Programmiersprachen (185.208, VL 2)

Inhalt: Syntax, Semantik, Datentypen, Kontrollstrukturen, Module, objektorientierte und funktionale Sprachen

Buch: *Programming Language Concepts*. Carlo Ghezzi und Mehdi Jazayeri, 3. Auflage, John Wiley & Sons

Voraussetzung: grundlegende Programmierkenntnisse

Zeit: Mittwoch 11. 3. – 29. 4. 2009 (6 mal)
$14^{00} - 16^{00}$ Uhr

Ort: EI 3A

Vortragender: Franz Puntigam, Inst. für Computersprachen
franz@complang.tuwien.ac.at
http://www.complang.tuwien.ac.at/franz

Inhalt

- Allgemeiner Überblick
- Syntax und Semantik
- Datentypen
- Kontrollstrukturen
- Modularität und Programmieren im Großen
- Objektorientierte Programmiersprachen
- Funktionale Programmiersprachen

Softwareentwicklungsprozess

- Anforderungsanalyse und Spezifikation
- Softwareentwurf (Design)
- Implementierung (Programmierung)
- Verifikation und Validierung (Test und Bewertung)
- Wartung

Sprachen — Entwicklungsumgebungen

Entwicklungsprozess durch Werkzeuge unterstützt:

Editor, Compiler, Simulator, Linker, Debugger, ...

Sprachen für Anforderungs- und Entwurfsdokumente,
Programm, Software-Dokumentation, ...

Idealfall: Werkzeuge verstehen diese Sprachen,
automatisieren Entwicklungsschritte,
prüfen Konsistenz und Vollständigkeit,
machen Verbesserungsvorschläge

CASE-Tools erreichen dieses Ideal nur teilweise.

Grund: Werkzeuge verstehen nur formale Sprachen.

Sprachen — Entwurfsmethoden

Entwurfsmethode = Richtlinien zum Entwurf von Software

Beispiele: strukturierte, top-down, objektorientierte Methoden

Programmierstile (*Paradigmen*) von Methoden abhängig

Implementierung ist schwierig wenn Entwurfsmethode und Paradigma der Programmiersprache verschieden sind.

Grenzen zwischen Entwurf und Implementierung sind fließend wenn Entwurfsmethode und Paradigma übereinstimmen.

Paradigmen von Programmiersprachen

- Prozedurale Programmierung
- Funktionale Programmierung
- Programmierung mit abstrakten Datentypen
- Programmierung auf der Basis von Modulen
- Objektorientierte Programmierung
- Generische Programmierung
- Deklarative Programmierung

Sprachen — Rechnerarchitekturen

Konventionelle (imperative, zustandsbasierte) Sprachen:

- *Abstraktionen* der Von-Neumann-Architektur
- Merkmale: sequentielle Befehlsausführung,
  destruktive Zuweisung

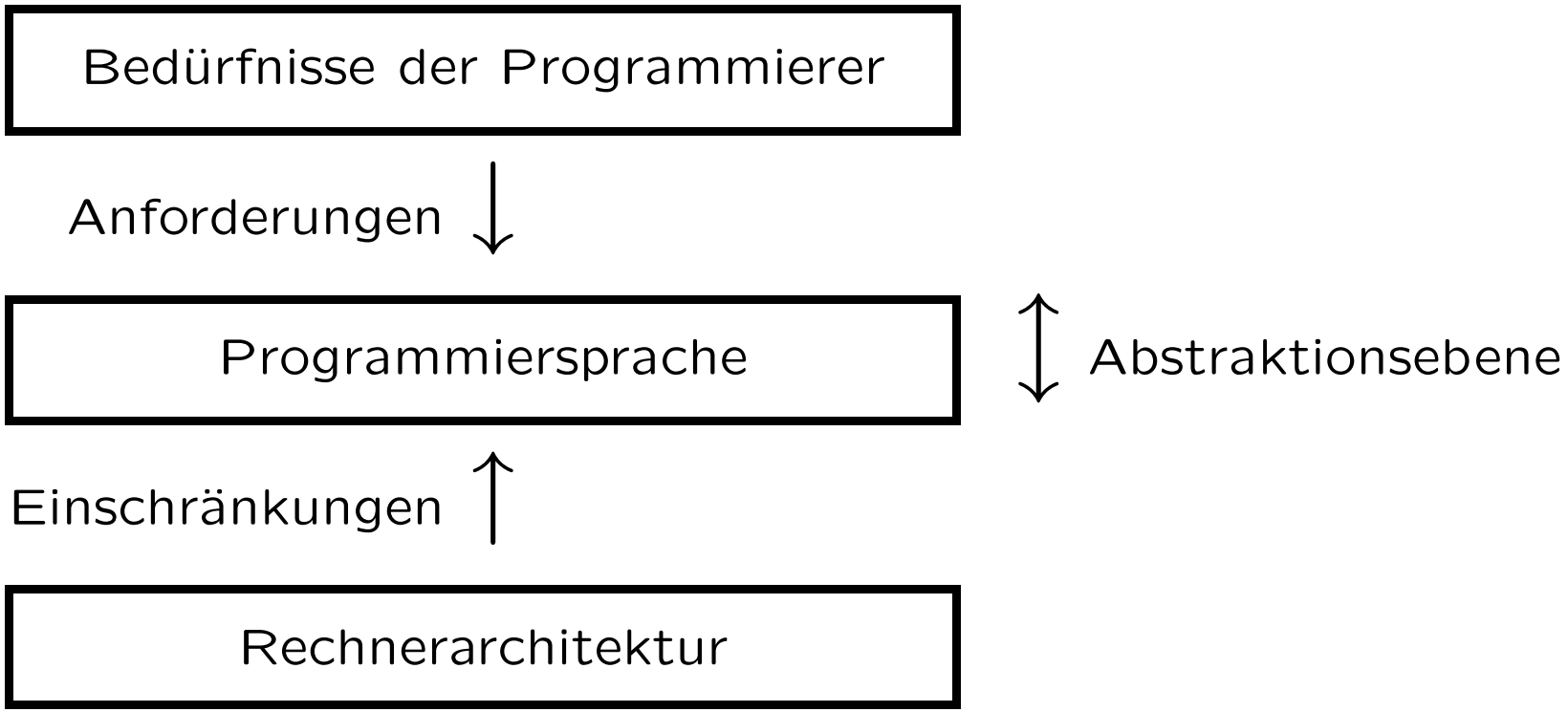
Funktionale und logische Sprachen:

- beruhen auf mathematischen Modellen
- Rechnerarchitekturen kaum erkennbar

Unterscheidung verschiedener Arten von Paradigmen:

- *Programmorganisation*: prozedural, objektorientiert, ...
- *Berechnungsmodell*: imperativ, funktional, logisch, ...

Anforderungen und Einschränkungen



Qualität von Programmiersprachen

- *Zuverlässigkeit der Software*
  Ausdruckskraft, Einfachheit, Sicherheit und Robustheit der Sprache, Lesbarkeit der Programme

- *Wartbarkeit der Software*
  Zusammenfügen zusammengehörender Merkmale, Vermeidung globaler Effekte

- *Effizienz der Software*
  Ausführungsgeschwindigkeit und Speichereffizienz, Wiederverwendbarkeit und Portabilität

FORTRAN	1954–57	Numerische Berechnungen
ALGOL 60	1958–60	Numerische Berechnungen
COBOL	1959–60	Verarbeitung von Geschäftsdaten
LISP	1956–62	Symbolische Datenverarbeitung
BASIC	1964	Ausbildung (Interaktiv)
SIMULA 67	1967	Simulation
Pascal	1971	Ausbildung (Allzweck)
PROLOG	1972	Künstliche Intelligenz
C	1972	Systemprogrammierung
CLU	1974–77	ADT-Programmierung
Modula-2	1977	Systemprogrammierung
Ada	1979	Allgemeine Anwendungen
Smalltalk	1971–80	Persönliche Datenverarbeitung
C++	1984	Allgemeine Anwendungen
Eiffel	1988	Allgemeine Anwendungen
Perl	1990	Skript-Sprache
Java	1995	Netzwerkanwendungen

```
#include <iostream.h>
#include "phone.h"

extern phone_list pb;
void insert();
number lookup();

main()
{   int request;
    cout << "Enter 1 to insert, 2 to lookup:" << endl;
    cin >> request;
    if (request == 1) insert();
    else if (request == 2) cout << lookup();
    else { cout << "invalid request." << endl;
           exit(1); }
}
```

Syntax und Semantik

Syntax:

- Syntax-Regeln beschreiben das *Aussehen* von Ausdrücken, Anweisungen und Programmen.
- Kleinste syntaktische Einheiten sind Zeichen.

Semantik:

- Semantik-Regeln beschreiben die *Bedeutung* von Ausdrücken, Anweisungen und Programmen.
- Semantische Grundeinheiten und Regeln sind oft wesentlich komplizierter als syntaktische.
- Die meisten bedeutsamen Unterschiede zwischen Sprachen sind semantischer Natur.

Variablen

Variablen sind benannte Speicherbereiche, die mit Werten belegt werden können.

Wichtige semantische Eigenschaften einer Variable:

Gültigkeitsbereich (Scope): Der Teil eines Programms, von dem aus auf die Variable zugegriffen werden kann

Typ: Beschreibt die Art der Werte, die in der Variable gespeichert werden können

Lebenszeit: Beschreibt, wann die Variable erzeugt und wann ihr Speicherbereich wieder freigegeben wird

Werte und Referenzen

Jede Variable (in C++) hat einen l-Wert und einen r-Wert.
Der l-Wert entspricht einer Referenz auf die Variable,
der r-Wert entspricht dem Inhalt der Variablen.

- `x = y;` Der r-Wert von `y` wird in den durch den l-Wert von `x` bestimmten Speicherbereich kopiert.
- `x = &y;` Der l-Wert von `y` wird in den durch den l-Wert von `x` bestimmten Speicherbereich kopiert.
- `x = 3;` Der r-Wert 3 (ein Literal) wird in den durch den l-Wert von `x` bestimmten Speicherbereich kopiert.
- `3 = y;` Fehler, da das Literal 3 keinen l-Wert hat.

Programmorganisation

Ein Programm besteht aus mehreren *Modulen*.

Die explizite Angabe verwendeter Module impliziert eine Hierarchie von Modulen — hierarchische Entwicklungsmethoden.

Programmiersprachen können zusichern, dass unabhängig voneinander erstellte Module zusammenpassen.

Nur für ganz kleine Programme sind Module verzichtbar, sonst ist die Sprachunterstützung von Modulen sehr wichtig.

Daten und Algorithmen

- Komplexe Datenstrukturen kommen in den meisten Programmen vor.
- Fast alle Sprachen bieten Mechanismen (*Datentypen*) zur einfacheren Handhabung komplexer Datenstrukturen an.
- Programmiersprachen beschreiben die Ausführung eines Programms oder Algorithmus.
- *Kontrollstrukturen* (Schleifen, bedingte Ausführung, ...) helfen, komplexere Algorithmen auf strukturierte Art und Weise zu beschreiben.

⟨program⟩ ::= {⟨statement⟩*}
⟨statement⟩ ::= ⟨assignment⟩ | ⟨conditional⟩ | ⟨loop⟩
⟨assignment⟩ ::= ⟨identifier⟩ = ⟨expr⟩;
⟨conditional⟩ ::= **if** ⟨expr⟩ {⟨statement⟩+} |
if ⟨expr⟩ {⟨statement⟩+} **else** {⟨statement⟩+}
⟨loop⟩ ::= **while** ⟨expr⟩ {⟨statement⟩+}
⟨expr⟩ ::= ⟨identifier⟩ | ⟨number⟩ | (⟨expr⟩) |
⟨expr⟩ ⟨operator⟩ ⟨expr⟩

⟨operator⟩ ::= + | − | ∗ | / | = | ≠ | < | > | ≤ | ≥
⟨identifier⟩ ::= ⟨letter⟩ ⟨ld⟩*
⟨ld⟩ ::= ⟨letter⟩ | ⟨digit⟩
⟨number⟩ ::= ⟨digit⟩+
⟨letter⟩ ::= a | b | c | . . . | z
⟨digit⟩ ::= 0 | 1 | 2 | . . . | 9

Abstrakte/konkrete Syntax, Pragmatik

```
while (x == y)
    x = ...;
```

```
while (x == y)
    {
        ...
    }
```

```
while x = y do
    begin
        ...
    end
```

```
if x = y then
    ...
end
```

```
if x = y then
    ...
else
    ...
end
```

```
while x = y do
    ...
end
```

Statische und dynamische Semantik

Syntax definiert wohlgeformte Programme.
Semantik definiert die Bedeutung wohlgeformter Programme.

Nicht alle wohlgeformten Programme haben eine Bedeutung.
Beispiel: **while** (1.75) { . . . }

Wohlgeformte Programme mit Bedeutung werden *vor* der Ausführung von solchen ohne Bedeutung unterschieden.

Statische Semantik definiert, welche wohlgeformten Programme Bedeutung haben.

Dynamische Semantik definiert die Effekte der Ausführung wohlgeformter Programme mit Bedeutung.

Formale Semantik

Semantik $\neq$ Implementierung

Meta-Sprache zur formalen Spezifikation der Semantik basiert auf mathematischen Methoden.

Formale Semantik gut als Referenz, aber meist schwer lesbar.

Operationale Semantik: Pseudo-Implementierung auf virtueller Maschine.

Axiomatische Semantik: Zustandsübergänge mittels Logik beschrieben.

Denotationale Semantik: Zustandsübergänge und aktueller Speicherzustand durch Funktionen beschrieben.

Axiomatische Semantik

{*Precondition*} *Statement* {*Postcondition*}
Beispiel: $\{y \geq 0\}\ x = y + 1;\ \{x > 0\}$

{wahr}
if $x \geq y$ **then** $max = x;$ **else** $max = y;$
{(max = x und $x \geq y$) oder (max = y und $y > x$)}

{P} S1; S2; {Q} wenn {P} S1; {R} und {R} S2; {Q}

{P} **while** B **do** L {P und nicht B} wenn {P} L {P}

Denotationale Semantik

dsem(S1; S2;, mem) = mem2 wenn dsem(S1;, mem) = mem1
und dsem(S2;, mem1) = mem2

dsem(**if** B **then** L1 **else** L2, mem) = mem1 wobei
mem1 = dsem(L1, mem) wenn mem(B) = wahr;
mem1 = dsem(L2, mem) wenn mem(B) = falsch

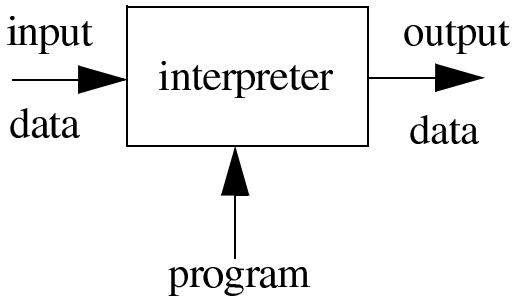
dsem(**while** B **do** L, mem) =
mem wenn mem(B) = falsch;
dsem(**while** B **do** L, dsem(L, mem)) wenn mem(B) = wahr

Interpretation und Übersetzung

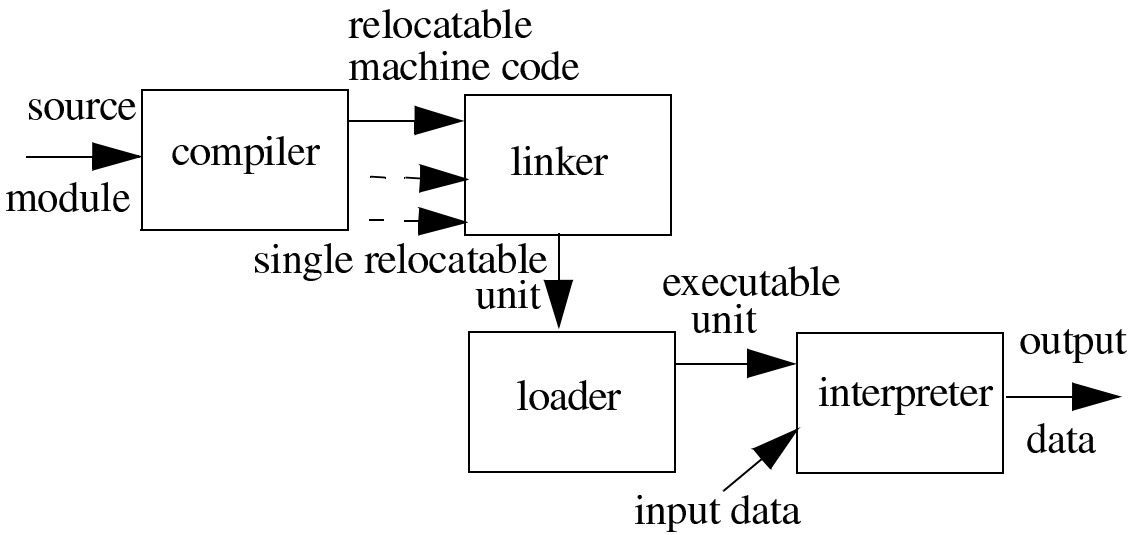
Sowohl Interpreter als auch Prozessoren (Hardware) führen folgende Schritte in einer Schleife aus:

1. Lade den nächsten Befehl (oder das nächste Statement)
2. Bestimme, was zu tun ist
3. Führe die Aktionen aus

Programme in einer high-level Sprache werden mittels Compiler, Linker und Loader in äquivalente Programme in einer lower-level Sprache übersetzt; diese werden dann interpretiert.



(a) Interpretation



(b) Translation (followed by interpretation)

FIGURE 2.3 Language processing by interpretation (a) and translation (b)

Interpretation und Übersetzung (2)

Reine Übersetzung: übersetzte Programme ("native code") von der Hardware ausgeführt (hohe Ausführungsgeschwindigkeit, statische Typüberprüfungen möglich)

Reine Interpretation: Programmtext direkt vom Interpreter ausgeführt (portabel, interaktive Softwareentwicklung unterstützt, oft Speichereffizient)

Mischformen: Programme in Zwischencode übersetzt; dieser von einem Interpreter ausgeführt (Kompromiss)

Mehrere Übersetzungsschritte (Macros) können die Ausdruckskraft einer Sprache erhöhen.

Binden

Das Zuweisen eines Wertes an ein Attribut (Name, Typ, ...) eines Sprachelements (Variable, Statement, ...) heisst *Binden*.

Eigenschaften von Bindungen:

- Zeipunkt des Bindens (Zeitpunkt der Sprachdefinition, Implementierung, Übersetzung oder Ausführung)
- Stabilität der Bindung (fix oder änderbar)

Eine fixe Bindung vor der Ausführung ist *statisch*,
eine änderbare Bindung während der Ausführung *dynamisch*.
Fixe Bindungen während der Ausführung sind auch möglich.

Variablen

Variable ist Tupel $\langle$Name, Scope, Typ, l-Wert, r-Wert$\rangle$

Name: Zeichenfolge zur Bezeichnung der Variablen in einem Programm (oder Programmabschnitt)

Scope (oder Gültigkeitsbereich): Der Programmabschnitt in dem der Name bekannt (und gültig) ist

Typ: Angaben zum Wertebereich der Variablen

l-Wert: Die mit der Variable assoziierte Speicheradresse

r-Wert: Der kodierte Wert, der an der durch den l-Wert bestimmten Adresse gespeichert ist

Namen und Gültigkeitsbereiche

Ein Variablenname wird oft durch eine *Deklaration* eingeführt. Der Gültigkeitsbereich erstreckt sich von der Deklaration bis zu einem durch die Sprachdefinition bestimmten Ende.

Dynamic scoping: Gültigkeitsbereich erstreckt sich bis zur erneuten Ausführung einer Deklaration desselben Namens (APL, LISP, SNOBOL4, . . .)

Static (lexical) scoping: Gültigkeitsbereiche werden durch die lexikalische Programmstruktur bestimmt (C, Java, . . .)

In blockstrukturierten Sprachen endet der Gültigkeitsbereich mit dem Block, in dem die Variable deklariert wurde.
Innere Blöcke können Namen in äußeren Blöcken verdecken.

Dynamic scoping

```
{  /* Block A */
   int x;
   ...
}
...
```

```
{  /* Block B */
   int x;
   ...
}
...
```

```
{  /* Block C */
   ...
   x = ...;
   ...
}
```

Static scoping

```
#include <stdio.h>
main()
{  int x, y, z;
   scanf("%d %d %d", &x, &y, &z);
      /* drei Dezimalzahlen werden gelesen und
         in den l-Werten von x, y und z gespeichert */
   {  /* Block zum Vertauschen von x und y */
      int z;
      z = x;
      x = y;
      y = z;
   }
   printf("%d %d %d", x, y, z);
}
```

Typen

Der Typ einer Variablen beschreibt die *Menge der Werte*, die im l-Wert der Variable gespeichert werden können, und die *Operationen*, die angewendet werden können, um solche Werte zu erzeugen, zu modifizieren und darauf zuzugreifen.

Einige Typnamen werden zum Zeitpunkt der Sprachdefinition an Wertemengen und Operationenmengen gebunden.
Werte und Operationen werden zum Zeitpunkt der Sprachimplementation an Maschinen-Repräsentationen gebunden.

Durch Typdeklarationen werden neue Typen zur Übersetzungszeit eingeführt.

Z.B. Pascal: **type** vector = **array**[1..10] **of** integer

Benutzerdefinierter Typ in C++

```
class stack_of_char {
public:
   stack_of_char(int sz) {
      top = s = new char[size = sz];
   }
   ~stack_of_char() {delete[] s;}
   void push(char c) {*top++ = c;}
   char pop() {return *--top;}
   int length() {return top - s;}
private:
   int size;
   char* top;
   char* s;
};
```

Statische Typisierung

Eine Sprache ist *statisch typisiert* wenn Typen statisch an Variablen gebunden werden (C, Modula-2, Ada, Java, . . .).

Z.B. Pascal: **var** x, y: integer;
c: character;

Statische Typisierung ermöglicht statische Typüberprüfungen.

Z.B.: x := c; ist illegal

Variablentypen müssen nicht immer explizit spezifiziert sein.

Z.B.: in FORTRAN bestimmt der Variablenname den Typ;
in ML wird der Typ durch *Typinferenz* ermittelt.

Dynamische Typisierung

Eine Sprache ist *dynamisch typisiert* wenn Typen erst zur Laufzeit an Variablen gebunden werden (LISP, Smalltalk, . . .).
Solche Variablen sind *polymorph*.

Ein Typ bleibt an eine Variable gebunden bis ein neuer Wert an die Variable zugewiesen wird. Dynamische Typinformation muss zur Laufzeit mitgeführt werden.

Dynamische Typüberprüfungen erfolgen zur Laufzeit.
Z.B.: $x := y + 3;$ ist illegal wenn y vom Typ "String" ist.
y kann vom Typ "Integer" oder "Real" sein.

Eine Sprache, in der keine Typüberprüfung erfolgt, ist *untypisiert* (Assembler).

l-Werte

Der Speicherbereich wird während der Lebenszeit eines Daten-Objekts benötigt.
Das Binden eines Speicherbereichs (l-Wertes) an eine Variable heißt *Speicher-Allokation*, das Aufheben der Bindung am Ende der Lebenszeit *Speicher-Deallokation*.

Man spricht von *statischer Allokation* wenn die Allokation vor der Programmausführung und die Deallokation nach Programmbeendigung erfolgt.
Sonst spricht man von *dynamischer Allokation*.

Dynamische Allokation (bzw. Deallokation) erfolgt entweder *explizit*, oder *implizit* bei Erreichen einer Variablendeklaration (bzw. am Ende des Gültigkeitsbereichs einer Variablen).

r-Werte

Der r-Wert einer Variablen ist in imperativen Sprachen meist dynamisch änderbar; nicht aber in funktionalen und logischen Sprachen.

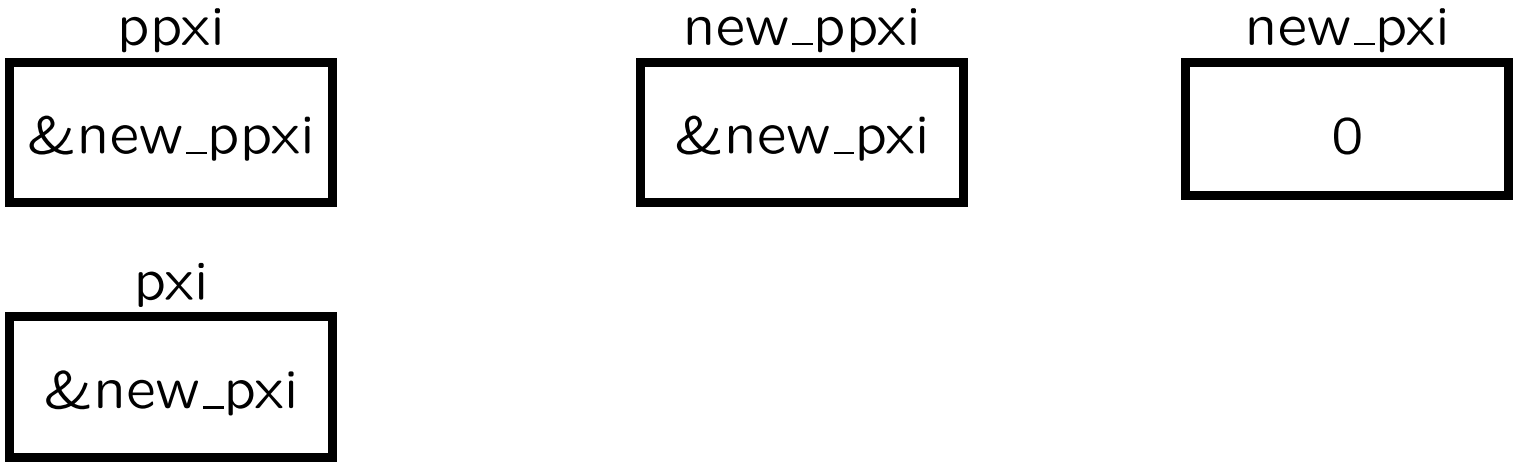
In den meisten Sprachen kann man Konstanten definieren; das sind Namen für r-Werte.
Z.B. in Pascal erfolgt die Bindung zwischen Name und r-Wert vor der Ausführung, in C oder Ada zur Laufzeit.

Einige Sprachen (wie ML) verlangen, dass jede Variable mit einem r-Wert initialisiert wird; andere Sprachen unterstützen die Initialisierung bei der Variablendeklaration.

Meist ist nicht definiert, welchen r-Wert eine uninitialisierte Variable hat (Fehlermeldung, implizite Initialisierung, . . .).

Zeiger

Ein Zeiger ist ein r-Wert, der als l-Wert verwendbar ist.




```
type PI = ^integer;
var pxi: PI;
...
new(pxi);
pxi^ = 0;
```

```
type PPI = ^PI;
var ppxi: PPI;
...
new(ppxi);
ppxi^ = pxi;
```

```
int x = 0;
int* px = &x;
int** ppx = &px;
int y = *px;
**ppx = 5;
```

Routinen

Routine hat Namen, Gültigkeitsbereich, Typ, l-Wert, r-Wert

Kopf: Name + Typen = *Signatur* (oft auch Parameternamen)
Z.B.: fun: $T_1 \times T_2 \times \cdots \times T_n \rightarrow R$

Rumpf: *lokale* Deklarationen + ausführbare Anweisungen;
r-Wert entspricht Rumpf, l-Wert dessen Speicheradresse

Eine Sprache behandelt Routinen als *first class Objekte* wenn sie Routinen-Variablen (bzw. Zeiger auf Routinen) unterstützt.

Aufruf: spezifiziert aktuelle Parameter; z.B.: i = sum(3);

C-Funktion und Zeiger auf Funktion

```
int sum(int n)
{
   int i, s;
   s = 0;
   for (i = 1; i <= n; i++)
      s = s + 1;
   return s;
}
```

```
int i;
int (*ps)(int);
...
ps = &sum;
i = (*ps)(5);
   /* i = sum(5); */
```

Routinen: Deklaration vs. Definition

```
int A(int x, int y);  /* Deklaration von A */
float B(int z)        /* Definition von B */
{  int w, u;
   ...
   w = A(z, u);       /* A ist hier sichtbar */
   ...
};
int A(int x, int y)   /* Definition von A */
{  float t;
   ...
   t = B(x);          /* B ist hier sichtbar */
   ...
};
```

Aufruf von Routinen

Instanz einer Routine: Code-Segment + *activation record* (*stack frame*)

Umgebung: lokale Umgebung = Objekte im activation record
nichtlokale Umgebung = alle anderen sichtbaren Objekte

Rekursion: ein neuer activation record kann angelegt werden bevor die Ausführung der Routine beendet ist. Bindung zwischen Code-Segment und activation record dynamisch.

Formale Parameter: Parameter im Kopf einer Routine

Aktuelle Parameter: Parameter (Argumente) im Aufruf

Bindung von Parametern

```
procedure Example(A: T1; B: T2 := W; C: T3);
-- Ada-Prozedur mit Default-Wert für Parameter B

Example(X, Y, Z);
-- Aufruf; Parameter über Position assoziiert

Example(C => Z, A => X, B => Y);
-- Parameter über Namen assoziiert

Example(X, C => Z);
-- Mischform; Default-Wert wird verwendet
```

Generische Routinen

```
template <class T> void swap(T& a, T& b)
/* generische Routine zum Vertauschen von a und b */
{
  T temp = a;
  a = b;
  b = temp;
}
```

```
int i, j;
...
swap(i, j);
```

```
float f, g;
...
swap(f, g);
```

Überladen und Aliasing

Ein Name mit mehr als nur einer Bedeutung ist *überladen*.

Zwei Namen, die dasselbe Objekt bezeichnen, sind *aliases*.

```
int i, j, k;
float a, b, c;
...
i = j + k;
a = b + c;
a = b + c + b();
a = b() + c + b(i);
```

```
int x = 0;
int *i = &x;
int *j = &x;
...
*i = 10;
```

SIMPLESEM (Operationale Semantik)

Einfacher Assembler zur Spezifikation der Semantik

Statische Struktur: Befehlszeiger (instruction pointer, ip),
Befehlsspeicher (code memory, C), Datenspeicher (D)

Befehle: set Ziel, Quelle
set 10, D[20]
set 15, read
set write, D[50]
set 99, D[15]+D[33∗C[41]]−8
set D[10], D[20]
jump 47
jumpt 47, D[3]>D[8]
jump D[13]

SIMPLESEM und Sprachen

Statische Sprachen:

Speicher vor Ausführung allokiert
Beispiele: Frühe Versionen von FORTRAN und COBOL
Wir beschreiben die Sprachen C1, C2 und C2′

Stack-basierte Sprachen:

Speicher zur Laufzeit von einem Stack (LIFO) allokiert
Beispiele: ALGOL 60, . . .
Wir beschreiben die Sprachen C3 und C4

Dynamische Sprachen:

Speicher zur Laufzeit von einem Heap allokiert
Beispiele: die meisten heutigen Sprachen
Wir beschreiben die Sprache C5

C1: Sprache mit einfachen Statements

```
main()
{
   int i, j;
   get(i, j);
   while (i != j)
      if (i > j)
         i -= j;
      else
         j -= i;
   print(i);
}
```

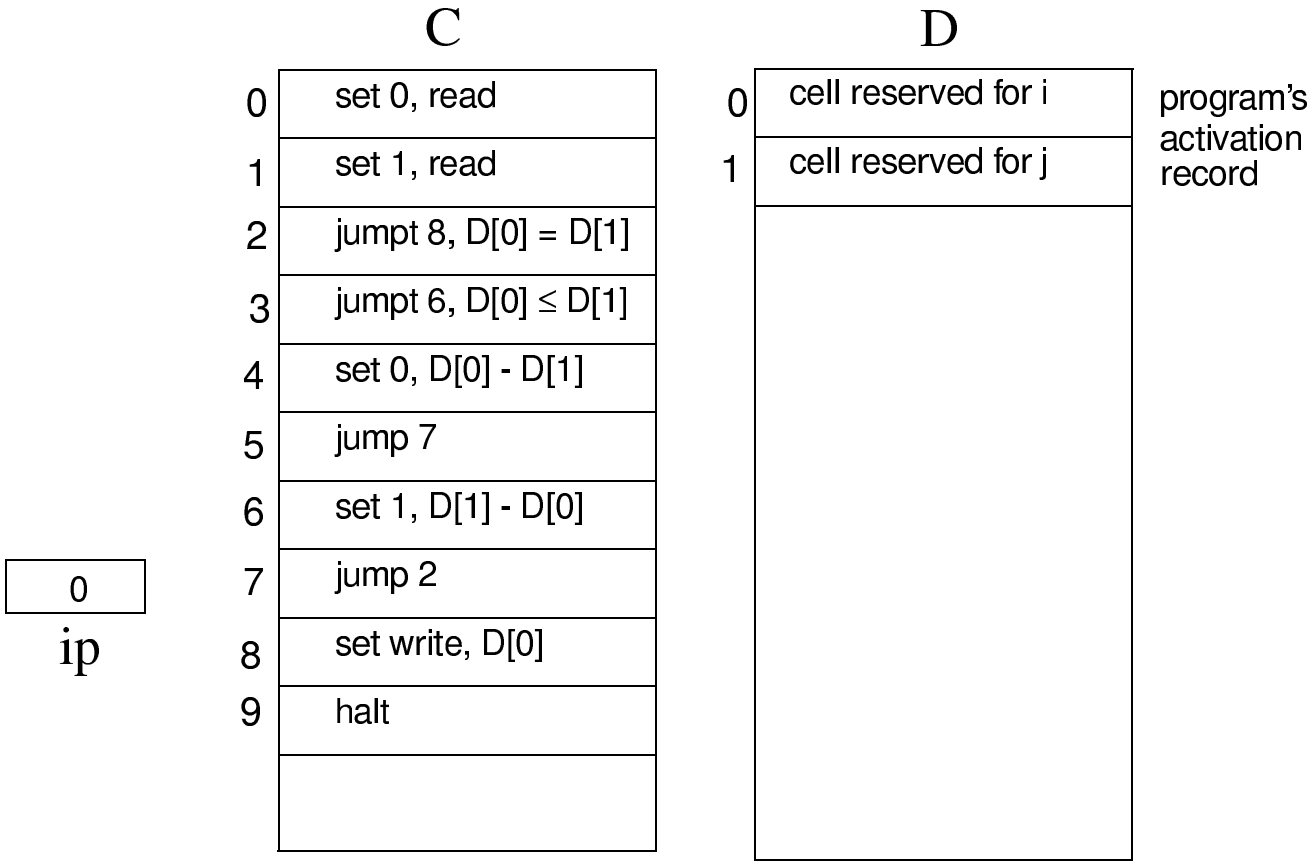


FIGURE 2.13 Initial state of the SIMPLESEM machine for the C1 program in Figure 2.12

C2: Einfache Routinen

```
int i=1, j=2, k=3;

alpha()
{
   int i=4, l=5;
   ...
   i += k + l;
   ...
};

beta()
{
   int k=6;
   ...
   i = j + k;
   alpha();
   ...
};

main()
{
   ...
   beta();
   ...
}
```

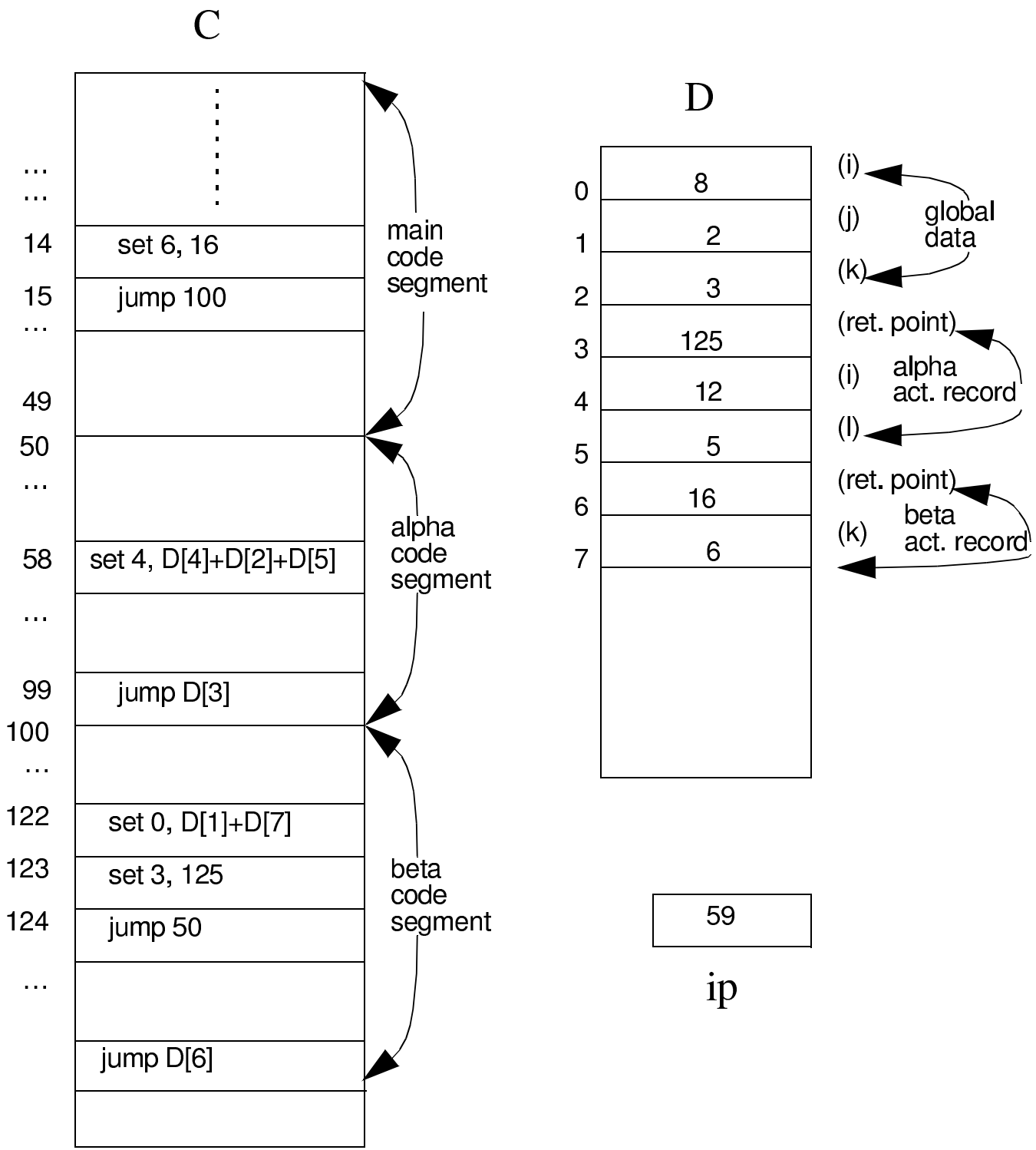


FIGURE 2.15 A snapshot of SIMPLESEM during execution of the program of Figure 2.14

C2': Getrennte Übersetzung

```
/* file 1 */

int i=1, j=2, k=3;
extern beta();
main()
{
   ...
   beta();
   ...
}
```

```
/* file 2 */

extern int k;
alpha()
{
   int i=4, l=5;
   ...
   i += k + l;
   ...
}
```

```
/* file 3 */

extern int i,j;
extern alpha();
beta()
{
   int k = 6;
   ...
   i = j + k;
   alpha();
   ...
}
```

C3: Rekursion und Ergebnis-Werte

```
int n;
int fact()
{
   int loc;
   if (n > 1) {
      loc = n--;
      return loc * fact();
   }
   else
      return 1;
}
```

```
main()
{
   get(n);
   if (n >= 0)
      print(fact());
   else
      print("input error");
}
```

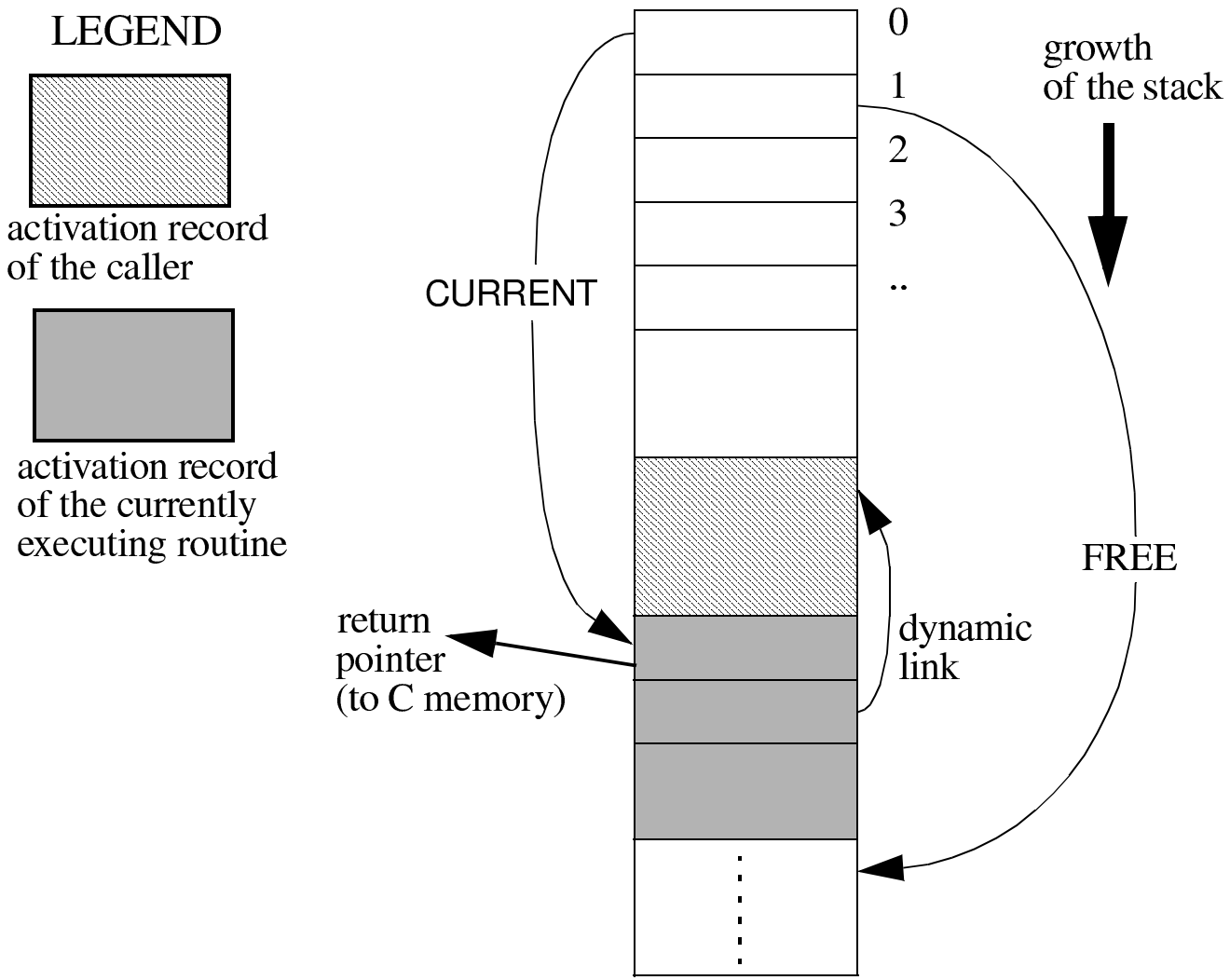


FIGURE 2.18 Structure of the SIMPLESEM D memory implementing a stack

Aufruf einer Routine

set 1, D[1]+1	allokiere Platz für Ergebnis
set D[1], ip+4	setze “return pointer”
set D[1]+1, D[0]	setze “dynamic link”
set 0, D[1]	setze “CURRENT”
set 1, D[1]+AR	setze “FREE”
jump addr	springe zur Routine

Rückkehr von einem Aufruf

set 1, D[0]	setze “FREE”
set 0, D[D[1]+1]	setze “CURRENT”
jump D[D[1]]	springe an die Rückkehradresse

```
0  set 2, read                 reads the value of n; 2 is the absolute address where
                               global variable n is stored
1  jumpt 10, D[2] < 0          tests the value of n
2  set 1, D[1] + 1             call to fact starts here; but first space for the result is
                               saved
3  set D[1], ip + 4            set return pointer
4  set D[1] + 1, D[0]          set dynamic link
5  set 0, D[1]                 set CURRENT
6  set 1, D[1] + 3             set FREE (3 is the size of fact's activation record)
7  jump 12                     12 is the starting address of fact's code
8  set write, D[D[1] - 1]      D[1] - 1 is the address where the result of the call to
                               fact is stored
9  jump 11                     end of the call
10 set write, "input error"
11 halt                        this is the end of the code of main
12 jumpt 23, D[2] ≤ 1          this is the start of the code of fact; tests the value of n
13 set D[0] + 2, D[2]          assigns n to loc
14 set 2, D[2] - 1             decrements n
15 set 1, D[1] + 1             call to fact starts here; but first space for the result is
                               saved
16 set D[1], ip + 4            set return pointer
17 set D[1] + 1, D[0]          set dynamic link
18 set 0, D[1]                 set CURRENT
19 set 1, D[1] + 3             set FREE (3 is the size of fact's activation record)
20 jump 12                     12 is the starting address of fact's code
21 set D[0] - 1, D[D[0] + 2] * D[D[1] - 1]    store returned value
22 jump 24
23 set D[0] - 1, 1             return 1
24 set 1, D[0]                 this and the next 2 instructions correspond to the
                               return from the routine
25 set 0, D[D[0] + 1]
26 jump D[D[1]]
```

FIGURE 2.19 SIMPLESEM code for the program in Figure 2.17

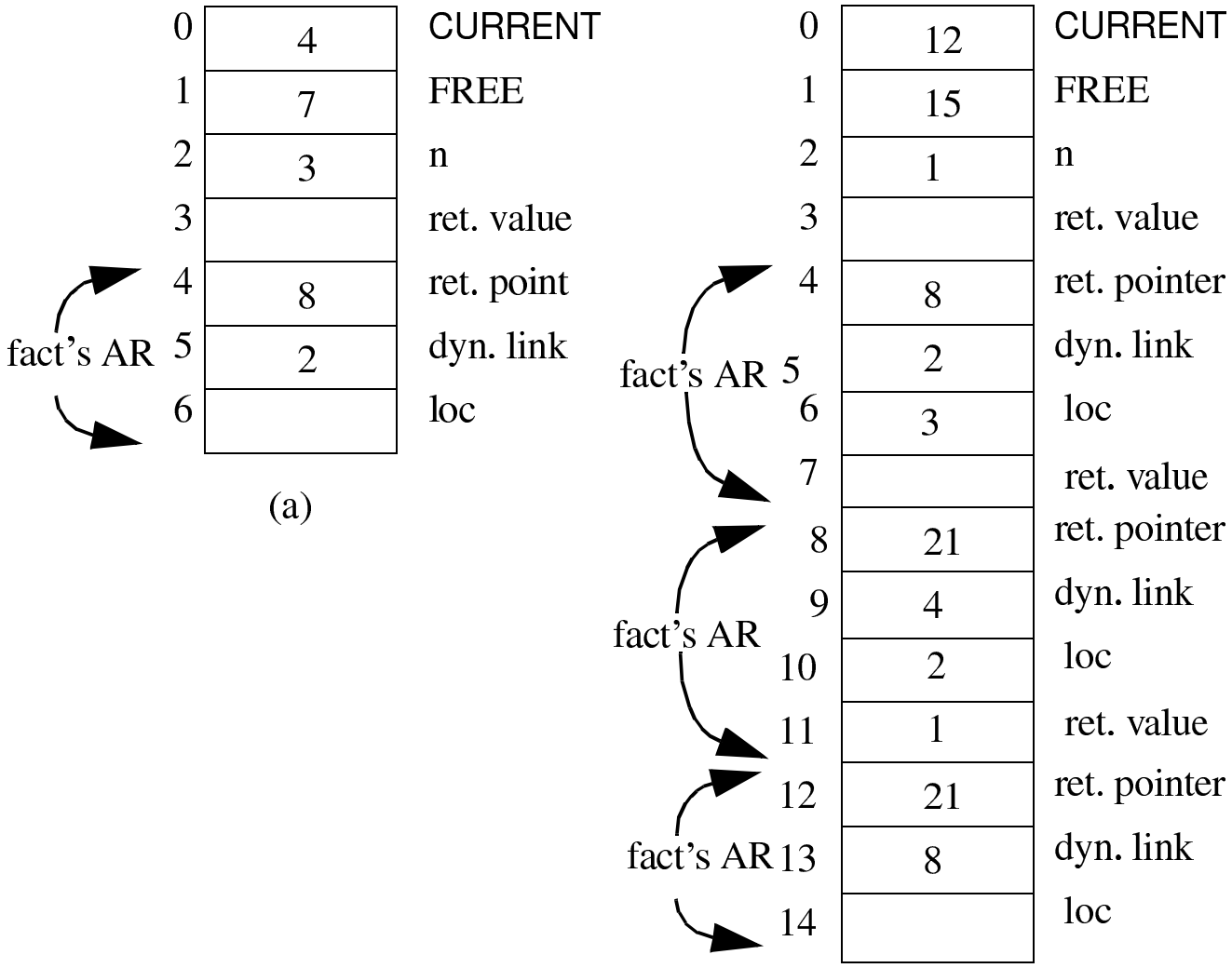


FIGURE 2.20 Two snapshots of the D memory

```
int f( );
{                                   //block 1
    int x, y, w;                    //1
    while (…)
    {                               //block 2
        int x, z;                   //2
        …
        while (…)
        {                           //block 3
            int y;                  //3
            …
        }                           //end block 3
        if (…)
        {                           //block 4
            int x, w;               //4
            …
        }                           //end block 4
    }                               //end block 2
    if (…)
    {                               //block 5
        int a, b, c, d;             //5
        …
    }                               //end block 5
}                                   //end block 1
```

FIGURE 2.21 An example of nested blocks in C4'

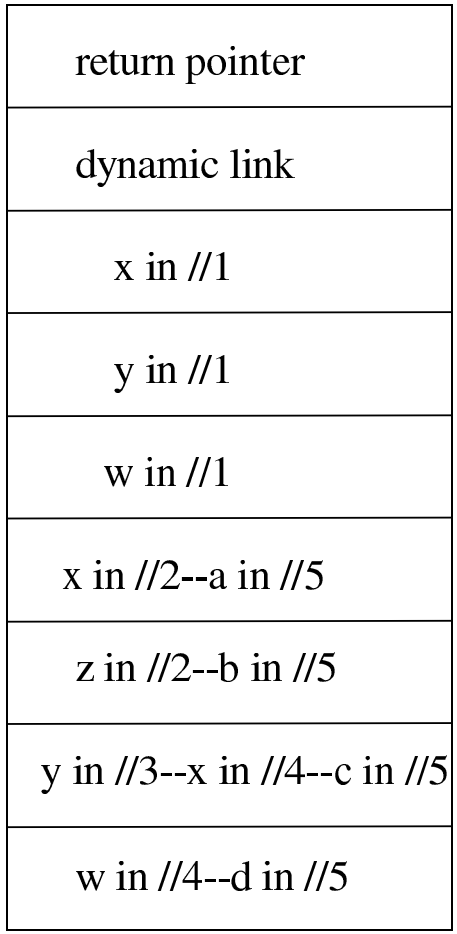


FIGURE 2.22 An activation record with overlays

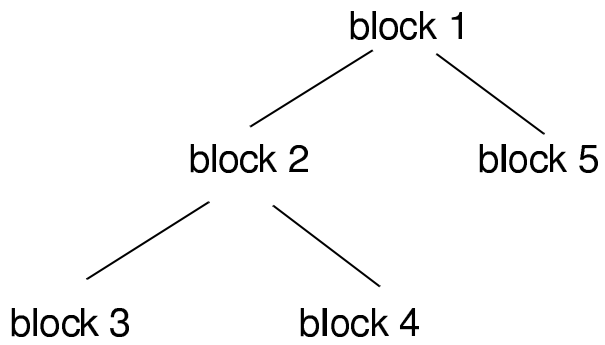
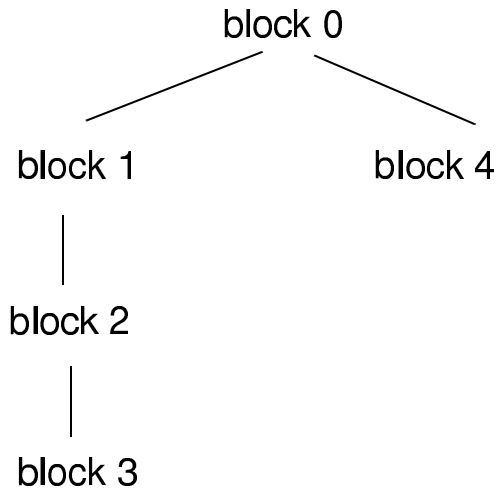


FIGURE 2.23 Static nesting tree for the block structure of Figure 2.21

```
//file start              //block 0
int x, y, z;
f1()
{                          //block 1
    int t, u;              // 1
    f2()
    {                      //block 2
        int x, w;          // 2
        f3()
        {                  //block 3
            int y, w, t;   // 3
            …
        }                  //end block 3
        x = y + t + w + z;
    }                      //end block 2
…
}                          //end block 1
main();
{                          //block 4
    int z, t;              // 4
    …
}                          //end block 4
//file end                 //end block 0
```

(a)



(b)

FIGURE 2.24 A C4" example (a) and its static nesting tree (b)

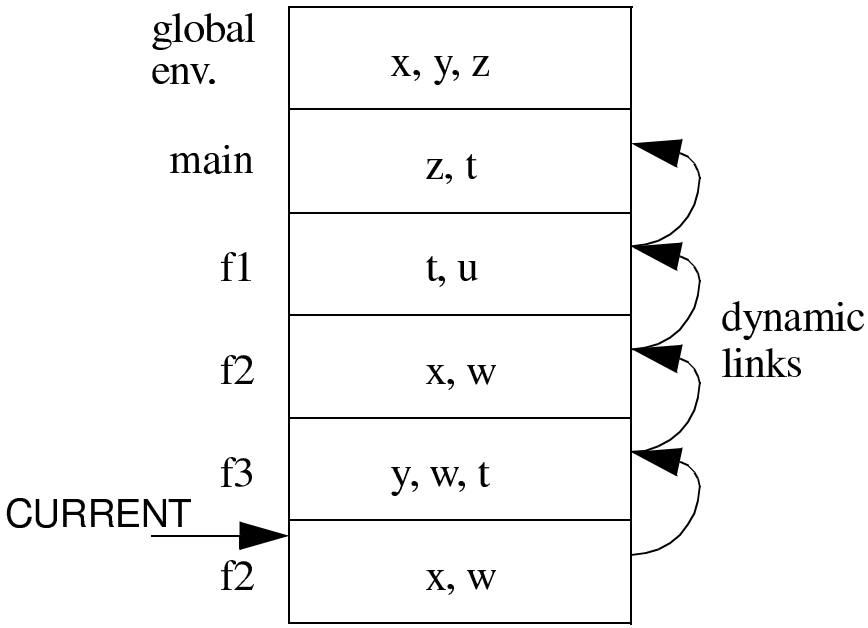


FIGURE 2.25 A sketch of the run-time stack

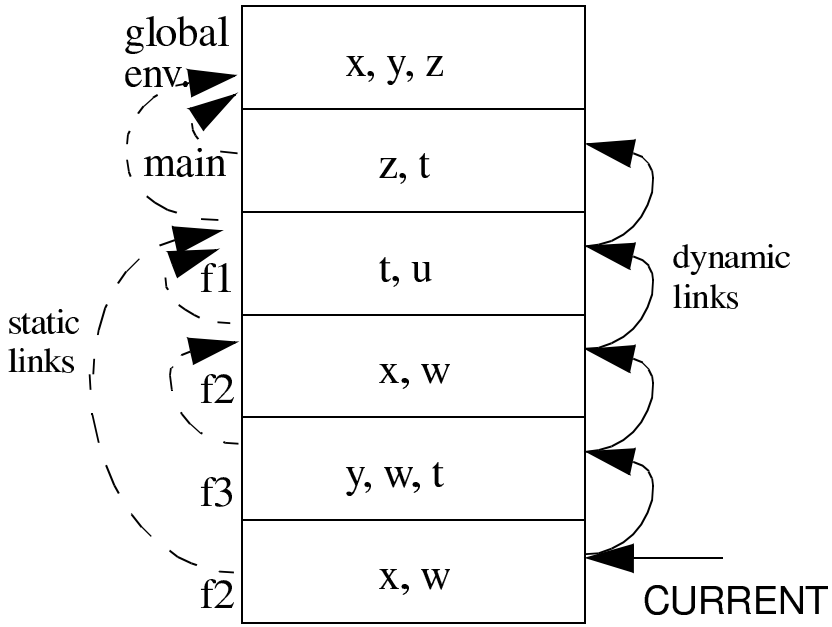


FIGURE 2.26 The run-time stack of Figure 2.25 with static links

Static Link

Jede Referenz ist statisch an ein Paar $\langle d, o \rangle$ gebunden;
d ist die Distanz, o der Offset

Berechnung des *frame pointers*:

$$\text{fp(d)} \;=\; \text{if d=0 then D[0] else D[fp(d}-1\text{)+2]}$$

Beispiele: fp(0) = D[0],
fp(1) = D[D[0]+2],
fp(2) = D[D[D[0]+2]+2]

Die Adresse einer Variable gebunden an $\langle d, o \rangle$ ist D[fp(d)+o]

Aufruf einer Routine in C4”

set 1, D[1]+1	allokiere Platz für Ergebnis
set D[1], ip+5	setze “return pointer”
set D[1]+1, D[0]	setze “dynamic link”
set D[1]+2, fp(d)	setze “static link”
set 0, D[1]	setze “CURRENT”
set 1, D[1]+AR	setze “FREE”
jump addr	springe zur Routine

C5': Dynamische Arrays

Beispiel: **type** V **is array** (Integer **range** <>) of Integer;
A: V (1 .. N); -- N nicht statisch bekannt

Zur Übersetzungszeit wird im activation record Platz für einen Descriptor reserviert. Dieser enthält einen Zeiger auf den Speicher für das Array sowie je eine Zelle für die Indexgrenzen.

Bei Ausführung der Deklaration eines dynamischen Arrays wird der activation record um Platz für das Array erweitert. Der Zeiger im Descriptor wird entsprechend gesetzt.

Beispiel für Zugriff A(I) := 0;

set[D[D[0]+m] + D[D[0]+s]], 0

(m = Offset des Descriptors von A; s = Offset von I)

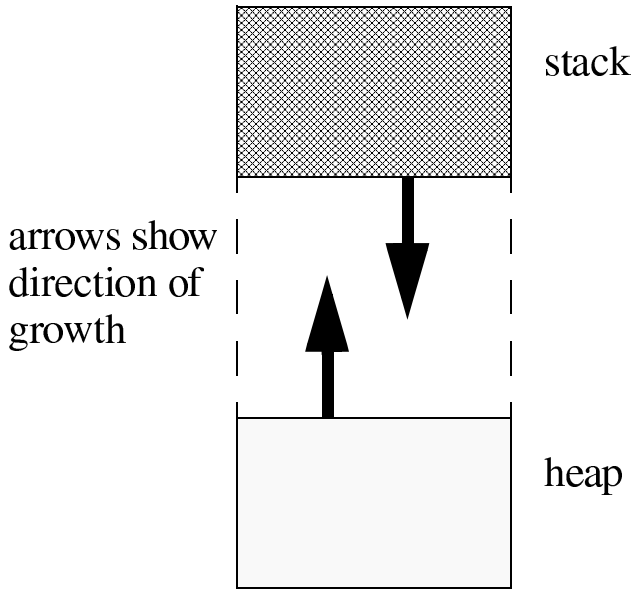


FIGURE 2.27 Organization of the D memory to accommodate a heap

```
sub2() {
    declare x;
    …
    … x …;
    … y …;
    …
}
sub1() {
    declare y;
    …
    …x …;
    … z …;
    sub2();
    …
}
main() {
    declare x, y, z;
    z = 0;
    x = 5;
    y = 7;
    sub1;
    sub2;
    …
}
```

FIGURE 2.28 An example program in a dynamically scoped language

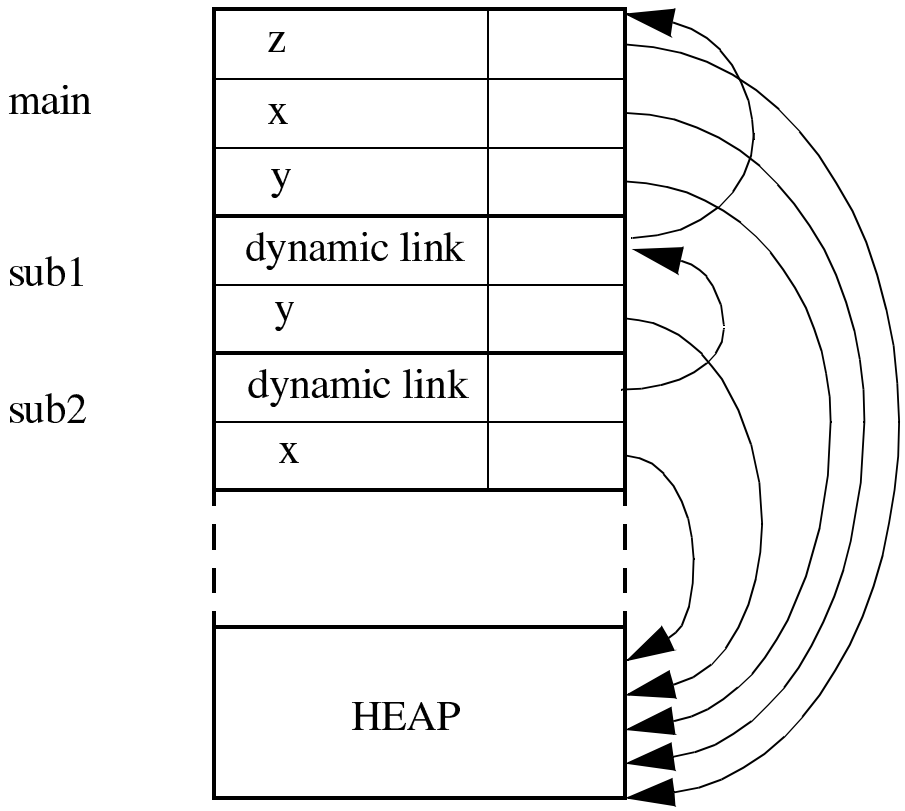


FIGURE 2.29 A view of the run-time memory for the program of Figure 2.28

Parameterübergabekonventionen

Call by reference: l-Wert des aktuellen Param. übergeben

Call by copy: Parameter sind lokale Variablen

- **Call by value:** übergebener r-Wert in Speicherbereich der aufgerufenen Routine kopiert
- **Call by result:** Rückgabewert in l-Wert einer Variablen des Aufrufers kopiert
- **Call by value-result:** r-Wert einer Variablen des Aufrufers in Speicherbereich der Routine kopiert; am Ende Inhalt des Speicherbereichs in l-Wert der Variablen kopiert

Call by name: In aufgerufener Routine jedes Vorkommen des formalen durch aktuellen Parameter ersetzt

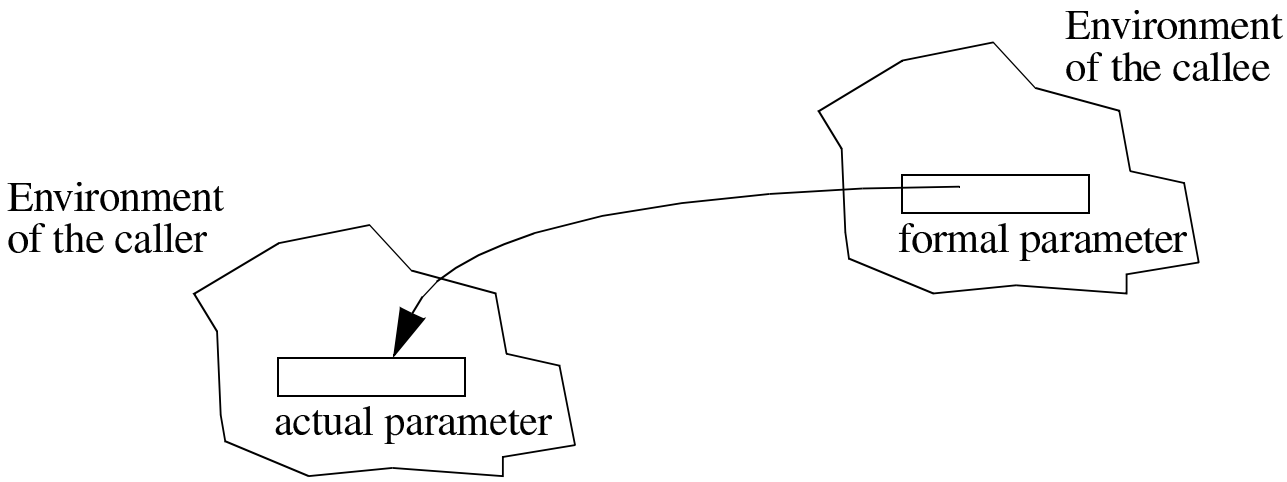


FIGURE 2.30 A view of call by reference

Implementierung von Call by Reference

Activation record der aufgerufenen Routine um eine Zelle pro Parameter erweitert (Offset off)

Übergabe einer Variablen $\langle d, o \rangle$:

set D[0]+off, fp(d)+o

Übergabe eines call-by-reference-Parameters $\langle d, o \rangle$:

set D[0]+off, D[fp(d)+o]

Zuweisung an call-by-reference-Parameter $x = 0;$

set D[D[0]+off], 0

Call by Reference / Value-Result

Diese Konventionen unterscheiden sich wenn

- zwei formale Parameter aliases sind;

  Beispiel: aktuelle Parameter: a[i] und a[j]
  formale Parameter: x und y
  Rumpf der Routine: x = 0; y++;

- ein formaler Parameter und eine Variable, die sowohl für Aufrufer als auch Aufgerufenen sichtbar ist, aliases sind.

  Beispiel: aktueller Parameter: a (Variable)
  formaler Parameter: x
  Rumpf der Routine: a = 1; x = x + a;

Call by Name

```
                              |   int c; /* globale Variable */
swap(int a, b) {              |   swap(int a, b) {
   int temp = a;              |      int temp = a; a = b;
   a = b;                     |      b = temp; c++;
   b = temp;                  |   };
};                            |   y() {
   ...                        |     int c, d;
   swap(i, a[i]);             |     swap(c, d);
   ...                        |   };

temp = i;                     |   temp = c; c = d;
i = a[i];                     |   d = temp;
a[i] = temp;                  |   c++;                  !!
```

Routinen als Parameter

```
1  int u, v;          |   7  b(routine x)          |  19  main()
2  a()                |   8  {                     |  20  {
3  {                  |   9     int u, v, y;       |  21     b(a);
4     int y;          |  10     c() {              |  22  };
5     ...             |  11        ...             |
6  };                 |  12        y = ...;        |
                      |  13        ...             |
                      |  14     };                 |
                      |  15     x();               |
                      |  16     b(c);              |
                      |  17     ...                |
                      |  18  }                     |
```

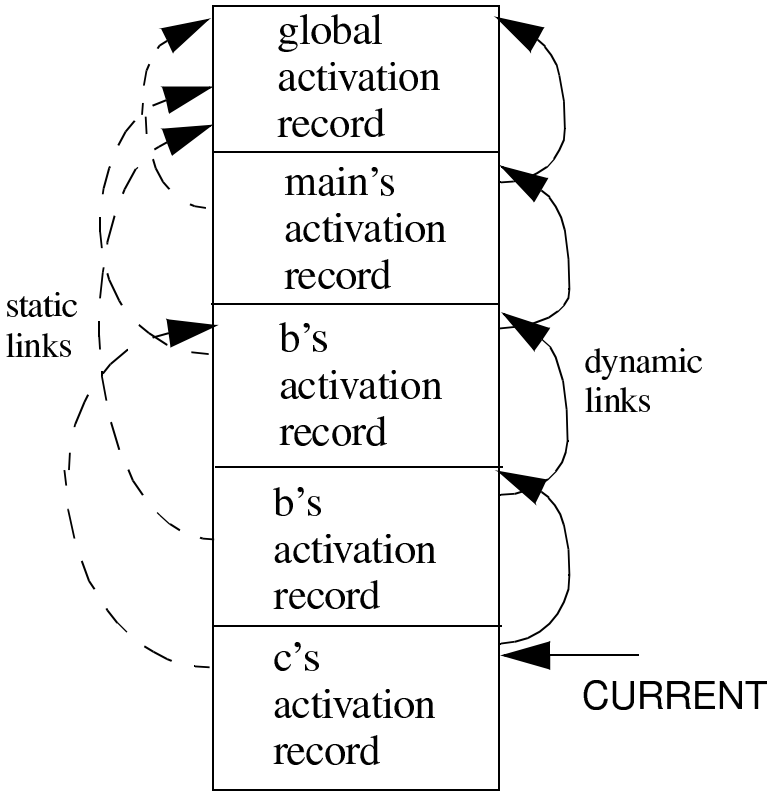


FIGURE 2.33 A sketch of the run-time stack for the program of Figure 2.32