KARADENİZ TEKNİK ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI

SİMGESEL YAKLAŞIMLARI KULLANARAK SAYISAL KÖK BULMA YÖNTEMLERİ İÇİN GENEL BİR YORUMLAYICININ TASARIMI VE GERÇEKLENMESİ

YÜKSEK LİSANS TEZİ

Bilgisayar Müh. Baki GÖKGÖZ

TEMMUZ 2016
TRABZON

KARADENİZ TEKNİK ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

BİLGİSAYAR MÜHENDİSLİĞİ

SİMGESEL YAKLAŞIMLARI KULLANARAK SAYISAL KÖK BULMA YÖNTEMLERİ İÇİN GENEL BİR YORUMLAYICININ TASARIMI VE GERÇEKLENMESİ

Baki GÖKGÖZ

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

Tezin Enstitüye Verildiği Tarih : 11 / 07 /2016
Tezin Savunma Tarihi : 26 / 07 /2016

Tez Danışmanı : **Yrd. Doç. Dr. Hüseyin PEHLİVAN**

Trabzon 2016

KARADENİZ TEKNİK ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

Bilgisayar Mühendisliği Anabilim Dalında
Baki GÖKGÖZ Tarafından Hazırlanan

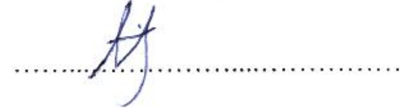
SİMGESEL YAKLAŞIMLARI KULLANARAK SAYISAL KÖK BULMA YÖNTEMLERİ İÇİN GENEL BİR YORUMLAYICININ TASARIMI VE GERÇEKLENMESİ

başlıklı bu çalışma, Enstitü Yönetim Kurulunun 17 / 05 / 2016 gün ve 1653 sayılı kararıyla oluşturulan jüri tarafından yapılan sınavda
YÜKSEK LİSANS TEZİ
olarak kabul edilmiştir.

Jüri Üyeleri

Başkan : Prof. Dr. Vasif V. NABİYEV ....

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

Üye : Yrd. Doç. Dr. M. Mohammad R. A. MİLANİ ....


Prof. Dr. Sadettin KORKMAZ
Enstitü Müdürü

ÖNSÖZ

Günümüzde, yazılım dünyasındaki hızlı gelişime bağlı olarak metinsel veriler üzerinde kullanılabilecek dönüştürücülere duyulan ihtiyaç çok daha fazla artmıştır. Bunun sonucu olarak derleyici ya da yorumlayıcı gibi çeviriciler yazılım geliştirici şirketler için önemli bir araç olmaya başlamıştır. Web programcılığı, veritabanı programcılığı, sistem programcılığı ve daha birçok yazılım uygulamaları alanında derleyiciler ve yorumlayıcılar kullanılmakta olup profesyonel olarak yazılım uygulamaları geliştiricileri bu araçlar ile yakından ilgilenmektedir.

Programlama dillerinin gelişim sürecine bakıldığında, başlangıçta makine kodları ile geliştirilen programların zorluğu ve karmaşıklığı programcıların kod yazımını çok zorlaştırmaktaydı. Daha sonra geliştirilen yüksek seviyeli programlama dilleri ile kod yazım işlemleri daha kolay hale gelmiştir. Kod yazım aşamasında, önce programcı tarafından yazılan kaynak kod çeviriciler aracılığı ile bilgisayarların anladığı makine diline çevirilir.

Bu tez çalışmasında, simgesel yaklaşımlar ve otomatik kod üretim araçları kullanılarak, sayısal yöntemlerle doğrusal olmayan denklemlerde kök hesaplamaları için bir programlama dili tanımlanmış ve bu dil için bir yorumlayıcı geliştirilmiştir.

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

Baki GÖKGÖZ
Trabzon 2016

TEZ ETİK BEYANNAMESİ

Yüksek Lisans Tezi olarak sunduğum "SİMGESEL YAKLAŞIMLARI KULLANARAK SAYISAL KÖK BULMA YÖNTEMLERİ İÇİN GENEL BİR YORUMLAYICININ TASARIMI VE GERÇEKLENMESİ" başlıklı bu çalışmayı baştan sona kadar danışmanım Yrd. Doç. Dr. Hüseyin PEHLİVAN'ın sorumluluğunda tamamladığımı, verileri/örnekleri kendim topladığımı, deneyleri/analizleri ilgili laboratuarlarda yaptığımı/yaptırdığımı, başka kaynaklardan aldığım bilgileri metinde ve kaynakçada eksiksiz olarak gösterdiğimi, çalışma sürecinde bilimsel araştırma ve etik kurallara uygun olarak davrandığımı ve aksinin ortaya çıkması durumunda her türlü yasal sonucu kabul ettiğimi beyan ederim. 26/07/2016

Baki GÖKGÖZ

İÇİNDEKİLER

Sayfa No

ÖNSÖZ III
TEZ ETİK BEYANNAMESİ IV
İÇİNDEKİLER V
ŞEKİLLER DİZİNİ X
TABLOLAR DİZİNİ XII
SEMBOLLER DİZİNİ XIV
1. GENEL BİLGİLER 1
1.1. Giriş 1
1.2. Literatür Taraması 3
1.3. Çeviriciler 7
1.3.1. Derleyiciler 7
1.3.2. Yorumlayıcılar 8
1.3.3. Derleyici ve Yorumlayıcının Karşılaştırılması 9
1.3.4. Çeviricilerin Genel Yapısı ve Bileşenleri 10
1.4. Derleyici Aşamaları 12
1.4.1. Temel Derleyici Aşamaları 14
1.4.2. Kelimesel Analiz 15
1.4.3. Sözdizimsel Analiz(Ayrıştırma) 19
1.4.4. Türetim 21
1.4.5. Ayrıştırma Ağaçları 22
1.5. Ayrıştırma 24
1.5.1. Ayrıştırma Algoritmaları 24
1.5.2. Top – Down Ayrıştırma 25
1.5.3. Bottom – Up Ayrıştırma 28
1.6. Gramer Kuralları İçin Otomatik Ayrıştırıcı Üreteçleri 29
1.6.1. Bison 30
1.6.2. Lex 30
1.6.3. Yacc 31
1.6.4. SableCC 31

1.6.5. JavaCC ... 31
1.7. Ayrıştırıcı Üretimi ... 35
1.8. Sözdizim Sınıfları ... 36
1.8.1. Sözdizim Ağacı ... 37
1.8.2. Değerlendirme Yöntemleri ... 38
1.9. Anlamsal Analiz ... 40
1.9.1. Sembol Tablosu ... 41
1.9.2. Tip Kontrol Tablosu ... 41
1.10. Sayısal Yöntemler ... 42
1.10.1. Basit İterasyon Yöntemi ... 42
1.10.2. İkiye Bölme Yöntemi ... 44
1.10.3. Regula Falsi Yöntemi ... 46
1.10.4. Newton-Raphson Yöntemi ... 48
1.10.5. Sekant Yöntemi ... 51
1.10.6. Halley Yöntemi ... 53
2. YAPILAN ÇALIŞMALAR ... 55
2.1. Giriş ... 55
2.2. Geliştirilen Matematiksel Programlama Dilinin Genel Yapısı ... 57
2.3. Geliştirilen Uygulamaya Ait Arayüz ve Çalışma Aşamaları ... 59
2.4. Gramer Tabanlı Değerlendirme Aşamaları ... 61
2.4.1. Kelimesel Ayrıştırıcı ... 62
2.4.2. Sözdizimsel Ayrıştırıcı ... 64
2.4.3. Anlama Yönelik Analiz ... 71
2.4.4. LL(k) Gramerine Ait Dönüşüm İşlemleri ... 74
2.4.5. Yorumlama Aşamaları ... 75
2.5. Sayısal Yöntemleri Programlama ... 83
2.5.1. Basit İterasyon Yönteminin Programlanması ... 84
2.5.2. İkiye Bölme Yönteminin Programlanması ... 86
2.5.3. Regula Falsi Yönteminin Programlanması ... 88
2.5.4. Newton-Raphson Yönteminin Programlanması ... 90
2.5.5. Secant Yönteminin Programlanması ... 92
2.5.6. Halley Yönteminin Programlanması ... 94
3. SONUÇ ... 96

4. ÖNERİLER 97
5. KAYNAKLAR 98
ÖZGEÇMİŞ

Yüksek Lisans Tezi

ÖZET

SİMGESEL YAKLAŞIMLARI KULLANARAK SAYISAL KÖK BULMA YÖNTEMLERİ İÇİN GENEL BİR YORUMLAYICININ TASARIMI VE GERÇEKLENMESİ

Baki GÖKGÖZ

Karadeniz Teknik Üniversitesi
Fen Bilimleri Enstitüsü
Bilgisayar Mühendisliği Anabilim Dalı
Danışman: Yrd. Doç. Dr. Hüseyin PEHLİVAN
2016, 102 Sayfa

Günümüzde, yaygın olarak kullanılan C, C++ ve Java gibi dillerle yapılan programlama faaliyetlerinin bazı aşamaları için otomatik olarak kod üreten birçok araç bulunmaktadır. Bu araçlar yardımıyla, hem derleme hem de yorumlama süreçlerinin parçası olan sözcüksel analiz, sözdizimi analizi, anlamsal analiz ve ara dil dönüşümü gibi işlemler daha kolay bir şekilde gerçekleştirilebilmektedir.

Bu çalışmada sayısal kök bulma yöntemlerinin otomatik kod üretim araçları ile nasıl programlanacağı gösterilmiştir. Programlama süreci türev alma, fonksiyonel dönüşüm ve iterasyon ifadelerinin üretimi gibi değişik simgesel programlama aktivitelerinden oluşmaktadır.

Kök hesabı yapılacak bir matematiksel ifade, Java programlama dilinde otomatik kod üreten JavaCC aracı kullanılarak, öncelikle birkaç analiz işleminden geçirilir ve sonra nesne yapılarıyla temsil edilir. Problemin çözümüne yönelik seçilen sayısal yöntemin gerektirdiği bütün hesaplamalar bu nesne yapıları üzerinden yürütülür.

Anahtar Kelimeler: Simgesel hesaplama, Ayrıştırıcılar, Sözdizimi sınıfları, JavaCC.

Master Thesis

SUMMARY

DESIGN AND IMPLEMENTATION OF A GENERAL INTERPRETER FOR NUMERICAL ROOT FINDING METHODS USING SYMBOLIC APPROACHES

Baki GÖKGÖZ

Karadeniz Technical University
The Graduate School of Natural and Applied Sciences
Computer Engineering Graduate Program
Supervisor: Asst. Prof. Dr. Hüseyin PEHLİVAN
2016, 102 Pages

Nowadays, there are many tools that automatically generate code for some certain stages of programming activities conducted with the use of modern languages such as C, C++ and Java. Using these tools, the operations such as lexical analysis, syntax analysis, semantic analysis and intermediate language translation, which are parts of both compilation and interpretation processes, are performed more easily.

In this work, it is described how to program numerical root-finding methods via automatic code generation tools. The programming process consists of distinct symbolic programming tasks such as differentiation, functional translation and generation of iteration expressions.

A mathematical expression solved for the roots is firstly processed through some analysis operations and then represented by object structures, using JavaCC, which is an automatic code generation tool. All the relevant computations involved by a numerical method adopted for the solution of the problem are carried out on these object structures.

Keywords*:* Symbolic computation, Parser, Syntax class, JavaCC.

ŞEKİLLER DİZİNİ

Sayfa No

Şekil 1.1. Derleme ve çalıştırma işlemleri 8
Şekil 1.2. Yorumlama ve çalıştırma işlemleri 8
Şekil 1.3. Melez bir derleyici yapısı 10
Şekil 1.4. Çevirici genel mimarisi 10
Şekil 1.5. Ön uç ve arka uç'un rolleri 11
Şekil 1.6. Ön uç ve Arka uç genel yapısı 11
Şekil 1.7. Ön uç içyapısı 12
Şekil 1.8. Arka uç içyapısı 12
Şekil 1.9. Bir Derleyicinin Aşamaları 15
Şekil 1.10. Kelimesel Analiz 16
Şekil 1.11. Sözdizimsel analiz 20
Şekil 1.12. Ayrıştırma ağacı 22
Şekil 1.13. Belirsiz gramer yapısı 23
Şekil 1.14. Belirsiz gramer yapısının oluşturduğu iki farklı ayrıştırma ağacı 23
Şekil 1.15. Belirsiz gramer yapısını ortadan kaldıran yeni gramerimiz. 24
Şekil 1.16. Bir gramere ait BNF dosyası 32
Şekil 1.17. Temel sınıf özelliklerinin tanımlandığı kod bloğu 34
Şekil 1.18. Sözcüksel kurallar ve özellikler bölümü kod bloğu 34
Şekil 1.19. JavaCC dosyasındaki kurallar bildirimi bölümü 36
Şekil 1.20. Gramer sözdizim sınıflarının tanımlaması 36
Şekil 1.21. Gramer kural tanımlamalarına sözdizim ağacı üreten ifadelerin eklenmesi 37
Şekil 1.22. $x3 - x - 1$ fonksiyonuna ait grafik 43
Şekil 1.23. $x3 + 4x2 - 10$ fonksiyonuna ait grafik 45
Şekil 1.24. Regula Falsi Metodu 46
Şekil 1.25. $2x3 - 2.5x - 5$ fonksiyonuna ait grafik 48
Şekil 1.26. Newton-Raphson yönteminin grafiksel gösterimi 49
Şekil 1.27. $x2 - 4$ fonksiyonuna ait grafik 50
Şekil 1.28. Secant Yöntemine ait grafik 51
Şekil 1.29. $4x3 - 16x2 + 17x - 4$ fonksiyonuna ait grafik 52

Şekil 1.30. $x3-3x+2$ fonksiyonuna ait grafik ... 54
Şekil 2.1. Kök kesaplama uygulamasının genel yapısı ... 56
Şekil 2.2. Uygulama arayüzü ... 61
Şekil 2.3. Fonksiyon dönüşümü ... 83
Şekil 2.4. $x+12-\exp-2*x-1$ fonksiyonuna ait hesaplanan kök değerleri ... 86
Şekil 2.5. $x3+4x2-10$ fonksiyonuna ait hesaplanan kök değerleri ... 88
Şekil 2.6. $3*x+\sin x-\exp(x)$ fonksiyonuna ait hesaplanan kök değerleri ... 90
Şekil 2.7. $x=\exp x-3x+4$ fonksiyonu için hesaplanan kök değerleri ... 92
Şekil 2.8. $3*x+\sin x-\exp(x)$ fonksiyonu için hesaplanan kök değerleri ... 93
Şekil 2.9. $\exp x-3x+4$ fonksiyonuna ait grafik ... 95

TABLOLAR DİZİNİ

Sayfa No

Tablo 1.1. Ayrıştırma Tablosu 26
Tablo 1.2. Ayrıştırıcı Üreteçleri 30
Tablo 1.3. Sözdizim ağacı değerlendirme yöntemlerinin karşılaştırılması 38
Tablo 1.4. instanceof işleci ile değerlendirme 38
Tablo 1.5. Sözdizim sınıflarına eval() metotlarının eklemesi 40
Tablo 1.6. $x3 - x - 1$ fonksiyonuna ait hesaplanan kök değerleri 43
Tablo 1.7. Basit İterasyon Yöntemine ait algoritma 43
Tablo 1.8. $fx = x3 + 4x2 - 10$ fonsiyonuna ait hesaplanan kök değerleri 44
Tablo 1.9. İkiye Bölme Yöntemine ait algoritma 45
Tablo 1.10. $2x3 - 2.5x - 5$ fonksiyonuna ait hesaplanan kök değerleri 47
Tablo 1.11. Regula Falsi Yöntemine ait algoritma 48
Tablo 1.12. $x2 - 4$ fonksiyonuna ait hesaplanan kök değerleri 50
Tablo 1.13. Newton-Raphson Yöntemine ait algoritma 51
Tablo 1.14. $4x3 - 16x2 + 17x - 4$ fonksiyonuna ait hesaplanan kök değerleri 52
Tablo 1.15. Sekant Yöntemine ait algoritma 53
Tablo 1.16. $x3 - 3x + 2$ fonksiyonuna ait hesaplanan kök değerleri 54
Tablo 1.17. Halley Yöntemine ait algoritma 54
Tablo 2.1. Matematiksel programlama dili için EBNF grameri 58
Tablo 2.2. Uygulamaya ait token tanımlaması 63
Tablo 2.3. Token dizisi 63
Tablo 2.4. Fonksiyon dizisi, arguman ve parametre tanımı 65
Tablo 2.5. Tablo 2.4. için JavaCC gramer tanımlaması 65
Tablo 2.6. Fonksiyonun sağ tarafına ait gramer tanımı 66
Tablo 2.7. Tablo 2.6. için JavaCC gramer tanımlaması 67
Tablo 2.8. “:” ile tanımlama bloğu ve “;” ile ardışık tanımlamalar 67
Tablo 2.9. Tablo 2.8. için JavaCC gramer tanımlaması 68
Tablo 2.10. Matematiksel ifadeler için gramer 68
Tablo 2.11. Tablo 2.10. için JavaCC gramer tanımlaması 68
Tablo 2.12. Fonksiyona parametre dizisi verilmesi, türev alma işlemi ve fonksiyonun 69

dönüştürülmsi işlemlerine ait gramer ... 69
Tablo 2.13. Tablo 2.12. için JavaCC gramer tanımlaması ... 70
Tablo 2.14. Değerlendirme işleminde koşul ifadelerini kullanımı ... 70
Tablo 2.15. Tablo 2.14. için JavaCC gramer tanımlaması ... 71
Tablo 2.16. Toplama işlemine ait tip kontrolü ... 73
Tablo 2.17. Çıkarma işlemine ait tip kontrolü ... 73
Tablo 2.18. Matematiksel ifadeler için bir LL(1) grameri ... 75
Tablo 2.19. FixedPointVisitor sınıfı (FNEvalVisitor.java) ... 76
Tablo 2.20. FixedPointVisitor sınıfı (DeriveVisitor. java) ... 79
Tablo 2.21. FixedPointVisitor sınıfı (SimplfyVisitor. java) ... 80
Tablo 2.22. FixedPointVisitor sınıfı (PrintVisitor. java) ... 81
Tablo 2.23. FixedPointVisitor sınıfı (FixedPointVisitor.java) ... 82
Tablo 2.24. Basit İterasyon yönteminin programlanması ... 84
Tablo 2.25. İkiye bölme yöntemini programlama ... 87
Tablo 2.26. Regula Falsi yöntemini programlama ... 89
Tablo 2.27. Newton-Raphson yönteminin programlanması ... 91
Tablo 2.28. Secant yönteminin programlanması ... 93
Tablo 2.29. Halley's yönteminin programlanması ... 94

SEMBOLLER DİZİNİ

AST Soyut Sözdizim Ağacı (Abstract Syntax Tree)

BNF Backus–Naur Form

CFG Bağlam Bağımsız Gramer (Context – Free Grammer)

DFA Belirli Sonlu Otomata (Deterministic Finite Automata)

EBNF Genişletilmiş (Extented) Backus–Naur Form

JTB Java Ağaç Oluşturucu (Java Tree Builder)

LL(k) Soldan sağa ayrıştırma – Sola dayalı türetim – k kelime kontrolü (Left-to- Right Parsing – Leftmost Derivation – k lookahead)

LR(k) Soldan sağa ayrıştırma – Sağa dayalı türetim – k kelime kontrolü (Left-to-Right Parsing – Rightmost Derivation – k lookahead)

1. GENEL BİLGİLER

1.1. Giriş

Matematiksel hesaplamalara ihtiyaç duyulan bütün mühendislik disiplinlerinde karşılaşılan problemler için denklemsel ifadelerin oluşturulması ve bu ifadelere uygun çözümler geliştirilmesi önemli bir yere sahiptir. Doğrusal olmayan denklemler, polinomlar, integraller ve diferansiyeller gibi ifadelerin el yordamıyla çözümleri oldukça zor olabilirken, klasik yöntemlerin sunduğu çözümler uzun, karmaşık ve hata içerme olasılığı yüksek işlemler gerektirebilir. Böyle durumlarda matematiksel ifadelerin bilgisayarlar yardımıyla yaklaşık çözümlerinin hesaplanabilmesi için sayısal yöntemlerden yararlanılır.

Sayısal yöntemler matematiksel problemlerin formüle edilip, çeşitli matematiksel işlemler yardımıyla çözümünü bulan yöntemlerdir. Çok sayıda sayısal yöntemler geliştirilmiş olmasına rağmen bu yöntemlerin hemen hemen tamamı çok miktarda yinelemeli hesaplamalar içermektedir. 1940'lı yıllarda bilgisayarların ortaya çıkışıyla birlikte sayısal yöntemlerin daha verimli bir biçimde kullanılması ve gelişim göstermesinin önü açılmıştır. Mühendislik alanlarındaki problemlerin çözümü için sayısal yöntemlerle çözüm yollarının artışında ve yeni yöntemlerin araştırılmasında, yüksek hıza sahip ve karşılaşılan problemleri verimli bir biçimde çözebilecek bilgisayar sistemlerinin gelişmesinin büyük etkisi olmuştur [1].

Simgesel hesaplama, klasik yöntemlerle çözümü zor, zaman alıcı ve hata yapma olasılığı yüksek olan matematiksel problemlerin bilgisayar programları yardımıyla tam ve hatasız olarak çözümünü bulmaya dayanır. Bu hesaplama türünde işlem adımlarına başlamadan önce matematiksel denklemlerin tam olarak ifade edilmesi ve daha sonra bilgisayar programları yardımıyla çözülebilecek algoritmalara dönüştürülmesi gerekmektedir [2,3]. Sayısal yöntemler belirli bir hata payı içerirler; uygulamada kök değerleri hesaplanırken belirli bir hata oranı veya belirli bir iterasyon sayısına göre işlemler yinelenmektedir.

Günümüzde simgesel hesaplama sistemleri genel amaçlı ve özel amaçlı olmak üzere iki gruba ayrılır. Genel amaçlı sistemlere Matlab, Maple, Macsyma, Mathematica, Axiom ve MuPad gibi hesaplama araçları örnek olarak verilebilir. Genel amaçlı simgesel hesaplama araçlarının ihtiyacı karşılamadığı durumlar için özel amaçlı simgesel hesaplama

araçları geliştirilmiştir; bu gruba, gruplar teorisi alanında GAP ve Magma, komütatif cebir ve cebirsel geometri çalışmaları için CoCoA ve Macaulay, yüksek enerji fiziği hesaplamaları için Schoonship araçları örnek olarak verilebilir.

Her simgesel hesaplama sistemi belirli bir programlama dili içerir; yani problemlerin matematiksel modellerine ait denklemlerin çözümünü gerçekleştirecek algoritmaların kodlanmasında özel bir programlama dili kullanılır. Simgesel programlama ile yapılan hesaplamalar, sistemlerin genişletilebilmesinde önemli bir rol oynar.

Kod yazım süreçlerini kolaylaştıran yüksek seviyeli programlama dillerinin ortaya çıkmasıyla birlikte derleyiciler yazılım geliştiriciler için vazgeçilmez araçlar olmuştur. Yüksek seviyeli programlama dilinde yazılmış kodları makine diline çevirme aşamalarını gerçekleştiren derleyiciler, işletim sistemleri kadar karmaşık bir kod yapısına sahiptir [4]. Yorumlayıcılar da çeviriciler sınıfındandırlar, kaynak kodu bloklar halinde veya satır satır çalıştırırlar ve çalıştırma işleminden sonra yeni bir dosya oluşturmazlar [5]. Yorumlayıcılarla çalıştırılan dillere örnek olarak Haskell, Basic, Lisp ve ASP verilebilir.

Programlama dillerinin yapılarındaki işlevsel değişiklikler, yeni teknolojilerin mevcut sistemlere uyumunun sağlanması gibi gereklilikler derleyicilerin yapısındaki karmaşıklığın en önemli sebepleri arasında sayılabilir. Otomatik kod üretim araçlarının geliştirilmesiyle birlikte derleyici ve benzeri yazılım uygulamaları için gereken programlama yükünün önemli bir bölümü bu araçlara devredilebilmektedir. Örneğin, bir programlama dili ile oluşturulan kaynak veri üzerinde analiz, yorumlama ve çevrim gibi aktiviteler için otomatik kod üretimi yapılabilmektedir. Otomatik kod üretim araçları kaynak veri üzerinde belirtilen aşamaları kolaylaştırmak için geliştirilmiştir. Derleyici derleyici olarak isimlendirilen bu araçlarla geliştirilmiş çok sayıda derleyiciler ve yorumlayıcılar mevcuttur. Bu araçlar ile daha dar alanlarda, özellikle üniversiteler gibi eğitim alanlarında kullanılan bazı programlama dillerinin tasarım ve uygulamasına önemli katkılar sağlamışlardır. Bunlara örnek olarak Fang [6], Triangle [7] ve MiniJava [8] dilleri verilebilir.

Programlama dilleri kaynak verinin sözdizimini tanımlayan içerikten bağımsız gramer (CFG) yapıları ile temsil edilirler. Bu yapıların gösterimi genellikle BNF notasyonunda yapılmaktadır ve onlar için ayrıştırıcılar üretebilen çok sayıda otomatik kod üretim aracı bulunmaktadır. Otomatik kod üretim araçlarının değişik programlama dillerine yönelik geliştirilen birçok çeşidi vardır. Sadece Java dilinde kaynak kod üretebilen çok sayıda araç mevcuttur; ANTLR [9], SableCC [10], JTB [11], JavaCC [12], JLex [13] ve

JFlex [14] . Örnek olarak JavaCC aracı ile üretilecek kaynak kod, diğer yazılımlara kolayca entegre edilerek, girdi verilerini işleyebilecek kelimesel çözümleyici ve ayrıştırıcı bileşenleri olarak hizmet verebilmektedir.

1.2. Literatür Taraması

Teknolojinin çok hızlı bir şekilde geliştiği ve her geçen gün bu gelişimin biraz daha hız kazandığı günümüzde, bilgisayarların ve bilgisayar özelliğne sahip yeni zihazların kullanımı günlük hayatın vaz geçilmez bir parçası haline gelmiştir.

Matematik, tarihsel süreçte toplumların temel ihtiyaçlarının giderilmesinde, hayatı kolaylaştırmada kullanılırken günümüzde bilim ve teknolojideki hızlı gelişim insanoğlunun hayatını her alanda etkilemiştir. Matematiksel problemlerin çözümüne ihtiyaç duyulan bütün alanlarda; fizikte, matematikte, kimyada, mühendislikte, genetikte ve benzeri birçok alanda karşılaşılan problemlerin çözümünde çok hızlı ve hatasız çözüm üreten bilgisayarlar kullanılmaktadır

Bu çalışmanın konusu olan simgesel hesaplama ile kök değerlerinin bulunması işlemleri birçok alanda kullanılmaktadır. Simgesel hesaplama, 1953 yılından bu yana bilgisayarlarda kullanılmasına karşın, bilimsel gelişmelerde kullanımı açısından uzun bir geçmişe sahiptir [15].

M.Ö. 3. yüzyılda Öklid tarafından bulunan tamsayıların en büyük ortak bölenlerin bulunması algoritmasının ve genelleştirmelerinin bugün kullanılan simgesel hesap sistemlerinin en temel algoritmaları arasında yer almaktadır.

Leibniz 1673 ile 1676 tarihleri arasınad Pariste matematiksel hesaplamalar üzerine çalışmalar yapmıştır. Leibniz matematiksel metotları ve ifadeleri algoritmalara ve formüllere dönüştürebilmek için karakteristik genel bir sembolik dil aramıştır [15].

Analitik Makine üzerindeki çalışmaları ile bilinen ve makina hakkındaki notları, bir bilgisayar tarafından işlenmek üzere yazılan ilk algoritmayı içerdiği için dünyanın ilk bilgisayar programcısı olarak kabul edilen Ada Lovelace 1842 yılında simgesel hesaplama ile ilgili; makinaların sayısal değerleri harflermiş gibi veya başka simgelermiş gibi düzenleyip birleştirebileceğini ifade etmiştir.

Matematikçiler matematiksel işlemler için birçok algoritma geliştirmişlerdir. Fakat bu algoritmalardan tam olarak faydalanılması bilgisayarın insanoğlunan hayatına girmesiyle başlamıştır. Örneğin; Polinomların çarpanlara ayırma problemi de uzun bir

tarihe sahiptir. Tamsayılar üzerinde tek değişkenli polinomların çarpanlara ayrılması için ilk algoritma 1793 yılında Schubert tarafından bulunmuştur. 1882 yılında bu algoritma Kronecker tarafından yeniden bulunmuş ve cebirsel katsayılı çok değişkenli polinomlara genişletilmiştir. Belirtilen algoritmalardan tam anlamıyla bilgisayarlar yardımıyla faydalanılmıştır.

Otomatik hesaplama için teknolojinin gelişmesinden çok erken bilgisayarların sembolik hesaplamanın yanı sıra sayısal hesaplama yapabileceği fark edilmiştir. 1953 yılında, elektronik bilgisayar icadından sonra, ilk uygulama iki yüksek lisans tezi olarak gerçekleşmiştir. Bunlardan biri Massachusetts Teknoloji Enstitüsünden [16] J. F. Nolan tarafından diğeri ise Temple Üniversitesinde [17] H. G. Kahrimanian tarafından geliştirlmiştir.

1950 lerin sonunda liste işleme dilleri geliştirilmiştir. Bu dillerden en yaygın ve uzun ömürlü olanı Lisp'dir. Lisp, 1958 yılında John McCarty tarafından geliştirilmiştir. Ayrıca Lisp en eski, ikinci üst seviye programlama dilidir. Lisp, simgesel hesaplama çalışmalarında çok önemli rol oynamıştır. Simgesel integral hesaplanmasıyla ilgili ilk program, Lisp ile 1961 yılında Slagle tarafından yazılmıştır. Bu uygulama doktora çalışması olarak geliştirlmiştir [18].

Lisp dilinin programlama olarak sunduğu imkânlar ile bilgisayarlar kullanılarak simgesel matematik problemlerinin çözülebileceğinin anlaşılmıştır. Bu aşamadan sonra simgesel hesaplama alan 1960'lı yıllardan itibaren hızlı bir gelişme göstermiştir.

Schubert ve Kronecker algoritmaları polinomları çarpanlarına ayırmak için kullanıldı fakat bu algoritmaların bilgisayarda bile çok yavaş çalıştıkları görüldü. Berlekamp tarafından 1967 yılında sonlu cisimler üzerindeki polinomların çarpanlara ayrılması için hızlı bir algoritma geliştirilmesi bu probleme yeni bir duruma sokmuştur[19]. 1969 yılında Zassenhaus, Berlekamp algoritması ile elde edilen çarpanlardan tamsayılar üzerindeki çarpanların elde edilebileceğini gösterdi [20]. 1975-76 yıllarında benzer yöntemlerin üzerinde çalışan Musser, Wang ve Rothschild çok değişkenli ve cebirsel katsayılı polinomları ve bu polinomların çözümüne ait algoritmaları geliştirmişlerdir [21, 22].

Risch 1968-70 yılları arasında üstel, logaritmik, trigonometrik ve rasyonel fonksiyonlari kapsayan genel bir fonksiyonlar sınıfı için belirsiz integral probleminin algoritmik çözümünü geliştirmiştir [23]. Günümüzde Risch algoritmasının daha kapsamlı fonksiyon sınıflarına uygulama çalışmalar devam etmektedir.

Integral hesaplama problemine benzeyen diğer bir problemde $\sum_{k=1}^{n} f(k)$ biçimindeki serilerin toplamlarının kapalı olarak ifade edilmesidir. 1978 yılında Gosper yaptığı çalışmalar sonucunda bu problemin çözümüne ait algoritmayı geliştirmiştir [24]. Daha gelişmiş ve genel bir çözüm algoritması 1990 yılında Zeilberger tarafından geliştirilmiştir [24].

1960'lı yılların sonunda ve 1970'li yılların başında ise ilk genel amaçlı simgesel hesap sistemleri geliştirilmiştir. Bu sistemlerden ilki 1968 yılında Reduce [25], 1970'de Macsyma [26] ve Reduce 2 [27], 1971'de Scratchpad [28].

2002'de C++ ortamında GiNac isimli sembolik hesaplama uygulaması Cristian Bauer [29] tarafında geliştrilmiştir. Geliştirilen bu uygulama, simgesel olarak enbüyük ortak bölen hesaplama, çok değişkenli polinom işlemleri, seri açılımları ve matrislerle hesaplama işlemleri yapılabilmektedir.

2004 yılında ifade sadeleştirmeyi anlama başlıklı bir çalışma Jacques Carette [30] tarafından yayınlanmıştır. Yapılan çalışmada sadeleştirme işlemine ait tanımlamalar kesin olarak yapılması gerekliliği göz önünde bulundurularak, genel bir sadeleştirme kavramını tanımı verilmeye çalışılmıştır.

Mohammed Shatnawi 2007 yılında matematiksel ifadelerin aranılması için ayrıştırma ağaç normalizasyonu ile eşitlik kontrolü çalışmasını yapmıştır [31]. İnternet ortamındaki verilerde matematiksel ifadelerin hızlı bir şekilde çoğalmasından dolayı bu veri tipleri için özel arama sorgularının oluşturulması bir ihtiyaç haline gelmiştir. Bazı matematiksel yapıların birden çok özdeşliğe sahip olabileceği durumlar bulunmaktadır. Bundan dolayı yapılan çalışmada matematiksel ifadelerin ayrıştırma ağaçları üzerinde kural tabanlı normalizasyon işlemleri yapılarak özdeş yapılar eşleştirilmiştir. Yapılan çalışmada matematiksel ifadelerin ayrıştırılmasında JavaCC aracı kullanılmıştır.

Yoshinari Miyazaki 2010 yılında web üzerinden mühendislik eğitimine katkı sağlamak amacı ile matematiksel ifadeler için Bilgi Erişimi aracı uygulamasını geliştirmiştir [32]. Web tabanlı bu uygulamada veri tabanı olarak Mysql, programlama dili olarak Java ve sunucu olarak Tomcat yapıları kullanılmıştır. Düzenli ifadeler kullanılarak matematiksel ifadelerin veri tabanı üzerinde arama sorgusu gerçekleştirilmiştir.

2012 yılında Rohit Singh ve arkadaşları sorgu tabanlı otomatik cebir problemi üretme uygulama çalışması yapmışlar ve bu çalışmayı yayınlamışlardır [33]. Yaptıklari çalışmada söz dizim yapılarına ait kural kümeleri kullanılarak ispat problemleri için

şablonlar belirlenmiş ve bu şablon yapıları kullanılarak geliştirdikleri uygulamaya ait dili ile sorgulama işlemleri yaparak sorularoluşturmaktadırlar.

Fateman 2015 yılında Lisp programlama dilinde yaptığı uygulamada algoritmik türev alma çalışmasını yapmıştır [34]. Yaptıkları çalışma otomatik türev alma işlemini gerçekleştirmektedir. Geliştirdikleri uygulamaya ADIL ismini vermişlerdir. Ayrıca geliştirilen uygulamanın kullnımının oldukça basit olduğu belirtilmiştir.

2013 yılında Yavuz TEKBAŞ "Otomatik Kod Üretim Araçları Yardımıyla Matematiksel İfadelerin Türevlerinin Hesaplanması ve Sadeleştirilmesi" isimli çalışmayı yüksek lisans tezi olarak sunmuştur [35]. Java programlama dili ile geliştirilen çalışmada, ifadelerin ayrıştırılması için otomat kod üretim aracı olan JavaCC aracı kullanılmıştır. Çlışmada belirlenen gramer kurallarına göre türev alma, dönüştürme ve sadeleştirme işlemleri gerçekleştirilmektedir.

2015 yılında Mir Mohammad Reza Alavi Milani doktora tez çalışmasında matematiksel ifadelerin adım adım değerlendirilmesi ile ilgili genel bir metodoloji çalışması yapmıştır [36]. Çalışmada matematiksel ifadeler için genel bir dil bilgisi çalışması yapılmıştır. Uygulama geliştirme dili olarak Java kullanılmıştır. İfadelerin ayrıştırma işlemleri için JavaCC ile üretilen ayrıştırıcı yapısı kullanılmıştır.

İletişim teknolojisinin hızlı bir şekilde ilerlemesi ve bu teknolojinin her alanda çok yoğun olarak kullanımı beraberinde güvenlik problemlerini de ortaya çıkarmıştır. İletişim teknolojisinde verilere ulaşımın engellenmesi ve güvenli bir şekilde alıcıya ulaştırılması amacıyla şifreleme işlemleri kullanılmaktadır. Kriptoloji, bilgi alış verişinde bulunan birden fazla tarafın veri iletişim güvenliğini sağlayan, temel yapı olarak çözümü çok zor matematiksel problemlere dayanan sistemlerin tamamıdır. 1976 yılında yayımlanan "New Direction in Cryptography" isimli makalede Diffie ve Hellman açık anahtarlı kriptografi yapısını tanımlamışlardır. Sonraları bu sistem birçok ticari uygulamada kullanıldı. Algoritmanın çalışma mantığı, iletişim kuran iki tarafın belirli bir anahtarı güvenli şekilde birbirlerine göndermelerinden sonra bu anahtarı kullanarak şifrelenmiş bilgileri birbirlerine iletmelerini sağlamaktır. Diffie-Hellman sistemi ayrık logaritma problemi üzerine kurulmuştur. Bu sistemin güvenirliliği çok büyük asal sayılara dayanmaktadır [37]. Bu yöntemde şifreleme işlemleri için asal sayılar kullanılmaktadır. Seçilen asal sayı ne kadar büyük olursa bu sayının çarpanlarına ayrılması ve ve şifrenin çözülmesi o kadar zorlaşır. Bu yöntemde kök hesaplama işlemelerine de ihtiyaç duyulmaktadır. Geliştirilen uygulama bu yöntem de gerekli olan matematiksel işlemler için kullanılabilir. Bu algoritma yapısının

bulunmasından sonra 1977 yılında R. Rivest, A. Shamir ve L. Adleman isminde üç bilim adamının oluşturduğu ekip tarafında RSA şifreleme algoritması geliştirilmiştir [38]. Bununla birlikte kök hesaplama işlemlerinin kullanıldığı alanlara örnek olarak; bir elektrik devresindeki akımın sıfır olduğu anlar, etkileşen iki yükü birleştiren doğru boyunca elektrik alanının sıfır olduğu nokta, diş sayıları farklı dişlilerin çakışma noktalarının veya bir denklem sisteme ait özdeğerleri hesaplama işlemleri gibi örnekler verilebilir.

Mühendislik uygulamalarında veya matematiksel işlemlerin kullanıldığı diğer disiplinlerde yüksek dereceli polinomlar, üstel fonksiyonlar ya da çok miktarda veri içeren denklemsel yapılar ile karşılaşılabilmektedir. Bu fonksiyonlar için bazı durumlarda bu tür denklemsel ifadeleri sıfır yapan durumların hesaplanması veya bu matematiksel fonksiyonları en küçük veya en büyük yapan değerlerin bulunmasına ihtiyaç duyulabilmektedir. Yine bu gibi durumlarda kök hesaplama işlemleri için geliştirilen sayısal yöntemlerin bilgisayar programları ile programlanarak kullanılması işlemlerde çok büyük kolaylık sağlamaktadır.

1.3. Çeviriciler

Bu bölümde dil çeviricilerinin genel yapısı ayrıntılı olarak anlatılmış ve çevirici örnekleri olan derleyiciler ile yorumlayıcılara ait genel çalışma yapısı ve bunların karşılaştırılması yapılmıştır.

1.3.1. Derleyiciler

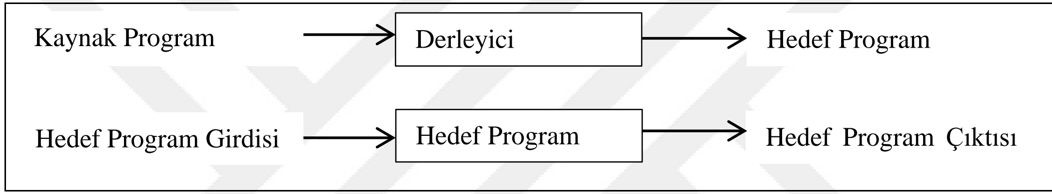
Bir dil çeviricisi, kısa ve genel tanımıyla bir dilde yazılmış bir programı (kaynak dil) alarak diğer bir programlama diline (hedef dil) çeviren araçtır. Bu çevirme işleminde kaynak dil yüksek seviyeli bir dil ve hedef dil assembly veya makine dili gibi düşük seviyeli bir programlama dili ise bu çeviricilere derleyici denir [39]. Hedef dil için üretilen bu kodlar genellikle ortama göre çalıştırılabilen kod olarak üretilmektedir.

Günümüzde derleyiciler her hangi bir dilde geliştirilen kaynak koddan ortam (işletim sistemleri) bağımlı kodların oluşturulmasında kullanılırlar. Bu kodların oluşturulma sürecinde derleyiciler doğrudan işletim sisteminin anlayacağı kodları üretirler veya işletim sisteminin sahip olduğu ve derleyiciler tarafından üretilen kodları bağlayarak işletim

sisteminin çalıştırabileceği kodları üreten bağlayıcıların (linker) anlayabileceği ara kodları oluştururlar.

Derleyicilerin oluşturduğu hedef dildeki kodların kullandığı hafıza miktarının az olması ve hızlı çalışması onlar için başarı göstergesi olarak kabul edilen etkenlerdendirler. Bundan dolayı derleyiciler oluşturulan kodlardan en iyi verimi alabilmek için kod optimizasyonunu gerçekleştirirler. Derleyiciler gerçekleştirdiği diğer önemli bir iş ise kod dönüşüm sürecinde, kaynak programdaki hataların yakalanması ve bu hataların rapor edilmesidir.

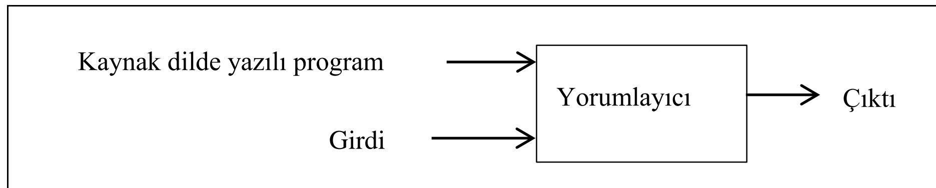
Yüksek seviyeli bir programlama dili ile kodlanmış bir program Şekil 1.1'de belirtildiği gibi iki aşamada çalıştırılır [39].



Şekil 1.1. Derleme ve çalıştırma işlemleri

1.3.2. Yorumlayıcılar

Yorumlayıcı, derleyiciler gibi dil dönüştürücülerindendir, fakat çalışma yapısı derleyicilerden farklıdır. Kaynak kodu bloklar halinde veya satır satır hedef dile çeviren dil işlemcilerine yorumlayıcı denir. Yorumlayıcılar; bir programlama dilindeki kaynak kodu çalıştırmak, farklı bir dildeki koda çevirmek, derlenmiş kodları sırası geldiğinde çalıştırmak gibi işlemleri gerçekleştirmektedirler.



Şekil 1.2. Yorumlama ve çalıştırma işlemleri

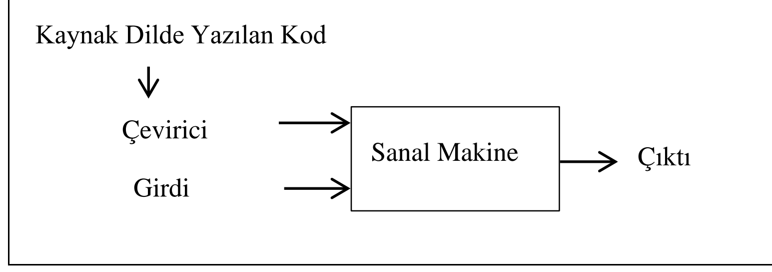
1.3.3. Derleyici ve Yorumlayıcının Karşılaştırılması

Genel olarak derleyiciler ve yorumlayıcılar benzer işleri yapmaktadır. Programcı tarafından yüksek seviyeli bir programlama dilinde yazılan kaynak kodu bilgisayarların anladığı bir dile çevirme işlemi dil çeviricileri olarak adlandırılan derleyici (compiler) ve yorumlayıcı (interpreter) programları tarafından gerçekleştirilir. Bu çevirme süreçleri derleyici ve yorumlayıcıda farklılıklar göstermektedir.

Derleyiciler, girdi olarak aldığı kaynak kodun tamamını hedef dildeki koda çevirir. Böylece kaynak kod girdisi en başta bir defa derlenir bu derleme aşamasında hedef dildeki (genellikle makine dili) kodları içeren, çalıştırılabilir bir dosya oluşturulur. Derleyici tarafından oluşturulan bu dosya sonraki aşamalarda hiçbir değişiklik yapılmadan çalıştırılabilir bir yapıdadır. Eğer girdi olarak verilen kaynak kodda herhangi bir değişiklik yapılırsa, kaynak kodun yeniden derleme işlemine tabi tutulması gerekmektedir.

Yorumlayıcılar, girdi olarak alınan kaynak kodu birinci satırından itibaren her satır için çeviri işlemini yaptıktan sonra kodun çalıştırma işlemini gerçekleştirirler. Bu çevirim işlemi sürecinde herhangi bir çalıştırılabilir dosya oluşturulmadığı için kaynak kodun kullanıldığı her aşamada tekrar yorumlanması gerekmektedir.

Günümüzde melez (hybrid) derleme olarak adlandırılan, bazı programlama diller için dil işlemcileri tarafından derleme ve yorumlama aşamalarının birleştirildiği yapılar vardır. Bu programlama dillerine örnek olarak Java programlama dili söylenebilir. Java programlama dilinde kaynak kod ilk önce bytekod olarak adlandırılan bir ara yapıya dönüştürülür. Bytekodlar daha sonra Java sanal makinesi (Java Virtual Machine(JVC)) tarafından yorumlanır. Bunun faydası ise herhangi bir makinede derlenen kodların başka bir makinede yorumlanabilmesidir. Ayrıca bu durum ilgili programlama dilini güçlü kılan özelliklerdendir. Girdilerin çıktılara daha hızlı eşlenmesini sağlamak için çalışma anında (Just-in-Time(JIT)) bir çeşit derleyici olan Java derleyicileri, bytekodların tamamını doğal makine diline çevirirler [40]. Bu dönüştürme işlemi özellikle çok sık kullanılan sınıf metotları için gerçekleştirilir. İlgili metot ilk çağrıldığında metoda ait bytekodlar makine diline çevrilir ve bu şekilde saklanır. Bu metot daha sonra çağrıldığında ise metoda ait makine kodu çağrılacağından dolayı çağırma işlemi daha kısa sürede gerçekleşecektir.



Şekil 1.3. Melez bir derleyici yapısı

Derleyici ve yorumlayıcının avantaj ve dezavantajlarının bir kısmı aşağıda belirtilmiştir [41];

- Kaynak kodun çevrildikten sonra çalıştırılma aşamasında, her defasında yorumlayıcının belleğe yüklenmesine ihtiyaç duyar. Bu durum kaynak koda bellekte daha az yer tahsis edilmesine neden olmaktadır. Derleyicide ise derleyici sadece kaynak kodun derleme aşamasında belleğe yüklenir, daha sonraki çalıştırma aşamalarında belleğe yüklenmez dolayısıyla bellekte kaynak koda daha fazla alan tahsis edilmiş olur.
- Makine diline çevrilmiş programlar, yorumlanarak çalıştırılan programlara göre daha hızlı çalışırlar.
- Yorumlayıcılar derleyicilerden daha iyi hata tespiti yaparlar. Bunun nedeni ise derleme anında tespit edilemeyen bazı hataların yorumlama anında tespit edilebilmesidir.
- Derlenen programların kodunda değişiklik yapma, yorumlanan programların kodunda değişiklik yapmaya göre daha yavaş ve zordur.

1.3.4. Çeviricilerin Genel Yapısı ve Bileşenleri

Çeviriciler girdi olarak aldıkları kaynak kodu çeşitli ara işlemlerden geçirirler bu ara işlemler sürecinde kod ile ilgili hatalar varsa bunlar rapor edilir ve ayıklanır daha sonra kaynak kod makine diline çevrilir. Aşağıda Şekil 1.4'de çeviricilerin genel mimari yapısı gösterilmiştir [39].

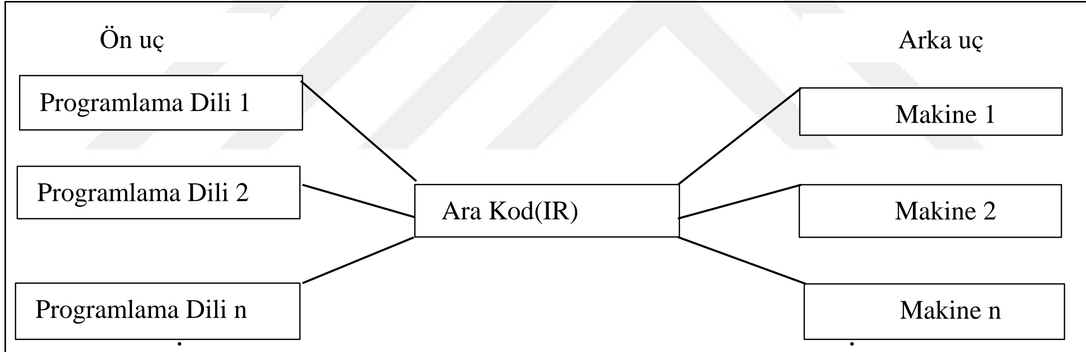
Şekil 1.4. Çevirici genel mimarisi

Derleyici farklı aşamaları bir ön uç ve arka uç oluşturmak üzere bir arada gruplandırabilir.

Ön uç öncelikle kaynak dile bağımlı ve hedef dilden bağımsız yapılardan oluşmaktadır. Ön uç genellikle sözcüksel analiz, kelimesel analiz ve anlamsal analiz kısımlarından oluşur. Ayrıca kod optimizasyonunun bir kısmı ön uç içinde yapılabilir.

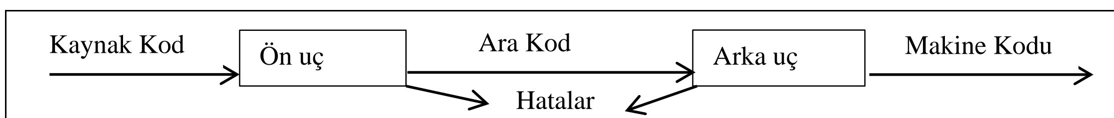
Arka uç, ön uç gibi birçok kısımdan oluşur, fakat arka uç kaynak koddan bağımsız olup hedef koda bağımlı bir yapıdadır. Bununla birlikte arka uç, kod üretimi ve kod optimizasyonunu içermektedir. Ön uç ve arka uç derleyici için birçok avantaj sağlamaktadır [42];

- Aynı ön uç ile farklı bir makinedeki farklı bir arka uç bağlantısı yapılarak aynı kod için farklı bir derleyici oluşturulabilir.
- Farklı ön uçlar tutularak aynı arka uç ile birçok farklı dile ait kodlar aynı makine üzerinde derlenebilir. Bu avantajlar için genel yapı Şekil 1.5'de gösterilmiştir.



Şekil 1.5. Ön uç ve arka uç'un rolleri

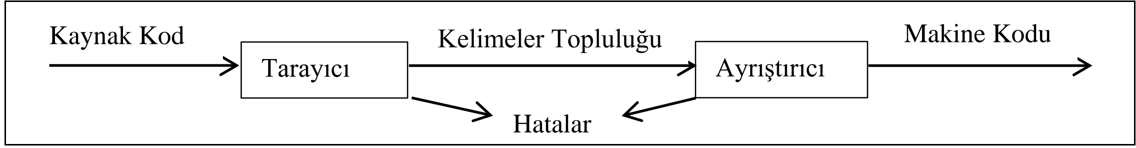
Yukarıda bahsedilen bileşenlerden ön uç kaynak kodu alır ve IR olarak adlandırılan ara koda dönüştürme işlemini gerçekleştirir. Arka uç ise ön uç tarafından oluşturulan ara kod (IR) u değişik alt işlemlere tabi tutarak hedef koda çevirir. Şekil 1.6'da Ön uç ve arka uç bileşenlerinin temel yapısı gösterilmiştir.



Şekil 1.6. Ön uç ve Arka uç genel yapısı

1.3.4.1. Ön Uç (Front end)

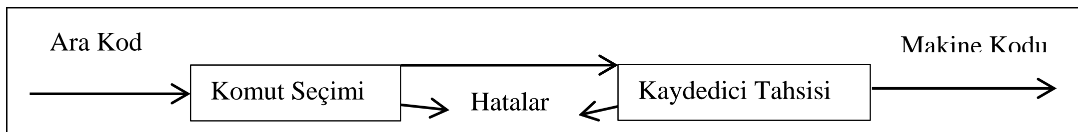
Ön uç, girdi olarak aldığı kaynak kodu ilk olarak tokenlarına ayırır. Bu ayırma işleminden sonra tokenları anlamlı parse ağaç yapıları şekline çevirir ve daha sonra onları IR (ara kod) olarak adlandırılan ara kod yapılarına dönüştürür. Şekil 1.7'de Ön uç'un iç yapısı ayrıntılı olarak gösterilmiştir.



Şekil 1.7. Ön uç içyapısı

1.3.4.2. Arka Uç (Back end)

Derleyicinin bu bileşeni ön uç tarafından oluşturulan ara kodu alır bazı işlemlere tabi tuttuktan sonra bulunduğu makineyle eşleştirme işlemini yapar ve böylece ara kod makine koduna çevrilir. Ayrıca arka uç kod optimizasyonu işlemini de gerçekleştirmektedir. Aynı arka uç yapısı kullanılarak aynı makinede farklı ön uç yapılarının ürettiği ara kodlar derlenebilmektedir. Şekil 1.8'de Arka uç' un içyapısı ayrıntılı olarak gösterilmiştir.



Şekil 1.8. Arka uç içyapısı

1.4. Derleyici Aşamaları

Modern bir derleyici, genellikle her bir bileşeni farklı bir soyut dilde işlem gören süreçlere ayrılmıştır [43]. Derleme işlemi ise bu süreçler ile bu süreçlerden farklı bir grup aktiviteden oluşmaktadır. Bu aşamalar altı bölümden ve iki grup aktiviteden meydana gelmiş olup bu bölümde bu aşmalar ve aktiviteler detaylı bir şekilde anlatılmıştır.

Derleyici süreçleri [43];

- Sözcüksel analiz (lexical analysis)
- Söz dizimsel analiz (syntax analysis)
- Anlamsal analiz (semantic analysis)
- Ara kod üretimi (intermediate code generation)
- Kod optimizasyonu (code optimization)
- Kod üretimi (code generation)

Aktiviteler ise;

- Sembol tablosu yönetimi (symbol table management)
- Hata kotarıcı (error handler)

Derleme aşamalarının tamamını iki sınıfa ayırabiliriz. Bunlardan birinci aşama analiz aşaması ikinci aşama ise sentez aşamasıdır [43]. Derleyicinin ilk üç aşaması analiz bölümünü geri kalan aşamalar ise sentez bölümünü oluşturur. Derleyicinin her bir aşamasında, ilk başta girdi olarak alınan kaynak kod bir biçimden başka bir biçime çevrilir. Derleyicinin her iki aktivitesi altı aşama ile etkileşimde bulunurlar. Derleyicilerin aşamalarını anlatmaya ayrıntılı olarak geçmeden önce kısaca özetleyelim.

Kaynak kod derleyiciye geldiği zaman, ilk olarak, kaynak koda ait karakterler soldan sağa doğru teker teker okunup işlenerek token olarak adlandırılan yapılara dönüştürülür. Bu işlem sözcüksel analizci tarafından gerçekleştirilir. Kaynak kodda kullanılan anahtar kelimelerin tespiti de sözcüksel analizci tarafından gerçekleştirilir. Sözcüksel analizci tarafından kaynak koddan oluşturulan tokenların derleyici tarafından kod sentezi için kullanılan gramatiksel ifadeler olarak sınıflandırılması işlemi sözdizimsel analizci aracılığı ile gerçekleştirilir. Bu aşamadan sonra ise çeşitli işlemlere tabi tutulan kaynak kodun anlamsak olarak hatalı olup olmadığının kontrolü anlamsal analizci tarafından gerçekleştirilir. Bazı derleyicilerde bu aşamalardan sonra kaynak koddan bir ara kod oluştururlar. Kod optimizasyon aşamasında ise kaynak kodun daha hızlı ve verimli çalışabilmesi için bazı işlemlerden geçirilir. Son aşama ise kod üretim aşmasıdır bu aşamada assembly kodu olarak adlandırılan hedef kod üretilir.

Bu kısımda yukarıda bahsettiğimiz derleyicinin temel iki aktivitesini ve yaptıkları görevleri kısaca açıklayalım.

Derleyicinin temel işlevlerinden biri kaynak kodda belirtilen tanımlayıcıları (identifier) kaydetmek ve bu tanımlayıcıların niteleyicileri hakkında belirlenen bilgileri toplamaktır. Her bir tanımlayıcı için sembol tablosunda bir yer ayrılır ve tanımlayıcı ile ilgili niteleyici bilgiler sembol tablosunda ayrılan bu yere yerleştirilir. Bu bilgiler derleme

süreçlerinin birçok safhasında kullanılır. Bunu bir örnekle açıklayacak olursak mesala kaynak kodda belirtilen bir değişken için kod üretim aşamasında bu değişken için hafızada ne kadar yer tahsis edileceği, belirtilen değişkene ait sembol tablosunda tutulan bilgilere bakılarak karar verilir.

Hata tespitinde ise derleyicinin belirtilen her bir aşamasında hatalarla karşılaşılabilir. Derleme aşmalarında işlem yapan sözcüksel analizci, söz dizimsel analizci ve anlamsal analizci bir programdaki hataların büyük bir kısmını tespit edebilir.

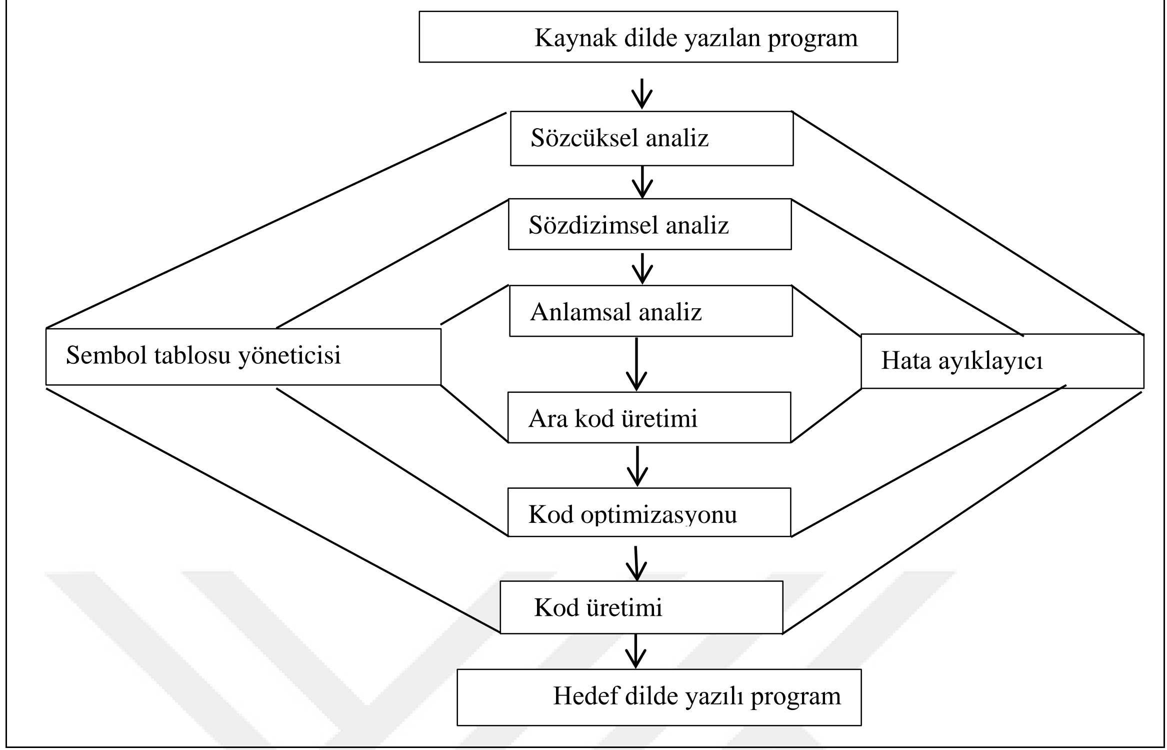
Sözcüksel analizci anlamlı token oluşturamayan karakter katarlarını tespit eder. Dilin sözdizimsel (syntax) yapısına uymayan token gruplarını sözdizimsel analizci tespit eder. Anlamsal analiz sırasında sözdizimine uyan fakat bir anlam ifade etmeyen yapılar tespit edilir. Örneğin, uzunluğu 4 byte olarak belirlenen bir değişkene 4 byte'lık alana sığmayan bir sayının atanması veya bir bölme işleminde bölenin 0 olması gibi. Bundan sonraki kısımda bu aşamalar ana başlıklar altında detaylı olarak anlatılmıştır.

1.4.1. Temel Derleyici Aşamaları

Bir derleyicinin tasarım süreci üç ana aşamadan oluşur

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

Altı aşamalı ve iki aktiviteli derleme aşamaları Şekil 1.9' da ayrıntılı olarak gösterilmektedir [44]. Derleyici yapısının bu kadar çok aşamadan oluşmasının sebebi girdi olarak alınan kaynak kodun her aşamada dönüşüme uğratılarak farklı ortamlar için kod üretimi gibi çeşitli amaçları gerçekleştirmek için kullanılmak istenmesidir.



Şekil 1.9. Bir Derleyicinin Aşamaları

Yorumlayıcılar için analiz süreci üç aşamadan oluşur [45]:

• Kelimesel Analiz

• Sözdizimsel Analiz

• Anlamsal Analiz

Sentez aşaması ise aşağıda belirtilen aşamaların tamamını veya bir kısmını içerir [21]:

- Makineden bağımsız kod optimizasyonu
- Bellek tahsisi
- Makine kodu oluşturma
- Makine kodunun optimizasyonu

1.4.2. Kelimesel Analiz

Belirli bir biçime sahip kaynak verilerin değerlendirme süreci kelimesel analiz ile başlar. Bu aşamada kaynak veri, ilgili programlama dilinin sözcük yapısına uygun olarak token adı verilen parçalara ayrılır. Bir ya da birkaç karakterden oluşabilen tokenlar, düzenli ifadeler yardımıyla kolayca tanımlanabilir.

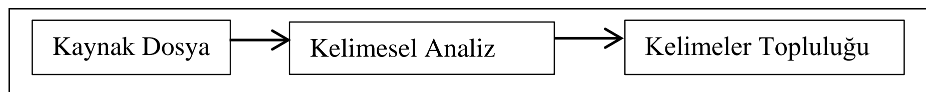
Sözcüksel analiz aşamasında, derleyici kaynak kodu oluşturan karakterler akışını girdi olarak alır bu karakterleri bazı işlemlerden geçirerek lexeme denilen anlamlı diziler şeklinde gruplar. Her lexeme için, sözcüksel analiz aşamasında bir sonraki aşama olan sözdizimsel analiz aşamasına geçirilen biçiminde bir token çıktı olarak üretilir. Üretilen tokendaki birinci eleman olan token-name sözdizimi analizi işlemleri esnasında kullanılan soyut bir semboldür ve ikinci kısım bu token için sembol tablosunda ayrılan bir girdiyi gösterir. Sembol tablosunda bütün elemanlar için girdi bilgisi tutulur bu bilgiler anlamsal analiz ve kod üretimi aşamaları için gereklidir.

Birçok programlama dilinde kullanılan token çeşitleri şunlardır [46]:

- Tanımlayıcılar (a, b, average, vb.)
- Anahtar kelimeler (if, do, for, vb.)
- Tam sayı sabitler (12, 0xFF, 0188, vb.)
- Kayan noktalı sabitler (5.6, 3.6e8, vb.)
- String sabitler ("merhaba\n", vb.)
- Karakter sabitler ('x', 'y', vb.)
- Özel semboller (( ) : := + - vb.)
- Yorumlar (Yok sayılacak)
- Derleyici emirleri (Dosyaları eklemek için direktifler, makro tanımlamalar, vb.)
- Satır bilgisi
- Beyaz boşluk (White space)
- Dosya sonu

Kelimesel analiz aşamasında kaynak kod içerisinde bulunan ve çözümleyici tarafından belirlenen dil gramer kurallarına uygun olmayan yapılar bulunup hata mesajı olarak raporlanır. Bu aşamada kaynak kodda hata tespiti yapılması durumunda kaynak kodun analizine devam edilmez yani hata düzeltilmeden bir sonraki aşamaya geçilmez.

Kelimesel analiz safhasının gerçekleştirilmesi için çevirme yapılarından yararlanılır. Bu yapılar; düzenli ifadeler ve sonlu otomatalardır. Şekil 1.10’da kelimesel analiz aşamasının yapısı gösterilmiştir.



Şekil 1.10. Kelimesel Analiz

1.4.2.1. Düzenli İfadeler

Düzenli ifadeler [47], karakter katarlarından meydana gelen kümelerin matematiksel olarak tanımlanmasını sağlayan bir gösterim metodudur [48]. Düzenli ifadeler bütün karakterler kümesinin içinde belli bir karakter kümesinin seçiminin yapılmasını sağlayan bir mekanizma olarak ta tanımlanabilir. Düzenli ifadeler ilk olarak 1956 yılında Stephen C. Kleene tarafından formel bir model olarak sunulmuştur.

Birçok programlama dilinde, sözcüksel tokenlar düzenli ifade olarak adlandırılan yapılarla tanımlanabilen çok basit bir yapıya sahiptir. Ayrıca düzenli ifadeler birçok editör programında kelime işleme aracı olarak kullanılmaktadır. Düzenli ifadeler sonlu durum otomatları ile tanınan dilleri ifade edebilme gücüne sahiptirler. Bununla birlikte düzenli ifadeler yaygın olarak yazılım araçlarında da kullanılmaktadır. Arama motorlarında, bilgi edinim (information retrieval), kelime işleme (word processing), veri doğrulama (data validation) ve sözdizim belirginleştirme(syntax highlighting) vb. işlemlerde yine düzenli ifadelerden yararlanılmaktadır. Programcı eşleşecek tokenları ve programlama dilinde uygulanacak işlemleri düzenli ifadeleri kullanarak belirtir.

Düzenli ifadeler kullanılacağı zaman bazı özel kavramlar kullanılır. Bu mekanizmada kullanılan temel kavramlar aşağıda sıralanmıştır [49]:

• Seçim : “ | ” işareti kullanılarak ifadeler seçeneklere ayrılır. Örneğin; (a | b) ifadesinde “a” ya da “b” ile eşleşme yapılır.

• Birleştirme : “ · ” işareti kullanılarak ifadeler birleştirilir. Örneğin; (a | b) · a ifadesinde “aa” ya da “ba” ifadeleri ile eşleşme yapılır.

• Epsilon : “ ε ” işareti ile kullanılarak boş değer gösterimi sağlanır. Örneğin; (a | ε) ifade-sinde aya da boş değer eşleşmesi yapılır.

- Tekrarlamalar: Bir karakterin ya da grubun ardından gelerek öncesindeki yapının kaç kez tekrarlanacağını belirtir. En temel tekrarlamalar “ * ” , “ + ” ve “ ? ” dir.

“ * “ ; hiç bulunmayacağını ya da herhangi bir sayıda tekrarlanacağını belirtir.

“ + “ ; en az bir kez olmak üzere herhangi bir sayıda tekrarlanacağını belirtir.

“ ? “ ; hiç bulunmayacağını ya da bir kez bulunacağını gösterir.

Düzenli ifadelerdeki bu kavramlar kullanılarak programlama diline ait kelimesel yapılar tanımlanır.

Programlama dilleri için genel olan bazı yapıların, düzenli ifade formatında gösterimleri aşağıdaki biçimdedir:

```
IF : " if "
DEĞİŞKEN : [a-z,0-9]*
SAYI : [0-9]+
GERÇEL :( [0-9]+ " . " [0-9]* ) | ( [0-9]* " . " [0-9]+ )
YORUM : ( " -- " [a-z]* " \n ")
```

Tanımlanan bu kurallara göre oluşturulacak yapıda bir takım belirsizlikler oluşabilmektedir. Örneğin "ifa" yapısı düzenli ifade tarafından IF ve DEĞİŞKEN mi yoksa yalnızca DEĞİŞKEN olarak mı algılanması gerektiği konusunda belirsizlik meydana gelmektedir. Bu belirsizlik probleminin çözümü için kelimesel çözümleyiciler iki önemli kural geliştirmişlerdir. Bunlar;

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

Bu kurallar sayesinde " ifa " kelimesi en uzun eşleşme kuralına göre DEĞİŞKEN, kural önceliği kuralına göre IF ve DEĞİŞKEN olarak algılanır. Kelimesel çözümleyiciler bu kurallardan sadece birini kullanırlar.

1.4.2.2. Kelimesel Çözümleyici Üreteçleri

Düzenli ifadeleri kolay bir şekilde otomatik olarak DFA yapılarına dönüştürmek amacıyla kelimesel çözümleyici üreteçleri geliştirilmiştir. Bu araçlar kullanılarak karakter kümeleri, token olarak adlandırılan kelimeler topluluğuna dönüştürülür. Bu yapıların en yaygın olarak kullanılanları;

- Flex,
- Lex,
- JLex,
- Yacc,
- Javacc,
- Sablecc,
- Antlr,
- Quex.

Bu araçlar tanımlanan yapıları, kod içinde belirleyebilmek için otomatik kod üretirler. Günümüzde farklı birçok programlama dillerinde otomatik kod üretebilen birçok

üreteç vardır. C/C++ programlama dilleri için Lex, Flex, Quex, Yacc kelimesel çözümleyici üreteçleri otomatik kod üretirken, Java programlama dili için ise JLex [13], JavaCC [12], Sablecc, Antlr [9] gibi onlarca otomatik kod üretim aracı mevcuttur.

Otomatik kod üretim araçları yüksek seviyeli programlama dilleri için otomatik kod üretmekle birlikte bu yapıların kendilerine özgü sözdizim yapıları bulunmaktadır. Örneğin bu tez için geliştirilen projede kullanılan Java programlama dili için kod üreten JavaCC üreteci, Java'nın sözdiziminden bağımsızdır. Java programlama dilinde otomatik kod üretme aracı olan JavaCC uyumlu örnek kod bloğu;

```
options {…}
PARSER_BEGIN(FNParser)
 public class FNParser {…}
PARSER_END(FNParser)
TOKEN: {<IF: "if">
  | <THEN: "then">     | <ELSE: "else">   | <PRINT: "print">
  | <CPRINT: "cprint">| <PLUS: "+">      | <MINUS: "-">
  | <TIMES: "*">       | <DIVIDE: "/">    | <MOD: "%">
  | <POWER: "^">       | <AND: "&&">      | <OR: "||">
  | <NOT: "!">         | <EQ: "==">       | <NE: "!=">
  | <LE: "<">          | <LT: "<=">       | <GT: ">=">
  | <GE: ">">          | <QM: "?">        | <COMMA: ",">
  | <SEMI: ";">        | <COLON: ":">     | <SQRT: "sqrt" >
  | <DRV: "drv">     | <FIX: "fix">   | <OTHER: "otherwise">
  | <ID: (["a"-"z","A"-"Z"])(["a"-"z","A"-"Z","0"-"9"])*>
  |<NUM:(["0"-"9"])+>| <DNUM: (["0"-"9"])+"."(["0"-"9"])+>
  | <STR: ("\"" (~["\""] | "\\" "\"")* "\"")> …}
```

1.4.3. Sözdizimsel Analiz(Ayrıştırma)

Kaynak veri biçiminin tanımlandığı bu aşamada token dizisi kullanılarak sözdizim analizi yapılır. Ayrıştırıcının (parser) iki ana görevi vardır; kaynak verinin biçimsel denetimi ve nesne ağacının üretimi. Biçimsel denetim için verinin sözdizimini tanımlayan BNF ve CFG gibi gramer türleri kullanılır ve bu tanımlamalar içerisine nesne ağacını üreten ifadeler eklenir.

Bu aşamada yapılan işlemler;

- Programlama diline ait sözdizimsel kurallarının tanımlanması,
- Programa ait kaynak kodun, tanımlanan programlama dilinin sözdizimsel yapısına uygun olup olmadığını kontrol edip var olan hataları belirlemek ve bu hataları rapor etmek,
- Kelimesel analiz aşamasında üretilen kelimeler topluluğuna hiyerarşik bir yapı azandırıp, bir sonraki derleme aşamasına geçiş için yeni bir veri yapısı üretmek.

Şekil 1.11'de bu aşamanın adımları gösterilmiştir.

Şekil 1.11. Sözdizimsel analiz

1.4.3.1. İçerikten Bağımsız Gramer (Context Free Grammar - CFG)

Bilgisayar bilimlerinde, programlama dili tasarımı sürecinde kullanılan bir gramer türüdür. Kısaca bir dilin kurallarını (dilbilgisini, grammar) tanımlamak için kullanılır.

İçerikten bağımsız diller, programlama dilleri olarak sıklıkla kullanılmaktadır, genellikle kullanılan programlama dillerini çoğu birer içerikten bağımsız dil özelliğine sahiptir. Programlama dillerinin içerikten bağımsız gramer yapılarını kullanabilmeleri için kendilerinin de içerikten bağımsız gramer yapısına sahip olmaları gerekmektedir. Yani bu dillerin kurallarının tanımlandığı gramerlerde içerikten bağımsız gramerlerdir (context free grammar).

Bu anlamda, JavaCC gibi programlama ortamlarında, bir dil tasarlamak ve içerikten bağımsız kurallar yazarak dili tanımlamak mümkündür.

CFG genelde yüksek seviyeli programlama dilleri için dil tasarımında yaygın olarak kullanılmak birlikte, kullanım alanı bununla bununla sınırlı değildir. Örneğin, CFG bilgisayar ağlarına yapılan saldırıların modellenmesinde de kullanılabilmektedir [50]. CFG, yinelemeli (recursive) yapılardan oluşan bir grup kurala denir.

Bir bağlamdan bağımsız gramer (CFG, context-free grammar) içiçe kelime dizeleri üretebilen bir biçimsel dili temsil eder. Genellikle BNF (Backus Normal Form) notasyonu ile tanımlanan bu gramerin her bir kuralı $X \longrightarrow w$ biçimindedir; X bir nonterminal

simgesine, w ise terminal (bir kelime) ve/veya nonterminal dizilerine karşılık gelir (w boş olabilir) [51].

Temel olarak bir içerik bağımsız gramer yapısı aşağıda maddeler halinde verilen dört özelliğe sahip olmaları gerekmektedir.

Bunlar;

- Sonlular (terminals),
- Devamlılar (nonterminals),
- Bağlantılar (relation) ve
- Başlangıç sembolu (starting symbol) olarak sıralanabilir.

Bir gramerin tanımı sırasında kullanılan bu kümeler aşağıdaki gibi yazılabilir:

G = (V, Σ, R, S)

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

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

Üretim kurallarının formatı aşağıdaki biçimdedir;

```
SEMBOL ::= SEMBOL SEMBOL … SEMBOL
```

Üretim kuralının sağ tarafında en az bir tane olmak şartı ile sınırsız sayıda terminal ya da non-terminal ifade bulunabilir [52].

1.4.4. Türetim

Genel olarak iki çeşit türetim yöntemi vardır. Bunlardan birincisi en soldan türetim (left most derivation) olup, bu tür türetimlerde türetimin herhangi bir adımında en soldaki sonlandırılmamış sembol sağ tarafıyla yer değiştirilir. İkinci türetim şekli en sağdan türetim (right most derivation) olup, türetimin her adımında en sağdaki sonlandırılmamış sembol sağ tarafıyla yer değiştirilir. Girdi verisi olarak alınan kaynak kodun tasarlanan programlama dilinin gramer yapısına ait olup olmadığının belirlenmesi amacıyla türetim işlemleri gerçekleştirilmektedir. Türetim işlemine başlangıç sembolü ile başlanır ve üretim kurallarına uygun şekilde eşleştirme gerçekleştirilir.

Türetim işleminin farklı yöntemleri bulunmaktadır;

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

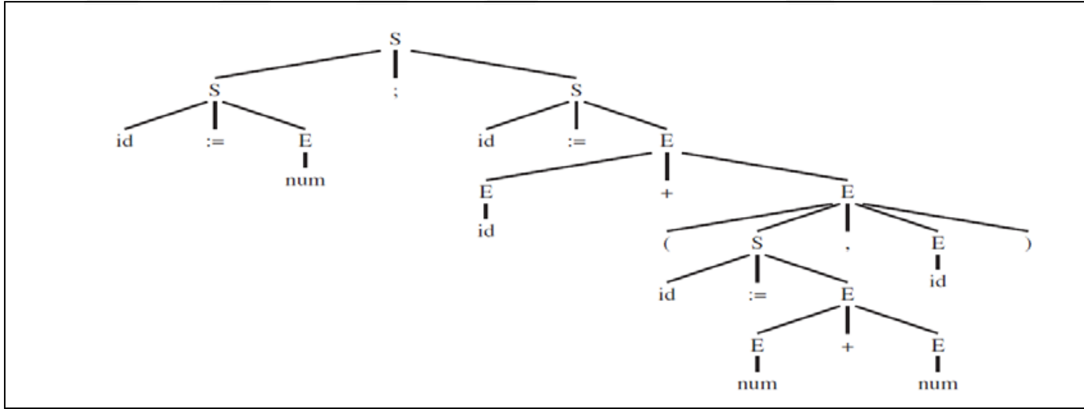
• Sağa Dayalı Türetim yönteminde ilk olarak en sağdaki non-terminal sembolü genişletilmeye çalışılır.

1.4.5. Ayrıştırma Ağaçları

Ayrıştırma (parse) işlemleri bilgisayar bilimlerinde çeşitli işlemleri gerçekleştirmek için kullanılmaktadır. Özellikle de programlama dillerine ait gramer işlemlerinin hemen hepsinde ihtiyaç duyulan bir yapıdır. Örneğin bir programlama dilinde yazılan kodların anlaşılabilmesi için ilk olarak koda ait kelimelerin ayrıştırılması (parse) gerekmektedir.

Ayrıştırma ağacı bir kelime topluluğunun, ilgili dile ait tanımlanan gramer kurallarına göre yapısal olarak ayrıştırılıp bir ağaç yapısı şeklinde ifade edilmesidir. Oluşturulan ayrıştırma ağacında bulunan iç düğümler gramerde bulunan non-terminalleri ifade ederken, yaprak düğümler gramerin terminal yapılarıyla ifade edilir.

Bir ayrıştırma ağacı Şekil 1.12'de [43] gösterildiği gibi birbirinden türetilmiş ve birbiri ile bağlı düğümlerden oluşturulur. İki farklı türetim aynı ayrıştırma ağacına sahip olabilir [43].



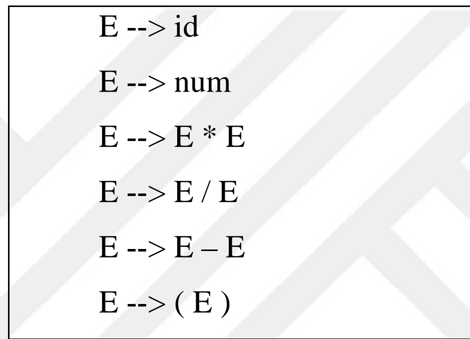
Şekil 1.12. Ayrıştırma ağacı

1.4.5.1. Belirsiz Gramer

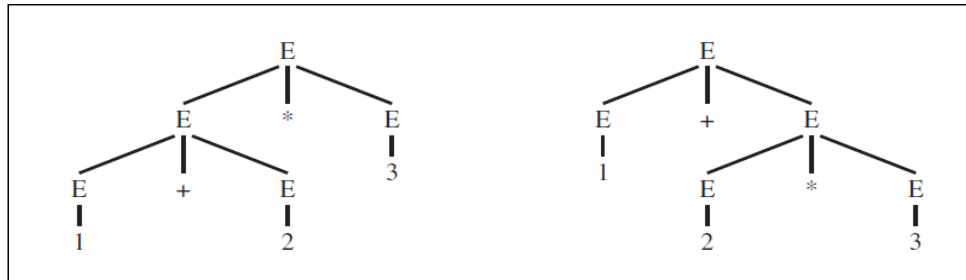
Bir gramer ancak ve ancak iki veya daha fazla farklı ayrıştırma ağacı (parse trees) olan bir cümlesel biçim (sentential form) üretiyorsa belirsizdir (ambiguous) [53].

Ayrıştırma işleminin analiz aşamasında, ayrıştırma yönüne bağlı olarak, birden fazla ayrıştırma ağacı meydana getiren gramer yapıları belirsiz (ambiguous) gramer olarak adlandırılır [54].

Belirsiz gramer yapılarının birden fazla sözdizim ağacı üretmesinden dolayı derleme esnasında bu durum sorun oluşturmaktadır. Bazı gramer kuralları birden fazla sözdizim ağacı üretebilmekte ve bu durum farklı sonuçlara sebep olmaktadır. Bu farklı sonuçlardan kurtulmak için programlama dilleri bu belirsizlikleri ortadan kaldıracak çözümleri bulması gerekmektedir. Belirsizlik durumuna 1+2*3 örneğinin şekil 1.13'deki [54] gramer yapısı ile Şekil 1.14'te [53] ayrıştırılması ve farklı iki sonuç ürettiği gösterilmiştir.



Şekil 1.13. Belirsiz gramer yapısı



Şekil 1.14. Belirsiz gramer yapısının oluşturduğu iki farklı ayrıştırma ağacı

Şekil 1.13'de gösterildiği gibi ayrıştırma işleminin yönüne göre iki farklı sonucu veren iki farklı sözdizim ağacı oluşmuştur. Şekil 1.14'de baktığımızda birinci sözdizim ağacı 9 sonucunu üretirken ikinci ağaç ise 7 sonucunu üretmektedir. Dolayısıyla buradaki sebep belirsiz gramer yapısıdır.

Bu durum gramer kurallarının işleçlerin öncelik düzeylerini dikkate almamasından kaynaklanır. Eğer ayrıştırma ağacında işleçlerin (operators) öncelik seviyeleri gösterilirse belirsizlik olmaz [53].

Şekil 1.15'de oluşturduğumuz yeni bir gramer yapısıyla belirsizliği ortadan kaldırabiliriz [53].

E --> E + T	T --> E * T	F --> id
E --> E – T	T --> E / T	F --> num
E --> T	T --> F	F --> (E)

Şekil 1.15. Belirsiz gramer yapısını ortadan kaldıran yeni gramerimiz.

1.5. Ayrıştırma

Token dizisi üzerinden gramer kurallarına uyarak ayrıştırma işlemi yapar. Günümüzde pek çok ayrıştırma algoritması kullanılmaktadır bu algoritmalar aşağıda sınıflandırılmıştır.

1.5.1. Ayrıştırma Algoritmaları

- Top – Down Ayrıştırma
  - Recursive Descent Ayrıştırma
  - LL Ayrıştırma
- Bottom – Up Ayrıştırma
  - LR Ayrıştırma
    - SLR Ayrıştırma
    - LALR Ayrıştırma
    - Canonical LR Ayrıştırma
- Derleyici Derleyiciler
  - Bison
  - Lemon
  - Lex
  - JavaCC

1.5.2. Top – Down Ayrıştırma

Ayrıştırma ağacını(parse tree), yapraklardan (leaves) başlayarak oluşturur. Sıra, en sağ türevin (rightmost derivation) tersidir. Bir $xA\alpha$ sağ cümlesel formu (right sentential form) verildiğinde, ayrıştırıcı (parser), sadece A'nın ürettiği ilk jetonu (token) kullanarak, en sol türevdeki (leftmost derivation) sonraki cümlesel formu (sentential form) elde etmek için doğru olan A'ya ait kuralı (A-rule) seçmelidir [53].

1.5.2.1. Aşağıya Özyinelemeli (Recursive Descent) Ayrıştırma

Gramerde(grammar) her bir nonterminal için o nonterminal tarafından üretilebilen cümleleri(sentences) ayrıştırabilen(parse) bir altprogram(subprogram) vardır [53].

EBNF, özyineli azalan ayrıştırıcıya (recursive-descent parser) temel oluşturmak için idealdir, çünkü EBNF nonterminal sayısını minimize eder [53]. Pek yaygın olmamasına rağmen algoritmanın en önemli avantajı basit olması ve ayrıştırıcı üreteçlerinin kullanılmadığı durumlarda kolaylıkla oluşturulabilmesidir. Recursive descent ayrıştırma algoritmasının çalışabilmesi için ilgili gramer yapısındaki her bir alt parçasındaki birinci terminal sembolün, bir sonraki işlem için kullanılması gereken kural için ilgili bilgiye sahip olması gerekmektedir. Eğer alt ifadelerdeki ilk terminaller bu bilgiye sahip değilse ayrıştırıcı çalışmayacaktır. Recursive descent ayrıştırma algoritmasının gerileme (backtracking) yapılmayan şekline Predictive Ayrıştırma adı verilir.

Recursive descent ayrıştırma algoritmasında, First ve Follow Set yapıları ayrıştırma tablolarını oluşturmak amacıyla geliştirilmiştir.

Gramer yapısını oluşturan terminaller ve non-terminallerin gramerin yapısında tanımlı olan herhangi bir kuralın hangi terminal sembolü ile başlayacağı FIRST set yapısı ile belirlenir. Oluşturulan bir gramer yapısında aynı sol tarafa ait en az iki kuralının, aynı FIRST sete sahip olması ilgili gramerin bu algoritma aracılığıyla ayrıştırma işleminin gerçekleştirilemeyeceğini gösterir.

1.5.2.2. LL Ayrıştırma

Tez için geliştirilen uygulamada LL ayrıştırma yöntemi kullanıldığı için bu bölüm daha detaylı bir şekilde ele alınmıştır. Bu tür ayrıştırma işlemlerinde ayrıştırma işleminin

herhangi bir basamağında en soldaki sonlandırılmamış sembol sağ tarafıyla yer değiştirilir bu işlem ilgili yapının sonuna kadar devam eder.

Bu gramer yapısı genel olarak LL(k) şeklinde gösterilir bu harflerin tanımı şöyledir;

L: Left-to-Right scan, soldan sağa doğru tarama yaklaşımından,

L: Leftmost Derivation, soldan eksiltme yaklaşımından,

k: sayısı ise ayrıştırma işlemi yapılırken kaç kelimeyi göz önünde bulundurularak karar verileceğini ifade eder.

LL Ayrıştırma;

- Girdi olarak alınan cümleyi saklayacağı bir ara belleğe,
- Ayrıştırma sonunda elde edilen terminal ve non-terminalleri saklayabileceği bir yığına,
- Girdi cümlesi ve elde edilen yığına bakarak hangi gramer kuralının uygulandığını gösteren bir ayrıştırma tablosunu barındırır.

Bu ayrıştırma yönteminde ayrıştırma tablosundaki kurallara bakılarak giriş cümlesindeki sembollere uygunluğu değerlendirilir sembollere en fazla uyanı bir yığına aktarılır ve bu işlem girdi olarak alınan cümlenin sonuna kadar uygulanır.

Bu ayrıştırma işleminde ayrıştırıcı başlangıç sembolü olarak adlandırılan “S” harfine, sonlandırıcı olarak kullanılan “$” sembolünü içerir.

Basit bir gramer örneği olarak bir LL ayrıştırıcısının ürettiği ayrıştırma tablosu aşağıda gösterilmiştir [53].

Gramer yapısı:

```
1. S → F
2. S → ( S + F )
3. F → x
```

Giriş cümlesi (x + x)

Tablo 1.1. Ayrıştırma Tablosu

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

Örneğe bakıldığında ayrıştırıcı ilk olarak giriş cümlesinden “(” ifadesini ve yığından başlangıç değeri olan “S” değerini bellekten okur. Tablo 1.1’den bu ifadenin 2 numaralı kural olduğunu tanımlar, sonra kural numarasını geri değer olarak döndürüp, yığın içerisindeki değişim işlemini gerçekleştirir.

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

Ayrıştırıcı bir sonraki işlem adımında ise “(“ ifadesini giriş cümlesinden ve yığından çıkarır. Bu adımdan sonraki durum aşağıdaki gibi olur.

[S , + , F ,) , $]

Ayrıştırıcı bundan sonra bir sonraki ifadeyi belirler (x), belirlenen bu ifadenin 1 numaralı kural ile sonrasında 3 numaralı kural ile eşit olduğuna karar verir ve bu karara göre tabloda ve yığında gerekli değişiklikleri yapar ve bu işlemlerden sonra yığının son hali şu şekilde olur:

[F , + , F ,) , $]

[x, + , F ,) , $]

Ayrıştırıcı geriye kalan son iki adımda “x” ve “+” ifadelerini giriş cümlesinden ve yığından siler. [F ,) , $]. Ardından ayrıştırıcı “x” ifadesini 3 numaralı kural ile eşleştirir. Ayrıştırıcı son olarak F ve “)” yığından ve giriş cümlesinden silerek ayrıştırma işlemini tamamlar. Bu işlemlerin sonucunda yalnızca “$” ifadesi kalıyor ve giriş cümlesinin sonuna gelinmişse, giriş cümlesinin ilgili gramere uygun olduğu anlamına gelmektedir.

Ayrıştırma işleminin başarılı bir şekilde sonlandırılmasından sonra geriye ayrıştırma işleminden elde edilen kural dizisi döndürülür. Yukarıda incelediğimiz örnek için kural dizisi [2, 1, 3, 3] biçimindedir.

S → (S + F) →(F + F) → (x + F) → (x + x).

Bir grameri LL(k) gramer yapan özellikleri;

- Soldan yinelemeli bir yapıda olmamalıdır; eğer bir gramer soldan yinelemeli ise bu gramer LL(k) gramer olamaz bu gibi durumlarda gramer soldan yinelemeli olan kural sağdan yineleme yoluyla tekrar yazılır ve böylece soldan yineleme sorunundan kurtarılmış olur.

- Sol faktörleme; Eğer iki kural aynı yapı ile başlıyorsa bu durumda LL(k) için problem oluşturur. Bu problemden kurtulmak için sol faktörleme işlemine başvurulur. Bu durum aşağıda bir örnek ile açıklanmıştır.

S → if E then S else S

```
S → if E then S
```

Yukarıdaki problemi ortadan kaldırmak için aşğıdaki gibi bir faktörleme işlemi herçekleştirilir bu işlemden sonra yapı LL(k) ayrıştırmasına uygun hale gelmiş olur.

```
S → if E then S X
X →
X → else S
```

1.5.3. Bottom – Up Ayrıştırma

Aşağıdan yukarıya ayrıştırma yöntemine ait çok sayıda algoritma geliştirilmiştir. Geliştirilen bu algoritmaların büyük çoğunluğu sağdan türetim (LR) denilen ayrıştırma yönteminin çeşitleridir. Bu ayrıştırma yöntemi ayrıştırma ağacının yapraklarından başlayarak köke doğru devam eder bu şekilde ayrıştırma işlemi gerçekleştirilir.

1.5.3.1. LR Ayrıştırıcılar

LR ayrıştırma yöntemi LL ayrıştırma yöntemine göre daha güçlüdür. Bunun sebebi ise LL ayrıştırma yöntemi tahmine dayalı bir ayrıştırma tekniğine sahiptir bu zayıflığından dolayı daha güçlü bir yapıya sahip olan LR ayrıştırma yöntemi geliştirilmiştir. Bu bölümde LR ayrıştırma yöntemi detaylı bir biçimde incelenmiştir.

LR ayrıştırma yöntemi ile girdi verisi olarak alınan cümle soldan sağa okunur, fakat türetim yönü sağa dayalı biçimde gerçekleştirilir. LR ayrıştırıcılar diğer ayrıştırıcılarla karşılaştırıldıkları zaman kullandıkları ayrıştırma tablosunun daha küçük olduğu ve bu tablonun kaynak kodunun daha az yer kaplayan bir yapıda olduğu görülür. LR algoritması aslı, Donald Knuth tarafından tasarlanmış ve 1965'te yayınlanmıştır [53], [55].

LR ayrıştırıcıların avantajları [56];

- Programlama dillerini tanımlayan gramerlerin tamamına yakınında kullanılabilir.
- Diğer aşağıdan yukarıya ayrıştırma algoritmalardan daha kapsamlı bir gramerler sınıfı için işlemler gerçekleştirir, bununla birlikte diğer aşağıdan yukarıya ayrıştırıcılar kadar verimlidir.
- Çok kısa bir zamanda sentaks sözdizimsel hataları saptayabilir ve bunları rapor edebilir.
- LR ayrıştırıcı gramerler sınıfı, LL ayrıştırıcıları tarafından ayrıştırılabilen sınıfın

üst kümesidir.

LR ayrıştırıcı gramer yapısını güçlü kılan bu avantajlarının yanında birde dezavantajı vardır. Bu dezavantaj programlama dili için ihtiyaç duyulan ve ayrıştırma kurallarının saklandığı ayrıştırma tablosunun elle üretiminin neredeyse imkansız olması durumu olarak ifade edilebilir. Bu problem, otomatik ayrıştırma tablosu oluşturan otomatik ayrıştırıcı üreteç yazılım yapılarının geliştirilmesiyle ortadan kalkmıştır.

LR ayrıştırma yöntemini kullanan algoritmalar, ayrıştırma tablosunun üretilme yoluna göre birkaç grupta incelenir.

Bunlar;

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

1.5.3.2. LR Ayrıştırma İşlemi

Temel olarak LR ayrıştırıcı işlemi beş ana elemandan oluşur.

Bunlar;

- Giriş stringini saklayan bir arabelleğe,
- Ayrıştırma işlemlerinin çıktısını gösteren ayrıştırma ağacına,
- Gramer yazım kurallarının ve ayrıştırma işleminde nerede olduğumuzu gösteren bir yığına,
- Ayrıştırıcı programına,
- Ayrıştırma tablosuna sahiptir.

LR ayrıştırma işlemleri süreçlerinde kullanılan ayrıştırıcı programın ve ayrıştırma tablolarının, ilgili programlama dili için elle hazırlanması çok zordur. Bu problemi ortadan kaldırmak için bu elemanlar otomatik ayrıştırıcı üreteç yazılım programı araçları ile oluşturulurlar.

1.6. Gramer Kuralları İçin Otomatik Ayrıştırıcı Üreteçleri

Bu bölümde bu tezde de kullanılan ve derleyicilerin önemli kod yükünü üzerine alan otomatik kod üretim araçlarından bahsedilmiştir. Günümüzde yüksek seviyeli

programlama dilleri için tasarlanmış onlarca otomatik kod üretim aracı bulunmaktadır. Bu yapılara derleyici derleyici ismi verilmektedir.

Bu tezde geliştirilen projede Java programlama diliyle yazılmıştır. Yine bu tezde Java programlama dilinde otomatik kod üretebilen JavaCC kullanılmıştır. Daha öncede bahsedildiği gibi ayrıştırma algoritmalarının elle oluşturulması çok zor, hataya açık ve karmaşık yapısından dolayı bu ayrıştırma algoritmalarını otomatik olarak gerçekleştirecek otomatik ayrıştırıcı üreteçleri geliştirilmiştir.

Tablo 1.2'de [57] bazı otomatik ayrıştırıcı üreteçler, bu üreteçlerin oluşturuldukları algoritmalar ve kod ürettikleri programlama dilleri ile birlikte gösterilmiştir.

Tablo 1.2. Ayrıştırıcı Üreteçleri

Üreteç	**Ayrıştırma Algoritması**	**Programlama Dili**
ANTLR	LL(k)	C#, Java, Phyton
BISON	LALR	C,C++
JAVACC	LL(k)	Java
SABLECC	LALR	Java, C, C++, C#, Phyton
YACC	LALR	C

1.6.1. Bison

Tablo 1.2'de gösterildiği gibi bu ayrıştırıcı üreteç C ve C++ programlama dillerinde yazılan kodların derleme süreçlerinde otomatik kod üretici olarak ihtiyaç duyulan işlemleri gerçekleştirmektedir. Bison ilgili programlama dilindeki gramer yapılarının ayrıştırma işleminde Backus – Naur Form (BNF) olarak isimlendirilen ve Chomsky hiyerarşisi içinde Bağlam Bağımsız Diller (Context Free) sınıfında olan bir gösterim yapısını kullanmaktadır. Bison bir programlama dilinde otomatik ayrıştırıcı olarak kullanılabilmesi için bu programlama dilinin Bağlam Bağımsız Gramer (Context Free Gramer) yapısında olması gerekmektedir [58].

1.6.2. Lex

Bilgisayar bilimlerinde programlama dillerinin tasarımı ve geliştirilmesi süreçlerinde kullanılan ve dildeki sözcüklerin analiz (lexical analysis) işlemlerini gerçekleştiren kod

üretme programıdır. Daha çok derleyici üreticisi olan yacc programına alt yapı oluşturulmak için kullanılır.

Lex Program yapısı: Lex programlarının kod yapısı yan yana iki % sembolü kullanılarak ayrıştırılan üç farklı bölümden oluşur.

- Tanım bölümü: C kodunun üstünde kısmında yazılması gereken kod bölümünün tamamı buraya yazılır. Yazılan kodun tamamı “%{“ ile “%}” arasında yer almalıdır. Kullanılması zorunlu değildir.
- Kurallar bölümü: Bu bölümde örüntü yapılarının ve bu yapılarla karşılaşıldığında yapılacak işlemlerin tanımlandığı kurallar bölümüdür.
- Kullanıcı alt-programları bölümü: Bu bölümde ise fonksiyon içerikleri yer alır. Lex oluşturduğu koda bu alt-programları kopyalar.

1.6.3. Yacc

Yacc, Lex aracılığı ile üretilen çıktıyı alarak sözdizimsel kurallarla belirlenen ayrıştırma süreçlerini gerçekleştirecek kodu yapısını üreten bir yazılım programıdır. YACC basitçe ilgili programlama dilindeki sözdizim (syntax) tasarımı için kullanılır ve tasarlanan programlama dilindeki kelimelerin sıralama durumunun doğru bir biçimde girilip girilmediğini kontrol eder. Ayrıca oluşturulan sıralamadaki her bir kelimenin ne anlama geldiği de yacc programı yardımıyla belirlenebilmektedir. YACC programının yapısında temel olarak BNF (Backus Normal Form) kullanarak cümle dizimi belirlenmektedir.

1.6.4. SableCC

SableCC 1998 yılında Etienne Gagnon tarafından bir yüksek lisans tezi olarak geliştirilmiş ve dil geliştirmekte kullanılan, JAVA üzerinde çalışan, nesne yönelimli bir geliştirme ortamıdır. SableCC derleyiciler, yorumlayıcılar ve diğer metin ayrıştırıcıları oluşturmak için tamamen nesne yönelimli çalışma ortamı olan bir ayrıştırıcı üreticidir [59].

1.6.5. JavaCC

JavaCC, Java programlama dilinde kullanılan bir ayrıştırıcı üreteci ve sözcüksel analizci üretecidir. Ayrıştırıcılar ve sözcüksel analizciler karakter dizilerinin girişi ile

ilgilenen yazılım bileşenleridir [60]. JavaCC içerikten bağımsız gramer (CFG) yapısına uygun olarak Java programlama dilinde recursive descent ayrıştırma kodu üretir.

JavaCC ayrıştırıcı üretecinin aşamaları [57];

- BNF (Backus – Naur Form)'de kullanılacak programlama diline ait gramer yapısı oluşturulur.
- BNF dosyasından, JavaCC gramer dosyası oluşturulur.
- Geliştirilecek uygulama için oluşturulan JavaCC gramer dosyasından Java programlama diline çevirme işlemleri gerçekleştirilir.
- Son olarak oluşturulan Java programlama diline ait kod dosyası geliştirilecek uygulama içinde kullanılır.

Yukarıdaki dört aşama başlıklar halinde aşağıda incelenmiştir.

1.6.5.1. Gramer Yapısının Oluşturulması

Şekil 1.16'da örnek bir gramerin [57] BNF özelliğindeki dosyaya ait kod yapısı gösterilmiştir. Terminal ifadeler '<' ve '>' sembolleri arasında bulunmaktadır.

```
Expression::=AndExpression(<OR>
AndExpression)*AndExpression::=
EqComparisonExpression(<AND>
EqComparisonExpression )*
EqComparisonExpression::=
ComparisonExpression ( ( <EQ> | <NE> )
ComparisonExpression )* ComparisonExpression
::= AdditiveExpression ( ( <LT> | <GT> | <LE> |
<GE>)AdditiveExpression )* AdditiveExpression
::= MultiplicativeExpression ( ( <PLUS> |
<MINUS>)MultiplicativeExpression)*
MultiplicativeExpression::=
ArithmeticUnaryExpression
( ( <MULT> |
      ……
```

```
ArithmeticUnaryExpression ::= ( <PLUS> |
ArithmeticUnaryExpression|
BooleanUnaryExpression BooleanUnaryExpression
::=(<TILDE>|<EXCL>)
ArithmeticUnaryExpression | PrimitiveExpression
PrimitiveExpression ::= Literal | Function |
<IDENTIFIER> | <LPAREN> Expression
<RPAREN> Function ::= <IDENTIFIER>
ArgumentsLiteral::=<INTEGER_LITERAL> |
<FLOATING_POINT_LITERAL> |
<STRING_LITERAL> | BooleanLiteral
BooleanLiteral ::= <TRUE> | <FALSE>
…….
```

Şekil 1.16. Bir gramere ait BNF dosyası

1.6.5.2. Gramerin JavaCC Dosyasına Dönüştürülmesi

Gramer yapısına ait genişletilmiş BNF dosyasının oluşturulmasından sonra JavaCC dosyasına çevrim işlemine geçilir. JavaCC dosyası temel dört bölümden oluşmaktadır. Birinci bölüm JavaCC'ye ait gramer yapısında ne tür işlemler yapılacağını belirleyen bazı kurallar ve özelliklerden oluşmaktadır. İkinci bölüm, ayrıştırma işlemlerine ait sınıf tanımlamalarının yapıldığı kısımdır. Bu bölümde tanımlanan sınıf ismi ve ayrıştırıcı sınıfının ismi aynıdır. Üçüncü bölüm, sözcüksel yapılara ait bilgilerin bulunduğu kısımdır. Dördüncü bölümde ise uygulamaya ait Java metotlarının ve sözdizimsel yapılara ait özelliklerin ve kuralların bulunduğu kısımdır. Yukarıda belirtilen dört bölüm ayrı başlıklar altında kod yapıları ile birlikte aşağıda incelenmiştir.

1.6.5.2.1. Özellikler Bölümü

JavaCC, ilgili programa ait gramer dosyasındaki işlemlerin nasıl gerçekleştirileceği ile ilgili bir kısım özellikleri destekler. Buna örnek;

```
options {
    LOOKAHEAD = 1;}
```

Tanımlanan kural kodu ayrıştırma işlemleri sürecinde karar verebilmek için en az bir kelimeye ihtiyaç duyulduğunu belirtilir.

1.6.5.2.2. Sınıf Tanımlama Bölümü

Temel ayrıştırıcı sınıflarının tanımlandığı bu bölüm Java programlama diline ait kodlardan oluşmaktadır. Ayrıca bu kod bölümü ayrıştırma sürecinde her hangi bir değişim işlemine tabi tutulmaz. İlgili dile ait gramer dosyası JavaCC ile derlenir bu esnada buradaki Java kodlarında herhangi bir kontrol işlemi yapılmaz. Bundan dolayı bu bölümde yazılan kodlarda hata olması durumunda dahi kod yapısında herhangi bir değişiklik yapılmadan sonraki aşamaya aktarılır.

Şekil 1.17'de bu kod bölümüne ait örnek bir kod bloğu verilmiştir. Bu örnekte [57] PARSER_BEGIN ve PARSER_END ifadeleri arasında yer alan ve temel sınıf özelliklerinin yer aldığı kısım sınıf tanımlama bölümüdür.

```
PARSER_BEGIN (ExpressionEvaluator)
public class ExpressionEvaluator {
public static void main(String[] args)
        throws ParseException {…}
private static Stack stack = new java.util.Stack();
private static Hashtable state = null;
public static boolean popBoolean()
        throws ClassCastException {
Boolean a = (Boolean) stack.pop();
if (a == null) return false;
        else return a.booleanValue(); }…}
PARSER_END(ExpressionEvaluator)
```

Şekil 1.17. Temel sınıf özelliklerinin tanımlandığı kod bloğu

1.6.5.2.3. Sözcüksel Kurallar ve Özellikler Bölümü

Sözcüksel özellikler bölümü olarak adlandırılan bu bölümde kullanılacak terminallere ait tanımlama işlemleri gerçekleştirilir. JavaCC bu bölümde tanımlanan terminal yapıları aracılığıyla ayrıştırma sürecinde bulunması gereken anlamlı ve en kısa kelimesel yapıların neler olduğunu bilir. Şekil 1.18'de bu kısma ait örnek [57] bir kod bloğu gösterilmiştir.

```
TOKEN:{
      < ASSIGN: "=" > | < GT: ">" > | < LT: "<" > | < EXCL: "!" >
     | < TILDE: "~" > | < EQ: "==" > | < LE: "<=" >| < GE: ">=" >
     | < NE: "!=" > | < OR: "||" > | < AND: "&&" > | < PLUS: "+" >
     | < MINUS: "-" > | < MULT: "*" > | < SLASH: "/" >  }
```

Şekil 1.18. Sözcüksel kurallar ve özellikler bölümü kod bloğu

1.6.5.2.4. Sözdizimsel Özellikler ve Kurallar Bölümü

Bu bölümde ilgili dile ait gramer yapısında tanımlanan kurallar için metotlar yazılır. Bu metotlar Java metotlarına benzer şekilde yazılırlar. Bununla birlikte Java kodlarına ihtiyaç duyulan yerlerde Java kodları küme parantezleri arasına yazılabilir. JavaCC'nin

BNF ve Java metotlarını birleştirebilme yeteneğine sahip güçlü bir yapısı vardır. JavaCC gramer dosyasında belirtilmiş kurallar için Java programlama dili yapısında metotlar oluşturur. Metodun ismi tanımlanan kuralın başında bulunan non-terminal ile aynı isme sahiptir. Metot isminden sonra Java kod yapısında olduğu gibi küme parantezleri ile belirlenen kod bloğu vardır. Bu kod bloğunda JavaCC tarafından çeşitli tanımlamalara izin verilir. Ayrıca burada yapılan tanımlamalar ilgili metot için genel bir yapıdadır. Bu bloğun devamında yine terminal ve non-terminal yapıları tanımlanır. Java metotlarına ihtiyaç duyulursa yine bu kısımda tanımlama ile ilgili Java metotları oluşturulabilir. Sözdizimsel kurallar bölümüne ait kod bloğunun bir bölümü aşağıdaki örnekte gösterilmiştir.

```
Exp parse() : {Exp a;}
{ a = expr()(<EOF> | <EOL>) { return a; } }
Exp expr() : {Exp a, b;}
{ a = term() (<PLUS>b = term(){ a = new Plus(a, b); }
    | <MINUS> b = term()    { a = new Minus(a, b); })*
{ return a; }}
```

1.7. Ayrıştırıcı Üretimi

Günümüzde yüksek seviyeli programlama dillerinde kullanılan çok sayıda ayrıştırıcı üreteci programları vardır. Bu bölümde tezin proje aşamasında kullanılan JavaCC ile ayrıştırıcı üreteci anlatılmıştır.

LL(k) gramer kuralları *.jj uzantılı dosyalar içerisinde tanımlanarak ayrıştırıcı üreteci olan JavaCC programına bildirilirler. *.jj dosyası üç ana birimden oluşmaktadır.

Bunlar;

- Seçenek bildirimi,
- Ayrıştırıcı sınıfının bildirimi,
- Terminal ve gramer kurallarının bildirimi olmak üzere üç bildirim bölümü

vardır.

Terminal'ler < ve > simgeleri arasında belirtilirken, kurallar ilgili programlama dilindeki fonksiyon yapılarına benzer biçimde tanımlanırlar. Burada dikkat edilmesi gereken durum metotlara ilgili kurallara ait non-terminal ile uyumlu isimler verilmelidir. Non-terminal ya da başka bir isimli metot biçimde bildirilir. Şekil 1.19'da gramer kuralına bir örnek verilmiştir.

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

Şekil 1.19. JavaCC dosyasındaki kurallar bildirimi bölümü

1.8. Sözdizim Sınıfları

Kaynak verinin sözdizim analizi sonucunda üretilecek nesne ağacı düğümleri için sözdizim sınıfları kullanılır. Bir sözdizim sınıfı bir yada birkaç gramer kuralını temsil etmek üzere tanımlanabilir. Genel olarak, bir sözdizim sınıfının ismi, ilgili kuralın temsil ettiği işlemden (işleç ya da işlev) türetilir ve kuralın içerdiği nonteminaller için sözdizim sınıfına birer alan verisi eklenir.

Bir dile ait en temel yapı birimi alfabesini oluşturan sembollerdir. Bu semboller dile ait gramer kurallarına uygun olarak birleşerek tanımlanan dile ait doğru ve anlamlı kelimeleri oluştururlar. Bu kelimelerde ilgilin dilin gramer kurallarına uygun bir şekilde birleşerek doğru ve anlamlı cümleleri oluştururlar. Sözdizim sınıfları ise gramer kuralları aracılığıyla üretilebilen girdi verilerini nesneye dayalı programlama yapılarıyla ifade etmek için oluşturulurlar. Şekil 1.20'de örnek olarak bazı sözdizim sınıflarının tanımlaması gösterilmiştir.

```
abstract class Exp { }
  class Plus extends Exp {
      public Exp exp1, exp2;
   public Plus(Exp e1, Exp e2)
       {exp1 = e1; exp2 = e2;}
       }
  class Euler extends Exp
   {
      public Exp exp;
      public Euler(Exp e) {exp = e;} }
  class Num extends Exp { public double num;
      public Num(double n) {num = n;
    }
    ……}
```

Şekil 1.20. Gramer sözdizim sınıflarının tanımlaması

1.8.1. Sözdizim Ağacı

Bir sözdizim ağacının yapısı birbirine belirli hiyerarşik kurallar çerçevesinde bağlanmış birçok düğüm kümesinden oluşmaktadır. Bu ağacın yapısındaki her bir düğüm sözdizim sınıflarından türetilmiş bir işlem yapısını veya bir veri yapısını gösteren bir nesneyi barındırır. Bununla birlikte sözdizim ağaç yapılarının bildirim işlemi çoğunlukla hiyerarşik olarak bir üst sınıf türü yardımıyla yapılmaktadır. Aynı üst sınıftan miras alan sözdizim sınıfları bir nesne ağacının düğümlerini oluşturmak için kullanılabilir, ancak üst sınıfı miras alan bu düğümler ağaç üzerinde üst sınıf türüyle temsil edilmektedirler.

Gramere ait sözdizim ağaç yapısının oluşturulmasındaki amaç anlamsal analiz ve kod yorumlama aşamalarında gerçekleştirilecek işlemler için uygun bir yapı meydana getirmektir.

Bu aşamada JavaCC ilgili işlemleri gerçekleştirmektedir. Gramer dosyasındaki non-terminaller JavaCC aracılığıyla birer sınıf yapısına dönüştürülürler. Bu dönüştürme işlemi sayesinde oluşturulan söz dizim ağacının her bir düğümünde, düğümlerin kontrolünün daha kolay yapılabilmesini sağlayacak daha önceden tanımlanmış nesne yapıları bulunacaktır. Böylece sözdizim ağacının yapısında kolayca hareket etme yeteneğine sahip olunabilecektir.

JavaCC tanımlamalarına çeşitli Java programlama diline ait yapıların eklenerek bir nesne ağacı oluşturulması mümkündür. Şekil 1.21'de matematiksel bir ifadeyi temsil edecek nesne ağacını üretmek için, JavaCC dosyasındaki gramer tanımlamalarına Java programlama diline ait ifadelerinin eklendiği gösterilmiştir.

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

Şekil 1.21. Gramer kural tanımlamalarına sözdizim ağacı üreten ifadelerin eklenmesi

1.8.2. Değerlendirme Yöntemleri

Bir nesne ağacının değerlendirme işlemi en içteki düğümden başlanarak kök düğüme doğru ilerlenerek gerçekleştrilir. Düğüm yapılarının her biri için yapılacak değerlendirme ilgili düğümün içerdiği nesnenin türüne bağlıdır. Nesne türünün belirlenmesi ve temsil edilen işlemin gerçekleştirilmesi üç farklı yöntem ile yapılabilmektedir. Bu yöntemlerin avantajları ve dezavantajları Tablo 1.3'de gösterilmiştir.

Tablo 1.3. Sözdizim ağacı değerlendirme yöntemlerinin karşılaştırılması

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

1.8.2.1. Instanceof İşleci

Bu yöntemde bir düğüme ait nesnenin türü intanceof işleci kullanılarak belirlenebilmektedir. Düğüme ait nesne türü belirleme işlemi yapıldıktan sonra temsil edilen işlemin gerçekleştirilebilmesi için üst sınıf referans değişkeninden alt sınıf nesnesinin türetilmesi gerekmektedir. Bu işlemde tür türetme (cast) yapısı kullanılarak işlemi temsil eden ilgili alt sınıf nesnesi türetilebilir. Tablo 1.4'te tanımlanan eval() metodu nesne ağacını ve değerlendirmenin yapılacağı x değerini parametre olarak alıp gerekli işlemleri gerçekleştirmektedir. Bu işleme ait kod yapısının bir kısmı Tablo 1.6'da verilmiştir.

Tablo 1.4. instanceof işleci ile değerlendirme

```
public class EvalExp {
public static void main(String[] args) {
             EvalParse parser = new EvalParse(System.in);try {
             eval(parser.parse(),e.printStackTrace();}}
public static double eval(Exp exp,double x)
            {if(exp instanceof Plus)} …}
```

1.8.2.2. Visitor Tasarım Deseni

Sözdizim sınıflarına ait yerel yapılar ile bu yapılar üzerinde tanımlanacak işlemleri ayırmak için Visitor tasarım deseni kullanılır. Visitor deseni genel tanım olarak, oluşturulan sözdizim ağacındaki düğümler arasında geçişi sağlamayı gerçekleştirir. Bu geçiş işlemlerinin gerçekleştirilmesi için Visitor sınıfına ait her bir sözdizim sınıf türü için bir visit() metodu ve her bir sözdizim sınıfına da bir accept() metodu eklenmelidir. Eklenen bu metotlar sırasıyla, visit() metodu değerlendirilecek düğüm nesnesinin accept() metodunu ve sonra accept() metodu ise değerlendirmeyi yapacak visit() metodunu çağırır. Bu şekilde visit() ve accept() metotları nesne ağacının bütün düğümlerine erişilinceye kadar birbirlerini çağırma işlemini gerçekleştirirler. Visitor sınıfı her bir sözdizim sınıfı için bir visit() metot bildirimi içeren bir arayüz işlevi görür. visit() metotlarının tanımlaması Visitor arayüzünü gerçekleyen bir sınıf içerisinde yapılır. Sözdizim sınıf nesnelerinin farklı bir değerlendirmesi diğer bir Visitor arayüzü tanımlamasını gerektirir.

Oluşturulan bir nesne ağacının değerlendirme işlemine Visitor nesnesin ait bir visit() metodunun çağrılmasıyla başlanır. visit() metodu ilk önce ağacın kök düğümüne erişmeye çalışır ve bundan dolayı kök düğümdeki nesnenin accept() metodunu mevcut Visitor nesne referansı ile çağırır. accept() metodu ise içinde bulunduğu nesne referansını arguman alan diğer bir visit() metodunu çağırarak yeni bir düğüm nesnesinin değerlendirmesini başlatır. Burada visit() ve accept() metotlarının çağrımı, değerlendirilen düğümdeki nesne türünün bilinmesini gerektirmezler.

Bu desen nesneye dayalı sistemleri daha esnek hale getirmeyi amaçlayan birçok desenden bir tanesidir. Visitor deseni birden çok nesneden oluşan yapıları kolayca kullanabilme imkânı verir. Visitor kullanılmadan uygulanacak diğer yöntemler çeşitli sorunlara neden olacaktır. Bu yöntemin avantajı var olan sınıfların tekrar derleme ihtiyacı duyulmadan, ilgili nesneleri kontrol edebilen kodlama işleminin yapılabilmesidir.

1.8.2.3. Sözdizim Sınıflarına Metot Ekleme

İlk yöntemde bahsedilen instanceof işlecinin kullanılması yöntemi nesne yönelimli bir yöntem değildir. Nesne yönelimli kodlama yapılarında nesnenin herhangi bir parçasına erişebilmek için geleneksel olarak kullanılan yöntem birbiriyle ilişkili metotlar yazmaktır. Bundan dolayı bu yöntemde her bir sözdizim sınıfına, sınıfın temsil ettiği işlemi yerine

getirebilecek bir eval() metodu eklenir. Bir nesne ağacı düğümünün değerlendirmesi için düğümün içerdiği nesneye ait eval() metodunun çağrılması yeterlidir. Dolayısıyla değerlendirilecek düğüm nesne türünün belirlenmesine gerek yoktur. Kullanılan yöntemlerin avantajları ve dezavantajları vardır. Bu yöntemin avantajı tip biçimleme ve instanceof yapılarından kullanılmayıp düzenli bir yöntemle kodlanmış yapıya sahip olmasıdır. Yöntemin dezavantajı ise kullanılacak her yeni işlem için her bir sınıfa özel yeni bir metot tanımlaması gerektirmesidir. Tablo 1.5'de değerlendirmenin yapılacağı x değerini parametre olarak alan eval() metodu tanımlamaları bazı sözdizim sınıfları için verilmiştir.

Tablo 1.5. Sözdizim sınıflarına eval() metotlarının eklemesi

```
abstract class Exp {public abstract double eval(double x);  }
class Plus extends Exp {
       public Exp exp1, exp2;
       public Plus(Exp e1, Exp e2) {
              exp1 = e1;
              exp2 = e2;}
       public double eval(double x){
              return  (exp1.eval(x)+exp2.eval(x));}}
class Euler extends Exp {public Exp exp;
       public double eval(double x){
              return Math.pow(Math.E,exp.eval(x));}}
```

1.9. Anlamsal Analiz

Anlamsal analizin amacı, sözdizimsel analiz sonucu oluşan ayrıştırma ağaçlarından gramer yapısı olarak doğru, ancak anlamsal ifade olarak yanlış ağaçların elenmesidir. Bir anlamsal analiz sistemi bir metni analiz ettikten sonra o metinle ilgili soruları cevaplayabilmektedir.

Ayrıştırma ağacı oluşturulduktan sonra, anlamsal analiz işlemine tabi tutulur. Ayrıştırma ağacının yapraklarında bulunan token ifadeleri, bu aşamada derleyici için bir anlam ifade etmeye başlar. Örneğin; derleyici tokenları integer, değişken, string gibi yapılarla anlamlandırır böylece ilgili token programda derleyici tarafından yüklenen anlama göre işlemlere tabi tutulur. Bu işlemlerin sonucu olarak anlamsal olarak bazı hatalar meydana gelebilmektedir. Bu aşamada oluşabilecek hatalara semantik hatalar denir.

Örneğin; integer olan bir değişken, herhangi bir dönüşüm işlemine tabi tutulmadan string olan bir değişkene doğrudan eşitlenemez çünkü bunların token tipleri derleyici tarafından farklı olarak anlamlandırılmıştır.

Bu bölümde, kaynak programa ait kod yapısındaki anlamsal hatalar kontrol edilir ve kod üretim işlemleri için veri tipi bilgileri belirlenir. Tip kontrolü anlamsal analizin en önemli kısmıdır. Sözdizimsel analiz işlemlerinde kullanılan CFG, bu aşamada belirlenen anlamsal kurallar ile birleştirilir.

Anlamsal analiz aşaması aşağıdaki gibi özetlenebilir;

- Derleyici her bir yaprakta bulunan token ifadelerini anlamlandırır ve bu anlamlara göre kullanımları ile ilişkilendirir,
- Tokenların tip bilgilerinin doğruluğunu kontrol edilir,
- Sözdizim ağaç yapısını kontrol edilip yorumlama aşamasına geçmeden önce son kez kodu yapısı kontrol edilir.

Anlamsal analizde iki temel aşama bulunmaktadır.

Bu aşamalar;

- Sembol tablosunun oluşturulması,
- Tip kontrolünün gerçekleştirilmesi

1.9.1. Sembol Tablosu

Bu aşamada tanımlayıcılar, tanımlayıcı tipleri ve tanımlayıcı içerikleri bilgisi sembol tablosuna kaydedilir. Kaynak veri kodunda tip bildirimleri, değişken ve fonksiyon tanımlamaları gibi programsal yapılar tanımlandığında bu bilgileri sembol tablosunda oluşturulur. Kaynak veri kodunun sonraki bölümlerinde kullanılan tanımlayıcılarla ilgili kararlar sembol tablosuna bakılarak verilir.

1.9.2. Tip Kontrol Tablosu

Kaynak veri kodunda kullanılan tanımlayıcıların tip bilgileri sembol tablosuna ekleme işlemi gerçekleştirildikten sonra, bu tanımlayıcıların uygun şekilde kullanılıp kullanılmadığının kontrol edilme işlemi tip kontrolü olarak isimlendirilir.

1.10. Sayısal Yöntemler

Sayısal yöntemler matematik problemlerinin formülüze edilip, çeşitli matematiksel işlemlerle çözülmesini sağlayan tekniklerdir. Çok sayıda ve türde sayısal yöntem bulunmaktadır. Bu yöntemlerin genel özelliği çok miktarda karmaşık matematiksel işlemler içermeleridir. Bu yöntemlerin belirtilen özelliğinden dolayı matematiksel problemlerin çözümleme işlemlerinden yeni problemler oluşabilmektedir. Bu problemleri ortadan kaldırmak için bilgisayar programlarından yararlanılmaktadır.

Bu çalışmada sayısal yöntemler kullanılarak doğrusal olmayan denklemlerin çözümlemeleri incelenmiştir.

1.10.1. Basit İterasyon Yöntemi

Basit İterasyon Yöntemi ile $f(x) = 0$ denklemi içerisindeki her bir x değeri için ilgili fonksiyon $x = g(x)$ şekline getirilir. Bu dönüşümden elde edilen $g(x)$ fonksiyonları için yakınsama koşulu olan $|g'(x)| < 1$ ifadesi kontrol edilerek kök hesaplama işleminde kullanılacak fonksiyona karar verilir ve yinelenecek iterasyonlar için bir hata oranı (ε) belirlenir.

Yakınsama koşulu $[a, b]$ aralığındaki bütün x değerleri için $|g'(x)| < 1$ şeklinde genel bir kurala sahiptir. İlgili koşul sağlandıktan sonra $x_{n+1} = g(x_n)$ ifadesi ile kök hesaplama işlemi için kök hesaplama adımlarına başlanır ve $|x_n - x_{n+1}| < \varepsilon$ oluncaya kadar devam eder.

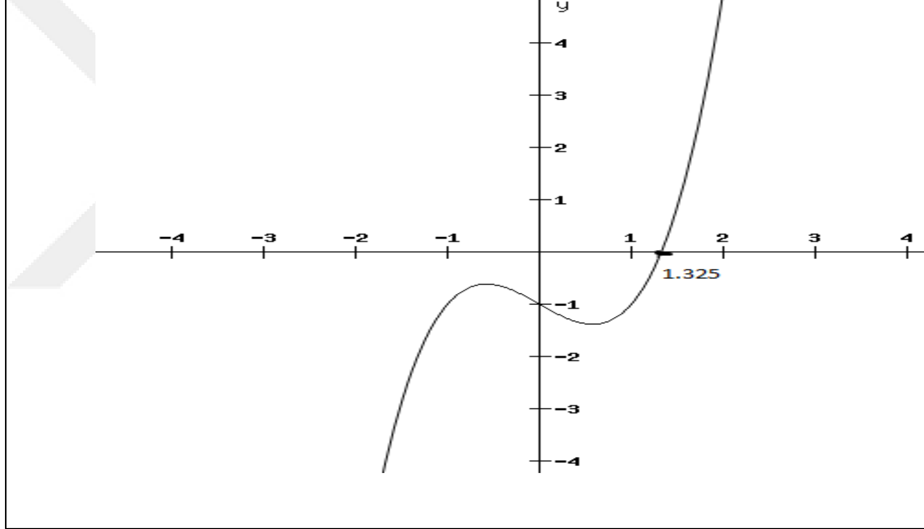
Bu yöntem için kök aralığı [0,2] olduğu bilinen $f(x) = x^3 - x - 1 = 0$ fonksiyonun çözümlenmesi; $x = x^3 - 1 = g_1(x)$, $x = 1/(x^2 - 1) = g_2(x)$, $x = (x + 1)^{1/3} = g_3(x)$ şeklinde kök hesaplaması için üç adet fonksiyon elde edilmiştir. Yine bu denklem için başlangıç değeri $x_0 = 1.9$ olarak belirlenmiştir. Daha sonra yakınsama koşulu için elde edilen bu denklemlerin her birine türev işlemi uygulanacaktır. Böylece koşulu sağlayan denklem ilgili fonksiyon için kök hesaplama işleminde kullanılacaktır.

$g'_1(x) = 3x^2$, $g'_2(x) = (-2x/(x^2 - 1)^2)$, $g'_3(x) = \frac{1}{3}(x + 1)^{-\frac{2}{3}}$ işlemlerinden sonra başlangıç değeri her bir denklemde yerine konularak koşul ifadesi kontrol edilir.

$|g'_1(x_0)| > 1, \quad |g'_2(x_0)| > 1, \quad |g'_3(x_0)| < 1$. Problemin çözümünde koşulu sağlayan denklem olan $x = (x+1)^{1/3} = g_3(x)$ kullanılır. Tablo 1.6'da hesaplanan kök değerleri gösterilmiştir. Ayrıca Şekil 1.22'de fonksiyonun grafiği gösterilmiştir.

Tablo 1.6. $x^3 - x - 1$ fonksiyonuna ait hesaplanan kök değerleri

İterasyon Sayısı	x	y
0	1.9	1.426
1	1.426	1.344
2	1.344	1.328
3	1.328	1.325
4	1.325	1.325



Şekil 1.22. $x^3 - x - 1$ fonksiyonuna ait grafik

Tablo 1.7. Basit İterasyon Yöntemine ait algoritma

Adım 1: Köke yakın tahmini bir x_0 başlangıç değerinin belirlenmesi
Adım 2: $f(x) = 0$ eşitle fonksiyondaki her bir x değeri için $x = g(x)$ formunda yeniden düzenle.
Adım 3: $\lvert g'(x) \rvert < 1$ yakınsama koşulunu kontrol edilmesi ve kullnılacak fonksiyonu belirle.
Adım 4: $x_{i+1} = g(x_i)$ denkleminde kök değerlerinin hesapla.
Adım 5: $\varepsilon_a = \lvert (x_{i+1} - x_i)/x_{i+1} \rvert * 100 \leq \varepsilon_s$ ise işlemi sonlandır, değilse $x_i = x_{i+1}$ Adım 3'e git
Adım 6: Bitir.
ε_a : Hesaplanan hata oranı
ε_s : Kök hesaplama işlemine başlamadan önce belirlenen hata oranı

1.10.2. İkiye Bölme Yöntemi

İkiye Bölme Yönteminde Ara Değer Teoremi kullanıldığı için yönteme geçmeden önce teoremin tanımlaması verilmiştir.

Ara Değer Teoremi:

$f \in C[a,b]$ ve K sayısı $f(a)$ ile $f(b)$ arasında herhangi bir sayı olsun. Buna göre (a,b) aralığında $f(c) = K$ eşitliğini sağlayan bir c sayısı vardır.

İkiye Bölme Yöntemi, $f(x)$ fonksiyonunun bir $[a,b]$ aralığında sürekli ve bir köke sahip olduğu biliniyorsa, $f(x) = 0$ formunda bir denklemin yaklaşık kökünü bulmak için bir parantez açma yöntemidir. Böyle bir durumda $f(x)$ fonksiyonu çözüm aralığının uç noktalarında zıt işaretli olacaktır.

Örneğin, $f(x) = x^3 + 4x^2 - 10 = 0$ denkleminin $[1,2]$ aralığında bir kökü olduğunu İkiye Bölme metodunu kullanarak bu aralıktaki kök değerinin, 0.001 hata duyarlılığı ile hesaplanması; çözüm için ilk önce başlangıç ve bitiş değerlerinin ilgili fonksiyonda değerlerine bakılır, $f(1) = -5 < 0$ ve $f(2) = 14 > 0$ olduğundan Ara Değer Teoremi'ne göre sürekli $f(x)$ fonksiyonunun verilen aralıkta en az bir kökü vardır.

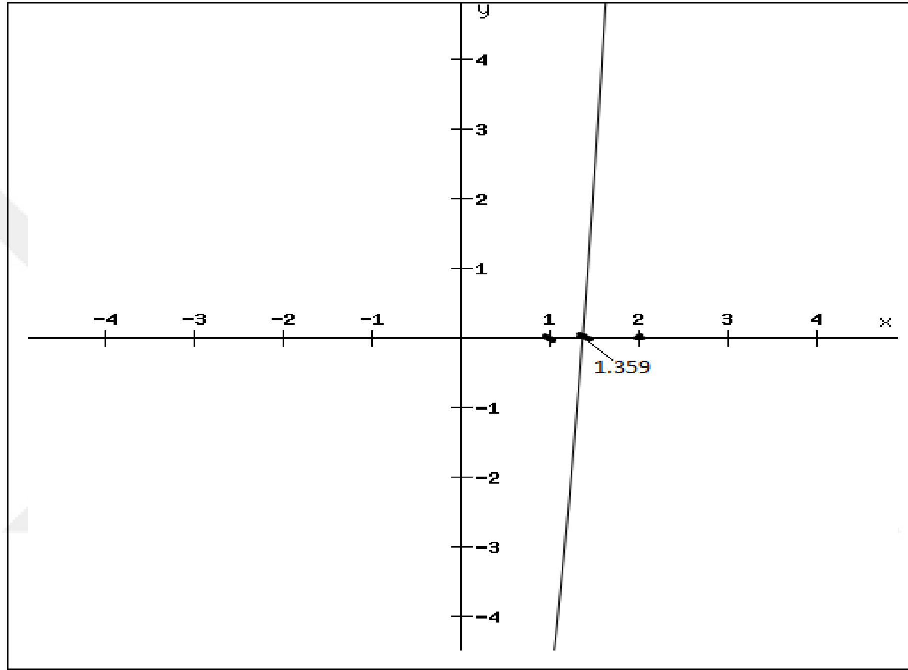
İkiye bölme yönteminin ilk basamağında $[1,2]$ aralığının orta noktası 1,5 değeri dikkate alınarak $f(1{,}5) = 2{,}375$ olarak hesaplanır. $f(1{,}5) > 0$ olduğundan elde edilecek kök değerinin $[1, 1.5]$ aralığında yer aldığına karar verilir. Böylece yeni aralık $[1, 1.5]$ olarak belirlenmiş olur. Elde edilen yeni aralığın orta noktası 1,25 olarak hesaplanır ve bu değer için ise $f(1{,}25) = -1{,}796875$ olarak hesaplanır. $f(1{,}25) < 0$ olduğundan kök değerinin $[1.25, 1.5]$ aralığında bulunduğu sonucuna varılır. $[1.25, 1.5]$ aralığının orta noktası 1,375 değeri icin $f(1{,}375) = 0{,}16211$ olarak hesaplanır $f(1{,}375) > 0$ olduğundan verilen denkleme ait kökün $[1, 1.375]$ aralığında yer aldığı sonucuna ulaşılır. Bu şekilde hata oranı göz önünde bulundurularak çözüm işlemleri yinelenerek bulunacak kök değerine daha fazla yaklaşılmaya çalışılır ve böylece en doğru değer bulunmuş olur. Tablo 1.7'da örneğe ait kök değerleri ile Şekil 1.23'de fonksiyonun grafiği gösterilmiştir

Tablo 1.8. $f(x) = x^3 + 4x^2 - 10$ fonsiyonuna ait hesaplanan kök değerleri

İterasyon Sayısı	x	z=(x+y)/2	y
0	1	1,5	2

Tablo 1.8’in devamı

1	1	1,25	1,5
2	1,25	1,375	1,5
3	1,25	1,312	1,375
4	1,312	1,344	1,375
5	1,344	1,359	1,375



Şekil 1.23. $x^3 + 4x^2 - 10$ fonksiyonuna ait grafik

Tablo 1.9. İkiye Bölme Yöntemine ait algoritma

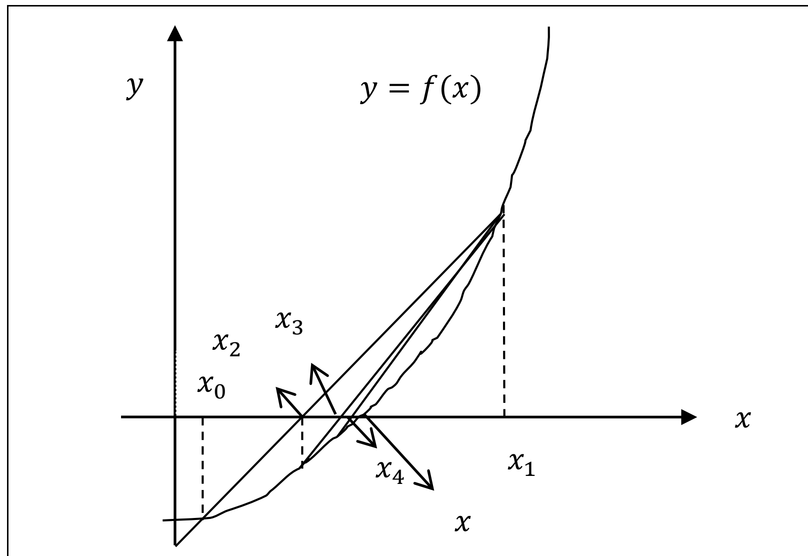
Adım 1: Aralığın başlangıç ve bitiş değerleri x_0 ve x_1 değerini belirle. Adım 2: $x_z = (x_0 + x_1)/2$ değerini hesapla. Adım 3: Eğer $f(x_0) * f(x_z) < 0$ ise yeni kök aralığı $[x_0, x_z]$, x_1=x_z Değilse x_0=x_z Adım 4: $\varepsilon_a = \lvert(x_{i+1} - x_i)/x_{i+1}\rvert * 100 \leq \varepsilon_s$ ise işlemi sonlandır, değilse Adım 2’e git Adım 5: Bitir ε_a : Hesaplanan hata oranı ε_s : Kök hesaplama işlemine başlamadan önce belirlenen hata oranı

1.10.3. Regula Falsi Yöntemi

Regula Falsi Yöntemi, İkiye Bölme Yöntemi ile Secant Yöntemi'nin birlikte kullanılmasıyla oluşan bir yöntemdir. Bu yöntemde Secant Yöntemi kullanılarak iterasyonlar oluşturulur, aynı zamanda İkiye Bölme yönteminde olduğu gibi her bir işlem adımında hesaplanacak kök değerini içeren aralık test edilerek ilerlenir. Bu şekilde kök değerlerinin hesaplandığı yönteme Regula Falsi yöntemi denir.

Regula Falsi Yöntemi ile önce $f(x_0)f(x_1) < 0$ koşulunu sağlayan x_0 ve x_1 başlangıç değerleri seçilir. Daha sonra Secant metodunda elde edilen iteratif formül kullanılarak, $(x_0, f(x_0))$ ve $(x_1, f(x_1))$ noktalarını birleştiren doğrunun x eksenini kestiği nokta olan x_2 değeri bulunur. x_3 değerini elde etmek için $f(x_0)$, $f(x_1)$ ve $f(x_2)$ değerlerinin işaretlerine bakılır. Eğer $f(x_1)f(x_2) < 0$ ise $(x_1, f(x_1))$ ve $(x_2, f(x_2))$ noktalarını birleştiren doğrunun x eksenini kestiği nokta x_3 değeri olarak bulunmuş olur. Eğer $f(x_0)f(x_2) < 0$ ise $(x_0, f(x_0))$ ve $(x_2, f(x_2))$ noktalarını birleştiren doğrunun x eksenini kestiği nokta x_3 değeri olarak hesaplanmış olur. Bu işlem adımları tekrarlanarak x_4, x_5, …x_n, değerleri hesaplanmış olur.

Regula Falsi yönteminine ait grafik Şekil 1.24'de gösterilmiştir. Grafikte görüldüğü gibi $[x_0 , x_1]$ aralığında aranan köke daha fazla yaklaşmak için kök hesaplama işlemleri yinelenmektedir. Bu yineleme işlemi duyarlılık oranına göre gerçekleştirilmektedir. Yine grafikte görüldüğü gibi beş iterasyonda kök hesaplama işlemi sonlandırılmaktadır.



Şekil 1.24. Regula Falsi Metodu

Şekil 1.24'de gösterildiği gibi Regula Falsi metodunda ardaşık değer noktalarını birleştiren doğru parcasının x eksenini kestiği yeni değer noktasının işaretinin kendinden önceki iki değerin işaretleri ile karşılaştırılması yapılmaktadır ve böylece hesaplanacak kök değerlerine ulaşılmaya çalışılmaktadır.

$2x^3 - 2.5x - 5$, fonksiyonu için $[1, 2]$ aralığında Regula Falsi yöntemi kullanılarak çözümü; bu problemin çözümünde ilk önce $f(1)f(2) < 0$ durumu kontrol edilir, eğer şart sağlanıyorsa bu aralıkta ilgili fonksiyona ait kök olduğu anlamına gelmektedir.

$f(1) = 2(1^3) - 2.5(1) - 5 = 5.5$ ve $f(1) = 2(2^3) - 2.5(2) - 5 = 6$ işlemler sonucunda elde edilen değerler çarpma işlemine tabi tutulursa $f(1)f(2) < 0, (5.5)(6) = -33 < 0$ koşulu gerçeklendiği için kök hesaplama işlemleri gerçekleştirilir.

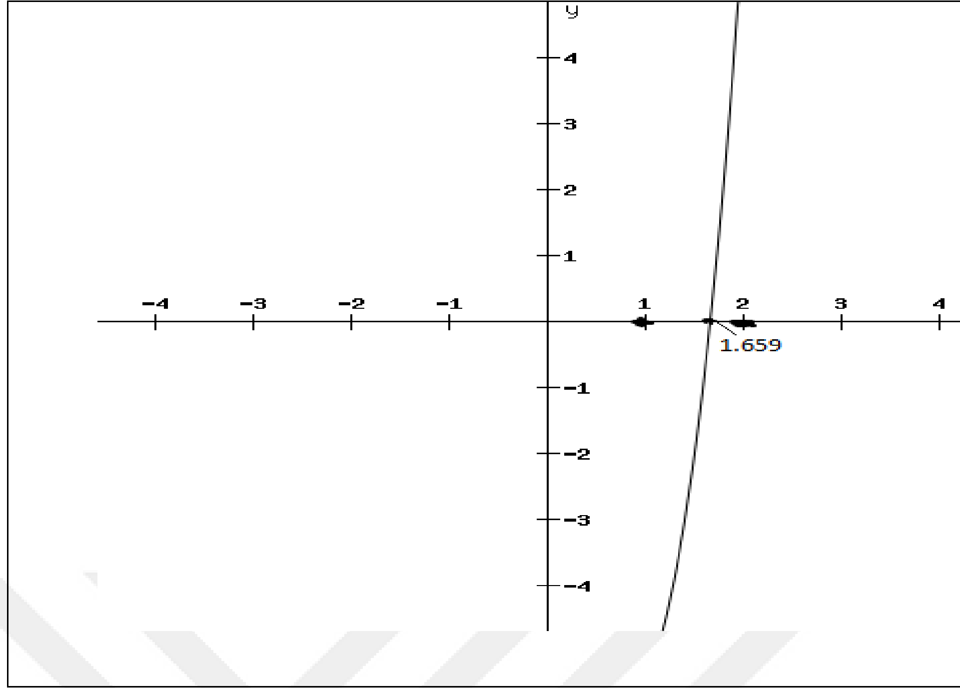
$x_2 = (f(2)1 - f(1)2)/\big(f(2) - f(1)\big) = 1.4782608747$ bu şekilde ilk adımda ilk kök değeri hesaplanır ve $x_1 = x_2$ atama işlemi gerçekleştirilir. $f(1)f(1{,}4782) < 0$, $f(2)f(1{,}4782) > 0$ koşullarına bakılarak kök aralığı belirlenir ve yeni aralık sıfırdan küçük koşulunu sağlayan $[1, 1.4782]$ aralığı olur.

$x_3 = (f(1.4782)1 - f(1)1.4782)/\big(f(1.4782) - f(1)\big) = 1.6198574305$ değeri elde edilir ve $x_1 = x_3$ atama işlemi gerçekleştirilir.

Bu aşamadan sonra elde edilen kök değerleri verilen koşullara göre aynı şekilde yinelenerek kök hesaplama işlemi gerçekleştirilir. Tablo 1.8'de fonksiyona ait kök değerleri ve Şekil 1.25'de fonksiyonun grafiği gösterilmiştir.

Tablo 1.10. $2x^3 - 2.5x - 5$ fonksiyonuna ait hesaplanan kök değerleri

İterasyon Sayısı	x_1	x_2	x_n
0	1	2	1.478
1	1.478	2	1.619
2	1.619	2	1.651
3	1.651	2	1.658
4	1.658	2	1.659



Şekil 1.25. $2x^3 - 2.5x - 5$ fonksiyonuna ait grafik

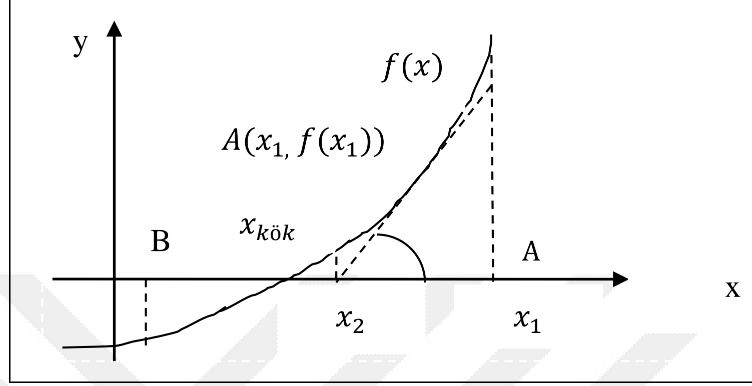
Tablo 1.11. Regula Falsi Yöntemine ait algoritma

Adım 1: Aralığın başlangıç ve bitiş değerleri x_0 ve x_1 değerini belirle.
Adım 2: Eğer $f(x_0) * f(x_1) < 0$, $x_2 = (x_0 * f(x_1) - x_1 * f(x_0))/f(x_1) - f(x_0)$.
Adım 3: Eğer $f(x_0) * f(x_2) < 0$ ise, x_1=x_2
Değilse x_0=x_2
Adım 4: $\varepsilon_a = \|(x_{i+1} - x_i)/x_{i+1}\| * 100 \leq \varepsilon_s$ ise işlemi sonlandır, değilse Adım 2'e git
ε_a : Hesaplanan hata oranı
ε_s : Kök hesaplama işlemine başlamadan önce belirlenen hata oranı

1.10.4. Newton-Raphson Yöntemi

Bu çalışmada kök hesaplama işlemleri için kullanılan sayısal yöntemlerden birisi de Newton-Raphson Yöntemidir. Bu yöntem $f(x)$ fonksiyonlarının grafiksel olarak çözümünü bulmak için kullanılmaktadır. Bu yöntemde, fonksiyonu sıfır yapan kök değeri, $f(x)$ fonksiyonunun x noktasındaki eğiminden yararlanarak bulunur. Şekil-1.26'da belirtilen alt indis numaraları ise işlem adımını belirtmek için kullanılmıştır. $f(x)$ fonksiyonuna A noktasında teğet olan doğrunun x eksenini kestiği x_2 noktası ilgili

fonksiyona ait kök hesaplama işlemini gerçekleştirmek için bir başlangıç değeri olarak kullanılmıştır. Şekil 1.26'da Newton-Raphson Yönteminin kök hesaplama işlemlerinde kullanım şekli ilgili değerler göz önünde bulundurularak çözüm adımlarının nasıl gerçekleştirildiği gösterilmiştir.



Şekil 1.26. Newton-Raphson yönteminin grafiksel gösterimi

Şekil 1.23'de x_1 noktası ile gösterilen eğrinin eğimi,

$$f'(x_1) = Tan(\alpha) = (f(x_2) - f(x_1))/(x_2 - x_1) \quad (1.1)$$

(1.1) denklemi ile ifade edilir. Bu denklemdeki $f'(x_1)$ terimi, $f(x)$ fonksiyonunun $x = x_1$ noktasındaki türevine ait değerdir. Gafikte ifade edilen x_2 değeri ise, ulaşılacak olan kök değerinin yaklaşık değeri olup bulunacak kök değerine belirtilen hata oranına ulaşılıncaya kadar kök hesaplama işlemlerine devam edilir. Denklem (1.1) den x_2 ifadesi (1.2) deki denklemsel ifade gibi yazılabilir.

$$x_2 = x_1 - (f(x_1) / f'(x_1)) \quad (1.2)$$

(1.2) denklemi genelleştirilirse (1.3) denklemi elde edilmiş olur.

$$x_{n+1} = x_n - (f(x_n) / f'(x_n)) \quad (1.3)$$

(1.3) denklemini kullanarak ilgili fonksiyona ait kök bulma işlemleri gerçekleştirilmiş olur.

Newton-Raphson yöntemi kullanılarak bir fonksiyona ait kök hesaplama işlemleri gerçekleştirilirken yapılan hata, (1.4) deki denklemsel yapı ile ifade edilebilmektedir.

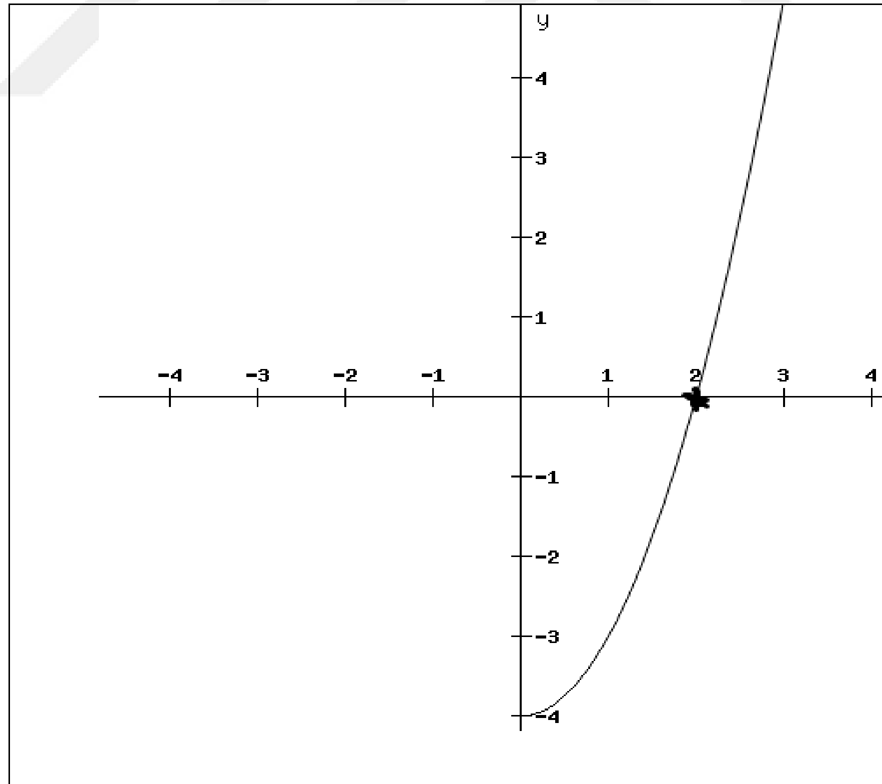
$$\varepsilon_{n+1} = x_{n+1} - x_n \quad (1.4)$$

Örneğin, $x^2 - 4 = 0$ denkleminin $x_0 = 6.0$ başlangıç değerine ve 0.001 hata oranına göre kök hesaplaması aşağıda gerçekleştirilmiştir; $f(x) = x^2 - 4$ denklemi türev

alma işlemine tabi tutularak $f'(x) = 2x$ denklemi elde edilir. (1.3) denklemi kullanılarak bu problemin çözümlenmesi; $x_{n+1} = x_n - ((x_n^2 - 4)/(2x_n))$ ifadesi şeklinde yazılır. Daha sonra soruda verilen hassaslık oranına ve başlangıç değerine göre çözümleme işlemi gerçekleştirilir. Tablo 1.9'da iterasyon sayısı ve fonksiyona ait hesaplanan kök değerleri ile Şekil 1.27'de fonksiyona ait grafik gösterilmiştir.

Tablo 1.12. $x^2 - 4$ fonksiyonuna ait hesaplanan kök değerleri

İterasyon Sayısı	x	y
0	6	3,333
1	3,333	2,267
2	2,267	2,016
3	2,016	2
4	2	2
5	2	2



Şekil 1.27. $x^2 - 4$ fonksiyonuna ait grafik

Tablo 1.13. Newton-Raphson Yöntemine ait algoritma

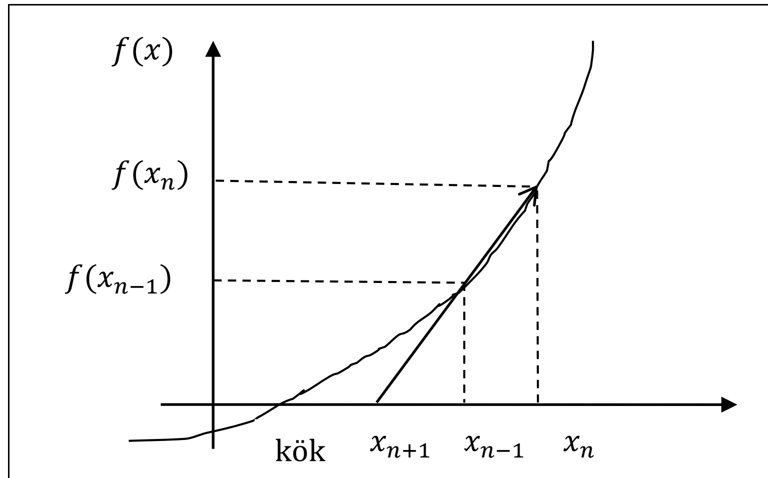
Adım 1: Köke yakın tahmini bir x_0 başlangıç değerinin belirlenmesi
Adım 2: $f(x)$ ve $f'(x)$ değerlerini hesapla.
Adım 3: $x_{i+1} = x_i - (\frac{f(x_i)}{f'(x_i)})$ yeni bir kök değeri hesapla.
Adım 4: $\varepsilon_a = \lvert (x_{i+1} - x_i)/x_{i+1} \rvert * 100 \leq \varepsilon_s$ ise işlemi sonlandır, değilse $x_i = x_{i+1}$ Adım 2'ye git
Adım 5: Bitir.
ε_a : Hesaplanan hata oranı
ε_s : Kök hesaplama işlemine başlamadan önce belirlenen hata oranı

1.10.5. Sekant Yöntemi

Newton-Raphson yöntemi için gerekli olan türev alma işlemi bazı polinom ve fonksiyonlarda zordur. Bu nedenle yöntemde türev alma yerine sonlu farklar türev formülü kullanılır. Newton–Raphson Yöntemine ait $x_{n+1} = x_n - (f(x_n)/f'(x_n))$ denklemindeki $f'(x_n)$ değeri yerine $f'(x_n) = (f(x_{n-1}) - f(x_n))/(x_{n-1} - x_n)$ değeri alınarak denklem (1.5) elde edilir. Elde edilen bu denklem kök hesaplama işlemlerinde kullanılır.

$$x_{n+1} = x_n - (f(x_n)\,((x_{n-1} - x_n))/(f(x_{n-1}) - f(x_n))) \quad (1.5)$$

Secant Yöntemine ait formülasyonu (1.5) denklemi şeklinde yazılır. Yöntemde hesaplama işlemlerine geçmeden önce x_n ve x_{n-1} değerleri başlangıçta verilir. Daha sonra ilgili denkleme ait kök hesaplamaları gerçekleştirilir.



Şekil 1.28. Secant Yöntemine ait grafik

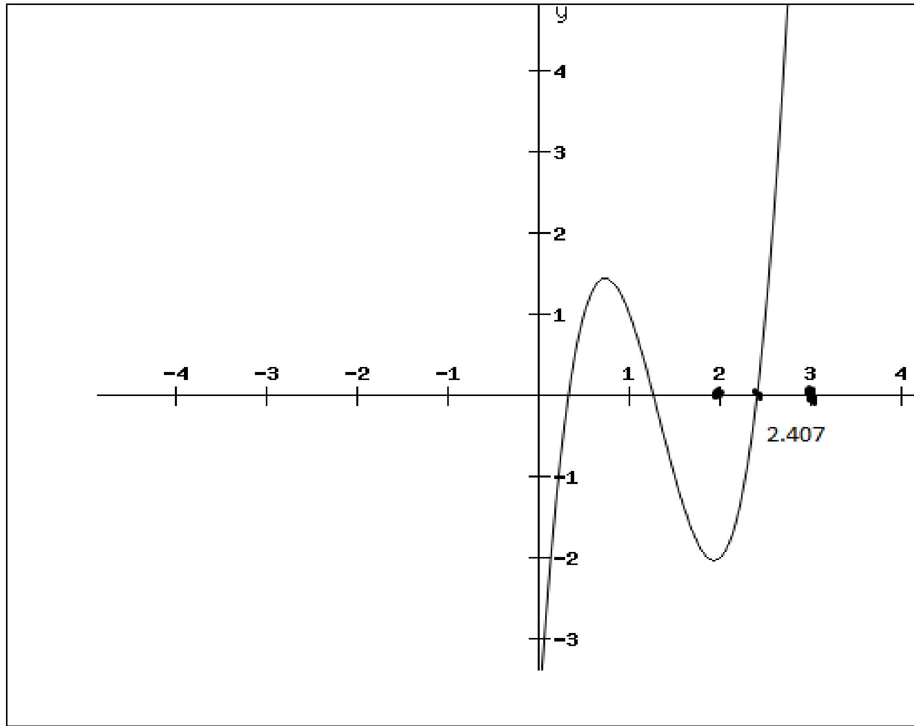
Örneğin, $4x^3 - 16x^2 + 17x - 4$ denklemine ait kök değerleri Secant Yöntemi kullanılarak aşağıda gerçekleştirilmiştir. Denklem için başlangıç değerleri $x_0 = 3$, $x_1 = 2.8$ ve hata oranı da 0.001 olarak belirlenmiştir. İlgili fonksiyonun çözümü için (1.5) deki denklem yapısı kullanılarak hesaplama yapılabilir.

$$x_{n+1} = x_n - \left(\frac{(4x_n^3 - 16x_n^2 + 17x_n - 4)\,(x_{n-1} - x_n)}{(4x_{n-1}^3 - 16x_{n-1}^2 + 17x_{n-1} - 4) - (4x_n^3 - 16x_n^2 + 17x_n - 4)}\right)$$

denkleminde işlemler gerçekleştirildiğinde Tablo 1.10'daki değerler elde edilecektir. Ayrıca fonksiyona ait grafik Şekil 1.29'da gösterilmiştir.

Tablo 1.14. $4x^3 - 16x^2 + 17x - 4$ fonksiyonuna ait hesaplanan kök değerleri

İterasyon Sayısı	x_0	x_2	x_1
0	3	2.562	2.8
1	2.8	2.459	2.562
2	2.562	2.415	2.459
3	2.459	2.4074	2.415



Şekil 1.29. $4x^3 - 16x^2 + 17x - 4$ fonksiyonuna ait grafik

Tablo 1.15. Sekant Yöntemine ait algoritma

Adım 1: Aralığın başlangıç ve bitiş değerleri x_0 ve x_1 değerini belirle.
Adım 2: $x_{i+1} = x_i - (f(x_i) * (x_i - x_{i-1})/(f(x_i) - f(x_{i-1}))$ değerini hesapla.
Adım 3: $\varepsilon_a = \lvert(x_{i+1} - x_i)/x_{i+1}\rvert * 100 \leq \varepsilon_s$ ise işlemi sonlandır, değilse Adım 2'e git
Adım 4: Bitir
ε_a : Hesaplanan hata oranı
ε_s : Kök hesaplama işlemine başlamadan önce belirlenen hata oranı

1.10.6. Halley Yöntemi

Halley Yöntemi $f(x) = 0$ ve $f'(x)$, $f''(x)$, $f'''(x)$ fonksiyonları sürekli olduğu zaman kök hesaplamarı için kullanışlı ve Newton-Raphson yöntemine göre daha hızlı çalışan bir yöntemdir. Yöntem ismini yöntemin geliştiricisi olan Edmond Halley'den almıştır ve Newton-Raphson yönteminin değiştirilmesiyle elde edilmiştir.

$$x_{n+1} = x_n - (f(x_n)\,/f'(x_n)\,) \quad (1.6)$$

Denklem (1.6)'daki Newton-Raphson Yöntemi Denklem (1.7)'deki gibi düzenlenerek Halley Yöntemi elde edilmiştir.

$$x_{n+1} = x_n - (2f(x_n)\,f'(x_n)\,/2[f'(x_n)]^2 - f(x_n)f''(x_n)) \quad (1.7)$$

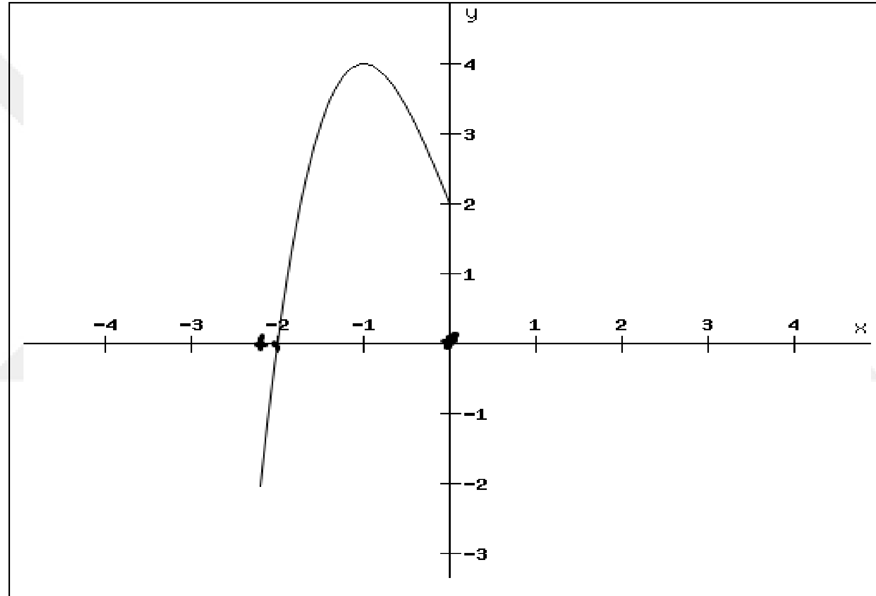
Halley's yönteminde x_0 gibi bir başlangıç değeri vardır. Halley's yönteminde kök hesaplama işlemi için $f(x) = x^3 - 3x + 2$ fonksiyonuna ait kök değerlerinin hesaplanması; bu fonksiyon için başlangıç değeri $x_{0=-}2{,}2$ dir.

$$h(x) = x - ((f(x)/f'(x))\left(1 - (f(x)f''(x)/2(f'(x))^2)\right)^{-1}) \quad (1.8)$$

Halley's iterasyon fonksiyonu olan Denklem (1.8) kullanılarak problemin çözümüne başlanmıştır. $(x) = x - (x^3 - 3x + 2/3x^2 - 3)(1 - (3x(2 - 3x + x^3)/(-3 + 3x^2)^2))$. Kök hesaplama işlemleri için gerekli matematiksel işlemler gerçekleştirildikten sonra denklem $h(x) = (x^3 + 2x^2 + 4x + 2)/(3 + 4x + 2x^2)$ şeklini almıştır. Bu denklem üzerinden kök hesaplama işlemleri için başlangıç değerinden başlanarak iteratif olarak işlemler gerçekleştirilmiştir. Hesaplama işlemleri sonucunda elde edilen kök değerleri Tablo 1.11'de gösterilmiştir. Şekil 1.30'da ise $x^3 - 3x + 2$ fonksiyonun grafiksel gösterimi verilmiştir.

Tablo 1.16. $x^3 - 3x + 2$ fonksiyonuna ait hesaplanan kök değerleri

İterasyon Sayısı	x_1
0	-2.2000000000000000
1	-2.0020618556701031
2	-2.0000000029138018
3	-2.0000000000000000
4	-2.0000000000000000
5	-2.0000000000000000



Şekil 1.30. $x^3 - 3x + 2$ fonksiyonuna ait grafik

Tablo 1.17. Halley Yöntemine ait algoritma

Adım 1: Aralığın başlangıç x_0 değerini belirle.
Adım 2: $x_{n+1} = x_n - (2f(x_n)\, f'(x_n)\ /2[f'(x_n)]^2 - f(x_n)f''(x_n))$ değerini hesapla.
Adım 3: $\varepsilon_a = \lvert(x_{n+1} - x_n)/x_{n+1}\rvert * 100 \leq \varepsilon_s$ ise işlemi sonlandır, değilse Adım 2'e git
ε_a : Hesaplanan hata oranı
ε_s : Kök hesaplama işlemine başlamadan önce belirlenen hata oranı

2. YAPILAN ÇALIŞMALAR

2.1. Giriş

Bilgisayar programları ile matematiksel problemlerin çözümünde çok farklı hesaplama metodolojileri ve programlama yöntemlerinin kullanımına ihtiyaç vardır. Genel olarak, sayısal ve simgesel yaklaşımlar hesaplama metodolojilerinin iki ana sınıfını meydana getirir. Sayısal yöntemler belirli bir hata payı içerir ve bir problemin çözümüne yönelik gerçekleştirilen hesaplamalar belirli bir hata oranı veya belirli bir iterasyon sayısına ulaşıncaya kadar yinelenir. Simgesel hesaplama yöntemleri ise matematiksel problemlerin bilgisayar programları yardımıyla tam ve hatasız olarak çözümünü bulmaya dayanır. Bu hesaplama işlemlerinin elle çözümü uzun, hataya açık ve bazen de klasik yöntemlerle çözümü mümkün olmayabilir.

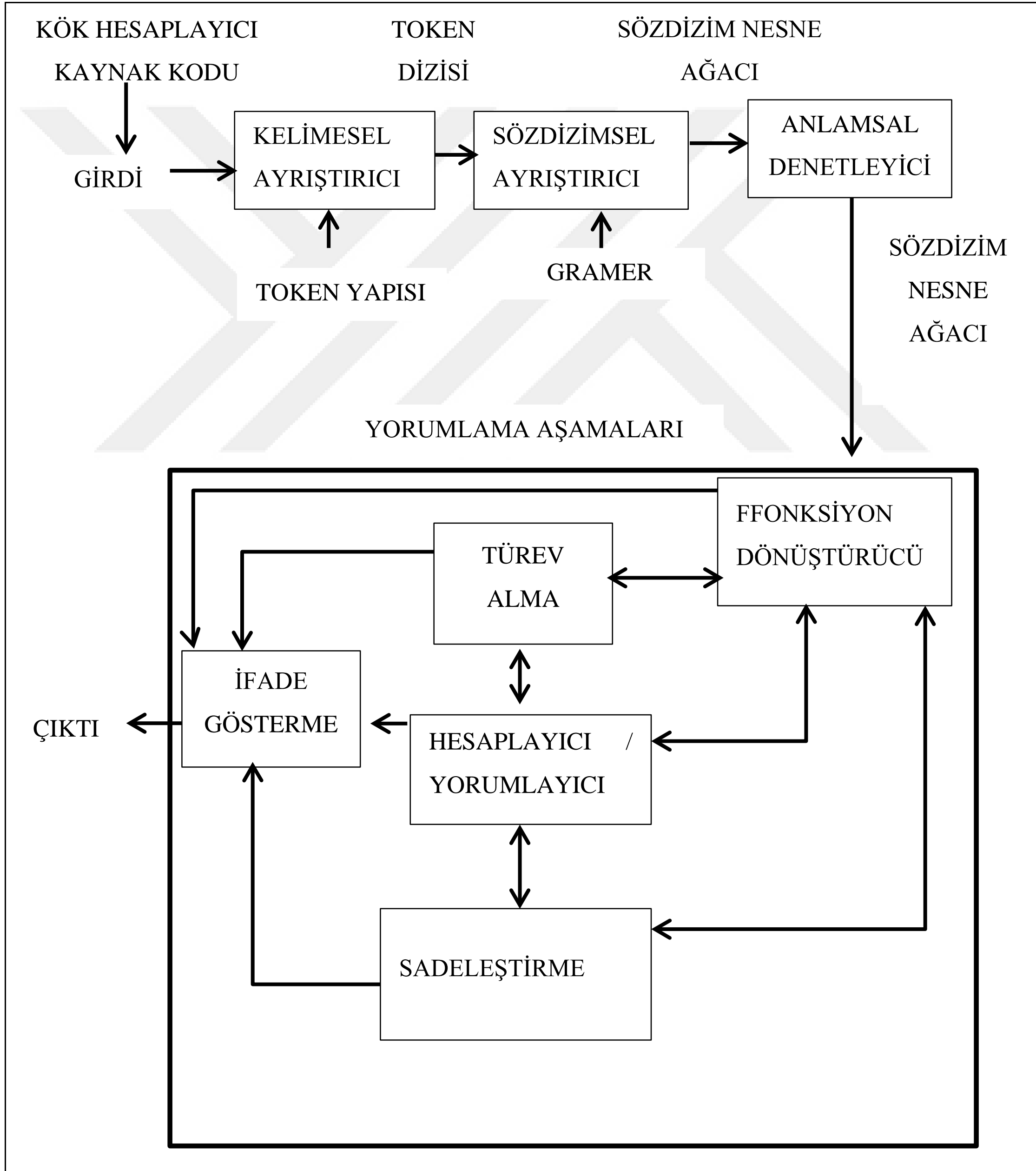
Bu çalışmada biçimsel bir dilde girişi yapılan kaynak verinin sözcüksel analiz, sözdizimsel analiz, anlamsal analiz gibi birçok aşamadan geçirilerek ilgili programlama dili gramerine göre biçimsel ve yapısal kontrolü ile sözdizim ağacına dönüşümü gerçekleştirilir. Kaynak verideki ifadeler bu sözdizim ağacına dayalı olarak geliştirilen bir yorumlayıcı yardımıyla değerlendirilir Bu aşamaların herhangi birinde bir hata ile karşılaşılması durumunda sonraki aşamaya geçilmeden hatanın kaynağı olan ifade rapor edilir. Kodlama süreçlerinde, özellikle analiz aşamalarına ait programlama yükünün önemli bir bölümünü üstlenen otomatik kod üretim araçlarından yararlanılmış olup tez kapsamında geliştirilen uygulama üzerinde bu araçların nasıl kullanıldığı gösterilmiştir. Matematiksel fonksiyonların köklerinin hesaplanması için türev alma ve sadeleştirme gibi birçok ara işleme sahip uygulamada, ifadelerin bileşenlerine ayrılması ve yorumlanması işlemlerinden sonuç versinin üretimine kadar bütün programlama süreçleri ayrıntılı olarak sunulmuştur.

Çalışma kapsamında geliştirilen uygulama belirli bir aralıktaki tek bir kök değerini hesaplamaktadır.

Çalışmada, kaynak verinin analizi için gerekli işlemler Java programlama dilinde otomatik kaynak kod üretebilen JavaCC aracı kullanılarak gerçekleştirilmiştir. Matematiksel ifadelerin sözdizimsel ve anlamsal yapısına uygun bir EBNF (Extended Backus Naur Form) grameri geliştirilmiştir ve soldan türetim gramer yapısı olan LL(k)

gramerine dönüştürülmüştür. Buna ilaveten bir matematiksel ifade içerisinde bulunabilecek işleç ve fonksiyonları temsil etmek üzere sözdizim sınıfları tanımlanmıştır.

Geliştirilen uygulama genel girdi verisini alarak token dizisine dönüştüren bir token üretici, token dizisinden verinin sözdizim yapısını kontrol ederek sözdizim ağacı üreten bir ayrıştırıcı, sözdizim ağacı üzerinde çalışan kök hesaplama, türev alma ve sadeleştirme işlemlerini ile ifade gösterici bileşenlerinden oluşmaktadır. Uygulamanın genel yapısı Şekil 2.1' de gösterilmiştir.



Şekil 2.1. Kök kesaplama uygulamasının genel yapısı

Şekil 2.1'de genel mimarisi gösterilen çalışmada, kök hesaplayıcı kaynak kodunun yazılacağı programlama dilinin, sayısal yöntemlerin belirli bir kök bulma problemine uygulanmasında ihtiyaç duyulan bütün matematiksel işlemleri desteklemesi gerekir. Programlaması yapılacak sayısal yöntemlere ait temel işlemler için gereken programlama dili yapıları, gramer kurallarının belirtildiği JavaCC dosyasında tanımlanmıştır. Gramer kuralları üzerinden ilgili programlama diline uygunluğu denetlenen kaynak veri ile sözdizim sınıflarından türetilmiş nesneleri hiyerarşik yapıda birbirine bağlayan birçok düğüme sahip nesne ağacı oluşturulur. Uygulamada sözdizim ağacı üzerindeki düğümlere visit ve accept metotlarıyla erişilmesini sağlayan Visitor sınıfları tanımlanmıştır.

Sayısal yöntemleri programlamak için bu çalışma kapsamında geliştirilen dil aritmetiksel işlemler, matematiksel operatörler, mutlak değer hesaplama, double, tamsayı ve string türleri, koşul ifadeleri, rekürsif fonksiyon tanımlama özelliklerine sahip olacaktır. Ayrıca geliştirilen dil, kökü hesaplanacak fonksiyonlar üzerinde gerekli fonksiyon dönüşüm işlemlerine imkân veren fonksiyon dönüştürme özelliğini de barındırmaktadır. Programlanan sayısal yöntemlerin bazılarında türev alma işlemine ihtiyaç duyulmaktadır. Bunun için geliştirilen dil ile türev alma işlemi de yapılabilmektedir. Türev alma işlemine özellikle Newton-Raphson, Halley ve Basit İterasyon yöntemlerinde ihtiyaç duyulmaktadır. Türev alma işlemini içeren çözümlemelerde sözdizim ağacını karmaşık yapan ve değerlendirme süresini etkileyen fonksiyon yapıları üzerinde türev işleminden sonra sadeleştirme işlemi uygulanmaktadır. Örneğin; $x^2 - 3$ ifadesinin türevi alındığında $2 * x - 0$ gibi bir sonuç ifadesi üretilecektir. Eğer bu sonuç sadeleştirilmeden geri döndürülürse kök hesaplama işleminde çok miktarda ifade içeren karmaşık yapıda bir sözdizim ağacı geri döndürülecek ve söz konusu fonksiyonun okunurluluğu azalacaktır. Bu nedenle, ilgili denklem $2 * x$ biçimine sadeleştirilerek sözdizim ağacı üzerinde gerekli düzenleme yapılır. Sayısal yöntem iterasyonları sadeleştirme işleminden geçen fonksiyona uygulanarak kök hesaplama işlemine devam edilir ve hesaplanan kök değerleri gösterilir. Ayrıca geliştirilen uygulamada kök hesaplama işlemi gerçekleştirilen fonksiyonların grafiksel gösterimleri de yapılmıştır.

2.2. Geliştirilen Matematiksel Programlama Dilinin Genel Yapısı

Geliştirilen uygulamada programlanacak kaynak kod fonksiyon dizilerinden oluşmaktadır. Aşağıda programlama diline ait EBNF notasyonunda verilen gramer,

fonksiyonlar üzerinde yapılacak değişik hesaplamalarda kullanılır. Bu hesaplamalar; $f(x)$'i belirli bir x değeri için hesaplama veya türevini alma. Matematiksel olarak tanımlanabilen sayısal yöntemleri programlayabilmek için bir gramer yapısına ihtiyaç vardır. Oluşturulacak bu gramer matematiksel yöntemlerin kodlanabildiği basit bir programlama dilini temsil etmektedir. Bu programlama dilinde veri türleri (Int, String ve Double), aritmetik ve mantıksal işleçler ile birkaç temel fonksiyon (abs ve otherwise gibi) desteklenmiştir. Aşağıda genel yapısı EBNF notasyonunda verilen gramer devam eden bölümlerde detaylı bir şekilde açıklanmıştır.

Tablo 2.1. Matematiksel programlama dili için EBNF grameri

```
<function_name>(<parameters>):
                   {<statements>}=<expressions>
```

Yukarıda Tablo 2.1'de genel yapısı verilen programlanacak fonksiyon dizileri için birkaç fonksiyon örneği;

$f\ () = 0$; Belirtilen fonksiyon yapısı programlanacak fonksiyon dizileri içinde en sade olanıdır. Fonksyonun yapsına bakıldığında; parametre ve statement yapılarına sahip değildir. Expression olarak sıfıra sahiptir.

$f\ (x) = x + 1$; Bu fonksiyonda yine sade bir yapıya sahiptir. Fonksyonun yapsına bakıldığında; bir adet parametre ve expression yapılarına sahiptir. Statement yapısına sahip değildir.

$f\ (x) \mid x < 0 = 0 \mid otherwise = x$; Bu fonksiyonda ilk iki fonksiyona göre daha karmaşık bir yapıya sahiptir. Fonksyonun yapsına bakıldığında; bir adet parametre, expression yapılarına ve birden fazla durumu belirten koşul yapılarına sahiptir.

$f\ (x, y) : \{z = y - x\} = z$;

Bu fonksiyonda ilk iki fonksiyona göre daha karmaşık bir yapıya sahiptir. Fonksyonun yapsına bakıldığında; iki adet parametre, expression yapısına ve statement yapılarına sahiptir. Ayrıca belirtilen fonksiyon koşul durumlarınıda alabilmektedir.

Geliştirilen uygulama yukarıda bir kısmı belirtilen fonksiyon yapılarını sayısal yöntemleride kullanarak programlamaktadır. Böylece ilgili fonksiyona ait kök bulma işlemlerini gerçekleştirmektedir.

Uygulama için geliştirilen gramerin JavaCC'ye bildirimi yapılarak, sonraki bölümlerden olan Kelimesel Ayrıştırıcı bölümünde ayrıntılı olarak açıklandığı gibi, kaynak verinin nesne ağacını üretebilen ve değerlendirmesini yapabilen bir yorumlayıcı geliştirilmiştir. Yukarıda Tablo 2.1'de EBNF notasyonunda genel yapısı tanımlanan fonksiyonlar dizisinin kuralları Sözdizimsel Ayrıştırıcı bölümünde detaylı olarak anlatılacağı için bu bölümde ayrıntıya girilmemiştir

Tez çalışmasına ait geliştirilen uygulamada, Bisection Yöntemi, Newton-Raphson Yöntemi, Halley's Yöntemi, Sekant Yöntemi, Regula Falsi Yöntemi ve Basit İterasyon Yöntemi kök hesaplama işlemlerini gerçekleştirmek için programlanmıştır. Geliştirilen uygulamaya gerekli bazı değişiklikler ve eklemeler yapılarak diğer sayısal yöntemlerde entegre edilerek kök hesaplama işlemleri gerçekleştrilebilir. Ayrıca değişik matematiksel işlemlerin çözümünün gerçekleştirilmesine ihtiyaç duyulan birçok alanda gelişirilen bu uygulama dil özellikleri, gramer yapısının zenginleştirilmesi ile fonksiyon yapısı geliştirilerek kullanılabilir.

Bu çalışma için geliştirilen ve genel yapısı Şekil 2.1'de gösterilen uygulamanın içerdiği bileşenler uygulamada işlem sıralarına ve belirtilen bileşen isimlerine uygun bir şekilde ayrıntılı olarak açıklanmıştır.

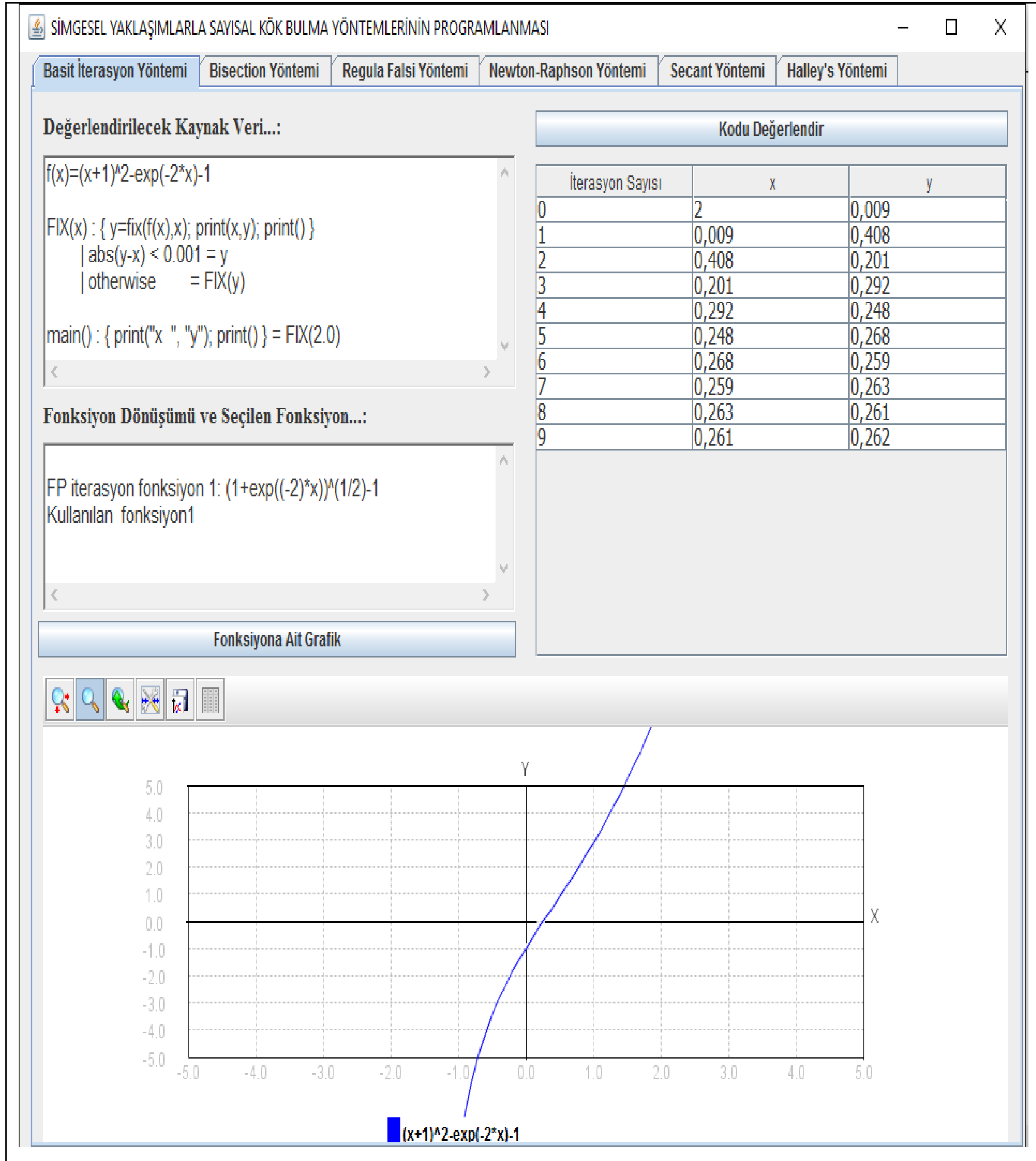
2.3. Geliştirilen Uygulamaya Ait Arayüz ve Çalışma Aşamaları

Şekil 2.2'de gösterilen form arayüzüne sahip uygulamanın çalışma aşamaları şu şekilde özetlenebilir;

- İlgili sayısal yönteme ait kaynak kod arayüzden girilir daha sonra uygulama bu yapıyı alarak hazırlanan gramer yapısına uygun olarak bir nesne ağacı oluşturur.
- Bu aşamadan sonra kök hesaplama işlemi için uygulanacak sayısal yöntemlerde bazen türev alma işlemine ihtiyaç duyulabilmektedir. Bunun için uygulamadaki türev sınıfına ait metotlar yardımıyla nesne ağacının düğümlerinde gezinilerek girilen ifadenin türevi alınır ve böylece üretilen nesne ağacından yeni bir nesne ağacı türetilmiş olur. Ayrıca kullanılan sayısal yöntem için ardışık türev alma işlemleri gerçekleştirilebilir.
- Değerlendirme işlemine tabi tutulan girdi verisi üzerinde türev alma işlemi gerçekleştirilmişse türev alma işleminden sonra elde edilecek kök değerlerinde hatalı sonuçları engellemek için oluşturulan nesne ağacı sadeleştirme işlemine tabi tutularak başka bir nesne ağacı elde edilir. Eğer başka işleme ihtiyaç yoksa oluşturulan yeni nesne

ağacı üzerinden ilgili fonksiyona ait kök değerleri hesaplanır. Daha sonra hesaplanan kök değerleri arayüzde bir tabloya yazdırılarak kök değerlerinin gösterim işlemi gerçekleştirilmiş olur.

- Kök hesaplama işlemi için programlanan sayısal yöntemlerde Basit İterasyon Yöntem'inde olduğu gibi fonksiyon dönüşüm işlemine ihtiyaç duyulabilir. Bu durumda geliştirilen uygulama ilgili fonksiyon dönüşüm işlemlerini gerçekleştirerek yeni bir nesne ağacı oluşturulur. Yine burada türev alma ve sadeleştirme işlemlerine ihtiyaç duyulacaktır, türev alma ve sadeleştirme işlemlerinden kök hesaplama işlemleri gerçeklenerek arayüzde gösterilir.
- Programlanan sayısal yöntem için fonksiyon dönüşümü, türev alma ve sadeleştirme işlemlerine ihtiyaç duyulmaz ise, tanımlı olan matematiksel işlemler kullanılarak kök hesaplama işlemi gerçekleştirilir. Şekil 2.2'deki formda gösterilen kriterlere göre kök değerleri hesaplanarak tablo şeklinde listelenir ve böylece kök hesaplama işlemi tamamlanır.
- Şekil 2.2'de uygulama için geliştirilen form arayüzünde görüldüğü gibi ilgili matematiksel fonksiyonun kök değerlerine ait grafiği de geliştirilen uygulama yardımıyla elde edebiliriz. Bu bölümde bu çalışma için geliştirilen uygulamanın genel tanıtımı yapılmıştır. Sonraki bölümlerde ise uygulama örneklerle detaylı bir şekilde anlatılmıştır.



Şekil 2.2. Uygulama arayüzü

2.4. Gramer Tabanlı Değerlendirme Aşamaları

Biçimsel dillerdeki gramer yapıları karakter dizileri veya kelimeler olarak adlandırılan kurallar topluluğu olarak ifade edilir. Bu dillerin yapılarında kullanılan içerikten bağımsız gramerler (CFG), düzenli gramerler, analitik gramerler gibi birçok gramer türü vardır.

Derleme veya yorumlama işlemleri için gramer yapısı oluşturulduktan sonra, bu gramerin yapısına uygun;

• Kelimesel Analiz,

• Sözdizimsel Analiz,

• Somut Sözdizim Ağacı Oluşturma,

• Anlamsal Analiz,

• Yorumlama işlemleri gerçekleştirilir.

Bu bölümde matematiksel ifadelerin sözdizim yapısına uygun bir CFG grameri tanımlanarak işleç anlamlarını içerecek biçimde LL(1) gramerine dönüşümü gösterilmiştir.

Uygulamada kullanılan gramer yapısından ayrıntılı olarak konunun devamında bahsedilmiştir.

2.4.1. Kelimesel Ayrıştırıcı

Girdi verisi olarak alınan, kök hesaplama işlemine tabi tutulacak kaynak kodun değerlendirme sürecinin ilk basamağı kelimesel analiz bölümüdür. Bu bölümde kaynak kod, uygulumada kullanılan programlama dili olan Java programlama dilinin sözcük yapısına uygun ve token olarak isimlendirilen parçalara ayrılır. Oluşturulan bu token parçaları bir ya da birkaç karakterden oluşabilmektedir. Daha sonra bu token parçaları düzenli ifadeler yardımıyla tanımlanmaktadır. Geliştirilen uygulamada kullanılan ve derleyici derleyici olarak isimlendirilen JavaCC aracı da token tanımlamalarında düzenli ifadeleri kullanır. Örneğin, tamsayı sayıları temsilen NUM reel sayıları temsilen DNUM ve değişken yapıları için de ID isimlerinde birer token tanımlama işlemleri Tablo 2.2'de gösterilmiştir. Ayrıca matematiksel operatörler, trigonometrik ifadeler, mantıksal operatörler ve parantez işlemi ile ilgili yapılara ait token tanımlama işlemlerinin bir kısmı yine Tablo 2.2’de gösterilmiştir. Özel karakterlerin de (* ve + gibi) kullanılabildiği düzenli ifadeler ile bir token sonsuz sayıda sözcük içerebilen bir seti temsil edebilmektedir. Ayrıca, bir veri türünün farklı örneklerini kapsayan birçok farklı karakter dizisini (ya da kelimeyi) temsil edebilmek için aynı token kullanılabilmektedir. Örneğin, Geliştirilen uygulmaya ait token tanımlamalarında gösterildiği gibi bütün tam sayılar ("5" ve "50" gibi) NUM veya bütün metinsel karakterleri (“x” ve “a”) ID temsil edebilen tokenlar ile ifade edilebilmektedir.

Tablo 2.2. Uygulamaya ait token tanımlaması

```
TOKEN: {
<IF: "if">              | <THEN: "then">     | <ELSE: "else">
| <PRINT: "print">      | <CPRINT: "cprint"> | <PLUS: "+">
| <MINUS: "-">          | <TIMES: "*">       | <DIVIDE: "/">
| <OR: "||">            | <NOT: "!">         | <EQ: "==">
| <RCURLY: "}">         | <LPAREN: "(">      | <RPAREN: ")">
| <FALSE: "false">      | <TRUE: "true">     | <ABS: "abs">
| <SIN: "sin" >         | <COS: "cos">       | <DRV: "drv">
| <FIX: "fix">          | <OTHER: "otherwise">
| <ID: (["a"-"z","A"-"Z"])(["a"-"z","A"-"Z","0"-"9"])*>
| <NUM: (["0"-"9"])+>| <DNUM: (["0"-"9"])+"."(["0"-"9"])+>}
     SKIP: { " " | "\t" | "\r" | "\n" }
```

JavaCC notasyonuna göre Tablo 2.2'de gösterilen token tanımlamalarında token ismi ve temsil edilen sözcük setine ait düzenli ifade birlikte verilir. Bu aşamada düzenli ifadelerin kullanılmasının nedeni işleme tabi tutulan karakter dizisi içerisinde kabul edilebilir bir token yapısı bulabilmek için bazı durumlarda geriye dönük arama işlemi yapılmasına ihtiyaç duyulabilmesidir. Bununla birlikte tanımlanan düzenli ifadeler, ilgili programlama diline ait tüm ifadeleri içermelidir.

Genel olarak, her bir işleç ya da anahtar sözcük farklı bir token sınıfı ile temsil edilir. Tam sayılar, virgüllü sayılar, sözcük setleri ve diğer işlemler için ayrı token sınıfları tanımlanır.

Matematiksel ifadelerden oluşan kaynak veri metin formatında sisteme girilmiştir; örneğin, $e^{-2x} - x^2 - 3x$ ifadesinin sisteme girişi $\exp(-2*x) - x\hat{}2 - 3*x$ biçiminde yapılır; ifade biçimleri dilin sözdizimi ile ilgilidir ve takip eden bölümde açıklanmıştır. Bu örnek ifade, Tablo 2.2'deki tanımlamalara göre Tablo 2.3'te gösterilen token dizisine ayrıştırma işlemlerine tabi tutulur.

Tablo 2.3. Token dizisi

```
EXP, LPR, MINUS, NUM, TIMES, ID,
 RPR, MINUS, ID, POWER, NUM, MINUS, NUM, TIMES, ID
```

Tablo 2.2'de tanımlanan token yapıları JavaCC yapısı kullanılarak kontrol işlemlerini gerçekleştirecek Java koduna dönüştürülmüştür. Dönüşüm işlemi sonucunda oluşturulan kod yapısı kelimesel analiz işlemlerini gerçekleştirmek için kullanılmıştır.

Token tanımlaması yapılmayan sözcükler, JavaCC ile üretilen kod tarafından tespit edilerek uygun bir hata mesajı gösterilir. Hata mesajı gönderildikten sonra analiz işlemi sonlandırılacaktır.

2.4.2. Sözdizimsel Ayrıştırıcı

Genel tanım olarak bir programlama dilinde (language) önceden tanımlanmış olan öğelerin (kelime, işlem, sembol ya da değerlerinin) uygun ve anlamlı bir dizilim meydana getirmesi ile ilgilenen aşamadır.

Kök hesaplama işlemine tabi tutulacak girdi verisinin değerlendirme süereçlerinin ikinci bölümünü oluşturan bu kısımda kaynak veri biçimi tanımlanmaktadır. Ayrıca yine bu aşamada token dizisi kullanılarak sözdizim analiz işlemleri gerçekleştirilmektedir. Sözdizimsel işlemleri gerçekleştiren ayrıştırıcının iki ana görevi vardır. Bunlar; kaynak verinin biçimsel denetimi yapmak ve nesne ağacının üretimi işlemlerini gerçekleştirmektir. Biçimsel denetim için verinin sözdizimini tanımlayan BNF ve CFG gibi gramer türleri kullanılır ve bu tanımlamalar içerisine nesne ağacını üreten ifadeler eklenir.

JavaCC aracı veri biçimlerini tanımlamak ve nesne ağacını oluşturmak için kullanımı kolay bildirimsel yapılar içerir. Kaynak veriyi biçimsel olarak temsil eden bir gramerin her bir kuralına yönelik, kuralın içerdiği token ve non-terminalleri kapsayacak şekilde birer metot ve sözdizim sınıfı (syntax class) tanımlanmıştır. Sözdizimsel analiz aşaması için gramer yapısı belirlendikten sonra oluşturulan bu gramer yapısının LL(k)'ya uygun hale getirilmesi gerekmektedir. Bunun nedeni JavaCC, LL(k) algoritması ile çalışmaktadır. Bundan dolayı ilgili gramere yapısının, LL(k) algoritmasına uygun olabilmesi için, soldan yinelemeli durumları elimine edilerek sol faktörleme (left factoring) işlemi gerçekleştirilmiştir. Tüm bu işlemlerden sonra ilgili gramer yapısı JavaCC ortamına aktarılmış ve geliştirilen uygulamaya ait sözdizimsel analiz işlemlerini gerçekleştirebilecek Java kodu üretilmiştir.

Bu bölümün devamında uygulama için geliştirilen gramer yapısı ve bu gramere ait Java programlama dilinde otomatik kod üretebilen, JavaCC kod yapısı ayrıntılı olarak açıklanmıştır.

Tablo 2.4. Fonksiyon dizisi, arguman ve parametre tanımı

```
<function_list>       ->     <function_definition>(<function_list>)?
<function_definition>-> <function>(<statement_list>)?<equation>
<statement_list>       -> ":" "{" (<statement>)? "}"
<function>             -> id "(" (<pattern_list>)? ")"
<pattern>              -> id ("+" num)? | num
```

Uygulama için gerekli olan gramer yapısının bir kısmı Tablo 2.4'de gösterilmiştir. Bu gramer yapısına göre sırasıyla kök hesaplaması yapılacak ve dışarıdan girdi verisi olarak alınan fonksiyonlar için, `<function_list>` kuralı ile fonksiyonlar dizisi, `<function_definition>` kuralı ile fonksiyon tanımlama, `<statement_list>` kuralı ile fonksiyon statement listesi, `<function>` kuralı ile fonksiyon yapısı, `<pattern_list>` kuralı ile model listesi ve `<pattern>` kuralı ile de formal parametre biçimlerinin tanımı yapılmıştır. `<function>` kuralına ait tanımlama; değişken, sol parantez, enfazla bir arguman listesi ve sağ parantez yapısını belirten durumlar gramer yapısında tanımlanmıştır. `<pattern>` kuralına ait tanımlamada görüldüğü gibi; değişken, sayı veya değişken + sayı şeklinde ki formal parametre biçimlerini içerebilmektedir. Bununla birlikte nesne ağacının oluşturulma biçimi ise (hiyerarşik yapısı), kaynak verinin oluşturulmasında ve değerlendirilmesinde kullanılan gramer kurallarının uygulanma (çağrılma) sırasına bağlıdır.

Tablo 2.4'de EBNF notasyonunda belirtilen gramerin JavaCC notasyonu ile tanımlanma işlemlerinin bir kısmı Tablo 2.5'de gösterilmiştir; S(FnDef[] eql, int i), R() ve L() metotları `<function_list>` kuralına karşılık olarak tanımlanmış olup bu kuralların uygulanması ile ilgilenir.

Tablo 2.5. Tablo 2.4. için JavaCC gramer tanımlaması

```
void S(FnDef[] eql, int i) :{ FnDef s; }
          {s=S2()(S(eql,i+1) {
          if (i>=9) { eql[i] = s; }}
Exp R() :{Token t; Exp[]
          el= new Exp[10];boolean b=false;}
          {t=<ID> <LPAREN> (R2(el,0) { b = true; })?
```

Tablo 2.5’in devamı

```
         <RPAREN> { return new Fn(t.image, b ? el : null); }}
 Exp L() :{ Token t, t2; Exp e; }
          {t=<ID>(<PLUS> t2=<NUM>
          {return new Plus(new Var(t.image), new
          Num(Integer.parseInt(t2.image))); })?
          {return new
          Var(t.image);}
          |t=<NUM>{returnnew
Num(Integer.parseInt(t.image));}}
```

Tablo 2.6. Fonksiyonun sağ tarafına ait gramer tanımı

```
<equation>       -> "=" <expression> | <equation_list>
<equation_list> -> "|" <bool_expression> "=" <expression>
                        (<equation_list>  )?
```

Geliştirilen uygulama için gerekli olan gramer yapısının diğer bir kısmı ise Tablo 2.6.’da gösterilmiştir. Tablo 2.6’daki ilgili gramer yapısına göre sırasıyla `<equation>` kuralı ile fonksiyonun sağ tarafına ait durumlar, `<equation_list>` kuralı ile “|” işleci ile başlayan fonksiyon denklemleri için gramer yapısında tanımlama yapılır. Tanımlanan bu gramer yapısının uygulamada kullanılabilmesi için JavaCC dosyasında tanımlanması gerekmektedir. Tablo 2.6'da EBNF notasyonunda verilen gramerin JavaCC notasyonu ile tanımlamasının bir kısmı Tablo 2.7'de gösterilmiştir; G(BExp[] bl, Exp[] el, int i) ve G2(BExp[] bl, Exp[] el, int i) metotları `<equation>` ve `<equation_list>` kuralına karşılık olarak tanımlanmıştır ve bu kuralların uygulanması ile ilgilenir. Yukarıda belirtildiği gibi gramer yapısına ait nesne ağacının oluşturulması yani hiyerarşik yapısının belirlenmesi, kaynak verinin oluşturulmasında kullanılan gramer kurallarının uygulanma (çağrılma) sırasına bağlıdır. Bir sözdizim ağacı (ya da genel olarak nesne ağacı) hiyerarşik yapıda birbirine bağlanmış birçok düğümden oluşmaktadır. Nesne ağacına üzerinde bulunan bu düğümlerden her bir düğüm sözdizim sınıflarından türetilmiş ve bir işlemi ya da veriyi temsil eden bir nesne içermektedir. Bu çalışma için geliştirilen bu uygulamada olduğu gibi sözdizim ağaçlarının bildirimi genellikle bir üst sınıf türü yardımıyla yapılır.

Tablo 2.5’de gösterilen metotlar her biri Exp adı verilen bir üst sınıftan miras alınarak oluşturulan sınıflarla temsil edilmiştir.

Tablo 2.7. Tablo 2.6. için JavaCC gramer tanımlaması

```
void G(BExp[] bl, Exp[] el, int i) :
      { Exp e; BExp b; }
      {<AEQ> e=E()
      { bl[i] = new BNum(true);
       el[i] = e; }| G2(bl,el,i)}
void G2(BExp[] bl, Exp[] el, int i) :
      { Exp e; BExp b; }
      { <GUARD> b=B() <AEQ> e=E()
      ( G2(bl,el,i+1) { if (i>=9)
      { bl[i] = b; el[i] = e; }…}
```

Tablo 2.8. “:” ile tanımlama bloğu ve “;” ile ardışık tanımlamalar

```
<statement_list>      -> ":" "{" ( <statement> )? "}"
<statement>           -> <expression_definition>
                         ( ";" <expression_definition> )*
<expression_definition> -> "print""("(<expression_list>)?")"
<expression_definition> -> "cprint""("B"?"EL(":"EL)?")"
<expression_list>       -> <expression>( "," <expression>)*
```

Tablo 2.8’de `<statement_list>` kuralıyla bölümde “:” ile başlayan tanımlama bloğu, `<statement>` “;” ile ayrılmış ardışık tanımlamalar, `<expression_definition>` kuralı ile ifade tanımlamaları ve `<expression_list>` kuralı ilede ifade dizilerine ait tanımlamalar gramer yapısına eklenmiştir. Tablo 2.8'deki gramerin JavaCC notasyonu ile tanımlamasının bir kısmı Tablo 2.9'da gösterilmiştir; ilgili tabloda gösterilen D(), D2() ve D3() metotları `<statement_list>`, `<statement>`, `<expression_definition>` ve `<expression_list>` kurallarına karşılık olarak tanımlanmıştır ve belirtilen bu kuralların uygulamada yukarıda işleleri verilen istenilen işlemleri gerçekleştrirlmesiyle ilgilenir.

Tablo 2.9. Tablo 2.8. için JavaCC gramer tanımlaması

```
Stm D() :{ Stm s=null; }
           {<COLON> <LCURLY> ( s=D2() )? <RCURLY>{ return s; }
              …}
Stm D2() :{ Stm a, b; }
              {a=D3() ( <SEMI> b=D3()
              { a = new LStm(a,b); } )*
              {return a;}
              …}
Stm D3() :{ Token t; Exp e; BExp be;
           {t=<ID> <AEQ> e=E() { return new AStm(t.image,e); }
             |<PRINT> <LPAREN> ( EL(el,0)
             { b = true; } )?<RPAREN>{
          <RPAREN> {return new CStm(be, el, b ? el2 : null); }
             …}
```

Tablo 2.10. Matematiksel ifadeler için gramer

```
<expression> -> ("+" | "-")? <term> ( ("+" | "-") <term> )*
      <term> -> <pow> ( ("*" | "/") <pow> )*
      <pow> -> <factor> ( "^" <pow> )?
```

Matematiksel ifadeler için Tablo 2.10'de gösterilen bir gramer tanımlanmıştır. Bu gramerde <expression>, <term> ve <pow> kuralları ve alternatifleri nesne ağacında, her biri Exp adı verilen bir üst sınıftan miras alınarak oluşturulan Plus, Minus, Times, Divide, Power gibi sınıflarla temsil edilmiştir. Örneğin, <expression> kuralının iki alternatifi olduğundan, nesne ağacı Plus ve Minus sınıflarından türetilen nesneler içerebilecektir. Oluşturulan nesne ağacına ait her bir düğüm sözdizim sınıflarından türetilmiş olup ve bir işlemi veya bir veriyi temsil eden bir nesne içerir.

Tablo 2.11. Tablo 2.10. için JavaCC gramer tanımlaması

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

Tablo 2.11’in devamı

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

Tablo 2.10'daki gramerin JavaCC notasyonu ile tanımlamasının bir kısmı Tablo 2.11'de gösterilmiştir; expr() ve element() metotları sırasıyla <expression> ve <term> kurallarına karşılık olarak tanımlanmıştır ve bu kuralların uygulanması ile ilgilenir. Ayrıca, Tablo 2.11'de görüldüğü gibi, kural uygulamaları içerisinden biçimi doğrulanmış kaynak veri bileşenini nesne ağacına taşımak için Java ifadeleri eklenmiştir. Sözdizim sınıflarından yararlanan bu ifadelerle nesne ağacına hiyerarşik bir yapı kazandırılır.

Tablo 2.12. Fonksiyona parametre dizisi verilmesi, türev alma işlemi ve fonksiyonun dönüştürülmesi işlemlerine ait gramer

```
<function_list>->id("("(<function>)?")")?
                |num|dnum|str|"("<expression>")"
                | "abs" "(" <expression> ")"
                | <türev_definition>
<function> -> <expression> ("," <function>)?
<türev_definition> -> "D"("^" num)?"("id")""(" E")"
                          | "fix" "(" E "," id ")"
```

Tablo 2.12’de <function_list>, <function> ve <türev_definition> kuralıyla çağrılan fonksiyona parametre dizisinin verilmesi, türev alma işlemi ve fonksiyonun dönüştürülme işlemlerine ait ifadeler gramer yapısına eklenir. Tablo 2.12'deki gramerin JavaCC notasyonu ile tanımlamasının bir kısmı Tablo 2.13'de gösterilmiştir; F()

ve F2(Exp[] el, int i) metotları `<function_list>` kuralına karşılık olarak tanımlanmış olup bu kuralların uygulanması ile ilgilenir.

Tablo 2.13. Tablo 2.12. için JavaCC gramer tanımlaması

```
Exp F() :
        { Token t; Expe;Exp[] el= new Exp[10];
         boolean b = false;
         int n=1;}
         {t=<ID> (<LPAREN> (F2(el,0) { b = true; })? <RPAREN>
         { return new Fn(t.image, b ? el : null); })?
         … }
         | t=<NUM> { return new Num(Integer.parseInt(t.image)); }
         | t=<DNUM> {return new DNum(Double.parseDouble(t.image));
             …}
         | <FIX> <LPAREN> e=E() <COMMA> t=<ID> <RPAREN>
         { return new FixExp(e, t.image); }}
void F2(Exp[] el, int i) :{ Exp e; }
            {e=E()(<COMMA>F2(el,i+1){if(i>=9)…}
```

Tablo 2.14. Değerlendirme işleminde koşul ifadelerini kullanımı

```
<bool_expression>  ->  <and_expression> ( "||" <and_expression> )*
<and_expression>   ->  <not_expression>( "&&" <not_expression>)*
<not_expression>   ->  "!" "(" <bool_expression> ")" |
<comparison_expression> | "otherwise"
<comparison_expression> ->  <expression> ("==" | "!=" | "<" | "<="
                            | ">=" | ">") <expression>
```

Tablo 2.14’te `<bool_expression>` kuralı ile true veya false değeri tanımlama bloğu, `<and_expression>` kuralı ile and operatörü tanımlaması, `<not_expression>` kuralı ile eşitlik durum kontrol tanımlaması ve `<comparison_expression>` kuralı ilede ifadelerin karşılaştırılma işlemlerine ait tanımlamalar gramer yapısına eklenmiştir. Tablo 2.14'deki gramerin JavaCC notasyonu ile tanımlamasının bir kısmı Tablo 2.15'de gösterilmiştir; ilgili tabloda gösterilen B(), A() ve N() ve metotları `<bool_expression>`, `<and_expression>`,

<not_expression> ve <comparison_expression> kurallarına karşılık olarak tanımlanmıştır ve belirtilen bu kuralların uygulamada yukarıda işleleri verilen işlemleri gerçekleştrirlmesiyle ilgilenir.

Tablo 2.15. Tablo 2.14. için JavaCC gramer tanımlaması

```
BExp B() :{ BExp be1, be2; }
          {be1=A() ( <OR> be2=A()
          {be1 = new OrExp(be1, be2); } )*
          {return be1;}}
BExp A() :{ BExp be1, be2; }
          {be1=N()(<AND> be2=N()
          {be1 =new AndExp(be1, be2);})*{return be1;}}
BExp N() :{ BExp be; }
         {<NOT> <LPAREN> be=B() <RPAREN>
         { return new NotExp(be); }
         | be=N2() { return be; }
         | <OTHER> { return new BNum(true); }}
```

Tablo 2.1'de gösterildiği gibi uygulamaya ait sözdizimsel ayrıştırıcı aşamasından sonra uygulamanın bütün aşamalarında sözdizim nesne ağacı derlenip tekrar tekrar kullanılmaktadır. Kaynak verinin sözdizim analizi sonucunda üretilecek nesne ağacı düğümleri için sözdizim sınıfları kullanılır. Bir sözdizim sınıfı bir ya da birkaç gramer kuralını temsil etmek üzere tanımlanabilir. Genel olarak, bir sözdizim sınıfının ismi, ilgili kuralın temsil ettiği işlemden (işleç ya da işlev) türetilir ve kuralın içerdiği nonteminaller için sözdizim sınıfına birer alan verisi eklenir.

2.4.3. Anlama Yönelik Analiz

Bu aşamada sözdizim aşamasında oluşturulan parse ağacı, semantik analiz işlemine tabi tutulur. Ağacın yapraklarında bulunan token yapıları, bu aşamada derleyici için bir anlam ifade etmeye başlar. Örneğin; derleyici tarafından integer olan bir tokene integer, değişken olan tokene ise değişken olarak anlam verilir. Bunun sonucunda anlam olarak belli hatalar oluşabilir. Bu aşamada oluşabilecek hatalara semantik hatalar denir. Örneğin; integer olan bir değişken, string olan bir değişkene direkt olarak eşitlenemez çünkü

bunların token tipleri farklıdır. Bu aşamada, kaynak koddaki anlamsal hatalar kontrol edilir ve kod üretimi için veri tipi bilgileri belirlenir. Sembol tablosunun oluşturulması ve tip kontrolünün gerçekleştirilmesi anlamsal analizin en önemli kısımlarıdır. Anlamsal bilgi, bağlamdan bağımsız dil ile gösterilmez. Söz dizimsel analizde kullanılan CFG, anlamsal kurallar ile birleştirilir.

Sembol tablosunun oluşturulması: Bu aşamada tanımlayıcılar, tanımlayıcı tipleri ve tanımlayıcı kapsamları bilgisi sembol tablosuna yerleştirilir. Kaynak kodda tip bildirimleri, değişken ve fonksiyon tanımlamaları yapıldığında bu tanımlayıcı bilgileri sembol tablosunda oluşturulur. Kodun sonraki kısımlarında kullanımına rastlanan tanımlayıcıların gerekli bilgileri sembol tablosundan bakılıp, karar verilir. Program içindeki her bir yerel değişkenin geçerli olduğu bir alan vardır. Örneğin fonksiyon içinde tanımlanmış bir değişkenin geçerlilik alanı fonksiyonun tanımlandığı yerdir. Tanımlayıcıların tipleri, geçerlilik alanları gibi bilgilerin tutulduğu sembol tablosuna çevre adı verilir ve → işareti ile gösterilir.

Tip kontrolünün gerçekleştirilmesi: Kod içinde kullanılan tanımlayıcıların tip bilgileri sembol tablosuna eklendikten sonra, bu tanımlayıcıların uygun şekilde kullanılıp kullanılmadığının tespit edilmesi işlemi tip kontrolü olarak adlandırılır. Uygulamada tip kontrolünün gerçekleştirilebilmesi için sembol tablosu program içerisinde tanımlanmış bazı tür bilgilerini içermelidir.

• Değişken isimleri ve formal parametreler tip bilgileri ile ilişkilendirilmeli,

• Metot isimleri, bu metodun aldığı parametreler, sonuç tipi ve yerel değişkenleri ile ilişkilendirilmeli,

• Eğer nesneye dayalı bir dil ise sınıflar da, içinde tanımlanmış değişken ve metotlarla ilişkilendirilmelidir.

2.4.3.1. Kaynak Kod Tip Kontrolü

Bu aşama oluşturulan nesne ağacındaki düğümlere ait ifadeler TypeVisitor sınıfı yardımı ile ifadelerin tiplerine bağlı olarak doğru biçimde kullanılıp kullanılmadığı kontrol edilir. Uygulamada tip kontrol işlemi fonksiyon tanımlamalarının sağ tarafı (Right- hand Side) ile gerçekleştirilmiştir. Bu bölümde yorumlanacak kaynak veriye ait ifadelerin beklenen tip değerleri ile sembol tablosundaki gerçek tip bilgileri karşılaştırılır. Hatalı durumlarda tip kontrol işlemi durdurulurak hata bildirimi yapılır. Tip kontrolünü

gerçekleştirmek için TypeVisitor sınıfı oluşturulmuştur. Tablo 2.16'da toplama işlemindeki tip kontrolü ve gerçeklenen kontrole göre geriye döndürülen değer yapısına ait kod bloğu verilmiştir.

Tablo 2.16. Toplama işlemine ait tip kontrolü

```
public Object visit(Plus e) {
       Object a = e.a.accept(this); Object b = e.b.accept(this);
       if (a == null || b == null)return null;
       if (a instanceof String || b instanceof String)
            return new String("");
       else if (a instanceof Double || b instanceof Double)
            return new Double(0); else return new Integer(0);}
```

Tablo 2.16'da verilen kod bloğu incelendiğinde toplama işlemi için iki adet object türünde değişken tanımlanmıştır. Bu değişkenlerin alması muhtemel bütün değer türleri göz önünde bulundurularak kontrol işlemi gerçekleştirilmiştir. İlk olarak değişkenlerin herhangi biri null değeri alması durumu kontrol edilmiş ve bu kontrol sonucunda şart sağlanıyorsa geriye null değeri döndürülmüştür. İkinci koşul ise değişkenlerin String tipte olup olmama durumu kontrol edilmiş olup şart sağlanıyorsa geriye string bir yapı döndürülmüştür. Üçüncü koşulda ise ilgili değişkenler üzerinde Double veri tipi kontrolü yapılmış ve yine geriye bir değer döndürülmüştür. Son olarak ise Integer tipte bir değer geri döndürülmüştür. Döndürülen bu değerlere göre toplama işlemini sonucu ilgili yerde gösterilecektir.

Tablo 2.17. Çıkarma işlemine ait tip kontrolü

```
public Object visit(Minus e) {
          Object a = e.a.accept(this);
          Object b = e.b.accept(this);
          if (a == null || b == null)return null;
          if (a instanceof String || b instanceof String) {
          System.out.println("Typeerror:"+new
         PrintVisitor().visit(e));
         error = true; return null;}…}
```

Tablo 2.17’de verilen kod bloğunda ifade edildiği gibi çıkarma işlemi için yine iki adet object türünde değişken tanımlanmıştır. Bu değişkenlerin alması muhtemel bütün değişken tipleri göz önünde bulundurularak tip kontrol işlemi gerçekleştirilmiştir. İlk olarak değişkenlerin herhangi biri null değeri alması durumu kontrol edilmiş ve bu kontrol sonucunda şart sağlanıyorsa geriye null değeri döndürülmüştür. İkinci koşul ise değişkenlerin String tipte olup olmama durumu kontrol edilmiştir. Bu durumda tanımlanan değişkenlerden herhangi biri String tipte bir değişken ise TypeVisitor sınıfı hata mesajı vericektir. Çünkü sayısal bir değer ile metinsel bir ifade çıkarma işleminde kullanılamaz. Daha sonra error değişkenine true değeri atanır ve geriye null değeri döndürülür. Bir diğer durumda ise tanımlanan değişkenler üzerinde Double veri tipi kontrolü yapılmış, kontrol işlemine göre geriye bir değer döndürülmüştür. Son olarak ise Integer tipte bir değer geri döndürülmüştür. Döndürülen bu değerlere göre çıkarma işlemi gerçekleştirilir veya hatalı bir işlem olması durumunda hata mesajı verilerek işlem sonlandırılır. Bu şekilde AST sınıfında tanımlanan bütün visit metodları için kontrol işlemi gerçekleştirilir. Bölümün başında belirtildiği gibi bu kontrol işlemleri oluşturulan nesne ağacı yapısındaki düğümler üzerinden gerçeklenmektedir.

2.4.4. LL(k) Gramerine Ait Dönüşüm İşlemleri

JavaCC tarafından üretilen ayrıştırıcı üreteçleri ile analiz edilen matematiksel ifadeleri LL(k) karakterli bir gramer yapısına uygun bir şekilde dönüştürülmelidir. Genel olarak, LL(k) gramer yapılarında girdi verilerinden en fazla k adet terminal bileşeni okunarak kural seçim işlemi gerçekleştirilebilir (k>0). Bu işlem yöntemi izlenirken ilgili LL(1) gramer yapısı aşağıda maddeler şeklinde ifade edilen özellikleri barındırmış olacaktır.

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

İlgili gramer yapısılarına yeni kurallar ekleme yoluyla basit bir şekilde LL(1) gramerine dönüşüm işlemi gerçekleştirilebilmektedir. Tablo 2.18’de gösterilen $ sembolü girdi verisinin sonlandığını ifade etmektedir. Ayrıca Tablo 2.18’de gösterilen ilgili gramer

yapısı için oluşturulacak bir ayrıştırıcının sahip olması gereken metotların isimlerini de içermektedir.

Tablo 2.18. Matematiksel ifadeler için bir LL(1) grameri

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

2.4.5. Yorumlama Aşamaları

Girdi verisi olrak alınan kaynak kod bu aşamaya kadar gelmesinin anlamı ilgili kodun yapısal olarak hatasız olduğudur. Herhangi bir hatası bulunmayan kaynak verinin son işlem aşaması burada gerçekleştirilir. Bu aşamada Şekil 2.1’de gösterilen Kök Hesaplayıcı bileşenin alt bileşenleri olan fonksiyon dönüştürücü, türev alıcı, sadeleştirici, sadeleştirici bileşenleri tarafından kök hesaplama işlemi gerçekleştirilecek kaynak kod yorumlanır ve sonuç üretilir.

2.4.5.1. Kök Hesaplayıcı(Örnekleri ilgili yerlerde anlat örnek basit iterasyon)

Kök hesaplayıcı bileşeninde diğer bileşenlerin entegre dilerek uygulamada çalıştırıldığı bu aşamada ilgili fonksiyona ait kök değerlerinin hesaplama işlemleri gerçekleştirilir. Şekil 2.1’de gösterildiği gibi, türev alma, fonksiyon dönüştürme, sadeleştirme, ifade gösterme elemanlarıyla bağlantılı bir şekilde ilgili işlemleri

gerçekleştirir. Tablo 2.19'de kök bulma uygulamasına ait java kodunun bir kısmı verilmiştir

Tablo 2.19. FixedPointVisitor sınıfı (FNEvalVisitor.java)

```
public class FNEvalVisitor implements Visitor
{
       public Object visit(DrvExp e){
         DeriveVisitor dvisitor=new DeriveVisitor(fList,t,e.id);
         Exp de = (Exp)(dvisitor.visit(exp));
         SimplifyVisitor svisitor = new SimplifyVisitor();
          … }
      public Object visit(FixExp e) {
         SimplifyVisitor svisitor = new SimplifyVisitor();
         PrintVisitor pvisitor = new PrintVisitor();
         DeriveVisitor dvisitor=new DeriveVisitor(fList,t,e.id);
          … }
     }
```

Tablo 2.19'de bir kısmı verilen FNEvalVisitor.java sınıfına ait iki önemli metot vardır. Bunlar; DrvExp ve FixExp metotlarıdır. Bu metotlardan DrvExp metodu kullanılarak kök hesaplaması yapılacak olan denklemsel ifadeye ait türev alma işlemi için türev alma sınıfı olan DeriveVisitor.java sınıfı türetilir bununla birlikte türev alma işleminde sadeleştirme işlemine ihtiyaç duyulmasında ise SimplifyVisitor.java sınıfı türetilerek kullanılır böylece uygulamada kök hesaplama işlemi gerçekleştirilmiş olur. Aşağıdaki örnekte matematiksel ifade uygulamaya, komut satırı üzerinden verilmiştir.

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

Bu denklemsel ifade üzerinde türev alma işlemlerinin gerçekleştirilmesinde uygulamaya ait her bir bileşenin hangi verileri üreteceği aşağıdabelirtilmiştir. Öncelikli olarak uygulamaya ait ayrıştırıcı metotları aşağıda belirtilen nesne ağacını üretir.

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

Kök hesaplama işleminde oluşturulan bu tree nesne ağacından yeni bir nesne ağacı türetmeye ihtiyaç vardır. Bunun için üretilen bu nesne ağcından yine DrvExp metodu kullnılarak yeni bir nesne ağacı türetilir.

Bu aşamadan sonra yine ilgili metotlar kullanılarak türev alma işlemi sonucunda elde edilen nesne ağacından sadeleştirme işlemi ile yeni bir nesne ağacı oluşturulur. İfade üretici sınıf metotlarıyla matematiksel notasyonda sadeleştirilen nesne ağacı ifadesine dönüşüm ifade elde edilir.

Geliştirilen uygulamada Basit İterasyon Yöntemi ile kök hesaplama işlemleri gerçekleştirilmektedir. İlgili yöntemde uygulama için f(x)=0 denkleminde değişik x değerlerine bakılırak `g1(x)`, `g2(x)`.. ile ifade edilen yeni denklemler elde edilir ve bu denklemlerden `|g1'(x)|<1` ise `g1(x)` fonksiyonu, `|g2'(x)|<1` ise `g2(x)` fonksiyonu FNEvalVisitor ile belirlenerek kök hesaplamada kullanılır. Bu yönteme ait kök hesaplama işlemlerinde türev alma, sadeleştirme ve kullanılacak denkleme karar verme işlemleri gerçeklenir. Bu işlemler Tablo 2.15'de ifade edilen FixExp metodu ile gerçekleştirilir. İlgili metot ile SimplifyVisitor, PrintVisitor, DeriveVisitor, FixedPointVisitor sınıfları türetilerek ve matematiksel sınıflar kullanılarak kök hesaplama işlemi gerçekleştrilir.

2.4.5.2. Türev Alıcı

Geliştirilen uygulamada matematiksel denklemlerin kök bulma işlemlerinde türev alım işlemine ihtiyaç duyulmaktadır. Bundan dolayı, uygulamaya bir türev alıcı eklenmiştir. Türev alma işleminde oluşturulan nesne ağacı en içteki düğümden başlanarak kök düğüme doğru değerlendirme işlemine tabi tutulur. Burada iki işlem yapılmaktadır bunlardan birincisi, nesne türünün belirlenmesi diğeri ise gösterilen işlemin gerçekleştirilmesidir. Bu iki işlem ise üç farklı metot ile yapılmaktadır. Bunlar;

- Java instanceof işleci,
- Gramere ait sözdizim sınıflarına metot ekleme işlemi,
- visitor tasarım deseni yöntemleriyle gerçekleştirilmektedir.

Geliştirilen bu uygulamada türev alma işleminin yanı sıra türev alma işlemi için uygun sadeleştirme işlemini gerçekleştirme ve işlem yapılan ifadenin gösterilmesi işlemleride gerçekleştirilmiştir. Ayrıca bu işlemlere özel farklı Visitor desenleri gerçekleştirilmiştir.

Uygulamada türev alma işlemi için

$$y = x^x = lny = xlnx = \frac{y'}{y} = x * lnx + \left(x * \left(\frac{1}{x}\right)\right) = lnx + 1 = x^x(1 + lnx) \quad (2.1)$$

Denklem yapısı kullanılmıştır. Örneğin; $y = 3 * x^2$ denkleminin geliştirilen uygulama ile türev alma işle uygulandığında oluşacak nesne ağacı aşğıda gösterilmiştir. Türev alma işleminde denklem 2.1'de ifade edilen formül kullanılmıştır.

`Exp1= new Times(new num(3),new Pow(new var("x"),new num(2)))` değerlendirilecek fonksiyona ait ağaç yapısı `Exp1` ile gösterilmiştir.

`Exp2=new Times(new Times(new num(3),new Pow(new var("x"),new num(2))),new Plus(new Times(new num(0),new ln(new num(3),new var(x)) new Times(new num (2), new divide(new num(3),new Times(new num(3), new var(x)))))))`

Türev alma işleminden sonra `Exp2` nesne ağacı elde edilmiştir

Oluşturulan nesne ağacına ait düğümleri değerlendirmek için uygulamada DeriveVisitor arayüzü ve bu arayüze ait işlemleri gerçekleştiren driver sınıfı kullanılmıştır. İlgili arayüzler Tablo 2.20'de gösterilmiştir. İfade edilen DeriveVisitor.java ile yapılan işlemler DeriveVisitor sınıfının nesne ağacına erişilerek yeni nesne ağaçlarının türetilmesi işlemlerini gerçekleştirmektir. Uygulamada kullanılan bazı sayısal yöntemler için ikinci dereceden türev alma işlemine ihtiyaç duyulmaktadır. Bunun anlamı ise oluşturulan nesne ağacından yeniden farklı bir nesne ağacının tekrar tekrar türetilerek ilgili fonksiyona ait istenen miktarda türev alıma işlemi ile yeni denklemsel yapıların elde edilmesidir.

Örneğin, `x^3-3*x+2` denklemine ait kök hesaplama işleminin Halley's yöntemi ile hesaplanması durumunda denkleme ait birinci ve ikinci dereceden türev alma işlemine ihtiyaç duyulacaktır. Uygulama girdi verisi olarak alınan denklem yapısını alarak derler ve bir ağaç yapısı oluşturur. Daha sonra bu ağaç yapısından türetme yoluyla ilgili denklemin birinci derceden türevi olan `((3*(x^2))-3)` denklemsel ifadesi elde edilir. Fakat bu işlem kullnılan sayısal yöntem için yeterli olmayacaktır bundan dolayı oluşturulan yeni nesne ağacının yeniden türevi alınarak yeni bir nesne ağacı türetilecektir. Türetilen bu

nesne ağacından ise `(3*(2*x))` denklemi elde edilerek ilgili fonksiyona ait kök hesaplama işlemi gerçekleştirilecektir.

Tablo 2.20. FixedPointVisitor sınıfı (DeriveVisitor. java)

```
public class DeriveVisitor implements Visitor {
     FnDef[] fL = new FNParser(System.in).Prog();
     DeriveVisitor fp = new DeriveVisitor(fL, new Table(10), "x");
 public DeriveVisitor(FnDef[] fL, Table tb, String var){
     fList = fL; t = tb;
     deriveVar = var;}
 public Object visit(Plus e) {Exp a = (Exp)(e.a.accept(this));
     Exp b = (Exp)(e.b.accept(this));
     return new Plus(a, b);}
 public Object visit(Minus e) { Exp a = (Exp)(e.a.accept(this));
            Exp b = (Exp)(e.b.accept(this));
            return new Minus(a, b);}
     … }
```

2.4.5.3. Sadeleştirici

Matematiksel fonksiyonların türev alma işlemi gerçekleştirilirken ilgili fonksiyon ifadesinde sadeleştirme işlemlerine ihtiyaç duyulabilmektedir. Bunun nedeni ise türev alma işleminden sonra üretilen yeni fonksiyonda sadeleşecek kısımlar bulunur ve kısımlar sadeleştirilmezse fonksiyonun yapısı karmaşık ve gereksiz çok miktarda veri içerebilir bu durum ilgili fonksiyonun okunurluğunu azaltır. Karşılaşılacak muhtemel problemleri engellemek için türevi alınan ifade sadeleştirme işlemine tabi tutulur. Uygulamada tanımlanan DeriveVisitor sınıfına ait metotları aracılığı ile nesne ağacının düğümlerine ulaşılarak ifadenin en sade şekline dönüştürülmesi gerekmektedir. Bu sadeleştirme işlemi yapılmadığı zaman elde edeceğimiz nesne ağacı Türev Alma bölümünde gösterilen `Exp2` ifadesi gibi çok karmaşık bir fonksiyon yapısı ile karşılaşılacaktır. Örneğin; Karmaşık denklem yapısındaki `Exp2` fonksiyonu sadeleştirilirse aşağıda ifade edilen `Exp3` nesne ağacı elde edilecektir.

```
Exp3=new Times(new num(6),new var(x))
```

Türev alma bölümünde türev işleminden sonra elde edilen `Exp2` ile `Exp3` nesne ağacı karşılaştırıldığında sadeleştirilen `Exp3` nesne ağacının çok daha sade ve okunaklı olduğu görülecektir.

Tablo 2.21'de sadeleştirici arayüzü ve bu arayüzü gerçekleştiren sadeleştirme sınıfına ait kod yapısının bir bölümü gösterilmiştir. Tablo 2.21'de gösterilen Visitor arayüzünden yeni bir sınıf gerçeklenir ve türev alıcının üreteceği nesne ağacı düğümleri üzerinde yeni visit() metotları tanımlanır. Bu metotlarla erişilen düğümün sözdizim sınıfı türüne göre sadeleştirme işlemleri gerçekleştirilebilir.

Tablo 2.21. FixedPointVisitor sınıfı (SimplfyVisitor. java)

```
public class SimplifyVisitor implements Visitor  {
        public void visit(Stm s) { }
        public Object visit(IfExp e)
                {return new Num(0); }
        public Object visit(Fn e)
                { return new Num(0); }
        public Object visit(Exp e)
                {return e.accept(this);}
        public Object visit(Plus e){
                Exp a = (Exp)(e.a.accept(this));
                Exp b = (Exp)(e.b.accept(this));
               if (a instanceof Num && ((Num)a).n < 0 }…}
```

2.4.5.4. İfade Gösterici

Türevi alınacak ifadeye ait nesne ağacının düğümleri ilgili sadeleştirme metotlarıyla değerlendirildikten sonra oluşan ifadenin matematiksel gösterim aşamasına geçilir. Tablo 2.21'de ilgili ifadeye ait nesne ağacından matematiksel ifadeye dönüşüm yapan bir ifade gösterici sınıfı ve metotları verilmiştir. Nesne ağacı düğümlerini değerlendirmek için Tablo 2.22'de tanımlanan PrintVisitor arayüzü ile bu arayüzü gerçekleyen ifade gösterici sınıfı kullanılmıştır.

Sadeleştirici sonucu elde edilen nesne ağacı ifade gösterici ile tür dönüşümü işlemine tabi tutularak aşağıdaki String yapıya sahip olur.

```
String str = ”6*x”
```

Tablo 2.22. FixedPointVisitor sınıfı (PrintVisitor. java)

```
public class PrintVisitor implements Visitor {
     public Object visit(Exp e) {return e.accept(this);}
     public Object visit(Plus e) {
           String a, b;
           a = (String)e.a.accept(this);
           b = (String)e.b.accept(this);
             return a + "+" + b;}
     public Object visit(Minus e) {
           String a, b;
           a = (String)e.a.accept(this);
           return a + "-" + b;} }…}
      public Object visit(Times e) {
          String a, b;
          if (e.a instanceof Num || e.a instanceof Var)
          a = (String)e.a.accept(this);
          else
          a = "(" + e.a.accept(this) + ")";
          if (e.a instanceof Num || e.a instanceof Var)
          b = (String)e.b.accept(this);
          else
          b = "(" + e.b.accept(this) + ")";
          return a + "*" + b;}
```

2.4.5.5. Fonksiyon Dönüştürücü

Geliştirilen uygulamada programlanan sayısal yöntemlerden, Basit İterasyon Yönteminde kök değerlerinin hesaplanabilmesi için girdi verisi olarak alınan `f(x)=0` şeklindeki denklemsel ifadeden `g1(x)=x`, `g2(x)=x`… `gn(x)=x` şeklinde bir veya birden fazla fonksiyon elde edilmesi gerekmektedir. Bu işlem için verilen kaynak veri olarak alınan `f(x)` fonksiyonuna ait her x ifadesi için `g(x)=x` şeklinde bir denklem ifadesi geliştirilen uygulama tarafından elde edilir. Daha sonra elde edilen fonksiyonlar koşul değerlendirmesine tabi tutularak `|g'(x)|<1` koşulunu sağlayan `g(x)` fonksiyonu `f(x)` fonksiyonuna ait kök hesaplama işlemlerinde kullanılır.

Geliştirilen uygulamada ilgili kaynak veri için fonksiyon dönüştürme ve fonksiyon kontrol işlemlerinin gerçeklenmesi `FixedPointVisitor.java` sınıfı tarafından gerçekleştirilmektedir.

Yorumlama işleminin bu aşamasında yorumlamaya ait bütün bileşenler kullanılmaktadır. Bunun nedeni fonksiyon dönüştürme işlemi için aynı zamanda türev alma, sadeleştirme, hesaplama ve ifade gösterici bileşenlerinin kullanılmak zorunda olunmasıdır. Tablo 2.23'da Kök bulma uygulamasına ait java kodunun bir kısmı verilmiştir.

Tablo 2.23. FixedPointVisitor sınıfı (FixedPointVisitor.java)

```
public class FixedPointVisitor implements Visitor {
        FixedPointVisitor fp=new FixedPointVisitor(new Num(0), fL, new
        Table(10), "x");
        Exp[] e = { new DNum(1.0), null }
        SimplifyVisitor svisitor = new SimplifyVisitor();
public FixedPointVisitor(Exp initE,FnDef[] fL,Table tb,String var){
        e_index = 0;
        initExp = initE;
        fList = fL; t = tb;
        deriveVar = var;
public Object visit(Minus e) {
        exp[e_index] = new Plus(e2, e.b);
       …}}
```

Aşağıda Şekil 2.3'de görüldüğü gibi değerlendirilecek kaynak kod geliştirilen uygulamada arayüzden giriliyor. Değerlendirme işleminden sonra iki adet fonksiyon elde edilmiştir. Uygulama Fonksiyon dönüştürme sınıfı matematiksel ifade uygulamaya, komut satırı üzerinden verilmiştir.

Değerlendirilecek Kaynak Veri...:

```
f(x)=x*ln(x)+exp(1-x)-1

FIX(x) : { y=fix(f(x),x); print(x,y); print() }
      | abs(y-x) < 0.001 = y
      | otherwise       = FIX(y)

main() : { print("x  ", "y"); print() } = FIX(2.0)
```

Fonksiyon Dönüşümü ve Seçilen Fonksiyon...:

```
FP iterasyon fonksiyon 1: (1-(exp(1-x)))/(ln(x))
FP iterasyon fonksiyon 2: (2.718281828459045)^((1-(exp(1-x)))/(x))
Kullanılan  fonksiyon2
```

Şekil 2.3. Fonksiyon dönüşümü

Şekil 2.3’de görüldüğü gibi Basit İterasyon yöntemi için gerekli olan fonksiyon dönüşümü geliştirilen uygulama ile uygun olan `g(x)` fonksiyonunu belirlenmiştir. Daha sonra belirlenen bu fonksiyon üzerinden kök hesaplama işlemlerini gerçekleştirmiştir.

2.5. Sayısal Yöntemleri Programlama

Bu bölümde yukarıda biçimsel ve yapısal olarak tanımlanan programlama dili kullanılarak sayısal kök bulma yöntemlerinin genel programlaması gösterilecektir.

Genel yapısı Tablo 2.1'de verilen gramer fonksiyonlar üzerinde yapılacak değişik hesaplamalarda kullanılır. Bunlar; `f(x)`'i belirli bir x değeri için hesaplama, türevini alma, sadeleştirme ve fonksiyon dönüşüm işlemlerinin gerçeklenmesidir. Genel olarak programlanan yöntemler; f(x,y) gibi iki parametreli veya f(x) biçiminde tek parametreli bir fonksiyon olarak programlanabilmektedir. Ayrıca programlanan yöntemlerin algoritmalarında ortak olarak; print komutu ile hesaplanan değerler yazdırılmış, abs işlevi ile mutlak değer karşılaştırması yapılmış, otherwise işlevi ise ilk iki koşul ile kapsanmayan tüm diğer durumları temsil edilmiştir. Ayrıca kök hesaplama işlemleri gerçekleştirilen fonksiyona ait grafiği de uygulama arayüzünden elde edilebilmektedir.

Tablo 2.1'deki gramerin JavaCC'ye bildirimi yapılarak, kaynak verinin nesne ağacını üretebilen ve değerlendirmesini yapabilen bir yorumlayıcı geliştirilmiştir. Geliştirilen uygulamada değerlendirme işlemi yorumlayıcı tarafından Visitor tasarım deseni ile gerçekleştirilmektedir. Sözdizim sınıflarına ait yerel veriler ile bu veriler üzerinde

tanımlanan işlemleri biribirinden ayırmak için Visitor tasarım deseni kullanılmaktadır. Visitor sınıfında tanımlanan metotlar ile sözdizim ağacının düğümlerine erişilmektedir.

2.5.1. Basit İterasyon Yönteminin Programlanması

Geliştirilen uygulamada kök hesaplama işlemlerinde kullanılmak için programlanan sayısal yöntemlerden birinci yöntem Basit İterasyon Yöntemidir. Bu yöntem ile $f(x) = 0$ biçiminde ifade edilen denklem yapısı içerisindeki her bir x değişkeni için girdi verisi olarak alınan fonksiyon $= g(x)$ şekline getirilerek fonksiyon dönüşüm işlemi gerçekleştirilir. Geliştirilen uygulamada Basit İterasyon Yöntemi için fonksiyon dönüşüm işlemlerini gerçekleştiren `FixedPointVisitor.java` sınıfı uygulamaya eklenmiştir. Önceki bölümlerde bu sınıftan detaylı olarak bahsedildiği için bu bölümde sınıfın ayrıntısına girilmemiştir. Daha sonra oluşan fonksiyonlardan yakınsama koşulu olan $|g'(x)| < 1$ ifadesi kontrol edilerek kök hesaplama işlemleri için kullanılacak fonksiyona karar verilir.

Basit İterasyon yöntemi diğer yöntemlere göre programlanması karmaşık olan bir yöntemdir. Tablo 2.24'de gösterildiği gibi geliştirilen uygulama kök hesaplama işlemi gerçekleştirilecek fonksiyonu, fonksiyona ait algoritmayı ve fonksiyonun yazdırılması ile başlangıç değerini tek bir kaynak veri alarak değerlendirir. Değerlendirme sonucunda fonksiyona ait kök hesaplama işlemleri gerçekleştirilir. Basit İterasyon yönteminin programlanması Tablo 2.24'de gösterilmiştir.

Tablo 2.24. Basit İterasyon yönteminin programlanması

```
f(x)=x*ln(x)+exp(1-x)-1
FIX(x) : { y=fix(f(x),x); print(x,y); print() }
            | abs(y-x) < 0.001 = y
            | otherwise = FIX(y)
main() : { print("x  ", "y"); print() } = FIX(2.0)
```

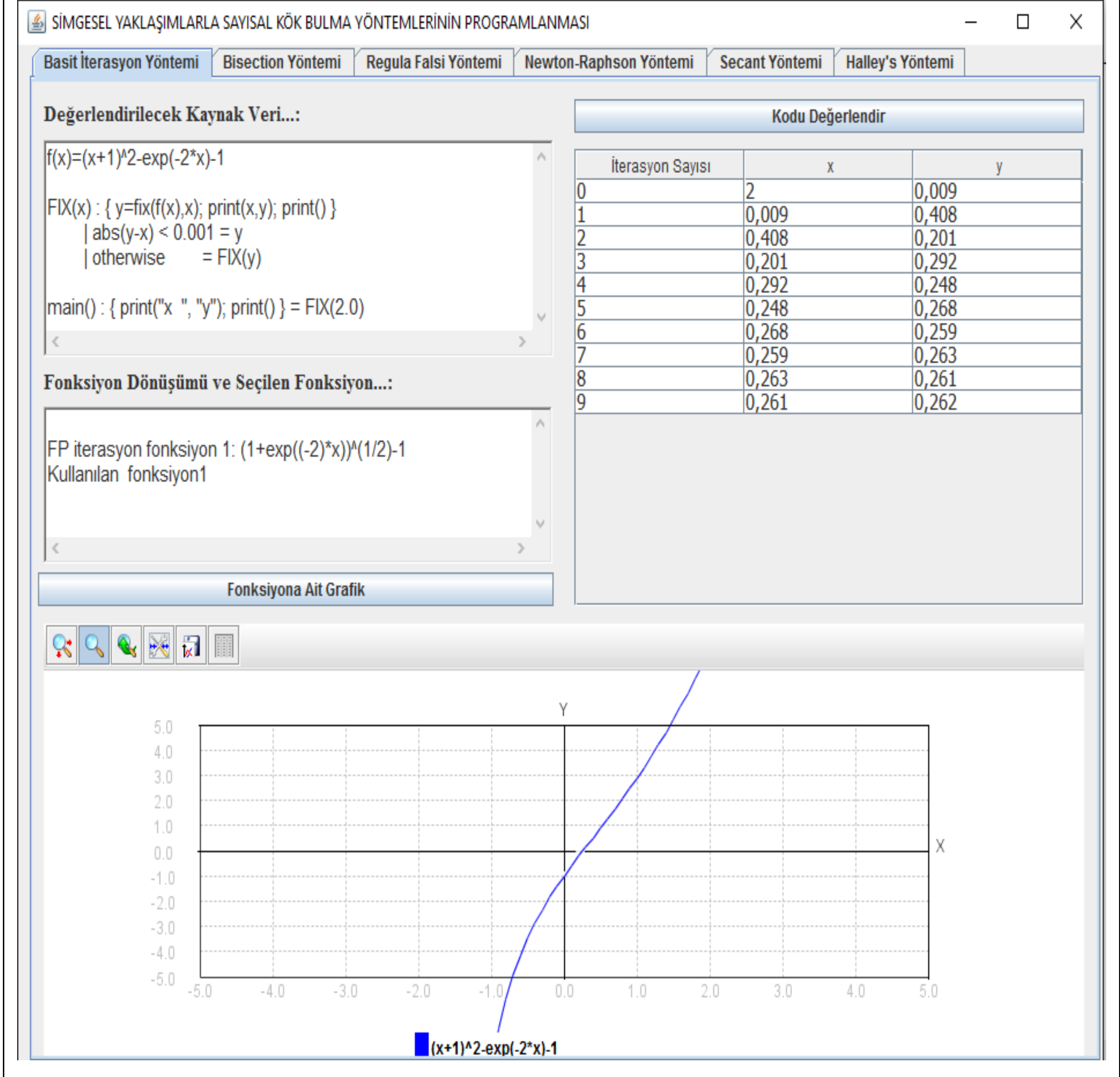
Tablo 2.24'deki kaynak kodun yapısına bakıldığında ilk satırda `f(x)=x*ln(x)+exp(1-x)-1` şeklinde fonksiyon tanımlaması yapılmıştır. Uygulama bu fonksiyonu ayrıştırma işlemine tabi tutarak "=" ifadesinden sağ ve sol tarafa

dallanan bir ağaç yapısı oluşturur. Daha sonra eşitliğin sağ tarafı kök hesaplama işleminde kullanır. Ayrıca programlanan bu yöntemin tek parametre aldığı yine kaynak koddan görülmektedir. FIX(x):{y=fix(f(x),x)} bu satırda ise Basit İterasyon yönteminde ilgili fonksiyona ait kök değerlerinin nasıl hesaplandığı gösterilmektedir. Burada y=fix(f(x),x) tanımlamasına bakıldığında "fix" olarak ifade edilen yapı ilk olarak javaCC dosyasında token olarak <FIX><LPAREN>e=E()<COMMA>t=<ID><RPAREN> şeklinde tanımlanmış ve gramer kuralı belirlenmiştir. Gramerin yapısına bakıldığınada "fix" yapısı, bir ifadeyi ve bir değişkeni parametre olarak alıp işlem yapan bir fonksiyon şeklinde tanımlanmıştır. abs(y-x) < 0.001 = y burada kök hesaplama işlemi yapıldıktan sonra hesaplanan değerlerin farkının mutlak değeri belirlenen hassaslık ile karşılaştırılır. İlgili şart sağlanana kadar kök hesaplama işlemi devam eder.

main():{print("x","y");print()}=FIX(2.0) bu kısımda hesaplanan değerlerin yazdırılması ve başlangıç değerinin alınması işlemini gerçekleştirir.

Basit İterasyon Yöntemi kullanılarak Şekil 2.4'te örnek bir fonksiyona ait kök hesaplama işlemi gerçekleştirilmiştir. Şekil 2.4'te gösterildiği gibi kök hesaplama işlemi gerçekleştirilecek kaynak veri belirtilen formatta arayüzden girilirek uygulama tarafından değerlendirme işlemine tabi tutulur.

Bu yönteme örnek olarak, $f(x) = (x+1)^2 - \exp(-2 * x) - 1$ fonksiyonu ve başlangıç değeri $x0 = 2$ olan Basit İterasyon yöntemi ile hesaplanan değerler Şekil 2.4'te gösterilmiştir.



Şekil 2.4. $(x + 1)^2 - \exp(-2 * x) - 1$ fonksiyonuna ait hesaplanan kök değerleri

2.5.2. İkiye Bölme Yönteminin Programlanması

Geliştirilen uygulamada kök hesaplama işlemleri için programlanan sayısal yöntemlerden bir diğeri ise İkiye Bölme Yöntemidir. Tablo 2.25'de gösterildiği gibi kök hesaplama işlemine tabi tutulacak kaynak veri; kök hesaplaması yapılacak fonksiyonu, İkiye Bölme yöntemine ait algoritmayı ve fonksiyonun hesaplanacağı aralığa ait başlangıç ve bitiş değerleri ile yazdırma bileşenlerini içermektedir. Kaynak verinin üzerinde değerlendirme işlemi gerçekleştirildikten sonra fonksiyona ait kök değerleri hesaplanmış ve uygulama için geliştirilen arayüzde gösterilmiştir.

İkiye Bölme Yöntemi Tablo 2.25'de verilen kaynak kod yapısına uygun olarak, `BS(x,y)` biçiminde bir fonksiyon olarak programlanmıştır.

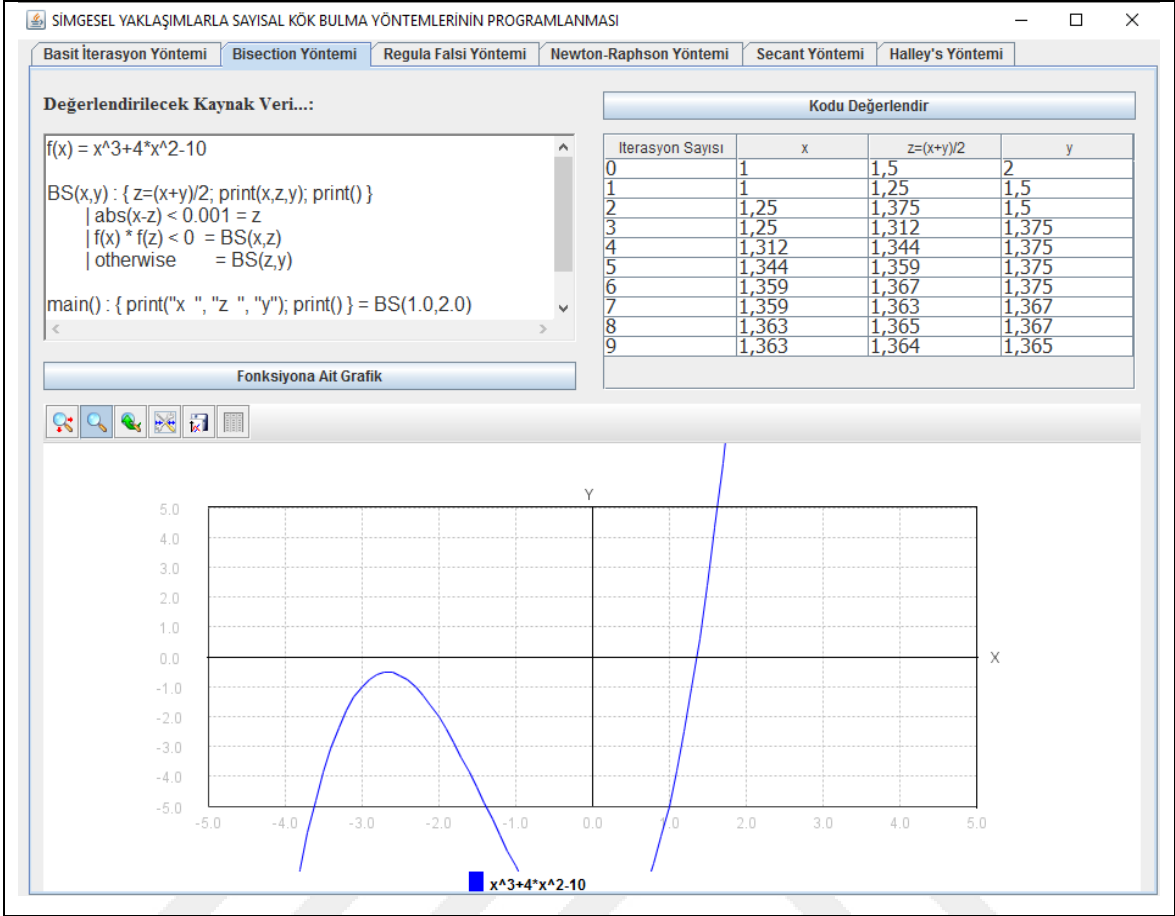
Tablo 2.25. İkiye bölme yöntemini programlama

```
f(x) = x^3+4*x^2-10
BS(x,y) : { z=(x+y)/2; print(x,z,y); print() }
             | abs(x-z) < 0.001 = z
             | f(x) * f(z) < 0  = BS(x,z)
             | otherwise        = BS(z,y)
main() : { print("x  ", "z  ", "y"); print() } = BS(1.0,2.0)
```

Tablo 2.25'de İkiye Bölme Yöntemi'ni programlamak için gösterilen kaynak verinin ilk satırında `f(x)=x^3+4*x^2-10` şeklinde tanımlanan denklemin sağ tarafı kök hesaplama işlemlerinde kullanılacak şekilde geliştirilen uygulama tarafından ayrıştırılır. Programlanan yöntemin genel yapısını belirten `BS(x,y)` yapısına bakıldığında ilgili yöntemin iki parametre aldığı görülmektedir. `z=(x+y)/2` ifadesi ile kök değerleri hesaplanmaktadır. `abs(x-z) < 0.001 = z` ile hata oranı kontrol edilmektedir. Bu değere göre kök hesaplama işlemleri tekrarlanmaktadır. Koşul sağlandığında kök hesaplama işlemi sonlandırılmaktadır. `f(x)*f(z)<0=BS(x,z)` ifadesi ile fonksiyonun kök aralığı kontrol edilip güncellenmektedir. `otherwise= BS(z,y)` bu satır ile ilk iki koşul haricindeki bütün koşullar ifade edilmekte olup yine bu satır ile fonksiyona ait kök aralığı güncellenmektedir. `print("x","z","y")` ifadesi ile yazdırma işlemi, `BS(1.0,2.0)` ile başlangıç değerleri belirlenmektedir.

Aşağıdaki örnek ile Programlanan İkiye Bölme Yöntemi kullanılarak nasıl kök hesaplama işlemi gerçekleştirildiği gösterilmiştir. Şekil 2.5'te görüldüğü gibi kök hesaplama işlemi gerçekleştirilecek kaynak veri arayüzden girilir. Daha sonra ilgili fonksiyonun kök değerleri programlanan İkiye Bölme Yöntemi ile adım adım hesaplanıp arayüzde gösterilir.

Örneğin, [1,2] aralığında bir kökünün olduğu bilinen $f(x) = x\text{^}3 + 4 * x\text{^}2 - 10$ fonksiyonu için İkiye Bölme (yarılama) yöntemi hesaplanan değerleri Şekil 2.5'de gösterilmiştir.



Şekil 2.5. $x^3 + 4x^2 - 10$ fonksiyonuna ait hesaplanan kök değerleri

2.5.3. Regula Falsi Yönteminin Programlanması

Bu çalışmada kök hesaplama işlemleri için programlanan sayısal yöntemlerden olan Regula Falsi Yöntemi, İkiye Bölme Yöntemi ile Secant Yöntemi'nin birlikte kullanılmasıyla oluşan bir yöntemdir. Programlanan bu yöntemde Secant Yöntemi kullanılarak iterasyonlar oluşturulurken, aynı zamanda İkiye Bölme yönteminde olduğu gibi her bir işlem basamağında hesaplanacak kök değerini içeren aralık kontrol edilerek işlem adımlarına devam edilerek ilgili fonksiyona ait kök değerleri hesaplanır.

Tablo 2.26'da programlanan yönteme ait kaynak veri kümesi gösterilmiştir. Kaynak veri; kök hesaplaması yapılacak fonksiyonu, kök hesaplama işleminde kullanılacak algoritma yapısını ve hesaplanan kök değerlerinin yazdırılması bileşenlerinden oluşmaktadır. Değerlendirme sonucunda ilgili fonksiyona ait kök hesaplama işlemleri programalanan sayısal yöntem tarafından gerçekleştirilir. Tablo 2.26'de verilen gramere

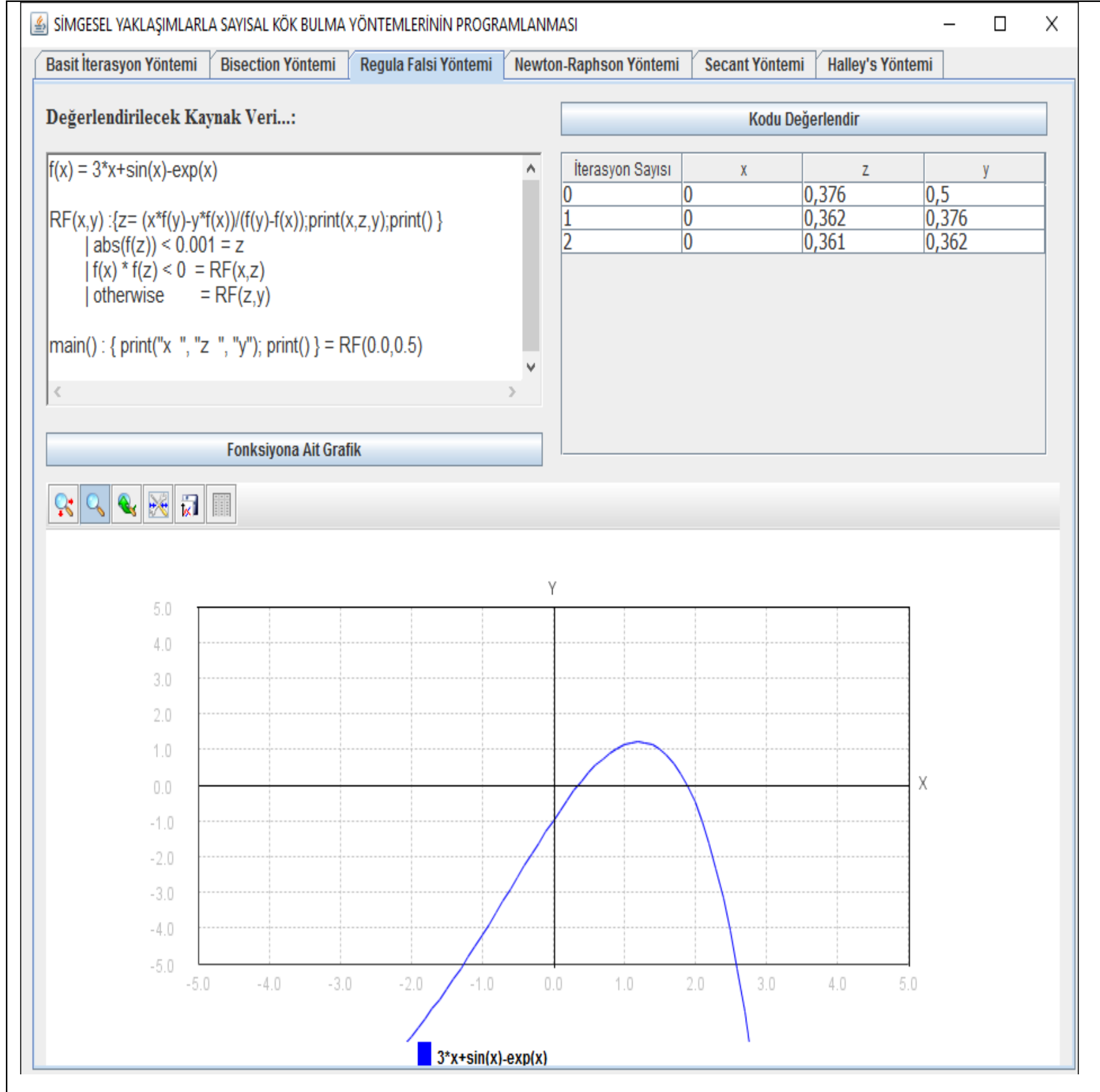
uygun olarak, Regula Falsi Yöntemi RF(x,y) biçiminde bir fonksiyon yapısı olarak programlanmıştır.

Tablo 2.26. Regula Falsi yöntemini programlama

```
f(x) = 3*x+sin(x)-exp(x)
RF(x,y) :{z= (x*f(y)-y*f(x))/(f(y)-f(x));print(x,z,y);print() }
             | abs(f(z)) < 0.001 = z
             | f(x) * f(z) < 0  = RF(x,z)
             | otherwise        = RF(z,y)
main() : { print("x  ", "z  ", "y"); print() } = RF(0,0.5)
```

Aşağıdaki bir örnek uygulama ile Regula Falsi Yöntemi programlanarak kök hesaplama işleminin gerçekleştirilmesi gösterilmiştir. Şekil 2.6’te gösterildiği gibi kök hesaplama işlemi gerçekleştirilecek kaynak kod belirtilen formatta arayüzden girilerek değerlendirme işlemi gerçeklenir. Değerlendirilen kaynak veri programlanan sayısal yöntem kullanılarak ilgili fonksiyonun kök değerleri hesaplanıp arayüzde gösterilir. Tablo 2.26’daki kaynak veride belirtilen `RF(x,y):{z=(x*f(y)-y*f(x))/(f(y)-f(x)` yapısında görüldüğü gibi programlanan yöntem iki adet parametre almaktadır. `z=(x*f(y)-y*f(x))/(f(y)-f(x)` bu denklemsel hesaplama yapısında ise kök hesaplama işleminde bilinen iki kök değeri ve bu değerlere ait fonksiyonlar kullanılarak diüer bir kök değerinin hesaplandığı belirtilmiştir. `abs(f(z))<0.001=z` burada ise hesaplanan kök değerinin herbirine ait fonksiyonun mutlak değeri belirlenen sayı ile karşılaştırılır. İfadede belirtilen koşul sağlayana kadar kök hesaplama işlemlerine devam edilir. `f(x)*f(z)<0=RF(x,z)` Bu satırda İkiye bölme yönteminde olduğu gibi hesaplanan değerlere ait fonksiyonlarının çarpımlarının sonucunun pozitif veya negatif olmasına göre hesaplanacak fonksiyonun kök değerleri aralığı güncellenir. Kaynak verinin devamındaki kısımalr ise diğer yöntemlerde olduğu gibi yazdırma ve ilk değerleri tanımlama kısımlarından oluşmaktadır.

Örneğin, $f(x) = 3 * x + \sin(x) - \exp(x)$ fonksiyonuna ait [0,0.5] aralığında bir kökünün olduğu bilinmektedir. Regula Falsi yöntemi ile fonksiyona ait hesaplanan kök değerleri Şekil 2.6’te gösterilmiştir.



Şekil 2.6. $3 * x + \sin(x) - \exp(x)$ fonksiyonuna ait hesaplanan kök değerleri

2.5.4. Newton-Raphson Yönteminin Programlanması

Newton-Raphson Yöntemi'nin programlanması için geliştirilen kaynak veri Tablo 2.27'de gösterilmiştir. Programlanan diğer sayısal yöntemlerde olduğu gibi ilgili tabloda belirtilen kaynak veri; hesaplama işlemi yapılacak fonksiyonu, fonksiyona ait algoritmayı ve fonksiyonun yazdırılması ile başlangıç değerinden oluşmaktadır. Belirtilen kyanak veri değerlendirilerek ilgili fonksiyona ait kök değerleri hesaplanır. Hesaplanan kök değerleri geliştirilen arayüzde gösterilir. Bu yöntem Tablo 2.27'de gösterildiği gibi `NR(x,y)` biçiminde bir fonksiyon olarak programlanmıştır.

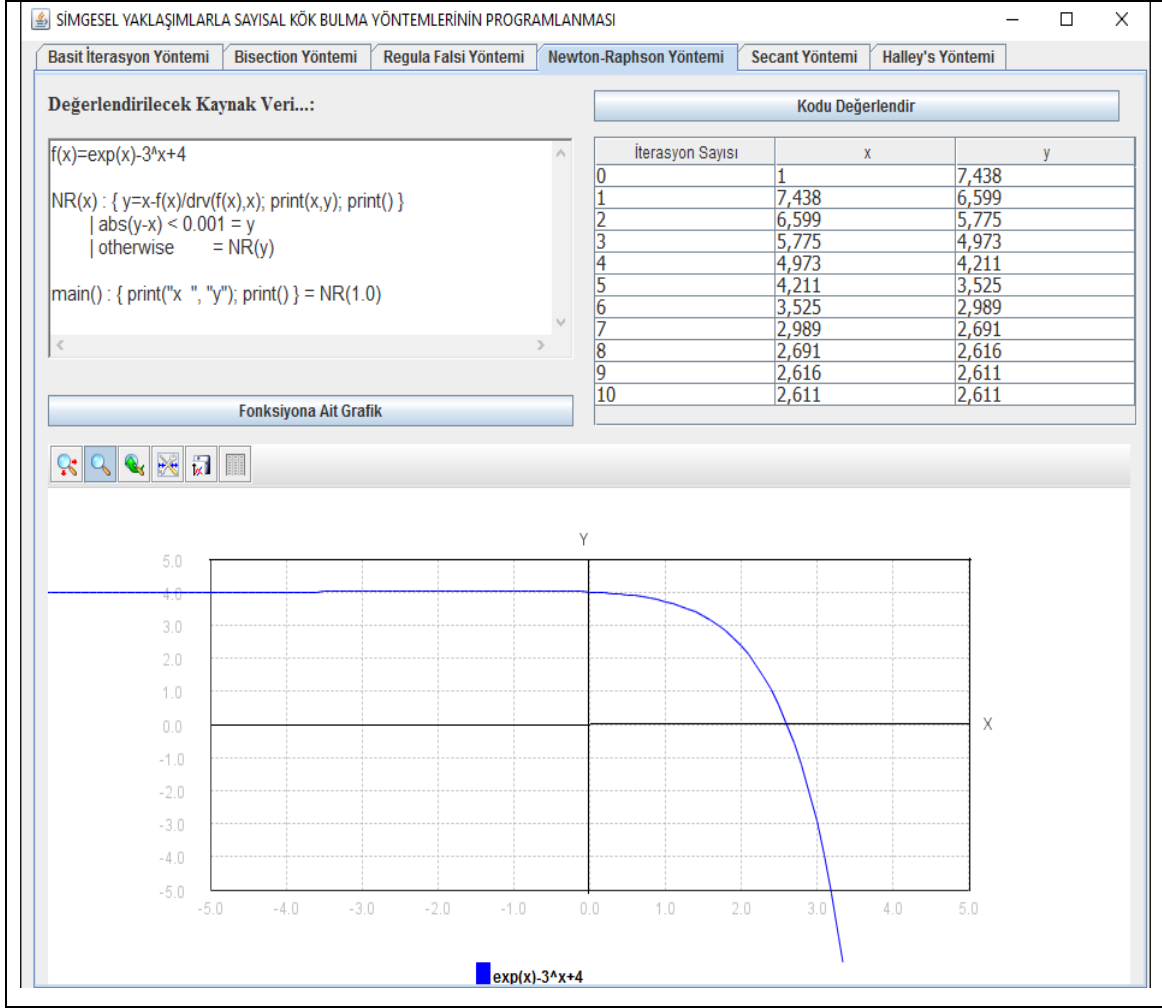
Tablo 2.27. Newton-Raphson yönteminin programlanması

```
f(x)=exp(x)-3^x+4
NR(x) : { y=x-f(x)/drv(f(x),x); print(x,y); print() }
            | abs(y-x) < 0.001 = y
            | otherwise        = NR(y)
main() : { print("x  ", "y"); print() } = NR(1.0)
```

Tablo 2.27'deki kaynak verideki ilk satırda kök hesaplama işlemi gerçeklenecek fonksiyon gösterilmiştir. `NR(x)` ifedesi ile programlanan yöntemin tek bir parametre aldığı belirtilmiştir. `y=x-f(x)/drv(f(x),x)` ifadesinde kök hesaplama işlemine ait denklem yapısı belirtilmiştir. Newton-Raphson yönteminde kök değerlerinin hesaplanabilmesi için türev alma işlemine ihtiyaç duyulmaktadır. Bunun için denklemde kullanılan "`drv`" ile türev alma işlemi gerçekleştirirlir. JavaCC dosyasında "`drv`" yapısına ait token tanımlaması `<DRV:"drv">` şeklinde, gramer tanımlaması ise `<DRV><LPAREN>e=E()<COMMA>(t=<NUM>{n=Integer.parseInt(t.image);}<COMMA>)?t=<ID><RPAREN>` şeklinde tanımlanmıştır. Tanımlanan gramer yapısına bakıldığında "`drv`" ifadesi parametre olarak bir ifadeyi ve yine ikinci parametre olarakda bir değişken veya bir sayıyı parametre olarak alan fonksiyon yapısında tanımlandığı görülmektedir. Tanımlanan bu gramer yapısına uygun olarak kaynak veride "`drv`" yapısı kullanılmıştır. Kaynak kodun devamında yine JavaCC dosyasında token tanımlaması ve gramer yapısı oluşturulan "`abs`" ve "`otherwise`" yapıları koşul şartlarını kontrol etmek için kullanılmıştır. Kaynak verinin son satırında ise hesaplanan kök değerlerini yazdırma işlemi gerçeklenmektedir. Ayrica türev alma işlemlerinin gerçekleştirilmesinde `DeriveVisitor.java` sınıfı kullanılmıştır.

Newton-Raphson Yöntemi için örnek bir uygulama Şekil 2.7'de gösterilmiştir. Kök hesaplama işlemi gerçekleştirilecek fonksiyon belirtilen formatta arayüzden girilir. Kaynak veri Newton-Raphson sayısal yöntemi kullanılarak ilgili fonksiyonun kök değerleri hesaplanıp arayüzde gösterilir.

Başlangıç değeri $x0 = 6.0$ olan $f(x) = \exp(x) - 3^x + 4$ fonksiyonu için Newton-Raphson yöntemi ile Şekil 2.7'de verilen değerler hesaplanmıştır.



Şekil 2.7. $x = \exp(x) - 3^x + 4$ fonksiyonu için hesaplanan kök değerleri

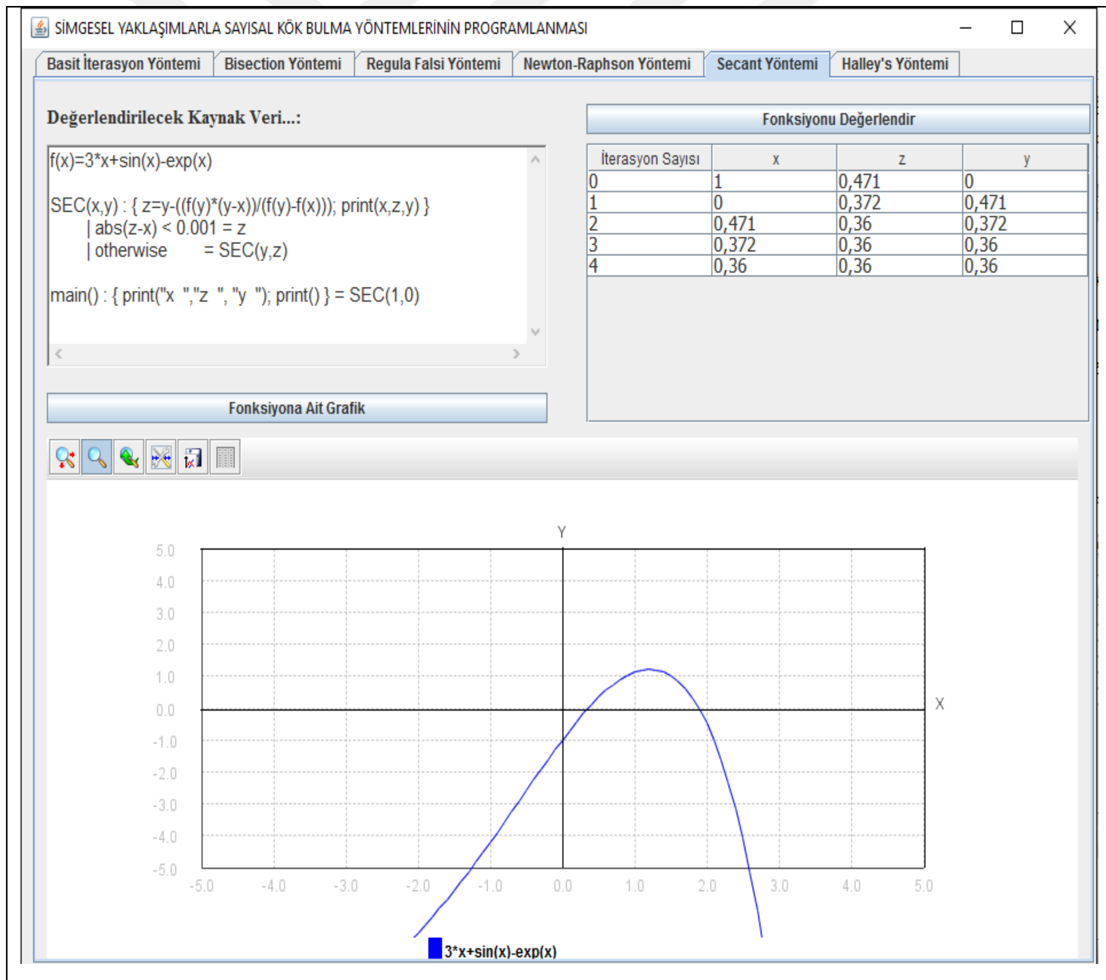
2.5.5. Secant Yönteminin Programlanması

Newton-Raphson Yöntemi programlanırken türev alma işlemine ihtiyaç duyulmuştur. Fakat bazı matematiksel fonksiyonlar için türev alma işleminın kullanılması zordur. Bundan dolayı bu denklemlere ait kök hesaplama işlemlerinde Sekant Yöntemi programlanarak kullanılmıştır. Tablo 2.28’de Secant (Kirişler) yönteminin programlanması için belirtilen kaynak kod yapısı gösterilmiştir. Kaynak kodun değerlendirmesi sonucunda ilgili fonksiyona ait kök hesaplama işlemi gerçekleştirilerek arayüzde kök değerleri gösterilmiştir. Tablo 2.28'de kaynak kod yapısı belirtilen Secant Yöntemi `SEC(x,y)` biçiminde bir fonksiyon olarak programlanmıştır. Yöntemin genel yapısına bakıldığında iki parametre alan bir fonksiyon olarak programlandığı görülmektedir.

Tablo 2.28. Secant yönteminin programlanması

```
f(x)=3*x+sin(x)-exp(x)
SEC(x,y) : { z=y-((f(y)*(y-x))/(f(y)-f(x))); print() }
              | abs(y-x) < 0.001 = z
              | otherwise        = SEC(z,y)
main() : { print("x  ","z  ", "y  "); print() } = SEC(0.0,1.0)
```

Şekil 2.8'de örnek bir uygulamada programlanan Secant Yöntemi ile kök hesaplama işleminin nasıl gerçekleştirildiği gösterilmiştir. Örneğin, $f(x) = 3 * x + \sin(x) - \exp(x)$ fonksiyonuna için [0.0,1.0] aralığında Secant yöntemi kullanılarak hesaplanan kök değerleri ve kaynak kod Şekil 2.8'de gösterilmiştir.



Şekil 2.8. $3 * x + \sin(x) - \exp(x)$ fonksiyonu için hesaplanan kök değerleri

2.5.6. Halley Yönteminin Programlanması

Programlanan diğer yöntemlerde olduğu gibi Halley yöntemine ait kaynak veri; fonksiyon, algoritma ve yazdırma işlemlerinden oluşmaktadır. Programlanan bu sayısal yöntemde Newton-Raphson Yöntemi'nde olduğu gibi türev alma işlemine ihtiyaç duyulmaktadır. Türev alma işlemi için bu yöntemde "`drv`" yapısı kullanılmıştır. Bu yöntemde Newton-Raphson yönteminden farklı olarak iki defa türev alım işlemi gerçekleştirilmektedir. Yönteme ait algoritma geliştirilirken bu durum göz önünde bulundurulmuştur. Bu yöntem Tablo 2.29'da gösterildiği gibi HL(x) biçiminde bir fonksiyon olarak programlanmıştır..

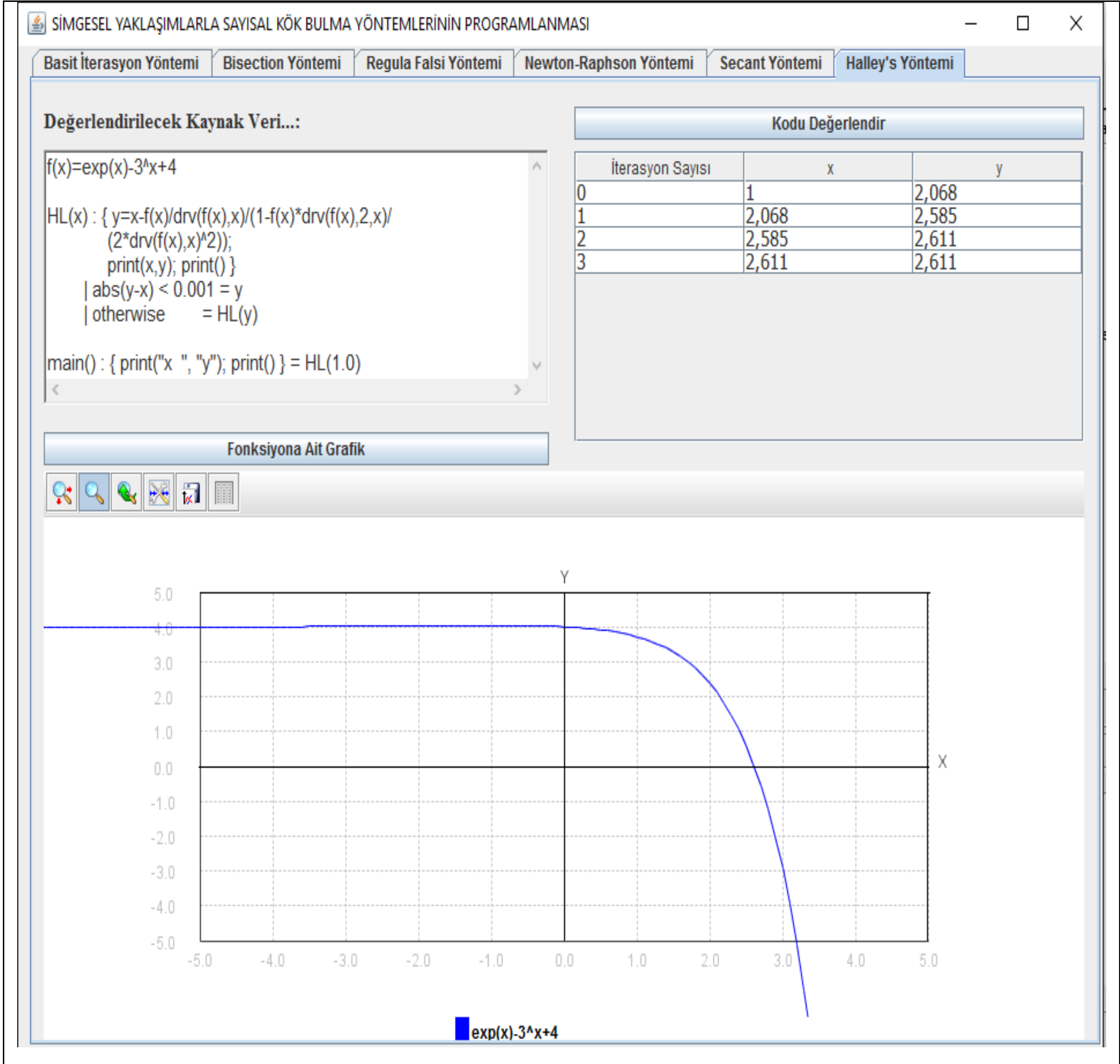
Tablo 2.29. Halley's yönteminin programlanması

```
f(x)=exp(x)-3^x+4
HL(x):{y=x-f(x)/drv(f(x),x)/(1-f(x)*drv(f(x),2,x)/(2*drv(f(x),x)^2));
             print(x); print() }
             | abs(y-x) < 0.001 = y
             | otherwise        = HL(y)
main() : { print("x  ", "y"); print() } = HL(1.0)
```

Halley's yöntemi kullanılarak örnek bir fonksiyonun çözümü Şekil 2.9'da gösterilmiştir. Kök hesaplama işlemi gerçekleştirilecek fonksiyon belirtilen formatta arayüzden girilir. Son olarak fonksiyonun aralığı ve yazdırma kısmının belirlendiği durum girilerek ilgili fonksiyonun kök değerleri Halley's yöntemi ile hesaplanarak arayüzde gösterilmiştir.

Başlangıç değeri $x0 = 6.0$ olan $f(x) = \exp(x) - 3^x + 4$ fonksiyonu için Halley's yöntemi ile Şekil 2.6'da verilen değerler hesaplanmıştır.

Örneğe dikkat edilirse Newton-Raphson Yöntemi ile çözülen aynı örnek olmasına rağmen işlem adımları daha azdır. Halley's yöntemiyle daha çabuk sonuca ulaşılmıştır.



Şekil 2.9. $\exp(x) - 3^x + 4$ fonksiyonuna ait grafik

Sonuç olarak programlanan sayısal yöntemlere ait kaynak kod yapılarına bakıldığında genel olarak benzerlik gösterdikleri görülmektedir. Programlanan yöntemlerin algoritmalarındaki farklar; yöntemin aldığı parametre sayısı ve yöntemde kök hesapla işlemi için kullanılan denklem olarak söylenebilir.

3. SONUÇ

Bu çalışmada Java programlama dilinden otomatik olarak kod üretebilen JavaCC kullanılarak ilgili dile ait gramer kuralları belirlenmiş böylece sayısal yöntemlerin simgesel olarak hesaplandığı bir uygulama geliştirilmiştir. Geliştirilen bu uygulamada öncelikle matematiksel fonksiyonlara ait ifadeler için EBNF notasyonunda bir gramer yapısı hazırlanmıştır ve bu yapı ile JavaCC yapısında tanımlama işlemi gerçeklenmiştir. Daha sonra oluşturulan bu gramer yardımıyla oluşturulabilecek bütün matematiksel ifadeleri temsil edebilecek bir nesne ağaç yapısında bir ayrıştırıcı oluşturulmuştur. Bu nesne ağacı gramer tarafından anlamlandırılan token yapılarından oluşmaktadır. Bu düğüm yapıları kullanılarak türev alma, sadeleştirme, ifade gösterme ve kök hesaplama işlemleri simgesel olarak hesaplanmıştır.

Bu çalışmada sayısal kök bulma yöntemlerinin otomatik kod üretim araçları ile nasıl programlanacağı gösterilmiştir. Programlama süreci türev alma, fonksiyonel dönüşüm ve iterasyon ifadelerinin üretimi gibi değişik simgesel programlama aktivitelerinden oluşmaktadır. Kök hesabı yapılacak bir matematiksel ifade, Java programlama dilinde otomatik kod üreten JavaCC aracı kullanılarak, öncelikle birkaç analiz işleminden geçirilir ve sonra nesne yapılarıyla temsil edilir. Problemin çözümüne yönelik seçilen sayısal yöntemin gerektirdiği bütün hesaplamalar bu nesne yapıları üzerinden yürütülür.

Çalışmada geliştirilen uygulama ile matematiksel işlemlerin kullanıldığı bütün mühendislik ve bilimsel hesaplamaların yapıldığı alanlardaki çeşitli denklem sistemleri, diferansiyel denklemleri ve belirsiz integrallerin simgesel çözümleme işlemleri gerçekleştirilebilir. Ayrıca problemin çözümünde sonuca götüren her bir hesaplama adımı gösterilebilir. Genellikle günümüzde yaygın olarak kullanılan genel ve özel amaçlı pek çok simgesel hesaplama sistemi ara adımları göstermeyip sadece işleme ait sonucu göstermektedirler.

4. ÖNERİLER

Kök hesaplama işlemlerinde kullanılmak üzere programlanan sayısal yöntemler için yapay zekâ öğrenmesinden faydalanılabilir. Böylece belirlenen bir yöntem ile çözülemyen problem için başka bir yöntem uygulama tarfından kullanıcıya önerilebilir.

Yapılan çalışmada, ilgili sayısal yöntem kullanılarak bir fonksiyonun belirtilen aralık veya başlangıç değeri için köklerinden sadece biri hesaplanmaktadır. Çalışma, fonksiyonun diğer aralıklar veya başlangıç değerleri için bütün köklerini hesaplayacak şekilde geliştirilebilir.

Uygulamayı daha kullanışlı hale getirmek için web ortamına taşınıp kök hesaplama işlemleri ve grafik yapılarına görsellik kazandırılabilir.

Uygulamaya daha fazla gramer yapısı eklenerek daha detaylı bir uygulama elde edilebilir.

Geliştirilecek yeni sayısal yöntemlerin çözüm işlemlerinde kullanılabilir.

Programın daha hızlı çalışması için mevcut sözdizim ağaç üretimi daha optimum bir hale getirilebilir.

Bu çalışma geliştirilmesiyle çözümü zor, karmaşık spesifik problemlerin çözümünü kolaylaştırır.

5. KAYNAKLAR

1. Chapra, S. C. ve Canale, R. P., Numerical Methods for Engineers, Sixth Edition, McGraw-Hill, New York, 2010.

2. Gathen, J. V. Z. ve Gerhard, J., Cambridge University Pres, Third Edition, Modern Computer Algebra, Cambridge, 2013.

3. Decker, W., Some Introductory Remarks on Computer Algebra, Proceedings of the Third European Congress of Mathematics, Barcelona, 2000.

4. Aho A., Lam M. ve Sethi R., Ullman J., Compilers Principles Techniques and Tools, Second Edition, Pearson, Boston, 2007.

5. Karan O., Design and Implementation of Interpreters, Yüksek Lisans Tezi, Haliç Üniversitesi, Fen Bilimleri Enstitüsü, İstanbul, 2005.

6. http://www.cse.ohiostate.edu/~xue/courses/655/Fang_Language.pdf Fang Language and Its Interpreter. 10 Ocak 2016.

7. Watt, D. ve Brown, D., Programming Language Processors in Java: Compilers and Interpreters, Prentice Hall, Reading, Massachusetts, 2000.

8. Roberts, E., An Overview of MiniJava, ACM SIGCSE Bulletin, 33 (2001) 1-5.

9. Parr, T. ve Fischer K. S., LL(*): The Foundation of the ANTLR Parser Generator, Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, Haziran 2011, California, Bildiriler Kitabı: 425-436.

10. Gagnon, E. M., ve Hendren, L. J., SableCC, An Object-Oriented Compiler Framework, Technology of Object-Oriented Languages and Systems, 26 (1998) 140-154.

11. http://compilers.cs.ucla.edu/jtb/JTB: Java Tree Builder. 10 Mart 2016.

12. https://javacc.dev.java.net JavaCC. 12 Mart 2016.

13. Berk, E., JLex: A Lexical Analyzer Generator for Java(TM), 2003.

14. Gerwin, K., JFlex User’s Manual, July 2005.

15. Boyle, A., ve Caviness, B. F., Report of a Workshop on Symbolic and Algebraic Computation, Washington, DC, April 1988.

16. Nolan, J. F., Analytical Differentiation on a Digital Computer, SM Thesis, Massachusetts Institute of Technology, Massachusetts, 1953.

17. Kahrimanian, H. G., Analytical Differentiation by a Digital Computer, MA Thesis, Temple University, Philadelphia, 1953.

18. Slagle, J. R., A Heuristic Program That Solves Symbolic Integration Problems in Freshman Calculus, Journal of the ACM (JACM), 10, 4 (1963) 507-520.

19. Berlekamp, E., Factoring Polynomials Over Finite Fields, Bell System Technical Journal, 46, 8 (1967) 1853-1859.

20. Zassenhaus, H., On Hensel Factorization, I. Journal of Number Theory, 1, 3 (1969) 291-311.

21. Musser, D., Multivariate Polynomial Factorization, Journal of the ACM (JACM), 22, 2 (1975) 291-308.

22. Wang, P. S., ve Rothschild, L. P., Factoring Multivariate Polynomials Over the Integers, Mathematics of Computation, 29, 131 (1975) 935-950.

23. Risch, R., The Problem of Integration in Finite Terms, Transactions of the American Mathematical Society, 139 (1969) 167-189.

24. Paule, P., ve Schorn, M., A Mathematica Version of Zeilberger's Algorithm for Proving Binomial Coefficient Identities, Journal of Symbolic Computation, 20, 5-6 (1995) 973-698.

25. Hearn, A. C., Reduce: A User-Oriented Interactive System For Algebraic Simplification, Symposium on Interactive Systems for Experimental Applied Mathematics: Proceedings of the Association for Computing Machinery Inc. Symposium, Ağustos 1967, Washington, Bildiriler Kitabı: 79-90.

26. Martin, W., ve Fateman, R., The Macsyma System, Symsac '71 Proceedings of the Second ACM Symposium on Symbolic and Algebraic Manipulation, Mart 1971, Los Angeles, Bildiriler Kitabı: 59-75.

27. Hearn, A. C., Reduce 2: A System And Language For Algebraic Manipulation, Proceedings of the Second ACM Symposium on Symbolic and Algebraic Manipulation, Mart 1971, Los Angeles, Bildiler Kitabı: 128-133.

28. Griesmer, J. H., ve Jenks, R. D., SCRATCHPAD/1: An Interactive Facility For Symbolic Mathematics, Proceedings of the Second ACM Symposium on Symbolic and Algebraic Manipulation, Mart 1971, Los Angeles, Bildiriler Kitabı: 17-18.

29. Bauer, C., Frink, A., ve Kreckel, R., Introduction To The GiNaC Framework For Symbolic Computation Within The C++ Programming Language, Journal of Symbolic Computation, 33, 1 (2002) 1-12.

30. Carette, J., Understanding Expression Simplification, Proceedings of the International Symposium on Symbolic and Algebraic Computation, Temmuz 2004, Santander, Bildiriler Kitabı: 72-79.

31. Shatnawi, M., ve Youssef, A., Equivalence Detection Using Parse-Tree Normalization for Math Search, Second International Conference on Digital Information Management ICDIM '07, October 2007, France, Lyon, Bildiriler Kitabı: 643-648.

32. Miyazaki, Y., Iguchi, Y., ve Watanabe, T., Information-Retrieval Tool for Math Expressions as Infrastructure of Training Engineers via E-Learning, 8th IFAC Symposium on Advances in Control Education, Ekim 2009, Kumamoto City, Bildiriler Kitabı: 302-306.

33. Singh, R., Gulwani, S., ve Rajamani, S., Automatically Generating Algebra Problems, AAAI'12 Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence,Temmuz 2012, Canada, Toronto, Bildiriler Kitabı: 1968-1975.

34. Fateman, R., Algorithm Differentiation in Lisp: ADIL, ACM Communications in Computer Algebra, 48,3/4 (2015) 78-89.

35. Tekbaş, Y., Otomatik Kod Üretim Araçları Yardımıyla Matematiksel İfadelerin Türevlerinin Hesaplanması ve Sadeleştirilmesi, Yüksek Lisans Tezi, Karadeniz Teknik Üniversitesi Fen Bilimleri Enstitüsü, Trabzon, 2013.

36. Milani, M. M. R., Design and Applications of Grammar-based Methodologies for Automatic Generation and Step-by-Step Solving of Mathematical Expressions. Trabzon: Doktora Tezi, K.T.Ü., Fen Bilimleri Enstitüsü, Trabzon, 2015.

37. Diffie, W., ve Hellman, M. E., New Directions in Cryptography, IEEE Transactions on Information Theory, 22, 6 (1976) 644-654.

38. Yerlikaya, T., Buluş, E. ve Arda, D., Asimetrik Kripto Sistemler ve Uygulamaları, II. Mühendislik Bilimleri Genç Araştırmacılar Kongresi, İstanbul, 2005.

39. Aho A.V. ve ULLMAN J.D., Principle of Compiler Design, Narosa Publishing Company House, New Delhi, 1985.

40. El-Kadri M., Groza V., Abielmona R. ve Assaf M., A Just-in-Time Compiler for a Reconfigurable Testing Platform, Proceedings of the IEEE Instrumentation and Measurement Technology Conference, Sorrento, Italy, 2006.

41. Scott, M. L., Programming Language Pragmatics, First Edition, Elsevier, London, 2000.

42. Puntambekar A.A., Compiler Design, Second Edition, Technical Publications, 2012.

43. Appel A. W., Modern Compiler Implementation in Java, Cambridge University Press, 2002.

44. Aho A., Lam M., ve Sethi R., Ullman J., Compilers Principles Techniques and Tools, Second Edition, PEARSON, Boston, 2007.

45. Hunter R., The Esence of Compilers, Prentice Hall, Great Britatin, 1999.

46. Hutton B., Computer Language Implementation, Computer Science,16 (2010).

47. Friedl, J., E., F., Mastering Regular Expressions, Second Edition, O'Reilly Media, Cambridge, 2003.

48. Kleene S. C., Representation of Events in Nerve Nets and Finite Automata, Princeton University Press, New Jersey, 1956.

49. Kakde, O. D., Algorithms for Compiler Design, First Edition, Charles River Media, Massachusetts, 2002.

50. Al-Mamory S.O., Zhang H., ve Abbas A. R., Modeling Network Attacks for Scenario Construction, IEEE International Joint Conference on Neural Networks, 3,6 (2008) 1495-1502.

51. http://ceng.ktu.edu.tr/labs/bs/[1]bs_gramer_degerlendirme.pdf Gramer Tabanlı Değerlendirme Yöntemleri. 12 Mart 2016.

52. Lerdo, H. G., A Translator Writing System for a Java Oriented Compilers Course, Yüksek Lisans Tezi, University of Florida, Princeton University with Jens Palsberg Purdue University, Florida, 2003.

53. Sebesta R. W., Consepts of Programming Languages, Ninth Edition, University of Colorado, USA, 2010.

54. https://en.wikipedia.org/wiki/LL_parser LL Parser. 21 Nisan 2016.

55. Knuth D.E., On the Translation of Languages from Left to Right, California Institute of Technology, 8, 6 (1965) 607-639.

56. Özçetin M., Java 1.5 Programlama Dili için Bir Ayrıştırıcı Tasarımı ve Gerçekleştirimi, Yüksek Lisans Tezi, Çanakkale Onsekiz Mart Üniversitesi, Fen Bilimleri Enstitüsü, Çanakkale, 2012.

57. Kodaganallur V., Incorporating Language Processing Into Java Applications: A JavaCC Tutorial, Journal IEEE Software, 21, 4 (2004) 70-77.

58. Donnelly C., ve Stallman, R., The Yacc-Compatible Parser Generator, Bison Version 3.0.4, 23 January 2015.

59. http://epaperpress.com/lexandyacc/download/LexAndYaccTutorial.pdf SableCC. 25 Nisan 2016.

60. http://www.engr.mun.ca/~theo/JavaCC-Tutorial/javacc-tutorial.pdf JavaCC. 16 Nisan 2016.

ÖZGEÇMİŞ

1978 yılında Erzurum'da doğdu. İlköğrenimini Tekirdağ'da Mediha Mehmet Tetikol İlköğretim Okulunda, Orta ve lise öğrenimini Tekirdağ İmama-Hatip Lisesinde tamamladı. 2003 yılında Abant İzzet Baysal Üniversitesi Bilgisayar Programcılığı'ndan mezun oldu. 2004 yılında Uluslarası Kıbrıs Üniversitesi Mühendislik Fakültesi Bilgisayar Mühendisliği Bölümü'nde burslu lisans programına başladı ve 2009 yılında bu bölümden mezun oldu. Bir dönem çeşitli yazılım firmalarında çalıştıktan sonra 2012 yılının mart ayından beri Gümüşhane Üniversitesi Torul Meslek Yüksekokulunda Öğretim Görevlisi olarak çalışmaktadır. 2013'ten beri Karadeniz Teknik Üniversitesi Fen Bilimleri Enstitüsü Bilgisayar Mühendisliği Anabilim Dalı'nda yüksek lisans öğrenimine devam etmektedir.