T.C.
ÇANAKKALE ONSEKİZ MART ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ
YÜKSEK LİSANS TEZİ

JAVA 1.5 PROGRAMLAMA DİLİ İÇİN BİR AYRIŞTIRICI TASARIMI VE GERÇEKLEŞTİRİMİ

Mümüne ÖZÇETİN
Bilgisayar Mühendisliği Anabilim Dalı
Tezin Sunulduğu Tarih: **07/02/2012**

Tez Danışmanı:
Doç. Dr. İsmail KADAYIF

ÇANAKKALE

YÜKSEK LİSANS TEZİ SINAV SONUÇ FORMU

MÜMÜNE ÖZÇETİN tarafından **DOÇ. DR. İSMAİL KADAYIF** yönetiminde hazırlanan "**JAVA 1.5 PROGRAMLAMA DİLİ İÇİN BİR AYRIŞTIRICI TASARIMI VE GERÇEKLEŞTİRİMİ"** başlıklı tez tarafımızdan okunmuş, kapsamı ve niteliği açısından bir Yüksek Lisans tezi olarak kabul edilmiştir.

Doç. Dr. İsmail KADAYIF

Danışman

Yrd. Doç. Dr. İbrahim TÜRKYILMAZ

Jüri Üyesi

Yrd. Doç. Dr. Can AKTAŞ

Jüri Üyesi

Sıra No :

Tez Savunma Tarihi: 07/02/2012

Prof. Dr. İsmet KAYA

Müdür

Fen Bilimleri Enstitüsü

İNTİHAL (AŞIRMA) BEYAN SAYFASI

Bu tezde görsel, işitsel ve yazılı biçimde sunulan tüm bilgi ve sonuçların akademik ve etik kurallara uyularak tarafımdan elde edildiğini, tez içinde yer alan ancak bu çalışmaya özgü olmayan tüm sonuç ve bilgileri tezde kaynak göstererek belirttiğimi beyan ederim.

Mümüne ÖZÇETİN

TEŞEKKÜR

Bu tezin gerçekleştirilmesinde, çalışmam boyunca benden bir an olsun yardımlarını esirgemeyen saygı değer danışman hocam Doç. Dr. İsmail KADAYIF'a, çalışma süresince tüm zorlukları benimle göğüsleyen F. Oğuz ÖZKEROĞLU'na, Arş. Gör. Selen AKTAN başta olmak üzere tüm çalışma arkadaşlarıma ve hayatımın her evresinde bana destek olan değerli aileme sonsuz teşekkürlerimi sunarım.

Mümüne ÖZÇETİN

SİMGELER VE KISALTMALAR

BNF	Backus–Naur Form
LL	Left Most Derivation (En Soldan Türetim)
LR	Rightmost Derivation (En Sağdan Türetim)
LALR	Look-ahead LR parser
CUP	Constructor of Useful Parser (Kullanışlı Ayrıştırıcı Yapıcı)
JIT	Just in Time (Çalışma Anında)
CFG	Context Free Gramer (İçerikten Bağımsız Gramer)

ÖZET

JAVA 1.5 PROGRAMLAMA DİLİ İÇİN BİR AYRIŞTIRICI TASARIMI VE GERÇEKLEŞTİRİMİ

Mümüne ÖZÇETİN

Çanakkale Onsekiz Mart Üniversitesi

Fen Bilimleri Enstitüsü

Bilgisayar Mühendisliği Anabilim Dalı Yüksek Lisans Tezi

Danışman : Doç. Dr. İsmail KADAYIF

07/02/2012, 41

Ayrıştırıcı, bazı özel kurallara göre yazılan bir metni ayrıştıran ve metni bellekte özel bir veri yapısıyla temsil eden bir programdır. Ayrıştırıcılar bilgisayar biliminde değişik alanlarda kullanılabilir. Onlar bir programlama dilinde yazılan programı başka bir programlama dilinde yazılan programa dönüştüren derleyicilerde ara bir adım olarak kullanılabilir. Ayrıca onlar doğal dil işlemede de yaygın olarak kullanılmaktadır.

Bu tezde, Java 1.5 belirtimine dayalı programlar için bir ayrıştırıcı tasarlamaya ve gerçekleştirmeye çalıştık. Yazılım araçları olarak otomatik tarayıcı üreteci olan JFlex ve otomatik ayrıştırıcı üreteci olan CUP'ı kullandık. JFlex, tokenları tanımlamak için düzenli ifadeleri kullanır. CUP, dilin sözdizimsel kurallarını belirtmek için içerikten bağımsız gramerleri kullanır. CUP, çoğu programlama dilini ayrıştırabilen yukarıdan aşağı LALR ayrıştırıcıları üretir. Bizim ayrıştırıcımız birden fazla Java dosyasını ayrıştırabilir ve onları tek bir ayrıştırıcı ağacı olarak temsil edebilir. Ayrıştırıcımızın doğru çalıştığını test etmek için şu adımlar uygulanır. Orijinal Java dosyalarını ayrıştırdıktan sonra, ayrıştırma ağacını dolaşarak yeni Java dosyaları diske yazdırılır. Bu yazdırılan dosyalar, dosyalarımızın ilk grubunu oluşturur. Daha sonra ilk grupta yazdırılan Java dosyaları, ayrıştırıcımıza girdi teşkil eder ve ilgili ayrıştırma ağacı dolaşılarak ikinci grubu oluşturan yeni dosyalara yazdırılır. Birinci gruptaki her bir Java dosyası ikinci gruptaki ilgili Java dosyası ile karşılaştırılır. Her iki gruptaki karşılıklı dosyaların birebir aynı olduğu görülür.

Anahtar sözcükler: LALR, Ayrıştırıcı, CUP, Sözcüksel Analiz, JFlex

ABSTRACT

DESIGN AND IMPLEMENTATION OF A PARSER FOR JAVA 1.5 PROGRAMMING LANGUAGE

Mümüne ÖZÇETİN

Çanakkale Onsekiz Mart University

Graduate School

Computer Engineering Thesis, Master of Science

Advisor : Assoc. Prof. Dr. Ismail KADAYIF

07/02/2012, 41

A parser is a program that parses a text written according to some specific rules and represents the text in memory with an appropriate data structure. Parsers can be used in a variety of areas in computer science. They can be used as an intermediate step in compilers that convert a programming language program into another programming language program. They are also extensively used in natural language processing.

In this thesis, we try to design and implement a parser targeting at programs based on Java 1.5 specifications. We use JFlex and CUP software tools as an automatic scanner generator and an automatic parser generator. JFlex employs regular expressions to identify tokens. CUP uses context free grammars to specify the language's syntactic rules. It produces top-down LALR parsers via which most of the programming languages can be parsed. Our parser can parse multiple Java files and represent them as a single parse tree. To test if our parser runs correctly we apply the following steps. After parsing original Java files we traverse the parse tree and dump it into new Java files. These dumped files constitute our first group of files. Then the dumped Java files in the first group are fed into our parser and the corresponding parse tree is dumped into another set of files, which constitute our second group of files. We compare each Java file in the first group with the corresponding Java file in the second group. After extensive experiments, we see that the compared Java files have a verbatim text.

Keywords: Lexical Analysis, Compiler, LALR, JFlex, CUP

İÇERİK **Sayfa**

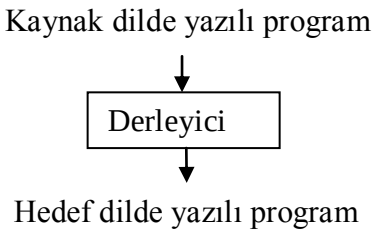
BÖLÜM 1
GİRİŞ

Ayrıştırıcı (parser), verilen bir yazıyı analiz ederek yazının verilen gramer kurallarına uygun yazılıp yazılmadığını kontrol eden bir bilgisayar programıdır. Burada bahsi geçen yazı, herhangi bir doğal dilde yazılı bir metin olabileceği gibi herhangi bir programlama dilinde yazılı olan bir program kodu da olabilir. Ayrıştırıcıların bilgisayar biliminde birçok kullanım alanları vardır.

Yüksek seviyeli bir programlama dilinde yazılı bir programı başka bir program şekline dönüştüren derleyicilerde ayrıştırıcı ara bir adım olarak kullanılır. Makine dili kodlarını inceleyerek yüksek seviyeli programlama diline dönüştüren geriye dönük derleme yapan derleyiciler de ayrıştırıcıdan faydalanabilirler. Ayrıca programlama dillerinde derleyici eklentileri kullanılarak güvenlik, statik kontrol, dil tasarımı, iyileştirme, biçimlendirme ve öğretme gibi faydalı özelliklerin dile kazandırılmasında ayrıştırıcılar kullanılabilir (Nystrom, 2003). Ayrıştırıcıları aşağıda derleyicilerin bir ara adımı olarak *(söz dizimsel analizci)* tanıtmaya çalışacağız.

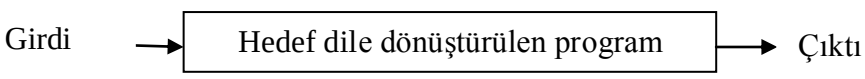
1.1. Derleyici

Bir dilde yazılı bir programı başka bir dilde yazılı bir program şekline dönüştüren programa *derleyici* denir (Şekil 1). Derleyicilerin önemli bir rolü dönüşüm esnasında tespit edilen, kaynak programdaki hataları raporlamaktır.



Şekil 1 . Derleyici

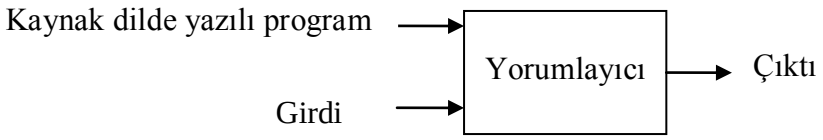
Hedef dil olarak yüksek seviyeli bir dil seçilebileceği gibi en alt seviye dili olan makine dili de seçilebilir. Hedef program yürütülebilir bir makine-dili programı ise girdileri işlemek için kullanıcı tarafından çağrılabilir ve girdilere ilişkin çıktılar üretebilir.



Şekil 2 . Çalışan Hedef Program

Yorumlayıcı, dil işlemcisinin diğer bir türüdür. Bu süreçte kullanıcılar tarafından

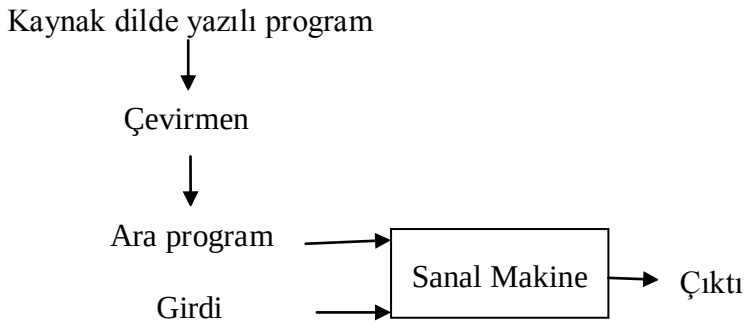
sağlanan girdiler de dikkate alınarak kaynak programda belirtilen işlemleri doğrudan yürütmek için çalışma zamanında makine dilinde adım adım üretilen kodlar çalıştırılır.



Şekil 3 . Yorumlayıcı

Derleyici tarafından üretilen makine dili programı, genelde girdileri çıktılara eşlemede yorumlayıcıdan çok daha hızlı çalışır. Ancak bir yorumlayıcı bir derleyiciden daha iyi hata tespitleri yapar. Bunun temel sebebi derleme zamanında tespit edilemeyen bazı hataların yorumlama zamanında tespit edilebilmesidir.

Bazı diller için dil işlemcileri derleme ve yorumlamayı birleştirir. Bu tip derlemelere genellikle *melez (hibrid) derleme* denir (Şekil 4). Bunun en güzel örneklerinden biri Java'dır. Java kaynak programı ilk olarak *byte kod* denilen bir ara biçime çevrilir. Byte kodlar ise daha sonra bir sanal makine tarafından yorumlanır. Bunun temel faydası, bir makinede derlenen byte kodların diğer bir makinede (farklı bir platform) yorumlanabilmesidir. Hatta byte kodlar internet üzerinden indirilip Java sanal makinesine sahip bir bilgisayarda çalıştırılabilir. Girdilerin çıktılara daha hızlı eşlenmesini başarmak için çalışma anında (*Just-in-Time (JIT))* (El-Kadri ve ark., 2006), (Fulton ve Stoodley, 2007) derleyici denilen bazı Java derleyicileri, byte kodları doğal makine diline topluca dönüştürür. Bu işlem özellikle sıkça çağrılan sınıf metotlarına uygulanır. İlk çağrıda metoda ait byte kodlar makine diline dönüştürülerek saklanır. Aynı metoda yapılan sonraki çağrılarda ise ilgili metoda ait makine kodu çağrılacağından metot çağrısı daha kısa sürede tamamlanacaktır. Smalltalk'da JIT derlemeyi başarıyla gerçekleştiren Deutsch ve Schiffman bu konunun öncüleri olmuştur (Cramer ve ark., 1997).



Şekil 4. Melez Bir Derleyici

1.1.1. Bir Derleyicinin Yapısı

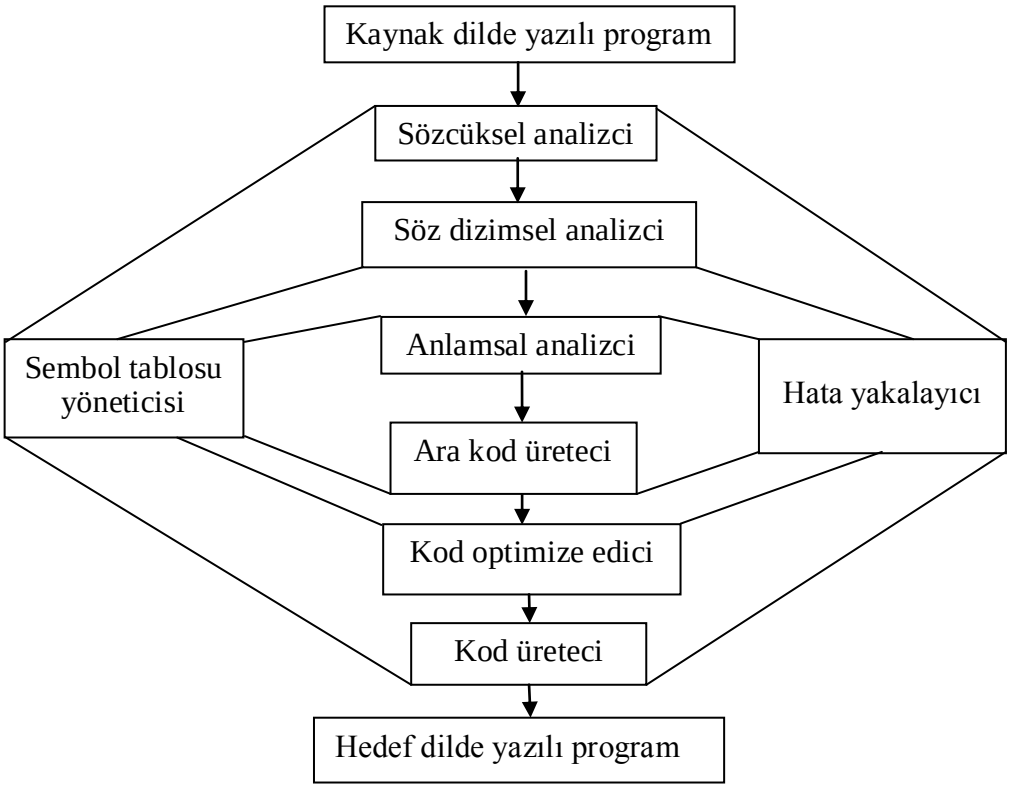
Modern bir derleyici, genelde her biri farklı bir soyut dilde işlem gören aşamalara bölünmüştür (Appel, 1997). Derleme işlemi bu aşamalar ile bu aşamalardan farklı bir takım aktivitelerden oluşur. Altı aşamalı ve iki aktiviteli derleme işlemi Şekil 5'te gösterilmektedir (Aho ve ark, 2007). Aşamaları sırasıyla sözcüksel analizci (lexical analyzer), söz dizimsel analizci (syntax analyzer), anlamsal analizci (semantic analyzer), ara kod üreteci (intermediate code generator), kod optimize edici (code optimizer) ve kod üreteci (code generator) şeklinde sayabiliriz. Aktiviteler ise sembol tablosu yönetimi (symbol table management) ve hatalarla uğraşma (error handler) şeklindedir. Derleyicinin her bir aşaması, kaynak programı bir temsilden başka bir temsile dönüştürür. Pratikte bu aşamaların bazıları gruplanarak tek aşama şekline getirilebilir. Derleyicinin ilk üç aşaması analiz kısmını geri kalan aşamalar ise sentez kısmını oluşturur. Derleyicinin her iki aktivitesi de altı aşama ile etkileşimde bulunur. Derleyici aşamalarını ve aktivitelerini aşağıda kısaca özetlemeye çalışalım.

Kaynak programı oluşturan karakterlerin soldan sağa doğru okunup token olarak gruplandırılma işlemi sözcüksel analizci tarafından yapılır. Yine programda kullanılan anahtar sözcüklerin tespiti de sözcüksel analizci vasıtasıyla gerçekleştirilir. Kaynak programdaki tokenların derleyici tarafından kod sentezi için kullanılan gramatiksel ibareler olarak gruplanması işlemi sözdizimsel analizci tarafından yerine getirilir. Kaynak programın anlamsal olarak doğru olup olmadığı ve daha sonraki aşamalar için gerekli olan tip bilgilerinin toplanması anlamsal analizci tarafından yapılır. Bazı derleyiciler sözcüksel analiz, sözdizimsel analiz ve anlamsal analiz aşamalarını takiben kaynak kodun ara temsiline yönelik bir kod oluşturur. Bu ara temsili, soyut bir dilde program yazımı şeklinde düşünebiliriz. Bu şekilde ara kod üretimi işleminden, ara kod üreteci sorumludur. Kod optimize edici ise daha hızlı çalışabilen makine kodu üretimi için ara kod üzerinde bilinen bir takım kod optimizasyonlarını uygular. Derleyicin son aşaması kod üretimi olup, bu aşamada assembly kodu olarak da bilinen yeri değiştirebilen hedef kodun üretimi gerçekleştirilir.

Şimdi de derleyicinin temel iki aktivitesinden kısaca bahsedelim. Önce sembol tablosu yönetimini açıklayalım. Derleyicinin temel fonksiyonlarından biri kaynak programda tanımlanan tanımlayıcıları (identifier) kaydetmek ve bu tanımlayıcıların niteleyicileri hakkında belirlenen bilgileri toplamaktır. Her bir tanımlayıcı için sembol tablosunda bir yer ayrılır ve tanımlayıcı ile ilgili niteleyici bilgiler bu yere yerleştirilir. Bu bilgiler derleyicinin birçok aşamasında kullanılır. Örneğin, kod üretimi aşamasında bir değişken için bellekte ne kadar yer ayrılacağı, ilgili değişkene ait sembol tablosundaki

bilgilere bakılarak karar verilir. Hataların tespiti ise hata uğraşına girmektedir. Her bir aşama, çeşitli hatalarla karşılaşabilir. Sözcüksel analizci, söz dizimsel analizci ve anlamsal analizci bir programda derleyici tarafından tespit edilebilen hataların büyük bir kısmını tespit edebilir. Sözcüksel analizci anlamlı token oluşturamayan karakter katarlarını tespit eder. Dilin sözdizimsel (syntax) yapısına uymayan token gruplarını sözdizimsel analizci tespit eder. Anlamsal analiz sırasında sözdizimine uyan fakat bir anlam ifade etmeyen yapılar tespit edilir. Örneğin, uzunluğu 4 byte olarak belirlenen bir değişkene 4 byte'lık alana sığmayan bir sayının atanması veya bir bölme işleminde bölenin 0 olması gibi.

Bu tezin konusu derleyicinin ilk iki aşaması olduğundan bu iki aşamayı aşağıda daha ayrıntılı bir şekilde açıklayacağız.



Şekil 5. Bir Derleyicinin Aşamaları

1.2. Sözcüksel Analiz

Derlemenin ilk safhasına sözcüksel analiz veya tarama denir. Sözcüksel analizci, kaynak programı oluşturan karakterler akışını girdi olarak alır ve karakterleri lexeme denilen anlamlı dizilere gruplar. Her lexeme için, sözcüksel analizci bir sonraki aşama olan sözdizimsel analiz aşamasına geçirilen (token-name, attribute-value) biçiminde bir tokenı çıktı olarak üretir. Tokendaki ilk bileşen olan token-name sözdizimi analizi sırasında kullanılan soyut bir semboldür ve ikinci bileşen bu token için sembol tablosunda ayrılan

bir girdiye işaret eder. Sembol tablosundaki girdi bilgisi anlamsal analiz ve kod üretimi için gereklidir.

Çoğu programlama dilinde kullanılan token türleri şunlardır: (Hutton, 2006)

- Tanımlayıcı (x, y, average, vb.)
- Ayrılmış veya anahtar kelimeler (if, else, while, vb.)
- Tam sayı sabitler (42, 0xFF, 0177 vb.)
- Kayan noktalı sabitler (5.6, 3.6e8, vb.)
- String sabitler ("hello there\n", vb.)
- Karakter sabitler ('a', 'b', vb.)
- Özel semboller (( ) : := + - vb.)
- Yorumlar (İhmal edilecek)
- Derleyici direktifleri (Dosyaları dahil etmek için direktifler, makro tanımlamalar, vb.)
- Satır bilgisi
- Beyaz boşluk (White space)
- Dosya sonu

Sözcüksel analiz, sözcüksel olarak geçersiz olabilen metinle başa çıkabilmelidir. Örneğin;

- Bir sayı çok büyük olabilir, bir string veya bir tanımlayıcı çok uzun olabilir.
- Bir sayı tamamlanmamış olabilir (26., 26e gibi).
- Bir stringte son bilgi unutulmuş olabilir.
- Bir yorumun sonu unutulmuş olabilir.
- Özel bir sembol tamamlanmamış olabilir
- Metinde geçersiz karakterler görünebilir, örneğin yanlışlıkla bir binary dosyayı sözcüksel olarak analiz etmeye çalışabiliriz.
- Derleyici direktifleri geçersiz olabilir (Hutton, 2006).

1.2.1. Düzenli İfadeler

Düzenli ifadeler, karakter katarlarından oluşan kümelerin cebirsel olarak tanımlanmasını sağlayan bir gösterim yöntemidir (Kleene, 1956). Çoğu programlama dilinde, sözcüksel tokenlar düzenli ifade denilen kalıplarla tanımlanabilen son derece basit bir yapıya sahiptir. Düzenli ifadeler birçok editörde kelime işlemede kullanılmaktadır. Aynı zamanda düzenli ifadeler birçok yazılım araçlarında da kullanılmaktadır. Örneğin Unix *grep* komutu, düzenli ifadelerle tanımlanan, kalıplara uyan sözcükleri dosyalarda

aramak için geliştirilmiştir. Arama motorlarında, bilgi çıkarımında, kelime işlemede ve veri doğrulamada düzenli ifadelerden faydalanılmaktadır. Düzenli ifadeye bir örnek verecek olursak, [A-Za-z][A-Za-z0-9]* kalıbını kullanarak çoğu programlama dilinde değişken isimlerini belirleyebiliriz (Hutton, 2006).

Düzenli ifadeler sözcüksel analiz için kullanılan özel amaçlı çok benzer dört yazılım aracı olan Lex, Flex, JLex ve JFlex tarafından da kullanılmaktadır. Programcı düzenli ifadeleri kullanarak eşleşecek tokenları ve geleneksel programlama dilinde gerçekleştirilecek eylemleri belirtir. Lex (Lesk ve Schmidt, 2011) ve Flex (Paxson, 1995), C tabanlıdır. JLex (Berk, 2003) ve JFlex (Klein, 2009), Java tabanlı karşılıklarıdır. Lex/Flex veya JLex/JFlex yazılım araçları, diğer C veya Java koduyla birleştirilebilen, C veya Java programı üretir. Flex ve JFlex, aşağı yukarı Lex ve JLex'in genişletilmiş GNU versiyonlarıdır. Biz çalışmamızda sözcüksel analiz için JFlex'i kullandık. Derleyicilerde sözcüksel analiz için yukarıda bahsedilen yazılım araçlarını kullanarak otomatik olarak üretilen programa *tarayıcı (scanner)* denir.

Bir tarayıcı şu 3 şeyi yapabilmelidir:

1. Bütün beyaz boşlukları (boşluk, tab, satır atlama) ve açıklamaları belirleyip sanki bunlar dosyada yokmuş gibi davranabilmeli
2. Karakter dizisi içindeki tüm tokenları bulabilmeli
3. Bulunan token için lexeme ve tokenin bulunduğu konumun satır ve sütun numaraları gibi özellikler sözdizimsel analizciye döndürebilmelidir.

1.3. Sözdizimsel Analiz (Syntactic Analysis)

Derlemenin ikinci aşaması sözdizimsel analiz aşamasıdır. Bu aşama hem programlama dilleri hem de doğal dil işlemede (Kovacs ve Barabas, 2010) ara bir adım oluşturur. Bu aşamaya *ayrıştırma (parsing)* aşaması da denir. Bu aşamada, program kendi yapısal bileşenlerine analiz edilmektedir – çeşitli bildirim tipleri, statementlar, ifadeler vb. Yapısal hatalar yine bu aşamada tespit edilmektedir. Ayrıştırıcı (sözdizimi analizi gerçekleştiren süreç, parser), gerekirse sözdizimi hata mesajları da üretir. Ayrıştırıcı tokenları sözcüksel analizciden (tarayıcı kodu çağırmak suretiyle) elde eder; programın yapısının sözdizimsel analizini yapar ve programın yapısını belirleyen bir *"soyut sözdizimi ağacı" (parse tree)* üretir.

Ayrıştırma işlemi, *yukarıdan aşağıya (top-down)* ve *aşağıdan yukarı (bottom-up)* olmak üzere ikiye ayrılmaktadır. Bu sınıflandırma ayrıştırma ağacı inşasındaki düğümlerin oluşturma sırasıyla ilgilidir. Aşağıdan yukarı ayrıştırıcılarda ağaç oluşturma yapraklardan başlayıp köke doğru devam ederken, yukarıdan aşağı ayrıştırıcılarda bu süreç kökten

başlayıp yapraklara doğru devam eder. Aşağıdan yukarı ayrıştırma daha geniş gramer sınıflarını ve dönüştürme yöntemlerini yönetebilir. Bu yüzden gramerlerden doğrudan otomatik olarak ayrıştırıcı üretmek için yazılım araçları çoğunlukla aşağıdan yukarı metodları kullanır. Genel olarak yukarıdan aşağı ayrıştırma LL ayrıştırma, aşağıdan yukarı ayrıştırma da LR ayrıştırma olarak bilinir. Gerek LL ayrıştırıcılar gerekse LR ayrıştırıcılar *içerikten bağımsız gramerler (Context Free Grammar, CFG)* temelli sözdizimsel analiz yaparlar.

Her ne kadar gerek LR gerekse LL ayrıştırıcıların geçerliliğini yitirdiği konusunda eleştiriler varsa da (Might ve Darais, 2010), LL ve LR en yaygın kullanılan ayrıştırma yöntemleridir (Sippu ve Soinien, 1988), (Sippu ve Soinien, 1990). Yaygın kullanılan programlama dilleri için yazılı açık kaynak kodlu derleyiciler incelendiğinde LL ve LR yöntemlerinden hangisini kullanmanın daha iyi olduğu konusunda kesin bir karara varmak zordur. Sun/Oracle Java derleyicisi LL yöntemini kullanırken, Eclipse Java derleyicisi Jikes tarafından üretilen LALR ayrıştırıcı kullanır. Google'un Go programlama dili için hem LL'nin elle gerçekleştirimi olan özyinelemeli alçalan (recursive descent) ayrıştırıcı hem de Bison ile üretilen LALR ayrıştırıcı mevcuttur. GCC önceleri Bison ile üretilen LALR ayrıştırıcı kullanırken, 2004 yılından itibaren GCC 3.4.0 ile birlikte C++ için ve 2006 yılından itibaren GCC 4.1.0 ile birlikte C/Objective C için özyinelemeli alçalan ayrıştırıcı kullanmaya başladı. Python elle yazılı LL(1) kullanırken, Ruby ve PHP Bison ile üretilen LALR ayrıştırıcı kullanmaktadır.

LR ayrıştırıcılar temelde iki avantaj sunarlar. Bunlardan ilki, LL ayrıştırıcılara kıyasla LR ayrıştırıcıların ayrıştırma yeteneği daha güçlüdür. Bütün deterministik CFG dillerini ayrıştırabilirler ve üretim kurallarındaki soldan özyineleme soruna sebep olmaz. İkincisi, neredeyse yaygın olan bütün programlama dillerinin bir LALR ayrıştırıcısı mevcuttur.

LL ayrıştırıcılara sıra gelecek olursa; LL ayrıştırıcılarının en büyük iki avantajından biri elle yazılabilecek kadar basit olmaları ve diğeri ise etkin hatadan kurtulma (error recovery) (Slivnik ve Vilfan, 2004) mekanizmalarının ayrıştırıcıya kolayca entegre olmasında kolaylık sağlamalarıdır. Her ne kadar elle yazılabilseler de, LL'ler için üreteçler mevcuttur. ANTLR, LL(*) (Parr ve Fischer, 2011) ayrıştırma kullanırken LISA (Henriques ve ark., 2005), hem LL hem de LR ayrıştırma kullanabilir (üretilen ayrıştırıcı sadece birini kullanır, ikisini değil). LL ayrıştırıcılar bazı ilavelerle daha güçlü kılınabilirler. Örneğin, Slivnik (Slivnik, 2011) LL'deki çakışmaları önlemek için küçük LR ayrıştırıcı kullanmıştır. Melez görünümlü bu ayrıştırıcı, derleyici gözünde yukarıdan aşağıya ayrıştıran ve iyi hatadan kurtarma özelliği gösteren bir ayrıştırıcıdır.

1.3.1. İçerikten Bağımsız Gramerler

Her ne kadar CFG dil işlemede yaygın kullanılmasına rağmen, kullanım alanı sadece bununla sınırlı değildir. Örneğin, CFG ağdan gelen saldırıların modellenmesinde kullanılabilir (Al-Mamory ve ark., 2008). CFG, *yinelemeli (recursive)* yapılardan oluşan bir grup kurala denir. Bunu bir örnekle açıklarsak;

```
S-->aB | bA
A-->a  | aS  | bAA
B-->b  | bS  | aBB
```

Yukarıda 8 kuralla verilen CFG, 0'dan büyük ve eşit sayıda "a" ve "b" karakterleri içeren bir dili formal olarak ifade etmektedir. Buradaki kuralların her birine bir türetim kuralı denmektedir. CFG'ler iki türlü sembol içermektedir. Bunlardan ilki *sonlandırılmamış (nonterminal)* semboller olarak adlandırılır. Yukarıdaki örnekte bu tür sembollerin oluşturduğu kümeyi {S,A,B} olarak ifade edebiliriz. Sonlandırılmamış sembollerden biri başlangıç sembolü olarak isimlendirilir ve türetme işlemi bu başlangıç sembolünden başlar. Örnek gramerimizde S, başlangıç sembolüdür. Gramerde ifade edilen diğer sembollere *sonlandırılmış (terminal)* semboller denir. Gramerimizdeki sonlandırılmış sembollerin oluşturduğu küme ise {a,b}'dir.

Genel olarak iki tür türetim vardır. Bunlardan ilki *en soldan türetim (left most derivation)* olup, bu tür türetimlerde türetimin herhangi bir adımında en soldaki sonlandırılmamış sembol sağ tarafıyla yer değiştirilir. İkinci türetim şekli *en sağdan türetim (right most derivation)* olup, türetimin her adımında en sağdaki sonlandırılmamış sembol sağ tarafıyla yer değiştirilir. En soldan ve en sağdan türetimlere birer örnek verelim. Bunun için yukarıdaki grameri göz önünde bulundurarak "bbbabaaa" cümlesini her iki yöntemle türetmeye çalışalım.

```
S --> bA --> bbAA --> bbbAAA -->
bbbaAA --> bbbabAAA --->
bbbabaAA --> bbbabaaA -->
bbbabaaa
```

Şekil 6. En Soldan Türetim

```
S --> bA --> bbAA --> bbAa -->
bbbAAa --> bbbAbAAa --->
bbbAbAaa --> bbbAbaaa -->
bbbabaaa
```

Şekil 7. En Sağdan Türetim

Şekil 6'da ilgili cümlenin en soldan türetimi gösterilirken Şekil 7'de cümlenin en sağdan türetimi gösterilmektedir.

1.3.2. LR Ayrıştırıcılar

Çok farklı aşağıdan yukarı ayrıştırma algoritmaları tasarlanmıştır. Bunların çoğu LR denilen sürecin çeşitleridir. LR ayrıştırıcılar diğerlerine göre daha küçük ayrıştırma tablosu ve daha az hacimli program koduna ihtiyaç duyarlar. Orijinal LR algoritması, Donald Knuth tarafından tasarlanmıştır ve 1965'te yayınlanmıştır (Sebesta, 2010), (Knuth, 1965). LR ayrıştırıcıların avantajlarından bazıları aşağıda verilmektedir :

1. Programlama dillerini tanımlayan neredeyse tüm gramerler için kullanılabilirler.
2. Diğer aşağıdan yukarı ayrıştırma algoritmalarından daha geniş gramer sınıflarını kabul etmelerine karşın onlar kadar verimli çalışırlar.
3. Yazım hatalarını mümkün olduğunca hızlı tespit ederler.
4. LR gramer sınıfları, ayrıştırılabilen LL gramer sınıflarının bir üst kümesidir.

LR ayrıştırmanın tek dezavantajını bir programlama dili için gerek duyulan ayrıştırma tablosunun elle üretiminin hemen hemen imkânsız olması olarak ifade edebiliriz. Bu dezavantaj, grameri girdi kabul eden ve otomatik ayrıştırma tablosu oluşturan otomatik ayrıştırıcı üreteç yazılım araçlarının geliştirilmesiyle giderilmiştir.

Daha önce de belirtildiği gibi LR ayrıştırıcılar türetim ağacını yapraklardan köke doğru inşa ederler. Bu husus her defasında türetim kurallarından birinin sağ tarafını verilen cümleye ait cümlesel formun içerisinde bulup sol tarafıyla yer değiştirmeye karşı gelir. Diğer bir ifadeyle her bir adımda giriş sözcüğüne karşı gelen cümlesel form içerisinde bir handle aranarak handle sol tarafıyla yer değiştirilir. Bu işlem gramerin başlangıç sembolüne ulaşıncaya kadar tekrarlanır. *Handle*in sağlaması gereken özelliği şöyle ifade edebiliriz. Bir türetim kuralının sağ tarafına karşı gelen bir altstring olması ve cümlesel formda yerine ilgili türetim kuralının sol tarafı yazılması (indirgeme) halinde en sağdan türetimin ters sırada bir adımının elde edilmesi gerekir.

LR ayrıştırma işlemini bir örnekle gösterelim. "bbbabaaa" cümlesini yukarıda verilen gramer dikkate alınarak LR yöntemiyle ayrıştırmaya çalışalım. LR yönteminin ilgili cümleye uygulanışı Çizelge 1'de gösterilmektedir.

Çizelge 1. "bbbabaaa" Cümlesinin LR Yöntemiyle Ayrıştırılması

Sağ cümlesel form	Handle	İndirgeme kuralı
bbb**a**baaa	a	A --> a
bbbAb**a**aa	a	A--> a
bbbAbA**a**a	a	A--> a
bbbA**bAA**a	bAA	A --> bAA
bb**bAA**a	bAA	A-->bAA
bbA**a**	a	A-->a
b**bAA**	bAA	A-->bAA
bA	bA	S-->bA
S		

Çizelgede de gösterildiği üzere her adımda sağ cümlesel formda bir handle aranır ve bulunan handle sol tarafına indirgenir. Çizelgede handlelar koyu olarak gösterilmektedir. Her adımda sadece bir handle belirlenmelidir. Neyin handle olacağına hem altstringin içeriğine hem de altstringin cümlesel formdaki konumuna bakılarak karar verilir. Örneğin birinci adımda dört tane "a" sembolünden yalnızca ilki (koyu olanı) handle olup diğer üç tanesi handle değildir. Bunun sebebi sadece ilk "a" sembolü sağ tarafı olan "A" ya indirgendiğinde en sağdan türetimin ters yönde bir adımı gerçekleştirilmiş olur. Çizelgeyi dikkatlice incelersek, tablodaki adımların sırası Şekil 7'deki en sağdan türetim adımlarının ters yöndeki sırasına karşı geldiğini görebiliriz.

1.3.3. LR Ayrıştırma İşlemi

LR ayrıştırıcı işlemi 5 bileşenden oluşur. Bunlardan ilki giriş stringini tutan giriş tamponudur. İkincisi ise ayrıştırmanın çıktısı olan ayrıştırma ağacıdır. Üçüncüsü ise gramer sembollerinin ve durum bilgilerinin tutulduğu yığıt. Dördüncüsü ayrıştırıcı sürücü programı. Sonuncusu ise ayrıştırma tablosudur. Programlama dilleri için sürücü programın ve ayrıştırma tablosunun elle hazırlanması çok zor olduğundan bunlar otomatik ayrıştırıcı üreteç yazılım araçları tarafından hazırlanır.

1.3.3.1. LR Ayrıştırma Tablosu

Şekil 8'de verilen gramere ait ayrıştırma tablosu Çizelge 2'de gösterilmektedir.

(1) E--> E + T
(2) E-->T
(3) T--> T * F
(4) T--> F
(5) F--> (E)
(6) F--> id

Şekil 8. Örnek Gramer

Bir LR ayrıştırma tablosu 2 kısımdan oluşur; ACTION ve GOTO. Tabloda satır etiketleri durum sembollerini tutarken, sütun etiketleri gramerin sonlandırılmış sembollerini tutar. Ayrıştırma tablosu ayrıştırıcının ne yapması gerektiğini belirler. Tablonun ACTION kısmının 2 ana ayrıştırma eylemi, kaydırma ve indirgemedir. Tablomuzda kaydırmalar "s" ile başlamaktadır ve yığıta ilgili giriş sembolünün ve durum bilgisinin itilmesi gerektiğini belirtir. İndirgemeler ise r harfi ile başlamaktadır ve yığıtın üzerinde bir handlenin bulunduğunu ve bu handlenin yığıttan çekilip yerine sol tarafının itilmesi (indirgeme) gerektiğini belirtir. Tablonun GOTO kısmı ise yığıttan çekilen handlenin yerine sol tarafı itildiğinde ayrıştırıcının hangi duruma geçiş yapacağını belirtir. Bu durum bilgisi de yığıta itilir. Dolayısıyla her zaman yığıtın en üstünde bir durum bilgisi tutulur. ACTION tablosundaki "acc" etiketi ayrıştırma işleminin başarıyla sonlandığını belirtir. Tabloda geri kalan boş bölmeler "hata" durumlarına karşı gelir. Ayrıştırma sırasında bu bölmelerden birine tekabül edilirse giriş stringinin ayrıştırılamayacağına karar verilir.

LR ayrıştırma tabloları elle üretilebilmesine rağmen, gerçek programlama dillerinin grameri için bu durum uzun sürebilir, sıkıcı olabilir ve hata eğilimli olabilir. Gerçek derleyiciler için, LR ayrıştırma tabloları her zaman yazılım araçlarıyla üretilmektedir.

Çizelge 2. Şekil 8'deki Gramere Ait Ayrıştırıcı Tablosu

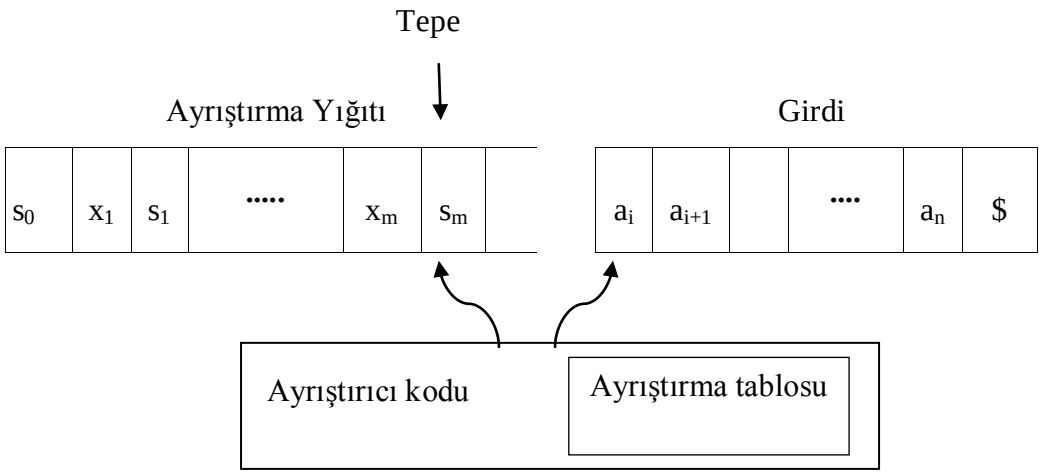
Durum	ACTION						GOTO		
	id	**+**	*****	**(**	**)**	**$**	**E**	**T**	**F**
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

1.3.3.2. LR Ayrıştırma Algoritması

Şekil 9'da ayrıştırma işleminin mantıksal gösterimi sunulmaktadır. Yığıt içeriği $s_o x_1 s_1 x_2$ ... $x_m s_m$ şeklindedir. Buradaki s'ler durum sembollerini, x'ler ise gramer sembollerini göstermektedir. Yığıtın en dibinde başlangıç durumuna ait sembol, en üstünde ise mevcut duruma ait (en son erişilen durum bilgisi) sembol tutulmaktadır. Giriş tamponu $ sembolü ile sonlandırılmış olup bir ip işaretçisiyle indekslenir. İlk başta ip, tampondaki ilk sembole işaret eder. Yığıtın tepe işaretçisinin gösterdiği durum sembolü ve ip işaretçisinin giriş tamponunda gösterdiği sembole göre ayrıştırma tablosu indekslenerek

ayrıştırıcının ne yapması gerektiğine karar verilir. Bununla ilgili algoritma Şekil 10'da verilmektedir.

Aşağıdaki şekil LR ayrıştırıcının yapısını gösterir. Bir LR ayrıştırıcı için ayrıştırma yığıtının içeriği şöyledir:



Şekil 9. Bir LR Ayrıştırıcının Yapısı

Girdinin en sağ tarafında $ işaretinin farkına varalım. Bu işaret ayrıştırıcının oluşturulması esnasında konularak, ayrıştırmanın normal sonlandırılması için kullanılmaktadır. Bu ayrıştırıcı konfigurasyonunu kullanarak, formal olarak ayrıştırma tablosu tabanlı LR ayrıştırıcı algoritması Şekil 10'daki gibidir.

```
ip yi w$ nin ilk sembolu işaret edecek şekilde düzenle
repeat forever begin
s stack in en üstündeki state ve a ip tarafından işaret edilen sembol olsun
if action[s,a] = shift s' then begin
önce a yi sonra s'ni stacke it;
ip yi sonraki sembole işaret edilecek şekilde ilerlet
end
else if action[s,a] = reduce A-->β then begin
2*|β| sembolu stack ten çıkart
s'stack in üstündeki state olsun
önce A yi sonra goto[s',A] yi stack e it
output production A-->β
end
else if action[s,a] = accept then return
else error()
end
```

Şekil 10. LR Ayrıştırma Algoritması

LR ayrıştırma tabloları, grameri girdi kabul eden *Yacc (Johnson, 1975)* gibi yazılım araçları kullanılarak kolaylıkla oluşturulabilmektedir. Yacc için gramerler *BNF*'nin bir çeşidi kullanılarak tanımlanmıştır. Bu teknik John Backus ve Peter Naur öncülüğünde ALGOL60'ı tanımlamak için kullanıldı. Bir BNF gramer içerikten bağımsız dilleri ifade etmek için kullanılabilmektedir. Modern programlama dillerinde çoğu yapı BNF'de temsil edilebilmektedir. CUP, iyi bilinen Yacc ayrıştırıcı üreteç sistemi gibi aynı rolü üstlenmek için tasarlanmıştır, ama C veya C++'dan ziyade tamamen Java'da yazılmıştır ve Java'yla çalışır. Sistemin işleyişindeki ek ayrıntılar ayrıştırıcı üretecinde ve çalışma zamanı kaynak kodda bulunabilir.

CUP orijinal olarak Scott Hudson (Hudson, 1999) tarafından Ağustos 1995'te yazılmıştır. 1996 Temmuz'unda Frank Flannery tarafından daha da geliştirilmiştir. CUP'daki devam eden eklemeler 1997 Aralık'tan günümüze, C. Scott Ananian, tarafından yapılmaktadır.

BÖLÜM 2
ÖNCEKİ ÇALIŞMALAR

Bu bölümde tez çalışmamızda kullandığımız JFlex ve CUP yazılım araçlarını tanıtacağız.

2.1. JFlex'e Giriş

JFlex, Java programlama dilinde yazılı bir otomatik tarayıcı üretecidir. Princeton Üniversitesi'nde Elliot Berk (Klein, 2009) tarafından, çok kullanışlı JLex (Berk, 2003) aracının yeniden yazılmış halidir.

2.1.1. Tasarım Amaçları

JFlex'in ana tasarım amaçları şunlardır (Klein, 2009) :

• Tam unicode desteği
• Hızlı üretilen tarayıcılar
• Hızlı tarayıcı üretimi
• Kullanışlı belirtim sözdizimi
• Platform bağımsızlığı
• JLex uyumluluğu

2.1.2. JFlex'e Genel Bakış

JFlex, bir JFlex dosyasını girdi olarak alır ve tarayıcı kodu olarak bir Java dosyası oluşturur. Zorunlu olmasa da bu JFlex girdi dosyasının uzantısı ".jflex" şeklindedir. Yylex default üretilen sınıfın adı olup, kodu Yylex.java dosyasına yazılmaktadır. Üretilen sınıfın ismi %class direktifi kullanılarak değiştirilebilir. Tarayıcı için iki yapıcı (constructor) oluşturulur. İlki, parametre olarak bir java.io.Reader nesnesi alır. İkincisi ise, parametre olarak bir java.io.InputStream nesnesi kabul eder ve parametre olarak aldığı nesneden java.io.InputStreamReader oluşturarak birincil yapıcıyı çağırır. Parametre, sözcüksel olarak analiz edilecek girdiyi sağlayan dosyaya karşı gelen bir nesnedir. Tarayıcı sınıfı, token sağlayan bir metoda sahiptir. Aksi belirtilmedikçe bu fonksiyonun ismi yylex()'dir. Ancak %function direktifi kullanılarak bu fonksiyonun ismi değiştirilebilir. Yine aksi belirtilmezse token sağlayan metodun döndürdüğü verinin tipi Yytoken'dir. Dönüş tipi %type direktifi kullanılarak değiştirilebilir. Bu metot, düzenli ifadelere girdiyi eşleştirerek eşleşen düzenli ifade için öngörülen eylemi gerçekleştirdikten sonra döngüyü devam ettirir. Eylem bir geri dönüş ifadesi içerirse, metot gösterilen değeri geri döndürür (Hutton, 2006).

2.1.3. JFlex'le Nasıl Çalışılır?

JFlex'in çalışma mantığını göstermek için bu bölümde Java dili için belirtimin bir kısmı sunulmaktadır. Örneğimiz, Java dil yapısının tamamını tanımlamamasına karşın örnek bir JFlex dosyasının hangi belirtimleri içerebileceğini göstermesi açısından önemlidir.

```
/* Java dili için dil sözdizimsel belirtiminin bir kısmı */
import java_cup.runtime.*;
%%
%class Scanner
%implements sym
%unicode
%cup
%line
%column
%{
StringBuffer string = new StringBuffer();
private Symbol symbol(int type) {
return new Symbol(type, yyline, yycolumn);
}
private Symbol symbol(int type, Object value) {
return new Symbol(type, yyline, yycolumn, value);
}
%}
LineTerminator = \r|\n|\r\n
InputCharacter = [^\r\n]
WhiteSpace = {LineTerminator} | [ \t\f]
Identifier = [:jletter:][:jletterdigit:]*
/* comments */
TraditionalComment = "/*" [^*] ~"*/" | "/*" "*"+ "/" /* whitespace */
```

```
%state STRING
%%
/* keywords */
<YYINITIAL> "abstract" { return symbol(sym.ABSTRACT); }
<YYINITIAL> "boolean" { return symbol(sym.BOOLEAN); }
<YYINITIAL> "break" { return symbol(sym.BREAK); }
/* identifiers */
{Identifier} { return symbol(sym.IDENTIFIER); }
/* literals */
{DecIntegerLiteral} { return symbol(sym.INTEGER_LITERAL); }
\" { string.setLength(0); yybegin(STRING); }
/* operators */
"=" { return symbol(sym.EQ); }
{WhiteSpace} { /* ignore */ }
}
<STRING> {
\" { yybegin(YYINITIAL);
return symbol(sym.STRING_LITERAL,
string.toString()); }
[^\n\r\"\\]+ { string.append( yytext() ); }
\\t { string.append('\t'); }
\\n { string.append('\n'); }
\\r { string.append('\r'); }
\\\" { string.append('\"'); }
\\ { string.append('\\'); } }
/* error fallback */
.|\n { throw new Error("Illegal character <"+yytext()+">"); }
/* whitespace */
```

```
{WhiteSpace} { /* ignore */ }
}
<STRING> {
\" { yybegin(YYINITIAL);
return symbol(sym.STRING_LITERAL,
string.toString()); }
[^\n\r\"\\]+ { string.append( yytext() ); }
\\t { string.append('\t'); }
\\n { string.append('\n'); }
\\r { string.append('\r'); }
\\\" { string.append('\"'); }
\\ { string.append('\\'); } }
/* error fallback */
.|\n { throw new Error("Illegal character <"+yytext()+">"); }
```

Bu belirtimden JFlex, tarayıcı için kod içeren bir sınıfla bir Java dosyası üretir. Bu sınıf okunan girdiden bir java.io.Reader alan bir yapıcıya sahip olacaktır. Sınıf ayrıca tarayıcıyı çalıştıran ve girdiden bir sonraki tokenı elde etmede kullanılacak bir yylex() fonksiyonuna sahiptir. (Bu örnekte fonksiyon gerçekten next_token() adını alır çünkü belirtim %cup anahtarını kullanır).

JLex'te olduğu gibi belirtim %% tarafından bölünmüş 3 parçadan oluşur:

```
Kullanıcı Kodu
%%
Seçenekler ve Bildirimler
%%
Sözdizimsel Kurallar ve Eylemler
```

Kullanıcı kodu kısmı dosyanın başından başlayıp %% ile başlayan ilk satıra kadar olan kısım olup, ilgili metin üretilen tarayıcı sınıfına ait kodun başına aynen kopyalanır. Paket tanımlaması ve import ifadeleri dışında burada yapılacak pek bir şey yoktur. Kod

javadoc sınıf yorumuyla biterse, üretilen sınıfa bu yorum eklenir, aksi takdirde JFlex otomatik olarak bir tane oluşturur.

Seçenekler ve bildirimler kısmı ikinci bölümü oluşturmaktadır. Bu bölüm bir takım seçenekleri, üretilecek tarayıcıya ait koda eklenecek kodu, sözdizimsel durumları ve makro tanımlamalarını içerir. Her JFlex seçeneği bir % ile başlayan bir satırla belirtilir. Örneğimizde aşağıdaki seçenekler mevcuttur:

- `%class Scanner`, JFlex'e üretilecek sınıfa "Scanner" adını vermesini ve kodunun "Scanner.java" dosyasına yazmasını söyler.
- `%implements sym`, CUP tarafından üretilen ve CUP dosyasında tanımlanan sonlandırılmış sembollere ait tamsayı sabitlerin tanımlandığı *sym* sınıfını kullanmasını belirtir. Tarayıcı bir token bulduğunda, bu tokena karşı gelen sabiti kullanarak oluşturulan bir Symbol nesnesini ayrıştırıcıya gönderir. Tarayıcıda bulunan token ayrıştırıcıda bir Symbol nesnesi olarak temsil edilir. Symbol sınıfı *java_cup.runtime*'de tanımlıdır. Symbol nesnesi, ait olduğu token'a karşı gelen bir tamsayı yanında o tokenın bulunduğu konumun satır ve sütun numarasını ve varsa tokena ait bir değeri de tutar.
- `%unicode`, tarayıcının üzerinde çalışacağı karakterler kümesini tanımlar. Taranan metin dosyaları için %unicode kullanılmalıdır.
- `%cup`, üretilen tarayıcının CUP tarafından üretilen bir ayrıştırıcıyla çalışacağını belirtir.
  - `%line` satır saymayı aktif hale geçirir (güncel satır numarası yyline değişkeniyle erişilebilir).
- `%column` sütun saymayı aktif hale geçirir (güncel sütun numarası yycolumn değişkeniyle erişilebilir).

`%{...%}` içinde kapsanan kod, üretilen tarayıcı sınıfı kaynak koduna aynen kopyalanır. Burada tarayıcı eylemlerinde kullanılan üye değişkenleri ve fonksiyonlarına ait bildirimler yapılır. Bizim örneğimizde, string literallerin parçalarını depolayacağımız bir `StringBuffer` tipinde bir *string* isimli bir değişken ve bulunan tokenlara bilgileri tutan `java_cup.runtime.Symbol` tipinde nesne oluşturan *symbol* isminde iki yardımcı fonksiyon tanımlanmıştır.

Seçenekler ve bildirimler kısmı makro bildirimlerle devam eder. Makrolar, sözcüksel belirtimlerde kullanılan ve belirtimlerin daha kolay anlaşılmasına yardımcı olan düzenli ifadeler için kısaltmalardır. Bir makro tanımlama bir tanımlayıcı, ardından "=" ve onun

ardından da bir düzenli ifade belirtimi şeklindedir. Bu düzenli ifade başka bir makro içerebilir.

Örnek makrolardan bazılarının ayrıntıları aşağıda belirtilmektedir:

- LineTerminator, bir ASCII CR (Macintosh türevi sistemler için), ASCII LF (Unix türevi sistemler için) veya ASCII CR'yi izleyen LF (Windows türevi sistemler için) ile eşleşen düzenli ifade anlamına gelir.
- InputCharacter, CR veya LF olmayan herhangi bir karakteri tanımlayan düzenli ifade anlamına gelir.
- TraditionalComment, yuvalanmamış yorumları tanımlayan düzenli ifade anlamına gelir.
- Identifier, *jletter* sınıfının bir karakteriyle başlayan ve *Jletterdigit* sınıfının sıfir veya daha fazla sayıda karakterinin tekrarıyla sonlanan stringleri tanımlayan düzenli ifade anlamına gelir.

İkinci bölümün son kısmında durum bildirileri yapılır. Örneğimizde `%state` *STRING*, *STRING* isminde bir durumun bildirimi yapmaktadır. Bir JFlex dosyasında birden fazla durum bildirimi yapılabilir. Sözdizimsel bir durum başlangıç koşulu şeklinde düşünülebilir. Eğer tarayıcı *STRING* durumunda ise aranılan token için sadece başına *STRING* koşulu getirilen düzenli ifadeler göz önünde bulundurulur. *YYINITIAL* ön tanımlı bir durum olup, tarayıcının başlangıçta bulunduğu durumu belirtir.

Şimdi de sözdizimsel kuralların belirtildiği son bölümü kısaca açıklayalım. Bu bölümde düzenli ifadeler ve bu düzenli ifadelerle eşleşen tokenlar tespit edildiğinde yerine getirilen eylemler belirtilir. Örneğimizde tarayıcı *YYINITIAL* durumunda bulduğu token "`abstract`" anahtar kelimesi ise, bu anahtar kelimeye karşı gelen bir *Symbol* nesnesi oluşturularak ayrıştırıcıya geri döndürülür. Yine tarayıcı *STRING* durumunda ve bulunan token " karakteri (Java'da stringler iki " arasındaki karakterlerden oluşur) ise tarayıcı `YYINITIAL` durumuna getirilir ve bulunan tokena (Java string literali) karşı gelen bir *Symbol* nesnesi oluşturularak ayrıştırıcıya geri döndürülür.

2.2. CUP

LL ayrıştırıcılarının bir kısmı özyinelemeli alçalan yöntemiyle elle gerçekleştirilmesine karşın gerek LL ayrıştırıcıların gerekse LR ayrıştırıcıların büyük bir kısmı ayrıştırıcı üreteci tarafından üretilir. Ayrıştırıcı üreteci girdi olarak ayrıştırılacak metinlerin yazım kurallarının tanımlandığı bir dosyayı girdi olarak alır ve çıktı olarak bir ayrıştırıcı kodu üretir. Ayrıştırıcı kodu C, C++ veya Java gibi yüksek seviyeli bir dilde

oluşturulur.

Biz bu çalışmamızda CUP (Hudson,1999) ayrıştırıcı üretecini kullandık. Diğer ayrıştırıcı üreteçlerinden bazıları Yacc, Bison (Donnelly ve ark., 2011), JavaCC (Anonim, 2011) ve ANTLR (Parr, 2007)'dir.

CUP, basit belirtimlerden otomatik olarak LALR ayrıştırıcısı üreten derleyicilere yönelik bir yazılım aracıdır. CUP, belirtimleri içeren bir dosyayı girdi olarak alır ve bu belirtimlere uygun ayrıştırıcıyı otomatik olarak üretir. Üretilen ayrıştırıcı kodu Java programlama dili temellidir.

2.2.1. Belirtim Sözdizimi

Bir CUP belirtim dosyası temel olarak 5 kısımdan oluşur. Bunlar sırasıyla,

- paket ve import belirtimleri,
- kullanıcı kod bileşenleri,
- sembol listeleri (sonlandırılmış ve sonlandırılmamış semboller),
- öncelik bildirimleri, ve
- gramer.

olup, her birinin kısa açıklaması aşağıda verilmektedir.

2.2.2. Paket ve Import Belirtimleri

CUP dosyası `package` ve `import` belirtimleriyle başlar. Bu belirtimler zorunlu olmayıp, gerek duyulursa dosyaya konur. Bunların belirtimleri Java'daki gibi olup, normal bir Java programında bulunan `package` ve `import` bildirimleri ile aynı role sahiptir.

Paket bildirim, CUP tarafından oluşturulan `sym` ve `parser` sınıflarının konacağı paketi (bağıl yolu) belirler.

2.2.3. Kullanıcı Kod Bileşenleri

İsteğe bağlı `package` ve `import` bildirimlerinin ardından yine isteğe bağlı kod bileşenleri kısmı yer alır. Bu kısımda oluşturulacak olan ayrıştırıcı koduna eklenecek olan kullanıcı kodu yer alır. Bu bölüm bir dizi bildirimler şeklindedir. Gramere gömülen kodda kullanılan rutinler ve değişkenler bu bölümde belirtilir.

Bu bölümdeki ilk belirtim `action code {: ... :};` şeklindedir. `{: ... :}`, `action class` sınıfına ait koda eklenecek kod parçasını belirtir.

`action code` bildiriminden sonra isteğe bağlı `parser code {: ... :};` bildirimi yer alır. Bu bildirimde yer alan kod oluşturulan ayrıştırıcı koduna eklenir.

Bu bölümdeki üçüncü bildirim `init with {: ... :};` olup bu bildirimde belirtilen kod ilk token isteğinde bulunmadan önce ayrıştırıcı tarafından çalıştırılır.

Bu bölümün son bildirimi `scan with {: ... :};` şeklinde olup, ayrıştırıcının tarayıcıdan sonraki token isteğinin nasıl olacağını belirtir.

CUP 0.10j ten itibaren action code, parser code, init code, ve scan with bölümleri herhangi sırada görünebilir. Ancak onların sembol listelerinden önce olması gerekir (Hudson, 1999).

2.2.4. Sembol Listeleri

Belirtim dosyasının ilk zorunlu bölümü sembol listelerinin yer aldığı bölümdür. Bu bölüm kullanıcı kod bileşenlerinin yer aldığı bölümden sonra gelmelidir. Buradaki bildirim listeleri, gramerde tanımlı sembollerin sonlandırılmış ve sonlandırılmamış sembol olarak etiketlenmesinden ve sembollerin tiplerinin belirlenmesinden sorumludur. Sonlandırılmış semboller ayrıştırıcı isteği üzerine tarayıcı tarafından bulunup ayrıştırıcıya döndürülür. Her bir sonlandırılmış sembol bir *`Symbol`* nesnesi ile temsil edilir. Eğer sonlandırılmış sembolün bir değeri varsa (örneğin tamsayı literali) bu değer tarayıcı tarafından ilgili nesnenin *`value`* değişkenine konur. Sembol listesi bildirimleri aşağıdaki gibi yapılır.

```
terminal classname name1, name2, ...;
non terminal classname name1, name2, ...;
terminal name1, name2, ...;
```

2.2.5. Öncelik ve Birleşirlik (Associativity) Bildirimi

İsteğe bağlı olan bu bölümde sonlandırılmış sembollerin öncelik ve birleşirlikleri belirlenir. Öncelik ve birleşirlik bildirimi belirsiz (ambiguous) gramerler ile ayrıştırmada faydalı olur. Öncelik ve birleşirlik bildirimi için 3 tip vardır:

```
precedence left     terminal[, terminal...];
precedence right    terminal[, terminal...];
precedence nonassoc terminal[, terminal...];
```

İlk bildirim ilgili listedeki sonlandırılmış sembollerin soldan öncelikli olduğunu, ikinci bildirim sağdan öncelikli olduğunu ve sonuncu bildirim ise birleşirliklerinin olmadığını belirtir. Sonlandırılmış semboller arasındaki öncelik sırası ise bildirim sırasının tersidir. Yani bildirim sırasının en üstündeki listedeki sonlandırılmış semboller en az öncelikli, bildirim sırasının en altındaki listedeki sonlandırılmış semboller ise en yüksek önceliğe sahiptir. Dolayısıyla aşağıda verilen iki bildirime göre şunları söyleyebiliriz.

```
precedence left  ADD, SUBTRACT;
precedence left  TIMES, DIVIDE;
```

Toplama, çıkarma, çarpma ve bölme operatörlerinin her biri soldan önceliklidir. Toplama ile çıkarmanın ve çarpma ile bölmenin önceliği aynıdır. Çarpma ve bölmenin önceliği hem toplamadan hem de çıkarmadan daha yüksektir.

Öncelik bildirimleri ayrıştırma esnasında kaydırma mı yoksa indirgeme mi yapılacağının karar verilmesini sağlar. Eğer önceliği yüksek olan bir sonlandırılmış sembol yığıta atılmış ise daha düşük öncelikli sonlandırılmış sembol yığıta atılmadan önce yüksek öncelikli sonlandırılmış sembolün belirlediği işlem yapılır (indirgeme). Eğer yığıta atılan sonlandırılmış sembolden daha yüksek öncelikli sonlandırılmış sembole rastlanırsa bu durumda yüksek öncelikli sonlandırılmış sembol yığıta itilir (kaydırma). CUP aynı zamanda her bir türetim kuralına bir öncelik atar. Türetim kuralının önceliği o kuralda yer alan en son sonlandırılmış sembolün önceliğine eşittir. Birleşirlik aynı zamanda `kaydırma/indirgeme çatışması` (shift/reduce conflict) çözümünde kullanılır. Eğer kaydırılan sonlandırılmış sembolün önceliği indirgenen kuralın önceliğinden büyükse bu durumda kaydırma işlemine, aksi takdirde indirgeme işlemine başvurulur.

2.2.6. Gramer

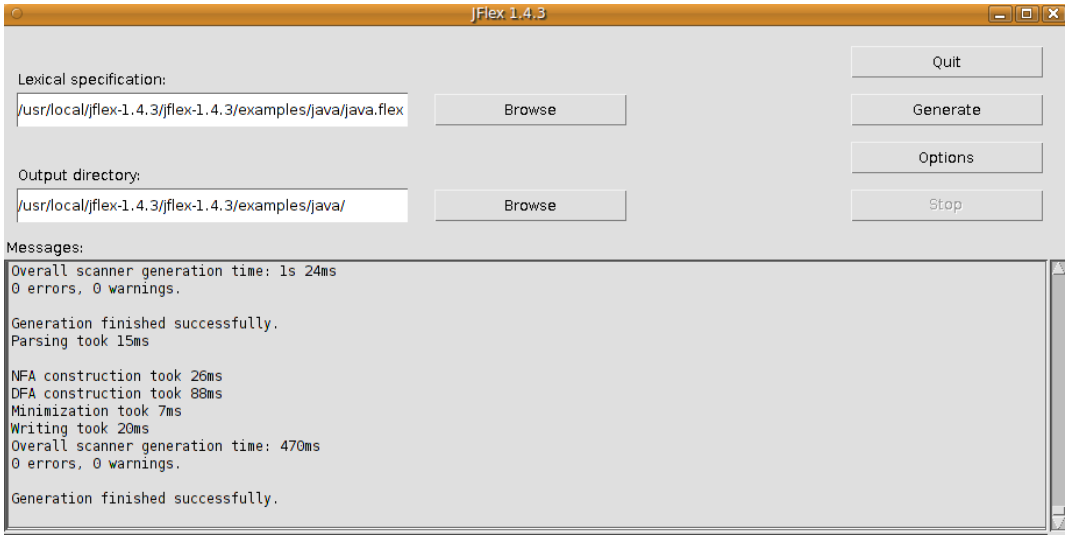
CUP dosyasının son bölümü dilin gramer tanımlamasıdır. Bu tanımlama CFG şeklindedir. Bu bölüm isteğe bağlı olarak `start with non-terminal;` biçimindeki belirtimle başlar. Bu belirtim, dilin başlangıç sembolü olan sonlandırılmamış sembolü belirtir. Eğer bir başlangıç sonlandırılmamış sembolü bildirimi açıkça yapılmazsa bu durumda dilin başlangıç sembolünün ilk üretim kuralının sol tarafındaki sonlandırılmamış sembol olduğu kabul edilir. Başarılı bir ayrıştırma işleminin sonunda, CUP, *`java_cup.runtime.Symbol`* tipinde bir nesne geri döndürür. Bu nesne ayrıştırma işlemi sonucunda oluşturulan ayrıştırma ağacını temsil eder. Ayrıştırma ağacı daha sonraki derleme aşamalarına girdi teşkil eder. Örneğin, bu ağaç dolaşılmak suretiyle tip uyuşmazlığı, değişkenlere tanımlanmadan önce bir değer atanıp atanmadığı gibi mantıksal hata kontrollerinde kullanılır. Hata kontrolü aşamasından sonra yine ayrıştırma ağacına bakılarak ara kod üretimi yapılır. Üretim kurallarının nasıl tanımlanacağı ve bir üretim kuralı için indirgeme yapıldığında ayrıştırma ağacının bir düğümüne karşı gelecek olan bir nesnenin nasıl oluşturulacağına ilişkin bilgiler sonraki bölümde açıklanacaktır..

BÖLÜM 3
MATERYAL VE YÖNTEM

3.1. JFlex ve CUP Kullanımı

Çalışmamızda JFlex sözcüksel analiz üretecini ve CUP ayrıştırıcı üretecini nasıl kullandığımıza ve ayrıştırma ağacını nasıl oluşturduğumuza bakalım. JFlex'ten Java için bir tarayıcı elde etmek için konsoldan şu komutu yazabiliriz:

`java /../jflex-1.4.3.tar.gz_FILES/jflex-1.4.3/lib/JFlex.jar JFlex.Main java.flex` veya grafik arayüzü kullanarak JFlex'i çalıştırdığımızda karşımıza aşağıdaki şekilde görüldüğü gibi bir ekran gelecektir. Burada `Lexical specification` alanında *`java.flex`* dosyasını gösteriyoruz ve `Generate` butonuna basıyoruz. Eğer *`java.flex`* dosyasında herhangi bir hata yoksa aşağıdaki gibi bir görüntü elde edilerek Scanner.java dosyası oluşturulmaktadır.



Şekil 11. JFlex'in Çalıştırılması

Scanner dosyamızı elde ettikten sonra ayrıştırıcı dosyasını ve sembol dosyası olan sym.java'yı elde etmek için CUP ayrıştırıcı üretecini kullanacağız. Bunun için konsola şu komutu yazıyoruz :

`java -classpath /.../java_cup_v10k java_cup.Main<java15.cup`

`java15.cup` dil kurallarının tanımlı olduğu dosyadır. Eğer herhangi bir hata yoksa parser.java ve sym.java olmak üzere 2 dosya oluşturulmaktadır.

3.2. JFlex ve CUP Dosya İçerikleri

Gerek JFlex dosyası içeriği gerekse CUP dosyası içeriği çok geniş kapsamlı olduğundan biz bu bölümde ilgili dosyaların önemli gördüğümüz kısımlarını açıklayacağız. JFlex dosyasının içeriği hakkında bilgi bir önceki bölümde verilmişti. Ayrıca CUP dosyasındaki sonlandırılmamış semboller sayısı ile bunlara karşı gelen bizim yazdığımız sınıfların sayısı çok olduğundan biz burada sadece CUP dosyasında yazılı olan türetim kurallarının bazılarını sınıf kodlarıyla birlikte açıklamaya çalışacağız.

`java15.cup` dosyamızın içeriğine bakacak olursak aşağıdaki gibi `package` ve `import` bildirimleri ile başlamaktadır ve bu kısım oluşan `parser.java` dosyasının başına kopyalanmaktadır.

```
package Parse;
import java_cup.runtime.*;
parser code  {:
  public void report_error(String message, Object info) {
    StringBuffer m = new StringBuffer("Error: ");
    if (info instanceof java_cup.runtime.Symbol)
      m.append( "("+info.toString()+")" );
      m.append(" : "+message);
    System.out.println(m);
  }
  public void report_fatal_error(String message, Object info) {
    report_error(message, info);
    throw new RuntimeException("Fatal Syntax Error");  }
:};
```

Şimdi de `parser code{: ... :}` kısmına bir göz atalım. Bu kısımda belirtilen kod parçacığı, ayrıştırma esnasında bir hata tespit edildiğinde hata konusunda uyarı verilmesini sağlamaktadır. Bu belirtimin ardından sonlandırılmış sembollerin tanımlandığı kısım yer almaktadır. JFlex dosyasında tanımlanan tokenlar CUP dosyasında sonlandırılmış birer sembol (terminal) olarak temsil edilmektedir. JFlex dosyasında tanımlı her bir tokena karşı CUP dosyasında bir sonlandırılmış sembol tanımı yapılmalıdır. Aşağıdaki koddan da görülebileceği gibi sonlandırılmış bazı sembollerin değeri de olabilir. Örneğin `IDENTIFIER` sonlandırılmış sembolünün bir değeri olup, değer tipi `java.lang.String`'dir. Oysa `DOT` sonlandırılmış sembol olmasına rağmen bir değeri yoktur. O yüzden CUP dosyasında bir tip bildirimi de verilmemiştir.

Tarayıcı bir token tespit ettiğinde bu tokena karşı gelen `Symbol` sınıfından bir nesne oluşturularak ayrıştırıcıya gönderilir. Her `Symbol` nesnesinin kaç numaralı tokena karşı geldiğini gösteren bir `type` alanı vardır. Tokenların numaraları ayrıştırıcı üreteci tarafından oluşturulan `sym.java` dosyasında tanımlanmaktadır. Symbol tipinde bir nesnenin ilgili tokenun satır numarasını (`left`) ve sütun numarasını (`right`) temsil eden alanları mevcuttur. Ayrıca değere sahip bir tokenun (örneğin `IDENTIFIER`) değerinin tutulduğu `value` diye de bir alanı vardır.

CUP dosyamızda bazı sonlandırılmış semboller için şu belirtimler yer almalıdır.

```
terminal java.lang.String IDENTIFIER;
terminal DOT, MULT, SEMICOLON;
terminal IMPORT; // import_declaration
terminal PACKAGE; // package_declaration
```

CUP dosyamızda sonlandırılmış sembollerin belirtiminden sonra sonlandırılmamış sembollerin (non terminal) belirtimi yer almaktadır:

```
non terminal goal;
non terminal ID name,simple_name, qualified_name;
non terminal irCompUnit compilation_unit;
non terminal irPackage package_declaration_opt,package_declaration;
non terminal irImport import_declarations_opt, mport_declarations;
non terminal TypeDec type_declarations_opt, type_declarations;
non terminal ClassDec class_declaration;
non terminal TypeDec interface_declaration;
non terminal EnumDec enum_declaration;
non terminal irImport import_declaration;
non terminal irImport single_type_import_declaration;
non terminal irImport type_import_on_demand_declaration;
non terminal TypeDec type_declaration;
non terminal irImport static_single_type_import_declaration;
non terminal irImport static_type_import_on_demand_declaration;
```

Her `non terminal` anahtar kelimesinin ardından sonlandırılmamış sembolün tipi belirtilmelidir. Bu tipler, sonlandırılmamış ifadeler için bizim oluşturduğumuz sınıflardır. Örneğin bizim `compilation_unit` sonlandırılmamış sembolümüzün eylem kod parçacığı (action) geriye bir `irCompUnit` nesnesi döndüreceğinden `irCompUnit` sembolünün tipini `irCompUnit` olarak belirtmeliyiz. Varsayılan geri dönüş tipi `Object`'tir. Farklı tiplere sahip olan bazı sonlandırılmamış semboller aynı tip nesne geri

döndürmek zorunda kalabilir. Bu durumda ortaya çıkan tip uyuşmazlığı problemini adaptör sınıflar oluşturarak ya da arayüz sınıflar oluşturarak çözdük.

Yukarıda olduğu gibi aynı tipe sahip olan sonlandırılmamış sembolleri de aralarına virgül koyarak tek bir satırda yazabiliriz.

CUP dosyasında sonlandırılmamış sembol bildirimleri sona erdikten sonra bu bildirimlerin sözdizimsel gramer (Syntactic Grammar) kısmı başlamaktadır. Her bir gramer kuralı, CFG'nin bir kuralına karşı gelir. Örneğimizde, gramerimizin başlangıç sembolü "goal" olarak belirtilmiştir. Her türetim kuralının bir aksiyon kısmı vardır. Kuralın aksiyon kısmında ilgili kural için yapılması gerekenler belirtilir.

```
start with goal;
goal ::= compilation_unit:cu{:
                    RESULT=cu;
                   :};
compilation_unit::= package_declaration_opt:pd
import_declarations_opt:impdec
     type_declarations_opt:tdec
          {:
            RESULT=new irCompUnit(pd,impdec,tdec,null);
          :};
```

Burada compilation_unit sonlandırılmamış sembolü; package_declaration_opt, import_declarations_opt ve type_declarations_opt sonlandırılmamış sembollerinden oluşmaktadır. ::='in sağ tarafındaki sembollere :'dan sonra daha kısa isimler vererek ilgili sembole karşı gelen değeri veya nesneyi temsil edebiliriz. Örnekte pd, impdec ve tdec bizim verdiğimiz yeni isimlendirmeler olup sırasıyla package_declaration_opt, import_declarations_opt ve type_declarations_opt sonlandırılmamış sembollere karşı gelen nesneleri temsil etmektedirler. Bu isimlendirmeler RESULT için oluşturulan nesneye parametre teşkil ederler.

Aşağıda irCompUnit.java sınıfına ait kod verilmektedir.

```
Public class irCompUnit exteds IR {
      irPackage pack; irImport imprt; TypeDec typeDec;
      irCompUnit next;

      public irPackage getPack() {
            return pack;     }
      public irImport getImprt() {
            return imprt;
            }
      public TypeDec getTypeDec() {
            return typeDec;  }
```

```
	public irCompUnit getNext() {
		return next;	}
	public void setNext(irCompUnit next) {
		this.next = next;
		}

	public irCompUnit(irPackage pack, irImport imprt, TypeDec tdec,
						irCompUnit next){
		super();
		this.pack = pack;
		this.imprt = imprt;
		this.typeDec = tdec;
		this.next = next;
		}

	public void visit(BufferedWriter out) throws IOException {
	if (this.getPack() != null) {
		this.getPack().visit(out);
		}
	if (this.getImprt() != null) {
		this.getImprt().visit(out);
		}
	if (this.getTypeDec() != null) {
		this.getTypeDec().visit(out);
		}
	if(this.getNext()!=null)
	{   getNext().visit(out);
	}
}
```

`irCompUnit` sınıfına bakarsak, `IR` sınıfından türetildiğini görürüz. Bu sınıf, `package_declaration_opt`'un tipi olan `irPackage`, `import_declarations_opt`'ın tipi olan `irImport`, `type_declarations_opt`'ın tipi olan `TypeDec` ve kendi türünden `next` değişkenini parametre kabul eden bir yapıcıya sahiptir. `visit(BufferedWriter out)` metodu test amacına yönelik olup alanları kontrol ederek eğer `null` dışında bir değere sahipse rekürsif olarak o alanın visit metodunu çağırmaktadır

Bütün sınıflar `IR` soyut sınıfından türetilmektedir. IR, satır ve sütunu temsil eden, `line` ve `col` olmak üzere 2 özellik içerir. Bunlar ilgili nesneye karşı gelen yapıya ait terminallerden birinin satır ve sütün numaralarına karşı gelir. Bu bilgiler ayrıştırma hatasının giriş dosyasında hangi pozisyona tekabül ettiğinin belirlenmesinde kullanılır.

```
public abstract class IR {
	int line,col;
	public IR() {
		this(0, 0);
		}
	public IR(IR initFromThisOne) {
		this(0, 0);
		if (initFromThisOne != null) {
			this.line = initFromThisOne.line;
			this.col = initFromThisOne.col;
			}
```

```
            }
      public IR(int line, int col) {
            super();
            this.line = line;
            this.col = col; }
            }
```

```
package_declaration_opt ::= package_declaration:pd
              {:
               RESULT=pd;
         :}|
              {:
               RESULT=null;
         :};
package_declaration ::= PACKAGE name:nm SEMICOLON
         {:
           RESULT=new irPackage(nm);
     :};
```

compilation_unit sonlandırılmamış sembolüne indirgenen sembollerden ilki olan *package_declaration_opt'*a bakacak olursak, ya *package_declaration* değerine sahip olacağını ya da değer almayacağını görürüz. Eğer *package_declaration* değerine sahipse eylem kodunda, *package_declaration* eylem kodunda oluşturulan *irPackage* nesnesi geri döndürülür. Görüldüğü gibi *package_declaration* sonlandırılmış sembol PACKAGE, sonlandırılmamış sembol *name* ve sonlandırılmış sembol SEMICOLON'dan oluşmaktadır. Daha somut bir şekilde ifade edecek olursak bu ifade Java dosyalarındaki paket bildiriminin tanımlandığı yerdir ve **package** parserapplication; tanımlamasında PACKAGE package'a, name parserapplication'a ve SEMICOLON ise ";"a karşı gelmektedir. irPackage sınıfımız aşağıdaki gibidir.

```
public class irPackage extends IR {

     ID initial_dir;

     public irPackage(ID initial_dir) {
          super(initial_dir);
          this.initial_dir = initial_dir;
     }

     public ID getInitial_dir() {
          return initial_dir;
     }
     public void visit(BufferedWriter out) throws IOException {
          out.write("package ");
          if (getInitial_dir() != null) {
               getInitial_dir().visit(out);
          }
          out.write(";" + "\n");
     }
}
```

`irPackage` sınıfının yapıcı fonksiyonu sadece bir parametre almaktadır. Burada dikkat etmemiz gereken nokta, CUP dosyasında `name` sonlandırılmamış bir sembol olarak belirtilirken bu sembole karşı oluşturulan nesnenin `irPackage` sınıfında `ID` tipinde `initial_dir` adlı değişkenle saklanmasıdır. CUP dosyasında `name` sonlandırılmamış sembolün tipi ile `irPackage` sınıfında tanımlı `initial_dir` değişkeninin tipinin aynı olup `ID` olduğuna da dikkat edelim. Sınıfta tanımlı `visit` metodu bir `irPackage` nesnesinin dosyaya döküm kodlarını içermektedir. Bu, önce dosyaya "package" anahtar kelimesi yazdırılmakla, sonra `ID` değişkeninin `visit` metodu çağrılarak ilgili paket yolunun yazdırılması ve en sonunda da bir noktalı virgül ";" yazdırılmakla gerçekleştirilir.

`compilation_unit` sonlandırılmamış sembolünü tanımlayan sembollerden ikincisi olan `import_declarations_opt` şu şekildedir:

```
import_declarations_opt ::= import_declarations:idec
              {: RESULT=idec;
            :}|
              {: RESULT =null;
            :};

import_declarations ::= import_declaration:impdec
          {:    RESULT=impdec;        :}   |

          import_declarations:i1 import_declaration:i2
          {:
           irImport i3=i1;
           while(i3.getNext()!=null) {
                  i3=i3.getNext();
                        }
           i3.setNext(i2);
           RESULT=i1;
             :};

import_declaration ::=
          single_type_import_declaration:s{: RESULT=s;
      :}  |

          type_import_on_demand_declaration:s{: RESULT=s;
      :}  |

          static_single_type_import_declaration:s{: RESULT=s;
      :}  |

          static_type_import_on_demand_declaration:s{: RESULT=s;
       :};

single_type_import_declaration ::=
          IMPORT name:nm SEMICOLON     {:
           RESULT= new irImport(nm,irImport.SINGLE_TYPE,null);
      :};
```

```
type_import_on_demand_declaration ::=
          IMPORT name:nm DOT MULT SEMICOLON{:
           RESULT= new irImport(nm,irImport.ON_DEMAND_TYPE,null);
      :};

static_single_type_import_declaration ::=
 IMPORT STATIC name:nm SEMICOLON{:
           RESULT = new irImport(nm,irImport.SINGLE_STATIC,null);
      :};

static_type_import_on_demand_declaration ::=
          IMPORT STATIC name:nm DOT MULT SEMICOLON{:
          RESULT= new irImport(nm,irImport.DEMAND_STATIC,null);
     :};
```

compilation_unit sonlandırılmamış sembolünü tanımlayan sembollerin sonuncusu olan type_declarations_opt, boş olmadığı durumda 4 alternatiften biri olabilmektedir. Bu dört alternatifin de tipi aynı olmalıdır. Bunun için class_declaration, enum_declaration, interface_declaration için oluşturulan sınıfların hepsini ortak bir arayüzden (TypeDec) gerçekleştirdik.

```
type_declaration ::=
          class_declaration:c{:     RESULT =c;   :}

     |    enum_declaration:e{:      RESULT =e;   :}
     |    interface_declaration:i{: RESULT =i;   :}
     |    SEMICOLON{:               RESULT =null;    :};
```

TypeDec arayüz sınıfımızın içeriği de şu şekildedir:

```
public interface TypeDec {
     TypeDec getNext();
     void setNext(TypeDec td);
     public void visit(BufferedWriter out) throws IOException ;
}
```

Şimdi de name, simple_name, qualified_name sonlandırılmamış sembollerine bakalım:

```
name ::=  simple_name:sn     {: RESULT=sn; :}
     |    qualified_name:qn {:  RESULT=qn; :};

simple_name ::=   IDENTIFIER:id
          {: RESULT = new ID(idleft,idright,id,null); :};

qualified_name ::=
          name:nm DOT IDENTIFIER:id
          {:
                  ID bid=nm;
           while(bid.getNext()!=null) {
               bid=(ID)bid.getNext();
          }
           bid.setNext(new ID(idleft,idright,id,null));
           RESULT=nm;
          :};
```

name için 2 alternatif olduğunu görürüz. simple_name sadece bir ismin belirtimine karşı gelir ve bu durumda bir ID nesnesi oluşturulur. qualified_name ise bir noktayla birbirinden ayrılan birden fazla ismin belirtildiği duruma karşı gelir. Bu durumda ise her bir isme karşı bir ID nesnesi oluşturulur. Bir isme erişebilmeyi sağlamak için o isimden bir önceki isme karşı gelen ID nesnenin next işaretçisinden yararlanılır.

`ID` sınıfımıza bakacak olursak `String` tipinde bir değişken ile `irExpr` tipinde `next` değişkenlerini parametre kabul eden yapıcıya sahiptir.

```
public class ID extends irExpr {

     String name;
     irExpr next;
public ID(int line, int col, String name, ID next) {

          super(line, col);
          this.name = name;
          this.next = next;
     }
```

3.3. Ana Program

Ana programımızın çalışması şu şekildedir:

```
public class Main {//Programda kullandığımız genel değişkenler

     public static String filename;
     public static String filename_out;
     public static PrintWriter out;
     public static boolean syntax_error;

     public static Program program;

     public static void main(String argv[]) throws IOException {
//Programda kullandığımız yerel değişkenler
          int i;
          Scanner scanner;
          parser parser;
          Symbol parse_result;
          irCompUnit compilation_unit, last_compilation_unit;
          FileReader input;
//programın parametre alıp almadığını kontrol ediyoruz.
          if (argv.length == 0) {
               System.out
                         .println("**Error: Usage: file1.java [file2.java
...] . ");
               System.out.println();
               System.out.println("Halting ...");
               return;
          }

          syntax_error = false;
          last_compilation_unit = null;
          parse_result = new Symbol(0, 0, 0, null);

          System.out.println("Starting parse...");
//Bu kısım birden fazla dosyanın taranabilmesini sağlar.
          for (i = 0; i < argv.length; i++) {
               filename = argv[i];
               try {
                    input = new FileReader(filename);
               } catch (FileNotFoundException e) {
          System.out.println("***Error: Unable to open input file "
                              + filename + ".");
                    System.out.println();
```

```
                    System.out.println("Halting...");
                    return;
               }

               /**** parse file... *****/
               System.out.println("Parsing " + filename + "...");
               scanner = new Scanner(input);
               parser = new parser(scanner);


// Ayrıştırma işleminin gerçekleştiği ve herhangi bir hata olması
durumunda hata ile ilgili uyarıcı mesaj sağlayan bölüm burasıdır.
                try {
                    parse_result = parser.parse();
               } catch (Exception e) {

          System.out.println("***Parser Error: " + e.getMessage());
                    syntax_error = true;
                              }
//Hata varsa dosya kapatılır ve programdan çıkılır.
               if (syntax_error) {
                    try {
                         input.close();
                    } catch (IOException e) {
                         /* ignore */
                    }
                    return;
               }
               compilation_unit = (irCompUnit) parse_result.value;

               if (program == null)
                    program = new Program(compilation_unit);
               else
                    last_compilation_unit.setNext(compilation_unit);

               last_compilation_unit = compilation_unit;

               try {
                    input.close();
               } catch (IOException e) {
                    e.printStackTrace();
               }
          }

          System.out.println("Starting dumping parse tree...");
          compilation_unit = program.units;
//Taranan dosyalar sonucunda elde edilen ayrıştırma ağaçları için
//dosyalar bu kısımda oluşturulmaktadır.
          i = 0;
          while (compilation_unit != null) {
     filename_out = argv[i].substring(0, argv[i].indexOf(".java"));
     filename_out = filename_out + "_out.java";

          try {
                    FileWriter fstream = new FileWriter(filename_out);
                    BufferedWriter out = new BufferedWriter(fstream);
                     compilation_unit.visible(out);
                     out.close();
               } catch (FileNotFoundException e) {
          System.out.println("***Error: Unable to open input file "
                              + filename_out + ".");
```

```
                System.out.println("Halting...");
                return;
            }
        System.out.println("Dumping into " + filename_out + " file");
        i++;
        compilation_unit = compilation_unit.getNext();
        }
```

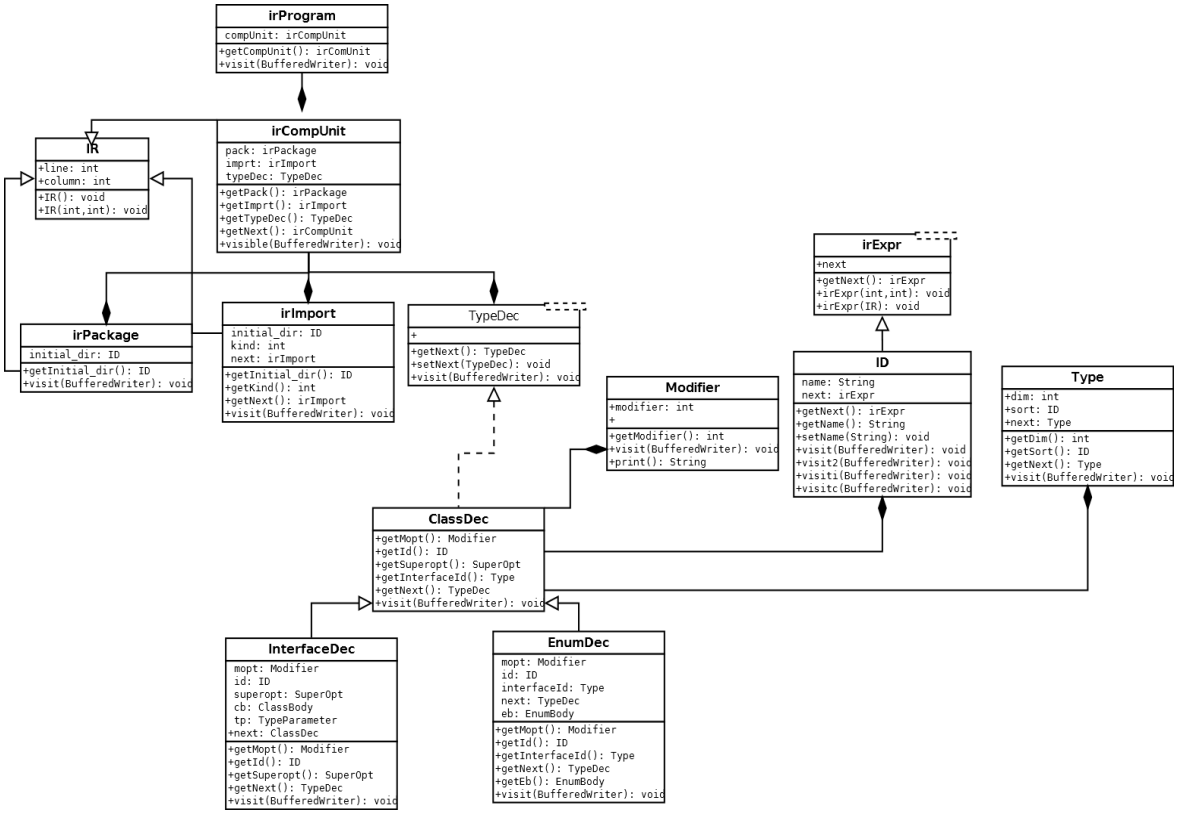
Programımız birden fazla Java dosyasını ayrıştırabilmekte ve bunlara ait üretilen ayrıştırma ağaçlarını birleştirerek çıktı olarak tek bir ayrıştırma ağacı üretmektedir. Programın parametre alıp almadığı kontrol edilmektedir ve parametre almadığı tespit edilirse bir uyarı mesajı yazdırılarak programdan çıkılması sağlanmaktadır. Burada parametre girdi olarak verdiğimiz Java kaynak kodları içeren dosyalardır. Eğer programa en az bir Java dosyası girdi olarak verilirse ayrıştırma işlemi bir döngü içerisinde yapılmaktadır. Ayrıştırılacak her bir dosya önce açılır, daha sonra açılan dosyadan tarayıcı sınıfının yapıcısı olan `Scanner` çağrılıp tarayıcı nesnesi oluşturularak `scanner` değişkenine atanır. Bunu izleyen adımda `scanner` değişkeni ayrıştırıcı sınıfının `Parser` yapıcısına parametre olarak gönderilerek oluşturulan ayrıştırıcı nesnesi `parser` değişkenine atanır. `parser` değişkeninin işaret ettiği nesnesinin `parse()` metodu çağrılarak ilgili Java dosyasının ayrıştırma işlemi yapılır. Ayrıştırma esnasında bir hata bulunursa ilgili dosya kapatılarak, ayrıştırma işlemi sonlandırılıp programdan çıkılır. Ayrıştırma işlemi başarılı ise ilgili Java dosyasının ayrıştırma işlemi sonucunda elde edilen ayrıştırma ağacı `compilation_unit` değişkenine atanır. Ayrıştırılan ağaçlar birbirlerine bağlanmak suretiyle ayrıştırma işleminin bütününe karşı gelen ayrıştırma ağacı `Program` sınıfının yapıcısına parametre olarak gönderilerek `Program` sınıfına ait bir nesne oluşturulur. Oluşturulan bu nesnenin adresi `program` değişkeninde tutulur.

Ayrıştırma işleminden sonra `while` döngüsüyle ayrıştırma ağacı dolaşılarak ayrıştırma ağacına karşı gelen Java kodları dosyalara dökülür. Bu süreçte ayrıştırılan her bir Java dosyasına karşılık diskte yeni bir Java dosyası oluşturulur. Oluşturulan her bir Java dosyası ayrıştırma işlemine girdi teşkil eden Java dosyasıyla kıyaslanarak ayrıştırma işleminin doğru yapılıp yapılmadığının kontrolü yapılır.

3.4. Bazı Sınıflar için UML Sınıf Diyagramı

Ayrıştırma işleminde kullanılan yaklaşık 100 farklı sınıf kodu yazdığımızdan sınıfların tümü arasındaki ilişkiyi göstermek tezde çok yer kaplayacaktı. Bu yüzden burada önemli gördüğümüz sınıflar arasındaki ilişkileri göstermeye çalıştık. Şekil 12'de ayrıştırma işleminde kullanılan bazı sınıflara ait UML sınıf diyagramı gösterilmektedir

Ayrıştırma ağacımızın en üst seviyesinde Program sınıfı bir irCompUnit nesnesi içerir. İçi dolu ok, sınıflar arasındaki içerme (composition) ilişkisini, içi boş ok türeme ilişkisini ve içi boş kesikli ok gerçeklenme (implement) ilişkisini gösterir.



Şekil 12 . Java 1.5 Grameri İçin Oluşturduğumuz Ayrıştırma Ağacını Oluşturan Programın UML Sınıf Diyagramı

BÖLÜM 4
SONUÇLAR VE ÖNERİLER

Ayrıştırıcı, verilen bir yazıyı analiz ederek yazının verilen gramer kurallarına uygun yazılıp yazılmadığını kontrol eden bir bilgisayar programıdır. Burada bahsi geçen yazı herhangi bir doğal dilde yazılı bir metin olabileceği gibi herhangi bir programlama dilinde yazılı olan bir program kodu da olabilir. Ayrıştırıcı aynı zamanda eğer yazıda bir yazım hatası varsa bu hatanın yerini bularak nasıl düzeltilebileceği konusunda da kullanıcıya tavsiyede bulunur. Eğer yazıda hata yoksa ayrıştırıcı, yazıyı bilgisayar belleğinde uygun bir şekilde temsil etmek için bir veri yapısına dönüştürür. Bu veri yapısı genel olarak ayrıştırma ağacı olarak bilinir.

Ayrıştırıcıların bilgisayar biliminde birçok kullanım alanları vardır. Yüksek seviyeli bir programlama dilinde yazılı bir programı makine diline dönüştüren derleyiciler ayrıştırıcıyı ara bir adım olarak kullanırlar. Makine dili kodlarını inceleyerek yüksek seviyeli programlama diline dönüştüren derleyiciler de ayrıştırıcıdan faydalanabilirler. Yine ayrıştırıcılar doğal dil işlemede kendilerine kullanım alanı bulurlar. Ayrıca, karmaşık verilerin analiz edilerek bellekte etkin bir biçimde saklanarak işlenmesi ayrıştırıcılar yardımıyla olmaktadır.

Bu tezde Java programlama dili için bir ayrıştırıcı tasarlanarak programı yazıldı. Yazdığımız ayrıştırıcı için CUP otomatik ayrıştırıcı üretecini kullandık. CUP, aşağıdan yukarıya doğru ayrıştırıcı ağacı üretmeye yarayan ve LALR temelli bir yazılım aracıdır. LALR ayrıştırıcıları neredeyse tüm programlama dillerinde yazılan programları ayrıştırabilirler. Oldukça hızlı çalışan LALR temelli ayrıştırıcılar ayrıştırma yeteneği açısından LR(1) ayrıştırıcılara yakındır. LALR ayrıştırıcılarının LR(1) ayrıştırıcılarına göre en önemli avantajı daha az belleğe ihtiyaç duymalarıdır (Cockett, 2007). Bunu, LR(1)'deki bazı durumları birleştirerek tek durum olarak temsil etmek suretiyle başarmaktadırlar. Daha az bellek ihtiyacı kısıtlı belleğe sahip gömülü sistemler için çok önemlidir. Java temelli olan CUP, tarayıcı olarak JFlex tarafından üretilen tarayıcılarla çalışır. Gramer modeli olarak CUP CFG'yi kullanırken, JFlex düzenli ifade temellidir.

Yazdığımız ayrıştırıcı birden fazla Java dosyasını ayrıştırabilmekte ve bu dosyadaki sınıflara ait kodların bellekte tek bir ayrıştırıcı ağacıyla temsil edilebilmesini sağlamaktadır. Yaptığımız testler ayrıştırıcımızın başarılı çalıştığını göstermektedir. Test amaçlı yazdığımız ağaç dolaşım kodları ayrıştırıcı ağacını dolaşarak ağacı dosyalara Java kodu olarak döker. Bu dökülen Java kodlarını tekrar ayrıştırıcımızla ayrıştırıp farklı dosyalara döküp iki gruptaki karşılıklı dosyaların aynı olduğunu gözlemledik. Bu gözlem

ayrıştırıcımızın yüzde yüz doğru çalıştığını kanıtlamasa da doğru çalıştığına işaret etmektedir.

Kod geliştirme aşamasında karşılaştığımız probleme değinelim. CUP'ın çıktısı olarak elde ettiğimiz parser.java dosyası başarıyla derlenmesine rağmen çalıştırma zamanında bir hata mesajıyla sonlandı. Bu sorunun temel sebebi parser.java dosyasının metotlarından birinin byte kod uzunluğunun 64K'yı aşmasıydı. Bu problemin üstesinden gelmek için 535 duruma (case) sahip olan ilgili metodun kodunu yaklaşık eşit uzunlukta ikiye bölerek iki farklı metoda paylaştırdık.

Geliştirdiğimiz ayrıştırıcının bazı kullanım alanlarına değinelim. Ayrıştırıcımız küçük bir takım değişikliklerle Java programlama diline özel amaçlı bazı eklentilerin yapılmasında bize yardımcı olabilir. Bu eklentiler dilin daha güvenli olmasını sağlayabilir ve kaynak kodun daha optimize byte kodlara dönüştürülmesinde derleyiciye ipuçları verebilir.

Yine ayrıştırıcımızın diğer bir kullanım alanı bazı ilavelerle Java programlama dilinde yazılı bir programı aynı fonksiyonellikteki bir C++ programına çevirmede ara bir adım olarak kullanabiliriz. Ayrıca ayrıştırıcımız Java programlama diline Türkçe bir arayüz yazılmasında da kullanılabilir. Belirlediğimiz Türkçe anahtar kelimeler kullanılarak Java benzeri bir programın Java dilinde yazılı bir programa dönüştürülmesi sağlanabilir.

KAYNAKLAR

Nystrom N., Clarkson M.R., and Myers A.C. Warsaw, 2003. Polyglot: An Extensible Compiler Framework for Java. *Proceedings of the 12th International Conference on Compiler Construction*, Poland. LNCS 2622, 138–152.

El-Kadri M., Groza V., Abielmona R. and Assaf M., 2006. A Just-in-Time Compiler for a Reconfigurable Testing Platform. *Proceedings of the IEEE Instrumentation and Measurement Technology Conference*, Sorrento, Italy. 628-632.

Fulton M. and Stoodley M., 2007. Compilation Techniques for Real-Time Java Programs. *IEEE International Symposium on Code Generation and Optimization*, San Jose, California. 221-231.

Cramer T., Friedman R., Miller T., Seberger D., Wilson R., Wolczko M., 1997. Compiling Java Just in Time. *IEEE Micro,* (*May/June) 36:43.*

Appel A.W., 1997. *Modern Compiler Implementation in Java*. Cambridge University Press, New York. 11:273.

Aho A., Lam M., Sethi R., Ullman J., 2007. *Compilers Principles Techniques and Tools (Second Edition).* PEARSON, Boston. 1-5.

Hutton B., 2006, *Computer Science 330, Computer Language Implementation*. Retrieved 30 November 2010, from http://www.cs.auckland.ac.nz/~bruce-h/lectures/330ChaptersPDF/Chapt1.pdf, http://www.cs.auckland.ac.nz/~bruce-h/lectures/330ChaptersPDF/Chapt2.pdf

Kleene S., 1956, *Representation of Events in Nerve Nets and Finite Automata*. Princeton University Press, New Jersey, 3–42.

Lesk M.E., Schmidt E. (n.d.). *Lex - A Lexical Analyzer Generator*. Retrieved April 8, 2011, from http://www.cs.utexas.edu/~novak/lexpaper.htm

Paxson V., 1995. *Flex, version 2.5, A fast scanner generator*. Retrieved December 12, 2011, from http://dinosaur.compilertools.net/flex/index.html

Berk E., updated on February 7, 2003. *JLex: A Lexical Analyzer Generator for Java(TM)*. Retrieved March 17, 2010, from http://www.cs.princeton.edu/~appel/modern/java/JLex/

Klein G., 2009. *JFlex User's Manual*. Retrieved February 10, 2010, from http://jflex.de/manual.html

Kovacs L., Barabas P., 2010. Implementation of Sentence Parser for Hungarian Language in Natural Language Processing. *IEEE International Symposium on Applied Machine Intelligence and Informatics,* Herlany. 59-63.

Might M. and Darais D., 2010. *Yacc is dead*. Retrieved December 10, 2011, from http://arxiv.org/abs/1010.5023v1

Sippu S. And Soisalon-Soinien E., 1988. *Parsing Theory, vol. I: Languages and Parsing*. Berlin Heidelberg, Germany: Springer-Verlag.

Sippu S. and Soisalon-Soinien E., 1990. *Parsing Theory, vol. II: LL(k) and LR(k) Parsing*. Berlin Heidelberg, Germany: Springer-Verlag.

Slivnik B. and Vilfan B., 2004. Improved error recovery in generated LR parsers. *Informatica*. vol. 28, no. 3, 257-263.

Parr T. and Fischer K.S., 2011. LL(*): The Foundation of the ANTLR Parser Generator. *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

Henriques P.R., Pereira M.J.V., Mernik M., Lenic M., Gray J.G., Wui H., 2005. Automatic generation of language-based tools using the LISA system. *IEEE Proceedings-Software,* vol. 152, no. 2, 54-69.

Slivnik B., 2011. The embedded left LR parser. *Proceedings of the IEEE Federated Conference on Computer Science and Information Systems*. Issue Date: 18-21 Sept. 2011, 863–870.

Al-Mamory S.O., Hongli Z., and Abbas A.R., 2008. Modeling Network Attacks for Scenario Construction. *IEEE International Joint Conference on Neural Networks*. Issue Date: 1-8 June 2008, 1495-1502.

Sebesta R., 2010. *Concept of Programming Languages (Ninth Edition)*. PEARSON, New Jersey. 209-220.

Knuth D.E.,1965. *On the Translation of Languages from Left to Right*, California Institute of Technology, Pasadena, California

Johnson S., 1975. *Yacc: Yet Another Compiler-Compiler*. Retrieved December 22, 2011, from http://www.csa.syr.edu/~chapin/cis657/yacc.pdf

Hudson S., 1999. *CUP User's Manual*. (Modified by Frank Flannery, C. Scott Ananian, Dan Wang with advice from Andrew W. Appel). Retrieved May 12, 2011, from http://www.cs.princeton.edu/~appel/modern/java/CUP/

Donnelly C., Stallman R., 2011. *Bison: The Yacc-compatible Parser Generator*. Retrieved October 12, 2011, from http://www.gnu.org/software/bison/manual/pdf/bison.pdf.

Parr T., 2007. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 376.

Cockett R., 2007. *Forming the LALR(1)-item automaton from the LR(0)-item automaton*, Retrieved July 2, 2011, from pages.cpsc.ucalgary.ca/~robin/class/411/LALR1/LALR1_follow_sets.html

Anonim, *Java Compiler Compiler - The Parser Generator*. (n.d.). Retrieved December 3, 2011, from https://javacc.dev.java.net

ÇİZELGELER **Sayfa No**

ŞEKİLLER **Sayfa No**

ÖZGEÇMİŞ

KİŞİSEL BİLGİLER:

Adı Soyadı: Mümüne ÖZÇETİN

Doğum Yeri: Seydişehir

Doğum Tarihi: 08.11.1987

EĞİTİM DURUMU:

Lisans Öğrenimi: ÇOMÜ Bilgisayar Mühendisliği (2008)

Bildiği Yabancı Diller: İngilizce

BİLİMSEL FAALİYETLERİ:

İŞ DENEYİMİ:

Çalıştığı Kurumlar ve Yıl: ÇOMÜ MMF Bilgisayar Mühendisliği (Araştırma Görevlisi) 2009-Devam ediyor.

İLETİŞİM:

E-posta Adresi: ozcetin87@gmail.com

Telefon: 0(544) 637 6203