315 Programlama Dilleri

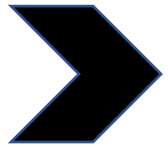
Yrd. Doç. Dr. Ahmet Arif AYDIN

Syntax ve Semantics Kavramları

Syntax(Söz dizimi)

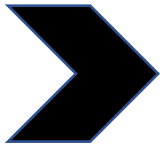
Bir programlama dilinin

- **ifadeleri (expressions)**
- **deyimleri (statement)**
- **program birimleri (program unit)**
- yazım biçimi
- yazım kuralları



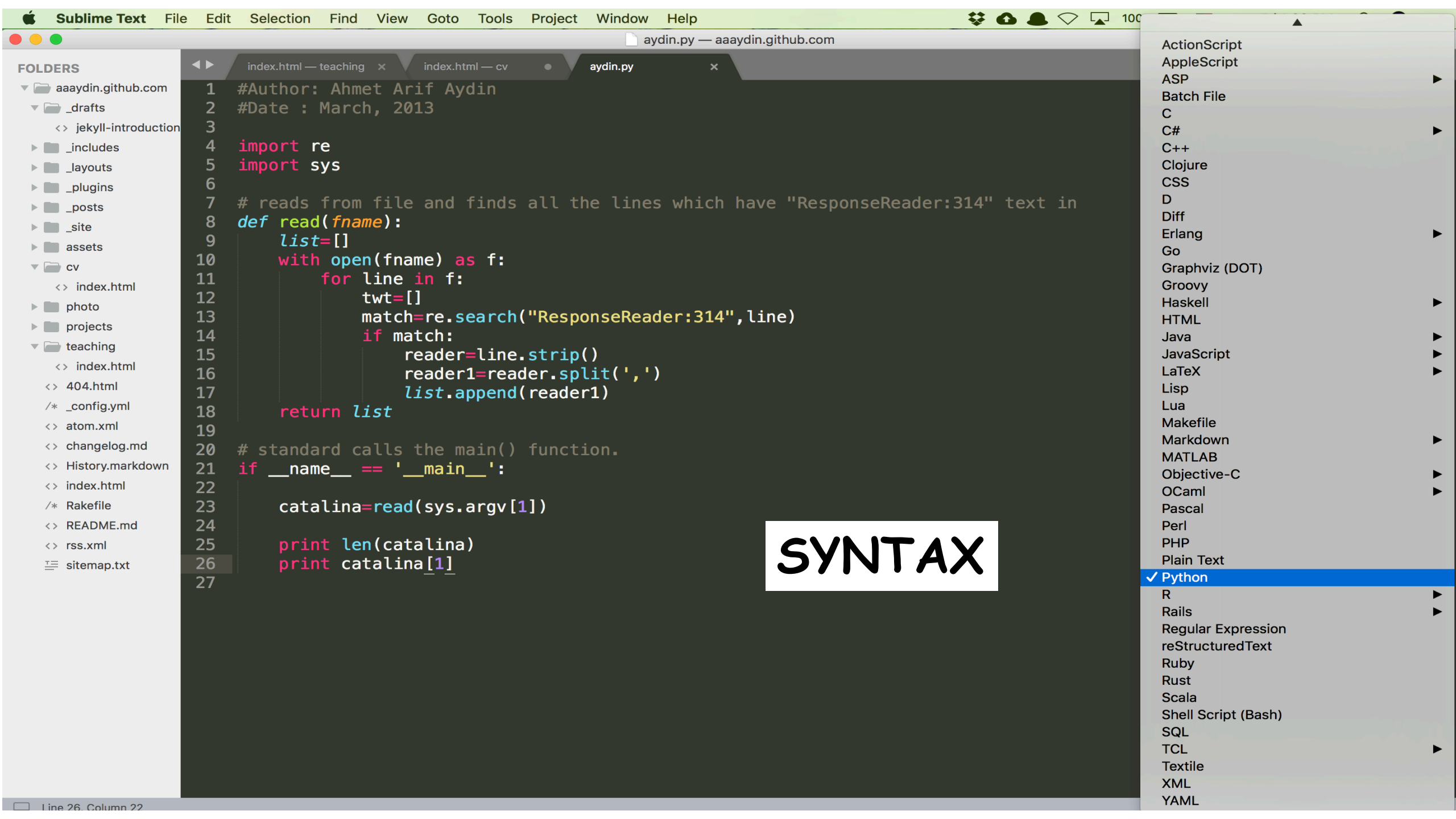
SYNTAX

Syntax Kuralları

SYNTAX 

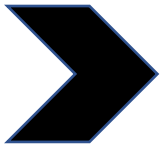
- C ++
  - ifadelerin ardından noktalı virgül (;) kullanılması
- Java
  - **Fonksiyon tanımlama**

```
public Integer fonksiyon()
{
 return değer;
}
```


Sublime Text File Edit Selection Find View Goto Tools Project Window Help
aydin.py — aaaydin.github.com
FOLDERS
aaaydin.github.com
_drafts
jekyll-introduction
_includes
_layouts
_plugins
_posts
_site
assets
cv
index.html
photo
projects
teaching
index.html
404.html
_config.yml
atom.xml
changelog.md
History.markdown
index.html
Rakefile
README.md
rss.xml
sitemap.txt
index.html — teaching
index.html — cv
aydin.py
#Author: Ahmet Arif Aydin
#Date : March, 2013
import re
import sys
# reads from file and finds all the lines which have "ResponseReader:314" text in
def read(fname):
list=[]
with open(fname) as f:
for line in f:
twt=[]
match=re.search("ResponseReader:314",line)
if match:
reader=line.strip()
reader1=reader.split(',')
list.append(reader1)
return list
# standard calls the main() function.
if __name__ == '__main__':
catalina=read(sys.argv[1])
print len(catalina)
print catalina[1]
SYNTAX
ActionScript
AppleScript
ASP
Batch File
C
C#
C++
Clojure
CSS
D
Diff
Erlang
Go
Graphviz (DOT)
Groovy
Haskell
HTML
Java
JavaScript
LaTeX
Lisp
Lua
Makefile
Markdown
MATLAB
Objective-C
OCaml
Pascal
Perl
PHP
Plain Text
Python
R
Rails
Regular Expression
reStructuredText
Ruby
Rust
Scala
Shell Script (Bash)
SQL
TCL
Textile
XML
YAML
Line 26, Column 22

Semantics(Anlam)

Programlama dilinde bulunan ifadelerin, deyimlerin ve program birimlerinin manası



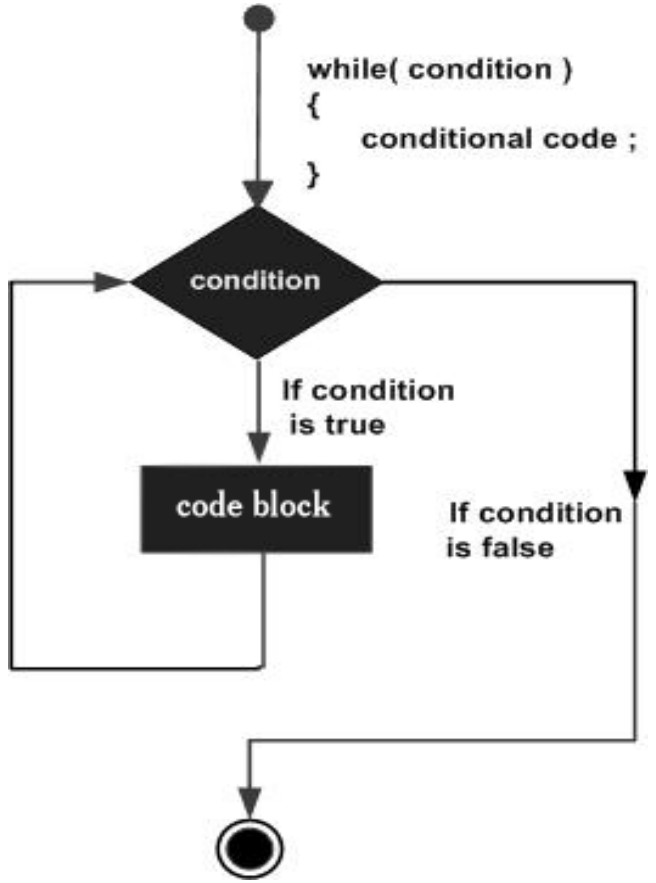
SEMANTICS

Kullanılan yapının *nasıl bir sonuç vereceğini tanımlar*

Java While Döngüsü

SYNTAX

```
While (Boolean Deger True veya False)
{
    işlemler
}
```



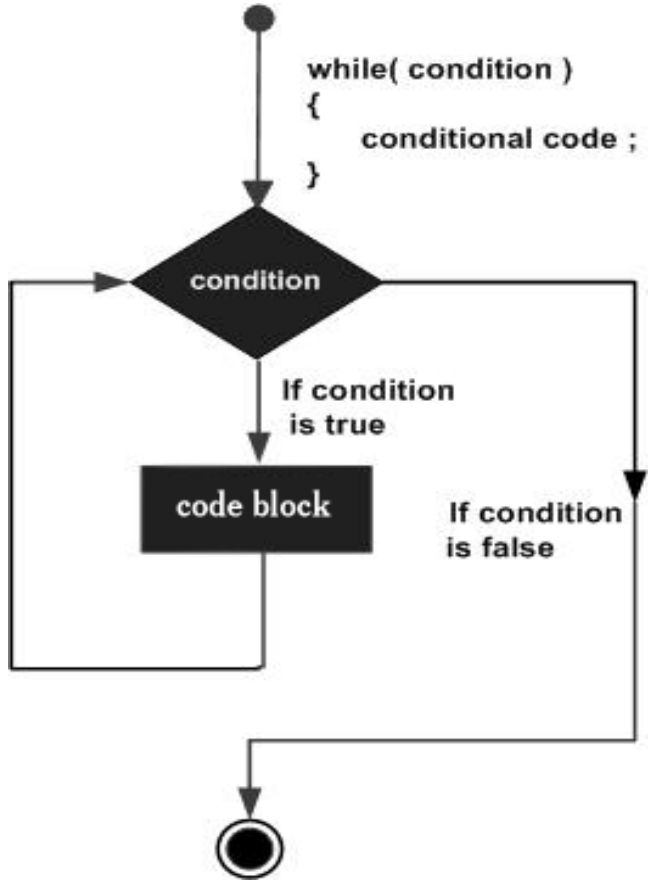
https://www.tutorialspoint.com/java/java_while_loop.htm

Java While Döngüsü

SEMANTICS

Eğer şart doğru ise işlemleri gerçekleştir
Değilse while döngüsünden sonraki satıra geç

İyi bir program için tavsiye edilen semantik ve syntax'in uyumlu bir biçimde çalışmasıdır.

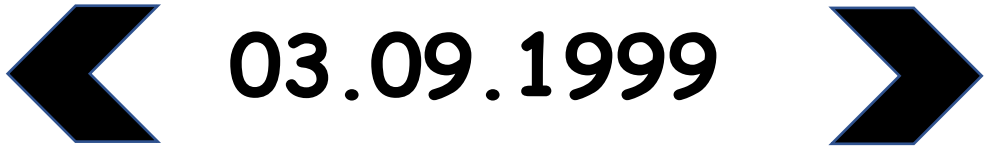


https://www.tutorialspoint.com/java/java_while_loop.htm

Syntax'i Tanımlama

- Doğal diller ve programlama dilleri kullanılan alfabenin karakter setlerinden oluşmaktadır.
- Her dilin kendine ait özellikleri ve söz dizimi kuralları (syntax) bulunmaktadır.
- Dilin kuralları manayı (semantics) tanımlamaktadır.

Türkiye'de
3 Eylül 1999



ABD'de
9 Mart 1999

Syntax

dizi

k--

0

5

--k

```
4  public static void main(String args[]){
5
6      int [] dizi = new int[6];
7
8      int k=5;
9
10     while (k>1)
11         {
12
13         dizi[k]=k--;
14         }
15
16 }
17
```


```
4  public static void main(String args[]){
5      int [] dizi = new int[6];
6      int k=5;
7
8      while (k>1)
9          {
10             dizi[k]=--k;
11         }
12
13 }
```

Lexeme ve Token

Programlama dilinde kod yazılırken

Her bir sözcük (lexeme) olarak tanımlanır

Anlamsal olan en küçük birime ***token denir.***

Lexeme'ler token içerisinde kategorize edilir.

Lexeme ve Token

```
not= ödev*0.3 + vize*0.3 + final*0.4
```

Lexeme	Token
not, ödev, vize, final	Tanımlayıcı (identifier)
=	eşittir
*	çarpma işlemi
+	toplama işlemi
;	noktalı virgül

Dillerin Formal Tanımlayıcıları

Dillerin tanımlanması için genel olarak iki yol bulunmaktadır:

1. Dil Tanıyıcılar (Language Recognizers)
2. Dil Üreticileri (Language Generators)

Dillerin Tanıyıcılar

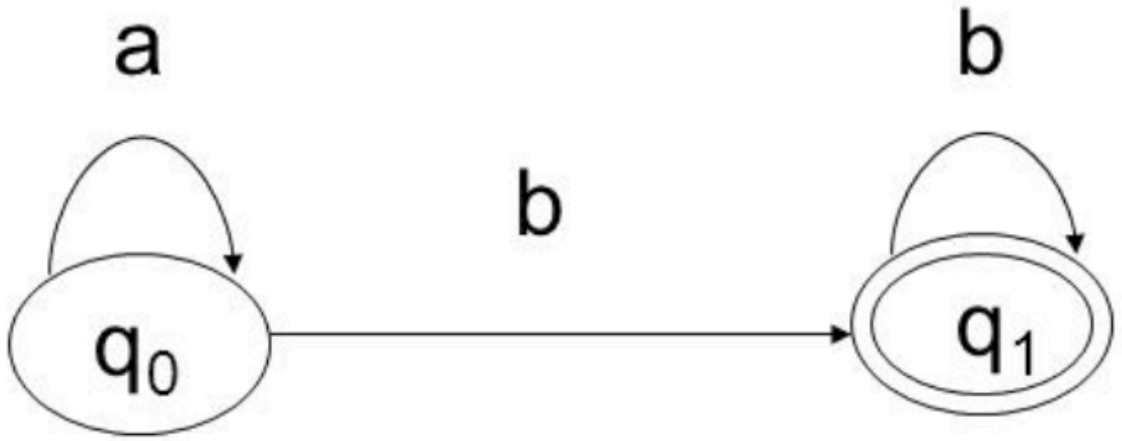
1. Dil Tanıyıcılar (Language Recognizers)
   - Giriş olarak verilen bir karakterler kümesinin bir dilde olup olmadığını kontrol eder.
     - L : bir dil
     - ∑ : Alfabe
     - R : tanımlayıcı cihaz

Sonlu Durum Otomata

1. Dil Tanıyıcılar (Language Recognizers)
   - **Finite State Automata**

$a^* b^+$
$a^n b^m$ n>=0 m>=0

FSA:



Regular language: {b, ab, bb, aab, abb, …}

Dil Üreticiler

2- Dil Üreticiler (Language Generators)

- Bir dilde bulunabilecek cümleleri üretebilen cihazlardır.
- Dil tanıyıcılarla beraber çalışmaktadır.

Syntax için Biçimsel Diller

1. Context-free Grammar
2. Backus-Naur Form

Context-free Grammar

Noam Chomsky

American linguist

- Dil bilimci olan Chomsky 1956 yılında iki gramer sınıfı geliştirdi
  - Context-free gramer
  - Regular gramer

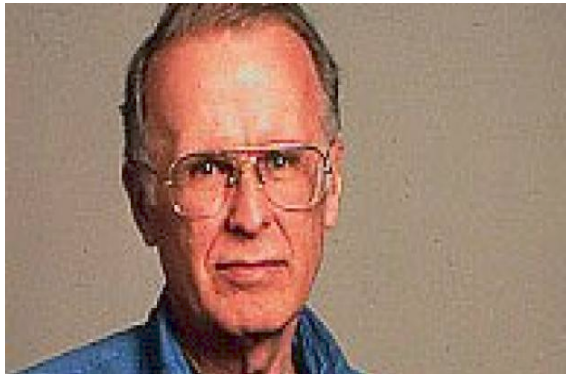
Chomsky geliştirdiği sınıfların programlama dillerine uygulanabileceğini bilmiyordu.

Contex-free sınıfı ile de bütün programlama dillerinin

syntax'i tanımlanabilmektedir.

Backus-Naur Form (BNF)

- 1959 yılında ALGOL 58 ile beraber John Backus programlama dilleri syntax için yeni bir biçimsel gösterim (notation) ortaya konuldu.
- 1960 yılında ALGOL 60 ile beraber Peter Naur, John Backus tarafından sunulan gösterimde bazı değişiklikler yaparak geliştirdi ve bu form Backus-Naur Form (BNF) olarak anılmaya başlandı.



John Backus

Peter Naur

Meta language

- Bir dili tanımlamak için kullanılan baska bir dile **üst-dil** (metalanguage) denir.
- Programlama dillerinin metalanguage'i BNF dir.
- BNF syntaxi tanımlamak için soyut ifadeler kullanmaktadır

Örnek: Atama işlemi

<atama> → <değişken> = <ifade>

yapılacak olan işlemin adını tanımlar

<atama> işleminin nasıl yapılacağını göstermektedir

lexeme, token ve referenslar bulunmaktadır.

Örnek: Atama işlemi

Yukarıdaki yapının hepsine birden ***kural (rule)*** denir.

Bu kural ile *atama soyut tanımı* değişken, eşittir ve ifade ile belirlenmektedir.

BNF

<if-yapısı> →if (<karşılaştırma>) <işlem>

< if-yapısı > →if (< karşılaştırma >) < işlem > else < işlem >

BNF

<if-yapısı> →if (<karşılaştırma>) <işlem>

< if-yapısı > →if (< karşılaştırma >) < işlem > else < işlem >

Birden fazla kuralı birleştirmek için OR | sembolu kullanılır.

< if-yapısı > → if (< karşılaştırma >) < işlem >
| if (< karşılaştırma >) < işlem > else < işlem >

Bu yapı if yapısının birden fazla formunun olduğunu tanımlar.

BNF: Recursion

BNF de sıralı listelerin veya ardışık yapıları (1, 2, 3,.....)

tanımlamak için öz yineleme (recursion) kullanılır.

<liste> → tanımlayıcı

| tanımlayıcı , <liste>

BNF

- BNF de bulunan soyut tanımlamalar Non-terminal sembols (NT) olarak ifade edilir.
- Kurallar da bulunan Lexeme ve token Terminal (T) olarak adlandırılır.
- BNF (veya Gramer) kurallar toplulugu olarak ifade edilir.

<atama> → <değişken> = <ifade>

Gramer ve Türetme

Gramer

<program> → başla < ifade-listesi > bitir

<ifade-listesi> → <atama>

| < atama > ; < ifade-listesi >

< atama > → <değişken> = <ifade>

< değişken > → X | Y | Z

< ifade > →< değişken > + < değişken >

| < değişken > – < değişken >

| < değişken >

- => sembolü ***türetir*** diye okunur.
- Her bir aşamada NT (Non-terminal) olan semboller Terminal olanlarla değiştirilerek türetme gerçekleştirilir.
- Türetilen her bir string **sentential form** olarak tanımlanır.
- Bütün NT semboller T olanlarla değiştirilinceye kadar işlem devam eder.
- İşlem soldan sağa doğru yapıldığı için ***leftmost derivations*** olarak adlandırılır.

Grammer ve Türetme

<program> → başla < ifade-listesi > bitir

<ifade-listesi> → <atama>

| < atama > ; < ifade-listesi >

< atama > → <değişken> = <ifade>

< değişken > → X | Y | Z

< ifade > →< değişken > + < değişken >

| < değişken > - < değişken >

| < değişken >

<program> => başla < ifade-listesi > bitir

=> başla < atama >;< ifade-listesi > bitir

=> başla < değişken >=< ifade >;< ifade-listesi > bitir

=> başla X = < ifade > ; < ifade-listesi > bitir

=> başla X = < değişken > + < değişken > ; < ifade-listesi > bitir

=> başla X= Y + < değişken > ; < ifade-listesi > bitir

=> başla X = Y + Z ; < ifade-listesi > bitir

=> başla X =Y +Z ; < atama > bitir

=> başla X = Y + Z ; < değişken > = < ifade > bitir

=> başla X =Y +Z ; Y =< ifade > bitir

=> başla X =Y +Z ; Y =< değişken > bitir

=> başla X =Y +Z ; Y = Z bitir

Örnek Gramer: Atama

<atama> → <değişken> =<ifade>

<değişken> → A|B|C

<ifade> → < değişken > + < ifade >

| < değişken > * < ifade >

| (< ifade >)

| < değişken >

Örnek Gramer: Atama

<atama> → <değişken> =<ifade>

<değişken> → A|B|C

<ifade> → < değişken > + < ifade >

| < değişken > * < ifade >

| (< ifade >)

| < değişken >

<atama> => <değişken> = < ifade >

=> A = <ifade>

=> A = <değişken> * <ifade>

=> A = B * (<ifade>)

=> A = B * (<değişken> + <ifade>)

=> A = B * (A + <ifade>)

=> A = B * (A + <değişken>)

=> A = B * (A + C)

A =B*(A+C) türetilir.

Parse Trees: Ayrıştırma Ağaçları

Bir dilin hiyerarşik yapısını tanımlayan

aşamalı yapıya *ayrıştırma ağacı (parse tree)* denir.

Gramerler doğal olarak

bir dilin cümlelerinin hiyerarşik yapısını tanımlar

Ayrıştırma Ağacı: A=B*(A+C)

<atama> => <değişken> = < ifade >

=> A = <ifade>

=> A = <değişken> * <ifade>

=> A = B * (<ifade>)

=> A = B * (<değişken> + <ifade>)

=> A = B * (A + <ifade>)

=> A = B * (A + <değişken>)

=> A = B * (A + C)

<atama> → <değişken> =<ifade>

<değişken> → A|B|C

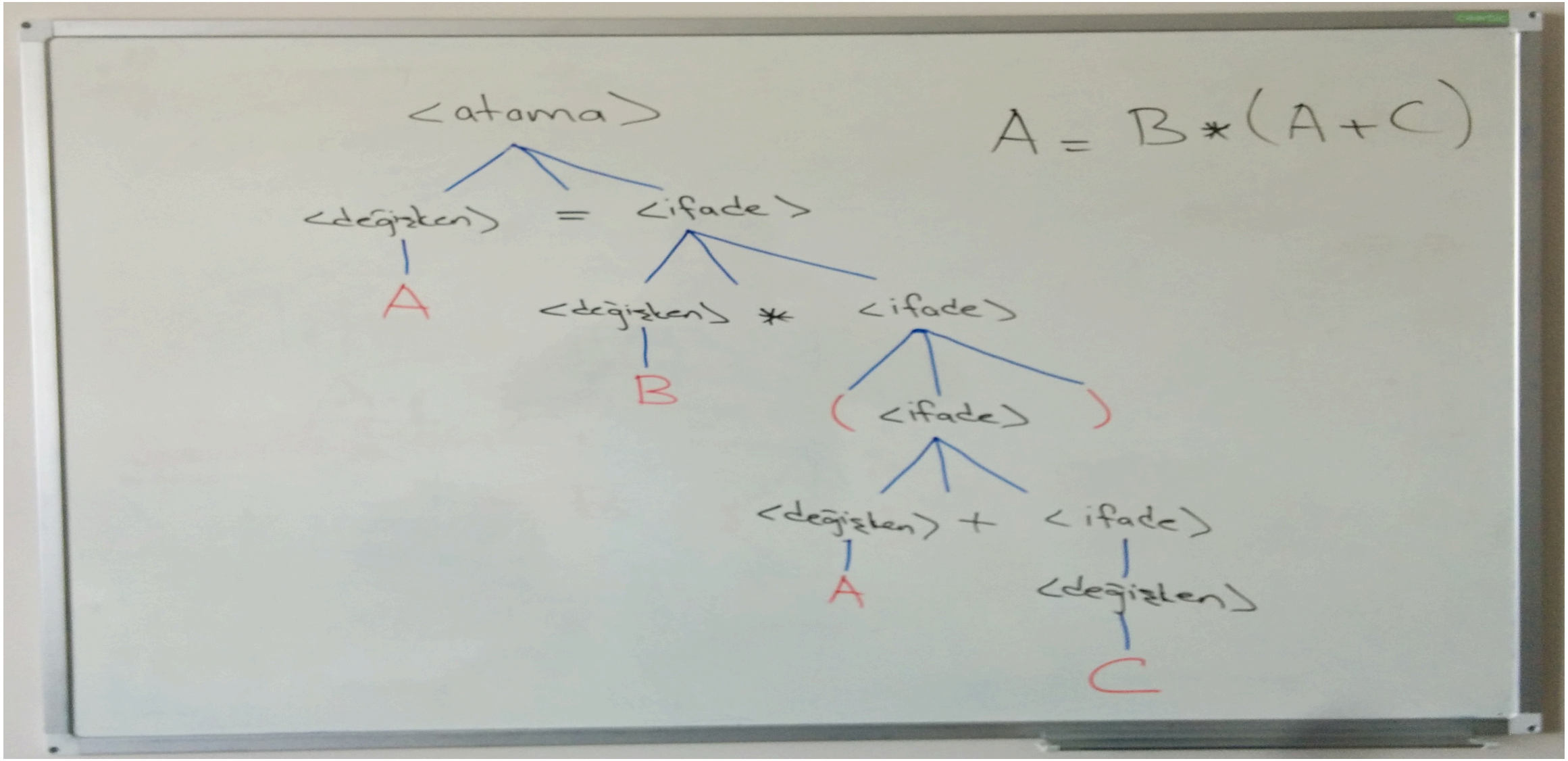
<ifade> → < değişken > + < ifade >

| < değişken > * < ifade >

| (< ifade >)

| < değişken >

Ayrıştırma Ağacı: A=B*(A+C)



Belirsizlik: Ambiguity

Bir gramerde bir işlem için birden fazla parse tree üretilmesine belirsizlik denir.

<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <expr>
| <expr> * <expr>
| (<expr>)
| <id>

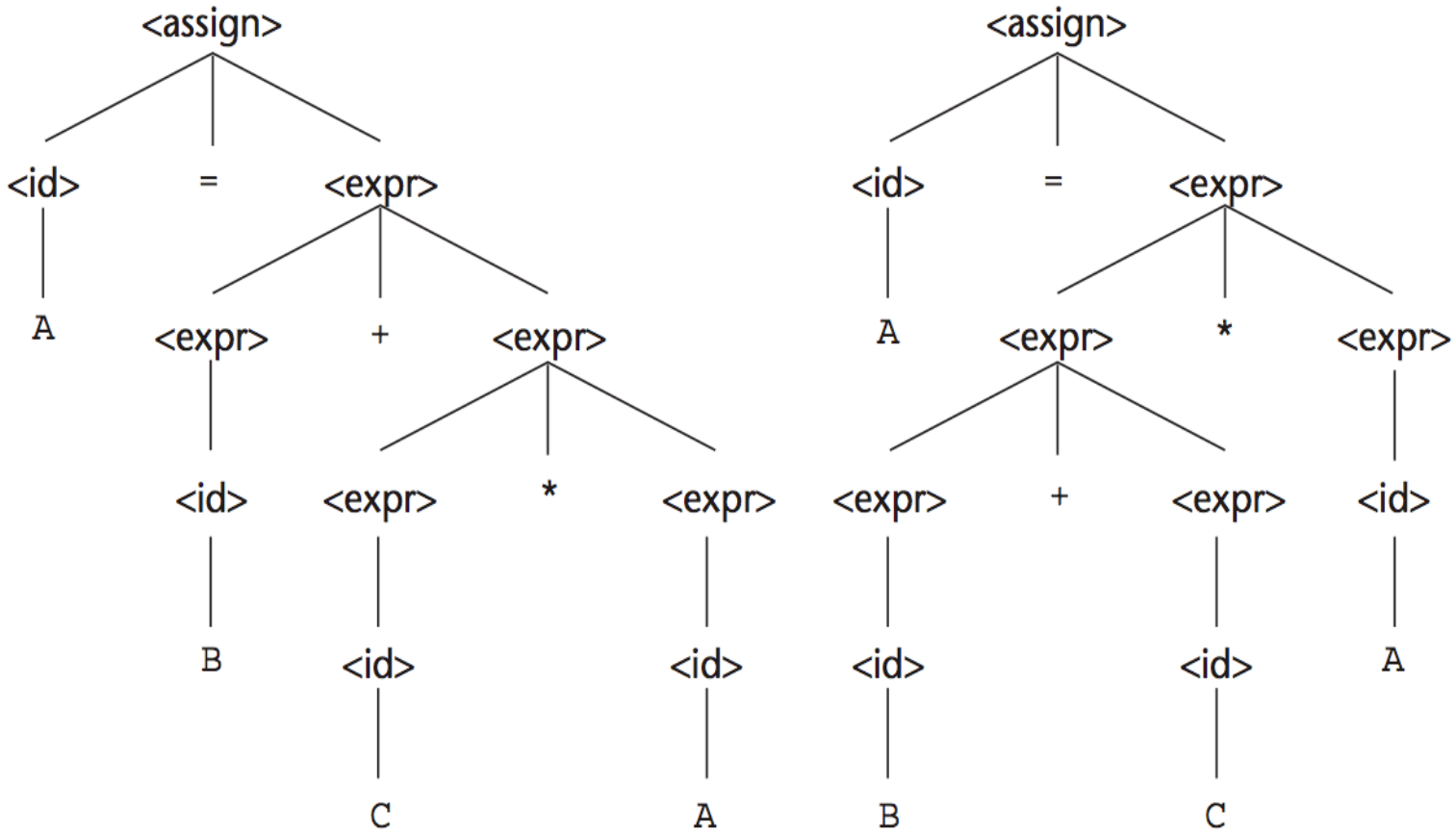
A = B + C *A

Belirsizlik: Ambiguity

Bir gramerde bir işlem için birden fazla parse tree üretilmesine belirsizlik denir.

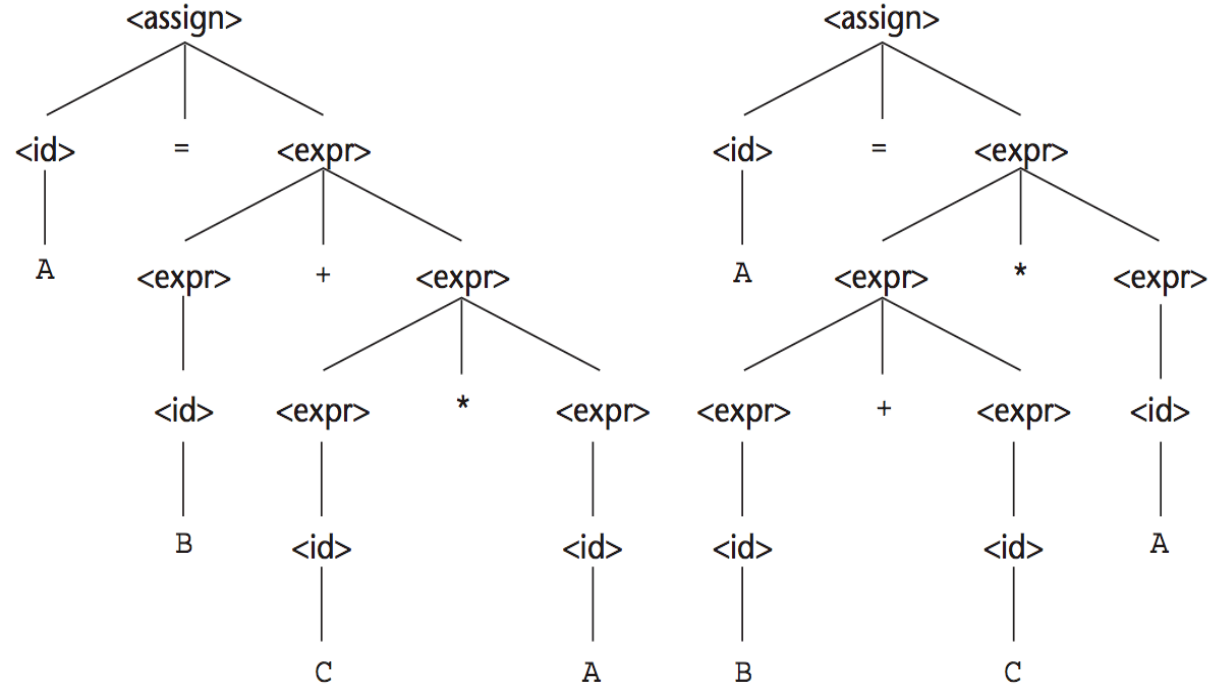
<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <expr>
| <expr> * <expr>
| (<expr>)
| <id>

A = B + C *A



Operatör Önceliği

- Bir gramerde operatör önceliği tanımlanmazsa belirsizlik ortaya çıkabilir.
- Operatör önceliği dili tanımlayanlar tarafından belirlenir.
- Ayrıştırma ağacından operatör önceliği çıkarılabilir (üst seviyede görülen operatörün önceliği bulunmaktadır)



Belirsizlik: Ambiguity

Bir gramerde ki belirsizlik kurallar yardımıyla ortadan kaldırılabilir.

<assign> → <id> = <expr>

<id> → A | B | C

<expr> → <expr> + <expr>
| <expr> * <expr>
| (<expr>)
| <id>

Ambiguous

<assign> → <id> = <expr>

<id> → A | B | C

<expr> → <expr> + <term>
| <term>

<term> → <term> * <factor>
| <factor>

<factor> → (<expr>)
| <id>

Unambiguous

Unambiguous Grammar

Belirsiz olmayan gramer left-most veya right-most derivations ile aynı ayrıştırma ağacını üretir.

A = B + C * A

<assign> => <id> = <expr>
=> A = <expr>
=> A = <expr> + <term>
=> A = <term> + <term>
=> A = <factor> + <term>
=> A = <id> + <term>
=> A = B + <term>
=> A = B + <term> * <factor>
=> A = B + <factor> * <factor>
=> A = B + <id> * <factor>
=> A = B + C * <factor>
=> A = B + C * <id>
=> A = B + C * A

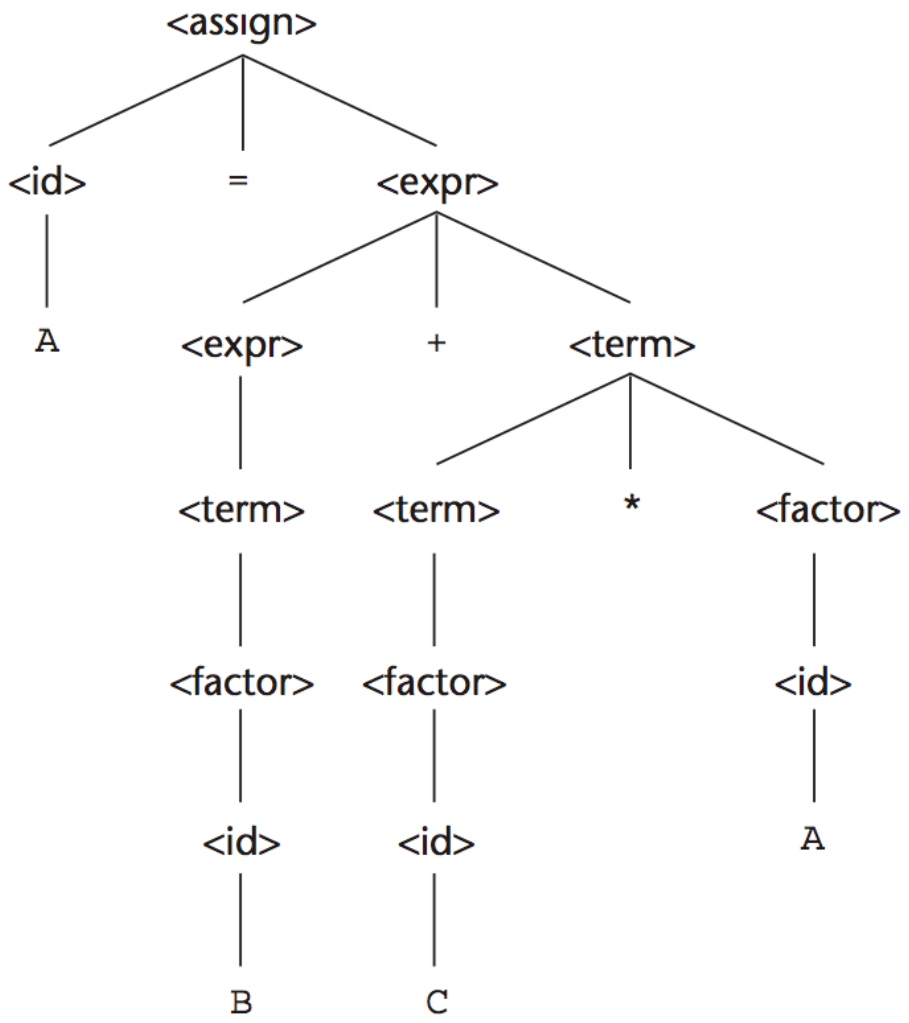
<assign> => <id> = <expr>
=> <id> = <expr> + <term>
=> <id> = <expr> + <term> * <factor>
=> <id> = <expr> + <term> * <id>
=> <id> = <expr> + <term> * A
=> <id> = <expr> + <factor> * A
=> <id> = <expr> + <id> * A
=> <id> = <expr> + C * A
=> <id> = <term> + C * A
=> <id> = <factor> + C * A
=> <id> = <id> + C * A
=> <id> = B + C * A
=> A = B + C * A

Unambiguous Grammar

A = B + C * A

<assign> => <id> = <expr>
=> A = <expr>
=> A = <expr> + <term>
=> A = <term> + <term>
=> A = <factor> + <term>
=> A = <id> + <term>
=> A = B + <term>
=> A = B + <term> * <factor>
=> A = B + <factor> * <factor>
=> A = B + <id> * <factor>
=> A = B + C * <factor>
=> A = B + C * <id>
=> A = B + C * A

<assign> => <id> = <expr>
=> <id> = <expr> + <term>
=> <id> = <expr> + <term> * <factor>
=> <id> = <expr> + <term> * <id>
=> <id> = <expr> + <term> * A
=> <id> = <expr> + <factor> * A
=> <id> = <expr> + <id> * A
=> <id> = <expr> + C * A
=> <id> = <term> + C * A
=> <id> = <factor> + C * A
=> <id> = <id> + C * A
=> <id> = B + C * A
=> A = B + C * A



Operator Birleşirliği: Associativity

- Toplama ve carpma işlemleri associative' dir
  - (A + B) + C = A + (B + C).
  - (A *B) * C = A * (B *C).
- Çıkarma ve bölme işlemleri associative degildir.

EBNF: Extended BNF

BNF:

<expr> → <expr> + <term>
| <expr> - <term>
| <term>
<term> → <term> * <factor>
| <term> / <factor>
| <factor>
<factor> → <exp> ** <factor>
<exp>
<exp> → (<expr>)
| id

EBNF:

<expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}
<factor> → <exp> { ** <exp>}
<exp> → (<expr>)
| id

- EBNF BNF in tanımlayıcı özelliklerine ek olarak okunabilirliği ve yazılabilirliği geliştirmiştir.
- Gösterimi kolaylaştıran meta semboller bulunmaktadır.
  - [ ] köşeli parantez. **seçilebilirlik**
    - **if** (<şart>) <işlem> [**else** <şart>]
  - { } parantez, **yineleme** (recursion)
  - | (or)

Attribute Grammar

- Context-free gramer ile tanımlanabilir
- Bir dilin üretilmesi için gereken kurallar topluluğuna Gramer denir.
- Bir dilin cümleleri NT (non-terminal) olan başlangıç sembolü ile başlar
- Tanımlanan kuralları kullanarak çıkarım (derivation) yapılır.

Attribute Grammar

Gramer = (T, NT, Kurallar, Başla(S))

T = {a, b, c}

NT = {S, B, C}

Kurallar:

1. S —>aSBC
2. S—>aBC
3. CB—>BC
4. aB—>ab
5. bB—>bb
6. bC—>bc
7. cC—>cc

$a^2b^2c^2$

Atama işlemi: Attribute Grammar

1. Syntax rule: <assign> → <var> = <expr>
   Semantic rule: <expr>.expected_type ← <var>.actual_type
2. Syntax rule: <expr> → <var>[2] + <var>[3]
   Semantic rule: <expr>.actual_type ←
   if (<var>[2].actual_type = int) and
   (<var>[3].actual_type = int)
   then int
   else real
   end if
   Predicate: <expr>.actual_type == <expr>.expected_type
3. Syntax rule: <expr> → <var>
   Semantic rule: <expr>.actual_type ← <var>.actual_type
   Predicate: <expr>.actual_type == <expr>.expected_type
4. Syntax rule: <var> → A | B | C
   Semantic rule: <var>.actual_type ← look-up (<var>.string)

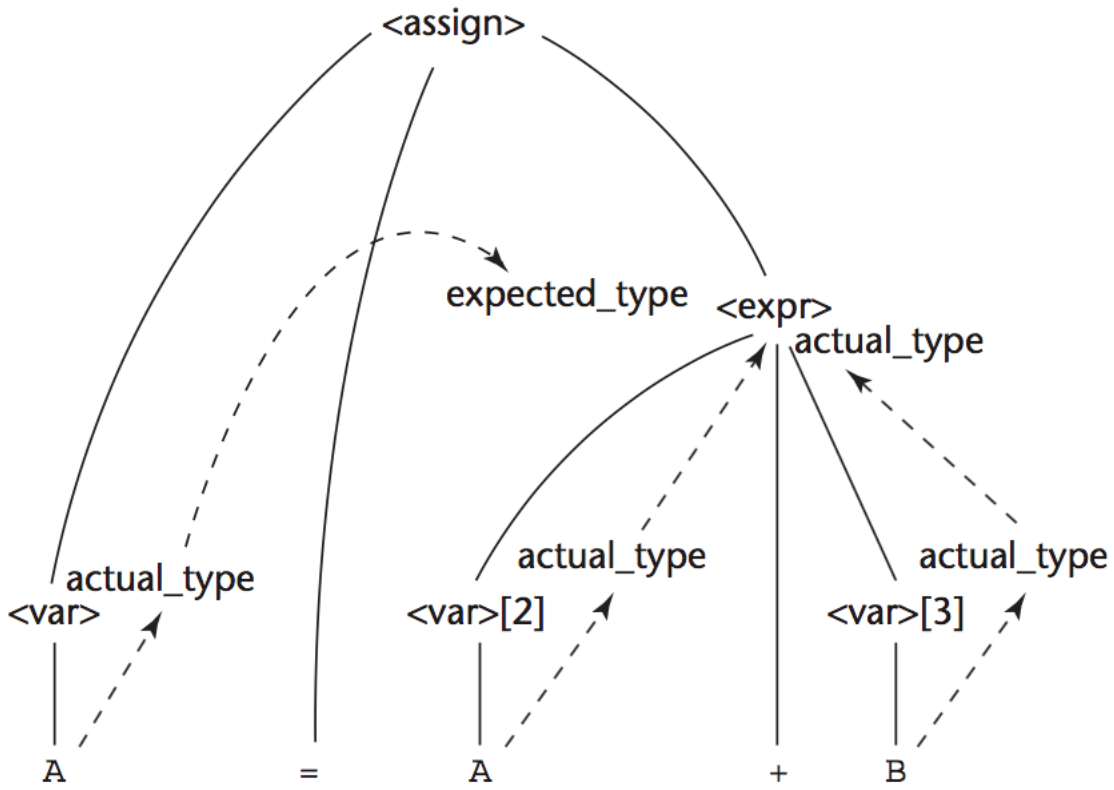
The look-up function looks up a given variable name in the symbol table and returns the variable's type.

Syntax: Yazım Kuralı

Semantic: Anlam

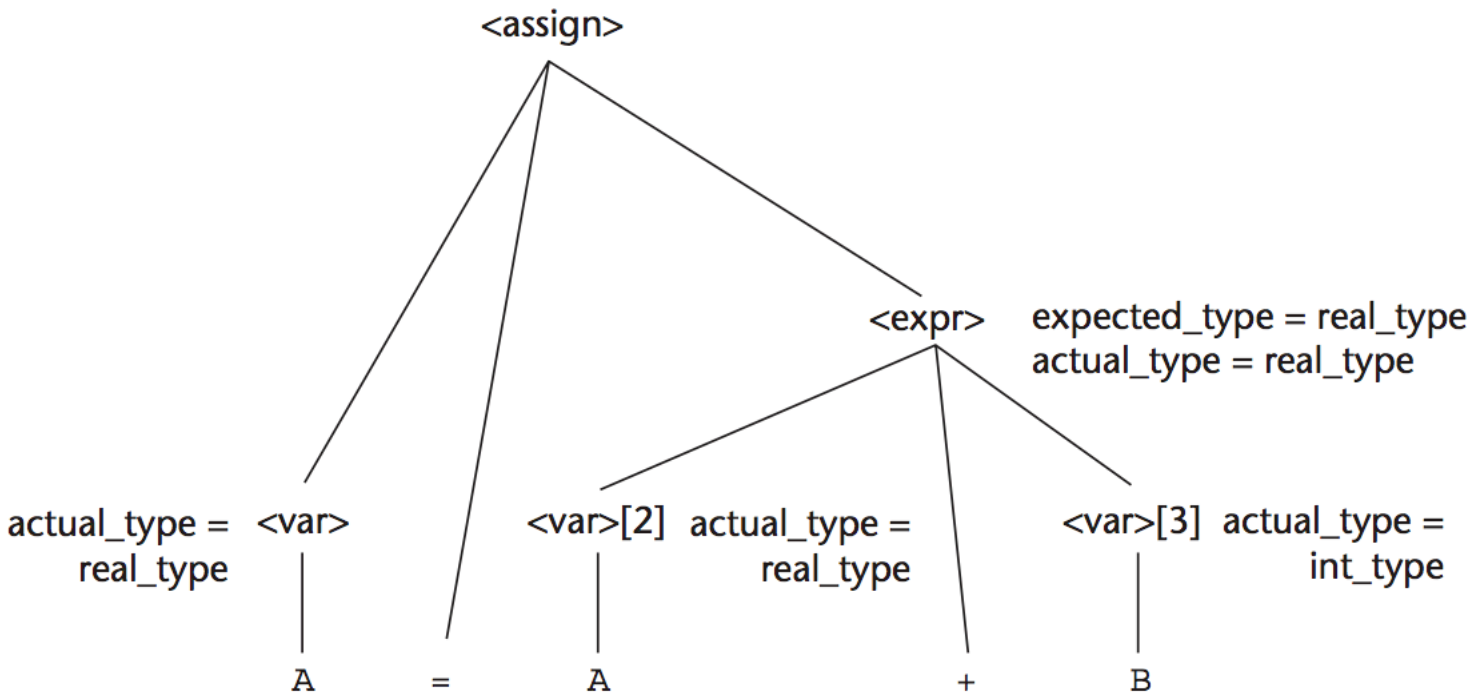
Predicate : Karşılaştırma belirtimi

Nitelik Değerlerinin Hesaplanması



- Niteliklerin hepsi önceden var olan değerlerden alınıyorsa (inherited) ayrıştırma agacı *top-down* olarak kök dizinden (root) başlayıp dallara dogru hesaplama yapılır.
- Alternatif olarak *bottom-up* olarak dallardan başlayıp kök dizine doğru çıkılır.

Tam Baglanmış Ayrıştırma Ağacı



Proğramların Manasının Tanımlanması

1- İşlemsel Anlam (Operational Semantics)

- **Program = soyut makine programi**
- Bir ifadenin veya programın manasını makine üzerinde çalısırken tanımlar.
- Makinedeki etki durumunda oluşan değişimler dizisi olarak tanımlanır.
- Makinenin durumu ise işlem sırasında kaydedilen değerler kolleksiyonudur.

Programların Manasının Tanımlanması

2- Fonksiyonel Anlam (Denotational Semantics)

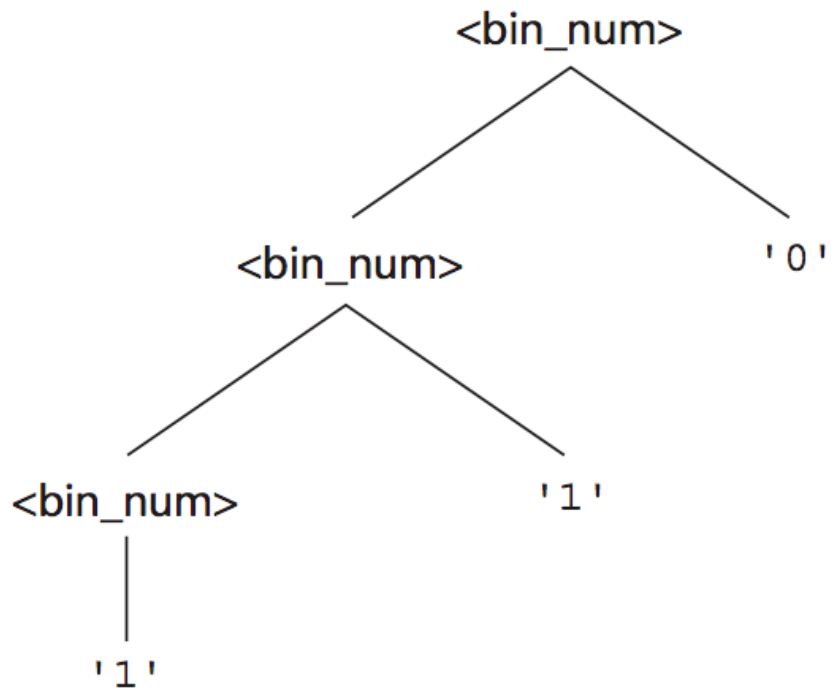
- **Program = Matematiksel ifade**
- Temelinde Recursive fonksiyon teorisi bulunmaktadır
- Nesneler ve eşleştirme fonksiyonları matematiksel olarak tanımlandığı için Denotational ismi verilmiştir.

Programların Manasının Tanımlanması

2- Fonksiyonel Anlam (Denotational Semantics)

<bin_num> → '0'
| '1'
| <bin_num> '0'
| <bin_num> '1'

M_{bin}('0') = 0
M_{bin}('1') = 1
M_{bin}(<bin_num> '0') = 2 * M_{bin}(<bin_num>)
M_{bin}(<bin_num> '1') = 2 * M_{bin}(<bin_num>) + 1



Programların Manasının Tanımlanması

3- Kural Tabanlı Anlam (Axiomatic Semantics)

- **Program = Özellikler kümesi**
- Programlar ile alakalı teorem ıspatında kullanılabilir