Gramer Tabanlı Değerlendirme Yöntemleri

1. Giriş

Bir gramer biçimsel bir dilde karakter dizileri yada kelimeler oluşturabilen kurallar seti olarak ifade edilir. Kurallar, dil alfabesini kullanarak, dilin sözdizimi ile uyumlu kelimelerin nasıl üretileceğini belirtir. Gramerlerin birçok türü vardır; bağlamdan bağımsız gramerler, düzenli gramerler, analitik gramerler, vs. Bu föyde bağlamdan bağımsız gramerler tarafından üretilebilen veriler için ayrıştırıcı (parser) ve değerlendirici (evaluator) geliştirme aşamaları bir uygulama üzerinden anlatılmıştır.

2. Bağlamdan Bağımsız Gramerler ve Analiz Aşamaları

Bir bağlamdan bağımsız gramer (CFG, context-free grammar) içiçe kelime dizeleri üretebilen bir biçimsel dili temsil eder. Genellikle BNF (Backus Normal Form) notasyonu ile tanımlanan bu gramerin her bir kuralı `X → w` biçimindedir; `X` bir nonterminal simgesine, `w` ise terminal (bir kelime) ve/veya nonterminal dizilerine karşılık gelir (`w` boş olabilir). BNF notasyonunda nonterminal ve terminal'lerin çok farklı gösterimleri olmakla birlikte, aşağıdaki gramer tanımlamalarında bir nonterminal tek karakterli büyük harf kullanarak, bir terminal ise çift tırnak içerisinde verilmiştir. Tablo 1(a)'da `x;y;z` gibi kelime dizelerini üreten bir gramer örneği gösterilmiştir.

Tablo 1: `x;y;z` kelime dizelerini üreten bir gramer ve javaCC bildirimi

(a) Gramer	(b) LL(1) grameri	(c) javaCC bildirimi

`S → S ";" S`
`S → "x"`
`S → "y"`
`S → "z"`
	`S → T ( ";" S )?`
	`T → "x" | "y" | "z"`
		`void S() : { }`
		`{ T() (";" S())? }`
		`void T() : { }`
		`{ "x" | "y" | "z" }`

Gramerler BNF notasyonuna meta karakterler eklenerek geliştirilen EBNF notasyonunda da tanımlanabilir. Tablo 2'de bazı meta karakterler ve anlamları listelenmiştir.

Tablo 2: EBNF meta karakterleri ve anlamları

Simge	Anlamı	Örnek	Veriler
`*`	Veriyi herhangi sayıda tekrarla	`a*`	`{Ø,a,aa,aaa,...}`
`+`	Veriyi en az bir defa tekrarla	`b+`	`{b,bb,bbb,...}`
`|`	Veri alternatifleri oluştur	`a|b`	`{a,b}`
`?`	Veriyi en fazla bir defa tekrarla	`c?`	`{Ø,c}`
`()`	Veriyi grupla	`(a|b)`	`{a,b}`
`[]`	Bir veri aralığı oluştur	`[a-c]`	`{a,b,c}`

Bir gramer için ayrıştırıcı geliştirmede Java dilinde kaynak kod üretebilen `javaCC` aracı kullanılabilir [1, 2]. Bu araç sadece LL(k) gramerleri üzerinde çalışır. LL(k) gramerlerinde girdi verisinden en fazla

k terminal okuyarak kural seçimi yapılabilir (k>0). Tablo 1(a)'daki gramer bir LL(1) gramerine dönüştürülerek Tablo 1(b)'de gösterilmiştir. Bir LL(1) grameri aşağıdaki özelliklere sahiptir.

- Aynı nonterminal ile başlayan kurallar farklı terminaller üretir.
- Bir kural ile bu kuralın hemen ardından çağrılan diğer bir kural farklı terminaller üretir.
- Kurallar soldan rekursif değildir.

LL(k) gramerleri `.jj` uzantılı dosyalar içerisinde tanımlanarak `javaCC` aracına bildirilirler. Bu dosyada seçenek bildirimi, ayrıştırıcı sınıfının bildirimi, terminal ve gramer kurallarının bildirimi olmak üzere üç bildirim bölümü vardır. Tablo 1(c)'de sadece kural bildirim bölümü gösterilmiştir. Terminal'ler < ve > simgeleri arasında belirtilirken, kurallar programlama dili fonksiyon yada metotlarına benzer yapıda bildirilir. Burada metotlara ilgili kural nonterminali ile uyumlu isimler verilir. nonterminal yada başka bir isimli metot biçimde bildirilir.

```
void nonTerminal() :
  { /* Java bildirimleri */ }
  { /* Kural tanımlaması */ }
```

Aşağıdaki bölümlerde matematiksel ifadeleri ayrıştıran ve değerlendiren bir uygulama geliştirilmiştir. Bu uygulamayı kodlama süreci gramerin yazılması, token'ların bildirilmesi, ayrıştırıcının tanımlanması, sözdizim sınıflarının oluşturulması, sözdizim ağacının üretilmesi ve ifadenin ağaç üzerinden değerlendirilmesi gibi 6 aşamadan meydana gelir.

3. Gramer Yazımı

Aritmetik ifadeler birçok işleç ve matematiksel fonksiyon kullanılarak yazılabilir. Tablo 2'de 5 farklı işleç ve 5 farklı fonksiyon içerebilen aritmetik ifadeler için bir EBNF grameri verilmiştir. Aritmetik ifadeyi meydana getiren parantez, işleç ve fonksiyonlar çift tırnak içerisinde gösterilmiştir. Gramer kurallarının karşısında parantez içerisinde verilen sınıf isimleri işleç, fonksiyon ve bunların uygulanacağı verileri nesneye dayalı programlama yapılarıyla temsil etmede kullanılacaktır. Bu sınıflar aşağıdaki bölümlerde sözdizim sınıfları olarak adlandırılmıştır.

Tablo 3: Aritmetik ifadeler için bir EBNF grameri

E → E "+" E	(Plus)	E → "exp" "(" E ")"	(Euler)
E → E "-" E	(Minus)	E → "log" "(" E ")"	(Log)
E → E "*" E	(Times)	E → "sqrt" "(" E ")"	(Sqrt)
E → E "/" E	(Divide)	E → "sin" "(" E ")"	(Sin)
E → E "^" E	(Power)	E → "max" "(" E "," E ")"	(Max)
E → "x"	(Var)	E → (D)+ ("." (D)+)?	(Num)
E → "+" E		E → "(" E ")"	
E → "-" E		D → ["0"-"9"]	

Tablo 3'de verilen gramerde işleçlerin değerlendirme sırasını düzenleyen özellikleri (öncelik düzeyleri ve birleşme yönleri) göz önüne alınmamıştır. Bir işlecin öncelik düzeyi kendisini üreten gramer kuralının konumuna, birleşme yönü ise bu kuralın rekursif olduğu yöne bağlıdır. Aynı nonterminal ile başlayan kuralların konumlarının da aynı olduğu kabul edilir. Konum farklılığı için bir kuralın diğer bir kural içerisinden çağrılabilir olması gerekir. Tablo 3'deki gramer tanımlamasına göre bütün işleçler aynı özelliklere sahip olacaktır. Ancak aritmetik işlemlerde bu işleçler için çok farklı öncelik düzeyleri ve birleşme yönleri kullanılır. Bu özellikler işleç öncelik sırasına göre en yüksekten başlanarak Tablo 4'de özetlenmiştir.

Tablo 4: İşleç öncelik ve birleşme yönleri

İşleçler	Anlamı	Birleşme yönü
^	Üs alma	Sağdan sola
+, -	Artı, eksi işareti	Sağdan sola
*, /	Çarpma, bölme	Soldan sağa
+, -	Toplama, çıkarma	Soldan sağa

Tablo 4'de görüldüğü gibi işleçlerin 4 farklı öncelik düzeyi ve 2 farklı birleşme yönü vardır. Bu nedenle, Tablo 3'deki gramer kurallarının konum farklılığı ve doğru rekursiflik yönü oluşturulacak biçimde yeniden düzenlenmesi gerekir. Ayrıca aritmetik ifadeler javaCC ile üretilen ayrıştırıcılar tarafından analiz edileceğinden yeni gramer LL(k) karakterli bir gramer olmalıdır. Böyle bir LL(1) karakterli gramer daha fazla sayıda kural ile Tablo 5'teki gibi tanımlanabilir (`$` simgesi girdi verisinin sonunu işaretler). Tablo 5 aynı zamanda bu gramer için kodlanacak bir ayrıştırıcıda bulunması gereken metotların isimlerini de içermektedir.

Tablo 5: Aritmetik ifadeler için işleç anlamlarını da içeren bir LL(1) grameri

Kural	Metot
`S → E $`	`parse()`

`E → T E'`
`E' → ("+" | "-") T E'`
`E' →`
	`expr()`

`T → F T'`
`T' → ("*" | "/") F T'`
`T' →`
	`term()`

`F → ("+" | "-")? P`	`unary()`
`P → R ("^" P)?`	`power()`
`R → "x"`	`element()`
`R → "exp" "(" E ")"`	
`R → "log" "(" E ")"`	
`R → "sqrt" "(" E ")"`	
`R → "sin" "(" E ")"`	
`R → "max" "(" E "," E ")"`	
`R → (D)+ ("." (D)+)?`	
`R → "(" E ")"`	
`D → ["0"-"9"]`	

4. Token Üreteçleri

Bir token üreteci (scanner) girdi verisini kelimesel yönden analiz ederek bir token dizisi üretir. Bir token girdi verisi içinde bulunan ve daha küçük parçalara ayrılamayan her bir kelimeyi (terminal) temsil eder. Bir karakterli olabileceği gibi birkaç karakterden de meydana gelebilir. Örneğin, matematiksel ifadeler içinde yer alabilen `"+"`, `"^"` ve `"max"` kelimelerin için farklı token'lar tanımlanır. Diğer yandan, bir veri türünün değişik örneklerine karşılık gelen birçok farklı karakter dizisini (yada kelimeyi) temsil edebilmek için aynı token kullanılır. Örneğin, bütün sayılar (`"5"` ve `"0.01"` gibi) `NUMBER` token'ı ile temsil edilir. Bölüm 2'de geliştirilen LL(1) gramerinin içerdiği kelimeleri temsil edecek token'lar javaCC'de Tablo 6'daki gibi tanımlanabilir. Burada token üretecine ait bir tanımlama bloğu olmadığına dikkat ediniz.

Tablo 6: Gramer terminal'leri için javaCC token bildirimleri

```
//EvalParse.jj
SKIP : { " " | "\t" | "\r" }

TOKEN : { < NUMBER : (["0"-"9"])+
            ("." (["0"-"9"])+)? > }
TOKEN : { < EOL : "\n" > }

TOKEN : /* OPERATORS */
{
    < PLUS: "+" >
  | < MINUS: "-" >
  | < TIMES: "*" >
  | < DIVIDE: "/" >
```

```
(Devamı)
  | < POWER: "^" >
  | < EXP: "exp" >
  | < LOG: "log" >
  | < SQRT: "sqrt" >
  | < SIN: "sin" >
  | < MAX: "max" >
  | < X: "x" >
  | < COM: "," >
  | < LPR: "(" >
  | < RPR: ")" >
}
```

Bu tanımlamalar ışığında "`(x+1)*2-x^sin(x)`" ifadesi için aşağıdaki token dizisi üretilir.

```
LPR X PLUS NUMBER RPR TIMES NUMBER
    MINUS X POWER SIN LPR X RPR
```

5. Ayrıştırıcılar

Bir ayrıştırıcı (parser) girdi verilerinin gramer kuralları ile üretilebilirliğini token dizisi üzerinden analiz eder. Böyle bir analizde girdi verisi sadece doğru sözdizimli olup olmadığı yönünden incelenir, verinin değerlendirilebilirliği (yada yorumlanabilirliği) konusu aydınlatılmaz. Bu durum sözdizimsel olarak doğru bir girdi verisinin değerlendirilmeye başlanmadan önce anlamsal analizden geçirilmesini gerektirir. Ancak, matematiksel ifadelerden oluşan girdi verileri için yapılan sözdizim analizi aynı zamanda verinin değerlendirilebilir olmasını garanti edecektir. Bölüm 2'de geliştirilen LL(1) gramerine göre sözdizim analizi yapan bir ayrıştırıcının javaCC tanımlaması Tablo 7'de verilmiştir. <EOL> ve <EOF> token'ları sırasıyla satır ve girdi sonu karakterlerini temsil eder.

Tablo 7: Gramer kurallarının javaCC tanımlaması

```
//EvalParse.jj
void parse() : { }
{
  expr() (<EOF> | <EOL>)
}

void expr() : { }
{
  term()( <PLUS> term()
        | <MINUS> term()
  )*
}

void term() : { }
{
  unary() ( <TIMES> unary()
          | <DIVIDE> unary()
  )*
}

void unary() : { }
{
    <PLUS> power()
```

```
(Devamı)
  | <MINUS> power()
  | power()
}

void power() : { }
{
  element() ( <POWER> power() )?
}

void element() : { }
{
    <NUMBER>
  | <X>
  | <LPR>  expr() <RPR>
  | <EXP>  <LPR>  expr() <RPR>
  | <LOG>  <LPR>  expr() <RPR>
  | <SQRT> <LPR>  expr() <RPR>
  | <SIN>  <LPR>  expr() <RPR>
  | <MAX>  <LPR>  expr() <COM>
                  expr() <RPR>
}
```

6. Sözdizim Sınıfları

Sözdizim sınıfları gramer kurallarıyla üretilebilen girdi verilerini nesneye dayalı programlama yapılarıyla temsil etmek için tanımlanır. Bir işleçli aritmetik ifade üreten gramer kuralını temsil edecek sözdizim sınıfının tanımlamasına işlecin uygulanacağı veriler de eklenir. Tablo 8 bazı sözdizim sınıflarının tanımlamasını gösterir.

Tablo 8: Gramer sözdizim sınıflarının tanımlaması

```
//AST.java
abstract class Exp { }

class Plus extends Exp {
  public Exp exp1, exp2;

  public Plus(Exp e1, Exp e2) {
    exp1 = e1; exp2 = e2;
  }
}
...

class Euler extends Exp {
  public Exp exp;

  public Euler(Exp e) {
    exp = e;
  }
}
```

(Devamı)

```
...

class Num extends Exp
{
  public double num;

  public Num(double n) {
    num = n;
  }
}

class Var extends Exp
{
  public Var() { }
}
```

7. Sözdizim (Nesne) Ağacı

Bir sözdizim ağacı (yada genel olarak nesne ağacı) hiyerarşik yapıda birbirine bağlanmış birçok düğümden meydana gelir. Her bir düğüm sözdizim sınıflarından türetilmiş ve bir işlemi yada veriyi temsil eden bir nesne içerir. Sözdizim ağaçlarının bildirimi genellikle bir üst sınıf türü yardımıyla yapılır. Aynı üst sınıftan miras alan sözdizim sınıfları bir nesne ağacının düğümlerini oluşturmada kullanılabilir, ancak bu düğümler ağaç üzerinde üst sınıf türüyle temsil edilir.

JavaCC tanımlamalarına çeşitli Java ifadeleri ekleyerek bir nesne ağacını üretmek mümkündür. Tablo 9'da bir matematiksel ifadeyi temsil edecek nesne ağacını üretmek için, Bölüm 4'teki gramer tanımlamasına hangi Java ifadelerinin ekleneceği gösterilmiştir.

Tablo 9: Gramer kural tanımlamalarına sözdizim ağacı üreten ifadelerin eklenmesi

```
//EvapParse.jj
Exp parse() : { Exp a; }
{
  a = expr() (<EOF> | <EOL>)
    { return a; }
}

Exp expr() : { Exp a, b; }
{
  a = term() (
     <PLUS>  b = term()
      { a = new Plus(a, b); }
   | <MINUS> b=term()
```

(Devamı)

```
Exp element() : {
  Token t;
  Exp a, b;
}
{
   t = <NUMBER>
      { return (new Num(Double.
          parseDouble(t.image))); }
  | <X>
      { return (new Var()); }
  | <LPR> a = expr() <RPR>
      { return a; }
```

```
      { a = new Minus(a, b); }
  )*
  { return a; }
}
...

Exp power() : { Exp a, b; }
{
  a = element() (
     <POWER> b = power()
      { a = new Power(a, b); }
  )?
  { return a; }
}
```

```
  | <EXP>  <LPR> a = expr() <RPR>
      { return (new Euler(a)); }
  | <LOG>  <LPR> a = expr() <RPR>
      { return (new Log(a)); }
  | <SQRT> <LPR> a = expr() <RPR>
      { return (new Sqrt(a)); }
  | <SIN>  <LPR> a = expr() <RPR>
      { return (new Sin(a)); }
  | <MAX>  <LPR> a = expr() <COM>
                 b = expr() <RPR>
      { return (new Max(a, b)); }
}
```

Bu ifadeler "`-x^2+x*sin(log(x)+3)`" girdi verisi için aşağıdaki nesne ağacını üretecektir.

```
Exp exp = new Plus(new Times(new Num(-1), new Power(new Var(),
            new Num (2))), new Times(new Var(), new Sin(
              new Plus(new Log(new Var()), new Num(3)))))
```

8. Değerlendirme Yöntemleri

Bir nesne ağacı en içteki düğümden başlanarak kök düğüme doğru değerlendirilir. Her bir düğüm için yapılacak değerlendirme düğümün içerdiği nesnenin türüne bağlıdır. Nesne türünün belirlenmesi ve temsil edilen işlemin gerçekleştirilmesi üç farklı yöntem ile yapılabilmektedir. Bu yöntemlerin üstünlükleri ve sakıncaları Tablo 10'da özetlenmiştir [1].

Tablo 10: Sözdizim ağacı değerlendirme yöntemlerinin karşılaştırılması

Yöntem	**Nesne Türetme**	**Sınıf Derleme**
intanceof İşleci	Evet	Hayır
Sözdizim Sınıflarına Metot Ekleme	Hayır	Evet
Visitor Tasarım Deseni	Hayır	Hayır

8.1. *instanceof* İşleci

Bu yöntemde bir düğümün içerdiği nesnenin türü *intanceof* işleci ile belirlenir. Nesne türü belirlendikten sonra temsil edilen işlemin gerçekleştirilebilmesi için üst sınıf referans değişkeninden alt sınıf nesnesinin türetilmesi gerekmektedir. Tür türetme (*cast*) yapısı kullanılarak ilgili alt sınıf nesnesi türetilebilir. Tablo 11'de tanımlanan `eval()` metodu nesne ağacını ve değerlendirmenin yapılacağı `x` değerini parametre olarak almaktadır.

Tablo 11: *instanceof* işleci ile değerlendirme

```
// EvalExp.java
public class EvalExp {
  public static void main(String[] args) {
    EvalParse parser = new EvalParse(System.in);
    try {
      // Evaluate f(x) for x = args[0];
      System.out.println(
        eval(parser.parse(), Double.parseDouble(args[0])));
    } catch(ParseException e) {
        e.printStackTrace();
```

```
      }
  }

  public static double eval(Exp exp, double x) {
    if (exp instanceof Plus)
      return eval(((Plus)exp).exp1, x) + eval(((Plus)exp).exp2, x);
    else if (exp instanceof Minus)
      return eval(((Minus)exp).exp1, x) - eval(((Minus)exp).exp2, x);
    ...
    else if (exp instanceof Power)
      return Math.pow(eval(((Power)exp).exp1, x),
        eval(((Power)exp).exp2, x));
    else if (exp instanceof Euler)
      return Math.pow(Math.E, eval(((Euler)exp).exp, x));
    ...
    else if (exp instanceof Max)
      return Math.max(eval(((Max)exp).exp1, x),
        eval(((Max)exp).exp2, x));
    else if (exp instanceof Num)
      return ((Num)exp).num;
    else
      return x;
  }
}
```

8.2. Sözdizim Sınıflarına Metot Ekleme

Bu yöntemde her bir sözdizim sınıfına, sınıfın temsil ettiği işlemi gerçekleştiren bir `eval()` metodu eklenir. Bir nesne ağacı düğümünün değerlendirmesi için düğümün içerdiği nesneye ait `eval()` metodunun çağrılması yeterlidir. Dolayısıyla değerlendirilecek düğüm nesne türünün belirlenmesine gerek yoktur. Tablo 12'de değerlendirmenin yapılacağı `x` değerini parametre olarak alan `eval()` metodu tanımlamaları bazı sözdizim sınıfları için verilmiştir.

Tablo 12: Sözdizim sınıflarına `eval()` metotlarının eklemesi

```
// AST.java
abstract class Exp {
  public abstract double
    eval(double x);
}

class Plus extends Exp {
  public Exp exp1, exp2;

  public Plus(Exp e1, Exp e2) {
    exp1 = e1; exp2 = e2;
  }

  public double eval(double x) {
    return (exp1.eval(x) +
            exp2.eval(x));
  }
}
...

class Euler extends Exp {
```

(Devamı)

```
...

class Num extends Exp
{
  public double num;

  public Num(double n) {
    num = n;
  }

  public double eval(double x) {
    return num;
  }
}

class Var extends Exp
{
  public Var() { }

  public double eval(double x) {
```

```
  public Exp exp;

  public Euler(Exp e) {
    exp = e;
  }

  public double eval(double x) {
    return Math.pow(Math.E,
      exp.eval(x));
  }
}
```

```
    return x;
  }
}
```

Tablo 13'de ise değerlendirmeyi kontrol eden ana sınıf gösterilmiştir.

Tablo 13: Metot ekleme ile değerlendirme ana sınıfı

```
// EvalExp.java
public class EvalExp {
  public static void main(String[] args) {
    EvalParse parser = new EvalParse(System.in);
    try {
      // Evaluate f(x) for x = args[0];
   System.out.println(parser.parse().eval(Double.parseDouble(args[0])));
    } catch(ParseException e) {
        e.printStackTrace();
      }
  }
}
```

8.3. Visitor Tasarım Deseni

Visitor tasarım deseni sözdizim sınıflarının yerel verileri ile bu veriler üzerinde tanımlanacak işlemleri biribirinden ayırır. Bir `Visitor` sınıfı tanımlayarak sözdizim ağacının düğümlerine erişir. Bunun için `Visitor` sınıfına her bir sözdizim sınıf türü için bir `visit()` metodu ve her bir sözdizim sınıfına da bir `accept()` metodu eklenmelidir. `visit()` metodu değerlendirilecek düğüm nesnesinin `accept()` metodunu ve ardından `accept()` metodu ise değerlendirmeyi yapacak `visit()` metodunu çağırır. Bu şekilde `visit()` ve `accept()` metotları nesne ağacının bütün düğümlerine erişilinceye kadar birbirlerini çağırmaya devam eder. `Visitor` sınıfı her bir sözdizim sınıfı için bir `visit()` metot bildirimi içeren bir arayüz işlevi görür. `visit()` metotlarının tanımlaması `Visitor` arayüzünü gerçekleyen bir sınıf içerisinde yapılır. Sözdizim sınıf nesnelerinin farklı bir değerlendirmesi diğer bir `Visitor` arayüzü tanımlamasını gerektirir.

Bir nesne ağacının değerlendirmesine Visitor nesnesin ait bir `visit()` metodunun çağrılmasıyla başlanır. `visit()` metodu ilk önce ağacın kök düğümüne erişmeye çalışır ve bu nedenle kök düğümdeki nesnenin `accept()` metodunu mevcut Visitor nesne referansı ile çağırır. `accept()` metodu ise içinde bulunduğu nesne referansını argüman alan diğer bir visit() metodunu çağırarak yeni bir düğüm nesnesinin değerlendirmesini başlatır. Burada `visit()` ve `accept()` metotlarının çağrımı, değerlendirilen düğümdeki nesne türünün bilinmesini gerektirmezler. Tablo 14'te bazı sözdizim sınıfları `accept()` metotları eklenerek gösterilmiştir.

Tablo 14: Sözdizim sınıflarına `accept()` metotlarının eklemesi

```
// AST.java
abstract class Exp {
```

(Devamı)

```
...
```

```
 public abstract double
          accept(Visitor v);
}

class Plus extends Exp {
 public Exp exp1, exp2;

 public Plus(Exp e1, Exp e2) {
   exp1 = e1; exp2 = e2;
 }

 public double accept(Visitor v) {
    return v.visit(this);
  }
}
...

class Euler extends Exp {
 public Exp exp;

 public Euler(Exp e) {
   exp = e;
 }

 public double accept(Visitor v) {
   return v.visit(this);
 }
}
```

```
class Num extends Exp {
 public double num;

 public Num(double n) {
   num = n;
 }

 public double accept(Visitor v) {
   return v.visit(this);
 }
}

class Var extends Exp {
 public Var() { }

 public double accept(Visitor v) {
   return v.visit(this);
 }
}
```

Nesne ağacını değerlendirmek için kullanılacak `Visitor` arayüzü ile bu arayüzü gerçekleyen EvalVisitor sınıfı Tablo 15'te verilmiştir.

Tablo 15: Sözdizim ağacı için bir Visitor arayüzü ve gerçeklemesi

```
// Visitor.java
public interface Visitor {
  public double visit(Exp exp);
  public double visit(Plus exp);
  public double visit(Minus exp);
  public double visit(Times exp);
  public double visit(Divide exp);
  public double visit(Power exp);
  public double visit(Euler exp);
  public double visit(Log exp);
  public double visit(Sqrt exp);
  public double visit(Sin exp);
  public double visit(Max exp);
  public double visit(Num exp);
  public double visit(Var exp);
}

// EvalVisitor.java
public class EvalVisitor
             implements Visitor {
 double x;

 public EvalVisitor(double a) {
```

(Devamı)

```
 public double visit(Exp exp) {
   return exp.accept(this);
 }

 public double visit(Plus exp) {
   double a =
           exp.exp1.accept(this);
    double b =
           exp.exp2.accept(this);
    return (a + b);
 }
  ...

 public double visit(Euler exp) {
   double a = exp.exp.accept(this);
    return Math.pow(Math.E, a);
  }
  ...

 public double visit(Num exp) {
   return (exp.num);
```

```
    x = a;
  }
```

```
  }

  public double visit(Var exp) {
    return x;
  }
}
```

Tablo 16'da ise değerlendirmeyi kontrol eden ana sınıf gösterilmiştir.

Tablo 16: Visitor deseni ile değerlendirme ana sınıfı

```
// EvalExp.java
public class EvalExp {
  public static void main(String[] args) {
    EvalParse parser = new EvalParse(System.in);
    try {
      // Evaluate f(x) for x = args[0];
      System.out.println(new EvalVisitor(
             Double.parseDouble(args[0])).visit(parser.parse()));
    } catch(ParseException e) {
        e.printStackTrace();
      }
  }
}
```

9. Deney Hazırlığı

1. Gramerleri ve EBNF notasyonunda gramer yazımını öğreniniz.
2. LL(k) grameri ve özelliklerini kavrayınız.
3. JavaCC aracı ve gramer tanımlama formatını inceleyiniz.
4. Miras (inheritage) ve çokbiçimlilik (polymorphism) gibi nesneye dayalı programlama kavramlarını gözden geçiriniz.
5. Gramer kuralları için sözdizim sınıfları tanımlamayı ve bu sınıflarla nesne ağacı oluşturmayı öğreniniz.
6. Visitor tasarım desenini ve kullanım alanını araştırınız.

10. Deney Tasarımı ve Uygulaması

1. Deney uygulama dosyaları bilgisayar masaüstünde *gramer_degerlendir.rar* dosyasında bulunmaktadır. `EvalExp`, `EvalExp2` ve `EvalExp3` altdizinlerini içeren bu arşiv dosyasını masaüstünde açın.
2. İlk önce `EvalExp` dizinindeki dosyaları inceleyin.
3. `AST.java` dosyasındaki sözdizim sınıflarının biçimlerini inceleyiniz ve Tablo 3'deki tanımlamalarla karşılaştırınız.
4. `EvalParse.jj` dosyasındaki gramer tanımlamalarını inceleyiniz ve Tablo 5'deki tanımlamalarla karşılaştırınız. İşleç önceliği ve birleşme yönü tanımlamalarının nasıl yapıldığını kavramaya çalışın.
5. `EvalParse.jj` dosyasına nesne ağacı üretim ifadelerinin nasıl eklendiğini anlamaya çalışın.
6. EvalExp.java dosyasında girdi ifadesini okuma, nesne ağacını oluşturma ve ifadenin değerlendirilmesi işlemlerinin nasıl yapıldığını anlamaya çalışın.
7. Uygulamayı çalıştırmak için bir DOS penceresi açın ve çalışma dizinini masaüstünde açılan arşiv içerisindeki `EvalExp` altdizinine değiştirin.

8. `EvalParse.jj` ve `.java` uzantılı dosyaları aşağıdaki gibi derleyin.
```
$> javacc EvalParse.jj
$> javac *.java
```
9. `Expr.txt` dosyasında bulunan matematiksel ifadeyi `x=2.0` değeri için aşağıdaki komut ile değerlendirin.
```
$> java EvalExp 2.0 < Expr.txt
```
10. Benzer matematiksel ifadeler DOS penceresinden aşağıdaki gibi de değerlendirilebilmektedir.
```
$> java EvalExp 3.0 (ENTER'dan sonra aşağıdaki ifadeyi girin.)
x^2+5*sin(x-1)       (ENTER'a basın.)
13.54648713
```
11. `EvalParse.jj` dosyası `ln`, `cos` ve `min` fonksiyonlarına ait herhangi bir tanımlama içermemektedir. Bu fonksiyonlar için gerekli gramer tanımlamalarını `EvalParse.jj` dosyasına ekleyiniz.

12. `AST.java` dosyasına `ln`, `cos` ve `min` fonksiyonlarını temsil edecek sözdizim sınıflarını ekleyiniz.

13. `EvalExp.java` dosyasına `ln`, `cos` ve `min` fonksiyonlarını değerlendirecek Java ifadelerini ekleyin ve uygulamayı matematiksel ifadeler girerek test edin.

14. `EvalParse.jj` dosyasındaki gramer tanımlamasına göre `+|-`, `*|/`, `+|-` ve `^` işleçlerinin öncelikleri ve birleşme yönlerini aşağıda verilen ifade ve diğer ifadelerle test edin.
```
-3^3+2^3*2+50/10/5
```
15. `EvalParse.jj` dosyasındaki `unary()` ile `power()` metotlarının çağrım sırasını değiştirin ve `x=3` için `-x^2` ifadesini değerlendirerek önceki durumdan farkını gözlemleyin.

16. Üs alma (`^`) işlecinin birleşme yönü sağdan sola doğrudur. Bu işlecin birleşme yönü soldan sağa doğru olacak şekilde gramer tanımlamasını değiştirin ve değişikliği `2^3^2` ifadesini değerlendirerek test edin.

17. `EvalExp` sınıfına matematiksel ifadeleri prefix notasyonunda gösterecek biçimde bir `prefix()` metodu ekleyin ve aşağıdaki ifadelerle test edin.
```
x^2-3*x        => - ^ x 2 * 3 x
x+sin(x+1)     => + x sin + x 1
```
18. `EvalExp` sınıfına matematiksel ifadelerin nesne ağacını gösterecek biçimde bir `tree()` metodu ekleyin ve metodu yukarıdaki iki ifade ile test edin.
```
x^2-3*x  => new Minus(new Power(new Var(),new Num(2)),
               new Times(new Num(3),new Var()))
x+sin(x+1) => new Plus(new Var(),
                  new Sin(new Plus(new Var(),new Num(1))))
```
19. Şimdi arşiv içerisindeki diğer altdizinler incelenecektir. Her üç altdizindeki `EvalParse.jj` dosyasının yapısı aynıdır.

20. `EvalExp2` altdizinindeki `AST.java` ve `EvalExp.java` dosyalarını inceleyin. Sözdizim sınıflarına eklenen `eval()` metotlarına dikkat ediniz.

21. Sözdizim sınıflarına girdi ifadesini prefix notasyonuna dönüştüren `prefix()` metotları ekleyin ve değişiklikleri test edin.

22. Sözdizim sınıflarına girdi ifadesine ait nesne ağacını gösteren `tree()` metotları ekleyin ve değişiklikleri test edin.

23. `EvalExp3` içindeki `AST.java`, `Visitor.java`, `EvalVisitor.java` ve `EvalExp.java` dosyalarını inceleyin. Sözdizim sınıflarına eklenen `accept()` metotlarına dikkat ediniz.

24. `EvalVisitor` sınıfındaki `visit()` metotlarını girdi ifadesini prefix notasyonuna dönüştürecek biçimde değiştirin ve değişiklikleri test edin.

25. `EvalVisitor` sınıfındaki `visit()` metotlarını girdi ifadesine ait nesne ağacını gösterecek biçimde değiştirin ve değişiklikleri test edin.

11. Deney Soruları

1. Tablo 5'teki gramere aşağıdaki gibi ardışık atama ifadelerini üretebilecek biçimde yeni kurallar ekleyiniz.
   ```
   y = x^2-3; z = y*3+x/sin(2*x)
   ```
2. Tamsayılar ve x değişkeni üzerinde toplama, mod alma ve faktöriyel işlemlerinin tanımlanabildiği ifadeleri üreten bir grameri EBNF notasyonunda yazınız. İşleçlerin birleşme yönünü soldan sağa doğru, öncelik düzeylerini de +, %, ! sırasında alınız; bu özellikler aşağıdaki örnek üzerinde gösterilmiştir.
   ```
   3+x%5!%2 == 3+((x%(5!))%2)
   ```
3. Bu grameri LL(1) gramerine dönüştürünüz.
4. Her bir kuralı temsil eden sözdizim sınıfını tanımlayınız.
5. Grameri nesne ağacı üreten ifadelerle birlikte JavaCC'de tanımlayınız ve `.jj` dosyasını oluşturunuz.
6. Değerlendirme sınıflarını her üç yöntem için de kodlayınız.

12. Deney Raporu

1. Gramer kuralı ile sözdizim sınıfı ilişkisini kısaca anlatınız.
2. Hangi değerlendirme yönteminin daha kullanışlı olduğunu tartışınız.
3. Matematiksel ifadeleri postfix notasyonunda gösterecek biçimde Visitor desenini bütün arayüz ve metotları ile birlikte tanımlayınız.
4. Deney sorularını cevaplayınız.

13. Kaynaklar

[1] A. W. Appel, Modern Compiler Implementation in Java, Cambridge University Press, 2002.
[2] Java Compiler Compiler – The Java Parser Generator, http://javacc.java.net/, 2012.