KARADENİZ TEKNİK ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI

SAYISAL HESAPLAMALARA YÖNELİK BİR PROGRAMLAMA DİLİNİN TASARIMI VE GERÇEKLEMESİ

DOKTORA TEZİ

Mehmet Cemil AYDOĞDU

TEMMUZ 2024
TRABZON

KARADENİZ TEKNİK ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI

SAYISAL HESAPLAMALARA YÖNELİK BİR PROGRAMLAMA DİLİNİN TASARIMI VE GERÇEKLEMESİ

Mehmet Cemil AYDOĞDU
ORCID : - - -

Karadeniz Teknik Üniversitesi Fen Bilimleri Enstitüsünce
"DOKTOR (BİLGİSAYAR MÜHENDİSLİĞİ)"
Unvanı Verilmesi İçin Kabul Edilen Tezdir.

Tezin Enstitüye Verildiği Tarih : 04 / 07 /2024
Tezin Savunma Tarihi : 30 / 07 /2024

Tez Danışmanı : Doç. Dr. Hüseyin PEHLİVAN
ORCID : - - -

Trabzon 2024

ÖNSÖZ

"Sayısal Hesaplamalara Yönelik Bir Programlama Dilinin Tasarımı ve Gerçeklemesi" isimli bu tez Karadeniz Teknik Üniversitesi Fen Bilimleri Enstitüsü Bilgisayar Mühendisliği Anabilim Dalı, Doktora Programı'nda hazırlanmıştır.

Tez çalışmam boyunca hiçbir desteğini esirgemeyen ve bilim insanı olma yolundaki ilk adımlarımı atarken kıymetli bilgileriyle ve tecrübesiyle yolumu aydınlatan danışman hocam Sayın Doç. Dr. Hüseyin PEHLİVAN'a teşekkürü bir borç bilirim. Ayrıca, yapıcı eleştirileri ve önerileri ile tezime büyük katkıları bulunan saygıdeğer hocalarım Sayın Dr. Öğr. Üyesi Murat AYKUT'a ve Sayın Dr. Öğr. Üyesi Mehmet ÖZTÜRK'e, eğitim öğretim hayatımda katkısı olan tüm hocalarıma teşekkür ederim.

Son olarak, doğduğum günden beri elimi hiç bırakmayan, hayatım boyunca aldığım tüm kararlarda bilgi ve tecrübelerinden faydalandığım ve her kararımda arkamda olan, maddi ve manevi desteklerini hiçbir zaman esirgemeyen ve bugünlere gelmemde en büyük paya sahip olan annem Zehra AYDOĞDU ve babam Kahraman AYDOĞDU'ya, ve özellikle her türlü desteğiyle yanımda olan hayat arkadaşım ve meslektaşım Dr. Özge AYDOĞDU'ya ve canım oğlum Mert Aren AYDOĞDU'ya teşekkür eder, minnettarlığımı sunarım.

Bu tez çalışmasının bundan sonraki çalışmalara katkı sağlamasını temenni ederim.

MEHMET CEMİL AYDOĞDU

Trabzon 2024

TEZ ETİK BEYANNAMESİ

Doktora Tezi olarak sunduğum “Sayısal Hesaplamalara Yönelik Bir Programlama Dilinin Tasarımı ve Gerçeklemesi” başlıklı bu çalışmayı baştan sona kadar danışmanım Doç. Dr. Hüseyin PEHLİVAN’ın sorumluluğunda tamamladığımı, verileri/örnekleri kendim topladığımı, deneyleri/analizleri ilgili laboratuvarlarda yaptığımı/yaptırdığımı, başka kaynaklardan aldığım bilgileri metinde ve kaynakçada eksiksiz olarak gösterdiğimi, çalışma sürecinde bilimsel araştırma ve etik kurallara uygun olarak davrandığımı ve aksinin ortaya çıkması durumunda her türlü yasal sonucu kabul ettiğimi beyan ederim. 30/07/2024

Mehmet Cemil AYDOĞDU

İÇİNDEKİLER

Sayfa No

ÖNSÖZ ... III
TEZ ETİK BEYANNAMESİ ... IV
İÇİNDEKİLER ... V
ÖZET ... VIII
SUMMARY ... IX
ŞEKİLLER DİZİNİ ... X
TABLOLAR DİZİNİ ... XI
SEMBOLLER VE KISALTMALAR DİZİNİ ... XIII
1. GENEL BİLGİLER ... 1
1.1. Sayısal Çözümleme ... 2
1.2. Programlama Dilleri ... 3
1.3. Sayısal Çözümleme için Kullanılan Programlama Dilleri, Çerçeveler ve Kütüphaneler ... 8
1.4. Programlama Dili Tasarımı ve Geliştirme Süreçleri ... 13
1.4.1. Programlama Dili Tasarımı ... 14
1.4.2. Programlama Dili Geliştirme Süreci ... 14
1.5. Dil İşlemcisi ... 15
1.5.1. Derleyiciler ... 15
1.5.2. Yorumlayıcılar ... 16
1.5.3. Derleyici ve Yorumlayıcı Kıyaslaması ... 17
1.6. Yorumlayıcı Aşamaları ... 18
1.6.1. Ayrıştırma ... 18
1.6.2. Sözdizimi ... 18
1.6.3. Dilbilgisi ... 20
1.6.4. Dillerin Hiyerarşisi ve Resmi Bilgisi ... 21
1.6.5. Sözcüksel Analiz ... 24
1.6.6. Sözcüksel Analiz Aşamaları ve Çalışması ... 27
1.6.7. Sözdizimi Analizi ... 29
1.6.8. Sözdizimi Analizi Aşamaları ... 30
1.6.9. Değerlendirme ... 32
1.6.10. Instanceof Operatörü ... 32

1.6.11. Eval Yöntemi 32
1.6.12. Ziyaretçi Tasarım Deseni 33
1.6.13. Anlamsal Analiz 35
1.6.14. Yorumlama 36
1.7. Ayrıştırıcı Üreteci Programları 38
1.7.1. Yacc 40
1.7.2. Bison 40
1.7.1. SableCC 41
1.7.1. ANTLR 41
1.7.1. JavaCC 42
1.8. Programlama Dili Değerlendirme Kriterleri 45
1.8.1. Okunabilirlik 46
1.8.2. Yazılabilirlik 46
1.8.3. Güvenilirlik 46
1.8.4. Sürdürülebilirlik 47
1.8.5. Ortogonallik 47
1.8.6. Tekdüzelik 47
1.8.7. Genişletilebilirlik 47
1.8.8. Verimlilik 48
1.8.9. Taşınabilirlik 48
2. YAPILAN ÇALIŞMALAR 49
2.1. Geliştirilen Dilin Söz Dizimi 51
2.2. Ayrıştırma 53
2.2.1. Dilbilgisi 53
2.2.2. Sözcüksel Analiz 54
2.2.3. Ayrıştırıcı 56
2.2.4. Nesne Sınıfları 57
2.2.5. Sözdizim Ağacı 61
2.3. Değerlendirici 63
2.3.1. Anlamsal Analiz 64
2.3.2. Yorumlama 68
2.3.3. Yorumlayıcı Entegrasyonu 74
2.4. Dilin Özellikleri 76
2.5. Örnek Programlar 78

2.5.1. Fibonacci Dizisi 79
2.5.1. Yarılama Yöntemi 80
2.5.1. Yer Değiştirme Yöntemi 81
2.5.1. Matris Transpozu 82
2.5.1. Determinant 83
3. BULGULAR VE TARTIŞMA 86
4. SONUÇLAR 92
5. ÖNERİLER 94
6. KAYNAKLAR 95
ÖZGEÇMİŞ

Doktora Tezi

ÖZET

SAYISAL HESAPLAMALARA YÖNELİK BİR PROGRAMLAMA DİLİNİN TASARIMI VE GERÇEKLEMESİ

Mehmet Cemil AYDOĞDU

Karadeniz Teknik Üniversitesi
Fen Bilimleri Enstitüsü
Bilgisayar Mühendisliği Anabilim Dalı
Danışman: Doç. Dr. Hüseyin PEHLİVAN
2024, 101 Sayfa

Sayısal çözümleme, birçok bilimsel ve mühendislik disiplininde kritik öneme sahip problemlerin çözümünde kullanılan temel bir araçtır. Matematik, mühendislik ve fen bilimleri gibi disiplinlerde karşılaşılan sorunların çoğu, matematiksel modellerle temsil edilerek sayısal çözümleme yöntemleri yardımıyla çözülmektedir. Bununla birlikte, sayısal yöntemlerin gerektirdiği matematiksel işlemler, modern dillerde uygulandığında karmaşık ve okunabilirliği zayıf büyük kod parçalarına yol açmaktadır. Özel amaçlı diller ise bu işlemleri kütüphane işlevleri aracılığıyla ele alarak diğer programlama ortamlarıyla zayıf entegrasyon, daha az programlama esnekliği ve pratiği sunmaktadır. Bu tez çalışmasında, bazı temel yapılar ve işlevler sağlanarak sayısal çözümleme yöntemlerinin programlanmasını kolaylaştıran bir programlama dilinin (MxPL) tasarımı ve geliştirilmesi amaçlanmıştır. MxPL dili sayısal çözümleme yöntemlerinin uygulanması için kullanıcı dostu, açık kaynak kodlu ve matematiksel notasyonlara benzer bir sözdizimine destek vermektedir. Çalışmada, matematiğin ve doğrusal cebirin olağan gösterimleriyle uyumlu bir dilbilgisi sunulmuş, ardından kaynak veri ayrıştırıcısı, doğrulayıcısı ve yorumlayıcısı geliştirilmiştir. Bazı program örnekleri üzerinde karmaşık sayısal çözümleme yöntemlerinin kodlanmasındaki performansı gösterilmiştir. Modern programlama dilleri ile temel dil özelliklerine dayalı bir karşılaştırmalı analiz yapılmıştır.

Anahtar Kelimeler: MxPL, Sayısal çözümleme, Programlama dili tasarımı

PhD. Thesis

SUMMARY

DESIGN AND IMPLEMENTATION OF A PROGRAMMING LANGUAGE FOR NUMERICAL COMPUTATIONS

Mehmet Cemil AYDOĞDU

Karadeniz Technical University
The Graduate School of Natural and Applied Sciences
Computer Engineering Graduate Program
Supervisor: Assoc. Prof. Hüseyin PEHLİVAN
2024, 101 Pages

Numerical analysis is a fundamental tool for solving critical problems in many scientific and engineering disciplines, making a solid foundation in it crucial. Problems in fields like mathematics, engineering, and natural sciences are often represented using mathematical models and solved using numerical analysis methods. However, the mathematical operations required by numerical methods, when implemented in modern programming languages, often lead to large pieces of code that are complex and poorly readable. Special-purpose languages, on the other hand, handle these operations through library functions, offering poor integration with other programming environments, less flexibility and practice. This thesis focuses on the design and development of a programming language (called MxPL) that facilitates the programming of numerical analysis methods by providing some basic constructs and functions. The MxPL language supports a user-friendly, open-source, and mathematical notation-like syntax, for the implementation of numerical analysis methods. The study presents a formal grammar compatible with the usual representations of mathematics and linear algebra, and then develops a parser, verifier and interpreter. Its performance in coding complex numerical analysis methods is demonstrated on some program examples. A comparative analysis based on basic language features is made with modern programming languages.

Key Words: MxPL, Numerical analysis, Programming language design

ŞEKİLLER DİZİNİ

Sayfa No

Şekil 1. Assembly Derleyicisi 16
Şekil 2. Yorumlayıcı 17
Şekil 3. Ayrıştırıcı öğeleri 29
Şekil 4. JavaCC 43
Şekil 5. MxPL genel çalışma şeması 50
Şekil 6. Sözdizim sınıf şeması 59

TABLOLAR DİZİNİ

Sayfa No

Tablo 1. Chomsky hiyerarşisi 21
Tablo 2. BNF ve EBNF örnek dilbilgisi 24
Tablo 3. Sözcüksel analiz terimleri 25
Tablo 4. Düzenli ifade örnekleri 26
Tablo 5. Sözcüksel analiz örneği 27
Tablo 6. $x = 5 + 2 * 3$ ifadesi için ayrıştırma ağacı 31
Tablo 7 Tarayıcı ve ayrıştırıcı üreteçleri özellikleri 39
Tablo 8. JavaCC yapılandırma dosyası 45
Tablo 9. MxPL dilbilgisi 53
Tablo 10. MxPL diline ait jeton listesi 54
Tablo 11. $f(x) = x + 1$ İfadesi için tarayıcı örneği 55
Tablo 12. $A = [1,2,3]$ İfadesi için tarayıcı örneği 55
Tablo 13. JavaCC dilbilgisi metotları 56
Tablo 14. Sözdizim sınıf tanımlamaları 60
Tablo 15. Sözdizim ağacını oluşturan aksiyon tanımları 62
Tablo 16. Ziyaretçi arayüzü 64
Tablo 17. Sembol tablosu 65
Tablo 18. Toplama işlemi için tür denetimi 67
Tablo 19. Toplama işlemi için *visit()* metodu 69
Tablo 20. Çarpma işlemi için *visit()* metodu 70
Tablo 21. Matris indis tanımları 71
Tablo 22. Matris indis kontrolleri yapan *chkIndex()* metodu 71
Tablo 23. Tembel değerlendirme sınıfı gövdesi 72
Tablo 24. Tembel değerlendirme için fonksiyon denkleminin belirlenmesi 73
Tablo 25. Yorumlayıcı çalıştırma parametreleri 74
Tablo 26. Yorumlayıcı çalıştırılabilir sınıf tanımı 75
Tablo 27. Yorumlayıcının komut satırı parametreleri ve işlevleri 76
Tablo 28. Matris veri türü 77
Tablo 29. Matris elemanlarına erişim örnekleri 78
Tablo 30. MxPL Fibonacci dizisi fonksiyon örnekleri 79

Tablo 31. Yarılama yöntemi için MxPL program örneği 81
Tablo 32. Yer değiştirme yöntemi için MxPL program örneği 82
Tablo 33. Matris transpozu fonksiyon örneği 83
Tablo 34. Determinant hesabı yapan Python kod örneği 84
Tablo 35. MxPL'de determinant hesaplayan program örneği 84
Tablo 36. Değerlendirme metrikleri ve programlama paradigmaları ile dillerin karşılaştırılması 86
Tablo 37. Belirli dil özellikleri üzerinden programlama dillerinin karşılaştırılması 88
Tablo 38. MxPL'in bazı sayısal hesaplama ortamları ile karşılaştırılması 90
Tablo 39. Popüler sayısal ortamlarda ve MxPL'de bazı operatörler ve işlemleri 91

SEMBOLLER VE KISALTMALAR DİZİNİ

EBNF : Genişletilmiş Backus-Naur formu

BNF : Backus-Naur formu

JavaCC : Java Compiler Compiler

1. GENEL BİLGİLER

Matematik mühendislik, finans ve bilişim gibi çeşitli teknoloji alanlarının temelinde yer alan bir sayı, uzay ve cebir bilimidir. Bu yüzden bilim ve teknoloji ilerledikçe matematik alt yapısının sağlamlığı ve eğitim süresi boyunca okullarda kavranması önem kazanmaktadır. Sayısal çözümleme ya da diğer adıyla nümerik analiz, klasik matematiğin bir dalıdır. Matematiğin analitik çözüm üretemediği veya ürettiği çözümün uygulama açısından yüksek karmaşıklık gösterdiği durumlarda sayısal hesaplama yöntemlerine başvurulur.

Sayısal çözümleme, matematiksel problemlerin yaklaşık çözümlerini bulmak için algoritmalar ve hesaplama teknikleri geliştiren önemli bir matematik disiplinidir. Bu problemler genellikle cebirsel denklemlerin köklerini bulma, integral hesaplama, diferansiyel denklemleri çözme gibi konuları içerir. Sayısal çözümleme tekniklerinin etkin bir şekilde uygulanması, güçlü ve kullanıcı dostu yazılım araçlarına ihtiyaç duymaktadır.

Matematiksel hesaplamalar için destek sunan yazılımlar ve buna bağlı olarak geliştirilen teknolojiler sayısal çözümleme yöntemlerinin uygulanabilmesini kolaylaştırmayı amaçlamaktadır. Ayrıca bu yazılımlar, farklı yöntemlerin sonuçlarını karşılaştırarak bir kıyaslama yapma imkânı da sağlayabilmektedir. Bununla birlikte, kullanıcılar, ara yüzün kısıtlamalarından kurtulmak için hazır araçlar kullanmak yerine bir programlama ortamında hesaplama algoritmalarını kodlamayı tercih edebilmektedir. Genel olarak algoritma kodlama bir yöntemin kavranmasında önemli bir rol oynar. Bu tekniklerin etkin bir şekilde uygulanması, kullanıcıların karmaşık algoritmaları kolayca ifade edebilecekleri, uygulayabilecekleri ve analiz edebilecekleri güçlü ve kullanıcı dostu yazılım ortamlarını gerektirmektedir. Mevcut programlama dilleri, sayısal çözümlemenin pedagojik ve şeffaflık gereksinimlerini tam olarak karşılayamamaktadır.

Bu eksikliğin temel nedeni, genel amaçlı programlama dillerinin, sayısal hesaplamaya özgü kavramları ve işlemleri doğal bir şekilde ifade etmek için tasarlanmamış olmasıdır. Örneğin, bu dillerde matrisler ve vektörler gibi temel veri yapıları için yerleşik destek bulunmayabilir veya sayısal yöntemlerin doğruluğunu ve kararlılığını etkileyen yuvarlama hataları ve sayısal hassasiyet gibi konuları ele almak için özel yapılar sunulmayabilir. Dahası, mevcut programlama dilleri, sayısal çözümleme algoritmalarının altında yatan matematiksel kavramları açık ve öz bir şekilde ifade etmekte zorlanabilmekte, özellikle

konuya yeni başlayanlar için öğrenme eğrisini daha dik hale getirerek algoritmaların uygulanmasında ve hata ayıklamasında zorluklara yol açabilmektedir. Bu durum, matematiksel ve doğrusal cebir notasyonundan uzaklaşmadan algoritmaların doğrudan programlanabilmesini sağlayacak alana özel bir programlama diline duyulan ihtiyacı ön plana çıkarmaktadır. Böyle bir dil, sayısal yöntemlerin uygulanmasını basitleştirerek, kullanıcıların algoritmaların matematiksel temellerine odaklanmalarını ve programlama dili sentaksı ve uzun kod bloklarıyla boğuşmalarını engelleyecektir. Bunun sonucunda, hem kodun okunabilirliği ve anlaşılabilirliği artacak, hem de geliştirme süreci hızlanacaktır. Ayrıca, sayısal yöntemlerin literatürdeki sunumuyla uygulama arasındaki uçurum kaldırılarak, araştırmacıların ve öğrencilerin yeni algoritmaları daha hızlı ve etkili bir şekilde öğrenmeleri ve uygulamaları mümkün hale gelecektir. Özetle, matematiksel notasyonu doğal bir şekilde destekleyen ve sayısal hesaplamalar için optimize edilmiş bir programlama dili, sayısal çözümleme alanında verimliliği ve ilerlemeyi önemli ölçüde artıracağı açıktır.

1.1. Sayısal Çözümleme

Sayısal çözümleme ya da sayısal analiz, karmaşık matematiksel problemlere sayısal yöntemler ile yaklaşık çözümler elde etme disiplinidir. Bilim ve mühendislik alanlarında karşılaşılan problemlerin çok büyük bir kısmı, analitik çözümlerle tam olarak ifade edilemeyecek kadar karmaşık veya imkansızdır. Bu tür problemlere çözüm bulmak için sayısal yaklaşımlar hayati önem taşımaktadır.

Pratik uygulamalarda bir bilim insanı ya da mühendis çalışmasının ya da projesinin sonuçlarını sayısal bir şekilde elde edebilmektedir. Örneğin, bir deney sonucunda elde edilen bir dizi tablo halindeki verilerden bir kestirimin yapılması ya da bir doğrusal cebirsel denklem sisteminin çözülmesi gerekebilir. Sayısal çözümlemenin amacı bu tarz problemlere sayısal yanıtlar verebilmek için en etkili yöntemleri sağlamaktır (Sastry, 2012).

Sayısal çözümleme, bilim ve mühendislik alanlarında çalışanlar için temel bir araç haline gelmiştir. Bu disiplin, karmaşık problemlere yaklaşık çözümler bulmak için güçlü bir yöntem sunar.

Sayısal yöntemler 1950'den önce yalnızca manuel hesaplamalarla uygulanabilmekteydi (Sastry, 2012). Fakat sıkıcı ve tekrarlanan aritmetik işlemler gerektirmesi, bilhassa karmaşık problemlerdeki manuel hesaplamalardaki hatanın fazla olmasından kaynaklı güvensizlik bu işlemleri otomatik ve hızlı yapacak teknolojilerin

gerekliliğini ortaya çıkarmıştır. Hesaplama teknolojisindeki muazzam ilerlemeler, sayısal çözümlemenin daha da popüler hale gelmesine yol açarak karmaşık problemlerin çözümü için çok sayıda bilimsel yazılım geliştirilmesine olanak sağlamıştır. Bu yazılımlar, problemlerin çözümünü hızlı, kolay, güvenilebilir ve verimli bir şekilde gerçekleştirilmesini sağlamaktadır.

Sayısal çözümleme, bilim ve mühendislikte geniş bir uygulama yelpazesine sahiptir. Bu alanlarda karşılaşılan problemlere örnek olarak şunlar verilebilir:

- Diferansiyel denklemlerin çözümü: Fizik, kimya ve mühendislikte birçok sistem, diferansiyel denklemlerle modellenmektedir. Sayısal analiz yöntemleri, bu denklemlerin sayısal çözümlerini bulmak için kullanılmaktadır. Bisection Yöntemi, Newton-Raphson Yöntemi, Gauss Eliminasyon Yöntemi, LU Ayrıştırması bu problemlerde kullanılan yöntemlere örnektir.
- Optimizasyon problemleri: Bir fonksiyonun maksimum veya minimum değerini bulmak gibi problemler, sayısal optimizasyon yöntemleri kullanılarak çözülebilir.
- Veri analizi: İstatistiksel yöntemler, büyük veri kümelerinden anlamlı bilgiler çıkarmak için kullanılır. Sayısal analiz, bu yöntemlerin uygulanmasında önemli rol oynar.
- Sayısal Türev: Bir fonksiyonun belirli bir noktadaki türevini yaklaşık olarak hesaplanması. Örnek olarak İleri Farklar ve Merkezi Farklar yöntemleri verilebilir.
- Sayısal İntegral: Bir fonksiyonun belirli bir aralıktaki integralini yaklaşık olarak hesaplanması gibi problemler. Trapez Yöntemi, Simpson Yöntemi, Monte Carlo İntegrasyonu gibi yöntemler bu probleme örnek teşkil etmektedir.
- Adi Diferansiyel Denklemler: Tek bir bağımsız değişkene bağlı diferansiyel denklemlerin çözülmesi. (Örnek: Euler Yöntemi, Runge-Kutta Yöntemi)
- Kısmi Diferansiyel Denklemler: Birden fazla bağımsız değişkene bağlı diferansiyel denklemlerin çözülmesi. (Örnek: Sonlu Farklar Yöntemi, Sonlu Elemanlar Yöntemi).

1.2. Programlama Dilleri

Programlama dili, programcılar tarafından bilgisayarlarla iletişim kurmak için kullanılan bir bilgisayar dilidir. Hem bilgisayar sistemlerinin hem de bu sistemlerin en önemli bileşeni olan bilgisayar programlarının geliştirilmesinde kritik role sahip olan

problem çözme araçlarıdır. Daha basit tabirle belirli bir görevi gerçekleştirmek için herhangi bir sentaksta yazılmış bir dizi talimat olarak da tanımlanabilir.

Bir programlama dili, sınırlı dilbilgisi kurallarıyla programcı ile makine arasında bir köprü görevi görür. Bu köprü, kısa ve öz talimatların karmaşık sonuçlara dönüşmesi ve eğitim, tıp ve güvenlik gibi alanları etkileyen bilgisayar sistemlerini ve programlarını şekillendirmesi nedeniyle hassas bir simetriyi bünyesinde barındırmaktadır. Bir programlama dili ile farklı mimarilere bilgisayar sistemine ya da birden çok bilgisayar sistemlerine kolaylıkla entegre edilebilecek programlar yazılması mümkün kılınabilinmektedir. Bilgisayar tabanlı sistemlerin bilim ve mühendisliğin birçok farklı alanında yaygın kullanımı, programlama dillerinin basitlik, verimlilik, soyutlama, kompaktlık, doğallık ve yerellik gibi birçok çekici özelliğe sahip olmasını zorunlu kılmaktadır. Ayrıca dil geliştiricilerinin okunabilirlik, yazılabilirlik, güvenilirlik, taşınabilirlik ve genişletilebilirlik gibi kaynak kodu geliştirme süreçlerinin verimliliğine doğrudan etki eden ve programlama dillerinin etkinliğini ve yaygın kullanımını artıran tasarım ilkelerini benimsemesi de önem taşımaktadır. Tasarım ilkeleri ile geniş bir programcı kitlesinin kaynak kodlara kolay adaptasyonunu sağlanabilmektedir (Louden & Lambert, 2011; Harper, 2016).

Programlama dilleri, literatürde farklı kriterlere göre sınıflandırılmaktadır. Bu sınıflandırmalardan biri de dillerin seviyelerine göre yapılan tasniftir. Bu tasnifte, diller yüksek seviyeli, orta seviyeli ve alçak seviyeli diller olmak üzere üç farklı kategoriye ayrılmaktadır.

Yüksek seviyeli diller insanların anlamasını ve yazmasını kolaylaştırmak için tasarlanmış bir programlama dilleridir. Bu diller, doğal dile daha yakındır ve daha soyut olan ve temeldeki bilgisayar donanımının belirli ayrıntılarına daha az bağımlı olan komutlar ve talimatlar kullanmaktadır. Yani daha yüksek düzeyde soyutlama sağlayarak geliştiricilerin bilgisayar sisteminin düşük düzeydeki ayrıntılarıyla uğraşması yerine, sorunları çözmeye odaklanmasına olanak tanımaktadırlar. Yüksek seviyeli dillerin insanlar tarafından anlaşılması daha kolaydır çünkü bir programdaki mantığı ve talimatları temsil etmek için birçok sembol, harf ve ifade kullanır. Bu diller programcıların daha okunabilir, bakımı kolay ve taşınabilir kod yazmasını sağlarlar. Yüksek seviyeli dillerin bazı örnekleri arasında Python (Van Rossum & Drake Jr., 1995), Java (Arnold vd., 2005) JavaScript (Haverbeke, 2018), PHP (Tatroe & MacIntyre; 2020), C# (Hejlsberg, 2010), C++ (Stroustrup, 1986), Objective C (Kochan, 2011), Cobol (Sammet, 1985), Perl (Wall vd., 2000), Pascal (Wakerly,

1979), LISP (DeMichiel & Gabriel, 1987), FORTRAN (Balman, 1981) ve Swift (Fojtik, 2020) gibi programlama dilleri yer almaktadır.

Orta seviyeli diller yüksek seviyeli ve alçak seviyeli diller arasında denge sunan bir programlama dilidir. Bu diller, düşük seviyeli dillere göre anlaşılması ve yazılması daha kolay olan soyutlamalar sağlarken donanım kaynaklarına doğrudan erişime de izin verir. Orta düzey bir programlama dilinin avantajları, yüksek seviyeli programlamanın özelliklerini desteklemesi, kullanıcı dostu bir dil olması, makine dili ve insan diliyle yakından ilişkili olmasıdır. Orta düzey dillerin örnekleri arasında, yüksek seviyeli dillerin basitliği ile alçak seviyeli dillerin ayrıntılı kontrolünü birleştiren C (Kernighan & Ritchie, 1988) ve C++ yer almaktadır.

Alçak seviyeli diller, donanımdan soyutlama yapmayan, makine komutları olan 0 veya 1 formlarıyla temsil edilen bir programlama dilleridir. Bu kategoriye giren diller Makine dili ve Assembly dilidir.

Bir diğer programlama dillerinin tasnifi ise dillerin yapısına göre ayrılmasıdır. Bu tasnif emirsel, nesneye dayalı, fonksiyonel ve mantıksal diller olmak üzere dört farklı kategoriden oluşmaktadır.

Emirsel diller prosedürel programlama dilleri olarak da ifade edilen dillerdir. Emirsel bir dil, istenen çıktıyı elde etmek için bir dizi ifadeyi veya komutu takip eder. Her adım dizisine prosedür adı verilir ve bu dillerden birinde yazılan bir programın içinde bir veya daha fazla prosedür bulunur. Emirsel dil, sorunları çözmek için sıralı, adım adım bir yaklaşım izleyen bir programlama dili türüdür. Böyle bir dilde kod, belirli bir sırayla çağrılabilen ve yürütülebilen prosedürler veya işlevler halinde düzenlenir. Yürütme akışı yukarıdan aşağıya bir yaklaşımı takip eder; bu, programın ilk talimattan başlayıp sırayla ilerlediği anlamına gelmektedir. Değişkenler verileri depolamak ve işlemek için kullanılır ve yürütme akışını kontrol etmek için döngüler ve koşullar gibi kontrol yapıları mevcuttur. En yaygın örnekleri olarak C ve PASCAL programlama dilleridir.

Nesne yönelimli programlama dilleri ise bir programı, nitelikler ve yöntemler olarak bilinen, veri ve program öğelerinden oluşan bir nesne grubu olarak ele almaktadır. Bu tarz diller nesnelere ve bu nesneler üzerinde işlem yapmaya odaklıdırlar. Nesneler bir program içinde veya başka programlarda yeniden kullanılabilir; bu durum kodun yeniden kullanılması ve ölçeklendirilmesi daha kolay olduğundan, onu karmaşık programlar için popüler bir dil türü haline getirmektedir. Java, Python, PHP, C++, C# ve Ruby (Matsumoto & Ishituka, 2002) en popüler nesne yönelimli programlama dilleridir.

Fonksiyonel programlama dilleri, ifadelerin yürütülmesine odaklanmak yerine, matematiksel fonksiyonların ve değerlendirmelerin çıktısına odaklanır. Yeniden kullanılabilen bir kod modülü olan her fonksiyon, belirli bir görevi gerçekleştirir ve buna bağlı bir sonuç döndürür. Sonuç, fonksiyona hangi verilerin girildiğine bağlı olarak değişiklik göstermektedir. Fonksiyonel diller kullanıcıya daha yakın olan dillerdir, ancak programların üzerinde çalışacağı makinelerin mimarisiyle nispeten ilgilenmezler. Literatürde yer alan popüler fonksiyonel programlama dilleri Scala (Wampler, 2021), Erlang (Armstrong, 1996), Haskell (Jones, 2003), Elixir (Fedrecheski, 2016) ve F# (Syme, 2011) ‘dır.

Bir diğer kategori olan mantıksal programlama dilleri bir bilgisayara ne yapması gerektiğini söylemek yerine bilgisayara nasıl karar vereceği konusunda talimat vermek için bir dizi olgunun ve kuralın ifade edilmesine dayanmaktadır. Prolog (Sterling & Shapiro, 1994), Absys, Datalog ve Alma-0 literatürde yer alan mantıksal programlama dillerine örnek dillerdir.

Bunların dışında literatürde programlama dilleri kullanım amaçlarına göre de sınıflandırılmaktadır. Genel kabul özel amaçlı ve genel amaçlı diller olmak üzere iki ana kategori altında toplandığı yönündedir.

Çok çeşitli alanların ihtiyaçlarını karşılayabilen programlama dillerine genel amaçlı programlama dilleri denilmektedir. Genel amaçlı diller, daha genel bir dizi sorunu çözebilecek yazılım uygulamaları geliştirmek için tasarlanmıştır. Genellikle farklı üst düzey yapılar tarafından desteklenen belirli bir programlama stilini tercih ederler. Bu diller birden fazla amacı yerine getirebilir; örneğin matematiksel hesaplamalara, araştırma çalışmalarına ve uygulama geliştirmeye aynı anda uygun olabilirler. Mesela bir genel amaçlı programlama dili olan Java, oyun geliştirmenin yanı sıra etkileşimli web sayfaları geliştirmek için de kullanılabilmektedir. Bir diğer dil olan C++ uygulama yazmak için kullanılabileceği gibi sistem programları geliştirmek için de yaygın kullanıma sahiptir. Python, Perl ve Ruby web programlamanın yanı sıra masaüstü uygulamalarının geliştirilmesi için de kullanılabilir. Ancak genel amaçlı programlama dillerinden her birinin kendine özgü bir uzmanlık alanı veya en iyi olduğu bir alan olmaktadır. Aslında genel amaçlı programlama dillerinin yukarıda verilen diğer programlama paradigmalarını da kapsadığı söylenebilinmektedir. Bir genel amaçlı programlama dili, birden fazla programlama paradigmasının simetrisini destekleyerek esneklik ve uyarlanabilirlik sağlayabilmektedir.

Özel amaçlı diller özellikle yorumlama veya yürütme sırasında üstün operasyonel performans sağlayan gerçek zamanlı uygulamaların geliştirilmesine hizmet etmektedir. Belirli bir sınıftaki sorunları çözmek için daha verimli ve daha kolay bir yol sağlayabilirler ve sistem güvenliğini artırmak için çalışma zamanı kodunu gizleyebilirler. Bu dilleri etki alanına özgü, uygun gösterimler ve soyutlamalar yoluyla, belirli bir sorun alanına odaklanan ve genellikle bununla sınırlı olan ifade gücü sunan bir programlama dili veya çalıştırılabilir belirtim dili olarak tanımlamak mümkündür (Oliveira, 2009). Özel amaçlı dillerin programlama alanları arasında belge biçimlendirmesi (Richter, 2014; Lamport, 1994, Berners-Lee vd., 2023), ilişkisel veri tabanı sorgusu (IBM Security QRadar, 2017; Johnson, (2005); Feuerstein & Pribyl, 2005), sembolik hesaplama (Hignam & Hignam, 2017; Mapplesoft, 2021; Trott, 2014; Bosma & Cannon, 1996), donanım açıklaması (Perry, 2002; Thomas & Moorby, 2002; Augustsson vd., 2001; Müller vd., 2007) ve animasyon ve modelleme (McKinley, 2006; Blain, 2020; Reinhart & Breton, 2009; Tavakkoli, 2018) yer almaktadır. Örneğin Giner-Miguelez ve arkadaşları (Giner-Miguelez vd., 2023) yaptıkları çalışmalarında makine öğrenimi veri kümelerini; veri yapıları veri kaynakları ve sosyal kaygılar açısından tam olarak tanımlayabilmek yeni bir alana özgü (özel amaçlı) dil geliştirmişlerdir. Bu dil ile yeni bir proje için en uygun veri kümesinin seçilmesi veya diğer makine öğrenimi sonuçlarının daha iyi kopyalanması gibi işlemlerde kolaylık sağlanacağını iddia etmişlerdir. Bir başka çalışmada adaptif web sistemlerinin yüksek düzeyde tanımlanması için özel bir dil geliştirilmiştir (Sadat-Mohtasham & Ghorbani, 2008). Kokash ve arkadaşları model tabanlı ilaç geliştirilmesinde uygulanması için özel amaçlı bir dil geliştirmişlerdir (Kokash vd., 2015). Lopez ve Martins ise kablosuz sensör ağları için tasarım gereği güvenli bir programlama dili ve çalışma zamanı sisteminin tasarımını, spesifikasyonunu ve uygulamasını sunmuşlardır (Lopez & Martins, 2016). Başka çalışmada, PObC++ adı verilen dağıtık bellekli paralel bilgi işlem platformları için nesne yönelimli paralel programlama dili geliştirilmiştir (Pinho & de Carvalho Junior, 2014). Louise tarafından çok çekirdekli sistemler için C türevi genel amaçlı bir veri akışı dili geliştirilmiştir (Louise, 2017). Vasic ve arkadaşları ise kimyasal kinetiği programlamaya yönelik CRN++ adında bir özel amaçlı dil geliştirmişlerdir (Vasic vd., 2020). Bunların dışında iş, ekonomi ve blok zinciri gibi birçok alanda literatürde yer alan özel amaçlı dil geliştirilmesi çalışmaları da mevcuttur.

1.3. Sayısal Çözümleme için Kullanılan Programlama Dilleri, Çerçeveler ve Kütüphaneler

Literatürde matematik, mühendislik ve eğitim alanlarında karmaşık problemlerin sayısal çözümlemesi yapabilmek, bunlara uyun modeller oluşturulabilmek ve sayısal çözümleme yöntemlerini kullanabilmek için geliştirilmiş birçok ortam ve yazılım paketi yer almaktadır. Ayrıca alana yönelik belirli sorunları çözmek amaçlı programlama dilleri de geliştirilmiştir.

Günümüzde matematiksel problemlerin çözümünün yapılabildiği birçok genel hesaplama yazılımları mevcuttur. Bunlardan en popüler olanı Matlab (Hignam & Hignam, 2017)' tır. Matlab, algoritmaların geliştirilmesinde ve bilimsel çalışmalarda yararlanılan popüler bir yazılımdır. Birçok amaç için kullanılabilen çok yönlü bir yazılım paketidir. Matlab basit matris odaklı bir yazılımdır ve büyük problemleri sayısal yöntemlerle çözmek için çok etkilidir (Kaukic, 2005). Matlab, hesaplamalarda yardımcı olan çok çeşitli sayısal fonksiyonlar sağlamaktadır. Matris manipülasyonuna, sayısal hesaplamalara, grafiksel gösterime, görüntü işlemeye ve büyük matematik problemlerinin çözümüne ve veri analizine izin vermektedir. Fakat büyük problemlerin pratik hesaplamaları için verimsiz kalmaktadır.

Maple (URL-2, 2021) matematiksel hesaplamalar için tasarlanmış bir yazılımdır. Hem sembolik hem de sayısal hesaplamaları yapılmasına olanak sağlamaktadır. Sayısal çözüm gerektiren problemler için Maple, gerçek sayılarla hesaplamalar yapmaya yönelik araçlar sağlar. Bu, denklemlerin köklerini bulma, integralleri sayısal olarak değerlendirme ve matris işlemlerini gerçekleştirme gibi görevleri içerir. Fakat Maple, limit, integral ve formüllerle türev hesabı, cebirsel ifadeleri basitleştirilmesi gibi sembolik işlemlerde çok daha efektiftir.

Mathematica (Trott, 2014), Maple' a benzer şekilde, teknik hesaplama için tasarlanmış bir yazılım sistemidir. Denklemlerin çözümlerinin bulunması, integral hesabı, bilinen veri noktaları arasındaki değerleri tahmin etmek için fonksiyonların sayısal olarak farklılaştırılması ve enterpolasyonların gerçekleştirilmesi, doğrusal denklem sistemlerinin çözülmesi, matris işlemlerinin gerçekleştirilmesi gibi sayısal çözümleme işlemleri için tasarlanmış geniş bir fonksiyon koleksiyonuna sahiptir. Ancak Mathematica ticari amaçlı kullanılmaktadır, açık kaynak kodlu bir yazlım değildir. Bu da geniş bir kitle tarafından erişiminde sorunlarına neden olmaktadır. Ayrıca geniş bir fonksiyon kütüphanesine sahip olması ve sentaks yapısı nedeniyle öğrenilmesi oldukça karmaşıktır.

GNU Octave, sayısal hesaplamalar sağlama amacı taşıyan yüksek seviyeli bir programlama dili ve kütüphanedir (Nagar & Nagar, 2018). Doğrusal ve doğrusal olmayan problemlerin sayısal çözümlenmesi için kullanılmaktadır. GNU Octave, Matlab'ın ortak açık kaynak seçeneklerinden biridir (Brewster & Gobbert, 2011). Karakteristik çağrıların bir listesini veya bir komut dosyasını içerir. Sözdizimi tamamen matris tabanlıdır ve matris işlemleri için çeşitli fonksiyonlar içermektedir. Büyük ve karmaşık hesaplamalarda yavaş kalabilmesi ve donanımsal/yazılımsal uyumluluk sorunları yaşayabilmesi GNU Octave'ın sorunlarından bazılarıdır.

SCILAB açık kaynaklı bir sayısal analiz paketi ve sayısal odaklı bir dildir (Enterprises, 2012). Bilimsel ve mühendislik amaçlı dostane bir ortam sağlar. İstatistiksel hesaplamalar, mühendislik formülasyonları, yüz tanıma, dinamik yapılar ve sembolik programlama gibi sayısal ve bilimsel hesaplamalar için kullanılır. Matlab gibi sinyal ve görüntü işleme üzerinde çalışabilmektedir. SCILAB, Matlab'a açık kaynaklı bir alternatiftir ve Matlab gibi tamamen matematiksel işlevlerde daha hızlı çalışır.

Analytica ise sayısal modeller oluşturulması ve analiz edilmesi için kullanılan yetenekleri kısıtlı bir ücretli araçtır. Karmaşık sayısal çözümleme yöntemlerini desteklemekten ziyade daha çok ticaret alanlarında modelleme yapma ve görselleştirme amaçlı kullanılmaktadır (Cox vd., 1999).

FreeMat hızlı mühendislik ve bilimsel işleme sayısal analiz için geliştirilmiş paket yazılımdır. Matlab ve Octave gibi diğer ticari paketlerle uyumlu olacak şekilde tasarlanmıştır (Brewster & Gobbert, 2011). Matlab ve IDL gibi ticari sistemlere benzer, ancak açık kaynaktır. Matlab yer alan özelliklerin birçoğunu desteklemektedir. Özdeğer ve tekil değer ayrıştırmaları, bölme operatörleri aracılığıyla doğrusal denklem sistemlerini çözme desteği yeteneklerine sahip olmasına rağmen, sayısal çözümleme tekniklerinin büyük çoğunluğunu desteklememektedir. Ayrıca, FreeMat, N boyutlu dizi manipülasyonu gerçekleştirebilmektedir, fakat N sayısı 6 ile sınırlıdır (Basu, (2009). FreeMat bazı sayısal yerleşik işlevlerden yoksundur (Brewster & Gobbert, 2011). Sözdizimi ile karşılaştırıldığında FReeMat, GNU Octave ve Matlab en çok benzerliğe sahiptir.

FreeMat gibi FlexPro'da veri analizi ve ölçüm için kullanılan bir araçtır. Ölçüm verilerinin hızlı ve kolay analizi ve sunumu için tasarlanmış veri analiz yazılımıdır.

Bu uygulamalar aynı zamanda eğitimsel ve bilimsel çalışmalarda yardımcı araç olarak da yaygın olarak kullanılmaktadır. Ancak öncelikle ticari amaçlarla geliştirilmişlerdir ve genel amaçlı programlama ortamlarına kolayca entegre edilemezler. Fonksiyon

kütüphaneleri ayrıntılı olarak belgelenmiş olmasına rağmen, matematiksel gösterimlerden çok farklı olabilecek programlama yapılarına ilişkin söz dizimi yeterince açıklanmamıştır. Genel amaçlı dillerle karşılaştırıldığında kaynak veriler, farklı anlamlara sahip olabilen sözdizimi bileşenleri içerebilmektedir. Ayrıca bu yazılımlardaki hazır hesaplama araçları, kullanıcının koda müdahale etmesine, yöntem üzerinde modifikasyon yapmasına olanak sağlamamaktadır. Bu dezavantajlar hazır paket yazılımlar dışında alternatif teknolojilere ihtiyaç oluşturmaktadır.

Hazır paket yazılımların dışında genel amaçlı dillerde sayısal çözümleme yöntemlerinin daha efektif kullanılmasını sağlayan birçok sayısal çözümleme kütüphaneleri mevcuttur. Bunlardan biri ALGLIB' dir. ALGLIB (Shearer & Wolfe, 1985) platformlar arası bir sayısal çözümleme ve veri işleme kütüphanesidir. C++, C#, Java, Python ve Delphi programlama dillerini desteklemektedir. ALGLIB ile veri analizi, optimizasyon ve doğrusal olmayan çözümleme, enterpolasyon ve doğrusal/doğrusal olmayan en küçük kareler uydurma, doğrusal cebir (doğrudan algoritmalar, EVD/SVD), doğrudan ve yinelemeli doğrusal çözümleme gibi birçok işlemler gerçekleştirilebilmektedir. Fakat bu işlemlerin birçoğu ücret gerektiren ticari sürümü ile yapılabilmektedir.

Python programlama dili için bilimsel hesaplama alanında NumPy (Harris vd., 2020) özel bir kütüphane olarak çok önemli bir konuma sahiptir. NumPy yüksek performanslı sayısal hesaplama gerçekleştirebilmek için geliştirilmiştir. Çok boyutlu matrisleri ve dizileri desteklemektedir. Bunlar kütüphanenin temelini oluşturmaktadır ve NumPy içerisinde bu ögeler üzerinde işlem yapacak üst düzey matematiksel fonksiyonlar yer almaktadır. NumPy, güçlü ve hızlı bir kütüphane olmasına rağmen, iş akışınıza entegre etmek için önceden önceden kurulmuş bir Python ortamı ve NumPy kitaplığını indirip yüklemek için Python Paket Yöneticisi (Pip) aracını kullanmanız gerekmektedir. Bununla birlikte, NumPy'ın büyük ölçüde fonksiyona dayalı yapısı, sayısal analiz yöntemleri, matris operasyonları ve ilgili algoritmaların pedagojik bir bakış açısıyla incelenmesi için ideal bir seçim olmayabileceği söylenebilir. SciPy ise NumPy' nin üzerine inşa edilmiş Python programlama dili için geliştirilmiş bir genel amaçlı bir kütüphanedir (Virtanen vd., 2020). İstatistik, optimizasyon, sinyal işleme ve görüntü işleme gibi sayısal çözümleme yöntemlerinin sıklıkla kullanıldığı alanlara bu yöntemleri gerçekleştiren özel fonksiyonlar sağlamaktadır.

Breeze (URL-1, 2023) ise sayısal hesaplama için oluşturulmuş bir Scala programlama dili kütüphanesidir. Ayrıca alana özel tasarlanmış ama yine de sayısal yöntemleri de içeren

makine öğrenmesi ve yapay zekâ uygulamaları için TensorFlow (Abadi, 2016) ve PyTorch (Imambi, 2021), ekonometri ve istatistiksel modelleme için Statsmodels (Seabold & Perktold, 2010) ve veri analizi ve veri işleme için Pandas (Snider & Swedo, 2004) kütüphaneleri de literatürde yer almaktadır.

Literatürde paket yazılım araçları ve kütüphaneler dışında sayısal hesaplamalar için geliştirilmiş programlama dilleri de yer almaktadır. Julia (Bezanson, 2017) sayısal ve bilimsel hesaplamada kullanılan açık kaynaklı bir programlama dilidir ve Massachusetts Teknoloji Enstitüsü (MIT) tarafından sertifikalanmıştır. İçerdiği çoğu özellik sayısal çözümleme ve veri bilimi için tasarlanmıştır. Sözdizimi, Python ve Ruby gibi programlama dillerine benzerdir.

Başka bir programlama dilleri ailesi matematiksel modelleme ve optimizasyon için geliştirilmiştir. Bazı özel örnekler AIMMS (Bisschop & Roelofs, 1999), AMPL (Fourer & Gay, 2002), LINGO (Schrage, 1999), GAMS (Bisschop & Meeraus, 1982), Mosel (Heipcke, 2002), LPL (Hürlimann 2007), OMNI (Kallrath, 2004), PCOMP (Kallrath, 2004) ve OPL' dir (Kallrath, 2004). Genel olarak modelleme dilleri, eşitlikler ve mantıksal ifadeler gibi uygun bir ilişkiler kümesi içeren matematiksel modeller aracılığıyla gerçek dünya sorunlarını temsil eder. Bunlar esas olarak problemleri matematiksel olarak resmileştirmek için gereken temel programlama yapılarıyla donatılmıştır. Ancak bu yapılar, sözdizimi ve anlambilim açısından tamamen matematiksel değildir; bazı kod bileşenleri, ifadelerin sıkı bir şekilde değerlendirilmesiyle birlikte farklı türde döngü ve seçim ifadeleri içerebilmektedir. Ayrıca diğer dil ortamlarına entegrasyon konusunda fazla esneklik ve destek sunmazlar. Bu bir zorluk olarak karşımıza çıkmaktadır.

Sayısal çözümleme ve bunu bilgisayarlar aracılığı ile gerçekleştirmede kullanılan en önemli öge matrislerdir. Matris manipülasyonu alanı, farklı yaklaşımları kapsayan zengin bir araştırma geçmişine sahiptir. Çalışmalar, genel amaçlı matris fonksiyon kitaplıkları, özel matris yazılımı ve özel matris programlama dilleri biçiminde ortaya çıkan, doğrudan veya dolaylı olarak matrisleri araştırır. Bunun bir örneği, matrisleri eş doğrusal cebir işlemleri yoluyla işlemek için özel olarak tasarlanmış resmi bir dil olan MATLANG' dir (Brijder, 2019). Özellikle MATLANG, bilimsel hesaplama bağlamlarında sıklıkla kullanılan matris ters çevirme gibi hesaplama açısından yoğun hesaplamaları hariç tutarak temel cebirsel işlemlere odaklanır.

Tensör cebri derlemesindeki önceki araştırmalara dayanarak yazarlar, matrisler de dahil olmak üzere seyrek dizileri işlemek için özel olarak tasarlanmış bir programlama

modelini hedefleyen yeni bir derleyici önermektedir (Henry vd., 2021). Katkıları, seyrek veri yapıları üzerinde çalışan dizi programları için genel bir derleyicinin yapımını göstermektedir.

Bir çalışmada yapısal ızgaralar üzerinde kısmi diferansiyel denklem (Partial differential equation, PDE) çözücülerini ifade etmek ve derlemek için özel olarak tasarlanmış yeni bir alana özgü dil (DSL) olan Mat2Stencil' i sunulmuştur (Cao vd., 2023). Bu yenilikçi çerçeve, yapılandırılmış seyrek matris soyutlamasından yararlanarak geniş bir yelpazedeki çözücü bileşenlerinin modüler, esnek ve kullanıcı dostu yapısına olanak tanımaktadır. Makale bir matris için dilbilgisi oluştururken, çalışma öncelikli olarak yapılandırılmış ızgaralardaki PDE çözücülerine odaklanmıştır. Benaddy ve El Habil temel matris işlemleri için özel olarak tasarlanmış yeni bir programlama dili önermiştir (Benaddy & El Habil, 2016). Sezgisel sözdizimi ve herhangi boyutlu matrisleri desteklemesi nedeniyle dilin lise matematikçileri için uygun olduğunu öne sürerken, makalelerinde geliştirme süreci ve kullanılan araçlara ilişkin ayrıntılı bir değerlendirme yer almamaktadır. Ayrıca sunulan örnek, matris cebri kavramlarının kapsamlı bir şekilde anlaşılmasını teşvik etmek yerine, önceden tanımlanmış matris işlemlerini gerçekleştirmeye yönelik bir yönelimi önermektedir.

Doornik ve arkadaşları özellikle ekonometri ve istatistik için tasarlanmış bir matris programlama dili olan Ox' u tanıtmışlardır (Doornik, 2001). Bu çok yönlü dil, C/C++' dan ilham alan bir sözdizimine ve kapsamlı bir matematiksel ve istatistiksel işlevler kitaplığına sahiptir ve matrislerin ve istatistiksel verilerin verimli bir şekilde işlenmesini kolaylaştırır. Matris işlemlerine odaklanmasına rağmen Ox, ekonometri ve istatistik topluluklarında yaygın bir şekilde benimsenmiştir. Akademik kullanım için ücretsiz bir sürüm mevcut olsa da Ox, kapalı kaynak lisanslama modeli altında faaliyet göstermektedir ve potansiyel olarak daha geniş kullanımını ve topluluk odaklı gelişimini sınırlamaktadır. Dil, en son sürümü Ox 6 (Doornik, 2009) olmak üzere sürekli bir evrim geçirmiştir.

MATX (Koga & Furuta, 1992) bilimsel ve mühendislik hesaplamaları için tasarlanmış yüksek performanslı bir dil olarak ortaya çıkmaktadır. Hem verimliliğe hem de kullanıcı dostu olmaya öncelik verirken, işlevselliği öncelikle temel matris tabanlı işlemlere odaklanmış görünmektedir. MATX' in matris işlemlerini açıkça hesaplamaması dikkat çekicidir; bu da eğitim amaçlı ve topluluk odaklı geliştirme çabalarına yönelik erişilebilirliğini sınırlayabilmektedir.

Literatür incelendiğinde sayısal çözümleme ve yöntemlerinin konu alan akademik çalışmalarında olduğu görülmektedir. Carroll çalışmasında çeşitli yazılımları ve programlama dillerinin sayısal metotların öğrenimine etkisini inceleyerek bazı eğiticilerin deneyimlerini sonuç olarak sunmuştur (Carroll, 1992). 2006 yılında Visual Basic programlama dilinde sayısal liberasyonların simülasyonu için bir uygulama Hassan ve arkadaşları tarafından gerçekleştirilmiştir (Hassan vd., 2006). Wlodkowski, Matcad ortamında sayısal metotların öğrenimine yönelik bir çalışma yapmış ve öğrencilerin sayısal yöntemin arkasındaki temel matematik bilgisini pekiştirmelerini sağladığı sonucuna varmıştır (Wlodkowski, 2006). Sayısal algoritmaların grafiksel simülasyonu adlı çalışmalarında matematik eğiticilerine kolaylık sağlamak için java dilinde geliştirdikleri ve GraSMa adını verdikleri yazılımı sunmuşlardır (Balsa vd., 2012).

Bunların dışında bilgisayarda uygulamalı matematik konusu ile ilgili birçok kitap yazılmıştır (Burden & Faires, 2011; Gilat & Subramaniam, 2014; James vd., 1993; Hoffmann, 1993; Fausett, 1999).

Sonuç olarak tarih boyunca bilgisayar ile hesaplama için pek çok dil ve uygulama geliştirilmiştir. Gerek daha genel çözümlerin bulunması gerekse daha hızlı yöntemlerin geliştirilmesi açısından araştırmalar sürmektedir. Bununla birlikte sayısal hesaplamanın imkânlarından faydalanılarak özellikle eğitim, bilim ve mühendislik alanında kullanılacak bilgisayar yazılımlarının geliştirilmesi için çalışmalar devam etmektedir. Literatürdeki çalışmalarda genellikle hazır fonksiyonlar, hazır program ara yüzleri kullanıcılar kısıtlanmaktadır. Bundan dolayı eğitim aracı olarak kullanıldıklarında sağladıkları fayda tartışılır olmaktadır. Bununla birlikte mevcut yöntemlerin farklı problemlere uyarlanması, yeni yöntemlerin geliştirilmesini kolaylaştırmamaktadır. Bu sebeple alan farklı yaklaşımlarla yeni çalışmaların yapılmasına ihtiyaç duymaktadır.

1.4. Programlama Dili Tasarımı ve Geliştirme Süreçleri

Bir programlama dili tasarlanması ve geliştirilmesi, yeni bir programlama dilinin oluşturulmasını içeren, dilin hedef kitlesini, kullanım alanlarını ve özelliklerini belirlemekten başlayan, dilin sözdizimi ve anlambilimini tanımlama ile devam eden, derleyici veya yorumlayıcı gibi araçlar geliştirmeye ve dilin belgelenmesi ile sonlanan birden çok aşamalı süreçler dizisidir. Aşamaların çokluğu dil geliştirme ve tasarlama sürecini daha da karmaşık hale getirmektedir.

1.4.1. Programlama Dili Tasarımı

Programlama dili tasarımı hedef kitle ve amaç belirleme, temel ve detaylı tasarım, araç geliştirme ve belgelendirme süreçlerinden oluşmaktadır. Bu süreçler bu bölümde kısaca açıklanmıştır.

Hedef Kitle ve Amaç Belirleme: Geliştirilen dilin kimler tarafından ve hangi amaçla kullanılacağının belirlenmesi ilk aşamadır. Bu aşama, dilin genel özellikleri ve tasarımıyla birlikte, sözdizimini de etkileyecek önemli bir faktördür.

Gereksinim Analizi: Bu aşama hedef kitle ve amaç belirlendikten sonra, dilin hangi özellikleri ve imkanları sunması gerektiğinin detaylı analizin gerçekleştiği aşamadır. Buradaki analiz, belirlenen amaç için mevcut programlama dillerinin eksikliklerini göz önünde bulundurarak gereksinimleri belirlenmesi için gerçekleştirilmektedir.

Temel ve Detaylı Tasarım: Gereksinim analizi sonucunda dile ait temel özellikler ve dilin sağlayacağı imkanlar belirlenir. Dilin temel unsurları olan sözdizimi ve anlambilimi, amaca uygun gelen veri yapıları, tip sistemi, kontrol akışı mekanizmaları ve giriş/çıkış işlemleri bu aşamada önce genel hatlarıyla tasarlanmaktadır. Daha sonra temel tasarım aşamasında belirlenen unsurlar detayları belirlenir ve unsurlar somutlaştırılır. Geliştirilen dile ait sözdizimi ve anlambiliminin formal bir tanımlaması yapılır. Gerekli fonksiyonlar, kontrol mekanizmaları ve API' ler ile veri yapıları ve veri tip sistemi detaylıca belirlenir.

Araç Geliştirme: Dil detaylıca tasarlandıktan sonra, dilin kaynak kodunu işleyebilecek ve çalıştırabilecek araçlar yani dil işlemcileri bu aşamada geliştirilir. Bu araçlar, derleyici, yorumlayıcı, sanal makine veya diğer türde bir yürütme ortamı olabilir. En çok kullanılanları derleyiciler ve yorumlayıcılardır.

Belgelendirme: Geliştirilen dile ait sözdizimi, anlambilimi, kütüphaneler ve dilin işlenmesinde kullanılan araçlar hakkında kapsamlı bir belgelendirme hazırlanır. Bu belge, dilin kullanıcılarının dili öğrenmelerine ve din kullanılmasına en büyük yardımcı olacak kaynaktır

1.4.2. Programlama Dili Geliştirme Süreci

Programlama dili geliştirme süreci prototip geliştirme, test ve hata ayıklama ve yayınlama ve bakım adımlarından meydana gelmektedir. Her bir adım kısaca aşağıdaki gibi açıklanmıştır.

Prototip Geliştirme: Dilin tasarımı tamamlandıktan sonra, dilin temel özelliklerini ve imkanlarını içeren bir prototip geliştirilir. Prototip, dilin işlevselliğini test etmek ve tasarımda herhangi bir hata veya eksiklik olup olmadığını belirlemek için kullanılır.

Test ve Hata Ayıklama: Prototip tamamlandıktan sonra, dilin kapsamlı bir şekilde test edilmesi ve hatalardan arındırılması elzemdir. Bu aşamada, çeşitli senaryolar belirlenerek dil icra edilir ve oluşan ya da oluşabilecek hatalar tespit edilip düzeltilir. Dilin test edilmesi ve hata ayıklanması sırasında kullanıcıların geri bildirimleri de ayrıca toplanır.

Yayınlama ve Bakım: Tüm aşamalar tamamlandıktan sonra, kullanıcıların erişebileceği şekilde dil yayınlanır. Fakat yayınlanma sonrasında da gerekli güncellemeler ve hata düzeltmeleri sunulmalıdır.

1.5. Dil İşlemcisi

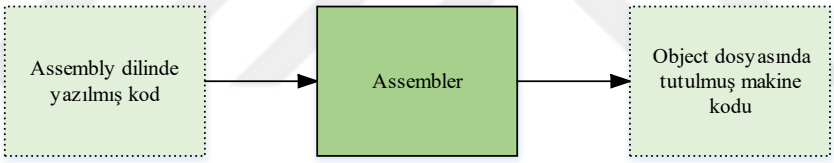
Programlama dili implementasyonu, bir programlama dilinin kaynak kodunu, bilgisayarın anlayabileceği ve çalıştırabileceği bir forma dönüştürülmesini gerektirmektedir. Bu yüzden insan-okunabilir kod ile bilgisayarın gerçekte çalıştırdığı talimatlar arasındaki köprü görevi gören dil işlemcilerine ihtiyaç vardır. Literatürde dil işlemcileri derleyici ve yorumlayıcı olarak ikiye ayrılmaktadır. Bunların dışında birleştiriciler, ön işlemciler, ayrıştırıcılar ya da hibrid modeller gibi farklı türde dil işlemcileri de mevcuttur (Sebesta, 2016). Bu başlık altında en çok kullanılan işlemciler olan derleyiciler ve yorumlayıcılar incelenmiştir.

1.5.1. Derleyiciler

Yüksek seviyeli bir dilde yazılmış kaynak programın tamamını tek seferde bir bütün olarak okuyan ve bunu makine dilinde eşdeğer bir programa çeviren dil işlemcisine derleyici denir. Derleyici kavramı 1950’ lerde Amerikalı bilgisayar bilimcisi Grace Hopper tarafından geliştirilmiştir (Sebesta, 2016). Hopper matematiksel gösterimi makine koduna çeviren A-0 Sistemi adı verilen ilk derleyiciyi yaratmıştır. Bu icat, programcıların makine kodu yerine insan tarafından okunabilen dillerde kod yazmasına olanak tanımıştır ve böylelikle programlamada devrim yaratıştır. Ayrıca yazılım geliştirmek daha hızlı ve daha erişilebilir hale gelmiştir.

Derleyicinin temel görevlerinden biri çevirme işlemi sırasında kaynak dilde tespit edilen hataların olup olmadığını dair rapor vermektir. Bir derleyicide kaynak kod hatasız olması durumunda başarıyla nesne koduna çevrilir. Kaynak kodda herhangi bir hata mevcutsa, derleyici derleme sonunda hataları satır numaralarıyla belirtir. Derleyicinin kaynak kodunu başarılı bir şekilde yeniden derleyebilmesi için hataların ortadan kaldırılması gerekmektedir.

Dil işlemcisi, bilgisayarın üst düzey bir dilde yazılmış olan kaynak programın tamamını tek seferde okuyarak programı çalıştırmasını ve anlamasını sağlar. Bilgisayar daha sonra bu kodu makine diline çevrildiği için yorumlayabilir. Her programlama dili için farklı bir derleyiciler mevcuttur. Ancak her derleyicinin gerçekleştirdiği temel görevler aynıdır. Bunlar, kaynak kodunu makine koduna çevirme süreci, sözcüksel analiz, sözdizimi analizi, anlamsal analiz, kod oluşturma ve optimizasyon dahil olmak üzere çeşitli aşamalardır. Genel olarak bir derleyicinin çalışma prensibi Şekil 1'deki gibidir.

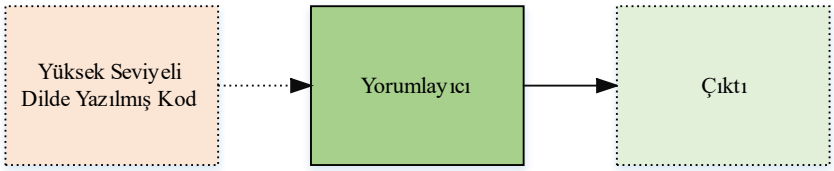


Şekil 1. Assembly Derleyicisi

1.5.2. Yorumlayıcılar

Kaynak programın tek bir ifadesinin makine koduna çevrilmesi bir dil işlemcisi tarafından yapılır ve bir sonraki satıra geçmeden hemen önce çalıştırılır, buna yorumlayıcı denir. Yani bir yorumlayıcı kaynak kodunu alır ve ardından satır satır okur, her kod satırını makine koduna çevirir ve bir sonraki satıra geçmeden önce onu çalıştırır. İfadede bir hata varsa bu ifadede çeviri işlemini sonlandırılır ve bir hata mesajı görüntülenir. Yorumlayıcı ancak hatanın giderilmesinden sonra uygulama için bir sonraki satıra geçer. Yorumlayıcı, bir programlama veya komut dosyası dilinde yazılmış talimatları, bunları önceden bir nesne

koduna veya makine koduna dönüştürmeden doğrudan yürütür ve bu nedenle başlangıçta bir derleyiciden daha hızlıdır. Yorumlayıcının çalışma prensibi Şekil 2'de verilmiştir.



Şekil 2. Yorumlayıcı

1.5.3. Derleyici ve Yorumlayıcı Kıyaslaması

Derleyici, programın tamamını tek bir bütün olarak alan ve kaynak kodunun tamamını CPU için çalıştırılabilir makine koduna dönüştüren bir dil işlemcisidir. Yorumlayıcı ise derleyiciden farklı olarak bir kaynak programı satır satır alıp çevirerek çalıştırmaktadır. Bir satır çevrildikten sonra diğerine geçmektedir.

Derleyici, kaynak kodunun genel olarak yürütülmesi açısından nispeten daha hızlıdır; ancak programlama kodunun tamamını derlemek için analiz yapmak çok zaman alır. Derleyiciyle karşılaştırıldığında, yorumlayıcı programlama kodunun genel olarak yürütülmesi açısından daha yavaştır, ancak kaynak kodu değerlendirmek daha az zaman alır.

Derleyici tüm programı taramayı tamamladığında hata mesajını üretir. Hata programdaki herhangi bir ifadede bulunabileceğinden; bu nedenle derleyiciyle hata ayıklamak nispeten zordur. Bir yorumlayıcı ise hatayla karşılaşılıncaya kadar programı dönüştürmeye devam eder; bu nedenle hata ayıklaması daha kolaydır.

Derleyicide programı çalıştırmak istendiğinde her defasında kaynak kodunu çevirecek bir dil programına ihtiyaç vardır. Fakat yorumlayıcıda ise programı çalıştırmak için kaynak kodu dönüştürmek için her defasında bir işlemci programına ihtiyaç duyulmaz.

Derleyici nesne kodunu (object code) saklar. Bu yüzden bellek ihtiyacı fazladır. Yorumlayıcı ise nesne kodunu tutmaz ve bellekten tasarruf sağlar.

Derleyici genelde ticari amaçlar için daha faydalıyken, yorumlayıcının ise öğrenme amaçlı kullanılması daha faydalıdır.

1.6. Yorumlayıcı Aşamaları

Bir yorumlayıcı görevini genel olarak 2 ana aşamada icra etmektedir. Bunlar ayrıştırma ve değerlendirme aşamalarıdır. Her aşama kendi içerisinde belirli işlemleri gerçekleştiren alt rutinlere ayrılmaktadır. Bu bölümde alt rutinler ve ana aşamalar ana başlıklar halinde detaylıca açıklanmıştır.

1.6.1. Ayrıştırma

Ayrıştırma, kaynak verilerin dilbilgisel analizi sürecidir ve sözcüksel analiz, sözdizimi analizinden oluşmaktadır. Kaynak kodunun programlama dilinin dilbilgisi kurallarına uyup uymadığını kontrol ettiği bir süreçtir. Bu nedenle ayrıştırma işlemi, bir programlama dilinin sözdizimi için resmi bir dilbilgisi tasarımını gerektirmektedir. Fakat ilk önce geliştirilen dilin amacına ve görevine uygun gelen sözdiziminin belirlenmesi ve buna bağlı olarak dilbilgisinin tasarlanması elzemdir.

1.6.2. Sözdizimi

Bir programlama dilinin iki özelliği sözdizimi ve anlambilimidir. Sözdizimi bir dilin anlatım yapısıyla ilgili bir konudur ve bir program listesinin o dilin geçerli bir programı olarak ilan edilmesi için uyması gereken kurallar dizisidir. Semantiği, geçerli programın arkasındaki anlam veya mantıktır. Bir bilgisayarın kabul edeceği programları yazabilmek için dilin sözdizimini bilmek gerekmektedir. Bir dilin sözdizimine uyulmazsa kod, yorumlayıcı tarafından anlaşılmayacaktır.

Aslında sözdizimi ifadelerin, ifadelerin ve program birimlerinin biçimi veya yapısıdır. Bir programlama dilinin amacına ve tasarlanmasındaki çıkış noktasına göre değişmektedir. Örneğin matematiksel veya cebirsel ifadelerin çözümünü üreten bir dil sözdiziminde matris yapılarını ve fonksiyonlarını içeren ifadeler ya da gerekli fonksiyonların tanımlayıcıları gibi ifadeler yer almalıdır. Bu nedenle bir programlama dilinin sözdizimi birden çok yapı taşları içermektedir.

Bunların ilki anahtar kelimelerdir (keywords). Anahtar kelimeler, dil içerisinde önceden tanımlanmış anlamları olan ayrılmış kelimelerdir. Örnekler arasında "if", "else",

"for", "while", "function", "class" vb. yer almaktadır. Bunları yanlış kullanmak sözdizimi hatalarına yol açabilmektedir.

Tanımlayıcılar ise değişkenlere, fonksiyonlara, sınıflara ve diğer program öğelerine verilen kullanıcı tanımlı adlardır. Her dilin harfle başlamak, büyük/küçük harf duyarlılığı kullanmak gibi kendi adlandırma kuralları mevcuttur.

Operatörler de veriler üzerinde işlem yapan sembollerdir. Yaygın türleri toplama, çıkarma, çarpma, bölme (+, -, *, /) aritmetik operatörleri; eşit, eşit değil, küçük, büyük, küçük veya eşit, büyük veya eşit (==, !=, <, >, <=, >=) karşılaştırma operatörleri; ve, veya, olumsuz (&&, ||, !) olan mantıksal operatörleridir.

Yapı taşlarından bir diğeri değerlerdir (values). Bunlar programın kullandığı sayılar (tamsayılar, kayan sayılar), metin dizeleri, boolean (Doğru/Yanlış) değerler gibi gerçek verilerdir.

Noktalı virgül ";", süslü parantez "{ }", parantez "( )" gibi noktalama işaretleri, kod bloklarını veya işlev çağrılarını sınırlamak için kullanılabilir. Kullanımları dile ve dilin amacına göre değişmektedir.

İfadeler ise bilgisayara ne yapması gerektiğini söyleyen eksiksiz talimatlardır. Anahtar kelimelerin, tanımlayıcıların, operatörlerin ve değerlerin dilin sözdizimine göre birleştirilmesiyle oluşturulurlar.

Mevcut programlama dillerinin sözdizimleri ve yapıtaşlarını kullanma şekilleri değişkenlik göstermektedir. Mesela Python gibi bazı diller, kod bloklarını (döngüler ve koşullu ifadeler gibi) tanımlamak için girinti kullanmaktadır. Geçerli bir sözdizimi için doğru girinti çok önemlidir. C ve Java gibi dillerde noktalı virgül ifadeleri sonlandırmak için kullanılmaktadır. Noktalı virgülü atlamak sözdizimi hatalarına yol açabilir. Bazı operatörlerin diğerlerinden daha yüksek önceliği vardır, bu da bir ifadede ilk önce değerlendirildikleri anlamına gelir. Doğru söz dizimini yazmak için operatör önceliği önem arz etmektedir.

Bir dilin sözdizimini tanımlamaya yönelik üç yaygın teknik vardır. Bunlar dilbilgisi, normal ifadeler (Sebesta, 2016) ve sonlu durum makineleridir (Thain, 2020). En yaygın olanı dilbilgisi tanımlamaktır.

1.6.3. Dilbilgisi

Programlama dilleri dünyasında dilbilgisi, bir programın sözdizimini veya yapısını tanımlamada hayati bir rol oynamaktadır. Dilbilgisi, geçerli program ifadelerinin nasıl oluşturulduğunu belirleyen bir kurallar dizisidir ve insan dilinde cümle yapısını nasıl tanımladığına benzerdir.

Dilbilgisi temel olarak iki temel unsurdan oluşur. Bunlardan ilk terminal (uç) sembolleridir. Uç semboller, dilbilgisi kullanılarak oluşturulan cümlelerin bileşenleri olan ve a, b, c gibi küçük harflerle gösterilen sembollerdir. Terminal, dilde görünebilen, jeton olarak da bilinen ayrı bir semboldür. Terminallere örnek olarak anahtar kelimeler, operatörler ve tanımlayıcılar verilebilir.

Diğer unsur ise terminal olmayan semboller adlandırılmaktadır. Terminal olmayan semboller, cümlenin oluşumunda yer alan ancak cümlenin bileşeni olmayan sembollerdir ve yardımcı semboller ya da değişkenler de denilmektedir. Terminal olmayan bir dilde oluşabilen ancak gerçek bir sembol olmayan bir yapıyı temsil eder. Terminal olmayanlara örnek olarak bildirimler, ifadeler verilebilir. Bu semboller A, B, C vb. gibi büyük harflerle temsil edilir.

Bir dilbilgisinin biçimsel tanımını yapabilmek için $D = < N, T, P, S >$ ifadesi kullanılmaktadır.

Burada;

- D, dilbilgisini ifade etmektedir.
- N, sonlu boş olmayan terminal olmayan semboller kümesidir.
- T, sonlu terminal sembolleri setidir.
- P, sonlu boş olmayan üretim kuralları setidir. Bilgisayar bilimindeki bir üretim veya üretim kuralı, yeni sembol dizileri oluşturmak için yinelemeli olarak gerçekleştirilebilecek bir sembol değişimini belirten bir yeniden yazma kuralıdır. $\alpha -> \beta$ biçimindedir; burada α, bir terminal veya terminal olmayan sembolleri dizisi olan β ile değiştirilebilen bir terminal olmayan semboldür.
- S, başlangıç sembolüdür.

Dilbilgisi, bir dilin dizelerini oluşturmak için kullanılan bir dizi üretim kuralı olduğundan, D dilbilgisi tarafından oluşturulan L(D) dili, D'den üretilen tüm dizilerin kümesini temsil etmektedir ve aşağıdaki 3 maddeyi sağlamalıdır.

1. Belirli bir D dilbilgisi, ona karşılık gelen L(D) dili benzersizdir.
2. D dilbilgisine karşılık gelen L(D) dili, D' den üretilebilecek tüm dizeleri içermelidir.
3. D dilbilgisine karşılık gelen L(D) dili, D' den üretilemeyen herhangi bir dize içermemelidir.

1.6.4. Dillerin Hiyerarşisi ve Resmi Bilgisi

Dillerin hiyerarşisi, programlama dillerini belli kriterlere göre tasnif eden ve onları organize eden bir sistem olarak tanımlanmaktadır. Hiyerarşi ile dillerin karmaşıklık düzeyleri ve diller arasındaki ilişkiler daha net anlaşılabilmektedir. Literatürde en yaygın olarak biçimsel dilleri sınıflandırmak için 1950'lerin ortalarında tanınmış dilbilimci olan Noam Chomsky tarafından ortaya atılan Chomsky Hiyerarşisi kullanılmaktadır. Bu hiyerarşi dilleri Tip0, Tip1, Tip2 ve Tip3 olmak üzere Tablo 1'deki gösterildiği gibi dört ana kategoriye ayırmaktadır.

Tablo 1. Chomsky hiyerarşisi

Tip	Dil	Makine	Karmaşıklık	Örnek
0	Özyinelemeli Diller	Turing Makinesi	Karar verilemez	Herhangi bir hesaplanabilir fonksiyon
1	Bağlama Duyarlı Diller	Doğrusal Bağlı Otomata	Üstel	$a^n b^n c^n$
2	Bağlam-Bağımsız Diller	Aşağı Sürüklemeli Otomata	Polinomial	$a^n b^n$
3	Düzenli Diller	Sonlu Otomata	Doğrusal	$a*$

Programlama dilleri, doğal dillerden farklı olarak daha kesin ve net bir tanımlama yapabilmek için resmi dilbilgisi kullanmaktadır. Resmi dilbilgisi, bir programın yapı

taşlarını ve bunların nasıl birleştirilebileceğini kategorize eden matematiksel sistemler olarak tanımlanabilmektedir. Literatürde Back Naus Form (BNF) ve İçerikten Bağımsız Dilbilgisi (Context Free Grammar, CFG) olarak aslında küçük farklılıklar dışında aynı olan iki tane resmi dilbilgisi yer almaktadır.

Noam Chomsky'nin ortaya attığı hiyerarşide yer alan içerikten bağımsız ve düzenli olarak dilbilgisi sınıflarından ikisinin, programlama dillerinin sözdizimini tanımlamak için yararlı olduğu ortaya çıkmıştır.

İçerikten bağımsız dilbilgisi (CFG), bir dilde izin verilen cümleleri resmi olarak tanımlayan kuralların bir listesidir. Terminal ve terminal olmayan semboller kuralların ana parçalarıdır. Her kuralın sol tarafı her zaman tek bir terminal değildir. Bir kuralın sağ tarafı, söz konusu terminal olmayanın izin verilen bir biçimini tanımlayan cümlesel bir biçimdir. Örneğin $A -> xXy$ kuralı, terminal olmayan A'nın bir x terminalini ve ardından bir terminal olmayan X'i ve bir y terminalini temsil ettiğini belirtir. Bir kuralın sağ tarafı, kuralın hiçbir şey üretmediğini belirtmek için $\mathcal{E}$ veya λ olabilir. CFG' de de ilk kural özel bir kuraldır: bir programın en üst düzeydeki tanımıdır ve terminal olmayanı başlangıç sembolü olarak bilinir.

Örnek bir CFG için, {a, b} alfabesindeki palindrom dizileri için bir dilbilgisi ele alınırsa, kurallar $S \rightarrow \lambda, S \rightarrow a, S \rightarrow b, S \rightarrow aSa, S \rightarrow bSb$ gibi olmaktadır. Örnek dilbilgisinde başlangıç sembolü S'dir. $S \rightarrow \lambda$ kuralını kullanarak λ elde edilir. λ bir (boş) terminal sembolü dizisi olduğundan, λ' nın L(D)' de olduğu sonucuna varılır. Benzer şekilde, a ve b dizilerini tek adımda türetilmektedir, dolayısıyla bunlar da L(D)' dedir. Aşağıda çok adımlı türetme abbba' nın L(D)' de olduğunu göstermektedir.

S |- aSa |- abSba |- abbba

BNF ise bir programlama dilinin sözdizimi kurallarını belirlemek için bir diğer dilbilgisi kullanma tekniğidir ve adını geliştiricileri John Backus ve Peter Naur'dan sonra Backus-Naur' dan almıştır. BNF içerikten bağımsız diller için eşdeğer bir gösterimdir.

BNF, programlama dilleri için bir meta dildir. Backus-Naur Formu veya Backus Normal Form anlamına gelen BNF, programlama dillerinin ve diğer resmi dillerin sözdizimini resmi olarak tanımlamak için kullanılan bir gösterimdir. Bir dilde oluşturulabilecek geçerli dizeler kümesini tanımlamanın bir yolunu sağlar ve esasen kod öğelerinin nasıl yapılandırılacağına ilişkin bir dizi kural gibi davranır.

BNF' nin temel bileşenleri CFG' ninkilerle aynıdır, fakat gösterim şekillerinde farklılıklar mevcuttur. Burada terminal olmayan semboller açılı parantezlerle "< >" temsil

edilir. Terminal semboller anahtar kelimeler ("if", "else", "while"), operatörler (+, -, *, /), noktalama işaretleri (;, {, }) gibi kodda görünen gerçek semboller, anahtar kelimeler veya değişmez değerlerdir (sayılar, dizeler). Üretim Kuralları terminal olmayanların, terminal ve diğer terminal olmayan dizilere nasıl genişletilebileceğini tanımlar. Bir üretim kuralının sol tarafından (Left-hand side, LHS) sağ tarafı (Right-hand side, RHS) ayırmak için burada da ok sembolü "→" kullanılmaktadır. LHS sadece bir tane terminal olmayandan oluşmaktadır. RHS ise oluşturulabileceği alternatif yolları gösterir.

EBNF' de {a, b} üzerindeki palindromlar kümesi aşağıdaki gibi gösterilebilir.

<palindromlar> → <boş>

<palindromlar> →a

<palindromlar>→ b

<palindromlar> → a<palindromlar>a

<palindromlar>→ b<palindromlar>b

<boş> gösterimi boş dizgeyi (*λ*) gösterir. Bu, palindromlar kümesinin yinelemeli bir tanımı olarak görülebilir. Temel durumlar *λ*, *a* ve *b*' dir. Özyinelemeli durumlar şu şekildedir; eğer bir palindrom s varsa, o zaman her iki ucunda birleştirilmiş bir a bulunan s bir palindromdur ve her iki ucunda bir b birleştirilmiş olan s bir palindromdur.

Dil sözdizimini belirlemek için Genişletilmiş Backus-Naur Formu (Extended BNF, EBNF) adı verilen genişletilmiş BNF bir sürümü kullanıma sunulmuştur. EBNF, programlama dillerinin ve diğer resmi dillerin sözdizimini açıklarken daha fazla esneklik ve ifade gücü sağlayan BNF' nin bir uzantısıdır. Üretim kuralları EBNF ile yazıldığında biraz daha basittir çünkü boş dizisine gerek yoktur ve genellikle çok fazla özyinelemeli üretim kuralı yoktur.

Gösterimdeki küçük bir fark, üretim kurallarını yazarken bazen sağ ok "→" yerine eşittir işaretinin "=" kullanılmasıdır. Alternatif bir gösterim de aynı şeyi ifade etmek için peş peşe iki nokta üst üste ve ardından eşittir işareti "::=" kullanmaktır. Ancak daha da önemlisi, EBNF çeşitli alternatiflerden birini seçme, sıfır veya bir kez ekleme ve sıfır veya daha fazla kez içermeyi desteklemektedir.

Alternatif seçme için dikey çubuk "|" sembolü kullanılmaktadır. Dikey çubuk daha önce dilbilgisinin iki tam versiyonu arasında seçim yapmak için kullanılıyordu. Ancak EBNF' de çubuk tek bir üretim kuralı dahilinde kullanılır. Bir sembolün isteğe bağlı olduğunu belirtmek için kullanılan gösterim, onu köşeli parantez "[ ]" içine almaktır. Birden fazla kez dahil edilebileceğini belirtmek için küme parantezleri "{ }" ifade içerisine alınarak

yazılmaktadır. Tablo 2'de EBNF ve BNF biçimlerinde aynı amacı gerçekleştiren dilbilgisi örneği verilmiştir.

Tablo 2. BNF ve EBNF örnek dilbilgisi

BNF	EBNF
<expr> → <expr> + <term> | <expr> - <term> | <term> <term> → <term> * <factor> | <term> / <factor> | <factor> <factor> → <exp> ** <factor> <exp> <exp> → (<expr>) | id	<expr> → <term> {(+ | -) <term>} <term> → <factor> {(* | /) <factor>} <factor> → <exp> { ** <exp>} <exp> → (<expr>) | id

1.6.5. Sözcüksel Analiz

Genellikle sözcük oluşturma veya tarama olarak adlandırılan sözcüksel analiz, derleyici oluşturma ve dil işlemenin karmaşık sürecinde temel aşamayı oluşturur. Rolü, bir karakter akışındaki (kaynak kodu) temel anlam birimlerini dikkatlice kazıp kategorize eden titiz bir arkeoloğunkine benzer. Jeton olarak adlandırılan bu kategorize edilmiş birimler, sonraki derleme ve yorumlama aşamalarının dayandığı temel yapı taşları olarak hizmet eder.

Sözcüksel analiz, bir programlama dilinin yorumlanma sürecinin ilk aşamasını oluşturur. Bir karakter akışını jeton adı verilen anlamlı birimlerin yapılandırılmış bir dizisine dönüştürerek, insan tarafından okunabilen kaynak kod ile sonraki aşamalar arasında köprü görevi görür. Başka bir deyişle, sözcüksel analiz girdi olarak aldığı kaynağı jeton adı verilen küçük fakat anlamlı birimlere parçalayan bir süreçtir. Bu aşamada, tasarlanan dilbilgisine başvurulmadan parçalama işlemleri gerçekleştirilmektedir.

Sözcüksel analiz aşamasında bilinmesi gereken üç önemli terim mevcuttur. Bunlardan ilki jetonlardır. Jeton, daha fazla parçalanamayan, önceden tanımlanmış bir karakter dizisidir. Bir birimi temsil eden soyut bir sembol gibidir. Bir jetonun isteğe bağlı bir nitelik değeri olabilir. Tanımlayıcılar (kullanıcı tanımlı), sınırlayıcılar/noktalama işaretleri (;, ,, {},

vb.), operatörler (+, -, *, / vb.), özel semboller (@, #, $ vb.), anahtar kelimeler ve sayılar gibi farklı türde jetonlar vardır.

Bir diğeri ise sözcük birimleridir (lexemes). Sözcük birimi, kaynak programda bir jetonun düzeniyle eşleşen bir karakter dizisidir. Örneğin (,) noktalama işaretinin jeton olduğu noktalama işareti türünde sözcük birimidir.

Desen ise, bir tarayıcının geçerli bir jetonu tanımlamak amacıyla giriş programındaki bir sözcük birimini eşleştirmek için izlediği bir dizi kuraldır. Bu, sözlüksel analizcinin bir sözcük birimini doğrulamak için kullandığı jetonun açıklamasına benzer. Örneğin, anahtar kelimedeki karakterler, anahtar kelimeyi tanımlayan desendir. Bir tanımlayıcıyı tanımlamak için, bir tanımlayıcı oluşturmak için önceden tanımlanmış kurallar dizisi desendir. Aşağıdaki Tablo 3' te sözcüksel analizdeki bu terimlerin örnekleri verilmiştir.

Tablo 3. Sözcüksel analiz terimleri

Jeton	**Sözcük birimi**	**Desen**
Anahtar kelime	while	w-h-i-l-e
Relop	<	<, >, >=, <=, !=, ==
Tam sayı	7	(0 - 9)*-> en az bir rakam içeren rakam dizisi
Katar	"Hi"	" " arasına yazılmış karakterler
Noktalama işareti	,	; , . ! etc.
Tanımlayıcı	number	A - Z, a - z Bir karakter tarafından başlatılan bir karakter ya da sayı dizisi

Sözcük analizcisinin yalnızca eldeki dile ait geçerli dizi/jeton/sözcük biriminin sınırlı bir kümesini araması ve tanımlaması gerekmektedir. Yani dil kuralları tarafından tanımlanan modeli aramaktadır. Sözcüksel analiz için bu model, dizi kümelerini tanımlamak için tasarlanmış cebirsel bir gösterim olan düzenli ifadeler (regular expressions, RE)

kullanılarak aranmakta ve yazılmaktadır. Düzenli ifadeler, desenleri belirtmek için önemli bir gösterimdir. Düzenli ifadeler, sonlu sembol dizileri için bir model tanımlayarak sonlu dilleri ifade etme yeteneğine sahiptir. Düzenli ifadelerle tanımlanan dilbilgisi, düzenli dilbilgisi olarak bilinir. Düzenli dilbilgisi ile tanımlanan dil, düzenli dil olarak bilinir. Düzenli bir R ifadesi, L(R) ile yazılan, R'nin dili adı verilen bir dizi setini belirtmektedir. Sözcüksel analizci oluşturucuları için yaygın olarak kullanılan düzenli ifade sözdizimini kullanarak, çeşitli yaygın düzenli ifade yapıları tarafından tanınan dil tanımlanabilmektedir. Tablo 4'te verilen düzenli ifade için tanımlanan düzenli dile ait belli başlı örnekler verilmiştir.

Tablo 4. Düzenli ifade örnekleri

Düzenli İfade	**Düzenli Dil**
düzenli	{düzenli}
d(o|i)g	{ dog, dig }
moo*	{moo, mooo, moooo,… }
(moo)*	{ε, moo, moomoo, moomoomoo, … }
a(b|a)*a	{aa, aaa, aba, aaaa, aaba, abaa, …}
s?	{s, ε}
(RE)+	{RE, RERE, RERERE, … }
[a-z]	{a, b, c, d, e, f, … z}
[^x]	x karakteri dışındaki karakterlerden biri

Bir kaynak veri için sözcüksel analiz işlemi gerçekleştirildiğinde, bu verinin birim kelime dizisini çıktı olarak üretir. Sözcüksel analiz işlemi, Tablo 5' te örnek bir jeton listesi kullanılarak bir girdi ifadesi üzerinde gösterilmiştir.

Tablo 5. Sözcüksel analiz örneği

Girdi:	A = 1+ 3
Jeton Listesi:	PLUS: '+' AEQ: '=' NUM: ([0-9])+ ID : [a-zA-Z]([a-zA-Z]|[0-9])*
Çıktı:	ID AEQ NUM PLUS NUM

1.6.6. Sözcüksel Analiz Aşamaları ve Çalışması

Bir derleyicinin başlangıç aşaması olan sözcüksel analiz birkaç önemli alt aşamaya ayrılabilir. Bu aşamalar, ham karakter akışını (kaynak kodu) ayrıştırıcının işleyeceği anlamlı jetonlar dizisine verimli bir şekilde dönüştürmek için uyum içinde çalışmaktadır

Tarama (Doğrusal Analiz): Bu aşama, kaynak kodunun karakter karakter incelemesini gerçekleştirir ve genellikle verimliliği artırmak için ara belleğe alma gibi teknikler uygulanır. Temel görevlerinden biri programın anlambilimine katkıda bulunmadığı için atılan boşluklar (boşluklar, sekmeler, yeni satırlar) ve yorumların da dahil olduğu ilgisiz karakterlerin kaldırılmasıdır. Bir diğer görevi ise, temel dil öğelerinin tanınmasıdır. Bu görev, anahtar sözcüklere, operatörlere, tanımlayıcılara ve değişmez değerlere (tamsayılar, kayan sayılar, dizeler vb.) karşılık gelen karakter dizilerinin tanımlanmasını içermektedir. Bu aşama esasen takip edilecek daha karmaşık örüntü tanıma için girdiyi hazırlamaktadır.

Desen Eşleştirme ve Jeton Oluşturma: Bu aşama, karakter akışındaki anlamlı sözcük birimlerini tanımlamak için genellikle düzenli ifadeler olarak ifade edilen önceden tanımlanmış kalıplardan yararlanılmaktadır. Örneğin, tanımlayıcıları tanımak için *[a-zA-Z_][a-zA-Z0-9_]** modeli kullanılabilir. Başarılı desen eşleştirmesinin ardından bir jeton oluşturulur. Bu jeton genellikle aşağıdakilerden oluşur:

- "IDENTIFIER", "KEYWORD", "OPERATOR", "INTEGER_LITERAL" vb. gibi eşleşen modelin kategorisini belirten jeton türü.
- Desen tarafından tanınan gerçek karakter dizisi olan sözcük birimi (örneğin, "myVariable" tanımlayıcısı veya "123" tam sayısı).

- Jetonun kaynak koddaki konumunu belirleyerek hata raporlamaya ve hata ayıklamaya yardımcı olan satır numarası ve konumu.

Sözcüksel Hata İşleme: Sözcüksel analiz, sözcüksel hataların, yani dilin sözcük kurallarının ihlallerinin belirlenmesinde ve raporlanmasında çok önemli bir rol oynamaktadır. Yaygın sözcüksel hatalar şunları içerir:

- Dilin alfabesinin parçası olmayan, tanınmayan karakterler olan geçersiz karakterler
- Bir rakamla başlayan bir tanımlayıcı gibi dilin sözcük yapısına uymayan hatalı biçimlendirilmiş jetonlar.
- Sonlandırılmamış değişmez değerler.

Bu tür hatalarla karşılaşıldığında sözcüksel analizci, genellikle hatalı karakteri/jetonu ve konumunu belirten bilgilendirici hata mesajları sağlamayı amaçlamaktadır. Hatayı kurtarmak için iyi tanımlanmış bir sınırlayıcı bulunana kadar karakterlerin atılması ya da karakter ekleyerek, silerek veya değiştirerek hatayı düzeltmeye çalışmak stratejilerine başvurulabilmektedir.

Sembol Tablosu Yönetimi: Bir yorumlayıcının veya derleyicinin çeviri yaşam döngüsü boyunca farklı aşamalar tarafından kullanılan önemli bir bileşen, sembol tablosudur. Sembol tablosu, programda kullanılan tanımlayıcılarla ilgili türleri, kapsamları ve bellek konumları gibi bilgileri saklamaktadır. Bu bilgi anlamsal analiz ve kod oluşturma gibi sonraki aşamalar için çok önem teşkil etmektedir.

Bir yorumlayıcının sözcüksel analiz uygulamasında iki farklı teknik bulunmaktadır. Bunlardan ilki Sonlu Durum Makineleridir. Genellikle bir dilin sözcüksel yapısını temsil etmek ve jeton tanıma sürecini yönlendirmek için kullanılır. Diğeri ise sözcük analizi oluşturucularıdır (Sözcüksel Araçlar). Lex ve Flex gibi araçlar, geliştiricilerin düzenli ifadeler kullanarak sözcüksel kuralları belirlemesine olanak tanıyarak sözcüksel analizörlerin geliştirilmesini basitleştirir. Bu araçlar sözcüksel analiz aşaması için otomatik olarak verimli kod üretir.

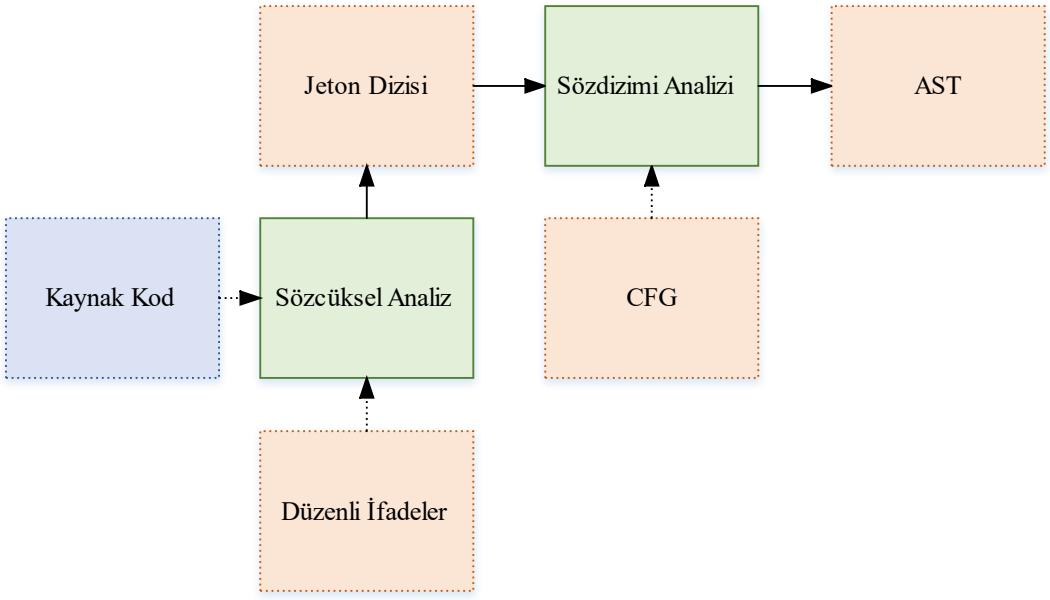
Genel olarak sözcüksel analiz, yorumlama sürecinin bekçisi görevi görür. Dilin sözcük kurallarını uygular, temel dil yapılarını tanımlar ve sonraki ayrıştırma ve anlamsal analiz aşamaları için yapılandırılmış bir jeton akışı sağlar. Verimli ve doğru sözcüksel analiz, bir yorumlayıcının genel performansı ve doğruluğu açısından temeldir.

1.6.7. Sözdizimi Analizi

Ayrıştırma olarak da bilinen sözdizimi analizi, derleyici/yorumlayıcı tasarımında temel bir aşamadır. Başlıca rolü, kaynak kodunun yapısını resmi dilbilgisi kurallarına göre analiz ederek sözdizimsel doğruluğunu sağlamaktır. Bu aşamada dile ait tanımlanan resmi dilbilgisinin rolü çok büyüktür.

Sözdizimi analizi aşaması dilbilgisi doğrulaması ve yapısal temsilden sorumludur. Dilbilgisi doğrulaması, kaynak kodunun programlama dilinin dilbilgisi kurallarına uygunluğunun sağlanmasıdır. Aslında bu, yorumlayıcının dilin sözcük dağarcığını ve dilbilgisini doğru kullanıp kullanmadığınızı kontrol etmesi olarak düşünülebilir. Yapısal temsil ise kodun, genellikle ayrıştırma ağacı veya soyut sözdizimi ağacı (Abstract Syntax Tree, AST) biçiminde yapılandırılmış bir temsilinin oluşturulmasıdır. Bu ağaçlar, farklı kod öğeleri arasındaki hiyerarşik ilişkiyi tasvir ederek sonraki aşamaların kodun anlamını anlamasını ve işlemesini kolaylaştırmaktadır.

Bir sözdizimi analizcisi veya ayrıştırıcısı, girdiyi jeton akışları biçiminde bir sözcüksel çözümleyiciden alır. Ayrıştırıcı, koddaki hataları tespit etmek için kaynak kodunu (jeton akışı) üretim kurallarına göre analiz eder. Bunun için resmi dilbilgisinden yardım alır. Bu aşamanın çıktısı bir ayrıştırma ağacıdır. Şekil 3' te ayrıştırıcının çalışma prensibi gösterilmektedir.



Şekil 3. Ayrıştırıcı öğeleri

Ayrıştırıcı resmi dilbilgisinden türetme işlemini kullanarak faydalanmaktadır. Türetme, temel olarak girdi dizesini elde etmek için kullanılan bir dizi üretim kuralıdır. Ayrıştırma sırasında bazı cümlesel girdi biçimleri için hangi terminalin değiştirileceğine karar vermek ve terminal olmayanın değiştirileceği üretim kuralına karar vermek olarak iki farklı karar mekanizması yer almaktadır.

Hangi terminal olmayanın üretim kuralıyla değiştirileceğine karar vermek için en soldan türetme ve en sağdan türetme olmak üzere iki seçenek mevcuttur.

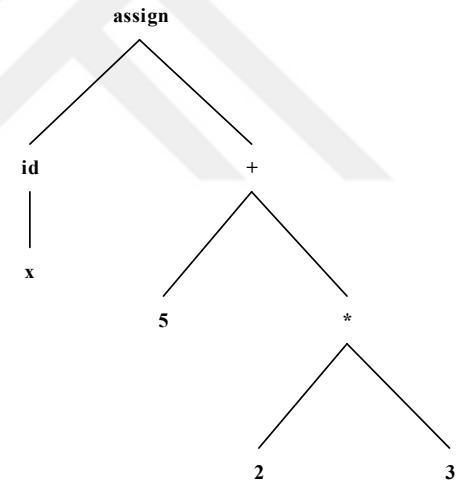
1.6.8. Sözdizimi Analizi Aşamaları

Bir sözdizimi analizcisi ya da ayrıştırıcının çalışma prensibi birkaç aşama ile maddeler halinde verilmiştir.

1. Ayrıştırıcı, kaynak kodunu zaten anahtar kelimeler, tanımlayıcılar, operatörler ve değişmez değerler gibi anlamlı birimlere (jetonlara) bölmüş olan sözcüksel analizciden girdi olarak alır.
2. Daha sonra, dilin sözdizimini yöneten kuralları tanımlayan, genellikle bağlamdan bağımsız bir dilbilgisi, BNF ya da EBNF olan resmi bir dilbilgisi kullanır. Dilbilgisi, geçerli program yapıları oluşturmak için jetonların nasıl birleştirilebileceğini belirtir.
3. Ayrıştırıcı, bağlamdan bağımsız dilbilgisi, BNF ya da EBNF aracılığıyla tanımlanan üretim kurallarını takip eder. Üretim kurallarının uygulanma şekli yani türetme, ayrıştırmayı iki türe ayırır: yukarıdan aşağıya ayrıştırma (LL(k)) ve aşağıdan yukarıya ayrıştırma (LR(k)). Yukarıdan Aşağıya Ayrıştırma, dilbilgisinin başlangıç sembolüyle başlar ve giriş jetonu akışını türetmeye çalışır. Aşağıdan yukarıya ayrıştırma ise giriş jetonlarıyla başlar ve bunları başlangıç sembolüne indirgemek için dilbilgisi kurallarını tersten kullanır. LL(k) ve LR(k)'nın her ikisinin de ortak olarak aldığı k parametresi ayrıştırıcının bir kural seçmek için ihtiyaç duyduğu ileri bakış miktarıdır. k parametresine bağlı olarak aynı anda k kadar jeton seçimi yapılmaktadır.
4. Ayrıştırma ilerledikçe bir AST oluşturulur. AST, bir ayrıştırıcının bir kod dizesini resmi bir dilbilgisine göre nasıl yorumladığının görsel bir temsilidir. Kodun sözdizimsel yapısını yansıtan hiyerarşik bir ağaç yapısıdır. AST; dil bilgisinin başlangıç sembolünü temsil eden kök düğüm, dilbilgisindeki terminal olmayan

sembolleri temsil eden (örneğin, "ifade", "ifade", "if-ifadesi") dahili düğümler, giriş kodundaki gerçek jetonları veya sözcükleri temsil eden (örneğin, anahtar kelimeler, tanımlayıcılar, operatörler) yaprak düğümleri (terminaller) ve ayrıştırma sırasında dil bilgisi kurallarının uygulanmasından elde edilen ebeveyn-çocuk ilişkilerini göstermek için düğümleri bağlayan kenarlardan oluşmaktadır. Basit bir $x = 5 + 2 * 3$ aritmetik ifadesi için dilbilgisi ve ona karşılık gelen ayrıştırma ağacını Tablo 6' da ki gibidir.

Tablo 6. $x = 5 + 2 * 3$ ifadesi için ayrıştırma ağacı

Girdi: $x = 5 + 2 * 3$	
Dilbilgisi	Ağaç Gösterimi
assignment -> identifier '=' expression expression -> term ('+' | '-') term term -> factor ('*' | '/') factor factor -> identifier | number	

Ayrıştırıcı bu işlemleri yaparken dikkat etmesi gereken en önemli durum belirsiz dilbilgisidir Bir girdi verisi (katar/dizi) için birden fazla ayrıştırma ağacı üretilebilen dilbilgisiler ile karşılaşılabilmektedir. Bu dilbilgisi için belirsiz dilbilgisi adı verilmektedir ve belirsiz dillerin de tanımlanasına neden olmaktadır. Belirsiz diller, ayrıştırıcılar için büyük bir sorun teşkil etmektedir. Belirsizliği otomatik olarak algılayıp, düzelten yöntemlerin olmaması nedeniyle dil bilgisinin belirsizliği ortadan kaldıracak şekilde yeniden tasarlanması ya da bu duruma dikkat edilerek en baştan yazılması gerekmektedir.

1.6.9. Değerlendirme

Yorumlayıcı oluşturma alanında, değerlendirme süreci iki ayrı ancak birbirine bağlı aşama olan anlamsal analiz ve kod yorumlama olarak kavramsallaştırılabilir. Bu aşamalar tipik olarak iki özel dil işleme bileşeni olan sırasıyla anlamsal çözümleyici ve kod yorumlayıcı aracılığıyla gerçekleştirilir. Her iki bileşen de ayrıştırma aşamasında oluşturulan ayrıştırma ağacı dolaşarak çalışmaktadır. AST verilerini farklı formlarda yorumlayan bu bileşenlerin uygulanması için çeşitli yöntemler mevcuttur. Bunlardan en yaygınları *instanceof* operatörü, Eval yöntemi eklenmesi ve davranışsal tasarım deseni olan Ziyaretçi (Visitor) Tasarım Desenidir. Bu bölümde değerlendirme adımlarına geçmeden önce bu desenler öncelikli olarak açıklanmıştır.

1.6.10. Instanceof Operatörü

Bu yöntemde, bir düğümün içerdiği nesnenin türünü belirlemek için *instanceof* operatörü kullanılır. *instanceof* operatörü, bir nesnenin belirli bir sınıfın örneği olup olmadığını kontrol eder ve sonuç olarak *true* veya *false* değerini döndürür. Bu sayede, düğümün hangi türde bir nesne içerdiği dinamik olarak belirlenebilir. Ancak, nesnenin gerçekleştirdiği işlemi doğru bir şekilde gerçekleştirebilmek için, üst sınıf referans değişkeninden alt sınıf nesnesinin türüne dönüştürülmesi gerekebilir. Bu işlem, tür dönüştürme (cast) yapısı kullanılarak gerçekleştirilir. Tür dönüştürme işlemi, bir nesnenin farklı bir tür olarak ele alınmasını sağlar. Bu sayede, alt sınıf nesnesine ait metotlara ve özelliklere erişim sağlanabilir ve ilgili işlem gerçekleştirilebilir. Ancak, tür dönüştürme işleminin dikkatli bir şekilde kullanılması önemlidir. Eğer bir nesne yanlış türe dönüştürülmeye çalışılırsa, *ClassCastException* hatası meydana gelir. Bu nedenle, tür dönüştürme işlemi gerçekleştirmeden önce *instanceof* operatörü ile nesnenin türünün kontrol edilmesi önerilir.

1.6.11. Eval Yöntemi

Bu yöntem, her bir sözdizimi sınıfına, o sınıfın temsil ettiği işlemi gerçekleştiren bir *eval()* metodu ekleyerek çalışır. Bu yaklaşım, nesne yönelimli programlamanın temel prensiplerine dayanır ve kodun daha okunaklı ve bakımı kolay olmasını sağlar.

Örneğin, bir Toplama sınıfı, iki sayıyı toplayan bir *eval()* metoduna sahip olabilirken, bir Çıkarma sınıfı ise çıkarma işlemini gerçekleştiren bir *eval()* metoduna sahip olacaktır.

Bir nesne ağacı düğümünün değerlendirilmesi gerektiğinde, düğümün içerdiği nesnenin *eval()* metodu çağrılır. Böylece, hangi işlemin gerçekleştirileceğini belirlemek için düğümün türünü kontrol etmeye gerek kalmaz. *eval()* metodu, nesnenin türüne göre otomatik olarak doğru işlemi gerçekleştirecektir. Bu da kodun daha az karmaşık ve daha kolay anlaşılır olmasını sağlar.

Bu yöntemin bir diğer avantajı da yeni işlem türleri eklemenin daha kolay olmasıdır. Yeni bir işlem eklemek için, sadece ilgili işlemi gerçekleştiren bir *eval()* metodu içeren yeni bir sınıf oluşturmak yeterlidir. Mevcut koda herhangi bir değişiklik yapmaya gerek yoktur. Bu da kodun daha esnek ve genişletilebilir olmasını sağlar. Buna karşın birçok kod tekrarı oluşabilir ve test edilmesi zordur.

1.6.12. Ziyaretçi Tasarım Deseni

Amaca yönelik bir ziyaretçi sınıfının oluşturulması dayanır. Bu yöntemde ziyaretçi sınıfının *visit* metotları ile düğümün *accept* metotları birbirlerini çağırır. Bu yöntem eval yöntemine göre daha esnektir, daha az kod tekrarı içerir ve test edilmesi daha kolaydır. Bu yöntem iki ana bileşenden oluşur:

Ziyaretçi tasarım deseni, davranışsal tasarım deseni ailesinin önde gelen üyelerinden biridir ve Erich Gamma tarafından tanıtılmıştır (Gamma vd., 1995). Ziyaretçi tasarımı hiyerarşik veri yapısını yorumlamak için kullanılan oldukça yaygın bir davranışsal tasarım şablonudur. Önemi, ortak bir tür hiyerarşisini paylaşan bir nesne koleksiyonu üzerinde işlem uygulama ihtiyacı ortaya çıktığında ve aynı anda işlem mantığı ile altta yatan nesne yapısı arasında net bir ayrım sağlamayı amaçladığında ortaya çıkar. Bu desen, Açık/Kapalı İlkesine bağlı kaldığı için, işlem kümesinin değişime veya genişlemeye açık olduğu durumlarda özellikle değerlidir.

Ziyaretçi deseni, özünde, işlem yürütme sorumluluğunu "ziyaretçi" olarak bilinen ayrı bir arayüzüne devrederek bu ayrımı başarır. Bu yetkilendirme iki temel bileşen aracılığıyla gerçekleştirilir:

1. *visit()* Metodu: Somut ziyaretçi sınıfları tarafından uygulanan bu metotlar, her işlem için belirli mantığı kapsar. Her *visit()* metodu, tipik olarak nesne yapısı içindeki somut bir eleman türüne karşılık gelir ve türe özgü işlemlere olanak tanır.

2. *accept()* Metodu: Her "ziyaret edilebilir" sınıf içinde tanımlanan (yani, nesne yapısı içindeki somut sınıflar) *accept()* metodu, ziyaretçiler için giriş noktası görevi görür. Bir ziyaretçi nesnesini argüman olarak alır ve o ziyaretçi üzerinde uygun *visit()* metodunu çağırarak hem ziyaretçinin türüne hem de somut elemanın türüne göre işlemi etkili bir şekilde dağıtır.

Bu mekanizma sayesinde işlem mantığı tamamen ziyaretçi sınıfları içinde yer alır ve nesne yapısının kendisi dokunulmadan kalır. Sonuç olarak, mevcut nesne yapısında değişiklik yapılmasına veya nesnenin metotları içinde koşullu ifadeler eklenmesine gerek kalmadan, yalnızca yeni ziyaretçi sınıfları tanımlanarak yeni işlemler eklenebilir.

Yorumlayıcının değerlendirme adımında Ziyaretçi Deseni iki amaca hizmet etmek için kullanılmaktadır.

1. İşlem ve Yapının Ayrıştırılması: Ziyaretçi deseni, ağaç üzerinde dolaşma (traversal) işlemini, her düğümde gerçekleştirilecek belirli eylemlerden ayrıştırır. Bu sayede, dolaşma algoritması nesne yapısının iç yapısından bağımsız hale gelir ve daha modüler ve yeniden kullanılabilir bir yapı elde edilir.
2. Eylem Gruplandırması: Ziyaretçi deseni, bir aşamaya ait tüm eylemleri (işlemleri) tek bir ziyaretçi (visitor) sınıfı altında toplar. Bu durum, kodun daha düzenli ve anlaşılır olmasını sağlar. Ayrıca, yeni eylemler eklemek veya mevcut eylemleri değiştirmek gerektiğinde, sadece ilgili ziyaretçi sınıfı üzerinde değişiklik yapılır, ana nesne yapısı etkilenmez.

Ayrıştırma ağacı değerlendirmesinde, Ziyaretçi tasarım şablonu, kodun düzenliliğini ve esnekliğini artıran önemli avantajlar sunar. Bu şablon, yeni değerlendiricilerin mevcut yapıya müdahale etmeden eklenmesini kolaylaştırarak, açık/kapalı prensibini destekler.

Ziyaretçi deseni uygulandığında, her bir ayrıştırma ağacı düğüm sınıfı için özel bir *visit()* metodu içeren bir ziyaretçi nesnesi oluşturulur. Bu yapı, her bir AST sınıfının, gelen ziyaretçi nesnesini uygun *visit()* metoduna yönlendiren bir *accept()* metodu bulundurmasını gerektirir.

Program akışı, ziyaretçi ve AST düğümleri arasında dinamik olarak ilerler. Ziyaretçi tarafından çağrılan düğümün *accept()* metodu, düğümün türüne göre doğru *visit()* metodunu çağırarak çift yönlü bir gönderim (double dispatch) mekanizması oluşturur. Bu süreç, tüm AST yapısı gezilinceye kadar devam eder.

Ziyaretçi tasarım şablonunun AST değerlendirmesinde sağladığı temel avantajlar şunlardır:

- Kod Düzenliliği ve Okunabilirliği: Ziyaretçi deseni, her bir değerlendirme işlemini ilgili ziyaretçi sınıfı içinde izole ederek kodun daha düzenli ve okunabilir olmasını sağlar.
- Esneklik ve Genişletilebilirlik: Yeni bir değerlendirme işlemi eklemek için, yeni bir ziyaretçi sınıfı oluşturulması yeterlidir. Mevcut AST yapısında veya diğer değerlendiricilerde herhangi bir değişiklik yapılması gerekmez.
- Açık/Kapalı Prensibine Uygunluk: Ziyaretçi deseni, mevcut yapı değiştirilmeden yeni işlevlerin eklenebilmesini sağlayarak, açık/kapalı prensibini destekler.

Sonuç olarak Ziyaretçi tasarım şablonu, AST değerlendirmesinde esneklik, genişletilebilirlik ve kod düzenliliği sağlayarak, karmaşık yazılım sistemlerinin geliştirilmesinde ve sürdürülmesinde önemli avantajlar sunar. Bu yöntem eval yöntemine göre daha esnektir, daha az kod tekrarı içerir ve test edilmesi daha kolaydır.

1.6.13. Anlamsal Analiz

Anlamsal analiz, derleyici tasarımında sözdizimsel doğruluk ile program anlamı arasındaki boşluğu dolduran çok önemli bir aşamadır. Ayrıştırma, bir programın dilbilgisi kurallarına uyup uymadığını kontrol ederken anlamsal analiz, programın dilin tanımına göre anlamlı olmasını sağlamak için daha derinlere iner.

Anlamsal Analiz, programın bildirimlerinin ve ifadelerinin anlamsal olarak doğru olduğundan emin olmayı amaçlar. Dil bilgisinin gerektirdiği şekilde ayrıştırıcı tarafından çağrılan prosedürlerin bir koleksiyonudur. Verilen kodun tutarlılığını kontrol etmek için hem önceki aşamanın sözdizimi ağacı hem de sembol tablosu kullanılır. Tür kontrolü, derleyicinin her operatörün eşleşen işlenenlere sahip olduğundan emin olduğu anlamsal analizin önemli bir parçasıdır.

Anlamsal analizci verilen programın anlamsal olarak dil tanımıyla tutarlı olup olmadığını kontrol etmek için sözdizimi ağacını ve sembol tablosunu kullanır. Tür bilgilerini toplar ve sözdizimi ağacında veya sembol tablosunda daha sonra kullanmak üzere saklar. Çıktısı açıklamalı ve muhtemelen değiştirilmiş bir ayrıştırıcı ağacıdır.

Dilbilgisi yapısının ötesinde dillerin, bileşenlerin nasıl etkileşime girdiğine ilişkin kuralları vardır. Anlamsal analiz bu kuralları uygulayarak aşağıdaki gibi hataları tespit eder:

- Tür uyuşmazlığı. Örneğin Bir tam sayının beklendiği yerde bir kayan nokta değişkeni kullanma.
- Bildirimden önce değişkenlere referans verilmesi.
- İşlevlere yanlış argümanların iletilmesi.
- Kapsamı dışında bir değişkene erişim.
- Ayrılmış tanımlayıcının kötüye kullanılması.
- Bir kapsamda birden fazla değişken bildirimi.
- Gerçek ve resmi parametre uyuşmazlığı.

Anlamsal analiz sırasında;

1. Her üretim kuralıyla ilişkili anlamsal işlevleri değerlendirerek ayrıştırma ağacını dolaşılır. Bu süreç, öznitelik değerlerini ağaçta yukarı ve aşağı yayarak anlamsal bilgiyi programın farklı bölümlerine etkili bir şekilde taşır.
2. Öznitelik yayılımı sırasında, anlamsal işlevler içinde kodlanan anlamsal kuralları uygulanır. Bunlar;
    - Bildirimden önce değişkenlere referans verilmesi.
    - İşlevlere yanlış argümanların iletilmesi.
    - Kapsamı dışında bir değişkene erişim.

Bir örnekle açıklamak gerekirse sözcük ve sözdizimi analizinden sorunsuz geçen *x = y + 5;* ifadesi için anlamsan analizci *y*' nin bildirilip bildirilmediğini ve türünün eklemeyle uyumlu olup olmadığını kontrol eder. *x'* in bildirilip bildirilmediğini ve türünün toplama sonucuyla uyumlu olup olmadığını kontrol eder. Her iki denetim de başarılı olursa sembol tablosunu *x* türüyle günceller.

1.6.14. Yorumlama

Yorumlama aşaması, bir yorumlayıcıda işlerin pratiğe döküldüğü yerdir. Yorumlayıcının, kodun yapısını ve anlamını anlamaktan, talimatları gerçekten yürütmeye ve sonuç üretmeye geçiş yaptığı bir aşamadır. Genellikle değerlendirici, ayrıştırılmış kodun yapılandırılmış bir temsilini, örneğin önceki aşamalarda genellikle anlamsal bilgilerle (tür

ek açıklamaları, tanımlayıcı bağlamaları) zenginleştirilmiş bir AST alır. Değerlendiricinin bir çalışma zamanı ortamına veya durumuna erişmesi gerekir. Bunun için değişkenler ve fonksiyon adlarıyla ilişkili değerleri aramak için sembol tablosu, fonksiyon çağrılarını, parametreleri ve dönüş adreslerini yönetmek için fonksiyon çağrı yığını ve Dinamik olarak ayrılan veriler için yığın belleği içerir.

Yorumlama aşamasında AST veya diğer kod temsilini sistematik olarak dolaşılmaktadır. Karşılaşılan her düğüm veya talimat için yorumlama gerçekleştirilir. Talimat/düğüm ve işlenenler alınır. Talimatın anlamını yani hangi işlemin gerçekleştirileceği yorumlanır. Belirtilen işlem gerçekleştirilir. Bu işlemler aşağıda maddeler halinde verilen işlemlerden biri olabilir.

- Aritmetik/Mantıksal hesaplamalar.
- Değişken aramaları ve atamaları.
- Fonksiyon çağrıları.
- Bellek yönetimi (ayırma/boşaltma).
- Kontrol akışı değişiklikleri (dallanma, döngü)

Değerlendirici, değişken değerlerindeki, çağrı yığınındaki vb. değişiklikleri yansıtan, talimatlar yürütüldükçe ortamı/durumu günceller. Bir yorumlama aşamasının birincil çıktısı programın yürütülmesinin sonucudur. Bunlar kodda tanımlandığı gibi görüntülenen değerler, değiştirilen veriler veya gerçekleştirilen eylemler olabilmektedir. Değerlendirme sırasında hatalar oluşursa (örneğin, sıfıra bölme, tür uyuşmazlığı), bunları tespit etmek ve rapor etmekten değerlendirici sorumludur.

Örnek bir x = 5 + 2 * 3; ifadesi için değerlendirici şu işlemleri gerçekleştirmektedir:

- 2 * 3 ifadesini değerlendirir (çarpma önceliklidir), sonuç 6 olur.
- 5 + 6 ifadesini değerlendirir, sonuç 11 olur.
- Sembol tablosunda x değişkenini arar. Yoksa yeni bir girdi oluşturur.
- 11 değerini x ile ilişkili sembol tablosu girdisine depolar

Değerlendirici aşaması, bir yorumlayıcının kalbidir ve talimatları titizlikle yürüterek ve programın çalışma zamanı ortamını yöneterek kodu hayata geçirir.

1.7. Ayrıştırıcı Üreteci Programları

Ayrıştırıcı tasarımı, bilgisayar bilimleri içerisinde programlama dili geliştirme alanının temel ve zorlu bir uğraşını temsil eder. Bir programlama dilinin başarılı bir şekilde uygulanması, büyük ölçüde etkili ve verimli bir ayrıştırıcıya dayanır. Geliştiricilere bu karmaşık görevi kolaylaştırmak amacıyla hem lisanslı hem de açık kaynak kodlu çok sayıda ayrıştırıcı üreteci yazılımı mevcuttur. Bu araçlar, geliştiricilerin ayrıştırıcıları sıfırdan oluşturma yükünü ortadan kaldırarak, programlama dili tasarımı ve uygulamasının daha üst düzey yönlerine odaklanmalarını sağlar.

Ayrıştırıcı üreteçleri, geliştiricilerin programlama dilleri ve diğer formal diller için ayrıştırıcıları kolayca oluşturmalarını sağlayan güçlü araçlardır. Bu alandaki iki popüler seçenek Yacc (Yet Another Compiler Compiler) ve Lex' tir (Lexical Analyzer Generator). Yacc, belirtilen bir dilbilgisine göre C kodunda bir ayrıştırıcı üretirken, Lex, düzenli ifadelere dayalı olarak bir sözcüksel analiz aracı (tokenizer veya lexer) üretir. Her iki araç da C programlama dili ile entegre çalışarak C tabanlı ayrıştırıcıların geliştirilmesinde sıklıkla tercih edilir.

Yacc ve Lex' in C tabanlı olmasının yanı sıra, Java tabanlı ayrıştırıcı üreteçleri de mevcuttur. JavaCC ve SableCC, bu tür araçlara örnek olarak verilebilir. Hem JavaCC hem de SableCC, ücretsiz ve açık kaynaklıdır, bu da onları geniş bir geliştirici kitlesi için erişilebilir kılmaktadır. Ayrıca, internette ve çeşitli kaynaklarda bu araçların kullanımı hakkında kapsamlı belgelendirme ve örnekler bulunmaktadır, bu da öğrenme sürecini kolaylaştırır. Geliştiriciler, projelerinin gereksinimlerine ve dil tercihlerine bağlı olarak C tabanlı (Yacc, Lex) veya Java tabanlı (JavaCC, SableCC) ayrıştırıcı üreteçleri arasından seçim yapabilirler.

Ayrıştırıcı üreteçleri, girdi olarak aldıkları biçimsel gramer tanımını kullanarak, belirtilen dilin yapısını tanıyabilen ve işleyebilen ayrıştırıcı programları otomatik olarak oluştururlar. Bu üreteçler, genellikle iki temel ayrıştırma yaklaşımından birini benimser: yukarıdan aşağıya ayrıştırma ve aşağıdan yukarıya ayrıştırma.

Yukarıdan aşağıya ayrıştırıcı üreteçleri, girdiyi çözümlemeye başlangıç sembolü ile başlar ve dilbilgisi kurallarını kullanarak, girdiyi türetmeye çalışır. Bu yaklaşım, genellikle özyinelemeli türetmeli ayrıştırma (LL ayrıştırma) tekniğini kullanır.

Aşağıdan yukarıya ayrıştırıcı üreteçleri ise, girdiyi çözümlemeye en alt seviyedeki yapı taşlarıyla (genellikle tokenlar) başlar ve dilbilgisi kurallarını kullanarak, bu yapı

taşlarını daha üst seviyedeki yapılara indirgemeye çalışır. Bu yaklaşım, genellikle kaydırmalı-indirgemeli ayrıştırma (LR ayrıştırma) tekniğini kullanır.

Ayrıştırıcı üreteçleri, sadece kullandıkları ayrıştırma tekniği bakımından değil, aynı zamanda geliştirme dili, destekledikleri dilbilgisi özellikleri ve ürettikleri ayrıştırıcıların performans karakteristikleri açısından da farklılık gösterir. Örneğin, bazı üreteçler belirli programlama dillerine (C, Java gibi) özgü ayrıştırıcılar üretirken, bazıları daha genel amaçlı olabilir.

Sonuç olarak, geliştiriciler bir ayrıştırıcı üreteci seçerken, proje gereksinimlerini, performans beklentilerini ve kişisel tercihlerini göz önünde bulundurarak en uygun seçeneği belirlemelidirler. Tablo 7 tarayıcı ve ayrıştırıcı üreteçlerinin bir özetini içermektedir.

Tablo 7 Tarayıcı ve ayrıştırıcı üreteçleri özellikleri

TARAYICI ÜRETEÇLERİ			
	Algoritma	**Dil**	**Lisans**
Lex	DFA	C	Proprietary, CDDL
Flex	DFA	C & C++	BSD
JFlex	DFA	Java	GNU GPL

AYRIŞTICI ÜRETEÇLERİ				
	Algoritma	**Dilbilgisi**	**Dil**	**Lisans**
Yacc	LALR(1)	YACC	C	CPL & CDDL
Bison	LALR(1), LR(1), IELR(1), GLR	YACC	C, C++	GNU GPL
JavaCC	LL(k)	EBNF	Java, C++, JavaScript (GWT derleyici ile)	BSD
SableCC	LALR(1)	EBNF	C, C++, C#, Java, OCaml, Python	GNU LGPL
ANTLR	Adaptive LL(*)	EBNF	C#, Java, Python, JavaScript, C++, Swift, Go, PHP	BSD

1.7.1. Yacc

Yacc, özellikle LALR (Look-Ahead LR) ayrıştırma algoritmasını kullanarak ayrıştırıcılar oluşturmak için yaygın olarak tercih edilen güçlü bir araçtır (Dos Reis, 2012). Yacc, girdi olarak bir BNF (Backus-Naur Form) dilbilgisi alır ve bu dilbilgisi kurallarına göre verimli bir şekilde çalışan bir ayrıştırıcı oluşturmak için C kaynak kodu üretir.

Yacc' ın kendisi sözcüksel analizi (lexing) ele almaz; bunun yerine, genellikle Lex veya Flex gibi ayrı bir sözcüksel analiz aracıyla birlikte kullanılır. Lex ve Flex, düzenli ifadeler kullanarak tanımlanan kalıplara göre girdi metnini jetonlarına ayırmak için kullanılır. Yacc, daha sonra bu jetonları girdi olarak alır ve daha büyük bir programlama dilinin yapısal temsilini oluşturmak için bu jetonları birleştirir.

Yacc, C programlama dilinde geliştirilen yazılımlarda ayrıştırıcı oluşturmak için sıklıkla tercih edilen bir araçtır ve birçok derleyici ve yorumlayıcının ayrıştırma aşamasının temelini oluşturur. Lex veya Flex ile birleştirildiğinde, Yacc, yazılım geliştiricilere programlama dili işleme için eksiksiz ve etkili bir çözüm sunar.

1.7.2. Bison

Yacc, ayrıştırıcı üretiminde önemli bir rol oynamış olsa da lisanslı yapısı ve platform bağımlılığı bazı sınırlamalar getirmektedir. Bu durum, Yacc' ın işlevselliğini koruyup açık kaynak avantajlarını sunan Bison (GNU Bison veya sıklıkla anıldığı şekliyle Buffalo Bison) gibi alternatiflerin ortaya çıkmasına zemin hazırlamıştır (Thain, 2020).

Bison, Yacc ile tam uyumluluk sağlayarak geliştiricilere sorunsuz bir geçiş imkânı sunmaktadır. Yacc için yazılmış herhangi bir gramer tanımı dosyası, Bison tarafından da herhangi bir değişiklik yapılmadan işlenebilir. Bu uyumluluk, mevcut Yacc projelerinin kolayca taşınabilmesi ve açık kaynak ekosisteminden faydalanabilmesi açısından büyük önem taşır.

Bison'un açık kaynak kodlu yapısı, geliştiricilere kaynak koda erişim, düzenleme ve dağıtma özgürlüğü sağlayarak topluluk tabanlı geliştirmeyi ve daha geniş bir platform desteğini mümkün kılar. Bu özellikler, Bison' u ayrıştırıcı üreteci arayan geliştiriciler için güçlü ve esnek bir seçenek haline getirir.

1.7.1. SableCC

SableCC, özellikle Java programlama dili için tasarlanmış, açık kaynak kodlu bir ayrıştırıcı üreteci aracıdır (Gagnon & Hendren, 1998). Yacc ve Bison gibi araçlara güçlü bir alternatif sunan SableCC, LALR ayrıştırma algoritmasını kullanarak Java dilinde ayrıştırıcılar oluşturur.

Açık kaynak kodlu yapısı, SableCC' yi çekici kılan önemli bir faktördür; çünkü geliştiricilere aracı kullanma, değiştirme ve dağıtma konusunda esneklik sağlamaktadır. Java tabanlı olması, Java projelerine sorunsuz bir şekilde entegre edilebilmesi ve Java geliştiricilerinin aşina oldukları bir ortamda çalışabilmesi açısından avantaj teşkil eder.

Yacc ve Bison gibi köklü araçların yanında SableCC, Java odaklı projelerde ayrıştırıcı geliştirme sürecini basitleştiren ve aynı zamanda açık kaynak dünyasının avantajlarını sunan değerli bir araç olarak öne çıkmaktadır.

1.7.1. ANTLR

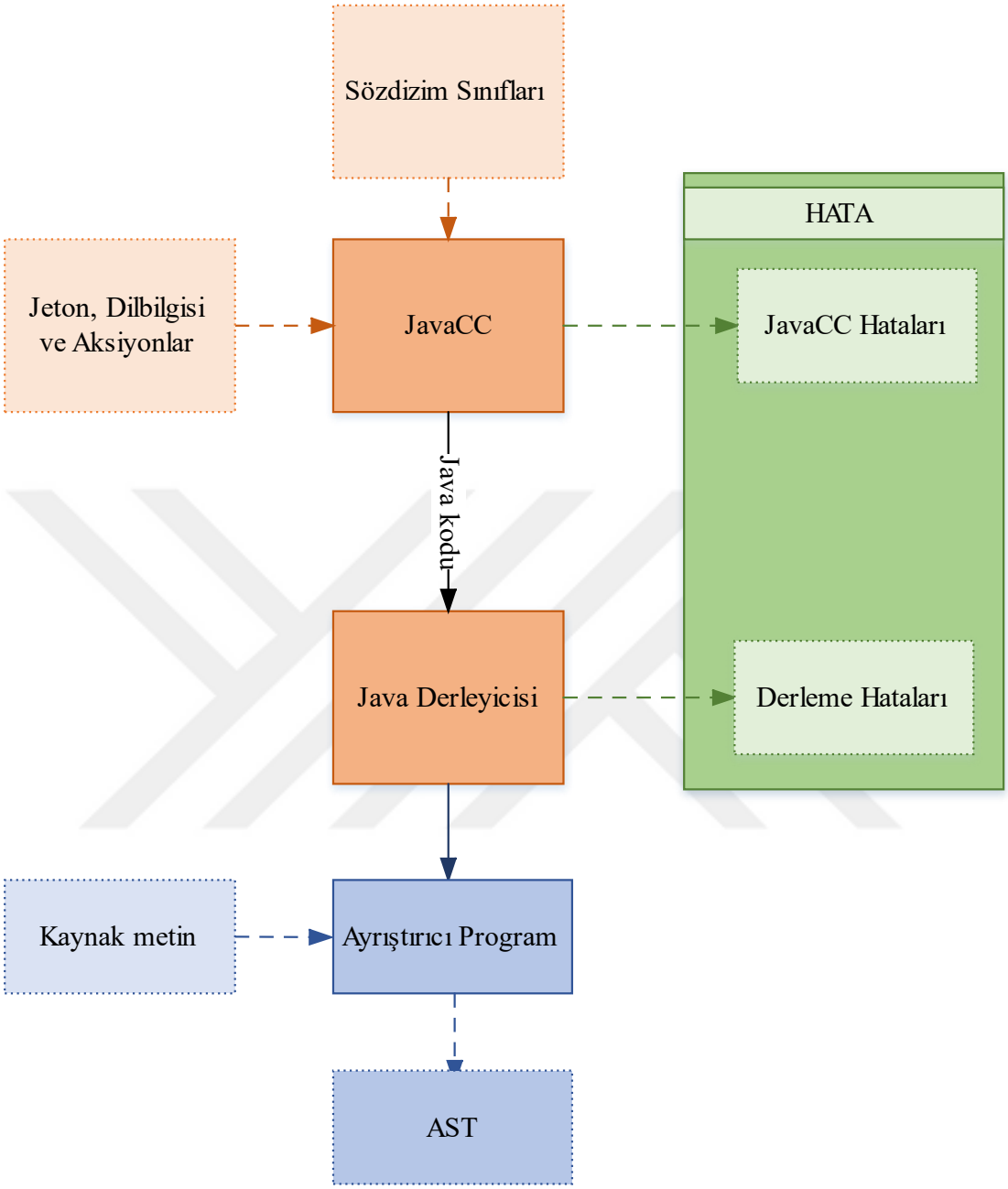
ANTLR (ANother Tool for Language Recognition), programlama dili tanıma ve ayrıştırıcı oluşturma alanında güçlü ve esnek bir araç olarak öne çıkmaktadır (Parr & Quong, 1995). Yacc/Bison gibi, ANTLR da geliştiricilerin bir dilbilgisi tanımından yola çıkarak hedeflenen dile özgü ayrıştırıcılar oluşturmasını sağlamaktadır. Ancak, Yacc/Bison' un LALR ayrıştırma algoritmasını kullanmasının aksine, ANTLR LL(*) ayrıştırma algoritmasını temel alır.

Çoklu dil desteği, ANTLR' ı diğer ayrıştırıcı üreteçlerinden ayıran önemli bir avantajdır. Java, C#, Python, JavaScript gibi birçok popüler programlama dili için ayrıştırıcı üretimi sağlayabilir. Bu özellik, ANTLR' ı farklı platformlarda ve projelerde kullanılabilen çok yönlü bir araç haline getirir.

Yacc/Bison ile karşılaştırıldığında, ANTLR daha dik bir öğrenme eğrisine sahip olabilir. ANTLR' nin sunduğu genişletilmiş özellikler ve esneklik, beraberinde daha karmaşık bir yapı ve kavramlar getirebilir. Bu nedenle, özellikle ayrıştırıcı geliştirme konusunda deneyimli olmayan kullanıcılar için ANTLR' ı öğrenmek ve kullanmaya başlamak biraz daha fazla zaman ve çaba gerektirebilir.

1.7.1. JavaCC

JavaCC (Java Compiler Compiler), Java programlama dili için özel olarak tasarlanmış, açık kaynaklı ve güçlü bir ayrıştırıcı üreteci aracıdır (Dos Reis, 2012). Geliştiriciler, JavaCC' yi kullanarak EBNF formatında yazılmış bir dil bilgisi kullanarak karmaşık ayrıştırıcılar ve sözcüksel analizciler oluşturabilirler. JavaCC, bu dil bilgisi tanımını işleyerek girdi verilerini ayrıştırabilen optimize edilmiş Java kodu üretir. JavaCC ile ayrıştırıcı üretimi Şekil 4' teki gibidir.



Şekil 4. JavaCC

JavaCC' nin temel yapıtaşları aşağıdaki gibi maddeler halinde verilebilir.

- JavaCC, ayrıştırıcının yapısını ve davranışını tanımlayan sözcük listesi, dil bilgisi kuralları ve diğer yapılandırmaları içeren .jj uzantılı bir dosya kullanır.
- JavaCC, düzenli ifadeler kullanarak esnek ve etkili bir şekilde jetonları (sözcük birimleri) tanımlama olanağı sunar.

- JavaCC, içerikten bağımsız dilbilgisi kullanarak Java dilinin yapısını ve ayrıştırma kurallarını tanımlar.

JavaCC kullanılmasının geliştiricilere sağladığı birçok yarar vardır. Bunlar;

- Ayrıştırıcı oluşturma sürecini otomatikleştirerek zamandan tasarruf sağlar.
- Tutarlı ve okunabilir Java kodu üretir, bu da bakım ve iş birliğini kolaylaştırır.
- Ayrıştırıcının yapısı ve işleyişi JavaCC tarafından net bir şekilde tanımlandığı için hata ayıklama süreci daha kolay ve etkili hale gelir.
- Java'nın nesne yönelimli özelliklerini kullanarak modüler ve yeniden kullanılabilir ayrıştırıcılar oluşturma imkânı sunar.
- Doğru tanımlanmış düzenli ifadeler ve dilbilgisi kurallarıyla hatasız çalışan ayrıştırıcılar üretir.

JavaCC, ayrıştırıcı yapılandırmasını tanımlamak için ".jj" uzantılı bir dosya kullanır. Bu dosya ayrıştırıcının davranışını özelleştirmek için isteğe bağlı seçeneklerin belirtildiği *OPTIONS* listesi ile başlar. Örneğin, öngörülü ayrıştırma sırasında kaç jetona bakılacağı, hata ayıklama modunun aktifleştirilip devre dışı bırakılacağı veya oluşturulan dosyaların hangi dizine yazılacağı gibi ayarlar yapılabilir. Daha sonra, üretilecek ayrıştırıcının ana sınıfının gövdesi tanımlanır. Bu tanımlama, *PARSER_BEGIN* ve *PARSER_END* etiketleri arasına yazılır. Bu etiketlerin arasına yazılan herhangi bir Java kodu, JavaCC tarafından oluşturulan ayrıştırıcı sınıfına aynen aktarılır. Ardından, sözcüksel analiz için gerekli olan atlanacak karakterler ve jeton listesi, düzenli ifadeler kullanılarak tanımlanır. *TOKEN* etiketi altında tanımlanan jeton listesi, gerekirse düzenli ifadeler içerebilir. *SKIP* etiketi ise, genellikle boşluk ve satır sonu karakterleri gibi ayrıştırma sırasında göz ardı edilecek karakterleri belirtmek için kullanılır. Uygulamaya bağlı olarak, atlanacak karakterlere başka karakterler de eklenebilir. Son olarak dilbilgisi kuralları *Start()* fonksiyonu başlangıç olacak şekilde fonksiyonlar halinde tanımlanır. Tablo 8'de JavaCC yapılandırma dosyası için bir örnek verilmiştir.

Tablo 8. JavaCC yapılandırma dosyası

Ayarlar

OPTIONS{
LOOKAHEAD = 1;
...
}

Ayrıştırıcı Program Sınıfı

PARSER_BEGIN(parser_name)
. . .
class parser_name {
. . .
}
. . .
PARSER_END(parser_name)

Jeton Tanımlamaları

SKIP :{ " " | "\t" | "\n" }
 	
TOKEN :{
| < #DIGIT: ["0"-"9"] >
| < NUM: (<DIGIT>)+ >
| < PLUS : "+" >
| < MINUS: "-" >
...
}

Dil Bilgisi Metotları
void Start() : {} { (| | |)* }

1.8. Programlama Dili Değerlendirme Kriterleri

Programlama dillerinin kalitesini ve etkililiğini değerlendirmek için kullanılan standartlar ve ilkeler dizisine dil değerlendirme kriterleri adı verilmektedir. Bu kriterler,

programlama dillerinin çeşitli uygulamalara ve bağlamlara uygunluğunu belirlemek amacıyla özelliklerini, tasarımını, sözdizimini ve anlambilimini değerlendirmek için kullanılmaktadır. Dil değerlendirme kriterleri, programlama dillerinin çeşitli yapılarının ve yeteneklerinin altında yatan kavramların, bakım da dahil olmak üzere yazılım geliştirme süreci üzerindeki etkilerine de odaklanarak dikkatlice incelenmesi için çok faydalı kriterlerdir (Sebesta, 2016).

1.8.1. Okunabilirlik

Okunabilirlik, belirli bir dilde yazılmış kodu okumanın ve anlamanın ne kadar kolay olduğunu ifade eden bir kriterdir. Okunabilirliği iyi olan bir dil, anlaşılması kolay, anlamlı adlandırma kuralları kullanan ve iyi yapılandırılmış bir koda sahip bir sözdizimine sahiptir. Öte yandan, okunabilirliği zayıf olan bir dil, kodda karışıklığa ve hatalara yol açmaktadır ve bu da bakımını ve genişletilmesini daha zor hale gelebilir.

1.8.2. Yazılabilirlik

Bu kriter, bir programlama dilinin kod yazmak için ne kadar kolay kullanılabileceğini ölçmektedir. Dilin sözdizimiyle ve geliştiricilerin verimli kod üretmek için bunu ne kadar hızlı kavrayabileceğiyle yakından ilgilidir. İyi yazılabilirliğe sahip bir dil, kısa bir sözdizimine, hatırlanması kolay anahtar kelimelere ve yapılara ve karmaşık fikirleri ifade etmeyi kolaylaştıran güçlü soyutlamalara sahiptir.

1.8.3. Güvenilirlik

Güvenilirlik, bir programın giriş verilerine dayanarak belirtilen hesaplamaları ne kadar doğru gerçekleştirdiğini ölçmektedir. Bir dilin, hataların ve öngörülemeyen durumların varlığında bile tutarlı ve öngörülebilir sonuçlar üretme yeteneğini ifade etmektedir.

Güvenilirliği iyi olan bir dilin beklenmeyen sonuçlar veya çökmeler üretme olasılığı daha düşüktür, bu da onu sağlam ve güvenilir yazılım oluşturmak için daha uygun hale getirir.

1.8.4. Sürdürülebilirlik

Bu kriter, bir yazılım programının zaman içinde kolayca muhafaza edilebilme ve güncellenebilme yeteneğini ifade etmektedir. Yazılım geliştirme sürecinde ortaya çıkabilecek hataları belirleme ve düzeltmenin yanı sıra yeni özellikler ekleme yeteneğini de içerir.

1.8.5. Ortogonallik

Bu, farklı dil özelliklerinin birbirleriyle nasıl etkileşime girdiğinin tutarlılığını ve öngörülebilirliğini ifade eder. Programı oluşturmak için nispeten az sayıda ilkel yapının çeşitli yollarla birleştirilebileceği anlamına gelmektedir. Ortogonal dil, programdaki görünümünün içeriğinden bağımsızdır.

1.8.6. Tekdüzelik

Dilin benzer anlamlara sahip yapıları kullanımında ne kadar tutarlı olduğunu değerlendirmektedir.

1.8.7. Genişletilebilirlik

Programcıların yeni işlevler sunmasına ve onu belirli ihtiyaçlara uyarlamasına olanak tanıyarak dilin yeteneklerini temel özelliklerinin ötesine genişletme yeteneği.

1.8.8. Verimlilik

Bu ölçüm, bir programın ne kadar hızlı çalıştırılabileceğini ve ne kadar bellek gerektirdiğini ölçer.

1.8.9. Taşınabilirlik

Bu, bir platform veya ortam için yazılan kodun önemli değişiklikler olmadan başka bir platformda veya ortamda yürütülebilmesinin kolaylığını ifade eder.

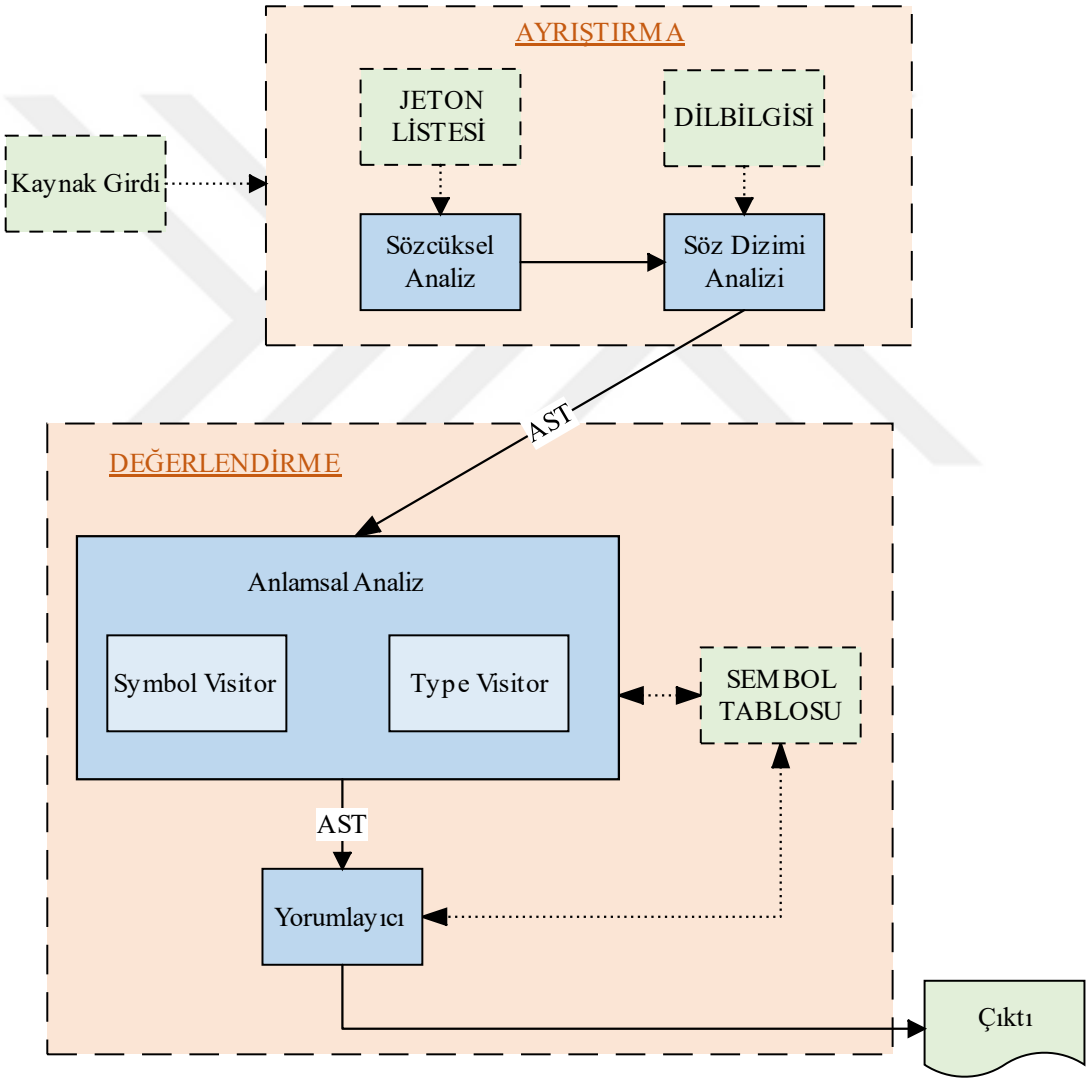
2. YAPILAN ÇALIŞMALAR

Sayısal çözümleme, diferansiyel denklemlerin çözümünden matris cebrine kadar geniş bir yelpazede matematiksel modelleme ve bilimsel hesaplama problemini ele almamızı sağlayan, yaklaşık çözümler üreten vazgeçilmez bir araçtır. Özellikle mühendislik eğitiminde önemli bir yer tutan sayısal çözümleme yöntemleri, günümüzde genellikle genel amaçlı programlama dilleri veya matematiksel hesaplama ortamları kullanılarak öğretilmekte ve uygulanmaktadır. Ancak, mevcut bu alternatifler, sayısal çözümlemenin tam anlamıyla kavranmasının ve etkili bir şekilde kullanılmasının önünde engel teşkil etmektedir.

Matlab gibi özel yazılımlar, eğitim ve bilimsel araştırmalarda güçlü araçlar olarak kabul görmelerine rağmen, bazı önemli kısıtlamalar sergilemektedirler. Ticari amaçlarla geliştirilmeleri nedeniyle genellikle yüksek maliyetlidirler ve bu durum, özellikle bireysel kullanıcılar ve küçük ölçekli araştırma grupları için önemli bir engel teşkil etmektedir. Dahası, özellikle hazır fonksiyonlara olan ağır bağımlılık ve matematiksel söz diziminden uzaklaşan programlama yapıları, bu ticari yazılımlarda geliştirilen kodların genel amaçlı dillere dönüştürülmesini oldukça zorlaştırmakta ve bazen pratik olarak imkânsız hale getirmektedir. Bu durumun ötesinde, ticari yazılımların sunduğu kullanıcı dostu arayüz ve hazır fonksiyonlar, öğrenme ve araştırma süreçlerinde bazı dezavantajlar da beraberinde getirebilmektedir. Matematiksel söz diziminden farklılaşan programlama dilleri ve sınırlı esneklik, kullanıcıların algoritma üzerinde tam kontrol sahibi olmasını engelleyerek, öğrenme sürecinde pasifliğe ve algoritmik düşünme becerilerinin körelmesine yol açabilmektedir. Özellikle eğitim ortamlarında, hazır fonksiyonlara aşırı bağımlılık, öğrencilerin problem çözme yeteneklerini ve algoritma geliştirme konusundaki derinlemesine öğrenmelerini olumsuz etkileyebilmektedir.

Günümüzde yaygın olarak kullanılan programlama dilleri, basit matris cebri gibi matematiksel işlemleri gerçekleştirme kapasitesine sahip olsalar da, bu işlemlerin kodlanması genellikle karmaşık ve öğrenmesi zor bir süreçtir. Genel amaçlı dillerin yapısı gereği, bu tür matematiksel işlemler için yazılan kodlar hacimli ve anlaşılması güç bir hal alabilir. Her ne kadar bu diller, önceden tanımlanmış kütüphane fonksiyonları aracılığıyla matematiksel işlemleri kolaylaştırsalar da bu fonksiyonların kullanımı bile belirli bir programlama bilgisi gerektirmekte ve matematiksel ifadenin doğrudan bir yansıması

olmaktan uzaktır. Dolayısıyla, özellikle programlamaya yeni başlayan öğrenciler, programlamanın teknik detaylarına odaklanmak zorunda kalabilir ve bu durum, matematiksel kavramların özünü anlamalarını engelleyebilir. Bu bağlamda, matematiksel ifadelere daha yakın bir yapıya sahip ve öğrenme sürecini kolaylaştıran özel amaçlı dillerin önemi ön plana çıkmaktadır. Bu diller, öğrencilerin programlamanın karmaşıklığına kapılmadan matematiksel kavramları daha etkili bir şekilde öğrenmelerine ve uygulamalarına imkân tanıyabilir.



Şekil 5. MxPL genel çalışma şeması

Bu tez çalışmasında, sayısal çözümleme tekniklerine yönelik, programlamaya dayalı matematiği ve doğrusal cebri orta düzeyde kullanabilen yeni bir programlama dili geliştirilmesi amaçlanmaktadır. Geliştirme süreci, Şekil 5' de görüldüğü üzere, sözcük analizi, söz dizimi analizi, ayrıştırma ağacı oluşturma, anlamsal analiz ve yorumlama gibi temel aşamalardan oluşmaktadır. Bu süreç, genel hatlarıyla ayrıştırma ve değerlendirme olmak üzere iki ana başlıkta incelenebilir. Ayrıştırma sürecinde JavaCC aracı kullanılarak bir Ayrıştırma Ağacı (AST) oluşturulur. Değerlendirme aşamasında ise, ziyaretçi tasarım deseni yaklaşımıyla AST, farklı amaçlar için analiz edilir ve yorumlanır.

Tez çalışmasının temel motivasyonu; sayısal çözümleme yöntemlerinin programlanmasında kolaylık sağlayan, açık, matematiksel sözdizimine benzer, kullanıcı dostu, platform bağımsız ve kolayca öğrenilebilen basit bir dil sistemi geliştirmektir. Geliştirilen programlama dilinin matematiksel ifadelere yakın ve kullanıcı dostu söz dizimi, çeşitli karmaşık matris işlemlerini içeren programlama kodu örnekleri ile açıkça ortaya konmaktadır. Çalışmanın son bölümünde ise, geliştirilen dil; C, Java ve Python gibi genel amaçlı programlama dilleri ile Breeze, MATLAB, NumPy, R ve ALGLIB gibi özel sayısal hesaplama ortamlarıyla çeşitli özellikler ve ölçütler bakımından karşılaştırılmıştır. Bu karşılaştırmalı analiz, dilin performansını ve sunduğu avantajları değerlendirmede önemli bir rol oynamaktadır.

2.1. Geliştirilen Dilin Söz Dizimi

Programlama dillerinin ifade yapısı, sözdizimi veya sentaks olarak da bilinen dilin kurallarını ve bu kurallara göre nasıl kod yazılabileceğini belirleyen bir sistem tarafından yönetilmektedir. Bu sistem, programda hangi tür ifadelerin kullanılabilir olduğunu, bu ifadelerin nasıl sıralanması gerektiğini ve programın nasıl yorumlanacağını tanımlamaktadır. Tez çalışmasında geliştirilen programlama dili matematiksel işlemleri ve sayısal çözümleme yöntemlerinde gerekli yapıları genel amaçlı dillere göre daha kolay temsil edecek şekilde tasarlanmıştır. Geliştirilen dil matematiksel gerekli tanımlamaların ve manipülasyonların için kullanışlı notasyonların basit bir sözdizimini sağlar.

Sayısal çözümleme yöntemlerinde problemlerin çözümünde en sık kullanılan ögelerden biri matrislerdir. Bir matris, m sayıda satırı ve n sayıda sütunu olan iki boyutlu bir sayı dizisidir ve buna m × n matrisi olarak adlandırılır. Tez çalışmasındaki dilde matrisler sıradan matematiksel gösterimlere uygun olarak aşağıdaki şekilde tanımlanmıştır:

A = [[a_{11}, a_{12}, . . . , a_{1n}], [a_{21}, a_{22}, . . . , a_{2n}], [a_{m1}, a_{m2}, . . . , a_{mn}]]

Bir matris aynı zamanda tek boyutlu bir dizi veya iki boyutlu bir dizi olarak da düşünülebilir. Boyutları 1 × 3 ve 2 × 3 olan bu tür matrislerin iki örneği aşağıda verilmiştir:

A = [[1, 2, 3]]

B = [[1, 2, 3], [4, 5, 6]]

Ayrıca 1x1 boyutunda bir matris olarak kabul edilen tek değerli matrislerin tanımlanmasına da izin verilmektedir. Benzer şekilde bir dizi, tek satırlı bir matrise karşılık gelir. Bazı örnekler aşağıdaki gibidir:

A = [1,2] → [[1,2]]

B = [1, 2, 3] → [[1, 2, 3]]

Geliştirilen dilde değişkenlere rasyonel tipte değerler atamak da mümkündür. Bir matris elemanı ya da herhangi bir değişken, iki tam sayının oranı olarak temsil edilen rasyonel bir sayı olabilmektedir. Rasyonel sayılar, aşağıda belirtildiği gibi bölme sembolü "/" kullanılarak oluşturulabilir:

A = [1/2] → [[1/2]]

B = [1/2, 1/3] → [[1/2, 1/3]]

C = 1/2

Programlama dillerinin temel yapılarından biri fonksiyonlardır. Çalışmada geliştirilen dil fonksiyonlar için basit ve anlaşılır bir sözdizimi içermektedir. Bir fonksiyon tanımının genel sözdizimi şu şekilde sağlanabilir:

$$fonksiyonadı(p\_1, p\_2 \ldots . p\_n\ )\{blok\} = gövde$$

Bu sözdiziminde fonksiyonlar "=" eşitlik işaretinin her iki yanında bulunan denklemler olarak tanımlanır. "fonksiyonadı" sözcüğü p_1,p_2….p_n resmi parametrelerine sahip bir fonksiyonun adına karşılık gelir. Küme parantezleri arasında yazılan "blok" kelimesi fonksiyonun ana gövdesini temsil eder, gerekmediği takdirde çıkarılabilir. "gövde" ise fonksiyonun dönüş değerini değerlendirir.

Dilin kaynak kodunun değerlendirilmesi, diğer birçok programlama dilinde olduğu gibi her zaman main adı verilen bir fonksiyonun çağrılmasıyla başlatılır.

2.2. Ayrıştırma

Ayrıştırma, kaynak verilerin dilbilgisel analizi sürecidir. Ayrıştırma işleminde ilk adım dile ait sözdizimi için resmi bir dilbilgisi tasarlanmasıdır. Tez çalışmasında önce dile ait bir dilbilgisi tasarlanmıştır. Daha sonra tasarlanan dilbilgisi JavaCC aracı formatına çevrilerek ayrıştırıcı geliştirilmiştir. Bu bölüm dilinin sözcüksel ve sözdizimsel açıklaması aracılığıyla sözcük oluşturucunun ve ayrıştırıcının uygulanmasının ayrıntılarına girmektedir.

2.2.1. Dilbilgisi

Çalışmada geliştirilen resmi dilbilgisi Tablo 9' da verilmiştir. Geliştirilen dile ait sunulan biçimsel dilbilgisi operatörle ilgili operatörlerin olağan önceliğine ve birleşme yönlerine ilişkin bilgileri içeren anlamsal yapıyı da belirtecek şekilde tasarlanmıştır.

Tablo 9. MxPL dilbilgisi

Program::=Function(Program)?
Function::=Header(Block)?Body
Header ::=Id'(' (Parlist)?')'
Parlist ::=Param(','Parlist)?
Param::=Id|Val
Block::='{' (StmList)?'}'
StmList ::=Statement(StmList)?
Statement ::=IfStm|WhileStm| (FunctionCall ';')| (Assignment ';')
Assignment ::=Var'='Expression
Body::='=' (Expression|EquationList)
EquationList ::='{'Expression','Expression'}' (EquationList)?
IfStm::='if' '('Expression')'Block
WhileStm::='while' '('Expression')'Block
FunctionCall ::=Id'('ExprList?')'
ExprList ::=Expression(','Expression)*
Expression::=('+' | '−')?Term(('+' | '−')Term)*
Term::=Power(('*' | '/' | '%')Power)*
Power ::=LogicOr('^'LogicOr)*

Tablo 9’ un devamı

```
LogicOr ::=LogicAnd(’||’LogicOr)?
LogicAnd::=Comparison(’&&’LogicAnd)*
Comparison::=Element((’<’ | ’<=’ | ’>’ | ’>=’ | ’==’ | ’!=’)Element)?
Element ::=Var |FunctionCall |Val | ’(’Expression’)’
Val ::=Num|Matrix
Matrix::=’[’ (NumList | (’[’NumList ’]’(’,’ ’[’NumList ’]’)*))’]’
NumList ::=Num(’,’NumList)*
Id::=#’[A−Za−z][A−Za−z0−9_]*’
Num::=Digits(’.’Digits)?
Digits ::=#’[0−9]+’
Var ::=Id(’[’Expression’,’Expression’]’)?
```

2.2.2. Sözcüksel Analiz

Sözcüksel analiz, kaynak veriyi sözcük birimleri adı verilen daha küçük, daha yönetilebilir birimlere bölmekten sorumludur. Bu sözcük birimleri, sözdizimi analizi adı verilen bir sonraki aşama için yapı taşları görevi görmektedir. Sözcük üreteci bir karakter dizisini bir jeton dizisine dönüştürerek boşluk karakterlerini ve yorumları kaldırdığı için dile ait bir kelime listesi üretilmesi gerekmektedir. Geliştirilen dil için oluşturulmuş kelime listesi Tablo 10’ da verilmiştir. Tablo 10’ da yer alan TOKEN bloğunda verilen her bir tanımlama için kelimeler listelenmişken, SKIP bloğundaki kelimeler için bir üretim yapılmaz. Çünkü kelime listesi içinde yer alan SKIP bloğu, boşluk karakterleri ile yorum satırları gibi jeton üretimi yapılmayacak ifadelerin tanımlarını içermektedir.

Tablo 10. MxPL diline ait jeton listesi

```
TOKEN: {
<PLUS: "+"> | <MINUS: "-"> | <TIMES: "*"> | <DIVIDE: "/"> | <POWER: "^">
| <MOD: "%">
| <AND: "&&"> | <OR: "||"> | <NOT: "!">
| <AEQ: "="> | <EQ: "=="> | <NE: "/="> | <LE: "<"> | <LT: "<="> | <GT: ">=">
| <GE: ">">
```

Tablo 10’ un devamı

```
| <COMMA: ","> | <SEMI: ";"> | <COLON: ":"> | <GUARD: "|">
| <LCURLY: "{"> | <RCURLY: "}"> | <LPAREN: "("> | <RPAREN: ")">
| <LBRCT: "["> | <RBRCT: "]">
| <ABS: "abs"> | <SQRT: "sqrt"> | <ROUND: "round"> | <DRV: "drv">
| <PRINT: "print"> | <OTHER: "otherwise">
| <DROPROW: "droprow"> | <DROPCOL: "dropcol">
| <CRT: "matrix"> | <SIZE: "size">
| <IF: "if"> | <ELSE: "else"> | <WHILE: "while">
| <QUIT: "quit">
| <ID: (["a"-"z","A"-"Z"])(["a"-"z","A"-"Z","0"-"9"])*>
| <NUM: (["0"-"9"])+>
| <DNUM: (["0"-"9"])+"."(["0"-"9"])+>
| <STR: ("\"" (~["\""] | "\\" "\"")* "\"")>
}
SKIP: { " " | "\t" | "\r" | "\n" }
```

Tablo 10’ daki her bir kelime dile ait bir yapının temsilidir. Burada ID kelimesi bir değişkenin, argümanın veya fonksiyonun adını temsil eder. Bir harf ile başlayıp rakam ya da harf ile devam eden kelimeleri içerir. NUM ve DNUM kelimeleri ise sırası ile tam sayı ve ondalıklı sayıları tanımlar.

Tablodaki kelime listesi ile sözcüksel analiz süreci f(x)= x+1 ifadesi için aşağıdaki Tablo 11’ deki, A=[1,2,3] ifadesi için de Tablo 12’ deki jeton dizilerini üretmektedir.

Tablo 11. $f(x) = x + 1$ İfadesi için tarayıcı örneği

Girdi:	$f(x) = x + 1$
Çıktı:	<ID> < LPAREN> <ID> < RPAREN > < AEQ> <ID> <PLUS> <NUM>

Tablo 12. $A = [1,2,3]$ İfadesi için tarayıcı örneği

Girdi:	$A = [1, 2, 3]$
Çıktı:	<ID> <AEQ> <LBRACKETS> <NUM> <COMMA> <NUM> <COMMA> <NUM <RBRACKETS>

Sözcük üreteci kaynak veride taradığı her kelime için yukarıdan aşağıya tüm jeton tanımlarını inceler ve ilk eşleşmeye karşılık gelen jeton üretilir. Bu sebeple ID jetonu bir harf ile başlayan tüm kelimelerle eşleşeceği için dile ait anahtar kelimeler ID jetonundan daha üstte tanımlanmalıdır.

2.2.3. Ayrıştırıcı

Çalışmada girdi verilerini sözcük üreteci ile jeton dizisine çevirdikten sonra gelen işlem ayrıştırmadır. Ayrıştırma işlemi JavaCC ile oluşturulan bir LL(k) ayrıştırıcı ile sağlanır. Geliştirilen dilde k değeri 1 (k=1) olarak seçilmiştir. Böylelikle LL(1) ayrıştırıcısı, kaynak verilerden tüketilen bir jetona bağlı olarak sadece bir dilbilgisi kuralı seçebilmektedir. Yani, JavaCC dilbilgisi tanımlamasında her dilbilgisi kuralına karşı bir metot tanımı oluşturulur. Bu şekilde, her dilbilgisi kuralını temsil edecek şekilde ayrıştırıcıya bazı metot tanımlamaları eklenebilmiştir. Bu tanımlamalar, kuralların dilbilgisinde birbirini çağırdığı sırayla tutulur. Tablo 13' te LL(1) ayrıştırıcı için JavaCC formatında dilbilgisi kuralları verilmiştir. İlgili kurallar tanımlanırken Tablo 10' da verilen jeton isimleri kullanılmıştır.

Tablo 13. JavaCC dilbilgisi metotları

```
Program Start() :{} {Program()<EOF>}
Program Program() :{} {Function()(Program())?}
Function Function() :{} {Header()(Block())?Body()}
Header Header() :{} {<ID><LPAREN>(Parlist())?<RPAREN>}
Stm Block() :{} {<LCURLY>(Stmlist())?<RCURLY>}
QList Body() :{} {<AEQ>Expr() |Eqlist()}
EList Parlist() :{} {Param()(<COMMA>Parlist())?}
Exp Param() :{} {<ID>(<PLUS><NUM>)?|<NUM>}
..
```

Tablo 13' teki metotların ilk olarak çağrılanı start() metodudur. Bu metot geliştirilen dilde ayrıştırıcının başlama metodu olarak ifade edilmiştir.

Kaynak dosyanın tümüne denk gelen Program() metodu ve <EOF> ile dosyadan okunan ya da giriş olarak girilen kaynak dosyanın sonlandığını belirten sonlandırma jetonu içermektedir. Bir Program() metodu, bir ya da daha fazla fonksiyondan oluşmaktadır.

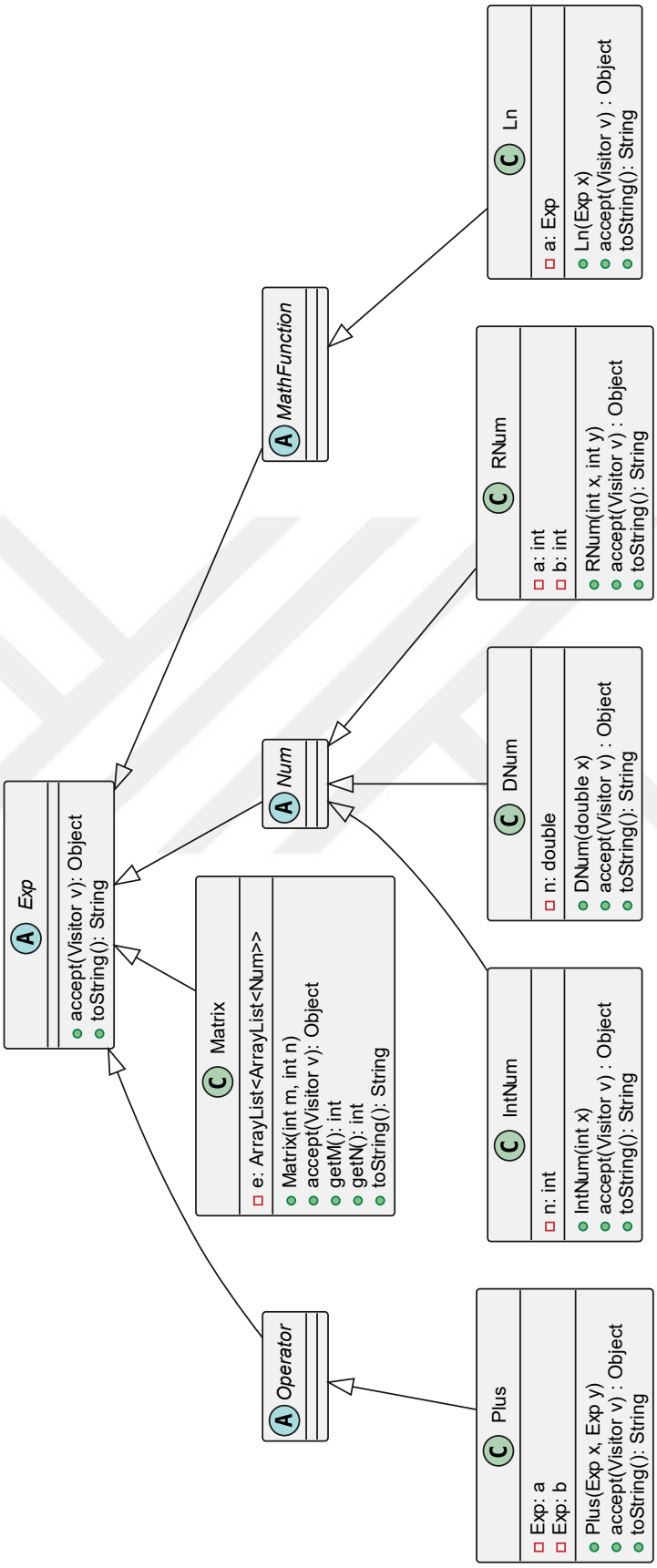
Bir fonksiyon (Function()) ise Header, Block ve Body isimli üç elementten oluşur. Header, fonksiyonun fonksiyon adı ve parametrelerinden oluşan başlık bilgisidir. Fonksiyon adı <ID> jetonundan oluşan karakter dizisidir. Daha sonra <LPAREN> jetonu ile fonksiyon adından sonra gelen sol parantez "(" gelmektedir. Bir fonksiyon parantez içerisinde parametre ya da parametreler içerebileceğinden Parlist() ile parametre bilgisi verilmektedir ve Parlist() ardından gelen "?" ile de fonksiyonun hiç parametre olmadığı durumlar sağlanmış olunmaktadır. Header()' da ki en son gelen jeton olan <RPAREN> ile de fonksiyonun sağ parantezi ") " yani kapama parantezi eklenerek başlık bilgisi sonlandırılır. Fonksiyonun Block metodu dilin buyurgan ifadelerinin kullanılabildiği blok kısmını tanımlar. Bu kısımda tanımlamalar, koşullu ifadeler, döngüler ifadelere ait gramer metotlarını içerebileceği tanımlanmıştır. Body metodunda ise fonksiyon gövdesinin sadece bir ifade veya birden fazla koşullu ifade içerebileceği tanımlanmıştır. Parlist metodunda fonksiyonun parametrelerinin virgül ile ayrılacak listelendiği tanımlanırken, Param metodu ile parametre kuralları tanımlanır.

2.2.4. Nesne Sınıfları

Java gibi nesne yönelimli programlama dillerinde kaynak kodunun ağaç tabanlı bir gösterimini nesne yönelimli programlama yapılarını kullanarak oluşturmak için nesne sınıfları kullanılmaktadır. Yani kısaca Java'da dilin terminalleri ve terminal olmayanları nesnelerle temsil edilebilmektedir. Bu sınıfların tasarımı, programlama dilinin dilbilgisi kuralları tarafından yönlendirilir Genellikle bir operatör veya anahtar kelime içeren her dilbilgisi kuralı için bir nesne sınıfı ya da bir diğer adıyla sözdizimi sınıfı tanımlanır. Alternatifleri olan dilbilgisi kuralları, ortak bir üst sınıftan miras alan ve böylece birbirlerinin yerine kullanılabilen nesne sınıflarıyla temsil edilir. Bu yaklaşım ayrıştırma ağacının oluşturulmasını ve işlenmesini kolaylaştırır.

Belirli durumlarda, özellikle bir dilin kapsamlı bir temsilini oluştururken, birden fazla dilbilgisi kuralı tek bir sözdizimi sınıfında birleştirilebilir. Örneğin, bir programlama dilinde bir ifadeyi tanımlamak için birden fazla dilbilgisi kuralı kullanılıyorsa, bu kuralların tümünü kapsayan tek bir sözdizimi sınıfı tanımlamak yeterli olabilir. Bu yaklaşım, ifade veya diğer

kurallar tarafından başvurulan yapılar gibi tekrar eden kalıplar için özellikle yararlıdır. Geliştirilen dile ait bazı söz dizimi sınıflarını gösteren sınıf şeması Şekil 6’ da verilmiştir.



Şekil 6. Sözdizim sınıf şeması

Şekil 6’ dan da görüldüğü gibi, matematiksel ifadeleri temsil etmek için kullanılan yapı, temelde Exp adlı bir sınıfa dayanmaktadır. Bu sınıf, tüm matematiksel ifadelerin atası olarak düşünülebilir. Diğer bir deyişle, her bir matematiksel ifade, en temel özelliklerinden miras aldığı Exp sınıfının bir türevidir.

Örneğin, tamsayıları temsil etmek için kullanılan IntNum sınıfı, doğrudan Exp sınıfından türetilmez. Bunun yerine, öncelikle Exp sınıfından türetilen Num adlı bir ara sınıftan türetilmektedir. Bu, IntNum sınıfının, sayılarla ilgili daha genel özellikleri barındıran Num sınıfının özelliklerini miras aldığı anlamına gelmektedir. Sonuç olarak, IntNum sınıfı hem Num sınıfının hem de en temel sınıf olan Exp sınıfının özelliklerine sahip olmaktadır. DNum ise ondalıklı sayılar için kullanılan bir sınıftır. O da aynı şekilde doğrudan Exp sınıfından türetilmek yerine, Num ara sınıfından türetilmektedir. Geliştirilen dildeki bahsedilen Exp sınıfı Tablo 14 ’ deki gibidir.

Tablo 14. Sözdizim sınıf tanımlamaları

```
public class Exp {
 public Exp copy() { return new Exp(); }
 public Object accept(Visitor v) { return null; }
 public String toString() { return "new Exp()"; }
}
public abstract class Num extends Exp {
}
public class IntNum extends Num {
   public int n;
   public IntNum(int x){
      n=x;
   }
   public Object accept(Visitor v) {
      return v.visit(this);
   }
         ...
}
```

Tablo 14’ ün devamı

```
public class RNum extends Num {
   public int a;
   public int b;
   public RNum(int x, int y) {
      a = x;
      b = y;
      if(b<0){
         a=-1*a;
         b=-1*b;
      }
   }
   public Object accept(Visitor v) {
      return v.visit(this);
   }
...
}
public abstract class Operator extends Exp {
}
public class Minus extends Operator {
  public Minus(Exp x, Exp y) {
    a = x;
    b = y;
 }
..
}
```

2.2.5. Sözdizim Ağacı

Çalışmanın bu aşamasında, daha önce tanımlanan sözdizimi sınıfları kullanılarak kaynak kodun sözdizimi ağacı (ayrıştırma ağacı veya AST) oluşturulmaktadır. Bu süreçte JavaCC aracı sözdizimi ağaçlarının üretiminde önemli bir rol oynamaktadır. Diğer birçok ayrıştırıcı üretecinde olduğu gibi JavaCC de ayrıştırıcı metotlarına semantik eylemlerin

eklenmesine olanak vermektedir. Bu anlamsal eylemler, Java kod parçacıkları olarak, sözdizimi ağacındaki belirli bir düğümün oluşturulmasından sorumludur. Tablo 15'te, sözdizimi ağacını oluşturmak için eklenen kod bloklarıyla birlikte bazı ayrıştırıcı metotlarını verilmiştir.

Tablo 15. Sözdizim ağacını oluşturan aksiyon tanımları

```
Program Start() :{ Program prog; }
 { prog=Program() <EOF> { return prog; } }
Program Program() : { Function def; Program prog = null; }
 { def=Function() (prog=Program() )? { return new Program(def, prog); } }
Function Function() : { Header fn; Stm s=null; QList eq; Exp e; }
 { fn=Header() ( s=Block() )? eq=Body() { return new Function(fn, s, eq); } }
Header Header() : { Token t; EList eq = null; boolean b = false; }
 {
  t=<ID> <LPAREN> (eq=Parlist() { b = true; })? <RPAREN>
  { return new Header(t.image, b ? eq : null); }
 }
Stm Block() : { Stm s=null; }
 { <LCURLY> ( s=Stmlist() )? <RCURLY> { return s; } }
QList Body() : { Exp e; QList eq; }
 { <AEQ> e=Expr() { eq = new QList(new BNum(true), e, null); } | eq=Eqlist() { return
eq; } }
EList Parlist() : { EList eq = null; Exp e; }
 { e=Param() (<COMMA> eq=Parlist() )? { return new EList(e, eq); } }
Exp Param() : { 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> { return new IntNum(Integer.parseInt(t.image)); }
 }
```

Tablo 15’ te olduğu gibi, sözdizimi ağacının kök düğümü Program türünde bir nesneye içermektedir. Bu durum, analizin başlangıç noktası olan start() yönteminin çağrılmasıyla doğrudan ilişkilidir. Fakat çağrılan diğer yöntemler farklı türde nesneler üretir. Ancak, ayrıştırma sürecinde çağrılan diğer yöntemler farklı türlerde nesneler üretir, bu da ağacın geri kalanının Program türünden farklı nesnelerden oluşabileceğini gösterir.

2.3. Değerlendirici

Değerlendirme süreci, bir programlama dilindeki bir programın yürütülmesi için gerekli adımları gerçekleştirmeden önce kaynak kodun anlaşılmasını ve doğrulanmasını içeren önemli bir aşamadır. Bu süreç genellikle iki temel göreve ayrılabilir: anlamsal analiz ve kod yorumlama.

Anlamsal analiz, programın dilin kurallarına uygunluğunu denetler ve değişken bildirimleri, tür uyumluluğu ve kapsam kuralları gibi konuları ele alır. Bu analiz sırasında, anlamsal hatalar tespit edilir ve düzeltilmesi için geliştiriciye bildirilir.

Kod yorumlama ise programın gerçek anlamını çıkarır ve talimatlarını adım adım çalıştırır. Bu adımda, programın davranışı belirlenir ve sonuçlar üretilir.

Çalışmada, bu bileşenlerin uygulanması için Ziyaretçi tasarım modeli tercih edilmiştir. Bu tasarım modeli, bir nesne yapısının (AST gibi) elemanlarını dolaşmak ve her bir eleman üzerinde belirli işlemleri gerçekleştirmek için etkili bir yol sunar.

Ziyaretçi tasarım modelinin kullanımı, anlamsal analizör ve kod yorumlayıcısının AST üzerindeki farklı düğüm türlerini ele almak için özel işlemler tanımlamasını sağlar. Bu da kodun daha modüler, okunabilir ve bakımı kolay olmasına yardımcı olur.

Bu bölümde, Accept ve Visitor adlı iki Java arayüzünün nasıl tanımlandığı ve uygulandığı ayrıntılı olarak ele alınacaktır. Bu arayüzler, Ziyaretçi tasarım modelinin uygulanmasında ve anlamsal analizör ve kod yorumlayıcısının AST ile etkileşiminde kritik bir rol oynar.

Accept arayüzü, her bir sözdizimi sınıfı tarafından uygulanır ve accept() metodunu tanımlar. Visitor arayüzü ise, her bir sözdizimi sınıfı için bir visit() metodu tanımlayan ayrı bir sınıf tarafından uygulanır. Tablo 16’da, IVisitor arayüzü ve visit() metot prototipleri gösterilmiştir. Metotların dönüş tipi, sözdizimi ağacında farklı değerlendirmelere olanak sağlamak için Object olarak belirlenmiştir.

Tablo 16. Ziyaretçi arayüzü

```
public interface IVisitor {
 public Object visit(Times e);
 public Object visit(Divide e);
 public Object visit(Mod e);
 public Object visit(Power e);
 public Object visit(Var e);
 public Object visit(Str e);
 public Object visit(Sqrt e);
 public Object visit(Matrix e);
...
```

2.3.1. Anlamsal Analiz

Ayrıştırma işlemi, kaynak kodun belirli bir programlama dilinin kurallarına uygun olarak yapılandırıldığını doğrular, ancak kodun anlamını veya mantıksal tutarlılığını denetlemez. Bu nedenle, programın çalıştırılmasından önce, kaynak kodun anlamsal olarak da doğru olduğundan emin olmak için daha derinlemesine bir analiz olan anlamsal analiz gerçekleştirilmesi gerekmektedir.

Anlamsal analiz, sözdizimi bileşenlerini dile özgü anlamlarıyla ilişkilendirerek programın anlamını çözümleme sürecidir. Başka bir deyişle, anlamsal analiz, programın gerçekten ne yapacağını anlamaya çalışır. Anlamsal analizin temel görevlerinden biri, kaynak kodda bulunan her bir tanımlayıcının veri türünü belirlemektir. Tanımlayıcılar, değişkenler, fonksiyonlar, sınıflar, parametreler vb. gibi programdaki farklı öğeleri temsil eder. Bu amaçla, kaynak koddaki tüm tanımlayıcıların bilgilerini içeren bir "sembol tablosu" oluşturulur. Sembol tablosu, bir programın anlamsal analizini ve daha sonraki derleme aşamalarını kolaylaştıran önemli bir veri yapısıdır.

Her bir tanımlayıcı, kaynak kodda ilk kez göründüğü anda (tanımlandığı veya kullanıldığı noktada) sembol tablosuna eklenir. Sembol tablosunda genellikle tanımlayıcının adı, türü, kapsamı (hangi kod bloğunda geçerli olduğu), parametreleri (fonksiyonlar için) gibi bilgiler saklanır.

Anlamsal analiz sırasında, yorumlayıcı sembol tablosunu kullanarak tür uyumluluğunu kontrol eder, değişkenlerin doğru bir şekilde kullanılıp kullanılmadığını denetler ve diğer anlamsal kuralları uygular. Örneğin, bir değişken tanımlanmadan önce kullanılıyorsa veya bir fonksiyona yanlış sayıda veya türde argüman gönderiliyorsa, anlamsal analiz bu hataları tespit eder ve geliştiriciye bildirir. Bunların hepsini gerçekleştirmek için sembol tablosu gerektiğinden geliştirilen dile ait sembol tablosundan bahsetmek gerekmektedir.

Bu çalışmada geliştirilen dil, fonksiyonlar için C programlama diline benzer şekilde düz bir kapsam yapısı benimserken, değişkenler için iç içe geçmiş blok yapısını desteklemektedir. Bu hibrit yaklaşım, global kapsam içinde yer alan ve yerel kapsamlı değişkenleri barındıran iç blokları beraberinde getirir. Bir değişkenin tüm kullanımları, tanımlandığı yerel kapsam içinde aynı veri türüne sahip olmalıdır. Değişkenin türü, atanan ilk değerin türünden çıkarılır. Bu nedenle, farklı türlerde değerler alan tanımlayıcılar farklı isimlere sahip olmalıdır. Başka bir deyişle, global değişkenler tek değerlidir ve yeniden değer atanamazlar. Bu kısıtlama, fonksiyon parametrelerinin tembel değerlendirme mekanizmasını desteklemek için gereklidir.

Tablo 17’ de gösterilen SymbolTable sınıfı, tüm tanımlayıcıları tür bilgileri ile birlikte saklamak için kullanılır. Bu sınıf, yeni bir blok kapsamına girmek için beginScope() metodunu, önceki kapsama dönmek için endScope() metodunu ve her değişken veya parametrenin adını türüne, her fonksiyonun adını ise parametre listesine ve dönüş türüne bağlamak için put() metodunu sağlar. Belirli bir tanımlayıcının sembol tablosundaki bilgileri, get() metodu ile elde edilebilir.

Tablo 17. Sembol tablosu

```
public class SymbolTable {
  Hashtable<String,FunctionHeader> func_headers = new Hashtable();
  ArrayList<SymbolList> smyTableList = new ArrayList();
  public SymbolList cur_scope = null;
  int cur_lvl=−1;
  public SymbolTable(){
    SymbolList symbolList = new SymbolList(null);
    smyTableList.add(symbolList);
```

Tablo 17’ nin devamı

```
    ++cur_lvl;
  }
  public void beginScope(int m){
    ++cur_lvl;
    SymbolList symbolList = new SymbolList(m<0 ? smyTableList.get(0) : cur_scope);
    smyTableList.add(symbolList);
    cur_scope = symbolList;
  }
  public void endScope() { ... }
  public void put(String id, Object obj){
    SymbolList node = search(id);
    if(node==null)
      cur_scope.put(id,obj);
    else
      node.put(id,obj);
  }
  public Object get(String id) { ... }
  public void beginLazy(int m){ cur_scope = smyTableList.get(m); }
  public void endLazy(){ cur_scope = smyTableList.get(cur_lvl); }
...
}
class SymbolList<K,T> extends Hashtable<K,T>{ ... }
class FunctionHeader { ... }
```

Bu çalışmada, anlamsal analiz iki aşamalı bir yaklaşımla ele alınmaktadır. İlk aşama, programdaki tüm tanımlayıcıların tür bilgilerini içeren bir sembol tablosu oluşturulmasını içerir. İkinci aşamada ise, oluşturulan sembol tablosu kullanılarak ifadelerin tür kontrolü gerçekleştirilir.

İki aşamalı analizin benimsenmesinin temel gerekçesi, fonksiyonların karşılıklı olarak özyinelemeli olabilmesidir. Bu durum, anlamsal analizi önemli ölçüde karmaşıklaştırabilir, çünkü bir fonksiyonun türü, henüz analizi tamamlanmamış başka bir fonksiyona olan bağımlılığı nedeniyle belirlenemeyebilir. İki aşamalı yaklaşım, bu soruna etkin bir çözüm sunar. Bu sayede, bir fonksiyon çağrısının tür denetimi sırasında, ilgili bilgilere sembol

tablosundan her zaman erişilebilmektedir. İlk aşamada, fonksiyon çağrılarının tür kontrolü ertelenir ve tüm fonksiyon tanımları işlenerek sembol tablosuna eklenir. Bu aşamada sadece fonksiyon imzaları (adı, parametre türleri ve dönüş türü) dikkate alınır. İkinci aşamada ise, tüm fonksiyon tanımları ve tür bilgileri sembol tablosunda mevcut olduğundan, ifadelerin tür kontrolü güvenli bir şekilde gerçekleştirilebilir. Bir fonksiyon çağrısına rastlandığında, sembol tablosu kullanılarak fonksiyonun imzası kontrol edilir ve çağrının tür açısından doğru olup olmadığı doğrulanır. Anlamsal analizin bu iki aşaması, IVisitor arayüzünden uygulanan SymbolVisitor ve TypeVisitor sınıfları tarafından gerçekleştirilir.

SymbolVisitor sınıfı, AST' yi dolaşarak ve her bir tanımlayıcı için sembol tablosu girdileri oluşturarak programdaki tanımlayıcıların anlamını izleme görevini üstlenir. Bu sınıf, farklı AST düğüm türlerini işlemek için bir dizi visit() metodu içermektedir.

Örneğin, Tablo 18' de gösterilen visit() metodu, değişken tanımlarıyla ilgilenmektedir. Bu metot, belirli bir kapsam düzeyindeki bir değişkenin adını ve türünü sembol tablosuna ekler. Ayrıca, bir değişkenin birden fazla türde değerle tanımlanıp tanımlanmadığını kontrol eder ve böyle bir durumda bir hata mesajı oluşturur. Bu sayede, aynı kapsam içinde aynı ada sahip ancak farklı türlerde birden fazla değişken tanımlanmasının önüne geçilir.

Tablo 18. Toplama işlemi için tür denetimi

```
public Object visit(Plus e) {
  Type a = (Type)e.a.accept(this);
  Type b = (Type)e.b.accept(this);
  if(a == null || b == null) return null;
  else if (a.isInt() && b.isInt()) return new IntType();
  else if(a.isString() || b.isString()) return new StringType();
  else if(a.isMatrix() || b.isMatrix()) {
    if(a.isMatrix() && b.isMatrix()) return new MatrixType();
  else return null; }
  else if(a.isDouble() || b.isDouble()){
    if (!a.isDouble()) e.a = new DNum((Num)e.a);
    if (!b.isDouble()) e.b = new DNum((Num)e.b);
    return new DoubleType(); }
  else if(a.isRational() || b.isRational()){
```

Tablo 18’ in devamı

```
    if (!a.isRational()) e.a = new RNum((Num)e.a);
    if (!b.isRational()) e.b = new RNum((Num)e.b);
    return new RatioType(); }
  return null;
}
```

Geliştirilen dilin semantik analiz sürecinde, farklı türlerdeki öğeler üzerinde işlem yapılırken tür uyumu kontrol edilir. Örneğin, toplama işleminde bir ifade tam sayı, diğeri rasyonel sayı ise sonucun türü, önceden tanımlanmış kurallar aracılığıyla rasyonel sayı olarak belirlenir. Bu tür çıkarımı, programın güvenilir ve hatasız çalışmasını sağlar. Ayrıca, tam sayı ifadesini temsil eden söz dizim ağacı düğümü, rasyonel sayı ifadesine dönüştürülür. Bu sayede, değerlendirme aşamasında tekrar tekrar tür kontrolü yapılmasının önüne geçilir ve program daha verimli çalışır.

Örnek İfadeler:

- X = 1+1.5;

  Girdi: Var(“X”, new Plus(new IntNum(1), new DNum(1.5));

  Çıktı: Var(“X”, new Plus(**new DNum(1)**, new DNum(1.5));

- X = 1+1/2;

  Girdi: Var(“X”, new Plus(new IntNum(1), new RNum(1,2));

  Çıktı: Var(“X”, new Plus(**new RNum(1,1)**, new RNum(1,2));

2.3.2. Yorumlama

Değerlendirme aşaması, anlamsal analiz sırasında tanımlayıcı bağlamalar ve örtülü dönüşümler gibi türe dayalı işlemlerle süslenmiş soyut sözdizimi ağacı üzerinde çalışır. Soyut sözdizimini doğrudan daha düşük düzeyde karşılık gelen kod parçalarına çevirmek mümkündür. Bu tür bir çeviri, talimat seçimi, kayıt tahsisi ve kod üretimi gibi genellikle hedef makineye bağımlı olan birçok başka aşamayı içerir. Geliştirilen dile ait programlar, bu tür makineye özgü sorunlara bağlı kalmaksızın, Java Sanal Makinesine dinamik olarak yüklenen bir değerlendirici tarafından yorumlanmaktadır. Yürütülen programlar üzerinde herhangi bir optimizasyon uygulanmaz çünkü bunlar ara gösterime çevrilmez. Çalışmadaki dilin değerlendiricisi, her sözdizimi sınıfı için bir *visit()* yöntemine sahip olan *EvalVisitor*

adlı bir ziyaretçi sınıfı tarafından temsil edilmektedir. Bu yöntemler, soyut sözdizimi ağacını yinelemeli olarak geçerek ve sözdizimi sınıflarına ait nesneleri içeren düğümleri yorumlayarak birbirleriyle iş birliği yapar. Tablo 19, Plus türündeki nesneleri değerlendiren *visit()* yönteminin uygulanmasını göstermektedir.

Tablo 19. Toplama işlemi için *visit()* metodu

```
public Object visit(Plus e) {
   Object a = e.a.accept(this);
   Object b = e.b.accept(this);
   if(a instanceof Num && b instanceof Num)
  { return ((Num)a).plus((Num)b); }
  else if (a instanceof String || b instanceof String)
  { return a.toString() + "" + b.toString(); }
...
}
```

Çalıştırılan bir programdaki yerel ve global kapsamlı tüm değişkenlerin anlık değerleri, 2.3.2 bölümünde sunulan *SymTable* sınıfının bir nesnesinde, tıpkı (sembol ve tip) çifti gibi, (değişken ve değer) çifti olarak saklanır. Nesnenin durumu, kapsamları tarafından belirlenen değişkenlerin (yani yerel, global ve formal olanlar) yaşam süreleri boyunca sürekli olarak güncellenir.

Anlamsal analiz, bir programın yapısını ve anlamını inceleyerek, programın dilin kurallarına uygun olup olmadığını denetleyen bir aşamadır. Bu süreçte, değişkenlere ve ifadelere atanan türler ve tipler üzerinde yüzeysel bir inceleme gerçekleştirilir. Yani, anlamsal analiz aşamasında, programın genel yapısının ve tür sisteminin tutarlılığı kontrol edilir, ancak gerçek değerler üzerinden detaylı bir inceleme yapılmaz. Ancak, bir programın hatasız çalışabilmesi için sadece tür ve tip uyumluluğunun kontrol edilmesi yeterli değildir. Programın çalışması sırasında ortaya çıkabilecek ve değerlerle ilgili olan hataların da tespit edilmesi gerekir. İşte bu noktada, yorumlama süreci devreye girer.

Yorumlama süreci, programın adım adım çalıştırılarak, her adımda değişkenlere atanan gerçek değerlerin kontrol edilmesini sağlar. Bu sayede, anlamsal analiz aşamasında

tespit edilemeyen, ancak programın çalışması sırasında ortaya çıkabilecek hatalar da yakalanabilir.

Örnek olarak matrislerin çarpımının değerlendirilmesi, Times türünde bir nesnenin formal parametresine sahip *visit()* metodu içinde gerçekleştirilir. İşlemin iki operandı da matris ise, matris çarpımı kuralına göre, ilk matris olan A_{nxm}' deki sütun sayısının ikinci matris olan B_{mxp}' deki satır sayısına eşit olup olmadığı doğrulanır. Bu doğrulama adımı, iki matrisi çarpmadan hemen önce boyut tutarlılığını garanti eder. İlgili kod parçası Tablo 20' de verilmiştir.

Tablo 20. Çarpma işlemi için *visit()* metodu

```
public Object visit(Times e) {
  Object a = e.a.accept(this);
  Object b = e.b.accept(this);
  if(a instanceof Matrix && b instanceof Matrix) {
    Matrix m1 = (Matrix) a, m2 = (Matrix) b;
    If (m1.getN() != m2.getM()){
      System.out.println("Error: Incorrect dimensions for matrix multiplication: line "+
e.line + " column " + e.column);
      return null;
  }
  }
  ...
}
```

Başka bir örnek olarak matrislerin indekslenmesi verilebilir. Geliştirilen dilde, matrislerin indekslenmesi için Index ve Range tipindeki nesnelerle temsil edilen iki olası sözdizimi sınıfı vardır. İlk nesne (Index) bir tam sayıya göre değerlendirilirken, ikinci nesne (Range) "start" ve "end" nitelikleriyle başlayıp bitebilen bir tam sayı aralığıyla ilişkilendirilir. Her iki sınıfın (Range ve Index) tanımları Tablo 21' de verilmiştir.

Tablo 21. Matris indis tanımları

```
public class Index extends Exp {
        public Exp a;
        public Index(Exp x){     a = x;  }
... }
 public class Range extends Exp {
         public Index start,end;
         public Range(Index x, Index y){
                start=x;  end=y;
         }
 ...
}
```

Yorumlama süreci sırasında, matris indekslerinin geçerliliğinin doğrulanması gerekir. Bu, indeks değerinin hem pozitif bir tam sayı hem de ilgili matris boyutundan küçük olmasını sağlamayı içerir. Tablo 22' te sağlanan *chkIndex()* metodu indeks geçerliliğini doğrulamak için kullanılır.

Tablo 22. Matris indis kontrolleri yapan *chkIndex()* metodu

```
public IntNum chkIndex(Index e, int n){
    Object a = e.a.accept(this);
    if(a instanceof IntNum)
     if(((IntNum)a).n < 0)
        System.out.println("Hata: Indis 0'dan küçük olamaz: line " + e.line + " column " +
e.column);
     else if(((IntNum)a).n >= n)
        System.out.println("Hata: Indis aşımı: line "+ e.line + " column " + e.column);
…
}
```

Ek olarak, bir matrisin boyutu, *rotate()* ve *merge()* gibi bazı özel işlevler aracılığıyla dinamik olarak değiştirilebilir. Statik analiz sırasında, mümkünse yalnızca eleman türü

doğrulanır. Bu nedenle, bir değişkene bir matrisle ilk atama yapıldıktan sonra, aynı yerel kapsamdaki o değişkenin daha sonraki tüm atamaları, yalnızca farklı boyutlarda olabilen matrislerle sınırlandırılır.

Değerlendirme sürecinin önemli bir aşaması, belirli bir kapsam dahilinde oluşturulan değişkenlerin takibini gerçekleştirmektir. Fonksiyonların gerçek parametrelerine erişim, tembel bir mekanizma ile sağlanır. Daha açık bir ifadeyle, eğer bir gerçek parametre bir ifade ise, çağrılan fonksiyonun gövdesi içerisinde, bu parametrenin karşılık geldiği biçimsel parametreye ilk erişim gerçekleştiğinde değerlendirilir. Dolayısıyla, yerel değişkenlerin kapsamı, genellikle tanımlandıkları fonksiyon gövdelerinin ötesine geçebilir. Tablo 23' te sunulan LazyEval sınıfı, belirli bir gerçek parametrenin değerlendirilmesinin gerekli olduğu kapsam numarasını saklamak amacıyla kullanılır.

Tablo 23. Tembel değerlendirme sınıfı gövdesi

```
class LazyEval {
  Exp e;
  int scope;
  public LazyEval(Exp a, int b)
    { e = a; scope = b; }
}
```

Tembel değerlendirme yöntemi (diğer bir deyişle "ihtiyaca göre çağırma"), fonksiyonların sabit bir değer içeren desenler kullanılarak tanımlanması durumunda uygulanmaz. Genel bir kural olarak, biçimsel bir parametre ya bir desen ya da basit bir değişkendir. Desen ise, sabit bir değer veya "n + k" (n bir değişken adı ve k pozitif bir tam sayı olmak üzere) formunda bir ifade olarak karşımıza çıkar. Bir fonksiyon, her farklı desen için ayrı bir gövdeye sahip olabilir ve bu da birden fazla denklemin ortaya çıkmasına neden olur. Bu tür bir fonksiyon çağrıldığında, öncelikle gerçek parametre değerlendirilir ve ardından desenler yukarıdan aşağıya doğru kontrol edilir. İlk eşleşen desen, ilgili fonksiyon gövdesinin (veya denklemin) değerlendirilmesini tetikler. Tablo 24' te yer alan *match()* fonksiyonu, çağrılan fonksiyonun gerçek parametreleri ile uyumlu biçimsel parametrelere sahip fonksiyonun uygun denklemini seçmek için kullanılır.

Tablo 24. Tembel değerlendirme için fonksiyon denkleminin belirlenmesi

```
// Object[] argn; // Gerçek parametreleri tutar
// String[] args; //Biçimsel parametreleri tutar
boolean match(Function fl, Function fr) {
  int argi = 0;
  EList eql = fl.eq; //Çağrılan fonksiyonun gerçek parametrelerinin listesi
  EList eqr = fr.eq; //Çağıran fonksiyonun biçimsel parametrelerinin listesi
  while (eql != null) {
  if (eql.e == null || eqr.e == null)
    return (eql.e == null && eqr.e == null);
  if (eql.e instanceof Var) { //Biçimsel parametre basit bir değişkendir
    args[argi] = ((Var)(eql.e)).id;
    argn[argi++] = new LazyEval(eqr.e, symboltable.getScope());
  } else if (eql.e instanceof Plus) { //Biçimsel parametre n+k biçimindedir
    args[argi] = ((Var)((Plus)(eql.e)).a).id;
    argn[argi++] =
    new LazyEval(new Minus(eqr.e, ((Plus)(eql.e)).b), symboltable.getScope());
  } else { //the formal param is a pattern with a constant value
    Object a = eqr.e.accept(this);
    args[argi] = "";
    argn[argi++] = a;
    if (a instanceof Integer && ((Integer)a).intValue()!=((Num)(eql.e)).n)
      return false;
    if (a instanceof Double && ((Double)a).doubleValue()!=((DNum)(eql.e)).n)
      return false;
  }
  eql = eql.eq;
  eqr = eqr.eq;
}
  return true;
}
```

Aşağıdaki örnekte, tembel değerlendirme mekanizması sayesinde, a fonksiyonu çağrıldığında 10/0 ifadesi anında hesaplanmaz. Dahası, fonksiyonun gövdesinde x parametresine herhangi bir şekilde erişilmediği için, 10/0 ifadesi hiçbir zaman değerlendirilmeyecek ve dolayısıyla sıfıra bölme hatası oluşmayacaktır.

```
a(x){print("a fonksiyonu")}=1
main(){ a(10/0); } = 1
```

2.3.3. Yorumlayıcı Entegrasyonu

Bu bölüm, önceki bölümlerde açıklanan yorumlayıcı bileşenlerinin entegrasyonunu ele almaktadır. Yorumlayıcı, üç ana bileşenin yani sözdizimi denetleyicisinin, anlamsal denetleyicinin ve kod değerlendiricisinin entegrasyonu ile oluşturulur. Bu bileşenler, üç farklı metodun çağrılmasıyla başlatılır: *parse()* metodu, *typeCheck()* metodu ve *evaluate()* metodu.

Tablo 25. Yorumlayıcı çalıştırma parametreleri

Argüman	**Çağrılan Metot**
-p	*parse()*
-c	*parse()* → *typeCheck()*
-r	*parse()* → *typeCheck()* → *evaluate()*

parse() metodu, kaynak kod üzerinde gerçekleştirilen sözcüksel ve sözdizimsel analiz süreçlerini kapsayarak, programın AST temsilini oluşturur. Devamında, *typeCheck()* metodu devreye girerek oluşturulan AST' nin, dilin tip sistemi kurallarına uygunluğunu doğrular. Son aşamada ise, *evaluate()* metodu, doğrulanmış AST' yi yorumlayarak programın çalıştırılmasını sağlar. Herhangi bir ayrıştırma, tür uyuşmazlığı veya çalışma zamanı hatası durumunda, ilgili hata raporları kullanıcıya sunulur. Bu metotların detaylı uygulama adımları Tablo 26' da sunulmaktadır.

Tablo 26. Yorumlayıcı çalıştırılabilir sınıf tanımı

```
public class Interpreter {
  static Program parse(String filePath){
    try {
      System.setIn(new FileInputStream(filePath));
      return new Parser(System.in).Start(); }
    catch(ParseException | FileNotFoundException ex){
      System.out.println(ex.getMessage());
      return null; }
  }
  static int typeCheck(Program p){
    TypeVisitor type = new TypeVisitor(p);
    type.start();
    return type.error == true ? −1 : 0; }
  static int evaluate(Program p){
    EvalVisitor eval = new EvalVisitor(p);
    eval.start();
    return eval.error == true ? −1 : 0; }
  public static void main(String[] args) {
    String command = args[1];
    int cmd=0;
    if (command.equals("p")) cmd = 1;
    else if (command.equals("c")) cmd = 2;
    else if (command.equals("r")) cmd = 3;
    if (cmd > 0) {
    Program p = SyntaxControl(args[0]);
    if(p==null) return;
    if (cmd>1)
    if(semanticControl(p) <0) return;
    if (cmd>2)
    if(evaluate(p) <0) return; } }
}
```

Yorumlayıcının oluşturulması, tüm kaynak kod dosyalarının aynı dizin altında toplanması ve ardından aşağıdaki komutların sırasıyla çalıştırılması ile gerçekleştirilir:

```
$> javacc Parser.jj
$> javac *.java
```

JavaCC ayrıştırıcı üreteci, Parser.jj dosyasında bulunan tanımlara göre ayrıştırıcı sınıflarını oluşturur. Yorumlayıcının derleme süreci, çalıştırılabilir Interpreter.class dosyasını üretir. Bu çalıştırılabilir dosya, program kodunu içeren dosyanın yolu ve istenen çıktı seçeneği komut satırında belirtilerek çalıştırılabilir. Yorumlayıcının komut satırı parametreleri ve işlevleri Tablo 27' de açıklanmıştır.

Tablo 27. Yorumlayıcının komut satırı parametreleri ve işlevleri

Seçenekler	**Anlamı**
-p	Program kodunu ayrıştırma ve AST yazdırma.
-c	Program kodunu derleme ve semantik analizleri yapar
-r	Program kodunu koşma (varsayılan)

Aşağıda, örnek bir program dosyası için yorumlayıcı programın nasıl çalıştırılacağına dair bir örnek verilmiştir:

```
$> java Interpreter -r sample_program.mx
```

2.4. Dilin Özellikleri

Bu programlama dili, basitlik ve esneklik göz önünde bulundurularak tasarlanmıştır ve aşağıdaki özelliklere sahiptir:

Genel Özellikler:

- *main()* fonksiyonu, programın başlangıç noktasıdır ve çalıştırma buradan başlar.
- Değişkenler için tür tanımlaması gerekmez. Bir değişkenin türü, ilk atanan değerin türüne göre otomatik olarak belirlenir.
- Global değişkenlerin değeri sabittir.
- Emirsel blokta koşul ifadeleri (*if*) ve döngüler tanımlanabilir (*while*).
- Tembel değerlendirme stratejisine sahiptir.

Tembel değerlendirme:

Tembel değerlendirme, bir programlama dili özelliğidir ve ifadelerin değerlendirilmesini, değerlerine gerçekten ihtiyaç duyulana kadar ertelemeyi sağlar. Başka bir deyişle, bir ifade ancak sonucu gerçekten gerektiğinde değerlendirilir. Geliştirilen dil bu mekanizmaya sahiptir.

Veri Türleri:

Dil, çeşitli veri türlerini desteklemektedir:

- Tamsayı (Integer): Tam sayıları temsil etmek için kullanılır.
- Ondalıklı Sayı (Floating-Point): Ondalıklı sayıları temsil etmek için kullanılır.
- Rasyonel Sayı (Rational): Kesirli sayıları hassas bir şekilde temsil etmek için kullanılır.
- Karakter Dizisi (String): Metin verilerini depolamak için kullanılır.

Matris (Matrix): Sayısal verileri (Tamsayı, Ondalıklı ve Rasyonel sayı) iki boyutlu bir yapıda depolamak için kullanılır. Dildeki bir matris gösterimi Tablo 28' deki gibidir.

Tablo 28. Matris veri türü

A = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13, 14, 15, 16]];	

	Kolon 0	Kolon 1	Kolon 2	Kolon 3
Satır 0	1	2	3	4
Satır 1	5	6	7	8
Satır 2	9	10	11	12
Satır 3	13	14	15	16

Tablo 29' da matrisler elemanlarına geliştirilen dilde nasıl erişildiğine dair örnekler yer almaktadır.

Tablo 29. Matris elemanlarına erişim örnekleri

	Girdi	**Çıktı**	**Açıklama**
1.	A[1, 1]	6	A matrisinin tek elemanına erişmek:
2.	A[1, :]	5, 6, 7, 8	A matrisinin ilk satırına erişmek
3.	A[:, 1]	2, 6, 10, 14	A matrisinin ilk sütununa erişmek
4.	A[1:2, :]	6,10	A matrisin belli bir parçasına erişmek

Tablo 29 da yer alan iki nokta üst üste operatörü ":", ayrıca aralık operatörü olarak da bilinir, bir matrisin tüm satırına veya sütununa erişmek için kullanılır. İki nokta üst üste operatörü virgülün sağ tarafına yerleştirildiğinde, matrisin tüm satırına erişilmesi gerektiğini belirtir. Bu durum, 2. örnekte, A[1, :] ifadesinin A matrisinin ikinci satırının tamamını döndürdüğü yerde görülebilir. Tersine, iki nokta üst üste operatörü virgülün sol tarafına yerleştirildiğinde, matrisin tüm sütununa erişilmesi gerektiğini belirtir. Bu durum 3. örnekte, A[:, 1] ifadesinin A matrisinin ikinci sütununun tamamını döndürdüğü yerde görülebilir. Dahası, iki nokta üst üste operatörü iki tam sayı arasına yerleştirildiğinde, tam sayılar arasındaki belirli satır veya sütunlara erişilmesi gerektiğini belirtir. 4. örnekte, A[1 : 2, :] ifadesi A matrisinin ikinci ve üçüncü satır aralığını döndürür.

Hazır Fonksiyonlar:

- *print(a,b,..)*: Konsola çıktı vermek için kullanılır. Parametresiz çağrıldığında yeni bir satıra geçerken, bir veya daha fazla parametre ile çağrıldığında bu parametreleri aralarında boşluk bırakarak yazdırır.
- *size(x)*: Girdi olarak bir $\boldsymbol{x}$ matrisi alır ve bu matrisin boyutlarını temsil eden bir matris döndürür.
- *matrix(n, m)*: n ve m tamsayı değerlerini girdi olarak alarak $n \; x \; m$ boyutlarında, tüm elemanları sıfır olan yeni bir matris döndürür.
- *merge(A,B)*: A ve B matrislerini girdi olarak alır ve bu matrislerin birleştirilmesiyle oluşan yeni bir matris döndürür

2.5. Örnek Programlar

Bu kısımda, dilin söz diziminin açıklanması ve işlevselliğini göstermek amacıyla çeşitli örneklere yer verilmiştir.

2.5.1. Fibonacci Dizisi

Örneklerin ilki Fibonacci dizinin geliştirilen dilde kodlanmasına yöneliktir. Aşağıdaki Tablo 30' da geliştirilen dilde Fibonacci dizisinin nasıl hesaplanabileceğini gösteren üç farklı fonksiyon örneği bulunmaktadır.

Tablo 30. MxPL Fibonacci dizisi fonksiyon örnekleri

Matematiksel gösterim biçimi
fib(0) = 0 fib(1) = 1 fib(k) = fib(k-1) + fib(k-2)
Parçalı fonksiyon gösterim biçimi
fib(k) = { 0, k==0 } { 1, k==1 } { fib(k-1) + fib(k-2), otherwise }
Emirsel yinelemeli fonksiyon biçimi

fib(k) {
  n1=0;
  n2=1;
  n3=0;
  i=0;
  while(i<k)
{
  n3=n1+n2;
  n1=n2;
  n2=n3;
  i=i+1;
  }
}=n3

Bu örnekler, geliştirilen dilde Fibonacci dizisinin nasıl hesaplanabileceğine dair farklı yaklaşımları göstermektedir. Her fonksiyonun performansı ve karmaşıklığı farklılık gösterebilir.

2.5.1. Yarılama Yöntemi

Yarılama yöntemi (İkiye bölme yöntemi), sürekli bir fonksiyonun kökünü başka bir deyişle fonksiyonun sıfır olduğu noktayı bulmak için kullanılan iteratif bir yöntemdir. Bu yöntem, verilen bir aralıkta fonksiyonun işaret değiştirdiğini (pozitiften negatife veya tam tersi) tespit ederek çalışır ve bu aralığı adım adım daraltarak köke yaklaşır. Adımları Algoritma 1' deki gibidir:

Algoritma 1: Yarılama Yöntemi

Girdi: Kökü bulunacak olan fonksiyon f, tolerans değeri ε

Çıkış: f fonksiyonunun kök değeri c

1) Başlangıç: Kökünün bulunacağı aralığı belirlenir. Bu aralık [a, b] olsun ve f(a) ile f(b) zıt işaretli olsun.

2) Orta Noktayı Hesaplama: Aralığın orta noktası (1)' deki gibi hesaplanır:

$$c = (a + b) / 2 \quad (1)$$

3) İşaret Kontrolü gerçekleştirilir. Eğer;

   a. $f(c) = 0$ ise: c, fonksiyonun köküdür.

   b. $f(c)$ ile $f(a)$ aynı işaretli ise: Kök, $[c, b]$ aralığındadır. $a = c$ yaparak aralık daraltılır.

   c. $f(c)$ ile $f(b)$aynı işaretli ise: Kök, $[a, c]$ aralığındadır. $b = c$ yaparak aralık daraltılır.

4) Tekrar: 2. ve 3. adımlar, belirlenen bir tolerans değerine ε ulaşılana kadar veya istenen iterasyon sayısına kadar tekrarlanır. Tolerans değeri, bulunan kökün gerçek köke ne kadar yakın olması gerektiğini belirtir.

Algoritma 1'de adımları verilen yarılama yöntemine ait geliştirilen dilde yazılmış program Tablo 31' de verilmiştir.

Tablo 31. Yarılama yöntemi için MxPL program örneği

```
f(x) = exp(x)-3^x+4

BS(x,y) { z=(x+y)/2; print(x,z,y); print(); } =
    { z , abs(x-z) < 0.001 }
    { BS(x,z) , f(x) * f(z) < 0 }
    { BS(z,y) , otherwise }

main() { print("x  ", "z  ", "y"); print(); } = BS(1.0,3.0)
```

2.5.1. Yer Değiştirme Yöntemi

Yer değiştirme yöntemi, sürekli bir fonksiyonun kökünü, yani fonksiyonun sıfır olduğu noktayı bulmak için kullanılan iteratif bir yöntemdir. Bu yöntem de, verilen bir aralıkta fonksiyonun işaret değiştirdiğini tespit ederek çalışır. Ancak, Yarılama Yöntemi' nin aksine, aralığı her adımda yarıya bölmek yerine, fonksiyon değerlerini kullanarak köke daha hızlı yaklaşmayı hedefler. Adımları Algoritma 2' deki gibidir:

Algoritma 2: Yer Değiştirme Yöntemi

Girdi: Kökü bulunacak olan fonksiyon f, tolerans değeri ε

Çıkış: f fonksiyonunun kök değeri c

1) Başlangıç: Kökünün bulunacağı aralığı belirlenir. Bu aralık [a, b] olsun ve f(a) ile f(b) zıt işaretli olsun.

2) Orta Noktayı Hesaplama: Aralığın orta noktası (2)' deki gibi hesaplanır:

$$c = \frac{a*f(b)-b*f(a)}{f(b)-f(a)} \quad (2)$$

3) İşaret Kontrolü gerçekleştirilir. Eğer;

   a. $f(c) = 0$ ise: c, fonksiyonun köküdür.

b. $f(c)$ ile $f(a)$ aynı işaretli ise: Kök, $[c, b]$ aralığındadır. $a = c$ yaparak aralık daraltılır.

c. $f(c)$ ile $f(b)$aynı işaretli ise: Kök, $[a, c]$ aralığındadır. $b = c$ yaparak aralık daraltılır.

4) Tekrar: 2. ve 3. adımlar, belirlenen bir tolerans değerine ε ulaşılana kadar veya istenen iterasyon sayısına kadar tekrarlanır. Tolerans değeri, bulunan kökün gerçek köke ne kadar yakın olması gerektiğini belirtir.

Algoritma 2'de adımları verilen yarılama yöntemine ait geliştirilen dilde yazılmış program Tablo 32' te verilmiştir.

Tablo 32. Yer değiştirme yöntemi için MxPL program örneği

```
f(x) = exp(x)-3^x+4

RF(x,y) {
  z = (x*f(y)-y*f(x))/(f(y)-f(x));
  print();
  print(x, z, y);
} = { z, abs(x-z) < 0.001 }
   { RF(x,z), f(x) * f(z) < 0 }
   { RF(z,y), otherwise }

main() { print("x", "z", "y"); } = RF(1.0,3.0)
```

2.5.1. Matris Transpozu

Bir matrisin transpozunu hesaplamak, doğrusal cebirde sıkça karşılaşılan temel bir işlemdir. Transpoz işlemi, orijinal matrisin satırlarını sütunlara, sütunlarını da satırlara dönüştürerek yeni bir matris oluşturur. Tablo 33' de, bir matrisin transpozunu hesaplamak için kullanılan iki farklı programlama yaklaşımını (emirsel ve fonksiyonel) gösteren örnek fonksiyonlar bulunmaktadır:

Tablo 33. Matris transpozu fonksiyon örneği

Fonksiyonel tarzda yazım

transpose(x){
  s=size(x); n=s[0,0];
} =
  { [], n==0 }
  { append(rotate(x[:,1]), transpose(x[:,1:]), otherwise }

Emirsel tarzda yazım

transpose(x) {
 s = size(x);
 t = matrix(s[0,1], s[0,0]);
 i = 0;
 while(i < s[0,0]){
 t[:,i] = x[i,:];
 i = i + 1; }
 } = t

Yukarıda verilen transpose(x) fonksiyonu, iki yardımcı fonksiyon kullanır: size(x) ve matrix(n, m). size(x) fonksiyonu, x matrisinin boyutunu bir matris olarak döndürür. matrix(n, m) fonksiyonu, n x m boyutunda yeni bir sıfır matrisi oluşturur.

2.5.1. Determinant

Bir kare matrisin determinantı, matrisin elemanları kullanılarak hesaplanan ve matrisin bazı özelliklerini temsil eden bir sayıdır. Bu değer, bir matrisin tersini bulmak ve doğrusal denklem sistemlerini çözmek gibi çeşitli uygulamalarda kullanılır. Tablo 34, bir matrisin determinantını hesaplayan Python dilinde yazılmış bir fonksiyon örneği sunmaktadır. Ayrıca geliştirilen dilde hem emirsel hem de fonksiyonel yaklaşımların bir arada kullanıldığı matrisin determinantını hesaplayan bir fonksiyon örneği, Tablo 35’ te sunulmuştur.

Tablo 34. Determinant hesabı yapan Python kod örneği

```
def determinant(A, total=0):
  indices = list(range(len(A)))
  if len(A) == 2 and len(A[0]) == 2:
    val = A[0][0] * A[1][1] - A[1][0] * A[0][1]
    return val

  for fc in indices:
    As = A.copy()
    As = As[1:]
    height = len(As)

    for i in range(height):
      As[i] = As[i][0:fc] + As[i][fc+1:]

    sign = (-1) ** (fc % 2)
    sub_det = determinant(As)
    total += sign * A[0][fc] * sub_det
  return total
```

Tablo 35. MxPL'de determinant hesaplayan program örneği

```
cof(x,j) {
  y=x[1:,:];
  z=merge(y[:,0:j-1],y[:,j+1:]);
 } = (-1^(j))*x[0,j]*det(z,j)
det(x,j) {
  s = size(x);
  n = s[0,0];
 }
 = { x[0,0], n==1 }
    { 0, j==n }
    { cof(x,j) + det(x,j+1), otherwise }
```

Tablo 35’ deki MxPL kodu örneği, fonksiyonel ve emirsel yaklaşımların hibrit kullanıldığı, iki fonksiyondan ve toplam 31 sözcükten oluşmaktadır. Buna karşılık Python örneğinde, iki döngü içeren emirsel bir fonksiyon, toplam 66 sözcükle tanımlanmıştır. Bu durum, MxPL'nin Python'a kıyasla daha öz bir yapıya sahip olduğunu ve daha az kodla aynı işlevi gerçekleştirebildiğini göstermektedir.

3. BULGULAR VE TARTIŞMA

Bu bölümde geliştirilen dilin, modern programlama dillerinin temel prensipleri ve özellikleri de dahil olmak üzere bazı önemli yönleri kullanılarak karşılaştırmalı olarak analizi gerçekleştirilmiştir.

İlk analiz Bölüm 1.9 ‘da detaylı olarak anlatılan programlama dillerinin değerlendirilmesine kullanılan ölçütler (ayrıca tasarım prensipleri olarak da adlandırılır) bazında gerçekleştirilmiştir. Okunabilirlik, taşınabilirlik ve verimlilik gibi ölçütler, C, C++, Java, Python ve Haskell dilleri ile geliştirilen dili karşılaştırmak için kullanılmaktadır. Dil ölçütleri üzerinde önemli ölçüde farklı etkilere sahip programlama paradigmaları da karşılaştırmaya dahil edilmiştir. Karşılaştırmaya dayalı sonuçlar Tablo 36’ da verilmiştir. Tablo 36’ da “X” ifadeleri dillerin bu özelliğe sahip olduğunu, boş alanlar da bu özelliğin o dilde olmadığını belirtmektedir.

Tablo 36. Değerlendirme metrikleri ve programlama paradigmaları ile dillerin karşılaştırılması

Ölçütler	**MxPL**	**C**	**C++**	**Java**	**Python**	**Haskell**
Emirsel	X	X	X	X	X	
Nesneye Yönelik			X	X	X	
Fonksiyonel	X				X	X
Verimlilik	X	X	X	X	X	X
Genişletilebilirlik		X	X	X	X	X
Sürdürülebilirlik	X	X	X	X	X	X
Ortogonallik	X				X	X
Taşınabilirlik	X			X	X	
Okunabilirlik	X	X	X	X	X	X
Güvenilebilirlik	X	X	X	X	X	X
Tekdüzelik	X			X		X
Yazılabilirlik	X	X	X	X	X	X

Tablo 36 programlama paradigmaları açısından incelenecek olunursa MxPL’in C, C++, Java ve Python gibi emirsel programlamayı desteklediği gözükmektedir. Yani

programlar bir dizi komut veya adım olarak belirtilmektedir. C++, Java ve Python, sınıflar ve nesneler aracılığıyla verileri ve yöntemleri düzenlemeye odaklanan nesne yönelimli programlamayı desteklerken; Geliştirilen dil böyle programlama yapısına sahip değildir. Bu da karmaşıklık, nesnelerin verileri ve yöntemleri bir araya getirmesinden kaynaklı verilerin istenmeyen bir şekilde değiştirilmesine yol açabilecek yan etkiler ve buna bağlı zorlu hata ayıklama ve kodun öngörülebilirliğini azaltılması gibi dezavantajlara sahip olmadığı anlamına gelmektedir. Ayrıca nesneye yönelik programlamadaki dinamik bellek ayırma ve çöp toplama mekanizmaları, sayısal hesaplamaların gerçek zamanlı performans gereksinimleri için sorun oluşturabilmesi Geliştirilen dil için de bir avantaj olarak söylenebilir. Bunların dışında MxPL, Haskell ve bir dereceye kadar Python ile birlikte, fonksiyonları birinci sınıf yapılar olarak ele alan ve yan etkilerden kaçınmaya odaklanan fonksiyonel programlamayı desteklemektedir.

Tüm diller, verimliliği yüksek öncelikli olarak değerlendirmektedir. Ancak verimlilik, uygulama ve kullanım durumuna göre değişiklik gösterebildiği bir gerçektir. C++, Java ve Python, geniş kütüphanelere ve çerçevelere sahip oldukları için genellikle daha genişletilebilir kabul edilmektedir. Geliştirilen dil ise geniş kütüphane desteği sağlandığı takdirde genişletilebilir özelliğe erişeceği öngörülmektedir. C ve diğerleri gibi ortogonallikten ödün vererek esneklik ve performans sağlamayı amaçlayan pragmatik dillerin aksine, Sistemin farklı bileşenlerinin birbirinden bağımsız olarak çalışabilme yeteneğini ifade eden ortogonallığı geliştirilen dil desteklemektedir. Yani, bu dilde bir bileşende yapılan değişiklik, diğer bileşenleri etkilemeyecek veya minimum düzeyde etkileyecek şekilde tasarlanmıştır. Ayrıca geliştirilen dil platform bağımsızdır. Yaygın olarak bulunan Java Çalışma Zamanı Ortamı (JRE) ile çalışabildiği için taşınabilirlik ve kullanım kolaylığı sağlamaktadır. Bu sayede dil yorumlayıcısı, farklı cihazlarda sorunsuz bir şekilde çalışabilmektedir.

Sonuç olarak MxPL, emirsel ve fonksiyonel programlamanın bir karışımını destekleyen, verimliliği, güvenilirliği ve sürdürülebilirliği vurgulayan çok yönlü bir dil özelliklerine sahip olduğu söylenebilmektedir.

Diğer karşılaştırmalı analiz, belirli dil özelliklerine dayalı olarak da gerçekleştirilmiştir (Biswa vd., 2016). Geliştirilen dil, zorunlu ve fonksiyonel programlama paradigmalarına özgü ve ayırt edici bazı yapılar ve özellikler sunmaktadır. Tür çıkarımı, örüntü eşleme ve tembel değerlendirme gibi özellikler tamamen fonksiyonel iken, açık tür tanımlamaları, döngüler ve atamalar gibi özellikler zorunludur. Tablo 37, MxPL ile birlikte C, C++, Java,

Python ve Haskell gibi popüler programlama dillerini çeşitli programlama dili özellikleri açısından karşılaştırmaktadır. Yine bu tabloda da "X" ifadeleri dillerin bu özelliğe sahip olduğunu, boş alanlar da bu özelliğin o dilde olmadığını belirtmektedir.

Tablo 37. Belirli dil özellikleri üzerinden programlama dillerinin karşılaştırılması

Ölçütler	**MxPL**	**C**	**C++**	**Java**	**Python**	**Haskell**
Atama	X	X	X	X	X	X
Döngü İfadesi	X	X	X	X	X	
Tembel Değerlendirme	X					X
Örüntü Eşleme	X					X
Özyineleme	X	X	X	X	X	X
Tür Çıkarımı	X				X	X
Bölümlenmiş Fonksiyonlar	X					X
Koşullu İfadeler	X	X	X	X	X	X
Yerel Yan Etkiler	X	X	X	X	X	X
Açık Tür Tanımlaması		X	X	X	X	
Rasyonel Tür	X					X
Tanımlayıcı Kapsamı	İç içe	Düz	İç içe	İç içe	İç içe	İç içe
Fonksiyon Kapsamı	Düz	Düz	Düz	Düz	Düz	İç içe

Tablo 37 genel olarak incelendiğinde MxPL'in, hem zorunlu hem de fonksiyonel programlama paradigmalarını destekleyen hibrit bir dil olduğu gözükmektedir. C/C++, sistem programlama gibi performansın kritik olduğu alanlarda yaygın olarak kullanılan, daha çok zorunlu dillere odaklanan diller olduğu söylenebilirken; Java' nın ise nesne yönelimli programlamaya güçlü bir şekilde odaklanan ve geniş bir kullanım alanına sahip genel amaçlı bir dil olduğu gözükmektedir. Python, okunabilirliği ve çok yönlülüğü ile bilinen hem zorunlu hem de fonksiyonel programlama stillerini destekleyen bir dildir. Haskell ise tembel değerlendirme ve güçlü statik tür sistemi ile bilinen saf fonksiyonel bir programlama dilidir.

Tablo 37 özelliklere göre analiz edilirse atama, döngü ifadesi, koşullu ifadeler, yerel yan etkiler hemen hemen tüm dillerde bulunan temel yapı taşlarıdır ve zorunlu programlama yaklaşımının merkezinde yer almaktadır. Tembel değerlendirme, örüntü eşleme, tür çıkarımı, bölümlenmiş fonksiyonlar fonksiyonel programlama dillerinde daha yaygındır ve

MxPL, Haskell ve bir dereceye kadar Python tarafından desteklenmektedir. MxPL tembel değerlendirmeyi destekleyerek performans ve ifade gücü avantajları sağlamaktadır. Bu da bu dillerin ifade gücünü ve özlü kod yazma yeteneğini artırabilmektedir. MxPL, örüntü eşlemeyi desteklemektedir. Bu özellik, veri yapılarıyla çalışmayı kolaylaştırmakta ve daha güvenli kod yazımına yardımcı olmaktadır. MxPL, Haskell ve Python gibi dillerde bulunan ve kodun daha kısa ve öz olmasını sağlayan tür çıkarımını destekler. C, C++ ve Java ise statik tipli dillerdir ve açık tür tanımlamaları gerektirir. MxPL, Haskell gibi fonksiyonel programlama dillerinde yaygın olan bölümlenmiş fonksiyonları desteklemektedir. Bu özellik, fonksiyonları daha küçük ve yönetilebilir parçalara ayırmayı kolaylaştırır. Özyineleme, tüm dillerde bulunan temel bir kavramdır ve fonksiyonel programlamada özellikle önemlidir. Açık tür tanımlaması C++, Java ve Python gibi statik tipli dillerde yaygınken, MxPL ve Haskell' ın tür çıkarımını desteklemesi, türlerin her zaman açıkça belirtilmelerine gerek kalmayabileceğini göstermektedir. Bu durum kodun daha uzun olmasına neden olabilir ancak tür güvenliği ve hata ayıklama açısından avantajlar sağlar. Python ise dinamik tipli bir dildir. Rasyonel tür, MxPL ve Haskell' da bulunması ise, bu dillerin sayısal hesaplamalar veya sembolik matematik gibi alanlarda avantaj sağlayabileceğini göstermektedir. Kapsam açısından bakılırsa, MxPL ve Haskell hariç tüm dillerde fonksiyon kapsamı kullanılırken, tanımlayıcı kapsamı açısından farklılıklar vardır. Sonuç olarak, MxPL, hem zorunlu hem de fonksiyonel programlamanın güçlü yönlerini birleştirmeyi amaçlayan, ifade gücü ve esneklik sunan hibrit bir dil olduğu Tablo 38'den görünmektedir.

Tablo 38' de sunulan üçüncü karşılaştırmalı analiz, geliştirilen dilin temel özelliklerini gelişmiş bazı sayısal ortamlarla karşılaştırmaktadır. Bu önde gelen ortamların, çeşitli yetenek ve beceri seviyeleri bulunmaktadır. Tablo 38' de görüldüğü gibi, MATLAB, kapsamlı toolbox ve gelişen topluluğuyla öne çıkmaktadır. Bununla birlikte, bazı kullanıcılar, yüksek maliyeti nedeniyle erişilebilirliği daha düşük bulabilir. NumPy, MATLAB' a güçlü bir alternatif olan ve önemli hesaplama yeteneklerine sahip, açık kaynaklı ve ücretsiz bir kütüphanedir. Taşınabilirliği, Python ortamına olan bağımlılığı ile sınırlıdır ve kullanıcı dostu olması genellikle tartışmalıdır, bazıları onu MATLAB' dan daha az sezgisel bulmaktadır. Breeze ve R, her ikisi de ortama bağımlı olan diğer açık kaynak alternatifleridir. Performans iyileştirmeleriyle ilgili bazı üstün yönlere sahiptirler, örneğin R'nin istatistiksel yeteneği veya Breeze' in hızı ve bellek verimliliği.

Tablo 38. MxPL'in bazı sayısal hesaplama ortamları ile karşılaştırılması

Özellik	**Breeze**	**Numpy**	**MATLAB**	**ALGLIB**	**R**	**MxPL**
Dil	Scala	Python	Matlab	C/C++/Fortran	R	MxPL
Ücret	Ücretsiz	Ücretsiz	Ücretli	Ücretsiz/Ücretli	Ücretsiz	Ücretsiz
Güçlülük	Hız, bellek verimliliği	Çok yönlülük, kullanım kolaylığı	Güçlü ortam, Geniş toolbox	Çapraz Dil Uyumluluğu	İstatistikse l odak, grafikler	Öğrenme kolaylığı, taşınabilirlik
Zayıflık	Sınırlı İşlevsellik	Matlab'dan daha az kullanıcı dostu	Yüksek maliyet	Python kadar başlangıç dostu değil	Python'dan daha dik bir öğrenme eğrisi	Sınırlı İşlevsellik
Gereksinimler	Scala ortamı	Python ortamı	Matlab Ortamı	C/C++/Fortran ortamı	R ortamı	Java ortamı

Genellikle karmaşık veya yetersiz belgelenmiş sözdizimi ve farklı anlamlar taşıyabilen sözdizimsel bileşenler nedeniyle, birçok geliştirici için nispeten uzun bir öğrenme eğrisine sahiptirler. MxPL, minimum dış bağımlılıklar aracılığıyla öğrenme ve programlama kolaylığına ve platform bağımsızlığına öncelik vererek kendini farklılaştırır. Sadece yaygın olarak bulunan Java Çalışma Zamanı Ortamı'na (JRE) ihtiyaç duyan dil yorumlayıcısı, çeşitli cihazlarda sorunsuz bir şekilde çalışabilmektedir ve bu da onu taşınabilir ve kullanım kolaylığı arayan kullanıcılar için iyi bir seçim haline getirmektedir.

Son karşılaştırma, dillerin sözdizimi yapılarına odaklanmaktadır.

Tablo 39, yaygın aritmetik ve matris işlemleri için geliştirilen dil ile diğer programlama ortamlarının temel sözdizimlerini karşılaştırmaktadır. ALGLIB, sözdizimi kullandığı dile göre değiştiği için tabloya dahil edilmemiştir. Dikkat çeken bir nokta, geliştirilen dilin matrislerde satır ana sıralamasını kullanmasıdır. Bu, MATLAB, Breeze ve R gibi sütun ana sıralamasını kullanan diğer ortamlardan farklıdır. Bununla birlikte MxPL, Numpy ve Breeze gibi 0 tabanlı indekslemeyi benimserken, MATLAB ve R 1 tabanlı indeksleme kullanmaktadır.

Tablo 39. Popüler sayısal ortamlarda ve MxPL'de bazı operatörler ve işlemleri

İşlem	**Breeze**	**Matlab**	**Numpy**	**R**	**MxPL**
Temel İndeksleme	a(0,1)	a(1,2)	a[0,1]	a[1L,2L]	A[0,1]
Matris Sütununu Çıkarma	a(::, 2)	a(:,3)	a[:, 2]	a[,2]	A[:,2]
Eleman Bazında Toplama	a + b	a + b	a + b	a + b	A+B
Matris Çarpımı	a * b	a * b	dot(a, b)	a %*% b	A*B
Matris Boyutu	a.rows, a.cols	size(a)	a.shape	dim(x)	size(a)
Dikey Birleştirme	DenseMatrix.vertcat(a,b)	[a ; b]	vstack((a,b))	rbind(a, b)	append(a,b)
Yatay Birleştirme	DenseMatrix.horzcat(d,e)	[d , e]	hstack((d,e))	cbind(d, e)	merge(a,b)

Tablodaki “a, b, A ve B” matrisleri temsil etmektedir. “L” eki, R dilinde tam sayı değerlerini belirtmek için kullanılmaktadır.

4. SONUÇLAR

Bu tez çalışması, sayısal çözümleme yöntemleri ve cebirsel hesaplamaları destekleyen yüksek seviyeli bir programlama dili tanıtmaktadır. Matrisler, tamsayılar, çiftler ve özellikle rasyonel sayılar gibi çeşitli veri türleri üzerinde işlem yapabilen bu dil, karmaşık algoritmaları matematiksel gösterimlere benzer bir sözdizimiyle ifade etmeyi kolaylaştırır. Böylece, sayısal çözümleme problemlerini anlamak ve çözmek için etkili bir yol sunar. Dil ayrıca, tür çıkarımı, desen eşleştirme ve tembel değerlendirme gibi işlevsel özellikleri de desteklemektedir. Çalışmada, geliştirilen dilin tasarım ve uygulama süreçleri detaylı bir şekilde ele alınmıştır. Bu kapsamda, biçimsel dilbilgisinin tasarımı, sözcük üreteci ve ayrıştırıcının oluşturulması, sözdizimi ağacı üretimi, tür doğrulama ve kod değerlendirme gibi aşamalar, kod örnekleri eşliğinde açıklanmıştır. Ayrıştırma ve sözdizimi ağacı oluşturma aşamalarında JavaCC aracından yararlanılmıştır. JavaCC kullanılarak LL(1) dil bilgisi kurallarının tanımlanması, jeton listesinin oluşturulması ve dilbilgisi metotlarına aksiyonlar eklenerek ayrıştırma sırasında AST' nin oluşturulması gibi işlemler, örneklerle birlikte ayrıntılı olarak açıklanmıştır. Anlambilimsel analiz ve değerlendirme aşamaları ise Java dilinde ziyaretçi tasarım deseni kullanılarak gerçekleştirilmiş ve bu aşamalar da örnek kod parçaları ve açıklamalar ile sunulmuştur.

MxPL'in sözdizimi ve anlambiliminin daha iyi anlaşılmasını sağlamak amacıyla, Fibonacci dizisi hesaplaması ile matris transpozu ve determinantı gibi temel matris işlemlerini gerçekleştiren örnek programlar sunulmuştur. Ayrıca, geliştirilen dilin belirli dil özellikleri ve performans metrikleri, Java ve Python gibi yaygın kullanılan programlama dilleri ile karşılaştırmalı olarak analiz edilmiştir

Programlama dilleri geliştikçe, fonksiyonel ve emirsel yaklaşımların en iyi özelliklerini birleştirmek, etkili ve kullanışlı diller yaratma yolunda önemli bir hedef haline gelmiştir. Bu çalışma, her iki yaklaşımın avantajlarını tek bir çatı altında birleştiren hibrit bir dil yapısı tasarlayarak ve uygulayarak, bu yapıları sorunsuz bir şekilde entegre eden hibrit bir dilbilgisi ve semantik geliştirmeyi hedeflemektedir. Tez çalışması kapsamında geliştirilen örnek uygulamalar, bu entegrasyonun işlerliğini ve verimliliğini ortaya koymaktadır.

Geliştirilen dilin bir diğer önemli özelliği ise, özel bir matris veri yapısı sunarak ve ilgili işlemleri kolaylaştırarak matris manipülasyonunu basitleştirmesidir. Programlama

dillerinde, karmaşık matris hesaplamaları genellikle üst düzey kütüphane rutinlerine bırakılırken, temel işlemler bile okunması ve bakımı zor, uzun kod blokları veya tamamen hazır fonksiyonlar gerektirebilir. Bu dil ise, matrislerle çalışmayı daha pratik ve anlaşılır hale getirerek geliştiricilerin program üzerindeki hakimiyetini artırır ve böylece kısa ve sade ifadelerle çok daha fazla iş yapılmasına imkân tanır. Ek olarak, dilimiz, çoğu dilde sadece kütüphanelerle kısıtlı bir şekilde sunulan rasyonel sayılarla doğrudan işlem yapma yeteneğine sahiptir. Bu özellik, rasyonel sayıdan ondalık sayıya dönüşümdeki hata paylarını ortadan kaldırarak daha hassas ve matematiksel olarak doğru bir hesaplama ortamı sunar.

Bu çalışmanın temel amacı matematiksel sözdizimine benzer ve kolay öğrenilebilen basit bir dil geliştirmektir. Performans optimizasyonları çalışmanın kapsamı dışında tutulmuştur. Bu nedenle dilin performans analizi ve iyileştirmeleri başka bir çalışmanın konusu olabilir. Ayrıca dilin matematik eğitiminde yararlı bir araç olabileceği göz önüne alındığında bu konunun eğitim bilimleri alanında da araştırılması düşünülebilir.

5. ÖNERİLER

Doktora tezi olarak sunulan bu çalışmada, sayısal hesaplamalara yönelik kullanıcı dostu ve matematiksel notasyona benzer MxPL olarak adlandırılan bir programlama dili geliştirilmiştir. Bu tez konusu ile çalışma yapmak isteyecek araştırmacılara ve uygulayıcılara iletilmek istenen öneriler aşağıdaki gibi belirtilmiştir:

- Geliştirilen dil, doğrusal cebir ve matematiğin diğer konuları için temel işlemleri gerçekleştiren çeşitli temel işlevleri destekleyecek şekilde genişletilebilmeye açıktır. Bu tür işlevler özellikle bilimsel problemlere dayalı matematiksel modellerin operasyonel gereksinimlerini karşılamaya hizmet edecektir.
- Metin dosyalarından okuma veya metin dosyalarına yazma gibi ortak bir G/Ç işlemleri kümesi oluşturmak da önemlidir.
- Belirli bir fonksiyonun köklerinin sabit nokta yinelemeleri yoluyla hesaplanmasında olduğu gibi, matematiksel fonksiyonların düzenlenmesini farklı şekillerde mümkün kılan dil yapılarına da ihtiyaç vardır. Bu gereksinimler, dil sözdizimine yeni veri türleri ve anahtar kelimeler eklenerek veya bir dil kütüphanesi geliştirilerek yerine getirilebilir.
- Dilin mevcut çalışması, tür kontrolü ve kapsam çözümü de dahil olmak üzere semantik analizin temel yönlerine odaklanmaktadır. Canlılık veya mevcut ifade analizi (Nielson vd., 1999) gibi daha gelişmiş anlamsal analiz teknikleri, dilin güvenliğini, güvenilirliğini ve doğruluğunu arttırmada ve desteklemede kritik bir rol oynayacaktır. Bu teknikler programın anlaşılmasını daha da derinlemesine inceleyerek daha zengin anlamı ortaya çıkaracak ve güçlü büyümeyi teşvik edecektir. Bu nedenle gelecekteki araştırmalar için önemli bir yön, daha gelişmiş semantik analiz özelliklerinin dahil edilmesidir.
- Gelişmiş semantik analiz tekniklerini WALA (URL-3, 2023), LiSA (Negrini, 2023) ve Julia (Bezanson, 2017) gibi statik analiz araçlarıyla birleştirmek, olası sorunlu yazılım davranışlarına dair içgörü sağlamada yardımcı olabilir. Geliştirilen Dil ekosisteminde bu kombinasyon stratejisi özellikle büyük ölçekli kod tabanları için yararlı olabilir.

6. KAYNAKLAR

Abadi, M. (2016). TensorFlow: learning functions at scale. *In Proceedings of the 21st ACM SIGPLAN international conference on functional programming*, (pp. 1-1).

Armstrong, J. (1996). Erlang—a Survey of the Language and its Industrial Applications. *In Proc. INAP, Vol. 96*, pp. 16-18).

Arnold, K., Gosling, J. & Holmes, D. (2005). *The Java Programming Language*. Addison Wesley Professional.

Augustsson, L., Schwarz, J. & Nikhil, R.S. (2001). *Bluespec Language Definition*. Sandburst Corparation.

Balman, T. (1981). Computer assisted teaching of FORTRAN. *Computers & Education, 5*(2), 111-123.

Balsa, C., Alves, L., Pereira, M. J., Rodrigues, P. J., & Lopes, R. P. (2012). Graphical simulation of numerical algorithms - An approach based on code instrumentation and Java Technologies. *Proceedings of the 7th International Conference on Computer Science & Education [CSEDU]*, Porto, Portugal.

Basu, S. (2009). *FreeMat v4. 0 Documentation*.

Benaddy, M., & El Habil, B. (2016, September). A programming language for matrices operations and LATEX code generation. *In Proceedings of the 2016 International Conference on Engineering & MIS (ICEMIS)*, (pp. 1-4). Agadir, Morocco.

Berners-Lee, T., Cailliau, R., Pellow, N., & Secret, A. (2023, Aralık 19). The World Wide Web Initiative, CERN http://info.cern.ch/hypertext/WWW/TheProject.html adresinden alınmıştır.

Bezanson, J., Edelman, A., Karpinski, S., & Shah, V. B. (2017). Julia: A fresh approach to numerical computing. *SIAM review, 59*(1), 65-98.

Bisschop, J., & Meeraus, A. (1982). *On the development of a general algebraic modeling system in a strategic planning environment*. Springer Berlin Heidelberg.

Bisschop, J., & Roelofs, M. (1999). *AIMMS: The Language Reference*. Haarlem, The Netherlands, Paragon Decision Technology B.V.

Biswa, K., Jamatia, B., Choudhury, D., & Borah, P. (2016). Comparative analysis of C, Fortran, C# and Java programming languages. I*nternational Journal of Computer Science and Information Technologies, 7*(3), 1004–1007.

Blain, J.M., (2020). *The Complete Guide to Blender Graphics: Computer Modeling & Animation, 6th ed.*, Boca Raton, FL, USA, CRC Press.

Bosma,W., & Cannon, J.J. (1996). *Handbook of Magma Functions*. Sydney, Australia, The University of Sydney.

Brewster, M. W., & Gobbert, M. K. (2011). *A comparative evaluation of Matlab, Octave, FreeMat, and Scilab on tara*. UMBC Faculty Collection.

Brijder, R., Geerts, F., Van den Bussche, J., & Weerwag, T. (2019). MATLANG: Matrix operations and their expressive power. *ACM Sigmod Record, 48*, 60–67.

Burden, R. L., & Faires, J. D. (2011). *Numerical analysis, 9th ed.*. Brooks/Cole Cengage Learning.

Cao, H., Tang, S., Zhu, Q., Yu, B., & Chen, W. (2023). Mat2Stencil: A modular matrix-based DSL for explicit and implicit matrix-free PDE solvers on structured grid. *Proceedings of the ACM on Programming Languages, 7*, 686–715.

Carroll, J. (1992). The role of computer software in numerical analysis teaching. *ACM SIGNUM Newsletter, 27*(2), 2–31.

Cox, L. A., Paige, K. N., & Popken, D. (1999). Software review of Analytica 1.2. *Human and Ecological Risk Assessment An International Journal, 5*(2), 305-316.

DeMichiel, L. G., & Gabriel, R. P. (1987). The common lisp object system: An overview. *In European Conference on Object-Oriented Programming*, pp. 151-170, Berlin, Heidelberg, Springer Berlin Heidelberg.

Doornik, J. (2001). *A Object-oriented matrix programming using Ox, 4th ed.*. London, UK, Timberlake Consultants Press.

Doornik, J. (2009). *An Object-Oriented Matrix Programming Language Ox 6.* London, UK, Timberlake Consultants Press.

Dos Reis, A. J. (2012). *Compiler Construction Using Java, JavaCC, and Yacc*. John Wiley & Sons.

Enterprises, S. (2012). *Scilab: Free and Open Source software for numerical computation.* Scilab Enterprises, Orsay, France.

Fausett, L. V. (1999). *Applied numerical analysis using MATLAB.* Prentice Hall.

Fedrecheski, G., Costa, L. C., & Zuffo, M. K. (2016). Elixir programming language evaluation for IoT. *In 2016 IEEE International Symposium on Consumer Electronics (ISCE)*, pp. 105-106, IEEE.

Feuerstein, S. & Pribyl, B. (2005). *Oracle PL/SQL Programming*. Newton, MA, USA , O'Reilly Media, Inc..

Fojtik, R. (2020). *S*wift a new programming language for development and education. *In Digital Science*, (pp. 284-295), Springer International Publishing.

Fourer, R., & Gay, D. M. (2002). Extending an algebraic modeling language to support constraint programming. *INFORMS Journal on Computing, 14*(4), 322-344.

Gagnon, E. M., & Hendren, L. J., (1998). SableCC, An Object-Oriented Compiler Framework. *Technology of Object-Oriented Language and System, 26*, 140-154.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns: Elements of reusable object-oriented software*. Boston, Addison-Wesley Longman Publishing Co., Inc.

Gilat, A., & Subramaniam, V. (2014). *Numerical methods for engineers and scientists, 3rd ed.*. Wiley.

Giner-Miguelez, J., Gómez, A., & Cabot, J. (2023). A domain-specific language for describing machine learning datasets. *Journal of Computer Languages, 76*, 101209.

Harper, R. (2016). *Practical Principles for Programming Languages, 2nd ed.*. New York, NY, USA, Cambridge University Press.

Harris, C. R., Millman, K. J., Van DerWalt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., … Oliphant, T. E. (2020). Array programming with NumPy. *Nature, 585*(7825), 357–362.

Hassan, A. B., Abolarin, M. S., & Jimoh, O. H. (2006). The application of Visual Basic programming language to simulate numerical iterations. *Leonardo Journal of Sciences, 9*, 125–136.

Haverbeke, M. (2018). *Eloquent javascript: A modern introduction to programming.* San Francisco, California, USA, No Starch Press.

Heipcke, S. (2002). *Applications of Optimization with Xpress-MP*. Blisworth, UK, Dash Optimization.

Hejlsberg, A., Torgersen, M., Wiltamuth, S., & Golde, P. (2010). *C# Programming language*. Addison-Wesley Professional.

Henry, R., Hsu, O., Yadav, R., Chou, S., Olukotun, K., Amarasinghe, S., & Kjolstad, F. (2021). Compilation of sparse array programming models. *Proceedings of the ACM on Programming Languages, 5*, 1–29.

Higham, D. J., & Higham, N. J. (2017). *The MATLAB guide, 3rd ed.*. SIAM.

Hoffmann, J. D. (1993). *Numerical methods for engineers and scientists.* McGraw-Hill.

Hürlimann, T., (2007). *Index Notation in mathematics and Modelling Language LPL: Theory and Exercices.* Department of Informatics University of Fribourg.

IBM Security QRadar, (2017). *Ariel Query Language (AQL) Guide v7.3.1*. New York, USA, IBM Corp..

Imambi, S., Prakash, K. B., & Kanagachidambaresan, G. R. (2021). PyTorch. *Programming with TensorFlow: solution for edge computing applications*, 87-104.

James, M.L., Smith, G.M., Wolford, J.C. (1993). *Applied Numerical Methods for Digital Computation*, Harper Collins.

Johnson, S., (1999). Query-by-example (QBE). *In Database Management Systems*. New York, NY, USA, McGraw-Hill Publisher.

Jones, S. P. (2003). *Haskell 98 language and libraries: The revised report*. Cambridge University Press.

Kallrath, J. (2004). *Modeling Languages in Mathematical Optimization*. Boston, MA, USA, Springer.

Kaukic, M. I. C. H. A. L. (2005). Open source software resources for numerical analysis teaching. *International Journal for Mathematics Teaching and Learning*, 1473-0111.

Kernighan, B. W., & Ritchie, D. M. (1988). *The C programming language*. Prentice Hall PTR.

Kochan, S. G. (2011). *Programming in objective-C.* Addison-Wesley Professional.

Koga, M., & Furuta, K. (1992, March). MaTX: A high-performance programming language (interpreter and compiler) for scientific and engineering computation. I*n Proceedings of the IEEE Symposium on Computer-Aided Control System Design*, 15-22, Napa, CA, USA.

Kokash, N., Moodie, S. L., Smith, M. K., & Holford, N. (2015). Implementing a domain-specific language for model-based drug development. *Procedia Computer Science, 63*, 308–316.

Lamport, L. (1994). *LateX: A Document Preparation System. 2nd ed.*, Boston, MA, USA, Addison- Wesley.

Lopes, L., & Martins, L. F. (2016). A safe-by-design programming language for wireless sensor networks. J*ournal of Systems Architecture, 63*, 16–32.

Louden, L.K.C., & Lambert, K.A. (2011). *Programming Languages: Principles and Practices, 3rd ed.*. Boston, MA, USA, Cengage Learning.

Louise, S. (2017). An OpenMP backend for the ΣC streaming language. *Procedia Computer Science, 108*, 1073-1082.

Matsumoto, Y., & Ishituka, K. (2002). *Ruby programming language*. Addison Wesley Publishing Company.

McKinley, M., (2006). *The Game Animator's Guide to Maya*. Sybex.

Müller, W., Rosenstiel, W. & Ruf, J. (Eds.) (2007). *SystemC: Methodologies and Applications.* Berlin/Heidelberg, Germany, Springer Science & Business.

Nagar, S., & Nagar, S. (2018). *Introduction to Octave*. Apress.

Negrini, L.; Ferrara, P., Arceri, V., & Cortesi, A. (2023). LiSA: A generic framework for multilanguage static analysis. *In Challenges of Software Verification*, Springer Nature, Singapore, 19–42.

Nielson, F., Nielson, H. R., & Hankin, C. (1999). *Principles of program analysis, 1st ed.*. Berlin/Heidelberg, Germany, Springer.

Oliveira, N., Pereira, M. J., Henriques, P. R., & Cruz, D. (2009). Domain specific languages: A theoretical survey. *INForum'09-Simpósio de Informática.*

Oppel, A. (2015). *SQL: A Beginner's Guide. 4th ed.* New York, USA, McGraw-Hill Education.

Parr, T. J., & Quong, R. W. (1995). ANTLR: A predicated-LL(k) parser generator. *Software Practice and Experience, 25*(7), 789–810.

Perry, D.L. (2002). *VHDL: Programming by Example, 4th ed.*. New York, NY, USA, McGraw-Hill Education.

Pinho, E. G., & de Carvalho Junior, F. H. (2014). An object-oriented parallel programming language for distributed-memory parallel computing platforms. *Science of Computer Programming, 80*, 65-90.

Reinhart, C., & Breton, P.F. (2009). Experimental validation of Autodesk 3ds Max® Design 2009 and DAYSIM 3.0. *Leukos 6*, 7–35.

Richter, W.T. (2014). TEX and Scripting Languages. *In TUGboat, Proceedings of the Practical TEX 2014 Conference*, 25, 71–88. San Francisco, CA, USA.

Sadat-Mohtasham, S. H., & Ghorbani, A. A. (2008). A language for high-level description of adaptive web systems. *Journal of Systems and Software, 81*(7), 1196-1217.

Sammet, J. E. (1985). Brief Summary of the Early History of cobol. *Annals of the History of Computing, 7*(4), 288-303.

Sastry, S. S. (2012). *Introductory methods of numerical analysis*. PHI Learning Pvt. Ltd..

Schrage, L. E. (2006). *Optimization modeling with LINGO*. Chicago, Il, USA, Boston, MA, USA.

Seabold, S., & Perktold, J. (2010). Statsmodels: econometric and statistical modeling with python. *SciPy*, *7*(1).

Sebesta R. W. (2016). *Concepts of programming languages, 11th ed.*. India, Pearson.

Shearer, J. M., & Wolfe, M. A. (1985). ALGLIB, a simple symbol-manipulation package. *Communications of the ACM, 28*(8), 820-825.

Snider, L. A., & Swedo, S. E. (2004). PANDAS: current status and directions for research. *Molecular psychiatry, 9*(10), 900-907.

Sterling, L., & Shapiro, E. Y. (1994). *The art of Prolog: Advanced programming techniques, 2nd ed.*. The MIT Press.

Stroustrup, B. (1986). An overview of C++. I*n Proceedings of the 1986 SIGPLAN Workshop on Object-oriented Programming*, 7–18, New York, USA.

Syme, D., Petricek, T., & Lomov, D. (2011). The F# asynchronous programming model. *In International Symposium on Practical Aspects of Declarative Languages,* 175-189, Berlin, Heidelberg, Springer Berlin Heidelberg.

Tatroe, K., & MacIntyre, P. (2020). *Programming PHP: Creating dynamic web pages*. Sebastopol, California, USA, O'Reilly Media.

Tavakkoli, A., (2018). *Game Development and Simulation with Unreal Technology, 2nd ed.*. London, UK, Routledge.

Thain, D. (2020). *Introduction to Compilers and Language Design: Second Edition*. University of Notre Dame.

Thomas, D.E. & Moorby, P.R. (2002). *The Verilog Hardware Description Language, 5th ed.*. New York, NY, USA, Springer.

Trott, M. (2014). *The Mathematica Guidebook for Programming*. New York, USA, Springer-Verlag.

URL-1, (2023, 10 Aralık). Breeze, https://github.com/scalanlp/breeze/wiki/Quickstart adresinden alınmıştır.

URL-2. (2021, Ağustos 11). Software for Mathematics, Online Learning, Engineering. (n.d.-b). Maple. https://www.maplesoft.com/ adresinden alınmıştır.

URL-3. (2023, Aralık 10). WALA. https://github.com/wala/WALA adresinden alınmıştır.

Van Rossum, G., & Drake Jr, F. L. (1995). *Python tutorial*. Centrum voor Wiskunde en Informatica Amsterdam, The Netherlands.

Vasić, M., Soloveichik, D., & Khurshid, S. (2020). CRN++: Molecular programming language. *Natural Computing, 19*, 391–407.

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., ... & Van Mulbregt, P. (2020). SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature methods, 17*(3), 261-272.

Wakerly, J. (1979). The programming language Pascal. *Microprocessors and Microsystems, 3*(7), 321–326.

Wall, L., Christiansen, T., & Orwant, J. (2000). *Programming Perl, 3rd ed.*, O'Reilly Media.

Wampler, D. (2021). *Programming Scala.* O'Reilly Media, Inc..

Wlodkowski, P. (2006). Teaching numerical methods in engineering with Mathcad. *American Society for Engineering Education, no.2006-1549*.

ÖZGEÇMİŞ

Mehmet Cemil AYDOĞDU, İlkokulu Mareşal Fevzi Çakmak İlk Okulu'nda, Ortaokul ve Liseyi ise Kanuni Anadolu Lisesi'nde tamamlamıştır. Anadolu Üniversitesi Bilgisayar Mühendisliği (İngilizce) Bölümü'nden 2010 yılında mezun olmuştur. 2013-2014 eğitim–öğretim yılının güz döneminde Karadeniz Teknik Üniversitesi Fen Bilimleri Enstitüsü Bilgisayar Mühendisliği Anabilim Dalı'nda yüksek lisans programına, bahar döneminde de aynı bölümde araştırma görevlisi olarak çalışmaya başlamıştır. 2016 yılında "Belirsiz Limit İfadelerinin Otomatik Üretimi ve Adım Adım Çözümü" adlı yüksek lisans tezini sunarak yüksek lisans eğitimi sonlandırmıştır. 2016 yılında Karadeniz Teknik Üniversitesi Fen Bilimleri Enstitüsü Bilgisayar Mühendisliği Anabilim Dalı'nda doktora eğitimine başlamıştır. Bilgisayar Mühendisliği Bölümü'nde Araştırma Görevlisi olarak çalışmaya devam etmektedir. İyi derecede İngilizce bilmektedir.

Aydoğdu, M. C., Aydoğdu, Ö., & Pehlivan, H. (2024). MxPL: A Programming Language for Matrix-Related Operations. *Symmetry, 16*(2), 181.