CENG 218 Programlama Dilleri

Bölüm 10: Kontrol II - Prosedürler ve Ortamlar

Öğr.Gör. Şevket Umut Çakır

Pamukkale Üniversitesi

Hafta 12

Hedefler

- Prosedürün tanımını ve aktivasyonunun doğasını anlamak
- Prosedür semantiğini anlamak
- Parametre geçme mekanizmalarını öğrenmek
- Prosedür ortamlarını, aktivasyonlarını ve tahsis edilmesini anlamak
- Dinamik bellek yönetimini anlamak
- İstisna işleme ve ortamlar arasındaki ilişkiyi anlamak
- Tinyada'da parametre modlarını işleme koymayı öğrenmek

Giriş

- Prosedürler ve fonksiyonlar: Yürütme ertelenen ve arabirimleri açıkça belirtilen bloklar
- Birçok dil fonksiyonlar ve prosedürler arasında güçlü sözdizimsel ayrımlar yapar
- Önemli bir anlamsal ayrım için de bir durum yapabilir:
  - Fonksiyonlar bir değer üretmelidir, sadece yan etkisi yoktur.
  - Prosedürler hiçbir değer üretmez ve yan etkiler üreterek çalışır

Giriş

- Bu nedenle prosedür çağrıları komutlardır, fonksiyon çağrıları ifadelerdir.
- Çoğu dil anlamsal ayrımları zorlamaz
  - Fonksiyonlar, geri dönüş değerlerinin yanı sıra yan etkileri de üretebilir
  - Prosedürler, yan etkilere neden olurken parametreleri yoluyla değerler üretebilir
- Çoğu dilde olmadığı için aralarında önemli bir ayrım yapmayacağız.

Giriş

- Fonksiyonel Diller, bir fonksiyon kavramını genelleştirir
  - Fonksiyonların kendileri birinci sınıf veri nesneleridir
- Fonksiyonlar çöp toplama dahil dinamik bellek yönetimi gerektirir
- **Aktivasyon kaydı(Activation record)**: Bir prosedürün bir defa yürütülmesini sağlamak için ihtiyaç duyulan verilerin kümesi

Prosedür Tanımı ve Aktivasyonu

- **Prosedür**: Bir grup eylem veya hesaplama grubunu soyutlamak için bir mekanizma
- Prosedürün **gövdesi(bodsy)**: eylem grubu
- Prosedür Adı: Gövdeyi temsil eder
- Bir **tanımlama(specification)** (veya **arayüz(interface)**) ve bir gövde sağlayarak bir prosedür tanımlanır.
- **Tanımlama(Specification)**: Prosedür adını, resmi parametrelerin türlerini ve isimlerini ve geri dönüş türünü (varsa) içerir.

Prosedür Tanımı ve Aktivasyonu

- Örnek: C++ kodunda

```
void intswap(int& x, int& y){//tanımlama
    int t = x; // gövde
    x = y;     // gövde
    y = t;     // gövde
}
```

- Bazı dillerde bir prosedür tanımlaması, gövdesinden ayrılabilir
  - Örnek:

```
void intswap(int&, int&); //sadece tanımlama
```

Prosedür Tanımı ve Aktivasyonu

- Bir prosedürü, adını belirterek ve resmi parametrelerine karşılık gelen çağrı argümanlarını belirleyerek bir prosedürü **çağırırsınız(call)** (veya **etkinleştirirsiniz(activate)**).
  - Örnek: `intswap(a, b);`
- Bir prosedüre bir çağrı, **çağrılan(callee)** prosedürün gövdesinin başlangıcına kontrolünü aktarır.
- Yürütme gövdenin sonuna ulaştığında, kontrol **çağırana(caller)** döndürülür.
  - Bir `return` ifadesi kullanarak gövdenin sonundan önce geri dönülebilir

Prosedür Tanımı ve Aktivasyonu

- Örnek:

```
void intswap(int& x, int& y){
    if (x == y) return;
    int t = x;
    x = y;
    y = t;
}
```

- FORTRAN'da prosedürler **alt rutinler(subroutines)** olarak adlandırılır
- **Fonksiyonlar(Functions)**: ifadelerde belirirler ve **geri dönen değerleri(returned values)** hesaplarlar

Prosedür Tanımı ve Aktivasyonu

- Bir fonksiyon, parametrelerini ve yerel olmayan değişkenlerini değiştirebilir veya değiştirmeyebilir
- C ve C++'da, tüm prosedürler örtülü olarak işlevlerdir
  - Değer döndürmeyenler `void` olarak ilan edilir
- Ada ve FORTRAN'da prosedürler ve fonksiyonlar için farklı anahtar kelimeler kullanılır.
- Bazı diller yalnızca fonksiyonlara izin verir
  - Tüm prosedürlerin dönüş değerleri olmalıdır
  - Bu fonksiyonel dillerde yapılır

Prosedür Tanımı ve Aktivasyonu

```
-- Ada prosedürü
procedure swap (x, y: in out integer) is
    t: integer;
begin
    if (x = y) then return;
    end if;
    t := x;
    x := y;
    y :=t;
end swap;

-- Ada fonksiyonu
function max ( x, y: integer) return integer is
begin
    if (x > y) then return x;
    else return y;
    end if;
end max;
```

Prosedür Tanımı ve Aktivasyonu

- ML'de prosedür ve fonksiyon bildirimleri, sabit bildirimlere benzer bir biçimde yazılır.

```
fun swap (x, y) =
    let val t = !x
    in
        x := !y;
        y := t
    end;
```

- Bir prosedür bildirimi, **sabit bir prosedür değeri(constant procedure value)** oluşturur ve sembolik bir adı bu değerle ilişkilendirir

Prosedür Tanımı ve Aktivasyonu

- Bir prosedür, programın geri kalanıyla parametreleri aracılığıyla ve ayrıca **yerel olmayan referanslar(nonlocal references)** aracılığıyla iletişim kurar (prosedür gövdesi dışındaki değişkenlere referanslar)
- Yerel olmayan referansların anlamlarını belirleyen **kapsam kuralları(scope rules)** Bölüm 7'de ele alınmıştır.

Prosedür Semantiği

- Anlamsal olarak, bir prosedür, bildirimi yürütülmesinden ayrı olan bir bloktur.
- Ortam, bellek tahsisini belirler ve yürütme sırasında adların anlamını korur
  - Bir bloğun yerel nesneleri için ayrılan belleğe **aktivasyon kaydı(activation record)** (veya **yığıt çerçevesi(stack frame)**) denir.
  - Bloğun çalıştırılırken etkinleştirildiği söyleniyor
- Yürütme sırasında bir bloğa girildiğinde, kontrol bloğun aktivasyonuna aktarılır

Prosedür Semantiği

- Örnek: C kodunda
  - A ve B Blokları karşılaşıldıkça yürütülür
- Bir bloktan çıkıldığında, kontrol çevreleyen bloğa geri aktarılır ve çıkış bloğunun aktivasyon kaydı serbest bırakılır.

```
A: {
     int x, y;
     //...
     x = y * 10;
     B: {
          int i;
          i = x / 2;
          //...
     } /* B sonu */
} /* A sonu */
```

Prosedür Semantiği

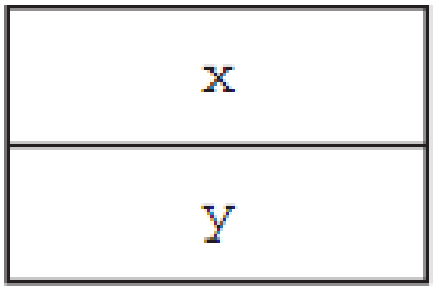


Figure 10.1 The activation record for block A

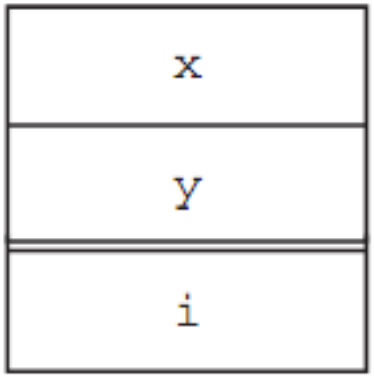


Figure 10.2 The activation records for blocks A and B

Prosedür Semantiği

- Örnek: B, A içinden çağrılan bir prosedürdür

```
1  int x;
2  void B(void) {
3      int i;
4      i = x / 2;
5      //...
6  } /* B sonu */
7  void A(void) {
8      int x, y;
9      //...
10     x = y * 10;
11     B();
12 } /* A sonu */
13 main(){
14     A();
15     return 0;
16 }
```

Prosedür Semantiği

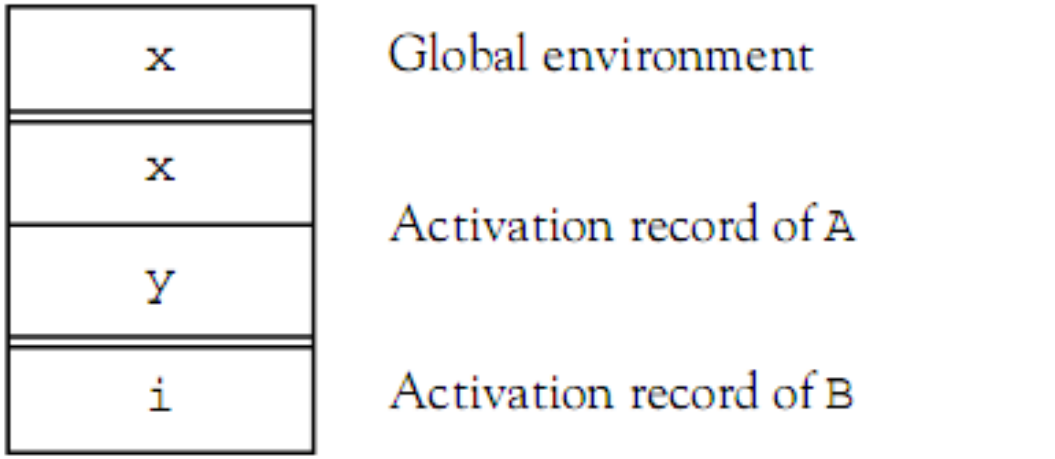


Figure 10.3 The activation records of the global environment, block A, and a procedure call of B

Prosedür Semantiği

- B'nin **tanımlayan ortamı(defining environment)**(veya **statik ortamı**), küresel ortamdır
- B'nin **çağrı ortamı(calling environment)**(veya **dinamik ortamı**), A'nın aktivasyon kaydıdır
- Prosedür olmayan bloklar için tanımlama ve çağırma ortamları her zaman aynıdır
- Bir prosedür, aynı tanımlayıcı ortamı koruyacağı herhangi bir sayıda çağrı ortamına sahip olabilir

Prosedür Semantiği

- Prosedür dışı bir blok, çevresindeki blokla yerel olmayan referanslar aracılığıyla iletişim kurar
  - Sözcüksel kapsam, çevreleyen bloktaki kendi bildirimlerinde yeniden bildirilmeyen tüm değişkenlere erişmesine izin verir
- Bir prosedür bloğu, tanımlayıcı bloğu ile yalnızca yerel olmayan değişkenlere referanslar yoluyla iletişim kurabilir.
  - Çağıran ortamdaki değişkenlere doğrudan erişmenin bir yolu yoktur
  - Çağıran ortam ile **parametreleri** aracılığıyla iletişim kurar

Prosedür Semantiği

- **Parametre listesi** prosedürün tanımı ile bildirilir
  - Prosedür çağrıldığında parametreler argümanlarla değiştirilene kadar herhangi bir değer almazlar.
- Parametreler ayrıca **biçimsel parametreler(formal parameter)** olarak adlandırılırken, argümanlar **gerçek parametreler(actual parameters)** olarak adlandırılır
- Prosedürlerin yalnızca parametrelerini kullanarak iletişim kurması gerektiği ve hiçbir zaman yerel olmayan bir değişkeni kullanmaması veya değiştirmemesi gerektiği söylenebilir.
  - Bağımlılıklardan (kullanım) veya yan etkilerden (değişim) kaçınmak için

Prosedür Semantiği

- Bu değişkenler için iyi bir kural olsa da, fonksiyonlar ve sabitler için iyi değildir
- **Kapalı form(Closed form)**: yalnızca parametrelere ve sabit dil özelliklerine bağlı prosedürler
- **Kapanış(Closure)**: bir fonksiyonun kodu ve tanımlayıcı ortamının temsili
  - İşlevin gövdesine göre tüm olağanüstü yerel olmayan referansları çözmek için kullanılabilir
  - Çalışma zamanı ortamı, gerektiğinde tüm fonksiyonlar için kapanışları hesaplamalıdır

Parametre Geçme Mekanizmaları

- argümanların parametrelere bağlanmasının doğası, prosedür çağrılarının anlamını etkiler
- Diller, mevcut parametre geçirme mekanizmalarının türleri ve izin verilen uygulama etkilerinin aralığı açısından önemli ölçüde farklılık gösterir.
- Dört mekanizma tartışılacaktır:
  - Değer olarak geçme(Pass by value)
  - Referans olarak geçme(Pass by reference)
  - Değer sonucu olarak geçme(Pass by value-result)
  - İsim olarak geçme(Pass by name)

Değer Olarak Geçme

- **Değer olarak geçme(Pass by value)**: parametre geçişi için en yaygın mekanizma
  - Argümanlar çağrı anında değerlendirilir ve değerleri parametrelerin değerleri haline gelir.
- Değere göre geçişin en basit biçiminde, değer parametreleri prosedürün yürütülmesi sırasında sabit değerler olarak davranır.
- Değer olarak geçme: yordam gövdesindeki tüm parametrelerin karşılık gelen argüman değerleriyle değiştirildiği bir işlem

Değer Olarak Geçme

- Bu mekanizma genellikle fonksiyonel dillerde kullanılan tek mekanizmadır
- C++ ve Pascal'daki varsayılan mekanizmadır ve esasen C ve Java'daki tek mekanizmadır.
- Değer olarak geçme için biraz farklı bir yorum kullanırlar
  - Parametreler, argüman değerleriyle verilen başlangıç değerleriyle prosedürün yerel değişkenleri olarak görülür.
  - Değer parametreleri atanabilir ancak prosedür dışında hiçbir değişikliğe neden olmaz

Değer Olarak Geçme

- Ada'da parametreler atanamayabilir
- Değere olarak geçme, prosedürlerin dışında değişikliklerin parametrelerin kullanılmasıyla gerçekleşemeyeceği anlamına gelmez.
  - Bir işaretçi veya başvuru türü parametresi, değeri olarak bir adres içerir ve bu, prosedürün dışındaki belleği değiştirmek için kullanılabilir.
- Örnek: C kodunda

```
void init_p(int* p){
        *p = 0;
}
```

Değer Olarak Geçme

- Not: doğrudan işaretçi parametresine atamak, prosedürün dışındaki argümanı değiştirmez.

```
void init_ptr(int * p){
        p = (int*) malloc(sizeof(int)); /* hata - bir
        ↪  etkisi yok */
}
```

- Bazı dillerde, belirli değerler örtülü olarak işaretçiler veya referanslardır
  - Örnek: C'de diziler örtülü olarak işaretçilerdir, bu nedenle dizide depolanan değerleri değiştirmek için bir dizi değeri parametresi kullanılabilir

Değer Olarak Geçme

- Java'da nesne türleri örtülü olarak işaretçilerdir, bu nedenle verilerini değiştirmek için bir nesne parametresi kullanılabilir
  - Parametrelere doğrudan atamalara izin verilmez

Referans Olarak Geçme

- **Referans olarak geçme(pass by reference)**: değişkenin konumunu ileterek parametreyi argüman için bir takma ad haline getirir
  - Parametrede yapılan herhangi bir değişiklik argümanda da meydana gelir
- FORTRAN için varsayılandır
  - Veri türünden sonra bir ve(ampersand) işareti (&) kullanılarak C++'da belirtilebilir
  - Değişken adından önce `var` anahtar sözcüğü kullanılarak Pascal'da belirtilebilir

Referans Olarak Geçme

- Örnek:
  - `inc(a)` 'ya yapılan bir çağrıdan sonra, a'nın değeri 1 artmıştır, böylece bir yan etki meydana gelmiştir.

```
void inc(int& x) {
    x++;
}
```

Şekil: Referans olarak geçme C++ örneği

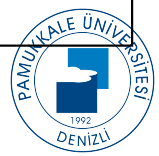
```
procedure inc(var x: integer)
begin
    x := x + 1;
end;
```

Şekil: Referans olarak geçme Pascal örneği

Referans Olarak Geçme

- Örnek: çoklu takma ad da mümkündür
  - yuck prosedürünün içinde çağrıdan sonra, `x`, `y` ve `a` aynı değişkene karşılık gelir, yani `a`

```
int a;
void yuck(int& x, int& y){
    x = 2;
    y = 3;
    a = 4;
}
//...
yuck(a, a);
```



Referans Olarak Geçme

- Bir işaretçi olarak açıkça bir referans veya konum ileterek C'de referans olarak geçme elde edebilir

```
void inc (int* x){ /* referans olarak göndermenin C
↪  benzetmesi */
    (*x)++; /* *x'e 1 ekler */
}
//...
int a;
//...
inc(&a); /* a'nın adresini inc fonksiyonuna gönder */
```

  - A değişkeninin adresini açık bir şekilde almanın ve daha sonra bunu inc gövdesi içinde açıkça başvurudan kaldırmanın gerekliliğine dikkat edin.

Referans Olarak Geçme

- Değişken olmayan argümanlara nasıl başvurulmalıdır?
- Örnek: C ++ kodunda

```
void inc(int& x)
{ x++; }
//...
inc(2); // ??
```

  - FORTRAN, geçici bir tamsayı konumu oluşturur, onu 2 değeriyle başlatır, ardından inc işlevini uygular
  - Bu, C ++ ve Pascal'da bir hatadır

Değer Sonucu Olarak Geçme

- **Değer sonucu olarak geçme(Pass by value-result)**:
  - Argümanın değeri kopyalanır ve prosedürde kullanılır
  - Prosedür çıktığında parametrenin son değeri argüman konumuna geri kopyalanır
  - **içe kopyala(copy-in)**, **dışa kopyala(copy-out)** veya **geri yükleme(copy-restore)** olarak da adlandırılır
- Değer sonucu olarak geçme, yalnızca takma ad kullanılırken referans olarak geçmeden ayırt edilebilir

Değer Sonucu Olarak Geçme

- Örnek: C kodunda
  - Referans ile geçerse, a, p çağrıldıktan sonra 3 değerine sahiptir
  - Değer sonucu olarak geçirilirse, p çağrıldıktan sonra a değeri 2'ye sahiptir

```
void p(int x, int y){
    x++;
    y++;
}
main(){
    int a = 1;
    p(a, a);
    //...
}
```

Değer Sonucu Olarak Geçme

- Ele alınması gereken konular şunları içerir:
  - Sonuçların argümanlara geri kopyalanma sırası
  - Argümanların konumlarının yalnızca girişte hesaplanıp saklanmadığı veya çıkışta yeniden hesaplanıp hesaplanmadığı
- Diğer bir seçenek de, **sonuç olarak geçme(pass by result)** mekanizmasıdır:
  - Gelen değer yok, sadece giden bir değer var

İsim Olarak Geçme ve Gecikmeli Değerlendirme

- **İsim olarak geçme(Pass by name)**: Algol60'da tanıtıldı
  - Prosedürler için bir tür gelişmiş satır içi işlem olarak tasarlanmıştır
  - Esasen normal sıralı gecikmeli değerlendirmeye eşdeğerdir
  - Uygulanması zordur ve diğer dil yapılarıyla, özellikle diziler ve atamalarla karmaşık etkileşimleri vardır
- Bölüm 3'te incelenen tembel değerlendirmenin temeli olarak anlaşılmalıdır.

İsim Olarak Geçme ve Gecikmeli Değerlendirme

- İsim olarak geçmede argüman, çağrılan prosedürde bir parametre olarak fiili kullanımına kadar değerlendirilmez.
  - Argümanın adı, karşılık geldiği parametrenin adını değiştirir
- Çağrı noktasındaki bir argümanın metni, prosedürde karşılık gelen parametre adına her ulaşıldığında değerlendirilen bir fonksiyon olarak görülür.
  - Bununla birlikte, argüman her zaman çağıran ortamda değerlendirilir

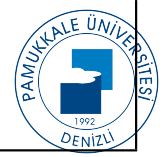
İsim Olarak Geçme ve Gecikmeli Değerlendirme

- Örnek: C kodunda
  - Bu kodun sonucu, a[1]' i değiştirmeden a[2]'ye 3 atamaktadır

```
int i;
int a[10];
void inc(int x){
    i++;
    x++;
}
main(){
    i = 1;
    a[1] = 1;
    a[2] = 2;
    inc(a[i]);
    return 0;
}
```

İsim Olarak Geçme ve Gecikmeli Değerlendirme

```
#include <stdio.h>
int i;
int p(int y) {
    int j = y;
    i++;
    return j + y;
}
void q(void) {
    int j = 2;
    i = 0;
    print("%d\n", p(i + j));
}
main() {
    q();
    return 0;
}
```



İsim Olarak Geçme ve Gecikmeli Değerlendirme

- Tarihsel olarak, isim olarak geçilen argümanların, prosedür çağrısındaki değerlendirilmesi gereken fonksiyonlar olarak yorumlanmasına **thunks** adı verilir
- Yan etkiler istendiğinde isim olarak geçme sorunludur
- Belirli durumlarda isim olarak geçmekten faydalanabilir
  - **Jensen'in cihazı(Jensen's device)**: bir işlemi dizinin tamamına uygulamak için isim olara geçmeyi kullanır

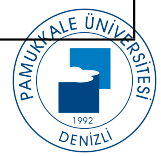
İsim Olarak Geçme ve Gecikmeli Değerlendirme

- Örnek: C kodunda Jensen'in cihazı:

```
int sum(int a, int index, int size) {
    int temp = 0;
    for (index = 0; index < size; index++)
        temp += a;
    return temp;
}
```

  - Eğer a ve index, isim olarak geçme parametreleriyse; bu kod x[0]'dan x[9]'a kadar tüm elemanların toplamını hesaplar

```
int x[10], i, xtotal;
//...
xtotal = sum(x[i], i, 10);
```



Parametre Geçme Mekanizması ve Parametre Tanımlaması

- Ada'nın parametre iletişimi için iki gösterimi vardır, `in` parametreleri ve `out` parametreleri
  - Herhangi bir parametre `in`, `out` ya da `in out` olarak bildirilebilir
    - `in` parametresi yalnızca giren değeri temsil eder
    - `out` parametresi yalnızca çıkan değeri temsil eder
    - `in out` parametresi hem giren hem de çıkan değerleri temsil eder
- Uygun değerler uygun şekilde iletildiği sürece herhangi bir parametre uygulaması kullanılabilir.
  - Girişte bir `in` değeri ve çıkışta bir `out` değeri

Parametre Geçme Mekanizması ve Parametre Tanımlaması

- Bu protokolleri ihlal eden herhangi bir program **hatalıdır(errorneous)**
  - `in` parametresine yasal olarak yeni bir değer atanamaz
  - `out` parametresi yasal olarak prosedür tarafından kullanılamaz
- Bir çevirmen, parametre tanımlamasının birçok ihlalini önleyebilir

Parametrelerin Tür Denetimi

- Güçlü türlü dillerde, argümanların tür ve sayı olarak belirtilen parametrelerle uyuştuğundan emin olmak için prosedür çağrıları kontrol edilmelidir.
- Bunun anlamı şudur ki:
  - Prosedürlerin değişken sayıda parametresi olmayabilir
  - Parametreler ve argümanlar arasındaki tür uyumluluğu için kurallar belirtilmelidir
- Referans olarak geçmek için, parametreler genellikle aynı türe sahip olmalıdır
  - Bu, değer olarak geçme için rahatlatılabilir

Prosedür Ortamları, Etkinleştirmeleri ve Tahsisi

- Sözcük kapsamına sahip blok yapılı bir dil ortamı, yığıt(stack) tabanlı bir şekilde yürütülebilir
  - Aktivasyon kaydı, bir bloğa girildiğinde ortam yığıtında(environment stack) oluşturulur ve bloktan çıkıldığında serbest bırakılır.
- Aynı yapı, tanımlama ve çağırma ortamlarının farklı olduğu prosedür aktivasyonlarına genişletilebilir.
- Yerel olmayan referansları çözmek için **kapanış(closure)** gereklidir

Prosedür Ortamları, Etkinleştirmeleri ve Tahsisi

- Programların davranışını tam olarak anlamak için bu yürütme modelini anlamalı
  - Prosedür çağrılarının semantiği bu modele yerleştirilmiştir
- Tamamen yığıt(stack) tabanlı bir ortam, prosedür değişkenleri ve prosedürlerin dinamik olarak oluşturulmasıyla başa çıkmak için yeterli değildir.
  - Bu olanaklara sahip diller (özellikle işlevsel diller), çöp toplama ile daha karmaşık tam dinamik bir ortam kullanmalıdır.

Tam Statik Ortamlar

- Fortran77'de, tüm bellek tahsisi yükleme zamanında gerçekleştirilebilir
- Programın yürütülmesi sırasında tüm değişken konumları sabittir
- Fonksiyon ve prosedür tanımları yuvalanamaz(nested)
  - Tüm prosedürler / fonksiyonlar globaldir
- Özyinelemeye izin verilmez
- Bir fonksiyon veya alt yordamla ilişkili tüm bilgiler statik olarak tahsis edilebilir

Tam Statik Ortamlar

- Her prosedür veya fonksiyon, yerel değişkenler ve parametreler için alan içeren sabit bir aktivasyon kaydına(activation record) sahiptir.
- Global değişkenler COMMON ifadeleri ile tanımlanır
  - Ortak bir alana işaretçiler tarafından belirlenirler

Tam Statik Ortamlar

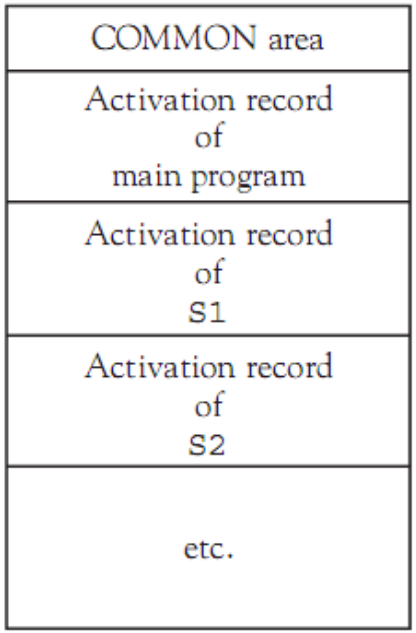


Figure 10.5 The runtime environment of a FORTRAN program with subprograms S1 and S2

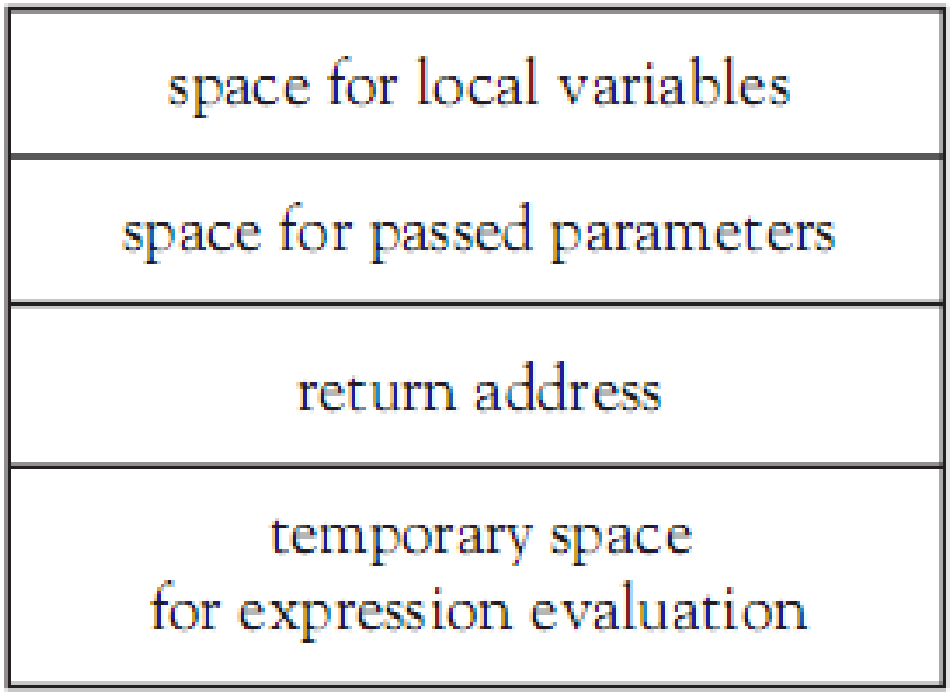


Figure 10.6 The component areas of an activation record

Tam Statik Ortamlar

```
      REAL TABLE (10), MAXVAL
      READ *, TABLE (1), TABLE (2), TABLE (3)
      CALL LRGST (TABLE, 3, MAXVAL)
      PRINT *, MAXVAL
      END

      SUBROUTINE LRGST (A, SIZE, V)
      INTEGER SIZE
      REAL A (SIZE), V
      INTEGER K
      V = A (1)
      DO 10 K = 1, SIZE
      IF (A (K) GT. V) V = A (K)
10    CONTINUE
      RETURN
      END
```

Tam Statik Ortamlar

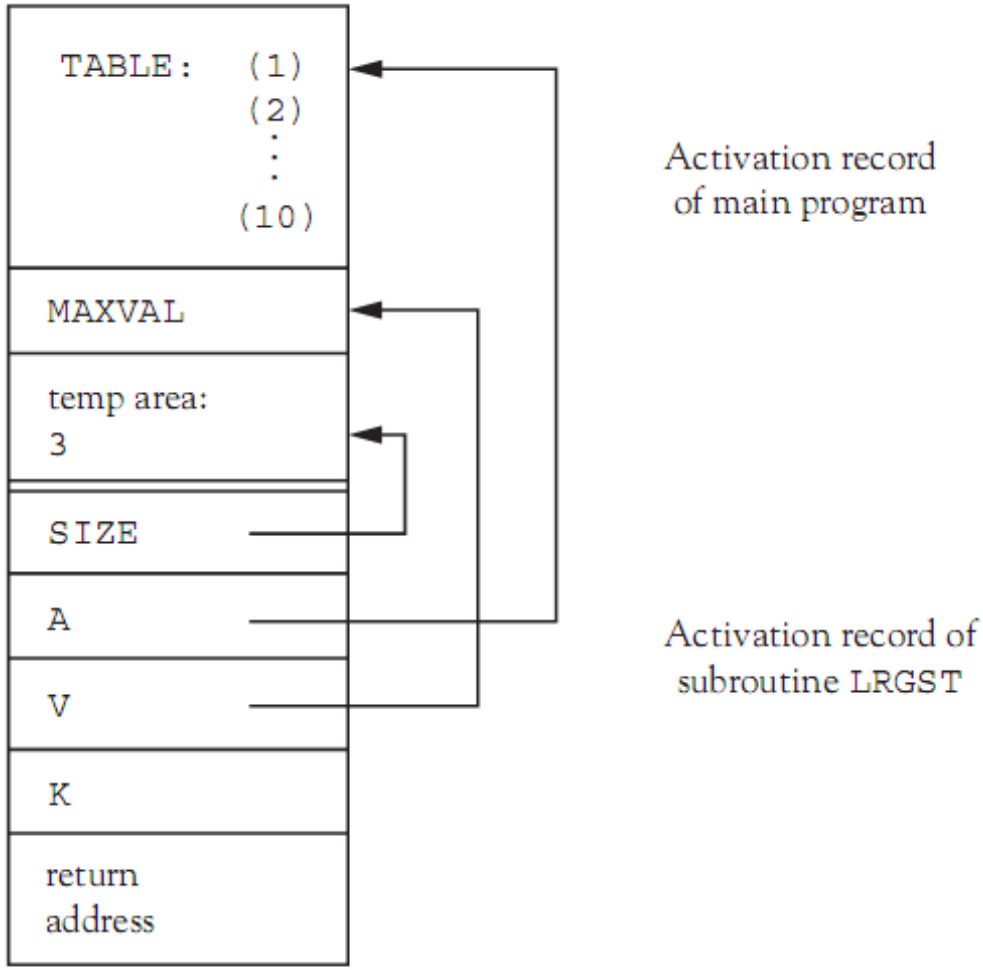


Figure 10.7 The runtime environment immediately after a call to the subroutine LRGST

Yığıt Tabanlı Çalışma Zamanı Ortamları

- Özyinelemeli blok yapılı bir dilde, prosedür bloklarının aktivasyonları statik olarak tahsis edilemez
  - Bir prosedür, önceki aktivasyonundan çıkılmadan tekrar çağrılabilir, bu nedenle her prosedür girişinde yeni bir aktivasyon yaratılmalıdır.
- Her prosedürün aktivasyon kaydı için sabit bir yeri olmadığından, mevcut aktivasyona bir işaretçi gereklidir.
  - Genellikle **ortam işaretçisi(environment pointer)** veya **ep** adı verilen bir kayıt defterinde tutulur

Yığıt Tabanlı Çalışma Zamanı Ortamları

- Ayrıca mevcut aktivasyonun girildiği bloğun aktivasyon kaydına bir işaretçi tutmalıdır.
  - Bir prosedür çağrısı ise, bu çağıranın aktivasyonudur.
- ep, önceki aktivasyona işaret edecek şekilde geri yüklenmelidir
  - Önceki konumun işaretçisine **kontrol bağlantısı(control link)** (veya **dinamik bağlantı(dynamic link)**) denir

Yığıt Tabanlı Çalışma Zamanı Ortamları

- C kodunda örnek:

```
void p(void)
{ /* ... */}
void q(void)
{
    //...
    p();
}
main()
{
    q();
    //...
}
```

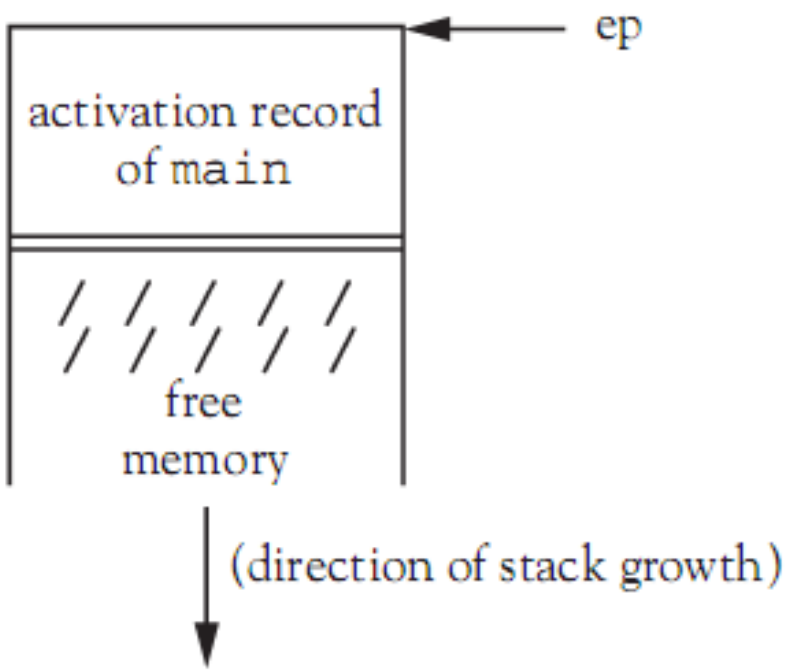


Figure 10.8 The runtime environment after `main` begins executing

Yığıt Tabanlı Çalışma Zamanı Ortamları

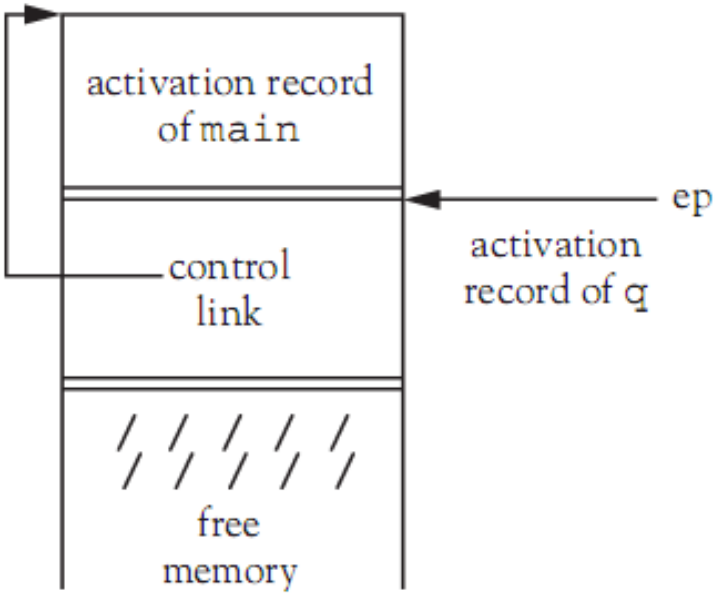


Figure 10.9 The runtime environment after q begins executing

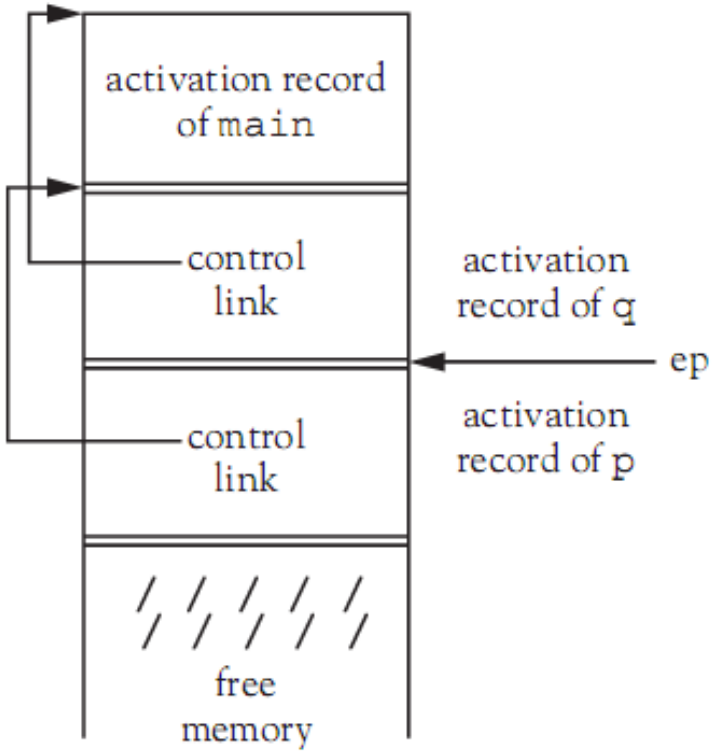


Figure10.10 The runtime environment after p is called within q

Yığıt Tabanlı Çalışma Zamanı Ortamları

- Her aktivasyon kaydındaki alanların yandaki bilgileri içermesi gerekir:

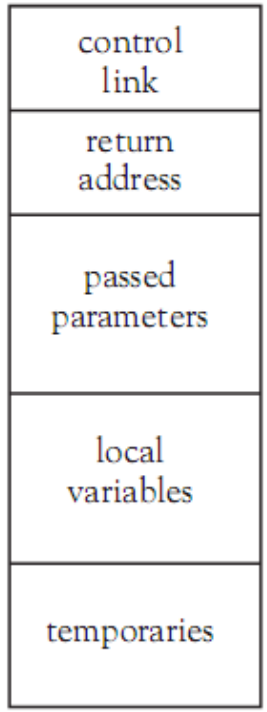


Figure 10.11 The component areas of an activation record with a control link

Yığıt Tabanlı Çalışma Zamanı Ortamları

- Yerel değişkenler, statik olduklarından her seferinde aynı sırayla geçerli etkinleştirme kaydında tahsis edilir.
- Böylece her değişken, kaydın başlangıcına göre aktivasyon kaydında aynı pozisyona tahsis edilir.
  - Buna yerel değişkenin **ötelemesi(offset)** denir

Yığıt Tabanlı Çalışma Zamanı Ortamları

```
int x;
void p(int y){
    int i = x;
    char c;
    //...
}
void q(int a){
    //...
    p(1);
}
main() {
    q(2);
    return 0;
}
```

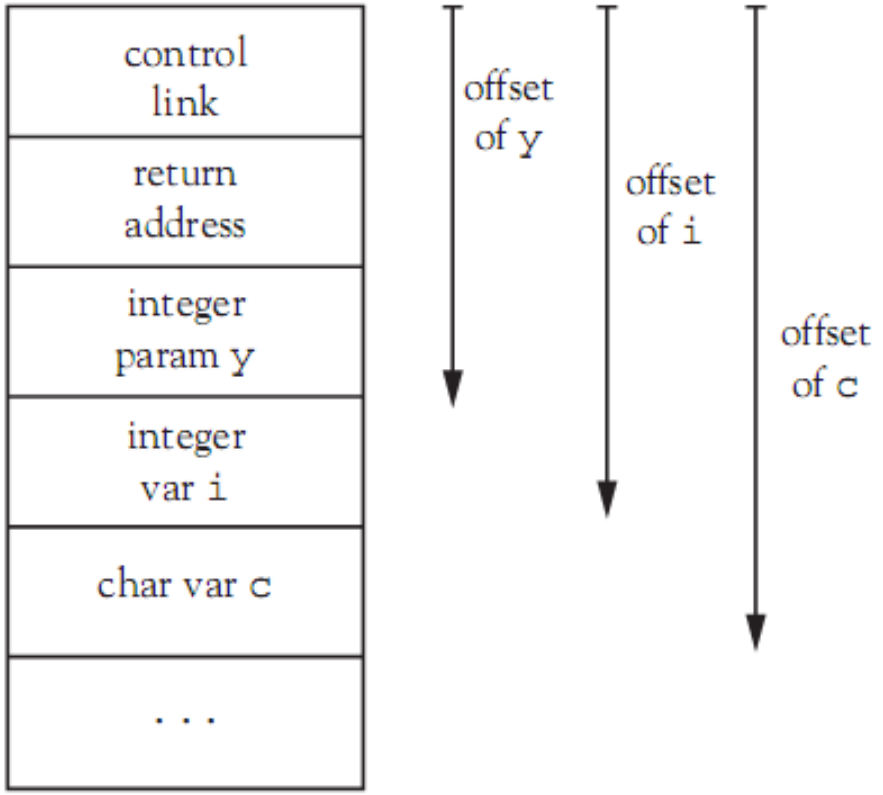


Figure 10.12 An activation record of p showing the offsets of the parameter and temporary variables

Yığıt Tabanlı Çalışma Zamanı Ortamları

- Yordamlar FORTRAN veya C'de yuvalanamadığından(nested), bir yordamın dışındaki yerel olmayan başvurular aslında geneldir ve statik olarak tahsis edilir
  - Aktivasyon yığıtında ek yapıya gerek yoktur
- İç içe yordamlara izin verildiğinde, yerel değişkenlere yerel olmayan başvurular çevreleyen bir yordam kapsamında izin verilir.

Yığıt Tabanlı Çalışma Zamanı Ortamları

```
procedure q is
  x: integer;
  procedure p(y: integer) is
    i: integer := x;
  begin
    --...
  end p;
  procedure r is
    x: float;
  begin
    p(1);
    --...
  end r;
begin
  r;
end q;
```

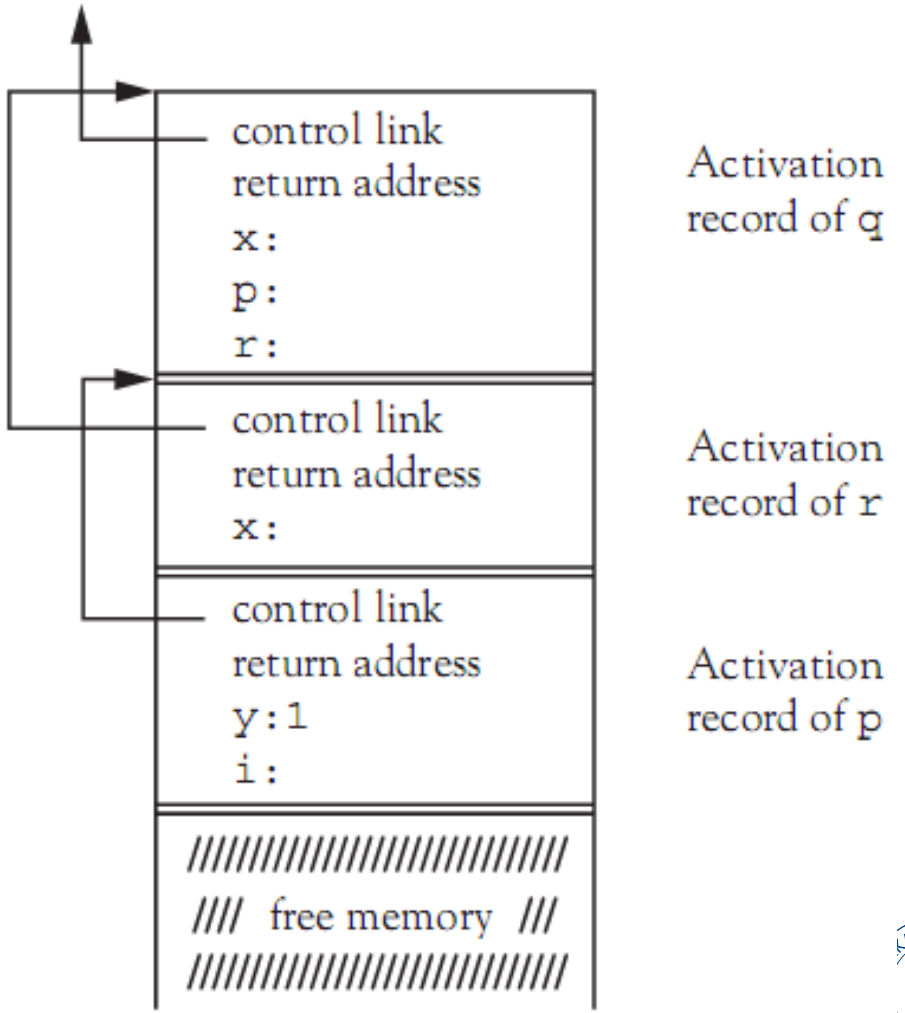


Figure 10.13 The runtime environment with activations of p, r, and q

Yığıt Tabanlı Çalışma Zamanı Ortamları

- Sözcüksel kapsam elde etmek için, bir prosedür **sözcüksel(lexical)** veya **tanımlayıcı(defining)** ortamıyla bir bağlantı sağlamalıdır.
  - Bu bağlantıya **erişim bağlantısı(access link)** (veya **statik bağlantı(static link)**) denir
- Her aktivasyon kaydının bir erişim bağlantı alanı olması gerekir
- Bloklar derinlemesine iç içe geçtiğinde, yerel olmayan bir referans bulmak için birden fazla erişim bağlantısını izlemek gerekebilir.

Yığıt Tabanlı Çalışma Zamanı Ortamları

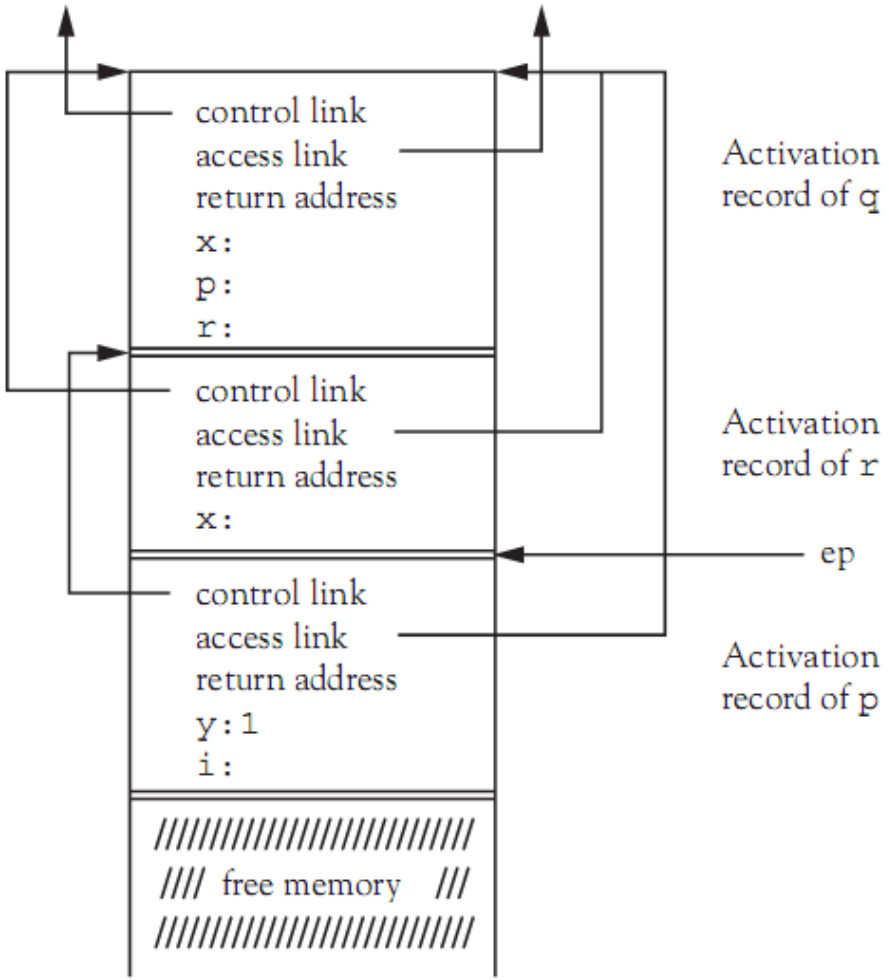


Figure 10.14 The runtime environment with activations of p, r, and q with access links

Yığıt Tabanlı Çalışma Zamanı Ortamları

- Ada kodunda örnek:
  - X'e q içinden erişmek için, q'nun aktivasyon kaydındaki erişim bağlantısını takip etmelisiniz.
  - Ardından, p aktivasyon kaydına erişim bağlantısını kullanın.
  - Ardından küresel ortama erişim bağlantısını takip edin

```
procedure ex is
  x: ...;
  procedure p is
    ...
    procedure q is
    begin
      ... x ...;
    end q;
  begin -- p
  end p;
begin -- ex
  ...
end ex;
```

Yığıt Tabanlı Çalışma Zamanı Ortamları

- Bu işleme **erişim zincirleme(access chaining)** adı verilir
- İzlenmesi gereken erişim bağlantılarının sayısı, erişim ortamı ile erişilen değişkenin tanımlayıcı ortamı arasındaki yuvalama seviyeleri (veya **yuvalama derinliği(nesting depth)**) farkına karşılık gelir.
- Kapanış(Closure) (yerel olmayan referansları çözme mekanizmasıyla birlikte prosedür kodu) önemli ölçüde daha karmaşıktır

Yığıt Tabanlı Çalışma Zamanı Ortamları

- Kapanış için iki işaretçi gerekir:
  - Kod veya talimat işaretçisi(instruction pointer) (ip)
  - Erişim bağlantısı veya tanımlayıcı ortamının ortam işaretçisi (ep)
- Kapanış <ep, ip> ile gösterilir

Yığıt Tabanlı Çalışma Zamanı Ortamları

```
1  with Text_IO; use Text_IO;
2  with Ada.Integer_Text_IO;
3  use Ada.Integer_Text_IO;
4  procedure lastex is
5    procedure p(n: integer) is
6      procedure show is
7      begin
8        if n > 0 then p(n - 1);
9        end if;
10       put(n);
11       new_line;
12     end show;
13   begin -- p
14     show;
15   end p;
16 begin -- lastex
17   p(1);
18 end lastex;
```

Şekil: Yuvalanmış prosedürlerin bulunduğu bir Ada programı

Yığıt Tabanlı Çalışma Zamanı Ortamları

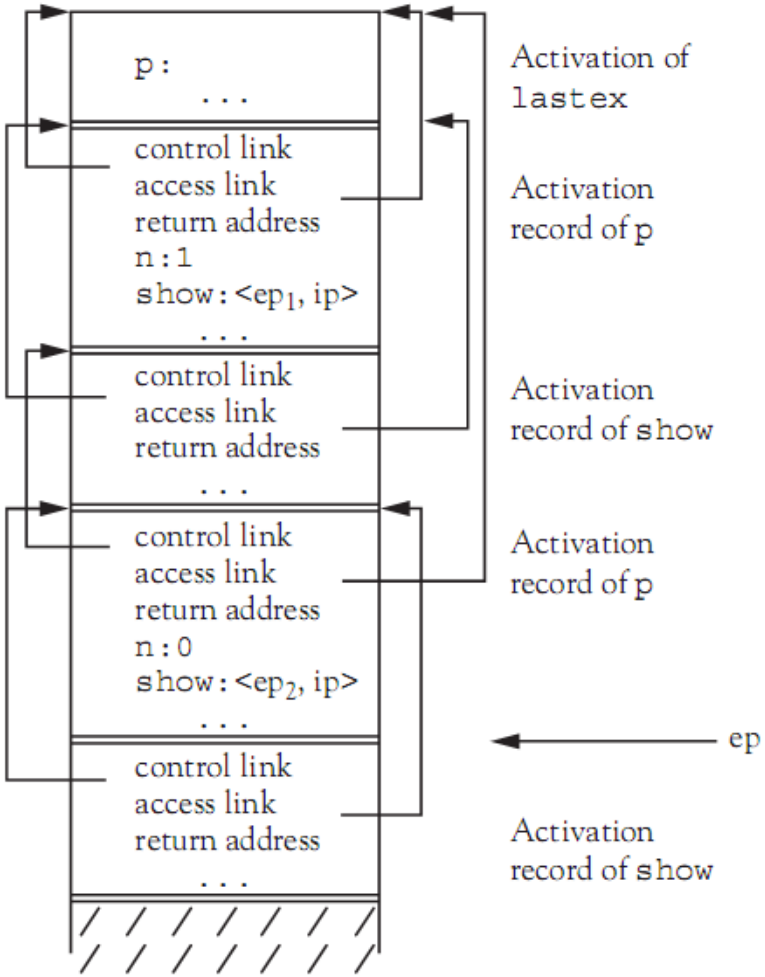


Figure 10.16 Environment of `lastex` during the second call to `show` (line 8 of Figure 10.15)

Dinamik Olarak Hesaplanan Prosedürler ve Tam Dinamik Ortamlar

- Prosedürler için <ep, ip> kapanışlarının kullanılması, bu parametreler değer parametreleri olduğu sürece, parametre olarak iletilen prosedürlerin bulunduğu diller için bile uygundur.
  - Parametre olarak geçirilen prosedür bir <ep, ip> çifti olarak geçirilir
  - Erişim bağlantısı, kapanışın ep kısmıdır
- Bu yaklaşım Ada ve Pascal'da kullanılmaktadır.
- Yığıt tabanlı bir ortamın sınırlamaları vardır

Dinamik Olarak Hesaplanan Prosedürler ve Tam Dinamik Ortamlar

- Bir işaretçiyi yerel bir nesneye döndürebilen herhangi bir prosedür, prosedürden çıkıldığında sarkan bir referansla sonuçlanacaktır.
- C kodunda örnek:

```
int * dangle(void) {
    int x;
    return &x;
}
```

- `addr = dangle()` ataması, `addr`'nin aktivasyon yığıtında güvenli olmayan bir konuma işaret etmesine neden olur

Dinamik Olarak Hesaplanan Prosedürler ve Tam Dinamik Ortamlar

- Java buna izin vermiyor
- Ada95, **Erişim Türü Yaşam Süresi Kuralını(Access-type Lifetime Rule)** belirterek bunu bir hata yapar:
  - T erişim türüne ait bir sonuç veren bir x'access özelliğine yalnızca x en az T kadar uzun süre varlığını sürdürebiliyorsa izin verilir.
- Prosedürler dinamik olarak oluşturulabilir ve diğer prosedürlerden döndürülebilirse, birinci sınıf değerler haline gelirler
  - Bu esneklik genellikle fonksiyonel bir dilde istenir

Dinamik Olarak Hesaplanan Prosedürler ve Tam Dinamik Ortamlar

- Yerel olarak tanımlanmış bir prosedürün kapanışı, mevcut aktivasyon kaydına işaret eden bir ep'ye sahip olacağından, yığın tabanlı bir ortam kullanılamaz.
- Bu kapanış, oluşturma prosedürünün aktivasyonunun dışında mevcutsa, ep artık mevcut olmayan bir aktivasyon kaydına işaret edecektir.

Dinamik Olarak Hesaplanan Prosedürler ve Tam Dinamik Ortamlar

```
type WithdrawProc is
  access function (x: integer) return integer;
InsufficientFunds: exception;
function makeNewBalance (initBalance: integer)
  return WithdrawProc
is
  currentBalance: integer;
  function withdraw (amt: integer) return integer is
  begin
    if amt <= currentBalance then
      currentBalance := currentBalance - amt;
    else
      raise InsufficientFunds;
    end if;
    return currentBalance;
  end withdraw;
begin
  currentBalance := initBalance;
  return withdraw'access;
end makeNewBalance;
```

Dinamik Olarak Hesaplanan Prosedürler ve Tam Dinamik Ortamlar

- Aşağıdaki kod yürütüldükten sonra:

```
withdraw1, withdraw2: WithdrawProc;
withdraw1 := makeNewBalance(500);
withdraw2 := makeNewBalance(100);
```

  - Aşağıdaki kod yürütülürse:

```
newBalance1 := withdraw1(100);
newBalance2 := withdraw2(50);
```

  - `newBalance1` için 400, `newBalance2` için 50 değerini almalıyız

- Yerel `currentBalance` değişkeninin iki örneği ortamdan kaybolduysa, bu çağrılar çalışmayacaktır.

Dinamik Olarak Hesaplanan Prosedürler ve Tam Dinamik Ortamlar

- LISP'de, fonksiyonlar ve prosedürler birinci sınıf değerlerdir
  - Hiçbir genellik veya ortogonalite olmamalıdır
- **Tamamen dinamik(Fully dynamic)** ortam: aktivasyon kayıtları, yalnızca çalıştırılan program içinden artık ulaşılamadığında silinir
- Ulaşılamayan depolama alanını geri kazanabilmelidir
  - Bunun için iki yöntem **referans sayıları(reference counts)** ve **çöp toplamadır(garbage collection)**.

Dinamik Olarak Hesaplanan Prosedürler ve Tam Dinamik Ortamlar

- Aktivasyon kayıtlarının yapısı yığın gibi değil ağaç gibi olur
  - Çağıran ortama kontrol bağlantıları, artık mutlaka hemen önceki aktivasyona işaret etmiyor
- Bu, Scheme ve diğer fonksiyonel dillerin uygulandığı modeldir.

Dinamik Olarak Hesaplanan Prosedürler ve Tam Dinamik Ortamlar

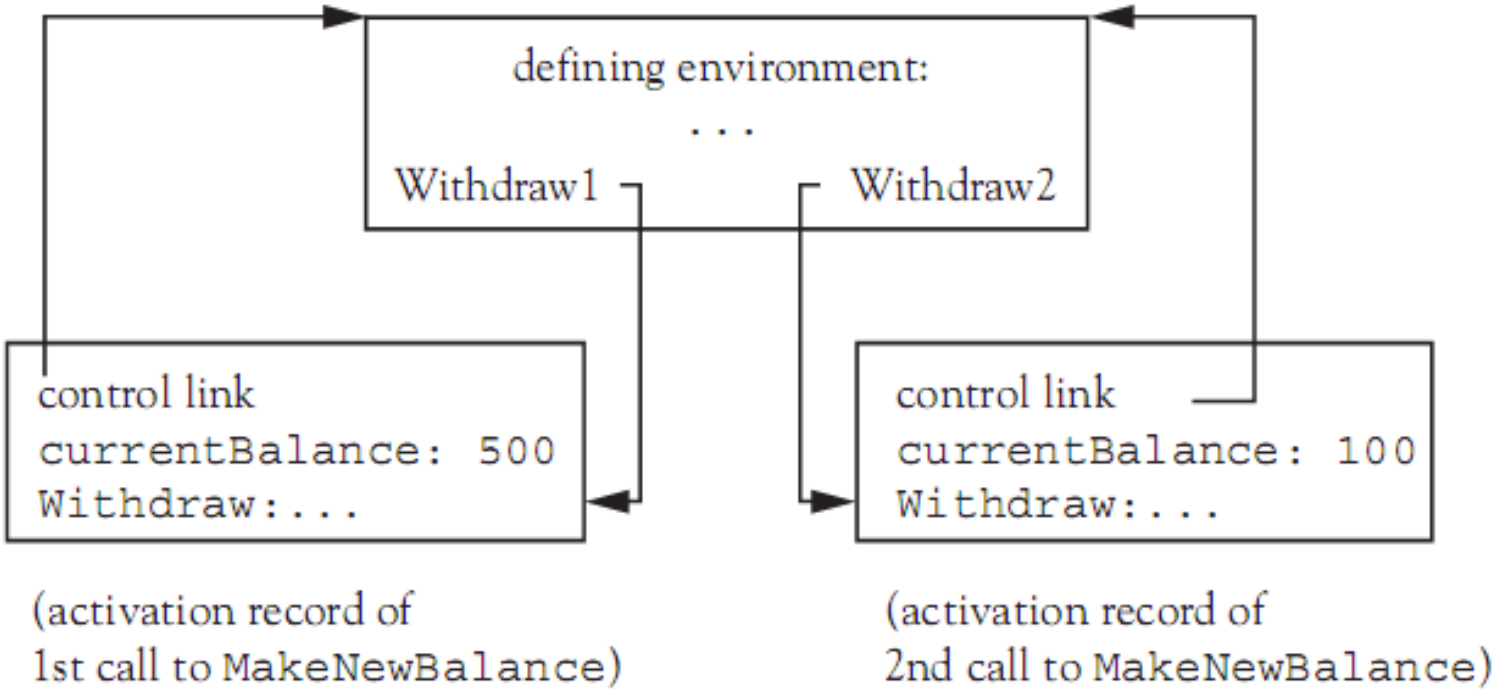


Figure 10.17 A tree-like runtime environment for a language with first-class procedures

Dinamik Bellek Yönetimi

- C gibi tipik bir zorunlu dilde, depolamanın otomatik olarak tahsisi ve serbest bırakılması yalnızca bir yığıt(stack) üzerindeki etkinleştirme kayıtları için gerçekleşir.
- Açık dinamik tahsis ve işaretçilerin kullanımı, yığıttan(stack) ayrı bir bellek **yığını(heap)** kullanılarak manuel programlayıcı kontrolü altında mevcuttur.
  - Yığın(heap) için otomatik çöp toplama arzu edilir
- Prosedürlerin ve fonksiyonların kullanımına önemli kısıtlamalar uygulamayan herhangi bir dil, otomatik çöp toplama sağlamalıdır.

Dinamik Bellek Yönetimi

- Otomatik bellek yönetiminin iki kategorisi vardır:
  - Artık kullanılmayan depolama alanının **geri kazanılması(reclamation)** (daha önce **çöp toplama(garbage collection)** olarak adlandırılıyordu)
  - Tahsis için mevcut boş alanın **bakımı(maintenance)**

Boş Alanın Bakımı

- Yürütülen bir programa sağlanan bitişik bir bellek bloğundaki boş alan, boş blokların bir listesi kullanılarak korunur.
  - Bir bağlı liste aracılığıyla yapılabilir

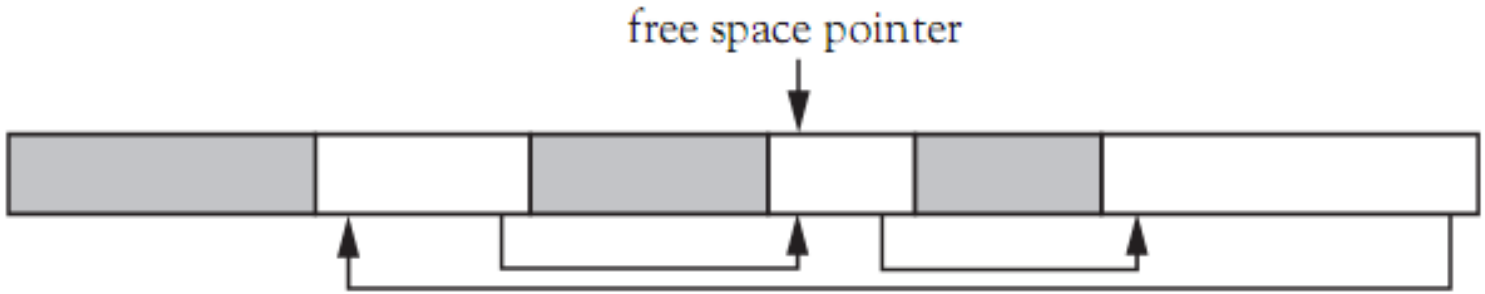


Figure 10.18 The free space represented as a linked list

Boş Alanın Bakımı

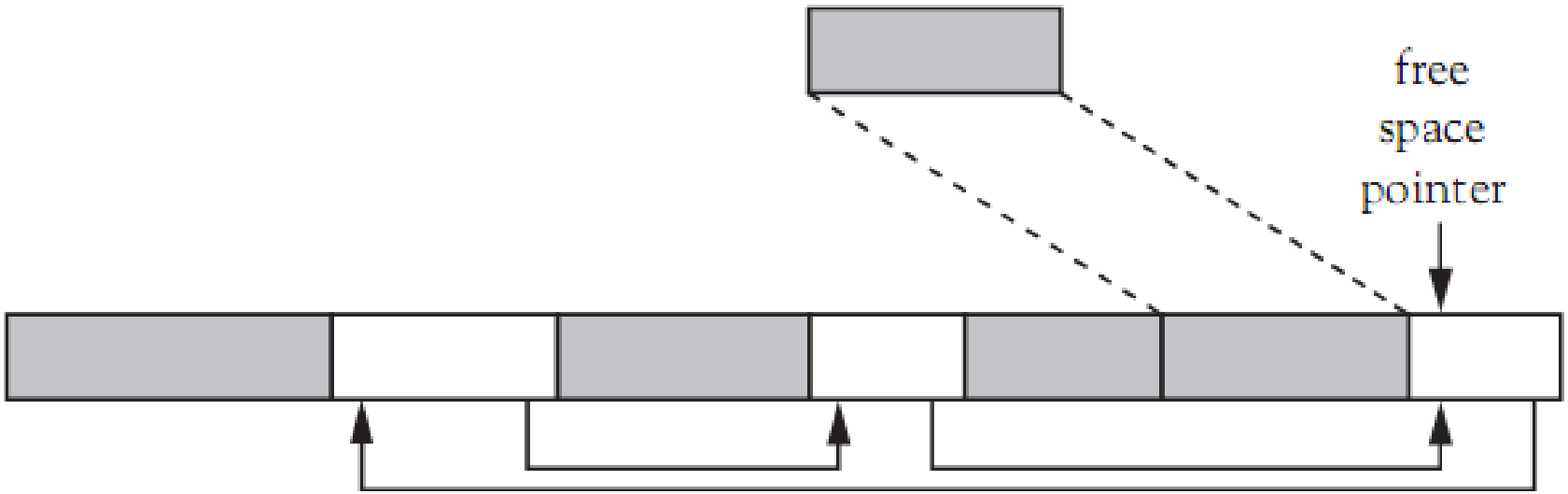


Figure 10.19 Allocating storage from the free space

Boş Alanın Bakımı

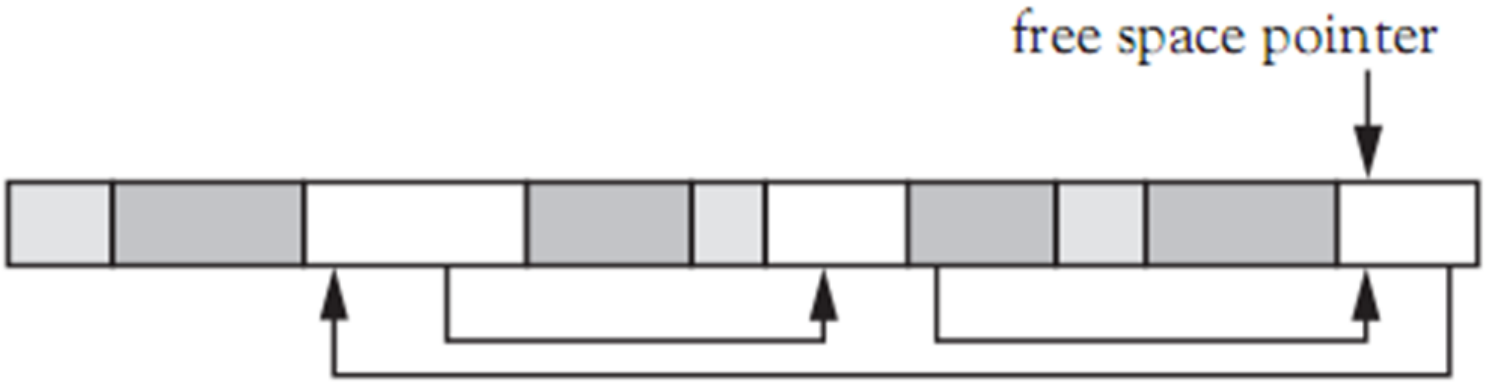


Figure 10.20 Returning storage to the free list

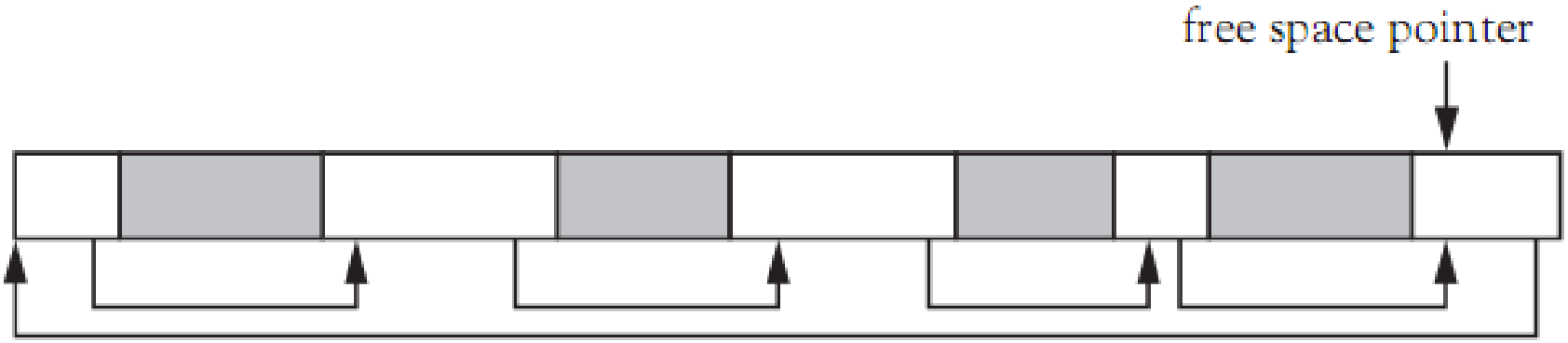


Figure 10.21 The free list after the return of storage

Boş Alanın Bakımı

- **Birleştirme(Coalescing)**: boş belleğin en büyük bitişik bloğunu oluşturmak için hemen bitişik boş bellek bloklarını birleştirme işlemi
- Serbest bir liste **parçalanabilir(fragmented)**
  - Bu, büyük bir bloğun tahsisinin başarısız olmasına neden olabilir
- Bellek, tüm serbest bloklar bir araya getirilerek ve tek bir blok halinde birleştirilerek zaman zaman **sıkıştırılmalıdır(compacted)**.
- Depolama sıkıştırması önemli ölçüde ek yük gerektirir

Boş Alanın Bakımı

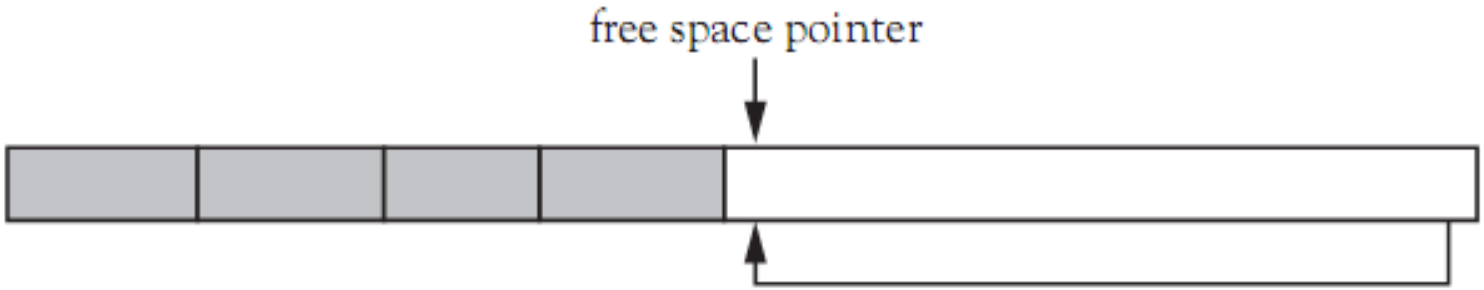


Figure 10.22 The free list after compaction

Depolama Geri Kazanımı

- Bir depolama bloğuna artık başvurulmadığında bunu tanımanın iki ana yöntemi:
  - Referans sayma
  - İşaretle ve süpür
- **Referans sayma(Reference counting)**: istekli bir geri kazanma yöntemi
  - Her tahsis edilmiş depolama bloğu, diğer bloklardan bu bloğa yapılan referansların sayısını depolayan bir alan içerir.
  - Sayı 0'a düştüğünde, blok serbest listeye geri döndürülebilir

Depolama Geri Kazanımı

- Referans saymanın sakıncaları:
  - Sayımları tutmak için ekstra bellek gerekli
  - Sayımların bakım eforu oldukça fazladır, çünkü referans sayıları özyinelemeli olarak azaltılmalıdır.
  - Dairesel başvurular, başvurulmayan belleğin asla ayrılmamasına neden olabilir

Figure 10.23 A circular linked list

Depolama Geri Kazanımı

- **İşaretle ve süpür(Mark and sweep)**: tahsis edicide yer kalmayıncaya kadar herhangi bir depolamayı geri almayı erteleyen tembel bir yöntem
  - Referans gösterilebilecek tüm depolamayı arar
  - Referans gösterilmeyen tüm depolamayı boş listeye geri taşır
- İşaretleme ve süpürme iki geçişte gerçekleşir:
  - İlk geçiş, tüm işaretçileri özyinelemeli olarak takip eder ve ulaşılan her depolama bloğunu işaretler
  - İkinci geçiş, bellekte doğrusal olarak gezinir ve işaretlenmemiş blokları boş listeye geri döndürür

Depolama Geri Kazanımı

- İşaretleme ve süpürmenin dezavantajı:
  - Hafızadan çift geçiş, işlemde birkaç dakikada bir meydana gelebilecek önemli bir gecikmeye neden olur
- Bir alternatif, kullanılabilir belleği ikiye bölmek ve depolamayı bir seferde yalnızca yarısından ayırmaktır.
  - İşaretleme geçişi sırasında ulaşılan bloklar diğer yarısına kopyalanır
  - Dur ve kopyala olarak adlandırılır
  - İşlem gecikmelerini çok az iyileştirir

Depolama Geri Kazanımı

- **Nesilsel çöp toplama(Generational garbage collection)**: 1980'lerde icat edildi
  - Önceki düzene kalıcı bir depolama alanı ekler
  - Yeterince uzun süre hayatta kalan tahsis edilmiş nesneler kalıcı alana kopyalanır ve sonraki geri kazanma işlemleri sırasında asla tahsisi kaldırılmaz.
  - Aranacak bellek miktarını azaltır

  Bu yöntem, özellikle sanal bellek sistemiyle çok iyi çalışır

İstisna İşleme ve Ortamlar

- İstisnaların oluşturulması ve işlenmesi prosedür çağrılarına benzer ve benzer şekillerde uygulanabilir
- Ancak bazı farklılıklar vardır:
  - Bir işleyici ararken yığın çözülebileceğinden(unwound), bir istisna oluşturmak için yığın üzerinde bir aktivasyon oluşturulamaz.
  - Bir işleyici bulunmalı ve dinamik olarak çağrılmalıdır
  - İşleyici eylemleri istisna değerine değil istisna türüne dayandığından istisna türü bilgileri saklanmalıdır.

İstisna İşleme ve Ortamlar

- Bir istisna ortaya çıktığında, hiçbir aktivasyon kaydı oluşturulmaz
  - İstisna nesnesi ve tür bilgileri, bir kayıt gibi bilinen bir konumda depolanır.
  - Bir işleyici arayan genel koda bir atlama gerçekleştirilir
  - İşleyici bulunmazsa çıkış kodu çağrılır
  - Bir istisnanın başarılı bir şekilde işlenmesi için iade adresi de bilinen bir yerde saklanmalıdır.
- Bu süreç ilk farkı ele alıyor

İstisna İşleme ve Ortamlar

- İkinci farkla başa çıkmak için, işleyicilere işaretçiler bir tür yığıt(stack) üzerinde tutulmalıdır.
  - İlişkili bir işleyiciye sahip bir prosedüre girildiğinde, işleyiciyi gösteren bir işaretçi yığıta(stack) kaydedilmelidir.
  - Çıkıldığında, işleyici işaretçisinin yığıttan(stack) çıkarılması gerekir
- Bu yığıtı(stack) doğrudan uygulamak için, ya yığın(heap) üzerinde ya da kendi bellek alanında başka bir yerde muhafaza edilmelidir (çalışma zamanı yığıtında(stack) değil)

İstisna İşleme ve Ortamlar

- Üçüncü fark, istisna yapılarının kendisinde ek yük eklemeden gerekli tür bilgisinin nasıl kaydedileceği ile ilgilidir.
  - Bir arama tablosu kullanılabilir
- İşleyicilerin temel uygulaması nispeten basittir
  - Belirli bir bloğun tüm işleyici kodunu, switch ifadesi olarak uygulanan tek bir işleyicide toplayın
- Ana sorun, işleyici yığınının bakımının potansiyel olarak önemli bir çalışma süresi cezasına neden olmasıdır.

İstisna İşleme ve Ortamlar

- Örnek:

```
void factor() throw (Unwind, InputError) {
    try
    { /*...*/ }
    catch (UnexpectedChar u)
    { /*...*/ }
    catch (Unwind u)
    { /*...*/ }
    catch (NumberExpected n)
    { /*...*/ }
}
```

Örnek Olay: TinyAda'da Parametre Modlarını İşleme

- TinyAda'da:
  - Bir tanımlayıcı bir parametre olabilir
  - Parametreler, değişkenlerle aynı kısıtlamalara tabidir
  - Statik ifadelerde ne parametreler ne de değişkenler görünebilir
- Parametreler, oynadıkları roller açısından, parametre modları olarak da tanımlanabilir.
  - Modlar, anlambilimsel analiz sırasında yeni bir dizi kısıtlama uygulamamıza izin verir

Parametre Modlarının Sözdizimi ve Anlambilimi

- TinyAda'nın parametre modları için sözdizimi kuralları:

```
parameterSpecification = identifierList ":" mode <type>name

mode = [ "in" ] | "in" "out" | "out"
```

Table 10.1 The static semantics of TinyAda's parameter modes

Parameter Mode	Role in a Procedure	Semantic Restrictions
`in`	Input only	May appear only in expressions, or only in the position of an `in` parameter when passed as an actual parameter to a procedure.
`out`	Output only	May appear only as the target of an assignment statement, or only in the position of an `out` parameter when passed as an actual parameter to a procedure.
`in out`	Input and output	May appear anywhere that a variable may appear.

Parametre Bildirimlerini İşlemek

- SymbolEntry sınıfı, bir mod alanı içerecek şekilde değiştirildi
  - Olası değerler SymbolEntry.IN, SymbolEntry.OUT ve SymbolEntry.IN_OUT

Parametre Referanslarını İşlemek

- İfadedeki bir isim bir parametre ise, modu OUT olamaz.
- Bir atama ifadesinin hedefindeki bir ad bir parametre ise, modu IN olamaz
- Gerçek parametrelerin sayısı ve türleri, prosedürün resmi parametreleriyle eşleştirilmelidir.
  - Yeni kısıtlamaların uygulanması için mod artık incelenmelidir.