q = max(|t| - |obl|, 0)$,
$d = \textbf{if}\ |t| \leq |obl|\ \textbf{then}\ |par| - |obl|\ \textbf{else}\ max(|par| - |t|, 0)$

Das Schema erzeugt für jeden aktuellen Parameter e_i Code, der den Wert von e_i in der Adressumgebung ρ der aufrufenden Funktion berechnet.
Dies entspricht einer Parameterübergabe **by value**.

Betreten von Funktionen: Call By Reference

Andere imperative Programmiersprachen wie Pascal oder C++ unterstützen auch eine Übergabe der Parameter **by reference**.

In diesem Fall soll die Adresse des aktuellen Parameters am Platz des zugehörigen formalen Parameters x abgelegt werden.

Jeder Zugriff auf den formalen Parameter x im Rumpf der Funktion erfordert dann eine **Indirektion** über seine Relativadresse.

Für einen Referenzparameter x, dem wir das Etikett R zuordnen, berechnen wir seine Adresse deshalb als:

$$\text{code}_A^\rho\ x = \textbf{loadr}\ j \qquad \text{, falls } \rho(x) = (R, j).$$

Übersetzung eines Funktionsaufrufs

In C wird für den Ausdruck g, der die Anfangsadresse der aufzurufenden Funktion liefern soll, ebenfalls Code zur Berechnung des *Wertes* erzeugt. Dies erlaubt den Aufruf einer Funktion über Funktionszeiger.

Einfache Funktionsnamen dagegen werden in C als **Referenzen** aufgefasst, deren Wert gleich der Adresse ist:
$\text{code}_W^\rho\ f = \text{code}_A^\rho\ f = \textbf{loadc}\ \rho(f) = \textbf{loadc}\ \_f$

Hier bietet sich ein Spezialbefehl an: **calld** $a \equiv$ **mark**, **loadc** a, **call**

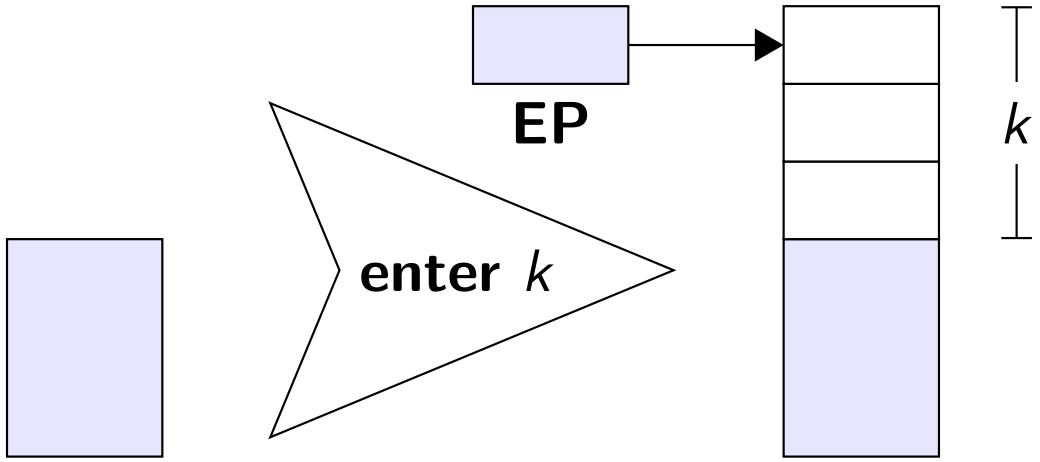
Betrachten wir den rekursiven Funktionsaufruf $fac(n-1)$ im Beispielprogramm. Das Übersetzungsschema liefert die Befehlsfolge:
alloc 0, **loadr** -3, **loadc** 1, **sub**, $\underbrace{\textbf{mark}, \textbf{loadc}\ \_fac, \textbf{call}}_{\textbf{calld}\ \_fac}$, **slide** 0 1

Übersetzen einer Funktionsdefinition

Zu Beginn erzeugen wir Code für die Schritte B8 und B9:
B8: Mit einem **alloc**-Befehl reservieren wir Platz für die lokalen Variablen.

B9: Den neuen EP setzen wir mit dem Befehl **enter** k, wobei k der maximale Platzbedarf für Zwischenergebnisse ist.

Der Befehl überprüft auch, ob auf dem Keller noch genügend Platz zur Abarbeitung des aktuellen Funktionsaufrufs vorhanden ist.
testoverflow steht für **if** $EP \geq maxS$ **then error**(„Stack Overflow“);



$EP \leftarrow SP + k$; **testoverflow**

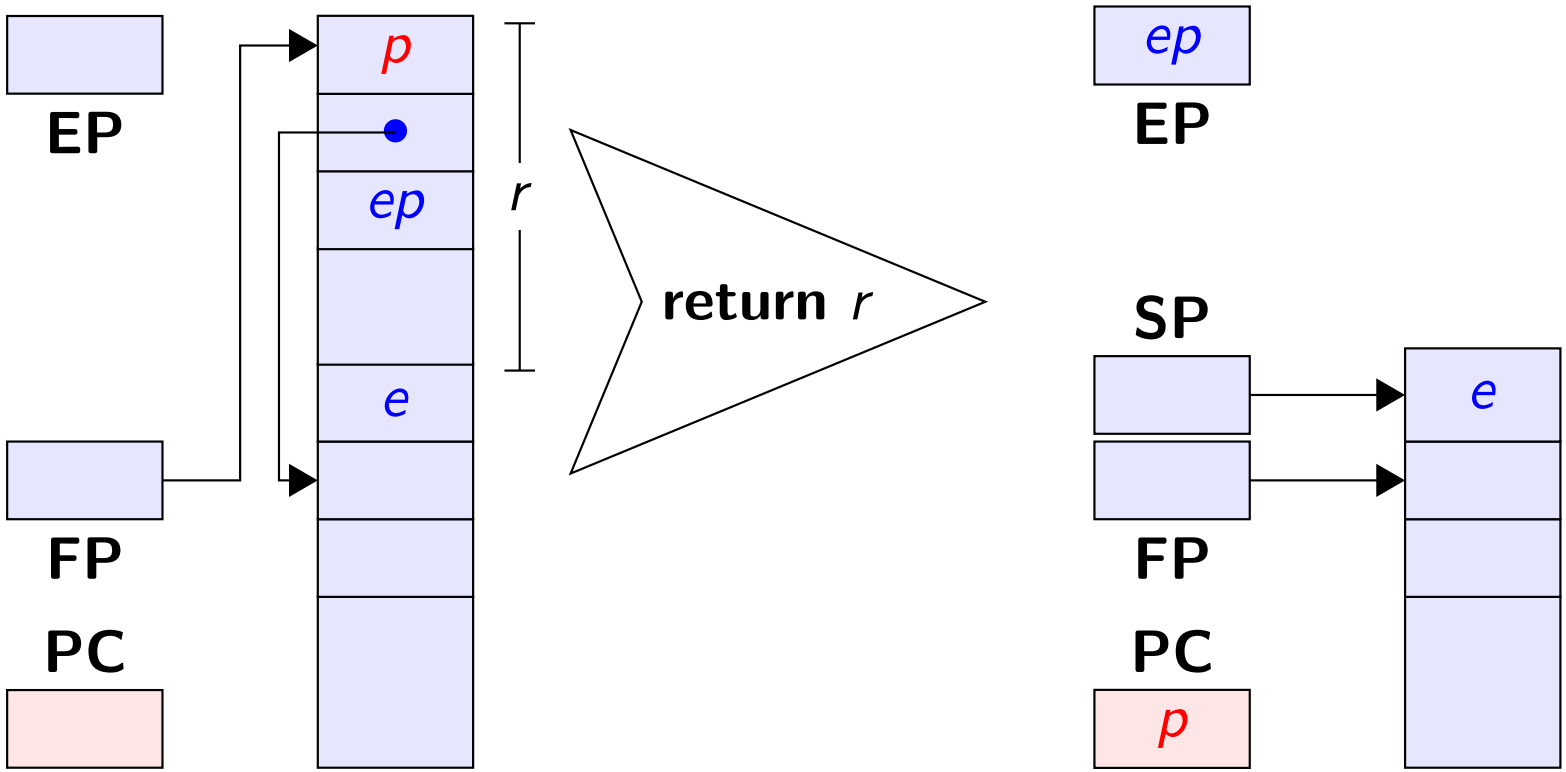
Der Befehl **enter** k.

Übersetzen einer Funktionsdefinition

Die Aufgabe (V1) wird wie eine Wertzuweisung behandelt, da wir die Anfangsadresse eines Rückgabewerts in der Adressumgebung verwalten.

Die Aufgaben (V2)–(V4) fassen wir im Befehl **return** *r* zusammen.
Es werden *r* Zellen oberhalb des Rückgabewerts freigegeben.
(Die Zellen unterhalb des Rückgabewerts gibt der **slide**-Befehl frei.)
Es wird überprüft, ob auf dem Keller noch genügend Platz für die aufrufende Funktion frei ist.

Übersetzen einer Funktionsdefinition



$PC \leftarrow S[FP]; \quad EP \leftarrow S[FP-2];$ **testoverflow**
$SP \leftarrow FP-r; \quad FP \leftarrow S[FP-1];$

Der Befehl **return** r.

Übersetzen einer Funktionsdefinition

Übersetzung der C-Anweisungen **return**; bzw. **return** e;

$\text{code}^{\rho}(\textbf{return};) = \textbf{return}\ (|obl|+3)$

$\text{code}^{\rho}(\textbf{return}\ e;) = \text{code}^{\rho}_{W}\ e,\ \textbf{storer}\ \rho(ret)\ |t|,\ \textbf{return}\ r$

wobei e ein Ausdruck vom Typ t ist,
$|obl|$ der Platzbedarf der Funktion für ihre obligatorischen Parameter und
$r = 3 + max\{|obl|-|t|, 0\}$.

Übersetzen einer Funktionsdefinition

Insgesamt übersetzen wir die **Definition einer Funktion** g mit Rückgabetyp t, Deklaration formaler Parameter *params*, Deklaration lokaler Variablen *locals* und Anweisungsfolge *ss* wie folgt:

$$\text{code}^{\rho}\ (t\ g\ (params)\ \{locals\ ss\}) = \\ \_g\text{: } \textbf{alloc}\ l, \textbf{enter}\ k, \text{code}^{\rho_g}\ ss, \textbf{return}\ r$$

wobei

- k der max. Platzbedarf für Zwischenergebnisse der Funktion g ist;
- l ist die Anzahl Speicherzellen für die lokalen Variablen,
- ρ_g ist die Adressumgebung für g, die man aus ρ, *params* und *locals* sowie der Größe des Rückgabetyps t erhält, und
- $r = 3 + max\{|obl| - |t|, 0\}$, falls $|obl|$ der Platzbedarf für die obligatorischen Parameter ist.

Übersetzen einer Funktionsdefinition

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

Insgesamt übersetzen wir dann den Rumpf der Funktion *fac* so:

_*fac*: **alloc** 0, **enter** 3, **loadr** -3, **loadc** 0, **leq**, **jumpz** A, **loadc** 1, **storer** -3, **return** 3, **jump** B, A: **loadr** -3, **alloc** 0, **loadr** -3, **loadc** 1, **sub**, **mark**, **loadc** _*fac*, **call**, **slide** 0 1, **mul**, **storer** -3, **return** 3, B: **return** 3

Natürlich kann man die Befehle **alloc** 0 und **slide** 0 1 weglassen.
Außerdem ist der Befehl **jump** B nicht erreichbar und kann entfallen.
Auch das letzte **return** 3 wird nie erreicht.

Übersetzung eines ganzen Programms

Als letztes beschreiben wir, wie ein ganzes C-Programm p übersetzt wird.

Alle Register haben vor der Programmausführung den Wert 0.
Die Ausführung des Programms beginnt also mit dem Befehl in C[0].

Das Programm p ist eine Folge von Deklarationen von globalen Variablen und Funktionen, von denen eine die Funktion **int** *main*() definiert.

Der Code für das Programm p enthält damit:

1. Code zum Anlegen der globalen Variablen;
2. Code für die Funktionsdefinitionen;
3. Code für den Aufruf der Funktion *main*;
4. den Befehl **halt**, um die Programmausführung zu beenden.

Übersetzung eines ganzen Programms

Alle Variablendeklarationen mögen vor den Funktionsdeklarationen stehen.
Das Programm p is also von der Form:
$p \equiv dd\ df_1\ \ldots\ df_n$
wobei dd eine Folge von Variablendeklarationen ist und
$df_1, \ldots, df_n$ die Deklarationen der Funktionen $f_1, \ldots, f_n \equiv main$ sind.

Dann ergibt sich das Paar aus der Adressumgebung für die globalen Variablen von p und der ersten freien Relativadresse aus:

$(\rho_0, k) = \textbf{elab\_globals}\ (\emptyset, 1)\ dd$

$\emptyset$ steht für die leere Adressumgebung.

Die Adressumgebung vor Abarbeitung der i-ten Funktionsdefinition ist:
$\rho_i = \rho_{i-1} \oplus \{f_i \mapsto (G, \_f_i)\} \quad (i = 1, \ldots, n)$,
dabei ist $\_f_i$ die Anfangsadresse der Funktion f_i.

Übersetzung eines ganzen Programms

Dann erhalten wir als Übersetzungsschema für das Programm p:

$$\text{code } p = \textbf{alloc } k, \textbf{ enter } 3, \textbf{ calld } \_f_n, \textbf{ slide } (k-1)\ 1, \textbf{ halt},$$
$$\text{code}^{\rho_1}\ df_1, \ldots, \text{code}^{\rho_n}\ df_n$$

Vor dem Aufruf der Funktion *main* reservieren wir oberhalb der Speicherzelle mit Adresse 0 insgesamt $k-1$ Speicherzellen für globale Variablen sowie eine Zelle für den Rückgabewert von f_n.
Da SP vor der Ausführung den Wert 0 hat, müssen wir ihn um k erhöhen.
Bis zur Ausführung des ersten Befehls **call** werden drei weitere Zellen auf den Keller gelegt. Deshalb erhält der **enter**-Befehl das Argument 3.

Nach Abarbeitung des ersten Aufrufs liegt der Rückgabewert in der Speicherzelle mit Adresse k. Der Befehl **halt**, der ihn an das Betriebssystem zurückliefern soll, erwartet ihn aber in der Speicherzelle 1.
Deshalb müssen wir ihn nach dem Aufruf um $k-1$ Zellen nach unten verschieben. Das leistet der Befehl **slide** $(k-1)\ 1$.

Warnungen

Wir haben –zur besseren Verständlichkeit– stark vereinfacht.
Einige harte Realitäten wurden ignoriert.
Insbesondere muss die **Wortgröße** festgelegt werden.

Z.B. haben wir Konstanten immer als Operanden direkt im Befehl untergebracht, in einigen Befehlen sogar zwei.

Die zur C-Maschine verwandte P-Maschine für Pascal unterscheidet, ob ein konstanter Operand in den Befehl hineinpasst oder nicht. Zu große Konstanten werden in einer **Konstantentabelle** gespeichert, die oberhalb der Halde angelegt wird. In den Befehlen stehen dann nur Verweise in diese Tabelle. Es gibt dann verschiedene Befehle für eine Operation, je nachdem, ob die Operanden im Befehl oder in der Konstantentabelle stehen.

Teil IV

Zwischencode-Erzeugung: Datenabstraktion und objekt-orientierte Programmiersprachen

Datenverbunde und Datenfelder

Werden mehrere Daten als eine Einheit aufgefasst, so sprechen wir von **strukturierten Daten** oder einem **Datenverbund** (*record*, *struct*).

Sind alle Daten vom gleichen Typ, so können sie auch als **Datenfeld** (*array*) realisiert werden.

Speicherbelegung für Datenverbunde

Betrachten wir die Speicherbelegung und Adressierung bei **Verbunden** oder **Strukturen**.

Sei eine Verbundvariable x deklariert durch: **struct** t {**int** a; **int** b;} x;

Der Verbundvariablen x ordnen wir wieder die Adresse der ersten freien Speicherzelle zu.
Die **Komponenten** von x erhalten Adressen relativ zum Anfang der Struktur ($a \mapsto 0$, $b \mapsto 1$). Diese Relativadressen hängen nur vom Typ t ab.

Wir sammeln Relativadressen in der Funktion **offsets**, die Paaren (t, c) aus Verbundtyp und Komponente die zugehörige Relativadresse zuordnet.

Hat ein Verbundtyp t die Komponenten $c_1 \ldots c_k$ mit Typen $t_1 \ldots t_k$. Dann:
$\mathbf{offsets}(t, c_1) = 0$ und
$\mathbf{offsets}(t, c_i) = \mathbf{offsets}(t, c_{i-1}) + |t_{i-1}|$, für $i > 1$

Speicherbelegung für Datenverbunde

Natürlich können die Komponenten wieder zusammengesetzt sein.

Deshalb benötigen wir eine Hilfsfunktion (in C: **sizeof**), die uns für jeden Typ t seine Größe $|t|$ (die Anzahl der benötigten Speicherzellen) berechnet.

Die **Größe eines Verbundtyps** t ergibt sich als Summe der Größen seiner Komponenten t_i:

$$|t| = \sum_{i=1}^{k} |t_i|$$

Deklarierte Variable werden wir weiter konsekutiv im Keller ablegen.

Adresse einer Verbundkomponente

Die **Adresse einer Verbundkomponente** wird dann berechnet durch:

1. Laden der Anfangsadresse des Verbundes;
2. Erhöhung der Adresse um die Relativadresse der Komponente.

Sei e ein Ausdruck von einem Verbundtyp t mit Komponente c.
Dann wird zur Berechnung der Adresse von $e.c$ der folgende Code erzeugt:

$\text{code}_A^\rho\ (e.c) = \text{code}_A^\rho\ e$, **loadc** m, **add** , wobei $m = \textbf{offsets}(t, c)$.

Wert einer Verbundkomponente

Wie ermittelt man den **Wert einer Verbundkomponente**?
Für Komponenten, deren Typ zusammengesetzt ist, reicht das Laden des Inhalts der adressierten Zelle offenbar nicht aus.
Stattdessen muss ein ganzer Block oben auf den Keller geladen werden.

Das Übersetzungsschema, um den Wert eines Ausdrucks e der Größe $|t|$ zu berechnen, lautet daher:

$$\text{code}_W^\rho\ (e) = \text{code}_A^\rho\ e, \textbf{ load } |t|$$

Die Zuweisung eines Werts vom Verbundtyp t ergibt sich deshalb zu:

$$\text{code}_W^\rho\ (e_1 \leftarrow e_2) = \text{code}_W^\rho\ e_2, \text{code}_A^\rho\ e_1, \textbf{ store } |t|$$

Speicherbelegung für Datenfelder

Als nächstes wollen wir ein **Datenfeld** (*array*) übersetzen.

- In einem Feld haben alle Komponenten den gleichen Typ.
- Die Komponenten haben ganze Zahlen als Standardnamen (**Index**), die auch errechnet werden können.

Die Programmiersprache C stellt nur **statische Felder** zur Verfügung, d.h. die Anzahl der Feldkomponenten ist zur Übersetzungszeit bekannt.
Die Indexuntergrenze ist in C stets 0.

Betrachten wir die Feld-Deklaration: **int** *a* [8];

Wieviele Speicherzellen belegt das Feld *a* zur Laufzeit?
Offenbar besteht *a* aus den 8 Komponenten

a[0], a[1], a[2], a[3], a[4], a[5], a[6], a[7],

von denen jede –da vom Typ **int**– eine Zelle belegt.
Damit benötigt das Feld elf Speicherzellen für seine Komponenten.
Die Elemente legen wir aufeinanderfolgend in den Keller.
Als Adresse für *a* nehmen wir die Adresse der Komponente *a*[0].

Speicherbelegung für Datenfelder

Die Größe $|t'|$ eines Feldtyps $t' \equiv t[k]$ errechnet sich als Produkt aus Anzahl der Komponenten k und Größe $|t|$ des Komponententyps:
$|t'| = k \cdot |t|$

Die Speicherbelegungsfunktion kann tatsächlich zur Übersetzungszeit, –aus dem Deklarationsteil des C-Programms– berechnet werden.
Z.B. Abspeichern des Wertes 42 in die Komponente $a[0]$ durch:
loadc 42, **loadc** $\rho(a)$, **store**, **alloc** -1 Adresse $\rho(a)$ statisch bekannt!

Interessant wird die Übersetzung der Wertzuweisung $a[i] \leftarrow 42$;
Die **int**-Variable i erhält ihren Wert erst zur Ausführungszeit!
Dazu müssen Befehle erzeugt werden, die

- zuerst den aktuellen Wert von i ermitteln
- und diesen Betrag zur Anfangsadresse $\rho(a)$ addieren, um die richtige Komponente des Felds zu auszuwählen:

loadc 42, **loadc** $\rho(a)$, **loada** ρ(i), **add**, **store**, **alloc** -1

Speicherbelegung für Datenfelder

Sei a ein Ausdruck für ein Feld von Komponenten des Typs t.

Für die Anfangsadresse der Komponente $a[e]$ muss erst die Anfangsadresse des Felds a ermittelt werden. Dann wird der Wert von e berechnet. Dieser Index muss mit dem Platzbedarf $|t|$ für eine Komponente multipliziert werden, bevor er zur Anfangsadresse des Feldes addiert wird.
(Adresse von $a[e]$) = (Adresse von a) + $|t| \cdot$ (Wert von e)

Als Code für die Adresse eines indizierten Feldausdrucks erhalten wir:
$\text{code}_A^\rho\ a[e] = \text{code}_A^\rho\ a$, $\text{code}_W^\rho\ e$, **loadc** $|t|$, **mul**, **add**

Der Wert eines indizierten Feldausdrucks $a[e]$ kann dann leicht bestimmt werden, indem der Inhalt der adressierten Speicherzelle geladen wird. Besteht die Komponente des indizierten Felds aus mehreren Speicherzellen, so nutzen wir die verallgemeinerten **load** und **store** Befehle.