KARADENİZ TEKNİK ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI

OTOMATİK KOD ÜRETİM ARAÇLARI YARDIMIYLA MATEMATİKSEL İFADELERİN TÜREVLERİNİN HESAPLANMASI VE SADELEŞTİRİLMESİ

YÜKSEK LİSANS TEZİ

Bil. Müh. Yavuz TEKBAŞ

OCAK 2013

KARADENİZ TEKNİK ÜNİVERSİTESİ

FEN BİLİMLERİ ENSTİTÜSÜ

BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI

OTOMATİK KOD ÜRETİM ARAÇLARI YARDIMIYLA MATEMATİKSEL İFADELERİN TÜREVLERİNİN HESAPLANMASI VE SADELEŞTİRİLMESİ

Bilgisayar Mühendisi Yavuz TEKBAŞ

Karadeniz Teknik Üniversitesi Fen Bilimleri Enstitüsünce
"BİLGİSAYAR YÜKSEK MÜHENDİSİ"
Unvanı Verilmesi İçin Kabul Edilen Tezdir.

Tezin Enstitüye Verildiği Tarih: 24.12.2012
Tezin Savunma Tarihi: 11.01.2012

Tez Danışmanı: Yrd. Doç. Dr. Hüseyin PEHLİVAN

Trabzon 2013

Karadeniz Teknik Üniversitesi Fen Bilimleri Enstitüsü

Bilgisayar Mühendisliği Anabilim Dalında

Yavuz TEKBAŞ tarafından hazırlanan

OTOMATİK KOD ÜRETİM ARAÇLARI YARDIMIYLA MATEMATİKSEL İFADELERİN TÜREVLERİNİN HESAPLANMASI VE SADELEŞTİRİLMESİ

başlıklı bu çalışma, Enstitü Yönetim Kurulunun 12 / 06 / 2012 gün ve 1460 sayılı kararıyla oluşturulan jüri tarafından yapılan sınavda

YÜKSEK LİSANS TEZİ

olarak kabul edilmiştir.

Jüri Üyeleri

Başkan : Doç. Dr. Mustafa ULUTAŞ …...……

Üye : Yrd. Doç. Dr. Hüseyin PEHLİVAN …...……

Üye : Yrd. Doç. Dr. H.İbrahim OKUMUŞ ……...……

Prof. Dr. Sadettin KORKMAZ

Enstitü Müdürü

ÖNSÖZ

Günümüzde, hızla gelişen teknolojiyle birlikte çeviricilere daha çok ihtiyaç duyulmaktadır. Yeni diller ve yeni mimariler arasındaki farklılıklara bağlı olarak gelişmektedir ve önemle çalışılan ve sağlıklı bir gelişim sergileyen geliştirilen bir alandır.

Bütün programlama dilleri insanların anlayabildiği biçimlerden (kaynak kodlar), bilgisayarların anlayabildiği biçimlere (makine kodları) çevrilirler. Bu aşama programcının programını kolaylıkla yazabilmesi için gerekmektedir. Kaynak kod yazmak, makine kodu yazmaktan çok daha kolaydır. Çevirme işlemi çeviriciler tarafından gerçeklenir.

Yapmış olduğum çalışmamda otomatik kod üretim araçları yardımıyla matematiksel ifadelerin türevlerinin hesaplanması ve sadeleştirilmesi yapılmış ve gerçeklenmiştir.

Bu çalışmada danışmanlığımı üstlenen Sayın Yrd. Doç. Dr. Hüseyin PEHLİVAN hocama yardımlarından dolayı teşekkür ederim. Ayrıca her zaman bana destek olan eşime ve aileme desteklerinden dolayı teşekkür ederim.

TEZ BEYANNAMESİ

Yüksek Lisans Tezi olarak sunduğum "OTOMATİK KOD ÜRETİM ARAÇLARI YARDIMIYLA MATEMATİKSEL İFADELERİNTÜREVLERİNİN HESAPLANMASI VE SADELEŞTİRİLMESİ" başlıklı bu çalışmayı baştan sona kadar danışmanım Yrd. Doç. Dr. Hüseyin PEHLİVAN'ın sorumluluğunda tamamladığımı, verileri/örnekleri kendim topladığımı, deneyleri/analizleri ilgili laboratuarlarda yaptığımı/yaptırdığımı, başka kaynaklardan aldığım bilgileri metinde ve kaynakçada eksiksiz olarak gösterdiğimi, çalışma sürecinde bilimsel araştırma ve etik kurallara uygun olarak davrandığımı ve aksinin ortaya çıkması durumunda her türlü yasal sonucu kabul ettiğimi beyan ederim. 11/01/2013

Yavuz TEKBAŞ

İÇİNDEKİLER

Yüksek Lisans Tezi

ÖZET

OTOMATİK KOD ÜRETİM ARAÇLARI YARDIMIYLA MATEMATİKSEL İFADELERİN TÜREVLERİNİN HESAPLANMASI VE SADELEŞTİRİLMESİ

Yavuz TEKBAŞ

Karadeniz Teknik Üniversitesi
Fen Bilimleri Enstitüsü
Bilgisayar Mühendisliği Anabilim Dalı
Danışman: Yrd. Doç. Dr. Hüseyin PEHLİVAN
2013, 69 Sayfa

C, C++ ve Java gibi birçok programlama dilinde kod üretebilen otomatik kod üretim araçları biçimsel dillerin analiz ve değerlendirme süreçlerinin en önemli aktörlerinden biridir. Programlama dilleri, doküman formatları ve emir kümeleri gibi çeşitli biçimsel dillerin sözdizimlerini betimlemek için içerikten bağımsız gramerler (CFG) kullanılır.

Bu çalışmada bir simgesel hesaplama uygulaması olarak matematiksel ifadelerin türevlerinin hesaplanması ve sadeleştirilmesinde otomatik kod üretim araçlarının nasıl kullanılacağı gösterilmiştir. Matematiksel ifadelerin sözdizimsel ve anlamsal yapısına uygun bir CFG grameri geliştirilmiş olup işleç ve fonksiyonları temsil etmek üzere sözdizim sınıfları tanımlanmıştır.

Çalışmada, Java CC aracı yardımıyla, matematiksel ifadeleri bileşenlerine ayırarak sözdizimi analizi ve ardından nesne ağacı dönüşümü yapan ayrıştırıcılar için otomatik kaynak kod üretimi gerçekleştirilmiştir. Nesne ağacının değerlendirilmesi aşamasında ise ağaç düğümlerinin içerdiği ifadelerin türevlerinin hesaplanması ve sadeleştirilmesi tartışılmış ve kodlamaları gösterilmiştir.

Anahtar Kelimeler: Simgesel hesaplama, LL(k) grameri, Ayrıştırıcılar, Sözdizimi sınıfları, Visitor tasarım deseni

Master Thesis

SUMMARY

CODE PRODUCTION TOOLS USING AUTOMATIC CALCULATION OF DERIVATIVES AND SIMPLIFICATION MATHEMATİCAL EXPRESSİONS

Yavuz TEKBAŞ

Karadeniz Technical University
The Graduate School of Natural and Applied Sciences
Computer Engineering Graduate Program
Supervisor: Yrd. Doç. Dr. Hüseyin PEHLİVAN
2013, 69 Pages

Automatic code generation tools which can produce code for syntax analyzers in many programming languages such as C, C++ and Java are the most important actors of the analysis and evaluation processes of formal languages. Context-free grammars (CFG) are used to describe the syntax of various formal languages such as programming languages, document formats and instruction sets.

This study addresses how automatic code generation tools can be used in the field of the symbolic computation, applying them for the calculation of derivations of mathematical expressions and the simplification of the resulting expressions. A CFG grammar is developed for the syntactic and semantic structures of mathematical expressions and the abstract syntax classes representing the operators and functions are defined.

In the study, by means of JavaCC , separating mathematical expressions into tokens, automatic code generation for parsers which perform syntax analysis and then produce object tree is carried out. At the phase of evaluating the object tree, the derivations and simplifications of the expressions contained in the tree nodes are discussed and the corresponding code instances are illustrated.

Key Words: Symbolic computation, LL(k)grammar, Parser, Classes of syntax, the Visitor pattern

ŞEKİLLER DİZİNİ

Sayfa No

TABLOLAR DİZİNİ

Sayfa No

SEMBOLLER DİZİNİ

DFA	Belirli Sonlu Otomata (DeterministicFiniteAutomata)
CFG	Bağlam Bağımsız Gramer (Context – FreeGrammer)
AST	Soyut Sözdizim Ağacı (AbstractSyntaxTree)
LL(k)	Soldan sağa ayrıştırma – Sola dayalı türetim – k kelime kontrolü (Left-to-Right Parsing – LeftmostDerivation – k lookahead)
LR(k)	Soldan sağa ayrıştırma – Sağa dayalı türetim – k kelime kontrolü (Left-to-Right Parsing – RightmostDerivation – k lookahead)
JTB	Java Ağaç Oluşturucu (Java Tree Builder)

1. GENEL BİLGİLER

1.1.Giriş

Simgesel hesaplama matematiksel işlemlerin simgeler üzerinden bilgisayar yardımıyla tam ve hatasız olarak gerçekleştirilmesiyle ilgilenir. Bunun için öncelikle hesaplama yapılacak ilgili matematiksel alandaki nesnelerin tam olarak temsil edilebilmeli ve problem çözüm yöntemlerinin bilgisayar programları ile yürütülebilecek algoritmalara dönüştürülebilmelidir [1,2]. Ancak bazı matematiksel hesaplamalar, uygulayıcının ilgili alandaki temel işlemleri yapabilmesinin ötesinde farklı beceri ya da deneyimler gerektirir. Örneğin, belirsiz entegral hesaplamaları için öğretilen standart yöntemlerle bir algoritma oluşturmaz. Belirsiz entegrallerin algoritmik çözümü temel fonksiyonlar (üstel, logaritmik, trigonometrik ve rasyonel) sınıfı için gerçekleştirilmiş olup daha genel fonksiyon sınıflarına genelleme çalışmaları sürmektedir [3,4].

Simgesel hesaplama sistemleri genel ve özel amaçlı olmak üzere iki sınıfa ayrılabilir. Bu sistemlerin her biri matematiksel ifadelerin yazılabileceği bir programlama diline sahiptir. Genel amaçlı sistemler matematiğin çeşitli dallarındaki problemleri çözmeye yönelik etkileşimli bir hesaplama ortamı sunar. Matlab, Maple, Macsyma, Mathematica, Axiom ve MuPad gibi araçlar genel amaçlı sistem örnekleridir. Özel amaçlı hesaplama sistemleri özel uygulama gereksinimlerini karşılamaya yönelik sınırlı sayıda işlem kapasitesi ile donatılmıştır. Örneğin, gruplar teorisi alanında GAP ve Magma, komütatif cebir ve cebirsel geometri çalışmaları için CoCoA ve Macaulay, yüksek enerji fiziği hesaplamaları için Schoonship araçları geliştirilmiştir.

Otomatik kod üretim araçları biçimsel dillerin analiz, yorumlama veya derleme süreçlerini kolaylaştırmak amacıyla geliştirilmiştir. Programlama dilleri, doküman formatları ve emir kümeleri gibi çeşitli biçimsel dillerin sözdizimlerini betimlemek üzere içerikten bağımsız gramerler (CFG) kullanılır. Genellikle BNF notasyonunda yazılan içerikten bağımsız gramerler için etkin ayrıştırıcılar üretebilen çok sayıda otomatik kod üretim aracı mevcuttur. Birçok modern programlama dillerinde kod üretebilecek çeşitleri bulunan otomatik kod üretim araçlarının sadece Java programlama dili için onlarca örneği vardır; ANTLR [5], JavaCC [6], SableCC [7], JTB [8], JLex [9] ve JFlex [10]. JavaCC, Java prog-

ramlama dilinde geliştirilecek uygulamalarla kolayca entegre edilebilecek ve girdi verileri üzerinde çalışacak kelimesel çözümleyici ve ayrıştırıcı bileşenleri için kaynak kod üretir

1.2.Dil Çeviricileri

Bu bölümde çevirici sistemlerinin genel yapısı anlatılmış ve çevici sistemlerinin örnekleri olan derleyiciler ile yorumlayıcıların davranışları karşılaştırılmıştır.

1.2.1.Derleyiciler

Derleyici, basitçe herhangi bir dilde yazılmış olan kodu (kaynak kodu yada source code) istenilen başka bir kod haline dönüştüren programdır. Genelde üretilen bu kod ortama göre çalıştırılabilir kod (executable code) olarak üretilmektedir. Ancak bir derleyicinin daha doğru tanımı bir dildeki kodu başka dile çeviren program olarak yapılabilir.

Derleyiciler günümüzde daha çok bir dilde yazılmış koddan, işletim sistemi ve donanım bağımlı kodların üretilmesinde kulllanılmaktadırlar. Bu üretim sırasında ya doğrudan işletim sisteminin anlayacağı ve çalıştıracağı kodları üretirler ya da işletim sisteminde bulunan veya yine dil bağımlı olarak çalışan bağlayıcı (linker) programların anlayacağı ara kodları üretirler.

Derleyiciler bu kod üretmesi sırasında, üretilen kodun en verimli şekilde üretilmesi için kod iyileştirmesi (optimisation) da yapmaktadırlar. Yani hedef dildeki çalışma süresi ve hafıza ihtiyacı en az olan kodu üretmek bir derleyicinin daha başarılı olma kriterlerinden birisidir. Aynı zamanda kaynak kodda (source code) bulunan hataların yakalanması bu hataların programcıya bildirilmesi de derleyicilerin diğer görevlerinden birisidir.

1.2.2 Yorumlayıcılar

Yorumlayıcı, bir programı veya bir komutu çalıştırmaya yarayan programlar için kullanılır. Genellikle derleyiciler ile karışan ve çoğu zaman aynı görevi icra eden yorumlayıcılar da derleyiciler gibi kodu bir dilden başka bir dile çevirme işlemini yerine getirirler. Basitçe görevleri ve tipleri aşağıdaki şekilde listelenebilir:

1. Kaynak kodu çalıştırmak
2. Bir kaynak kodu çalıştırılabilir farklı bir kod haline tercüme etmek

3. Daha önceden hazırlanmış olan (çoğu zaman önceden derlenmiş bir koddur) kodları yeri geldiğinde çalıştırmaktır.

1.2.3. Derleyici ile Yorumlayıcının Karşılaştırılması

Basitçe, bir kaynak kodu hedef koda çevirdikten sonra çalıştıran ve dolayısıyla koddaki hataları yakalama işlemini ve kodun iyileştirilmesini daha kodu çalıştırmadan yapan çeviricilere derleyici, kodu satır satır veya bloklar halinde çalıştırıp sırası gelmeyen satırları hiç çalıştırmayan bu satırlardaki hataları hiçbir zaman göremeyen ve kodun bütününe ait iyileştirmeleri yapamayan çeviricilere de yorumlayıcı (interpreter) adı verilmektedir.

Derleyiciler, programın analizi ve işlenmesi için daha fazla zaman gerektirirlerken yorumlayıcılar daha az analiz ile programı çalıştırırlar. Derleyiciler için sonuç, genelde makine bağımlı bir kod iken yorumlayıcılar için genelde orta seviye bir koddur. Derleyicilerin çıktısı, doğrudan işletim sistemi üzerinde çalışırken, yorumlayıcıların çıktısı genelde çalışmak için ilave bir programa ihtiyaç duyar. Derleyici çıktıları genelde hızlı çalışırken, yorumlayıcı çıktıları daha yavaş çalışır.

Derleyici, kaynak kodun tamamını makine koduna çevirir. Kaynak kod dosyası bir kez derlenir ve sonucunda makine kodları içeren, koşturulabilir bir dosya üretilir. Oluşturulan dosya bir sonraki aşamada hiçbir işleme gerek duymadan koşturulabilir. Kaynak kodda yapılan bir değişiklikte ise kaynak kod yeniden derlenmek zorundadır.

Yorumlayıcı kaynak kodun ilk satırından başlayarak her satır için çeviri işlemini gerçekleştirip ardından kodun icrasını gerçekleştirir. Yorumlama sonucunda herhangi bir koşturulabilir veya benzeri dosya üretilmediğinden her defasında kaynak program üzerinden yorumlama işlemi tekrarlanır. Ayrıca kaynak kodun icrası anında kaynak programın yanı sıra yorumlayıcının da belleğe yüklenmesi gereklidir.

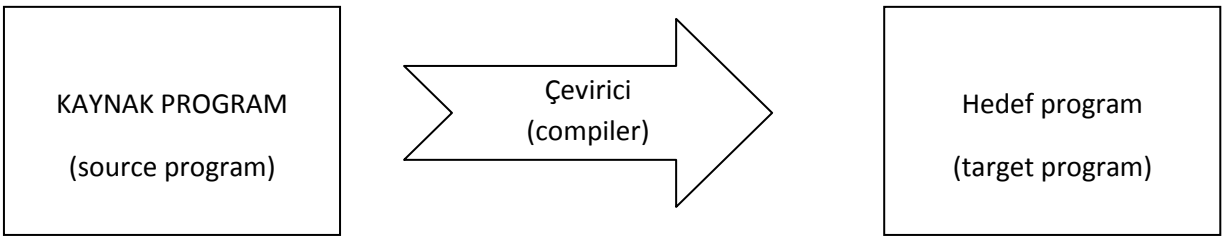
Derleyici ile yorumlayıcının birbirlerine göre avantajları aşağıdaki gibi sıralanabilir [13];

• Koşma anında yorumlayıcının belleğe yüklenmesi, kaynak programa daha az bellek alanı tahsis edilmesine yol açar. Derleyicinin ise sadece derleme anında belleğe yüklenip sonrasında yüklenmeye ihtiyaç duymaması kaynak programa daha fazla bellek alanı tahsis edilmesini sağlar.

• Derlenmiş kaynak kodu, derleyiciye ihtiyaç duymadan tekrar koşturulabilirken, yorumlanan kaynak kod tekrar çalıştırılmak istendiğinde yorumlayıcı programa gereksinim duyar.
• Makine koduna çevrilmiş programlar, yorumlanarak çalıştırılan programlara göre daha hızlı çalışırlar.
• Yorumlamalı programların kodunda, derlenmiş programlara göre daha kolay ve hızlı şekilde değişiklik yapılabilir.

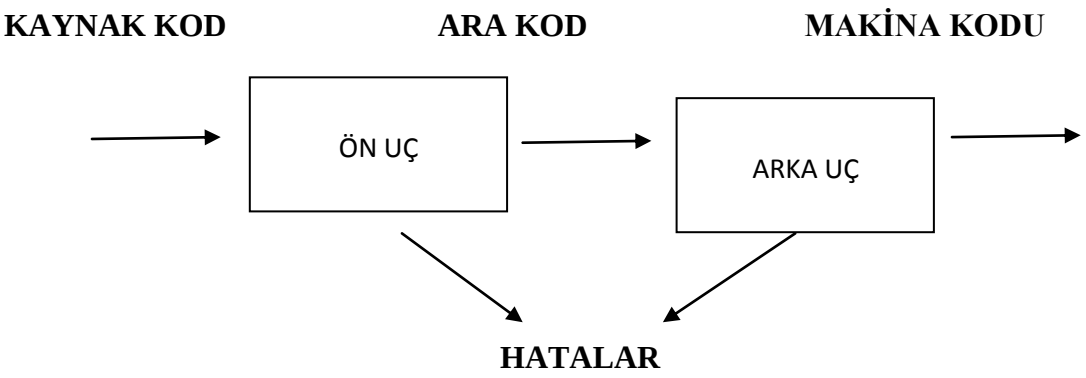
1.2.4. Çeviricilerin Genel Yapısı ve Bileşenleri

Şekil 1 de genel mimarisi gösterilen bir çeviriciye kaynak kod verildikten sonra çeşitli ara işlemlerden geçirilir ve varsa hatalar ayıklandıktan sonra makine koduna dönüşümler yapılmaktadır.



Şekil 1.1. Çevirici genel mimarisi

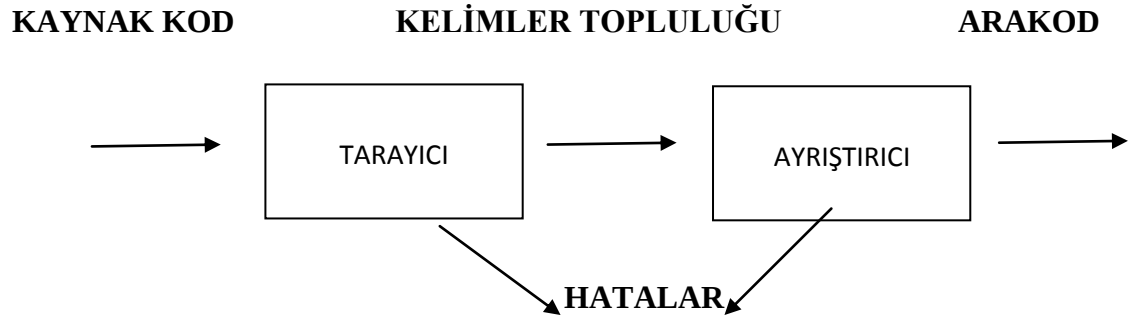
Bu ara işlemleri iki temel bileşene ayırabiliriz. Bunlar önuç ve arka uçdur. Genel olarak bunların işlevlerini ifade edersek önuç kaynak kodu alır IR diye bir ara koda dönüştürür. Arka uç ise önuçun oluşturduğu Ara kod (IR) u alır çeşitli alt işlemlerden geçirerek hedef koduna dönüştürür.



Şekil 1.2. Önuç ve Arka uç

1.2.4.1. Ön Uç (Front end)

Önuç, kaynak kodu tokenlarına ayırdıktan sonra bunları anlamlı parse ağaçlarına döker ve ardından da son aşamada bu işlemden geçmiş kodları IR (ara koda) diye bir ara koda dönüştürür.

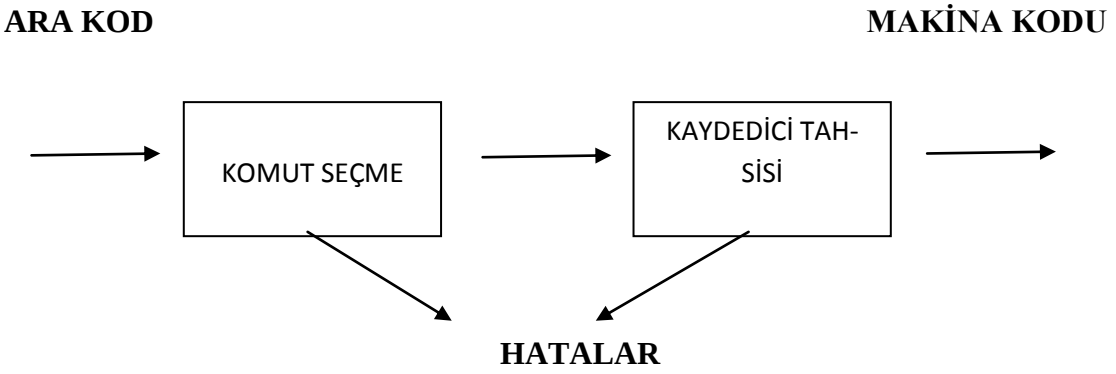


Şekil 1.3. Önuçun iç yapısı

1.2.4.2. Arka Uç (Back end)

Burada önuçta IR gibi bir ara koda dönüştürülen kaynak kod arka uçtaki işlemler vasıtasıyla IR ara kodu Makine koduna çevirilir. Arka uçun içyapısına bakarsak şunlar karşımıza çıkacaktır.

Arka uç yapısında Instruction Selection yapısında Ara kodun yapısındaki kodları bulunduğu makineyle eşleştiriyor. Yani bulunduğu makine kodlarıyla IR ara kodunu eşleştiriyor.



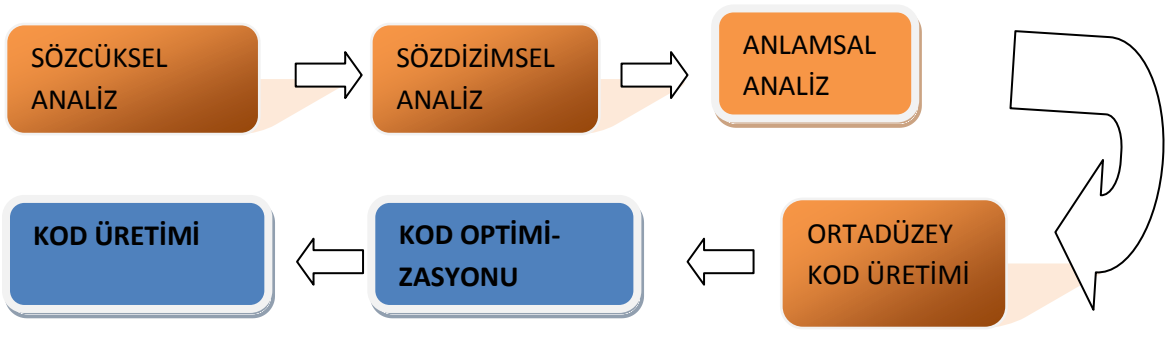
Şekil 1.4. Arka uçun iç yapısı

1.3. Çevirici Aşamaları

1.3.1. Temel Çevirici Aşamaları

Bir derleyicinin tasarım süreci üç temel aşamadan oluşur;

- Çeviri (Translation),
- Doğrulama (Validation),
- Eniyileme (Optimization).



Şekil 1.5. Çevirici aşamaları

Derleyici tasarımının bu kadar ayrı safhalara bölünmesinin nedeni her aşama sonucunda elde edilen veri yapılarının farklı platformlar için kod üretimi gibi çeşitli amaçlara

yönelik olarak kullanılmasıdır ve kaynak kodun bir başka kaynak koda dönüşümünde ilk önce analiz aşaması olan front end adı verilir. Analiz aşamasından sonra parçalanmış halde olan kodun yeniden birleştirilip, koşma işlemi gerçekleştirilmesi gerekmektedir. Sentez işleminin yapıldığı bu aşamaya ise derleyicinin arka yüzü (backend) adı verilir. Bold yazılı olmayan bölüm front end bold yazılı yer olan bölüm ise back end' i temsil etmektedir.

1.3.2. Kelimesel Analiz

Yüksek seviyeli bir programlama dilinde yazılan kaynak kodu okuyan ve kelimesel analizini yapan ilk bölümü olan token yapılarıdır. Kaynak kodun kendisi karakterlerin bir dizisi olarak göz önüne alınır ve söz konusu dilde bir anlamı olan en küçük karakter kümelerine ayrılır. Her bir küme uygun bir token sınıfı ile temsil edilir. Fonksiyon, değişken ve parametre isimleri için ayrı ayrı bölünmüştür. Kod içinde bulunan boşluk karakterleri tokenları belirlemede kullanılır. Aynı şekilde açıklama satırları da dikkate alınmaz.

Bu aşamada kaynak kod içerisinde bulunan ve çözümleyici tarafından belirlenen dile uygun olmayan yapılar tespit edilip hata mesajı olarak gösterilir. Kaynak kodda hata çıkması durumunda kodun analizi için bir sonraki aşamaya geçilmez.

Kelimesel analizin gerçekleştirilmesi için dönüştürme mekanizmalarından faydalanılır. Bu mekanizmalar; düzenli ifadeler ve sonlu otomatalardır.

Şekil 1.6. Kelimesel Analiz

1.3.2.1. Düzenli ifadeler

Düzenli ifadeler bütün karakterler kümesinin içinde belli bir karakter kümesini seçmeye yarayan bir mekanizmadır.

Düzenli ifadeleri kullanırken bazı özel kavramlar kullanılır. Bu mekanizmada kullanılan temel kavramlar ise şunlardır [19]:

• Seçim : “ | ” işareti kullanılarak ifadeler seçeneklere ayrılır. Örneğin; (a | b) ifadesinde “a” ya da “b” ile eşleşme yapılır.
• Birleştirme : “ · ” işareti kullanılarak ifadeler birleştirilir. Örneğin; (a | b) · a ifadesinde “aa” ya da “ba” ifadeleri ile eşleşme yapılır.
• Epsilon : “ ε ” işareti ile kullanılarak boş değer gsterimi sağlanır. Örneğin; (a | ε) ifadesinde aya da boş değer eşleşmesi yapılır.
Tekrarlamalar: Bir karakterin ya da grubun ardından gelerek öncesindeki yapının kaç kez tekrarlanacağını belirtir. En temel tekrarlamalar “ * ” , “ + ” ve “ ? ” dir.
“ * “ ; hiç bulunmayacağını ya da herhangi bir sayıda tekrarlanacağını belirtir. “ + “ ; en az bir kez olmak üzere herhangi bir sayıda tekrarlanacağını belirtir. “ ? “ ; hiç bulunmayacağını ya da bir kez bulunacağını gösterir. Düzenli ifadelerdeki bu kavramlar kullanılarak programlama diline ait kelimesel yapılar tanımlanır.
Programlama dilleri için genel olan bazı yapıların, düzenli ifade formatında gösterimleri aşağıdaki biçimdedir:
IF : “ if ”
*DEĞİŞKEN : [a-z,0-9]**
SAYI : [0-9]+
GERÇEL : ([0-9]+ " . " [0-9]) | ([0-9]* " . " [0-9]+)*
YORUM : (" -- " [a-z] " \n ")*
Bu kurallar çerçevesinde oluşturulacak yapıda bazı belirsizlikler meydana gelebilmektedir. Örneğin “ifa” yapısı düzenli ifade tarafından IF ve DEĞİŞKEN mi yoksa sadece DEĞİŞKEN olarak mı algılanması konusunda belirsizlik oluşmaktadır. Bu belirsizliği ortadan kaldırmak için kelimesel çözümleyiciler iki önemli kural geliştirmişlerdir.

1- En uzun eşleşme: Bu kurala göre en uzun eşleşmeye sahip ifade öncelikli seçilir.
2- Kural önceliği: Düzenli ifade tanımlama sırasına göre ilk karşılaşılan ifade öncelikli seçilir.
3-

1.3.2.2. Kelimesel Çözümleyici Üreteçleri

Genellikle BNF notasyonunda yazılan içerikten bağımsız gramerler için etkin ayrıştırıcılar üretebilen çok sayıda otomatik kod üretim aracı mevcuttur. Bu araçlar kelimeler topluluğunu kendi içinde kelimeler topluluğuna (token) dönüştürülür.

Programlama dillerinde geliştirilecek uygulamalarla kolayca entegre edilebilecek ve girdi verileri üzerinde çalışacak kelimesel çözümleyici ve ayrıştırıcı bileşenleri için kaynak kod üretir.

Birçok modern programlama dillerinde kod üretebilecek çeşitleri bulunan otomatik kod üretim araçlarının sadece Java programlama dili için onlarca örneği vardır; ANTLR , JavaCC , SableCC ,JTB , JLex ve JFlex .

Java programlama dilinde otomatik kod üretme aracı olan JavaCC uyumlu örnek kod bloğu ;

```
PARSER_BEGIN(Derivation)
public class Derivation
 {……}

PARSER_END(Derivation)
SKIP : { " " | "\t" }
TOKEN : { < NUMBER : (["0"-"9"])+ ("." (["0"-"9"])+)? > }
TOKEN : { < EOL : "\n" > }
TOKEN : /* OPERATORS */
{   < PLUS: "+" >
  | < MINUS: "-" >
  | < MULTIPLY: "*" >
  | < DIVIDE: "/" >
  | < POWER: "^" > }
```

şeklindedir.

1.3.3. Sözdizimsel Analiz (Ayrıştırma)

Bu safhada yapılacak işlemler;

- Programlama dilinin sözdizimsel kurallarının tanımlanması,
- Programın kaynak kodunun, tanımlanan dilin sözdizimsel yapısına uyup uymadığını kontrol edip var olan hataları belirlemek,

- Kelimesel analiz aşamasında üretilen kelimeler topluluğuna hiyerarşik bir yapı kazandırıp, bir sonraki derleme aşamasına geçiş için yeni bir veri yapısı üretmekdir.

Şekil 1.7. Sözdizimsel Analiz

1.3.3.1. İçerikten Bağımsız Gramer (Context Free Gramer)

Bir içerikten 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 $X \rightarrow w$ biçimindedir; X bir nonterminal simgesine, w ise terminal (bir kelime) ve/veya nonterminal dizilerine karşılık gelir. (w boş olabilir).

CFG notasyonu, temsil ettiği programlama dilindeki kelimeler topluluğundan oluşmaktadır. Her bir kelime ise önceden belirlenmiş alfabede bulunan sembollerin sonlu dizisinden oluşmaktadır. Ayrıştırma işleminde kelimeler dizisi kaynak programın, semboller dizisi kelimesel yapıların ve alfabe de kelimesel çözümleyicilerin geri döndürdüğü kelime türlerinin bir kümesidir.

Girdi olarak verilen bir karakter dizisinin, CFG'ye ait kurallardan nasıl üretileceğinin ortaya çıkartılması, özyinelemeli tanımlar konusunda belli bir karakter dizisinin kurallar kullanılarak elde edilişinin ispat edilmesi ile aynı anlamı taşımaktadır

Temel olarak bir içerik bağımsız gramer dört özellik içermelidir. Bunlar sonlular (terminals), devamlılar (nonterminals), bağlantılar (relation) ve başlangıç sembolu (starting symbol) olarak sıralanabilir. Bir gramerin tanımı sırasında kullanılan bu kümeler aşağıdaki şekilde yazılabilir:

G = (V , ∑ , R , S)

Bu gösterimdeki grammer (G) , V devamlıları, ∑ sonluları, R bağlantıları, S ise başlangıç sembolünü göstermektedir.

Örneğin S -> aSb | ab şeklinde tanımlanan bir dilde: G= ({S} , {a,b} , {S -> aSb | ab} , S) gösterimi kullanılabilir.

1.3.4. Türetim

Kaynak kodun ilgili programlama dilinin gramerine ait olup olmadığını belirlemek için türetim işlemi gerçekleştirilir. Türetim işlemine başlangıç sembolü ile başlanır ve üretim kurallarına uygun şekilde eşleştirme gerçekleştirilir. Türetim işleminin farklı yöntemleri bulunmaktadır;

- Sola Dayalı Türetim yönteminde ilk olarak en soldaki non-terminal sembolü genişletilmeye çalışılır.
- Sağa Dayalı Türetim yönteminde ilk olarak en sağdaki non-terminal sembolü genişletilmeye çalışılır.

  while(x<10) { x = x + 1 ; } kodu için sola dayalı türetim;

  1. While

  2. " while " "(" Exp ")" Stmt

  3. " while " "(" Primary_Exp Symbol Primary_Exp ")" Stmt

  4. " while " "(" Id (x) Symbol Primary_Exp ")" Stmt

  5. " while " "(" Id (x) Symbol (<) Integer (10) ")" Stmt

  6. " while " "(" Id (x) Symbol (<) Integer (10) ")" Block

  7. " while " "(" Id (x) Symbol (<) Integer (10) ")" "{" Stmt "}"

  8. " while " "(" Id (x) Symbol (<) Integer (10) ")" "{" Assignment "}"

  9. " while " "(" Id (x) Symbol (<) Integer (10) ")" "{" Id (x) "=" Exp "}"

  10. " while " "(" Id (x) Symbol (<) Integer (10) ")" "{" Id (x) "=" Primary_Exp Symbol Primary_Exp "}"

  11. " while " "(" Id (x) Symbol (<) Integer (10) ")" "{" Id (x) "=" Id (x) Symbol (+) Primary_Exp "}" 14

  12. " while " "(" Id (x) Symbol (<) Integer (10) ")" "{" Id (x) "=" Id (x) Symbol (+) Integer (1) "}"

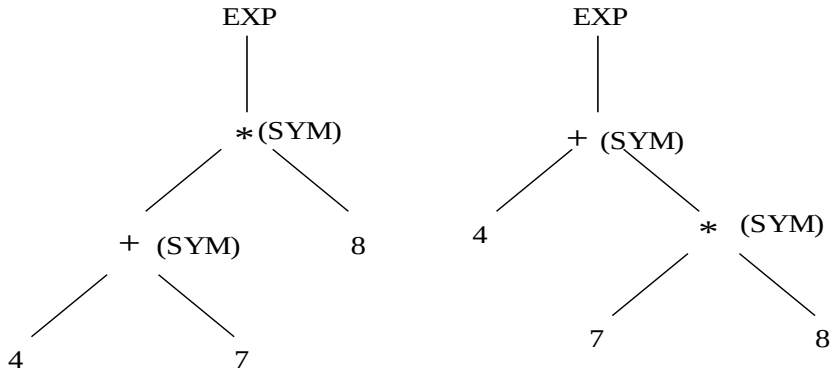
1.3.5. Ayrıştırma Ağaçları

Ayrıştırma ağacı bir kelime topluluğunun, gramer kurallarına göre yapısal olarak parçalanıp ağaç şeklinde ifade edilmesidir. Ayrıştırma ağacında bulunan iç düğümler gramerde bulunan non-terminalleri ifade ederken, yaprak düğümler gramerin terminal yapılarıyla ifade edilir.

1.3.5.1. Belirsiz Gramer

Derleme esnasında belirsiz gramerler sorun teşkil eder. Bazı gramer kuralları birden fazla sözdizim ağacı üretebilmekte ve bizi farklı sonuçlara götürmektedir. Bu amaçla programlama dilleri bu belirsizliklerlerden kurtulması gereklidir.

Örneğin; İfade olarak “4+7*8” alalım ve ağacımıza yerleştirelim.



(I) (II)

Görüldüğü üzere ayrıştırma işleminin yönüne bağlı olarak iki farklı sözdizim ağacı oluşmuştur. Farklı sözdizim ağaçları üzerinden yapılacak değerlendirmeler farklı değerler üretir. Yani, bir numaralı ağaç için 88, iki numaralı ağaç için 60 elde edilir. Bu durum gramer kurallarının işleçlerin öncelik düzeylerini dikkate almamasından kaynaklanır. Eğer ayrıştırma ağacında işleçlerin (operators) öncelik seviyelerigösterilirse belirsizlik olmaz.

1.4. Ayrıştırma

Token dizisi üzerinden gramer kurallarına uyarak ayrıştırma işlemi yapar ve günümüzde kullanılan ayrıştıma yöntemleri bazıları aşağıda açıklanmıştır.

1.4.1. Ayrıştırıcılar

Ayrıştırma işleminin gerçekleştirilmesi üzerine birçok algoritma geliştirilmiştir. Bu algoritmalar 3 temel sınıfta ayrılabilir:

- ✓ Yukardan Aşağıya (Top – Down) Ayrıştırma
  - Recursive Descent Ayrıştırma
  - LL Ayrıştırma
- ✓ Aşağıdan Yukarıya (Bottom – Up) Ayrıştırma
  - LR Ayrıştırma
    - SLR Ayrıştırma
    - LALR Ayrıştırma
    - Canonical LR Ayrıştırma
- ✓ Derleyici Derleyiciler
  - Bison
  - Lemon
  - Lex
  - JavaCC

1.4.2. Top – Down Ayrıştırma

Sezgisel olarak ayrıştırıcı başlangıç non-terminalinden başlar ve kaynak kodu en sondaki terminallere kadar kurallara uygun şekilde ayırmaya çalışır.

1.4.2.1. Aşağıya Özyinelemeli (Recursive Descent) Ayrıştırma

Basitçe bir gramerdeki her nonterminal için bir alt program vardır. Alt programlar nonterminaller tarafından üretilen cümleleri ayrıştırır. Ayrıştırma algoritmasının basit bir yapısı vardır. Aşağıda verilen grameri göz önüne alalım.

$S \rightarrow$ for E {S;}

$S \rightarrow$ print E

$E \rightarrow$ num = num

Bu gramerin kuralları için aşağıdaki gibi alt program tanımlamaları yapılabilir.

```
final int FOR=1, LBRACKET=2, RBRACKET=3, PRINT=4, NUM=5, EQ=6;
int tok = getToken();
void advance() {tok=getToken();}
void eat(int t) {if (tok==t) advance(); else error();}
void S() {switch(tok) {
case FOR: eat(FOR); E();eat(LBRACKET);S();eat(RBRACKET);break;
case PRINT : eat(PRINT);E();break;}
void E(){eat(NUM);eat(EQ);eat(NUM)}
```

Aşağıya Özyinelemeli ayrıştırmanın çalışması için gramerin her bir alt ifadesindeki ilk terminal sembolünün, bir sonraki adımda hangi kuralın kullanılacağını belirleyebilmesi için gerekli bilgiye sahip olması gerekir. Bu bilgiye sahip olunmazsa ayrıştırıcı çalışmaz. Bu algoritmada, ayrıştırma tablosunu oluşturabilmek için FIRST ve FOLLOW set kavramları geliştirilmiştir.

FIRST set kavramı, terminaller ve non-terminallerden oluşmuş bir gramerdeki herhangi bir kuralın hangi terminal sembolü ile başlayacağını belirler. Bir gramerin aynı sol tarafa sahip en az iki kuralının, aynı FIRST sete sahip olması o gramerin bu algoritma yardımıyla ayrıştırılamayacağını gösterir.

FIRST set kavramına örnek olarak;

- $S \rightarrow$ For E {S;}
- $S \rightarrow$ print E
- $E \rightarrow$ num = num

FIRST (S) = { for, print }

FIRST (E) = { num }

FOLLOW set kavramı ise non-terminali takip eden ilk terminal olarak tanımlanır. Dolayısıyla yalnızca non-terminal ifadeler için hesaplama yapılabilir.

$Z \rightarrow d$ $\quad Z \rightarrow X\ Y\ Z$

$Y \rightarrow$ $\quad Y \rightarrow c$

$X \rightarrow Y$ $\quad X \rightarrow a$

Tablo 1.1. FIRST ve FOLLOW seti

	FIRST	**FOLLOW**
X	**a c**	**a c d**
Y	**c**	**a c d**
Z	**a c d**	

1.4.2.2. LL Ayrıştırma

Ayrıştırma işlemini, soldan sağa doğru ve sola dayalı türetim ile gerçekleştirmektedir. LL(k) kullanılması k tane kelimeye bakılıp karar verileceğini anlamını taşımaktadır.

L eft-to-Right parse
L eft-most Derivation
(k) look-ahead şeklindedir.
İlk 'L' girdinin soldan sağa doğru işleme tabi tutulacağını ifade eder.
İkinci 'L' sola dayalı bir türetim yapılacağını ifade eder.
(k) sayısı ise ayrıştırma yapılırken kaç kelimeye bakılıp karar verileceğini (tahmin edileceğini) ifade eder.
LL Ayrıştırma;
• Giriş cümlesini saklayacak bir arabelleğe,
• Ayrıştırılmış terminal ve non-terminal yapıları saklamak için bir yığına,
• Girdi ve yığında bulunan verilere dayanarak hangi gramer kuralının uygulandığını gösteren bir ayrıştırma tablosuna sahiptir.

Ayrıştırma tablosunda bulunan kurallardan, giriş cümlesinde bulunan sembollere en çok uyanını bularak yığına atar. Bu işlem girilen cümlenin sonuna kadar yinelemeli bir şekilde devam ettirilir. Ayrıştırıcı işleme başlarken "S" başlangıç sembolüne, "$" terminaline sahiptir. Bu terminal özel bir durum olup, yığının başını aynı zamanda giriş cümlesinin sonunu temsil eder. Basit bir gramer örneği bir LL ayrıştırıcısının ürettiği ayrıştırma tablosu aşağıdaki gibidir.

1. S → F
2. S → **(** S + F **)**
3. F → m

Giriş cümlesi (m+m)

Tablo 1.2. Ayrıştırma Tablosu

	(	)	m	+	$
S	2	-	1	-	-
F	-	-	3	-	-

Ayrıştırıcı, ilk cümleden "(" kelimesini ve yığından S değerini bellekten okur. Tablodan bu kelimenin 2 numaralı kural olduğunu belli eder ve sonra kural numarasını geri değer olarak döndürüp ve yığın içinde değişim işlemini gerçekler.

[(S , + , F ,) , $]

Diğer bir sonraki adımda "(" kelimesini giriş cümlesinden ve yığından siler.

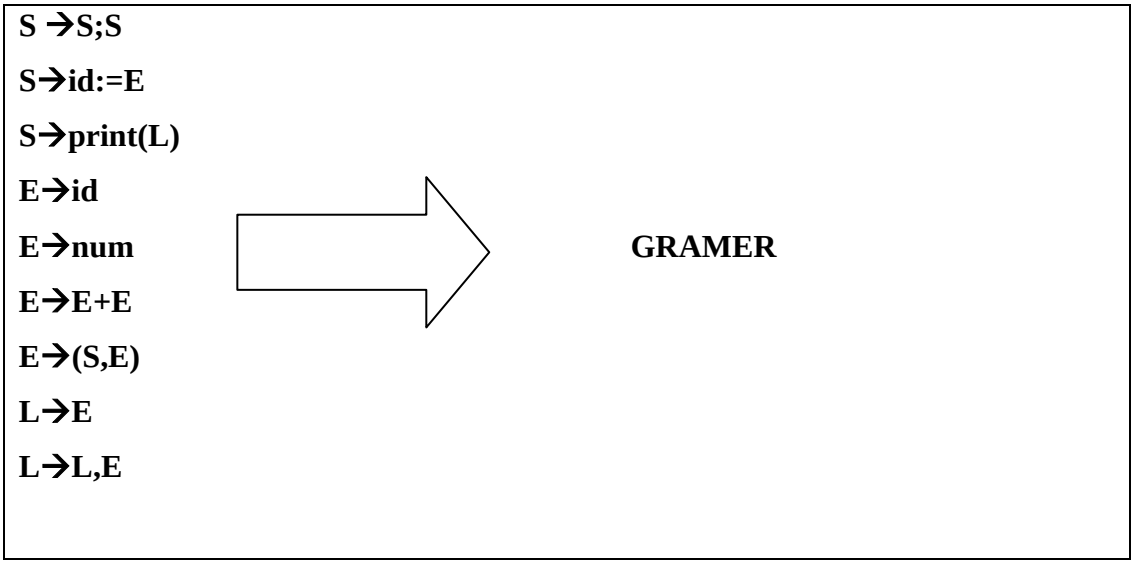
[S , + , F ,) , $]

Ayrıştırıcı bir sonraki kelimeyi belirleyerek (m), bunun 1 numaralı kural ile sonrasında 3 numaralı kural ile eşit olduğuna karar verir. Bu adımın sonunda yığın şu şekli alır:

[F , + , F ,) , $]

[m, + , F ,) , $]

Son iki adımda ayrıştırıcı "m" ve "+" kelimelerini giriş cümlesinden ve yığından siler. ***[F ,) , $]*** Ardından ayrıştırıcı "m" kelimesini 3 numaralı kural ile eşleştirir. Son olarak F ve ")" sembollerini yığından atarak işlemi tamamlar. Sonuçta sadece "$" sembolünün kalması ve giriş cümlesinin sonuna varılması, ayrıştırma sonucunda giriş cümlesinin gramere uygun olduğunu gösterir. Giriş cümlesinin ayrıştırılma işleminden sonra geriye kural dizisi döner. Bu örnek için bu dizi [2, 1, 3, 3] şeklinde olur. S → (S+F) →(F+F) → (m+F) → (m+m)



Şekil 1.8. Dizi ifaderinden oluşan dil grameri

Girdi verisi olarak id:=(id:=num,id+num);print(id+num,id)

S

S:S

id:=E;S

id:=(S;E);S

id:= (id:=E;E);S

id:= (id:=num;E);S

id:= (id:=num,E+E);S

id:= (id:=num,id+E);S

id:= (id:=num,id+num);S

id:= (id:=num,id+num);print(L)

id:= (id:=num,id+num);print(L,E)

id:= (id:=num,id+num);print(E,E)

id:= (id:=num,id+num);print(E+E,E)

id:= (id:=num,id+num);print(id+E,E)

Tablo'nun devamı
id:= (id:=num,id+num);print(id+num,E)
id:= (id:=num, id+num);print(id+num,id)

Şekil 1.9. Sola dayalı üretim

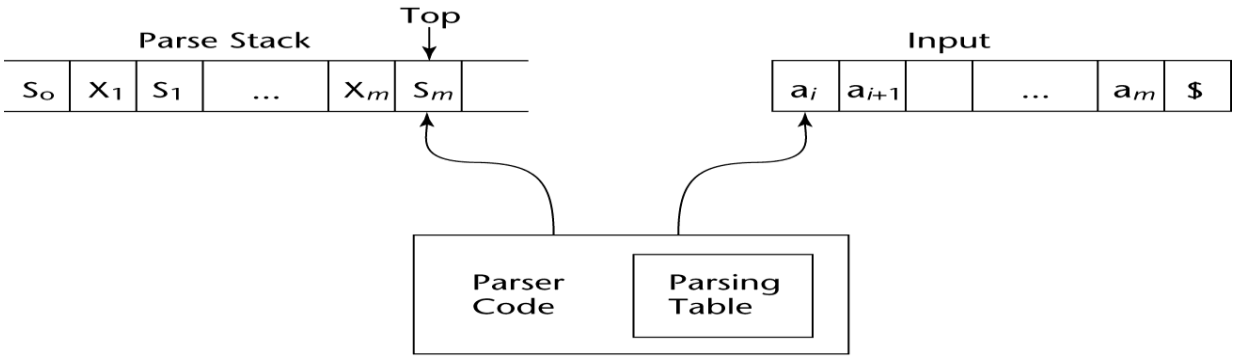
1.4.3. Bottom – Up Ayrıştırma

Ayrıştırıcı kaynak kodu alır ve kod içindeki terminal'lerden nonterminal'lere geçiş yaparak gramer kurallarına uygunluğu kontrol eder.

1.4.3.1. LR Ayrıştırma

LR algoritması girilen cümleler kümesini sol sağ şeklinde okur ve mümkün olan en kısa zamanda syntax errors saptayabilir. LR gramerler sınıfı(LR class of grammars), LL ayrıştırıcıları (LL parsers) tarafından ayrıştırılabilen sınıfın (class) üstkümesidir (superset)

- LR ayrıştırıcılar(LR parsers) şekil sürümlüdür(table driven),bu tablonun iki bileşeni vardır, bir ACTION tablosu ve bir GOTO tablosu
- ACTION tablosu, verilen bir ayrıştırıcı durumu(parser state) ve sonraki jeton(next token) için, ayrıştırıcının(parser) hareketini(action) belirler
- Satırlar(rows) durum(state) adlarıdır; sütunlar(columns) terminallerdir.
- The GOTO tablosu bir indirgeme hareketi(reduction action) yapıldıktan sonra ayrıştırma yığını(parse stack) üzerine hangi durumun(state) konulacağını belirler
- Satırlar(rows) durum(state) adlarıdır; sütunlar(columns) nonterminallerdir



Şekil 1.10. LR Ayrıştırma

Ayrıştırı haraketleri

Başlangıç yapılandırması: (S_0, $a_1 \ldots a_n$\$)

- If ACTION[S_m, a_i] = Shift S, sonraki yapılandırma:

  ($S_0X_1S_1X_2S_2 \ldots X_mS_ma_iS$, $a_{i+1} \ldots a_n$\$)
- If ACTION[S_m, a_i] = Reduce $A \rightarrow \beta$ and S = GOTO[S_{m-r}, A], r = β nın uzunluğu olmak üzere , sonraki yapılandırma: ($S_0X_1S_1X_2S_2 \ldots X_{m-r}S_{m-r}AS$, $a_ia_{i+1} \ldots a_n$\$)
- If ACTION[Sm, ai] = Accept, ayrıştırma(parse) tamamlanmıştır ve hata(error) bulunmamıştır
- If ACTION[Sm, ai] = Error, ayrıştırıcı(ayrıştırıcı) bir hata-işleme rutini(error-handling routine) çağırır

LR ayrıştırma tablosunun üretilme yoluna bağlı olarak birkaç grupta incelenir.Bunlar;

- SLR(Simple LR Parser)
- LALR(Look-Ahead LR Parser)
- CLR(Canonical LR Parser)

LL algoritmasındaki gramer yapısını kullanarak girdi verisini tekrar id:=(id:=num,id+num);print(id+num,id) alalım.

S

S;S

S;print(L)

S;print(L,E)

S;print(L,id)

S;print(E,id)

S;print(E+E,id)

S;print(E+num,id)

S;print(id+num,id)

id:= E;print(id+num,id)

id:=(S,E);print(id+num,id)

id:=(S,E+E);print(id+num,id)

id:=(S,E+num);print(id+num,id)

id:=(S,id+num);print(id+num,id) id:=(id:=E,id+num);print(id+num,id) id:= (id:=num,id+num);print(id+num,id)

Şekil 1.11. Sağa dayalı üretim

1.5. Derleyici derleyiciler

Programcılıkta, bir programlama dilinde yazılmış olan kaynak kodunu başka bir dile (genellikle makine koduna) çeviren yazılımdır. Derleyiciler üst seviyeli bir programlama dilinde yazılmış olan programları makine uyumlu bir dile, genellikle de makine diline çevirme işlemini gerçekleştirmektedirler. Kaynak programın her bir deyimi için, makine dilinde birçok deyim üretilmektedir. Program, derleme işlemi tamamen bittikten sonra çalıştırılabilmektedir. Her bir dilin kendine ait gramer yapısı ve otomatik kod üreten araçlar vardır.

C ,Fortran, Pascal, Objective-C, Java, Ada gibi programların compier compier uygulamalarını geliştirmek için yaygın olarak kullanılan ayrıştırıcı GNU Compiler Collection.

Java compier compier java uygulamalarını geliştirmek için kullanılan ayrıştırıcılar: Antlr, Byacc/Java, Coco/Java, Cup, Generıc Interpreter, Jaccıe, Javacc, Jax, Jay, Sable CC, Pat ve Oops gibidir.Biz bu uygulamada JavaCC ayrıştıcısını kullanacağız.

1.5.1. Bison

Bison ‘un, belli bir paterne sahip girdileri işleyebilen bir ayrıştırıcı program üretebilmesi için, program girdilerinin CFG kullanılarak ifade edilebilmesi gerekir. CFG, giriş diline ait kelimelerin sözdizimsel olarak gruplanması (syntactic grouping) ve her bir sözdizimsel gruplamanın hangi tür kelimelerden oluşacağını belirleyen kuralların (production) tanımlanmasıyla elde edilir. C dilindeki *ifade* (expression) yapıları sözdizimsel gruplamaya bir örnektir. C dilindeki ifade yapılarını tanımlamak için aşağıdaki kurallar kullanılabilir.

Tablo 1.3. Bison Ayrıştırma Yöntemleri

Kural	Örnek
Fonksiyon çağrısı bir ifadedir	*a = sum(b,c);* //sum(b,c) bir ifadedir
Tamsayı bir değer bir ifadedir	*a = b;* //b, bir ifadedir
Bir ifade, "ifade+ifade" formatında olabilir	*a = sum(b,c) + b;* //"sum(b,c) + b" bir ifadedir

1.5.2. Lemon

Lemon, bison ve yacc gibi yazılım geliştirme araçlarına benzer. Ancak bu programlarla uyumlu değildir. Gramerin girdi biçimi kodlama hatalarını önlemek için oldukça farklı bir sözdizim grameri kullanır.

1.5.3. Lex

Lex, sözcüksel analiz yapan bir bilgisayar programıdır ve yaygın olarak yacc derleyici üreticisinde kullanılır.

1.5.4. JavaCC

Java compier compier java uygulamalarında yaygın olarak kullanılan ve java programlama dilinde uygulama ve geliştirmeye imkân veren ayrıştırıcılardır. Bağlam bağlamsız gramere bağlı olarak java dilinden recursive descent ayrıştırma kodu üretir. Bir recursive descent ayrıştırma yazılan gramerin içinde altprogram olarak tanımlanır. Java ile bu uygulamalar dilin grameri hazırlanır sonra yazılan gramer JavaCC gramer dosyasına çevrilir. JavaCC gramer dosyası uygulama için java koduna otomatik olarak çevrilir ve oluşan dosya geliştiricek dosya içinde kullanılır.

JavaCC gramer dosya uzatısı. jj şeklindedir. Bu gramer dosyası 4 bölüme ayrılmaktadır. İlk bölüm gramer dosyasının nasıl bir işlem uygulayacağı kurallar ve özellikler bölümü, ikinci bölümde temel gruplandırma bölümü, üçüncü bölümde kelimeler topluluğu ve son bölümde ise kurallar, işlemler ve Java uygulamaları yapılmaktadır.

1.5.4.1. Kurallar ve özellikler bölümü

JavaCC, gramer dosyasında gerçekleşecek işlemlerin nasıl yönetileceğine dair bazı özellikleri destekler. Örneğin;

options{
LOOKAHEAD = 1;
}

Yukarda verilen kod ayrıştırma esnasında karar verebilmek için en az bir kelime gerektiği belirtilir.

1.5.4.2. Temel Gruplandırma Bölümü

Java kodlarından oluşur ve ayrıştırma esnasında dikkate alınmaz. Burada yazılan kodlar, gramer dosyasının JavaCC ile derlenmesi esnasında kontrol edilmez. Yanlış kodlar yazılmış olsa dahi bu kısım bir sonraki aşamaya değişiklik yapılmadan aktarılır. Bu bölümde PARSER_BEGIN ve PARSER_END arasında kalan temel gruplandırma bölümüdür. Örneğin;

```
PARSER_BEGIN(NestedParser)
import java.io.*;
public class NestedParser {
public static void main(String[] args) {
Reader sr = new StringReader(args[0]);
NestedParser p = new NestedParser(sr);
try {
p.Start();
} catch (ParseException pe) {
pe.printStackTrace();
}
}
}PARSER_END(NestedParser)
```

1.5.4.3. Kelimeler Topluluğu

Terminal yapılarının kullanıldığı bölümdür. Burada tanımlanan terminaller JavaCC'ye ayrıştırma işleminde bulunması gereken anlamlı en kısa kelimelerin neler olduğunu ifade eder. Örneğin ;

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

TOKEN : /* OPERATORS */
{
   < PLUS: "+" >
  | < MINUS: "-" >
  | < MULTIPLY: "*" >
  | < DIVIDE: "/" >
  | < POWER: "^" >
}

TOKEN :
{

   < SIN: "sin">
  |< TAN: "tan">
  |< SEC: "sec">
  |< ARCSIN: "arcsin">
  |< ARCCOT: "arccot">
  |< COSH: "cosh">
  |< SINH: "sinh">

}
```

1.5.4.4. Sözdizimsel Kurallar Bölümü

Gramerde bulunan her bir kural için Java metotlarına benzer kodlar yazılır. Ayrıca blok içinde Java kodları gerek duyulduğu yere, küme parantezi içine alınarak yazılabilir. JavaCC gramer içinde tanımlanmış her bir kural için Java'da bir metot üretir. Metodun adı kuralın başında bulunan non-terminal ile aynı isimdedir. Metot isminden hemen sonra, JavaCC'nin küme parantezleri içinde tanımlamalara izin verdiği bir blok gelir. Bu blok içinde gerçekleştirilen tanımlamalar metot içinde geneldir. Bu yapının ardından non-terminal, terminal ya da non-terminallerle genişletilmiş BNF şeklinde tanımlanır. Örneğin;

```
value=expr()
(<EOF> | <EOL>)            { return value; }
}

Exp expr() : {
Exp x;
Exp y;
}
{
x=term()
(
<PLUS> y=term()    { x = new Plus(x, y); }
| <MINUS> y=term()   { x = new Minus(x, y); }      )*     { return x; }
```

1.5.5. JTB

JTB, JavaCC ayrıştırıcı üreteci ile çalışan otomatik bir sözdizim ağaç üretecidir. JavaCC'de geliştirilmiş gramer dosyasını argüman olarak alır ve içinde sözdizimsel ağacı oluşturabilecek şekilde yeniden yapılandırır. JTB gramer dosyasını işledikten sonra aşağıdaki yapıları üretir:

• jtb.out.jj dosyası; gramer dosyası ile aynı yapıda olup ayrıştırma esnasında sözdizim ağacını oluşturabilmek için anlamsal işlemler eklenir. Bu dosya üretildikten sonra diğer dosyaya gerek kalmaz.

• syntaxtree klasörü; gramer içinde tanımlı tüm kurallar için oluşturulan sınıfları içerir. JTB ile gramer dosyasında yer alan kural sayısı kadar sınıf üretilir. Bu sınıflar

kuralın sol tarafında yer alan isim ile aynı isme sahiptir. Bu sınıflar bir kez üretildikten sonra yeniden düzenlemeye gerek kalmaz.

JTB ile kural sınıflarından başka otomatik olarak üretilen başka sınıflar da vardır. Bu sınıflar:

o Node Sınıfı; tüm ağaç düğümlerinin üretildiği temel düğüm arayüzüdür.

o NodeListInterface Sınıfı; NodeList, NodeListOptional ve NodeSequence sınıflarının gerçeklenmesi için gerekli arayüzdür.

o NodeChoie Sınıfı; (A | B) durumunda kullanılır.

o NodeList Sınıfı; (A)+ durumunda kullanılır.

o NodeListOptional Sınıfı; (A)* durumunda kullanılır.

o NodeOptional Sınıfı; (A)? Durumunda kullanılır.

o NodeSequence Sınıfı; "…" (A)* gibi durumlarda kullanılır.

o NodeToken Sınıfı; "…" durumunda kullanılır.

Visitor klasörü; sözdizim ağacında gezebilmek için oluşturulmuş arayüzleri ve sınıfları içermektedir. JTB otomatik olarak iki arayüz oluşturur. Bunlar Visitor ve ObjectVisitor arayüz sınıfları olup DepthFirstVisitor ve ObjectDepthFirst sınıfları sırasıyla bu arayüzlerden oluşur.

Visitor arayüzü gramer içindeki her bir kural için bir fonksiyon tanımlanır. Visit() adı verilen bu fonksiyonların her biri daha önceden syntax tree klasörü içerisinde tanımlanmış sınıfların her birini argüman olarak alır. Aşırı yüklenmiş bu fonksiyon aldığı argümana göre ilgili kodu çalıştırır. Visit() fonksiyonu ağacın hangi düğümünde bulunulduğu hakkında bilgi verir. Bu sayede ağacın hangi düğümünde ne yapılacağına karar verilerek bu fonksiyonlar kullanıcının isteğine göre doldurulabilir. Visit() fonksiyonu altında accept() fonksiyonu çağırılarak ağaçta bir alt düğüme geçilmesi sağlanır. Ağacın düğümlerinde hatalı bir alt düğüme geçilmeye çalışılırsa accept() fonksiyonlarından geriye dönülmeden hata üretilir ve işlemden çıkılır. Gramer dosyası JTB ile yeniden yapılandırılarak kaynak kodun değerlendirildikten sonra ağaç biçiminde bir veri yapısının geriye döndürülmesi sağlanır. Böylece ihtiyaç duyulan anlamsal kontroller ağaç üzerinden gidilerek kolayca yapılabilir.

1.6. JavaCC ile Ayrıştırıcı Üretimi

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. Terminal'ler < ve > simgeleri arasında belirtilirken, kurallar programlama dili fonksiyon ya da metotlarına benzer yapıda bildirilir. Burada metotlara ilgili kural nonterminali ile uyumlu isimler verilir. Nonterminal ya da başka bir isimli metot biçimde bildirilir. Tablo 1.4 de gramer kuralına örnek verilmiştir.

Tablo 1.4. Kurallar Bildirimi

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

1.7. Sözdizim Sınıfları

Sözdizim sınıflarının dil gerçekleştirmenin en yaygın kısmıdır. JavaCC dilinin gramer yapısı incelenmiş ve karmaşık kodlamayı zorlaştırmayacak şekilde gramer sınıfları üretilmiş ve bir alt sınıfa miras kalmıştır.

1.7.1. Sözdizim (Nesne) Ağacı

Sözdizim ağacının oluşturulmasındaki amaç daha sonraki aşamalar olan anlamsal analiz ve kod yorumlama işlemlerinin gerçekleştirilmesinde uygun bir yapı oluşturabilmektir.

Bu aşamada JavaCC'nin yanı sıra JTB mekanizması devreye girmektedir. JTB'nin kullanılması ile birlikte gramer içinde tanımlanan tüm non-terminal ifadeler birer sınıf şekline dönüştürülmektedir. Böylece oluşturulan sözdizim ağacının her bir düğümünde daha

önceden tanımlı bir nesne yer alacaktır. Bu nesneler düğümlerin kontrolünü daha kolay hale getirecektir.

1.7.2. Sözdizim (Nesne) Ağacı 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 1.5'de özetlenmiştir.

Tablo 1.5. Yöntem üstünlükleri

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

1.7.2.1. 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. Şekil 1.10'da tanımlanan ifadenin nesne ağacını ve değerlendirmenin yapılacağı exp değerini parametre olarak almaktadır.

Tablo 1.6. instanceof işleci ile değerlendirme

```
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));
```

1.7.2.2. Sözdizim Sınıfına 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. Şekil1.11'de değerlendirmenin yapılacağı x değerini parametre olarak alan eval() metodu tanımlamaları bazı sözdizim sınıfları için verilmiştir.

Tablo 1.7. Sözdizim sınıflarına eval() metotlarının eklemesi

```
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));
}}
```

```
(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
```

class Euler extends Exp {
public Exp exp;
public Euler(Exp e) {
exp = e;}
public double eval(double x) {
return Math.pow(Math.E,
exp.eval(x));}}
	{
	public Var() { }
	public double eval(double x) {
	return x;
	}
	}

1.7.2.3. 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 1.8. Sözdizim sınıflarına accept() metotlarının eklemesi

abstract class Exp {
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); }}
	Devamı)
	class Num extends Exp
	{ public double num;
	public Num(double n) {
	num = n;
	}
	public double accept(Visitor v) {
	return v.visit(this);
	}}

Tablo 1.9. Sözdizim ağacı için bir Visitor arayüzü ve gerçeklemesi

// Visitor.java
public interface Visitor {
public double visit(Exp exp);
public double visit(Euler exp);
public double visit(Log exp);
public double visit(Sqrt exp);
}
// EvalVisitor.java
public class EvalVisitor
implements Visitor {
double x;
public EvalVisitor(double a) {
x = a;
}
	(Devamı)
	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);
	}
	public double visit(Var exp) {
	return x;
	}
	}

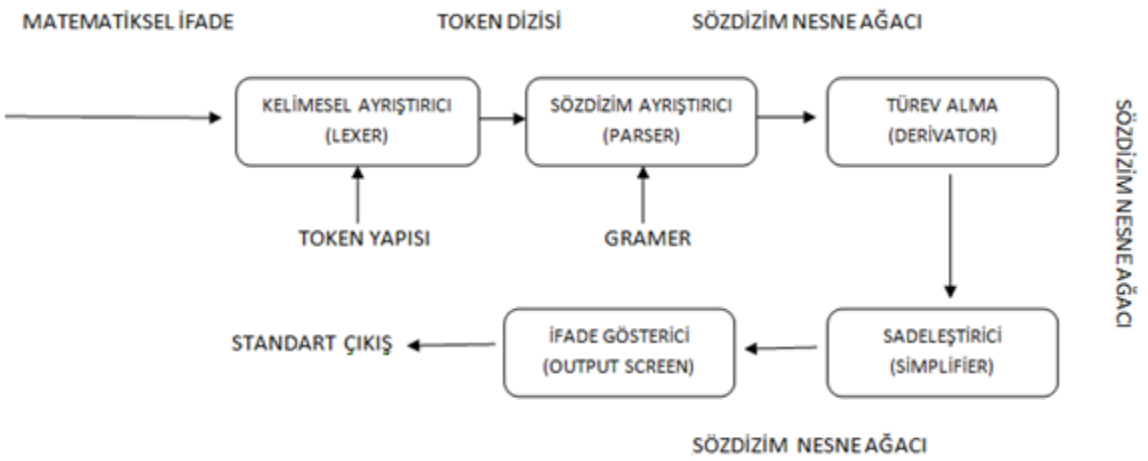
2. YAPILAN ÇALIŞMALAR, BULGULAR VE DEĞERLENDİRME

2.1. Giriş

Biçimsel bir dilde geliştirilen kaynak verinin sözdizim denetiminin yapılması ve yorumlanabilir üst düzey veri yapılarına dönüştürülmesi süreçlerinde, çok farklı programlama dillerinde kod üretebilen örnekleri bulunan derleyici derleyici araçlarından kullanımı önemli bir kodlama kolaylığı sağlar. Bu çalışmada, otomatik kod üretim araçları olarak da bilinen derleyici derleyicilerin simgesel hesaplama uygulamalarının geliştirilmesinde nasıl kullanılacağı örnek bir uygulama üzerinden gösterilmiştir. Matematiksel ifadelerin türevlerinin hesaplanmasını ve sadeleştirilmesini kapsayan uygulamada, ifadelerin bileşenlerine ayrılmasından sonuç ifadesinin üretimine kadar bütün kodlama aşamaları ayrıntılı olarak tartışılmış ve gerçeklenmiştir.

Bu çalışmaya konu olan simgesel türev hesaplama uygulaması Java dilinde kaynak kod üretebilen JavaCC aracı yardımıyla geliştirilmiştir. Matematiksel ifadelerin sözdizimsel ve anlamsal yapısına uygun bir CFG (context-free grammer) grameri yazılarak LL(k) gramerine dönüştürülmüş, ifade içerisinde yer alabilecek işleç ve fonksiyonları temsil etmek üzere sözdizim sınıfları tanımlanmıştır. Uygulama genel olarak matematiksel ifadeyi okuyarak token dizisine dönüştüren bir token üretici, token dizisinden ifadenin sözdizimini kontrol ederek sözdizim ağacı üreten bir ayrıştırıcı, sözdizim ağacı üzerinde çalışan türev alıcı, sadeleştirici ve ifade gösterici bileşenlerinden meydana gelir. Simgesel türev hesaplama uygulamasının genel mimarisi Şekil 2.1' de gösterilmiştir.

Aşağıdaki bölümlerde bu bileşenlerin daha ayrıntılı tanıtımı yapılmıştır.



Şekil 2.1. Simgesel türev hesaplama uygulamasının mimarisi

2.2. Gramer Yazımı

Bir gramer biçimsel bir dilde karakter dizileri ya da kelimeler oluşturabilen kurallar seti olarak ifade edilir. Kurallar, dil alfabesini kullanarak, dilin sözdizimi ile uyumlu kelimelerin nasıl üretileceğini belirtir. Gramerlerin birçok türü vardır; içerikten bağımsız gramerler (CFG), düzenli gramerler, analitik gramerler, vs. Bu bölümde matematiksel ifadelerin sözdizim yapısına uygun bir CFG grameri tanımlanarak işleç anlamlarını içerecek biçimde LL(1) gramerine dönüşümü gösterilmiştir.

2.2.1. CFG Düzenleme

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. Aritmetik ifadeler birçok işleç ve matematiksel fonksiyon kullanılarak yazılabilir. Tablo 2.1' 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 anılacaktır.

Tablo 2.1. Matematiksel ifadeler için bir CFG 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"]	

2.2.2. İşleç Anlamlarını Ekleme

Tablo 2. 1'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 özyinelemeli 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 2.1'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 2.2'de özetlenmiştir.

Tablo 2.2. İşleç öncelik düzeyleri 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 2. 2'de görüldüğü gibi işleçlerin 4 farklı öncelik düzeyi ve 2 farklı birleşme yönü vardır. Bu nedenle, Tablo2.1'deki gramer kurallarının konum farklılığı ve doğru özyineleme yönü oluşturulacak biçimde yeniden düzenlenmesi gerekir. Tablo 2.3'te gösterilen gramerde işleç öncelik düzeyleri ve birleşme yönleri göz önüne alınmıştır.

Tablo 2.3. İşleç öncelik düzeyleri ve birleşme yönlerini içeren CFG grameri

Kural	Kural
E → E "+" T | E "-" T | T	R → "log" "(" E ")"
T → T "*" F | T "/" F | F	R → "sin" "(" E ")"
F → "+" P | "-" P | P	R → "cos" "(" E ")"
P → R "^" P | R	R → (D)+ ("." (D)+)?
R → x	R → "(" E ")"
R → "exp" "(" E ")"	D → ["0"-"9"]
R → "ln" "(" E ")"	

2.3. LL(1) Gramerine Dönüşüm

Matematiksel ifadeler JavaCC ile üretilen ayrıştırıcılar tarafından analiz edileceğinden Tablo 2.3'teki gramer LL(k) karakterli bir gramere dönüştürülmelidir. Genel olarak, LL(k) gramerlerinde girdi verisinden en fazla k terminal okuyarak kural seçimi yapılabilir (k>0). Bu yol içerisinde bir LL(1) grameri aşağıdaki özelliklere sahip olacaktır.

- Bir kuralın her bir alternatifi farklı bir terminal ile başlar.
- NULL durumu “Evet” olan (yani, hiçbir şey üretmeyebilen) kuralların FIRST ve FOLLOW setleri ortak terminal içermez.
- Kurallar soldan özyinelemeli değildir.

Tablo 2.3'te verilen gramer yeni kurallar eklenerek kolaylıkla bir LL(1) gramerine dönüştürülebilir. Böyle bir gramer Tablo 4'te gösterilmiştir; $ simgesi girdi verisinin sonunu işaretler. Tablo 2.4 aynı zamanda bu gramer için kodlanacak bir ayrıştırıcıda bulunması gereken metotların isimlerini de içermektedir.

Tablo 2.4. Matematiksel ifadeler için işleç anlamlarını da içeren bir LL(1) grameri

Kural	Metot	Kural	Metot
S → E $	parse()	R → "x"	element()
E → T E’	expr()	R → "exp" "(" E ")"	
E’ → ("+" | "-") T E’		R → "ln" "(" E ")"	
E’ →		R → "log" "(" E ")"	
T → F T’	term()	R → "sin" "(" E ")"	
T’ → ("*" | "/") F T’		R → "cos" "(" E "," E ")"	
T’ →		R → (D)+ ("." (D)+)?	
F → ("+" | "-")? P	unary()	R → "(" E ")"	
P → R ("^" P)?	power()	D → ["0"-"9"]	

Tablo 2.4 ‘de verilen LL(1) gramerinin FIRST ve FOLLOW seti aşağıdaki şekilde oluşturulabilir. Tabloda görüldüğü gibi, her bir kural farklı bir terminal ile başlar ve Null durumu “Evet” olan kuralların FIRST ve FOLLOW setleri farklı terminaller içerir.

Tablo 2.5. FIRST ve FOLLOW setleri

Kural	Null Durumu	FIRST	FOLLOW
S	Hayır	“+”, “x”, “exp”, “ln”, “log”, “sin”, “cos”, “(“	
E	Hayır	“+”, “x”, “exp”, “ln”, “log”, “sin”, “cos”, “(“	
E’	Evet	“+”,”-“	“)”
T	Hayır	“+”, “x”, “exp”, “ln”, “log”, “sin”, “cos”, ”(“	“)”
T’	Evet	“*”,”/”	
F	Hayır	“+”, “x”, “exp”, “ln”, “log”, “sin”, “cos”, “(“	
P	Hayır	“+”, “x”, “exp”, “ln”, “log”, “sin”, “cos”, “(“	
R	Hayır	“+”, “exp”, “ln”, “log”, “sin”, “cos”, “(“	
D	Hayır		

2.4. Token Üreteci

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 (ya da kelimeyi) temsil edebilmek için aynı token kullanılır. Örneğin, bütün sayılar ("5" ve "0.01" gibi) NUMBER token'ı ile temsil edilir. Geliştirilen LL(1) gramerinin içerdiği kelimeleri temsil edecek token'lar javaCC'de Tablo 2.6’daki gibi tanımlanabilir. Burada token üretecine ait bir tanımlama bloğu olmadığına dikkat ediniz.

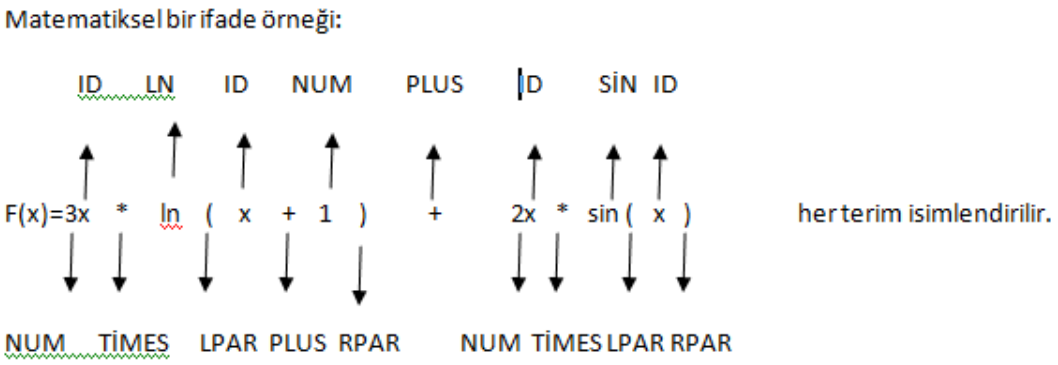
Tablo 2.6. Gramer terminal'leri için JavaCC token tanımlamaları

```
TOKEN :{
< SIN: "sin">
|< COS: "cos">
```

```
|< TAN: "tan">
|< COT: "cot">
|< SEC: "sec">
}
```

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

Verilen bir matematiksel ifadenin notasyonu örneği:
F(x)=3x*ln(x+1)+2x*sinx ifadesinin token dizisindeki açılımı aşağıdaki gibidir.



Şekil 2.2. Matematiksel ifadenin token dizisindeki karşıtı

2.5. Ayrıştırıcılar

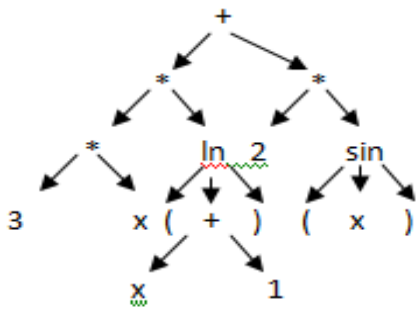
Bir ayrıştırıcı (parser) girdi verilerinin gramer kuralları ile üretilebilirliğini token dizisi üzerinden analiz eder. Böyle bir analizde girdi verisi sadece doğru sözdizimli olup olmadığı yönünden incelenir, verinin değerlendirilebilirliği (ya da yorumlanabilirliği) konusu aydınlatılmaz. Bu durum sözdizimsel 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 2,3'te geliştirilen LL(1) gramerine göre sözdizim analizi yapan bir ayrıştırıcının JavaCC tanımlaması Tablo 2. 7'da verilmiştir. <EOL> ve <EOF> token'ları sırasıyla satır ve girdi sonu karakterlerini temsil eder.

Tablo 2.7. Ayrıştırıcı için JavaCC tanımlaması

```
void parse() : { }
{ expr() (<EOF> | <EOL>)
}
void expr() : { }
{ term() ( <PLUS> term()        //SOLDAN
          | <MINUS> term() )*
}
```

Yukarıda örneği verilen matematiksel ifadenin gramer yapısına göre soyut Sözdizim Ağacı ise aşağıda verilmiştir.



Şekil 2.3. Matematiksel ifadenin ağaç yapısı

2.6. 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. Bazı sözdizim sınıflarının tanımlaması Tablo 2. 8'de gösterilmiştir

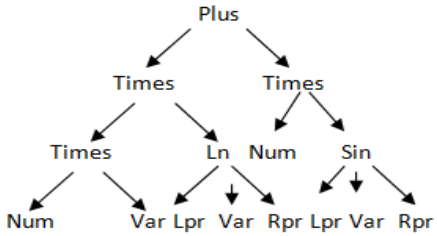
Tablo 2.8. Sözdizim sınıfları

```
abstract class Exp {
  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;
```



Şekil 2.4. Matematiksel ifadenin sözdizim sınıfı

2.7. Sözdizim (Nesne) Ağacı

Bir sözdizim ağacı (ya da 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 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. İfadenin sözdiziminin doğru olup olmadığı o dil, üreten JavaCC üreticisi inceleyek tespit edilebilir. Tablo 2.9'de bir matematiksel ifadeyi temsil edecek nesne ağacını üretmek için, Java ifadelerinin Tablo 2. 6'daki gramer tanımlamasına nasıl ekleneceği gösterilmiştir

Tablo 2.9. Sözdizim nesne ağacının üretilmesi

```
Exp parse() : {
Exp value;
                }
                {
value=expr()(<EOF> | <EOL>)          { return value; }}
Exp expr() : {
```

```
    Exp x;
    Exp y;
                }
                {
    x=term()
    (
      <PLUS> y=term()    { x = new Plus(x, y); }
    | <MINUS> y=term()   { x = new Minus(x, y); }
    )*
    { return x; }
  }
```

JavaCC'de token tanımlamaları ile ayrıştırıcı tanımlaması. jj uzantılı aynı dosya içerisine kaydedilir. Bu dosya içerisinde JavaCC'ye özel bazı çevre değişkenlerine değer ataması da yapılabilir. Tablo 2.6 ve Tablo 2.9'deki tanımlamalarla oluşturulan Derivation. jj dosyası Tablo 2.10.'da verilmiştir.

Tablo 2.10. Derivation.jj dosyası

```
options {
   STATIC = true;      // Ayrıştırıcı metotlarını static yapar
   LOOKAHEAD = 1;          // Bir token ile kural seçimi yapar
}
PARSER_BEGIN(Parser)
public class Parser { }
PARSER_END(Parser)
// Token tanımlamaları
...
```

```
// Ayrıştırıcı tanımlaması
...
```

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)))))
```

2.8. Türev Alıcı

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; Java *instanceof* işleci, sözdizim sınıflarına metot ekleme ve Visitor tasarım deseni. Bu çalışmada türev alma, sadeleştirme ve sadeleştirilen ifadeyi gösterme işlemlerinin her biri için farklı bir Visitor deseni gerçeklenmiştir.

Nesne ağacı düğümlerini değerlendirmek için Tablo 2.11'da tanımlanan DeriveVisitor arayüzü ile bu arayüzü gerçekleyen deriver sınıfı kullanılmıştır. Deriver sınıfının metotları ile nesne ağacına erişilerek türev hesaplanır ve yeni bir nesne ağacı üretilir.

Tablo 2.11. Visitor arayüzü (DeriveVisitor. java)

```
public interface DeriveVisitor {
  public Exp visit(Exp exp);
  public Exp visit(Plus exp);
  public Exp visit(Minus exp);
  public Exp visit(Times exp);
  public Exp visit(Divide exp);
  public Exp visit(Power exp);
  public Exp visit(Sin exp);
```

```
public Exp visit(Cos exp);
public Exp visit(Tan exp);
public Exp visit(Cot exp);
public Exp visit(Sec exp);
public Exp visit(Csc exp);
public Exp visit(Sqrt exp);
public Exp visit(Ln exp);
```

Tablo 2.12. Türev alıcı sınıfı (DeriverExp. java)

```
public class DeriveExp implements DeriveVisitor {
public Exp visit(Ln exp) {
Exp a = exp.exp.accept(this);
return new Divide(a, exp.exp);}
public Exp visit( Divide exp) {
Exp a = (Exp)(exp.exp1.accept(this));
Exp b = (Exp)(exp.exp2.accept(this));
return new Divide (new Minus(new Times(a,exp.exp2),new  Ti-
mes(exp.exp1,b)),new Times(exp.exp2,new Num(2.0)));
```

2.9. Sadeleştirici

Matematiksel ifadelerin türevi sadeleştirilmesi gereken birçok terim içerebilmektedir. Bu nedenle, Deriver sınıfının metotları ile üretilen nesne ağacının düğümlerine yeniden erişilerek sadeleştirilebilir ifadelerin değerlendirilme ve en sade halini olması gerekmektedir. Örneğin 2+x ifadesinin türevini aldığımızda ve sadeleştirme methodu kullanmadığımızda nesne ağacından matematiksel ifadeye dönüşüm 0+1 olarak elde ederiz bu da yanlış bir ifade olur. Buna istinaden ürettiğimiz sadeleştirme ağacı ise bize doğru sonuca ulaş-

mamızı sağlar ve doğru sonucun 1 olduğunu görürüz. Tablo 2.14'de sadeleştirilmeye ihtiyaç duyan bazı ifade örnekleri gösterilmiştir.

Nesne ağacı düğümlerini değerlendirmek için Tablo 2.13'de tanımlanan SimpVisitor arayüzü ile bu arayüzü gerçekleyen sadeleştirme sınıfı kullanılmıştır.

Tablo 2.13. Visitor arayüzü (SimpVisitor. java)

```
public interface SimpVisitor {
public Exp visit(Exp exp);
public Exp visit(Plus exp);
public Exp visit(Times exp);
public Exp visit(Power exp);
public Exp visit(Divide exp);
public Exp visit(Minus exp);
}
```

Tablo 2.14. Bazı ifadeler ve özdeşleri

İfade	Sadeleştirme sonucu
new Power(l, new Number(0.0))	new Number(1.0)
new Power(l, new Number(1.0))	l
new Power(new Number(1.0), r)	new Number(1.0)
new Power(new Number(0.0), r)	new Number(0.0)

Tablo 2.14'deki özdeşliklerden yararlanarak, Tablo 2.13'da verilen Visitor arayüzünden yeni bir sınıf gerçeklenir ve türev alıcının üreteceği nesne ağacı düğümleri üzerinde yeni visit() metotları tanımlanır. Bu metotlarla erişilen düğümün sözdizim sınıfı türüne göre sadeleştirme yapılabilir. Sadeleştirici sınıfı ve bazı metotları Tablo 2.15'de verilmiştir.

Tablo 2.15. Sadeleştirici sınıfı ve bazı metotlarının tanımlaması (SimplifyExp.java)

```
public class SimplifyExp implements  SimpVisitor{
public Exp visit(Exp exp) { return (Exp) exp.accept(this);}
public Exp visit(Plus exp) {
Exp a = exp.accept(this);
Exp b = exp.accept(this);
if (a instanceof Num){
if (((Num)a).num==0.0) return b;}
if (b instanceof Num)
{ if (((Num)b).num==0.0)
return a;}return exp;}
```

2.10. İfade Gösterici

Nesne ağacının düğümleri sadeleştirici metotlarıyla değerlendirildikten sonra ortaya çıkan ifadenin matematiksel notasyonda gösterim aşamasına geçilir. Tablo 2.15'de nesne ağacından matematiksel ifadeye dönüşüm yapan bir ifade gösterici sınıfı ve metotları verilmiştir. Nesne ağacı düğümlerini değerlendirmek için Tablo 2.16'de tanımlanan PrintVisitor arayüzü ile bu arayüzü gerçekleyen ifade gösterici sınıfı kullanılmıştır.

Tablo 2.16. Visitor arayüzü (PrintVisitor. java)

```
public interface PrintVisitor {
public String visit(Exp exp);
public String visit(Plus exp);
public String visit(Var exp);
public String visit(Euler exp);}
```

Tablo 2.17. İfade gösterici sınıfı ve bazı metotlarının tanımlaması (PrintExp.java)

```
public class Printer implements Visitor {
public Object visit(Times exp) {
String a = (String)(exp.exp1.accept(this));
String b = (String)(exp.exp2.accept(this));
return ("(" + a + ")*(" + b + ")");}
public Object visit(Minus exp) {
String a = (String)(exp.exp1.accept(this));
String b = (String)(exp.exp2.accept(this));
```

2.11. Bileşenlerin Entegrasyonu

Yukarıda tanıtılan her bir bileşen Tablo 2.18'deki gibi entegre edilerek bir simgesel türev uygulaması geliştirilmiştir. Uygulama hem dosya içerisinden hem de komut satırı üzerinden matematiksel ifadeyi okuyabilmektedir. Matematiksel ifadede dönüşüm nesne ağacı üzerinden yapılmaktadır.

Tablo 2.18. Simgesel türev uygulaması (Deriver.java)

```
public class Deriver {
public static void main(String[] args) {
Tablo'nun devamı
 Derivation parser = new Derivation(System.in);
 try {Exp exp = parser.parse();
 exp = new DeriveExp().visit(exp);
  exp = new SimplifyExp().visit(exp);
 String fexp = new PrintExp().visit(exp);
System.out.println(fexp);
} catch(ParseException e) {  e.printStackTrace(); }}}
```

JavaCC yardımıyla ayrıştırıcının üretimi ve türev uygulamasının derlemesi sırasıyla javacc ve javac araçları ile yapılır. Aşağıdaki örnekte matematiksel ifade uygulamaya komut satırı üzerinden verilmiştir.

```
$> javacc Parser.jj
$> java *.java
$> java ExprDerive
x^3 +x*sin(x+1)
(3.0)*(x^2.0)+sin(x+1.0)+(x)*(cos(x+1.0))
```

Bu matematiksel ifade üzerinde simgesel türev uygulamasının her bir bileşeninin hangi verileri üreteceği aşağıda gösterilmiştir. İlk önce ayrıştırıcı metotları aşağıdaki nesne ağacını üretir.

```
tree = new Plus(new Power(new Var(), new Number(3.0)), new Times(new Var(), new Sin(new Plus(new Var(), new Number(1.0)))))
```

Bu nesne ağacından türev alıcı sınıfı metotları ile diğer bir nesne ağacı üretilir.

```
tree2 = new Plus(new Times(new Number(3.0), new Power(new Var(), new Number(2.0))), new Plus(new Times(new Number(1.0), new Sin(new Plus(new Var(), new Number(1.0)))), new Times(new Var(), new Times(new Plus(new Number(1.0), new Number(0.0)), new Cos(new Plus(new Var(), new Number(1.0)))))))
```

Sadeleştirici sınıf metotları ile nesne ağacı yeniden dönüştürülür.

```
tree3 = new Plus(new Times(new Number(3.0), new Power(new Var(), new Number(2.0))), new Plus(new Sin(new Plus(new Var(), new Number(1.0))), new Times(new Var(), new Times(new Cos(new Plus(new Var(), new Number(1.0)))))))
```

İfade üretici sınıf metotlarıyla matematiksel notasyonda türev ifadesine dönüşüm yapılır.

```
str = "(3)*(x^2)+sin(x+1)+(x)*(cos(x+1))"
```

3. SONUÇ

Bu bildiride otomatik kod üretim araçları ile bir simgesel türev hesaplama uygulamasının nasıl geliştirileceği gösterilmiştir. Uygulamanın matematiksel ifadenin okunarak nesne ağacına dönüştürülmesi aşamalarında LL(k) ayrıştırıcıları için Java dilinde otomatik kaynak kod üretebilen JavaCC aracı kullanılmıştır. Uygulamada önce çeşitli fonksiyonların yer alabildiği matematiksel ifadeler için BNF notasyonunda bir LL(1) grameri yazılarak JavaCC formatında tanımlaması yapılmıştır. Bu gramer ile üretilebilecek bütün matematiksel ifadeleri nesne ağacı biçiminde temsil edebilen bir ayrıştırıcı üretilmiştir. Nesne ağacı üzerinde tanımlanan türev alıcı, sadeleştirici ve ifade gösterici bileşenleri ile ifadenin simgesel türevi hesaplanmıştır.

Otomatik kod üretim araçlarını belirli bir biçime sahip karakter dizlerini değerlendiren herhangi bir uygulamanın geliştirilmesinde kullanmak mümkündür. Bildiride gösterilen yöntem ile matematiğin kullanıldığı bütün mühendislik dallarındaki çeşitli denklem sistemleri, diferansiyel denklemleri ve belirsiz entegrallerin simgesel çözümleri yapılabilir ve çözüme götüren her bir hesaplama adımı gösterilebilir. Günümüzde kullanılan genel ve özel amaçlı pek çok simgesel hesaplama sistemi ara adımları göstermeyip doğrudan hesaplama sonuçlarını verirler.

4. ÖNERİLER

1. Üzerinde çalıştığımız yorumlayıcıda kısıtladığımız kurallar, ilerleyen çalışmalarda gramere eklenerek tam bir yorumlayıcı elde edilebilir.
2. Programın hızlanması için mevcut sözdizim ağaç üretimi kaldırılmalıdır.
3. Çalışmamız geliştirilmesiyle çözümü zor, karmaşık problemlerin çözümünü kolaylaştırır.

5. KAYNAKLAR

1. J.V.Z. Gathen, J. Gerhard, *Modern Computer Algebra*, Cambridge University Pres, Cambridge, 1999.
2. W. Decker, Some introductory remarks on computer algebra, Proceedings of the third European Congress of Mathematics, Barcelona, 2000.
3. M. Bronstein, *Symbolic Integration I – Transcendental Functions*, Springer, Berlin, 1997.
4. R. Risch, The solution of the problem of integration in finite terms, Bull. A.M.S. 76 (1970) 605-608.
5. T. J. Parr and R. W. Quong. ANTLR: A predicated-LL(k) parser generator. Softwa-re Practice and Experience, 25(7) (1995) 789–810.
6. JavaCC, https://javacc.dev.java.net/
7. E. M. Gagnon and L. J. Hendren. SableCC, an object-oriented compiler framework. In Technology of Object-Oriented Languages and Systems, pag. 140–154. IEEE Computer Society, 1998.
8. JTB: Java Tree Builder, http://compilers.cs.ucla.edu/jtb/.
9. E. Berk. JLex:A lexical analyzer generator for Java(TM), http://www.cs.princeton.edu/~appel/modern/java/JLex/
10. K. Gerwin. JFlex User's Manual, July 2005.

6. EKLER

Ek-1. JavaCC token tanımlamaları

Bir programlama diline ait kaynak kod içinde bulunan karakterler kümesini, anlamlı en kısa kelimeler (token) kümesine dönüştürme aşamasıdır.

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

TOKEN : /* OPERATORS */
{
< PLUS: "+" >
| < MINUS: "-" >
| < MULTIPLY: "*" >
| < DIVIDE: "/" >
| < POWER: "^" >
}
TOKEN:{
|< CSC: "csc">
|< ARCSIN: "arcsin">
|< ARCCOS: "arccos">
|< ARCTAN: "arctan">
|< ARCCOT: "arccot">
|< COSH: "cosh">
|< SINH: "sinh">
|< TANH: "tanh">
|< COTH: "coth">
|< LN: "ln">
|< LOG: "log"
```

Ek-2. JavaCC tanımlamaları

Java ifadelerinin gramer tanımlamasına nasıl ekleneceği gösterilmiştir.

```
void term() : { }
{ unary() ( <TIMES> unary()
            | <DIVIDE> unary())*
}
void unary() : { }
{   <PLUS> power()
  | <MINUS> power()
  | power()
}
| <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>}
```

Ek-3. JavaCC tanımlamaları

Sözdizim nesne ağacının üretilmesi gösterimi aşağıdaki kod bloğunda verilmiştir.

```
Exp term() : {
   Exp x;
   Exp y;
}
{
   x=power()
   (
     <MULTIPLY> y=power()    { x = new Times(x, y); }
   | <DIVIDE> y=power()      { x = new Divide(x, y); }
   )*
   { return x; }
}

Exp power() : {
   Exp x;
   Exp y;
}
{
   x=unary()
   (
     <POWER> y=unary()    { x = new Power(x, y); }
   )*
   { return x; }
}

Exp unary() : {
   Exp value;
}
```

```
{
    <MINUS> value=element()  { return (new Times(new Num(-1),value)); }
  | value=element()          { return value; }
}

Exp element() : {
  Token t;
  Exp value;
}
{
    t=<NUMBER>        { return (new Num(Double.parseDouble(t.image))); }
  | "x"                 { return (new Var()); }
        | "e"                   { return (new Euler(new Num(1.0)));}
  | “(“ value=expr() “)” { return new Euler(value); }
  | "(" value=expr() ")"          { return value; }
  | <SIN> "(" value=expr() ")"      { return (new Sin(value));}
  | <COS> "(" value=expr() ")"      { return (new Cos(value));}
  | <TAN> "(" value=expr() ")"      { return (new Tan(value));}
  | <COT> "(" value=expr() ")"      { return (new Cot(value));}
  | <SEC> "(" value=expr() ")"      { return (new Sec(value));}
  | <CSC> "(" value=expr() ")"      { return (new Csc(value));}
  | <ARCSIN> "(" value=expr() ")"   { return (new Arcsin(value));}
  | <ARCCOS> "(" value=expr() ")"   { return (new Arccos(value));}
  | <ARCTAN> "(" value=expr() ")"   { return (new Arctan(value));}
  | <ARCCOT> "(" value=expr() ")"   { return (new Arccot(value));}
  | <COSH> "(" value=expr() ")"     { return (new Cosh(value));}
  | <SINH> "(" value=expr() ")"     { return (new Sinh(value));}
  | <TANH> "(" value=expr() ")"     { return (new Tanh(value));}
  | <COTH> "(" value=expr() ")"     { return (new Coth(value));}
  | <LN>   "(" value=expr() ")"     { return (new Ln(value));}
  | <LOG> "(" value=expr() ")"      { return (new Ln(value));}}
```

Ek-4.Sözdizim sınıfları

Matematiksel ifadelerin türevlerini hesaplayabilmek için sözdizimsel ve anlamsal yapısına uygun bir CFG grameri geliştirilmiş olup işleç ve fonksiyonları temsil etmek üzere sözdizim sınıfları tanımlanmıştır. Kod bloğu aşağıda verilmiştir.

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

  public String accept(PrintVisitor v) {
   return v.visit(this);
  }

  public Exp accept(DeriveVisitor v) {
   return v.visit(this);
  }

  public Exp accept(SimpVisitor v) {
   return v.visit(this);
  }
}

class Minus extends Exp {
  public Exp exp1, exp2;

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

```
public double accept(EvalVisitor v) {
  return v.visit(this);
}
  public String accept(PrintVisitor v) {
  return v.visit(this);
}

public Exp accept(DeriveVisitor v) {
  return v.visit(this);
}

public Exp accept(SimpVisitor v) {
  return v.visit(this); }}
```

Ek-5. Visitor arayüzü (Derive.java)

Sözdizim sınıf nesnelerinin değerlendirmesi için bir türev hesaplama sınıfının Visitor arayüzü tanımlaması gösterilmiştir.

```
public interface DeriveVisitor {
  public Exp visit(Exp exp);
  public Exp visit(Plus exp);
  public Exp visit(Minus exp);
  public Exp visit(Times exp);
  public Exp visit(Divide exp);
  public Exp visit(Power exp);
  public Exp visit(Sin exp);
  public Exp visit(Cos exp);
  public Exp visit(Tan exp);
  public Exp visit(Cot exp);
  public Exp visit(Sec exp);
```

```
  public Exp visit(Csc exp);
  public Exp visit(Sqrt exp);
  public Exp visit(Ln exp);
  public Exp visit(Log exp);
  public Exp visit(Arcsin exp);
  public Exp visit(Arccos exp);
  public Exp visit(Arctan exp);
  public Exp visit(Arccot exp);
  public Exp visit(Sinh exp);
  public Exp visit(Cosh exp);
  public Exp visit(Tanh exp);
  public Exp visit(Coth exp);
  public Exp visit(Num exp);
  public Exp visit(Var exp);
  public Exp visit(Euler exp);
}
```

Ek-6. Türev Alıcı

Nesne ağacına düğümlerine erişilerek matemetiksel ifadelerin türevleri hesaplanması gösterilmiştir..

```
public Exp visit(Exp exp) {
return exp.accept(this);}
public Exp visit(Plus exp) {
Exp a = exp.exp1.accept(this);
Exp b = exp.exp2.accept(this);
return new Plus(a, b);}
public Exp visit(Minus exp) {
Exp a = exp.exp1.accept(this);
Exp b = exp.exp2.accept(this);
return new Minus(a, b);}
```

```
public Exp visit(Times exp) {
Exp a = (Exp)(exp.exp1.accept(this));
Exp b = (Exp)(exp.exp2.accept(this));
return new Plus(new Times(a, exp.exp2), new Times(exp.exp1, b))}
public Exp visit(Sin exp) {
Exp a = (Exp)(exp.exp.accept(this));
return new Times(new Times(new Num (1.0), a), new Cos (exp.exp));}
public Exp visit(Cos exp) {
Exp a = (Exp)(exp.exp.accept(this));
return new Times(new Times(new Num (-1.0), a), new Cos (exp.exp));}
public Exp visit(Ln exp) {
Exp a = exp.exp.accept(this);
return new Divide(a, exp.exp);}
public Exp visit( Divide exp) {
Exp a = (Exp)(exp.exp1.accept(this));
Exp b = (Exp)(exp.exp2.accept(this));
return new Divide (new Minus(new Times(a,exp.exp2),new Times(exp.exp1,b)),new Ti-
mes(exp.exp2,new Num(2.0)));}
public Exp visit (Var exp) {
return new Num(1.0);}
public Exp visit(Power exp) {
Exp a = (Exp)(exp.exp1.accept(this));
Exp b = (Exp)(exp.exp2.accept(this));
return new Plus(new Times(new Times(b,new Power(exp.exp1,exp.exp2)),new
Ln(exp.exp1)),new Times(a,new Power(new Minus(exp.exp2,new Num(1.0)),exp.exp2)));
}
public Exp visit(Sec exp) {
Exp a = (Exp)(exp.exp.accept(this));
return new Times(new Tan(exp.exp),new Sec(exp.exp));}
public Exp visit(Arcsin exp) {
Exp a = (Exp)(exp.exp.accept(this));
```

```
return new Divide(new Num (1.0),new Sqrt (new Minus(new Num(1.0),new
Power(exp.exp,new Num(2.0))));}
public Exp visit(Arccos exp) {
Exp a = (Exp)(exp.exp.accept(this));
return new Divide(new Num (-1.0),new Sqrt (new Minus(new Num(1.0),new Power
(a,new Num(2.0))));}
public Exp visit(Arccot exp) {
Exp a = (Exp)(exp.exp.accept(this));
return new Divide(new Num (-1.0),new Plus(new Num(1.0),new Power (a,new
Num(2.0)));}
public Exp visit(Arctan exp) {
Exp a = (Exp)(exp.exp.accept(this));
return new Divide(new Num (1.0),new Plus(new Num(1.0),new Power (a,new
Num(2.0)));}
public Exp visit(Tan exp) {
Exp a = (Exp)(exp.exp.accept(this));
return new Times(new Power(new Sec(exp.exp),new Num (2.0)),a);}
public Exp visit(Cot exp) {
Exp a = (Exp)(exp.exp.accept(this));
return new Times(new Times(new Num(-1.0),new Power(new
Csc(exp.exp),new Num (2.0))),a);}
public Exp visit(Csc exp) {
Exp a = (Exp)(exp.exp.accept(this));
return new Times(new Num(-1.0),new Times(new Times(new Csc(exp.exp),new
Cot(exp.exp)),a));}
public Exp visit(Sinh exp) {
Exp a = (Exp)(exp.exp.accept(this));
return new Times(new Times(new Num (1.0), a), new Cosh (exp.exp));
 }
 public Exp visit(Cosh exp) {
 Exp a = (Exp)(exp.exp.accept(this));
```

```
return new Times(new Times(new Num (1.0), a), new Sinh (exp.exp));
public Exp visit(Tanh exp) {
Exp a = (Exp)(exp.exp.accept(this));
return new Divide(new Num (1.0),new Minus(new Num(1.0),new Power(a,new
Num(2.0))));}
```

Ek-7. Bazı İfadeler ve özdeşleri

Sadeleştirilebilir ifadelerin değerlendirilme ve en sade halini olması gerekmektedir. Aşağıdaki tabloda ihtiyaç duyulan ifadeler gösterilmiştir.

İfade	Sadeleştirme sonucu
new Plus(new Number(0.0), r)	r
new Plus(l, new Number(0.0))	l
new Minus(new Number(0.0), r)	new Times(new Num(-1.0), r)
new Minus(l, new Number(0.0))	l
new Times(new Number(0.0), r)	new Number(0.0)
new Times(l, new Number(0.0))	new Number(0.0)
new Times(new Number(1.0), r)	r
new Times(l, new Number(1.0))	l
new Divide(new Number(0.0), r)	new Number(0.0)
new Power(l, new Number(0.0))	new Number(1.0)
new Power(l, new Number(1.0))	l
new Power(new Number(1.0), r)	new Number(1.0)
new Power(new Number(0.0), r)	new Number(0.0)

Ek-8. Sadeleştirme sınıfı ve bazı metotların tanımlaması

Sözdizim sınıfı türüne göre sadeleştirme yapılabilir. Sadeleştirici sınıfı ve bazı metotları aşağıda verilmiştir.

```
public Exp visit(Times exp)
{
Exp a = (exp.exp1.accept(this));
Exp b = (exp.exp2.accept(this));
if (a instanceof Num)
{  if (((Num)a).num==0.0)
return (new Num(0.0));
else if (((Num)a).num==1.0)
return b;}
else if (b instanceof Num)
Tablo'nın devamı {
if (((Num)b).num==0.0)
return (new Num(0.0));
else if (((Num)b).num==1.0)
return a; } return exp;
}public Exp visit(Power exp)
{Exp a = exp.exp1.accept(this);
Exp b = exp.exp2.accept(this);
if (a instanceof Num)
{
if (((Num)a).num==1.0)
return a;
}else if (((Num)a).num==0.0)          {
if(((Num)b).num<=0.0)
{System.out.println("turev yoktur");
System.exit(0);}
else if (((Num)a).num==0.0)
{if(((Num)b).num>0.0)
return (new Num(0.0));
}}
else if (b instanceof Num){
```

```
if (((Num)b).num==0.0)
{if (((Num)a).num>0.0||((Num)a).num<0.0){
return (new Num(1.0));
}} if (((Num)b).num==1.0)
{   return a;}}  return exp;     }
public Exp visit(Divide exp)
Tablo'nın devamı {Exp a = (Exp)(exp.exp1.accept(this));
Exp b = (Exp)(exp.exp2.accept(this));
if (a instanceof Num) {
if(((Num)a).num==0.0)
return (new Num(0.0));
else if (((Num)a).num==0.0) {
if (b instanceof Num)
if(((Num)b).num==0.0)
{  System.out.println("turev yoktur"); System.exit(0);
}}}else if (b instanceof Num) {
if(((Num)b).num==1.0)
return a;
else if (((Num)b).num==0.0){
System.out.println("turev yoktur");
System.exit(0);                    }}return exp;}}
```

Ek-9. Visitor arayüzü (PrintVisitor.java)

Sözdizim sınıf nesnelerinin değerlendirmesi için bir ifade gösterici sınıfının Visitor arayüzü tanımlaması gösterilmiştir.

```
public interface PrintVisitor {
  public String visit(Exp exp);
  public String visit(Plus exp);
```

```
public String visit(Minus exp);
public String visit(Times exp);
public String visit(Divide exp);
public String visit(Power exp);
public String visit(Sin exp);
public String visit(Cos exp);
public String visit(Tan exp);
public String visit(Cot exp);
public String visit(Sec exp);
public String visit(Csc exp);
public String visit(Sqrt exp);
public String visit(Ln exp);
public String visit(Log exp);
public String visit(Arcsin exp);
public String visit(Arccos exp);
public String visit(Arctan exp);
public String visit(Arccot exp);
public String visit(Sinh exp);
public String visit(Cosh exp);
public String visit(Tanh exp);
public String visit(Coth exp);
public String visit(Num exp);
public String visit(Var exp);
public String visit(Euler exp);}
```

Ek-10. İfade gösterici sınıfı ve bazı metotlarının tanımlaması (PrintExp.java)

Sadeleştirici metotlarıyla değerlendirildikten sonra ortaya çıkan ifadenin matematiksel notasyonda gösterim aşamasına geçilir.

```
public class PrintExp implements PrintVisitor {
public String visit(Exp exp) {
return exp.accept(this);}
public String visit(Times exp) {
String a = (String)(exp.exp1.accept(this));
String b = (String)(exp.exp2.accept(this));
return ("(" + a + ")*(" + b + ")");}
public String visit(Minus exp) {
String a = (String)(exp.exp1.accept(this));
String b = (String)(exp.exp2.accept(this));
return ("(" + a + ")-(" + b + ")");
}public String visit(Plus exp) {
String a = (String)(exp.exp1.accept(this));
String b = (String)(exp.exp2.accept(this));
return ("(" + a + ")+(" + b + ")");}
public String visit(Divide exp) {
String a = (String)(exp.exp1.accept(this));
String b = (String)(exp.exp2.accept(this));
return ("(" + a + ")/(" + b + ")");}
public String visit(Power exp) {
String a = (exp.exp1.accept(this));
String b = (exp.exp2.accept(this));
return ("(" + a + ")^(" + b + ")");}
public String visit(Sin exp) {
String a = (String)(exp.exp.accept(this));
return ("sin(" + a + ")");}
public String visit(Cos exp) {
String a = (String)(exp.exp.accept(this));
return ("cos(" + a + ")");}
public String visit(Cot  exp) {
String a = (String)(exp.exp.accept(this));
```

```
return ("cot(" + a + ")");}
public String visit(Tan exp) {
String a = (String)(exp.exp.accept(this));
return ("tan(" + a + ")");}
public String visit(Sec exp) {
String a = (String)(exp.exp.accept(this));
return ("sec(" + a + ")");}
public String visit(Csc exp) {
String a = (String)(exp.exp.accept(this));
return ("csc(" + a + ")");}
public String visit(Sqrt  exp) {
String a = (String)(exp.exp.accept(this));
return ("sqrt(" + a + ")");}
public String visit(Ln  exp) {
String a = (String)(exp.exp.accept(this));
return ("ln(" + a + ")");}
public String visit(Log exp) {
String a = (String)(exp.exp.accept(this));
return ("log(" + a + ")");}
public String visit(Euler exp){
String a = (String)(exp.exp.accept(this));
return "e(" + a + ")";}
public String visit(Arcsin exp) {
String a = (String)(exp.exp.accept(this));
return ("arcsin(" + a + ")");}
public String visit(Arccos exp) {
String a = (String)(exp.exp.accept(this));
return ("arccos(" + a + ")");}
public String visit(Arctan exp) {
String a = (String)(exp.exp.accept(this));
return ("arctan(" + a + ")");}
```

```
public String visit(Arccot exp) {
String a = (String)(exp.exp.accept(this));
return ("arccot(" + a + ")");}
public String visit(Sinh exp) {
String a = (String)(exp.exp.accept(this));
return ("sinh(" + a + ")");}
public String visit(Cosh exp) {
String a = (String)(exp.exp.accept(this));
return ("cosh(" + a + ")");}
public String visit(Tanh exp) {
String a = (String)(exp.exp.accept(this));
return ("tanh(" + a + ")");}
public String visit(Coth exp) {
String a = (String)(exp.exp.accept(this));
return ("Coth(" + a + ")");}
public String visit(Num exp) {
return exp.num + "";}
public String visit(Var exp) {
   return "x";}}
```

ÖZGEÇMİŞ

1985 yılında Giresun'da doğdu. İlköğrenimini Keşap Cumhuriyet İlkokulunda, Orta öğrenimini Giresun Mehmet Akif Ersoy Ortaokulu'nda ve Lise öğrenimini de Giresun Lisesinde tamamladı. 2003 yılında Girne Amerikan Üniversitesi Mühendislik Fakültesi Bilgisayar Mühendisliği Bölümü'nde burslu lisans programına başladı ve 2008 yılında bu bölümde dereceye girerek mezun oldu.

Bir dönem Giresun Devlet Hastanesinde Bilgi İşlem Bölümü'nde çalıştıktan sonra Karadeniz Teknik Üniversitesi Fen Bilimleri Enstitüsü Bilgisayar Mühendisliği Anabilim Dalı'nda yüksek lisans ve Giresun Üniversitesi Şebinkarahisar Meslek Yüksek Okulu'nda Bilgisayar Programcılığı Bölümü'nde Öğretim Görevlisi olarak çalışmaktadır. Yabancı dil olarak İngilizce bilmektedir.