4 Das Laufzeitsystem

Bevor die Übersetzung von der höheren Programmiersprache in einen Zwischencode behandelt werden kann, muss etwas genauer auf Fragen des Gültigkeitsbereichs von Deklarationen und der Speicherzuordnung von Variablen eingegangen werden. Die Realisierung dieser Konzepte geschieht im Laufzeitsystem, das zusammen mit den System-Schnittstellen, dem Speicher-Management bzw. dem darüber liegenden Software-Schichten den Zielrechner darstellt. Außerdem werden in diesem Abschnitt verschiedene Methoden der Parameterübergabe diskutiert.

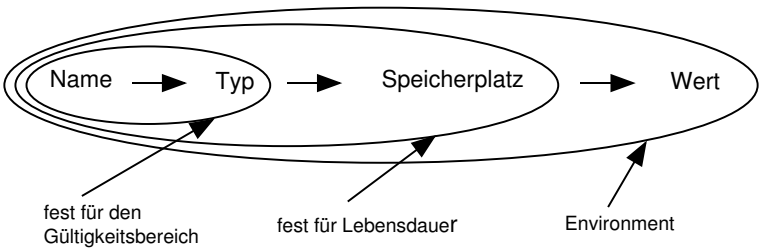
4.1 Gültigkeitsbereich und Lebensdauer

Deklarationen sind Sprachkonstrukte in höheren Programmiersprachen, bei denen einem Namen ein Typ zugeordnet wird. Dies geschieht in fast allen modernen Programmiersprachen explizit, d.h. der Programmierer schreibt etwa `var abc : integer;`. Diese Zuordnung hat nun einen gewissen Gültigkeitsbereich innerhalb des Programms und das zugeordnete Objekt hat eine eventuell sogar davon abweichende Lebensdauer.
Der **Gültigkeitsbereich** (Scope) dieser Deklaration von `abc` umfasst den Teil des Programms, in dem sich eine Verwendung des Namens `abc` auf diese Deklaration bezieht.
Üblicherweise wird der Gültigkeitsbereich statisch und lexikographisch, d.h. unabhängig von einem Programmlauf, allein durch den Programmtext, festgelegt.

Bem.: Man könnte den Gültigkeitsbereich von `abc` auch dynamisch festlegen, d.h. die zeitlich zuletzt erfolgte Deklaration von `abc` hätte an dieser Stelle Gültigkeit! Diese Methode wurde in einigen frühen Lisp-Versionen benutzt.

Bem.: In anderen Programmiersprachen wie etwa ML, Smalltalk oder Python kann auf eine Deklaration der Variablen verzichtet werden, da entweder ein komplexes Typ-Inferenzsystem den zugehörigen Typ einer Variablen berechnet oder aber die an die Variablen gebundenen Objekte selbst Typ-Informationen enthalten.
FORTRAN ist ein Beispiel einer Programmiersprache, die eine implizite Deklaration von Variablen erlaubt. Jede Variable, die mit einem der Buchstaben `I`, `J`, `K`, `L`, `M` oder `N` beginnt, ist vom Typ `integer`, alle anderen vom Typ `real`

Die **Lebensdauer** (Extend) des durch `abc` bezeichneten Objekts beginnt mit der Speicherzuordnung für das Objekt und endet mit der Freigabe des Speichers.
Meist geschieht die Speicherzuordnung beim Eintritt in eine Prozedur oder einen Block und beim Verlassen wird der Speicher wieder freigegeben. Es gibt aber auch viele andere Möglichkeiten, etwa die Speicherzuordnung durch expliziten Aufruf von Konstruktoren und die Freigabe des Speichers durch den Aufruf von Destruktoren oder aber auch durch automatische Garbage-Collector Verfahren.



4.2 Aktivierungs-Record, statische und dynamische Verkettung

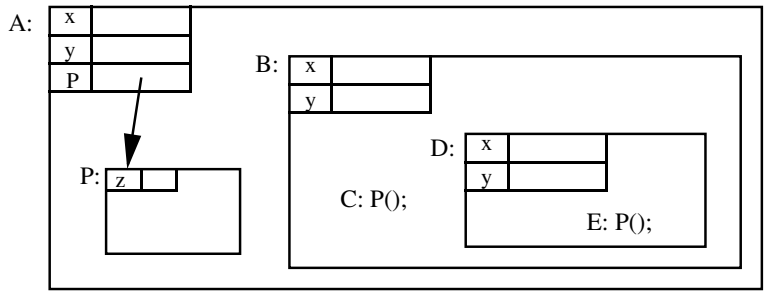
Im folgenden Beispiel sollen Gültigkeitsbereich und Lebensdauer von Variablen betrachtet und die notwendigen Implikationen auf das Laufzeitsystem des Programms etwas genauer untersucht werden.
Wir wollen eine Programmiersprache betrachten, die Gültigkeitsbereiche von Variablen über eine Blockstruktur regelt. Ein Block ist ein Programmteil, der Deklarationen von Variablen enthält, deren Gültigkeitsbereich und Lebensdauer auf diesen Block beschränkt ist. Probleme treten dann auf, wenn Blöcke ineinander geschachtelt werden können und besonders dann, wenn innerhalb von Blöcken Prozeduren definiert werden können.

Beispiel 4.1:
Beispiel für Schachtelung von Blöcken und Prozeduren

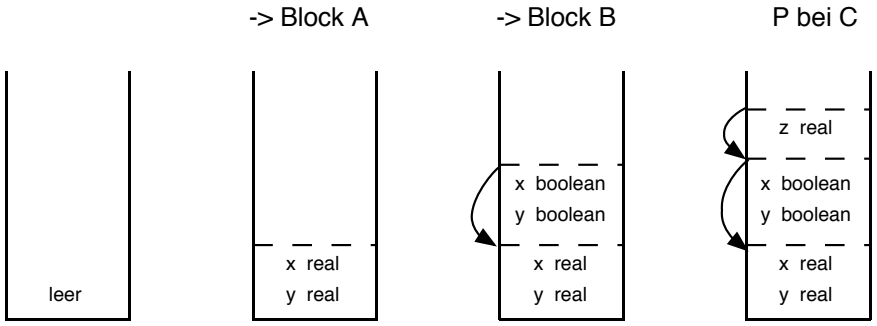
```
A:  begin
        real x,y;
        procedure P();
        begin
            real z;
            x:=x+z;
        end P;

B:      begin
            boolean x,y;
              ...
C:          P();
              ...
D:          begin
                integer x,y;
                  ...
E:              P();
                  ...
            end D;
        end B;
    end A;
```

Um die Schachtelung der einzelnen Blöcke bzw. Prozedurrümpfe formal zu beschreiben, verwendet man die sogenannte Stufenzahl. Alle Deklarationen im äußersten, umfassenden Block sind auf Stufe 1. Bei jedem Blockeintritt erhöht sich die Stufenzahl um 1, bei jedem Blockaustritt vermindert sie sich um 1. Bei Prozeduren gilt eine spezielle Regel - bei einer Prozedur auf Stufe i sind die Deklarationen im Rumpf sowie die Deklarationen der Formalparameter auf Stufe i, der Name der Prozedur ist jedoch noch auf Stufe $i-1$ deklariert.
Im Beispiel 4.1 sind offensichtlich die Deklarationen von `x` und `y` als `real` auf Stufe 1, die Deklarationen der beiden Variablen als `integer` dagegen auf Stufe 3. Die Prozedurname `P` ist auf Stufe 1, die lokale Variable `z` dagegen auf Stufe 2 deklariert.

Manchmal ist es nützlich, die Verschachtelungen in Form eines **Schachtelungsdiagramms** graphisch darzustellen.



Die großen Rechtecke stellen dabei jeweils einen Block oder einen Prozedurrumpf dar. In der linken oberen Ecke eines derartigen Rechtecks sind die in diesem Block deklarierten Namen mit symbolischen Platzhaltern für die jeweiligen Werte vermerkt. Die Stufenzahl wird dann durch die Tiefe der Verschachtelung der Rechtecke repräsentiert.
Jeder Blockrand ist halb durchlässig - man kann innerhalb eines Blocks nach außen sehen, jedoch nicht von außen in einen Block hinein. Wird z. B. innerhalb des Blocks `D` bei `E` der Name `P` benutzt, so ist damit die Prozedur auf Stufe 2 gemeint; wird der Name `z` benutzt, so ist an dieser Stelle keine Deklaration von `z` gültig! Die Benutzung der Namen `x` und `y` bezieht sich auf die in `D` deklarierten Variablen. Die in `B` und `A` deklarierten Variablen `x` und `y` sind von dieser Stelle aus nicht sichtbar (werden durch die Deklaration in `D` überdeckt).
Der Speicher für die deklarierten Variablen eines Blocks wird in einem **Aktivierungs-Record** (stack-frame) bereitgestellt.
Dieser enthält alle für **eine** Aktivierung einer ausführbaren Programmeinheit notwendigen Daten. Die Aktivierungs-Records werden üblicherweise auf einem Stack, dem sogenannten **Laufzeitstack** (run-time stack) abgelegt.
Man betrachte das Beispiel 4.1. Der Laufzeitstack würde sich dann wie folgt entwickeln:

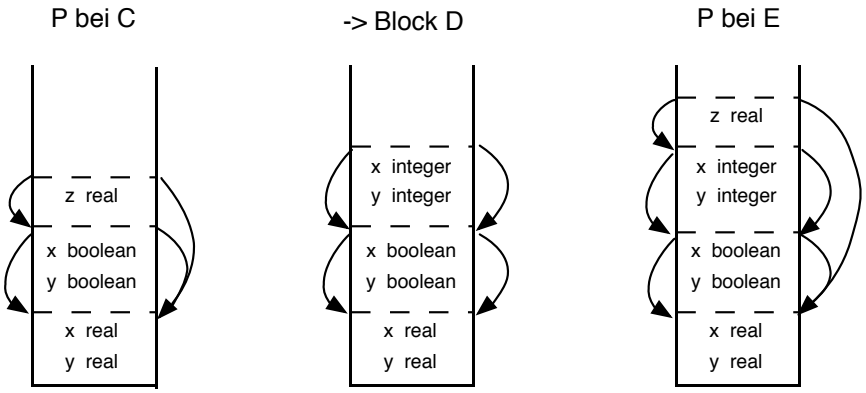


wobei jeder einzelne Aktivierungs-Record einen zusätzlichen Zeiger hat (**dynamischer Link**, durch Zeiger auf der linken Seite des Stacks angedeutet), der auf den Anfang des darunter liegenden Records auf dem Stack verweist. Damit kann nach Verlassen eines Blocks der zugehörige Aktivierungs-Record wieder gelöscht werden.
Betrachtet man die Situation bei Abarbeitung des Prozedurrumfes von `P` (die Aktivierungsrecords sind dabei wie im Bild rechts), so hat man bei der Abarbeitung des Befehls `x := x + z` in `P` zwei Möglichkeiten der Interpretation:

1) Durchsuchen des Stacks von „oben nach unten“, um den ersten Aktivierungs-Record zu finden, in dem ein `x` auftritt. Dies entspricht einem dynamischen Gültigkeitsbereich für die Deklarationen. Man folgt also der dynamischen Verkettung der Aktivierungs-Records (dem dynamischen Link) und würde die Deklaration von `x` bei `B:` als erste finden.

2) Durchsuchen des Stacks von „oben nach unten“, um den ersten Aktivierungs-Record eines `P` umfassenden Blocks zu finden, der eine Deklaration von `x` enthält. In diesem Fall würde also die Deklaration von `x` bei `A:` gemeint sein. Dies entspricht einem lexikalen Gültigkeitsbereich für die Deklarationen. Um diesen Record zu finden benötigt man eine weitere Verkettung der Aktivierungs-Records (**statischer Link**, durch Zeiger auf der rechten Seite des Stacks angedeutet)

Die weitere Entwicklung des Laufzeit-Stacks mit Angabe der statischen Verkettung der Aktivierungs-Records wäre dann wie folgt:



Befindet man sich zur Laufzeit des Programms in einem Block oder einer Prozedur, die auf Stufe i definiert wurde, so hat die Kette der statischen Links die Länge $i-1$. Eine nicht lokale Größe, die j Stufen zurück definiert wurde, ist in einem Aktivierungs-Record zu finden, der j Schritte in der Kette der statischen Links zurück liegt.

Also werden Variablen 2-dimensional adressiert:

- die *Stufenzahl* der Deklaration wird benötigt, um den richtigen Aktivierungs-Record zu finden, und
- ein vom Compiler festgelegter Abstand vom Anfang des Aktivierungs-Records (*Offset*) liefert die endgültige Adresse.

*Dies bedeutet aber auch, dass die Variablennamen **nicht** in dem Aktivierungs-Records abgespeichert werden müssen!*

Beispiel 4.2:
Angenommen im Programm aus Beispiel 4.1 werde im Block `A` eine neue Variable `i` deklariert, die im Block `D` benutzt wird. Die Deklaration von `i` ist auf Stufe 1, benutzt wird `i` im Block `D`, der sich auf Stufe 3 befindet. Die Stufendifferenz ist 2, also muss man, um auf `i` zuzugreifen, zweimal dem statischen Link folgen um zum Aktivierungs-Record zu kommen, in dem `i` abgelegt ist. *Diese Differenz ist aber zur Übersetzungszeit bekannt.*

Um die Zugriffsgeschwindigkeit zu erhöhen, verwendet man häufig ein Feld von Zeigern (**Display** genannt) auf die jeweils aktuellen Aktivierungs-Records.
Die Länge des Feldes entspricht genau der Stufenzahl des aktiven Blocks. Eine Variable `a`, die auf Stufe i deklariert wurde, ist in dem Aktivierungs-Record zu finden, auf den der i-te Eintrag im Display zeigt.
Erlaubt die Programmiersprache nur Prozedurdefinitionen auf Stufe 1, wie etwa in C, kann man auf den statischen Link völlig verzichten, denn es gibt nur Variablen auf Stufe 1 (globale Variablen) oder Variablen auf Stufe 2 (lokale Variablen).

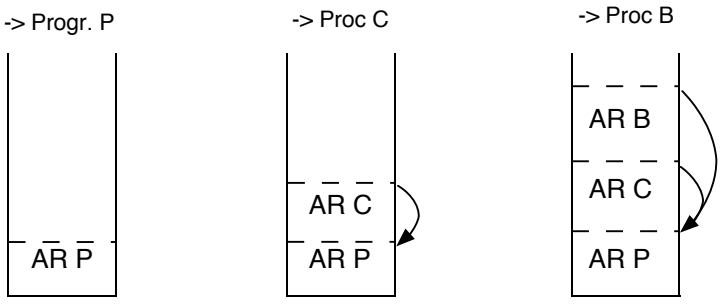
4.3 Prozeduren als Parameter oder als Rückgabewerte

Wenn die Programmiersprache Prozeduren als Parameter zulässt, ergeben sich weitere Probleme beim Laufzeitsystem.

Beispiel 4.3:
Man betrachte das folgende Programmfragment:

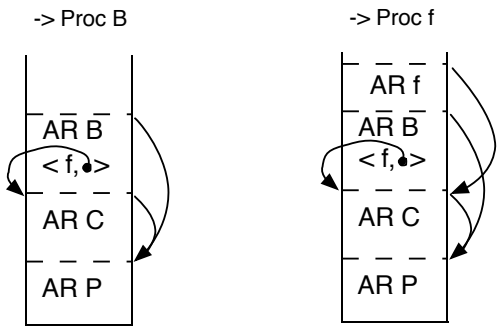
```
program P
    begin
    procedure B (function h(n:integer):integer);
        begin
            writeln(h(2))
        end;
    procedure C
        var m:integer;
        function f(n:integer):integer);
            begin
                f:=m+n;
            end;
        begin
            m:=0;
            B(f);
        end;
    begin
        C;
    end;
```

Nach dem Aufruf von `C` und danach von `B` ergibt sich folgendes Bild:



In `B` wird jetzt die als Parameter übergebene Funktion `f` aufgerufen. Allerdings tritt im Gültigkeitsbereich von `B` die Variable `m` *nicht* auf. Es ist also wichtig, dass die Funktion `f` ihr Environment mitbringt, d.h. als Parameter für den Aufruf der Prozedur `B` wird sowohl die Startadresse der Prozedur als auch ein Zeiger auf das Environment des Aufrufs, also einen Zeiger auf den Aktivierungs-Record von `C` übergeben.
Man nennt ein derartiges Paar auch **Closure**.

Damit ergibt sich folgendes Bild des Laufzeitstacks:



Ein ähnliches Problem ergibt sich, wenn die Programmiersprache nicht-lokale Sprünge, also Sprünge mit einem Sprungziel in einem umfassenden Block, zulässt.
Diese Sprünge müssen das Environment ändern. Man benötigt dazu die Stufendifferenz zwischen der Definition der Marke, also des Sprungziels, und dem entsprechenden `goto`-Befehl.

Hinweis: Sieht die Programmiersprache vor, dass Prozeduren auch als Rückgabewerte von Prozeduren auftreten, gibt es zusätzliche Probleme. Liefert z.B. eine Prozedur `F` eine Prozedur `P` zurück, die lokal in `F` ist, dann verschwindet nach dem Verlassen von `F` der Aktivierungs-Record von `F` und damit ein Teil des Environments von `P`. Damit dürfte klar sein, dass bei derartigen Sprachen, so etwa bei Scheme, keine einfache Stack-Implementation des Laufzeitsystems möglich ist.

4.4 Speicherorganisation

Wie wird man nun den virtuellen oder auch realen Adressraum eines Prozesses aufteilen? Üblicherweise ordnet das Betriebssystem einem Prozess zumindest die folgenden drei Speichersegmente zu, wobei jedes Speichersegment einen zusammenhängenden, sequentiellen Adressraum darstellt:

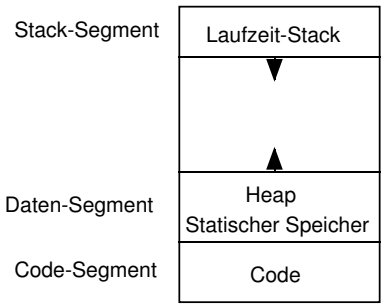
- Das **Code-Segment**, das das ausführbare Programm enthält und üblicherweise schreibgeschützt ist.
- Das **Stack-Segment**, das den Laufzeit-Stack für die Aktivierungs-Records enthält.
- Das **Daten-Segment**, das den statischen Speicher und den dynamischen Speicher (auch **Heap** genannt) enthält.

Der Laufzeit-Stack enthält den Speicher für Datenobjekte, deren Lebensdauer an die Aktivierung von Prozeduren bzw. Blöcken gebunden ist. Für jede Aktivierung wird im allgemeinen ein Speicherplatz einer festen Größe zugeordnet (Teil des Aktivierungs-Records). Die Adressierung der Objekte erfolgt meist indirekt über eine für alle Aktivierungs-Records gleiche Startposition, die z.B. über einen sogenannten **Frame-Pointer** festgelegt wird, und einen Offset, der bereits zur Übersetzungszeit bekannt ist.
Der statische Speicher enthält Speicherplatz für alle Datenobjekte, deren Lebensdauer sich vom Start des Programms bis zu dessen Ende erstreckt. Diese Speicherzuordnung kann bereits während der Übersetzungsphase erfolgen und die Adressierung kann damit direkt erfolgen.
Der dynamische Speicher enthält Speicherplatz für alle Datenobjekte, deren Lebensdauer nicht durch die Ablaufstruktur der Programmiersprache vorhersagbar ist. Typischerweise sind dies

Datenobjekte, die vom Programmierer selbst erzeugt werden können und deren Speicher, je nach Programmiersprache, auch explizit wieder freigegeben werden muss. In C gibt es dazu die Standardaufrufe `malloc` und `mfree`. Auch Konstruktoren in objektorientierten Programmiersprachen erzeugen Objekte, die in diesem Speicherbereich abgelegt werden.

In manchen dieser Sprachen, wie etwa Smalltalk oder auch Java, muss sich der Programmierer nicht mehr um die Rückgabe des frei gewordenen Speichers kümmern, diese Aufgabe erledigt ein sogenannter **Garbage Collector** im Hintergrund.
In anderen Sprachen wie etwa auch in Scheme werden die benötigten Datenobjekte sowohl automatisch erzeugt als auch wieder zurückgegeben. Auch hier arbeitet ein Garbage Collector, um den nicht mehr benötigten Speicherplatz dem freien Speicher zuzuordnen.
Die in der Verwaltung des Heaps verwendeten Verfahren sind meist recht aufwendig und werden später behandelt.

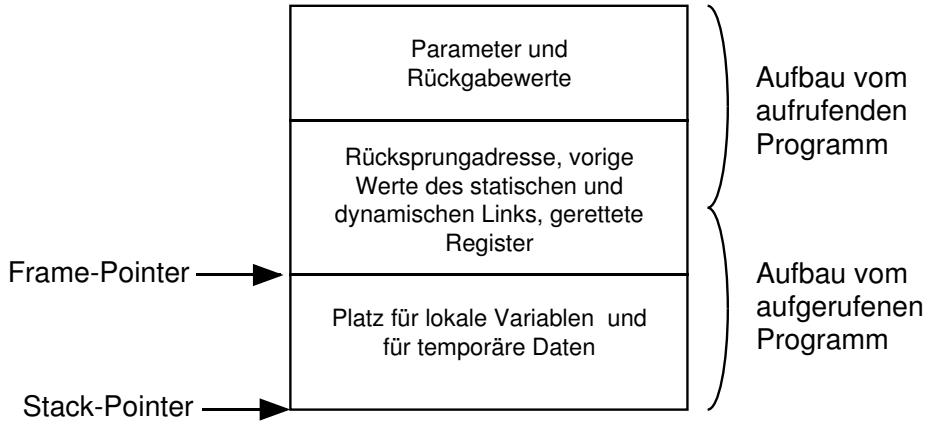
Die folgende Abbildung zeigt eine typische Verteilung der drei Segmente in dem hier angenommenen linearen und zusammenhängenden Adressraum eines Prozesses.



Das Stack-Segment wächst dabei von oben, d.h. zu den kleineren Adressen hin, während der Heap bei Bedarf nach oben, also zu den größeren Adressen hin wächst.

Wichtig zu bemerken ist, dass zugeordnete Datenobjekte eventuell im Daten-Segment verschoben werden müssen, um wieder größere, zusammenhängende Speicherbereiche zu erhalten.
Das bedeutet aber, dass z.B. in lokalen Variablen keine direkten Pointer oder Referenzen auf diese Datenobjekte gespeichert werden dürfen, da diese sonst nach einer Verschiebung auf unsinnige Speicherbereiche zeigen würden.
Eine Lösung ist die Verwendung sogenannter **Handles**, die einen Zeiger auf einen Zeiger auf das Datenobjekt darstellen und deren zweiter Zeiger bei einer Verschiebung vom System aktualisiert werden kann.

Abschließend soll noch beispielhaft der Aufbau eines Aktivierungs-Records gezeigt werden. Beim Aufruf einer Prozedur werden zunächst die Aktualparameter auf den Stack geschrieben und Platz für eventuelle Rückgabewerte geschaffen. Danach werden die momentanen Werte des dynamischen und statischen Links gerettet. Beim Sprung zur Prozedur wird dann die Rücksprungadresse ebenfalls auf den Stack geschrieben. In der angesprungenen Prozedur werden zunächst die Registerinhalte gerettet und dann, durch Setzen des Stack- und Frame-Pointers, der Aufbau des Aktivierungs-Records beendet.



Möglicher Aufbau eines Aktivierungs-Records

4.5 Parameterübergabe

Es gibt mehrere Methoden der Parameterübergabe, da es meist mehrere Möglichkeiten gibt, die Bedeutung eines Aktualparameters zu interpretieren.
Zum Beispiel kann ein Aktualparameter `a[i]` interpretiert werden als

1) der Wert des i-ten Elementes eines Feldes `a` (**r-Wert**), oder

2) der Ort, an dem das i-te Element eines Feldes `a` abgespeichert ist (**l-Wert**), oder aber auch

3) eine Zeichenkette, die immer dann einzusetzen ist, wenn der zugeordnete Formalparameter in der Prozedur auftritt (Makro-Substitution).

Bemerkung: Die Begriffe r-Wert und l-Wert beziehen sich auf eine übliche Anweisung etwa der Form `x := y + z` in einer Programmiersprache. Um diese Anweisung abzuarbeiten benötigt man die Werte von `y` und `z` und die Adresse von `x`. Die unterschiedliche Behandlung der auftretenden Namen `x`, `y` und `z` hängt also davon ab, ob sie links (l-Wert) oder rechts (r-Wert) vom Wertzuweisungszeichen `:=` auftreten. Natürlich gibt es einige Ausdrücke in Programmiersprachen, die etwa nur einen r-Wert haben, etwa eine Zahl oder ein Ausdruck wie `y + z`.

Es gibt die folgenden Möglichkeiten der Parameterübergabe:

call-by-value Formalparameter sind wie initialisierte lokale Variablen zu betrachten - die Aktualparameter werden im Environment des Aufrufs ausgewertet und die r-Werte werden zur Initialisierung der Formalparameter benutzt.
Dies ist die Standardmethode in C oder auch in Java.
Probleme gibt es nur bei „großen“ Parametern, etwa Feldern, die ja in die entsprechende lokale Version kopiert werden müssten.

call-by-reference Hat der Aktualparameter einen l-Wert, so wird dieser übergeben. Ansonsten wird der Aktualparameter ausgewertet und der Wert in einem temporären Speicherplatz, dessen Adresse übergeben wird, abgelegt.
Dies war die Standardmethode in Fortran.

call-by-copy-restore Die Aktualparameter werden ausgewertet und die r-Werte werden übergeben. Beim Verlassen der Prozedur werden die r-Werte der Formalparameter in die l-Werte der Aktualparameter (soweit vorhanden) zurückgespeichert.
Sie ist die Standardmethode in Ada und auch in einigen Fortran-Versionen.

call-by-name Jedes Auftreten eines Formalparameters wird im entsprechenden Aufruf der Prozedur durch den Aktualparameter textuell substituiert. Das entspricht bei einer Variablen als Aktualparameter einem call-by-reference. Bei komplizierteren Ausdrücken als Aktualparameter wird dieser in eine parameterlose Prozedur (**thunk**) umgewandelt und jedesmal aufgerufen, wenn der Formalparameter benötigt wird.
Dies war die Standardmethode in Algol 60 und ist wegen vieler überraschender Seiteneffekte nicht weiter verwendet worden.

Beispiel 4.4:
Man betrachte das folgende, in einer Pseudo-Sprache geschriebene Programm:

```
procedure sub (var a, b, c, d: integer);
begin
    b:=a+a;
    d:=a+c;
end;

program parameter-test;
begin
var x,y,z:integer;
    x:=1;
    y:=2;
    z:=7;
    sub(x,x,x+y,z);
    print z;
end
```

Der vom Programm ausgegebene Wert von `z` ist:

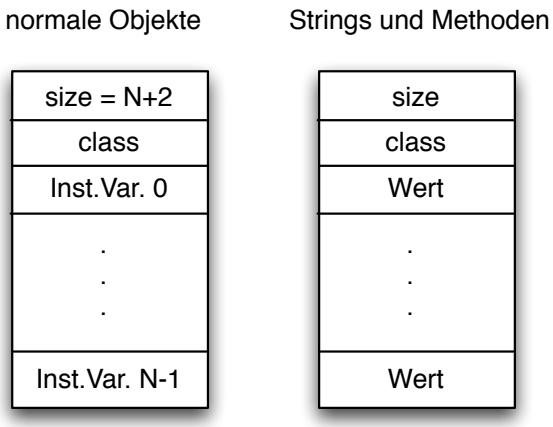
Art der Parameterübergabe	Wert
call-by-value	7
call-by-reference	5
call-by-copy-restore	4
call-by-name	6

4.6 Speicherverwaltung in objektorientierten Systemen

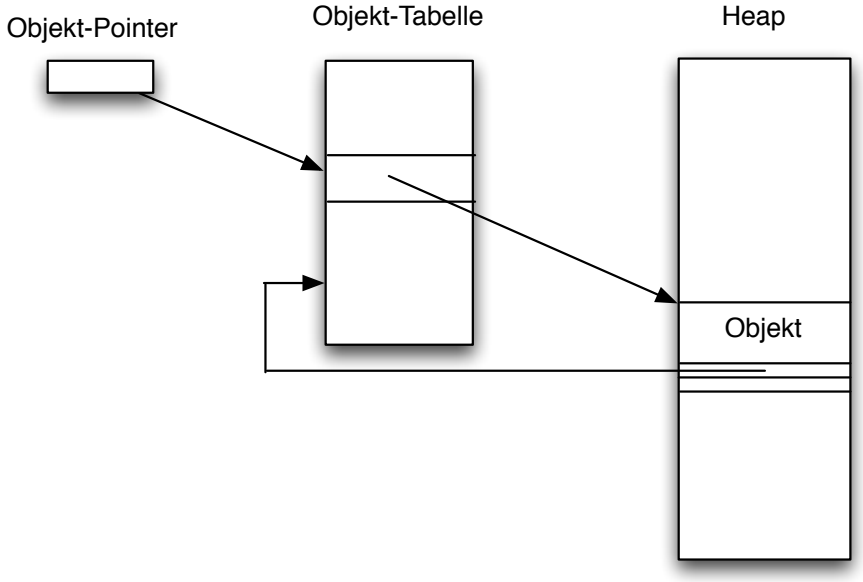
Für objektorientierte Programmiersprachen tritt der Laufzeit-Stack mit seinen Aktivierungs-Records gegenüber der Heapverwaltung in den Hintergrund, da die Lebensdauer von Objekten nicht an das Eintreten bzw. Verlassen eines Blocks gekoppelt ist.

Als Beispiel soll hier die Laufzeitumgebung für Smalltalk-80 beschrieben werden [**?**] , die charakteristisch für ein auf einer virtuellen Maschine ablaufendes objektorientiertes System ist.

Fast alle im Smalltalk-System existierenden Objekte sind im Heap gespeichert. Der Standardaufbau eines Objekts im Heap ist der folgende:

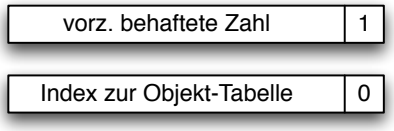


Abweichend davon werden Zeichenketten (Strings) und Methoden wie rechts dargestellt. Die Werte sind dabei entweder die Zeichen, aus denen der String zusammengesetzt ist oder aber Literale und der Bytecodes der übersetzten Methode.
Integerzahlen werden ebenfalls anders dargestellt.
Die im Heap befindlichen Objekte werden nicht direkt, sondern über eine Objekt-Tabelle indirekt adressiert. Der Grund für diese Vorgehensweise ist einfach einzusehen. Bei einer eventuell notwendigen Kompaktifizierung des freien Speichers im Heap müssen lebendige Objekte verschoben werden. Damit müssten aber auch in allen anderen Objekten, die dieses Objekt referenzieren, die entsprechende Adresse geändert werden. Dies ist natürlich viel zu aufwendig. Für jedes im Heap befindliche Objekt gibt es einen Eintrag in der Objekt-Tabelle und nur dort ist, neben anderen Informationen, die Adresse verzeichnet, ab der sich das Objekt im Speicher befindet. Eine Referenz auf dieses Objekt (ein *Objekt-Pointer*), etwa in einer Variablen oder einer Instanzenvariablen, ist nur ein Index in die Objekt-Tabelle. Muss das Objekt im Speicher verschoben werden, so muss danach nur in der Objekt-Tabelle die aktuelle Adresse nachgetragen werden.



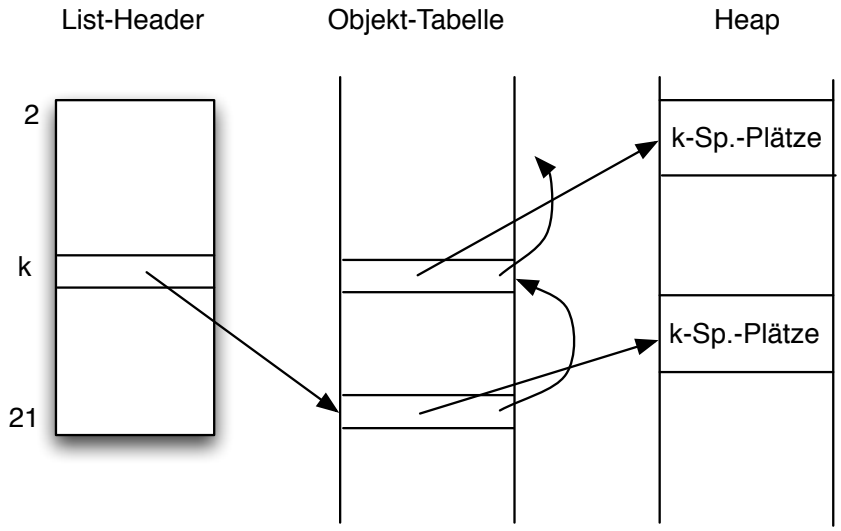
Damit ist es aber auch möglich, die Darstellung von Integer-Objekten zu beschreiben. Ist das unterste Bit eines Objekt-Pointers eine 1, so werden die restlichen Bits als vorzeichenbehaftete

Zahl, also als ein Objekt der Klasse Integer, interpretiert. Ist das unterste Bit eines Objekt-Pointers dagegen eine 0, so werden die restlichen Bits als Index in die Objekt-Tabelle betrachtet

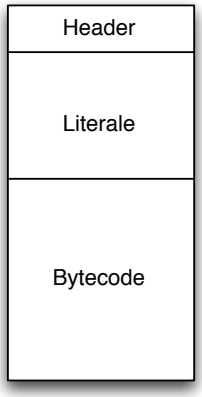


Objekt-Tabellen Einträge bestehen aus zwei Speicherplätzen, die Verwaltungsinformationen und den Ort (gegeben durch Segmentnummer und Adresse im Segment) enthalten, wo sich das Objekt im Heap befindet. Smalltalk-Systeme arbeiten üblicherweise sowohl mit Reference-Counting als auch mit einem „mark-and-sweep“ Verfahren zum Garbage-Collection. Der Reference-Count für jedes Objekt befindet sich ebenfalls in den Verwaltungsinformationen in der Objekt-Tabelle.
Freie Einträge in der Objekt-Tabelle werden durch das F-Bit markiert. Diese Einträge sind verkettet und über eine globale Variable erreichbar.
Alle Objekte im System werden auf diese Art gespeichert. Allerdings gibt es zwei Ausnahmen - Zeichenketten und übersetzte Methoden werden aus Effizienzgründen anders gespeichert. Um diese Art Objekte von den anderen unterscheiden zu können, gibt es das P-Bit. Da Objekte dieses Typ im Gegensatz zu allen anderen auch eine ungerade Zahl von Bytes im Speicher belegen können, gibt es weiterhin das O-Bit.

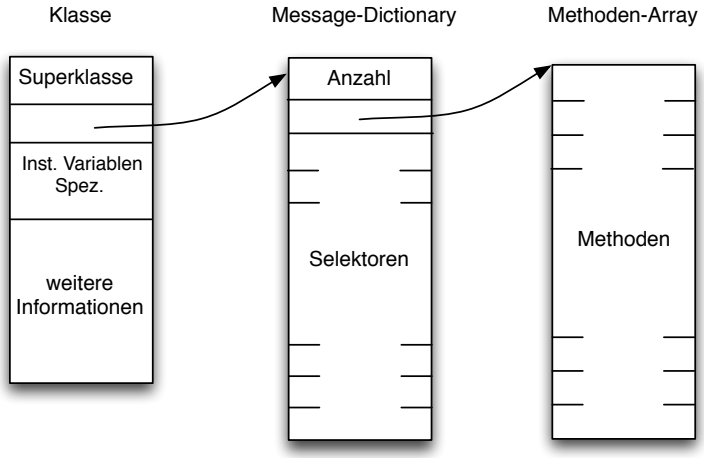
Um auf Speicheranforderungen bei Objekterzeugung schnell reagieren zu können, sind die freien Einträge in der Objekt-Tabelle wie auch die zugehörigen freien Bereiche im Heap verkettet. Es gibt dabei mehrere Ketten - für kleine Objekte, die insgesamt k (≤ 20) Speicherplätze benötigen, gibt es einen List-Header k, der auf eine verkettete Liste von Einträgen in der Objekt-Tabelle verweist, die jeweils auf frei Speicherplätze der Größe k im Heap verweisen.
Alle größeren freien Speicherbereiche (im Original > 20 Worte) sind ebenfalls auf diese Weise verkettet.



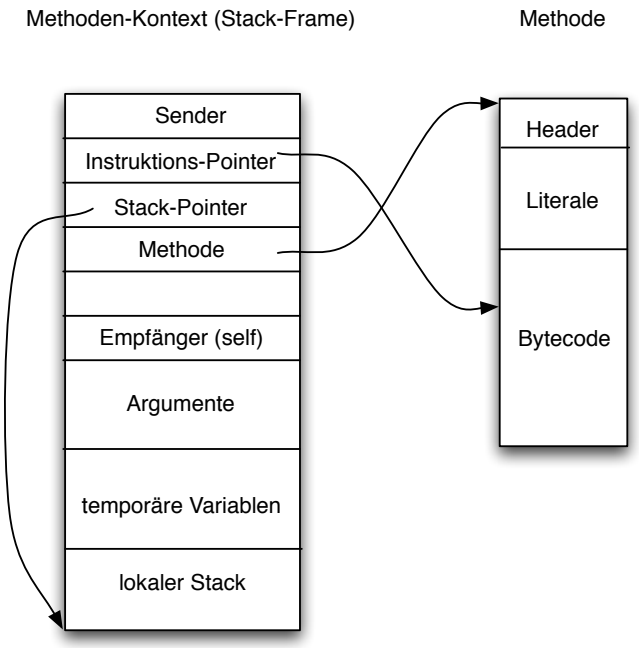
Übersetzte Methoden sind natürlich ebenfalls Objekte und befinden sich daher im Heap. Allerdings werden sie etwas anders gespeichert. Der Header enthält Informationen über die Anzahl der Argumente und die Anzahl der Literale der Methode. Ferner ist die Zahl der benötigten temporären Werte inklusive der Argumente vermerkt.



Da Klassen ebenfalls Objekte sind und die virtuelle Maschine auf die Methoden zugreifen muss, werden die Methoden in einem „Dictionary“ gespeichert werden. Der Zugriff erfolgt mittels einer Hash-Funktion über den Selektor der Nachricht.



Zur Ausführung von Methoden wird aber nun wieder ein Laufzeit-Stack benötigt. Im Smalltalk-System spricht man statt von Aktivierungs-Records von sogenannten „Kontexten“. In der Abbildung sei wieder angenommen, dass der Stack nach unten wächst.

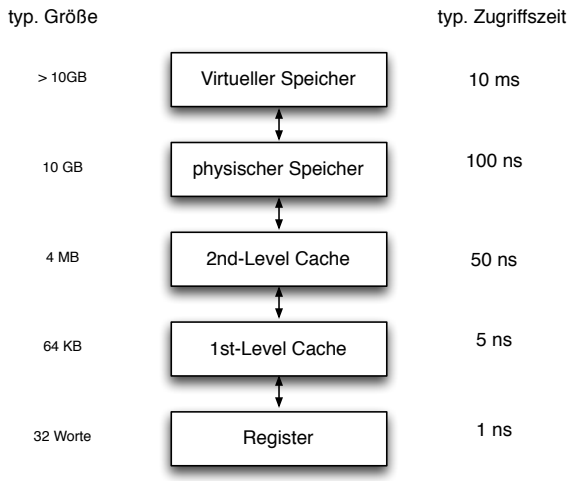

Methoden-Kontext (Stack-Frame)
Methode
Sender
Instruktions-Pointer
Stack-Pointer
Methode
Empfänger (self)
Argumente
temporäre Variablen
lokaler Stack
Header
Literale
Bytecode

4.7 Heap Management

Während der statische Speicherbereich alle Datenobjekte enthält, deren Lebensdauer sich vom Start des Programms bis zum Ende erstreckt und deren Speicherzuordnung somit vom Compiler vor dem eigentlichen Programmlauf festgelegt werden kann, wird der Heap-Speicher hauptsächlich zum Speichern von Daten und Objekten benutzt, die unabhängig von Blockeintritten oder Prozeduraufrufen zur Laufzeit des Programms erzeugt und später wieder vom Programm freigegeben werden.
Die Speicheranforderung geschieht etwa über eine `new`-Anweisung in einer Prozedur, um Speicherplatz für ein neues Datenobjekt anzufordern (Konstruktor). Dieses Datenobjekt existiert aber auch noch, wenn die Prozedur beendet wird. In einigen Sprachen muss der Programmierer selbst dafür Sorge tragen, den Speicherplatz wieder frei zu geben (etwa in in C oder C++), in anderen Sprachen gibt es ein Garbage-Collector Verfahren, das den Heap nach nicht mehr benötigten Datenobjekten durchsucht und diese wieder dem freien Heap-Speicher zuordnet.

4.7.1 Die Speicherhierarchie eines Rechners

An dieser Stelle ist es wichtig, sich etwas genauer mit dem Aufbau des Speichers zu beschäftigen.



Wie man sieht, befindet sich eventuell nur ein Teil des gesamten Programms und der Daten im Hauptspeicher. Von diesem Teil befindet sich wiederum nur ein Teil im schnelleren Cache usw. Der Transfer zwischen den verschiedenen Speicherstufen findet immer in größeren Einheiten statt, beim virtuellem Speicher vom Betriebssystem gesteuert in etwa 32 KB großen Blöcken, bei den Cache-Speichern von der Hardware gesteuert in kleineren Blöcken von etwa 256 Bytes. Die Speicher-Hardware arbeitet üblicherweise mit einer Varianten des *most recently used*-Algorithmus, das bedeutet, dass versucht wird, die zuletzt benutzen Speicherbereiche im Cachespeicher zu behalten.
Hieran erkennt man einen wichtigen Hinweis für den Programmierer und den Compilerbauer. Es ist wichtig, das Programm so zu schreiben (und zu übersetzen!), dass man größtmögliche Lokalität erreicht. Es gilt immer noch die 80-20 Regel, die besagt, dass 80% der Rechenzeit in 20% des Maschinencodes verbraucht wird. Wichtigste Kandidaten für diese 20 Prozent Maschinencode sind die Rümpfe innerer Schleifen.

4.7.2 Speicherverwaltung

Die Speicherverwaltung für den Heap-Speicher muss zwei wichtige Aufgaben erfüllen:

Allokation Wenn ein Programm Speicherplatz für ein Objekt anfordert, muss die Speicherverwaltung ein genügend großes zusammenhängendes Stück aus dem Heap-Speicher finden und einen Referenz auf diesen Speicherbereich zurückgeben. Sollte nicht ausreichend Speicher zur Verfügung stehen, kann die Speicherverwaltung eventuell zusätzlichen Speicher vom Betriebssystem anfordern und so die Speicheranforderung erfüllen.

Deallokation Die Speicherverwaltung muss Speicherbereiche, die vom Anwendungsprogramm nicht mehr benötigt werden, wieder dem Bereich des freien Speichers zuordnen.

Die Speicherverwaltung sollte diese Aufgaben schnell und effizient erfüllen und dabei selbst nicht zu viele Ressourcen verbrauchen.

Um eine Fragmentierung des Speichers zu vermeiden, werden häufig Varianten des bekannten Best-Fit oder First-Fit Verfahrens bei der Allokation von Speicher angewendet. Fordert die Programmiersprache eine manuelle Deallokation des Speichers, ist die Aufgabe der Speicherverwaltung nicht schwer. Man muss nur darauf achten, dass nebeneinander liegende freie Bereiche des Speichers wieder zu einem größeren Bereich verschmolzen werden. Allerdings hat die manuelle Deallokation einen schweren Nachteil. Bei ungenauer Programmierung kann man eventuell vergessen, nicht mehr benötigte Speicherbereiche zurückzugeben (*memory leak*) oder man referenziert bereits zurückgegebene Speicherbereiche (*dangling-pointer*). Während der erste Teil zu erhöhtem Speicherbedarf und so eventuell nach einiger Laufzeit zu einem Abbruch des Programms wegen Speichermangels führen kann, ist der zweite Fehler viel schwerer zu lokalisieren, da das Programm eventuell noch einige Zeit weiterläuft, um dann entweder fehlerhafte Daten zu liefern oder aber mit einer Exception zu enden. Ausserdem ist diese Art Fehler häufig nicht einfach reproduzierbar und somit schwer zu lokalisieren.

Eine weitere wichtige Methode der Speicherverwaltung ist das Reference Counting. Jedes dynamisch allokierte Objekt wird um einen Zähler (**reference count**) erweitert. Jedes mal, wenn eine neue Referenz zu einem Objekt erzeugt wird, inkrementiert man den Zähler; wird eine Referenz gelöscht, wird der Zähler dekrementiert. Erreicht der Reference Count eines Objektes den Wert 0, kann das Objekt deallokiert werden. Enthält das Objekt Referenzen zu anderen Objekten, so müssen die Reference Counter der anderen Objekte entsprechend dekrementiert werden. Diese Methode hat Probleme bei zirkularen Strukturen, ausserdem kostet sie bei jeder Operation, die eine Referenz verändert.

Auch bei dieser Methode muss sehr sorgfätig programmiert werden, da ein vergessenes Inkrementieren bzw. Dekrementieren zu den gleichen Problemen wie bei der manuellen Speicherverwaltung führt. Diese Methode wurde lange Zeit von Apple in Objective C verwendet. In Xcode 4.2 wurde ein automatisches Reference Counting eingeführt, dass im Wesentlichen ohne Unterstützung des Programmierers funktioniert. Die entsprechenden Befehle zum Manipulieren der Reference Counts werden dabei vom Compiler eingefügt.

Für den Programmierer ist dieses Verfahren vergleichbar mit einem Garbage Collector Verfahren. Die Grundidee ist dabei, dass Objekte, die zu einem Zeitpunkt nicht über Referenzen erreichbar sind, endgültig nicht mehr erreichbar sind und daher freigegeben werden können. Eine wichtige Voraussetzung ist dabei, dass die Programmiersprache Typ-sicher ist. Das bedeutet, dass für jede Variable der Typ des Objekts und damit die Größe des Objekts auf das sie verweist, zumindest zur Laufzeit bekannt ist. Ausserdem muss man natürlich wissen, welche Teile des Objekts wiederum Referenzen auf andere Objekte enthalten, um diese gegebenenfalls auch freizugeben. Das ist z.B für C und C++ nicht erfüllt, da in diesen Sprachen Pointer-Arithmetik zulässig ist und somit Pointer auch mitten in ein Objekt zeigen können.

4.8 Garbage Collection

Was sind die Qualitätsmerkmale eines Garbage-Collector Verfahrens?

- Das Verfahren sollte die Laufzeit des Anwendungsprogramms (in diesem Kontext spricht vom **Mutator**, da das Programm den Heap-Speicher verändert) nicht zu sehr verlangsamen. Da der Garbage Collector viele Datenobjekte untersuchen muss, sollte er gut mit dem Speichersystem zusammenarbeiten.

- Das Verfahren sollte selbst nicht zu viel Speicherplatz benötigen und die Fragmentierung des freien Speichers klein halten.

- Da das Verfahren quasi parallel zum Anwendungsprogramm läuft, sollte es keine langen Pausen in der Abarbeitung des Anwendungsprogramms produzieren. Das ist besonders wichtig für real-time Anwendungen, wo man in zeitkritischen Bereichen häufig den Garbage-Collector abschalten muss, um eine gewisse Antwortzeit zu garantieren.

- Das Verfahren sollte die Lokalität des Anwendungsprogramms verbessern oder zumindest nicht verschlechtern.

Es ist klar, dass einige dieser Merkmale in Widerspruch zueinander stehen, so dass man Kompromisse eingehen muss. Werden in einer Programmiersprache hauptsächlich viele kleine Objekte erzeugt, sollte die Allokation des Speichers schnell gehen. Die Umspeicherung (**Reallokation**) der Objekte kostet dann nicht viel. Anders sieht es aus, wenn hauptsächlich große Objekte erzeugt werden. Hier ist der Aufwand für die Reallokation beträchtlich.

4.8.1 Erreichbarkeit von Objekten

Die Basismenge der erreichbaren Objekte sind alle, die direkt, ohne Dereferenzierung eines Zeigers, vom Programm aus erreicht werden können. In Java wären dies die Klassenvariablen (static) und die Variablen auf dem Laufzeit-Stack. Alle Objekte in der Basismenge sind direkt erreichbar. Rekursiv ist dann jedes Objekt, dessen Referenz in einer Instanzvariablen eines erreichbaren Objekts gespeichert ist, ebenfalls erreichbar.
Es dürfte klar sein, dass speziell bei optimierenden Compilern, Probleme hinzukommen, da Variablenwerte zeitweise nur in einem Register gehalten werden oder erst durch Addition eines Offsets zu einer korrekten Referenz werden (*versteckte Referenzen*).

In diesem Fall muss der Compiler dem Garbage Collector helfen. Er kann

- Garbage Collection nur zu gewissen Zeitpunkten erlauben, wenn diese „versteckten“ Referenzen nicht existieren.

- dem Garbage Collector zusätzliche Informationen mitgeben oder

- er kann garantieren, dass sich alle erreichbaren Objekte immer aus der Basismenge rekursiv bestimmen lassen.

Es gibt vier Situationen, in denen der Mutator die Menge der erreichbaren Objekte verändert:

- Allokation von neuen Objekten. Die Menge erreichbarer Objekte wächst dadurch an.

- Übergabe von Parameter und Rückgabewerten. Die Menge erreichbarer Objekte bleibt in diesem Fall gleich.

- Zuweisen von Referenzen. Bei einer Zuweisung `a = b`, bei der die Typen von `a` und `b` Referenzen zu Objekten sind, bleibt das von `b` referenzierte Objekt erreichbar, sofern `a` erreichbar ist. Gleichzeitig wird das ursprünglich von `a` referenzierte Objekt jetzt einmal weniger referenziert. War die Referenzierung von `a` die letzte für das Objekt, so wird dieses Objekt und alle Objekte, die nur über dieses Objekt erreichbar waren, unerreichbar. Die Menge erreichbarer Objekte kann also kleiner werden.

- Beenden von Prozeduren. Der zugehörige Aktivierungsrecord wird frei gegeben. Gibt es in ihm Referenzen zu Objekten, die nur so erreichbar sind, werden diese Objekte ebenfalls unerreichbar und natürlich auch alle weiteren Objekte, die nur so erreichbar waren.

4.8.2 Garbage Collection über Reference Counting

Jedes Objekt hat ein zusätzliches Feld, das den Reference Count enthält. Der Zähler wird wie folgt benutzt:

- Bei Erzeugung eines Objekts wird dieser Zähler auf 1 gesetzt, da der Rückgabewert des Konstruktors eine Referenz ist.

- Wird ein Objekt als Parameter an eine Prozedur übergeben oder als Rückgabewert zurückgegeben, wird der Zähler inkrementiert.

- Bei einer Zuweisung `a = b` von Referenzen wird der Zähler des Objekts, auf das `b` verweist, inkrementiert und der Zähler des Objekts, auf das `a` verweist, dekrementiert.

- Beim Verlassen einer Prozedur müssen die Zähler aller Objekte, die über lokale Variable referenziert werden, dekrementiert werden. Verweisen mehrere lokale Variable auf das selbe Objekt, so muss der Zähler für jede Referenz dekrementiert werden.

- Wird der Reference Count eines Objekts auf 0 gesetzt, so müssen die Reference Counter alle Objekte, die über Instanzenvariablen dieses Objekts direkt erreichbar sind, ebenfalls dekrementiert werden.

Diese Methode hat den Nachteil, dass der Overhead durch einzuführende extra Operationen relativ groß ist und nicht direkt von der Zahl der Objekte abhängt. Von Vorteil ist allerdings, dass dieses Garbage Collector Verfahren inkrementell stattfindet, so dass dieses Verfahren besonders bei zeitkritischen Anwendungen Vorteile hat.
Es dürfte klar sein, dass diese Methode bei zirkulären Strukturen, bei denen kein Mitglied in der Basismenge liegt, versagt. Hier muss man besondere Vorkehrungen treffen.

4.8.3 Mark-and-Sweep Garbage Collection

Das Prinzip dieser Klasse von Garbage Collector Verfahren ist relativ simpel. Das Anwendungsprogramm (der Mutator) allokiert Speicher für neue Objekte. Nach einer gewissen Zeit, eventuell abhängig von der Größe des freien Speichers, beginnt der Garbage Collector die erreichbaren Objekte zur markieren. Dazu muss jedes Objekt Platz für ein Markierungsbit bereitstellen. Anschließend durchläuft der Garbage Collector den gesamten Heap und sammelt die nicht markierten Bereiche in einer Liste der freien Speicherbereiche, die anschließend vom Allokierungsalgorithmus wieder benutzt werden kann.
Eine Variante dieses Verfahren führt in der Sweep-Phase eine Kompaktifizierung des Speichers durch, indem es alle erreichbaren Objekte an ein Ende des Heaps verschiebt und damit einen großen zusammenhängenden freien Speicherbereich erzeugt.

Eine andere Strategie verfolgen die sogenannten Copying Garbage Collector Verfahren. Der gesamte Heap ist dabei in zwei Hälften aufgeteilt. Der Mutator arbeitet zunächst nur in einer Hälfte und allokiert dort Objekte, bis der freie Speicher in dieser Hälfte knapp wird. Der Garbage Collector kopiert dann die erreichbaren Objekte in die andere Hälfte des Heaps. Danach kann der Mutator weiter laufen wobei die Rollen der beiden Häften vertauscht sind.
Es dürfte klar sein, dass bei all diesen Verfahren das Anwendungsprogramm für einen längeren Zeitraum unterbrochen wird.
Es gibt Verfahren, die die erste oder die zweite Phase des Garbage Collecting in kleinere Prozesse zerlegen, um die Pausen für das Anwendungsprogramm nicht zu lang werden zu lassen. Da es hier aber zu Interaktionen mit dem Mutator kommen kann, müssen diese Verfahren beim Bestimmen der erreichbaren Objekte einen beträchtlichen Extra-Aufwand treiben, um zu verhindern, dass erreichbare Objekte fälschlicherweise dem freien Speicher zugeteilt werden. Werden andrerseits nicht erreichbare Objekte nicht dem freien Speicher zugeteilt und geschieht dies bei nicht zu vielen Objekten, macht das Verfahren keine Probleme, da bei der nächsten Aktivierung des Garbage Collectors diese Objekte garantiert nicht erreicht werden können (Man bedenke, dass deallokierte Objekte nie wieder vom Anwendungsprogramm referenziert werden können!).