KARADENİZ TEKNİK ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI

BELİRSİZ İNTEGRAL PROBLEMLERİNİN ÇÖZÜMÜ İÇİN GENEL BİR YORUMLAYICININ SİMGESEL HESAPLAMA YAKLAŞIMLARI KULLANILARAK TASARIMI VE GERÇEKLENMESİ

YÜKSEK LİSANS TEZİ

CEYHAN YILMAZ

HAZİRAN 2019
TRABZON

KARADENİZ TEKNİK ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI

BELİRSİZ İNTEGRAL PROBLEMLERİNİN ÇÖZÜMÜ İÇİN GENEL BİR YORUMLAYICININ SİMGESEL HESAPLAMA YAKLAŞIMLARI KULLANILARAK TASARIMI VE GERÇEKLENMESİ

Ceyhan YILMAZ

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

Tezin Enstitüye Verildiği Tarih : 21 / 05 /2019
Tezin Savunma Tarihi : 18 / 06 /2019

Tez Danışmanı : Dr. Öğr. Üyesi Hüseyin PEHLİVAN

Trabzon 2019

KARADENİZ TEKNİK ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

Bilgisayar Mühendisliği Anabilim Dalında
Ceyhan YILMAZ tarafından hazırlanan

BELİRSİZ İNTEGRAL PROBLEMLERİNİN ÇÖZÜMÜ İÇİN GENEL BİR YORUMLAYICININ SİMGESEL HESAPLAMA YAKLAŞIMLARI KULLANILARAK TASARIMI VE GERÇEKLENMESİ

başlıklı bu çalışma, Enstitü Yönetim Kurulunun 28 / 05 / 2019 gün ve 1806 sayılı kararıyla oluşturulan jüri tarafından yapılan sınavda YÜKSEK LİSANS TEZİ olarak kabul edilmiştir.

Jüri Üyeleri

Başkan : Prof. Dr. Abdulsamet HAŞILOĞLU .....

Üye : Dr. Öğr. Üye. Hüseyin PEHLİVAN .....

Üye : Dr. Öğr. Üye. İbrahim SAVRAN .....

Prof. Dr. Asim KADIOĞLU
Enstitü Müdürü

ÖNSÖZ

Bilgisayar bilimlerinin gelişimiyle birlikte metinsel verilerin işlenebilmesi önem kazanmaktadır. Bu amaç için kullanılabilecek çeviricilere ihtiyaç günden güne artmaktadır. Yeni ihtiyaçlar, yeni dillerin ve yeni yöntemlerin gelişmesine katkı sağlamaktadır. İhtiyaçlara yönelik yeni dil yapıları geliştirmek üzerinde itinayla çalışılan ve sürekli gelişim sergileyen bir alandır.

Programlama dilleri sayesinde insanların anlayabileceği türden yazılmış kaynak kodlar, özel çeviriciler aracılığıyla bilgisayarın anlayabileceği makine kodlarına dönüştürülür. Bu, program yapmak isteyen kişilerin programlarını daha kolay bir şekilde oluşturabilmesine yardımcı olur. İnsanların anlayabileceği dillerde kaynak kod yazmak, makine dilinde kod yazmakla kıyaslanamaz derecede kolaydır.

Belirsiz integral problemlerinin çözümü için genel bir yorumlayıcının simgesel yaklaşımlar kullanılarak tasarımı ve gerçeklenmesinin yapıldığı bu çalışma, Karadeniz Teknik Üniversitesi Fen Bilimleri Enstitüsü Bilgisayar Mühendisliği Anabilim Dalı'nda Yüksek Lisans tezi olarak hazırlanmıştır.

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

Ceyhan YILMAZ

Trabzon 2019

TEZ ETİK BEYANNAMESİ

Yüksek Lisans Tezi olarak sunduğum Belirsiz İntegral Problemlerinin Çözümü için Genel bir Yorumlayıcının Simgesel Hesaplama Yaklaşımları Kullanılarak Tasarımı ve Gerçeklenmesi başlıklı bu çalışmayı baştan sona kadar danışmanım Dr. Öğr. Üyesi Hüseyin PEHLİVAN'ın sorumluluğunda tamamladığımı, verileri/örnekleri kendim topladığımı, deneyleri/analizleri ilgili laboratuvarlarda yaptığımı/yaptırdığımı, başka kaynaklardan aldığım bilgileri metinde ve kaynakçada eksiksiz olarak gösterdiğimi, çalışma sürecinde bilimsel araştırma ve etik kurallara uygun olarak davrandığımı ve aksinin ortaya çıkması durumunda her türlü yasal sonucu kabul ettiğimi beyan ederim. 18/06/2019

Ceyhan YILMAZ

İÇİNDEKİLER

Sayfa No

Yüksek Lisans Tezi

ÖZET

BELİRSİZ İNTEGRAL PROBLEMLERİNİN ÇÖZÜMÜ İÇİN GENEL BİR YORUMLAYICININ SİMGESEL HESAPLAMA YAKLAŞIMLARI KULLANILARAK TASARIMI VE GERÇEKLENMESİ

Ceyhan YILMAZ

Karadeniz Teknik Üniversitesi
Fen Bilimleri Enstitüsü
Bilgisayar Mühendisliği Anabilim Dalı
Danışman: Dr. Öğr. Üyesi Hüseyin PEHLİVAN
2019, 53 Sayfa

Otomatik kod üretim araçları, sıklıkla kullandığımız C, C++, Java gibi programlama dilleri için derleme ve yorumlama sürecinin bileşenleri olan, analiz ve dönüşüm işlemlerini otomatik olarak yapabilen araçlardır.

Bu çalışmada, otomatik kod üretim araçları yardımıyla simgesel hesaplama yaklaşımları kullanılarak, matematikte ve mühendislik alanlarında önemli bir yer tutan belirsiz integral problemlerinin, çözümünde sonlu sayıda terim içeren türlerini çözebilen bir sistemin nasıl gerçekleştirilebileceği gösterilmiştir. İntegral hesabı yapılırken gereken bütün işlemleri adım adım gösterecek şekilde tasarlanan sistem Ayrıştırıcı, Simgesel Türev Alıcı, Simgesel Sadeleştirici ve Simgesel İntegral Alıcı olmak üzere dört temel bileşen barındırmaktadır.

İntegral ifadeleri üzerinde gerçekleştirilmesi gereken kelimesel çözümleme ve sözdizim analizi ve ayrıştırma işlemleri için kullanılan ayrıştırıcı, JavaCC ayrıştırıcı üreteci yardımıyla otomatik olarak üretilmiştir. Hesaplanacak integral ifadelerinin sözdizimini temsil edecek dilbilgisi kuralları, BNF (Backus Naur Form) notasyonunda tanımlanmıştır. Daha sonra bu kurallar dikkate alınarak, soyut sözdizim ağacını oluşturacak Java dili ifadeleri eklenip JavaCC notasyonuna dönüştürülmüştür. Sözdizim ağacı üzerinde işlem yapan diğer bileşenler, Ziyaretçi Tasarım Deseni yardımıyla tasarlanıp kodlanmıştır.

Anahtar Kelimeler: Simgesel Hesaplama, Ayrıştırıcılar, JavaCC, Formal Gramerler, İntegral

Master Thesis

SUMMARY

DESIGN AND IMPLEMENTATION OF A GENERAL INTERPRETER FOR SOLUTION OF INDEFINITE INTEGRAL PROBLEMS BY USING SYMBOLIC COMPUTATION APPROACHES

Ceyhan YILMAZ

Karadeniz Technical University
The Graduate School of Natural And Applied Sciences
Computer Engineering Graduate Program
Supervisor: Asst. Prof. Dr. Hüseyin PEHLİVAN
2019, 53 Pages

Automated code generation tools are tools that can automatically perform analysis and transformation, which are components of the compilation and interpretation process for frequently used programming languages such as C, C ++, Java.

In this study, it is shown how a system can be used which can solve the types of finite number of terms in the solution of indefinite integral problems which have an important place in mathematics and engineering fields by using symbolic computation approaches with the help of automatic code generation tools. The system, which is designed to show all the necessary operations step by step when performing the integral calculation, has four basic components, namely the Parser, the Symbolic Derivative, the Symbolic Simplification and the Symbolic Integral Solver.

The parser, which is used for lexical analysis and syntax analysis and parsing operations that must be performed on integrals, has been generated automatically with the help of JavaCC parser generator. Grammar rules that represent the syntax of integral expressions to be calculated have been defined in BNF (Backus Naur Form) notation. Then, taking these rules into account, Java language expressions to create the abstract syntax tree have been added and converted into JavaCC notation. Other components processing on the syntax tree have been designed and coded with the help of the Visitor Design Pattern.

Key Words: Symbolic Computation, Parsers, JavaCC, Formal Grammars, Integral

ŞEKİLLER DİZİNİ

Sayfa No

TABLOLAR DİZİNİ

Sayfa No

SEMBOLLER DİZİNİ

AST	Soyut Sözdizim Ağacı (Abstract Syntax Tree)
BNF	Backus-Naur Form
CFG	İçerikten Bağımsız Gramer (Context – Free Grammar)
EBNF	Genişletilmiş (Extended) Backus-Naur Form
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)
SDT	Sözdizim Yönlendirmeli Çeviri (Syntax Directed Translation)

1. GENEL BİLGİLER

1.1. Giriş

Matematiğe ihtiyaç duyulan tüm bilim ve mühendislik alanlarında Lineer denklemler, polinom denklemleri, diferansiyel denklemler, türev ve integral gibi denklemlerin oluşturulması ve çözülmesi gerekir. Bu denklemlerin birçoğunun insan eliyle çözümü uzun, karmaşık ve hata yapmaya müsait işlemler gerektirir. Hatta bu işlemlerin bir kısmı el ile hesaplanamayacak kadar karmaşık olabilir.

Matematiksel hesaplamaların yaklaşık çözümlerini bilgisayarlardan yardım alarak hesaplamak amacıyla sayısal analiz yöntemleri geliştirilmiştir. Ancak bu yöntemler belirli bir hata payı ölçüsünde sonuca yaklaşabilir. Hata payını kontrol altında sayısal analizin ana amaçlarından biridir.

Bilgisayar yardımıyla simgesel hesaplama çalışmaları ilk kez 1953'te başlamıştır, fakat simgesel hesaplama düşüncesi çok daha geçmişe dayanır.

Simgesel hesaplama yöntemleri matematiksel hesapların bilgisayar aracılığıyla tam olarak çözülebilmesi için geliştirilmiştir. Ancak öncelikle üzerinde çalışılacak olan matematiksel problemlerin tam olarak açıklanabilmesi gerekmektedir. Ardından problemlerin çözülebilmesi için faydalanılan yöntemler, bilgisayarlar aracılığı ile hesaplayacak algoritmalar şeklinde ifade edilmesi gerekir. Ancak bu noktada bilgisayarlar bilimlerinde en temel birimler olarak kabul edilen tamsayı ve reel sayıların dahi bilgisayarlar tarafından tam olarak temsil edilemediğini unutmamak gerekir.

Diferansiyel hesabının Newton tarafından geliştirilmesinden sonra, matematiğin en önemli alanlarından biri haline gelmiştir. Temel fonksiyonların türev ve integrallerini hesaplayabilecek becerilere, matematikle ilgilenen herkesin sahip olması gerekir.

Diferansiyel öğrenen herkesin fark ettiği gibi, türev hesabı birkaç temel kural yardımıyla her durumda yapılabilmektedir ancak, belirsiz integrallerin hesaplanabilmesi amacıyla kullanım alanı oldukça kısıtlı çok sayıda farklı yöntem vardır. Sonuca ulaşmak için var olan yöntemlerin hangi kombinasyonla uygulanacağının bilinmesi gerekir. Ayrıca bilinen temel kurallarla çözümü ifade edilemeyen problemler de vardır. Bilinen yöntemlerle çözüme ulaşılamadığında karşılaştığımız problemin çözümünün olup olmadığını kestiremeyebiliriz.

Tanımı gereği, bir algoritmanın uygulanması, uygulayıcının belirtilen işlemleri gerçekleştirebilme becerisinden farklı bir becerisine bağlı olmaması gerekir. Bilgisayarlar aracılığıyla uygulanmak istenen her yöntem için algoritmanın bu özelliğinin gerekliliği açıktır. Aynı zamanda belirsiz integralleri hesaplamak amacıyla geliştirilen bilinen yöntemlerin bir algoritma oluşturmadığı da ortadadır.

Simgesel hesaplamanın faydalarından biri de insan tarafından okunabilirliğidir. Problemin simgesel çözümü, insana sayısal çözümün sunacağı sayılardan daha çok anlam taşır. Ayrıca sayısal değerler gerektiğinde bunlar simgesel çözüm üzerinden üretilebilir.

Bilgisayarlar üzerinde simgesel hesaplamanın yaklaşık 70 yıllık tarihinde birçok problem çözüme kavuşmuş, bunun yanında birçok problem henüz çözülememiş veya algoritmik olarak çözülemezliği tespit edilmiştir. Daha geniş problem gruplarının çözülmesi ve çözümü var olanların geliştirilmesi amacıyla simgesel hesap çalışmalarına ilgi giderek artacaktır.

1.2. Literatür Taraması

Teknolojinin hızla gelişmesiyle birlikte günlük yaşantımızda sıklıkla kullandığımız bilgisayar özellikli cihazlar vücudumuzun bir uzvu gibi yaşantımızın ayrılmaz bir parçası haline gelmiştir.

İnsanlık tarihi boyunca insanlar, karşılaştıkları problemlere daha kolay ve hızlı çözüm bulmak, yaptıkları ve yapacakları eylemlerin sonuçlarını öngörebilmek istemişlerdir. Bu isteklerini gerçekleştirebilmek için bilimi ve özellikle benzetilecek olursa bilimin damarlarında bir kan görevi gören matematiği sıklıkla kullanmışlardır. Günümüzde teknolojinin de hızla gelişmesi hayatımızı her yönüyle etkilediği gibi, matematiksel

problemlerin çözümü de teknolojinin gelişiminden etkilenmiş ve fizikten kimyaya, mühendislikten genetiğe bütün alanlarda karşılaşılan matematiksel problemlerin çözümü bilgisayarlar yardımıyla hızlı ve hatasız bir şekilde yapılabilmektedir.

Yapılan çalışmanın konusu, simgesel hesaplama ile belirsiz integral problemlerinin çözümü, mühendislik ve matematik alanları başta olmak üzere birçok alanda kullanılmaktadır. 1953'ten beri simgesel hesaplama bilgisayarlar üzerinde kullanılmasına rağmen, simgesel hesaplamanın bilim tarihinde kullanımı oldukça geçmişe dayanır [1].

Leibniz'in matematiksel hesaplamalar üzerine yapmış olduğu çalışmalarda, matematiksel ifadeleri ve yöntemleri, algoritmalar ve formüllerle ifade edebilmek için karakteristik simgesel bir dil oluşturmaya çalışmıştır [1].

Bilgisayarların simgesel hesaplamanın yanında sayısal hesaplamalar da yapabileceği, bilgisayar teknolojisinin otomatik hesaplama için geliştirilmesinden daha önce fark edilmiştir.

J. F. Nolan'ın 1953'te yüksek lisans tezi olarak sunduğu çalışması, Massachusetts Teknoloji Üniversitesi'ndeki Whirlwind I üzerinde gerçekleştirdikleri makine türev hesabı, bu alandaki erken dönem çalışmalardan biridir [2].

Yine 1953'te Temple Üniversitesi'nden H. G. Kahrimanian tarafından kaleme alınan makalede türev için ilk bilgisayar programlarından biri tanımlanmıştır [3].

1950'li yılların sonlarına doğru John McCarty tarafından geliştirilen Lisp gibi liste işleme dilleri geliştirilmiştir. Lisp, simgesel hesaplama yöntemlerinin gelişmesinde de hatırı sayılır bir yere sahiptir.

1961'de James Robert Slagle tarafından Massachusetts Teknoloji Enstitüsünde doktora tezi olarak geliştirilen simgesel integral hesaplama programı Lisp ile yazılmıştır [4].

Lisp'in sunmuş olduğu avantajlarla simgesel hesaplamanın bilgisayarlar vasıtasıyla yapılabileceği fark edilmiş ve bilimsel bir alan olarak simgesel hesaplama hızlı bir gelişme sürecine girmiştir.

Genel amaçlı simgesel hesaplama sistemlerinden ilki olan Reduce [5] 1967 yılında, ardından 1971'te Macsyma [6] ve 1971'de de Scratchpad [7] geliştirilmiştir.

1969'da Robert H. Risch tarafından kaleme alınan makalede daha sonra Risch algoritması olarak adlandırılacak olan belirsiz integral probleminin çözümüne yönelik algoritmayı trigonometrik, üstel, logaritmik fonksiyonlar gibi genel fonksiyonlar için

geliştirmiştir [8]. Günümüzde halen Risch algoritması kapsamı genişletilip geliştirilmeye devam etmektedir.

1967'de E. R. Berlekamp tarafından yayınlanan makalede, sonlu alanlar üzerindeki polinomların çarpanlarına ayrılabilmesi için bir algoritma sunulmuştur [9].

David R. Musser tarafından 1974'te sunulan makalede, bir veya çok sayıda değişkene sahip, katsayıları tamsayı olan polinomların çarpanlara ayrılma algoritması açıklanmıştır [10]. Geliştirilen algoritma Hensel'in Lemma yapıları ve sonlu alanlar üzerinde çarpanlara ayırma yöntemlerini temel almaktadır.

1975 yılında yayınlanan makalelerinde Paul S. Wang ve Linda P. Rothschild, Berlekamp'ın geliştirdiği tek değişkenli polinomları çarpanlarına ayırma algoritmasını temel alıp, çok değişkenli tamsayı katsayılı polinomların çarpanlarına ayrılmasını sağlayan bir algoritma açıklamışlardır [11]. Bu algoritmanın hem tek değişkenli hem çok değişkenli polinomlar üzerine uygulandığında daha önce geliştirilmiş algoritmalardan çok daha hızlı olduğunu göstermişlerdir.

Jacques Carette 2004 yılında yapmış olduğu çalışmada, genel ifadeler için sadeleştirme kavramının ilk resmi tanımını verdiklerini belirtmişlerdir [12]. Bu çalışmada, sadeleştirme kavramının tanımı genel olarak verilmeye çalışılmıştır.

Mohammed Shatnawi Matematiksel ifade arama için ayrıştırma-ağaç normalizasyonu kullanılarak denklik kontrolünün yapılmasını 2017 yılında sunmuş olduğu makalede açıklamıştır [13]. Matematiksel ifadelerin birden çok birbirine denk ifade ile temsil edilebilmesi sebebiyle yapılan çalışmada, matematiksel ifadeler ayrıştırma ağaçları üzerinde normalizasyon işlemine tabi tutularak özdeş yapılar denkleştirilmeye çalışılmıştır. Bu çalışmada ifadeler JavaCC ayrıştırma üreteci kullanılarak ayrıştırılmıştır.

Rohit Singh ve arkadaşları 2012'de otomatik olarak cebirsel problem üretme metodolojisini yayınladıkları makalede açıklamışlardır [14].

Yavuz Tekbaş'ın 2013'te yüksek lisans tezi olarak sunduğu çalışmada, simgesel hesaplama yaklaşımları kullanılarak matematiksel ifadelerin türevlerinin simgesel olarak hesaplanması ve sadeleştirilmesi açıklanmıştır [15]. Bu çalışmada programlama dili olarak Java, ifadelerin ayrıştırılması için de otomatik ayrıştırıcı üreteci olan JavaCC kullanılmıştır.

2014 yılında Mir Mohammad Reza Alavi Milani, Hüseyin Pehlivan ve Sahereh Hossainpour, Taylor ve Maclaurin serilerinin derleyici derleyici araçları kullanılarak genişletilmesini açıklayan bir makale sunmuşlardır [16].

Richard Fateman 2015 yılında yayınlamış olduğu makalede, Lisp dilinde algoritmik türev alma uygulaması geliştirilmesini açıklamışlardır [17]. ADIL olarak adlandırdıkları uygulama türev alma işlemini otomatik olarak gerçekleştirebilmektedir.

2015 yılında sunmuş olduğu doktora tez çalışmasında Mir Mohammad Reza Alavi Milani matematiksel ifadelerin adım adım değerlendirilebilmesi için genel bir yapı oluşturmaya çalışmıştır [18]. Çalışmada matematiksel ifadeleri temsil edebilecek genel bir gramer oluşturulmaya çalışılmıştır.

Baki Gökgöz 2016 yılında yüksek lisans tezi olarak yapmış olduğu çalışmada sayısal kök bulma yöntemleri için simgesel yaklaşımlar yardımıyla bir yorumlayıcı tasarlayıp gerçekleştirmiştir [19].

Mohammed Yusuf Hassan 2017 de sunmuş olduğu yüksek lisans tezinde simgesel yaklaşımlar kullanılarak lineer olmayan denklem sistemlerinin adım adım çözümü için genel bir yorumlayıcının tasarımını açıklamıştır [20].

2017 yılında Seda Efendioğlu yüksek lisans tez çalışmasında polinomlar için genel bir simgesel hesaplama çatısı tasarlayıp gerçekleştirmiştir [21].

2017'de Sahereh Hossainpour ve Mir Mohammad Reza Alavi Milani yayınladıkları makalede matematiksel ifadelerin üretimi için gramer tabanlı bir metodoloji sunmuşlardır. Yine 2018'de sundukları makalelerde matematiksel ifadeler için adım adım çözüm ve matematiksel ifadeler için kural tabanlı üretim metodolojilerini açıklamışlardır.

2018 yılında Nawal Abdullahi Mohamed'in sunduğu yüksek lisans tezinde en küçük kareler metodu için simgesel yaklaşımlar kullanarak bir yorumlayıcı geliştirmiştir [22].

2019'da yayınlamış olan makalesinde Hüseyin Pehlivan, nümerik analiz yöntemlerinin programlanıp yorumlanmasını sağlayan bir programlama dilinin tasarımını gerçekleştirmiş ve bu dilin kaynak kodunu yorumlayabilen bir yorumlayıcı geliştirmiştir [23].

İnsanlık tarihinde bilimin gelişmesi matematiğin etkin kullanımıyla hızlanmıştır. Hayatımızın hemen her alanına girmiş olan matematik, bilgisayar teknolojisinin ilk ortaya çıkışından itibaren, bu yeni teknolojiyle birlikte ayrılmaz bir bütün haline gelmiştir. Bilim dünyasında ve günlük hayatımızda karşılaştığımız birçok problemi matematik sayesinde çözmemiz mümkün olmaktadır, ancak bazı problemler el yordamıyla çözüm için oldukça zor olabilmektedir. Böyle durumlarda bilgisayar teknolojisinden faydalanarak karmaşık

problemlerin hızlı ve etkin çözümü sağlanabilmektedir. Matematiğin karmaşık konularından biri olan integral hesabı için de bilgisayar teknolojisinden faydalanılabilir.

İntegral genel olarak;

- Eğriler arasında kalan alanı hesaplama,
- Eğri uzunluğunu hesaplama,
- Bir cismin hacmini hesaplama,
- Bir fonksiyonun ortalama değerini hesaplama

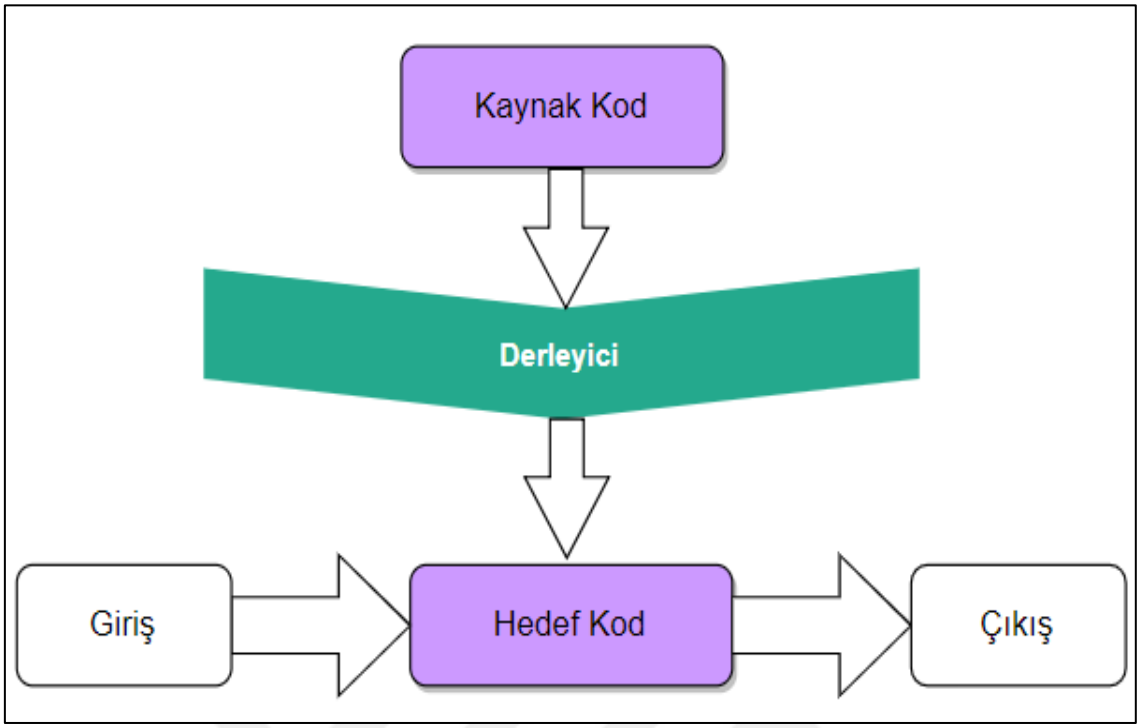
gibi hayatımızda sıklıkla karşılaştığımız problemlerin çözümünde kullanılır. Geliştirilen uygulamada integral problemlerinin çözümü için genel bir yapı oluşturulmuştur ve yukarıdaki gibi integral hesabının gerekli olduğu mühendislik uygulamaları ve matematiksel işlemlerde, problemleri çözmek için kullanılabilir.

1.3. Dil Çeviricileri

Dil çeviricilerinin genel özellikleri bu bölümde detaylı olarak incelenmiştir. Dil çeviri araçlarından derleyici ve yorumlayıcıların çalışma prensipleri anlatılmış ve birbirlerine göre üstünlük ve zayıflıkları gösterilmiştir.

1.3.1. Derleyiciler

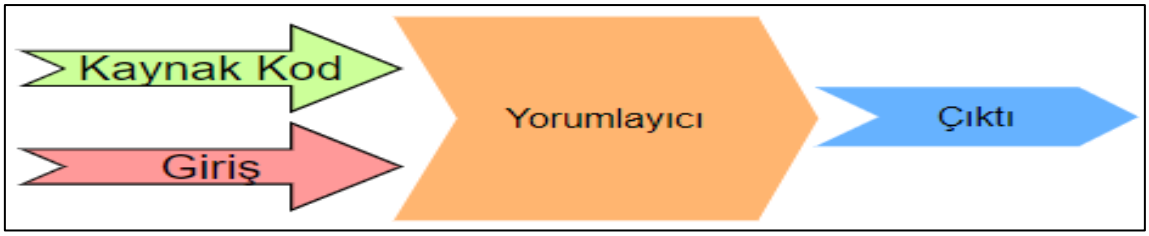
Derleyiciler yüksek seviyeli kaynak dilde yazılmış bir program kodunu, fonksiyonel olarak denk olan düşük seviyeli bir dile, genellikle makine diline, dönüştüren programladır. Yani derleyici bir bilgisayarda yüksek seviyeli bir dil uygulanabilmesi için kullanılan çevirici türlerinden biridir [24]. Kaynak dilde yazılmış olan program kodu bir kez derlenir ve sonucunda makinenin anlayacağı bir dile dönüştürülmüş çalıştırılabilir kod üretilmiş olur. Bu çalıştırılabilir kod, daha sonra hiçbir işlem gerektirmeksizin çalıştırılıp kullanılabilir. Ancak kaynak dilde yazılmış kod üzerinde değişiklik yapıldığında tüm kodun yeniden derlenmesi gerekmektedir. Şekil 1‘de derleyicinin çalışma prensibi gösterilmiştir.



Şekil 1. Derleyicinin Çalışma Prensibi

1.3.2. Yorumlayıcılar

Yorumlayıcı, kaynak dilde yazılmış program kodunu ilk satırından itibaren sırasıyla çevirir ve kodların yapması gereken eylemleri gerçekleştirir. Bu işlem esnasında herhangi bir dönüştürme işlemi gerçekleştirilmez ve çalıştırılabilir bir kod üretilmez. Bu sebeple kaynak dilde yazılmış program kodu her çalıştırıldığında yine bu kaynak dilde yazılmış program kodu üzerinden yorumlama işlemi gerçekleştirilir. Bu yorumlama işlemi sırasında yorumlayıcının da belleğe yüklenmesi gerekmektedir [25]. Şekil 2'de genel bir yorumlayıcının çalışma prensibi gösterilmiştir.



Şekil 2. Yorumlayıcının Çalışma Prensibi

1.3.3. Derleyici ile Yorumlayıcının Karşılaştırılması

Genel anlamda bakılacak olursa dil çeviricileri benzer işlevlere sahiptir. İster derleyici ister yorumlayıcı olsun her ikisi de yüksek seviyeli bir dilde yazılmış olan kaynak program kodunu makinenin anlayabileceği bir dile dönüştürme işlemi icra ederler. Ancak bu dönüştürme işlemini gerçekleştirirken bazı farklılıklar göstermektedirler.

Derleyiciler, kendisine verilen kaynak dildeki program kodunu alıp, bu kodun tamamını makinenin anlayacağı dilde bir koda dönüştürür. Daha sonra program çalıştırılmak istendiğinde Yeniden herhangi bir dönüştürme işlemine gerek kalmaksızın sadece derleyici tarafından dönüştürülmüş olan program kodu çalıştırılabilir. Ancak kaynak program kodunda bir değişiklik yapılırsa, tüm derleme işlemi yeniden icra edilmelidir.

Yorumlayıcılar, verilen kaynak program kodunu baştan aşağı satır satır çevirme işlemine tabi tutar ve dönüştürülen bu kodları çalıştırır. Bu aşamada herhangi bir dönüştürülmüş hedef kod oluşturulmadığı için program her çalıştırılmak istenildiğinde aynı dönüştürme işlemi gerçekleştirilmelidir.

Derleyiciler ve yorumlayıcıların farklarından bir kısmı aşağıdaki şekilde tanımlanabilir [26]:

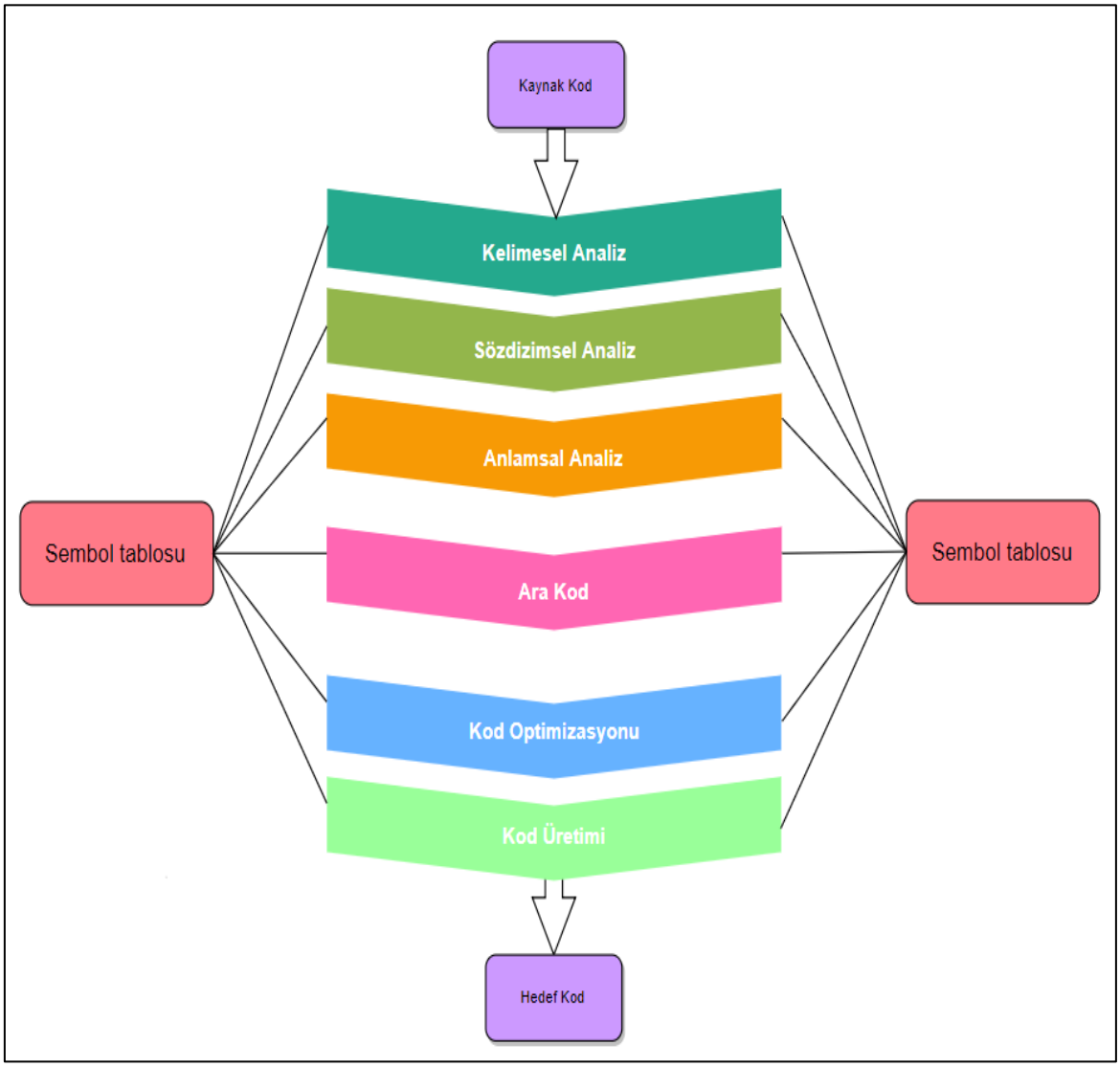
- Kaynak program kodu yorumlayıcı ile icra edilmek istendiğinde, her icra esnasında yorumlayıcı da belleğe yüklenmelidir. Bu durum bellek kullanımının verimsizleşmesine yol açabilir. Kaynak program kodu derleyici ile icra edilirse, derleyici sadece derleme esnasında bellekte yer kaplar, programın sonraki çalıştırılma zamanlarında derleyicinin yeniden belleğe yüklenmesi gerekmez.
- Derlenerek dönüştürülmüş programlar, yorumlanarak dönüştürülen programlara göre daha hızlı çalışırlar.
- Yorumlayıcılar, derleyicilere göre hata tespitinde biraz daha iyidirler. Derleme esnasında tespit edilemeyen hatalar yorumlama esnasında program kodu satır satır icra edildiği için anında tespit edilebilir.
- Kaynak program kodunda değişiklik yapmak yorumlayıcı dillerde derleyici dillere göre daha kolay ve hızlıdır.

1.4. Çeviricilerin Genel Yapısı ve Bileşenleri

Çeviriciler tipik olarak kendilerine verilen herhangi bir dilde yazılmış olan kaynak dildeki program kodunu işlemlere tabi tutarak kendisine denk olan başka bir dildeki program kodlarına dönüştüren araçlardır. Bu işlemler sırasında varsa kod hataları raporlanır ve ayıklanır, ardından kaynak dilde yazılmış program kodu düşük seviyeli makinenin anlayacağı bir dile, genellikle makine diline çevrilir. Bir çevirici genel olarak şu aşamalardan oluşur.

- Ön Uç aşamaları
    - Kelimesel analiz aşaması
    - Sözdizimsel analiz aşaması
    - Anlamsal analiz aşaması
- Ara kod üretim aşaması
- Arka Uç Aşamaları
    - Kod optimizasyon aşaması
    - Kod üretim aşaması

Aşağıda Şekil 3‘te genel bir çeviricinin genel yapısı ve aşamaları gösterilmiştir [27].



Şekil 3. Çeviricinin Genel Yapısı ve Aşamaları

1.4.1. Çevirici Aşamaları

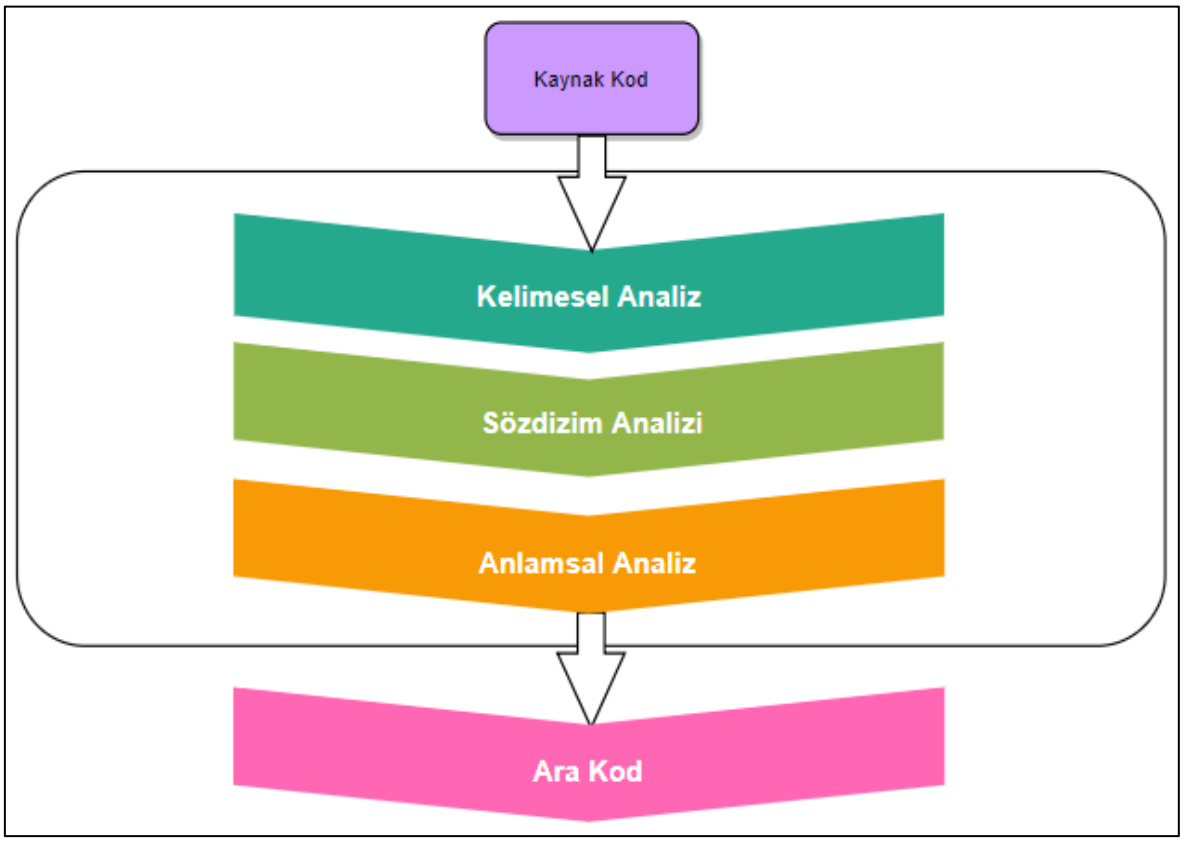
Çeviriciler ön-uç ve arka-uç olmak üzere iki kısımdan oluşmaktadır. Ön-uç kendisine girdi olarak verilen kaynak program kodunu hedef makineden bağımsız olarak analiz edildiği kısımdır. Bu analiz sonucunda bir ara kod üretilir. Üretilen bu ara kod arka-uç tarafından hedef makinenin mimarisine uygun olacak şekilde düşük seviyeli bir dile, genellikle makine diline dönüştürülür.

1.4.1.1. Ön-Uç (Front-End)

Ön-uç aşamaları, kaynak program kodunun alınıp ara kod üretimine kadar olan kısımdaki analiz işlemlerinden oluşmaktadır. Bu aşamalar:

- Kelimesel Analiz Aşaması
- Sözdizimsel Analiz Aşaması
- Anlamsal Analiz Aşaması

olarak üç ana başlıkta toplanabilir. Şekil 4'te ön uç aşamaları gösterilmiştir.



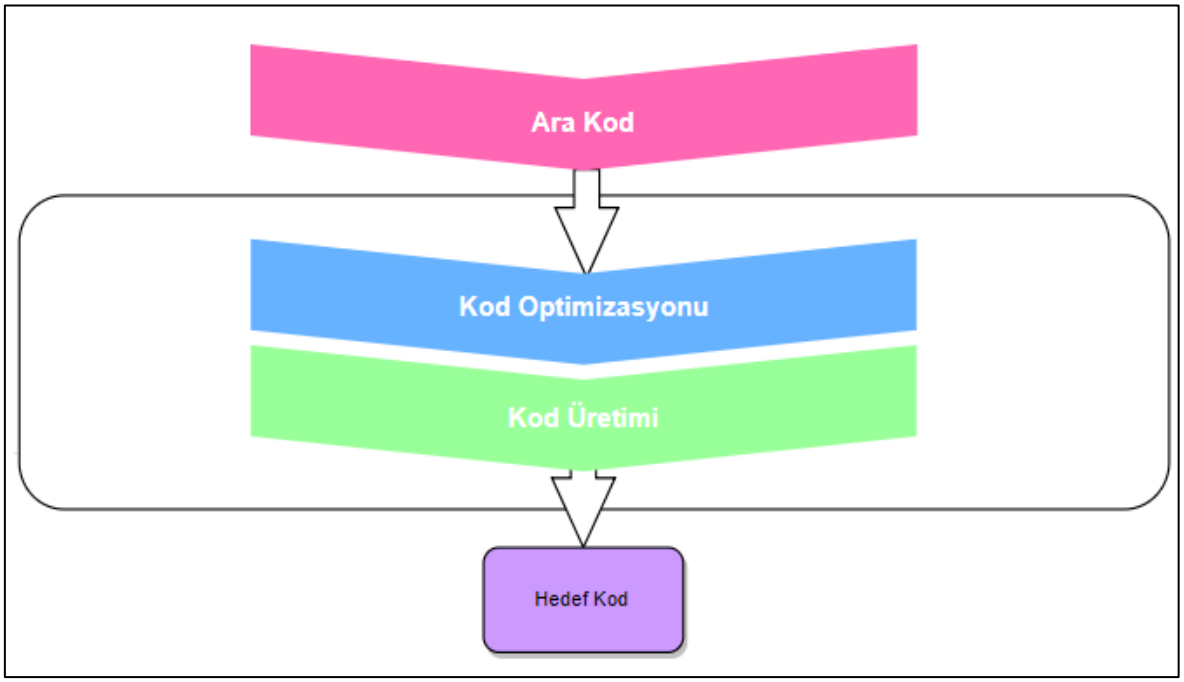
Şekil 4. Ön-Uç Aşamaları

1.4.1.2. Arka-Uç (Back-End)

Arka-uç temel olarak iki aşamadan oluşur. Bu aşamalardan ilki kod optimizasyonu aşamasıdır. Bu aşamada ara koddaki yapılara denk emirler en yüksek performans sağlanacak

şekilde assembly dilinde üretilir. Bu işlem makine mimarisine bağımlıdır. Daha önce üretilmiş soyut sözdizim ağacı için en performanslı emirleri üretmek kolay bir problem değildir.

İkinci aşama olan kod üretimi aşamasında, kod optimizasyonu aşamasında sonuç olarak üretilen emir dizisi birleştiriciye (Assembler) gönderilir. Birleştirici kendisine verilen kodu makinenin anlayabileceği makine diline dönüştürerek çalıştırılabilir bir forma kavuşturur. Arka-uç aşamaları Şekil 5'de gösterilmiştir.



Şekil 5. Arka-Uç Aşamaları

1.4.2. Kelimesel Analiz

Kelimesel analiz aşamasında, kaynak program kodunda bulunan henüz anlamlı olmayan karakterler dizisini, belirteçler (token) dizisine dönüştürme aşamasıdır.

Kelimesel çözümleyiciler yardımıyla, girdi olarak alınan karakterler dizisi, isimler, anahtar kelimeler dizisi. gibi daha önceden belirlenmiş olan yapılara dönüştürülür. Analizin bu aşamasında eğer tanımlanmışsa; açıklama satırı, özel karakterler gibi derleme işlemine tabi tutulmayacak yapılar varsa göz ardı edilebilir.

Kelimesel analiz sürecinde, kaynak program kodunda bulunan ve belirlenen dile uygun olmayan ifadeler varsa tespit edilir ve hata mesajı olarak gösterilir. Kaynak kodda herhangi bir hataya rastlanması durumunda bir sonraki aşamaya geçilmez.

Kelimesel analiz işleminin gerçekleştirilebilmesi için düzenli ifadelerden faydalanılır.

1.4.2.1. Düzenli İfadeler

Düzenli ifadeler, güçlü esnek ve verimli metin işlemenin anahtarıdır. Neredeyse bir programlama dili gibi genel bir desen notasyonu ile metni tanımanıza ve ayrıştırmanıza izin verir [28]. Özel araçlar tarafından sağlanan ek destekler kullanarak her türlü metni ve veriyi ekleyebilir, silebilir, izole edebilir ve işleyebilir.

Düzenli ifadeler tanımlarken ön tanımlı bazı terimler kullanılır. Kullandığımız bazı düzenli ifade terimleri şunlardır [29]:

- `|` işareti : Seçim işareti ifadelerden birinin seçileceğini belirtir. Örneğin; (x | y) ifadesinde ‘x’ veya ‘y’ ile eşleme yapılır
- `·` işareti : Birleştirme işareti ile ifadeler birleştirilir. Örneğin; (x | y) · z ifadesinde ‘xz’ veya ‘yz’ ifadeleri ile eşleme yapılır.
- `ε` işareti : Epsilon işareti kullanılarak boş değer gösterimi gerçekleştirilir.

Örneğin; (x | ε) ifadesi x veya boş değer ile eşleştirme yapar.

Bu ifadelere ek olarak tekrarlama işaretleri de düzenli ifadelerde sıklıkla kullanılır. Başlıca tekrarlama terimleri:

- `*` işareti : Bir ifadenin hiç bulunmayabileceğini veya herhangi bir miktarda tekrarlanabileceğini ifade eder.
- `+` işareti : Bir ifadenin en az bir veya daha fazla tekrarlanabileceğini ifade eder.
- `?` işareti : Bir ifadenin hiç bulunmayabileceğini veya yalnız bir kez bulunabileceğini ifade eder.

Bu terimler yardımıyla düzenli ifadeler kullanılarak geliştirilecek programlama diline ait kelimesel yapı belirlenir. Programlama dillerinde sıklıkla kullanılan bazı ifadeler düzenli ifade biçiminde aşağıdaki gibi gösterilebilir:

```
IF          : `if`
WHILE       : `while`
DEĞİŞKEN    : [a-z,0-9]*
TAMSAYI     : [0-9]+
REELSAYI    : ([0-9]+`.`[0-9]*)|([0-9]*`.`[0-9]+)
```

Yukarıdaki ifade ile belirtilen dilde istenmeyen bazı belirsizlikler oluşabilmektedir. Örneğin; `whiles` belirteci için `WHILE` ve `DEĞİŞKEN` ya da sadece `DEĞİŞKEN` olarak algılanması konusunda belirsizlik meydana gelebilmektedir. Belirsizlik durumundan kurtulmak için iki temel kural geliştirilmiştir.

- En Uzun Eşleşme Kuralı : En uzun eşleşme kuralına göre, ifade içindeki en uzun eşleşmeye öncelik verilir, yani ifade ile en uzun eşleşen ifade öncelikli olarak seçilir. Bu kurala göre `whiles` belirteci `DEĞİŞKEN` olarak seçilir.
- Kural Önceliği Kuralı : Kural önceliği kuralına göre, düzenli ifade tanımlama sırasına göre ilk karşılaşılan ifade öncelikli seçilir. Bu kurala göre `whiles` belirteci `WHILE` ve `DEĞİŞKEN` olarak algılanabilir.

Kelimesel analiz üreteçleri bu iki kuraldan sadece bir tanesini kullanırlar.

1.4.2.2. Kelimesel Analiz Üreteçleri

Kelimesel analiz üreteçleri genellikle BNF notasyonunda tanımlanmış içerikten bağımsız gramerler ile ifade edilen yapıları ayrıştırmak için kullanılan araçlardır. Bu araçlar aldıkları ham kelimeler topluluğunu işleyerek belirteçler (Token) topluluğuna dönüştürür. Bu amaçla geliştirilmiş farklı dilleri destekleyen Jlex [30], JavaCC [31], Yacc [27], SableCC [32], ANTLR [33] gibi birçok farklı kelimesel analiz üreteçleri geliştirilmiştir.

1.4.3. Sözdizimsel Analiz (Ayrıştırma)

Çözümleyicilerin ikinci aşaması sözdizim analizi aşamasıdır. Bu aşama temel olarak üş işlemden oluşur. Bunlar:

- Dilin sözdizim kurallarının belirlenmesi

- Kaynak program kodunun tanımlanan dilin sözdizim yapısına uygunluğunun kontrol edilip varsa hataların belirlenmesi
- Kelimesel analiz sürecinden gelen belirteç dizisinin derleme aşamasına uygun bir veri yapısına dönüştürülmesidir.

Şekil 6'da Sözdizim analizinin yapısı gösterilmiştir.

Şekil 6. Sözdizim Analizi

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

Programlama dili belirli kelimeler topluluğundan oluşur. Bu dilde kullanılan her bir terim daha önceden belirlenmiş bir alfabede yer alan simgelerin sonlu sayıdaki dizisidir. İçerikten bağımsız gramer (CFG), programlama dilinin sözdizimsel yapısını tanımlayan karakter dizileri oluşturmak için kullanılan özyinelemeli yeniden yazma veya oluşturma kurallarıdır [34]. CFG'ler dört kavram üzerine inşa edilir:

- Sonlu Terimler (Terminaller) dizisi : Kelimesel çözümleyiciler aracılığıyla oluşturulan terimlerden oluşur. Alfabedeki simgeler kullanılarak oluşturulan kelimeleri belirtir.
- Devamlı Terimler (Nonterminaller) dizisi : Kuralın sol kısmında kalan yapıyı ifade eder. Sonlu terimlere veya devamlı terimlere geçiş yapabilen ifadelerdir.
- Başlangıç Simgesi : Grameri başlatmaya yarayan devamlı terimi belirtir.
- Üretim dizisi : Sonlu ve devamlı terimlerden oluşan ve dilin gramer kurallarını belirleyen yapıdır. Ayrıştırma ağaçlarının oluşturulmasında da bu yapılar kullanılır.

Üretim kuralının sağ kısmında bir veya daha fazla sonlu veya devamlı terim bulunmalıdır [35].

1.4.4. Sözdizim Yönlendirmeli Çeviri (Syntax Directed Translation - SDT)

Sözdizim yönlendirmeli çeviri (SDT), kaynak program kodunun tamamen ayrıştırıcı tarafından yürütüldüğü, yani dilin sözdizimini temel alan bir derleyici uygulama yöntemini ifade eder. SDT için yaygın bir yöntem eylemi gramer kuralına ekleyerek bir karakter dizisini bir eylem dizisine çevirmektir. Böylece gramerin bir karakter dizisini ayrıştırarak bir dizi kural oluşturmaktadır. SDT, herhangi bir sözdizimine anlam yüklemek için basit bir yol sunar [36].

Hemen hemen tüm modern çeviriciler sözdizim yönlendirmelidir. SDT'ye genel yaklaşım bir ayrıştırma ağacı veya sözdizim ağacı oluşturmak ve ağacın düğümlerindeki özniteliklerin değerlerini belirli bir sırada ziyaret ederek hesaplamaktır.

1.4.5. Türetim

Türetim metotları sağa dayalı ve sola dayalı türetim olmak üzere ikiye ayrılır.

- Sola dayalı türetimde ilk olarak en solda bulunan devamlı terim genişletilmeye çalışılır.

Örneğin, verilen gramere göre a ∗ (a + b00) ifadesinin sola dayalı türetim adımları aşağıdaki gibi olmaktadır:

E → I | E + E | E * E | (E)
I → a | b | Ia | Ib | I0 | I1

1.	E	7.	a * (I + E)
2.	E * E	8.	a * (a + E)
3.	I * E	9.	a * (a + I)
4.	a * E	10.	a * (a + I0)
5.	a * (E)	11.	a * (a + I00)
6.	a * (E + E)	12.	a * (a + b00)

- Sağa dayalı türetimde ise ilk olarak en sağda bulunan devamlı terimler genişletilmeye çalışılır.

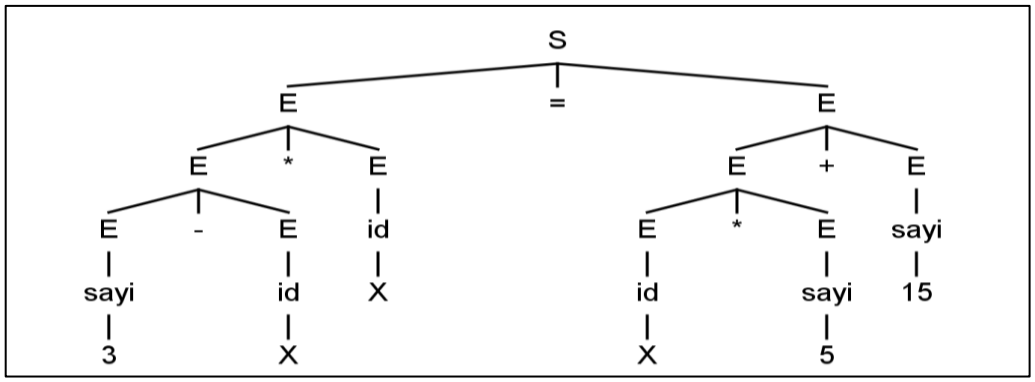
Örneğin yukarıda verilmiş olan gramer ve örnek için sağa dayalı türetim adımları aşağıdaki gibi olmaktadır.

1.	E	7.	E * (E + I00)
2.	E * E	8.	E * (E + b00)
3.	E * (E)	9.	E * (I + b00)
4.	E * (E + E)	10.	E * (a + b00)
5.	E * (E + I)	11.	I * (a + b00)
6.	E * (E + I0)	12.	a * (a + b00)

1.4.6. Ayrıştırma Ağaçları

Ayrıştırma ağacı (Parse Tree), terimler topluluğunun belirlenmiş gramer kurallarına göre ayrıştırılıp ağaç veri yapısı şeklinde temsil edilmesidir. Ayrıştırma ağaçlarındaki iç düğümler devamlı terimleri, yapraklar ise sonlu terimleri temsil eder.

Aşağıda Şekil 7'de örnek bir ayrıştırma ağacının yapısı gösterilmiştir.



Şekil 7. Ayrıştırma Ağacı Örneği

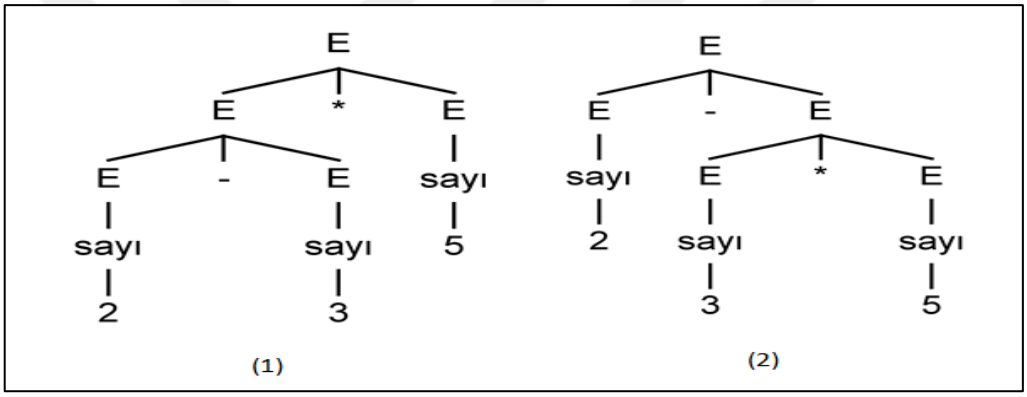
1.4.6.1. Belirsiz Gramer

Analiz aşamasında, aynı ifade için birden çok ayrıştırma ağacı oluşturulabiliyorsa, bu tür gramerlere belirsiz ya da çok anlamlı gramer denilir [37]. Bu tür gramerler, derleme esnasında istenmeyen durumlara yol açabilirler. Oluşturulabilecek birbirinden farklı

ayrıştırma ağaçlarında yapılacak olan işlemler, istenmeyen sonuçların üretilmesine sebep olmaktadır. Bu tür durumlarla karşılaşmamak için geliştirilen gramerlerin belirsizlik durumlarından arındırılması gerekmektedir. Örneğin aşağıdaki gramer verilmiş olsun.

E ::= E + E | E – E | E * E | E / E | sayı
sayı ::= [“0”-“9”]+

Verilen bu gramera uygun olarak 2-3*5 ifadesini tanımlamak için üretilen ayrıştırma ağaçları Şekil 8‘de gösterilmiştir. Bu ifade için iki farklı ayrıştırma ağacı üretilebildiğinden bu gramer belirsiz bir gramerdir.



Şekil 8. 2-3*5 İfadesi İçin Ayrıştırma Ağaçları

Geliştirilmiş bu gramer ile 2-3*5 ifadesi Şekil 8’deki (1) numaralı ayrıştırma ağacı üzerinden değerlendirilirse sonuç (-5), (2) numaralı ayrıştırma ağacı üzerinden değerlendirilirse sonuç (-13) olmaktadır. Bu şekilde istenmeyen durumların oluşmasını engellemek için gramerin işlem önceliklerini belirleyecek şekilde yeniden tasarlanması gerekmektedir. Yukarıda tanımlamış olduğumuz gramerin işlem öncelikleri belirlenerek belirsizlikten kurtarılıp yeniden tasarlanmış hali aşağıdaki gibi ifade edilebilir.

E ::= E + T | E – T | T
T ::= T * C | T / C | C
C ::= sayı
sayı ::= [“0”-“9”]+

1.5. Ayrıştırma

Ayrıştırıcılar belirteç dizisi üzerinde gramer kurallarına bağlı kalarak ayrıştırma işlemi yaparlar. Ayrıştırma işlemini yapmak için birçok ayrıştırma algoritması geliştirilmiştir. Bu bölümde ayrıştırma algoritmalarının bir bölümü tanıtılmıştır.

1.5.1. Ayrıştırma Algoritmaları

Ayrıştırma algoritmaları kendi aralarında genel olarak yukarıdan-aşağıya (Top-down) ve aşağıdan-yukarıya (Bottom-up) ayrıştırma olarak 2 gruba ayrılır. Her ayrıştırma grubu kendi içinde farklı ayrıştırma mekanizmalarına sahip yöntemleri barındırmaktadır.

1.5.1.1. Top-Down Ayrıştırma

Yukarıdan aşağıya ayrıştırıcı, ayrıştırma ağacını kökten başlayarak yapraklara doğru gezer ve oluşturur. Her düğüm dallarından önce ziyaret edilir. Belirli bir düğümden gelen dallar soldan sağa doğru sırayla takip edilir[38]. Bu soldan türetime karşılık gelir. Türetme açısından yukarıdan aşağıya bir çözümleyici şu şekilde tarif edilebilir:

Soldan türetimin bir parçası olan cümlesel bir form verildiğinde, ayrıştırıcının görevi soldan türetmedeki bir sonraki cümlesel formu bulmaktır. Soldaki cümle biçiminin genel formu xAα şeklindedir. Burada x bir karakter dizisini, A nonterminali ve α ise herhangi bir karakter dizisidir. x sadece terminalleri içerdiği için, A, cümle biçiminde en solda olmayandır. Bu nedenle bir sonraki cümle biçimini soldan türetmede almak için genişletilmesi gereken terimdir.

1.5.1.1.1. Aşağıya Özyinelemeli (Recursive Descent) Ayrıştırma

Aşağı özyinelemeli ayrıştırma adını birçoğu özyinelemeli olan ve aşağıdan yukarıya doğru bir ayrıştırma ağacı üreten bir altprogramlar koleksiyonundan oluştuğu için almıştır[38]. Bu özyineleme, birkaç farklı iç içe yapı türünü barındıran programlama dillerinin doğasının bir yansımasıdır. Örneğin, ifadeler genellikle başka ifadelerle iç içe

girmiştir. Ayrıca ifadelerdeki parantezler düzgün bir şekilde iç içe geçmesi gerekir. Bu tür ifadelerin sözdizimi özyinelemeli dilbilgisi kuralları ile doğal olarak açıklanabilmektedir.

EBNF notasyonu aşağı özyinelemeli ayrıştırıcılar için idealdir. İçerdikleri ifadenin bir veya daha fazla kez tekrarlanabileceğini belirten köşeli parantezler ve içlerindeki ifadenin bir kez veya hiç görünmeyebileceğini belirten parantezler EBNF'in birincil eklentileridir.

Aşağı özyinelemeli ayrıştırıcı, gramerdeki her bir nonterminal için bir altprograma sahiptir. Belirli bir nonterminal ile ilişkili altprogramın sorumluluğu aşağıdaki gibidir:

Ayrıştırıcının girişine bir karakter dizisi verildiğinde, bu nonterminalde köklenebilen ve yaprakları girişteki karakter dizisine uyan ayrıştırma ağacını izler. Aslında aşağı özyinelemeli ayrıştırma altprogramı ilişkili nonterminal tarafından oluşturulan dil için ayrıştırıcıdır.

1.5.1.1.2. LL Ayrıştırma

Ayrıştırma tablosunda tekrarlama içermeyen gramerler LL(k) olarak adlandırılır. LL(k) algoritması ile ayrıştırma ve türetim işlemleri soldan sağa doğru gerçekleştirilmektedir. LL(k) algoritması ile ayrılabilen gramerlere da LL gramer adı verilir. İçeriğindeki k'nın anlamı ise, ayrıştırıcının kaç belirtece bakıp karar vermesi gerektiğini belirtir.

LL Ayrıştırma:

- Giriş verisini tutabilecek bir arabelleğe
- Terminal ve nonterminalleri tutacak bir yığına
- Girdi ve yığın verilerinden hangi gramer kuralının uygulandığını gösteren ayrıştırma tablosu barındırır.

Ayrıştırma tablosundaki kurallardan, giriş verisindeki sembollere en uygun olanını tutarak yığına koyar. Giriş verisinin sonuna ulaşılıncaya kadar bu işlem tekrarlanır.

Ayrıştırıcı başlangıçta S başlangıç sembolüne ve $ terminaline sahiptir. Bu özel terminal yığının başını ve giriş cümlesinin sonunu temsil eder.

Basit bir gramer için LL ayrıştırıcının oluşturduğu ayrıştırma tablosu Tablo 1'de gösterilmiştir.

Girdi verisi : (a+a)

Gramer

1. S ::= F
2. S ::= (S+F)
3. F ::= a

Tablo 1. Ayrıştırma Tablosu

	(	**)**	**a**	**+**	**$**
S	2	-	1	-	-
F	-	-	3	-	-

Ayrıştırma işlemi başlamadan yığın [S, $] şeklindedir. Ayrıştırıcı girdi verisinden ilk "(" kelimesini ve yığından S değerini okuyarak işleme başlar. Ayrıştırma tablosuna bakarak bu kelimenin 2 nolu kurala ait olduğunu tespit eder. Kural numarasını geriye döndürür ve yığını günceller.

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

Sonra "(" kelimesini yığından siler.

[S, +, F,), $]

Ayrıştırıcı sonraki kelimeyi belirler, bu kelimenin önce 1 numaralı kural ile ardından 3 numaralı kural ile eşleştiğini tespit eder. Bu işlem sonucunda yığın şu şekilde güncellenir.

[F, +, F,), $]
[a, +, F,), $]

Ardından ayrıştırıcı "a" ve "+" kelimelerini yığından ve girdi verisinden siler.

[F,), $]

Son olarak F ve “)” sembollerini yığından atarak işlemi sonlandırır. Yığında sadece $ sembolünün kalması girdi verisinin sonuna gelindiğini belirtir ve girdinin gramere uygun olduğunu gösterir. Bu ayrıştırma işleminden sonra geriye kural dizisi döndürülür. Örneğimiz için bu dizi [2,1,3,3] şeklindedir.

S ::= (S+F) ::= (F+F) ::= (a+F) ::= (a+a)

Bir gramerin LL(k) olması için sahip olması gereken özellikler vardır.

- Gramer soldan özyinelemeli olmamalı. Soldan yineleme varsa, bu yinelemeden sakınmak için kural sağdan yinelemeli olacak şekilde yeniden düzenlenir.

S :: S*E | E
S ::= ES’
S’ ::= *ES’ | ‘ ’

- Soldan Faktörleme, birden fazla kuralın aynı ifade ile başlaması LL(k) gramerinde sorun teşkil eder. Bu durumu ortadan kaldırmak için soldan faktörleme işlemi yapılır. Örneğin:

S ::= if E then S else S
S ::= if E then S

kuralı için soldan faktörleme yapılırsa:

S ::= if E then S X
X ::= else S | ‘ ‘

şeklinde düzenlenir.

1.5.1.2. Bottom-Up Ayrıştırma

Aşağıdan yukarıya ayrıştırıcı, yapraklardan başlayıp köke doğru ilerleyerek bir ayrıştırma ağacı oluşturur [38]. Bu ayrıştırma sırası, en sağdan türetmenin tersine karşılık gelir. Yani türetmenin cümlesel formu sondan ilke doğru üretilir. Türetme açısından aşağıdan yukarıya ayrıştırma şu şekilde tarif edilebilir:

Doğru bir cümle formu verildiğinde, ayrıştırıcı, önceki cümle formunu sağdan sola doğru üretmek için dilbilgisindeki kuralın sağ tarafının sol tarafına indirgenmesi gereken kuralı belirlemelidir.

1.5.1.2.1. LR Ayrıştırma

LR ayrıştırıcı, özyinelemeli olmayan, kaydırmayı azaltan, aşağıdan yukarıya (Bottom-Up) bir ayrıştırıcıdır. İçerikten bağımsız gramerin geniş bir sınıfını kullanarak en etkin sözdizim analiz tekniğini inşa eder. LR ayrıştırıcıları ayrıca L'nin giriş akışının soldan sağa taranması anlamına geldiği LR (k) ayrıştırıcıları olarak da bilinir; R, sağdan türetme işleminin tersine inşası anlamına gelir ve k, kararlar için alınacak sembollerin sayısını belirtir.

LR ayrıştırıcıların sağladığı birkaç avantaj aşağıdaki gibi maddeler halinde özetlenebilir:

- Programlama dillerini tanımlamak için oluşturulmuş gramerlerin çoğunda kullanılabilir
- Kendisinden farklı aşağıdan yukarıya ayrıştırıcılar kadar verimli olmasının yanında diğerlerinden daha kapsamlı gramerler sınıfı için işlemler gerçekleştirebilir
- Sözdizim hatalarını oldukça hızlı bir şekilde tespit edebilir

LR ayrıştırıcıda, ayrıştırma kurallarının kaydedildiği ayrıştırma tablosunun el ile oluşturulması neredeyse imkansızdır. Bu durum otomatik ayrıştırıcı üreteçlerinin geliştirilmesiyle problem olmaktan çıkmıştır.

LL ayrıştırma yönteminin tahmine dayalı bir ayrıştırma yöntemine sahip olmasından dolayı LR ayrıştırma yöntemi, LL ayrıştırma yönteminden daha güçlü bir yapıya sahiptir.

LR ayrıştırıcıların oluşturdukları ayrıştırma tablosu diğer ayrıştırıcılardan genel olarak daha küçüktür ve bu tablonun kaynak kodları daha az yer kaplar.

1.6. Ayrıştırıcı Üreteçleri

Bilgisayar programlama dillerinin sayısı her geçen gün artmaktadır. Her biri farklı bir amaca yönelik olarak geliştirilmiş yüzlerce programlama dili vardır. Programlama dillerinin çeşitliliği çok olduğundan bu dillere uygun ayrıştırıcı üreteçlerinin de geliştirilmesi kaçınılmaz olmuştur. Günümüzde en çok kullanılan programlama dilleri için ayrıştırıcı üretebilen uygulamalardan bazıları bu bölümde tanıtılmıştır.

1.6.1. Bison

Bison, içerikten bağımsız grameri (CFG), LALR(1) ayrıştırıcı tabloları kullanan bir deterministik LR ayrıştırıcısına veya genelleştirilmiş LR(GLR) ayrıştırıcısına dönüştüren genel amaçlı bir ayrıştırıcı üretecidir. Deneysel bir özellik olarak IELR(1) ve kanonik LR(1) ayrıştırıcı tabloları da oluşturulabilir [39].

Bison aslen Robert Corbett tarafından geliştirilmiş olsa da Richard Stallman tarafından Yacc ile uyumlu hale getirilmiştir. Bison, Yacc ile yukarı doğru uyumludur, yani Yacc ile doğru yazılmış tüm gramerler Bison ile birlikte hiçbir değişikliğe gerek kalmaksızın çalışabilir. Bison'u kullanabilmek için C/C++ dillerine hakim olmak gerekir. Java da deneysel bir özellik olarak desteklenmektedir.

1.6.2. Lex

M. E. Lesk ve E. Schmidt tarafından geliştirilmiş olan Lex, kelime manası olarak sözcüksel analiz üreteci anlamına gelir. Lex, kontrol akışı girdi olarak verilen düzenli ifadelerin örnekleri tarafından yönlendirilen programlar yazmaya yardımcı olur.

Lex'in kaynağı düzenli ifadelerin ve buna karşılık gelen program parçalarının bir tablosudur [40].

1.6.3. ANTLR

ANTLR, yapılandırılmış metinleri veya ikili (binary) dosyaları okumak, işlemek, icra etmek veya çevirmek için kullanılabilecek güçlü bir ayrıştırıcı üretecidir. Her türlü dili, aracı ve yapıyı (Framework) oluşturmak için akademi ve endüstride yaygın olarak kullanılır. Örneğin; Twitter, günde 2 milyardan fazla sorgu içeren sorgu ayrıştırmaları için ANTLR'yi kullanır, Oracle, SQL Developer IDE ve geçiş araçlarında ANTLR'yi kullanır, Netbeans IDE, C++'ı ANTLR ile ayrıştırır [33].

Gramer adı verilen biçimsel bir dil tanımından, ANTLR bu dil için otomatik olarak ayrıştırma ağaçları oluşturabilen bir ayrıştırıcı oluşturur. Bu ayrıştırma ağacı yapısı gramerin girdiyle nasıl eşleştiğini temsil eder. Ayrıca ANTLR, uygulamaya özel kod yürütmek için bu ağaçların düğümlerini ziyaret etmek amacıyla kullanılabilecek ağaç gezinme mekanizmasını da otomatik olarak üretir.

1.6.4. Yacc

Yacc (Yet Another Compiler Compiler), Unix sistemlerinde mevcut olarak gelen bir LALR (1) ayrıştırıcı üretecidir. İçeriğindeki harflerin açılımına bakılacak olursa, Bir Başka Derleyici Derleyici (Yet Another Compiler Compiler), piyasaya sürüldüğünde bir derleyici derleyicinin pek fazla olmadığı düşünüldüğünde ismi oldukça ilginç gözükmektedir. Yacc'in popülaritesi düşünüldüğünde bir dizi Yacc türevi program geliştirilmiştir. Ancak Yacc sürümlerinin çoğu yalnızca C/C++ ayrıştırıcılarından oluşur. Bunun yanında bazı versiyonları C/C++ ve Java ayrıştırıcıları da oluşturabilir [27]. Yacc, JavaCC'den iki önemli noktada farklılık gösterir:

- Yukarıdan-aşağıya (Top-Down) ayrıştırıcı yerine aşağıdan yukarıya (Bottom-Up) ayrıştırıcı oluşturur.
- Kelimesel analiz üreteci içermez.

Yacc tarafından oluşturulan bir ayrıştırıcıyı kullanmak için, yine Yacc tarafından üretilen bir kelimesel analiz üreteci tarafından istenen biçimde belirteçler sağlayan yylex adlı bir kelimesel analiz yöntemi sağlanmalıdır. Kelimesel analiz el ile yazılabilir veya başka

bir kelimesel analiz üreteci ile oluşturulabilir. Genel olarak Yacc ile kullanılan kelimesel analiz üreteçleri Lex, Flex, Jlex versiyonlarıdır.

1.6.5. SableCC

SableCC, derleyiciler, yorumlayıcılar ve diğer metin ayrıştırıcıları oluşturmak için Java programlama dilinde tam özellikli nesne yönelimli çerçeveler üreten bir ayrıştırıcı üretecidir. Özellikle, oluşturulan çerçeveler sezgisel olarak tam olarak yazılmış soyut sözdizimi ağaçları ve ağaç yürüyüşü içerir. SableCC ayrıca, makine tarafından üretilen kod ile kullanıcı tarafından yazılmış kod arasında daha kısa bir geliştirme döngüsüne yol açan temiz bir ayrım yapılmasını sağlar [32].

1.6.6. JavaCC (Java Compiler Compiler)

Java derleyici derleyicinin kısaltması olarak bilinen JavaCC, Java uygulamalarıyla kullanılan en popüler ayrıştırıcı üretecidir. Ayrıştırıcı üreteci gramer özelliklerini okuyup, gramer ile eşleşmeleri tanıyabilen bir java programına dönüştüren bir araçtır. Ayrıştırma üretecinin kendisine ek olarak, JavaCC, JJTree adlı bir araç aracılığıyla ağaç oluşturma, hata ayıklama işlemleri gibi ayrıştırıcı oluşturma ile ilgili diğer standart yeteneklere sahiptir [41].

1.7. JavaCC ile Ayrıştırıcı Üretimi

JavaCC ile ayrıştırıcı üretirken, .jj uzantılı dosyalar içerisine LL(k) gramerleri tanımlanır. .jj uzantılı dosya içerisinde, ayrıştırıcı sınıfın, terminal ve gramer kurallarının belirtimi ayrı bölümler içinde yapılır. < ve > işaretleri arasında terminaller belirtilir, kurallar ise programlamada kullandığımız metotlara benzer yapıda tanımlanır. Tanımlanan metotlara gramerde belirlediğimiz nonterminal ile uyumlu bir isim verilir. Örnek bir gramer kuralı Tablo **2**'de verilmiştir.

Tablo 2. JavaCC Kural Bildirimi

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

1.8. Sözdizim Sınıfı

Sözdizim analiziyle oluşturulacak olan soyut sözdizim ağacının düğümleri sözdizim sınıflarından oluşmaktadır. Sözdizim sınıfları belirli gramer kurallarını temsil edecek şekilde tanımlanabilirler. Anlaşılmasının kolay olması amacıyla sözdizim sınıflarının isimleri ilgili kuralın ismine benzer şekilde türetilir. Gramer kuralında bulunan nonterminaller için sözdizim sınıfına birer veri eklenir. Dilin temel yapı taşı alfabesindeki sembollerdir. Semboller belirli kurallar dahilinde bir araya gelerek anlamlı sözcükler oluşturur. Bu anlamlı sözcükler de belirli gramer kuralları ile bir araya gelerek anlamlı cümleler oluştururlar.

Sözdizim sınıfları, gramer kuralları vasıtasıyla oluşturulabilen girdi verilerini nesneye dayalı programlama yapılarıyla temsil edebilmek amacıyla oluşturulurlar. Tablo 3'de bazı sözdizim sınıflarının yapısı gösterilmiştir.

Tablo 3. Bazı Sözdizim Sınıflarının Yapısı

```
abstract class Exp { }
class Plus extends Exp {
    public Exp left, right;
    public Plus(Exp left, Exp right) {this.left = left;
this.right = right;}}
class Tan extends Exp {
    public Exp;
    public Tan(Exp exp) {this.exp = exp;}}
class Var extends Exp {
    public String var;
    public Var(String var) {this.var = var;}…
}
```

1.8.1. Sözdizim Ağacı Değerlendirme Yöntemleri

Sözdizim sınıflarından oluşan sözdizim ağacı, en uçtaki düğümden kök düğüme doğru değerlendirilir. Düğümlerin değerlendirilmesi için yapılması gereken işlemler, düğümü oluşturan nesnenin türüne bağlıdır. Bu aşamada nesne türünün belirlenmesi için farklı yöntemler kullanılmaktadır. Kullanılan yöntemlerin karşılaştırılması Tablo 4'te gösterilmiştir.

Tablo 4. Yöntemlerin Karşılaştırılması

Yöntem	**Nesne Türetme**	**Sınıf Derleme**
instanceof operatörü	Var	Yok
Sözdizim sınıflarına metot ekleme	Yok	Var
Visitor tasarım deseni	Yok	Yok

1.8.1.1. instanceof Operatörü

Nesneye dayalı programlamanın gelişimi ile birlikte kalıtım ve çok biçimlilik kavramları ortaya çıkmıştır. Bu kavramlar kullanılarak birbirinden oluşturulan nesnelerin tiplerinin belirlenebilmesi için de farklı yöntemler geliştirilmiştir. instanceof operatörü test edilen sınıfın istenen türe ait bir sınıf olup olmadığını belirler. Nesnenin hangi türe ait olduğu tespit edildikten sonra gerekli işlemlerin gerçekleştirilmesi için gerekli tip dönüşümleri yapılması gerekmektedir. Tablo 5'te instanceof operatörü kullanılan bir değerlendirme metodu gösterilmiştir.

Tablo 5. instanceof Operatörü ile Değerlendirme

```
public static double eval(Exp exp, double x) {
    if (exp instanceof Times)
        return eval(((Times)exp).left, x) *
eval(((Times)exp).right, x);
    else if (exp instanceof Divide)
        return eval(((Divide)exp).left, x) /
eval(((Divide)exp).right, x);…}
```

1.8.1.2. Sözdizim Sınıflarına Metot Ekleme

Sözdizim ağacı değerlendirmek için kullanılabilecek bir diğer yöntemde, sözdizim sınıflarının her birine kendi sorumluluk alanının işlemlerini gerçekleştirecek bir metot eklenir. Eklenen bu metot, düğüm değerlendirilmesi aşamasında, değerlendirme sırası ilgili nesneye geldiğinde çağrılmalıdır. Bu yöntemde değerlendirilmesi gereken nesnenin türünün belirlenmesi gerekmez.

Tablo 6'te Sözdizim sınıflarına metot ekleme ile değerlendirme yapan bir metodun tanımı gösterilmiştir.

Tablo 6. Sözdizim Sınıflarına Metot Ekleme

```
abstract class Exp {
     public abstract double eval(double num);}
class Minus extends Exp {
     public Exp left, right;
     public Minus(Exp left, Exp right) {
          this.left = left;
          this.right = right;}
     public double eval(double num){
          return (left.eval(num)-right.eval(num));
     }
}…..
class Log extends Exp {
     public Exp;
     public double eval(double num){
          return Math.log(exp.eval(num));
     }
}
```

1.8.1.3. Visitor Tasarım Deseni

Visitor tasarım deseni yardımıyla veri ve veriye uygulanacak işlem birbirinden ayrılabilir. Bu tasarım kalıbına göre uygulanacak işlem, verileri doğrudan işlemek yerine verilere bir visitor(ziyaretçi) yardımıyla erişir. Bu veri sınıflarını değiştirmeden ek işlemler eklemenizi sağlar, böylece verileri ve kodu birbirinden ayırır. Örneğin soyut sözdizim ağacı gibi bir düğüm ağacı, ağacın gezinilmesine dayanan farklı işlemler gerektirebilir. Visitor

tasarım deseninde gezinen kod bu tür işlemleri kabul edebilir, bu nedenle kodda herhangi bir değişiklik olmaz. Bu tasarım desenindeki mantık, veri bulma kodunu değil, verilerin kodunu bulmasına dayanır.

Sözdizim ağacının Visitor sınıfı yardımıyla gezinilebilmesi için ağacı oluşturan bütün sözdizim sınıf türlerine visit() metodu ve bütün sözdizim sınıflarına da accept() metodu ilave edilmelidir. Ağacın bütün düğümlerine erişilinceye kadar visit() metodu değerlendirilecek nesnenin accept() metodunu, accept() metodu değerlendirme işlemini gerçekleştiren visit() metodunu çağırır. Visitor bütün sözdizim sınıfları için bir visit() metodu barındıran bir arayüzdür. Bu visit() metotlarının içeriği Visitor arayüzünden türeyen sınıf içerisinde yapılır.

Visitor tasarım deseni farklı nesnelerden oluşmuş yapıları kolayca kullanabilme imkanı sunar. Uygulanabilecek diğer yöntemler farklı sorunlar ortaya çıkarabilmektedir. Visitor tasarım deseninin bir diğer avantajı ise, hali hazırda tanımlanmış olan sınıfların tekrar derlenmeye gerek duyulmaksızın nesneleri değerlendirebilecek kodlama işleminin yapılabilmesidir.

Tablo 7'de Sözdizim Sınıflarına accept() metodunun eklenmesi ve Tablo 8'de Bir visitor arayüzünün tanımı ve bu arayüzden türetilmiş bir sınıfın tanımı gösterilmiştir.

Tablo 7. Sözdizim Sınıflarına accept() Metodunun Eklenmesi

```
abstract class Exp {
    public abstract double
    accept(Visitor v); }
class Plus extends Exp {
    public Exp left, right;
    public Plus(Exp left, Exp right) {
        this.left = left; this.right = right;}
    public double accept(Visitor v) { return v.visit(this);}
}
class Num extends Exp{
    public double num;
    public Num(double num) { this.num = num; }
    public double accept(Visitor v) {
        return v.visit(this);
    }
}
……
```

Tablo 8. Visitor Arayüzü ve Türetilen EvalVisitor Sınıfını

```
public interface Visitor {
public double visit(Exp exp);
……
public double visit(Euler exp);
public double visit(Log exp);
public double visit(Sqrt exp);
}
public class EvalVisitor implements Visitor {
    double num;
    public EvalVisitor(double num) { this.num = num; }
    public double visit(Euler exp) {
        double num = exp.exp.accept(this);
        return Math.pow(Math.E, num); }
    public double visit(Num exp) { return (exp.num); }
    public double visit(Var exp) { return var; }
}
```

1.9. İntegral

İntegral alan veya alanın genelleştirilmesi olarak yorumlanan matematiksel bir nesnedir. İntegral türevle birlikte hesaplamanın (Calculus) temel nesnesidir. Bir başka deyişle integral türevin tersi ve ilkeli olarak ifade edilebilir. Riemann integrali, en basit integral tanımıdır ve genellikle fizik ve temel hesaplamada karşılaşılan tek tanımdır [42].

Riemann integralinin f(x) fonksiyonun x değişkeni üzerinde a ile b noktaları arasındaki genel tanımı (1) numaralı denklemde gösterilmiştir.

$$\int_a^b f(x)dx \tag{1}$$

Eğer $f(x) = 1$ ise , bu integral sadeleştirilmiş biçimde aşağıdaki (2) numaralı denklemde olduğu gibi ifade edilebilir.

$$\int_a^b dx \tag{2}$$

İntegral hesaplama sürecine entegrasyon denir ve bir integralin yaklaşık hesaplanması sayısal entegrasyon olarak adlandırılır.

İki sınıf Riemann integrali vardır:

- (1) numaralı denklemde gösterilen üst ve alt limitleri olan belirli integraller
- (3) numaralı denklemde gösterilen Limitleri belirsiz olan belirsiz integraller

$$\int f(x)dx \tag{3}$$

Hesaplamanın birinci teoremi, belirli integrallerin belirsiz integraller cinsinden hesaplanmasını sağlar. Eğer $F(x)$, $f(x)$'in belirsiz integrali ise, $f(x)$'in [a,b] aralığındaki integrali (4) numaralı denklemle ifade edilebilir.

$$\int_a^b f(x)dx = F(b) - F(a) \tag{4}$$

Sabit bir terimin türevi sıfır olduğu için, belirsiz integraller (5) numaralı denklemde görüldüğü gibi sadece bir C entegrasyon sabitine kadar tanımlanabilir.

$$\int f(x)dx = F(x) + C \tag{5}$$

1.9.1. İntegral Alma Kuralları

İntegral problemleri analitik olarak hesaplanırken bazı temel formüller bilinmesi gerekmektedir [43]. Bu temel formüllerden genel olarak bilinen ve yaygın olarak kullanılan birkaç tanesi Şekil 9'da listelenmiştir.

$$\int dx = x + C \tag{6}$$

$$\int x^n \, dx = \frac{x^{(n+1)}}{(n+1)} + C, n \neq 1 \tag{7}$$

$$\int \frac{dx}{x} = \ln(x) + C = \ln(x) + \ln(C) = \ln(x * C) \tag{8}$$

$$\int a^x \, dx = \frac{a^x}{\ln(a)} + C \tag{9}$$

$$\int e^x dx = e^x + C \tag{10}$$

$$\int \sin(x) dx = -\cos(x) + C \tag{11}$$

$$\int \cos(x) dx = \sin(x) + C \tag{12}$$

$$\int \tan(x) dx = \ln(\sec(x)) + C \tag{13}$$

$$\int \cot(x) dx = \ln(\sin(x)) + C \tag{14}$$

$$\int \ln(x) dx = x\ln(x) - x + C \tag{15}$$

$$\int \frac{a}{a^2 + x^2} dx = \arctan(\frac{x}{a}) + C \tag{16}$$

$$\int \frac{a}{a^2 - x^2} dx = \frac{1}{2}\ln(\frac{x+a}{x-a}) + C \tag{17}$$

$$\int \frac{dx}{\sqrt{a^2 - x^2}} = \arcsin(\frac{x}{a}) + C \tag{18}$$

$$\int \frac{dx}{\sqrt{a^2 \pm x^2}} = \ln(x + \sqrt{a^2 \pm x^2}) + C \tag{19}$$

Şekil 9. Bazı İntegral Formülleri

Analitik olarak integral hesaplamak için yukarıda belirtilen formüllerin haricinde genel birkaç kural daha vardır. Bu kuralların en yaygın olarak kullanılanları Şekil 10'da gösterilmiştir.

$$\int af(x)dx = a\int f(x)dx, \qquad a\ sabit\ ise \tag{20}$$

$$\int (f(x) \pm g(x))dx = \int f(x)dx \pm \int g(x)dx \tag{21}$$

$$\int (f(x) * g(x))dx = f(x)\int g(x)dx - \int \left(\int g(x)dx\right) f'(x)dx \tag{22}$$

$$\int udv = uv - \int vdu \tag{23}$$

Şekil 10. Yaygın Kullanılan İntegral Alma Kuralları

(22) numaralı denklem kısmi integral olarak ifade edilir. Daha sade ve anlaşılır biçimde ifade edilecek olursa (23) numaralı denklemdeki gibi temsil edilebilir [44].

2. YAPILAN ÇALIŞMALAR, BULGULAR VE DEĞERLENDİRME

Basit matematiksel işlemlerin el yordamıyla yapılması mümkün iken, karmaşık matematiksel problemlerin çözümünün insan eliyle çözülemeyecek kadar zor olduğu, hatta klasik yöntemlerle dahi çözümünün hesaplanamaz olduğu görülebilir. Matematiksel işlemlerin bilgisayarlar aracılığıyla hesaplanmasında birçok farklı hesaplama yöntemi ve bu yöntemleri uygulayacak programlama yöntemlerine gereksinim duyulur. Hesaplama yöntemleri simgesel ve sayısal yaklaşımlar olarak iki kısma ayrılabilir. Sayısal yöntemler ile yapılan hesaplamalar, sonuca belirli bir hata payı miktarınca yaklaşabilmeyi esas alır ve işlemleri bu hata payına ulaşana kadar tekrarlar. Simgesel hesaplama yaklaşımıyla analitik çözüm yöntemi bulunan matematiksel problemlerin çözümü tam (hata payı olmadan) olarak hesaplanmaya çalışılır.

Bu çalışma dahilinde, simgesel integral hesabının bilgisayarlar yardımıyla yapılabilmesi için geliştirilen uygulamada, biçimsel diller yardımıyla geliştirilen kaynak program kodunun, kelimesel analiz, sözdizim analizi ve anlamsal analiz adımlarından geçirilmesi süreçlerinde, Java programlama dilinde otomatik kod üretebilen JavaCC ayrıştırıcı üreteci aracından yararlanılmıştır. Matematiksel ifadelerin gramer yapısına uygun olacak şekilde bir içerikten bağımsız gramer yazılmış ve LL(k) gramerine uygun olacak şekilde düzenlenmiştir. Sözdizim sınıfları ifadenin içerdiği operatör ve fonksiyonlar için oluşturulmuştur. Geliştirilen uygulama, kendisine girdi olarak verilen matematiksel ifadeyi okuyarak belirteçler dizisine dönüştüren bir belirteç üreteci, ifadenin sözdizim ağacını bu belirteçler dizisini ayrıştırarak oluşturan bir ayrıştırıcı, oluşan bu sözdizim ağacı üzerinde işlem yapabilen integral alıcı, türev alıcı, ifade sadeleştirici ve ifade gösterici gibi bileşenlerin kombinasyonundan oluşur. Simgesel integral hesaplama sisteminin genel yapısı ‘de gösterilmiştir.

2.1. Geliştirilen Matematiksel Programlama Dilinin Genel Yapısı

Geliştirilen uygulamada kullanılan kaynak kod bir fonksiyon tanımından oluşmaktadır. Geliştirilen programlama dilinin EBNF notasyonundaki grameri, fonksiyonlarda yapılacak diğer hesaplamalarda kullanılır. Simgesel yöntemleri programlayabilmek için bir gramer gereklidir. Tasarlanan gramer matematiksel metotların kodlanabildiği bir programlama dilini belirler. Bu dilde aritmetik operatörler, birkaç temel matematiksel fonksiyon ve int, string ve double gibi veri türleri kullanılabilir. EBNF notasyonunda belirlenen gramer aşağıda gösterilmiştir.

```
<integral> <LPR> <expression> <COMMA> <var> <RPR>
```

Yukarıda belirtilmiş gramer için fonksiyon örneği verecek olursak; integral(x^2+a*x , x) : x^2 + a*x fonksiyonun x değişkenine göre integralinin alınacağını ifade eder.

2.2. Gramer Yazımı

Gramer, biçimsel dillerde karakter dizileri üretebilen kurallar kümesi olarak tarif edilebilir. Dili oluşturan alfabenin birimleri kullanılarak oluşturulacak dilin sözdizimine uygun kelimelerin nasıl üretileceği kurallar yardımıyla belirlenir. Çeşitli gramer türleri vardır. Bunlardan bazıları:

- İçerikten bağımsız gramer (CFG)
- Düzenli gramerler
- Analitik gramerlerdir.

Yapılan çalışmada matematiksel ifadeleri tanımlayabilecek sözdizim yapısında bir CFG tanımlanmış ve operatör adımlarını barındıracak şekilde LL(1) gramerine dönüşümü gerçekleştirilmiştir.

2.2.1. CFG Düzenleme

İçerikten bağımsız gramer karakter dizisi yapılarını oluşturmak için özyinelemeli yazma kurallarından oluşur

Bir CFG aşağıdaki bileşenlerden meydana gelir.

- Bir dizi sonlu terimler (terminal) seti :Dilin alfabesinin kelimeler içinde görünen karakterlerini temsil eder
- Bir dizi devamlı terimler (non-terminaller) seti : Devamlı terimler tarafından üretilebilecek sonlu terimlerin yerini tutan sembollerdir.
- Bir dizi kurallar seti :Karakter dizisindeki devamlı terimlerin sonlu veya devamlı sembollerle yer değiştirme kurallarıdır.
- Başlangıç sembolü :Gramer tarafından oluşturulan, ilk karakter dizisinde görünen özel bir devamlı terim sembolüdür.

Sonlu terimlerin bir dizisini CFG den oluşturmak için:

- Başlangıç sembolünden oluşan karakter dizisinden başlanır
- Başlangıç sembolünü kuralın sağ tarafıyla değiştirerek sol taraftaki başlangıç sembolüyle kurallardan biri uygulanır
- Karakter dizisindeki devamlı sembolleri seçme ve devamlı olanlar sonlu olanlarla değiştirilene kadar bu işlem tekrarlanır.

Aşağıda aritmetik ifadeleri temsil edebilecek karakter dizisini üretebilen bir CFG gösterilmiştir.

1. E ::= E + E
2. E ::= E - E
3. E ::= E*E
4. E ::= E / E
5. E ::= x | [“0”-“9”]+

Bu gramerde işlem öncelikleri dikkate alınmamıştır ve çok anlamlılık/anlamsızlık durumları oluşabilir. Bu tür durumlardan kaçınmak için bu gramer işlem öncelikleri dikkate alınarak yeniden düzenlenmelidir. Gramer yeniden düzenlendiğinde aşağıdaki gibi olur.

1. E ::= E+T | E – T | T
2. T ::= T * F | T / F | F
3. F ::= x | [“0”-“9”]+

2.3. LL(k) Gramerine Dönüşüm

JavaCC ile oluşturulan ayrıştırıcı üreteçleri aracılığıyla analiz edilen matematiksel ifadeler LL(k) karakterli bir gramera dönüştürülmelidir. LL(k) gramerlerinde girdiden en az 1 adet olmak üzere en fazla k adet sonlu terim okunarak kural seçim işlemi gerçekleştirilir.

Bu yöntem uygulanırken LL(1) grameri aşağıdaki özellikleri sağlamış olur.

- Herhangi bir kuralın her alternatifi farklı bir devamlı terimle başlar.
- NULL durumu “Evet” olan kuralların FIRST ve FOLLOW setleri ortak sonlu terim barındıramaz.
- Tanımlanan kurallar soldan özyinelemeli değildir.

Gramera yeni kurallar ekleyerek LL(1) grameri oluşturulmuş olur.

Aşağıda matematiksel ifadeler için bir LL(1) gramer tanımlaması gösterilmiştir. $ sembolü giriş verisinin sonlandığını belirtir.

1. E ::= TE’
2. E’ ::= +TE’ | Boş
3. T ::= FT’
4. T’ ::= *FT’ | boş
5. F ::= (E) | int

2.4. Ayrıştırıcılar

Ayrıştırıcı kendisine verilen girdinin gramer kurallarına uygunluğunu belirteçler dizisini kullanarak kontrol eder. Bu aşamada, girdinin sözdiziminin doğruluğu kontrol edilir veri üzerinde anlamsal bir analiz yapılmaz. Sözdizimsel olarak gramere uyan girdinin değerlendirilmeden önce anlamsal analizden geçirilmesi gerekmektedir. Tablo 9'da JavaCC formatında tanımlanmış olan ve LL(1) gramera göre sözdizim analizi yapan ayrıştırıcının yapısı gösterilmiştir.

Tablo 9. JavaCC Ayrıştırıcı Yapısı

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

2.5. Sözdizim Sınıfları

Sözdizim sınıfları gramer kuralları yardımıyla ayrıştırılan verileri, nesnelerle temsil edebilmek için tanımlanır. Operatörlerin tanımlandığı sözdizim sınıflarına operatörün işlem yapacağı veriler de eklenir. Tablo 10'da bazı sözdizim sınıflarının tanımı verilmiştir.

Tablo 10. Sözdizim Sınıfları

```
abstract class Exp {
  public abstract double accept(Visitor v);}
class Plus extends Exp { public Exp lhs, rhs;
  public Plus(Exp lhs, Exp rhs) {
     this.lhs = lhs; this.rhs = rhs;}
  public double accept(Visitor v) {
     return v.visit(this);}}
class Minus extends Exp { public Exp lhs, rhs;
  public Minus(Exp lhs, Exp rhs){
     this.lhs = lhs; this.rhs = rhs;}
  public double accept(Visitor v){
     return v.visit(this);}}
```

2.6. Sözdizim Ağacı

Sözdizim ağacı adından da anlaşılacağı üzere birbirine hiyerarşik bir bağla bağlanmış düğümlerden oluşur. Düğümler bir işlemi veya veriyi temsil eden bir nesne barındırır. Sözdizim ağaçlarında her düğüm miras aldıkları üst sınıf türüyle temsil edilir ve sözdizim ağacı da bu üst sınıf yardımıyla oluşturulur.

JavaCC ile nesne ağacı üretmek, tanımlamalara ek Java ifadeleri ekleyerek mümkün olmaktadır. Tablo 11'de matematiksel ifadeleri temsil edebilecek nesne ağacını üretmek için Java ifadelerinin gramere nasıl ekleneceği gösterilmiştir.

Tablo 11. Sözdizim Ağacının Üretilmesi

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

JavaCC'de belirteç ve ayrıştırıcı tanımlaması. jj uzantılı tek bir dosya içerisinde tutulur. Bu dosyada JavaCC'ye özel çevre değişkenlerine de değer ataması yapılabilir.

Tanımlanan bu ifadeler yardımıyla girdi verisi "3*x+sin(x)" olan ifade için aşağıdaki nesne ağacı oluşturulacaktır.

```
Exp exp = new Plus(new Times(new Num(3.0),new Var()),
                   new Sin(new Var()))
```

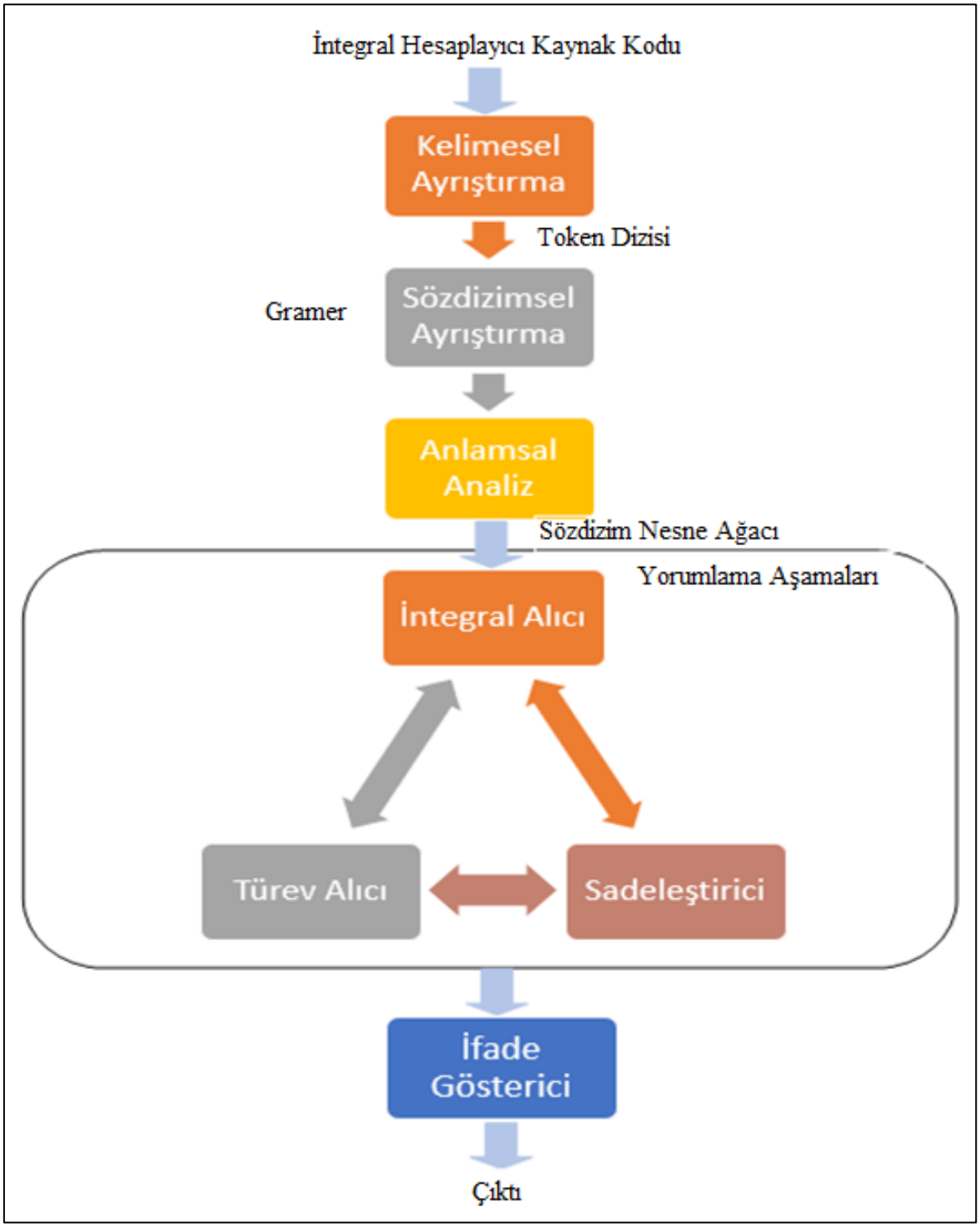
2.7. İntegral Alıcı

Matematiksel ifadelerin integrallerinin hesaplanması oldukça karmaşık ve zor olabilmektedir. İntegral hesabının kuralları tam olarak tanımlanmamış olduğundan simgesel yaklaşımlarla sadece belirli kuralları ve formülleri olan integral problemlerinin çözümü yapılabilmektedir. Simgesel hesaplamanın yetersiz kaldığı bazı durumlarda sayısal yöntemlerle hesaplamalar yapılabilmesine rağmen her iki yöntemle de çözümü bilinmeyen

problemlerin varlığı bilinmektedir. Yapılan çalışmada, analitik çözümü bilinen ve çözümünde sonlu sayıda terim içeren belirsiz integral problemlerinin hesaplanabilmesi için bir yorumlayıcı geliştirilmiştir. Geliştirilen yöntem:

- İntegral alıcı
- Türev alıcı
- İfade Sadeleştirici
- İfade Gösterici

bileşenlerinin birleşiminden oluşmaktadır. Geliştirilen integral alıcı uygulamasının genel yapısı Şekil 11‘da gösterilmiştir.



Şekil 11. İntegral Alıcı Genel Yapısı

Bölüm 1.9'da integral hesabına yönelik şimdiye kadar tanımlanabilmiş genel formül ve kuralların bir kısmı gösterilmiştir. Çalışmamızda tanımlanmış olan bu formül ve kurallara uyan belirsiz integral problemlerinin çözümü gerçekleştirilmeye çalışılmıştır.

İntegral hesaplama işlemi, verilen matematiksel ifadenin analiz işlemlerinden geçirilerek oluşturulan nesne ağacı üzerinde en uç düğümden kök düğüme doğru değerlendirme yapılarak gerçekleştirilir. Düğümlerde yapılacak işlem düğümün kendi türüne bağlı olmaktadır ve her düğümde bulunan nesne türünün tespit edilebilmesi için 3 farklı yöntem mevcuttur. Bunlar:

- instanceof operatörü
- Sözdizim sınıflarına metot ekleme
- Visitor tasarım desenidir.

Simgesel integral hesabı yapılırken gerçekleştirilecek tüm işlem adımlarında tür belirleme işlemi Visitor tasarım deseni yardımıyla yapılmıştır. İfadelerin integralleri hesaplanırken Tablo 12'de Visitor arayüzünün ve Tablo 13'da Visitor arayüzünden türetilmiş DeriveVisitor sınıfının genel yapısı gösterilmiştir.

Tablo 12. Visitor Arayüzünün Genel Yapısı

```
public interface Visitor {
    public Exp visit(Exp exp);
    public Exp visit(Plus exp);
    public Exp visit(Times exp);    public Exp visit(Sin
exp);….}
```

Tablo 13. IntegralVisitor Sınıfısın Genel Yapısı

```
public class IntegralVisitor implements Visitor {
    public Exp visit(Plus exp) {
        Exp l = exp.left.accept(this);
        Exp r = exp.right.accept(this);
        return new Plus(l, r); }
    public Exp visit(Times exp) {
        Exp u = exp.left;
        Exp dv = exp.right;
        Exp du = DeriveVisitor.visit(u);
        Exp v = dv.accept(this);
        return new Minus(new Times(u, v),new Times(v,
du).accept(this));...    }}
```

Örneğin; $x * sin(x) + 3 * e^x$ ifadesine integral alma işlemi uygulanmak istendiğinde ilk olarak aşağıdaki gibi bir nesne ağacı oluşturulur.

```
Exp exp = new Plus( new Times(new Var(),new Sin(new Var())),
                new Times(new Num(3.0),new Exp(new Var())));
```

Nesne ağacı oluşturulduktan sonra bu ağaç üzerinde integral hesabı kurallara uygun olarak yapıldığında oluşturulmuş olan bu ağaç aşağıdaki şekle dönüşür.

```
Exp resultExp = new Plus(new Minus(new Times(new Var(),
    new Negate(new Cos(new Var()))), new Times(new Num(1.0),
    new Negate(new Sin(new Var())))), new Times(new Num(3.0),
    new Divide(new Exp(new Var()),new Num(1.0))));
```

2.8. Türev Alıcı

Yapılan çalışmada, bazı matematiksel ifadelerin integrallerinin hesaplanma sürecinde türev alım işlemine de ihtiyaç duyulmaktadır. Bu sebeple, geliştirilen uygulamaya türev alıcı bir yapı ilave edilmiştir.

Bütün matematiksel ifadelerin türevlerinin hesaplanabilmesi için genelleştirilmiş formüller mevcuttur. Aşağıda türev işleminin genel matematiksel formülleri gösterilmiştir. Bu formüller sayesinde matematiksel olarak ifade edilen bütün problemlerin türevleri alınabilmektedir. Bu formülleri bilgisayar ortamında modelleyerek tüm matematik problemlerinin türevlerinin bilgisayarlar aracılığıyla hesaplanabilmesi mümkün olmaktadır.

$$\boldsymbol{\left(f(x) \pm g(x)\right)' = \left(f(x)\right)' \pm \left(g(x)\right)'} \quad \textbf{(24)}$$

$$\boldsymbol{\left(f(x) * g(x)\right)' = \left(f(x)\right)' * \left(g(x)\right) + \left(f(x)\right) * \left(g(x)\right)'} \quad \textbf{(25)}$$

$$\boldsymbol{\left(\frac{f(x)}{g(x)}\right)' = \frac{\left(f(x)\right)' * \left(g(x)\right) - \left(f(x)\right) * \left(g(x)\right)'}{\left(g(x)\right)^2}} \quad \textbf{(26)}$$

$$\left(f(x)^{g(x)}\right)' = \left(f(x)^{(g(x)-1)}\right) * \left(g(x) * f(x)\right)' + f(x) * \ln\left(f(x)\right) * g(x) \quad (27)$$

$$\left(f(g(x))\right)' = g'(x) * f'(g(x)) \tag{28}$$

Türev alma işleminde, oluşturulmuş olan nesne ağacı en uç düğümden kök düğüme doğru sırayla değerlendirilir. Düğümlerde yapılacak olan her bir işlem, düğümün türüne göre belirlenir.

Yapılan çalışmada gerçekleştirilecek tüm değerlendirme süreçlerinde tür belirleme işleminin gerçekleştirilmesi için Visitor tasarım deseni kullanılmıştır. İfadelerin türevleri hesaplanırken Tablo 14'de Visitor arayüzünün ve Tablo 14'te Visitor arayüzünden türetilmiş DeriveVisitor sınıfının genel yapısı gösterilmiştir.

Tablo 14. DeriveVisitor Nesnesinin Genel Yapısı

```
public class DeriveVisitor implements Visitor {
    public Exp visit(Ln exp) {
       return new Divide(exp.exp.accept(this), exp.exp);
    }
    public Exp visit(Divide exp) {
        Exp numerator = new Minus(new
                Times(exp.left.accept(this),exp.right),
                        new Times(exp.left,
                                  exp.right.accept(this)));
        Exp denominator = new Power(exp.right, new
                                    Num(2.0));
        return new Divide(numerator, denominator);
    }
}
```

Örneğin; sin(3*x) + x/4 ifadesine türev alma işlemi uygulanmak istendiğinde ilk olarak aşağıdaki gibi bir nesne ağacı oluşturulur.

```
Exp exp = new Plus(
              new Sin(new Times(new Num(3.0),new Var())),
              new Divide(new Var(),new Num(4.0)));
```

Daha sonra oluşturulan bu nesne ağacı türev alma işleminin kurallarına uygun şekilde değerlendirilerek aşağıdaki şekilde yeniden düzenlenerek resultExp ağacı elde edilir.

```
Exp resultExp = new Plus(
     new Times(new Plus(new Times(new Num(0.0),new Var()),
     new Times(new Num(3.0),new Num(1.0))),
     new Cos(new Times(new Num(3.0),new Var()))),
     new Divide(new Minus(new Times(new Num(1.0),
                          new Num(4.0)),
     new Times(new Var(),new Num(0.0))),
     new Power(new Num(4.0),new Num(2.0))));
```

2.9. Sadeleştirici

Matematiksel ifadelerin integrali veya türevi hesaplanırken sadeleştirilmesi gereken terimler oluşabilmektedir. Bu terimler sadeleştirilmediği takdirde, fonksiyon karmaşık bir hale gelir, gereksiz birçok veri barındırabilir ve bu durum ilgili fonksiyonun insanlar tarafından okunurluğunu azaltır. Ayrıca bilgisayar için de bu ifadeler her yeni işlemde değerlendirilmek durumunda kalacağı için ek yük getirmiş olur. Bu sebeple, IntegralVisitor sınıfının metotları ile üretilmiş nesne ağacının yeniden değerlendirilmesi ve bu değerlendirme esnasında varsa sadeleştirilebilir ifadelerin tespit edilip ifade en sade haline dönüştürülmesi gerekir. Sadeleştirme işlemi gerçekleştirilmediği takdirde, yukarıda resultExp nesnesinde olduğu gibi okunması zor bir fonksiyon ile karşılaşılır. Yukarıdaki resultExp nesnesinin temsil ettiği fonksiyon sadeleştirildiğinde aşağıdaki gibi bir yapı elde edilmelidir.

```
Exp resultExp = new Plus(new Times(new Num(3.0),
     new Cos(new Times(new Num(3.0),new Var()))),
     new Divide(new Num(1.0),new Num(4.0)));
```

Sadeleştirme işlemi yapılırken belirli kurallara uyulması gerekmektedir. Bu kurallar matematikte bildiğimiz birbiri yerine kullanılabilecek ifadelerin bir dizisinden oluşur. Bu çalışmada kullanılan sadeleştirme işleminin hangi özdeşlikleri kapsadığı Tablo 15‘te gösterilmiştir.

Tablo 15. Özdeş İfadeler

Sadeleştirilmemiş İfade	**Sadeleştirilmiş ifade**
sayı1 (+ | - | * | / | ^) sayı2	sayı
x + x	2 * x
x – x	0
x * x	x^2
x / x	1
x * 1	x
x * 0	0
x ^ 1	x
x ^ 0	1
1 ^ x	1
0 ^ x	0

Yukarıda belirtilmiş olan özdeşlikler haricindeki sadeleştirme kuralları daha sonraki çalışmalarda dikkate alınabilir. Ayrıca belirtilen durumların bazılarında belirsizlikler oluşmaktadır. Bu belirsizlikler dikkate alınarak hesaplamalar daha güvenilir şekilde yapılabilir.

Sadeleştirme işlemini gerçekleştirmek için yine daha önce oluşturmuş olduğumuz Visitor arayüzünden faydalanabiliriz. Visitor arayüzünden türetilecek SimplifyVisitor nesnesinin genel yapısı Tablo 16'da gösterilmiştir.

Tablo 16. SimplifyVisitor Genel Yapısı

```
public class SimplifyVisitor implements Visitor{
  public Exp visit(Exp exp) { return exp.accept(this);}
  public Exp visit(Plus exp) {
   Exp left = exp.left.accept(this);
   Exp right = exp.right.accept(this);
   if ((left instanceof Num)&&(right instanceof Num)){
    return new Num(((Num)left).num + ((Num)right).num);
   }
   return exp;
}
    ...
}
```

2.10. İfade Gösterici

İntegral hesabı yapıldıktan sonra nesne ağacı sadeleştirme işlemine tabi tutularak yeniden değerlendirilir. Sonuç olarak ortaya çıkan sadeleşmiş ifade matematiksel notasyonda gösterilmelidir. Bu gösterim için de yine integral hesabında olduğu gibi,

- instanceof operatörü
- Sözdizim sınıfına metot ekleme
- Visitor tasarım deseni

yöntemlerinden herhangi biri kullanılabilir. Oluşturulan nesne ağaçlarından matematiksel ifade üreten ifade gösterici sınıfının genel yapısı Tablo 17'de gösterilmiştir. İfade gösteriminde ayrıca dikkat edilmesi gereken bir husus, bu aşamada yukarıda olduğu gibi Visitor arayüzü aynen kullanılmaz. Yeni bir arayüz tanımlanması gerekmektedir. Geliştirilen yeni arayüze ait genel yapısı Tablo 18'de gösterilmiştir.

Tablo 17. PrintVisitor Sınıfının Genel Yapısı

```
public class PrintVisitor implements PVisitor {
    public String visit(Plus exp) {
        return ("(" + exp.left.accept(this) + ")+(" +
exp.right.accept(this) + ")");    }
    public String visit(Minus exp) {
        return ("(" + exp.left.accept(this) + ")-(" +
exp.right.accept(this) + ")");    }
    public String visit(Times exp) {
        return ("(" + exp.left.accept(this) + ")*(" +
exp.right.accept(this) + ")");
    }...}
```

Tablo 18. PVisitor Arayüzünün Genel Yapısı

```
public interface PVisitor {
    public String visit(Exp exp);
    public String visit(Expr exp);
    public String visit(Plus exp);
    public String visit(Minus exp);
    public String visit(Times exp);
    public String visit(Divide exp);
    public String visit(Power exp);
    public String visit(Sin exp);    ...}
```

3. SONUÇ

Bu çalışmada, simgesel yaklaşımlar ve otomatik kod üretim araçları yardımıyla bir simgesel integral hesaplama uygulamasının nasıl geliştirilebileceği gösterilmiştir. Matematiksel ifadenin uygulama aracılığıyla işlenip nesne ağacı şeklinde temsil edilmesi sürecinde LL(k) ayrıştırıcıları için otomatik kod üretebilen JavaCC ayrıştırıcı üreteç aracı kullanılmıştır. Uygulamada matematiksel fonksiyonları temsil edebilecek ifadeler için EBNF notasyonunda bir LL(1) gramer oluşturulmuştur ve bu gramer JavaCC dosya yapısında tanımlanmıştır. Bu gramer kullanılarak üretilebilecek tüm matematiksel ifadeleri nesne ağacı şeklinde temsil edebilecek bir ayrıştırıcı üretilmiştir. Bu nesne ağacı üzerinde işlem yapmak üzere geliştirilen integral alıcı, türev alıcı, sadeleştirici ve ifade gösterici yapılarıyla birlikte analitik çözümü bulunan ve çözümünde sonlu sayıda terim içeren matematiksel ifadelerin integrali hesaplanmıştır.

Matematiksel ifadeler gibi belirli bir yapıya sahip karakter dizilerini değerlendirmek için otomatik kod üretim araçlarından faydalanılabilir. Bu çalışmada geliştirilen yöntem, matematiksel hesaplamalara ihtiyaç duyulan mühendislikten eğitime, matematikten fiziğe tüm alanlarda karşılaşılabilecek olan diferansiyel denklem, türev, limit, integral ve daha birçok analitik problemleri simgesel yaklaşımlarla çözmek için kullanılabilir. Bunun yanında çözüme giden aşamalar adım adım gösterilebilir, bu sayede özellikle matematik eğitiminde öğrencilere yardımcı kaynak olarak kullanılabilir. Yaygın olarak kullanılan birçok simgesel hesaplama sisteminde ara işlem adımları gösterilmeden direkt sonuç gösterilmektedir. Ticari olarak geliştirilmiş ara adımların gösterimini yapan uygulamalar da mevcuttur.

4. ÖNERİLER

- Yapılan çalışmada geliştirilen yorumlayıcıya konulmuş olan kısıtlamalar, sonraki çalışmalarda kaldırılarak daha geniş kapsamlı bir yorumlayıcı elde edilebilir.
- Çalışma daha karmaşık matematiksel ifadeleri kapsayacak şekilde geliştirilerek daha diferansiyel denklemler gibi daha karmaşık problemlerin çözümünde kullanılabilir.
- İntegral probleminin henüz tüm durumları kesin olarak tanımlanabilmiş değildir. Bilimin gelişmesiyle integral hesabına yönelik yeni keşfedilecek olan matematiksel kurallar modellenerek daha geniş kapsamlı bir çalışma yapılabilir.
- Çalışmada geliştirilen yöntem paralelleştirilerek hesaplama süresinin kısaltılması sağlanabilir.
- Çalışmada kullanılan ağaç veri yapısı yerine işlemleri hızlandırabilecek şekilde optimize edilmiş yeni bir veri yapısı kullanılabilir.

5. KAYNAKLAR

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

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

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

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

5. 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.

6. 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.

7. 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.

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

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

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

11. Paul S. Wang, L. P. R. Factoring Multivariate Polynomials Over the Integers, Mathematics of Computation, Temmuz 1975, Bildiriler Kitabı: 935–950.

12. Carette, J., Understanding Expression Simplification, Proceedings of the International Symposium on Symbolic and Algebraic Computation, Temmuz 2004, Santander, Bildiriler Kitabı: 72-79.13. 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.

14. 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.

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

16. Mohammad, M., Alavi Milani, R., Pehlivan, H. ve Hosseinpour, S., Taylor And Maclaurin Series Expansion Using Compiler-Compiler Tools, Mathematical Sciences International Research Journal Volume, 3, 1 (2014) 470-477.

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

18. 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.

19. Gökgöz, B., Simgesel Yaklaşımları Kullanarak Sayısal Kök Bulma Yöntemleri İçin Genel Bir Yorumlayıcının Tasarımı Ve Gerçeklenmesi. Trabzon: Yüksek Lisans Tezi, K.T.Ü., Fen Bilimleri Enstitüsü, Trabzon, 2016.

20. Hassan, M. Y., Design And Implementation Of A General Interpreter For Step-By-Step Solving Nonlinear System Of Equations Using Symbolic Approaches. Trabzon: Yüksek Lisans Tezi, K.T.Ü., Fen Bilimleri Enstitüsü, Trabzon, 2017.

21. Efendioğlu, S., Polinomlar İçin Bir Simgesel Hesaplama Çatısının Tasarımı ve Gerçeklenmesi. Trabzon: Yüksek Lisans Tezi, K.T.Ü., Fen Bilimleri Enstitüsü, Trabzon, 2017.

22. Mohamed, N. A., Design And Implementatıon Of An Interpreter For The Least Squares Method Using Symbolic Approaches. Trabzon: Yüksek Lisans Tezi, K.T.Ü., Fen Bilimleri Enstitüsü, Trabzon, 2018.

23. Pehlivan, H., Sayısal Çözümleme Yöntemlerinin Programlanması ve Yorumlanması. Düzce Üniversitesi Bilim ve Teknoloji Dergisi, 7 (2019) 872-894.

24. Kakde, O.G., Compiler Design, 4. Baskı, University Science Press, Yeni Delhi, Hindistan, 2008.

25. Üstübioğlu, A., Bir Web Tabanlı Minihaskell Yorumlayıcısı. Trabzon: Yüksek Lisans Tezi, K.T.Ü., Fen Bilimleri Enstitüsü, Trabzon, 2009.

26. Scott, M. L., Programming Language Pragmatics, 4. Baskı, Elsevier, London, 2016.

27. Reis, A. J. Dos., Compiler Construction Using Java, JavaCC and Yacc. IEEE Computer Society Publications, Austin, Amerika, 2012.

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

29. Kakde, O. G., Algorithms for Compiler Design, First Edition, Charles River Media, Massachusetts, 2002.

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

31. https://javacc.org. 17 Mayıs 2019.

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

33. Parr, T., The Definitive ANTLR 4 Reference, The Pragmatic Programmers, LLC, Texas, USA, 2013.

34. https://www.cs.rochester.edu/~nelson/courses/csc_173/grammars/cfg.html Context-Free Grammars, 16 Mayıs 2019.

35. 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.

36. https://ipfs.io/ipfs/QmXoypizjW3WknFiJnKLwHCnL72vedxjQkDDP1mXWo6uco /wiki/Syntax-directed_translation.html Syntax Directed Translation. 5 Mayıs 2019.

37. Earley, J., An Efficient Context-Free Parsing Algorithm. Commun. ACM, 13 (1970) 94–102.

38. Sebesta R. W., Consepts of Programming Languages, 9. Baskı, University of Colorado, USA, 2010.

39. Donnelly, C., ve Stallman, R., Bison The Yacc Compatible Parser Generator, Free Software Foundation, Boston, USA, 2019.

40. http://dinosaur.compilertools.net/#lex Lex - A Lexical Analyzer Generator, 18 Mayıs 2019.

41. https://javacc.org The Java Parser Generator, 18 Mayıs 2019.

42. http://mathworld.wolfram.com/Integral.html Integral, 17 Mayıs 2019.

43. Granville, W. A., Elements of The Differential and Integral Calculus, The Athenaum Press, Boston, USA, 1911.

44. Abramowitz, M., ve Irene, S., Handbook of Mathematical Functions With Formulas, Graphs, and Mathematical Tables, Washington D.C., USA, 1972.

ÖZGEÇMİŞ

1983 yılında Trabzon'lu bir ailenin üçüncü çocuğu olarak Tekirdağ'ın Çorlu ilçesinde dünyaya geldi. İlköğrenimini Akçaabat Merkez İlkokulunda ve Akçaabat Mevlüt Selami Yardım Ortaokulunda, orta öğrenimini Akçaabat Anadolu Lisesi'nde tamamladı. 2010 yılında Karadeniz Teknik Üniversitesi Mühendislik Fakültesi Bilgisayar Mühendisliği Bölümü'nde lisans programını tamamladı. Kısa bir süre özel sektörde çalıştıktan sonra, Karadeniz Teknik Üniversitesi Fen Bilimleri Enstitüsü Bilgisayar Mühendisliği Anabilim Dalı'nda yüksek lisans programına başladı. 2012-2017 yılları arasında Iğdır Üniversitesi Mühendislik Fakültesi Bilgisayar Mühendisliği Bölümünde Araştırma Görevlisi olarak görev yaptı. Temmuz 2017'den itibaren Karadeniz Teknik Üniversitesi Uzaktan Eğitim Merkezinde Öğretim Görevlisi olarak görev yapmaktadır. Yabancı dil olarak İngilizce bilmektedir.