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

1) *DERS 01 - GİRİŞ:*

- 702 Programlama dili mevcut.
- İlk program ENIAC.
- Programlama Dilleri Kavramlarının Çalışılma Nedenleri; (**http://cs.loc.edu/~chu/COSI350/Ch1/Reason.html**)
  - ✓ **Fikirlerimizi uygularken daha kolay ve daha iyi yapabilmek için.** (*Programalama dillerinin detaylarını bilerek yazılım zenginleştirilebilir.*)
  - ✓ **Seçeneklerimizin ne olduğunu bilirsek iyiyi seçebiliriz.** (*Bilgimizi artırarak eldeki probleme en uygun programalama dilini seçebiliriz.*)
  - ✓ **Dil öğrenmede yetkinlik. Dillerin özelliklerini bilmeyen, belli bir dille çalışmaya alışmış kişi, farklı bir dili öğrenmesi gerektiğinde zorlanır.** (*Örnek: Nesneye yonelik programlama kavramını bilen bir kişi, Java'yı bu konsepti bilmeyen bir kişiye gore daha kolay öğrenebilir.*)
  - ✓ **Belli bir dilin önemli özelliklerini anlayarak daha iyi kullanabilmek için.** (*Diller komplex yapılardan olusur. Fakat önemli özellikler etkin kullanılarak yazılım gelistirilebilir.*)
  - ✓ **Bir kod yazabilir ve derleyicinin her şeyi yapmasını sağlayabiliriz, ancak uygulama ayrıntılarını bilmek bir dili daha akıllı bir şekilde kullanmamıza ve daha verimli bir kod yazmamıza yardımcı olur.**
  - ✓ **Dilleri daha iyi değerlendirebilirsek, doğru seçimler yaparız, doğru teknolojilerin gelişmesine destek olmuş oluruz.**
  - ✓ **Gerçekleştirimin anlaşılmasıyla programlama dilini daha iyi anlama.** (*Örnek: Alt programlar sıklıkla çağrılırsa, program hızı düşer. Bunu bilirsek daha iyi program tasarımı yapabiliriz.*)
  - ✓ **Hata bulurken özelliklerini bilmemiz faydalıdır.**
  - ✓ **Tıpkı doğal diller gibi, bir dilin gramerini ne kadar iyi biliyorsak, ikinci bir dil öğrenmek o kadar kolay olacaktır.**
  - ✓ Programlama dilleri kavramlarını inceleyerek, programcılar dillerin önceden bilmedikleri kısımlarını kolayca öğrenebilirler.
- Dil Değerlendirme Kriterleri; [**Readibility**/**Writability**/**Reliability**/**Cost**] (**http://ece.uprm.edu/~ahchinaei/courses/2010jan/icom4036/slides/03icom4036Intro.pdf**)
  - ✓ **Okunabilirlik;**
    - ➢ Programlar kolay okunabilir ve anlaşılır olmalı.
    - ➢ **Bütünün Basitliği;** Yönetilebilir özellikler ve yapılar, Aynı işi yapan özelliklerin çok olması (• Örnek: c = c + 1; c+ = 1; c++; ++c;)
    - ➢ **Orthogonality**; İlkel yapıların küçük sayıdaki yollar ile bir araya getirilerek birleştirilebilmesi (• Örnek: 4 veri tipi: integer, float, double, char ve 2 tip operatör: dizi ve pointer kullanarak birçok kontrol ifadesi yazılabilir.), Birbirinden bağımsız yapıların varlığı ve tanımlanması. (Pascal ve C ortogonal değildir.)

- **Kontrol ifadeleri;** İyi bilinen kontrol ifadelerinin varlığı (örn., while ifadesi)
- **Veri Tipleri ve Yapıları;** Veri yapılarını tanımlamak için yeterli sayıda kolaylığın olması.
- **Söz Dizim Tasarımı;** Bileşik ifadeleri oluşturmak için özel kelime ve metotların olması (class, for, while), Biçim ve anlam: kendi-kendini tanıtan yapılar, anlamlı anahtar kelimeler (static)

Orthogonality

- **Example :** Adding two 32-bit integers residing in memory or registers, and replacing one of them with the sum
- IBM (Mainframe) Assembly language has two instructions:

```
A   Register1, MemoryCell1
AR  Register1, Register2
```

More restricted
Less writable
Not orthogonal

meaning

Register1 ← contents(Register1) + contents(MemoryCell1)
Register1 ← contents(Register1) + contents(Register2)

- VAX Assembly language has one instruction:

```
ADDL operand1, operand2
```

orthogonal

meaning

operand2 ← contents(operand1) + contents(operand2)
Here, either operand can be a register or a memory cell.

✓ **Yazılabilirlik;**

- Program oluşturmak için yazımının kolay olması.
- **Basitlik;** Az yapının olması, küçük sayıda ilkellerin olması, bunları birleştirecek kuralların az olması.
- **Soyutlama Desteği;** Detayları yok sayarak karmaşık yapı ve işlemleri tanımlama ve kullanma yeteneği
- **Anlamlılık;** İşlemleri tanımlamak için uygun yolların olması, (Örnek: for ifadesinin yerine daha kolay yazılabilen while'ı kullanmak)
- **Okunabilirlik ve Yazılabilirlik;** Bir algoritmayı doğal bir şekilde ifade etme yolları bulunmayan diller, ister istemez doğal olmayan yaklaşımları kullanacaktır, böylece de okunabilirlik azalacaktır.

✓ **Güvenilebilirlik;**

- Teknik şartnamelere uygunluğu, tanımlara uyması.
  - **Tip Kontrolü;** Tip hataları için test etme. (Örneğin; yandaki uygulama tip kontrolü olmayan C programında çalışır. >>>>>>>>>>)
  - **İstisna (Exception) İşleme;** Çalışma zamanı hatalarının kesilmesi ve düzeltici önlemlerin alınması.
  - **Örtüşme (Aliasing);** Aynı bellek bölgesini işaret eden iki yada daha fazla farklı referansın olabilmesi iyi değildir.

For example, the following program in original C compiles and runs!

```
foo  (float  a)  {
  printf  ("a:  %g  and  square(a):  %g\n",  a,a*a);
}
main  ()  {
  char  z  =  'b';
  foo(z);
}
```

Output is : a: 98 and square(a): 9604

- ✓ **Maliyet;**
  - ➢ Dili kullanmak için programcıların eğitimi.
  - ➢ Program yazma maliyeti (özel uygulamalara kapalılık).
  - ➢ Programları derleme maliyeti.
  - ➢ Programları yürütme maliyeti.
  - ➢ Uygulama Sisteminin Maliyeti: Eğer program pahalıysa veya sadece pahalı donanımlarda çalışıyorsa, yaygın olarak kullanılmayacaktır.
  - ➢ Zayıf güvenilirlik yüksek maliyetlere neden olur.
  - ➢ Programların bakımı sonucu oluşan maliyet.
- ✓ **Diğer Kriterler;**
  - ➢ **Taşınabilirlik;** Bir programın bir gerçekleştirimden başka bir gerçekleştirime kolaylıkla taşınabilir olması.
  - ➢ **Genellik;** Geniş sahadaki uygulamalara uygulanabilirlik.
  - ➢ **İyi Tanımlanabilirlik;** Dilin resmi tanımının tam ve kesin olması.

❖ Dil Tasarımının Getiri-Götürüsü: (**https://www.slideshare.net/abreslav/trade-offs-22989326**)

- ✓ **Güvenilirliğe Karşı Çalıştırma Maliyeti;** Örneğin - Java dizi içindeki elemanların tamamına ulaşımda referansların ve indislerin kontrol edilmesini talep eder, bu da çalıştırma maliyetini arttırır.
- ✓ **Okunabilirliğe Karşı Yazılabilirlik;** Örneğin - APL birçok güçlü operatör yardımıyla oldukça karmaşık hesaplamaların yapılabilmesine imkan verir, fakat okunabilirlik azalır.
- ✓ **Yazılabilirliğe (esneklik) Karşı Güvenilirlik;** Örneğin - C++ işaretçileri güçlüdür ve oldukça esnektir fakat kullanımı güvenilir değildir.

❖ Programlama Alanları: [**Scientific Applications**/**Business Applications**/**Artificial Intelligence**/**Systems Prog.**/**Scripting Languages**] (**http://cs.loc.edu/~chu/COSI350/Ch1/Domain.html**)

- ✓ **Bilimsel Uygulamalar;**
  - ➢ Büyük sayıda noktalı hesaplama yapma
  - ➢ Doğru hesaplama en önemli özellik
  - ➢ Fortran (1950'ler), Algol 60 (1960'lar)
- ✓ **İş Uygulamaları;**
  - ➢ Rapor oluşturma, ondalık sayı ve karakterlerin kullanımı
  - ➢ COBOL (1960'lar) halen en popüleri
- ✓ **Yapay Zeka;**
  - ➢ Sayılar yerine semboller kullanılır, dizi yerine bağlantılı bilgi
  - ➢ Programlar çok daha esnek yapıya sahip olmalıdır; program çalışırken yeni kod üretip çalıştırabilmelidir.
  - ➢ LISP (1965), Prolog (1970'ler)
- ✓ **Sistem Programlama;**
  - ➢ Sürekli kullanım nedeniyle hızlı ve verimli çalışma gereksinimi
  - ➢ IBM'in ilk sistem programi PL/I (1970 ler), C (1970ler)
  - ➢ Hemen hemen tum işletim sistemleri C veya C++ ileyazılmıştır.
  - ➢ UNIX tamamen C ile yazılmıştır.
- ✓ **Web Yazılımı;**

- ➢ Markup (örn. HTML, XHTML) bir programlama dili degildir.
- ➢ Scripting Languages (örn., PHP, Javascript) dinamik icerik icin HTML dökümanına program kodları eklemek için
- ➢ Genel Amaçlı (örn. Java (applets, servlets))

❖ Dil Kategorileri: [**Imperative**/**Functional**/**Logic**/**Object Oriented**/**Markup**] (**http://www.info.univ-angers.fr/~gh/hilapr/langlist/classif.htm**)

✓ **Emirsel;**

- ➢ Merkezi özellikleri değişkenler, atama ifadeleri ve döngülerdir. (Örnek: C, Pascal)

✓ **Fonksiyonel;**

- ➢ Hesaplama yapmanın temelinde veriler ve parametrele fonksiyonları uygulamak. (Örnek: LISP, Scheme)

✓ **Mantık;**

- ➢ Kural tabanlı, kurallar belirli sıralama olmadan verilir. (Örnek: Prolog)

✓ **Nesneye Yönelik;**

- ➢ Veri soyutlama, kalıtım, polymorphism. (Örnek: Java, C++)

✓ **İşaretleme;**

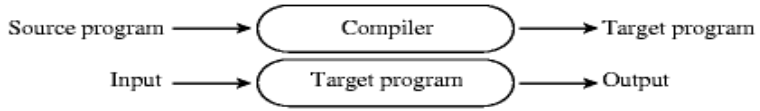
- ➢ Yeni; tam bir programlama dili değillerdir fakat web dökümanlarındaki bilginin yerleşimini belirtmede kullanılır. (Örnek: XHTML, XML)

2) DERS 02 - DESCRABING SYNTAX AND SEMANTICS:

2.1) Uygulama Yöntemleri: [Compilation/Pure Interpretation/Hybrid Implementation Systems]

❖ **Derleme;**

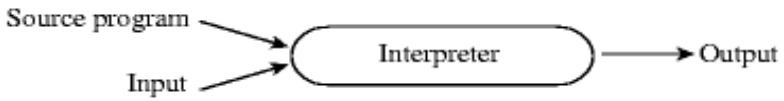
✓ Programlar makine diline çevrilir; JIT sistemleri içerir. (Kullanım: Büyük ticari uygulamalar) Yavaş Çeviri, Hızlı Çalıştırma !!! (C, C++, Cobol, Ada)



❖ **Hibrit Uygulama Sistemleri;**

❖ **Saf Yorumlama;**

✓ Programlar tercüman olarak bilinen başka bir program tarafından yorumlanır. (Kullanım: Küçük programlar veya verimlilik bir sorun olmadığında) Çeviri Yok, Yavaş Çalıştırma !!! (Script Dilleri)



- ✓ Derleyiciler ve saf tercümanlar arasında bir uzlaşma. (Kullanım: Verimlilik ilk sorun değilse küçük ve orta sistemler) Saf Yorumlama'dan Daha Hızlı !!!

2.2) Syntax (Söz Dizimi/Gramer) ve Semantics (Programlama Dilinin Davranışları):

❖ Bilgisayar Programları Oluşturma;

- ✓ Her programlama dili, **bir dizi ilkel işlem** sağlar.
- ✓ Her programlama dili, **karmaşık olan ilkel ifadeleri**, yasal bir biçimde birleştiren mekanizmalar sağlar.
- ✓ Her programlama dili, hesaplamalar veya ifadeler ile ilişkili **anlamların veya değerlerin çıkarılması** için mekanizmalar sağlar.

❖ Terminology;

- ✓ **Syntax,** ifadelerin ve program birimlerinin biçimi veya yapısı. (Örneğin; while)
- ✓ **Sematics,** ifadelerin ve program birimlerinin anlamı. (Örneğin; while'ın iç değerinin anlamı)
- ✓ **Sentence,** bazı alfabelerin bulunduğu söz dizisi.
- ✓ **Language,** cümle kümesi.
- ✓ **Lexeme,** bir dilin en düşük düzeyde sözdizimsel birimi (Örneğin; sum, begin)
- ✓ **Token,** Lexeme'nin kategorisi. (Örneğin; identifier)

Example in Java Language

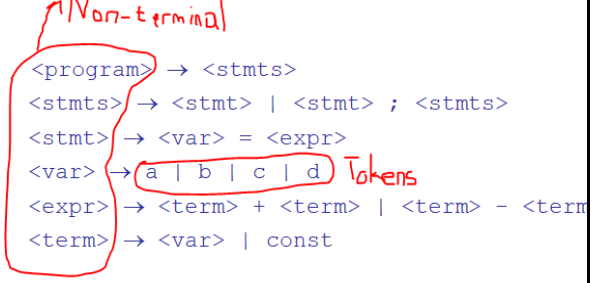
```
x = (y+3.1) * z_5 ;
```

```
x = (y+3.1) * z_5;
```

Lexemes	Tokens
x	identifier
=	equal_sign
(	left_paren
)	right_paren
for	for
y	identifier
+	plus_op
3.1	float_literal
*	mult_op
z_5	identifier
;	semi_colon

❖ BNF (Backus-Naur Form (1959))

- ✓ BNF'de soyutlamalar, sözdizimsel yapıların sınıflarını temsil etmek için kullanılır, sözdizimsel değişkenler gibi davranırlar (aynı zamanda nonterminal semboller veya sadece terminaller olarak da adlandırılır)
- ✓ Terminaller lexeme'ler ya da token'lardır.
- ✓ <LHS> → <RHS> >>>> Soldaki ifade için LHS; Non-terminal, RHS; Terminal dizisi ya da non-terminal ifade içerir.
- ✓ Sağda bir örneğini göreceğimiz "BNF" açılımında anlamlandırmak kolay olacaktır.




```
<program> => <stmts> => <stmt>
          => <var> = <expr>
          => a = <expr>
          => a = <term> + <term>
          => a = <var> + <term>
          => a = b + <term>
          => a = b + const
```

- ✓ ***Türetme (Derivation);***

- Verilen bir dizenin dilde geçerli bir programı temsil edip etmediğini kontrol etmek için, bunu dilbilgisinde türetmeye çalışırız.

- En soldaki türetme, her bir cümle içindeki en soldaki nonterminalin genişletilmiş olanıdır.

Derive string: **begin A := B; C := A * B end**

<program> ⇒ begin <stmt_list> end
⇒ begin <stmt> ; <stmt_list> end
⇒ begin <var> := <expression>; <stmt_list> end
⇒ begin A := <expression>; <stmt_list> end
⇒ begin A := B; <stmt_list> end
⇒ begin A := B; <stmt> end
⇒ begin A := B; <var> := <expression> end
⇒ begin A := B; C := <expression> end
⇒ begin A := B; C := <var><arith_op><var> end
⇒ begin A := B; C := A <arith_op> <var> end
⇒ begin A := B; C := A * <var> end
⇒ begin A := B; C := A * B end

sentential form

If always the leftmost nonterminal is replaced, then it is called **leftmost derivation**.

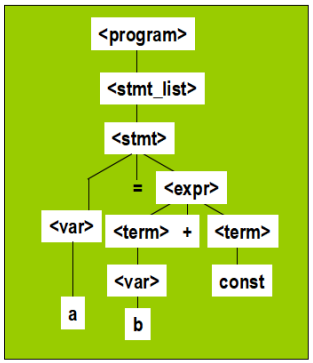
- Bir türetmedeki her sembol dizesi bir cümle biçimidir.

- Bir cümle, yalnızca terminal sembollerine sahip olan bir cümle biçimidir.

✓ ***Parse Tree;***

- A hierarchical representation of a derivation

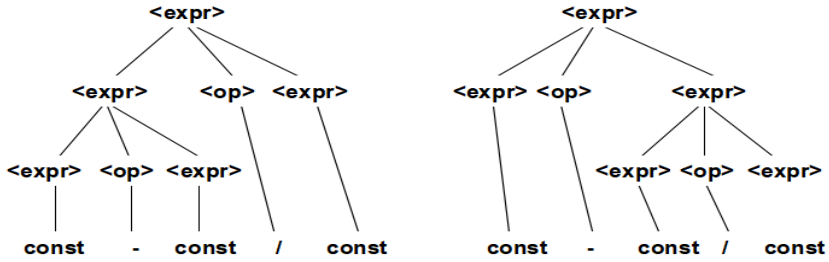
<program> → <stmt_list>
<stmt_list> → <stmt>
| <stmt> ; <stmt_list>
<stmt> → <var> = <expr>
<var> → a | b | c | d
<expr> → <term> + <term>
| <term> - <term>
<term> → <var> | const



✓ ***Belirsizlik (Ambiguous);***

- Bir dilbilgisi, yalnızca iki veya daha fazla ayrıştırma ağacı içeren bir cümle formu oluşturuyorsa belirsizdir.

<expr> → <expr> <op> <expr> | const
<op> → / | -



❖ ***Öncelik Kuralı (Precedence);***

✓ Bir ifadede birden fazla operatör varsa, C dili, işleçler için önceden tanımlanmış bir öncelik kuralına sahiptir. Bu işleçlerin önceliği kuralına "operatör önceliği" denir.

✓ C'de, aritmetik işleçlerin (*,%, /, +, -) önceliği ilişkisel işleçlerden (==,! =,>, <,> =, <=) daha yüksektir. İlişkisel işleçlerin önceliği de mantıksal işleçlerden daha yüksektir. (&&, ||).

```
(1 > 2 + 3 && 4)
This expression is equivalent to:
((1 > (2 + 3)) && 4)
i.e, (2 + 3) executes first resulting into 5
then, first part of the expression (1 > 5) executes resulting into 0 (false)
then, (0 && 4) executes resulting into 0 (false)
```

❖ ***Birleşme Kuralı (Associativity;***

- ✓ Bir ifadede aynı öncelikli (priority) iki operatör varsa, işleçlerin ilişkilendirilmesi yürütme sırasını gösterir.
- ✓ Burada, operatörler == ve! = Aynı önceliğe sahiptirler. Her ikisi de == ve! = Birlikteliği sağa sola, yani soldaki ifade ilk önce yürütülür ve sağa doğru hareket eder.

```
1 == 2 != 3
```

```
((1 == 2) != 3)
i.e, (1 == 2) executes first resulting into 0 (false)
then, (0 != 3) executes resulting into 1 (true)
```

❖ ***Extended BNF;***

- ✓ Bilgisayar bilimlerinde dil tasarımı konusunda kullanılan Backus Normal Şeklinin (BNF) özel bir halidir. Basitçe standart BNF'te yazılan kuralların birleştirilerek daha sade yazılmasını hedefler.
- ✓ Örneğin BNF olarak yazılan dilimize göre:
  - ➢ ***<IF> ::= if(<KOSUL>) | if(<KOSUL>) else,***
  - ➢ şeklinde bir satırımız bulunsun. Bu satırın anlamı dilimizde bir IF söz dizilimi (syntax), if komutu ve parantez içinde bir koşuldan oluşabilir veya bu if ve parantez içerisindeki koşulu bir else komutu izleyebilir.
  - ➢ Yukarıdaki bu BNF yazılımını EBNF olarak aşağıdaki şekilde yazabiliriz:
  - ➢ ***<IF> ::= if(<KOSUL>) [else]***
  - ➢ Yukarıdaki bu yeni satırda dikkat edileceği üzere köşeli parantezler arasında bir else komutu bulunmaktadır. Bunun anlamı, IF komutu "if(KOSUL)" olarak tanımlanır ve şayet istenirse bu komuta ilave olarak else komutu eklenebilir. Yani köşeli parantez içerisindeki komut isteğe bağlıdır.
- ✓ Yukarıdaki bu yeni yazılım aslında sadece gösterimde bir farklılık oluşturmaktadır. Bunun dışında, EBNF'in kullanım alanı ve işlevi BNF ile aynıdır.
- ✓ EBNF'in, BNF'den farkı yandaki tabloda işaretlenmiştir.

İfade		Kullanımı
Tanımlama	definition	=
Üleştirme	concatenation	,
Bitirme	termination	;
Seçim (Veya)	separation	|
Çift Tırnak	double quotation marks	" ... "
Tek Tırnak	single quotation marks	' ... '
İsteğe bağlı	option	[...]
Tekrarlı	repetition	{ ... }
Gruplama	grouping	(...)
Yorum	comment	(* ... *)
Özel dizilim	special sequence	? ... ?
Hariç	exception	-

- ✓ **<YORUM> ::= "/*" , { <harf> } , "*/"**
- ✓ **<harf> ::= a | b | … | z**
- ✓ Yukarıdaki EBNF tanımında a'dan z'ye kadar olan harfler, <harf> olarak tanımlanmış, ardından bu tanım <YORUM> içerisinde istenildiği kadar tekrarlanabilir anlamında {} işaretleri arasına yerleştirilmiştir.

3) DERS 03 - LEX: (http://bilgisayarkavramlari.sadievrenseker.com/2008/12/12/lex/)

- Bilgisayar bilimlerinde programlama dillerinin tasarımı ve geliştirilmesi sırasında kullanılan ve dildeki kelimelerin analizine ***(lexical analysis)*** yarayan kod üretme programıdır. Yani lex için hazırlanmış bir dosyayı lex programından geçirdikten sonra size C dilinde bir kod çıkar. Bu kodu C dilinde derledikten (compile) sonra çalışan bir programınız olur. Veya tercihen bu çıktıyı ***"yacc"*** programına alt yapı oluşturmak için de kullanabilirsiniz.
- LEX programının ismi inigilizcedeki lexical analyzer kelimesinden gelir. Kelime bilimi anlamına gelen Lexical kelimesinin ilk 3 harfinden kısaltılmıştır. LEX programının linux üzerinde çalışan ve yaygın bir sürümü ***flex*** ismindedir. flex programı da lex gibi çalışmaktadır ve hemen hemen aynı parametre ve özelliklerle kullanılabilir.
- LEX dosyalarının 3 ana bölümü bulunur. İlk bölümde fonksiyon ve değişken tanımlamaları ve projeye dahil edilecek (include) kütüphaneler tanımlanır.
- ikinci bölümde LEX dosyamızın omurgasını oluşturan düzenli deyimler (regular expression) kısmı yer alır. Burada her ihtimal için ayrı bir regular expression tanımlanarak ilgili regular expression'a girilmesi durumunda ne yapılacağı kodlanır.
- Son bölümde ise fonksiyon içerikleri yer alır.

```
%{ // yukarıda anlatılan ilk bölüm tanımlamalar yapılıyor
    #include "y.tab.h"
    #include <stdlib.h>
    void yyerror(char *);
%}
```

```
// ikinci bölüm regular expressionlar
%%
[0-9]+        {
                  yylval = atoi(yytext);
                  return INTEGER;
              }
[-+n]         { return *yytext; }
[ t]          ;        /* skip whitespace */
.             yyerror("Unknown character");
%%
```

```
%%
// son bölüm fonksiyon içerikleri
int yywrap(void) {
    return 1;
}
```

```
Ex1.l :
%%
zippy printf("I RECOGNIZED ZIPPY");

$cat test1
zippy
ali zip
veli and zippy here
zipzippy
ZIP

$cat test1 | ex1
I RECOGNIZED ZIPPY
ali zip
veli and I RECOGNIZED ZIPPY here
zipI RECOGNIZED ZIPPY
ZIP
```

```
%%
zip printf("ZIP");
zippy printf("ZIPPY");

$cat test1 | ex2
ZIPPY
ali ZIP
veli and ZIPPY here
ZIPZIPPY
```

```
%%
monday|tuesday|wednesday|thursday|friday|
saturday|sunday printf("<%s is a day.>",
  yytext);
$cat test3
today is wednesday september 27

$ex3 < test3
today is <wednesday is a day> september 27
```

!!!!!!!!!! Önemli Not: Lex spesifikasyon dosyasının sonunda ekstra boşluk ve / veya boş satır bırakmayın !!!!!!!!!!

❖ **Design Patterns:**

[abc] matches **a, b or c**
[a-f] matches **a, b, c, d, e, or f**
[0-9] matches any digit
X+ matches one or more of **X**
X* matches zero or more of **X**
[0-9]+ matches any integer
(...) grouping an expression into a single unit
(a|b|c)* is equivalent to **[a-c]***
X? X is optional (0 or 1 occurrence)
if(def)? matches **if** or **ifdef** (equivalent to **if|ifdef**)

[A-Za-z] matches any alphabetical character
. matches any character except newline character
\. matches the **.** character
\n matches the newline character
\t matches the tab character
**** matches the **** character
[\t] matches either a space or tab character
[^a-d] matches any character other than **a,b,c** and **d**

Real numbers
[0-9]*(\.)?[0-9]+
To include an optional preceding sign:
[+-]?[0-9]*(\.)?[0-9]+
Integer or floating point number
[0-9]+(\.[0-9]+)?
Integer, floating point or scientific notation.
[+-]?[0-9]+(\.[0-9]+)?([eE][+-]?[0-9]+)?

-**Lex** builds the **yylex()** function that is called, and will do all of the work for you.
-Lex also provides a count **yyleng** of the number of characters matched.
-**yywrap** is called whenever lex reaches an end-of-file
-**Lex** is a tool for writing lexical analyzers.
-**Yacc** is a tool for constructing parsers.

```
/* rule-order.l */
%%
for printf("FOR");
[a-z]+ printf("IDENTIFIER");

for input
for count := 1 to 10
the output would be
FOR IDENTIFIER := 1 IDENTIFIER 10
```

<<<<<<<<<

!!!!!!!!!! Eğer bir ifade belirlenmiş 2 kurala da uyuyorsa ilk önce hangi kurala girdiyse o uygulanır !!!!!!!!!!

<<<<<<<<<

3) *DERS 04 - YACC:* (http://bilgisayarkavramlari.sadievrenseker.com/2008/12/12/yacc/) || (https://www.youtube.com/watch?v=__-wUHG2rfM)

- YACC, bilgisayar bilimlerinin önemli dallarından birisi olan dil tasarımı ve dil geliştirilmesi sırasında (compiler teory) sıkça kullanılan bir kod üretici programdır. YACC basitçe dildeki sözdizim (syntax) tasarımı için kullanılır ve tasarladığımız dildeki kelimelerin sıralamasının istediğimiz şekilde girilip girilmediğini kontrol eder. Aynı zamanda sıralamadaki her kelimenin anlamını da yacc marifetiyle belirleyebiliriz.
- YACC temel olarak BNF (Backus Normal Form) kullanarak cümle dizimini belirtmektedir.
- LEX ile birlikte kullanıldığıdan bir dil tasarımının neredeyse yarısı olan lexical (kelime) ve syntax (cümle) analizi tamamlanmış olur. Bundan sonra dildeki her kelime ve cümle diziliminin anlamını (semantic) kodlamak kalır.
- YACC'i her ne kadar anlatmaya çalışsam da örnekler üzerinden ya da video izlenerek öğrenilmesi daha kolay duruyor. O yüzden bir konunun daha sonuna gelmiş bulunmaktayız.
- Sıkı Çalışın 100 Alın :D

YACC TOKEN'LARI VE ÖZELLİKLERİNİ TANIMLAMA			
%token	Token adlarının belirtir. (Örneğin; %token INTEGER)	%union	Semantic değerler için birden fazla veri türü belirleme.
%left	Sol ilişkilendirmeli operatörleri belirtir.	%start	Start sembolünü bildirir. (Varsayılan kurallar içindeki ilk değişken)
%right	Sağ ilişkilendirmeli operatörleri belirtir.	%prec	Bir kuralın önceliğini atar.
%nonassoc	Kendileriyle ilişkilendirilmeyen operatörleri tanımlar.	$$	Bu devamlıdan (nonterminal) dönecek olan değerdir. (Yani soldaki ifade)
%type	Değişkenlerin türünü bildirir.	$1 / $3	BNF yapısındaki ilk parametredir. / BNF yapısındaki 3. parametredir. Örneğin; | expr '+' expr { $$ = $1 + $3; } => $1 = "1.expr", $2 = "+", $3 = "2.expr"

Cheat Sheet => https://ufile.io/xaa2r

Ek Bilgi => http://comp.eng.ankara.edu.tr/lisans-egitimi/ders-sayfalari/ikinci-sinif-guz-donemi/com241-programlama-dilleri-kavramlari/

Örnekler => https://www.ibm.com/support/knowledgecenter/en/ssw_aix_72/com.ibm.aix.genprogc/ie_prog_4lex_yacc.htm

https://www.epaperpress.com/lexandyacc/prl.html

: 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)

1) *Ders07 - Pointer and Reference Types, Type Checking: (Sayfa 22 ve sonrası özet çıkarmanın gereksiz olmasından dolayı çoğunlukla eklenmedi !!!)*

- ***Tip Kontrolü,*** bir operatörün işlenenlerinin uyumlu tipte olmasını sağlama faaliyetidir.
- ***Uyumlu bir tür,*** operatör için yasal olan ya da dil kurallarının, derleyici tarafından oluşturulan kodla, örtük olarak dönüştürülmesine izin verilen bir yasal türe uygundur. Bu otomatik dönüşüme zorlama denir. Örneğin, Java'da bir float değişkeni ile int değişkeni toplandığında, int değişkeni float haline getirilir ve sonuç bu şekilde hesaplanır.
- ***Tip hatası,*** bir operatörün uygun olmayan bir işlenene uygulanmasıdır.
- ***Yazım tipi statik ise***, tüm tip denetimi derleyici tarafından statik olarak yapılabilir. Dinamik tip bağlaması, çalışma zamanında dinamik tip kontrolü gerektirir, örn. Javascript ve PHP
- Daha önceki düzeltme genellikle daha az maliyetli olduğu için, çalışma zamanı yerine derleme zamanında hataları saptamak daha iyidir.
- **Tür Uyumluluğu;**
  - ✓ İki değişkenin uyumlu tipte olmasının en önemli sonucu, her ikisinin de diğerine atanan değeri olabilir. Tip uyumluluğunu kontrol etmek için iki yöntem:

- Name Type Compatibility:
  - Two variables have compatible types only if they are in either the same declaration or in declarations that use the same type name.
- Advantage: Easy to implement
- Disadvantage: Highly restrictive
  - Subranges of integer types are not equivalent with integer types
  - Formal parameters must be the same type as their corresponding actual parameters

- Under a strict interpretation, a variable whose type is a subrange of the integers would not be compatible with an integer type variable
- Example:

```
type indexType = 1..10; {subrange type}
var count: integer;
index: indexType;
```

- The variables `count` and `index` are not name type compatible, and cannot be assigned to each other

Structure Type Compatibility

- Two variables have compatible types if their types have identical structure.
- Disadvantage: Difficult to implement
- Advantage: more flexible
- The variables `count` and `index` in the previous example, are structure type compatible.
- Structure type compatibility also disallows differentiating between types with the same structure

```
type celsius = float;
fahrenheit = float;
```

- They are compatible according to structure type compatibility but they may be mixed

  - ✓ Çoğu Programlama Dili bu yöntemlerin bir kombinasyonunu kullanır.
  - ✓ C, yapısal eşdeğerliği (structural equivalence), C++ ise isim eşdeğerliğini (name equivalence) kullanır.
  - ✓ Ada, isim uyumluluğunu kullanır, aynı zamanda iki tip yapı sağlar; -Subtypes, -Derived Types.

- **Derived Types:** Uyumsuz olduğu önceden tanımlanmış bazı türlere dayalı yeni bir tür. Ana türün tüm özelliklerini miras alırlar.

```
type celsius is new float
type fahrenheit is new float
```

- Yapıları aynı olmasına rağmen, iki tip birbiriyle uyumsuzdur. Ayrıca diğer kayar nokta (float) türleriyle de uyumsuzdurlar.
- **Subtypes:** Muhtemelen mevcut bir türün kısıtlı sürümünü gösterir. Bir alt tip ebeveyn türüyle uyumludur.
- `subtype small_type is Integer range 0..99;`
- Small_type değişkenleri tam sayı değişkenleriyle uyumludur.

❖ **C'de Tür Uyumluluğu;**

- ✓ C, ***structures*** ve ***unions*** dışındaki her tür için yapı türü uyumluluğunu kullanır.
- ✓ Her structures ve unions, başka hiçbir türle uyumlu olmayan yeni bir tür yaratır.
- ✓ Typedef'in herhangi bir yeni tip getirmediğini, ancak yeni bir isim tanımladığını unutmayın.
- ✓ C ++ isim eşdeğerliğini kullanır.

❖ **İç İçe Seçiciler (Nesting Selectors);**

- Java example

```
if (sum == 0)
   if (count == 0)
       result = 0;
else result = 1;
```

- Which **if** gets the **else**?
- Java's static semantics rule: **else** matches with the nearest **if**

- To force an alternative semantics, compound statements may be used:

```
if (sum == 0) {
  if (count == 0)
      result = 0;
}
else result = 1;
```

- The above solution is used in C, C++, and C#

- Python

```
if sum == 0 :
   if count == 0 :
     result = 0
   else :
     result = 1
```

2) ***Ders08 - Subprograms:***

❖ **Alt Programların Temelleri (Fundamentals of Subprograms);**

- ✓ Her alt programın tek bir giriş noktası vardır.
- ✓ Arama programı askıya alınmış durumda, çağrılan alt programın yürütülmesi halinde.
- ✓ Bu nedenle, belirli bir zaman diliminde yalnızca bir alt program yürütülmekte.

❖ **Temel Tanımlar;**

- ✓ Bir ***alt program tanımı,*** alt programı soyutlamanın eylemlerini ve arayüzünü açıklar.
- ✓ Python'da fonksiyon tanımları çalıştırılabilir; diğer tüm dillerde çalıştırılamaz.

```
if ….
  def fun1(…);
else
  def fun2(…);
```

- ✓ ***Alt program başlığı (subprogram header),*** adın, alt programın türünün ve biçimsel parametrelerin de dahil olduğu tanımın ilk bölümüdür.
- ✓ FORTRAN örneği >>> SUBROUTINE adı (parametreler), C örneği >>> void adder (parametreler)
- ✓ Bir ***alt program çağrısı (subprogram call),*** alt programın yürütülmesi için açık bir istektir.
- ✓ Bir ***alt programın parametre profili (aka signature),*** parametrelerinin sayısı, sırası ve türleridir.
- ✓ ***Protokol*** bir alt programın parametre profilidir ve eğer bir fonksiyonsa, dönüş tipidir.
- ✓ C ve C ++ 'daki işlev bildirimlerine genellikle ***prototipler (prototypes)*** denir.
- ✓ Bir ***alt program bildirimi (subprogram declaration),*** alt programın gövdesini değil, protokolünü sağlar.
- ✓ ***Resmi bir parametre (formal parameter),*** alt program başlığında listelenen ve alt programda kullanılan kukla (dummy) bir değişkendir.
- ✓ ***Gerçek bir parametre (actual parameter),*** alt program çağrısı deyiminde kullanılan bir değeri veya adresi temsil eder.

❖ **Parametreler;**

Example in Ada,

```
SUMER (LENGTH => 10,
       LIST => ARR,
       SUM => ARR_SUM);
```

Formal parameters: LENGTH, LIST, SUM.

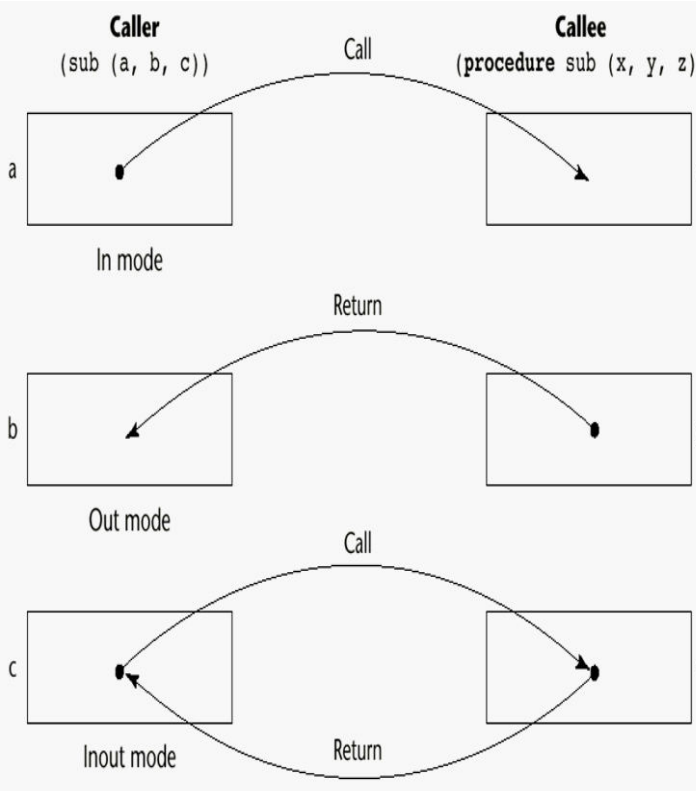
Actual parameters: 10, ARR, ARR_SUM.

- Ada, and FORTRAN 90 allow positional parameters and keyword parameters to be used together.

```
SUMER (10, SUM => ARR_SUM, LIST => ARR);
```

- Once a keyword appears in the call, all remaining parameters must be keyword parameters.

❖ **Parameter Passing; (https://courses.cs.washington.edu/courses/cse505/99au/imperative/parameters.html)**



- ***Pass-by-Value (In Mode);*** Bu yöntemde in-mode kullanır. Resmi parametrede yapılan değişiklikler arayana geri iletilmez. Çağrılan yöntem içindeki biçimsel parametre değişkeninde yapılan değişiklikler yalnızca ayrı depolama konumunu etkiler ve çağrı ortamındaki gerçek parametreye yansıtılmaz. Bu yöntem aynı zamanda değere göre çağrı olarak da adlandırılır.
- ***Pass-by-Reference (In/Out Mode);*** Bu teknik, iç/dış mod anlambilimini kullanır. Resmi parametrede yapılan değişiklikler, arayan (caller) parametreye geri iletilir. Formal parametredeki herhangi bir değişiklik, formal ortamdaki gerçek verilere bir referans aldığı için, çağrı (caller) ortamındaki gerçek parametreye yansıtılır. Bu yöntem hem zaman hem de mekanda etkilidir.
- ***Pass-by-Result (Out Mode);*** Bu yöntem out-mode anlamını kullanır. Kontrol arayan kişiye geri aktarılmadan hemen önce, formal parametrenin değeri gerçek parametreye geri iletilir. Bu yönteme bazen sonuçtan çağrı çağrılır. Genel olarak, sonuç tekniğine göre geçiş kopya ile uygulanır.
- ***Pass-by-Value Result (In/Out Mode);*** Bu yöntem, iç / dış mod anlamını kullanır. Bu, Pass-by-Value ve bir Pass-by-Result yöntemlerinin bir kombinasyonudur. Kontrol arayan kişiye geri aktarılmadan hemen önce, formal parametrenin değeri gerçek parametreye geri iletilir. Bu yönteme bazen değer-sonuç (value-result) çağrısı denir.

- ***Pass-by-Name (In/Out Mode);*** Bu teknik Algol gibi bir programlama dilinde kullanılır. Bu teknikte, bir değişkenin sembolik "adı" geçirilir ve bu hem erişilmesine hem de güncellenmesine izin verir. Bağımsız değişken ifadesi, formal parametre her iletildiğinde yeniden değerlendirilir. Prosedür, argüman ifadesinde kullanılan değişkenlerin değerlerini değiştirebilir ve böylece ifadenin değerini değiştirebilir. Yandaki örnekte de görülebileceği üzere "call by name" çağrısında fonksiyonun parametresine "n+10"un değeri değil, direkt olarak ismi geçiriliyor. Yani "k" ifade bir nevi "n" ile değiştirilmiş oluyor. Bundan sonraki işlemlerde "k"yi "n" olarak düşünebiliriz.

```
begin
integer n;
procedure p(k: integer);
   begin
   print(k);
   n := n+1;
   print(k);
   end;
n := 0;
p(n+10);
end;

Output:

call by value:   10 10
call by name:    10 11
```

❖ **Referencing Environment;**

- ✓ **Soru:** Geçirilen alt programı yürütmek için Başvuru Ortamı (Referencing Environment) nedir? (Lokal olmayan değişkenler için)
  - ***Sığ ciltleme (Shallow Binding):*** Geçirilen alt programı etkileyen call ifadesinin ortamı. Dinamik-kapsamlı diller için en doğal durum.
  - ***Derin ciltleme (Deep Binding):*** Geçirilen alt programın tanımı ortamı. Statik kapsamlı diller için en doğal.
  - ***Geçici ciltleme (Ad Hoc Binding):*** Alt programı geçen call deyiminin ortamı.

```
Example:
function sub1() {
  var x;           2:declared in        Passed subprogram S2
  function sub2() {                     Output:
    window.status = x;                  is called by S4
  } // sub2                             is declared in S1
  function sub3(){                      is passed in S3
    var x;
    x = 3;
    sub4(sub2);     3: passed in
  } // sub3
  function sub4(subx) {
    var x;
    x = 1;
    subx();         1:called by
  } // sub4
  x = 2;
  sub3();
} // sub1
```

❖ **ÖZET;**

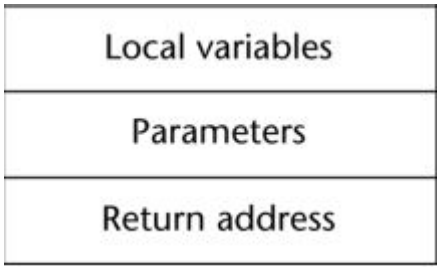
- ✓ Bir alt program tanımı, alt program tarafından temsil edilen eylemleri açıklar.
- ✓ Alt programlar, fonksiyonlar veya prosedürler olabilir.
- ✓ Alt programlardaki yerel değişkenler yığın dinamik veya statik olabilir.
- ✓ Üç parametre geçirme modeli: in mode, out mode & in-out mode.

- ✓ Bazı diller operatörün aşırı yüklenmesine izin veriyor
- ✓ Alt programlar kapsamlı (generic olabilir.
- ✓ Bir coroutine, çoklu girişleri olan özel bir alt programdır.

3) Ders09 - Implementing Subprograms:

❖ Implementing "Simple" Subprograms;

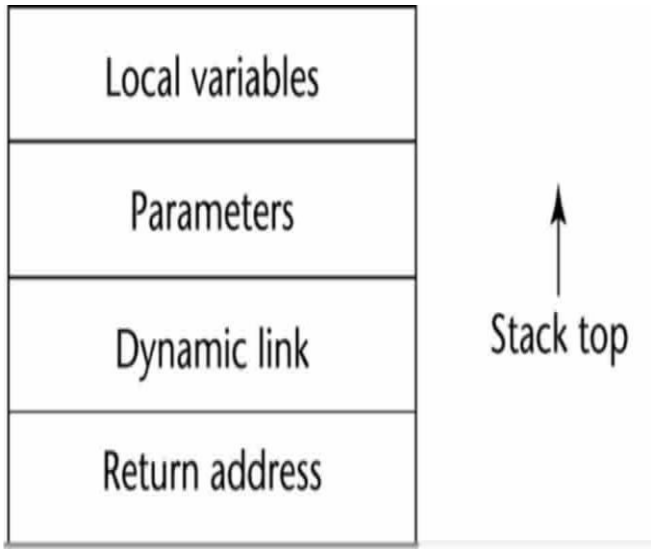
- ✓ Basit alt programlara sahip diller özyinelemeyi desteklemediğinden, belirli bir alt programın etkin bir versiyonunu bir seferde yalnızca bir tane olabilir.
- ✓ Bu nedenle, bir alt program için aktivasyon kaydının sadece tek bir örneği olabilir.
- ✓ Basit bir alt programın aktivasyon kaydı örneği sabit bir boyuta sahip olduğundan, statik olarak tahsis edilebilir. Ayrıca kod bölümüne de eklenebilir.



- ✓ Kodun ARI'lere eklenebileceğini unutmayın!! Ayrıca, dört program birimi farklı zamanlarda derlenebilir. Linker, ana program için çağrıldığında derlenen parçaları bir araya getirir.

❖ Stack-Dynamic Local Variables;

- ✓ ***Return address:*** Nereden çağrıldığını işaret eden adres. ***Dinamic link:*** Arayanın aktivasyon kayıt örneğinin tepesine gösterici.
- ✓ Statik kapsamlı dillerde, bu bağlantı, bir çalışma zamanı hatası oluştuğunda geri izleme bilgisi sağlamak için kullanılır.
- ✓ Dinamik kapsamlı dillerde, dinamik olmayan bağlantı yerel olmayan değişkenlere erişmek için kullanılır.



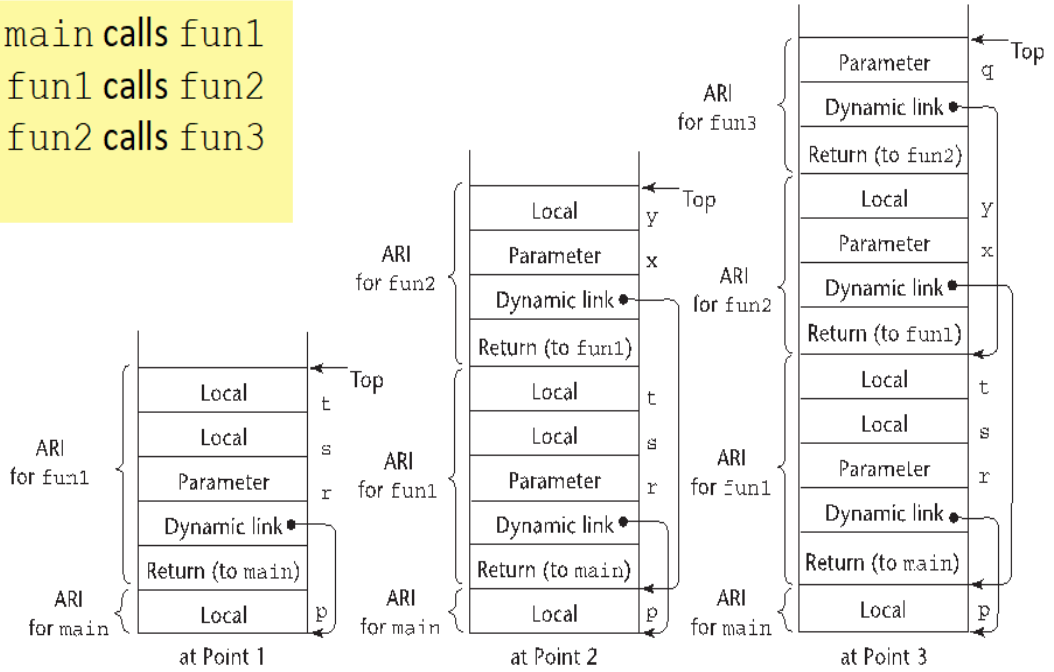
An Example Without Recursion

```
void fun1(float r) {
   int s, t;
   ...
   fun2(s);
   ...
}
void fun2(int x) {
   int y;
   ...
   fun3(y);
   ...
}
void fun3(int q) {
   ...
}
void main() {
   float p;
   ...
   fun1(p);
   ...
}
```

main calls fun1
fun1 calls fun2
fun2 calls fun3

An Example Without Recursion

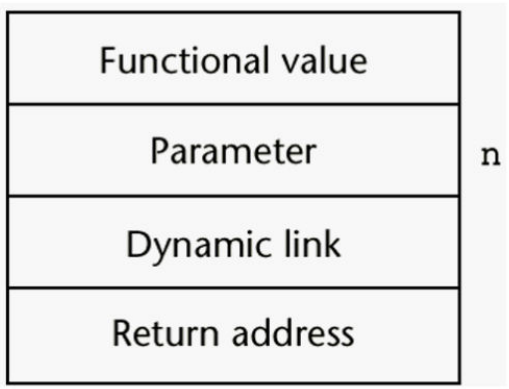
main calls fun1
fun1 calls fun2
fun2 calls fun3



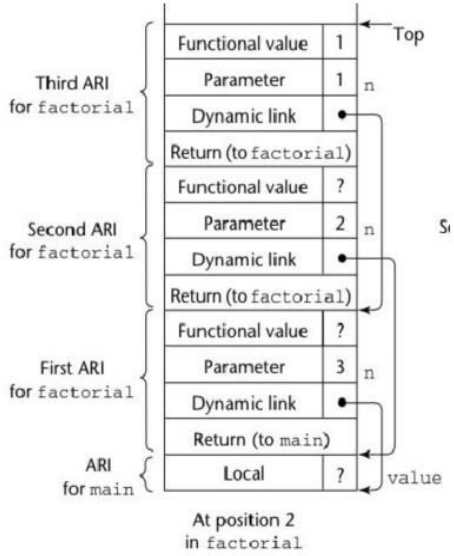
Recursion

```
int factorial (int n) {
     <-----------------------------1
   if (n <= 1)
       return 1;
   else return (n * factorial(n - 1));
     <-----------------------------2
}
void main() {
   int value;
   value = factorial(3);
     <-----------------------------3
}
```

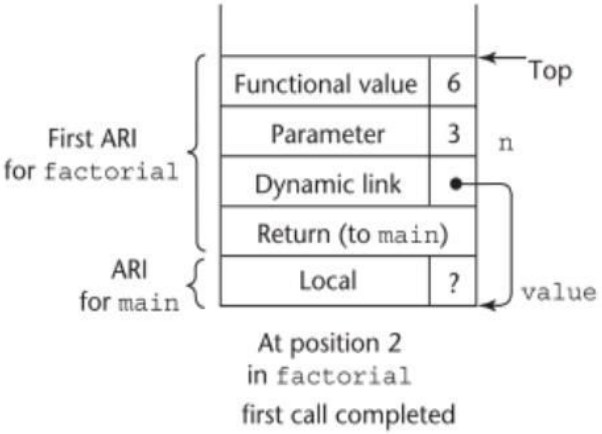
Activation Record for `factorial`



Stack contents during execution of `main` and `factorial`



Stack contents during execution of `main` and `factorial`



❖ **Nested Subprograms (İç İçe Alt Programlar);**

- ✓ ***Statik zincir (static chain),*** belirli aktivasyon kaydı örneklerini bağlayan statik bağlantıların bir zinciridir.
- ✓ Alt program A için bir aktivasyon kayıt örneğindeki ***statik bağlantı (static link),*** A statik ebeveyni aktivasyon kaydı örneklerinden birinin altına işaret eder.
- ✓ Bir aktivasyon kaydı örneğindeki statik zincir, onu statik atalarının tümüne bağlar.
- ✓ ***Static_depth,*** değeri iç içe geçme derinliği => olan statik bir kapsamla ilişkilendirilen bir tamsayıdır.

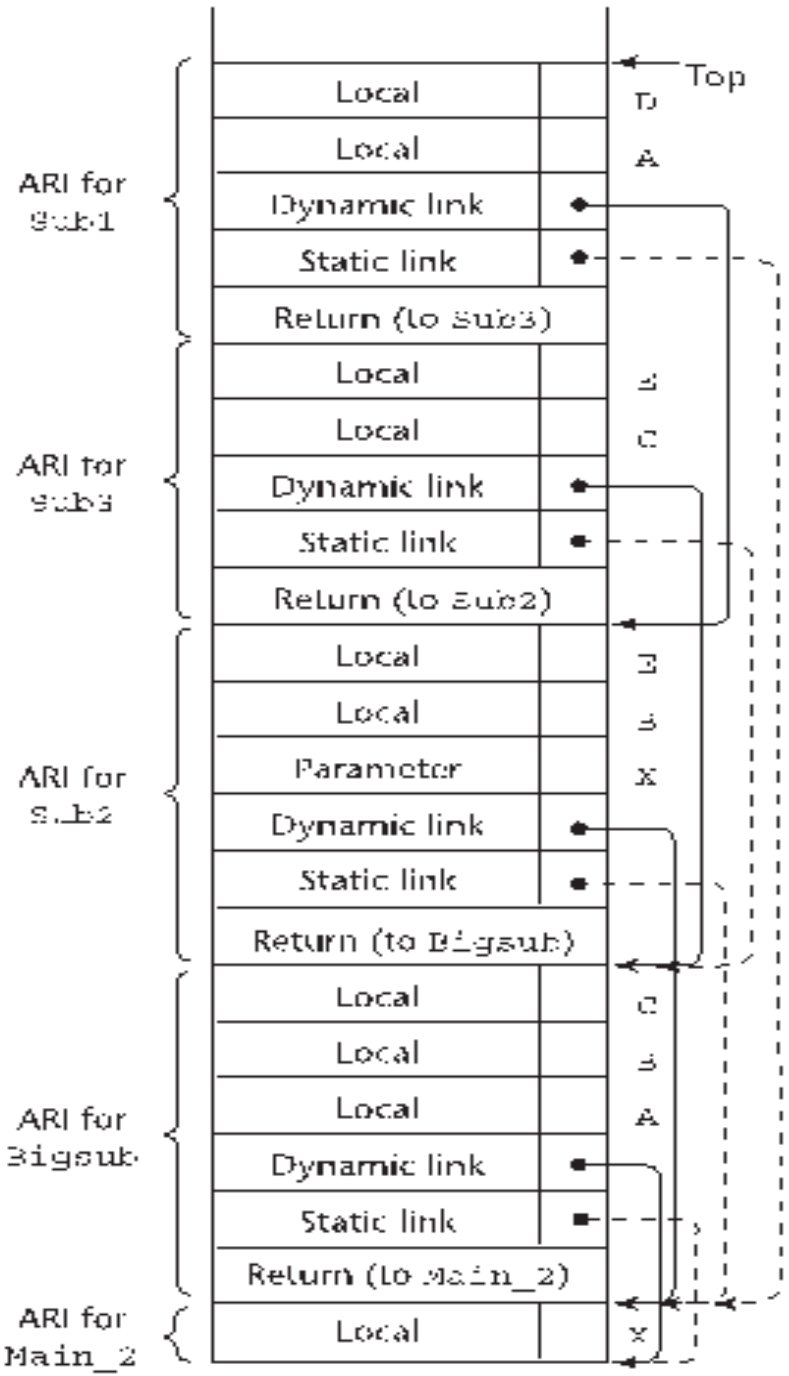
Example Ada Program*

```
procedure Main_2 is
  X : Integer;
  procedure Bigsub is
    A, B, C : Integer;
    procedure Sub1 is
      A, D : Integer;
      begin -- of Sub1
      A := B + C;  <-----------------------1
    end;  -- of Sub1
    procedure Sub2(X : Integer) is
      B, E : Integer;
      procedure Sub3 is
        C, E : Integer;
        begin -- of Sub3
        Sub1;
        E := B + A:    <--------------------2
        end; -- of Sub3
      begin -- of Sub2
      Sub3;
      A := D + E;  <-----------------------3
      end; -- of Sub2 }
    begin -- of Bigsub
    Sub2(7);
    end; -- of Bigsub
  begin
  Bigsub;
```

Example Ada Program (continued)

- Call sequence for `Main_2`

  `Main_2` calls `Bigsub`
  `Bigsub` calls `Sub2`
  `Sub2` calls `Sub3`
  `Sub3` calls `Sub1`



❖ **Implementing Blocks;**

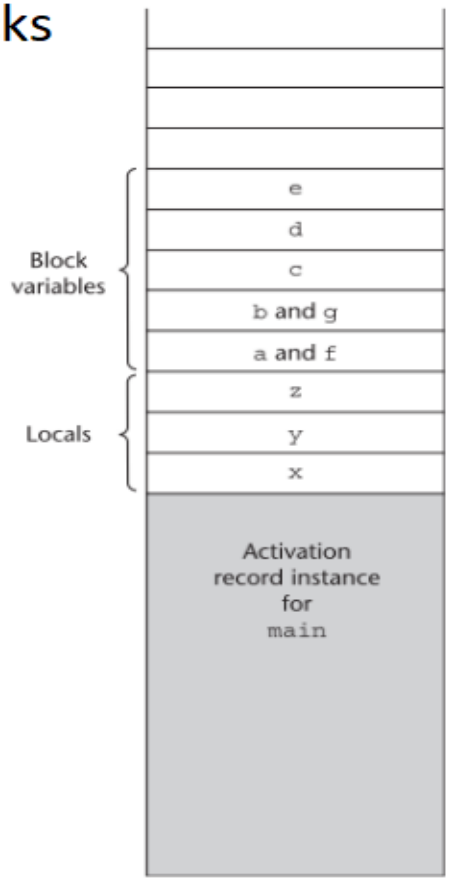
✓ Bloklara, her zaman aynı konumdan çağrılan parametresiz alt programlar olarak davranın. Her bloğun bir aktivasyon kaydı vardır; blok her çalıştırıldığında bir örnek oluşturulur.

✓ Bir blok için gereken maksimum depolama statik olarak belirlenebildiğinden, bu miktardaki alan aktivasyon kaydındaki yerel değişkenlerden sonra tahsis edilebilir.

Implementing Blocks

- Block variable storage when blocks are not treated as parameterless procedures

```
void main() {
  int x, y, z;
  while ( . . . ) {
    int a, b, c;
    . . .
    while ( . . . ) {
      int d, e;
      . . .
    }
  }
  while ( . . . ) {
    int f, g;
    . . .
  }
  . . .
}
```

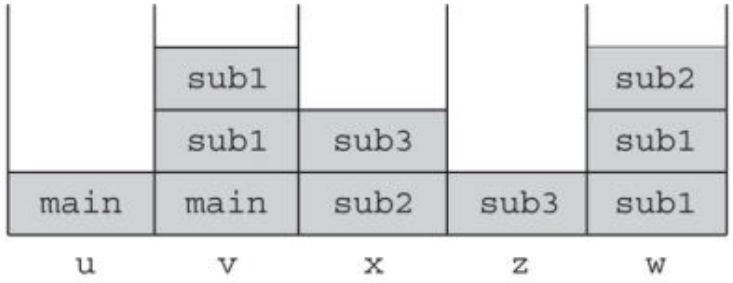


✓ **Derin Erişim (Deep Access):** yerel olmayan referanslar, dinamik zincirdeki aktivasyon kaydı örnekleri aranarak bulunur. Zincirin uzunluğu statik olarak belirlenemez. Her aktivasyon kaydı örneği değişken adlarına sahip olmalıdır.

✓ **Sığ Erişim (Shallow Access):** Local'leri merkezi bir yere koy. Her değişken adı için bir yığın. Her değişken adı için bir girişi olan merkezi bir tablo.

✓ **Shallow Access:**

```
void sub3() {
  int x, z;
  x = u + v;
  …
}
void sub2() {
  int w, x;
  …
}
void sub1() {
  int v, w;
  …
}
void main() {
  int v, u;
  …
}
```



(The names in the stack cells indicate the program units of the variable declaration.)

main calls sub1
sub1 calls sub1
sub1 calls sub2
sub2 calls sub3

4) Ders12 - Logical Programming:

❖ Mantık dilindeki programlar sembolik mantık biçiminde ifade edilir. Sonuç üretmek için mantıksal bir çıkarım işlemi kullanılır.

❖ Mantıksal bir ifade, doğru olabilir de olmayabilir de. Mantıksal ifadeler, Nesnelerden ve nesnelerin birbirleriyle ilişkilerinden oluşur.

❖ Formal mantığın temel ihtiyaçları için kullanılabilecek mantık:

- ➢ Önerileri ifade etme,
- ➢ Öneriler arasındaki ilişkileri ifade etme,
- ➢ Yeni önermelerin diğer önermelerden nasıl çıkarılabileceğini açıklama.

❖ Önerilerdeki nesneler basit terimlerle temsil edilir: sabitler (constants) veya değişkenler (variables).

❖ ***Sabit (Constant):*** bir nesneyi temsil eden bir sembol.

❖ ***Değişken (Variable):*** farklı zamanlarda farklı nesneleri temsil edebilen bir sembol. Zorunlu dillerdeki değişkenlerden farklı.

❖ **Compound Terms (Bileşik Terimler);**

➢ İki bölümden oluşan bileşik terim,

✓ ***Functor:*** ilişkiyi adlandıran işlev simgesi.

✓ ***Tuple (Demet):*** Sıralı parametrelerin listesi.

✓ Örneğin >>> student(john), like(seth, OSX), like(nick, windows), like(jim, linux).

❖ **Logical Operators;**

Name	Symbol	Example	Meaning
negation	¬	¬ a	not a
conjunction	∩	a ∩ b	a and b
disjunction	∪	a ∪ b	a or b
equivalence	≡	a ≡ b	a is equivalent to b
implication	⊃ ⊂	a ⊃ b a ⊂ b	a implies b b implies a

Quantifiers

Name	Example	Meaning
universal	∀X.P	For all X, P is true
existential	∃X.P	There exists a value of X such that P is true

Booleans

No	false value
fail	false value
not(4)	logical not
; / ,	logical or / and (short circuit)
true	true value
Yes	true value

❖ **Sebep Formu (Clausal Form);**

➢ Aynı şeyi ifade etmenin çok fazla yolu vardır. Mesela, öneriler için standart bir form kullanmak gibi.

➢ Sebep formu:

➢ $B_1 \cup B_2 \cup \ldots \cup B_n \subset A_1 \cap A_2 \cap \ldots \cap A_m$ Tüm A'lar doğru ise, en az bir B doğru demektir.

➢ Öncül (Antecedent): sağ taraf, Sonuç (Consequent): sol taraf.

❖ **Terms - Variables & Structures;**

➢ Değişken (Variable): büyük harfle başlayan herhangi bir harf, rakam ve alt çizgi dizisi.

- Örnekleme (Instantiation): bir değişkenin bir değere bağlanması. Sadece bir tam hedefi yerine getirmek için sürdüğü sürece devam eder.
- Yapı (Structure): atomik önermeyi temsil eder. functor (parametre listesi).

❖ **Örnekler;**

- parent(X,Y) :- mother(X,Y). >>> "X, Y'nin annesi ise X, Y'nin ebeveynidir."
- grandparent(X,Z):- parent(X,Y), parent(Y,Z). >>> "Y, Z'nin ebeveyni ve X'de Y'nin ebeveyni ise X, Z'nin atasıdır."

Example

```
speed(ford,100).
speed(chevy,105).
speed(dodge,95).
speed(volvo,80).
time(ford,20).
time(chevy,21).
time(dodge,24).
time(volvo,24).
distance(X,Y) :-   speed(X,Speed),
                   time(X,Time),
                   Y is Speed * Time.
```

A query: `distance(chevy, Chevy_Distance).`

Example

```
likes(jake,chocolate).
likes(jake, apricots).
likes(darcie, licorice).
likes(darcie, apricots).

trace.
likes(jake, X), likes(darcie, X).
((1)) 1 Call: likes(jake, _0)?
(1) 1 Exit: likes(jake, chocolate)
(2) 1 Call: likes(darcie, chocolate)?
(2) 1 Fail: likes(darcie, chocolate)
((1)) 1 Redo: likes(jake, _0)?
(1) 1 Exit: likes(jake, apricots)
(3) 1 Call: likes(darcie, apricots)?
(3) 1 Exit: likes(darcie, apricots)
X = apricots
```