

KARADENİZ TEKNİK ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI

SİMGESEL YAKLAŞIMLARLA LOGARİTMA İFADELERİNİN ÜRETİMİ
VE DEĞERLENDİRİLMESİ

YÜKSEK LİSANS TEZİ

Melike YÜKSEL

EKİM 2024

TRABZON



KARADENİZ TEKNİK ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI

SİMGESEL YAKLAŞIMLARLA LOGARİTMA İFADELERİNİN ÜRETİMİ
VE DEĞERLENDİRİLMESİ

Melike YÜKSEL

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

Tezin Enstitüye Verildiği Tarih :31/10/2024

Tezin Savunma Tarihi :02/10/2024

Tez Danışmanı : Doç. Dr. Hüseyin PEHLİVAN

Trabzon 2024

ÖNSÖZ

“Simgesel Yaklaşımlarla Logaritma İfadelerinin Üretimi ve Değerlendirilmesi” isimli bu tez Karadeniz Teknik Üniversitesi Fen Bilimleri Enstitüsü Bilgisayar Mühendisliği Anabilim Dalı, Yüksek Lisans Programı’nda hazırlanmıştır.

Tez çalışmam boyunca hiçbir desteğini esirgemeyen ve bilim insanı olma yolundaki ilk adımlarımı atarken kıymetli bilgileriyle ve tecrübesiyle yolumu aydınlatan danışman hocam Sayın Doç. Dr. Hüseyin PEHLİVAN’a teşekkürü bir borç bilirim. Ayrıca, ortak olarak çalışmalar yürüttüğümüz ve tez çalışması süresinde de hiçbir yardımdan kaçınmayan arkadaşım Esra ERDEN’e, eşim İbrahim YÜKSEL’e ve canım aileme her zaman bana destek oldukları için çok teşekkür ederim.

Bu tez çalışmasının bundan sonraki çalışmalara katkı sağlamasını temenni ederim.

Melike YÜKSEL

Trabzon 2024

TEZ ETİK BEYANNAMESİ

Yüksek Lisans Tezi olarak sunduğum “Simgesel Yaklaşımlarla Logaritma İfadelerinin Üretimi ve Değerlendirilmesi” başlıklı bu çalışmayı baştan sona kadar danışmanım Doç. Dr. Hüseyin PEHLİVAN’ın sorumluluğunda tamamladığımı, verileri/örnekleri kendim topladığımı, deneyleri/analizleri ilgili laboratuvarlarda 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. 02/10/2024

Melike YÜKSEL

İÇİNDEKİLER

	<u>Sayfa No</u>
ÖNSÖZ.....	III
TEZ ETİK BEYANNAMESİ.....	IV
İÇİNDEKİLER.....	V
ÖZET	VIII
SUMMARY	IX
ŞEKİLLER DİZİNİ.....	X
TABLolar DİZİNİ.....	XI
SEMBOLLER VE KISALTMALAR DİZİNİ	XII
1. GENEL BİLGİLER	1
1.1. Giriş.....	1
1.2. Tezin Amacı	2
1.3. Literatür Araştırması	2
1.4. Logaritmik İfadeler.....	4
1.5. Seviyelerine Göre Programlama Dilleri	6
1.5.1. Düşük Seviyeli Diller.....	6
1.5.2. Orta Seviyeli Diller	6
1.5.3. Yüksek Seviyeli Diller	6
1.6. Dil Çeviriciler.....	7
1.6.1. Yorumlayıcılar (Interpreter).....	7
1.6.2. Derleyiciler (Compiler).....	7
1.6.3. Derleyici Kısımları.....	7
1.6.4. Derleyici ve Yorumlayıcının Karşılaştırılması	9
1.7. Kaynak Veri Analizi.....	10
1.7.1. Kelimesel (Lexical) Analiz	10
1.7.2. Sözdizimsel Analiz	11
1.7.3. Anlamsal (Semantik) Analiz.....	12
1.8. Bağlam Bağımsız Gramer (Context-Free Grammar-CFG).....	12
1.8.1. BNF Notasyonu.....	13
1.8.2. EBNF Notasyonu	13
1.8.3. Türetim (Derivation).....	13

1.8.4.	Sözdizim Sınıfları	14
1.8.5.	Soyut Sözdizim Ağaçları (AST)	14
1.8.6.	Belirsiz (Ambiguous) Gramer	15
1.8.7.	FIRST ve FOLLOW Kümeleri	15
1.8.8.	Anlamsal denetleyici	17
1.9.	Ayrıştırma Algoritmaları	18
1.9.1.	Yukarıdan Aşağıya Ayrıştırma (Top-Down Parsing)	18
1.9.2.	Aşağıdan Yukarıya Ayrıştırma (Bottom-Up Parsing)	18
1.10.	Ayrıştırıcı Üreteçleri (Parser generators)	19
1.10.1.	JavaCC (Java Compiler Compiler)	19
1.11.	Sözdizim Ağacı Değerlendirme Metodolojileri	20
1.11.1.	Instanceof İşleci	21
1.11.2.	Sözdizim Sınıflarına Metot Ekleme	21
1.11.3.	Visitor (Ziyaretçi) Tasarım Deseni	22
1.12.	Doküman Formatlama	23
1.12.1.	Latex	23
1.13.	Java Programlama Dili	24
1.13.1.	BigDecimal ve BigInteger Sınıfları	24
2.	YAPILAN ÇALIŞMALAR	26
2.1.	Giriş	26
2.2.	Gramer Tasarımı	28
2.3.	Sözcük Üretici	29
2.4.	Ayrıştırıcılar	30
2.5.	Sözdizim Sınıfları	31
2.6.	BigDecimal ve BigInteger Sınıflarının Entegrasyonu	32
2.7.	Soyut Sözdizim Ağacı	32
2.8.	Logaritma İfadelerinin Üretimi ve Değerlendirilmesi	33
2.8.1.	Logaritma İfadelerinin Üretimi	35
2.8.2.	Logaritma İfadelerinin Değerlendirilmesi	35
2.8.3.	Parantez Açılımı Yapılarak Logaritma İfadelerin Genişletilmesi	36
2.8.4.	Bilinmeyenlerin Eşitliğin Bir Tarafına Toplanması	39
2.8.5.	Logaritmik İfadelerin Hesaplanması	40
2.9.	Özel Durumlar	40
2.9.1.	Logaritma Toplama Kuralı	40

2.9.2. Logaritma Çıkarma Kuralı	41
2.9.3. Logaritma Çarpma Kuralı	42
2.9.4. Logaritma Bölme Kuralı	43
3. BULGULAR VE TARTIŞMA	45
3.1. Yorumlayıcı Bileşenlerinin Entegrasyonu	45
3.2. Latex Entegrasyonu	46
3.3. Uygulama Bulguları	47
4. SONUÇLAR	55
5. ÖNERİLER	56
6. KAYNAKLAR	57
ÖZGEÇMİŞ	

Yüksek Lisans Tezi

ÖZET

SİMGESEL YAKLAŞIMLARLA LOGARİTMA İFADELERİNİN ÜRETİMİ VE DEĞERLENDİRİLMESİ

Melike YÜKSEL

Karadeniz Teknik Üniversitesi
Fen Bilimleri Enstitüsü
Bilgisayar Mühendisliği Anabilim Dalı
Danışman: Doç. Dr. Hüseyin PEHLİVAN
2024, 59 Sayfa

Bu çalışmada simgesel yaklaşımlar yardımıyla logaritma fonksiyonlarına dayalı olarak matematiksel ifadelerin üretimini, değerlendirmesini ve dokümantasyonunu yapabilen bir yazılım uygulamasının geliştirilmesi amaçlanmıştır. Logaritmik ifadeler için EBNF benzeri notasyonda bir LL(k) grameri tasarlanıp, ifade ayrıştırıcısının kaynak kodu, JavaCC formatında yapılacak tanımlamalar sonucu Java programlama dilinde üretilmiştir. Ayrıştırıcı metotları ile bir yandan logaritmik ifade bileşenlerinin sözdizim analizi gerçekleştirilirken, diğer yandan ilgili ifadenin taşıdığı anlama uygun olarak hiyerarşik yapıda bir soyut sözdizim ağacı oluşturulmuştur. Logaritma ifadeleri değişkensiz (tamsayılara dayalı) ve tek değişkenli (x simgesine dayalı) olmak üzere iki farklı biçimde üretilmiştir. Belirli bir soru kalıbına veya şablonuna göre oluşturulan logaritma problemlerinin çözümleri işlem adımları ile birlikte hem bir terminal üzerinde gösterilebilmekte hem de bir PDF dokümanına aktarılabilir. PDF dosyalarının üretimi, logaritmik ifadelere ait sözdizim ağaçları üzerinden LaTeX doküman formatlama diline yapılan metinsel dönüşümlerle gerçekleştirilmiştir.

Anahtar Kelimeler: Logaritmik ifadeler, biçimsel gramerler, ayrıştırıcılar, simgesel hesaplama, doküman formatlama

Master Thesis

SUMMARY

PRODUCTION AND EVALUATION OF LOGARITHMIC EXPRESSIONS WITH SYMBOLIC APPROACHES

Melike YÜKSEL

Karadeniz Technical University
The Graduate School of Natural and Applied Sciences
Computer Engineering Graduate Program
Supervisor: Asst. Prof. Dr. Hüseyin PEHLİVAN
2024, 59 Pages

This study aims to develop a software application that, via symbolic approaches, can produce, evaluate and document mathematical expressions based on logarithm functions. An LL(k) grammar is designed in EBNF-like notation for logarithmic expressions, and the source code of the expression parser is produced in the Java programming language as a result of definitions in JavaCC format. Using parser methods, on the one hand, the syntax analysis of logarithmic expression components is carried out, on the other hand, an abstract syntax tree with a hierarchical structure is created in accordance with the meaning of the relevant expression. As part of this pattern, the accept method is used to grant access to each tree node, and the visit method is used to evaluate the relevant node. Logarithmic expressions are generated in two different forms: constant (based on integers) and single-variable (based on the symbol x). The solutions to logarithm problems created according to a specific question pattern or template, along with the processing steps, can be both displayed on a terminal and transferred to a PDF document. The production of PDF files is carried out by textual transformations into the LaTeX document formatting language via syntax trees of logarithmic expressions.

Key Words: Logarithmic expressions, formal grammars, parsers, symbolic computation, document formatting.

ŞEKİLLER DİZİNİ

	<u>Sayfa No</u>
Şekil 1. Derleyici bileşenleri	8
Şekil 2. Kelimesel ve sözdizimsel analiz örneği.	11
Şekil 3. $\log(x) + \log(a) = c$ ifadesinin ağaç yapısı.....	15
Şekil 4. Uygulama genel yapısı	27
Şekil 5. Çarpma işleminin dağılma özelliği.....	38
Şekil 6. Toplama kuralı algoritması	41
Şekil 7. Çıkarma kuralı algoritması.....	42
Şekil 8. Çarpma kuralı algoritması	43
Şekil 9. Bölme kuralı algoritması	44
Şekil 10. Şablon 1'e göre üretilen ifadeler ve çözümleri	47
Şekil 11. Şablon 2'ye göre üretilen ifadeler ve çözümleri	48
Şekil 12. Şablon 3'e göre üretilen ifadeler ve çözümleri	49
Şekil 13. Şablon 4'e göre üretilen ifadeler ve çözümleri	49
Şekil 14. Şablon 5 'e göre üretilen ifadeler ve çözümleri	50
Şekil 15. Şablon 6'ya göre üretilen ifadeler ve çözümleri	51
Şekil 16. Şablon 7'ye göre üretilen ifadeler ve çözümleri	52
Şekil 17. Şablon 8'e göre üretilen ifadeler ve çözümleri	53
Şekil 18. Şablon 9 'a göre üretilen ifadeler ve çözümleri	53
Şekil 19. Şablon 10'a göre üretilen ifadeler ve çözümleri	54

TABLÖLER DİZİNİ

	<u>Sayfa No</u>
Tablo 1. Logaritmik özellikler.....	5
Tablo 2. Yorumlayıcı ve derleyicinin karşılaştırılması	9
Tablo 3. Örnek gramer.....	12
Tablo 4. FIRST ve FOLLOW kümeleri	17
Tablo 5. evalLog metodunun instanceof işleci kullanılarak değerlendirilmesi.....	21
Tablo 6. Sözdizim sınıflarına metot ekleyerek değerlendirme yapılması	22
Tablo 7. Visitor arayüzü	23
Tablo 8. Girdi verisi örnekleri	28
Tablo 9. Gramer tasarımı.....	29
Tablo 10. JavaCC formatında birim sözcük tanımlamaları	29
Tablo 11. Logaritma problemleri için birim sözcük ifadeleri.	30
Tablo 12. Ayrıştırıcı metotları.....	31
Tablo 13. Sözdizim sınıfları	32
Tablo 14. Üretilen örnek sözdizim ağaçları.....	33
Tablo 15. Sembol tablosu	33
Tablo 16. Exp soyut sınıfı	34
Tablo 17. Logn sınıfı	35
Tablo 18. ASTExpand() metodu	36
Tablo 19. ASTArrange() metodu.....	39
Tablo 20. LogApp sınıfı	46
Tablo 21. LaTeX tanımlamaları	47

SEMBOLLER VE KISALTMALAR DİZİNİ

CAS	: Computer Algebra System
EBNF	: Extented Backus-Naur Form
LL(k)	: Left to Left
JavaCC	: Java Compiler Compiler
Regex	: Regular Expression
AST	: Abstract Syntax Tree
CFG	: Context Free Grammer
BNF	: Backus-Naur Form
LR	: Left-to-right
YACC	: Yet Another Compiler Compiler
ANTLR	: Another Tool for Language Recognition
PLY	: Python Lex and Yacc
JJTree	: Java JavaBean Tree View
JJDoc	: Java JavaBeans Documentation
JVM	: Java Virtual Machine
JIT	: Just In Time
IDE	: Integrated Development Environment

1. GENEL BİLGİLER

1.1. Giriş

Mühendislik ve çeşitli bilimsel alanlarda sıklıkla karşımıza çıkan matematiksel problemlerin birçoğu analitik yaklaşımlarla çözülemeyecek kadar zordur. Bu gibi durumlarda sayısal hesaplama (analiz) yöntemleri etkili ve güvenilir çözümler sunabilmektedir. Genel olarak, sayısal hesaplama, matematiksel problemlerin yaklaşık çözümlerini bulmak için kullanılan bir dizi yöntemden oluşmaktadır (Chapra & Canale, 2010). Sayısal yöntemlerin dahi yetersiz kaldığı veya karmaşıklığın çok olduğu problemleri çözmek için simgesel hesaplama yöntemleri kullanılmaktadır. Bir simgesel hesaplayıcı, girdi olarak matematiksel ifade alan bir bilgisayar cebir sistemidir (Achary vd., 2012) ve temel cebir işlemlerinden diferansiyel denklemlerin çözümüne kadar birçok alana yardımcı olabilmektedir. Simgesel hesaplamada matematiksel işlemler ve kavramlar, sayılar yerine semboller kullanılarak temsil edilmektedir (Bauer vd., 1998). Bu sayede karmaşık problemler daha kolay anlaşılakta ve çözülmekte, sayısal hesaplamaların aksine, problemin tam çözümü elde edilebilmektedir (Von Zur Gathen & Gerhard, 2013). Simgesel hesaplama yöntemleri, matematiksel problemleri soyut, genelleştirilmiş formlara dönüştürerek analiz edilebilir hale getirmektedir. Problemlerin çözümü, bu soyut ifadelerin kullanılmasıyla açık bir şekilde ifade edilmekte ve ardından çözüm algoritması uygulanarak gerçekleştirilmektedir (Gathen & Gerhard, 2013). Sayısal hesaplama sistemlerinde problem ve sonuçlar daha düzenli ve öngörülebilir olma eğiliminde iken simgesel hesaplama sisteminde problem ve sonuçlar genellikle düzensiz ve öngörülemez büyüklüktedir (Fateman, 2000).

Simgesel hesaplama yöntemleri matematiksel problemlerin çözüm sürecinde birçok farklı yaklaşım sunarken logaritma fonksiyonunun gücünden yararlanmaktadır. Logaritma fonksiyonu, karmaşık ifadelerin basitleştirilmesinde ve simgesel çözüm algoritmalarının geliştirilmesinde önemli bir rol oynamaktadır. Simgesel hesaplamada logaritma, çarpma ve bölme işlemlerini kolaylaştırmanın yanı sıra üstel fonksiyonlarla ilgili problemlerin çözümünde de etkin bir şekilde kullanılmaktadır. Bu sayede logaritma fonksiyonu, simgesel hesaplama verimliliğini artırarak matematiksel çalışmaların daha etkili bir şekilde yürütülmesine imkan sağlamaktadır.

Logaritma fonksiyonu, üstel fonksiyonun tersidir. Logaritma, başlangıçta çarpma ve bölme işlemlerini kolaylaştırmak amacıyla kullanılırken günümüzde bilgisayar bilimleri, istatistik ve diferansiyel denklemler dahil olmak üzere matematiğin çeşitli alanlarında ve bilimsel disiplinlerde vazgeçilemez bir araç olmuştur. Logaritma fonksiyonu, matematikte üstel fonksiyonlarda ile büyüme ve azalma süreçlerine yönelik problemlerin çözümünde önemli bir yer tutmaktadır. Logaritma fonksiyonunun özellikleri ve sağladığı kolaylıklar, matematiksel çalışmaların daha verimli ve etkili bir şekilde yürütülmesini sağlamaktadır. Bu nedenle, logaritmanın matematikteki önemi yadsınamaz ve logaritma matematik eğitiminin temel bir unsuru olarak kabul edilmelidir.

1.2. Tezin Amacı

Simgesel hesaplama yöntemleri karmaşık matematik problemlerinin çözümünün belirlenmesi amacıyla birçok bilimsel disiplinde kullanılmaktadır. Simgesel hesaplama, nitelikleri sayesinde otomatik matematik problemi üretimine ve bir matematik probleminin çözüm sürecinin işlem adımlarının gösterilmesine imkan tanımaktadır. Bu çalışmada önceden tanımlanmış soru şablonlarına göre değişkensiz ve değişkenli olacak şekilde logaritma ifadelerini üreten ve adım adım çözümlerini hem terminal üzerinden gösteren hem de matematiksel notasyonlarla pdf dokümanı üzerine kaydeden bir uygulama geliştirilmiştir.

Çalışma neticesinde geliştirilen bu yazılım uygulaması sayesinde farklı biçimlerde hazırlanabilecek logaritma problemleri ile lise öğrencilerinin logaritmik ifade konusunu öğrenme süreçlerinin hızlandırılması ve logaritma problemi çözme becerilerinin artırılması amaçlanmaktadır.

1.3. Literatür Araştırması

Matematiğin son yıllardaki önemi ve kullanım alanlarının genişlemesi, bu alandaki araştırma ve geliştirme faaliyetlerinin sayısında gözle görülür bir artışa yol açmıştır. Bu durum, matematiğin temel bilimler kategorisinde yer alması ve birçok bilim dalının temelini oluşturmaya bağlanabilmektedir. Artan ilgi, matematiksel modelleme ve problem çözme becerilerinin öneminin artmasıyla, matematiğin karmaşık problemleri çözmek için güçlü bir araç olarak görülmesinden kaynaklanmaktadır. Simgesel hesaplama bu karmaşık problemlerin çözümünde güçlü bir araçtır. Problem çözme kapasitesini

artırması, geniş bir problem yelpazesine uygulanabilmesi, esneklik ve uyarlanabilirlik sunması ve kavramsal anlayışı geliştirmesi gibi birçok avantajı sayesinde matematik ve diğer disiplinlerde önemli bir rol oynamaktadır. Örneğin Taylor ve Maclaurin serilerinin sembolik hesaplama yöntemleri kullanılarak açılımının nasıl gerçekleştirildiği (Mohammad vd., 2014), bilgisayar cebir sistemleri için adım adım çözüm metodolojisinin anlatıldığı (Hosseinpour vd., 2018) ve bazı karmaşık problemlerin çözümünde sadeleştirme, çarpanlara ayırma gibi cebirsel işlemlerin kullanıldığı (Milani vd., 2020) gibi çalışmalar simgesel yaklaşımlar kullanılarak gerçekleştirilmiştir.

Matematik eğitiminde, bir problemin çözümü kadar o problemin benzerlerinin üretilmesinin de önemli olduğu savunulmaktadır (Kojima vd., 2009). Simgesel hesaplama ile matematik problemlerini otomatik olarak oluşturmak ve adım adım çözümünü göstermek mümkündür. Otomatik problem oluştururken, rastgele değişkenler, parametreler veya matematiksel işlemler kullanılabilir. Simgesel hesaplamanın otomatik soru üretimi ve problemlerin çözümüne yönelik işlem adımlarının gösterimi gibi süreçlerde kullanımı matematik eğitimi ve araştırmaları için birçok avantaj sağlamaktadır. Bu kapsamda belirsiz limit problemi üretimi (Aydoğdu, 2016), polinom ifadelerinin üretimi (Efendioğlu, 2017) ve integral problemi üretimi (Yılmaz, 2019) gibi matematiksel konular üzerine otomatik ifade üreten ve değerlendirebilen simgesel hesaplama çalışmaları gerçekleştirilmiştir.

Matematiksel problemlerin üretilmesi ve çözümünün bulunması kadar ilgili problemin nasıl çözüldüğü de önemlidir. Örneğin, çözümlerin karmaşıklığını azaltmak için cebirsel sadeleştirme işleminden yararlanılabilmektedir. Cebirsel sadeleştirme, kullanıcının bakış açısından karmaşık gelebilecek ifadelerin kavranmasını kolaylaştırma, tasarımcının bakış açısından ise daha kullanışlı ve verimli olan sistemler oluşturma hedefi ile iki farklı açıdan ele alınmaktadır (Moses, 1971). Böylece sadeleştirme işlemi sadece ifadenin hesaplanması aşamasında değil üretimi aşamasında da önemsenmektedir.

Teknolojinin gelişmesiyle birlikte bilgisayar sistemleri, matematiksel problemlere çözüm sunan yeni bir paradigmanın ortaya çıkmasına neden olmuştur. 1960'larda Martin Davis ve Donald Knuth öncülüğünde geliştirilen ilk bilgisayar cebir sistemi (CAS), simgesel hesaplama alanında bir çığır açmıştır (Davis, 1965; Knuth, 1966). CAS'ın ortaya çıkışı, matematiğin işlenişi ve problemlere yaklaşımı açısından önemli bir dönüm noktası olmuştur. Bu sistemler matematikçiler, bilim insanları ve mühendisler için önemli bir araç olup, problemleri daha hızlı, daha doğru ve daha verimli bir şekilde çözmelerine destek

olabilmektedir. Bu sayede matematikçiler ve bilim insanları, daha önce elle çözümü imkânsız olan problemleri çözmeye imkanına kavuşmaktadır.

Slagle'in 1963'te gerçekleştirdiği çalışması (Slagle, 1963), bilgisayar bilimlerinde ve matematikte sembolik hesaplamanın temelleri için önemli bir katkı sağlamıştır ve ilerleyen yıllarda bu alandaki gelişmelere zemin hazırlamıştır. 1988 yılında (Boyle & Caviness, 1988) yayımlanan çalışmada, sembolik hesaplama alanının gelecekteki araştırma yönleri ve potansiyel gelişmeleri tartışılmaktadır.

Kişisel bilgisayarların, daha iyi kullanıcı ara yüzlerinin ve yeni tasarlanmış programların ortaya çıkışı ile 1980'lerin sonlarında teknolojinin gelişmesine bağlı olarak simgesel programlama araçlarının popülaritesi artmıştır (Fateman, 2000). Öyle ki bu çeşit hesaplamalar için kullanılabilen Matlab, Maple, Mathematica ve Maxima (Coşkun, 2022) gibi genel amaçlı araçlar ile ticari boyut kazanmıştır. Diğer yandan SageMath gibi simgesel hesaplama için kullanılan açık kaynaklı matematiksel yazılımlar da bulunmaktadır. SageMath betik dili olarak Python'u kullanmaktadır ve çeşitli güçteki ve açık kaynaklı matematiksel yazılım sistemlerini (GAP, Maxima, R, Singular, SymPy, NumPy, scipy vb.) bir araya getirmektedir (Verschelde, 2023). Ayrıca nümerik analiz yöntemlerinin programlanıp yorumlanmasını sağlayan bir programlama dili tasarlanmış ve bu dilin kaynak kodunu yorumlayabilen bir yorumlayıcı geliştirilmiştir (Pehlivan, 2019).

Logaritma, bilim ve teknolojide yaygın uygulama alanına sahip bir temel matematik kavramıdır (Williams, 2011). Birçok matematik alanının temel kavramlarından biri olmasının yanı sıra günlük yaşamdaki problemlerin çözümünde de yararlı bir araçtır (Smith & Confrey, 1994; Weber, 2002). Matematik tarihine ilişkin pek çok kaynağa göre logaritma, çarpma ve bölme işlemlerini basitleştirmek ile özellikle büyük sayılar söz konusu olduğunda büyük miktarda hesaplama tasarrufu sağlamak amacıyla geliştirilmiştir (Cajori, 1913; Eves, 1983). Logaritmik ifadeler bilim, bilgisayar, finans, endüstri ve benzeri çeşitli alanlarda uygulanabildiğinden öğrenimi önemlidir (Burton, 2007; Dintarini, 2018). Ayrıca öğrenciler için öğrenimi zor olan kavramlardan biridir (Liang & Wood, 2005).

1.4. Logaritmik İfadeler

Logaritma bir sayının hangi kuvvetinin, hangi sayıyı verdiğini bulmamızı sağlamaktadır. Michael Stifel logaritma hesabının temeli olan üslü sayıların logaritmasını

ilk tanımlayan kişi olmuştur ve aynı zamanda 2 sayısının kuvvetlerini incelemiştir (Aleksandrov, 2014). John Napier farklı birçok taban kullanarak logaritmayı gerçek anlamda kullanan ilk kişi olmuştur (Aleksandrov, 2014). Napier'in logaritma ile olan çalışmaları genel olarak cebire dayanmaktadır ve logaritma ile ilgili ilk çalışmasını 1614 yılında "A Description of the Wonderful Table of Logarithms (Harika Logaritma Tablosunun bir Açıklaması)" adı altında yayınlamıştır. Napier bu çalışmasında Yunanca "logos" ve "arithmo" kelimelerini sayılar ve logaritmalarını ifade etmek üzere bir arada "logaritma" olarak kullanmıştır (Bruce, 2000).

Logaritma, farklı tabanlara göre çeşitli türlere ayrılır ve her türün kendine özgü kullanım alanları vardır. Onluk logaritma (tabanı 10 olan) fen ve mühendislikte, doğal logaritma (tabanı e sayısı olan) matematikte ve ikilik logaritma (tabanı 2 olan) ise bilgisayar bilimlerinde önemli rol oynar. Logaritma fonksiyonu ve kuralları, bilimsel ve mühendislik problemlerini çözmek için güçlü araçlar sunar.

Logaritma fonksiyonları için genel kurallar Tablo 1'de verilmiştir.

Tablo 1. Logaritmik özellikler

	Özellik	Örnek
Çarpma	$\log_a(xy) = \log_a x + \log_a y$ $\log_a b \cdot \log_b a = 1$ $\log_a b \cdot \log_b c \cdot \log_c d = \log_a d$	$\log_3(9 \cdot 27) = \log_3 9 + \log_3 27$ $\log_3 7 \cdot \log_7 3 = 1$ $\log_2 5 \cdot \log_5 7 \cdot \log_7 9 = \log_2 9$
Bölme	$\log_a \left(\frac{x}{y}\right) = \log_a x - \log_a y$	$\log_2 \left(\frac{64}{4}\right) = \log_2 64 - \log_2 4$
Üs	$\log_a(x^p) = p \log_a x$ $\log_a x^m = \frac{1}{m} \log_a x$ $\log_a x^n = \frac{n}{m} \cdot \log_a x$	$\log_2(2^7) = 7 \log_2 2$ $\log_{3^2} 5 = \frac{1}{2} \log_3 5$ $\log_{2^3} 3^4 = \frac{4}{3} \log_2 3$
Kök	$\log_a \sqrt[p]{x} = \frac{\log_a x}{p}$	$\log_{10} \sqrt{1000} = \frac{1}{2} \log_{10} 1000$
Taban Değiştirme	$\log_a x = \frac{\log_b x}{\log_b a}$ $\log_a x = \frac{\ln x}{\ln a}$	$\log_{10} 8 = \frac{\log_2 8}{\log_2 10}$ $\log_{10} 8 = \frac{\ln 8}{\ln 10}$

Bu tez çalışmasında logaritma ifadelerinin üretimi için logaritmik ifade şablonları kullanılmıştır. Soru şablonlarının uygulamaya bildirimi için Tablo 1'deki ifade örneklerine benzer bir sözdizimi kullanılmaktadır.

1.5. Seviyelerine Göre Programlama Dilleri

1.5.1. Düşük Seviyeli Diller

Düşük seviyeli diller, bilgisayarın donanımıyla doğrudan etkileşime giren programlama dilleridir. Makine koduna yakın bir dilbilgisi kullanırlar. Böylelikle programcının donanım üzerinde daha fazla kontrol sahibi olmasını sağlamış olurlar. Ayrıca düşük seviyeli diller belirli bir işlemci veya donanım mimarisi için tasarlandığı için diğer platformlarda çalışmaz. Düşük seviyeli dillere örnek olarak Makine kodu ve Assembly dili verilebilir. Düşük seviyeli diller işletim sistemleri, gömülü sistemler, sürücüler gibi sistemlerde kullanılmaktadır.

1.5.2. Orta Seviyeli Diller

Orta seviyeli diller, yüksek seviyeli dillerin soyutlama ve taşınabilirlik özelliklerini, düşük seviyeli dillerin donanıma yakın erişim ve performans avantajlarını birleştiren programlama dilleridir. Orta seviyeli diller programcuyu donanım üzerinden soyutlayarak daha yüksek seviyeli kavramlar üzerinde çalışmasına imkan tanımaktadır. Düşük seviyeli dillere göre daha karmaşık bir dil bilgisine sahiptir ve platform bağımsızlığı sağlayan kodlar üretmeye olanak vermektedir. Orta seviyeli dillere C ve C++ dili örnek verilebilir.

1.5.3. Yüksek Seviyeli Diller

Yüksek seviyeli programlama dilleri, bilgisayar programlamada soyutlama düzeyi yüksek olan ve insanların anlayabileceği şekilde tasarlanmış dillerdir. Bu diller programcıdan donanım detayları ve bellek yönetimi gibi karmaşık işlemleri gizleyerek programlama sürecini basitleştirmektedir. Genellikle yüksek seviyeli diller, görev odaklı geliştirilmiştir ve geniş bir kullanım alanına sahiptir. Java, Python ve C# dilleri yüksek seviyeli programlama dillerindendir.

1.6. Dil Çeviriciler

1.6.1. Yorumlayıcılar (Interpreter)

Yorumlayıcı, bir bilgisayar programlama dilinde yazılmış kaynak kodunu satır veya bloklar halinde okuyan, yorumlayan ve doğrudan çalıştıran bir yazılım veya bileşendir. Yorumlama aşamasında kaynak kod makine koduna çevrilmez, yorumlayıcı tarafından çalıştırma sırasında yorumlanır. Yorumlayıcı, kodun çalıştırılmasını adım adım yürüterek işlemi gerçekleştirmektedir.

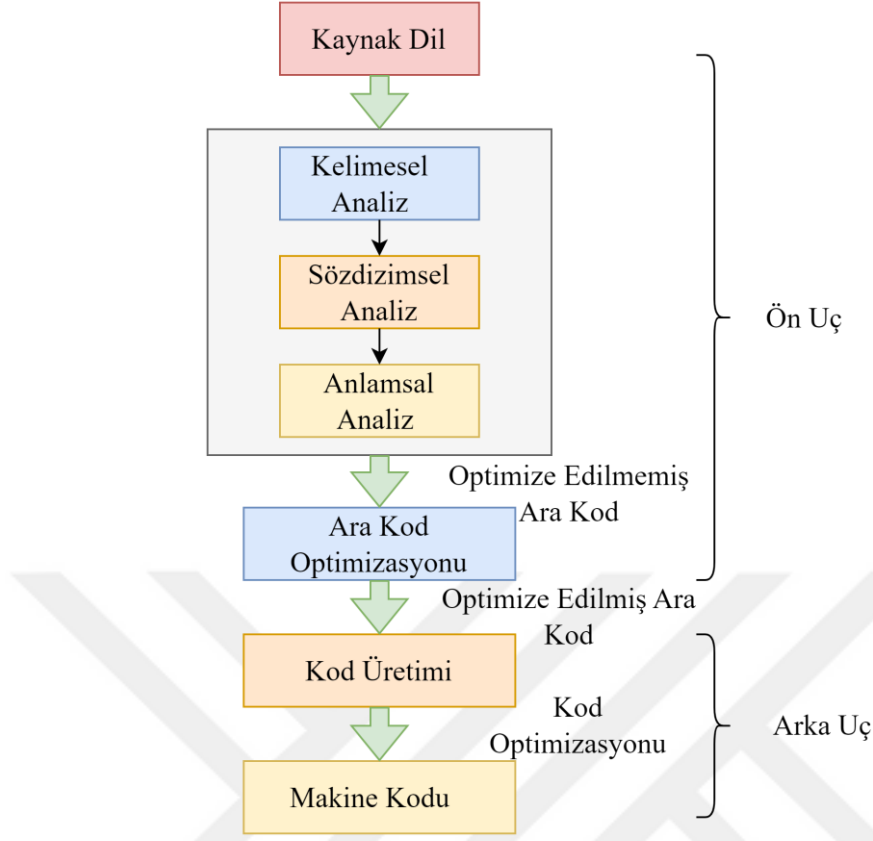
Her bir kod satırı yorumlayıcı tarafından okunmakta, dilin kurallarına ve sözdizimine uygunluğu kontrol edilmekte ve ardından bu kod satırı işlenip çalıştırılmaktadır. Bu süreçte kodun hata vermesi durumunda, hata mesajları üretilerek programın çalışması durdurulmaktadır. Yorumlayıcılar, programın çalıştırılması sırasında değişiklik yapılmasına olanak tanımaktadır. Python, JavaScript ve Ruby gibi programlama dilleri yorumlayıcı tarafından çalıştırılmaktadır.

1.6.2. Derleyiciler (Compiler)

Bilgisayarlar, yalnızca makine kodunu anlayabilmekte ve çalıştırabilmektedir. Programcılar ise genellikle daha yüksek seviyeli programlama dillerini kullanarak kod yazmaktadır. Derleyiciler, bu iki dil arasında köprü oluşturarak yüksek seviyeli kodları, bilgisayarların anlayabileceği düşük seviyeli makine koduna dönüştürmektedir. Kaynak kod, derleyici tarafından önceden işlenmekte ve hedef işlemci mimarisine uygun şekilde makine koduna dönüştürülmektedir.

1.6.3. Derleyici Kısımları

Derleyici, birden fazla aşamadan oluşan karmaşık bir yapıdır ve her aşama, kaynak kodun işlenmesinde belirli bir rolü oynamaktadır. Bu aşamalar ön kısım, ara kod optimizasyonu ve arka kısım olmak üzere üç tanedir. Şekil 1’de derleyici bileşenleri ve aşamaları gösterilmiştir.



Şekil 1. Derleyici bileşenleri

Ön kısım, kaynak kodunun işlenmesinden ve anlamının yorumlanmasından sorumlu olan ilk aşamadır. Bu aşamada kaynak kodun analizi gerçekleştirilerek ara kod oluşturulmaktadır. Bu ara kodun üretimi çoğu zaman söz dizim ağacı üzerinden yapılmaktadır. Ön kısımdaki ara adımlar “Kaynak Veri Analizi” bölümünde anlatılmıştır.

Ara kod, kaynak kodun anlamsal açıdan eşdeğer bir temsiline dönüştürülmüş halidir. Ara kod optimizasyonu, derleme sürecinin önemli bir aşamalarından biridir. Bu aşama ara kod üzerinde yapılan optimizasyonları kapsamaktadır. Optimizasyon, kodun daha hızlı çalışmasını sağlayarak çıktı programının performansını ve etkinliğini artırmaktadır. Ayrıca optimizasyon sonucunda kod daha küçük boyutlara indirgenebilmektedir. Bu durum, programın taşınabilirliğini ve bellekte daha az yer kaplamasını sağlamaktadır.

Arka kısım, derleme sürecinin son aşamasını ifade etmektedir. Bu aşamada, kaynak kodun makine koduna dönüştürülmesi gerçekleştirilmektedir. Böylece hedef platformda çalıştırılabilir bir program üretilmektedir. Arka kısım, derleyicinin en karmaşık ve kritik aşamalarından biri olup genellikle optimizasyon, kod üretimi ve kod optimizasyonu olarak üç ana bileşenden oluşmaktadır.

Optimizasyon aşamasında, üretilen ara kod üzerinde çalışma zamanındaki performansını artırmak, bellek kullanımını azaltmak ve kodun daha etkili hale gelmesini sağlamak için tasarlanmış çeşitli optimizasyon tekniklerini uygulamaktadır. Böylelikle kodun daha verimli hale getirilmesi amaçlanmaktadır.

Kod üretimi aşamasında optimizasyon işlemlerinden sonra, platformun özellikleri dikkate alınarak optimize edilmiş makine kodu üretilmektedir. Kod optimizasyonu sürecinde üretilen makine kodu, hedef platformda performansı artırmak, bellek kullanımını ve enerji tüketimini azaltmak için optimize edilmektedir.

1.6.4. Derleyici ve Yorumlayıcının Karşılaştırılması

Tablo 2’de Derleyicilerin ve yorumlayıcıların özelliklerine göre karşılaştırılması gerçekleştirilmiştir.

Tablo 2. Yorumlayıcı ve derleyicinin karşılaştırılması

Özellik	Yorumlayıcı	Derleyici
Çalışma Şekli	Kaynak kodu satır veya bloklar halinde makine koduna çevirerek çalışmaktadır.	Kaynak kodun tamamını, bir seferde makine koduna çevirmektedir.
Hız	Kaynak kod parçasını, çalıştırmadan hemen önce makine koduna çevirdiği için daha yavaştır.	Kaynak kod halihazırda makine koduna çevrilmiş olduğu için daha hızlıdır.
Bellek Kullanımı	Çalıştırma sırasında kaynak kodunu bellekte tutmak zorundadır.	Kaynak kodunu bellekte tutmak zorunda değildir.
Güvenlik	Programı satır satır çalıştırdığı için her satırda potansiyel bir güvenlik açığı meydana gelme ihtimali vardır, daha güvensizdir.	Programı bir seferde makine diline çevirerek çalıştırdığı için güvenlik açığı meydana gelme ihtimali düşüktür, daha güvenlidir.

Tablo 2'nin devamı

Hata Ayıklama	Hataları, çalışan kaynak kod satırı veya bloğuna göre göstermektedir.	Hataları, kaynak kodun tamamını baz alarak göstermektedir.
Uygulama Alanları	Hızlı geliştirme ve prototipleme ihtiyacı olan uygulamalarda kullanılmaktadır. (Örn: Web uygulamaları)	Performans ve güvenlik ihtiyacı üst seviyede olan uygulamalarda kullanılmaktadır. (Örn: İşletim sistemleri)

1.7. Kaynak Veri Analizi

1.7.1. Kelimesel (Lexical) Analiz

Kelimesel analiz, bir programlama diline ait kaynak kod içinde bulunan karakterler kümesini, birim sözcük (token) kümesine dönüştürme aşamasıdır. Birim sözcük parçaları bir ya da birkaç karakterden oluşabilmektedir ve düzenli ifadeler yardımıyla tanımlanmaktadır. Bu tanımlamalar ile kaynak veri analiz edilerek birim sözcük dizisi üretilmektedir.

Düzenli ifadeler (regex), metin işleme sürecinde güçlü, esnek ve verimli bir araçtır. Neredeyse küçük bir programlama dili gibi olan düzenli ifadeler, metin tanımlamaya ve ayırtırmaya olanak sağlamaktadır (Friedl, 2003). Düzenli ifadeler, metin editörlerinden programlama dillerine kadar birçok yazılım uygulamalarında kullanılmaktadır. Bu araçlar, karmaşık metin işleme görevlerini basitleştirerek zaman ve emek tasarrufu sağlamaktadır.

Düzenli ifadeler, metinlerde belirli kalıpları arayıp işlemek için harfler, rakamlar, özel karakterler ve karakter sınıfları gibi çeşitli öğeler kullanabilen güçlü araçlardır. Kullanılan bazı düzenli ifade kalıpları şunlardır:

- $\wedge$ metnin başlangıcını, $\$$ metnin sonunu ifade etmektedir
- $*$ önündeki bir karakterin sıfır veya daha fazla tekrarını, $+$ önündeki bir karakterin en az bir tekrarını, $?$ ise önündeki karakterlerin sıfır veya bir tekrarını ifade etmektedir.
- $\{n\}$ ifadesi ise önündeki karakterlerin n tekrarını, $\{n,m\}$ ise önündeki karakterlerin n ile m arasındaki tekrarını temsil etmektedir.

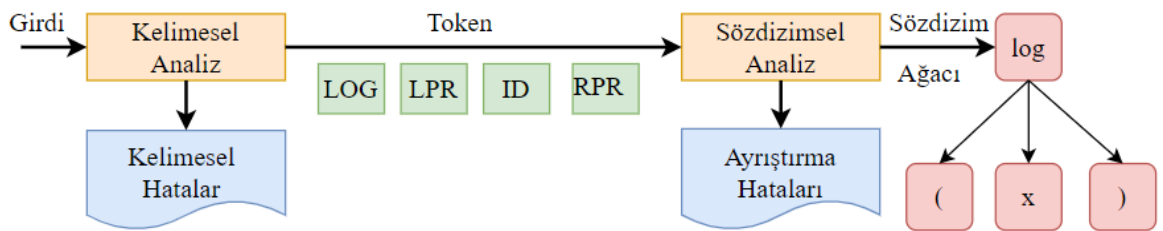
- [] karakter sınıfını temsil etmekte ve içindeki karakterlerden herhangi biriyle eşleşmektedir.
- \ ise özel karakterlerin gerçek anlamlarını temsil etmektedir.
- \d rakamları, \w alfanümerik karakterleri ve alt çizgiyi, \s boşluk karakterini ifade etmektedir.

$a(?:\w*|\d\{1\})?\d\$$ şeklinde verilen bir düzenli ifade örneği için; düzenli ifade "a" harfiyle başlamak zorundadır. “(?:...)” ifadesi, parantez içindeki kısmı gruplayarak seçim yapmaktadır, bu da performansı artırabilmektedir. Sonrasında ise isteğe bağlı olarak “\w*” karakteri kullanılarak bir harf veya “\d{1}” karakteri kullanılarak sadece bir rakam karakterinin alınmasını sağlamaktadır. “\d\$” karakteri kullanıldığı için ifadenin sonunun rakamla bitmesi gerekmektedir.

Örnek olarak “a1”, “at5”, “a44” ifadeleri regex ile eşleşmektedir. “a”, “adgcd” gibi ifadeler regex ile eşleşmemektedir.

1.7.2. Sözdizimsel Analiz

Sözdizimsel analiz, bir metnin dilbilgisi kurallarına uygun olup olmadığının kontrol edildiği aşamadır. Bu aşamada, tokenlar kullanılarak metnin yapısal bir temsili oluşturulmaktadır. Bu temsil, dilbilgisi kurallarına göre analiz edilerek metnin doğruluğu belirlenmektedir.



Şekil 2. Kelimesel ve sözdizimsel analiz örneği.

Şekil 2’de kelimesel analiz sonucunda oluşan bir token dizisi ve bu token dizisi için ayrıştırıcı tarafından üretilen söz dizim ağacı örneği verilmiştir.

1.7.3. Anlamsal (Semantik) Analiz

Semantik analiz, bir programlama dilinde veya doğal bir dilde yazılmış olan ifadelerin anlamını ve anlam ilişkilerini belirlemek için kullanılmaktadır. Semantik analiz aşamasında programlama dillerinin veya doğal dil ifadelerinin, belirli anlamsal kurallara uygun olup olmadığı o dilin tanımına göre kontrol edilmektedir. Semantik analiz yalnızca kelime öbeklerinin tek başına taşıdıkları anlamları değil, aynı zamanda birbirleriyle olan ilişkilerini, bağlamlarını ve nüanslarını da inceleyerek dilin gerçek özünü ortaya çıkarmayı amaçlamaktadır.

1.8. Bağlam Bağımsız Gramer (Context-Free Grammar-CFG)

Gramer terimi, bir dilin yapısını ve kurallarını tanımlayan bir kural kümesini ifade etmektedir. Bir programlama dilinin sözdizimsel yapısını tanımlayan karakter dizileri oluşturmak için kullanılan özyinelemeli üretim kuralları (production rules) grameri oluşturmaktadır. Grameri oluşturan bileşenler şu şekildedir: $G = (V, \Sigma, R, S)$.

- V , non-terminal adı verilen sembollerden oluşan sonlu bir kümedir. Bu semboller, gramerdeki kurallarda yer alıp kelime öbeklerini veya ifadeleri temsil etmektedir.
- Σ , V kümesinden ayrık ve terminal olarak adlandırılan sonlu bir kümedir. Terminaller, gramerdeki gerçek kelimeleri temsil eden sembollerdir.
- R , her biri bir değişken olup değişkenlerden ve terminallerden oluşan bir dizi kurallardan oluşan sonlu bir kümedir. Bu kurallar, gramerdeki kelimelerin nasıl oluşturulacağını belirlemektedir.
- $S \in V$, başlangıç sembolünü ifade etmektedir.

Tablo 3. Örnek gramer

$S \rightarrow AB$
$A \rightarrow aA \mid \varepsilon$
$B \rightarrow bB \mid \varepsilon$

Tablo 3'te verilen gramerde a ve b sembolleri terminal, S , A ve B sembolleri non-terminaldir. S başlangıç sembolünü, $S \rightarrow AB$ üretim dizisini ifade etmektedir.

Bağlam bağımsız gramerler, dilbilim ve bilgisayar bilimi alanlarında dilbilgisi kurallarını tanımlamak için kullanılan güçlü bir araçtır. Bu yapılar, dilbilgisel analiz ve sentez, programlama dillerinin sentaks analizi, metin analizi ve doğal dil işleme gibi birçok farklı alanda yaygın olarak kullanılmaktadır.

Bağlam bağımsız gramerler bir formal gramer BNF veya EBNF gibi notasyonlar yardımıyla tanımlanabilmektedir.

1.8.1. BNF Notasyonu

John Backus ve Peter Naur tarafından 1950'li yılların sonuna doğru bulunan Backus-Naur form, formal dillerin yazım kurallarını tanımlamak için kullanılan bir yazım kuralları kümesidir. Bilgisayar programlama dillerinin yazım kurallarını özetlemekte kullanılan en önemli yöntemlerden biridir.

1.8.2. EBNF Notasyonu

EBNF notasyonu, BNF notasyonunun ek semboller ile genişletilmiş biçimi olup kural, sayı ve uzunlukları BNF notasyonuna göre daha kısadır. Diğer bir ifadeyle, bilgisayar programlama dillerini ve biçimsel dilleri tanımlamanın bir başka yoludur. EBNF notasyonu matematiksel ve bilgisayar bilimleri alanlarında kullanılması için yeni bir resmi gramer üst dili olarak geliştirilmiştir (Quinlan vd., 2019).

1.8.3. Türetim (Derivation)

Türetim, formal dilbilgisi ve sentaks analizi bağlamında, dildeki ifadelerin nasıl oluşturulduğunu gösteren yöntemlerdir. Gramerin başlangıç sembolü, genellikle ilk üretim kuralının solundaki non-terminal sembolüdür. Türetim süreci başlangıç sembolü ile başlayıp herhangi bir non-terminal sembolünün, sağ taraftaki alternatif sembol dizilerinden biriyle genişletilerek devam ettirilir. Türetim işlemi, en sağdan ve en soldan olmak üzere iki şekilde yapılabilmektedir; En Soldan Türetim (Leftmost Derivation) ve En Sağdan Türetim (Rightmost Derivation).

En soldan türetim, sentaks analizi sürecinde bir ifadeyi oluştururken her zaman en sol taraftaki non-terminal sembollerini ile genişletmeyi tercih etmektedir.

Tablo 3'te verilen gramer için "aabb" kelimesinin en soldan türetme yöntemi ile adım adım türetimi, aşağıdaki gibi yapılmaktadır.

$$S \rightarrow AB \Rightarrow S \rightarrow aAB \Rightarrow S \rightarrow aaAB \Rightarrow S \rightarrow aaB \Rightarrow S \rightarrow aabB \Rightarrow S \rightarrow aabbB \Rightarrow S \rightarrow aabb$$

En soldan türetmenin aksine, en sağdan türetmede türetim adımları sağdan sola doğru ilerlemekte ve her adımda en sağdaki non-terminal ile sembol genişletilmektedir.

Tablo 3'te verilen gramer için türetim sağdan olacak şekilde ilk B non-terminali seçilmektedir. "aabb" kelimesini sağdan türetmek için aşağıdaki adımlar kullanılmaktadır.

$$S \rightarrow AB \Rightarrow S \rightarrow AbB \Rightarrow S \rightarrow AbbB \Rightarrow S \rightarrow Abb \Rightarrow S \rightarrow aAbb \Rightarrow S \rightarrow aaAbb \Rightarrow S \rightarrow aabb$$

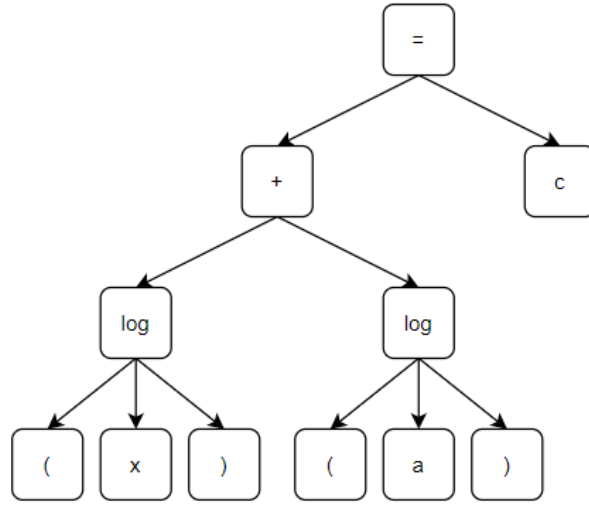
1.8.4. Sözdizim Sınıfları

Sözdizim sınıfları nesneye dayalı programlama yapılarında, gramer kurallarıyla üretilen girdi verilerini ifade etmek için kullanılmaktadır. Gramer kuralları sözdizim sınıflarının tanımlanmasında önemli bir role sahiptir. Genellikle her bir gramer kuralı için işlevini ve biçimini temsil eden bir sözdizim sınıfı tanımlanır. Alternatifi bulunan kuralları temsil eden sınıflar, aynı üst sınıftan miras alınarak tanımlanmaktadır.

1.8.5. Soyut Sözdizim Ağaçları (AST)

Soyut sözdizim ağacı, diğer bir deyişle ayrıştırma ağacı, bir programlama dilinin sözdizimini hiyerarşik ağaç benzeri bir yapı biçiminde oluşturulan bir veri yapısıdır. Bu yapı, derleyiciler için sembol tabloları ve kod üretimi için kullanılmaktadır. Sözdizim ağacı, dildeki tüm yapıları ve birbirleri ile ilişkilerini temsil etmektedir.

Bir sözdizimi ağacı, hiyerarşik bir yapıda birbirine bağlanan bir dizi düğümden oluşmaktadır. Her düğüm, bir sözdizimi sınıfından türetilmekte ve kaynak veri bileşenlerini temsil eden bir nesne tarafından oluşturulmaktadır. Ayrıştırma ağaçlarındaki iç düğümler non-terminal, yapraklar ise terminalleri ifade etmektedir. Şekil 3'te $\log(x) + \log(a) = c$ ifadesini gösteren sözdizim ağacında "log", "+", "=" ifadeleri iç düğümler olup non-terminalleri oluştururken, "(", "x", ")", "c" ifadeleri yapraklar olup terminallere karşılık gelmektedir.



Şekil 3. $\log(x) + \log(a) = c$ ifadesinin ağaç yapısı.

1.8.6. Belirsiz (Ambiguous) Gramer

Bir gramer, iki veya daha fazla ayrıştırma ağacı oluşturuyorsa bu gramer belirsiz olarak adlandırılır. Bir gramerin belirsiz olması, o gramerle oluşturulan bir ifadenin birden fazla şekilde yorumlanabileceği anlamına gelmektedir.

Belirsizliği gidermek için genellikle işleç önceliklerinin düzenlenmesi veya yeni non-terminal ve kurallar eklenmesi kullanılan yöntemlerdir. Ancak bu yöntemlerin etkinliği, belirsizliğin kaynağına ve gramerin yapısına bağlı olarak değişebilmektedir. Örneğin, " $x+y*z$ " ifadesi hem " $(x+y)z$ " hem de " $x+(yz)$ " şeklinde yorumlanabilmektedir. Bu da belirsizliğe sebep olmaktadır. Bu belirsizliği gidermek için işleç önceliklerini belirtebilir veya yeni non-terminal semboller kullanarak grameri yeniden düzenleyebilirsiniz.

1.8.7. FIRST ve FOLLOW Kümeleri

Bir gramerdeki her bir sembol veya sembol dizisi için, o sembol veya dizinin olası başlangıç terminal sembollerini içeren kümeye "FIRST" denmektedir. Kısacası non-terminal sembolünün türetebileceği ilk sembollerin kümesidir. Bir gramerdeki her bir sembol için, o sembolün takip edebileceği terminal sembollerin kümesine ise "FOLLOW" denilmektedir.

Tahmine dayalı (öngörülü) ayrıştırma yapan bir ayrıştırıcı, FIRST ve FOLLOW kümeleri yardımıyla oluşturulmaktadır. Bu kümeler, ayrıştırma tablosunun değerlerini doldurmamızı sağlayarak, ayrıştırma işlemini mümkün kılmaktadır.

FIRST kümesi oluşturma kuralları şu şekildedir:

1. FIRST kümesi oluşturmaya gramerin son üretim kuralı ile başlanır. Çünkü alttaki üretim kuralları doğrudan terminal sembolü içermesi daha mümkün olan kurallardır.
2. Eğer ilk sembol terminal ise FIRST kümesinde o terminal bulunur.
3. Eğer ilk sembol non-terminal ise FIRST kümesi o non-terminalin FIRST kümesidir.
4. Eğer ilk sembol ϵ sembolü ise FIRST kümesine ϵ sembolü eklenir.
5. Üretim kuralında $|$ (veya) varsa FIRST kümesine $,$ (virgül) eklenerek ekleme yapılmaya devam edilir.

FOLLOW kümesi oluşturma kuralları aşağıdaki gibi listelenmiştir.

1. Gramerin başlangıç sembolünün FOLLOW kümesi $\$$ sembolü ile başlamaktadır.
2. Sol taraftaki non-terminalin FOLLOW kümesi bulunurken gramerin sağ tarafında non-terminalin geçtiği tüm üretim kuralları değerlendirilmektedir.
3. Eğer üretim kuralının sağ tarafında non-terminali takip eden sembol terminal ise o terminal FOLLOW kümesine eklenmektedir.
4. Eğer üretim kuralının sağ tarafında non-terminali takip eden sembol non-terminal ise o non-terminalin FIRST kümesi FOLLOW kümesine eklenmektedir.
5. Eğer üretim kuralının sağ tarafında non-terminali takip eden bir sembol yok ise sol tarafındaki non-terminalin FOLLOW kümesi ilgili FOLLOW kümesine eklenmektedir.
6. Eğer FOLLOW kümesine bir non-terminalin FIRST kümesi eklenecek (4. kural) ve bu küme ϵ sembolü içeriyor ise sağ tarafta bulunan sıradaki sembolün hesaplaması yapılarak FOLLOW kümesine eklenmektedir.

$$E \rightarrow TE'$$

$$E' \rightarrow + TE' \mid \epsilon$$

$$T \rightarrow FT'$$

$$T' \rightarrow *FT' \mid \epsilon$$

$$F \rightarrow id \mid (E)$$

Yukarıdaki gramerin FIRST ve FOLLOW kümeleri Tablo 4’ te verilmiştir. F non-terminali için FIRST(F) kümesi, “id” veya “(” sembolleridir. Aynı şekilde, sembol T' için FIRST(T') kümesi, “*” veya “ε” (boşluk) sembolleridir. E non-terminalinin FOLLOW(E) kümesi 1. ve 2. kural sebebiyle “\$,” sembolleridir. FOLLOW(E') kümesi 5. kurala göre hesaplanmakta bu da FOLLOW(E) kümesine eşit olmaktadır. FOLLOW(T) kümesi hesaplanırken 4 numaralı kurala göre FIRST(E') ve 6 numaralı kuraldan FOLLOW(E') kümesinin değerlerine eşit olmaktadır.

Tablo 4. FIRST ve FOLLOW kümeleri

	FIRST	FOLLOW
E	id, (	\$,)
E'	+, ε	\$,)
T	id, (	+, \$,)
T'	*, ε	+, \$,)
F	id, (	+, *, \$,)

1.8.8. Anlamsal denetleyici

Anlamsal denetleyici bir programlama dilinde, tür denetimi, statik tip kontrolü ve bellek yönetimi gibi işlemleri gerçekleştiren bileşendir. Tür denetimi için kaynak koddaki tanımlamalara dayalı bir sembol tablosu kullanır. Sembol tablosu programdaki tüm tanımlayıcıların (değişkenler, fonksiyonlar vb.) adlarını, türlerini ve diğer özelliklerini saklayan bir veri yapısıdır. Bu özellikleri ile anlamsal denetleyici, anlamsal analiz için kullanılmaktadır.

Kaynak kodun anlamsal analizi sırasında, ayrıştırıcı tarafından oluşturulan nesne ağacından yararlanılır. Nesne ağacı, tüm düğümleri gezmekte ve kaynak veride bildirim yapıları her bir ismi, (isim, tür) çifti biçiminde sembol tablosunda saklamaktadır.

1.9. Ayırıştırma Algoritmaları

Ayırıştırma algoritmaları, bir metnin belirli dilbilgisi kurallarına uygunluğunu belirlemek için kullanılan yöntemlerdir. İki temel ayırıştırma yöntemi vardır.

1.9.1. Yukarıdan Aşağıya Ayırıştırma (Top-Down Parsing)

Bu yöntemde, metin başlangıç sembolünden başlayarak alt dallara doğru ayırıştırılmaktadır. Başlangıç sembolü gramerdeki üretim kurallarına göre alt dallara ayırıştırmayı tüm semboller ayırıştırılana kadar devam etmektedir. LL(k) ayırıştırıcı yukarıdan aşağıya ayırıştırıcı örneğidir.

LL(k) ayırıştırma, Lewis ve Stearns (1968) ve Knuth (1971) tarafından geliştirilmiştir. LL(k) gramerleri, yukarıdan aşağıya kesin olarak ayırıştırılabilen gramerlerdir ve sentaks analizi için "k" kadar ileriye bakış (lookahead) yapmaktadır. Bu, ayırıştırıcının bir sonraki tokeni göz önünde bulundurarak karar verdiği anlamına gelmektedir. "LL" terimi, parser'ın soldan sağa ayırıştırma yaptığını ve en soldaki türetimi oluşturulduğunu belirtmektedir.

LL ayırıştırmada, giriş dizisi okunurken, her adımda önceden tanımlanmış kurallara göre en soldaki alt dizinin hangi üretim kuralıyla türetildiği tahmin edilmeye çalışılmaktadır. Tahminin başarılı olduğu süreçte, tahmin edilen kural uygulanmaktadır. Eğer tahmin başarısız olursa, hata mesajı üretilmekte ve ayırıştırma işlemi durdurulmaktadır.

LL ayırıştırıcılar hızlı ve daha anlaşılır yapıya sahiptir. Geriye dönme işlemi olmadığı için bellek kullanımı daha azdır. Ayrıca tüm CFG'ler için LL ayırıştırıcı oluşturmak mümkün değildir. Çünkü gramerde sol faktörleme dönüşümleri gerekebilir. Ayrıca gramerlerde çok anlamlılık (ambiguity) gibi kısıtlamalar olabilmektedir.

1.9.2. Aşağıdan Yukarıya Ayırıştırma (Bottom-Up Parsing)

Aşağıdan yukarıya ayırıştırma, terminal sembollerden başlanarak daha karmaşık non-terminal sembollere doğru ilerleyen ayırıştırma yöntemidir. Tüm girdi dizisi işlenene ve başlangıç sembolü oluşturulana kadar devam etmektedir. Bu yöntemde ayırıştırma ağacı yapraklardan köke doğru üretilmektedir. LR ayırıştırıcı bu yönteme örnektir.

LR ayrıştırıcı, bilgisayar programlama dili derleyicileri ve diğer ilişkili araçlar tarafından yaygın olarak kullanılan bir bağlam bağımsız gramer için aşağıdan yukarıya ayrıştırıcı türüdür. LR ayrıştırıcısı, girdisini soldan sağa okumakta ve sağdan türetim gerçekleştirmektedir. Özetle LR ayrıştırıcı, en üst düzey gramer kurallarını yaprak düğümlerden başlayarak yukarıya doğru inşa ederek indirgemeye çalışmaktadır.

LR ayrıştırmada, giriş dizisi okunurken her adımda önceden tanımlanmış bir tabloya göre hangi üretim kuralının uygulanacağına karar verilir. Bu tablo, giriş dizisinin bir kısmını ve ayrıştırıcının mevcut durumunu göz önünde bulundurarak oluşturulur. Uygun kural seçildikten sonra, giriş dizisinden türetilen alt dizi işlenmiş kabul edilir ve ayrıştırıcının durumu güncellenir.

LR ayrıştırma, bir dilin sentaksını hızlı ve etkin bir şekilde incelemek için yaygın olarak kullanılmaktadır. Bu yöntem, bir derleyici veya yorumlayıcı tarafından bir programın kodunun doğru bir şekilde yorumlanmasını sağlamaktadır. Özellikle büyük ve karmaşık programlar için önemli bir role sahiptir. LR ayrıştırıcıları, pratikte kullanılan tüm deterministik ayrıştırıcılar arasında en güçlü olanıdır. LL ayrıştırıcılara göre bellek kullanımı daha fazladır ve daha yavaş çalışmaktadır. Ayrıca daha geniş CFG'leri desteklemekte, sol faktörleme gibi gramer dönüşümlerine ihtiyaç duymazlar ve tablo tabanlı bir yapıya sahip oldukları için rekursif iniş gibi karmaşık tekniklere gerek duymazlar.

1.10. Ayrıştırıcı Üreteçleri (Parser generators)

Ayrıştırıcı üreteçleri, bir formal dilin gramer kurallarına uygun metinleri sözcük dizileri üzerinden analiz ederek sözdizim ağacı oluşturan ayrıştırıcılar için kaynak kod üretebilen yazılım araçlarıdır. Bu araçlar derleyiciler, yorumlayıcılar, metin işleme araçları, programlama dilleri gibi birçok farklı alanda kullanılır.

Ayrıştırıcı üreteçleri, aşağıdan yukarıya ayrıştırma veya yukarıdan aşağıya ayrıştırma gibi farklı yöntemler kullanabilir. En yaygın kullanılan ayrıştırıcı üreteçleri şunlardır: YACC (URL-1), Bison (Buffalo) (URL-2), ANTLR (URL-3), PLY (URL-4) JavaCC.

1.10.1. JavaCC (Java Compiler Compiler)

JavaCC, YACC ve benzeri araçlar tarafından üretilen aşağıdan yukarıya ayrıştırıcıların aksine, yukarıdan aşağıya ayrıştırıcı üretmektedir. Yukarıdan aşağıya

ayrıştırıcıların, daha genel gramerlere izin vermesinin yanı sıra, hata ayıklamasının daha kolay olması, gramerdeki herhangi bir non-terminal ayrıştırması yapabilmesi ve ayrıştırma sırasında ayrıştırma ağacında yukarı ve aşağıya değer aktarabilmesi gibi başka avantajları da vardır (URL-5).

JavaCC çeşitli ayrıştırma stratejilerini desteklemektedir ve varsayılan olarak bir LL(1) ayrıştırıcısı üretmektedir. Bu özelliklerinin yanı sıra JavaCC, ayrıştırma sırasında oluşabilecek hataları önceden algılanması ve ayrıştırma işleminin doğru bir şekilde gerçekleştirilmesi için çeşitli araçlar sağlamaktadır.

Ayrıştırıcı üretimi için JavaCC kullanımının sağladığı avantajların bir kısmı aşağıda listelenmiştir;

- Gramerlerin geliştirilmesi daha kolaydır ve geliştirilen gramerler daha okunaklıdır.
- BNF'nin standart özelliklerine ek olarak, EBNF tanımlarına da izin vermektedir. Bu sayede daha karmaşık gramerler tanımlanabilmektedir.
- Ayrıştırma sırasında yok sayılabilen veya özel işleme tabi tutulabilen özel tokenların tanımlanmasına imkan sağlamaktadır.
- Tokenların büyük-küçük harfe duyarlılığını tanımlama yeteneği sunmaktadır. Meta-sözcükler kullanılmasına olanak vermektedir.
- Java programlarıyla entegre edilebilir Java ayrıştırıcısı üretir.
- JJTree ve JJDoc gibi araçlarla birlikte tedarik edilmektedir (URL-5). JJTree, parse ağacı oluşturma için güçlü bir araçtır. JJDoc ise, gramer dosyalarını dokümantasyon dosyalarına dönüştürmek için kullanılmaktadır.

1.11. Sözdizim Ağacı Değerlendirme Metodolojileri

Bir nesne ağacındaki her bir düğüm, en uçtaki düğümden başlanarak kök düğüme doğru değerlendirilmektedir. Bu değerlendirme düğümün içerdiği nesnenin türüne bağlı olarak gerçekleşmektedir. Nesne türleri ve temsil edilen işlemler için üç farklı yöntemden yararlanılabilir. Bunlar; instanceof işleci, sözdizim sınıflarına metot ekleme ve Visitor tasarım desenidir.

1.11.1. Instanceof İşleci

Instanceof işleci, Java programlama dilinde bir nesnenin belirli bir sınıfın örneği olup olmadığını kontrol etmek için kullanılmaktadır. Bu operatör, bir nesnenin belirli bir sınıfın veya o sınıfın alt sınıfının bir örneği olup olmadığını belirler.

Bu yaklaşım, kod ile belirli sınıflar arasındaki gevşek bağlantıyı desteklemektedir. Böylece Tablo 5'te verilen örnekteki Num ve Ratio sınıflarında değişiklik yapılmasını gerektirmez. Tür dönüşümlerinin aşırı kullanımı, özellikle de instanceof operatörüyle birlikte, kodun okunabilirliğini önemli ölçüde azaltabilir. Ayrıca tür varsayımlarının yanlış olması durumunda çalışma zamanı hatalarının potansiyeli artmaktadır. Bu yöntem sistem performansını azaltabilir ve nesne yönelimli değildir.

Tablo 5. evalLog metodunun instanceof işleci kullanılarak değerlendirilmesi

```
public Exp evalLog(Table t) {
    e1 = e1.evalLog(t);
    e2 = e2.evalLog(t); ... .....
    if (e1 instanceof Num && e2 instanceof Num)
        return new Num((((Num)e1).n.add(((Num)e2).n)));
    if (e1 instanceof Ratio && e2 instanceof Ratio)
        return new Ratio((((Ratio)e1).getR1().add(((Ratio)e2).getR1())));
    if (e1 instanceof Num && e2 instanceof Ratio)
        .....}
}
```

1.11.2. Sözdizim Sınıflarına Metot Ekleme

Bir nesnenin bileşenlerine erişmek için klasik yaklaşım hem alt nesnelere erişen hem de onlar üzerinde işlem yapan özel yöntemlerin kullanılmasıdır. Bu yaklaşımda her bir sözdizim sınıfına, o sınıfın özelliklerine göre metotlar eklenmektedir.

Tablo 6. Sözdizim sınıflarına metot ekleyerek değerlendirme yapılması

```

class Log extends Exp {
    public Exp evalLog(Table t) {
        return new Log(e.evalLog(t));
    }.....
class Logn extends Exp {
    public Exp evalLog(Table t) {
        return new Logn(e1.evalLog(t),e2.evalLog(t));
    }.....

```

Tablo 6 örneğinde verilen “evalLog()” metodu “Log” ve “Logn” sınıflarında o sınıfların ihtiyaçlarına göre tanımlanmıştır. Nesne ağacı düğümü değerlendirilirken nesnenin türüne bakılmaksızın o düğümün içerdiği nesneye ait “evalLog()” metodunun çağrılması yeterli olacaktır.

1.11.3. Visitor (Ziyaretçi) Tasarım Deseni

Ziyaretçi tasarım deseni, bir nesne hiyerarşisindeki nesneler üzerinde çeşitli işlemleri gerçekleştirmek için kullanılan davranışsal tasarım deseni. Bu desende, her işlem türü için ayrı bir "ziyaretçi" sınıfı bulunmakta ve bu sınıflar, ziyaret ettikleri nesnelerin türüne göre farklı davranışlar sergilemektedir. Bu sayede, nesneyi değiştirmeden yeni işlevler eklemek veya mevcut işlevleri değiştirmek mümkün olmaktadır. Ziyaretçi tasarım deseninin önemli bir faydası, ağaç düğümleri üzerinde yapılacak olan farklı türden bir değerlendirme için farklı bir desen örneğinin tanımlanmasının yeterli olmasıdır.

Visitor arayüzü gramer içindeki her bir kural sözdizimi için bildirimi yapılan özel metotlardan oluşur; visit adı verilen bu metotların her biri farklı bir sözdizim sınıfının referansını argüman alır ve sözdizim ağacının aynı türden düğümünün değerlendirmesi ile ilgilenir. visit() metodu ağacın hangi düğümünde bulunulduğu hakkında bilgi vermektedir. Bu sayede ağacın hangi düğümünde ne yapılacağına karar verilerek bu fonksiyonlar kullanıcının isteğine göre doldurulabilmektedir. Bu metotların gövdeleri içerisinden çağrımı yapılan accept() metotları ile ağacın bir sonraki alt düğümüne geçiş sağlanmaktadır. Bir ağacın düğümünden, değerlendirmesi hata üreten bir alt düğüme geçilmeye çalışılırsa accept() metotlarından geriye dönülmeden hata üretilir ve işlem

çıkılır. Böylece ihtiyaç duyulan anlamsal kontroller ağaç üzerinden gidilerek kolayca yapılabilir.

Sözdizim sınıflarına, kaynak verinin değerlendirilmesi sürecinde, sözdizim ağacının düğümleri arasında gezinmeyi sağlayan accept isminde birer metot eklenmelidir. accept() metotları için aşağıda gösterilen standart bir gövde tanımlaması yapılmaktadır.

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

Tablo 7’de gösterilen Visitor arayüzünde geri dönüş veri tipi Object olarak verilmiştir. Böylece soyut sözdizimi ağacı üzerinde yapılacak olan farklı değerlendirme türlerinde aynı Visitor’ün kullanılması sağlanmıştır.

Tablo 7. Visitor arayüzü

<pre>interface Visitor { public Object visit(LogEqnList e); public Object visit(Exp e); public Object visit(LogEqn e); public Object visit(Plus e); public Object visit(Minus e); public Object visit(Times e); public Object visit(Divide e);</pre>	<pre>public Object visit(Id e); public Object visit(Num e); public Object visit(Ratio e); public Object visit(Ln e); public Object visit(Log e); public Object visit(Logn e);} public Object visit(Power e); }</pre>
--	---

1.12. Doküman Formatlama

1.12.1. Latex

LaTeX, genellikle bilimsel yayınlar için doküman formatlamada kullanılan bir yazılım sistemidir. Bu sistem içerisinde WYSIWYG (What You See Is What You Get) kelime işlemcilerinde (örneğin, Microsoft Word) olduğu gibi doğrudan biçimlendirilmiş içerik tanımları yerine etiketler aracılığıyla biçim ve içerik ayrı olarak tanımlanmaktadır. İçerik ve biçim tanımlamalarını içeren kaynak kod derlendikten sonra doküman

formatlanıp yayına hazır hale getirilmektedir. Kaynak kod .tex uzantılı dosyalarda tanımlanmaktadır ve derlendiğinde çeşitli formatlarda dosyalara dönüştürülebilmektedir.

1.13. Java Programlama Dili

Java, nesneye yönelik, platform bağımsız, yüksek verimli ve çok işlevli bir programlama dili olup, 1995 yılında Sun Microsystems mühendislerinden James Gosling tarafından geliştirilmiştir ve günümüzde Oracle tarafından yönetilmektedir (URL-6). Bu dil, hem yorumlanan hem de derlenen bir yüksek seviyeli dil olarak öne çıkmaktadır.

Java, nesne yönelimli programlama paradigmasını benimsemektedir. Java kodları, Java Sanal Makinesi (JVM) tarafından yorumlanır. JVM işletim sisteminden bağımsız olduğu için Java kodları, Windows, Linux, macOS gibi farklı işletim sistemlerinde çalışabilir. JIT derleyici, Java kodlarını çalıştırma zamanında makine diline dönüştürerek kodların daha hızlı çalışmasını sağlamaktadır.

Java, uygulama geliştirmek için geniş bir yelpaze sunmaktadır. Java; Servlet, JSP ve Spring gibi web geliştirme çerçeveleri ile web uygulamaları, Swing ve JavaFX gibi kütüphaneler ile masaüstü uygulamaları geliştirmek için kullanılmaktadır. Ayrıca mobil uygulamalar, gömülü sistemler, kurumsal uygulamalar gibi sitemlerde de kullanılmaktadır.

Bu tez çalışmasında Java programlama dilinden yararlanılarak bir konsol uygulaması geliştirilmiştir. Java'da konsol uygulamaları, kullanıcı etkileşimlerini komut satırı arayüzü üzerinden gerçekleştiren programlardır. Bu tür uygulamalar, giriş ve çıkış işlemlerini “System.in” ve “System.out” komutları ile gerçekleştirip, File ve FileReader gibi sınıflar ile dosyaları okuyabilir ve yazabilirsiniz. Program bir metin dosyasına “.java” uzantılı olacak şekilde kaydedilerek, komut satırından ilgili dizine gidilerek “java uygulama_adı” şeklinde derlenir. Eclipse, IntelliJ IDEA, Visual Studio Code gibi bir IDE’ye sahipseniz bu işlemler çok daha kolay gerçekleştirilir.

1.13.1. BigDecimal ve BigInteger Sınıfları

BigInteger ve BigDecimal veri tipleri java programlama dilinde tanımlanmış olan, ilkel veri tiplerine göre daha kapsamlı veri saklanması olanak sağlayan veri tipleridir.

BigInteger, Integer ilkel veri tipi ile -2^{31} (-2147483648) ile $2^{31}-1$ (2147483647) sayıları dahil olmak üzere bu sayılar arasında tanımlanabilen bir tam sayı değerinin aksine herhangi büyüklükteki bir tamsayı değerinin saklanması imkanı vermektedir.

BigDecimal, Double ilkel veri tipi ile -1.8×10^{308} ile $+1.8 \times 10^{308}$ sayıları dahil olmak üzere bu sayılar arasında tanımlanabilen bir ondalık sayı değerinin aksine herhangi büyüklükteki bir ondalık sayı değerinin saklanmasına imkan vermektedir.

Java'da BigDecimal sınıfı, yüksek hassasiyetli aritmetik işlemler için kullanılmaktadır. BigDecimal sınıfının içerdiği bazı metotlar şunlardır:

- `signum()` metodu, bir sayının işaretini (pozitif, negatif veya sıfır) belirlemek için kullanılır. Sayı pozitif ise 1, negatif ise -1 ve 0 ise 0 dönmektedir.
- `negate()` metodu mevcut BigDecimal nesnesinin negatifini döndürür.
- `multiply()` metodu, iki BigDecimal nesnesini çarpmak için kullanılır.

Java'da BigInteger sınıfı, çok büyük tamsayılarla çalışmak için kullanılmaktadır.

BigInteger sınıfının içerdiği bazı metotlar şu şekildedir:

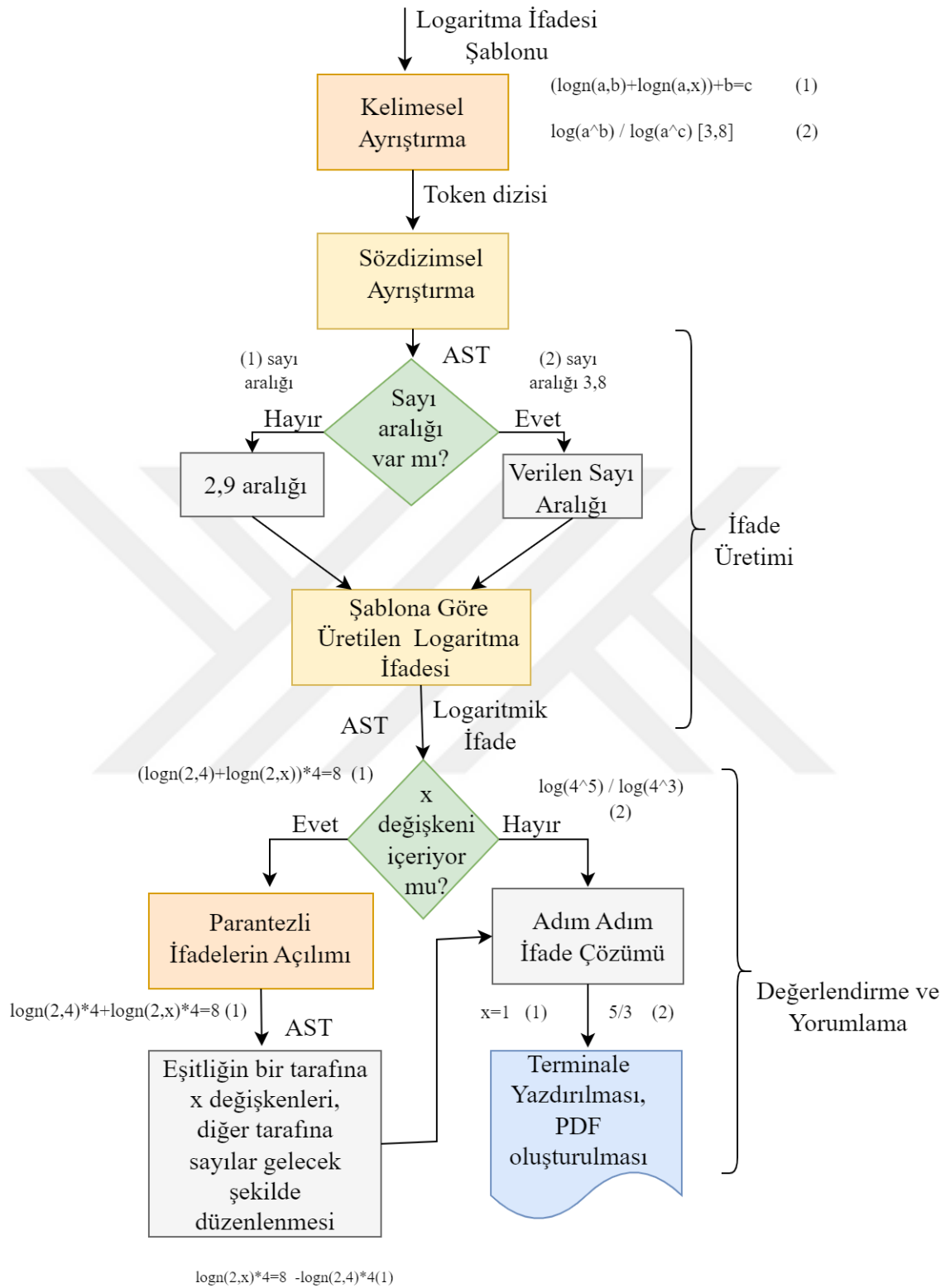
- `divide()` metodu, iki BigInteger nesnesini bölmek için kullanılmaktadır. Tamsayı bölme işlemi gerçekleştiğinden kalan kısmı dikkate alınmaz.
- `signum()`, `negate()`, `multiply()` metotları da BigDecimal sınıfının metotları gibidir.

2. YAPILAN ÇALIŞMALAR

2.1. Giriş

Bu tez çalışmasında logaritmik ifadeler için cebirsel fonksiyon yapılarından türetilen bir sözdizimine karşılık gelen EBNF benzeri notasyonunda bir LL(k) grameri tasarlanmıştır. LL(k) grameri için ihtiyaç duyulan ifade ayrıştırıcısının kaynak kodu, JavaCC formatında oluşturulmuş ve Java programlama dilinde üretilmiştir. Çeşitli ifade şablonlarını destekleyebilen gramer, değişkenli veya değişkensiz logaritma sorularını üretecek biçimde geliştirilmiştir. Gramer kuralları üzerinde gerçekleştirilen farklı genişletmeler ile temel logaritmik ifade bileşenleri birleştirilerek matematiksel denklemler üretilmiştir. Bu denklemlerin genel biçimlerini temsil eden diğer bir biçimsel gramer ile çözüm süreci yönetilmektedir. Ayrıştırıcı metotları, logaritmik ifadenin bileşenlerini sözdizimsel olarak analiz ederken, aynı zamanda ifadenin taşıdığı anlama uygun hiyerarşik bir soyut sözdizim ağacı oluşturmaktadır. Sözdizim ağacının değerlendirilmesi sürecinde, instanceof işleci ve sözdizim sınıflarına metot ekleme yöntemlerinden yararlanılmıştır.

Tek değişkenli ve değişkensiz denklemlerin üretimi için logaritmik ifade şablonları kullanılacak olup bu şablonların hazırlanmasında lise matematik müfredatındaki logaritma soru türleri ve örnekleri önemli bir rol oynamaktadır. Belirli bir şablona göre oluşturulan logaritma problemlerinin işlem adımlarını içeren çözümlerinin bir terminal üzerinde gösterimi ve ayrıca bir pdf dokümanına aktarımı sağlanmıştır. Pdf dokümanlarının üretiminde matematiksel metinleri ve formülleri biçimlendirmek için kullanılan Latex formatlama dilinden yararlanılmıştır. Sorunun çözüm süreci, ilgili ifadenin sadeleştirilmesi veya değişken değerinin belirlenmesi şeklinde icra edilmektedir.



Şekil 4. Uygulama genel yapısı

Şekil 4'te uygulamanın genel yapısını gösteren algoritmaya ilk olarak logaritma ifadelerini içeren bir şablon girdi olarak verilmiştir. Sırasıyla kelimesel, sözdizimsel ve

anlamsal analiz adımları gerçekleştirilmiştir. Gramerin içerdği fonksiyon ve operatörler için tanımlanan sözdizim sınıfları ile AST denilen hiyerarşik bir veri yapısı oluşturulmuştur. Bu veri yapısı üzerinde gezinilerek, belirli bir şablondan okunan değişkensiz veya x değişkenine dayalı logaritma ifadelerinin verilen sayı aralığına ya da varsayılan sayı aralığına göre üretimi gerçekleştirilmiştir. Sonrasında ise x değişkeni içeren ifadeler parantezli ifade açılımı, x değişkenlerinin eşitliğinin bir tarafına toplanacak şekilde yeniden oluşturulmuş ve işlem adımlarını içeren ifade çözümü gerçekleştirilmiştir. Ayrıca x değişkeni içermeyen yani değişkensiz ifadelerin adımsal çözümü gerçekleştirilmiştir. Son olarak ifadelerin ve çözümlerin terminale yazdırılması ve .pdf uzantılı bir doküman biçiminde oluşturulması sağlanmıştır.

2.2. Gramer Tasarımı

Çalışma kapsamında desteklenen değişkensiz ve x değişkenine dayalı örnek girdi verileri Tablo 8’de gösterilmiştir.

Tablo 8. Girdi verisi örnekleri

Biçim 1	Biçim 2
$a * \log(x) + b * \log(x) = c$ $a^{\logn(b,x)} + x^{\logn(b,a)} = c$ $\log(x+1) + \log(x-1) = a$ $a * \logn(3, 3^b)$ $c * \logn(d, x) = e * \logn(2, 2^f)$	$\log(a) + \log(a^b) + \log(a^c)$ $\logn(a, b) * \logn(b, c)$ $\log(a^b) / \log(a^c)$ $\ln(a) + \ln(a^b) + \ln(a^c)$ $(\logn(a, b) + \logn(a, c)) / \logn(a, c)$

Logaritmik denklemler üretmek için kullanılan LL(1) gramer tasarımı ise Tablo 9’da verilmiştir. Bu kurallara göre bir ifade içerisinde bulunabilecek fonksiyon, işleç, parantez, sayı gibi belirteçler tırnak işareti kullanılarak düzenlenmiştir. Gramerde “L” başlangıç simgesini, “Q” isteğe bağlı olarak eşitlik içeren bir ifadeyi, “R” sayı aralıklarını, “E” aritmetik ifadeyi, “T” terimleri (çarpma, bölme içeren ifadeyi), “P” üstel işlem içeren ifadeyi, “F” elemanları (temel ifadeleri) belirtmektedir.

Tablo 9. Gramer tasarımı

$L \rightarrow (Q)^*$
$Q \rightarrow E ("=" E)? (R)?$
$R \rightarrow "[" \text{ num } " , " \text{ num } "]"$
$E \rightarrow [" - " "+"] T (("+" "-") T)^*$
$T \rightarrow P (("*" "/") P)^*$
$P \rightarrow F ["^" P]$
$F \rightarrow \text{id} \text{num} "(" E ")"$
$F \rightarrow \text{"ln"} "(" E ")"$
$F \rightarrow \text{"log"} "(" E ")"$
$F \rightarrow \text{"logn"} "(" E " , " E ")"$

Bu gramer ile geniş bir yelpazede aritmetik ifadeler tanımlanabilmekte ve sayılar, aritmetik işlemler, üslü ifadeler, parantezler ve çeşitli logaritma fonksiyonları kullanılarak karmaşık ifadeler oluşturulabilmektedir.

2.3. Sözcük Üretici

Tablo 9’da gösterilen gramerin içerdiği sonlu sözcüklere (terminal) karşılık gelen birim sözcük tanımlamaları Tablo 10’da verilmiştir.

Tablo 10. JavaCC formatında birim sözcük tanımlamaları

```

TOKEN : /* OPERATORS */
{< PLUS: "+" > | < MINUS: "-" >
 | < TIMES: "*" > | < DIVIDE: "/" >
 | < POWER: "^" > | < LN: "ln" >
 | < LOG: "log" > | < LOGN: "logn" >
 | < COMMA: "," > | < EQUAL: "=" >
 | < LPR: "(" > | < RPR: ")" >
 | < LSR: "[" > | < RSR: "]" >
 | < ID: ([ "a" - "z" , "A" - "Z" ] ([ "a" - "z" , "A" - "Z" , "0" - "9" ] ) ) * >
 | < NUM: ([ "0" - "9" ] ) + >

```

Tablo 10'un devamı

< DNUM : (["0"-"9"])+ ("." (["0"-"9"])+)? >
< EOL : "\r\n" >}
SKIP :
{ " " "\t" "\r" "\n" < "--" (~ ["\r"]) * ("\r" "\r\n") > }

Tablo 11'de ise logaritma problemleri için üretilecek birim sözcük ifadeleri verilmiştir.

Tablo 11. Logaritma problemleri için birim sözcük ifadeleri.

Logaritma İfadeleri	Birim Sözcük Tanımlamaları
$\log(x) + \log(a) = c$	LOG LPR ID RPR PLUS LOG LPR ID RPR EQUAL ID
$a^{\log_n(b,x)} + x^{\log_n(b,a)} = c$ [3,8]	ID POWER LOGN LPR ID COMMA ID RPR PLUS ID POWER LOGN LPR ID COMMA ID RPR EQUAL ID LSR DNUM COMMA DNUM RSR
$\ln(a) + \ln(a^b) + \ln(a^c)$	LN LPR ID RPR PLUS LN LPR ID POWER ID RPR PLUS LN LPR ID POWER ID RPR

2.4. Ayrıştırıcılar

Logaritma ifadeleri için ayrıştırıcı metot tanımlamaları Tablo 12'de verilmiştir; parse(), ayrıştırıcının başlangıç metodunu ifade etmekte olup ayrıştırma işlemi <EOF> veya <EOL> birim sözcük dizilerinden biriyle karşılaşınca kadar devam etmektedir. Bu tanımlamalarda metot geri dönüş veri türleri ve sözdizim ağacını üreten Java ifadeleri gösterilmiştir.

Tablo 12. Ayrıştırıcı metotları

<pre> LogEqnList parse() : { LogEqnList el; } { el=L() { return el; }} LogEqnList L() : { LogEqnList el = new LogEqnList(); Exp e; } { (e=Q() { el.add(e); } ((<EOL>)+ <EOF> { return el; }))* { return el; }} Exp Q() : { Token t1, t2; Exp e1, e2=null; int n1=0, n2=0; } {e1=E() (<EQUAL> e2=E())? (<LSR> t1=<NUM> <COMMA> t2=<NUM> <RSR> { n1 = Integer.parseInt(t1.image); n2 = Integer.parseInt(t2.image); })? { return new LogEqn(e1, e2, n1, n2); } } </pre>	<pre> Exp T(): { Exp e1, e2; int k = 0; } {e1=P() (<TIMES> e2=P() { e1 = new Times(e1, e2); } <DIVIDE> e2=P() { e1 = new Divide(e1, e2); })* { return e1; } Exp F(): { Token t; Exp e1,e2; } { t=<ID> { return new Id(t.image); } t=<NUM> { return new Num(Integer.parseInt(t.image)); } <LPR> e1=E() <RPR> { return new Par(e1); } <LN> <LPR> e1=E() <RPR> { return new Ln(e1); } <LOG> <LPR> e1=E() <RPR> { return new Log(e1); } </pre>
---	---

2.5. Sözdizim Sınıfları

Logaritmik fonksiyon üretiminde kullanılan LogEqn, Log, Logn ve Num gibi bazı sözdizim sınıfı tanımlamaları Tablo 13'te gösterilmiştir.

Tablo 13. Sözdizim sınıfları

<pre> class LogEqn extends Exp { Exp e1, e2; public LogEqn(Exp a, Exp b) { e1 = a; e2 = b; } } class Log extends Exp { Exp e1, e2; public Log(Exp a, Exp b) { e1 = a; e2 = b; } ... </pre>	<pre> class Logn extends Exp { Exp e1, e2; public Logn(Exp a, Exp b) { e1 = a; e2 = b; } } class Num extends Exp { BigInteger n; public Num(int a){ n=BigInteger.valueOf(a); } public Num(BigInteger a){ n = a; } </pre>
---	---

2.6. BigDecimal ve BigInteger Sınıflarının Entegrasyonu

Uygulamadaki hesaplamalar çok geniş aralıkta, sayı değerlerinin tanımlanmasını gerektirdiği için ilkel veri tipi karşılıklarının genişletilmiş halleri olan BigInteger ve BigDecimal veri tiplerinden yararlanılmıştır. Tablo 13'te verilen Num sınıfı tanımlamasında BigInteger tipinde bir değişken tanımlanmıştır. Bu değişkene değer ataması yapılırken BigInteger.valueOf(a) metodundan yararlanılmıştır. Ayrıca uygulamanın birçok yerinde BigInteger ve BigDecimal sınıflarının signum, negate, multiply gibi metotları kullanılarak hesaplamalar gerçekleştirilmiştir.

2.7. Soyut Sözdizim Ağacı

Uygulamada üretilen logaritma ifadeleri için üretilen soyut sözdizim ağaçları Tablo 14'te verilmiştir. $a^{\log_n b^x} + x^{\log_n b^a} = c$ ve $a * \log(b * x) + c * \log(b^d) = e * \log(b^f)$ şeklindeki logaritmik denklemlerden üretilen ifadeler ve sözdizim ağaçları gösterilmiştir.

Tablo 14. Üretilen örnek sözdizim ağaçları

Soru Şablonu	$a^{\log n(b,x)} + x^{\log n(b,a)} = c$
Üretilen İfade	$4^{\log n(3,x)} + x^{\log n(3,4)} = 6$
Söz Dizim Ağacı	LogEqn(Plus(Power(Ratio(RDec(4),RDec(1)),Logn(Ratio(RDec(3),RDec(1)), Id(x))),Power(Id(x),Logn(Ratio(RDec(3),RDec(1)),Ratio(RDec(4),RDec(1))))),Ratio(RDec(6),RDec(1)),0,0)
Soru Şablonu	$a * \log(b * x) + c * \log(b^d) = e * \log(b^f)$
Üretilen İfade	$7 * \log(7 * x) + 6 * \log(7^6) = 4 * \log(7^5)$
Söz Dizim Ağacı	LogEqn(Plus(Times(Ratio(RDec(7),RDec(1)),Log(Times(Ratio(RDec(7),RDec(1)),Id(x)))),Times(Ratio(RDec(6),RDec(1)),Log(Power(Ratio(RDec(7),RDec(1)),Ratio(RDec(6),RDec(1))))),Times(Ratio(RDec(4),RDec(1)),Log(Power(Ratio(RDec(7),RDec(1)),Ratio(RDec(5),RDec(1))))),0,0)

2.8. Logaritma İfadelerinin Üretimi ve Değerlendirilmesi

Kaynak kodun değerlendirilmesi sırasında, ayrıştırıcı tarafından oluşturulan sözdizim ağacından yararlanılmaktadır. Bu ağacın içerdiği logaritma ifadesinde yer alan a, b ve c gibi parametrelere atanacak tamsayı değerleri için Tablo 15'teki Table sınıfı ile tanımlanmıştır.

Her bir örnek logaritma ifadesinin üretiminde bu parametrelere atanan (isim, değer) çifti biçimindeki değerler sembol tablosunda saklanmaktadır.

Tablo 15. Sembol tablosu

<pre> class Table {Hashtable table = new Hashtable(); public void put(String id, Ratio val) { table.put(id, val); } public Ratio get(String id) { return (Ratio)table.get(id); }} </pre>
--

Uygulamada şablona dayalı logaritma sorularının üretimini kolaylaştırmak için soyut (abstract) sınıf ve metotlar kullanmıştır. Bu soyutlaştırma sözdizim sınıflarının her birine uygulanmıştır. Soyut metotlardan türetilen her metot o sözdizim sınıfına göre kodlanarak somutlanmıştır.

Tablo 16. Exp soyut sınıfı

```
abstract class Exp {
    public abstract Exp getE1();
    public abstract Exp getE2();
    public abstract String toClass();
    public abstract Exp insert(Table t);
    public abstract Object accept(Visitor v);
    public abstract void addIDs(ArrayList<String> vs);
    public abstract Exp evalLog(Table t);
    public Ratio eval(Table t) { return new Ratio(); }
    public BigDecimal getDecimal() { return BigDecimal.ZERO; }
    public String toLatex(boolean x) { return ""; }
}
```

Tablo 16’da Exp soyut (abstract) sınıfı gösterilmiştir. getE1() ve getE2() metotları, bir düğümün sol ve sağ alt düğümlere geçişini sağlamaktadır. Uygulamanın geliştirme sürecinde kullanılan toClass() metodu, sözdizim ağacını metinsel formda üreterek kodlama faaliyetlerinin daha kolay yönetilebilmesine yardım eder. addIDs() metodu logaritmik şablonda kullanılan değişkenleri bir listeye eklememizi sağlamaktadır. Soru üretimi aşamasında, insert() metodu ile bu listedeki değişkenlere değer atanarak logaritma ifadesi karşılığı olan bir Exp nesnesi oluşturulmaktadır. evalLog() metodu sembol tablosunu kullanarak bu logaritma ifadesini sadeleştirirken, eval() metodu yine sembol tablosunu kullanarak ilgili ifadenin değerlendirmesini veya çözümünü gerçekleştirmektedir. toLatex() metodu ise kaynak verinin nesne temsilini Latex formatına dönüştürmektedir. Exp sınıfından türetilen sınıf örneği (Logn) Tablo 17’de verilmiştir.

Tablo 17. Logn sınıfı

<pre> class Logn extends Exp { Exp e1; Exp e2; public Logn(Exp a, Exp b) { e1 = a; e2 = b; } public Exp getE1() { return e1;} public Exp getE2() { return e2;} public String toLatex(boolean x) { return "log_{"+e1.toLatex(x)+"}("+ e2.toLatex(x)+")";} public String toClass() { return "Logn("+e1.toClass()+","+ e2.toClass()+")";} public Exp insert(Table t) { return new Logn(e1.insert(t),e2.insert(t)); } </pre>	<pre> public Exp evalLog(Table t) { return new Logn(e1.evalLog(t), e2.evalLog(t)); } public Ratio eval(Table t) { return new Ratio(Math.log(e2.eval(t).getDecimal().doubleValue()) / Math.log(e1.eval(t).getDecimal().doubleValue())); } public String toString() { return "logn(" + e1 + "," + e2 + ")";} public Object accept(Visitor v) { return v.visit(this);} public void addIDs(ArrayList<String> vs) { e1.addIDs(vs); e2.addIDs(vs); } </pre>
--	---

2.8.1. Logaritma İfadelerinin Üretimi

Logaritma ifadelerinin üretimi aşamasında ilk olarak bir metin dosyasından logaritma şablonları okunarak bir listeye aktarılır. Burada her bir şablon ifadede bulunan parametreler ilk önce sembol listesine eklenir. Daha sonra, x hariç tüm değişkenlere 2 ile 9 arasında veya belirtilen bir aralıkta rastgele sayılar atanır. Üretilen sayıların benzersizliği denetlenmektedir. Aynı şablona dayalı diğer bir örnek ifade üretimi için sembol tablosuna yeni parametre değerleri eklenmelidir. Uygulama her soru kalıbına göre değişik sayıda soru üretebilecek biçimde yapılandırılabilir.

2.8.2. Logaritma İfadelerinin Değerlendirilmesi

Logaritmik ifadelerin değerlendirmesi kolaydır. İfade bileşenleri logaritmik prensipler çerçevesinde birbirine bağlanarak çözüme ulaşılabilmektedir. Denklem çözüm

süreci nispeten daha zordur ve ilgili şablonun karmaşıklığına bağlı olarak çözümü mümkün yapmayabilmektedir. Genel olarak, basit denklem şablonları için çözüm süreci birkaç adımdan oluşur. İlk olarak denklemin içerdiği parantezli ifadelerin açılımı yapılarak sözdizim ağacı yeniden düzenlenir. Ardından denklemin bir tarafında x değişkenlerini içeren ifade bileşenleri toplanırken, diğer tarafına sadece sayılardan oluşan ifadeler taşınır. İfade değerlendirmesinde olduğu gibi iki taraftaki terimler kendi aralarında birleştirilerek çözümü yapılabilen basit bir cebirsel ifade elde edilir.

2.8.3. Parantez Açılımı Yapılarak Logaritma İfadelerin Genişletilmesi

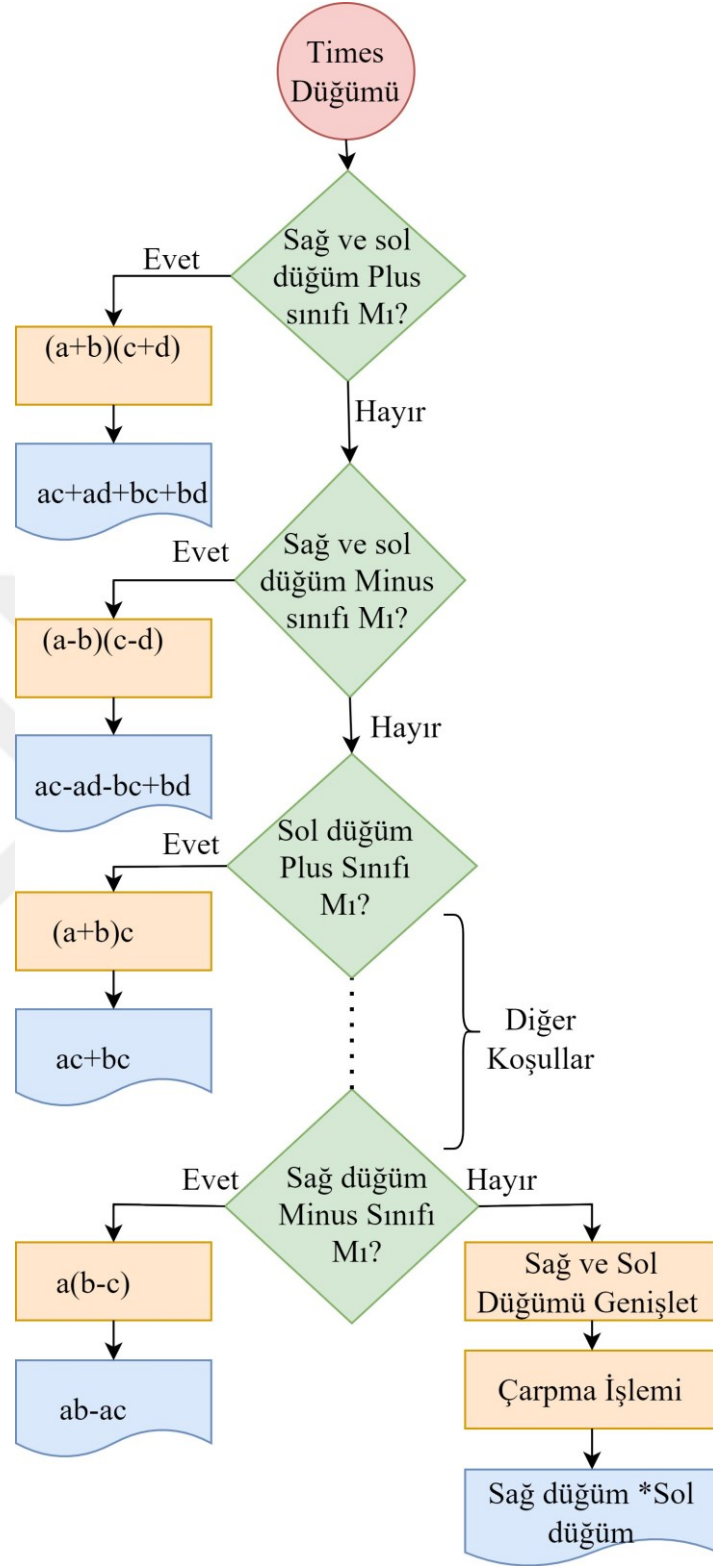
Logaritma ifadelerinin çözüm aşamasında ilk olarak parantezli bir ifadenin açılımı gerçekleştirilmektedir. Söz dizim ağacında söz konusu dönüşümleri gerçekleştiren ASTExpand() metodu Tablo 18’de gösterilmiştir.

Tablo 18. ASTExpand() metodu

```
public Exp ASTExpand(Exp e) {   Exp e1, e2;
    if (e instanceof Par) e = e.getE1();
    if (e instanceof Plus) { e1 = ASTExpand(e.getE1());
        e2 = ASTExpand(e.getE2());
        return new Plus(e1, e2);
    } .....
    if (e instanceof Times) { //(a+b)(c+d)=ac+ad+bc+bd
        if (e.getE1() instanceof Plus && e.getE2() instanceof Plus)
            return ASTExpand(new Plus(new Plus(new Plus(new Times(e.getE1().getE1(),
                e.getE2().getE1()), new Times(e.getE1().getE1(), e.getE2().getE2())),
                new Times(e.getE1().getE2(), e.getE2().getE1())),
                new Times(e.getE1().getE2(), e.getE2().getE2())));
        .....
        e1 = ASTExpand(e.getE1());   e2 = ASTExpand(e.getE2());
        ..... return e;   }
```

ASTExpand() metodu sırasıyla aşağıdaki işlemleri gerçekleştirir.

1. İfade parantez içindeyse (Par), alttaki ifadeyi ortaya çıkarmak için parantezler kaldırılmaktadır.
2. Toplama (Plus), çıkarma (Minus), çarpma (Times) ve bölme (Divide) gibi ifadeler için, alt ifadelerini yinelemeli olarak genişleterek ve aynı türden yeni nesneler oluşturarak genişletilmiş alt ifadelerle işlemektedir.
3. Parantezli ifadeler açılırken çarpma işleminin dağılma özelliğinden yararlanılmaktadır. Şekil 5'te sağ ve sol düğümün tipine göre gerçekleştirilen yöntemler gösterilmiştir. Örneğin sağ ve sol düğümün Plus sınıfı içerdiği $(\log 10 + \log 100) * (2 + \log 10^3)$ ifadesinin açılımı $\log 10 * 2 + \log 10 * \log 10^3 + \log 100 * 2 + \log 100 * \log 10^3$ şeklinde yapılarak genişletilmektedir. Sol düğümün Minus tipinde olduğu durumlarda $(\log_2 8 - x) * \log_3 9$ ifadesi $\log_2 8 * \log_3 9 - x * \log_3 9$ şeklinde açılarak genişletilmektedir. Bu koşulların geçerli olmadığı durumda alt düğümler yinelemeli olarak genişletilmekte ve sonrasında bu genişletilmiş ifadeler çarpılmaktadır.
4. Yukarıdaki koşullardan hiçbiri geçerli değilse (örneğin, ifade bir değişken veya sabit ise), işlev herhangi bir değişiklik yapmadan orijinal ifade döndürülmektedir.



Şekil 5. Çarpma işleminin dağılma özelliği

2.8.4. Bilinmeyenlerin Eşitliğin Bir Tarafına Toplanması

Bilinmeyenli (x değişkenli) logaritma ifadelerinin çözümü için sözdizim ağacının düzenlendiği `ASTArrange()` metodu Tablo 19’da verilmiştir. `ASTArrange()` metodu çağrılırken eşitliğin sağ ve sol düğümleri birbirinden çıkarılmaktadır. Sonrasında ise değişkenler eşitliğin bir tarafına sayılar ise diğer tarafına gelecek şekilde sözdizim ağacı düzenlenmektedir. Örneğin $\log(x) + \log 10 = 2$ ifadesi $\log(x) = 2 - \log 10$ olacak şekilde düzenlenmektedir. $5 * \log_2 x - \log_3 9 = \log_2 x + \log_3 27$ ifadesinin düzenlemesi ise $5 * \log_2 x - \log_2 x = \log_3 27 + \log_3 9$ biçiminde yapılmaktadır.

Tablo 19. `ASTArrange()` metodu

```
ASTArrange (new Minus(e2.getE1(), e2.getE2()), 1);
.....
public void ASTArrange(Exp e, int n) {
    if (e instanceof Par) e = e.getE1();
    if (e instanceof Plus) {
        ASTArrange(e.getE1(), n);
        ASTArrange(e.getE2(), n);
    } else if (e instanceof Minus) {
        ASTArrange(e.getE1(), n);
        ASTArrange(e.getE2(), (1) * -n);
    } else {
        boolean x = IdCheck(e);
        int r = x ? 0 : 1;
        if (eqs.get(r) instanceof Ratio && ((Ratio)
eqs.get(r)).getR1().compareTo(BigDecimal.ZERO) == 0)
            eqs.set(r, (x == n > 0) ? e : new Times(new Ratio(-1), e));
        else
            eqs.set(r, (x == n > 0) ? new Plus(eqs.get(r), e) : new Minus(eqs.get(r),
e));
    }
}
```

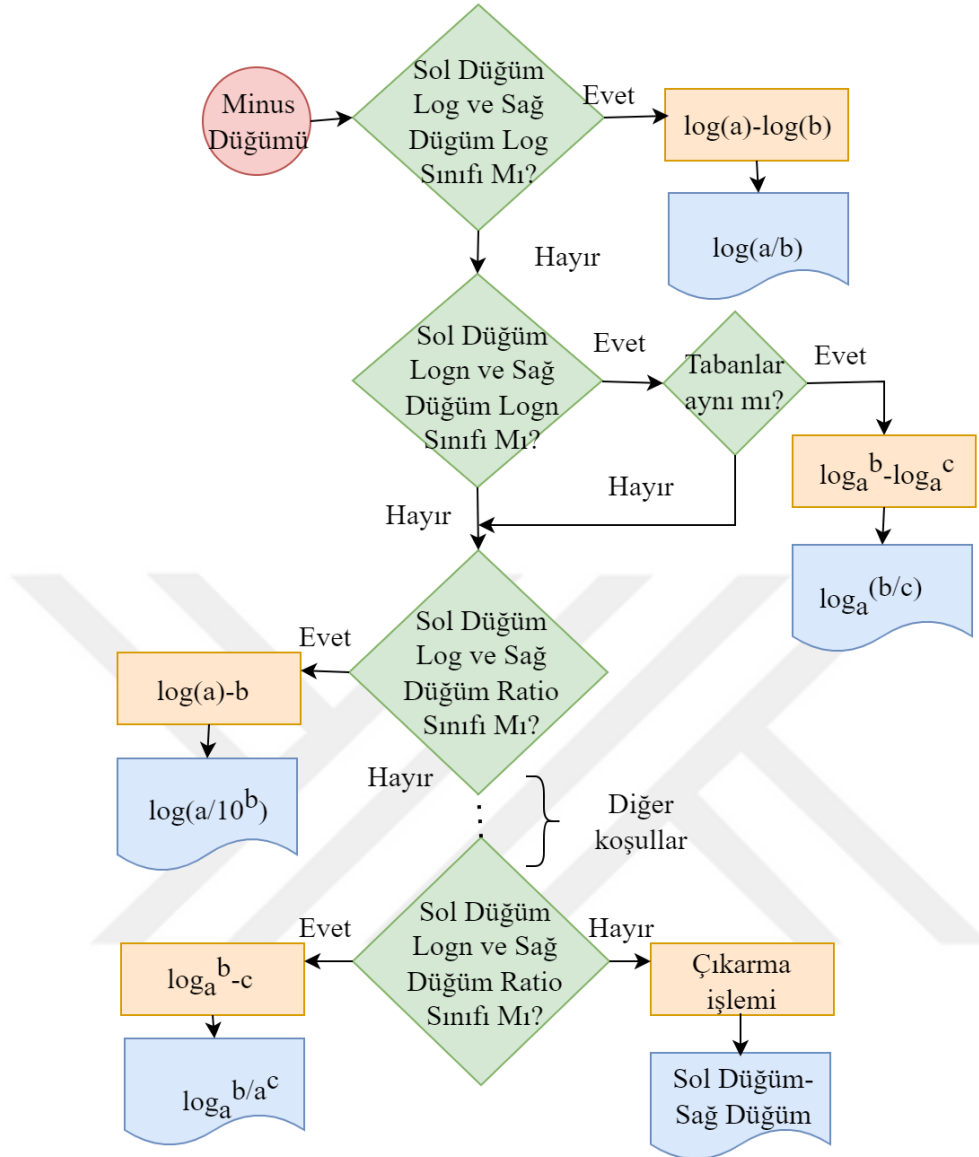
2.8.5. Logaritmik İfadelerin Hesaplanması

Parantezlerin kaldırılması ve bilinmeyenlerin eşitliğinin bir tarafına toplanması aşamasından sonra logaritma ifadelerinin hesaplaması aşaması gerçekleştirilmektedir. Bu aşama değişken içermeyen ifadeler için logaritma kurallarına göre yürütülmektedir. Bilinmeyen içeren ifadelerde işlemler biraz daha karmaşıktır. İlk aşamada eşitliğin bir tarafına x bilinmeyenini toplandığı ve diğer tarafına sayıların toplandığı ifadeleri ayrı ayrı logaritma kurallarına göre hesaplanıp yeni bir eşitlik oluşturulmaktadır. Sonrasında ifadenin sağ ve sol düğümünün Log, Ln ve Logn türünde olup olmadığı kontrol edilerek ifadeler hesaplanmaktadır.

2.9. Özel Durumlar

2.9.1. Logaritma Toplama Kuralı

Logaritma ifadeleri hesaplanırken toplama kuralı için, Şekil 6'daki algorithmadan yararlanılmıştır. Algorithmaya göre eğer düğüm Plus düğümü ise sağ ve sol düğümler kontrol edilmektedir. Düğümlerin tipine göre logaritma ifadelerinin kuralları işlemektedir. Örneğin; sağ ve sol düğüm Log sınıfı tipinde ise burada $\log(a)+\log(b)$ ifadesinin eşdeğeri $\log(a.b)$ şeklinde olacaktır. Tüm koşullar adım adım kontrol edilip bu koşullardan hiçbiri sağlanmaz ise ifade düğümlerin toplamına eşit olmaktadır.

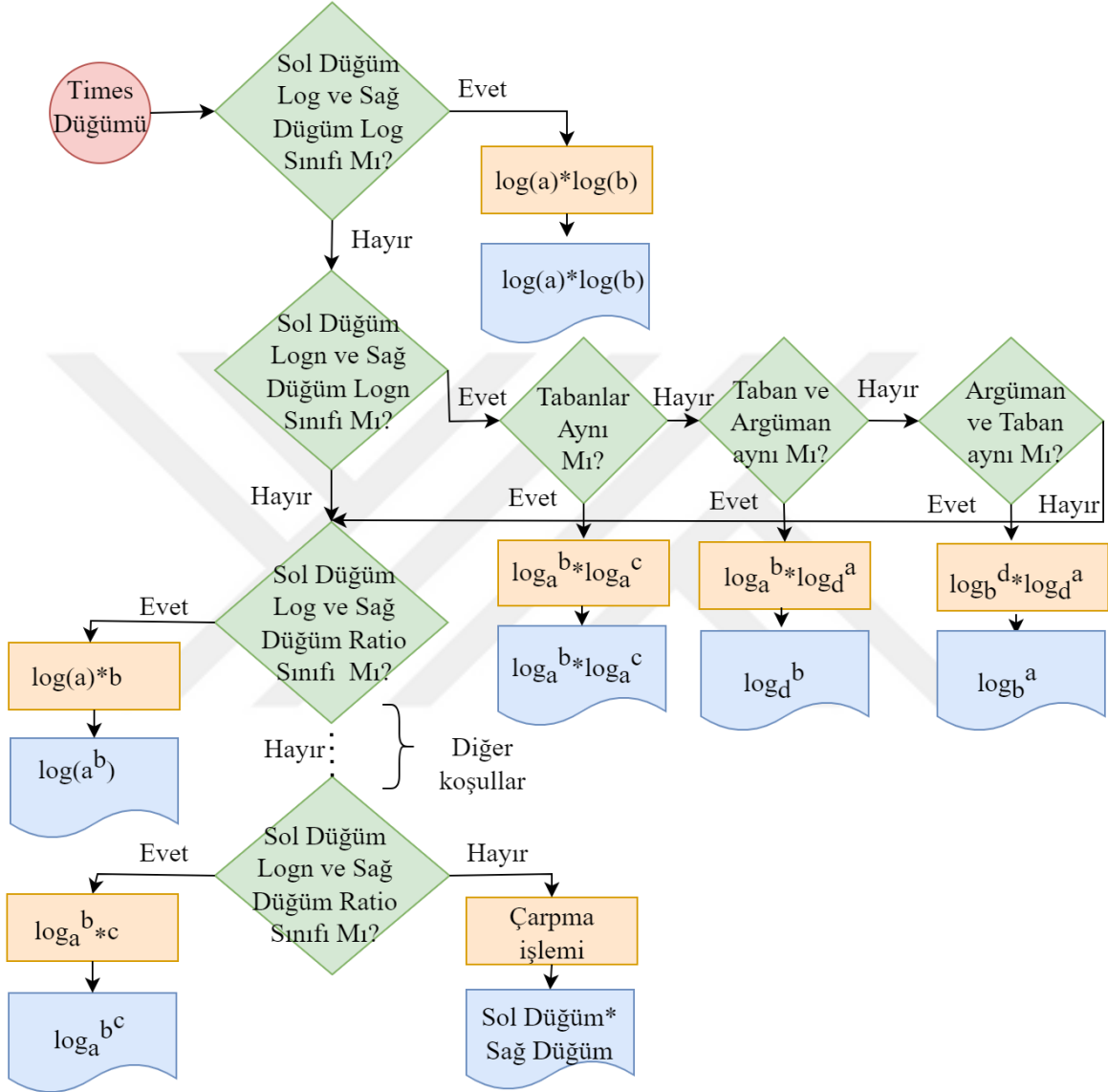


Şekil 7. Çıkarma kuralı algoritması

2.9.3. Logaritma Çarpma Kuralı

Logaritmik ifadeler için çarpma işleminin kurallarını ifade eden algoritması Şekil 8’de verilmiştir. Buna göre Times düğümünün sağ ve sol düğümlerinin belirttiği nesne türlerine göre hangi işlemin yapılması gerektiği örneklerle belirtilmiştir. Sağ düğümün Log tipinde sol düğümün ise Ratio tipinde nesneler içermesi durumunda $\log(a)*b$ ifadesi $\log(a^b)$ ifadesine dönüştürülecekti. Sağ ve sol düğümlerin her ikisi de Logn tipinde nesnelere sahipse, bir düğümdeki taban değerinin diğer düğümdeki argüman ile aynı olup olmadığına

göre karar vererek işlemleri gerçekleştirmektedir. Bu koşulların hiçbirisi sağlanmazsa ifade iki düğümün çarpımına eşittir.

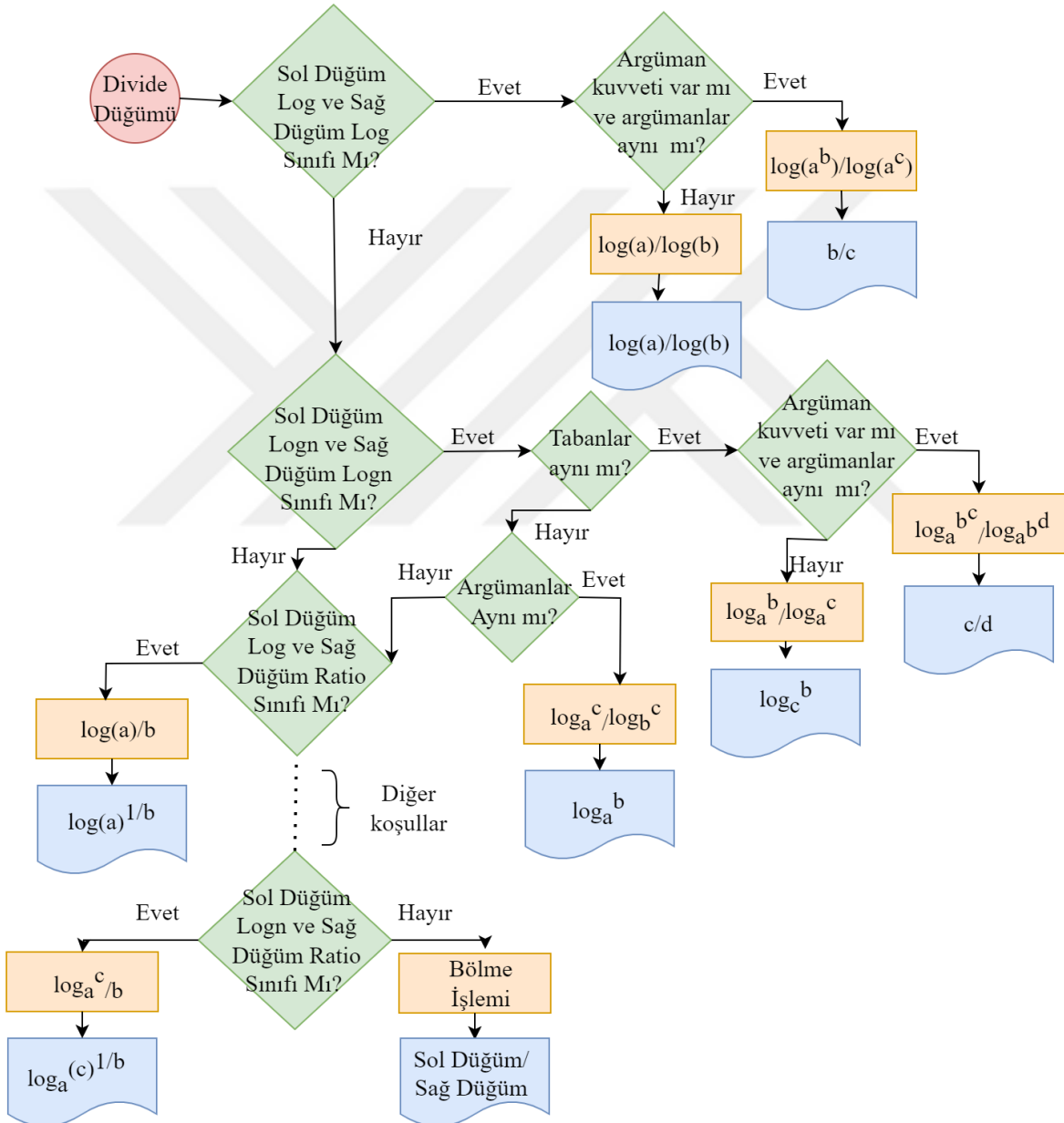


Şekil 8. Çarpma kuralı algoritması

2.9.4. Logaritma Bölme Kuralı

Logaritma ifadeleri için bölme işlemi kuralları Şekil 9’da ifade edilmiştir. Divide düğümünün sağ ve sol düğümlerinin tiplerine göre kurallar şekillenmektedir. Sağ ve sol düğümler Log tipinde nesne içeriyorsa, argümanların aynı olması ve üslerinin bulunması

durumunda eşdeğer ifade üslerin bölünmesi şeklinde hesaplanmaktadır. Aynı kural Logn sınıfı için de geçerli olmaktadır. Sağ ve sol düğümler Logn tipinde nesnelere sahipse, tabanların aynı olmadığı argümanların ise aynı olduğu durumda eşitlik iki ifadenin tabanlarının logaritmasına eşittir. Hiçbir koşulun sağlanmadığı durumda ifade iki düğümün bölümüne eşittir.



Şekil 9. Bölme kuralı algoritması

3. BULGULAR VE TARTIŞMA

3.1. Yorumlayıcı Bileşenlerinin Entegrasyonu

Bir metin dosyasından veya standart girişten okunan logaritma ifade şablonlarına dayalı örnek ifadelerin üretimi ve değerlendirilmesi ayrıştırıcı, değerlendirici ve PDF doküman üretici bileşenlerinin entegrasyonunu gerektirir. Bu işlemler LogExpParser, LogApp ve LogText sınıfları ile gerçekleştirilmiştir.

JavaCC formatında ayrıştırıcıyı tanımlayan log_exp.jj, yorumlayıcı sözdizim sınıflarını içeren AST.java, logaritma şablonlarını değerlendiren ve doküman üreten LogApp.java ve LogTex.java dosyaları, aynı dizinde bulunmalıdır. Belirtilen şablonlar için örnek ifadelerin üretilerek bir PDF dosyasına aktarımı bir terminal üzerinde aşağıdaki gibi yapılmaktadır.

```
$> javacc log_exp.jj
```

```
$> javac *.java
```

```
$> java LogTex
```

Benzer olarak LogApp ile üretilen logaritma ifadeleri ve çözümleri terminal üzerinde görüntülenmektedir.

Tablo 20'de verilen main() metodunda, ilk olarak bir metin dosyasından logaritma ifade şablonları okunmakta ve bu şablonlar, loglist nesnesinin içerdiği list listesine eklenmektedir. Ardından, logApp() metodu çağrılmaktadır. Bu metod, her bir şablon ifadede bulunan parametreleri vs adlı listeye eklemektedir. Örneğin, " $\log(a) + \log(b) + x = c$ " şablon ifadesi için a, b, c ve x değişkenleri listeye eklenmektedir. Sonraki aşamada, x değişkeni hariç tüm değişkenlere, eğer belirtilen bir sayı aralığı yoksa, 2 ile 9 arasında rastgele sayılar atanmaktadır. Üretilen sayıların benzersizliği ns listesi ile denetlenmektedir. Aynı şablonu kullanarak farklı bir ifade elde etmek için sembol tablosuna yeni sayısal değerler ataması yapılır. LogEqn, logaritmik denklemler için özel olarak tasarlanmış bir denklem sınıfıdır. Sonrasında, oluşturulan LogEqn türündeki ifadelerin sağ tarafında bir düğüm olup olmasına göre çözüm adımları gerçekleştirilir. Eğer ifade sağ tarafında bir düğüm içeriyorsa, bu genellikle x değişkenini içermesi anlamına gelir ve çözüm adımları buna göre daha farklı olacaktır. Sonuç olarak, ifade oluşturma ve çözüm adımları tamamlanır ve terminalde gösterilir.

Tablo 20. LogApp sınıfı

<pre> public static void main(String[] args) throws ParseException, FileNotFoundException { LogExpParser pr = new LogExpParser(new FileInputStream(new File("log.txt"))); LogEqnList loglist = pr.parse(); new LogApp().logApp(loglist); } public void logApp(LogEqnList loglist) { ArrayList<String> vs = new ArrayList<String>(); ArrayList<String> ns = new ArrayList<String>(); for (int i = 0; i < loglist.list.size(); i++) { vs.clear(); ((Exp) loglist.list.get(i)).addIDs(vs); } int MIN = 2, MAX = 9; Random rn = new Random(System.currentTimeMillis()); for (int i = 0; i < loglist.list.size(); i++) { if (leq.n1 > 0) { MIN = leq.n1; MAX = leq.n2; } int n; ns.clear(); </pre>	<pre> for (int k = 0; k < 5; k++) { Table table = new Table(); for (int j = 0; j < vs.size(); j++) { if (vs.get(j).equals("x")) continue; int n_count = 0 do { n = MIN + rn.nextInt(MAX - MIN + 1); LogEqn leq = (LogEqn) loglist.list.get(i); vs.clear(); leq.addIDs(vs); if (n_count++ > 9) break; } while (ns.contains(n + "")); ns.add(n + ""); table.put(vs.get(j), new Ratio(n)); } Exp e1 = leq.insert(table); Exp e2 = e1.evalLog(table); Exp logE; if (leq.e2 != null) { e2 = ASTExpand(e2); } else { logE = logExp(((LogEqn) e2).e1); } </pre>
--	--

3.2. Latex Entegrasyonu

LaTeX doküman formatlama işlemleri MiKTeX kütüphanesi yardımıyla yapılarak PDF dokümanları üretilmiştir. Tablo 21’de LaTeX formatına dönüşüm yapan bazı kod satırları gösterilmiştir. toLatex() metodu soyut sınıftan türetilmektedir ve her söz dizim sınıfı için o sınıfın özelliklerine göre tanımlanmaktadır.

Tablo 21. LaTeX tanımlamaları

```

Tex_Create();
PutTextLn("\\section{Logaritma ifadelerinin üretilmesi}");
.....
PutTextLn("\\n$\\textrm " + " Step 1:" + eqn.toLatex(false) + "$ ");
.....
PutTextLn("\\n$\\textrm " + " Solution:" + logE.toLatex(false) + "$ ");
PutTextLn("\\end{enumerate}");
PutTextLn("\\end{itemize} ");
.....

```

3.3. Uygulama Bulguları

Örnek bazı logaritma şablonlarının, üretilen logaritma ifadelerinin ve denklemlerinin çözümleri aşağıda verilmiştir.

Şablon 1: $\log(a) + \log(a^b) + \log(a^c)$

Şablon 1'e göre oluşturulan $\log(2) + \log(2^2) + \log(2^4)$ ifadesi $\log(2^7)$ ifadesine değerlendirilirken, uygulamada üretilen soru ve çözüm adımları Şekil 10'da verilmiştir.

1. $\log(2) + \log(2^5) + \log(2^9)$
Result : $\log(2^{15})$
2. $\log(7) + \log(7^3) + \log(7^4)$
Result : $\log(7^8)$
3. $\log(8) + \log(8^6) + \log(8^3)$
Result : $\log(8^{10})$
4. $\log(7) + \log(7^9) + \log(7^9)$
Result : $\log(7^{19})$
5. $\log(8) + \log(8^5) + \log(8^9)$
Result : $\log(8^{15})$

Şekil 10. Şablon 1'e göre üretilen ifadeler ve çözümleri

Şablon 2: $(\log_a x + \log_a y) * b = c$

Şablon 2'ye göre oluşturulan $(\log_2 x + \log_2 4) * 3 = 12$ ifadesi $3 * \log_2(4x) = 12 \Rightarrow 4x = 2^4 \Rightarrow x = 4$ şeklinde çözülürken, uygulamada üretilen soru ve çözüm adımları Şekil 11'de verilmiştir.

1. $(\log_3(3, 3) + \log_3(3, x)) * 3 = 4$
Step1 : $\log_3(3) * 3 + \log_3(x) * 3 = 4$
Step2 : $\log_3(x) * 3 = -1 * \log_3(3) * 3 + 4$
Evaluate : $\log_3(x^3) = \log_3(3^1)$
Solution : $x = ((3^1))^{\frac{1}{3}}$
Result : $x = 1.4422495702545926$
2. $(\log_4(4, 4) + \log_4(4, x)) * 4 = 2$
Step1 : $\log_4(4) * 4 + \log_4(x) * 4 = 2$
Step2 : $\log_4(x) * 4 = -1 * \log_4(4) * 4 + 2$
Evaluate : $\log_4(x^4) = \log_4(4^{-2})$
Solution : $x = ((4^{-2}))^{\frac{1}{4}}$
Result : $x = 0.5$
3. $(\log_3(3, 4) + \log_3(3, x)) * 4 = 4$
Step1 : $\log_3(4) * 4 + \log_3(x) * 4 = 4$
Step2 : $\log_3(x) * 4 = -1 * \log_3(4) * 4 + 4$
Evaluate : $\log_3(x^4) = \log_3(4^{-4} * 3^4)$
Solution : $x = ((4^{-4} * 3^4))^{\frac{1}{4}}$
Result : $x = 0.75$

Şekil 11. Şablon 2'ye göre üretilen ifadeler ve çözümleri

Şablon 3: $\log_a a^b + \log_e x = e * \log_c c^f$

Şablon 3'e göre oluşturulan $\log_2 2^8 + \log_4 x = 4 * \log_3 3^6$ ifadesi $8 + \log_4 x = 4 * 6 \Rightarrow 8 + \log_4 x = 24 \Rightarrow \log_4 x = 16 \Rightarrow x = 4$ şeklinde çözülürken, uygulamada üretilen soru ve çözüm adımları Şekil 12'de verilmiştir.

1. $\log_n(2, 2^4) + \log_n(3, x) = 3 * \log_n(5, 5^2)$
Step1 : $\log_2(2^4) + \log_3(x) = 3 * \log_5(5^2)$
Step2 : $\log_3(x) = -1 * \log_2(2^4) + 3 * \log_5(5^2)$
Evaluate : $\log_3(x) = \log_2(2^{-4}) + \log_5(5^6)$
Solution : $x = 3^{\log_2(2^{-4}) + \log_5(5^6)}$
Result : $x = 9$
2. $\log_n(4, 4^5) + \log_n(3, x) = 3 * \log_n(4, 4^2)$
Step1 : $\log_4(4^5) + \log_3(x) = 3 * \log_4(4^2)$
Step2 : $\log_3(x) = -1 * \log_4(4^5) + 3 * \log_4(4^2)$
Evaluate : $\log_3(x) = \log_4(4^1)$
Solution : $x = 3^{\log_4(4^1)}$
Result : $x = 3$
3. $\log_n(2, 2^5) + \log_n(5, x) = 5 * \log_n(4, 4^2)$
Step1 : $\log_2(2^5) + \log_5(x) = 5 * \log_4(4^2)$
Step2 : $\log_5(x) = -1 * \log_2(2^5) + 5 * \log_4(4^2)$
Evaluate : $\log_5(x) = \log_2(2^{-5}) + \log_4(4^{10})$
Solution : $x = 5^{\log_2(2^{-5}) + \log_4(4^{10})}$
Result : $x = 3125$

Şekil 12. Şablon 3'e göre üretilen ifadeler ve çözümleri

Şablon 4: $x = \frac{\log(a^b)}{\log(a^c)}$

Şablon 4'e göre oluşturulan $\frac{\log(2^5)}{\log(2^6)}$ ifadesi $\frac{5}{6}$ değerine eşit olurken, uygulamada üretilen soru ve çözüm adımları Şekil 13'te verilmiştir.

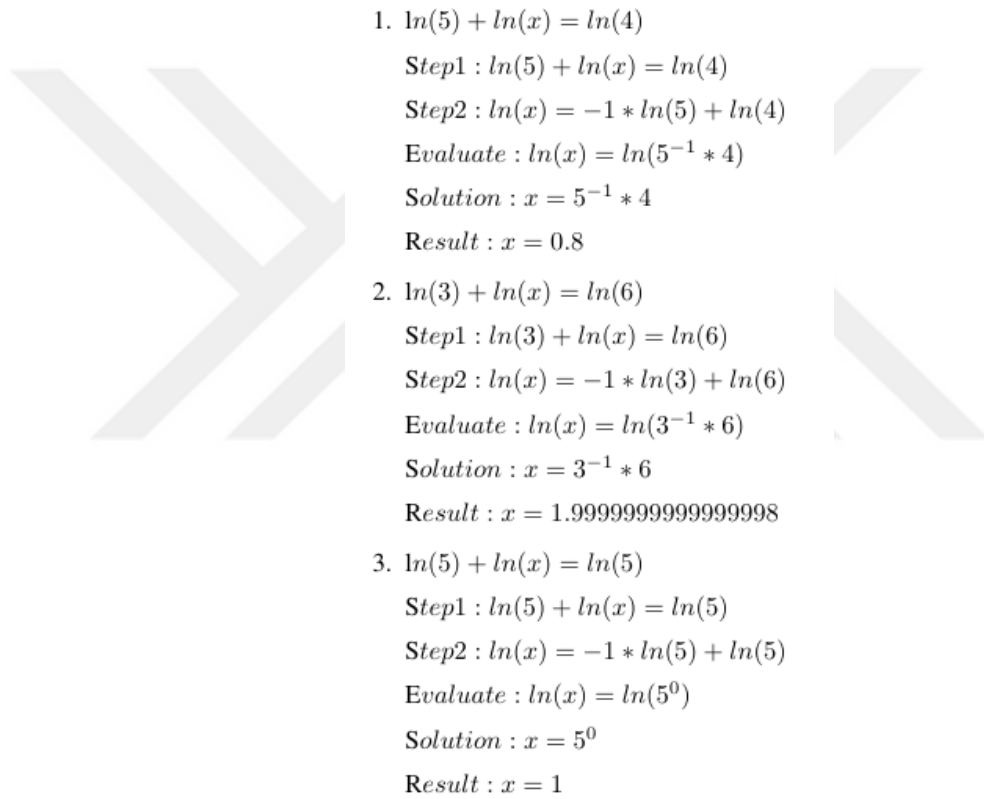
1. $\log(4^2)/\log(4^3)$
Result : $\frac{2}{3}$
2. $\log(4^3)/\log(4^2)$
Result : $\frac{3}{2}$
3. $\log(4^4)/\log(4^3)$
Result : $\frac{4}{3}$
4. $\log(2^4)/\log(2^2)$
Result : $\frac{4}{2}$
5. $\log(2^4)/\log(2^4)$
Result : $\frac{4}{4}$

Şekil 13. Şablon 4'e göre üretilen ifadeler ve çözümleri

Şablon 5: $\ln(a) + \ln(x) = \ln(b)$

Şablon 5'e göre oluşturulan $\ln(2) + \ln(x) = \ln(8)$ ifadesi $\ln x = \ln 8 - \ln 2 \Rightarrow$

$\ln x = \ln 8 - \ln 2 \Rightarrow x=3$ şeklinde çözülürken, uygulamada üretilen soru ve çözüm adımları Şekil 14'te verilmiştir.



1. $\ln(5) + \ln(x) = \ln(4)$
 Step1 : $\ln(5) + \ln(x) = \ln(4)$
 Step2 : $\ln(x) = -1 * \ln(5) + \ln(4)$
 Evaluate : $\ln(x) = \ln(5^{-1} * 4)$
 Solution : $x = 5^{-1} * 4$
 Result : $x = 0.8$
2. $\ln(3) + \ln(x) = \ln(6)$
 Step1 : $\ln(3) + \ln(x) = \ln(6)$
 Step2 : $\ln(x) = -1 * \ln(3) + \ln(6)$
 Evaluate : $\ln(x) = \ln(3^{-1} * 6)$
 Solution : $x = 3^{-1} * 6$
 Result : $x = 1.9999999999999998$
3. $\ln(5) + \ln(x) = \ln(5)$
 Step1 : $\ln(5) + \ln(x) = \ln(5)$
 Step2 : $\ln(x) = -1 * \ln(5) + \ln(5)$
 Evaluate : $\ln(x) = \ln(5^0)$
 Solution : $x = 5^0$
 Result : $x = 1$

Şekil 14. Şablon 5 'e göre üretilen ifadeler ve çözümleri

Şablon 6: $a * \log(b) - c * \log(x) - d * (\log(x) + \log(b)) = f$

Şablon 6'ya göre oluşturulan $5 * \log(10) - 1 * \log(x) - 3 * (\log(x) + \log(10)) = 6$ ifadesi $5 * \log(10) - 1 * \log(x) - 3 * \log(x) - 3 * \log(10) = 6 \Rightarrow$

$-4 * \log(x) = -2 * \log(10) + 6 \Rightarrow -4 * \log(x) = 4 \Rightarrow \log(x) = -1 \Rightarrow x=10^{-1}$ şeklinde çözülürken, uygulamada üretilen soru ve çözüm adımları Şekil 15'te verilmiştir.

1. $5 * \log(2) - 3 * \log(x) - 4 * (\log(x) + \log(2)) = 2$
Step1 : $5 * \log(2) - 3 * \log(x) - 4 * \log(x) + 4 * \log(2) = 2$
Step2 : $-1 * 3 * \log(x) - 4 * \log(x) = -1 * 5 * \log(2) + 4 * \log(2) + 2$
Evaluate : $\log(x^{-7}) = \log(2^{-1} * 10^2)$
Solution : $x = ((2^{-1} * 10^2))^{(\frac{1}{-7})}$
Result : $x = 0.5718603680956573$
2. $2 * \log(4) - 3 * \log(x) - 2 * (\log(x) + \log(4)) = 5$
Step1 : $2 * \log(4) - 3 * \log(x) - 2 * \log(x) + 2 * \log(4) = 5$
Step2 : $-1 * 3 * \log(x) - 2 * \log(x) = -1 * 2 * \log(4) + 2 * \log(4) + 5$
Evaluate : $\log(x^{-5}) = \log(4^0 * 10^5)$
Solution : $x = ((4^0 * 10^5))^{(\frac{1}{-5})}$
Result : $x = 0.09999999999999999$
3. $4 * \log(2) - 2 * \log(x) - 3 * (\log(x) + \log(2)) = 4$
Step1 : $4 * \log(2) - 2 * \log(x) - 3 * \log(x) + 3 * \log(2) = 4$
Step2 : $-1 * 2 * \log(x) - 3 * \log(x) = -1 * 4 * \log(2) + 3 * \log(2) + 4$
Evaluate : $\log(x^{-5}) = \log(2^{-1} * 10^4)$
Solution : $x = ((2^{-1} * 10^4))^{(\frac{1}{-5})}$
Result : $x = 0.18205642030260802$

Şekil 15. Şablon 6'ya göre üretilen ifadeler ve çözümleri

Şablon 7: $\frac{(\log_a b + \log_a c)}{(\log_a c * \log_c b)}$

$$\text{Şablon 7'ye göre oluşturulan } \frac{(\log_2 8 + \log_2 3)}{(\log_2 3 * \log_3 8)} \text{ ifadesi } \frac{\log_2 8 * 3}{\log_2 8} \Rightarrow \frac{8 * 3}{8} \Rightarrow 3$$

şeklinde çözülürken, uygulamada üretilen soru ve çözüm adımları Şekil 16'da verilmiştir.

1. $(\log n(3, 2) + \log n(3, 4)) / (\log n(3, 4) * \log n(4, 2))$
Step1 : $\frac{\log_3(8)}{\log_3(2)}$
Result : $\frac{8}{2}$
2. $(\log n(2, 4) + \log n(2, 3)) / (\log n(2, 3) * \log n(3, 4))$
Step1 : $\frac{\log_2(12)}{\log_2(4)}$
Result : $\frac{12}{4}$
3. $(\log n(3, 2) + \log n(3, 4)) / (\log n(3, 4) * \log n(4, 2))$
Step1 : $\frac{\log_3(8)}{\log_3(2)}$
Result : $\frac{8}{2}$
4. $(\log n(4, 2) + \log n(4, 3)) / (\log n(4, 3) * \log n(3, 2))$
Step1 : $\frac{\log_4(6)}{\log_4(2)}$
Result : $\frac{6}{2}$
5. $(\log n(2, 4) + \log n(2, 4)) / (\log n(2, 4) * \log n(4, 4))$
Step1 : $\frac{\log_2(16)}{\log_2(4)}$
Result : $\frac{16}{4}$

Şekil 16. Şablon 7'ye göre üretilen ifadeler ve çözümleri

Şablon 8: $a * \log_c x + b * \log_c x = \log_c b * \log_b a$

Şablon 8'e göre oluşan $3 * \log_2 x + 5 * \log_2 x = \log_2 5 * \log_5 3$ ifadesi $8 * \log_2 x = \log_2 3 \Rightarrow x^8 = 3 \Rightarrow x = 3^{\frac{1}{8}}$ şeklindeyken, uygulamada üretilen soru ve çözüm adımları Şekil 17'da verilmiştir.

1. $8 * \log_n(5, x) + 2 * \log_n(5, x) = \log_n(5, 2) * \log_n(2, 8)$
Step1 : $8 * \log_5(x) + 2 * \log_5(x) = \log_5(2) * \log_2(8)$
Step2 : $8 * \log_5(x) + 2 * \log_5(x) = \log_5(2) * \log_2(8)$
Evaluate : $\log_5(x^{10}) = \log_5(8)$
Solution : $x = ((8))^{(\frac{1}{10})}$
Result : $x = 1.2311444133449163$
2. $9 * \log_n(6, x) + 3 * \log_n(6, x) = \log_n(6, 3) * \log_n(3, 9)$
Step1 : $9 * \log_6(x) + 3 * \log_6(x) = \log_6(3) * \log_3(9)$
Step2 : $9 * \log_6(x) + 3 * \log_6(x) = \log_6(3) * \log_3(9)$
Evaluate : $\log_6(x^{12}) = \log_6(9)$
Solution : $x = ((9))^{(\frac{1}{12})}$
Result : $x = 1.200936955088045$

Şekil 17. Şablon 8'e göre üretilen ifadeler ve çözümleri

Şablon 9: $a * \log x + b * \log x = c$

Şablon 9'a göre oluşturulan $4 * \log x + 3 * \log x = 2$ ifadesi $7 * \log x = 2 \Rightarrow \log x^7 = 2 \Rightarrow x^7 = 10^2 \Rightarrow x = 10^{\frac{2}{7}}$ şeklinde çözülürken, uygulamada üretilen soru ve çözüm adımları Şekil 18'de verilmiştir.

1. $2 * \log(x) + 4 * \log(x) = 3$
Step1 : $2 * \log(x) + 4 * \log(x) = 3$
Step2 : $2 * \log(x) + 4 * \log(x) = 3$
Evaluate : $\log(x^6) = 3$
Solution : $x^6 = 10^3$
 $\Rightarrow x = 10^{3 * \frac{1}{6}}$
Result : $x = 3.162277658712097$
2. $4 * \log(x) + 3 * \log(x) = 4$
Step1 : $4 * \log(x) + 3 * \log(x) = 4$
Step2 : $4 * \log(x) + 3 * \log(x) = 4$
Evaluate : $\log(x^7) = 4$
Solution : $x^7 = 10^4$
 $\Rightarrow x = 10^{4 * \frac{1}{7}}$
Result : $x = 3.7275937183530883$

Şekil 18. Şablon 9 'a göre üretilen ifadeler ve çözümleri

Şablon 10: $\log_a b * \log_b c * \log_c a$

Şablon 10'a göre oluşturulan $\log_2 3 * \log_3 5 * \log_5 7$ ifadesi 1 değerine eşit olurken, uygulamada üretilen soru ve çözüm adımları Şekil 19'da verilmiştir.

1. $\log_n(5, 6) * \log_n(6, 9) * \log_n(9, 5)$
Step1 : $\log_6(6)$
Result : 1.0
2. $\log_n(3, 4) * \log_n(4, 7) * \log_n(7, 3)$
Step1 : $\log_4(4)$
Result : 1.0
3. $\log_n(2, 8) * \log_n(8, 9) * \log_n(9, 2)$
Step1 : $\log_8(8)$
Result : 1.0
4. $\log_n(5, 8) * \log_n(8, 9) * \log_n(9, 5)$
Step1 : $\log_8(8)$
Result : 1.0

Şekil 19. Şablon 10'a göre üretilen ifadeler ve çözümleri

4. SONUÇLAR

Çalışmada simgesel matematik yöntemleriyle çeşitli logaritmik ifade ve denklemlerinin üretimleri ve çözüm adımlarının gösterimi gerçekleştirilmiştir. Denklem ve ifade üretimlerinin dayandırıldığı logaritmik ifade şablonları için EBNF benzeri notasyonunda biçimsel bir gramer oluşturulmuş ve JavaCC ile denklemlerin sözdizimine uygun bir ayrıştırıcı geliştirilmiştir. Kaynak verinin analiz sürecinde oluşturulan soyut sözdizim ağaçları, değerlendirme ve dokümantasyon işlemlerinin ana girdisini oluşturmaktadır.

Belirli bir soru kalıbına veya şablonuna göre oluşturulan logaritma problemlerinin çözümleri işlem adımları ile birlikte hem bir terminal üzerinde gösterilebilmekte hem de bir PDF dokümanına aktarılmaktadır. PDF dosyaların üretimi, logaritmik ifadelere ait sözdizim ağaçları üzerinden LaTeX doküman formatlama diline yapılan metinsel dönüşümlerle gerçekleştirilmektedir. Sorunun çözüm süreci ilgili ifadenin sadeleştirilmesi veya değişken değerinin belirlenmesi şeklinde ifade edilmektedir. Farklı biçimlerde hazırlanabilecek logaritma problemleri ile lise öğrencilerinin problem çözme becerilerinin artırılacağı ve öğrenme süreçlerinin hızlandırılacağı öngörülmektedir.

5. ÖNERİLER

Belirli bir kalıba göre üretilen logaritma ifadelerindeki değişkenlerin değerleri önceden belirlenmiş bir sayı aralığına göre atanmaktadır. Bu sayı aralığı kullanıcı tarafından tanımlanabildiği gibi sistem tarafından da tanımlanabilmektedir. Tanımlanmış sayı aralığına göre değişken değeri atanması karmaşıklığı ön görülemeyecek düzeyde yüksek olan soruların üretimine sebep olmaktadır. Karmaşıklık düzeyini kontrol edebilmek adına ileride gerçekleştirilecek çalışmalarda değişken değerleri tanımlanmış sayı aralığı yerine tercih edilen sayıları içeren bir sayı listesinden atanabilir. Logaritmik ifadelerin çözümü basit görünse de karmaşık süreçler içermektedir. İfadelerin karmaşıklığının artması çözümlerin anlaşılabilirliğini azaltmaktadır. Uygulama karmaşık ifadeler odaklı geliştirilirse matematiksel öğrenim açısından daha faydalı olacaktır. Ayrıca uygulama bir bilinmeyenli logaritma ifadeleri için birden fazla kök olması durumu için geliştirilebilir.

Uygulama mevcut haliyle bir terminal üzerinde çalışmaktadır ve görselliği sorular ile aşamalı çözümlerini içeren bir pdf dokümanı ile sağlamaktadır. Uygulamanın bir kullanıcı arayüzü sağlayan masaüstü veya web uygulaması şeklinde düzenlenmesi kullanım kolaylığı sağlayacaktır. Mevcut gramerin farklı soru kalıplarını kapsayacak biçimde genişletilmesiyle farklı türde ifade üretimi ve çözümü uygulama üzerinden yapılabilecektir. İleride yapılacak olan çalışmalarla, yeni türde soru ifadesi üretimi ve çözümü gerçekleştirilmesi, daha kolay anlaşılır ve daha sezgisel bir kullanıcı arayüzünün tasarımıyla kullanım etkinliği yüksek uygulamalar geliştirilebilir.

6. KAYNAKLAR

- Achary, S., Murarka, P. R., & Reza, M. (2012). *Symbolic computation: A Java-based computer algebra system*.
- Aleksandrov, A. G. (2014). Residues of logarithmic differential forms in complex analysis and geometry. *Analysis in Theory and Applications*, 30(1), 34–50.
- Aydođdu, M. C. (2016). *Belirsiz limit ifadelerinin otomatik üretimi ve adım adım çözümü* (Yüksek lisans tezi). Karadeniz Teknik Üniversitesi, Fen Bilimleri Enstitüsü, Trabzon.
- Bauer, A., Clarke, E., & Zhao, X. (1998). Analytica: An experiment in combining theorem proving and symbolic computation. *Journal of Automated Reasoning*, 21(3), 295–325.
- Boyle, A., & Caviness, B. (1988). Future directions for research in symbolic computation. Washington, DC.
- Bruce, I. (2000). Napier's logarithms. *American Journal of Physics*, 68(2), 148–154.
- Burton, D. M. (2007). *The history of mathematics: An introduction* (6th ed.). McGraw-Hill.
- Cajori, F. (1913). History of exponential and logarithmic concepts. *The American Mathematical Monthly*, 20(1), 5–14.
- Chapra, S. C., & Canale, R. P. (2010). *Numerical methods for engineers* (6th ed.). McGraw-Hill.
- Coşkun, E. (2022). *Maxima ile sembolik hesaplama ve kodlama* (pp. 180). Trabzon: KTÜ Yayınevi.
- Davis, M. (1965). The undecidability of the Entscheidungsproblem. *Proceedings of the American Mathematical Society*, 16(6), 1045–1050.
- Dintarini, M. (2018). Understanding logarithm: What are the difficulties that students have?
- Efendiođlu, S. (2017). *Polinomlar için bir simgesel hesaplama çatısının tasarımı ve gerçekteşmesi* (Yüksek lisans tezi). Karadeniz Teknik Üniversitesi, Fen Bilimleri Enstitüsü, Trabzon.
- Eves, H. (1983). *Great moments in mathematics after 1650*. Washington, DC: The Mathematical Association of America.
- Fateman, R. J. (2000). Problem solving environments and symbolic computing.

- Friedl, J. E. F. (2003). *Mastering regular expressions* (2nd ed.). O'Reilly Media.
- Gathen, J. V. Z., & Gerhard, J. (2013). *Modern computer algebra* (3rd ed.). Cambridge University Press.
- Hosseinpour, S., Alavi Milani, M. M. R., & Pehlivan, H. (2018). A step-by-step solution methodology for mathematical expressions. *Symmetry*, 10(7), 285.
- Kojima, K., Miwa, K., & Matsui, T. (2009, December). Study on support of learning from examples in problem posing as a production task. In *Proceedings of the 17th International Conference on Computers in Education [CDROM]*. Hong Kong: Asia-Pacific Society for Computers in Education.
- Knuth, D. E. (1966). The remaining trouble spots in ALGOL 60. *Communications of the ACM*, 9(5), 311–318.
- Knuth, D. E. (1971). Top-down syntax analysis. *Acta Informatica*, 1, 79–110.
- Lewis, P. M., & Stearns, R. E. (1968). Syntax-directed transduction. *Journal of the ACM (JACM)*, 15(3), 465–488.
- Liang, C. B., & Wood, E. (2005). Working with logarithms: Students' misconceptions and errors. *Mathematics Education*, 8(2), 53–70.
- Milani, M., Orhan, Ö., & Milani, B. (2020). Automatic solution of some mathematical problems based on a symbolic computing methodology. *Electronics Letters on Science and Engineering*, 16(2), 130–142.
- Mohammad, M., Alavi Milani, R., Pehlivan, H., & Hosseinpour, S. (2014). Taylor and Maclaurin series expansion using compiler-compiler tools. *Mathematical Sciences International Research Journal*, 3(1), 470–477.
- Moses, J. (1971, March). Algebraic simplification a guide for the perplexed. In *Proceedings of the second ACM symposium on Symbolic and algebraic manipulation* (pp. 282–304).
- Quinlan, D., Wells, J. B., & Kamareddine, F. (2019). BNF-style notation as it is actually used.
- Pehlivan, H. (2019). Sayısal çözümleme yöntemlerinin programlanması ve yorumlanması. *Düzce Üniversitesi Bilim ve Teknoloji Dergisi*, 7, 872–894.
- Smith, E., & Confrey, J. (1994). Multiplicative structures and the development of logarithms: What was lost by the invention of function. In G. Harel & J. Confrey (Eds.), *The development of multiplicative reasoning in the learning of mathematics* (pp. 333–364). State University of New York Press.
- Slagle, J. R. (1963). A heuristic program that solves symbolic integration problems in freshman calculus. *Journal of the ACM (JACM)*, 10(4), 507–520.
- URL-1. (2024, Nisan 1). Yacc: <https://www.javatpoint.com/yacc> adresinden alınmıştır.

- URL-2. (2024, Nisan 1). Bison: <https://www.gnu.org/software/bison> adresinden alınmıştır.
- URL-3. (2024, Nisan 1). Antlr: <https://wwwantlr.org/> adresinden alınmıştır.
- URL-4. (2024, Nisan 1). Ply: <https://www.dabeaz.com/ply> adresinden alınmıştır.
- URL-5. (2024, Nisan 1). JavaCC: <https://javacc.github.io/javacc> adresinden alınmıştır.
- URL-6. (2024, Mayıs 24). Java: <https://tr.wikipedia.org/wiki/Java> adresinden alınmıştır.
- Von Zur Gathen, J., & Gerhard, J. (2013). *Modern computer algebra*. Cambridge University Press.
- Verschelde, J. (2023, June 20). *Introduction to symbolic computation* (Release 1.7.6).
- Yılmaz, C. (2019). *Belirsiz integral problemlerinin çözümü için genel bir yorumlayıcının simgesel hesaplama yaklaşımları kullanılarak tasarımı ve gerçekleşmesi* (Yüksek lisans tezi). Karadeniz Teknik Üniversitesi, Fen Bilimleri Enstitüsü, Trabzon.
- Weber, K. (2002). Developing students' understanding of exponents and logarithms. In *Proceedings of the Annual Meeting of the North American Chapter of the International Group for the Psychology of Mathematics Education*. Athens, GA: University of Georgia.
- Williams, H. R. A. (2011). *A conceptual framework for student understanding of logarithms*. Brigham Young University.

ÖZGEÇMİŞ

Melike Yüksel, İlkokulu ve Ortaokulu Küçükdere İlköğretim Okulu'nda, Liseyi ise Of Anadolu Lisesi'nde tamamlamıştır. Karadeniz Teknik Üniversitesi Bilgisayar Mühendisliği Bölümü'nden 2015 yılında mezun olmuştur. 2018 yılında Çalışma ve Sosyal Güvenlik Bakanlığında Yazılımcı olarak çalışmaya başlamıştır. 2022 Eylül ayından beri Göç İdaresi Başkanlığında Yazılım Geliştirme Uzmanı olarak halen çalışmaktadır. İyi derecede İngilizce bilmektedir.

Yüksel, M., Pehlivan, H. (2024). Production and Evaluation of Logarithmic Expressions with Symbolic Approaches. VIII. International Icontech Conference on Innovative Surveys in Positive Sciences, 287-301.