### A real example

BNF

Bilgisayar biliminde, Genişletilmiş Backus–Naur formu (EBNF) bağlamdan bağımsız dilbilgisini (context-free grammar) ifade etmek için kullanılan bir notasyondur. Başka bir ifadeyle, bilgisayar programlama dillerini ve biçimsel dilleri (formal languages) tanımlamanın bir yoludur. Backus–Naur formu (BNF) notasyonunu temel alır.  
  
Niklaus Wirth tarafından geliştirilmiştir. En yaygın kullanılan türevleri standartlaştırılmıştır

Below is a sample BNF grammar:

S := '-' FN |

FN

FN := DL |

DL '.' DL

DL := D |

D DL

D := '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

The different symbols here are all abbreviations: S is the start symbol, FN produces a fractional number, DL is a digit list, while D is a digit.

Valid sentences in the language described by this grammar are all numbers, possibly fractional, and possibly negative. To produce a number, start with the start symbol S:

S

Then replace the S symbol with one of its productions. In this case we choose not to put a '-' in front of the number, so we use the plain FN production and replace S by FN:

FN

The next step is then to replace the FN symbol with one of its productions. We want a fractional number, so we choose the production that creates two decimal lists with a '.' between them, and after that we keep choosing replacing a symbol with one of its productions once per line in the example below:

DL . DL

D . DL

3 . DL

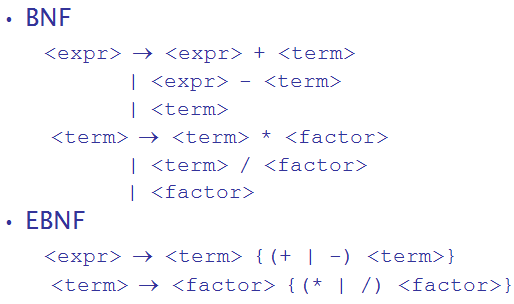
3 . D DL

3 . D D

3 . 1 D

3 . 1 4

Here we've produced the fractional number 3.14. How to produce the number -5 is left as an exercise for the reader. To make sure you understand this you should also study the grammar until you understand why the string 3..14 cannot be produced with these production rules.



<exp> ::= <exp> + <term> | <exp> - <term> | <term>

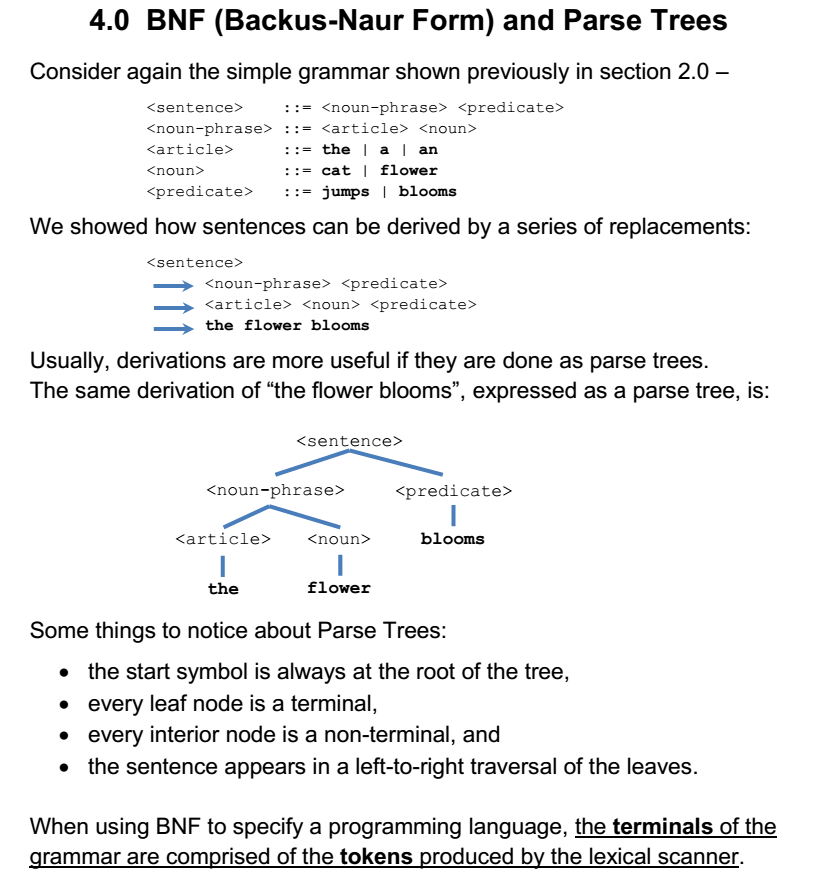
<term> ::= <term> \* <power> | <term> / <power> | <power>

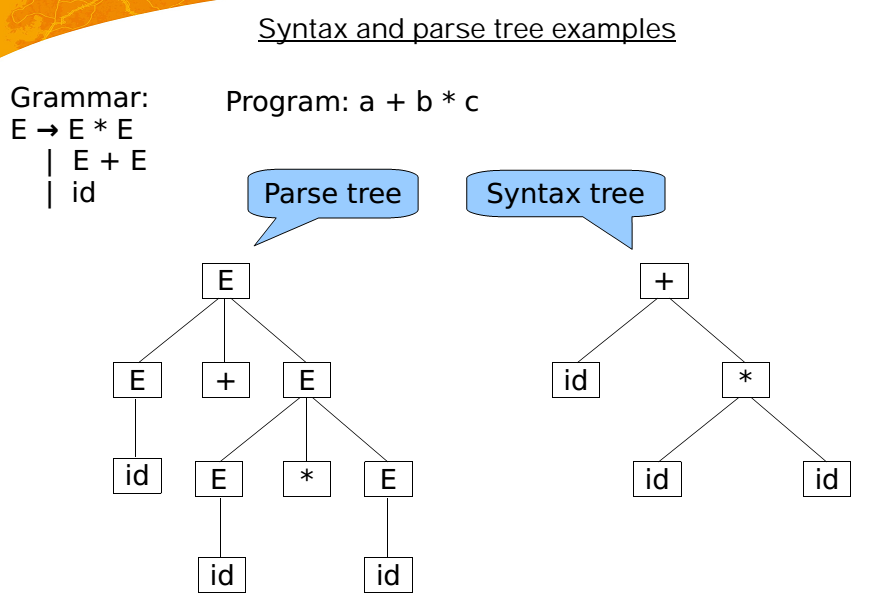
<power> ::= <factor> ^ <power> | <factor>

<factor> ::= ( <exp> ) | <int>

<int> ::= <digit> <int> | <digit>

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9





**Parçalama Ağacı (Parse Tree)**

Parçalam işlemi  (parsing) bilgisayar bilimlerinde çeşitli amaçlar için kullanılmaktadır. Özellikle de dil ile ilgili işlemlerin hemen hepsinde ihtiyaç duyulan bir işlemdir. Örneğin bir programlama dilinde yazılan komutların algılanması için öncelikle kelimeleirn parçalanması (parse) gerekir. Benzer şekilde dopal dil işleme (natural language processing) işlemlerinde de doğal dilde bulunan kelimelerin algılanması bir parçalamadan (ek ve köklerin ayrılmasından) sonra gerçekleşmektedir.

Çeşitli sebeplerle kullanılan parçalama ağaçları basitçe verilen bir dilbilgisine (grammer) göre verilen cümlenin (veya kelimenin) nasıl parçalandığını şekilsel olarak gösteren ağaçlardır:

Örneğin aşağıda [BNF yapısında](http://www.bilgisayarkavramlari.com/2008/12/09/backus-normal-form-bnf/)verilmiş dili ele alalım:

<dil>::=<işlem>

<işlem> ::= <işlem> + <terim> | <işlem> – <terim> | <terim>

<terim> ::= <terim> \* <unsur> | <terim> / <unsur> | <unsur>

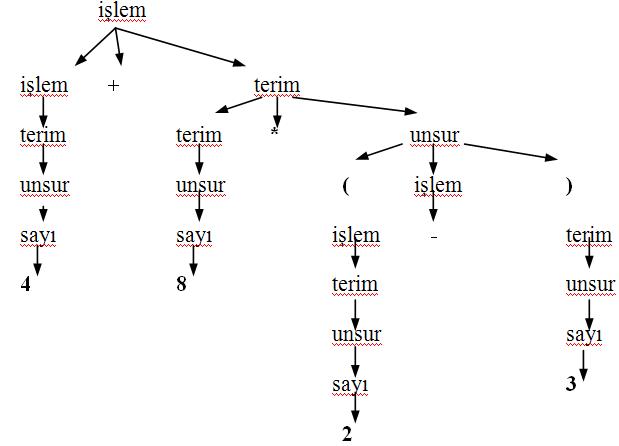
<unsur> ::= **sayı** | **(**<işlem> **)**

<sayı> ::= 1|2|3…|9|0

Bu dilde aşağıdaki örneğin nasıl parçalandığını inceleyelim:

4+8\*(2-3)

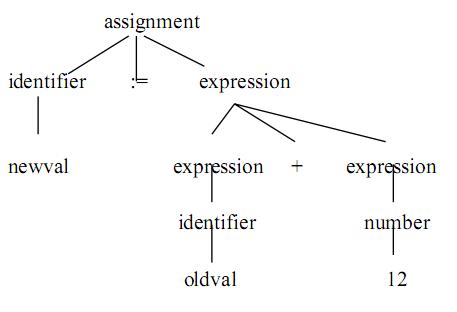
Bu dilde tanımlı olan yukarıdaki işlemin parçalama ağacı aşağıdaki şekildedir:



yukarıdaki şekilde devamlı (nonterminal) terimlerden sonuncu (terminal) terimlere kadar bir BNF dilinin nasıl açıldığı gösterilmiştir.

**Söz Dizimsel Analiz (Syntax Analysis)**

Parser, lexerın oluşturduğu token listesini alır ve bir ağaç yapısı üretir. Bu oluşturulan ağaca "parse ağacı" denir. Token listesinden oluşturulmuş olan parse ağacı yine tokenlerden oluşmuş bir yapıdır ve derleyicinin tanıdığı gramere göre oluşturulur. Parser, ağacı oluştururken gramere uygun olmayan bir token ile karşılaşırsa bunu hata olarak bildirir. Parserın ürettiği bu hataya "gramer hatası" ya da "parse hatası (parse error)" denir.



**Resim 1:** Parse ağacının yapısı

Parser,

* Token içinde bulunan ve tokenin ifade ettiği metin ile uğraşmaz.
* Tokenin karakter katarında hangi pozisyonda bulunduğu ile ilgilenmez.
* Lexer gibi tokenin ifade ettiği anlam ile uğraşmaz.

Parser, yalnızca token listesinin dizilişine göre gramere uygunluğu kontrol eder. Bunun sonucunda da bu token listesinden oluşan bir parse ağacı üretir.

Bir dilin söz dizimi Bağlamdan Bağımsız Dil Bilgisi (Context Free Grammar - CFG) ile belirlenir. CFG'de belirlenen kurallar genellikle özyinelemelidir. (recursive)

Söz dizimsel analizde, verilen bir programın CFG tarafından belirtilen kurallara uyup uymadığı kontrol edilir. Eğer uyuyorsa bu program için bir ayrıştırma ağacı oluşturulur.

Örnek: CFG, BNF (Back Naur Form) kullanılarak yazılır:

assignmnet -> identifier := expression

expression -> identifier

expression -> number

expression -> expression + expression

Ayrıştırma ağacının oluşturulmasına bağlı olarak farklı ayrıştırma teknikleri bulunmaktadır.

Ayrıştırma teknikleri iki grupta toplanabilir:

* Yukarıdan Aşağıya Ayrıştırma (Top - Down Parsing)
* Aşağıdan Yukarıya Ayrıştırma (Bottom - Up Parsing)

**Yukarıdan Aşağıya Ayrıştırma**

 Ayrıştırma ağacının yapısı kökten başlar ve yapraklara doğru ilerler. Etkin yukarıdan aşağıya ayrıştırıcılar kolaylıkla çizilebilir.

**Aşağıdan Yukarıya Ayrıştırma**

Ayrıştırma ağacının yapısı yapraklardan başlar ve köke doğru ilerler. Etkin aşağıdan yukarıya ayrıştırıcılar bir yazılım aracı ile oluşturulabilirler.

|  |
| --- |
| Resim 2: Lexer ve parser işleyişi |
| **Resim 2:** Lexer ve parser işleyişi |

**Anlamsal Analiz (Semantic Analysis)**

Parse ağacı oluştuktan sonra bu ağaç, semantik analize (semantic analysis) tabi tutulur. Ağacın her yaprağında bulunan tokenler, bu aşamada derleyici için bir anlam ifade etmeye başlar. Örneğin; derleyici tarafından integer olan bir tokene integer, değişken olan tokene ise değişken olarak anlam verilir. Bunun sonucunda anlam olarak belli hatalar oluşabilir. Bu aşamada oluşabilecek hatalara "semantik hatalar (semantic errors)" denir. Örneğin; integer olan bir değişken, string olan bir değişkene direkt olarak eşitlenemez çünkü bunların token tipleri farklıdır.

Bu aşamada, kaynak programdaki anlamsal hatalar kontrol edilir ve kod üretimi için veri tipi bilgileri belirlenir. Tip kontrolü anlamsal analizin en önemli kısmıdır. Anlamsal bilgi, bağlamdan bağımsız dil ile gösterilmez. Söz dizimsel analizde kullanılan CFG, anlamsal kurallar ile birleştirilir.

**Örnek:**

            newval := oldval + 12

newval verisinin tipi (oldval+12) ifadesinin tipi ile uyum göstermektedir.

**İngilizcede Anlamsal Analiz**

**Örnek:**

*Jack said Jerry left his assignment at home.*

Burada "his" kimin için kullanılmıştır? Jack mi, Jerry mi?

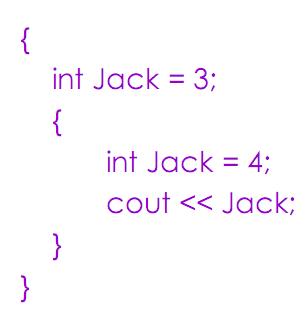
**Daha kötüsü:**

*Jack said Jack left his assignment at home.*

Bu durumda kaç tane Jack vardır? Hangisi ödevini evde bırakmıştır?

**Programlamada Anlamsal Analiz**

Programlama dilleri bu tür belirsizlikleri önlemek için katı kurallar tanımlarlar.



Bu C++ kodunun çıktısı "4" olur; en içteki tanımlama kullanılmıştır.

Derleyiciler, değişken bağlamanın (variable bindings) yanında pek çok anlamsal kontrol de yaparlar.

**Örnek:**

Jack left her homework at home.

**Tip uyuşmazlığı (type mismatch):** Her ve Jack arasında; bu iki sözcük farklı kişilerden bahsediyor olmalı, Jack muhtemelen erkek olmalı.

**Orta Düzey Kod Üretimi**

Semantik analiz sonucunda herhangi bir hata oluşmamış ise parse ağacı yardımı ile "intermediate code" ya da "intermediate representation" denilen ara bir kod oluşturulur. Bu ara kod oluşturulduktan sonra kodlar optimize edilir ve tekrar son çıktı için kod üretilir. Bu safha her derleyicide bulunmayabilir. Bazı derleyiciler ara kod oluşturmadan direkt olarak kod oluşturabilir.

Derleyiciler kaynak programı temsil eden açık orta düzey kod üretebilirler. Bu orta düzey kodlar genellikle bilgisayar mimarisinden bağımsızdır. Ancak bu kodların seviyesi makine kodu seviyesine yakındır.

**Örnek:**

newval := oldval \* fact + 1

id1 := id2 \* id3 + 1

MULT   id2, id3, temp1

ADD     temp1, #1, temp2

MOV     temp2, id1

**Kod Optimizasyonu**

İngilizce için tam karşılığı olmamakla birlikte redaksiyona (editing) benzemektedir.

Programları otomatik olarak değiştirerek

* - Daha hızlı çalışmasını
* - Daha az bellek kaplamasını
* - Genel olarak kaynak kullanımında tutumlu davranmayı sağlamaktadır.

**Örnek:**

MULT   id2, id3, temp1

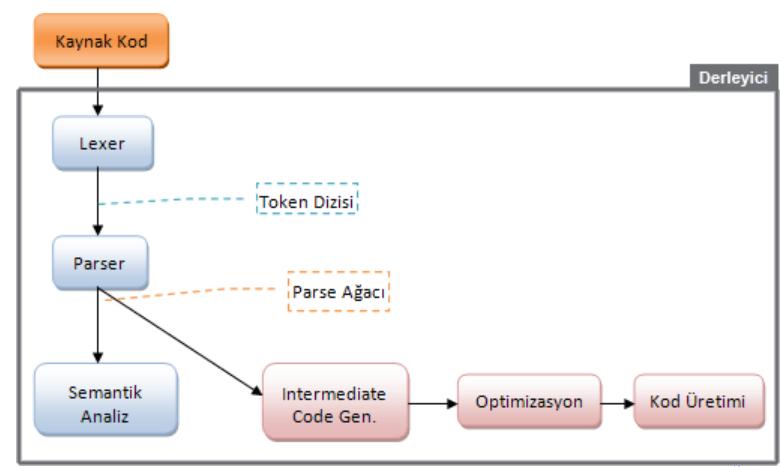
ADD     temp1, #1, id1

**Kod Üretimi**

İstenen mimaride hedef dili üretebilir. Genelde assembly kodu üretir. Bu daha sonra assembler

(çevirici, toplayıcı) tarafından çalıştırılabilir koda çevrilir.

Başka bir dile çeviri şeklinde de olabilir. Bu, insanların yaptığı çeviriyle özdeşlelştirilebilir.



**Resim 3:** Derleyici işleyiş diyagramı

Bu işlemlerden bizi en çok zorlayan kısım lexer ve parser kısımlarıdır çünkü bu parçalar, programlama dilinin gramerini ifade eden, tanımlanmasında hata kabul etmeyen, iş yükü çok olan parçalardır. Bu zorluklardan dolayı birçok "derleyici derleyicisi" denilen yardımcı programlar geliştirilmiştir.

Optimizasyonu her zaman kullanmak gerekli değildir. Ancak:

* Bilimsel programlarda
* İleri işlemcilerde (DSP - Digital Signal Processors)
* Küçük cihazlarda (hız = daha uzun pil ömrü) gerekebilir.

Kodun güvenilirliğini artırmak için derlemede ele alınan konular:

* Bellek güvenliği
* Uyumluluk sorunlarını (concurrency errors) tespit etmek.

Derleyici tasarımında kullanan teknikler, bir derleyicinin geliştirilmesine ek olarak, bilgisayar biliminde birçok probleme de uygulanabilir.

* Sözcüksel analiz için kullanılan teknikler, metin düzenleyici, bilgi elde etme (information retrieval), örüntü tanıma programları için de kullanılabilir.
* Ayrıştırmada kullanılan teknikler SQL gibi sorgu işleme sistemlerinde kullanılabilir.
* Derleyici tasarımında kullanılan tekniklerin birçoğu Doğal Dil İşleme (Natural Language Processing) sistemlerinde kullanılabilir.

Günümüzde derleyicilere daha çok ihtiyaç duyulmakta ve daha karmaşıklaşmaktadırlar. Yeni diller ve yeni mimariler arasındaki farklılıklara bağımlı olarak gelişmektedirler. Önemle çalışılan ve sağlıklı bir gelişim sergileyen, geliştirilen bir alandır.

EBNF' ye göre tanımlanmış atamalara izin veren basit bir programlama dili:

(\* a simple program in EBNF − Wikipedia \*)

program ::= 'PROGRAM' , white space , identifier , white space ,

'BEGIN' , white space ,

{ assignment , ";" , white space } ,

'END.' ;

identifier = alphabetic character , { alphabetic character | digit } ;

number ::= [ "-" ] , digit , { digit } ;

string ::= '"' , { all characters − '"' } , '"' ;

assignment ::= identifier , ":=" , ( number | identifier | string ) ;

alphabetic character ::= "A" | "B" | "C" | "D" | "E" | "F" | "G"

| "H" | "I" | "J" | "K" | "L" | "M" | "N"

| "O" | "P" | "Q" | "R" | "S" | "T" | "U"

| "V" | "W" | "X" | "Y" | "Z" ;

digit ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;

white space ::= ? white space characters ? ;

all characters ::= ? all visible characters ? ;

Yukarıdaki kurala göre söz dizimi doğru olan bir program:

PROGRAM DEMO1

BEGIN

A0:=3;

B:=45;

H:=-100023;

C:=A;

D123:=B34A;

BABOON:=GIRAFFE;

TEXT:="Hello world!";

END.

Bu dil kontrol akışları, matematiksel ifadeler ve G/Ç komutlarıyla kolayca genişletilebilir, küçük ve kullanılabilir bir programlama dili haline getirilebilir.

Standartta normal gösterim olarak önerilen şu işaretler kullanıldı:

|  |  |
| --- | --- |
| **Kullanım** | **Notasyon** |
| definition | **=** |
| concatenation | **,** |
| termination | **;** |
| separation | **|** |
| option | **[ ... ]** |
| repetition | **{ ... }** |
| grouping | **( ... )** |
| double quotation marks | **" ... "** |
| single quotation marks | **' ... '** |
| comment | **(\* ... \*)** |
| special sequence | **? ... ?** |
| exception | **-** |

As an example, consider this possible BNF for a [U.S.](http://en.wikipedia.org/wiki/United_States) [postal address](http://en.wikipedia.org/wiki/Address_(geography)):

**<**postal-address**>** **::=** **<**name-part**>** **<**street-address**>** **<**zip-part**>**

**<**name-part**>** **::=** **<**personal-part**>** **<**last-name**>** **<**opt-suffix-part**>** **<**EOL**>**

**|** **<**personal-part**>** **<**name-part**>**

**<**personal-part**>** **::=** **<**first-name**>** **|** **<**initial**>** "."

**<**street-address**>** **::=** **<**house-num**>** **<**street-name**>** **<**opt-apt-num**>** **<**EOL**>**

**<**zip-part**>** **::=** **<**town-name**>** "," **<**state-code**>** **<**ZIP-code**>** **<**EOL**>**

**<**opt-suffix-part**>** **::=** "Sr." **|** "Jr." **|** **<**roman-numeral**>** **|** ""

**<**opt-apt-num**>** **::=** **<**apt-num**>** **|** ""

This translates into English as:

* A postal address consists of a name-part, followed by a [street-address](http://en.wikipedia.org/wiki/Street_name) part, followed by a [zip-code](http://en.wikipedia.org/wiki/ZIP_Code) part.
* A name-part consists of either: a personal-part followed by a [last name](http://en.wikipedia.org/wiki/Last_name) followed by an optional [suffix](http://en.wikipedia.org/wiki/Suffix_(name)) (Jr., Sr., or dynastic number) and [end-of-line](http://en.wikipedia.org/wiki/End-of-line), or a personal part followed by a name part (this rule illustrates the use of [recursion](http://en.wikipedia.org/wiki/Recursion_(computer_science)) in BNFs, covering the case of people who use multiple first and middle names and/or initials).
* A personal-part consists of either a [first name](http://en.wikipedia.org/wiki/First_name) or an [initial](http://en.wikipedia.org/wiki/Initial) followed by a dot.
* A street address consists of a house number, followed by a street name, followed by an optional [apartment](http://en.wikipedia.org/wiki/Apartment) specifier, followed by an end-of-line.
* A zip-part consists of a [town](http://en.wikipedia.org/wiki/Town)-name, followed by a comma, followed by a [state code](http://en.wikipedia.org/wiki/U.S._postal_abbreviations), followed by a ZIP-code followed by an end-of-line.
* A opt-suffix-part consists of a suffix, such as "Sr.", "Jr." or a [roman-numeral](http://en.wikipedia.org/wiki/Roman_numerals), or an empty string (i.e. nothing).
* A opt-apt-num consists of an apartment number or an empty string (i.e. nothing).

Note that many things (such as the format of a first-name, apartment specifier, ZIP-code, and Roman numeral) are left unspecified here. If necessary, they may be described using additional BNF rules.

Anlamsallar(Semantics)

Anlamsalları tanımlamakiçin geçiş çapta

kabul edilen tek bir notasyon yoktur.

•

İşlemsel Anlamlar(Operational Semantics)

–

Bir programın ifadelerini bir makinede

çalıştırarak anlamlarını tanımlar; ya simüle eder

ya da gerçekten çaıştırır. Makinenin

durumundaki(bellek-memory, yazmaçlarregisters) değişiklik ifadenin anlamını tanımlar.

İşlemsel Anlamlar(Operational Semantics)

Bir yüksek seviye dil için işlemsel anlamarı

kullanmak için bir sanal makineye ihtiyaç vardır.

•

Bir donanımsal saf yorumlayacı çok pahalı

olacaktır.

•

Bir yazılımsal saf yorumlayıcının da bazı

problemleri vardır

–

Belirli bilgisayarın detaylı karakteristikleri

eylemleri anlamayı zorlaştıracaktır.

–

Bu tip bir anlamsal tanımlama makine-bağımlı

olacaktır.

Aksiyomatik Anlamlar(Axiomatic Semantics)

biçimsel mantık temellidir.

•

Orijinal amaç: biçimsel program doğrulama

•

Aksiyonlar veya girişim(inference) kuralları

dildeki her ifade için tanımlanır(bu

deyimlerden başka deyimlere

dönüştürmeye izin verir)

•

Deyimler iddialar(assertions) olarak

adlandırılır.

Bir ifadeden önceki iddia (a precondition)

değişkenler arasındaki ilişkileri ve kısıtları

tanımlar

•

Bir ifadeden sonraki iddia postcondition dır.

•

En zayıf ön koşul (weakest precondition)en

az kısıtlayıcı önkoşuldur ve bu koşul post

condition ı garanti eder.

Pre-, post form: {P} statement {Q}

•

Bir örnek

–

a = b + 1 {a > 1}

–

Bir mümkün önkoşul(precondition): {b > 10}

–

En zayıf önkoşul: {b > 0}

# Axiomatic Semantics

***Axiomatic semantics*** *was defined in conjunction with the development of a method to prove the correctness of programs.*

1. Compute the weakest precondition for each of the following assignment statements and postconditions: (*Sebesta, 7th ed., Problem Set 3.19*)
2. a = 3 \* (b - 1) + 2 {a>0}
3. p = (q + 9) / 4 {p > 7}

***Solution***

Weakest precondition is: What one can be sure about; at least..

Note that these are not mathematical inequalities. “=” stands for assignment. Not “equal to”. Assignment is uni-directional.

a) a = 3 \* (b - 1) + 2 {a>0}

3\*(b - 1)+2 > 0

3\*b -3 +2 > 0

3\*b>1

b> 1/3

1. p = (q + 9) / 4 {p > 7}

(q + 9) / 4 > 7

q + 9 > 28

q > 19

1. Compute the weakest precondition for each of the following sequences of assignment statements and their postconditions:
2. a = 3 \* b + 8;

b = a +13;

{b<1}

1. a = 2 \* ( b + a);

b = 2 \* a + 12;

{b > 24}

***Solution***

a) Starting from the most recent assignment

b = a +13; {b<1}

a + 13 < 1

a < -12

Now we have: a = 3 \* b + 8; {a< -12}

3\*b+8<-12

3\*b<-20

b < -20/3

b) Starting from the most recent assignment

b = 2 \* a + 12; {b > 24}

2\*a+12 >24

2\*a>12

a>6

Now we have: a = 2 \* ( b + a); {a>6}

2 \* ( b + a)>6

b+a>3

We can not deduce any further because we have no clue about b’s value before the second assignment.

# PL Lab – Week 3

## *Exercises*

### Parse Trees and Derivations

**1. <assign>→<id>:=<expr>**

**<id>→A|B|C**

**<expr>→<id>+<expr>**

**|<id>\*<expr>**

**|(<expr>)**

**|<id>**

**(Show a parse tree and leftmost derivation)**

**a. A:=A\*(B+(C\*A))**

<assign>

<id> **:=** <expr>

<id> **\*** <expr>

A

A

( <expr> )

<id> **+** <expr>

B

( <expr> )

<id> **\*** <expr>

C

<id>

A

<assign>→<id>:=<expr>

A:=<expr>

A:=<id>\*<expr>

A:=A\*<expr>

A:=A\*(<expr>)

A:=A\*(<id>+<expr>)

A:=A\*(B+<expr>)

A:=A\*(B+(<expr>))

A:=A\*(B+(<id>\*<expr>))

A:=A\*(B+(C\*<expr>))

A:=A\*(B+(C\*<id>))

A:=A\*(B+(C\*A))

**b. B:=C\*(A\*C+B)**

<assign>

<id> **:=** <expr>

<id> **\*** <expr>

B

C

( <expr> )

<id> **\*** <expr>

<id> **+** <expr>

A

<id>

C

B

<assign>🠚<id>:=<expr>

B:=<expr>

B:=<id>\*<expr>

B:=C\*<expr>

B:=C\*(<expr>)

B:=C\*(<id>\*<expr>)

B:=C\*(A\*<expr>)

B:= C\*(A\*<id>+<expr>)

B:= C\*(A\*C+<expr>)

B:= C\*(A\*C+<id>)

B:= C\*(A\*C+B)

**2. <assign>→<id>:=<expr>**

**<id>→A|B|C**

**<expr>→<expr>+<term>|<term>**

**<term>→<term>\*<factor>|<factor>**

**<factor>→(<expr>)|<id>**

**(Show a parse tree and leftmost derivation)**

**a. A:=(A+B)\*C**

<assign>

<id> **:=** <expr>

A

<term>

<term> **\*** <factor>

<id>

C

<factor>

( <expr> )

<expr> **+** <term>

<term>

<factor>

<id>

A

<id>

B

<factor>

<assign>🠚<id>:=<expr>

A:=<expr>

A:=<term>

A:=<term>\*<factor>

A:=<factor>\*<factor>

A:=(<expr>)\*<factor>

A:=(<expr>+<term>)\*<factor>

A:=(<term>+<term>)\*<factor>

A:=(<factor>+<term>)\*<factor>

A:=(<id>+<term>)\*<factor>

A:=(A+<term>)\*<factor>

A:=(A+<factor>)\*<factor>

A:=(A+<id>)\*<factor>

A:=(A+B)\*<factor>

A:=(A+B)\*<id>

A:=(A+B)\*C

**BNF-EBNF**

**1.**

**<S>🠚<A>a<B>b**

**<A>🠚<A>b|b**

**<B>🠚a<B>|a**

***(Çözüm için Not:***

***1-Mutlaka b ile başlayacak.***

***2-Mutlaka b ile bitecek.***

***3-İkiden fazla a olmalı.)***

**Which of the following are generated by the above grammar?**

a. baab ✓

b. bbbab🗶

c. bbaaaaa🗶

d. bbaab✓

**2. Give a BNF definition of variable names which begin with alphabetic characters.**

<alphabetic> → A|B|C..|Z|a|b|c|..|z

<numeric> → 0|1|2|3|4|5|6|7|8|9

<any\_char> → <alphabetic>|<numeric>

<name> → <name><any\_char>|<alphabetic>

**3. Write a grammar for the language consisting of strings that have n copies of the letter a followed by the same number of copies of the letter b, where n>0.**

<final>🠚ab|a<final>b

**4. The XL language which consists of all strings that begin with a single ‘X’ followed by one or more ‘X’s, ended by a single ‘L’.**

<my\_gram> → X<letterXs>L

<letterXs> → <letter>|<letter><letterXs>

<letter> → X

### BNF-EBNF Conversions

**3.** Give a BNF definition of Java method signatures; i.e., the method declarations that can be given in an interface, for example:

**public void** myMethod(**int** i);

String concatenate(String s1, String s2);

You may assume that all parameters either have type int or String, and that can return types are either int, String or void.

<prototype> → <encaps><blanks><type><blanks><name>(<args>);

<encaps> → public|private|protected|<null>

<type> → int|String|void

<arg\_type> → int|String

<args> → <arglist>|<null>

<arglist> → <arglist>,<arg>|<arg>

<arg> → <arg\_type><blanks><name>

<blanks> → *b*| *b*<blanks>

Note: <name> was already defined in question 2.

**4. Simplify** the BNF grammar you have defined for the question above, into EBNF form.

**Solution 1:**

Instead of converting from the BNF definition, it is more feasible to write it directly from the question.

<prototype> → [<encaps>] *b*{*b*}<type> *b*{*b*}<name>([<arglist>]);

<encaps> → public|private|protected

<type> → int|String|void

<arglist> → <arg>{,<arg>}

<arg> → (int|string) *b*{*b*}<name>

**Solution 2:**

It is also possible to write the expression on one line.

<prototype>→

[(public|private|protected)] *b*{*b*}(int|String|void) *b*{*b*}<name>

([(int|String) *b*{*b*} <name>{, (int|String) *b*{*b*}<name>}]);

Note: To separate the literal ‘)’ from the EBNF symbol ‘)’ the literal usage is underlined.

**5.** Translate the following grammar expressed in EBNF into BNF.

**<goal> → {ab[<bar>]}[<foo>](<bar>|<baz>)**

**<bar> → b[e]f**

**<foo> → d{de}**

**<baz> → g{g}**

<goal> → <non\_term\_1><non\_term\_2><non\_term\_3>

<non\_term\_1> → <null>|<non\_term\_1><in\_loop>

<in\_loop> → ab|ab<bar>

<bar> → bef|bf

<non\_term\_2> → <null>|<foo>

<foo> → <foo>de|d

<non\_term\_3> → <bar>|<baz>

<baz> → g|g<baz>