KARADENİZ TEKNİK ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI

BİR WEB-TABANLI MİNİHASKELL YORUMLAYICISI

YÜKSEK LİSANS TEZİ

Arda ÜSTÜBİOĞLU

ŞUBAT 2009
TRABZON

KARADENİZ TEKNİK ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI

BİR WEB-TABANLI MİNİHASKELL YORUMLAYICISI

Bilgisayar Mühendisi Arda ÜSTÜBİOĞLU

Karadeniz Teknik Üniversitesi Fen Bilimleri Enstitüsünce
"Bilgisayar Yüksek Mühendisi"
Unvanı Verilmesi İçin Kabul Edilen Tezdir.

Tezin Enstitüye Verildiği Tarih : 20.01.2009
Tezin Savunma Tarihi : 04.02.2009

Tez Danışmanı : Yrd. Doç. Dr. Hüseyin PEHLİVAN
Jüri Üyesi : Prof. Dr. Vasıf V. NABİYEV
Jüri Üyesi : Yrd. Doç. Dr. H. İbrahim OKUMUŞ

Enstitü Müdürü : Prof. Dr. Salih TERZİOĞLU

Trabzon 2009

ÖNSÖZ

Çeviriciler, teknolojinin gelişmesi ile büyük şirketlerin ilgi odağı olmuştur. Günümüzde web programcılığından veritabanı programcılığına, sistem programcılığından ağ programcılığına kadar birçok alanda derleyiciler ve yorumlayıcılar kullanılmaktadır. Bilgisayarın temelini oluşturan yazılım bu bileşenler olmadan asla düşünülemez. Dolayısıyla profesyonel anlamda bilgisayar ile ilgisi olan kişiler bu yapılarla yakından ilgilenmektedir.

Literatür taramalarında ve YÖK'ün Ulusal Tez Merkezi'nde yapılan araştırmalarda bu konuya yönelik çalışmaların az olması konunun önemini daha da artırmaktadır.

Geçmişten günümüze kadar geçen sürece bakıldığında, ilk zamanlarda makine kodları ile yazılan programlar teknolojinin gelişmesi ve araştırmacıların çalışmaları sayesinde günümüzde programlama dilleri ile tanımladığımız yapılar tarafından gerçekleştirilmektedir. Böylece çok zor olan assembly ile kod yazımı günümüzde yüksek seviyeli programlama dilleri ile daha etkin şekilde yapılabilmektedir.

Çalışmamda fonksiyonel programlama dili olan Haskell örnek alınıp, az olmasıyla birlikte literatürde bulunan çalışmalar incelenip bir yorumlayıcı gerçekleştirilmiştir. Çalışmanın daha kolay erişilebilir olması için web tabanlı olması öngörülmüştür. Sonuç olarak, ortaya konulan yaklaşımlarla oldukça yararlı bir çalışma gerçeklenmiştir.

Bu çalışmada danışmanlığımı üstlenen değerli hocam Yrd. Doç. Dr. Hüseyin PEHLİVAN'a ilgi, alaka ve yardımlarından dolayı, değerli hocam Prof. Dr. Vasif Vagifoğlu NABİYEV'e desteğinden dolayı teşekkürü bir borç bilirim. Ayrıca her zaman yanımda olan aileme ve dostlarıma da destekleri için teşekkür ederim.

Arda ÜSTÜBİOĞLU
Trabzon 2009

İÇİNDEKİLER

Sayfa No

ÖZGEÇMİŞ

ÖZET

Günümüzde programlama dillerinin derleme süreçlerini otomatikleştiren birçok araç bulunmaktadır. Bu araçlarla birlikte derleyici aşamaları olan kaynak kodun analizi, ara koda dönüşüm, optimizasyonu ve hedef koda dönüşüm işlemleri geçmişe oranla daha kolay ve etkin şekilde yapılabilmektedir.

Bu çalışmada web tabanlı bir yorumlayıcının gerçekleştirilmesi hedeflenmiştir. Yorumlama işlemleri, fonksiyonel bir programlama dili olan Haskell'in bir alt seti seçilerek tanımlanan MiniHaskell dili üzerinde yapılmıştır.

Yorumlayıcılar Kelimesel Analiz, Sözdizimsel Analiz, Anlamsal Analiz ve Yorumlama aşamalarından oluşmaktadır. Her bir aşama için gereken kodlar MiniHaskell dilinin gramer kurallarına bağlı olarak Java programlama dilinde kod üretebilen JavaCC ve JTB gibi araçlar yardımıyla yazılmıştır.

Yapılan çalışmada otomatik olarak üretilen kodlar ve bunlara dayalı geliştirilen kodlar Java applet programına entegre edilerek online olarak çalışabilir duruma getirilmiştir.

Anahtar Kelimeler: Yorumlayıcı, Haskell, JavaCC, JTB, Derleyici Tasarımı

SUMMARY

A Web-Based MiniHaskell Interpreter

Recently, there have been developed many tools which automates the process of compiler construction for programming languages. Compared to the past, it is easier and more efficient to carry out the analysis of the source code, the transformation of the intermadiate code, the optimization and the transformation of the target code using these tools.

In this work, it is aimed to implement a web based interpreter. All the interpreting operation are performed by defining a programming language called MiniHaskell a subset of the Haskell which is a functional programming language.

Interpreters consist of the phases of Lexical Analysis, Syntactical Analysis, Semantic Analysis and Interpretation. The main parts of the code required for each phase are written by means of tools such as JavaCC and JTB which can generate code for Java language, depending upon the grammar rules of the MiniHaskell language.

The codes produced automatically and manually in the work are integrated into a Java applet program so that it can run online.

Key Words: Interpreter, Haskell, JavaCC, JTB, Compiler Design

ŞEKİLLER DİZİNİ

Sayfa No

TABLOLAR DİZİNİ

Sayfa No

SEMBOLLER DİZİNİ

DFA	Belirli Sonlu Otomata (Deterministic Finite Automata)
NFA	Belirsiz Sonlu Otomata (Non- Deterministic Finite Automata)
CFG	Bağlam Bağımsız Gramer (Context – Free Grammer)
AST	Soyut Sözdizim Ağacı (Abstract Syntax Tree)
LL(k)	Soldan sağa ayrıştırma – Sola dayalı türetim – k kelime kontrolü (Left-to-Right Parsing – Leftmost Derivation – k lookahead)
LR(k)	Soldan sağa ayrıştırma – Sağa dayalı türetim – k kelime kontrolü (Left-to-Right Parsing – Rightmost Derivation – k lookahead)
JTB	Java Ağaç Oluşturucu (Java Tree Builder)

1. GENEL BİLGİLER

1.1. Giriş

Çevirici, herhangi bir programlama dilinde yazılmış olan kaynak kodu, başka bir programlama diline ait hedef koda dönüştüren programdır. Günümüzde çevirici programlara yazılım disiplinin hemen her alanından örnek vermek mümkündür. Tablo 1.1.'de çevirici programlara ilişkin kaynak kod ve hedef kod örnekleri bulunmaktadır.

Tablo 1.1. Örnek Çeviriciler

Kaynak Kod	**Çevirici Program**	**Hedef Kod**
LaTeX	Text Formatter	PostScript
SQL	Database Query Optimizer	Query Evaluation Plan
Java	Javac Compiler	Java Byte Code
Java	Cross Compiler	C++ Code
Düzenli İfadeler	JLex Scanner Generator	Java

Kaynak kodun yüksek ya da orta seviyeli bir programlama dili, hedef kodun ise makine kodu olması durumunda çevirici programa derleyici adı verilir. Derleyiciler yüksek seviyeli programlama dillerinin ortaya çıkmasıyla bilgisayar dünyasının vazgeçilmezleri arasına girmiştir. Yüksek seviyeli bir dilden makine diline uzanan süreç içinde kaynak kodu meydana getiren veriyi çeşitli dönüşümlerden geçiren derleyiciler, çok karmaşık programlama gerektirirler [1,2].

Günümüzde derleyiciler için ihtiyaç duyulan programlama yükünün önemli bir bölümünü üstlenen ve otomatik olarak kod üretebilen araçlar geliştirilmiştir. Unix sistemlerinin ortaya çıktığı 1970'li yıllardan itibaren lex [22,24] ve yacc [23,24] gibi araçlar bu alandaki gelişmelere öncülük etmiştir.

Derleyici derleyici olarak adlandırılan bu araçlar ile yazılmış birçok derleyici ve yorumlayıcı bulunmaktadır. Bu araçlar özellikle üniversite düzeyindeki öğretim faaliyetlerinde kullanılmak üzere birçok programlama dilinin tasarımına ve gerçeklenmesine katkı sağlamışlardır. Gerçeklenen programlama dillerine örnek olarak Fang, Triangle [7] ve MiniJava [5] dilleri verilebilir. Yorumlayıcılar da çevirici olarak nitelendirilirler. Fakat yorumlayıcılar, kaynak kodu parça parça icra eder ve kodun icrasından sonra yeni bir dosya oluşturmaz [16]. Yorumlayıcılarla çalıştırılan bazı programlama dilleri şunlardır: Basic, Pascal, APL, Haskell, Lisp, Cobol, Asp, Lava…

1.2. Derleyici ve Yorumlayıcının Karşılaştırılması

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

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

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

- Koşma anında yorumlayıcının belleğe yüklenmesi, kaynak programa daha az bellek alanı tahsis edilmesine yol açar. Derleyicinin ise sadece derleme anında belleğe yüklenip sonrasında yüklenmeye ihtiyaç duymaması kaynak programa daha fazla bellek alanı tahsis edilmesini sağlar.
- Derlenmiş kaynak kodu, derleyiciye ihtiyaç duymadan tekrar koşturulabilirken, yorumlanan kaynak kod tekrar çalıştırılmak istendiğinde yorumlayıcı programa gereksinim duyar.
- Makine koduna çevrilmiş programlar, yorumlanarak çalıştırılan programlara göre daha hızlı çalışırlar.
- Yorumlamalı programların kodunda, derlenmiş programlara göre daha kolay ve hızlı şekilde değişiklik yapılabilir.

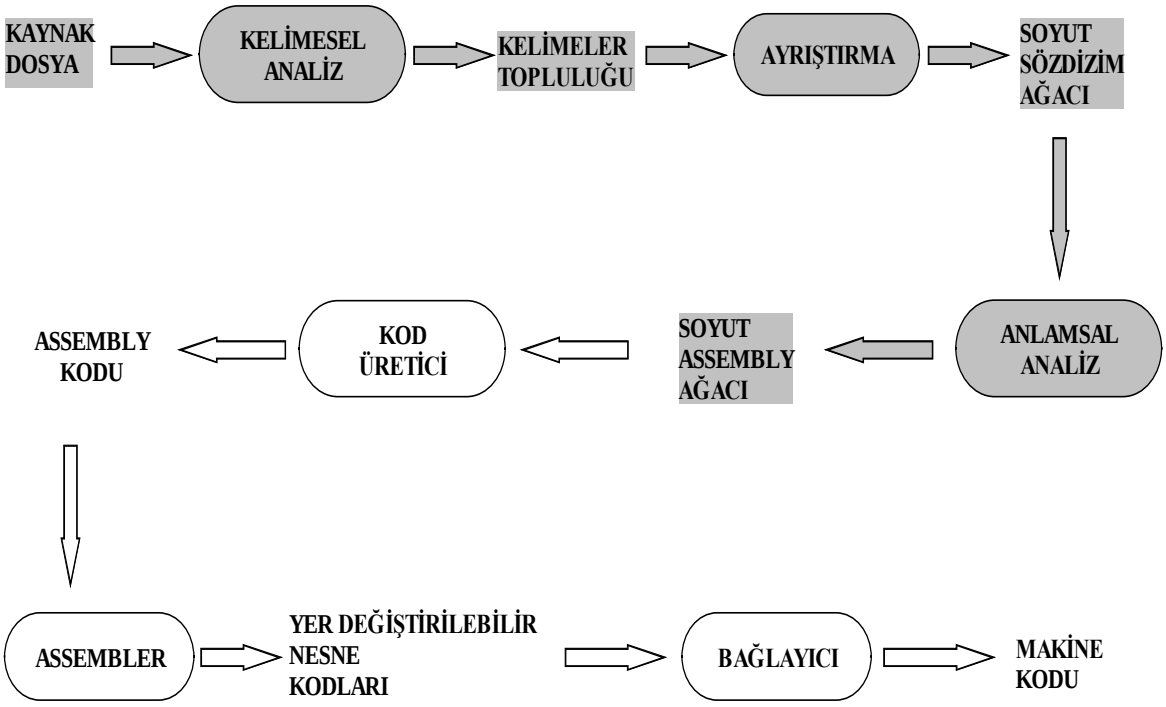
1.3. Derleyici Aşamaları

Bir derleyicinin tasarım süreci üç temel aşamadan oluşur;

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

Şekil 1.1.'de temel derleyici aşamaları daha ayrıntılı olarak gösterilmiştir. Koyu renkli kutular yorumlayıcılar için de gereken aşamaları temsil etmektedir.

Derleyici tasarımının bu kadar ayrık safhalara bölünmesinin nedeni her aşama sonucunda elde edilen veri yapılarının farklı platformlar için kod üretimi gibi çeşitli amaçlara yönelik olarak kullanılmasıdır.



Şekil 1.1. Derleyici Aşamaları

Bir programlama dilindeki kaynak kodun başka bir programlama diline çevrilebilmesi için derleyicinin ilk aşamada, kod yapısını ve içeriğini kontrol etmesi gerekmektedir. Analiz aşaması olarak bilinen bu aşamaya derleyicinin önyüzü (frontend) adı verilir. Analiz aşamasından sonra parçalanmış halde olan kodun yeniden birleştirilip,

icrasının gerçekleştirilmesi gerekmektedir. Sentez işleminin yapıldığı bu aşamaya ise derleyicinin arka yüzü (backend) adı verilir.

Şekil 1.1.'de koyu renkli kutular derleyicinin önyüzünü (analiz işlemini), geri kalan kutular ise arka yüzünü (sentez işlemini) temsil etmektedir.

Yorumlayıcılar analiz aşamasını gerçekleştirdikten sonra derleyiciler gibi sentez aşamasına geçmezler. Bu aşamadan sonra analiz edilmiş kodun icrasını gerçekleştirirler.

Yorumlayıcılar için analiz süreci üç aşamadan oluşur:

- Kelimesel Analiz
- Sözdizimsel Analiz
- Anlamsal Analiz

1.3.1. Kelimesel Analiz

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

Kelimesel çözümleyiciler, aldıkları karakter dizisinin, isimler dizisi, anahtar kelimeler, noktalama işaretleri gibi, daha önceden tanımlanmış yapılara dönüşümünü gerçekleştirir. Bu aşamada belirtildiği takdirde, kaynak kodda bulunan açıklamalar, özel karakterler gibi derleme işlemine dahil edilmeyecek yapıları göz ardı etmek mümkündür.

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

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

1.3.1.1. Düzenli İfadeler

Düzenli ifadeler [20], karakter topluluğunun içinden belirli karakter kümelerinin seçilmesine olanak sağlayan bir mekanizmadır.

Belirli bir kelimeyi veya karakterler kümesini bir metin içerisinde arama işlemi oldukça basittir. Tüm metin düzenleyiciler bu işlemi kolayca gerçekleştirebilmektedir. Düzenli ifadeler ise daha güçlü ve esnek yapılardır. Bu yapı ile birlikte belirli uzunluktaki

kelimeler, belirtilen şartlara uygun kelimeler, sayılar ve noktalama işaretleri kolaylıkla aranabilir.

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

- Seçim : “ | ” işareti kullanılarak ifadeler seçeneklere ayrılır. Örneğin; (a | b) ifadesinde “a” ya da “b” ile eşleşme yapılır.
- Birleştirme : “ · ” işareti kullanılarak ifadeler birleştirilir. Örneğin; (a | b) · a ifadesinde “aa” yada “ba” ifadeleri ile eşleşme yapılır.
- Epsilon : “ ϵ ” işareti ile kullanılarak boş değer gösterimi sağlanır. Örneğin; (a | ϵ) ifadesinde “a” ya da boş değer eşleşmesi yapılır.
- Tekrarlamalar : Bir karakterin ya da grubun ardından gelerek öncesindeki yapının kaç kez tekrarlanacağını belirtir. En temel tekrarlamalar “ * ” , “ + ” ve “ ? ” dir.

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

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

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

Düzenli ifadelerdeki bu kavramlar kullanılarak programlama diline ait kelimesel yapılar tanımlanır. Programlama dilleri için genel olan bazı yapıların, düzenli ifade formatında gösterimleri aşağıdaki biçimdedir:

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

Bu kurallar çerçevesinde oluşturulacak yapıda bazı belirsizlikler meydana gelebilmektedir. Örneğin “ifa” yapısı düzenli ifade tarafından IF ve DEĞİŞKEN mi yoksa sadece DEĞİŞKEN olarak mı algılanması konusunda belirsizlik oluşmaktadır. Bu belirsizliği ortadan kaldırmak için kelimesel çözümleyiciler iki önemli kural geliştirmişlerdir.

- En uzun eşleşme: Bu kurala göre en uzun eşleşmeye sahip ifade öncelikli seçilir.

- Kural önceliği: Düzenli ifade tanımlama sırasına göre ilk karşılaşılan ifade öncelikli seçilir.

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

1.3.1.2. Sonlu Otomata

Kelimesel yapıların düzenli ifadeler aracılığıyla belirlenmesi uygun bir yöntemdir, ancak bilgisayar programı olarak oluşturulabilmesi için yeterli değildir. Bu sebeple sonlu otomata yapısına çevrilmesi gerekmektedir.

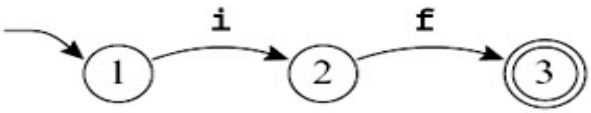
Sonlu Otomatalar, sonlu sayıda durumla birlikte karakterize edilmiş hesaplama modeli tanımlar. Bu yapı üç önemli kavram içermektedir [12].

- Sonlu durum kümesi,
- Mümkün harfleri içeren Alfabe ($\sum$),
- Her bir durumun, her bir harf için gideceği durumu belirleyen Sonlu Geçiş Kümesi.

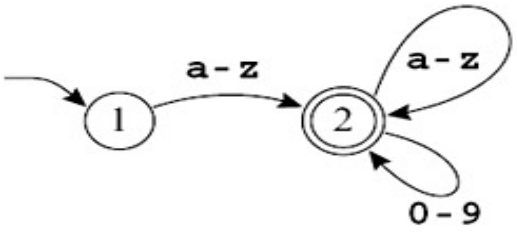
Sonlu Otomata; sonlu sayıda durum kümesine, bir durumdan diğerine geçişi sağlayan ve her biri bir sembol ile belirtilmiş kenarlara sahiptir. Durum kümesinde sadece bir tane başlangıç durumu, istenilen sayıda ise sonuç durumu bulunabilir.

Şekil 1.2.’de önceden düzenli ifade yardımıyla tanımlanmış yapıların, sonlu otomata ile gösterimi bulunmaktadır. Her bir yapı için durumlar ayrı ayrı numaralandırılmış ve başlangıç durumu “ 1 ” ile gösterilmiştir. Sonuç durumu ise iç içe iki çember ile belirtilmiştir.

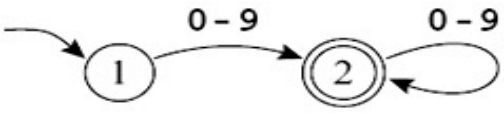
IF :



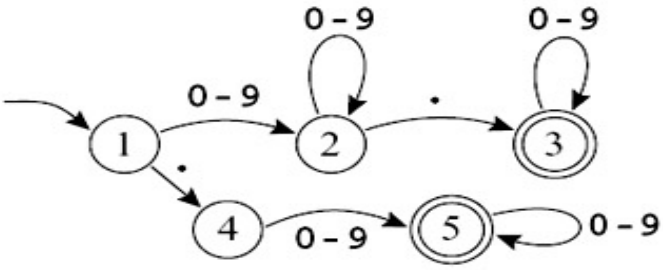
DEĞİŞKEN :



SAYI :



GERÇEL :



Şekil 1.2. Sonlu Otomata ile Gösterim

Sonlu Otomatalar; Belirli Sonlu Otomata (DFA) ve Belirsiz Sonlu Otomata (NFA) olmak üzere ikiye ayrılır.

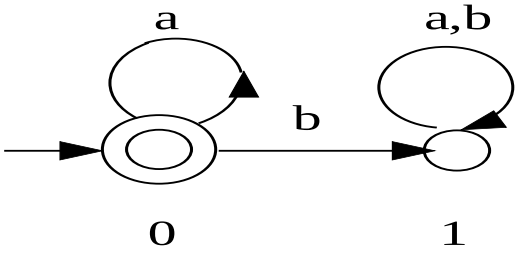
DFA, < Q , ∑ , q0 , f , A > grubundan oluşur. Öyle ki;

- Q: Durumların sonlu kümesini,
- ∑: Sembollerin sonlu kümesini,
- q0: Başlangıç durumunu,
- f: Q x ∑ → Q tanımlı geçiş fonksiyonunu,
- A: Kabul edilen durumları ifade etmektedir.

Şekil 1.3.'te Q={0,1},∑={a,b}, A={0} ve f fonksiyonu Tablo 1.2.'de verilmiş örneğin DFA'sı yer almaktadır.

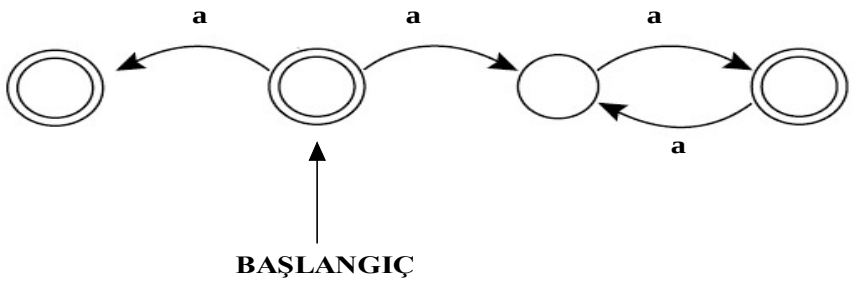
Tablo 1.2. f Fonksiyonu

Durum(q)	Giriş(a)	Sonraki Durum(f(q,a))
0	a	0
0	b	1
1	a	1
1	b	1



Şekil 1.3. Örnek DFA

NFA, bir durumdan, aynı sembol ile tanımlı ve farklı iki duruma geçiş yapabilen kenarlara ya da Ɛ ile etiketlenmiş sonlandırıcı kenarlara sahip yapılardır. Şekil 1.4.'te NFA'ya örnek bir sonlu otomata verilmiştir.



Şekil 1.4. Belirsiz Sonlu Otomata

Düzenli ifadeler şeklinde oluşturulmuş bir gramerin doğrudan DFA'ya çevriminin gerçekleştirilmesi zordur. Bu amaçla düzenli ifade çevrimi daha kolay olan NFA yapısına çevrilip ardından DFA yapısına dönüştürülür (Şekil 1.5.).

Şekil 1.5. RE - DFA Dönüşümü

1.3.1.3. Kelimesel Çözümleyici Üreteçleri

DFA üretimi bilgisayarlar aracılığı ile kolayca gerçekleştirilebilmektedir. Bu amaçla düzenli ifadeleri DFA'ya otomatik olarak çeviren kelimesel çözümleyici üreteçleri geliştirilmiştir. Bu araçlar karakter topluluklarını, kelimeler topluluğuna (token) dönüştürmede kullanılır. Bu yapılardan en çok kullanılanları;

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

Bu araçlar tanımlanan yapıları, kod içinde belirleyebilmek için otomatik kod üretirler. Günümüzde farklı programlama dillerinde kod üretebilen birçok üreteç vardır. Lex, Flex, Quex, Yacc kelimesel çözümleyici üreteçleri C/C++ programlama dillerinde, JLex [9], Javacc [4], Sablecc, Antlr [8] kelimesel çözümleyici üreteçleri ise Java programlama dilinde otomatik kod üretmektedir.

Üreteçler Java, C, C++ programlama dillerine ait kodları üretmelerine rağmen hepsinin kendine özgü sözdizim yapıları bulunmaktadır. Örneğin Java kodu üreten JavaCC

üreteci, Java'nın sözdiziminden bağımsızdır. Basit bir hesap makinesi uygulaması için, kelimesel yapıların belirlenmesinde aşağıdaki JavaCC uyumlu kod kullanılabilir:

```
PARSER_BEGIN(HesapMak)

public class HesapMak
 {… }
PARSER_END(HesapMak)

SKIP:
{
   " " | "\r" | "\t"
}

TOKEN:
{
   < SAYI: ( <RAKAM> ) + ( "." ( <RAKAM> )+ )? >
   |
   < RAKAM:  [ "0"-"9" ] >
  |
   < TOPLAMA: "+">
  |
   < CIKARMA: "-">
  |
   < CARPMA: "*">
  |
   < BOLME: "/">
  |
   < SATIRSONU: "\n" >
}
```

Üreteçler yorumlayıcı geliştirme, derleyici geliştirme, doğal dil işleme gibi birçok alanda kullanılmaktadır. Ürettikleri otomatik kod sayesinde programcıya büyük kolaylık sağlamaktadır.

1.3.2. Sözdizimsel Analiz (Ayrıştırma)

Derleyici tasarımının ikinci aşaması Sözdizimsel Analiz işlemidir. Bu aşamada gerçekleştirilecek işlemler;

- Programlama dilinin sözdizimsel kurallarının tanımlanması,
- Programın kaynak kodunun, tanımlanan dilin sözdizimsel yapısına uyup uymadığını kontrol edip var olan hataları belirlemek,
- Kelimesel analiz aşamasında üretilen kelimeler topluluğuna hiyerarşik bir yapı kazandırıp, bir sonraki derleme aşamasına geçiş için yeni bir veri yapısı üretmek (Şekil 1.6.).

Şekil 1.6. Sözdizimsel Analiz

1.3.2.1. Bağlam Bağımsız Gramer (Context Free Grammer – CFG)

Bir programlama dili kelimeler kümesinden oluşmaktadır. Her bir kelime ise önceden belirlenmiş alfabede bulunan sembollerin sonlu dizisinden oluşmaktadır. Ayrıştırma işleminde kelimeler dizisi kaynak programın, semboller dizisi kelimesel yapıların ve alfabe de kelimesel çözümleyicilerin geri döndürdüğü kelime türlerinin bir kümesidir.

CFG programlama dilinin sözdizimsel yapısını ifade etmekte kullanılan bir mekanizmadır. CFG'ler 4 temel kavramdan oluşur;

- Sonlu terminal dizisi V_t. Bu dizi kelimesel çözümleyiciler tarafından üretilen kelimelerden oluşur. Dil içinde, alfabedeki sembollerden oluşturulmuş kelimeleri ifade etmektedir. Bu ifadelerle geçiş sona erer.

- Sonlu non-terminal dizisi V_n. Üretim kuralının sol tarafında bulunan yapıyı ifade etmektedir. Terminal ya da non-terminal yapılara geçiş yapan ifadelerdir.
- Başlangıç sembolü S. Grameri başlatan non-terminali ifade etmektedir.
- Sonlu üretim (production) dizisi P. Terminal ve non-terminal yapılardan meydana gelen ve dilin gramer kurallarını tanımlayan yapıdır. Ayrıştırma ağaçlarının üretilmesinde bu yapılar kullanılır.

Üretim kurallarının formatı aşağıdaki biçimdedir;

SEMBOL ::= SEMBOL SEMBOL SEMBOL … SEMBOL

Üretim kuralının sağ tarafında en az bir tane olmak şartı ile sınırsız sayıda terminal ya da non-terminal ifade bulunabilir [18].

C programlama dilindeki "while" yapısı için bağlam bağımsız gramer aşağıdaki şekilde oluşturulabilir;

While ::= "while" "(" Exp ")" Stmt
Stmt ::= Block | Assignment | While
Block ::= "{" (Stmt) "}"*
Assignment ::= Id "=" Exp ";"
*Id ::= Letter (Letter | Digit | Symbol)**
Letter ::= ["a" – "z"] | ["A" – "Z"]
Digit ::= ["0" – "9"]
Symbol ::= ["-", "", "&&", "+", …]*
Exp ::= (Primary_Exp) (Symbol) (Primary_Exp) | Primary_Exp
Primary_Exp ::= Integer | True | False | Id | Paranthesized_Exp
*Integer ::= ["1" – "9"] (Digit)**
True ::= "True"
False ::= "False"
Paranthesized_exp ::= "(" Exp ")"

1.3.2.2. Türetim

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

Türetim işleminin farklı yöntemleri bulunmaktadır;

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

```
while(x<10)
{
    x = x + 1 ;
}
```

kodu için sola dayalı türetim;

1. *While*
2. *“ while ” “(“ Exp “)” Stmt*
3. *“ while ” “(“ Primary_Exp Symbol Primary_Exp “)” Stmt*
4. *“ while ” “(“ Id (x) Symbol Primary_Exp “)” Stmt*
5. *“ while ” “(“ Id (x) Symbol (<) Integer (10) “)” Stmt*
6. *“ while ” “(“ Id (x) Symbol (<) Integer (10) “)” Block*
7. *“ while ” “(“ Id (x) Symbol (<) Integer (10) “)” “{“ Stmt “}”*
8. *“ while ” “(“ Id (x) Symbol (<) Integer (10) “)” “{“ Assignment “}”*
9. *“ while ” “(“ Id (x) Symbol (<) Integer (10) “)” “{“ Id (x) “=” Exp “}”*
10. *“ while ” “(“ Id (x) Symbol (<) Integer (10) “)” “{“ Id (x) “=” Primary_Exp Symbol Primary_Exp “}”*
11. *“ while ” “(“ Id (x) Symbol (<) Integer (10) “)” “{“ Id (x) “=” Id (x) Symbol (+) Primary_Exp “}”*

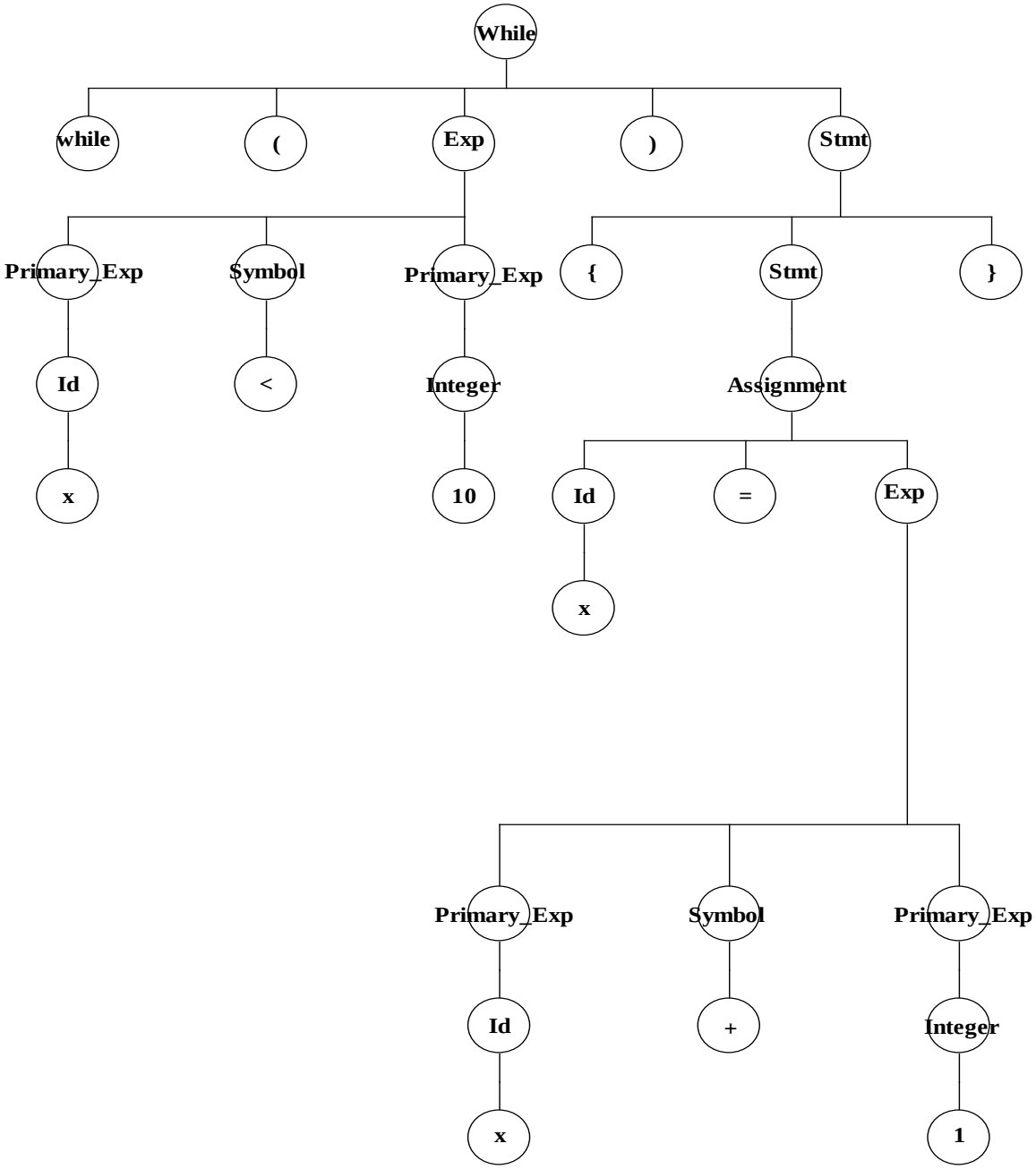
12. “ while ” “(“ Id (x) Symbol (<) Integer (10) “)” “{“ Id (x) “=” Id (x) Symbol (+) Integer (1) “}”

1.3.2.3. Ayrıştırma Ağaçları

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

```
while(x<10)
{
    x = x + 1 ;
}
```

kodu için ayrıştırma ağacı Şekil 1.7.’deki gibidir.



Şekil 1.7. Ayrıştırma Ağacı

1.3.2.4. Belirsiz Gramer

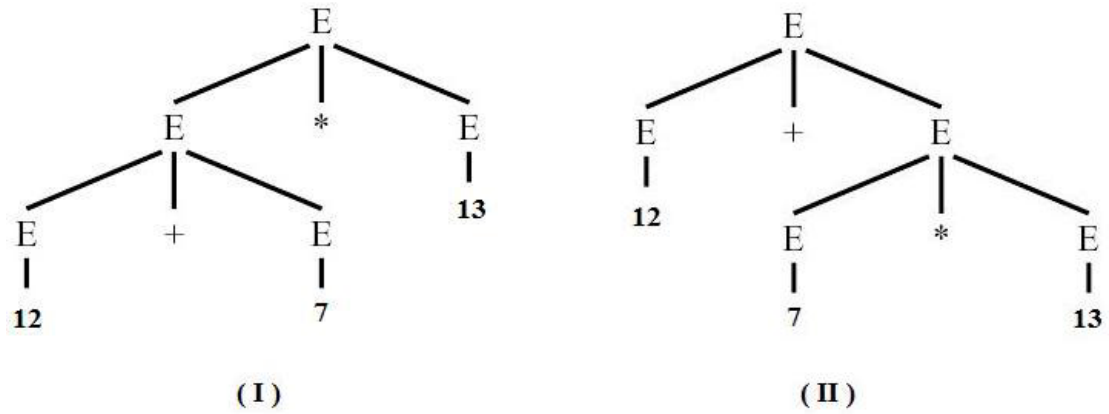
Analiz sürecinde, ayrıştırma yönüne bağlı olarak, birden çok ayrıştırma ağacı ortaya çıkaran gramerler belirsiz gramer olarak nitelendirilir [14].

Derleme esnasında belirsiz gramerler sorun teşkil etmektedir. Farklı ayrıştırma ağaçları üzerinden yapılacak olan değerlendirmeler, farklı sonuçlar elde edilmesine ve

programın anlamlandırılmasında sorunlara neden olmaktadır. Bu amaçla programlama dilleri için oluşturulan gramerlerin belirsizlikten kurtarılması gerekir. Örneğin,

E ::= *E* * *E*
E ::= *E* / *E*
E ::= *E* + *E*
E ::= *E* – *E*
E ::= *sayı*
sayı ::= *[“1” – “9”] [“0” – “9”]**

ile verilenen bir gramerin 12+7*13 ifadesi için ayrıştırma ağaçları üretilmiştir (Şekil 1.8.). Bu gramer iki farklı ayrıştırma ağacına sahip olmasından dolayı bir belirsizliğe sahiptir.



Şekil 1.8. Ayrıştırma Ağaçları

Söz konusu gramer ile 12+7*13 işlemi (I) numaralı ayrıştırma ağacı üzerinden yürütülürse 247 değeri hesaplanırken, (II) numaralı ayrıştırma ağacı kullanılırsa 103 değeri hesaplanır. Bu nedenle yukarıda verilen gramer işleç önceliklerini ve birleşme yönlerini dikkate alan ve sadece bir ayrıştırma ağacı üretecek biçimde yeniden düzenlenmelidir.

Şekil 1.8.’de ayrıştırma ağaçları gösterilen belirsiz gramerin, yeniden düzenlenmiş ve belirsizliği ortadan kalkmış biçimi

E ::= *E* + *T*

E ::= *E - T*

E ::= *T*

T ::= *T * C*

T ::= *T / C*

T ::= *C*

C ::= *sayı*

sayı ::= *[“1” – “9”] [“0” – “9”]**

gibi verilebilir.

1.3.2.5. Ayrıştırma Algoritmaları

Ayrıştırma işleminin gerçekleştirilmesi üzerine birçok algoritma geliştirilmiştir. Fakat bu algoritmalar 2 temel sınıfta toplanabilir:

- Top – Down Ayrıştırma
    - Recursive Descent Ayrıştırma
    - LL Ayrıştırma
- Bottom – Up Ayrıştırma
    - LR Ayrıştırma
        - SLR Ayrıştırma
        - LALR Ayrıştırma
        - Canonical LR Ayrıştırma

1.3.2.5.1. Recursive Descent Ayrıştırma

Basit olarak gramerde her bir non-terminal için, non-terminalin kendisi tarafından üretilebilen cümleleri ayrıştırabilen bir alt program yazılır. Pek yaygın olmamasına rağmen algoritmanın en önemli avantajı basit olması ve ayrıştırıcı üreteçlerinin kullanılmadığı durumlarda kolaylıkla oluşturulabilmesidir.

S → *while (E) { S ; }*

S → *print num*

$$E \rightarrow num == num$$

Yukarıda kuralları verilen gramerin recursive descent ayrıştırma kodu aşağıdaki gibi yazılabilir;

```
final int WHILE=1, LPAREN=2, RPAREN=3, LBRACKET=4, RBRACKET=5,  PRINT=6,
SEMICOLON=7,  NUM=8, EQUAL=9;

int tok = getToken();

void advance() {tok=getToken();}

void eat(int t) {if (tok==t) advance(); else error();}

void S() {switch(tok) {
      case WHILE: eat(WHILE); eat(LPAREN); E(); eat(RPAREN); eat(LBRACKET);
                        S(); eat(SEMI); eat(RBRACKET); break;
      case PRINT: eat(PRINT); eat(NUM); break;}
}

void E(){eat(NUM); eat(EQUAL); eat(NUM);}
```

Recursive descent ayrıştırmanın çalışması için gramerin her bir alt ifadesindeki ilk terminal sembolünün, bir sonraki adımda hangi kuralın kullanılacağını belirleyebilmesi için gerekli bilgiye sahip olması gerekir. Bu bilgiye sahip olunmazsa ayrıştırıcı çalışmaz.

Recursive descent ayrıştırma algoritmasının gerileme (backtracking) yapılmayan şekline Predictive Ayrıştırma adı verilir.

Bu algoritmada, ayrıştırma tablosunu oluşturabilmek için First ve Follow Set kavramları geliştirilmiştir.

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

FIRST set kavramına aşağıdaki gibi örnek verilebilir;

$$S \rightarrow \textit{while (E) \{ S ; \}}$$
$$S \rightarrow \textit{print num}$$
$$E \rightarrow \textit{num == num}$$

FIRST (S) = { while, print }
FIRST (E) = { num }

FOLLOW set kavramı ise non-terminali takip eden ilk terminal olarak tanımlanır. Dolayısıyla yalnızca non-terminal ifadeler için hesaplama yapılabilir. Şekil 1.9.'da aşağıda verilen gramer için hesaplanmış FIRST ve FOLLOW setleri bulunmaktadır.

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

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

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

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

Şekil 1.9. First ve Follow Setler

1.3.2.5.2. LL Ayrıştırma

Gramerin ayrıştırma tablosunda tekrarlama içermeyen yapılar LL(k) olarak isimlendirilir.

Ayrıştırma işlemini, soldan sağa doğru ve sola dayalı türetim ile gerçekleştirmektedir. Bu algoritma ile ayrıştırılabilen gramerler LL gramer olarak adlandırılır. LL(k) kullanılması k tane kelimeye bakılıp karar verileceğini anlamını taşımaktadır.

LL Ayrıştırma;

- Giriş cümlesini saklayacak bir arabelleğe,
- Ayrıştırılmış terminal ve non-terminal yapıları saklamak için bir yığına,
- Girdi ve yığında bulunan verilere dayanarak hangi gramer kuralının uygulandığını gösteren bir ayrıştırma tablosuna sahiptir.

Ayrıştırma tablosunda bulunan kurallardan, giriş cümlesinde bulunan sembollere en çok uyanını bularak yığına atar. Bu işlem girilen cümlenin sonuna gelene kadar yinelemeli bir şekilde devam ettirilir.

Ayrıştırıcı işleme başlarken "S" başlangıç sembolüne, "$" terminaline sahiptir. Bu terminal özel bir durum olup, yığının başını aynı zamanda giriş cümlesinin sonunu temsil eder.

Aşağıdaki basit bir gramer örneği için LL ayrıştırıcının ürettiği ayrıştırma tablosu gösterilmiştir;

Gramer :

1 – S → F

2 – S → (S + F)

3 – F → 5

Giriş cümlesi :

(5 + 5)

Tablo 1.3. Ayrıştırma Tablosu

	(	**)**	**5**	**+**	**$**
S	2	-	1	-	-
F	-	-	3	-	-

Ayrıştırma işlemine başlanmadan önce yığın ***[S , $]*** şeklindedir. Ayrıştırıcı ilk olarak, giriş cümlesinden "(" kelimesini ve yığından S değerini okur. Ayrıştırma

tablosundan bu kelimenin 2 numaralı kurala ait olduğunu belirler ve sonra kural numarasını geri değer olarak döndürür ve yığın içinde güncelleme işlemini gerçekler.

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

Bir sonraki adımda "(" kelimesini giriş cümlesinden ve yığından siler.

[S , + , F ,) , $]

Ayrıştırıcı bir sonraki kelimeyi belirleyerek (5), bunun 1 numaralı kural ile ardından 3 numaralı kural ile eşleştiğine karar verir. Bu adımın sonucunda yığın şu şekli alır:

[F , + , F ,) , $]

[5 , + , F ,) , $]

Sonraki iki adımda ayrıştırıcı "5" ve "+" kelimelerini giriş cümlesinden ve yığından siler.

[F ,) , $]

Ardından ayrıştırıcı "5" kelimesini 3 numaralı kural ile eşleştirir. Son olarak F ve ")" sembollerini yığından atarak işlemi tamamlar. Sonuçta sadece "$" sembolünün kalması ve giriş cümlesinin sonuna varılması, ayrıştırma sonucunda giriş cümlesinin gramere uygun olduğunu gösterir.

Giriş cümlesinin ayrıştırılma işleminden sonra geriye kural dizisi döner. Bu örnek için bu dizi [2, 1, 3, 3] şeklinde olur.

S → (S+F) →(F+F) → (5+F) → (5+5)

Bir gramerin LL(k) olabilmesi için bazı özelliklere sahip olması gerekir. Bunlar:

- Gramer soldan yinelemeli olmamalı; Soldan yinelemeden kurtulmak için, soldan yineleme içeren kural sağdan yineleme kullanılarak tekrar yazılır. Örneğin;

S	→	*S * E*	*S*	→	*E S'*
S	→	*E*	*S'*	→	** E S'*
			S'	→	

- Sol faktörleme (Left Factoring); İki kuralın aynı sembol ile başlaması LL(k) grameri için diğer bir sorundur. Bunu ortadan kaldırmak için sol faktörleme yapılır. Örneğin;

S	→	*if E then S else S*
S	→	*if E then S*

kuralı için aşağıdaki gibi sol faktörleme gerçekleştirilir:

$S \rightarrow$ *if E then S X*

$X \rightarrow$

$X \rightarrow$ *else S*

1.3.2.5.3. LR Ayrıştırma

LL ayrıştırma, uygulanan kuralı tahmin yoluyla tespit ettiği için zayıf bir ayrıştırma tekniğidir. Bunun yerine daha güçlü bir teknik olan LR ayrıştırma geliştirilmiştir.

LR ayrıştırma girilen cümleyi soldan sağa okur, ancak sağa dayalı türetim üretir. LR ayrıştırma:

- Giriş cümlesini saklayacak bir arabelleğe,
- Hangi durumda olunduğunu gösteren bir yığına,
- Yeni durumun ne olacağını belirleyen goto tablosuna,
- Giriş cümlesinde o anki terminale ve duruma göre uygulanacak gramer kurallarını tutan action tablosuna sahiptir.

LR ayrıştırma, tablo tabanlı bir yöntemdir. Bu yöntemin *action* ve *goto* olmak üzere iki bileşeni vardır. Action tablosu verilen bir ayrıştırıcı durumuna ve sonraki kelime için yapılması gereken işlemi belirler. Goto tablosu ise indirgeme işlemi yapıldıktan sonra yığın üzerine hangi durumun getirileceğini belirler.

Ayrıştırma algoritması aşağıdaki adımlardan oluşur;

- Başlangıç yapılandırması gerçekleştirilir.
- Verilen duruma ve giriş cümlesinde o anki terminal değerlerine bağlı olarak action tablosundan eşleştirme yapılır. Burada dört durum bulunmaktadır;
    - Kaydırma sn:
        - Terminal değeri giriş cümlesinden silinir,
        - n durumu yığına itilir ve şu anki durum olarak setlenir.
    - İndirgeme rk:
        - k numaralı kural çıktı olarak verilir,
        - m kuralının sağ tarafında bulunan her bir sembol yığından silinir,
        - m kuralının sol tarafı yeni durum olur ve yığının başına eklenir.

- Kabul etme: Giriş cümlesi kabul edilir.
- Hata: Hata olduğu tespit edilir ve raporlanır.

• Yukarıdaki adım hata meydana gelene kadar ya da giriş cümlesi kabul edilene kadar sürdürülür.

LR ayrıştırmanın birçok avantajı bulunmaktadır;

• Birçok programlama dili LR ayrıştırmanın türevleri kullanılarak ayrıştırılabilir,

• LR ayrıştırıcılar etkin bir biçimde gerçeklenebilir,

• Diğer ayrıştırıcılara göre sözdizimsel hataları daha kısa zamanda tespit eder ve raporlar.

LR ayrıştırmanın tek dezavantajı ise elle üretilemeyecek kadar zor bir yöntemdir. Bu amaçla bu işi gerçekleştiren ayrıştırıcı üreteçleri kullanılmaktadır.

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

• SLR(Simple LR Parser)

• LALR(Look-Ahead LR Parser)

• CLR(Canonical LR Parser)

Sırasıyla her bir sınıf kendinden önce gelen sınıfı kapsar ve daha çok gramer setini ifade eder.

Aşağıda verilen gramer ve giriş cümlesi için Şekil 1.10.'da ayrıştırma tablosu, Şekil 1.11.'de LR algoritması ile ayrıştırılması gösterilmiştir:

Gramer :

1 -	*S*	→	*S ; S*
2 -	*S*	→	*id := E*
3 -	*S*	→	*print (L)*
4 -	*E*	→	*id*
5 -	*E*	→	*num*
6 -	*E*	→	*E + E*
7 -	*E*	→	*(S , E)*
8 -	*L*	→	*E*
9 -	*L*	→	*L , E*

Giriş cümlesi : *a := 7;*

b := c + (d := 5+6 , d)

	id	num	print	;	,	+	:=	(	)	$	S	E	L
1	s4		s7								g2		
2				s3						a			
3	s4		s7								g5		
4							s6						
5				r1	r1					r1			
6	s20	s10						s8				g11	
7								s9					
8	s4		s7								g12		
9	s20	s10						s8				g15	g14
10				r5	r5	r5			r5	r5			
11				r2	r2	s16				r2			
12				s3	s18								
13				r3	r3					r3			
14					s19				s13				
15					r8				r8				
16	s20	s10						s8				g17	
17				r6	r6	s16			r6	r6			
18	s20	s10						s8				g21	
19	s20	s10						s8				g23	
20				r4	r4	r4			r4	r4			
21									s22				
22				r7	r7	r7			r7	r7			
23					r9	s16			r9				

Şekil 1.10. Verilen Gramer için Ayrıştırma Tablosu

YIĞIN	GİRİŞ CÜMLESİ	ACTION	
1	a := 7 ; b := c + (d := 5 + 6 , d) $	Kaydırma	
1 id_4	:= 7 ; b := c + (d := 5 + 6 , d) $	Kaydırma	
1 id_4 :=$_6$	7 ; b := c + (d := 5 + 6 , d) $	Kaydırma	
1 id_4 :=$_6$ num_{10}	; b := c + (d := 5 + 6 , d) $	İndirgeme	$E \rightarrow$ num
1 id_4 :=$_6$ E_{11}	; b := c + (d := 5 + 6 , d) $	İndirgeme	$S \rightarrow$ id:=E
1 S_2	; b := c + (d := 5 + 6 , d) $	Kaydırma	
1 S_2 ;$_3$	b := c + (d := 5 + 6 , d) $	Kaydırma	
1 S_2 ;$_3$ id_4	:= c + (d := 5 + 6 , d) $	Kaydırma	
1 S_2 ;$_3$ id_4 :=$_6$	c + (d := 5 + 6 , d) $	Kaydırma	
1 S_2 ;$_3$ id_4 :=$_6$ id_{20}	+ (d := 5 + 6 , d) $	İndirgeme	$E \rightarrow$ id
1 S_2 ;$_3$ id_4 :=$_6$ E_{11}	+ (d := 5 + 6 , d) $	Kaydırma	
1 S_2 ;$_3$ id_4 :=$_6$ E_{11} +$_{16}$	(d := 5 + 6 , d) $	Kaydırma	
1 S_2 ;$_3$ id_4 :=$_6$ E_{11} +$_{16}$ (8	d := 5 + 6 , d) $	Kaydırma	
1 S_2 ;$_3$ id_4 :=$_6$ E_{11} +$_{16}$ (8 id_4	:= 5 + 6 , d) $	Kaydırma	
1 S_2 ;$_3$ id_4 :=$_6$ E_{11} +$_{16}$ (8 id_4 :=$_6$	5 + 6 , d) $	Kaydırma	
1 S_2 ;$_3$ id_4 :=$_6$ E_{11} +$_{16}$ (8 id_4 :=$_6$ num_{10}	+ 6 , d) $	İndirgeme	$E \rightarrow$ num
1 S_2 ;$_3$ id_4 :=$_6$ E_{11} +$_{16}$ (8 id_4 :=$_6$ E_{11}	+ 6 , d) $	Kaydırma	
1 S_2 ;$_3$ id_4 :=$_6$ E_{11} +$_{16}$ (8 id_4 :=$_6$ E_{11} +$_{16}$	6 , d) $	Kaydırma	
1 S_2 ;$_3$ id_4 :=$_6$ E_{11} +$_{16}$ (8 id_4 :=$_6$ E_{11} +$_{16}$ num_{10}	, d) $	İndirgeme	$E \rightarrow$ num
1 S_2 ;$_3$ id_4 :=$_6$ E_{11} +$_{16}$ (8 id_4 :=$_6$ E_{11} +$_{16}$ E_{17}	, d) $	İndirgeme	$E \rightarrow E + E$
1 S_2 ;$_3$ id_4 :=$_6$ E_{11} +$_{16}$ (8 id_4 :=$_6$ E_{11}	, d) $	İndirgeme	$S \rightarrow$ id:=E
1 S_2 ;$_3$ id_4 :=$_6$ E_{11} +$_{16}$ (8 S_{12}	, d) $	Kaydırma	
1 S_2 ;$_3$ id_4 :=$_6$ E_{11} +$_{16}$ (8 S_{12} ,$_{18}$	d) $	Kaydırma	
1 S_2 ;$_3$ id_4 :=$_6$ E_{11} +$_{16}$ (8 S_{12} ,$_{18}$ id_{20}	) $	İndirgeme	$E \rightarrow$ id
1 S_2 ;$_3$ id_4 :=$_6$ E_{11} +$_{16}$ (8 S_{12} ,$_{18}$ E_{21}	) $	Kaydırma	
1 S_2 ;$_3$ id_4 :=$_6$ E_{11} +$_{16}$ (8 S_{12} ,$_{18}$ E_{21})$_{22}$	$	İndirgeme	$E \rightarrow (S, E)$
1 S_2 ;$_3$ id_4 :=$_6$ E_{11} +$_{16}$ E_{17}	$	İndirgeme	$E \rightarrow E + E$
1 S_2 ;$_3$ id_4 :=$_6$ E_{11}	$	İndirgeme	$S \rightarrow$ id:=E
1 S_2 ;$_3$ S_5	$	İndirgeme	$S \rightarrow S; S$
1 S_2	$	Kabul Etme	

Şekil 1.11. Giriş cümlesinin ayrıştırılmış hali

1.3.2.6. Ayrıştırıcı Üreteçleri

Recursive descent dışında kalan diğer ayrıştırma algoritmalarının gerçeklenmeleri elle oluşturulamayacak kadar zordur. Bu sebepten dolayı kelimesel çözümleyici üreteçlerinde olduğu gibi ayrıştırma işlemlerini de otomatikleştiren üreteçler geliştirilmiştir.

Tablo 1.4.'te [21] bazı üreteçler, oluşturuldukları algoritmalar ve kod ürettikleri programlama dilleri ile birlikte gösterilmiştir.

Tablo 1.4. Ayrıştırıcı Üreteçleri

Üreteç	**Ayrıştırma Algoritması**	**Programlama Dili**
ANTLR	LL(k)	C#, Java, Phyton
ANAGRAM	LALR	C, C++
BISON	LALR	C,C++
COCO/R	LL(k)	C, C++, C#, F#, Java, …
JAVACC	LL(k)	Java
SABLECC	LALR	Java, C, C++, C#, Phyton
YACC	LALR	C

Hesap makinesi uygulaması için sözdizimsel analiz JavaCC dilinde aşağıdaki biçimde gerçekleştirilebilir:

```
…
void ayristirma():
{}
{
   ifade() <SATIRSONU> | <SATIRSONU> | <EOF>
}
void ifade():
{}
{
   terim() <TOPLAMA> ifade() | <CIKARMA> ifade()
```

```
}
void terim():
{}
{
   tekli() <CARPMA> terim() | <BOLME> terim()
}
void tekli():
{}
{
   <CIKARMA> eleman() | eleman()
}
void eleman():
{}
{
   <SAYI> | "(" ifade() ")"
}
```

1.3.2.7. Soyut Sözdizim

Bir yorumlayıcı kodun programlama diline ait olup olmadığının tespitinden daha fazlasını yapmalıdır. Yorumlayıcı sözdizimsel olarak dile uygun olan kodun anlamsal olarak da değerlendirmesini yapmalıdır. Bu anlamsal işlemler ayrı kod ve kontrol gerektirmektedir.

Recursive descent, JavaCC ayrıştırıcılarında anlamsal işlem kodu ayrıştırma kodunun arasına yazılır. SableCC ayrıştırıcısında ise otomatik olarak anlamsal işlem kodlarının yazılması için sözdizim ağacı oluşturulur.

1.3.2.7.1. Anlamsal İşlemler

Gramer içinde her bir terminal ve non-terminal kendi anlamsal değerlerine bağlı olarak bir tip ile ilişkilendirilmelidir. Örneğin hesap makinesi uygulamamızda “eleman()” non-terminalinin döndüreceği değere bağlı olarak “ifade()” non-terminali de aynı değeri

geri döndürmelidir. Eğer yapılan işlemde tutarsızlık olursa gramer dile uygun olsa bile hata bildirimi yapılmalıdır.

Otomatik ayrıştırıcı üreteci olan JavaCC'de bu kontrolü sağlamak için yazılan sözdizimsel kod bloğuna, ek kod yazılması gerekir. Hesap makinesi örneğinin anlamsal işlemlere uygun olarak düzenlenmiş hali aşağıdaki gibidir;

```
void ayristirma():
{   double a;  }
{
   a = ifade()  <SATIRSONU>   |   <SATIRSONU>   |   <EOF>
}
double ifade():
{  double a;   double b;   }
{
   a=terim()
   (
      <TOPLAMA> b=ifade()   {a += b;}   |   <CIKARMA> b=ifade()   {a -= b;}
   )*
      { return a;}
}

double terim():
{  double a;   double b;   }
{
   a=tekli()
   (
      <CARPMA> b=terim()   {a *= b;}   |   <BOLME> b=terim()   {a /= b;}
   )*
      {return a;}
}
double tekli():
{  double a;   }
{
```

```
    <CIKARMA> a=eleman()    {return –a;}  |  a=eleman()       {return a;}
}
double eleman():
{  Token t;   double a;   }
{
   t=<SAYI>   |    "(" a=ifade() ")"    {return a;}
}
```

JavaCC için oluşturulan bu koda gerek kalmadan sözdizim ağacını oluşturabilmek için JTB (Java Tree Builder) [3], JJTree gibi hazır araçlar bulunmaktadır.

1.3.2.7.2. Soyut Sözdizim Ağacı

Anlamsal işlem aşamalarını içeren bir yorumlayıcı otomatik ayrıştırıcılarla yazılabilir. Ancak böyle bir programın okunması ve onarılması çok güçtür.

Birimselliği sağlamak amacıyla ayrıştırma işlemleri ile anlamsal işlemleri birbirinden ayırmak gerekmektedir. Bunu yapmanın bir yolu ayrıştırıcının somut sözdizim ağacı oluşturmasıdır. Teknik olarak sözdizim ağacı giriş cümlesinin her bir kelimesi için bir yaprak ve her bir gramer kuralı için bir düğüm içerir.

Fakat bunun kullanılması pek uygun değildir. Bu kullanımda birçok yaprak gereksiz bilgi tutulmasına sebep olacaktır. Bunlara hiçbir anlama sahip olmayan noktalama işaretleri örnek olarak verilebilir. Bunun yanı sıra somut sözdizim ağacının yapısı gramere bağımlı olacaktır. Gramerin soldan yinelemelerden kurtarılması, belirsizliğin kaldırılması gibi durumlarla büyüyeceği göz önünde bulundurulursa somut sözdizim ağacının bu iş için kullanımı uygun değildir.

Soyut sözdizim ağacı, ayrıştırma ile yorumlayıcının sonraki aşamaları arasında arayüz oluşturabilecek niteliktedir. Soyut sözdizim ağacı herhangi bir anlamsal yorumlama yapmadan kaynak programın ayrıştırma safhalarını bir sonraki aşamaya iletir.

Ayrıştırma işleminde somut sözdizim ağacı kullanılır. Soyut sözdizim ağacı ayrıştırma için herhangi bir anlam taşımaz. Bu yüzden belirsiz olmasının ya da soldan yinelemeli olmasının bir önemi yoktur.

Soyut sözdizim ağacı oluşturmada otomatik üreteçler kullanılmaktadır. Bu üreteçler sayesinde kendi içinde soyut sözdizim ağacı oluşturmayan ayrıştırıcılarda isteğe bağlı olarak kullanılabilir.

1.3.2.7.3. Visitor

Visitor deseni (pattern) basit anlamda oluşturulan sözdizim ağacındaki düğümler arasında geçişi sağlamayı gerçekleştirir.

Bu desen nesneye dayalı sistemleri daha esnek hale getirmeyi amaçlayan birçok desenden bir tanesidir. Visitor deseni birden çok nesneden oluşan yapıları kolayca kullanabilme imkânı verir. Visitor kullanmadan uygulanacak diğer yöntemler çeşitli sorunlara yol açacaktır. Aşağıdaki örnekte bu gözlenmektedir:

```
interface List{}
class Nil implements List{}
class Cons implements List {
  int head;
  List tail;
}
```

Yöntem1: Instanceof ve Tip Biçimlemesi

İlk yaklaşım verilen listenin Nil veya Cons nesnesi olup olmadığını *instanceof* ile kontrol eden bir döngü olabilir. Eğer Cons nesnesi ise alanlara tip biçimlemesi ile erişim sağlanır ve döngü devam eder.

```
List l;                          //Üzerinde çalıştığımız list nesnesi
int sum = 0;                     //döngü sonucundaki toplam
boolean proceed = true;
while(proceed){
if(l instanceof Nil)
    proceed = false;
else if (l instanceof Cons){
    sum = sum+((Cons) l ).head ;
```

```
    l = ((Cons) l ).tail;
}
}
```

Kodu bu şekilde oluşturmanın avantajı Nil ve Cons sınıflarından bağımsız biçimde olmasıdır. Dezavantajı ise yazılan kodun, nesnenin hangi sınıfa ait olduğunu sabit olarak her seferinde tip biçimleme ve *instanceof* yapılarını kullanarak belirliyor olmasıdır.

Yöntem2: İlişkili Metotlar Yazmak

İlk yöntem nesne yönelimli bir yol değildir. Nesne yönelimli programlarda nesnenin herhangi bir parçasına erişebilmek için geleneksel yöntem ilişkili metotlar yazmaktır. Yukarıdaki örneğe her bir sınıf için sum() fonksiyonu eklenirse;

```
interface List{
 int sum();
}
class Nil implements List{
public int sum(){
    return 0;
}
}
class Cons implements List{
int head;
List tail;
public int sum(){
   return head+tail.sum();
}
}
```

elde edilir. Bu durumda verilen bir List değişkeni için tüm değerlerin toplamı ***l.sum()*** kodu ile hesaplanabilir. Bu yöntemin avantajı tip biçimleme ve *instanceof* yapılarından vazgeçilip sistematik bir yöntemle yazılmış olmasıdır. Yöntemin dezavantajı ise her yeni işlem için her bir sınıfa özel yeni bir metot tanımlaması gerektirmesidir.

Yöntem3: Visitor Deseni

Visitor deseninde, mevcut sınıfların nesne yapılarını değiştirmeden, sadece nesne yapısında yeni bir işlem tanımlanır. Her bir sınıf içine accept() adı verilen yeni bir metot eklenir ve visit() adı verilen yeni bir işlem kodunun içine ise yapılması istenenler yazılır. Bu desene uygun kod şu şekildedir:

```
interface List {
   void accept(Visitor v);
}
interface Visitor {
   void visitNil(Nil x);
   void visitCons(Cons x);
}
class Nil implements List {
  public void accept(Visitor v) {
     v.visitNil(this);
  }
}
class Cons implements List {
  int head;
  List tail;
  public void accept(Visitor v) {
     v.visitCons(this);
  }
}
class SumVisitor implements Visitor {
  int sum = 0;
  public void visitNil(Nil x) {}
  public void visitCons(Cons x) {
     sum = sum + x.head;  x.tail.accept(this);
  }
}
```

Her bir accept() metodu argüman olarak visitor nesnesi alır. Visitor arayüzü her bir temel sınıf için bir başlığa sahiptir. Bu durumda verilen List değişkeni için tüm değerlerin toplamı şu şekilde hesaplanır;

```
SumVisitor sv = new SumVisitor();
l.accept(sv);
System.out.println(sv.sum);
```

Bu yöntemin avantajı var olan sınıfları yeniden derlemeden, nesneleri kontrol edebilen kodlar yazılabilmesidir.

1.3.2.7.4. JTB (Java Tree Builder)

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

JTB gramer dosyasını işledikten sonra aşağıdaki yapıları üretir:

- jtb.out.jj dosyası; gramer dosyası ile aynı yapıda olup ayrıştırma esnasında sözdizim ağacını oluşturabilmek için anlamsal işlemler eklenir. Bu dosya üretildikten sonra diğer dosyaya gerek kalmaz.
- syntaxtree klasörü; gramer içinde tanımlı tüm kurallar için oluşturulan sınıfları içerir. JTB ile gramer dosyasında yer alan kural sayısı kadar sınıf üretilir. Bu sınıflar kuralın sol tarafında yer alan isim ile aynı isme sahiptir. Bu sınıflar bir kez üretildikten sonra yeniden düzenlemeye gerek kalmaz.

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

- Node Sınıfı; tüm ağaç düğümlerinin üretildiği temel düğüm arayüzüdür.
- NodeListInterface Sınıfı; NodeList, NodeListOptional ve NodeSequence sınıflarının gerçeklenmesi için gerekli arayüzdür.
- NodeChoie Sınıfı; (A | B) durumunda kullanılır.
- NodeList Sınıfı; (A)+ durumunda kullanılır.
- NodeListOptional Sınıfı; (A)* durumunda kullanılır.

- NodeOptional Sınıfı; (A)? Durumunda kullanılır.
- NodeSequence Sınıfı; "…" (A)* gibi durumlarda kullanılır.
- NodeToken Sınıfı; "…" durumunda kullanılır.

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

Visitor arayüzü gramer içindeki her bir kural için bir fonksiyon tanımlanır. Visit() adı verilen bu fonksiyonların her biri daha önceden syntaxtree klasörü içerisinde tanımlanmış sınıfların her birini argüman olarak alır. Aşırı yüklenmiş bu fonksiyon aldığı argümana göre ilgili kodu çalıştırır.

Visit() fonksiyonu ağacın hangi düğümünde bulunulduğu hakkında bilgi verir. Bu sayede ağacın hangi düğümünde ne yapılacağına karar verilerek bu fonksiyonlar kullanıcının isteğine göre doldurulabilir.

Visit() fonksiyonu altında accept() fonksiyonu çağırılarak ağaçta bir alt düğüme geçilmesi sağlanır. Ağacın düğümlerinde hatalı bir alt düğüme geçilmeye çalışılırsa accept() fonksiyonlarından geriye dönülmeden hata üretilir ve işlemden çıkılır.

Gramer dosyası JTB ile yeniden yapılandırılarak kaynak kodun değerlendirildikten sonra ağaç biçiminde bir veri yapısının geriye döndürülmesi sağlanır. Böylece ihtiyaç duyulan anlamsal kontroller ağaç üzerinden gidilerek kolayca yapılabilir.

1.3.3. Anlamsal Analiz

Anlamsal analiz aşamasında;

- Yorumlayıcı değişken tanımlamalarını, kullanımları ile ilişkilendirir,
- İfadelerin tip bilgilerinin doğruluğunu kontrol eder,
- Sözdizim ağacını inceleyip yorumlama safhasına geçmeden önce son kez kodu kontrol eder.

Anlamsal analizde iki temel aşama bulunmaktadır. Bu aşamalar;

- Sembol tablosunun oluşturulması,
- Tip kontrolünün gerçekleştirilmesi.

1.3.3.1. Sembol Tablosu

Bu aşamada tanımlayıcılar, tanımlayıcı tipleri ve tanımlayıcı kapsamları bilgisi sembol tablosuna yerleştirilir. Kaynak kodda tip bildirimleri, değişken ve fonksiyon tanımlamaları yapıldığında bu tanımlayıcı bilgileri sembol tablosunda oluşturulur. Kodun sonraki kısımlarında kullanımına rastlanan tanımlayıcıların gerekli bilgileri sembol tablosundan bakılıp, karar verilir.

Program içindeki her bir yerel değişkenin geçerli olduğu bir alan vardır. Örneğin fonksiyon içinde tanımlanmış bir değişkenin geçerlilik alanı fonksiyonun tanımlandığı yerdir.

Tanımlayıcıların tipleri, geçerlilik alanları gibi bilgilerin tutulduğu sembol tablosuna çevre adı verilir ve → işareti ile gösterilir.

Aşağıda örnek olarak yazılmış kodun sembol tablosunun içeriğini gösterelim.

```
1)    {
2)        int a;
3)        a = 0;
4)        {
5)            int a;
6)            a = 1;
7)            System.out.println( a ) ;
8)        }
9)        a ++ ;
10)       System.out.println( a ) ;
11)   }
```

σ_0 = { a → Integer }

$\sigma_1 = \sigma_0$ + { a → Integer } şeklindedir.

7. satıra gelindiğinde aktif olan çevre σ_1 olduğu için a değişkenin değeri ekrana 1 olarak bastırılır. 10. Satıra gelindiğinde ise σ_1 çevresi geçerliliğini yitirip, σ_0 aktif olur. Bu sebepten dolayı ekrana tekrardan 1 değeri bastırılır.

Sembol tablosunun yönetimi, fonksiyonel ve emirsel (imperative) programlama dillerinde farklı şekilde gerçekleştirilir. Fonksiyonel programlama dillerinde yeni değer eski sembol tablosu korunacak şekilde yeni sembol tablosu oluşturularak yazılır. Emirsel dillerde ise yeni değer eski sembol tablosu üzerine güncellenerek yazılır. Yapılan bu güncelleme daha sonraki adımlarda eski sembol tablosuna geri dönebilmeyi sağlayacak niteliktedir.

Büyük programlarda birçok farklı değişkenin olmasından dolayı sembol tablosunda arama işlemi etkin bir şekilde gerçekleştirilmelidir.

Emirsel stilde arama işlemi için hash tabloları kullanılır. Bu yöntemle beraber arama hızlandırılır ve silme işlemi etkin şekilde yapılır.

Fonksiyonel stilde ise arama işlemi ikili ağaçlar yardımıyla etkin biçimde gerçekleştirilir.

Aşağıdaki kod bloklarında fonksiyonel ve emirsel stilde oluşturulan sembol tablolarının gerçeklenmesi için örnek arayüzler bulunmaktadır.

```
Public class Symbol {
  public String toString();
  public static Symbol symbol(String s);
}
```

Emirsel diller için;

```
public class Table {
  public Table();
  public void put(Symbol key, Object value);
  public Object get(Symbol key);
  public void beginScope();
  public void endScope();
  public java.util.Enumeration keys();
}
```

Fonksiyonel diller için;

```
public class Table {
```

```
  public Table();
  public Table put(Symbol key, Object value);
  public Object get(Symbol key);
  public java.util.Enumeration keys();
}
```

Yukarıda yer alan Symbol arayüzü gereksiz dizgi karşılaştırmalarından kurtulmak için oluşturulmuştur. Bu arayüz sayesinde her bir tanımlayıcı tek bir sembole karşılık düşürülür. Arama esnasında tüm yapıyı karşılaştırmak yerine sadece semboller karşılaştırılır.

1.3.3.1.1. Çoklu Sembol Tabloları

Bazı programlama dillerinde program esnasında birden fazla aktif çevre olabilir. Bunun nedeni programdaki her modülün ve her sınıfın kendi sembol tablosunun olmasıdır. Aşağıdaki Java programlama dilinde yazılmış kod bloğu çoklu sembol tablosuna örnek olarak verilmiştir:

```
package M;
 class sinif1 {
    static int degisken1 = 5;
 }
 class sinif2 {
    static int degisken1 = 10;
    static int degisken2 = sinif1.degisken1 + degisken1;
 }
 class sinif3 {
    static int degisken1 = sinif1.degisken1 + sinif2.degisken2;
 }
```

σ_0 = { degisken1 → Integer }

σ_1 = { sinif1 → σ_0 }

σ_2 = { degisken1 → Integer, degisken2 → Integer}

σ_3 = { sinif2 → σ_2 }

σ_4 = { degisken1 → Integer }

σ_5 = { sinif3 → σ_4 }

$\sigma_6 = \sigma_1 + \sigma_3 + \sigma_5$

Yukarıda yazılmış program için geçerli olan çevre çoklu sembol tablosuna sahip olan σ_6'dır.

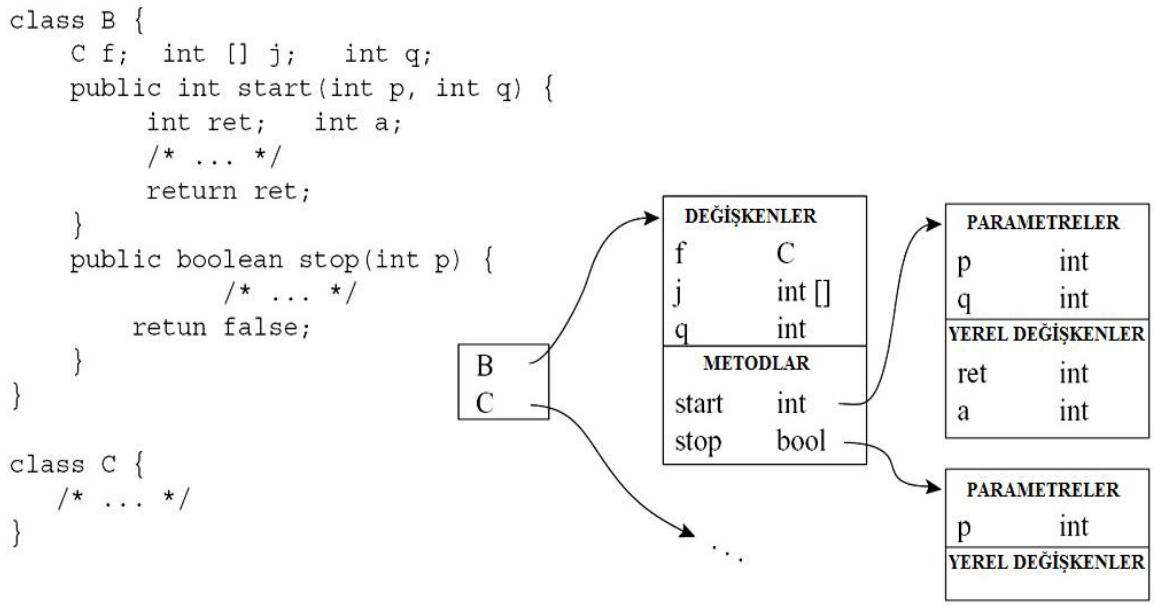
1.3.3.2. Tip Kontrolü

Kod içinde kullanılan tanımlayıcıların tip bilgileri sembol tablosuna eklendikten sonra, bu tanımlayıcıların uygun şekilde kullanılıp kullanılmadığının tespit edilmesi işlemi tip kontrolü olarak adlandırılır.

Tip kontrolünün gerçekleştirilebilmesi için sembol tablosu program içerisinde tanımlanmış bazı tür bilgilerini içermelidir.

- Değişken isimleri ve formal parametreler tip bilgileri ile ilişkilendirilmeli,
- Metot isimleri, bu metodun aldığı parametreler, sonuç tipi ve yerel değişkenleri ile ilişkilendirilmeli,
- Eğer nesneye dayalı bir dil ise sınıflar da, içinde tanımlanmış değişken ve metotlarla ilişkilendirilmelidir.

Şekil 1.12.'de tip kontrolü yapılabilmesi için bir kodun sembol tablosunda yer alması gereken bilgilere örnek verilmiştir.



Şekil 1.12. Kod bloğu ve ona ilişkin sembol

Şekil 1.12.'de tip kontrolü iki aşamada gerçekleştirilir. İlk aşamada sembol tablosu oluşturulur, ikinci aşamada ise durumların ve ifadelerin tipleri kontrol edilir. İkinci aşamaya geçilmeden önce sembol tablosunun her bir tanımlayıcı için doldurulmuş olması gerekir. Aksi halde sembol tablosuna eklenmemiş bir tanımlayıcının kullanılması durumunda hata meydana gelecektir.

Tip kontrolünün ilk aşamasında visitor tarafından sözdizim ağacındaki düğümler taranarak sembol tablosu oluşturulur. İkinci aşamasında yine sözdizim ağacı taranarak bu sefer gerekli kontroller yapılır.

Tip kontrolü ile yorumlama aşamasında çıkacak sorunlar önceden engellenmiş olur. İki çeşit tip kontrolü vardır;

- Statik kontrol
- Dinamik kontrol

Statik kontrolde tanımlayıcı tipleri yorumlama aşamasında kontrol edilirken, dinamik kontrolde koşma anında tip kontrolü gerçekleştirilir. Statik kontrol yapan bazı programlama dillerine C, C++, C#, Java, Fortan, ML, Pascal, Haskell ve dinamik kontrol yapan bazı programlama dillerine ise Groovy, JavaScript, Lisp, Perl, Php, Prolog örnek olarak gösterilebilir.

1.3.4. Yorumlama

Anlamsal analiz ile kod analizi tamamlandıktan sonra kod icra edilmeye (yorumlanmaya) hazır hale gelir.

Yorumlama işleminde de anlamsal analizde olduğu gibi sözdizim ağacından faydalanılır. Bu ağacın düğümlerinde ilerleyerek kodun icrası satır satır gerçekleştirilir.

Bu aşamada, anlamsal analizde kullanılmış sembol tablosunda artık tanımlayıcı özelliklerin yerine değişkenlerin değerleri tutulmaktadır. Bunun için ya sembol tablosu silinip yerine yeni bir tablo oluşturulabilir ya da sembol tablosunda bulunan alanlar güncellenebilir.

Değişkenlerin değerleri yorumlama esnasında değer tablosuna eklenir. Gerektiğinde de bu istenilen değerler tablodan çekilir ya da tablo üstünde güncelleme yapılıp saklanır.

1.4. JavaCC

JavaCC, Java programlama dilinde uygulama geliştirmeye imkân sağlayan kelimesel çözümleyici üreteci ve aynı zamanda ayrıştırıcı üretecidir [17]. Bağlam bağımsız gramere bağlı olarak Java dilinde recursive descent ayrıştırma kodu üretir.

JavaCC ile uygulama geliştirme dört adımdan oluşmaktadır [11]:

- Genişletilmiş BNF'de kullanılacak dilin grameri hazırlanır.
- Genişletilmiş BNF dosyası, JavaCC gramer dosyasına çevrilir.
- JavaCC gramer dosyasından uygulama için Java koduna dönüşüm gerçekleştirilir.
- Oluşturulan Java dosyası geliştirilecek uygulama içinde kullanılır.

1.4.1. Gramerin Belirlenmesi

Aşağıda örnek bir gramerin [11] genişletilmiş BNF özellik dosyası gösterilmiştir. Terminal yapılar '<' ve '>' işaretleri arasında bulunmaktadır.

*Expression ::= AndExpression (<OR> AndExpression)**
AndExpression ::= EqComparisonExpression (<AND>
*EqComparisonExpression)**
EqComparisonExpression ::= ComparisonExpression ((<EQ> | <NE>)

*ComparisonExpression)**

ComparisonExpression ::= AdditiveExpression ((<LT> | <GT> | <LE> | <GE>)
*AdditiveExpression)**

AdditiveExpression ::= MultiplicativeExpression ((<PLUS> | <MINUS>)
*MultiplicativeExpression)**

MultiplicativeExpression ::= ArithmeticUnaryExpression ((<MULT> | <SLASH>)
*ArithmeticUnaryExpression)**

ArithmeticUnaryExpression ::= (<PLUS> | <MINUS>)
ArithmeticUnaryExpression | BooleanUnaryExpression

BooleanUnaryExpression ::= (<TILDE> | <EXCL>) ArithmeticUnaryExpression
| PrimitiveExpression

PrimitiveExpression ::= Literal | Function | <IDENTIFIER> | <LPAREN>
Expression <RPAREN>

Function ::= <IDENTIFIER> Arguments

Literal ::= <INTEGER_LITERAL> | <FLOATING_POINT_LITERAL>
| <STRING_LITERAL> | BooleanLiteral

BooleanLiteral ::= <TRUE> | <FALSE>

Arguments ::= <LPAREN> (ArgumentList)? <RPAREN>

*ArgumentList ::= Expression (<COMMA> Expression)**

1.4.2. JavaCC Dosyasına Çevirim

Genişletilmiş BNF dosyası uygun şekilde hazırlandıktan sonra kolayca JavaCC özellik dosyasına geçiş yapılabilir. JavaCC özellik dosyası 4 ayrık bloktan oluşmaktadır. İlk blok JavaCC'nin gramer dosyasında nasıl işlem yapacağını belirleyen özelliklerden oluşur. İkinci blok, temel ayrıştırıcı sınıf tanımının yapıldığı yerdir. Burada belirtilen sınıf ismi ile birlikte ayrıştırıcı sınıfının ismi aynıdır. Ayrıca bu blokta yazılan kodlar değiştirilmeden oluşturulacak Java sınıfının içinde yer alır.

Üçüncü blok kelimesel özelliklerin yer aldığı kısımdır. Bu kısımda terminal değerler ve karakter ya da karakter dizisi şeklinde tanımlamalar yer alır. Son blokta ise sözdizimsel kurallar ve genişletilmiş Java metotları bulunur.

1.4.2.1. Özellikler Bloğu

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

```
options{
LOOKAHEAD = 2;
}
```

özellik kodu ile birlikte ayrıştırma esnasında karar verebilmek için en az iki kelime gerektiği belirtilir.

1.4.2.2. Temel Sınıf Tanımlama Bloğu

Bu blok tamamen Java kodlarından oluşur ve ayrıştırma esnasında dikkate alınmaz. Burada yazılan kodlar, gramer dosyasının JavaCC ile derlenmesi esnasında kontrol edilmez. Yanlış kodlar yazılmış olsa dahi bu kısım bir sonraki aşamaya değişiklik yapılmadan aktarılır.

Aşağıda bu bloğa örnek bir yapı verilmiştir. Örnekte [11] PARSER_BEGIN ve PARSER_END arasında kalan ve temel sınıf özelliklerinin bulunacağı yer sınıf tanımlama bloğudur.

```
PARSER_BEGIN(ExpressionEvaluator)
package ieee;
import java.util.*;
import java.io.*;
public class ExpressionEvaluator {
public static void main(String[] args) throws ParseException {
…
}
private static Stack stack = new java.util.Stack();
private static Hashtable state = null;
public static boolean popBoolean() throws ClassCastException {
```

```
Boolean a = (Boolean) stack.pop();
if (a == null) return false; else return a.booleanValue();
}
…..
}
PARSER_END(ExpressionEvaluator)
```

1.4.2.3. Kelimesel Özellikler Bloğu

Bu blokta terminallerin tanımları yapılır. Burada tanımlanan terminaller JavaCC'ye ayrıştırma işleminde bulunması gereken anlamlı en kısa kelimelerin neler olduğunu ifade eder. Aşağıda kelimesel özellikler bloğuna örnek [11] gösterilmiştir.

```
TOKEN :
{
< ASSIGN: "=" > | < GT: ">" > | < LT: "<" > | < EXCL: "!" >
| < TILDE: "~" > | < EQ: "==" > | < LE: "<=" >| < GE: ">=" >
| < NE: "!=" > | < OR: "||" > | < AND: "&&" > | < PLUS: "+" >
| < MINUS: "-" > | < MULT: "*" > | < SLASH: "/" >
}
TOKEN :
{
< IDENTIFIER: <LETTER> (<LETTER>|<DIGIT>)* >
|
< #LETTER: ["\u0041"-"\u005a", "\u005f", "\u0061"-"\u007a"] >
|
< #DIGIT:["\u0030"-"\u0039"] >
}
```

1.4.2.4. Sözdizimsel Kurallar Bloğu

Bu blok içerisine gramerde bulunan her bir kural için Java metotlarına benzer kodlar yazılır. Ayrıca blok içinde Java kodları gerek duyulduğu yere, küme parantezi içine alınarak yazılabilir. Genişletilmiş BNF gösterimini ve Java metotlarını birleştirmesi, JavaCC'nin güçlü yönlerinden biridir.

JavaCC gramer içinde tanımlanmış her bir kural için Java'da bir metot üretir. Metodun adı kuralın başında bulunan non-terminal ile aynı isimdedir. Metot isminden hemen sonra, JavaCC'nin küme parantezleri içinde tanımlamalara izin verdiği bir blok gelir. Bu blok içinde gerçekleştirilen tanımlamalar metot içinde geneldir. Bu yapının ardından non-terminal, terminal ya da non-terminallerle genişletilmiş BNF şeklinde tanımlanır. Tanımlamanın bu kısmında ihtiyaç duyuluyorsa yine Java metotları yazılabilir.

Aşağıda sözdizimsel kuralların tanımlandığı bloğa örnek [11] verilmiştir:

```
void Expression() :{}
 {
  AndExpression() ( <OR> AndExpression()
  {
          try {
                  boolean a = popBoolean();
                  boolean b = popBoolean();
                  stack.push(new Boolean(b || a));
              }
          catch (ClassCastException ex) {
                  System.out.println("Non boolean operand supplied for
                                                  <OR> operator");
                  throw new ParseException();
          }
})*
}
```

AndExpression ifadesinden sonra yazılan Java kodları, her defasında programcı tarafından daha önceden tanımlanmış bir yığın içinden iki değer çekip bunların OR işlemine tabi tutulmasını sağlamaktadır.

Yukarıda örnek verilmiş kod parçası sadece tek bir kurala aittir ve diğer kurallar da buna benzer şekilde oluşturulmak zorundadırlar.

1.4.3. Java Kodu Üretimi

JavaCC gramer dosyası oluşturulduktan sonra, ayrıştırıcı sınıflarının oluşturması basit bir şekilde gerçekleştirilir. JavaCC, Windows ve Unix ortamında çalışabilen bir betik (script) olarak geliştirilmiştir. Komut satırından;

javacc dosya_adi.uzantısı

komutu koşturularak ayrıştırıcı kodları üretilebilir.

1.5. Haskell

Haskell yorumlamalı bir dildir ve fonksiyonel programlama çevrelerinin en yeni ve güçlü temsilcisidir. Son 10 yılda üniversitelerde olduğu kadar endüstride de ilgi gösterilen diller arasına girmiştir [15].

Haskell, yüksek dereceli fonksiyonları ve tembel hesaplamayı (lazy evaluation) destekleyen fonksiyonel bir programlama dilidir [6]. Tembel hesaplamalı (lazy evaluation) dillerde ifadeler gerekmedikleri sürece hesaplanmazlar. Bu nedenle tembel diye nitelendirebileceğimiz bu diller geleneksel dillerden tümüyle farklı bir karaktere sahiptirler.

Tembel hesaplamanın karşıtı olarak, çoğu programlama dilinin (C, C++, Java gibi) hesaplama tekniği olan kesin belirtimli hesaplama bulunmaktadır. Kesin belirtimli dillerde hesaplama sonucunun önemli ya da önemsiz olduğuna bakılmaksızın her bir ifade hesaplanır. Bu durumun açıklaması için aynı işi yapan (hatalı şekilde oluşturulmuş dizinin boyutunu ekrana bastıran) Haskell ve Java programlama dillerinde yazılmış iki kod bloğu aşağıda verilmiştir.

Haskell kodu;

```
yazdir::Int
yazdir = length [(2 + 1),(5 `div` 0), (3 * 4)]
```

Java kodu;

```
void yazdir (){
  Integer[] arr = { (2 + 1),(5 / 0),(3 * 4) };
  System.out.println(arr.length);
}
```

Program, ilgili diziyi oluşturarak bu dizinin uzunluğunun ekrana bastırılmasını gerçekleştirmektedir. Haskell tembel hesaplama (lazy evaluation) mantığı ile çalışan bir dil olduğu için, length fonksiyonu içinde değerleri hesaplamadan sonucu ekrana bastırabilir.

Java kodunda ise dizi oluşturulurken (5 / 0) ifadesi hesaplanmaya çalışılır. İfadenin sonucu tanımsız olduğu için program, sıfıra bölme hatası verip icrayı sonlandıracaktır.

Haskell, modüller halinde program yazmayı destekler. Bu sayede yazılan kodlar geleneksel dillerde yazılan kodlara göre daha okunabilir, daha anlaşılabilir ve daha kısa biçimdedir.

Haskell yan etkiye (side effects) izin vermediğinden dolayı saf (pure) bir dil olarak nitelendirilir. Programlama terimi olarak ‘yan etki’ genel (global) bir değişkeni etkilemek ya da genel (global) değişkenin etkisinde kalmak anlamında kullanılır. Bu durum sorunlara neden olabileceğinden, olabildiğince uzak durulması gereken bir durumdur.

Tembel hesaplamalı bir dilin saf olması kaçınılmazdır. Fakat bunun tersi doğru olmayabilir [10].

Genel değişkenlere (örneğin ekrana, paralel porta vb.) erişimi tamamen engellemek, bir programlama dilini tamamen işlevsiz kılacağı için Haskell’de bu tip işlemler “monad” olarak adlandırılan bir yapı ile gerçekleştirilir. Kullanıcı etkileşimini sağlayan bir Haskell programı örneği aşağıda verilmiştir.

```
tahminet num = do
                    putStrLn "Tahmininizi giriniz:"
```

```
tahmin <- getLine
if ( (read tahmin) < num )
        then do
                putStrLn "Dusuk tahmin!"
                tahminet num
        else if ( (read tahmin) > num )
                        then do
                                putStrLn "Yuksek tahmin!"
                                tahminet num
                        else do putStrLn "Dogru tahmin!"
```

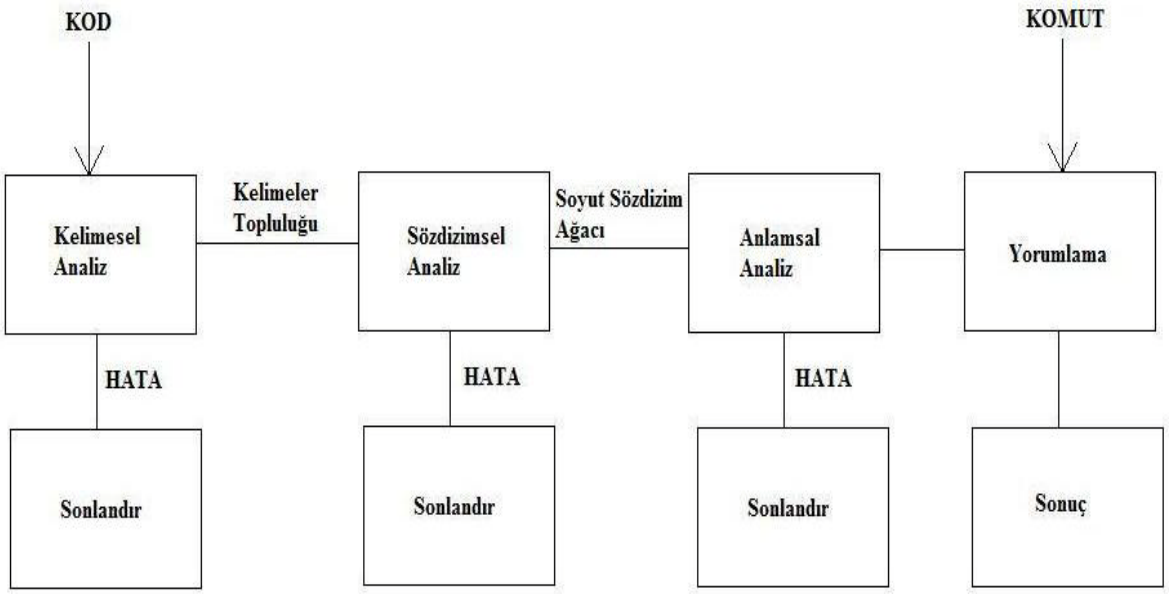
2. YAPILAN ÇALIŞMALAR, BULGULAR VE İRDELEME

2.1. Giriş

Çalışmada, Haskell'den türetilen basit bir fonksiyonel programlama dilinin (MiniHaskell) tanımlanması ve bu dilde geliştirilen kaynak programların web-tabanlı bir yorumlayıcı ile çalıştırılabilmesi amaçlanmıştır. Bu sistemde yorumlanacak kaynak kod internet ortamından alınacak ve yorumlama sonuçları kullanıcıya gösterilecektir.

İnternet üzerinden alınan kaynak kod, Kelimesel Analiz, Sözdizimsel Analiz ve Anlamsal Analiz aşamalarından geçirilerek ilgili fonksiyonel dilin gramer kurallarına uygunluğu yönünden değerlendirilir. Bu aşamaların herhangi birinde karşılaşılacak hata, kodun bir sonraki aşamaya geçmesini engelleyecektir. Kodun hatasız bir şekilde aşamaları tamamlaması durumunda ise kullanıcıdan komut istenip yorumlama işlemi gerçekleştirilecektir.

Yorumlama sisteminin web-tabanlı olarak çalışabilmesini sağlamak amacıyla Java dilinin kullanılmasına ve yorumlayıcının bir applet uygulaması biçiminde gerçeklenmesine karar verilmiştir.



Şekil 2.1. Web-tabanlı yorumlayıcı aşamaları

Günümüzde programlama dillerinin derleme süreçlerini otomatikleştiren birçok mekanizma geliştirilmiştir. Bu çalışmanın gerçekleştirilmesinde de Java dilinde kaynak kod üretebilen JavaCC (Java Compiler Compiler), JTB (Java Tree Builder) araçlarından yararlanılmıştır.

Çalışma içerisinde kelimesel ve sözdizimsel analizin gerçekleştirilmesinde JavaCC, anlamsal analiz, soyut ağaç üretimi ve yorumlama işlemlerinin gerçekleştirilmesinde JTB mekanizmaları kullanılmıştır.

2.2. Web-Tabanlı Yorumlayıcı

Çevirici programlar yazılırken önce çevirisi yapılacak kaynak dilin grameri belirlenir. Ardından bu gramer üzerinden gidilerek çeviri işlemi gerçekleştirilir.

Çevirici programlar olan yorumlayıcılar, derleyicilerle karşılaştırıldıklarında daha basit yapılı görünmelerine rağmen, küçük ölçekli bir dilin tüm işlev seti göz önüne alındığında çok karmaşık kodlama işlemleri gerektirebilirler. Hatta bazı dil işlevlerinin yazılım ile yürütülmesi mümkün olamayabilmektedir. Bu nedenle günümüzde geliştirilen yorumlayıcılar programlama dillerinin küçük bir işlev setini kapsamaktadır.

Bu yaklaşımdan yola çıkarak fonksiyonel programlama dili olan Haskell'in kısıtlı bir işlev setinin yorumlayıcısı gerçekleştirilmiştir.

Yorumlama için gramer belirlendikten sonra, bu gramere uygun;

- Kelimesel Analiz,
- Sözdizimsel Analiz,
- Somut Sözdizim Ağacı Oluşturma,
- Anlamsal Analiz,
- Yorumlama

işlemleri gerçekleştirilir.

2.2.1. Kelimesel Analiz

Yorumlayıcının ilk aşaması olan kelimesel analizde yorumlanacak kodun gramere uygun kelimeler içerip içermediğinin kontrolü yapılır. Bu aşamada programlama dilinin içerdiği tüm anahtar kelimeler ve yapılar tanımlanmıştır. Yorumlanacak kaynak kodda bu

tanımlamalar dışında bir yapı ile karşılaşıldığında sistem hata vererek analiz işlemi sonlandırılır.

MiniHaskell programlama diline için belirlenen gramerde kullanılan anahtar kelimeler ve yapılar şunlardır:

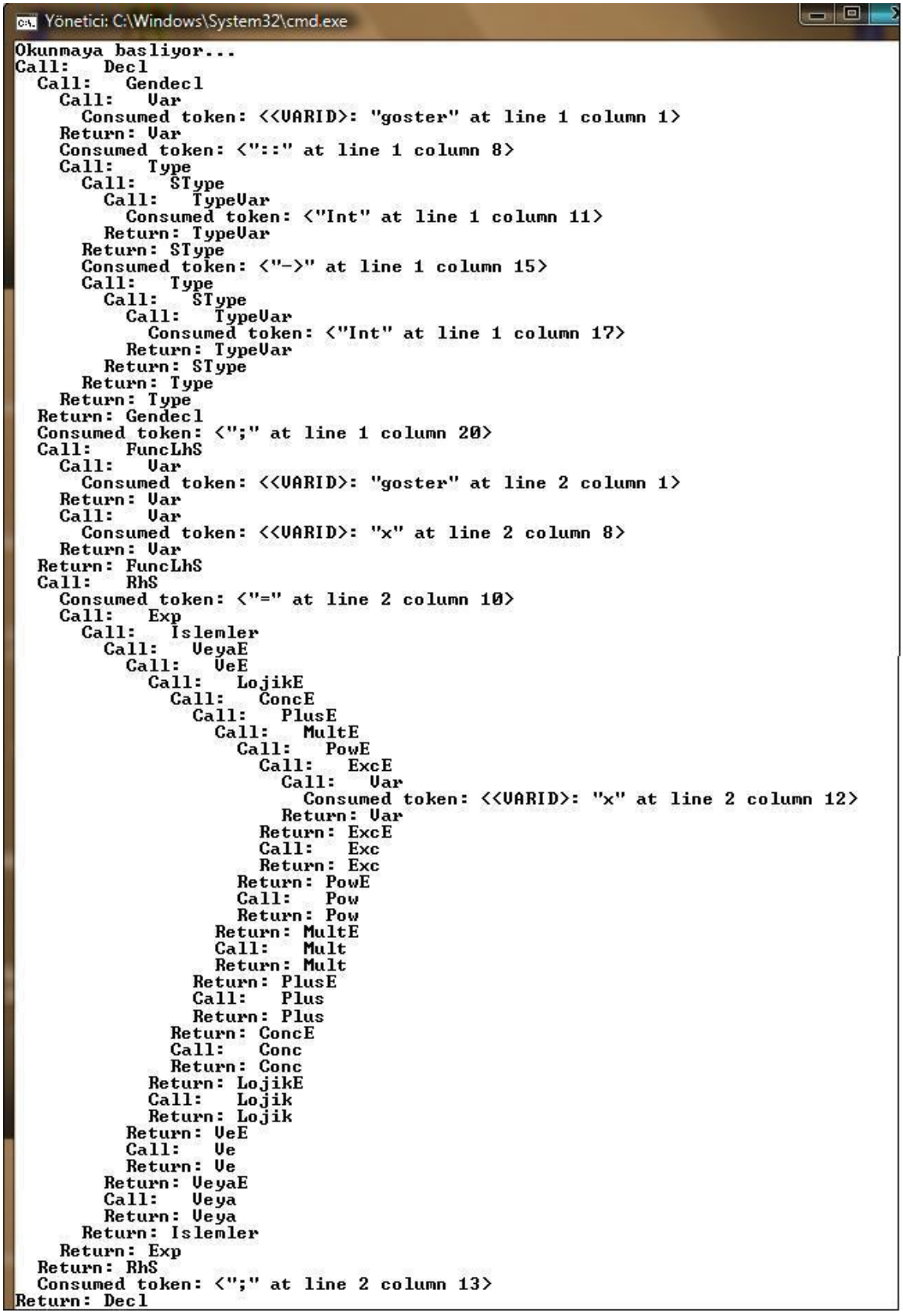
CASE : "case" *CLASS : "class"* *DATA : "data"*
DEF : "default" *DERIVING : "deriving"* *DO : "do"*
ELSE : "else" *IF : "if"* *IMPORT : "import"*
IN : "in" *INFIX : "infix"* *INFIXL : "infixl"*
INFIXR : "infixr" *INSTANCE : "instance"* *LET : "let"*
MODULE : "module" *NEWTYPE : "newtype"* *OF : "of"*
THEN : "then" *TYPE : "type"* *WHERE : "where"*
INTEGER : "Int" *STRING : "String"* *BOOLEAN : "Bool"*
DPOINT : ".." *DCOLON : "::"* *COLON : ":"*
POWER : "^" *NOTEQUAL : "/="* *EQUAL : "=="*
ASSIGNMENT : "=" *BACKSLASH : "\\"* *DOUBLEOR : "||"*
SMALLDASH : "<-" *DASHBIG : "->"* *EQBIG : "=>"*
DEXCLAMATION : "!!" *DOUBLEPLUS : "++"* *ONEOR : "|"*
BIGOREQUAL : ">=" *SMALLOREQUAL : "<="* *UNDERDASH : "_"*
AT : "@" *TILDA : "~"* *EXCLAMATION : "!"*
SHARP : "#" *DOLAR : "$"* *PERCENT : "%"*
AMPERSAND : "&" *TIMES : "*"* *PLUS : "+"*
DOT : "." *SLASH : "/"* *LITTLE : "<"*
BIG : ">" *QUESTIONM : "?"* *MINUS : "-"*
LPAREN : "(" *RPAREN : ")"* *COMMA : ","*
SEMICOLON : ";" *LSQPAREN : "["* *RSQPAREN : "]"*
QUOTE : " ' " *LBRACE : "{"* *RBRACE : "}"*
SMALL : ["a"-"z"] *LARGE : ["A"-"Z"]* *DIGIT : ["0"-"9"]*
DOUBLEAMPRSND : "&&"
*VARID : (SMALL) ((SMALL) (LARGE) (DIGIT))**
BOOL : "True" | "False" *INT : (DIGIT)+*
DQUOTE : "\""
GRAPHIC : (~["\""])+

STR : (DQUOTE) (GRAPHIC) (DQUOTE)

Bu yapılar JavaCC mekanizması kullanılarak kontrolü gerçekleştirecek Java koduna dönüştürülmüştür. Elde edilen bu kod kelimesel analizi gerçekleştirecektir.

Şekil 2.2.'de aşağıda bulunan Haskell programlama dilinde yazılmış örnek kodun kelimesel analiz çıktısı gösterilmiştir. Örnek kod içinde bulunan tanımlı kelimeler program tarafından belirlenmektedir. Bu kelimeler "Consumed Token:" etiketi ile gösterilmiştir.

```
goster :: Int -> Int
goster n = n
```



Şekil 2.2. Örnek kod için kelimesel analiz çıktısı

2.2.2. Sözdizimsel Analiz

MiniHaskell dilini tanımlayabilmek için öncelikle Haskell programlama dilinin gramer yapısı incelenmiştir. Haskell'in gramer kuralları arasında çok karmaşık kodlama gerektirecek ve kontrolü zorlaştıracak kısımları içermeyecek şekilde MiniHaskell dilinin gramer yapısı oluşturulmuştur.

Kısıtlamalar sonucunda oluşturulan yeni gramerin bağlam bağımsız gramer şeklindeki gösterimi aşağıdaki gibidir (tümü büyük harf olan yapılar terminalleri, diğer yapılar ise non-terminalleri ifade etmektedir):

*Decl ::= (FuncLhS RhS SEMICOLON Gendecl SEMICOLON)**

Gendecl ::= (Var)+ DCOLON

Type Var ::= VARID

Type ::= SType (DASHBIG Type)?

SType ::= TypeVar

LPAREN Type (COMMA Type) RPAREN*

LSQPAREN TypeVar RSQPAREN

TypeVar ::= INTEGER BOOLEAN STRING

*FuncLhS ::= Var (Var LPAREN Var COLON Var RPAREN UNDERDASH Literal)**

Apat ::= Var Literal GCon UNDERDASH LPAREN Pat (COMMA Pat) RPAREN LSQPAREN Pat (COMMA Pat)* RSQPAREN Islemler*

GCon ::= LPAREN RPAREN LSQPAREN RSQPAREN

Literal ::= INT STR BOOL

Pat ::= MINUS Literal GCon (Apat)+ Apat

Islemler ::= VeyaE Veya

*Veya ::= (DOUBLEOR VeyaE Veya)**

VeyaE ::= VeE Ve

*Ve ::= (DOUBLEAMPRSND VeE Ve)**

VeE ::= LojikE Lojik

*Lojik ::= ((EQUAL NOTEQUAL LITTLE SMALLOREQUAL BIG BIGOREQUAL) LojikE Lojik)**

LojikE ::= ConcE Conc

*Conc ::= ((DOUBLEPLUS DCOLON) ConcE Conc)**

*ConcE ::= PlusE Plus Plus → ((PLUS MINUS) PlusE Plus)**

PlusE ::= MultE Mult

*Mult ::= ((TIMES SLASH) MultE Mult)**

MultE ::= PowE Pow

*Pow ::= (POWER PowE Pow)**

PowE ::= ExcE Exc

*Exc ::= (DEXCLAMATION ExcE Exc)**

ExcE ::= Var (Aexp)+ Var Literal LPAREN FuncLhS RPAREN LPAREN Islemler RPAREN

RhS ::= ASSIGNMENT Exp (GdRhS)+

Exp ::= Islemler MINUS Exp BACKSLASH (Apat)+ DASHBIG Exp LET RhS IN Exp IF Exp THEN Exp ELSE Exp CASE Exp OF (Alt)+ (Aexp)+

Aexp ::= Var GCon Literal LPAREN Exp (COMMA Exp) RPAREN LSQPAREN Exp (COMMA Exp)* RSQPAREN LSQPAREN Exp (COMMA Exp)? DPOINT Exp RSQPAREN LSQPAREN Exp ONEOR Qual (COMMA Qual)* RSQPAREN*

Qual ::= Pat SMALLDASH Exp Exp

Alt ::= Pat DASHBIG Exp

GdRhS ::= Gd ASSIGNMENT Exp

Gd ::= ONEOR Exp

Sözdizimsel analiz için gramer belirlendikten sonra bu yapının LL(k)'ya uygun hale getirilmesi gerekmektedir. Çünkü JavaCC, LL(k) algoritması ile çalışmaktadır. Bu nedenle gramerin, LL(k) algoritmasına uygun olabilmesi için, soldan yinelemeli durumları elimine edilmiştir ve sol faktörleme (left factoring) yapılmıştır.

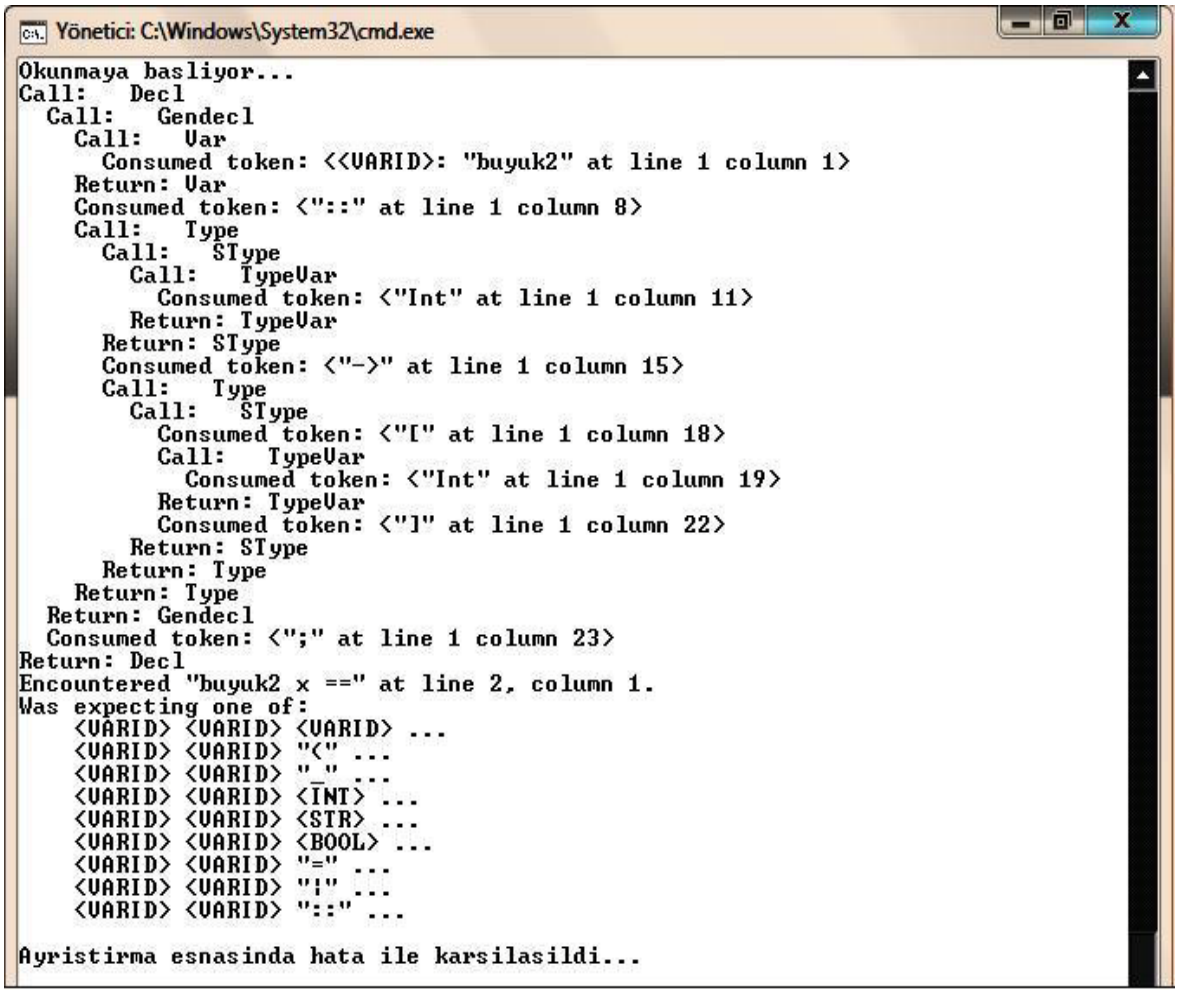
Tüm bu işlemler gerçeklendikten sonra gramer JavaCC ortamına aktarılmış ve MiniHaskell programlarının sözdizimsel analizini yapabilecek Java kodu üretilmiştir.

Şekil 2.3.'te aşağıda hatalı şekilde yazılmış olan MiniHaskell kodunun kelimesel ve sözdizimsel analiz sonucunu göstermektedir. Yorumlanacak kodda fonksiyonun x parametresinin hemen adından "=" kullanılması gereken yerde dilde bulunan bir yapı olmasına rağmen "==" yapısının kullanılması kodun kelimesel olarak doğru fakat sözdizimsel olarak hatalı olduğu anlamı taşımaktadır. Bu nedenle yorumlayıcı hata verip bir sonraki aşamaya geçmeyecektir.

Programda hatanın nerede ve hangi kelimede meydana geldiği “Encountered ” etiketi ile gösterilmektedir. Ayrıca hata tespit edilen ifadelerin yerine gramere uygun olarak gelebilecek ifadeler belirtilmiştir.

buyuk2 :: Int -> [Int]

buyuk2 x == [a | a <- [1..x], a > 2]



Şekil 2.3. Örnek kod için sözdizimsel analiz çıktısı

2.2.3. Somut Sözdizim Ağacı Oluşturma

Yorumlanacak kodun üçüncü aşamasında sözdizim ağacı oluşturulmuştur. Bu ağacın oluşturulmasındaki amaç daha sonraki aşamalar olan anlamsal analiz ve kod yorumlama işlemlerinin gerçekleştirilmesinde uygun bir yapı oluşturabilmek.

Bu aşamada JavaCC'nin yanı sıra JTB mekanizması devreye girmektedir. JTB'nin kullanılması ile birlikte gramer içinde tanımlanan tüm non-terminal ifadeler birer sınıf şekline dönüştürülmektedir. Böylece oluşturulan sözdizim ağacının her bir düğümünde daha önceden tanımlı bir nesne yer alacaktır. Bu nesneler düğümlerin kontrolünü daha kolay hale getirecektir.

JTB mekanizması iki farklı paket oluşturur. Bu paketler;

- Syntaxtree,
- Visitor

paketleridir.

2.2.3.1. Syntaxtree Paketi

Bu paket içerisinde JTB, gramerdeki tüm non-terminallar için birer sınıf meydana getirir. Bu sınıfların yanı sıra, bu sınıflarda kullanılan altı temel sınıf ve tüm sınıfların gerçeklediği iki arayüz oluşturur.

```
package syntaxtree;
public interface Node extends java.io.Serializable {
  public void accept(visitor.Visitor v);
  public Object accept(visitor.ObjectVisitor v, Object argu);
}
```

Yukarıdaki Node arayüzü, syntaxtree paketi içinde bulunan çoğu sınıf tarafından gerçeklenir. Node arayüzünü gerçeklemeyen diğer sınıflar (NodeList, NodeListOptional, ve NodeSequence sınıfları) ise NodeListInterface arayüzünü gerçekler. Temel olarak iki fonksiyondan oluşan bu arayüz, bu fonksiyonlar sayesinde ağaç içerisinde bir alt seviyeye geçişi sağlarlar.

NodeListInterface arayüzü Node arayüzü miras alınarak tanımlanır. Bu sınıfın Node arayüzünden farkı birden fazla düğümü aynı anda ifade edebilmesidir.

```
package syntaxtree;
import java.util.*;
public class NodeToken implements Node {
```

```
  public NodeToken(String s) { … }
  public NodeToken(String s, int kind, int beginLine, int beginColumn, int
endLine, int endColumn) { … }
  public NodeToken getSpecialAt(int i) { … }
  public int numSpecials(){ … }
  public void addSpecial(NodeToken s) { … }
  public void trimSpecials(){ … }
  public String toString()    { … }
  public String withSpecials(){ … }
  public void accept(visitor.Visitor v) {
    v.visit(this);
  }
  public Object accept(visitor.ObjectVisitor v, Object argu) {
    return v.visit(this,argu);
  }
  public String tokenImage;
  public Vector specialTokens;
  public int beginLine, beginColumn, endLine, endColumn;
  public int kind;
}
```

Yukarıda tanımlanan NodeToken sınıfı ile birlikte terminal yapılar ifade edilir. İçinde bulunan fonksiyonlar ve değişkenler yardımı ile terminal yapı hakkında veriler tutulur. NodeToken sınıfında da bulunan ***accept()*** fonksiyonları Node arayüzünde olduğu gibi ağacın alt seviyelerine geçiş için kullanılır.

JTB ile otomatik üretilen diğer temel sınıflar da benzer yapıda olup ifade ettikleri yapılara uygun fonksiyon ve değişkenlere sahiptirler.

```
package syntaxtree;
/**
 * Grammar production:
 * f0 -> <ONEOR>
 * f1 -> Exp()
 */
```

```
public class Gd implements Node {
  public NodeToken f0;
  public Exp f1;
  public Gd(NodeToken n0, Exp n1) {
    f0 = n0;
    f1 = n1;
  }
  public Gd(Exp n0) {
    f0 = new NodeToken("|");
    f1 = n0;
  }
  public void accept(visitor.Visitor v) {
    v.visit(this);
  }
  public Object accept(visitor.ObjectVisitor v, Object argu) {
    return v.visit(this,argu);
  }
}
```

Yukarıdaki kod bloğu ise JTB'nin gramer içindeki herhangi bir non-terminali için ürettiği sınıf kodudur. Bu kod içinde görünen açıklama, bu sınıfın gramer içinde hangi kurala karşı geldiğidir. Açıklama kısmında yer alan f0, f1, f2, … vb. değişkenler ağacın alt düğümlerini ifade etmektedir. Burada Gd düğümünün iki alt düğümden (f0,f1) oluştuğu söylenebilir.

Üretilen diğer sınıflarda olduğu gibi accept() metodu bu sınıfta da yer almaktadır. Bu metod sayesinde Gd sınıfının sahip olduğu iki alt düğüme erişim sağlanır.

2.2.3.2. Visitor Paketi

JTB tarafından üretilen bir diğer paket olan Visitor paketinde ağacın düğümlerinde dolaşmayı sağlayacak sınıflar ve arayüzler yer almaktadır. Bu sınıflardan miras alınarak ağacın bütün düğümleri ziyaret edilebilir.

Visitor paketinde otomatik üretilen iki arayüz (Visitor, ObjectVisitor) ve bu iki arayüzden oluşturulmuş iki sınıf (DepthFirstVisitor, ObjectDepthFirst) bulunmaktadır.

DepthFirstVisitor sınıfında bulunan kod miras alınarak ağaç içinde dolaşma gerçekleştirilebilir, fakat geriye hiçbir veri döndürme imkanı yoktur. Ağacın herbir düğümünden değer geri döndürülmek isteniyorsa ObjectDepthFirst sınıfı kullanılmalıdır. Bu iki sınıfın birbirinden tek farkı geriye değer döndürüp döndürememesidir.

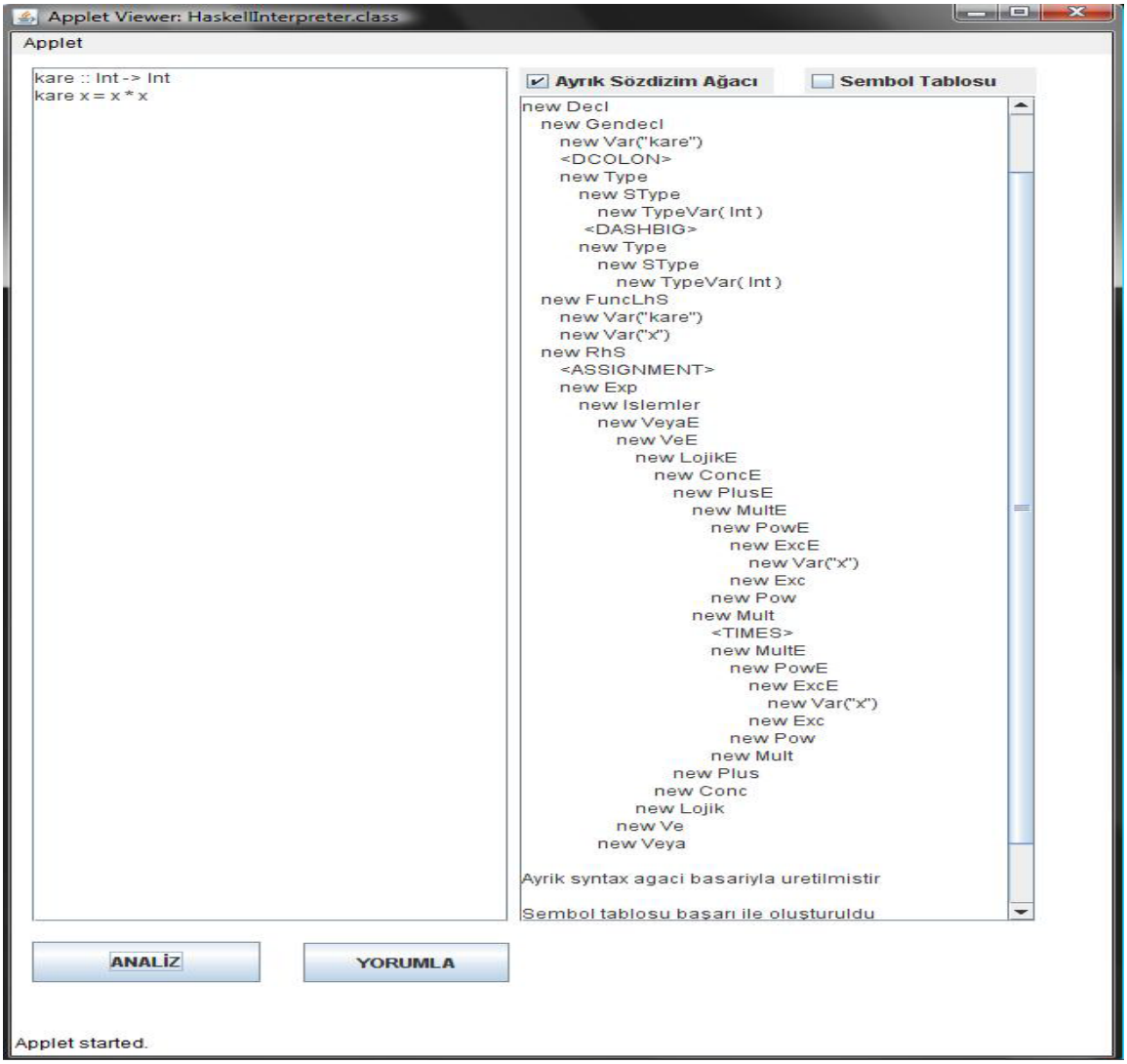
```
package visitor;
import syntaxtree.*;
import java.util.*;
public class DepthFirstVisitor implements Visitor {
  /**
   * f0 -> <VARID>
   */
  public void visit(Var n) {
    n.f0.accept(this);
  }
  /**
   * f0 -> <INT>
   *       | <STR>
   *       | <BOOL>
   */
  public void visit(Literal n) {
    n.f0.accept(this);
  }
…
}
```

DepthFirstVisitor sınıfında, ağacın düğümleri arasında dolaşabilme, her bir kural için ***visit()*** fonksiyonlarının tanımlanması ile gerçekleştirilebilmektedir. Bu fonksiyonun içinde ağacın alt düğümlerine geçişi sağlayan accept() fonksiyonu bulunmaktadır.

Yukarıdaki kod bloğu DepthFirstVisitor sınıfının sadece küçük bir kısmını teşkil etmektedir. DepthFirstVisitor sınıfı syntaxtree paketinde bulunan tüm sınıflar için ***visit()*** fonksiyonunu gerçeklemektedir.

Soyut sözdizim ağacı bu iki paketin kullanımı ile oluşturulur. DepthFirstVisitor sınıfı miras alınarak *AstPrint* sınıfı oluşturulmuştur. Bu sınıf, ağacın oluşumunu ve ağacın düğümlerini ekrana bastırmayı gerçekleştirir. Bunu yapabilmek için kelimesel ve sözdizimsel analiz sonucunda hata vermeyen Node nesnesinin accept() fonksiyonuna *AstPrint* sınıfından bir nesne argüman olarak verilir.

Şekil 2.4.'te, gösterimi kolay, basit bir Haskell programının somut sözdizim ağacı gösterilmiştir.



Şekil 2.4. Örnek Somut Sözdizim Ağacı

2.2.4. Anlamsal Analiz

Bu aşamada, kaynak kodun anlamsal olarak kontrolü gerçekleştirilir. Sözdizim ağacı oluşturulmuş kaynak kodun, düğümlerinde dolaşarak uygun veri tiplerinin kullanılıp kullanılmadığı kontrol edilir.

Anlamsal analizin gerçekleştirilebilmesi için öncelikle sembol tablosunun oluşturulması gerekir. MiniHaskell programlama dilinde yazılmış bir kodda bulunan her bir fonksiyonun kendisine ait bir sembol tablosu olmalıdır. Bu sembol tablolarında fonksiyonun aldığı parametrelerin türleri, fonksiyonun geri döndürdüğü tip bilgisi ve fonksiyon içinde kullanılan değişkenler ve türleri saklanmaktadır.

Sembol tabloları oluşturulduktan sonra, sembol tablosu içindeki veriler kullanılarak değişkenlerin uygun şekilde kullanılıp kullanılmadığı test edilir. Bu nedenle anlamsal analiz iki aşamada gerçekleştirilir;

- Sembol Tablosu oluşturma,
- Tip Kontrolü.

2.2.4.1. Sembol Tablosu Oluşturma

MiniHaskell programlama dilinde sembol tablolarının oluşturulabilmesi için fonksiyon bildirimlerinden ve fonksiyon tanımlamalarının sol tarafından (Left-Hand Side) faydalanılır.

Fonksiyon bildirimlerinden; fonksiyon parametrelerinin tür bilgileri ve fonksiyonun geri döndürdüğü tip bilgisi alınır. Fonksiyon tanımlamasının sol tarafından ise fonksiyonda kullanılacak değişkenler ve bunlara ait tip bilgileri alınır.

Sembol tablosu oluşturulurken JTB aracının ürettiği ObjectDepthFirst sınıfı kullanılır. Bu sınıf miras alınarak ağacın düğümlerinde gezmeye yarayan fonksiyonlar, ihtiyaca uygun şekilde yeniden yazılır.

Ağacın düğümlerinde dolaşırken belirlenen değişkenler ve tip bilgileri sembol tablosunda saklanacaktır. Bu nedenle bu bilgileri saklayabilecek bir sembol sınıfı oluşturulmalıdır. Aşağıda bu işi gerçekleştirmek için oluşturulmuş Symbol sınıfının tanımlanması verilmiştir.

```
package environment;
import java.util.Vector;
import java.util.Dictionary;
import java.util.Enumeration;

public class Symbol{
    public String func;
    public Vector params = new Vector();
    public Dictionary dict = new java.util.Hashtable();
    public Dictionary dict2 = new java.util.Hashtable();
    public boolean check = false ;
```

```
        public void  degisken_ekle(String degisken, String tur, Integer satir) {
          String s = (String)dict.get(degisken);
          if (s==null){
           dict.put(degisken,tur);
           dict2.put(degisken,satir);
          }
          else{
           System.out.println("Satir " + satir + " \"" + degisken.trim() + "\":" + "
Fonksiyon arguman tekrari");
           System.exit(-1);
          }
        }
        public void param_ekle(String tur){
         params.add(tur);
        }
        public boolean degisken_kontrol(String degisken){
         Integer s = (Integer) dict2.get(degisken);
         if(s==null)
           return false ;
         else
           return true;
        }
      }
```

Symbol sınıfında yorumlanacak kodda belirtilen fonksiyonun adını saklamak için "func" değişkeni kullanılmıştır. Bu değişken anahtar özelliği taşımakta ve her fonksiyon için farklı değerde olmak zorundadır. Aynı isimli iki fonksiyon tanımlanması bu sayede engellenmiştir.

Fonksiyonun aldığı parametrelerin tip bilgilerinin ve geri döndürdüğü değerin tip bilgisinin tutulması için "params" vektörü kullanılmıştır. Vektör tipinden bir değişken kullanılarak fonksiyonun istenilen sayıda parametre kullanmasına olanak sağlanmıştır.

"dict" ve "dict2" hash tablosu değişkenleri ile fonksiyon içinde kullanılan değişkenler ve bu değişkenlere ait bilgiler tutulmuştur. "dict" tablosunun içinde değişken

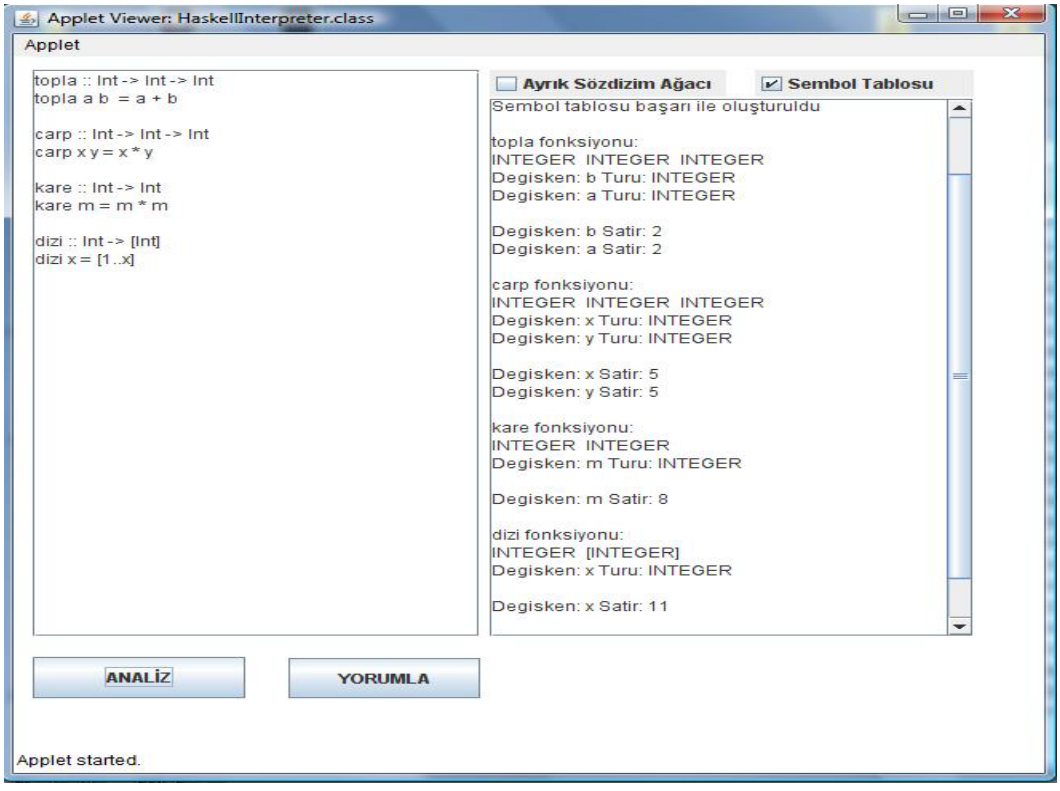
isimleri ve değişkene karşılık gelen tip bilgisi tutulmuştur. “dict2” tablosunda ise değişken isimleri ve kod içerisinde bulundukları satır numarası bilgisi tutulmuştur. Böylece hata tespit edildiğinde kodun kaçıncı satırında meydana geldiği gösterilebilecektir.

Sınıf içerisindeki degisken_ekle() ve param_ekle() fonksiyonları params, dict ve dict2 değişkenlerine veri ekleme işlemi için tanımlanmıştır. degisken_kontrol() fonksiyonu ise yorumlanacak kod içindeki fonksiyonda tanımlanmış değişkenin sembol tablosunda olup olmadığının kontrolünü gerçekleştirir.

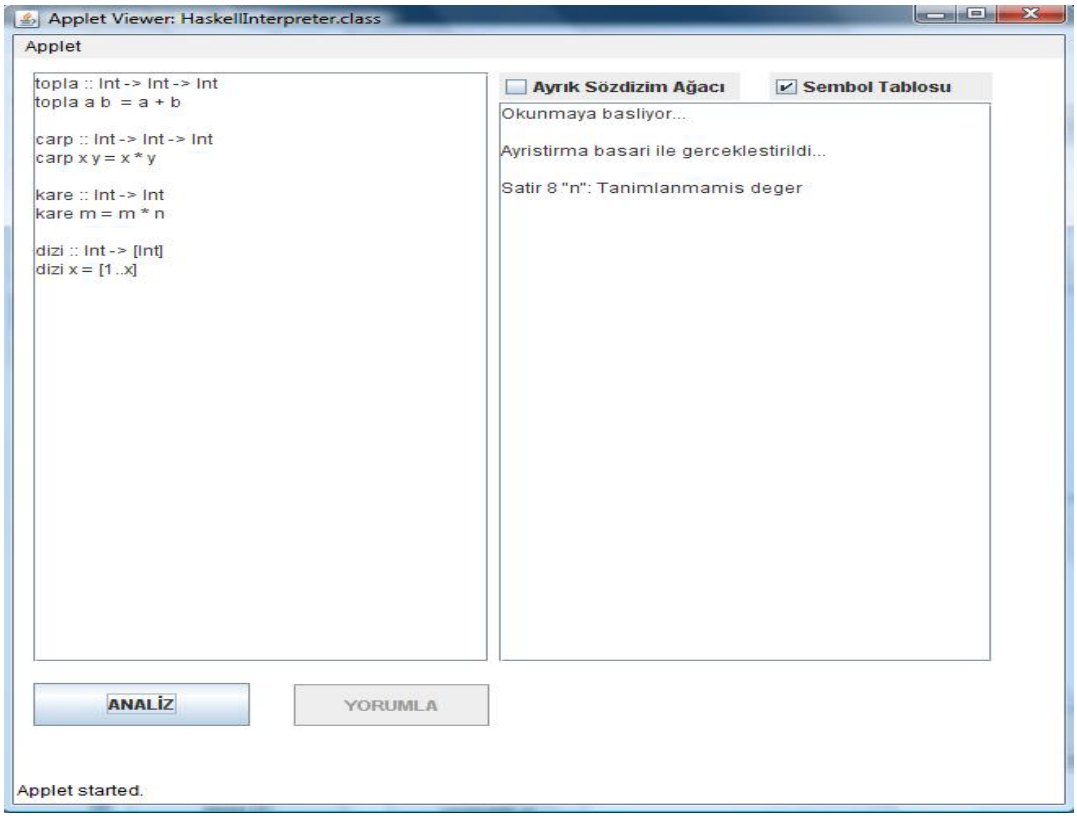
Yorumlanacak kod içinde tanımlanmış her bir fonksiyon için yeni bir Symbol nesnesi oluşturulur. Bu sayede farklı fonksiyonlar içinde tanımlanmış aynı isimli değişkenler birbirinden etkilenmez.

Şekil 2.5.’te, hatasız şekilde tanımlanmış örnek bir Haskell programının sembol tablosu analizi gösterilmiştir. Hatasız şekilde yazılmış kodun sembol tablosunda bulunan veriler şekil 2.5.‘te yer almaktadır.

Şekil 2.6.‘da ise hatalı şekilde tanımlanmış bir Haskell programının sembol tablosu analizi gösterilmiştir. Hata olarak kod içerisinde 8. satırda bulunan ancak tanımlanmadığı halde kullanılmaya çalışılan bir değişkenin (n değişkeni) var olduğunu belirtmektedir.



Şekil 2.5. Hatasız Haskell kodu sembol tablosu analizi



Şekil 2.6. Hatalı Haskell kodu sembol tablosu analizi

2.2.4.2. Tip Kontrolü

Bu aşama oluşturulan sembol tablosu yardımı ile ifadelerin tiplerine bağlı olarak doğru biçimde kullanılıp kullanılmadığını kontrol eder. Tip kontrolü genellikle fonksiyon tanımlamalarının sağ tarafı (Right- hand Side) ile gerçekleştirilir.

Bu aşamada yorumlanacak kod içindeki ifadelerin beklenen tip değerleri ile sembol tablosundaki gerçek tip bilgileri karşılaştırır. Uygun olmayan durumlarda tip kontrol işlemi durdurulur ve hata bildirimi yapılır.

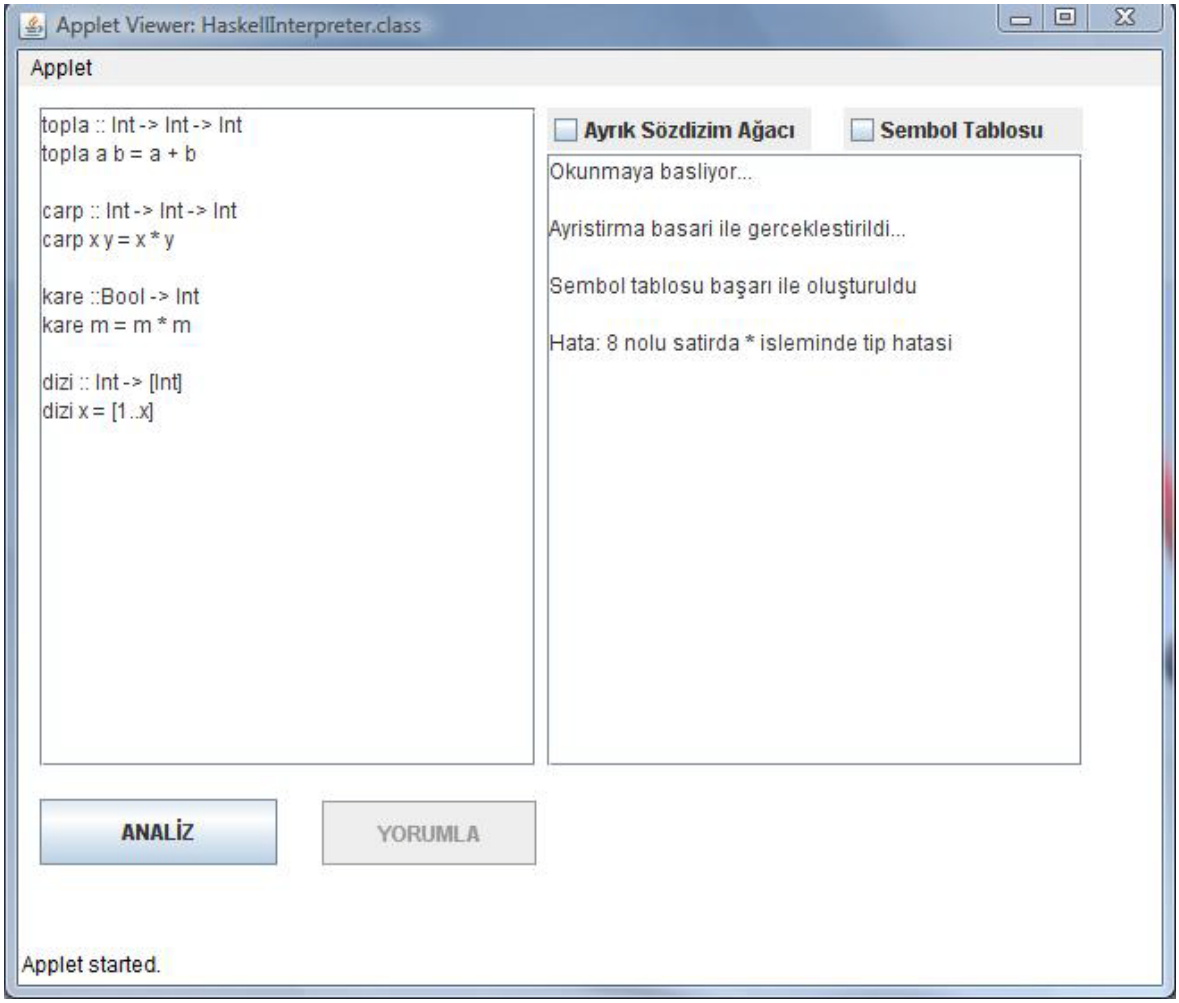
Tip kontrolünü gerçekleştirmek için ObjectDepthFirst sınıfı miras alınarak TypeCheck sınıfı oluşturulmuştur. ObjectDepthFirst sınıfından miras alınmasının nedeni sözdizim ağacındaki düğümlerden, tip bilgisinin geriye döndürülmek zorunda olunmasıdır.

Aşağıdaki kod bloğu toplama işlemi için gerçekleştirilen tip kontrolünü göstermektedir. Toplama işlemi için gereken kontrol, işlemin iki tarafının INTEGER tipinden olmasıdır.

```
public Object visit(ConcE n, Object argu) {
  Object _ret=null,_ret1=null;
  _ret = n.f0.accept(this, argu);
  _ret1 = n.f1.accept(this,argu);
  if(_ret1!=null){
      String text = (String)_ret ;
       if(!(text.substring(0 , text.indexOf(" ")).equals("INTEGER"))){
           System.out.println("Hata: " + text.substring(text.indexOf(" ")+1,text.length()) + " nolu satirda "+ sym +" isleminde tip hatasi");
           System.exit(-1);
           }
       else{
           _ret = (text.substring(0 , text.indexOf(" "))) + " " + text.substring(text.indexOf(" ")+1,text.length());
           }
  }
  return _ret;
}
```

TypeCheck sınıfı bütün kurallar için ağacın her düğümünde yukarıdaki koda benzer kontroller gerçekleştirmektedir.

Şekil 2.7.'de anlamsal olarak hatalı şekilde tanımlanmış MiniHaskell programının analizini gösterilmiştir.



Şekil 2.7. Örnek MiniHaskell kodunun anlamsal analizi

Yorumlanacak örnek programın "kare" fonksiyonunda, çarpma işlemi için INTEGER tipinden değişkenler beklenirken "m" değişkeninin BOOLEAN tipinde olması bir anlamsal hata oluşturacaktır.

2.2.5. Yorumlama

Hatası bulunmayan kaynak kodun son aşaması burada gerçekleştirilir. Bu aşamada verilen komuta göre kaynak kod yorumlanır ve sonuç üretilir. Bunu yapabilmek için kullanıcıdan alınacak komutun doğruluğu test edilmeli ve ardından komut içindeki değerler kullanılarak hesaplamaya geçilmelidir.

İlk aşama olarak kullanıcıdan alınan komutun doğruluğu test edilmelidir. Bunun anlamı komutun kaynak içinde bulunup bulunmadığının kontrolü, verilen komutun

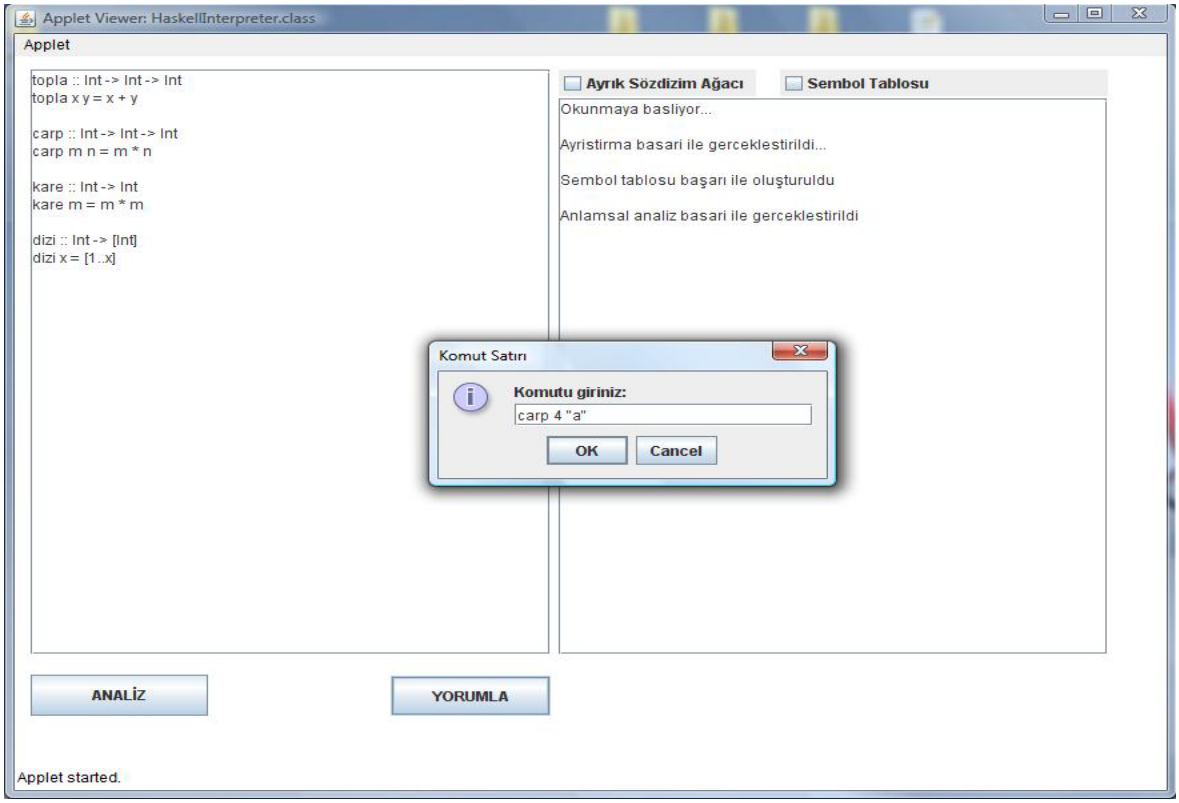
argümanlarının kaynak kodda bulunan fonksiyona uygunluğunun kontrolü işlemlerinin yapılması demektir. Böylece bu aşamada kontrolü yapılan komutun ve içerdiği argümanlarının ilerleyen aşamalarda hata üretmesi engellenmiş olur.

Bu kontrol public void komut_kontrol(String komut) fonksiyonu ile gerçekleştirildi. Fonksiyonla birlikte;

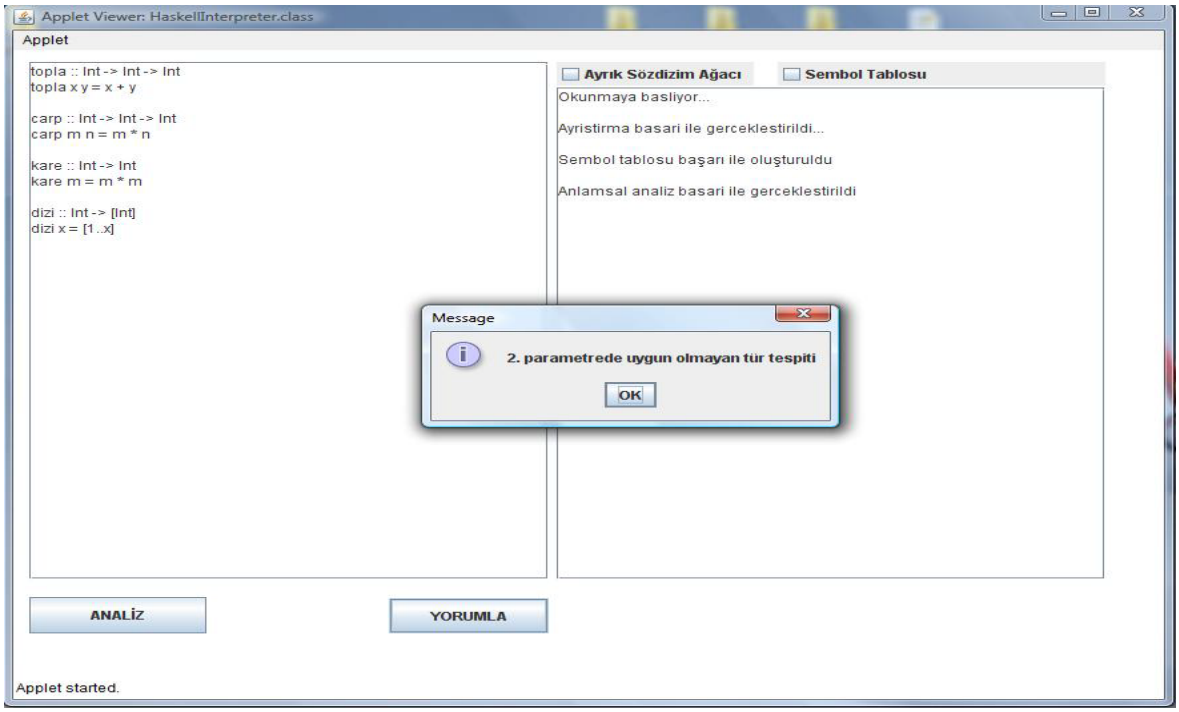
- Verilen komutun, kaynak kod içinde fonksiyon olarak bulunup bulunmadığı kontrolü,
- Verilen komutta bulunan argüman sayısının, kaynak kodda bulunan fonksiyonun argüman sayısı ile aynı olup olmadığının kontrolü,
- Verilen komutta bulunan argümanların türlerinin kaynak kodda bulunan fonksiyonun argüman türleri ile uyuşup uyuşmadığının kontrolü,

gerçekleştirilmiştir.

Şekil 2.8.(a)'da kaynak kodun analiz işlemi tamamlandıktan sonra kullanıcının komut giriş işlemi gösterilmiştir. Ancak girilen bu komutun ikinci parametresi kaynak kodda bulunan fonksiyon karşılığına uymamaktadır. Kaynak kod içinde *Int* olarak tanımlanmasına rağmen komutta bu değer *"a"* ile eşleştirilmeye çalışılmıştır. Dolayısıyla yanlış verilen bu komut kullanıcıya Şekil 2.8.(b)'de gösterildiği biçimde sunulmuştur.



(a)



(b)

Şekil 2.8.a)Hatalı komut girişi b)Hatalı komut uyarısı

Komutun kontrolünden sonra artık yorumlama işlemine geçilip sonucun kullanıcıya gösterilmesi gerekmektedir. Yorumlama işleminin gerçekleştirilmesi için Interpret sınıfı oluşturulmuştur. Yorumlama işleminde de ağacın düğümlerinde dolaşılacağı için ObjectDepthFirst sınıfı miras alınmıştır.

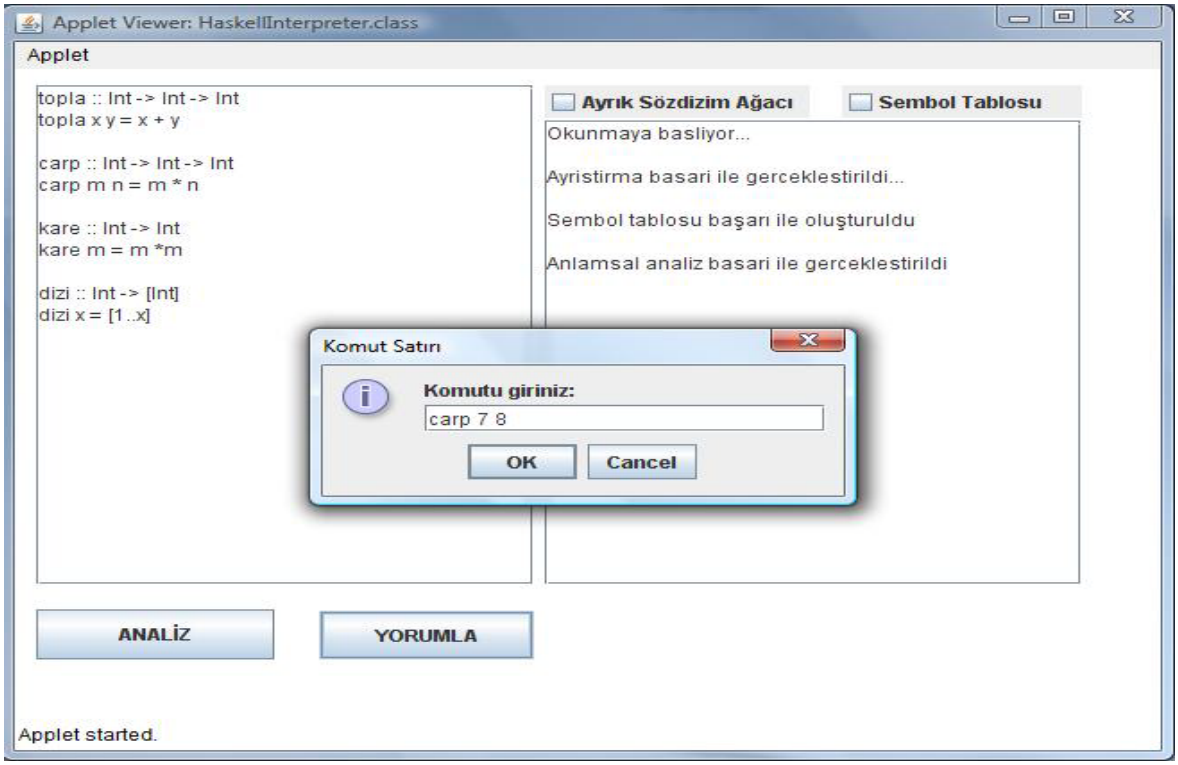
```
public class Interpret extends ObjectDepthFirst {
…
  public Object visit(ConcE n, Object argu) {
    Object _ret=null,_ret1=null;
    _ret = n.f0.accept(this, argu);
    _ret1 = n.f1.accept(this, argu);
    if(_ret1!=null){
      if(sym.equals("PLUS")){
        _ret = (Integer)_ret + (Integer)_ret1;
      }
      else if(sym.equals("MINUS")){
        _ret = (Integer)_ret - (Integer)_ret1;
      }
    }
    return _ret;
  }

public Object visit(Plus n, Object argu) {
  Object _ret=null,t1=null;
  for(Enumeration e = n.f0.elements();e.hasMoreElements();){
    NodeSequence ns = (NodeSequence)e.nextElement();
    NodeChoice nc = (NodeChoice)(ns.elementAt(0));
    if(nc.which==0){
      sym = "PLUS" ;
    }
    else if(nc.which==1){
      sym = "MINUS" ;
    }
```

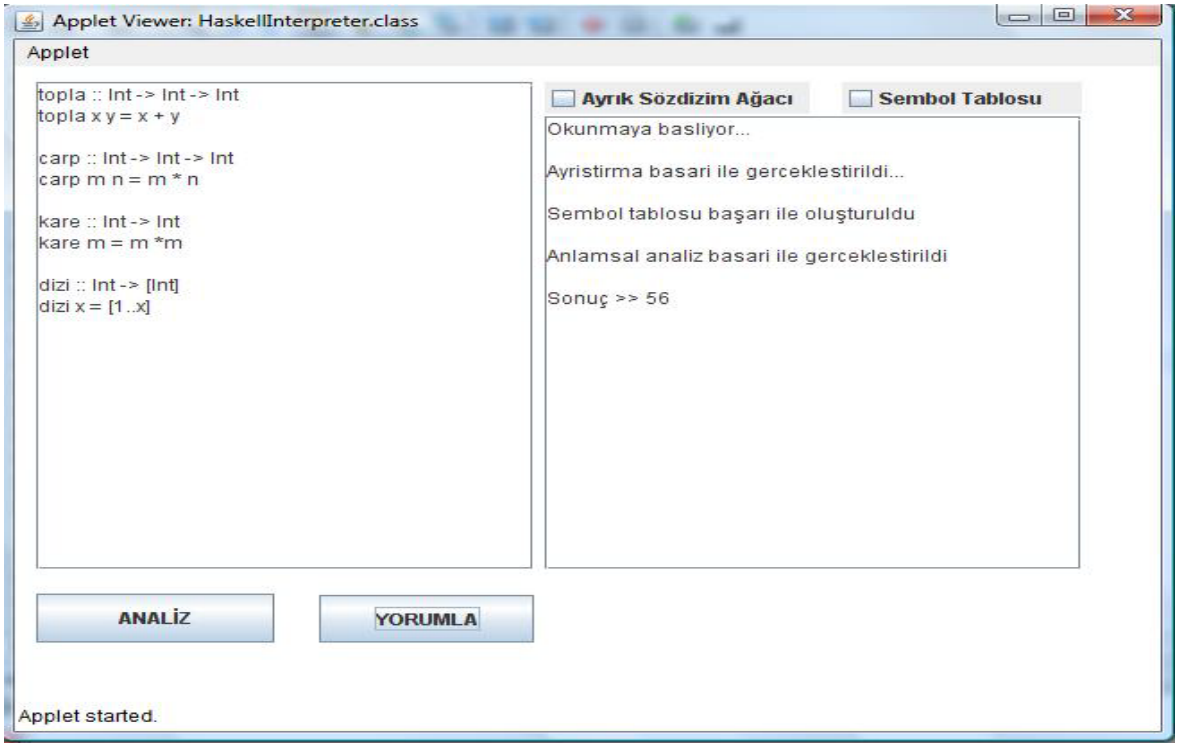
```
        ns.elementAt(0).accept(this, argu);
        t1 = ns.elementAt(1).accept(this, argu);
        _ret = ns.elementAt(2).accept(this, argu);
              if(_ret != null){
          if(sym.equals("PLUS")){
             _ret = (Integer)t1+(Integer)_ret;
          }
          else
          {
             _ret = (Integer)t1-(Integer)_ret;
          }
        }
        else{
          _ret = t1;
        }
        if(nc.which==0){
          sym = "PLUS" ;
        }
        else if(nc.which==1){
          sym = "MINUS" ;
        }
      }
      return _ret;
    }
...
  }
```

Yukarıda sadece toplama ve çıkarma işlemini gerçekleştiren yorumlama kodu bulunmaktadır. Bu düğümde gelen ifadelerin ağacın üst tarafına aktarılması ve işarete bağlı olarak işlemin gerçekleştirilmesi visit() fonksiyonunun tekrar yazılması (override) ile gerçekleştirilmiştir. Diğer tüm işlemlerde (düğümler) uygun şekilde tekrar yazılmıştır.

Şekil 2.9.'da hatasız şekilde oluşturulmuş bir kodun ve hatası olmayan bir komutun çıktısı gösterilmiştir.



(a)



(b)

Şekil 2.9. a) Hatasız komut girişi, b) Yorumlama Sonucu

2.2.6. MiniHaskell

Tüm gramerin kodlanması ve kontrolü zor olacağından Haskell'in bir alt seti işleme alınmıştır. Bu alt set aşağıdaki kısıtlamalarla oluşturulmuştur:

- Yeni veri tipi tanımlanması kısıtlandı. "data" anahtar kelimesi kullanılarak yeni veri türleri MiniHaskell'de oluşturulamaz.
- Haskell'de bulunan belirli veri türlerinin kullanımına izin verildi. Bu türler Boolean, Integer, String veri türleridir. Float, Double, Char gibi temel veri türlerinin MiniHaskell'de kullanımı kısıtlanmıştır.
- Haskell'de bulunan infix, prefix ve postfix tanımlamalarına MiniHaskell'de izin verilmemiştir.
- Yazılan ifadelerin infix formatta yazımına izin verilmiştir.
- Haskell'de "type" anahtar kelimesi ile oluşturulan veri gruplarına (tuple) izin verilmemiştir.
- Fonksiyon bildirimlerinin kullanıcı tarafından gerçeklenmesi zorunlu tutulmuştur.
- Haskell'de Monad diye adlandırılan (I/O) işlemleri gerçeklenmemiştir.
- Haskell'in kütüphane fonksiyonları tanımlanmamıştır.

3. SONUÇLAR

Bu çalışmada Haskell dilinden türetilen ve gramer kuralları belirlenen MiniHaskell programlama dili için Web tabanlı bir yorumlayıcı geliştirilmiştir.

Çalışmada önişlem olarak Haskell gramer kurallarının geniş bir setini kapsayan yeni bir gramer oluşturulmuştur. Gramerde yapılan bu kısıtlama ile gramer kurallarının sayısı azaltılarak yorumlama sürecinin kontrolü kolaylaştırılmıştır. Bu sayede daha verimli ve etkin bir yorumlayıcı elde edilmiştir.

Yapılan araştırmalarda bu konuya yönelik çalışmaların kısıtlı ve az olduğu gözlemlenmiştir. Özellikle geçmişte yapılan çalışmalarda çok küçük gramer setlerinin kullanımı göze çarpmaktadır.

Yorumlayıcı kodunun bir bölümü JavaCC ve JTB araçları yardımıyla üretilmiştir. Kelimesel analizin gerçekleştirilmesinde ve soyut sözdizim ağacının oluşturulmasında bu araçların ürettiği kodlardan yararlanılmıştır.

Yorumlayıcının her bir aşaması için yeni bir sınıf tanımlaması ile yeni bir kod bloğu yazılmıştır. Bu sayede çalışmanın, modüler olması sağlanmıştır. Her bir aşama bir önceki aşamadan, hata olmadığı sürece, gelecek kodu inceleyip bir sonraki aşamaya iletmektedir. Böylece her bir modül diğer modüle bağlı kalmadan çalışabilmektedir.

Sonuç olarak gerçekleştirilen yorumlayıcı, diğer çalışmalarla kıyasladığında içerik yönünden daha geniş bir çalışma olduğu görülmektedir. Çalışma sırasında geliştirilen koda dayanan 1 adet bildiri 1. Mühendislik ve Teknoloji Sempozyumu’ na sunulmuş ve bildiri kitapçığında basılmıştır.

4. ÖNERİLER

1. Çalışmamızda kısıtladığımız kurallar, ilerleyen çalışmalarda gramere eklenerek tam bir yorumlayıcı elde edilebilir.
2. Programın çalışmasını göstermek için üretilen somut sözdizim ağacı hız kaybına neden olmaktadır. Bu ağacın üretimi programdan çıkarılıp yorumlama süreci hızlandırılabilir.

5. KAYNAKLAR

1. Apel, A., W. ve Palsberg, J., Modern Compiler Implemenatation in Java, 2nd edition, Cambridge University Press, Cambridge, 2002.

2. Aho, A.,V., Sethi, R. ve Ullman, J., D., Compilers: Principles, Techniques and Tools, Addison-Wesley, Reading, Massachusetts, 1986.

3. URL: http://compilers.cs.ucla.edu/jtb/, JTB: Java Tree Builder, 22 Eylül 2008.

4. URL: http://javacc.dev.java.net/, JavaCC: Java Compiler Compiler, 22 Eylül 2008.

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

6. Thompson, S., Haskell: The Craft of Functional Programming, 2nd edition, Addison-Wesley, Reading, Massachusetts, 1999.

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

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

9. URL: http://www.cs.princeton.edu/~appel/modern/javal/JLex, Berk, E., JLex: A Lexical Analyzer Generator for Java, 15 Aralık 2007.

10. Hudak P., Hughes J., Jones S.P. ve Wadler P, A History of Haskell: Being Lazy with Class, The Third ACM SIGPLAN History of Programming Languages Conference (HOPL – III), 12,1 (2007) 12-55.

11. Kodaganallur V., Incorporating Language Processing into Java Applications: A JavaCC Tutorial, Software, IEEE, 21 (2004) 70-77.

12. URL: http://www.cs.odu.edu/~toida/nerzic/390teched/regular/fa/dfa-definitions.html, Definition of Deterministic Finite Automata, 22 Eylül 2008.

13. Scott, M., L., Programming Language Pragmatics, 1st edition, Elsevier, London, 2000.

14. Earley Jay, An Efficient Context- Free Parsing Algorithm, Communication of the ACM, 13,2 (1970) 94-102.

15. Pehlivan H. ve Üstübioğlu A., Web-Tabanlı Bir Yorumlayıcının Tasarımı ve Gerçeklenmesi, 1. Mühendislik ve Teknoloji Sempozyumu, Nisan 2008, Ankara, Bildiler Kitabı, 71-82.

16. Karan O., Design and Implementation of Interpreters, Yüksek Lisans Tezi, Haliç Üniversitesi, İstanbul, 2005.

17. URL: http://www.cs.nmsu.edu/~rth/cs/cs471/InterpretersJavaCC.html, Writing Interpreter with JavaCC, 22 Eylül 2008.

18. Lerdo, H., G., A Translator Writing System for a Java Oriented Compilers Course, Yüksek Lisans Tezi, University of Florida, Florida, 2003.

19. Kakde, O., D., Algorithms for Compiler Design, 1st edition, Charles River Media, Massachusetts, 2002.(haliç)

20. Friedl, J., E., F., Mastering Regular Expressions, 2nd edition, O'Reilly Media, Cambridge, 2003.

21. URL: http://en.wikipedia.org/wiki/Comparison_of_parser_generators, Comparison of Parser Generator, 10 Aralık 2008.

22. Lesk, M., E. ve Schmidt, E., Lex – A Lexical Analyzer Generator, Computing Science Technical Report No, 39, Bell Laboratories, 1975.

23. Johnson, S., C., Yacc: Yet Another Compiler Compiler, Computing Science Technical Report No, 32, Bell Laboratories, Murray Hill, New Jersey, 1975.

24. Levine, J., R., Mason, T. ve Brown, D., Lex & Yacc, O'Reilly & Associates, Inc., Sebastopol, California, 1992.

6. EKLER

Ek 1. Haskell.jj

PARSER_BEGIN(Haskell)

…

PARSER_END(Haskell)

SKIP :

{

" " | "\t" | "\n" | "\r"

}

TOKEN :

{

< DPOINT : ".." > | < DCOLON : "::" > | < COLON : ":" > | < POWER : "^" > |

< NOTEQUAL : "/=" > | < EQUAL : "==" > | < ASSIGNMENT : "=" > |

< BACKSLASH : "\\" > | < DOUBLEOR : "||" > | < SMALLDASH : "<-" > |

< DASHBIG : "->" > | < EQBIG : "=>" > | < DEXCLAMATION : "!!" > |

< DOUBLEPLUS : "++" > | < DOUBLEAMPRSND : "&&" > | < BIGOREQUAL : ">=" >

| < SMALLOREQUAL : "<=" > | < ONEOR : "|" > | < UNDERDASH : "_" > |

< AT : "@" > | < TILDA : "~" > | < EXCLAMATION : "!" > | < SHARP : "#" > |

< DOLAR : "$" > | < PERCENT : "%" > | < AMPERSAND : "&" > | < TIMES : "" > |*

< PLUS : "+" > | < DOT : "." > | < SLASH : "/" > | < LITTLE : "<" > | < BIG : ">" > |

< QUESTIONM : "?" > | < MINUS : "-" >| < LPAREN : "(" > | < RPAREN : ")" > |

< COMMA : "," > | < SEMICOLON : ";" > | < LSQPAREN : "[" > | < RSQPAREN : "]" > |

< QUOTE : "'" > | < LBRACE : "{" > | < RBRACE : "}" >

}

TOKEN:

{

< #SMALL : ["a"-"z"] > | < #LARGE : ["A"-"Z"] > | < #DIGIT : ["0"-"9"] > |

< VARID : <SMALL> (<SMALL> | <LARGE> | <DIGIT>) > |*

< BOOL : "True" | "False" > | < INT : (<DIGIT>)+ > | < #DQUOTE : "\"" > |

< #GRAPHIC : (~["\""])+ > | < STR : <DQUOTE> <GRAPHIC> <DQUOTE> >

```
}
void Decl():{}
{
  ( LOOKAHEAD(3) FuncLhS() RhS() <SEMICOLON>| LOOKAHEAD(3) Gendecl()
<SEMICOLON>)*
}
void Gendecl() : {}
{  ( Var() )+ <DCOLON> Type() }
void Var() : {}
{  < VARID >}
void Type() : {}
{  SType() ( <DASHBIG> Type() )? }
void SType() : {}
{   TypeVar() | <LPAREN> Type() ( <COMMA> Type() )* <RPAREN> | <LSQPAREN>
TypeVar() <RSQPAREN>}
void TypeVar() : {}
{ <INTEGER> | <BOOLEAN> | <STRING>}
void FuncLhS():{}
{  Var() ( Var() | <LPAREN> Var() <COLON> Var() <RPAREN> | <UNDERDASH> |
Literal())*}
void Apat():{}
{  LOOKAHEAD(2) Var() |  LOOKAHEAD(2) Literal() | LOOKAHEAD(2) GCon() |
<UNDERDASH> | LOOKAHEAD(3) <LPAREN> Pat() ( <COMMA> Pat() )* <RPAREN> |
<LSQPAREN> Pat() ( <COMMA> Pat() )* <RSQPAREN> | Islemler()}
void GCon():{}
{  <LPAREN> <RPAREN> | <LSQPAREN> <RSQPAREN>}
void Literal():{}
{  <INT> | <STR> | <BOOL>}
void Pat():{}
{  <MINUS> Literal() | LOOKAHEAD(3) GCon() (Apat())+ | Apat() }
void Islemler():{}
{   VeyaE() Veya()}
void Veya():{}
```

{ (LOOKAHEAD(2) <DOUBLEOR> VeyaE() Veya())}*

void VeyaE():{}

{ VeE() Ve()}

void Ve():{}

{ (LOOKAHEAD(2) <DOUBLEAMPRSND> VeE() Ve())}*

void VeE():{}

{ LojikE() Lojik()}

void Lojik():{}

{ (LOOKAHEAD(2) (<EQUAL> | <NOTEQUAL> | <LITTLE> | <SMALLOREQUAL> | <BIG> | <BIGOREQUAL>) LojikE() Lojik())}*

void LojikE():{}

{ ConcE() Conc()}

void Conc():{}

{ (LOOKAHEAD(2) (<DOUBLEPLUS> | <DCOLON>) ConcE() Conc())}*

void ConcE():{}

{ PlusE() Plus()}

void Plus():{}

{ (LOOKAHEAD(2) (<PLUS> | <MINUS>) PlusE() Plus())}*

void PlusE():{}

{ MultE() Mult()}

void Mult():{}

{ (LOOKAHEAD(2) (<TIMES> | <SLASH>) MultE() Mult())}*

void MultE():{}

{ PowE() Pow()}

void Pow():{}

{ (LOOKAHEAD(2) <POWER> PowE() Pow())}*

void PowE():{}

{ ExcE() Exc()}

void Exc():{}

{ (LOOKAHEAD(2) <DEXCLAMATION> ExcE() Exc())}*

void ExcE():{}

{LOOKAHEAD(3) Var() (LOOKAHEAD(2) Aexp())+ | LOOKAHEAD(3) Var() | Literal() | LOOKAHEAD(3) <LPAREN> FuncLhS() <RPAREN>| <LPAREN> Islemler() <RPAREN>

}

void RhS():{}

{ <ASSIGNMENT> Exp() | (GdRhS())+}

void Exp():{}

{ LOOKAHEAD(3) Islemler() | <MINUS> Exp() | <BACKSLASH> (Apat())+ <DASHBIG> Exp() | <LET> RhS() <IN> Exp() | <IF> Exp() <THEN> Exp() <ELSE> Exp() | <CASE> Exp() <OF> (LOOKAHEAD(2) Alt())+ | (LOOKAHEAD(3) Aexp())+}

void Aexp():{}

{ Var() | LOOKAHEAD(2) GCon() | Literal() | LOOKAHEAD(2) <LPAREN> Exp() (<COMMA> Exp()) <RPAREN> | LOOKAHEAD(3) <LSQPAREN> Exp() (<COMMA> Exp())* <RSQPAREN> | LOOKAHEAD(3) <LSQPAREN> Exp() (<COMMA> Exp())? <DPOINT> Exp() <RSQPAREN> | LOOKAHEAD(3) <LSQPAREN> Exp() <ONEOR> Qual() (<COMMA> Qual())* <RSQPAREN> }*

void Qual():{}

{ LOOKAHEAD(2) Pat() <SMALLDASH> Exp() | LOOKAHEAD(2) Exp() }

void Alt():{}

{ Pat() <DASHBIG> Exp()}

void GdRhS():{}

{ Gd() <ASSIGNMENT> Exp()}

void Gd():{}

{<ONEOR> Exp()}

Ek 2. AstPrint.java

```
package visitor;
import syntaxtree.*;
import java.util.*;
public class AstPrint extends DepthFirstVisitor {
  public Integer tab = new Integer(0);
  public void write(int t){
     for(int i=0;i<t;i++)
        System.out.print("\t");
  }
  …
 public void visit(Type n) {
     write(tab++);
     System.out.println("new Type ");
     n.f0.accept(this);
     if(n.f1.present())
          {
               write(tab);
               System.out.println(" <DASHBIG> ");
               n.f1.accept(this);
          }
     tab--;
   }
   public void visit(SType n) {//++
     write(tab++);
     System.out.println("new SType ");
     if(n.f0.which==0)
          {
            n.f0.accept(this);
          }
    if(n.f0.which==1)
          {
```

```
                NodeSequence m = (NodeSequence) n.f0.choice;
                write(tab);
                System.out.println(" <LPAREN> ");
                ((Node)m.elementAt(0)).accept(this);
                ((Node)m.elementAt(1)).accept(this);
                NodeListOptional m1 = (NodeListOptional) (m.elementAt(2));
                for(Enumeration e = m1.elements();e.hasMoreElements();)
                {
                        write(tab);
                        System.out.println("<COMMA> ");
                       ((Node)e.nextElement()).accept(this);
                }
                write(tab);
                System.out.println(" <RPAREN> ");
                ((Node)m.elementAt(3)).accept(this);
        }
  if(n.f0.which==2)
        {
            write(tab);
            System.out.println(" <LSQPAREN> ");
            n.f0.accept(this);
            write(tab);
            System.out.println(" <RSQPAREN> ");
        }
  tab--;
}
public void visit(FuncLhS n) {
        write(tab++);
        System.out.println("new FuncLhS ");
        n.f0.accept(this);
        for(Enumeration e = n.f1.elements();e.hasMoreElements();)
        {
          NodeChoice mn = (NodeChoice)e.nextElement();
```

```
        if(mn.which==0)
            {
                mn.accept(this);
            }
        else if(mn.which==1)
            {
                NodeSequence m = (NodeSequence)mn.choice;
                write(tab);
                System.out.println("<LPAREN> ");
                ((Node)m.elementAt(0)).accept(this);
                ((Node)m.elementAt(1)).accept(this);
                write(tab);
                System.out.println("<COLON> ");
                ((Node)m.elementAt(2)).accept(this);
                ((Node)m.elementAt(3)).accept(this);
                write(tab);
                System.out.println("<RPAREN> ");
                ((Node)m.elementAt(4)).accept(this);
            }
        else if(mn.which==2)
            {
                write(tab);
                System.out.println("<UNDERDASH> ");
                mn.accept(this);
            }
        else if(mn.which==3)
            {mn.accept(this);}
    }
    tab--;
  }
  ... // Diğer düğümler için kodlar
}
```

Ek 3. SymbolTable.java

```
package visitor;
import syntaxtree.*;
import environment.*;
import java.text.ParseException;
import java.util.*;
import sun.security.pkcs.ParsingException;
public class SymbolTable extends ObjectDepthFirst {
public boolean right = false, flg = false;
public Object visit(Gendecl n, Object argu) {
    Object _ret=null;
    for (Enumeration e = n.f0.elements();e.hasMoreElements();){
      _ret = (String)((Node)e.nextElement()).accept(this,argu);
      String line = ((String)_ret).substring(((String)_ret).indexOf("
")+1,((String)_ret).length());
      _ret = ((String)_ret).substring(0,((String)_ret).indexOf(" "));
      for (int i = 0; i< ((MySymbol)argu).index; i++){
         if(((String)_ret).compareTo(((MySymbol)argu).dizi[i].func)==0)
         {
                System.out.println("Satir " + line + " \"" + ((String)_ret).trim() + "\":" + "
Gecersiz fonksiyon tekrari");
                throw null;              }
       }
      ((MySymbol)argu).dizi[((MySymbol)argu).index].func=(String)_ret;
    }
    n.f1.accept(this, argu);
    n.f2.accept(this, argu);
    return _ret;
  }
public Object visit(Var n, Object argu) {
    Object _ret=null;
          if(!((MySymbol)argu).dizi[((MySymbol)argu).index].check && !right),
```

```
                    _ret = n.f0.tokenImage+" "+n.f0.beginLine;
            else{
        if(!(((MySymbol)argu).dizi[((MySymbol)argu).index].degisken_kontrol(n.f0.tokenImag
e)))
                    ((MySymbol)argu).undefined.add(n.f0.tokenImage+" "+n.f0.beginLine);
            }
         if(flg){
                _ret = n.f0.tokenImage + " " + n.f0.beginLine;
         }
    n.f0.accept(this, argu);
    return _ret;
  }
 public Object visit(SType n, Object argu) {
    Object _ret=null;
    if(n.f0.which == 0){
        _ret = n.f0.accept(this,argu);
        ((MySymbol)argu).dizi[((MySymbol)argu).index].param_ekle((String)_ret);
    }
    else if (n.f0.which == 2){
        NodeSequence m = (NodeSequence)n.f0.choice;
        (m.elementAt(0)).accept(this,argu);
        _ret = (m.elementAt(1)).accept(this,argu) ;
        ((MySymbol)argu).dizi[((MySymbol)argu).index].param_ekle("["+(String)_ret+"]");
        (m.elementAt(2)).accept(this,argu);
    }
    else if (n.f0.which == 1){
        n.f0.accept(this,argu);
    }
    return _ret;
  }
  public Object visit(TypeVar n, Object argu) {
    Object _ret=null;
    if(n.f0.which==0)
```

```
      _ret = "INTEGER";
    else if (n.f0.which == 1)
        _ret = "BOOLEAN";
    else if (n.f0.which == 2)
        _ret = "STRING";
    n.f0.accept(this, argu);
    return _ret;
  }
  public Object visit(FuncLhS n, Object argu) {
    Object _ret=null;
    ((MySymbol)argu).dizi[((MySymbol)argu).index].check =false;
    _ret = n.f0.accept(this, argu);
    if(!right){
         _ret = ((String)_ret).substring(0, ((String)_ret).indexOf(" "));
        for(int k =0; k<((MySymbol)argu).index+1;k++){
                if(((String)_ret).compareTo(((MySymbol)argu).dizi[k].func)==0&&!(((MySym
        bol)argu).dizi[k].dict.isEmpty())){
                ((MySymbol)argu).index++;
                ((MySymbol)argu).kopyala(k,((MySymbol)argu).index);
        }
    }
    int i = 0;
    int val = 0;
    for (Enumeration e = n.f1.elements();e.hasMoreElements();){
        NodeChoice m = (NodeChoice)e.nextElement();
        if(m.which==0){
                _ret = m.accept(this,argu);
                val = Integer.parseInt(((String)_ret).substring(((String)_ret).indexOf("
")+1,((String)_ret).length()));
                _ret = ((String)_ret).substring(0,((String)_ret).indexOf(" "));
                if(i>(((MySymbol)argu).dizi[((MySymbol)argu).index].params.size()-2)){
                        System.out.print("Satir " + val + " \"" + ((String)_ret).trim() + "\":" +
" Asiri degisken tanimlamasi");
```

```
        throw null;
            }
                else{
                    ((MySymbol)argu).dizi[((MySymbol)argu).index].degisken_ekle(
                (String)_ret,(String)((MySymbol)argu).dizi[((MySymbol)argu).index].p
                arams.elementAt(i++),val);
            }
    }
    else if(m.which==1){
            NodeSequence ns = (NodeSequence) m.choice;
            (ns.elementAt(0)).accept(this,argu);
            _ret = (ns.elementAt(1)).accept(this,argu);
            val = Integer.parseInt(((String)_ret).substring(((String)_ret).indexOf("
")+1,((String)_ret).length()));
            _ret = ((String)_ret).substring(0,((String)_ret).indexOf(" "));
            String temp =
    (String)((MySymbol)argu).dizi[((MySymbol)argu).index].params.elementAt(i) ;
            temp = temp.substring(1,temp.length()-1);
            if(temp.compareTo("INTEGER")!=0 && temp.compareTo("STRING")!=0 &&
temp.compareTo("BOOLEAN")!=0){
                System.out.println("Satir " + val + " \"" + _ret + "\":" + " Hatali dizi
tanimlamasi");
            throw null;
            }
            if(i>(((MySymbol)argu).dizi[((MySymbol)argu).index].params.size()-2)){
        String tm = (String)_ret;
        (ns.elementAt(2)).accept(this,argu);
                _ret = (ns.elementAt(3)).accept(this,argu);
                _ret = ((String)_ret).substring(0,((String)_ret).indexOf(" "));
                System.out.print("Satir " + val +" \"(" + tm + ":" + _ret + ")" + "\":" +
" Asiri degisken tanimlamasi");
            throw null;
             }
```

```
                else{

        ((MySymbol)argu).dizi[((MySymbol)argu).index].degisken_ekle((String)_ret,temp
,val);

                        (ns.elementAt(2)).accept(this,argu);
                        _ret = (ns.elementAt(3)).accept(this,argu);
                        val = Integer.parseInt(((String)_ret).substring(((String)_ret).indexOf("
")+1,((String)_ret).length()));
                        _ret = ((String)_ret).substring(0,((String)_ret).indexOf(" "));
        ((MySymbol)argu).dizi[((MySymbol)argu).index].degisken_ekle((String)_ret,(String)((
MySymbol)argu).dizi[((MySymbol)argu).index].params.elementAt(i++),val);
                        (ns.elementAt(4)).accept(this,argu);
                }
        }
        else if(m.which==2){
                NodeToken nt = (NodeToken) m.choice;
                nt.accept(this,argu);
                if(i>(((MySymbol)argu).dizi[((MySymbol)argu).index].params.size()-2)){
                        System.out.print("Satir " + nt.beginLine + " \"" + nt.tokenImage.trim()
+ "\":" + " Asiri degisken tanimlamasi");
           throw null;
                 }
                else{
        ((MySymbol)argu).dizi[((MySymbol)argu).index].degisken_ekle("_",(String)((MySymb
ol)argu).dizi[((MySymbol)argu).index].params.elementAt(i++),nt.beginLine);
                }
        }
        else if(m.which==3){
                _ret = m.accept(this,argu);
                Integer first_i = ((String)_ret).indexOf(" ")+1 ;
                Integer second_i = ((String)_ret).indexOf(" " , first_i)+1 ;
```

```
                val = Integer.parseInt(((String)_ret).substring( second_i
,((String)_ret).length()));
                if(i>(((MySymbol)argu).dizi[((MySymbol)argu).index].params.size()-2)){
                    System.out.print("Satir " + val + " \"" +
(((String)_ret).substring(first_i,second_i)).trim() + "\":" + " Asiri degisken tanimlamasi");
                        throw null;
                 }
                else{
                    if(!( ((String)_ret).substring(0,((String)_ret).indexOf(" ")).equals(
(String)((MySymbol)argu).dizi[((MySymbol)argu).index].params.elementAt(i) ))){
                            System.out.println("Hata: "+val+" numrali satirda hatali tip
bildirimi");
            throw null;
                    }

        ((MySymbol)argu).dizi[((MySymbol)argu).index].degisken_ekle(((String)_ret).substrin
g(first_i,second_i),(String)((MySymbol)argu).dizi[((MySymbol)argu).index].params.elementA
t(i++),val);
                }
        }
    }
    if(i<(((MySymbol)argu).dizi[((MySymbol)argu).index].params.size()-1)){
                    System.out.print("Satir " + val + ":" + " Eksik degisken bildirimi");
          throw null;
                 }
    }
    else{ // sonradan eklendi
        n.f1.accept(this,argu);
    }
    return _ret;
  }
```

Ek 4. TypeCheck.java

```
package visitor;
import syntaxtree.*;
import environment.*;
import java.util.*;
public class TypeCheck extends ObjectDepthFirst {
   public Integer in = new Integer(0);
   public Integer cycle = new Integer(0);
   public String sym = new String("");
   public String type,type1 = new String("");
   public boolean right = false;
  …
public Object visit(FuncLhS n, Object argu) {
     Object _ret=null;
     ((MySymbol)argu).dizi[in].check =false;
     if(right){
          _ret = n.f0.accept(this,argu);
          int degisken_sira = 0;
          int ind = Integer.parseInt( ((String)_ret).substring(((String)_ret).indexOf("
",((String)_ret).indexOf(" ")+1)+1,((String)_ret).length()) );
          _ret = ((String)_ret).substring(0, ((String)_ret).indexOf(" " ,
((String)_ret).lastIndexOf(" ")-1) );
          for (Enumeration e = n.f1.elements();e.hasMoreElements();){
                    NodeChoice m = (NodeChoice) e.nextElement();
                    if(m.which == 0){
                     _ret = m.accept(this,argu);

if(!((String)((MySymbol)argu).dizi[ind].params.elementAt(degisken_sira)).equals(((String)_r
et).substring(0,((String)_ret).indexOf(" "))))
                     {
```

```
                    System.out.println("Hata :" +
((String)_ret).substring(((String)_ret).indexOf(" "),((String)_ret).length())+ " numarali
satirda \" " +
                                        (String)((MySymbol)argu).dizi[ind].func +" \"
fonksiyonunda hatali parametre bildirimi");
          throw null;
           }
          }
          else if(m.which == 1){
                    NodeSequence ns = (NodeSequence)m.choice;
                    (ns.elementAt(0)).accept(this,argu);
                    _ret = (ns.elementAt(1)).accept(this,argu);
                    (ns.elementAt(2)).accept(this,argu);
                    boolean temp = right;
                    right = false ;
                    (ns.elementAt(3)).accept(this,argu);
                    right = temp;
                    (ns.elementAt(4)).accept(this,argu);
          }
          else if(m.which == 2){
                    System.out.println("Hata :" +
((String)_ret).substring(((String)_ret).indexOf(" "),((String)_ret).length())+ " numarali
satirda hatali \" _ \" ifadesi kullanimi");
          throw null;
          }
          else if(m.which == 3){
           _ret = m.accept(this,argu);
if(!((String)((MySymbol)argu).dizi[ind].params.elementAt(degisken_sira)).equals(((String)_r
et).substring(0,((String)_ret).indexOf(" "))))
           {
                    System.out.println("Hata :" +
((String)_ret).substring(((String)_ret).indexOf(" "),((String)_ret).length())+ " numarali
satirda \" " +
```

```
                                        (String)((MySymbol)argu).dizi[ind].func +" \"
fonksiyonunda hatali parametre bildirimi");
            throw null;
                 }
                }
                degisken_sira++;
                if(degisken_sira>((MySymbol)argu).dizi[ind].params.size()-1){
                        System.out.println("Hata : " +
((String)_ret).substring(((String)_ret).indexOf(" "),((String)_ret).length())+ " numarali
satirda \" " +
                                        ((MySymbol)argu).dizi[ind].func + " \" fonksiyonu icin asiri
parametre bildirimi");          throw null;
                }
        }
        if(degisken_sira<((MySymbol)argu).dizi[ind].params.size()-1){
                System.out.println("Hata : " + ((String)_ret).substring(((String)_ret).indexOf("
"),((String)_ret).length())+ " numarali satirda \" " +
                                        ((MySymbol)argu).dizi[ind].func + " \" fonksiyonu icin eksik
parametre bildirimi");          throw null;
        }
        _ret = (String)((MySymbol)argu).dizi[ind].params.lastElement() +
((String)_ret).substring(((String)_ret).indexOf(" "),((String)_ret).length());
    }
    else{
        n.f0.accept(this, argu);
        n.f1.accept(this, argu);
    }
    return _ret;
  }
…
}
```

Ek 5. Symbol.java

```
package environment;
import java.util.Vector;
import java.util.Dictionary;
import java.util.Enumeration;
public class Symbol{  public String func;
        public Vector params = new Vector();
        public Dictionary dict = new java.util.Hashtable();
        public Dictionary dict2 = new java.util.Hashtable();
         public Dictionary value_table = new java.util.Hashtable(); /
        public boolean check = false ;
public void  degisken_ekle(String degisken, String tur, Integer satir) {
      String s = (String)dict.get(degisken);
      if (s==null){ dict.put(degisken,tur);dict2.put(degisken,satir);}
      else{  System.out.println("Satir " + satir + " \"" + degisken.trim() + "\":" + " Fonksiyon
arguman tekrari");
      }     }
   public void param_ekle(String tur){        params.add(tur);   }
   public boolean degisken_kontrol(String degisken){
        Integer s = (Integer) dict2.get(degisken);
        if(s==null)       return false ; else                return true;
   }
   public void yazdir(){
        System.out.println(func+" fonksiyonu:");
        for (int i = 0; i<params.size(); i++){System.out.print(params.elementAt(i)+ "  " );}
        Enumeration f = dict.keys();
        for (Enumeration e = dict.elements();e.hasMoreElements();)
              System.out.println("Degisken: "+f.nextElement()+" Turu: "+e.nextElement());
              f = dict2.keys();
              for ( Enumeration e = dict2.elements();e.hasMoreElements();)
              System.out.println("Degisken: "+f.nextElement()+" Satir: "+e.nextElement());
}}
```

Ek 6. MySymbol.java

```
package environment;
import java.util.Enumeration;
import java.util.Vector;
public class MySymbol{
        public Integer index = new Integer(-1);
        public Symbol[] dizi = new Symbol[100];
        public Vector undefined = new Vector() ;
        public void kopyala(Integer index1, Integer index2)
        {
                dizi[index2].func = dizi[index1].func;
                for (int i = 0; i<dizi[index1].params.size(); i++){
                        dizi[index2].param_ekle((String)dizi[index1].params.elementAt(i));
                }

        }
        public void yazdir(){
                for (int i = 0; i< undefined.size(); i++){
                        System.out.println(undefined.elementAt(i));
                }
        }
   public void value_table_create(int in, String[] vars,String[] val){
     dizi[in].value_table.put(vars[0], " ");
     for(int i=1;i<vars.length;i++){
        if(dizi[in].dict.get(vars[i]).equals("INTEGER")){
           dizi[in].value_table.put(vars[i],new Integer(val[i]));
        }
        else if(dizi[in].dict.get(vars[i]).equals("STRING")){
            val[i] = val[i].substring(1, val[i].length()-1);
            dizi[in].value_table.put(vars[i],new String(val[i]));
        }
        else if(dizi[in].dict.get(vars[i]).equals("BOOLEAN")){
```

```
            dizi[in].value_table.put(vars[i],new Boolean(val[i]));
        }
        else if(dizi[in].dict.get(vars[i]).equals("[INTEGER]")){
            String[] gecici_dizi = val[i].substring(1,val[i].length()-1).split(",");
            Integer[] array = new Integer[gecici_dizi.length];
            for(int j=0;j<gecici_dizi.length;j++){
                array[j]=Integer.parseInt(gecici_dizi[j]);
            }
            dizi[in].value_table.put(vars[i],array);
        }
        else if(dizi[in].dict.get(vars[i]).equals("[STRING]")){
            String[] gecici_dizi = val[i].substring(1,val[i].length()-1).split(",");
            dizi[in].value_table.put(vars[i],gecici_dizi);
        }
        else if(dizi[in].dict.get(vars[i]).equals("[BOOLEAN]")){
            String[] gecici_dizi = val[i].substring(1,val[i].length()-1).split(",");
            boolean[] array = new boolean[gecici_dizi.length];
            for(int j=0;j<gecici_dizi.length;j++){
                array[j]=new Boolean(gecici_dizi[j]);
            }
            dizi[in].value_table.put(vars[i],array);
        }
      }
    }
}
```

Ek 7. ErrThread.java

```
public class ErrThread extends Thread
{
	boolean bRun = false;
	JTextArea outText = null;
	PrintStream out = null;
	PrintStream stdout = null;
	PipedInputStream stdoutStream = null;
	public ErrThread(JTextArea outText)
	{
		this.outText = outText;
		out = System.out;
		bRun = true;
		redirectStreams();
	}
	public void redirectStreams(){
		try{
			stdoutStream = new PipedInputStream();
			stdout = new PrintStream(new PipedOutputStream(stdoutStream));
			System.setOut(stdout);
		}catch(IOException e){
			System.err.print("ErrThread.redirectStreams:" + e + "\n");
		}catch(SecurityException e){
			System.err.print(e + "\n");
		}
	}
	public void end(){
		bRun = false;
	}

   @Override
	public void run()
```

```
        {
      try
        {
                BufferedReader stdoutReader = new BufferedReader(new
InputStreamReader(stdoutStream));
                String outLine = "";
                while(bRun){
           while ((outLine=stdoutReader.readLine()) != null) {
             outText.append(outLine+"\n"); // <--TextArea
                                }
                }
        }catch (IOException e){
                System.err.print(e + "\n");
        }

                stdout.close();
                System.setOut(out);
        }
}
```

Ek 8. Interpret.java

```
package visitor;
import environment.MySymbol;
import syntaxtree.*;
import java.util.*;
import java.lang.reflect.Array;
public class Interpret extends ObjectDepthFirst {
 …
 public Object visit(Veya n, Object argu) {
     Object _ret=null,temp1=null,temp2=null;
     for(Enumeration e = n.f0.elements();e.hasMoreElements();){
        NodeSequence ns = (NodeSequence)e.nextElement();
        ns.elementAt(0).accept(this, argu);
        temp1 = ns.elementAt(1).accept(this, argu);
        temp2 = ns.elementAt(2).accept(this, argu);
     }
     if(temp2!=null){
        _ret = (Boolean)temp1 || (Boolean)temp2;
     }
     else{
        _ret = temp1;
     }
     return _ret;
   }
 public Object visit(VeyaE n, Object argu) {
     Object _ret=null,_ret1=null;
     _ret = n.f0.accept(this, argu);
     _ret1 = n.f1.accept(this, argu);
     if(_ret1!=null){
        _ret = (Boolean)_ret && (Boolean)_ret1;
     }
     return _ret;
```

```
  }
public Object visit(Ve n, Object argu) {
    Object _ret=null,temp1=null,temp2=null;
    for(Enumeration e = n.f0.elements();e.hasMoreElements();){
       NodeSequence ns = (NodeSequence)e.nextElement();
       ns.elementAt(0).accept(this, argu);
       temp1 = ns.elementAt(1).accept(this, argu);
       temp2 = ns.elementAt(2).accept(this, argu);
    }
    if(temp2!=null){_ret = (Boolean)temp1 && (Boolean)temp2; }
    else{_ret = temp1;   }
    return _ret;
  }
public Object visit(VeE n, Object argu) {
    Object _ret=null,_ret1=null;
    _ret = n.f0.accept(this, argu);
    _ret1 = n.f1.accept(this, argu);
    if(_ret1!=null){
       if(sym.equals("EQUAL")){_ret = _ret.equals(_ret1);         }
       else if(sym.equals("NOTEQUAL")){ _ret = !(_ret.equals(_ret1));}
       else if(sym.equals("LITTLE")){_ret = (Integer)_ret < (Integer)_ret1; }
       else if(sym.equals("SMALLOREQUAL")){_ret = (Integer)_ret <= (Integer)_ret1; }
       else if(sym.equals("BIG")){_ret = (Integer)_ret > (Integer)_ret1; }
       else if(sym.equals("BIGOREQUAL")){_ret = (Integer)_ret >= (Integer)_ret1;  }
    }
    return _ret;
  }
public Object visit(Lojik n, Object argu) {
    Object _ret=null;
    for(Enumeration e = n.f0.elements();e.hasMoreElements();){
       NodeSequence ns = (NodeSequence)e.nextElement();
       NodeChoice nc = (NodeChoice)(ns.elementAt(0));
       if(nc.which==0){sym = "EQUAL" ;      }
```

```
      else if(nc.which==1){sym = "NOTEQUAL" ;         }
      else if(nc.which==2){sym = "LITTLE" ; }
      else if(nc.which==3){sym = "SMALLOREQUAL" ;  }
      else if(nc.which==4){sym = "BIG" ; }
      else if(nc.which==5){sym = "BIGOREQUAL" ; }
      ns.elementAt(0).accept(this, argu);
      _ret = ns.elementAt(1).accept(this, argu);
      ns.elementAt(2).accept(this, argu);
    }
    return _ret;
  }
  public Object visit(LojikE n, Object argu) {
    Object _ret=null,_ret1=null;
    _ret = n.f0.accept(this, argu);
    _ret1 = n.f1.accept(this, argu);
    if(_ret1!=null){
      if(sym.equals("DOUBLEPLUS")){
        Object s = new Object[Array.getLength(_ret)+Array.getLength(_ret1)];
        System.arraycopy(_ret, 0, s, 0, Array.getLength(_ret));
        System.arraycopy(_ret1, 0, s, Array.getLength(_ret), Array.getLength(_ret1));
        _ret = s;
      }
    }
    return _ret;
  }
  public Object visit(Conc n, Object argu) {
    Object _ret=null,t1=null;
    for(Enumeration e = n.f0.elements();e.hasMoreElements();){
      NodeSequence ns = (NodeSequence)e.nextElement();
      NodeChoice nc = (NodeChoice)(ns.elementAt(0));
      if(nc.which==0){ sym = "DOUBLEPLUS" ; }
      else if(nc.which==1){sym = "DCOLON" ; }
      ns.elementAt(0).accept(this, argu);
```

```
      _ret = ns.elementAt(1).accept(this, argu);
      ns.elementAt(2).accept(this, argu);
    }
    return _ret;
  }
  public Object visit(ConcE n, Object argu) {
    Object _ret=null,_ret1=null;
    _ret = n.f0.accept(this, argu);
    _ret1 = n.f1.accept(this, argu);
    if(_ret1!=null){
      if(sym.equals("PLUS")){_ret = (Integer)_ret + (Integer)_ret1;}
      else if(sym.equals("MINUS")){_ret = (Integer)_ret - (Integer)_ret1;}
    }
    return _ret;
  }
  public Object visit(Plus n, Object argu) {
    Object _ret=null,t1=null;
    for(Enumeration e = n.f0.elements();e.hasMoreElements();){
      NodeSequence ns = (NodeSequence)e.nextElement();
      NodeChoice nc = (NodeChoice)(ns.elementAt(0));
      if(nc.which==0){sym = "PLUS" ;}
      else if(nc.which==1){sym = "MINUS" ; }
      ns.elementAt(0).accept(this, argu);
      t1 = ns.elementAt(1).accept(this, argu);
      _ret = ns.elementAt(2).accept(this, argu);
            if(_ret != null){
                  if(sym.equals("PLUS")){_ret = (Integer)t1+(Integer)_ret;}
           else { _ret = (Integer)t1-(Integer)_ret;}
      }
      else{_ret = t1;}
      if(nc.which==0){sym = "PLUS" ; }
      else if(nc.which==1){sym = "MINUS" ; }
    }
```

```
    return _ret;
  }
  public Object visit(PlusE n, Object argu) {
    Object _ret=null,_ret1 = null;
    _ret = n.f0.accept(this, argu);
    _ret1 = n.f1.accept(this, argu);
    if(_ret1!=null){
       if(sym.equals("TIMES")){_ret = (Integer)_ret * (Integer)_ret1;}
       else if(sym.equals("SLASH")){
          if(_ret1.equals(0)){
             System.out.println("Sıfıra bölme hatası");
             throw null;
          }
           _ret = (Integer)_ret / (Integer)_ret1;
       }
    }
    return _ret;
  }
  public Object visit(Mult n, Object argu) {
    Object _ret=null,t1=null;
    for(Enumeration e = n.f0.elements();e.hasMoreElements();){
       NodeSequence ns = (NodeSequence)e.nextElement();
       NodeChoice nc = (NodeChoice) (ns.elementAt(0));
       nc.accept(this, argu);
       if(nc.which==0){sym = "TIMES" ;}
       else if(nc.which==1){sym = "SLASH" ; }
       t1 = ns.elementAt(1).accept(this, argu);
       _ret = ns.elementAt(2).accept(this, argu);
       if(_ret != null){
          if(sym.equals("TIMES")){_ret = (Integer)t1*(Integer)_ret;}
          else {_ret = (Integer)t1/(Integer)_ret; }
       }
       else{_ret = t1;        }
```

```
        if(nc.which==0){sym = "TIMES" ; }
        else if(nc.which==1){sym = "SLASH" ;   }
      }
      return _ret;
    }
…
    public Object visit(Pow n, Object argu) {
      Object _ret=null,temp1=null,temp2=null;
      for(Enumeration e = n.f0.elements();e.hasMoreElements();){
        NodeSequence ns = (NodeSequence)e.nextElement();
        ns.elementAt(0).accept(this, argu);
        temp1 = ns.elementAt(1).accept(this, argu);//powe
        temp2 = ns.elementAt(2).accept(this, argu);//pow
      }
      if(temp2!=null){_ret = (int)Math.pow((Integer)temp1, (Integer)temp2); }
      else{ _ret = temp1;  }
      return _ret;
    }
    public Object visit(PowE n, Object argu) {
      Object _ret=null,_ret1=null;
      _ret = n.f0.accept(this, argu);
      _ret1 = n.f1.accept(this, argu);
      if(_ret1!=null){
       Integer[] t = (Integer[])_ret;
       if((Integer)_ret1>=t.length){
          System.out.println("Dizi sınırları aşıldı");
          throw null;
       }
       return t[(Integer)_ret1];
      }
      else{return _ret;}
    }
…
```

```
  public Object visit(ExcE n, Object argu) {
    Object _ret=null;
    if(n.f0.which==0){
        NodeSequence ns = (NodeSequence)n.f0.choice;
        ns.elementAt(0).accept(this, argu);
        for(Enumeration e = ns.elements();e.hasMoreElements();){
           ((Node)e.nextElement()).accept(this, argu);
        }
    }
    else if(n.f0.which==1){ _ret = n.f0.choice.accept(this, argu);
    }
    else if(n.f0.which==2){_ret = n.f0.choice.accept(this, argu);
    }
    else if(n.f0.which==3){
        NodeSequence ns = (NodeSequence)n.f0.choice;
        ns.elementAt(0).accept(this,argu);
        ns.elementAt(1).accept(this, argu);
        ns.elementAt(2).accept(this, argu);
    }
    else if(n.f0.which==4){
        NodeSequence ns = (NodeSequence)n.f0.choice;
        ns.elementAt(0).accept(this,argu);
        _ret = ns.elementAt(1).accept(this, argu);
        ns.elementAt(2).accept(this, argu);
    }
    return _ret;
  }
…
}
```

ÖZGEÇMİŞ

1983 yılında Trabzon'da doğdu. İlköğrenimini Kanuni Süleyman İlkokulu ve Trabzon Anadolu Lisesi'nde, orta öğrenimini Trabzon Kanuni Anadolu Lisesi'nde tamamladı. 2001 yılında Karadeniz Teknik Üniversitesi Mühendislik Fakültesi Bilgisayar Mühendisliği Bölümü'nde lisans programına başladı ve 2005 yılında bu bölümden bölüm yedincisi olarak mezun oldu.

Kısa bir süre özel sektörde çalıştıktan sonra, Karadeniz Teknik Üniversitesi Fen Bilimleri Enstitüsü Bilgisayar Mühendisliği Anabilim Dalı'nda yüksek lisans programına ve aynı bölümde araştırma görevlisi olarak çalışmaya başladı. Halen bu görevini sürdürmektedir. Yabancı dil olarak İngilizce bilmektedir.