: B B M 3 0 1 - P R O G R A M L A M A D I L L E R I A R A S I N A V 2 :

1) *Ders05 - Names, Bindings, Type Checking and Scopes:*

- **Bellek (Memory),** hem talimatları hem de verileri depolar.
- **İşlemci (Processor),** belleğin içeriğini değiştirmek için işlemler sağlar.
- **Soyutlama;** Bellek için soyutlamalar **değişkenlerdir.**
- **İsimler (Names);**
  - ✓ ***Uzunluk:*** ***Önceki diller*** - Tek karakter, ***Fortran 95*** - En fazla 31 karakter, ***C#, Ada, Java*** - Sınırsız.
  - ✓ ***İsim Formları:*** Çoğu Programlama Dillerindeki isimler aynı biçime sahiptir. Harf ile başlar ve harf, rakam veya alt çizgi karakterleriyle devam eder. Bazılarında, bir değişkenin isminden önce özel karakter kullanılır. Bugünlerde çok popüler olan büyük-küçük harf kullanımı -> **Ör:** *toplamaFonksiyonu.* Fortran'ın eski sürümlerinde boşluklar göz ardı edilirdi -> **Ör:** T*oplama Fonksiyonu = ToplamaFonksiyonu.*
  - ✓ ***Özel Karakterler:*** ***PHP*** - Tüm değişkenler $ işareti ile başlamalıdır, ***Perl*** - Tüm değişken isimleri, değişken türünü belirten özel karakterler ($, @, %) ile başlar.
  - ✓ ***Büyük-Küçük Harf Duyarlılığı:*** Birçok dilde isimlerdeki büyük ve küçük harfler farklıdır -> **Ör:** Gül, gül, GÜL
  - ✓ ***Özel kelimeler:*** Program tarafından ayrılmış isimler (for, while, ..) değişken ismi olarak kullanılamaz.
- **Değişkenler;**
  - ✓ ***Değişken Öznitelikleri - İsim:*** Çoğu değişken isimlendirilir.
  - ✓ ***Değişken Öznitelikleri - Adres:*** İlişkili olduğu hafıza adresi. Aynı ismin farklı yerleri işaret etmesi mümkündür.
  - ✓ ***Değişken Öznitelikleri - Takma Adlar (Aliases):*** Birden çok tanımlayıcı aynı adresi referans alır. Aynı bellek konumuna erişmek için birden fazla değişken kullanılır. Bu tanımlayıcı isimler takma ad olarak adlandırılır.
  - ✓ ***Değişken Öznitelikleri - Tür:*** Değişkenin alabileceği değerler aralığı -> **Ör:** Java'da "int" değişkeni -2(üzeri 31) ile +2(üzeri 31) arasında değişir.
  - ✓ ***Değişken Öznitelikleri - Değer:*** Değişkenin ilişkilendirildiği konumun içeriği. **Ör:** l_value <-- r_value. l_value, değişkenin adresi. r_value, değişkenin değeri.

❖ **Bağlanma Kavramı (Binding);**

✓ ***Bağlanma (binding),*** varlık <-> özniteliği ya da operasyon <-> sembol arasındaki ilişkidir.

✓ ***Muhtemel Bağlanma Süreleri:***

➢ ***Dil tasarım zamanı*** - operatör işlemlerini sembollere bağlama,

➢ ***Dil uygulama zamanı*** - kayan nokta türünü bir gösterime bağlama,

➢ ***Derleme zamanı*** - bir değişkeni C veya Java'da bir türe bağlama.

➢ ***Bağlantı süresi*** - kütüphane alt programına yapılan çağrı alt program koduna bağlanır,

➢ ***Yükleme zamanı*** - bir değişken belirli bir bellek konumuna bağlanır,

➢ ***Çalışma zamanı*** - bir değişken, atama cümlesi üzerinden bir değere bağlanır.

Binding Times

- **Example:**
  - `count = count + 5`
- The type of `count` is bound at compile time
- The set of possible values of `count` is bound at compiler design time
- The meaning of the operator symbol `+` is bound at compile time, when the types of its operands have been determined
- The internal representation of the literal `5` is bound at compiler design time
- The value of `count` is bound at execution times with this statement

❖ **Statik ve Dinamik Bağlama *(https://www.geeksforgeeks.org/static-vs-dynamic-binding-in-java);***

✓ Bağlama, çalışma zamanından önce gerçekleştiğinde statiktir ve program çalışması boyunca değişmeden kalır.

✓ Bağlama, ilk yürütme sırasında gerçekleşirse veya programın çalışması sırasında değiştirilebilirse dinamiktir.

✓ ***Statik Tip Bağlama;***

➢ Şayet değişkenin (variable) tipi açıkça tanımlanıyor ve programcı tarafından belirleniyorsa bu tip tanımlamalara açıktan tanımlama (explicit decleration) şayet açıkça belirtilmiyor ancak içerisine konulan verinin tipine göre belirleniyorsa bu tip tanımlamalara da gizli bağlama ile tanımlama (implicit decleration) ismi verilir.

➢ Çoğu mevcut Programlama Dili, tüm değişkenlerin açık beyanlarına ihtiyaç duyar, istisnalar -> **Ör:** Perl, Javascript, ML

➢ Örneğin FORTRAN dilinde "I, J, K, L, M veya N" harfleriyle başlayan değişkenler tam sayı (integer) ve diğer bütün tanımlamalar ise reel sayı olarak belirlenmiştir ve içsel olarak bu tanımlanma kendiliğinden yapılmış programcının bir tanımlama yapmasına gerek bırakılmamıştır.

➢ Benzer şekilde PERL dilinde bazı özel semboller ile değişken tipleri belirlenir. Örneğin $ sembolü ile başlayan bir değişken sabit bir sayı tutabilir (scalar) buna karşılık @ sembolü ile başlayan değişkenler dizilerdir (arrays) yine benzer şekilde % işareti ile başlayan değişkenler ise özet değerleri (hashing) tutmaktadır. **Ör:** $apple = scalar, @apple = array, %apple = hash.

✓ ***Dinamik Tip Bağlama;***

➢ Yukarıda açıklanan sabit bağlamalara (static binding) karşılık değişken bağlamalarda (dynamic binding) değişkenin (variable) tipi atandıktan sonra değişebilir.

➢ Yani yukarıda açıktan (explicit) veya kapalı (implicit) olarak tip belirlendikten sonra değişememektedir. **Ör:** int x; tanımından sonra x değişkeninin değeri tamsayı olmaktadır.

- Buna karşılık hareketli bağlamalarda (dynamic binding) tip bir kere atandıktan sonra değişebilir.
- Örneğin, **bilgi = {2,3,4};** şeklindeki bir tanımla bilgi ismindeki değişkene bir dizi konulmuştur. Bu durumda bilgi değişkeninini bir dizi olduğu sonucuna varılır ve tipi bu şekilde atanır. Ancak yukarıdaki satırdan sonra aşağıdaki şekilde bir satır gelirse:
- **bilgi = "ali";** bu durumda değişkenin tipi dizgi (string) olarak yeniden atanmış olur ve bu satırdan sorna bu değişken üzerinde yapılan işlemler dizgi (string) işlemleri olarak kabul edilir.

❖ **Tip Çıkarımı (Type Inference)** **[http://bilgisayarkavramlari.sadievrenseker.com/2009/05/24/degisken-tip-baglama-dynamic-type-binding-muteharrik-sekil-bagi/];**

✓ Miranda, Haskell ve ML gibi programlama dillerinde fonksiyonların tip çıkarımı yapması durumudur. Örneğin ML dilinde aşağıdaki örneği ele alalım:

functionAlan(r):3.14*r*r; yanda r yarıçapında bir dairenin alanını hesaplayan fonksiyon verilmiştir. Bu fonksiyonda dönen değerin tipi reel sayı olacaktır çünkü fonksiyon içerisinde 3.14 gibi reel bir sayı ile çarpım yapılmıştır. İşte bu noktada programlama dili, fonksiyonun içeriğinden bir çıkarım yapmaktadır.

✓ ML programlama dilinde çıkarım yapılamayan durumlarda programcının bir tipi elle belirtmesi istenir. Örneğin:

functionCarp(x):x*x; yandaki fonksiyonda x değerinin tipi bilinmediği için ve fonksiyonun dönüş tipi tahmin edilemeyeceği için programcının fonksiyonu yandaki şekilde yazması gerekir: **functionKare(x): int = x*x;**

❖ **Sabit Değişkenler;**

✓ Programın çalışmasından önce hafızaya (RAM) yüklenen ve programın çalışması süresince hep hafızada kalan değişken türleridir.

✓ Sabit değişkenlerin bir avantajı hız açısından verimdir (time performance) çünkü değişklerin adres hesaplamaları oldukça basit olmaktadır. Buna karşılık esneklikten feragat edilmektedir. Yani değişkenler hafızaya çakılmakta ve programın tamamı bitene kadar oynamamaktadırlar. Bu durumda değişkenin kullanımı bitmiş olsa bile hafızada kalır ve yerine başka bir değişkenin yüklenmesi engellenir.

❖ **Yığın-Hareketli Değişkenler (Stack-Dynamic Variables);**

✓ Bir programlama dilinin tasarımında kullanılan değişken tutma tipidir. Basitçe değişken hafızaya çalışma sırasında (run-time) yüklenir. Ancak değişken hafızada sabit (static) olarak kalır fakat içeriği zamanla değişebilir.

❖ **Açık-Yığıt Hareketli Değişkenler (Explicit-Heap Dynamic Variables);**

✓ Bu değişken tipi ise, programcı tarafından açıkça belirtilerek kullanılabilen ve hareketli olarak (dynamic) ayrılarak geri bırakılabilen alanlardır. Programlama dillerinde dinamik hafıza yönetimi (dynamic memory management) özelliği sayesinde hafızanın istenilen boyutta programcı tarafından ayrılması (allocate) mümkündür. Örneğin C ve C++ dillerinde malloc,realloc veya calloc fonksiyonları ile bu işlem programcı tarafından yapılabilir.

Example:
– In C++

```
int *intnode;                  // Create a pointer
intnode = new int;             // Create the heap-dynamic variable
….
delete intnode;                // Deallocate the heap-dynamic variable
```

❖ **Kapalı-Yığıt Hareketli Değişkenler (Implicit-Heap Dynamic Variables);**

- ✓ Bu değişken tipi genellikle programlama dili içerisinde dinamik olarak oluşturulan ancak özel bir şekilde programcının belirtmesine gerek duyulmayan yapıları ifade eder. Örneğin C veya C++ dillerinde bulunan union yapısı buna bir örnektir. Bir değişkenin tipi union şeklinde tanımlandığı zaman bu değişkenin değerinin heap (yığıt) içerisinde tutulması tasarlanır ancak burada özel bir tanım gerekmez.

❖ **Scope** ***(https://www.youtube.com/watch?v=EKf0Jslyr4Q);***

- ✓ ***Statik Scoping;*** En yakın değişkene bindingin değeri atanır. Basit bir şekilde programın metni okunup bu işlem yapılabilir. Programın çalışırken (runtime) oluşturduğu stack içeriğine bakılmasına gerek yoktur.

Sadece metine bakması yeterli olduğu için bu tarz scopinglere "lexical scoping" adı da verilir. Static scope, kodun anlaşılmasını daha kolay hale getirdiği için daha modüler kodlar yazılmasını sağlar.

- ✓ ***Dinamik Scoping;*** Programcının bütün olası stack değerlerini ve karşılaşabileceği olasılıkları hesaplamasını gerektirdiği için itici olabilir.

- ✓ Örneğin aşağıdaki kod hem static hem de dinamik (dynamic) scoping ile çalıştırılabilir:

```
int x = 0;

int f () { return x; }

int g () { int x = 1; return f(); }
```

- ✓ Şayet static scoping kullanılırsa g fonksiyonunun döndüreceği değer "0" olur çünkü, static scopingin o sırada fonksiyon stackinde ne olduğu ileilgisi yoktur ve x değerinin son hali olan 0'ı alır.

- ✓ Tersi olarak dynamic scoping kullanarak bu kod çalıştırılacak olsaydı g fonksiyonunun döndüreceği değer 1 olacaktır. Çünkü g fonksiyonu terk edilmeden önce x in değeri 1 idi ve bu bilgi stackten alınır.

2) Ders06 - Functional Programming Language;

❖ **Lambda,** ifadeleri isimsiz fonksiyonları tanımlar. Lambda(x) x*x*x ifadesi x'in küpü işlemini yapar. -> **Ör:** (Lambda(x) x*x*x) (2) = 8.

❖ **Fonksiyon Birleşimi,** bir fonksiyonu diğer fonksiyonlar cinsinden yazma. **Mesela,** h(x) = f(g(x)), f(x) = x+10 ve g(x) = 3*x olduğunu varsayalım. O halde h(x) = (3*x) +10.

❖ **Hepsini Uygulama,** bir fonksiyona birden fazla parametre verip değerleri bir liste şeklinde çıkarmak. **Örneğin,** f(x) = x*x ise (h, (2,3,4)) ifadesi (4,9,16) sonucunu doğurur.

❖ **Lisp Veri Tipleri ve Yapıları;**

- ✓ LISP, 1959'da MIT'de John McCarthy tarafından geliştirildi.

- ✓ Veri nesne türleri: Sadece atomlar ve listeler.

- ✓ Liste formu: alt listelerin ve / veya parantez içine alınmış . **Ör:** (A B (C D) E)

- ✓ Lambda notasyonu, işlevleri ve işlev tanımlarını belirtmek için kullanılır. Fonksiyon uygulamaları ve verileri aynı forma sahiptir. **Örneğin,** liste (A B C) veri olarak yorumlanırsa, A, B ve C ismindeki 3 atomun basit bir listesi. Bir fonksiyon uygulaması olarak yorumlanırsa, A olarak adlandırılan fonksiyonun B ve C adlı 2 parametresine uygulandığı anlamına gelir.
- ✓ İfadeler "EVAL" işleviyle yorumlanır.

❖ **İlkel Fonksiyonlar;**

- ✓ Parametreler, belirli bir sırayla değerlendirilir. Parametrelerin değerleri fonksiyon gövdesine değiştirilir. Fonksiyon gövdesi değerlendirilir. Vücuttaki son ifadenin değeri, fonksiyonun değeridir.
- ✓ İlkel Aritmetik Fonksiyonlar: +, -, *, /, ABS, SQRT, REMAINDER, MIN, MAX **Örneğin;** (* 3 5 7) = 3*5*7 = 105.

❖ **Define (Fonksiyon Oluşturma Terimi);**

- ✓ Bir sembolü bir ifadeye bağlamak. **Örneğin,** (Define pi 3.14) => pi = 3.14, (Define two_pi (* 2 pi)) => two_pi = 2*pi, (Define (kare x) (* x x)) => (kare 5) = 5*5 = 25.
- ✓ DEFINE için değerlendirme süreci farklıdır! İlk parametre asla değerlendirilmez. İkinci parametre değerlendirilir ve ilk parametreye bağlanır.

❖ **Çıkış Fonksiyonları;** (DISPLAY ifade) ya da (NEWLINE)

❖ **Sayısal Öngörme İşlevleri;**

- ✓ **#T (veya #t) true ve #F (veya #f) false ifade edilir. // =, <>,>, <,> =, <= // EVEN ?, ODD ?, ZERO ?, NEGATİF? // NOT işlevi bir Boole ifadesinin mantığını dönüştürür.**

❖ **Kontrol Akışı - COND;**

```
(DEFINE (compare x y)
   (COND
     ((> x y) "x is greater than y")
     ((< x y) "y is greater than x")
     (ELSE "x and y are equal")
   )
 )
```

❖ **Liste İşlevleri;**

- ✓ ***CONS***, iki parametre alır; bunlardan ilki bir atom veya bir liste olabilir ve ikincisi bir liste olabilir; ilk öğeyi ve ikinci listeyi içeren yeni bir liste döndürür. **Ör:** (CONS 'A '(B C)) returns (A B C)
- ✓ ***LIST,*** herhangi bir sayıda parametre alır; parametrelerle bir liste oluşturur ve onu döndürür. **Ör:** (LIST 'apple 'orange 'grape) returns (apple orange grape)

- ✓ ***CAR,*** listenin ilk elemanını döndürür. **Ör:** (CAR '(A B C)) yields A, (CAR '((A B) C D)) yields (A B).
- ✓ ***CDR,*** listenin ilk elemanı haricindeki elemanları döndürür. **Ör:** (CDR '(A B C)) yields (B C), (CDR '((A B) C D)) yields (C D)
- ✓ ***CAR ve CDR tipi özel tanımlamalar;*** (CAAR x) = (CAR(CAR x)), (CADR x) = (CAR (CDR x)), (CADDAR x) = (CAR (CDR (CDR (CAR x)))), (CADDAR '((A B (C) D) E)) = (C)
- ✓ ***Öntanımlı İşlev - EQ?,*** iki sembolik parametre alır; Her iki parametre de atomsa ve ikisi de aynı ise #T döndürür; aksi halde #F. **Ör:** (EQ? 'A 'A) yields #T, (EQ? 'A 'B) yields #F

EQ? liste parametreleriyle çağrılır, sonuç güvenilir değildir. Ayrıca EQ? sayısal atomlar için çalışmaz.

- ✓ ***Öntanımlı İşlev - EQV?,*** EQ? gibi, sadece sembolik ve sayısal atomlar için çalışır; bir değer karşılaştırmasıdır. **Ör:** (EQV? 3.4 (+ 3 0.4))yields #T.
- ✓ ***Öntanımlı İşlevler - LIST?,*** bir parametre alır; Parametre bir liste ise #T döndürür; aksi halde #F. **Ör:** (LIST? '()) yields #T.
- ✓ ***Öntanımlı İşlevler - NULL?,*** bir parametre alır; Parametre boş liste ise #T döndürür; aksi halde #F. **Ör:** (NULL? '(())) yields #F.

❖ **Bazı Scheme Fonksiyon Örnekleri;**

Example Scheme Function: `member`

- `member` takes an atom and a simple list; returns `#T` if the atom is in the list; `#F` otherwise

```
(DEFINE (member atm lis)
(COND
   ((NULL? lis) #F)
   ((EQ? atm (CAR lis)) #T)
   ((ELSE (member atm (CDR lis)))
))
```

Example Scheme Function: `equalsimp`

- `equalsimp` takes two simple lists as parameters; returns `#T` if the two simple lists are equal; `#F` otherwise

```
(DEFINE (equalsimp lis1 lis2)
(COND
   ((NULL? lis1) (NULL? lis2))
   ((NULL? lis2) #F)
   ((EQ? (CAR lis1) (CAR lis2))
        (equalsimp(CDR lis1)(CDR lis2)))
   (ELSE #F)
))
```

Example Scheme Function: `equal`

- `equal` takes two general lists as parameters; returns `#T` if the two lists are equal; `#F` otherwise

```
(DEFINE (equal lis1 lis2)
  (COND
   ((NOT (LIST? lis1))(EQ? lis1 lis2))
   ((NOT (LIST? lis2)) #F)
   ((NULL? lis1) (NULL? lis2))
   ((NULL? lis2) #F)
   ((equal (CAR lis1) (CAR lis2))
        (equal (CDR lis1) (CDR lis2)))
   (ELSE #F)
))
```

Example Scheme Function: append

- append takes two lists as parameters; returns the first parameter list with the elements of the second parameter list appended at the end

```
(DEFINE (append lis1 lis2)
  (COND
    ((NULL? lis1) lis2)
    (ELSE (CONS (CAR lis1)
                (append (CDR lis1) lis2)))
))
```

(append '(A B) '(C D R)) returns (A B C D R)
(append '((A B) C) '(D (E F))) returns ((A B) C D (E F))

LET Example

```
(DEFINE (quadratic_roots a b c)
  (LET (
    (root_part_over_2a
     (/ (SQRT (- (* b b) (* 4 a c)))(* 2 a)))
    (minus_b_over_2a (/ (- 0 b) (* 2 a)))
  (DISPLAY (+ minus_b_over_2a root_part_over_2a))
  (NEWLINE)
  (DISPLAY (- minus_b_over_2a root_part_over_2a))
))
```

Tail Recursion in Scheme (cont'd.)

- Example of rewriting a function to make it tail recursive, using helper a function

Original:

```
(DEFINE (factorial n)
   (IF (= n 0)
      1
      (* n (factorial (- n 1)))
))
```

Tail recursive:

```
(DEFINE (facthelper n factpartial)
   (IF (= n 0)
      factpartial
      facthelper((- n 1) (* n factpartial)))
))
(DEFINE (factorial n)
   (facthelper n 1))
```

Functional Form - Composition

- If h is the composition of f and g, h(x) = f(g(x))

```
(DEFINE (g x) (* 3 x))
(DEFINE (f x) (+ 2 x))
(DEFINE h x) (+ 2 (* 3 x))) (The composition)
```

- In Scheme, the functional composition function compose can be written:

```
(DEFINE (compose f g) (LAMBDA (x) (f (g x))))
((compose CAR CDR) '((a b) c d)) yields c
 (DEFINE (third a_list)
    ((compose CAR (compose CDR CDR)) a_list))
```

is equivalent to CADDR

Functional Form – Apply-to-All

- Apply to All - one form in Scheme is map
  - Applies the given function to all elements of the given list;

```
(DEFINE (map fun lis)
  (COND
    ((NULL? lis) ())
    (ELSE (CONS (fun (CAR lis))
                (map fun (CDR lis))))
))

(map (LAMBDA (num) (* num num num)) '(3 4 2 6))
```

yields (27 64 8 216)