ÖZELLEŞTİRİLEBİLİR İMLAYA SAHİP YENİ VE ESNEK NESNE TABANLI BİR PROGRAMLAMA DİLİ GELİŞTİRME

FIRAT KÜÇÜK

DÜZCE ÜNİVERSİTESİ

FEN BİLİMLERİ ENSTİTÜSÜ

ELEKTRİK EĞİTİMİ ANABİLİM DALINDA

YÜKSEK LİSANS

DERECESİ İÇİN GEREKLİ ÇALIŞMALARI YERİNE GETİREREK

ONAYA SUNULAN TEZ

ŞUBAT 2008

Fen Bilimleri Enstitüsü'nün Onayı

Prof. Dr. Demet KAYA

Enstitü Müdürü

Bu tezin Yüksek Lisans derecesinde bir tez olarak gerekli çalışmaları yerine getirdiğini onaylarım.

Yrd. Doç. Dr. Recep DEMİRCİ

Elektrik Eğitimi Anabilim Dalı Başkanı

Bu tezin Yüksek Lisans Derecesinde bir tez olarak onaylanması, düşüncemize göre, amaç ve kalite olarak tamamen uygundur.

Yrd. Doç. Dr. İbrahim ŞAHİN

Tez Danışmanı

Jüri Üyeleri:

Yrd. Doç. Dr. İbrahim ŞAHİN .....

Yrd. Doç. Dr. Mustafa TURAN .....

Yrd. Doç. Dr. Recep DEMİRCİ .....

A NEW AND FLEXIBLE OBJECT ORIENTED PROGRAMMING LANGUAGE WITH CUSTOMIZABLE GRAMMAR

Küçük, Fırat

MSc., Department of Electrical Education

Supervisor: Yrd. Doç. Dr. İbrahim Şahin

February 2008, 179 pages

Effective use of human resources is the most important requirement in a software project. All of the project team members should known almost all details of the programming language that is used in the project and should be able to overcome all the syntactical problems quickly. On the other hand, IT companies and software centrals generally do not have adequate human resources who know the same Programming Language. The main goal of the study is to provide a programming environment in which programmers with different programming language backgrounds can produce code without changing their programming behaviors. For this purpose, a new Programming Language Infrastructure called Cezve was designed and implemented. It is designed to be a host for customizable programming languages. It runs on Java Virtual Machine (JVM), and can be customized according

to the programmers wishes. Users of it can design their own language and use this language on Cezve without changing their programming habits.

Cezve includes an integrated tree parser and lots of management classes. A new Programming Language for Cezve can be created or an existing one can be modified using the tree parser constructor supported by Language Infrastructure. Throughout this study, an example programming language called Javy was created to show how the language customization mechanism of Cezve works. Some nice features of JavaScript, Python, Ruby, Perl languages are integrated in Javy.

Keywords: Lexer, Parser, Programming Language Infrasctructure, Java Sanal Makinesi

ÖZELLEŞTİRİLEBİLİR İMLAYA SAHİP YENİ VE ESNEK NESNE TABANLI BİR PROGRAMLAMA DİLİ GELİŞTİRME

Küçük, Fırat

Yüksek Lisans, Elektrik Eğitimi Anabilim Dalı

Tez Danışmanı: Yrd. Doç. Dr. İbrahim Şahin

Şubat 2008, 179 Sayfa

İnsan kaynaklarının etkin kullanımı, bir yazılım projesinde en önemli gereksinimidir. Proje ekibinin tüm fertleri, yazılımın yapılacağı dilin neredeyse tüm detaylarına hakim olmalı ve imla problemlerinin çabucak üstesinden gelebilmelidir. Fakat bilişim şirketleri ve yazılım merkezleri çoğu zaman aynı dili bilen yeterli insan kaynağına sahip olamaz. Bu çalışmanın temel amacı farklı Programlama Dili geçmişi olan programcıların programlama alışkanlıklarını değiştirmeden kod üretmeleridir. Bu maksatla yeni bir Programlama Dili Altyapısı olan Cezve tasarlandı ve oluşturuldu. Cezve, özelleştirilebilir programlama dillerinie ev sahipliği yapmak için tasarlandı. Java Sanal Makinesi (JVM) üzerinde koşan dil alt yapısı, programcıların istekleri doğrultusunda özelleştirilebilir. Cezve kullanıcıları kendi dillerini

tasarlayabilir ve programlama alışkanlıklarını değiştirmeden tasarladıkları dili kullanabilirler.

Cezve bütünleşik bir ağaç ayrıştırıcı ve bir çok yönetici sınıfı içermektedir. Dil altyapısının desteklediği ağaç yapısı gözetilerek varolan Programlama Dilleri özelleştirilebilir veya baştan yeni bir Programlama Dili oluşturulabilir. Çalışmamız süresince dil özelleştirme mekanizmasını göstermek için Javy adında örnek bir programlama dili oluşturulmuştur. Dil altyapısının çalışma mekanizmasını göstermek adına örnek bir Programlama Dili oluşturulmuştur. JavaScript, Python, Ruby, Perl gibi dillerin bazı güzel özellikleri Javy'ye eklenmiştir.

Anahtar Kelimeler: Sözcük Ayrıştırıcı, Söz Dizim Ayrıştırıcı, Programlama Dili Altyapısı, Java Sanal Makinesi

TEŞEKKÜR

Çalışmam esnasında bana yardımcı olan danışman hocam, Sayın Yrd. Doç. Dr. İbrahim ŞAHİN'e, Ana Bilim Dalı Başkanımız Yrd. Doç. Dr. Recep Demirci'ye ve değerli hocam Yrd. Doç. Dr. Pakize ERDOĞMUŞ ve onun çok sevdiğim eşi Yrd. Doç. Dr. Beşir ERDOĞMUŞ'a,

Sürekli yanımda olduklarını bildiğim sevgili annem Nezahat KÜÇÜK'e ve sevgili ablam Yrd. Doç. Dr. Aysel KÜÇÜK'e, rahmetli babam Hüseyin KÜÇÜK, rahmetli ablam Yrd. Doç. Dr. Ayfer KÜÇÜK ve rahmetli abim Arş. Gör. Dr. Murat Küçük'e,

Bu çalışmamda bana manevi desteğini esirgemeyen ve hep destek olan arkadaşım Gönül KARAKUŞ'a, sınıf arkadaşlarım Ali ERDUMAN, Öğr. Gör. Ferzan KATIRCIOĞLU'na ve Yüksek Lisans'ım konusunda her zaman destek veren ve bana güvenen sayın Yrd. Doç. Dr. Mustafa TURAN'a,

Çalışmamı sürekli destekleyen kardeşlerim Boğaç TURGUT, Yılmaz UĞURLU, Ersin ARSLAN, Cafer ŞİMŞEK, Talat UYARER, Anıl KARADAĞ, Özgün TARHAN, Murat ARSLAN, Burak ERKAYMAN, Göktuğ ÖZTÜRK, Alparslan SARI, Mahmut YILDIZ, Tuğrul DURAN ve Şamil ALTUNDAL'a, proje

planlaması konusunda desteğini esirgemeyen Öğr. Gör. Elif YAVUZ'a ve anlayışı ile destek veren Mehmet DANIŞMAN'a,

Bunların dışında ANTLR sistemini sağlayan Prof. Dr. Terence PARR'a, yazılım için barınma hizmeti veren sf.net'e ve IntelliJ IDE yazılımını ücretsiz kullanımımıza tahsis eden JetBrains şirketine, Netbeans ekibine ve çalışmamızın sürekli arkasında olan btturk.net sitesine,

Saygı ve şükranlarımı sunarak teşekkür etmek istiyorum.

İÇİNDEKİLER DİZİNİ

TABLOLAR LİSTESİ

ŞEKİLLER LİSTESİ

1. GİRİŞ

Bilgisayar Dilleri kavramı, günümüzde kabına sığamamış cep telefonlarından kol saatlerine kadar tüm işlem birimi (Mikroişlemci, Mikrodenetleyici, Sayısal İşaret İşleyici (DSP) vb..) bulunan elektronik cihazlarda kullanım alanı bulmuştur. Şüphesiz bu yönelim, programlanabilir bir elektronik aygıtın özelleştirilebilme büyüsünden ileri gelmektedir.

Programlanabilir bir aygıt, kullanıldığı sistem içindeki bütün girdi, çıktı ve etkileşim birimlerinin muhtemel tüm kullanım kombinasyonlarını kullanabilme yetisine sahiptir. Yüzlerce mikroişlemci yönergesi ve onlarca tümleşik aygıta sahip olan günümüz bilgisayarlarındaki bu kombinasyonlar, uygulama geliştiricilerin hayal gücü ile sınırlıdır.

Kasım 1954'de [1] ilk Yüksek Düzeyli Programlama Dili (High-Level Programming Language) Fortran'ın çıkması ile artık bilgisayarlar daha kolay programlanabilir hale gelmiştir. Bu açılan yeni yüksek düzey programlama çağı ile birlikte artık programlar daha kısa bir süreç neticesinde yazılabilmiştir. Uygulama geliştiricilerin daha az kod ile daha çok iş yapma becerileri, bu tarihten itibaren yeni çıkan paradigmalara paralel olarak katlanarak arttı.

FORTRAN'dan önce bir çıktı aygıtına bir kelime yazdırmak için kullanılabilecek yegâne yol; işlem birimini makine dili yönergeleri ile ya da bu makine dili yönergelerine takma isimler (Mnemonic) verilerek oluşturulmuş Assembly Dili sözcükleri ile beslemekti. Bu işlem, FORTRAN gibi bir üst düzey dilin icadı ile PRINT veya WRITE ifadesi kadar sadeleştirilebildi. FORTRAN makine diline derleme yaptığından sonuçtaki uygulama Assembly ile yazılan uygulamaya oldukça yakın başarım (performans) sergilemekteydi [2]. Artık programlar daha kısa sürede daha az kodlama ile yazılabiliyordu.

Bilgisayar dilleri, daha sonraları yalnız programlama kavramı içerisine sıkışıp kalmadı. Kullanıcı Grafik Arabirimi (KGA/GUI) tanımlama, veritabanı sorgulama gibi daha birçok alanda kullanım bularak uygulama geliştiricilerin daha çok işlem yapabilen programları, daha kısa sürede yazmalarına imkân tanıdı. Bilgisayar Dillerinin alt kolu olan Programlama Dilleri ise artık yalnızca uygulama geliştirmek için değil, uygulamaları özelleştirmek için de sıkça kullanılır hale geldiler.

Bunların yanında programlama dillerine ait sanal makineler (JVM, .NET Framework) ve farklı mikroişlemci platformları arası derlenebilen diller (C/C++ vb.) sayesinde, işlemci mimarilerine kolayca uyarlanabilen uygulamalar yazılmasına

imkân tanınmıştır. Hiç şüphesiz bu yaklaşımlar da uygulama geliştirme sürecini olabildiğince kısaltmıştır.

Sanal Makineler (Virtual Machines) sayesinde sanal makineye özgü yazılmış uygulamalar, sıfır hata ile desteklenen platformlar arası anında taşınabilmektedir. Sanal makine üzerinde çalışmayan uygulamalar ise her platformda çalışan aynı programlama dilini derleyebilen derleyiciler sayesinde, farklı işlemci mimarileri üzerinde de çalışabilir hale gelmektedir. Örnek olarak; 86 milyon satır koda sahip Apple Mac OS X Tiger işletim sistemi, 210 gün gibi bir süre içerisinde Power PC platformundan Intel platformuna taşınabildi [3]. Bu aktarımı gerçekleyen en büyük etmen hiç şüphesiz platformlar arası çalışabilen C derleyicileriydi.

Programlama dilleri, bilişim sektörü başta olmak üzere tüm sektörlerde kullanım bulmakta ve bu sektörlere ait uygulama geliştirme süresini kısaltan daha performanslı programlama dillerine her zaman ihtiyaç duyulmaktadır. Uygulama geliştirme süresine etki eden en büyük etmen bir yazılım şirketinin veya yazılım üreten bir merkezin sahip olduğu insan kaynağıdır. Sahip olunan bilişim elamanı ve IT profesyoneli sayısı yeterli ve yetkin ise iyi bir sonuç ortaya çıkacaktır. Fakat yazılım merkezleri çoğunlukla aynı proje üzerinde aynı programlama ortanımı kullanabilecek gerekli olan insan kaynağını tedarik edemez. Bunun yanında bir projede çalısan bütün programcılar aynı imla kurallarını kullanarak program yazmak

istemeyebilirler. Programcılar daha üretken ve alışık oldukları programalama stilini ve imlasını kullanmak isteyeceklerdir.

1.1. Proje Amaçları

Çalışmada tez başlığında da vurgulandığı şekilde özelleştirilebilir bir imlaya sahip nesne modelli programlama dili oluşturma hedeflenmektedir. Bu maksatla nesne modelli ve betik örnek bir programlama dili geliştirilerek özelleştirilebilir bir yapı ile sunulacaktır. Oluşturulacak olan programlama dili özelleştirilebilmesinin yanında modern programlama dili paradigmaları sağlamalıdır. Bu sayede programcılara yeni yaklaşımlar ile program yazma imkanı sağlanabilir.

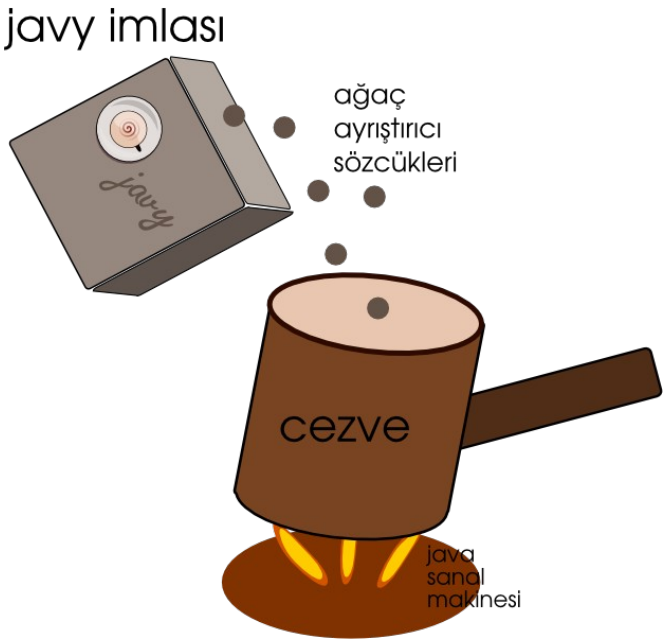
Tez çalışmasında aynı çatı altında farklı programlama dilleri bilen bilişim elemanlarını buluşturmak için özelleştirilebilir dillere imkan tanıyan bir programlama dili altyapısı geliştirilmiştir. **Cezve Dil Altyapısı** [4] adını verdiğimiz bu platform ile programcılar alıştığı programlama dili imlasından vazgeçmeden aynı proje kapsamında çalışabileceklerdir. Çalışma bir imla dosyası ile istenilen programlama dili tanımlanmasını sağlamaktadır. Dil altyapısının genel çalışma sistematiğini göstermek için, JavaScript [5], Python [6], Perl [7], PHP [8], Ruby [9] gibi dillerin programcılar tarafından sevilen özellikleri alınarak tasarlanmış **Javy** adı

verilen bir örnek programlama dili tez çalışmasına eklenmiştir. Projenin güncel hali, www.javy.org adresinden temin edilebilmektedir.

Bunlara ilaveten çalışma neticesinde ortaya çıkacak olan programlama dili, bir çok programlama dilindeki kullanım kolaylığı sağlayan özelliklerin bazılarını da bünyesinde barındıracaktır. Bu sayede programcıların daha az kod ile daha işlevsel programlar yapması ve daha kısa sürede çalışabilen ürünler elde etmesi hedeflenmektedir.

Proje, aynı zamanda bilgisayar programcılığı öğrencileri için öğretim materyali olarak kullanılabilir.Ana dillerinde kullanabilecekleri ve öğrenimi kolay programlama dilleri geliştirilerek öğrencilerin nesne modelli yaklaşımı öğrenmeleri sağlanabilir.

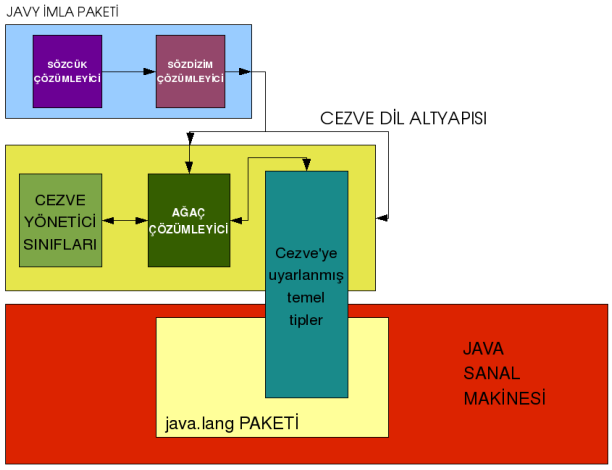
1.2. Sistem Bileşenleri



Şekil 1: Cezve Dil Altyapısı İllüstrasyonu

Çalışmada Java Sanal Makinesi üzerinde çalışan özelleştirilebilir imlaya sahip betik dillere ev sahipliği yapan bir altyapı ve bu altyapı üzerinde çalışan modern paradigmalara sahip bir dil oluşturulmuştur. Proje temel olarak 3 katman üzerine bina edilmiştir.

1. Özelleştirilebilir Programlama Dili
2. Cezve Dil Altyapısı
3. Java Sanal Makinesi



Şekil 2: Cezve Sistem Bileşenleri

Oluşturulan programlama altyapısına **'Cezve'** ve programlama diline de **'Javy'** adı verilmiştir. Javy, temel veri tipleri olarak Java'nın sahip olduğu birçok başvuru (referans) tipini kullanmaktadır. Bunlara ek olarak belli başlı Java başvuru tiplerinin Cezve uyarlamaları hazırlanmıştır. Oluşturulan ön tanımlı özelleştirmeye açık imla, başta Java olmak üzere Python, JavaScript ve Ruby gibi dillerden esinlenerek ortaya çıkarılmıştır. Ortaya çıkardığımız konsept dilde, bu dillerin programcılar tarafından beğenilerek kullanılan özellikleri alınmıştır.

2. ÖN BİLGİ

Javy Programlama Dili ve Cezve Dil Altyapısı, temelde birçok teknolojiye dayanır. Bunların en önemlileri, Zuse [10], Plankalkül'den [11] günümüzde kadar geliştirilmiş birçok programlama dili ve bu programlama dillerine ilişkin paradigmalardır. Javy, kendi özelliklerini miras aldığı C, C++ [12] gibi birçok programlama dili gibi temelde dillerin atası sayılabilecek olan diğer programlama dillerinden mesnet almaktadır. Bu programlama dillerinin açtığı önemli yönelimler sayesinde diller şekillenmiştir.

2.1. Bilgisayar Dilleri

Genelde yaygın görüş, bilgisayar dillerinin programlama dili olduğudur. Fakat bilgisayar dilleri kullanım amaçlarına göre farklı şekiller alabilirler. Örneğin; bir web sitesi arabirimi tasarlamak için HTML dili kullanılabilir. HTML, bir programlama dili değil, bir bilgisayar dilidir. Veritabanlarını sorgulamak için kullandığımız SQL, ya da küresel bazda geniş kullanım alanı bulan betimleme dili olan XML, web sitelerinin görünüm şablonlarının tasarlanabildiği CSS gibi birçok dil, programlama dili olmamasına karşın birer bilgisayar dilidir. Bilgisayar dillerinin belirli başlı kategorileri şu şekildedir.

- Programlama Dilleri : Simula [13], Algol [14], C, C++, Ada [15] ... vb
- Betik Diller: JavaScript, Python, Perl .. vb
- İş Denetim Dilleri ve Kabuklar: Bash, ash .. vb
- KGA ve Makro Betik Dilleri: expect
- Uygulamaya Özel Betik Diller: VimScript
- Web Programlama Dilleri: PHP, Ruby .. vb
- Metin İşleme Dilleri: sed, awk .. vb
- Programlama Dili Uzantıları/Gömülebilen Diller: ch ..
- Tanımlama Dilleri:
- Makine Kodları ve Byte Kodlar: Java ByteCode, MSIL .. vb
- Sorgu Dilleri: SQL, RDQL ..vb
- Betimleme Dilleri: HTML, SGML
- Dönüştürme Dilleri
- Donanım Tanımlama Dilleri: VHDL [16] .. vb
- Yapılandırma Dosyaları
- Veri Serileştirme Biçemleri

2.2. Başlıca Programlama Paradigmaları

Programlama Dilleri, günümüze kadar birçok değişiklikten geçmiştir ve günümüze gelene kadar birçok farklı yaklaşım sergilenmiştir. Programlama paradigmaları, nesneler veya dilin kullandığı ögeler arasındaki ilişkileri, belirli temellere oturtmak ve programcılara kolaylık sağlamak adına tasarlanmışlardır. Belirli başlı programlama paradigmaları şöyledir.

- Zaruri Programlama
- Yordam Tabanlı Programlama
- Tanım Tabanlı Programlama
- İşlevsel Programlama
- Nesne Yönelimli Programlama
- Görünüm Yönelimli Programlama
- Mantıksal Programlama

Programlama Dilleri, yazılma stillerine, değişken tanımlamalarına ve belli başlı genel imla kurallarına göre de kategorilere ayrılır. Bu ayrım güvenlik, optimizasyon, belgelendirme ve modülerlik açısından da önem taşır. Programlama dillerini tip denetimlerine göre şu şekilde listeleyebiliriz.

- Durağan Tip Denetimi
- Devingen Tip Denetimi
- Güçlü Tip Denetimi
- Zayıf Tip Denetimi
- Güvenli Tip Denetimi
- Güvensiz Tip Denetimi
- Ördek Tip Denetimi

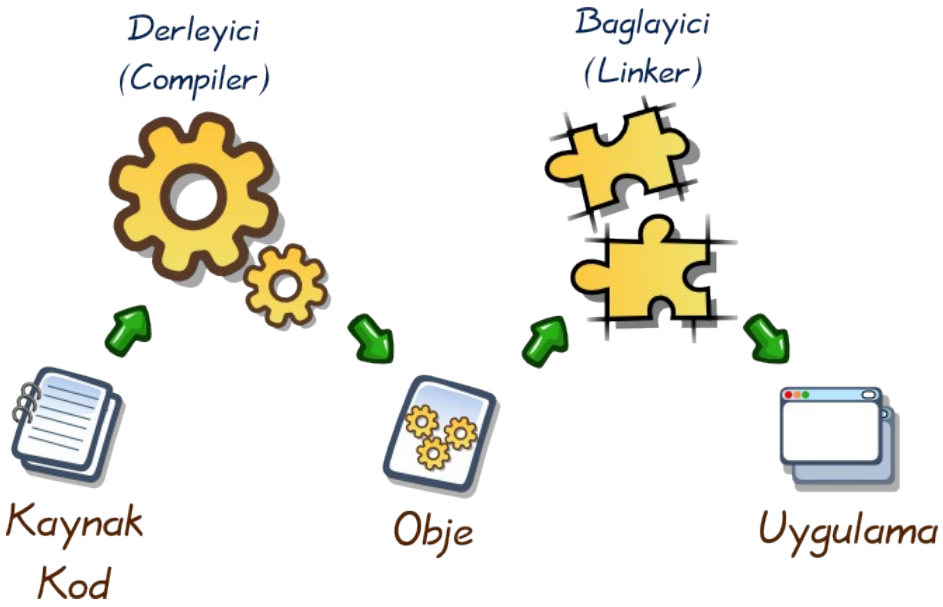
2.3. Dil İşleyicileri

Bir programlama dilinde yazılmış programa ait kaynak kod, ya bir yorumlayıcı ile anında ara dile derlenip çalıştırılacaktır ya da bir derleyici sistem ile çalıştırılabilir dosyaya dönüştürülecektir. Bu süreç, tamamen programlama dilinin yapısına bağlıdır. Yorumlayıcı ve Derleyici dil işleme sistemleri, sözcük ayrıştırma, söz dizim denetimi ve denetlenen söz dizimin anlamsallaştırılması gibi birçok asli vazifeye sahiptirler.

2.3.1. Derleyici

Derleyici (Compiler), bir betik program kaynağını alıp derlenmiş ikilik kod haline dönüştüren sistemlerdir [17] . Program platforma bağımlı makine diline veya sanal makineye bağımlı bir ara dile dönüştürülür.

Derlenilebilen bir sistemde genel bir yaklaşım şu şekildedir;



Şekil 3: Kod Derleme ve Bağlama

Örnek C programı:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
    printf("Merhaba Dünya");
    return EXIT_SUCCESS;
}
```

Yukarıdaki kod ekrana “Merhaba Dünya!” çıktısı veren basit bir C uygulamasıdır.

Programı Derleme:

```
gcc -std=c99 -c -Wall ornek.c
```

Dil derleyicisi ile kaynak kodunu içeren dosya işlenir. Sonuç bir nesne (object) dosyasıdır.

```
ornek.o
```

Nesne dosyası, özel biçemde hazırlanmış makine kodları içeren bir dosyadır. İçerdiği makine kodu şu şekilde analiz edilebilir:

```
objdump -d ornek.o
-----------------------------------------------------------------

ornek.o:     file format elf32-i386

Disassembly of section .text:

00000000 <main>:
   0:   8d 4c 24 04             lea    0x4(%esp),%ecx
   4:   83 e4 f0                and    $0xfffffff0,%esp
   7:   ff 71 fc                pushl  0xfffffffc(%ecx)
   a:   55                      push   %ebp
   b:   89 e5                   mov    %esp,%ebp
   d:   51                      push   %ecx
   e:   83 ec 04                sub    $0x4,%esp
  11:   c7 04 24 00 00 00 00    movl   $0x0,(%esp)
  18:   e8 fc ff ff ff          call   19 <main+0x19>
  1d:   b8 00 00 00 00          mov    $0x0,%eax
  22:   83 c4 04                add    $0x4,%esp
  25:   59                      pop    %ecx
  26:   5d                      pop    %ebp
  27:   8d 61 fc                lea    0xfffffffc(%ecx),%esp
  2a:   c3                      ret
```

Programın kullandığı kütüphaneler bağlanılıp tamamen çalıştırılabilir bir dosya haline gelmesi gerekmektedir.

Programı Bağlama:

```
gcc –o ornek ornek.o
```

Bağlama işlemi neticesinde, çalıştırılabilir **ornek** dosyası oluşur.

```
ornek
```

Örnek dosyasının bağlandığı kütüphaneler;

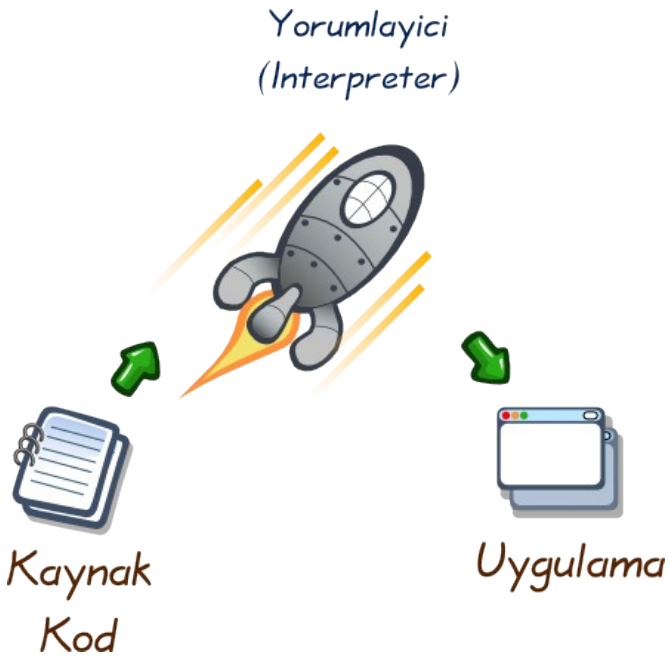
```
ldd –v ornek
-----------------------------------------------------------------
  linux-gate.so.1 =>  (0xffffe000)
  libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7ddc000)
  /lib/ld-linux.so.2 (0xb7f2b000)

  Version information:
  ./ornek:
    libc.so.6 (GLIBC_2.0) => /lib/tls/i686/cmov/libc.so.6
  /lib/tls/i686/cmov/libc.so.6:
    ld-linux.so.2 (GLIBC_PRIVATE) => /lib/ld-linux.so.2
    ld-linux.so.2 (GLIBC_2.3) => /lib/ld-linux.so.2
    ld-linux.so.2 (GLIBC_2.1) => /lib/ld-linux.so.2
```

Programı Çalıştırma:

```
./ornek
Merhaba Dünya
```

2.3.2. Yorumlayıcı

Yorumlayıcı (Interpreter), bir betik kaynak kodu dosyasını alıp çalıştıran programdır. Kaynak kodu ek bir derleme işlemine tabi tutulmaz. Yalnızca yorumlayıcı tarafından analiz edilir ve işletilir [18].



Şekil 4: Kod Yorumlama Sistemi

Örnek bir Python ifadesi:

```
for i in range(5): print i
```

Program ornek.py olarak kaydedilecektir. Ve yorumlayıcıya argüman olarak gönderilecektir.

```
python ./ornek.py
```

Yorumlayıcı, programı derlemeden veya bağlamadan doğrudan çalıştıracaktır.

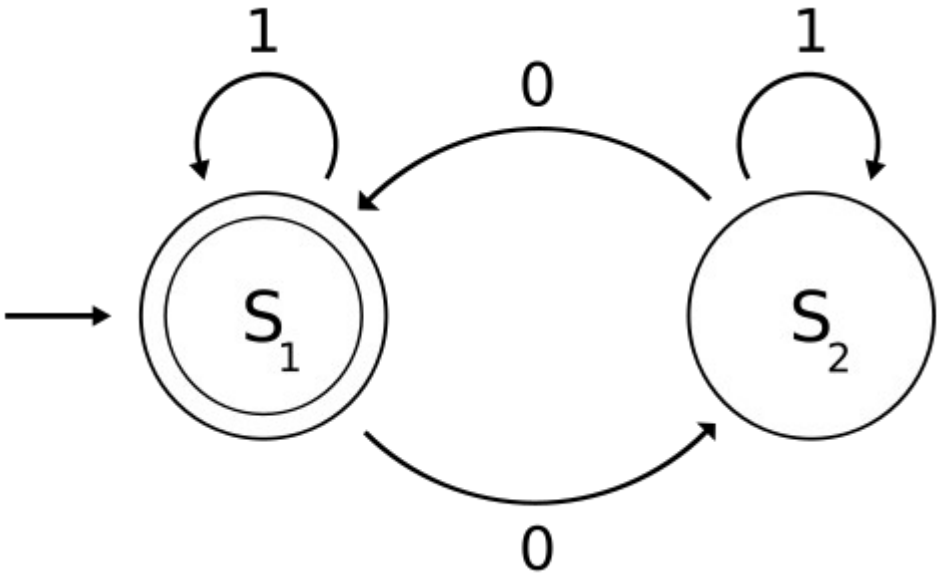
```
0
1
2
3
4
```

2.4. Durum Makinesi Kavramı

Sonlu durum makineleri kavramı denilince akla gelen ilk ifade, Sonlu Durum Makineleridir (Finite State Machines). Sonlu Durum Makineleri, sonlu sayıdaki durumu ve bunlar arasındaki geçişleri ve hareketleri ifade eden yapılardır [19].

2.4.1. Belirgin Sonlu Otomat

Belirgin Sonlu Otomat (Deterministic Finite Automata), bir Sonlu Durum Makinesidir. Her giriş ve durum sembolü için, bir ve yalnız bir adet sonraki duruma geçiş bulunur. Bir sonraki ifade için geçiş durumu daima bellidir [20].



Şekil 5: Örnek DFA Gösterimi

2.4.1.1. Düzenli İfadeler

Düzenli İfadeler (Regular Expressions), karakter katarları üzerinde eşleşme sağlayan bir dizi karakter katarıdır. Genelde arama özelleştirme için kullanılan bir ifade şeklidir. Düzenli ifadeler, yapıları itibari ile birer Belirgin Sonlu Otomat'tırlar (DFA). Oluşturulan sistemde belirgin olmayan bir yaklaşım bulunmamaktadır.

Örnek Üzerinde Arama Yapılacak Metin:

```
Korkma, sönmez bu şafaklarda yüzen al sancak;
Sönmeden yurdumun üstünde tüten en son ocak.
O benim milletimin yıldızıdır, parlayacak;
O benimdir, o benim milletimindir ancak.
```

"^O" tırnaklar arasındaki düzenli ifadenin sağladığı eşleşme şu şekildedir.

Korkma, sönmez bu şafaklarda yüzen al sancak; Sönmeden yurdumun üstünde tüten en son ocak. **O** benim milletimin yıldızıdır, parlayacak; **O** benimdir, o benim milletimindir ancak.

Düzenli ifadedeki **“^”** aksan işareti, satır başını vurgulamaktadır. Sonlu otomatın sonraki durumu ise **“O”** harfidir. Bu durumda karakter katarımızdaki **O** ile başlayan ifadeler eşleşmeyi sağlayacaktır.

İstiklal Marşındaki **“cak”** ile biten kelimeleri vurgulamak için. **”[a-z]*cak”** ifadesini yazmamız yeterli olacaktır. Bu belirgin durum makinesinin grafiği Şekil 6'daki gibidir.

Şekil 6: Düzenli İfade Durum Makinesi Gösterimi

2.4.1.2. BNF

Backus–Naur Form olarak bilinen bu ifade kümesi, formal dilleri tanımlamak için bağlamdan bağımsız bir yapıya sahiptir [21]. BNF imlası ve daha gelişmiş sürümü olan EBNF, düzenli ifadeler ile benzerlikler göstermektedir. Genel imla tanımla şekli:

```
<<sembol>> ::= <<semboller ve ifadeler>>
```

şeklindedir. Semboller üzerinde düzenli ifadelerdeki benzer ***** ve **+** gibi belirteçler ile tekrar miktarları belirtilebilmektedir. Düzenli ifadelere ek olarak, VEYA "|" ifadesi de bulunmaktadır. BNF imlası genelde ayrıştırıcı oluşturmak için bilgisayar dillerini oluşturmada kullanılan ayrıştırıcıların tanımlanmasında kullanılmaktadır. Örnek bir sözcük ayrıştırıcının işleyişi BNF ile şu şekilde tanımlanabilir:

```
kural ::= alt_kural_1* alt_kural_2
alt_kural_1 ::= 'A' | 'B'
alt_kural_2 ::= 'C'
```

"kural" BNF ifadesinde alt_kural_1 hiç kullanılmayabilir, bir kere veya birden fazla da kullanılabilir. Bu tanımlamayı * işleci tanımlar. Daha sonra ise mutlaka alt_kural_2 gelmelidir. Kısaca şu yargıya varabiliriz. Bu kurallar zinciri ile tanımlanmış tüm ifadeler alt_kural_2 yani 'C' ile bitmek zorundadır. "kural" ifadesinin ilk eşleşmesi olan "alt_kural_1" ifadesi 'A' veya 'B' karakterlerinden biri olmalıdır. Bu durumda bu kurallar dizisini eşleşeceği veriler şu şekilde olabilir:

- C
- BC
- AC
- ABC
- AAC
- BBC
- BAC
- AAAC
- AABC

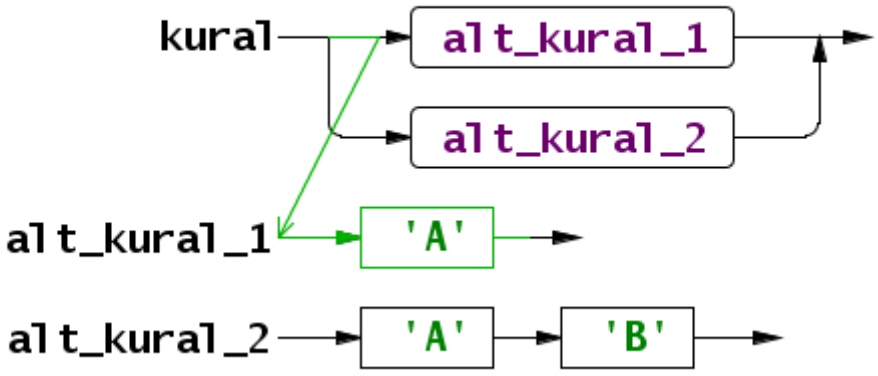
- …

2.4.2. Belirgin Olmayan Sonlu Otomat

Bir Durum Makinesinde, sonraki geçiş durumu belirgin değilse, burada belirsizlik (nondeterminism) mevcuttur. Belirgin olmayan bir ifadenin çözümü oldukça zordur. Bu nedenle formal dillerde ayrıştırma yapılırken belirsizliğin giderilmesi gerekmektedir.

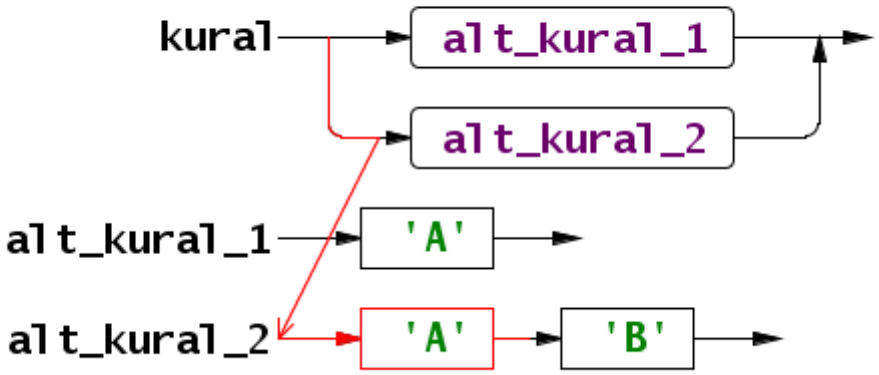
Belirgin olmayan (nondeteministic) bir BNF imla örneği şu şekilde olabilir.

```
kural        ::= alt_kural_1 | alt_kural_2;
alt_kural_1  ::= 'A';
alt_kural_2  ::= 'A' 'B';
```



Şekil 7: DFA Belirsizliği 1. Seçenek

başı 'A' ile başlayan ayrılaştırılacak bir ifade için tanımlanan durum, kural → alt_kural_1 şeklinde olabilir. Bunun yanında:



Şekil 8: DFA Belirsizliği 2. Seçenek

durum kural → alt_kural_2 şeklinde de olabilir. Bu bir belirsizliktir.

Bunun için bazı ayrıştırıcı sistemler, İleri Bakış (Look Ahead) miktarı ile bu sorunun üstesinden gelirler. Yukarıdaki örnekte ileri bakışı 2 olan bir sistem kurallarda 'A' harflerini eşleştirmek yerine 'A' 'B' harflerini eşleştirecektir. Bu da belirsizliği ortadan kaldıracaktır. Çünkü yalnızca alt_kural_2'de 'A' 'B' ifadesi bulunmaktadır.

2.5. Çözümleme Sistemleri

Çözümleme sistemleri, başta bilgisayar dilleri oluşturma gibi birçok alanda ayrıştırma işlemi yapabilen sistemlerdir. Programlama dillerinde bir derleyiciye veya bir yorumlayıcıya ait sistemin tanımlanması BNF kuralları ile yapılmaktadır. Eğer BNF kurallarından ayrıştırıcı oluşturan otomatik bir sistem varsa, bu tanımlama

işlemi bir ürün olarak programcıya da dönecektir. Genellikle bu işlemi yapan uygulamalar: Sözcük Ayrıştırıcı Oluşturucu (Lexer Generator), Sözdizim Ayrıştırıcı Oluşturucu (Parser Generator), Derleyici Derleyici (Compiler Compiler) gibi isimler alır.

Yapılan BNF tarzı kural tanımlamaları üzerinde yapılan ayrıştırma metoduna göre, Çözümleme Sistemleri kategorilere ayrılırlar.

2.5.1. Yukarıdan Aşağı Ayrıştırma

Yukarıdan aşağı (Top-Down Parsing) ayrıştırma sistemlerinde, ayrıştırıcı diğer sistemlerde olduğu gibi kurallar üzerinde yürür. Ve alt kurallara dallanarak devam eder [22].

```
A -> 'a' B C
B -> 'c' | 'c' 'd'
C -> 'd' 'f' | 'e' 'g'
```

şeklinde tanımlanan BNF kurallar kümesinde ayrıştırıcı, öncelikle A kuralından başlar ve daha sonra B kuralı içerisine girer. B kuralını işletip, tekrar A kuralı üzerinde kaldığı noktaya geri döner. Hemen sonra ise C kuralı ile karşılaşıp, bu alt kural içerisine girer. C kuralından çıktıktan sonra, A kuralı bitecektir.

Kurallar, kural ağacında yukarıdan aşağı doğru ayrıştırılmaktadır. Özyinelemeli İniş Ayrıştırıcı (Recursive Descent Parser), LL Ayrıştırıcı gibi çözümleme sistemleri bu kategoriye girer.

2.5.2. Aşağıdan Yukarı Ayrıştırma

Aşağıdan yukarıya ayrıştırma (bottom-up parsing) işleminde kurallar, en uç kuraldan kök kurala varılacak şekilde işletilir.

```
javy_kurali   -> ja_alt_kurali 'vy'
ja_alt_kurali -> j_alt_kurali 'a'
j_alt_kurali  -> 'j'
```

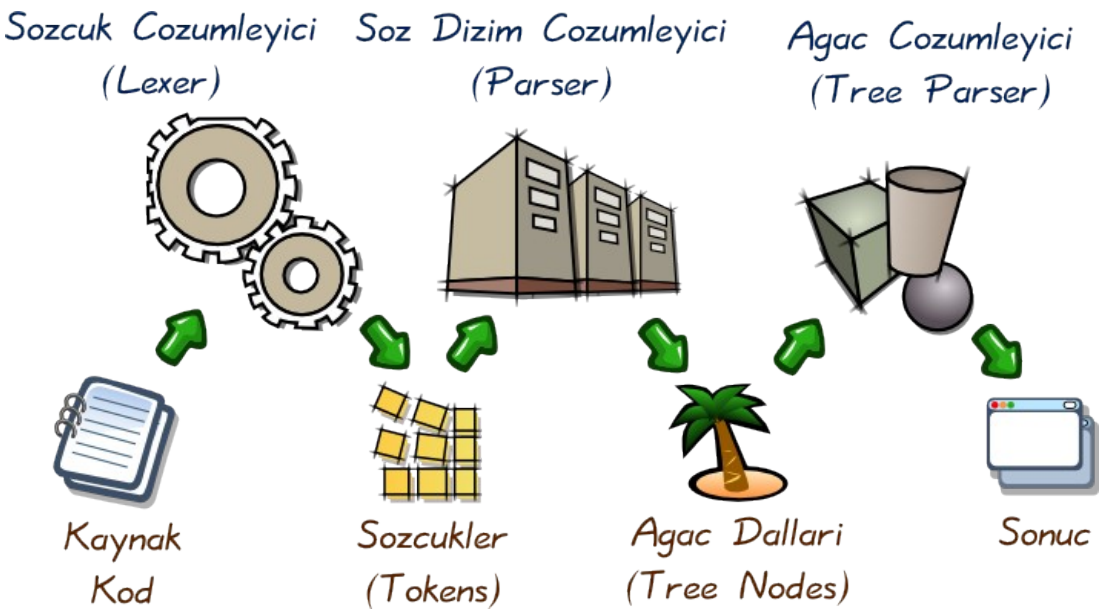
gibi bir kurallar kümesinde "javy" karakter katarı j_alt_kurali → ja_alt_kurali → javy_kurali sırasında ayrıştırılır. Yukarıdan aşağı ayrıştırma yapan bir sistemde ise bu yapı tam tersi bir hal alacaktır. javy_kurali → ja_alt_kurali → j_alt_kurali şeklinde ayrıştırma gerçeklenecektir.

Aşağıdan yukarı ayrıştırma da ayrıştırılan ifade yerine kural geçer. Ve sonuca ulaşılır:

'javy' → j_alt_kurali avy → ja_alt_kurali vy → javy_kurali

2.6. İmla Çözümleme Aşamaları

Bilgisayar Dilleri, Çözümleme Sistemlerinin başlıca kullanım alanıdır. Ve bir bilgisayar dilinin işlenebilen anlamsal hale gelebilmesi için birçok kere çözümlenmesi gereklidir. Örnek bir hesaplama dili yapmak istersek, kaynak dosyası şu aşamalardan geçmelidir:



Şekil 9: İmla Çözümleme Aşamaları

Bu ayrıştırma işlemini gerçekleştirmek için 3 adet farklı ayrıştırıcıya ihtiyacımız bulunmaktadır. Bu ayrıştırıcıları, el ile yazabileceğimiz gibi bir ayrıştırıcı oluşturucu sistem ile de oluşturabiliriz. Her ayrıştırıcı için farklı kurallar zinciri tanımlamamız gereklidir.

Sözcük Ayrıştırıcı (Lexer), Söz dizim ayrıştırıcı (Parser) ve Ağaç ayrıştırıcı (Tree Parser) için ayrı kural tanımlamaları yapılmalıdır.

Kaynak dosyası Sözcük Çözümleyici sisteme girer ve çözümleme sisteminden sözcükler (tokens) çıkar.

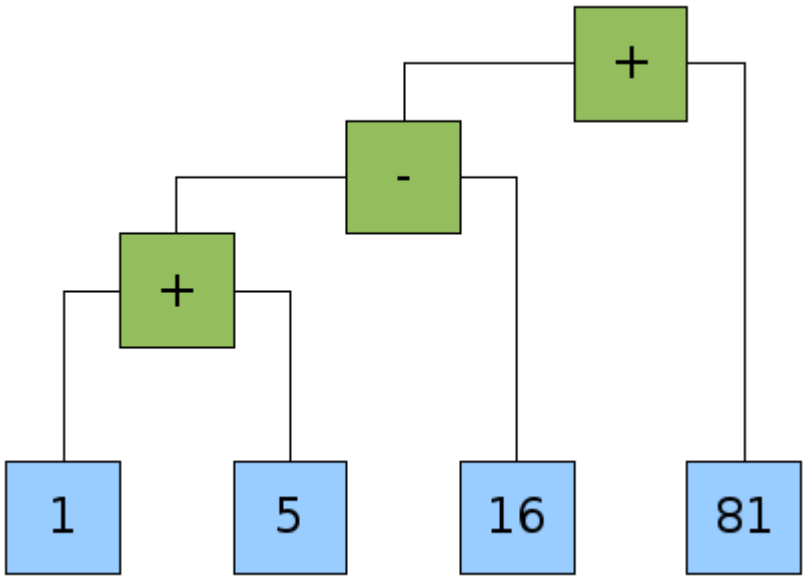
Örnek bir kaynak dosyası:

```
1 + 5 - 16 + 81
```

Sözcük ayrıştırıcının çıktısı ayrıştırılmış sözcük katarıdır.

```
SAYI(1) ARTI(+) SAYI(5) EKSI(-) SAYI(16) ARTI(+) SAYI(81)
```

Bu sözcük katarı ise söz dizim ayrıştırıcı tarafından ayrıştırılır.



Şekil 10: İşlem Ağaç Yapısı Gösterimi

Kuralların durumuna göre ortaya bir ağaç yapısı çıkar ve bu çıkan ağaç yapısı ağaç ayrıştırıcı tarafından çözümlenir.

2.7. ANTLR Çözümleyici Oluşturucu

Projede çözümleyici oluşturmak için ANTLR (ANother Tool for Language Recognition) [23] kullanılmıştır. ANTLR, San Francisco Üniversitesi'nde (USFCA) öğretim üyesi olarak çalışan Prof. Dr. Terrence PARR tarafından başlatılan ve birçok programcı tarafından destek verilen ayrıştırıcı, yorumlayıcı, çevirici ve tanımlayıcı oluşturmak için kullanılan açık kaynak bir dil üretecidir. ANTLR, LL(k) tipi bir ayrıştırıcı oluşturucudur. Yani ayrıştırma işlemi yukarıdan aşağı (top-down) doğru yapılır. LL(k) ifadesindeki k ise, ileri bakış miktarını vermektedir.

```
kural        : alt_kural_1 | alt_kural_2;
alt_kural_1  : 'A';
alt_kural_2  : 'A' 'B';
```

Örneğinde "AB" kelimesini ayrıştırırken, ileri bakış (Look Ahead) miktarı 1 olarak ayarlanırsa; sistem, belirsiz (nondeterministic) bir hal alır. ANTLR, çözümleme işlemini karakterleri birer birer tüketerek (consume) yapacaktır. Bu nedenle ilk alacağı karakter 'A' olacaktır. A'yı sağlayan iki kural olması bu sistemi belirsizliğe iter. Bu nedenle ANTLR, LL(k) olarak tasarlanmıştır. Yani istenilen LookAhead miktarı ayarlanabilmektedir. Yukarıdaki örnekte LA miktarı iki alınırsa belirsizlik çözülecektir.

ANTLR çözümleyici oluşturucu, EBNF imlasına benzer bir imla ile kural tanımlanmasına izin verir.

```
kural : 'A' | 'B';
```

“|” işleci kurallar veya sözcükler arasında veya işlemini gösterir. Yukarıdaki örnek; 'A' veya 'B' ile eşleşmesi durumunda doğruluğu sağlanacak bir kuraldır. Birçok programlama dilinde olduğu gibi ANTLR kural tanımlama imlasında da parantez içine alınan veriler işleç önceliği verir.

```
kural : 'B' ('A' | 'E') 'B' ('A' | 'E');
```

Yukarıdaki örnek ile parantez içine alınan veriler, diğer ifadeler ile karışmaması için ayrı tutulmuş ve işleç önceliği kazandırılmıştır. **kural**; 'BABA', 'BEBE', 'BABE', 'BEBA' gibi veriler ile eşleşecektir.

```
kural : 'BABA' | 'BEBE';
```

Yukarıdaki kural ise 'BABA' karakter katarı veya 'BEBE' karakter katarı ile eşleşme sağlayacaktır.

```
kural : 'S' 'A'* 'T';
```

* işleci solundaki veriye etkiyerek; ya hiç kullanılmamasını ya bir kez kullanılmasını ya da bir den fazla birçok kez kullanılmasını sağlamaktadır. Eşleşme 'ST', 'SAT', 'SAAT', 'SAAAT' …. biçimlerinde olacaktır.

```
kural : 'S' 'A'+ 'T'
```

+ işleci ise en az bir defa tekrar gerektiren durumlarda geçerli olacaktır. Bu kuralın sağladığı eşleşme; 'SAT', 'SAAT', 'SAAAT' …. biçimlerindedir.

```
kural : 'S' 'A'? 'T';
```

? işleci ise seçimli (opsiyonel) ifadeler için kullanılır. Kuralın eşleştiği karakter katarları ya 'ST' ya da 'SAT' şeklinde olacaktır.

```
kural : '0' .. '9';
```

.. işleci ise arasında kalan tüm veriler için, bir VEYA işlemi uygular. Bu kural 0 ile 9 arasındaki tüm rakamlar ile eşleşecektir. Kuralın diğer bir yazım şekli şöyle olabilir:

```
kural:'0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

ANTLR sözcük çözümleyici kurallarına sözcük veya token denilir. Sözcükler büyük harf ile başlarken, diğer tüm ayrıştırıcı kuralları, küçük harf ile başlamak durumundadır.

```
SozcukCozumleyiciKurali   : 'DENEME';
sozdizimCozumleyiciKurali : 'A' | 'B';
SOZCUK_COZUMLEYICI_KURALİ : 'ORNEK';
```

Bunların dışında, ağaç ayrıştırıcı için kullanılan işleçler bulunmaktadır: Bu işleçler, ifadenin yeniden yazılmasını sağlayan ”->” işleci, sözcüğün (Token) ağaç

ayrıştırıcıya aktarılmamasını sağlayan '!' işleci ve ağaç ayrıştırıcıya kuralın kök elemanı olduğunu belirten "^" işlecidir.

ANTLR Çözümleyi Oluşturucu (Parser Generator) sisteminin bir diğer önemli özelliği ise belirgin ifadeleri tahmin edebilmesidir. Bu bayrak açık olduğu takdirde, ANTLR yukarıdaki örnekteki gibi olan belirsizlikleri deneme yanılma yolu ile tahmin eder. Bu da, daha kompleks imla tanımları yapılabilmesine olanak tanımaktadır.

Tüm bu kuralları bir hesap makinesi dili tasarlayarak daha iyi bir şekilde açıklayabiliriz:

Bir betik veya uçbirim (konsol) girdisi, derleyici veya yorumlayıcı tarafında iki aşamada çözümlenir. Birinci aşama sözcük çözümleme (lexing), ikinci aşama ise söz dizim çözümlemedir (parsing). Bu çözümleme, çeşitli kurallara göre yapılmalıdır. Ve bu kuralları oluşturacağımız da bir betimleme dili olmalıdır. İşte bu betimleme dili **BNF**'tir.

Oluşturacağımız lexer ve parser kural dosyası şu şekildedir.

```
grammar HesapMakinesi;

options {
    output       = AST;
    ASTLabelType = Tree;
}
```

```
// ### SOZ DIZIM COZUMLEYICI - AYRISTIRICI (PARSER) KURALLARI

basla         :    (islem (SATIR_SONU islem)* SATIR_SONU?)?;
// basla kurali icin gerekli olanlar
//   * Bos bir ifadeye imkan taninabilmeli. Hicbir veri
//     girilmese de
//     soz dizim dogru olmalidir.
//   * Islem Betigin son islemi ise satir sonu gelmeyebilir.
//     Bunun haricinde her islemden sonra satir sonu
//     gelmelidir.
//   * Iki bagimsiz islem arada islec (operator) olmadan
//     asla arka arkaya gelmemelidir.

islem             : oncelikli_islem
                    ((ARTI^ | EKSI^) oncelikli_islem)*
                  ;
oncelikli_islem   : SAYI ((CARPI^ | BOLU^) SAYI)*;
// ### SOZCUK COZUMLEYICI (LEXER) KURALLARI

SATIR_SONU         :    ('\u000D' | '\u000A');
SAYI               :    RAKAM+;
ARTI               :    '+';
EKSI               :    '-';
CARPI              :    '*';
BOLU               :    '/';
BEYAZ_BOSLUKLAR    :    (' ' | '\t')+   {$channel = HIDDEN;};

// ### SOZCUK COZUMLEYICI ALT PARCALARI

fragment RAKAM     :    '0' .. '9';
```

BNF ile **DFA** (Deterministic Finite Automata - Belirgin Sonlu Otomat) oluşturabiliriz. Önce sözcük çözümleyicisi (lexer) için **BNF** kuralları oluşturalım.

```
ARTI : '+';
```

ARTI adlı bir sözcüğümüz (token) oldu. Bu tarayıcı sözcüğüne (token) atanan karakter ise '+'dır. Benzer şekilde diğer işleçlerimizi (operatör) de yazalım.

```
EKSI  : '-';
CARPI : '*';
BOLU  : '/';
```

Bunlara ilaveten satır sonlarını gösteren de bir kural eklememiz gerekir.

```
SATIR_SONU : ('\u000D' | '\u000A') ;
```

SATIR_SONU sözcüğü ya **0D** ya da **0A** onaltılık değerlere sahip olacaktır. **BNF** dilinde anlaşıldığı gibi, **'|'** düşey boru işareti **'yada' (OR)** manasına gelir.

Yazdığımız bu dilde yok sayılması gereken boşluk karakterlerini ise aşağıdaki kural ile tanımlayalım.

```
BEYAZ_BOSLUKLAR : (' ' | '\t')+ ;
```

DFA'nın bir başka şekli olan Düzenli İfadeler (regular expressions) ile uğraşmış olan geliştiriciler bilirler ki, + işleci birden fazla tekrar manasına gelir. * ise, 0'dan fazla tekrar manasına gelir. Örnek kural üzerinde gösterelim:

```
SAAT : 's' 'a'+ 't'
```

Yukarıdaki kural 'sat', 'saat', 'saat', 'saaaaaaaaat' ifadeleri ile eşleşecektir. Fakat 'st' ile eşleşmeyecektir. Çünkü + işleci, 1 ve 1'den fazla tekrar manasına gelir. * işleci ise, **0** veya daha fazla tekrar manasına geleceğinden 'st' ile de eşleşebilir.

```
SAAT : 's' 'a'* 't'
```

Yukarıdaki kural, 'st', 'sat', 'saat', 'saaaaaaat' gibi ifadeler ile eşleşir. Benzer şekilde **BEYAZ_BOSLUKLAR** sözcüğü (token) 'sekme', 'boşluk', 'sekme-boşluk',

'bosluk-sekme', 'sekme-sekme', 'sekme-sekme-sekme', 'boşluk-boşluk-sekme' gibi daha birçok eşleşmeye sahiptir. Yani betiğimizdeki tüm boşluk ifadelerini kapsar.

```
RAKAM : '0' .. '9';
```

RAKAM sözcüğü, 0'dan 9'a kadar tüm rakamlar ile eşleşir. Yukarıda aralık belirten '..' işleci aslında birçok **veya** işleminin kısa yoludur. Yukarıdaki kural şu şekilde de yazılabilirdi.

```
RAKAM : '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9';
```

Tüm kurallara ilaveten, son sözcük çözümleyici (lexer) kuralımız ise şu şekilde olmalıdır:

```
SAYI : RAKAM+;
```

SAYI sözcüğü (token), bir veya birden fazla rakamın birleşmesinden oluşacaktır. '1', '12', '1000', '0122' gibi birçok eşleşme **SAYI** sözcüğü ile ifade edilebilir.

Örnek hesaplama ifademiz:

```
1 + 56 - 31 / 89 * 21
```

Bu ifadeyi sözcük çözümleyici şu şekilde çözümler.

```
SAYI ARTI SAYI EKSI SAYI BOLU SAYI CARPI SAYI
```

Tabi şu anda yaptığımız ayrıştırıcı, gerekli sözcükleri çözümlüyor. Fakat söz dizim konusunda sorunları bulunmaktadır. Örneğin:

```
++4423412++++++++++3444++--++ 14324 142314
```

İfadesi de benzer şekilde sözcüklere ayrıştırılacaktır. Bu bağlamda söz dizim kurallarımız olması gerekir. Bu işlemi ise Parser dediğimiz söz dizim çözümleyicisi yapacaktır. Bu kurallar sayesinde işleç öncelikleri de belirleyebiliriz.

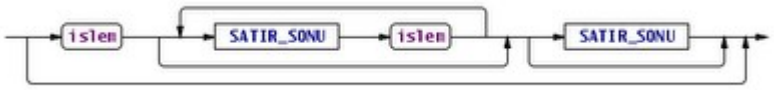
Öncelikle bir kök kuralımız olması gerekmektedir. Bu kök kural şunları gözetmelidir:

- Boş bir ifadeye imkân tanınabilmeli ve hiçbir veri girilmese de söz dizim doğru olmalıdır.
- İşlem betiğin son işlemi ise satır sonu gelmeyebilir. Bunun haricinde her işlemden sonra satır sonu gelmelidir.
- İki bağımsız işlem, arada operatör olmadan asla arka arkaya gelmemelidir.

Bu veriler ışığın da kuralımız şu şekilde olabilir:

```
basla : (islem (SATIR_SONU islem)* SATIR_SONU?)? ;
```

Kuralın **DFA** grafiği yukarıdaki Şekil 11'deki gibi olabilir. Kural direkt geçilebiliyor, bu da demek oluyor ki boş bir ifade de kabul edilebilir. Ve bir işlemden sonra **SATIR_SONU** gelebilir veya gelmeyebilir, ama iki işlemden sonra arada mutlaka **SATIR_SONU** olmalıdır. Fakat son işlemden sonra **SATIR_SONU** ister konulabilir, isterse konulmayabilir.



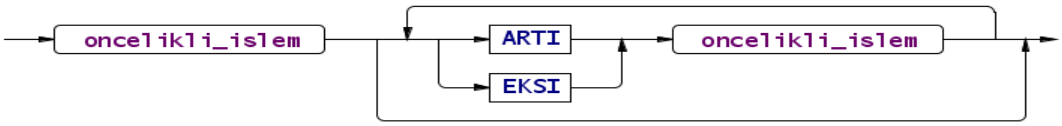
Şekil 11: basla Kuralı Söz Dizim Gösterimi

Yukarıdaki kurallardan sonra işlem kuralını açalım. İşlem kuralında önce bir toplama-çıkarma işlemi yapıyoruz. Böylelikle işleç önceliğini çarpma ve bölmeye vermiş oluruz.

Kural şu şekilde olmalıdır.

```
islem : oncelikli_islem ((ARTI | EKSI) oncelikli_islem)*;
```

Bir öncelikli işlemden sonra, başka bir veya birden fazla öncelikli işlem geliyor ise mutlaka **ARTI** veya **EKSI** ile birleştirilmelidir.

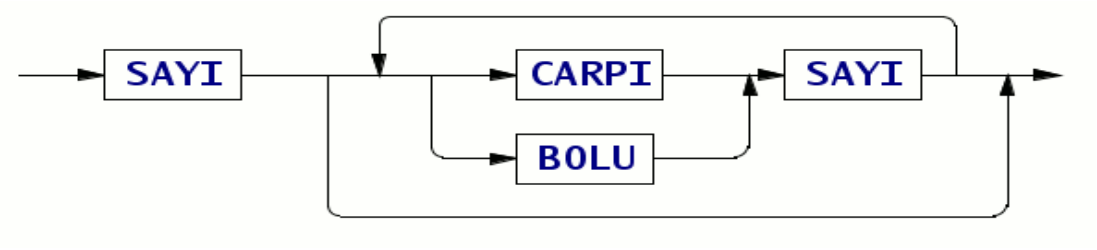


Şekil 12: islem Kuralı Söz Dizim Gösterimi

oncelikli_islem kuralımız ise Şekil 12'dekine benzer bir ifade olacaktır.

```
oncelikli_islem : SAYI ((CARPI | BOLU) SAYI)*;
```

Bir sayı tek veya birden çok çarpı/bölü işleçleri ile birleştirilmiş olmalıdır.



Şekil 13: oncelikli_islem Kuralı Söz Dizim Gösterimi

Şimdi bir örnek üzerinde kuralların kullanımından bahsedelim.

```
1 + 15 / 5 + 1 * 1
```

Öncelikle yukarıdaki ifadeyi söz dizimin kök kuralı olarak irdeleyelim.

```
islem
```

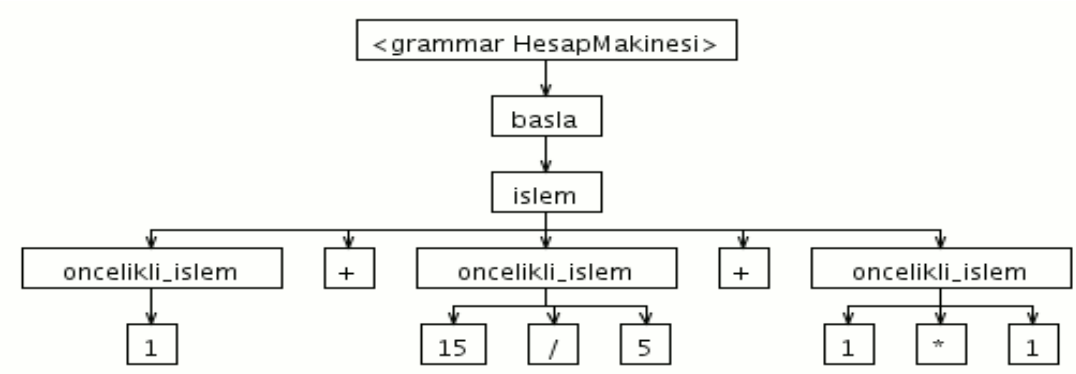
İfade hiçbir **SATIR_SONU** ile ayrılmamış tek bir işlemdir. 'islem' kuralı nazarından bakarsak şöyle çözümlenecektir.

oncelikli_islem **ARTI** oncelikli_islem **ARTI** oncelikli_islem

Görüldüğü gibi, islem ifadesi **3** farklı kola ayrıldı.

```
1. işlem : 1
2. işlem : 15 / 5
3. işlem : 1 * 1
```

Bu söz dizim kurallarının yanında, bir de ağaç yapısı oluşturmuş olduk. Ağaç yapısı ise şu Şekil 14'deki gibidir.



Şekil 14:Örnek İşlem için Ayrıştırma Ağacı Gösterimi

Bir ağaç söz dizim ayrıştırıcı ile oluşturulan ifadeleri inceleyebiliriz. Örnek işlemimizin **LISP** stili ağaç yapısı şu şekildedir:

```
(+ (+ 1 (/ 15 5)) (* 1 1))
```

Böyle bir ağaç yapısını ayrıştırmamız için gelen verileri analiz edecek bir de ağaç çözümleyiciye (tree parser) ihtiyacımız bulunmaktadır. Ağaç çözümleyicinin kuralları ise şu şekildedir:

```
tree grammar HesapMakinesiTreeParser;

options {
    tokenVocab   = HesapMakinesi;
    ASTLabelType = Tree;
}

// ### AGAC SOZ DIZIM COZUMLEYICI - AYRISTIRICI (PARSER)
KURALLARI

basla
    :   a = ifade {System.out.println(a);}
    ;

ifade returns [int ifd]
    :   ^(ARTI  a = ifade b = ifade {ifd = a + b;})
    |   ^(EKSI  a = ifade b = ifade {ifd = a - b;})
    |   ^(CARPI a = ifade b = ifade {ifd = a * b;})
    |   ^(BOLU  a = ifade b = ifade {ifd = a / b;})
```

```
    |   s = SAYI {ifd = (int) new Integer(s.getText());}
    ;
```

Yukarıdaki antlr tarzı BNF imlasında `(+ (+ 1 (/ 15 5)) (* 1 1))` ağaç yapısı yukarıdan aşağı doğru işletilir. Kıvrık parantezlerdeki Java kodları birçok ifade oluşturup, nihayet kök kural olan 'basla' biterken, konsola bulunan ifade yazdırılır.

Bu kurallar ile antlr kullanılarak 3 adet dosya elde ederiz:

1. HesapMakinesiLexer.java
2. HesapMakinesiParser.java
3. HesapMakinesiTreeParser.java

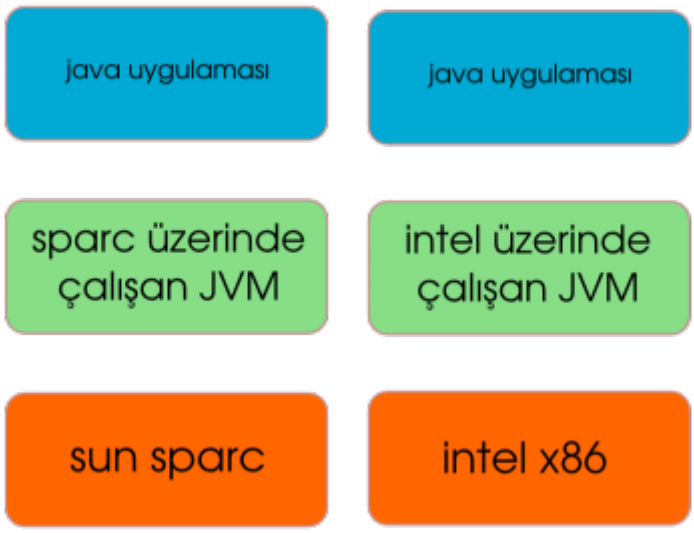
Bu 3 sınıfı çağıran ana programımız ise şu şekilde olabilir.

```
import org.antlr.runtime.*;
import org.antlr.runtime.tree.*;
import jline.*;

public class HesapMakinesi {

  public static void main(String[] args) throws Exception {

    ConsoleReader okuyucu = new ConsoleReader();
    String satir;

    while ((satir = okuyucu.readLine(">>> ")) != null) {
      CharStream dizge = new ANTLRStringStream(satir);
      HesapMakinesiLexer sozcukC = new
        HesapMakinesiLexer(dizge);
      CommonTokenStream sozcukler Stream(sozcukC);
      HesapMakinesiParser sozdizimC = new
        HesapMakinesiParser(sozcukler);
      HesapMakinesiParser.basla_return agac =
        sozdizimC.basla();

      // AGAC YAPISINI LISP STILI GORUNTULEMEK ICIN
      // ASAGIDAKI IFADE KULLANILABILIR
      System.out.println(
        "agac yapısı : " + ((Tree)agac.tree).toStringTree()
```

```
        );

        CommonTreeNodeStream dugumler = new
          CommonTreeNodeStream((Tree)agac.tree);
        HesapMakinesiTreeParser agacC = new
          HesapMakinesiTreeParser(dugumler);
        agacC.basla();
      }
    }
}
```

2.8. Java ve JVM Altyapısı

Java ile yazılan programlar, platform bağımsızlığına sahip uygulamalardır. Gnu/linux, Solaris, Windows, DOS, AIX, BeOS, MacOS, OS/2 gibi birçok işletim sisteminde kayıpsız çalışabilmektedir. Bu olağanüstü taşınılabilirlik, Java Sanal Makinesi (Java Virtual Machine) sayesinde olmaktadır. Java Sanal Makinesi tıpkı bir bilgisayar gibi çalışır. Bir bilgisayar nasıl uygulamaların makine kodlarını işliyor ise Java Sanal Makinesi ByteCode adı verilen kodları da işler [24].



Şekil 15: JVM'de Platformdan Bağımsız Uygulama Çalıştırma

Yukarıdaki grafikte, iki faklı işlemci mimarisi görülmektedir. Resmin sol kısmında, Sun SPARC işlemci mimarisine sahip bir sistemde, sisteme özgü derlenmiş Java sanal makinesi çalışmaktadır. Resmin sağ kısmında ise Intel platformunda çalışan bir sanal makine mevcuttur. Resmin tepe noktasında ortak bir java uygulaması vardır. Java uygulaması platform ne olursa olsun, o platforma ait java sanal makinesi üzerinde çalışmaktadır.

JVM üzerinde .class uzantısına sahip dosyalar çalışmaktadır. Bu dosyalar Java sınıf yapılarını içermektedir [25].

Örnek [26] bir Java uygulama döngüsüne ait ByteCode dönüşümü şu şekildedir:

```
outer:
 for (int i = 2; i < 1000; i++) {
  for (int j = 2; j < i; j++) {
    if (i % j == 0)
      continue outer;
  }
  System.out.println (i);
 }

 Code:
  0:   iconst_2
  1:   istore_1
  2:   iload_1
  3:   sipush  1000
  6:   if_icmpge       44
  9:   iconst_2
  10:  istore_2
  11:  iload_2
  12:  iload_1
  13:  if_icmpge       31
  16:  iload_1
  17:  iload_2
```

```
  18:  irem             # remainder
  19:  ifne    25
  22:  goto    38
  25:  iinc    2, 1
  28:  goto    11
  31:  getstatic       #84; //Field
java/lang/System.out:Ljava/io/PrintStream;
  34:  iload_1
  35:  invokevirtual   #85; //Method
java/io/PrintStream.println:(I)V
  38:  iinc    1, 1
  41:  goto    2
  44:  return
```

Genelde JVM denilince akla Sun Microsystems'e ait HotSpot gelmektedir. Fakat hali hazırda Java sınıflarını aynı şekilde işleyen birçok farklı sanal makine mevcuttur. Bunlardan bazıları şöyledir:

- Apache Harmony
- BEA JRockit
- GCJ
- Squawk
- IKVM
- JX
- Kaffe
- SableVM
- CACAO
- IcedTea

Java Sanal Makinesi, J2ME mikro sürümü ile mobil pazarda ve J2EE ile web ve kurumsal uygulamalar düzeyinde uygulama çatısı sunmaktadır.

2.9. Yapılan İlişkili Çalışmalar

İlişkili çalışmalara değinmek için öncelikle Javy programlama dilini tam anlamıyla tanımlamak gerekmektedir. Bu manada, bu programlama diline ait bir

künye oluşturabiliriz. Javy programlama dilinin varsayılan imlasını aşağıdaki kategorilere ayırabiliriz:

- **Paradigmalar:** Nesne Yönelimli, Yapısal, Zaruri
- **Tip Denetimi Disiplini:** Devingen, Zayıf, Güvenli
- **Çalıştığı İşletim Sistemi:** Çapraz Platform (Java'nın çalıştığı tüm platformlar)

Javy, çok imlalı bir dil altyapısı olduğundan, farklı paradigma ve tip denetimine sahip diller türetmekte mümkündür. Fakat Cezve Dil Altyapısının varsayılan imlası yukarıdaki sayılan özellikleri bünyesinde barındırmaktadır.

2.9.1. Java Platformunda Çalışan Betik Alternatif Betik Diller

Java kendi başına betik bir programlama dili değildir. Yapısı gereği, bu dil ile yazılan diller derlenmek zorundadır. Java'nın bu eksiğini gören birçok programcı, Java için bir betik dil projesi başlatmıştır. Bunlardan bazıları şu şekildedir:

- Groovy [27], Hem derlenebilir hem de betik olarak çalıştırılabilir yapısı ile son yılların en çok öne çıkan JVM tabanlı programlama dilidir. İmlası Python, Perl, Ruby ve Smalltalk gibi dillerden esinlenen öğeler içermekle beraber genel yapı itibari ile Java'da olduğu şekilde kıvrık parantezler barındırır.

- Jython [28] programlama dili adından da anlaşılacağı üzere Python programlama dilinin bir Java uyarlamasıdır. Bir çok Python kütüphanesi Jython için yeniden ele alınarak yazılmıştır.

- BeanShell [29], J2SE 1.4 sürümü Java imlasını betik bir şekilde çalıştırmayı hedefleyen bir betik programlama dilidir. Ve openoffice.org gibi benzer projelerde scripting dili olarak kullanılmaktadır.

- JavaScript temelde Java'ya bağlı bir betik programlama dili değildir. Fakat mozilla grubunun başlattığı Rhino motoru ile Java altyapısında çalıştırılabilir hale getirilmiştir.

- JRuby [30] adından oldukça söz ettiren bir Ruby JVM uyarlamasıdır. Temel mantık olarak Jython'un yaklaşımına oldukça benzemektedir. Ruby kütüphaneleri JRuby için baştan ele alınmaktadır. Bunun yanında Ruby On Rails gibi Ruby'nin beğenilen özellikleri de benzer şekilde aktarılmıştır. Java'nın da desteklediği bu proje Netbeans IDE üzerinde de çalışabilmektedir.

Diğer Java Tabanlı Betik Programlama Dilleri:

Sayılan belli başlı Java tabanlı programlama dillerine ilaveten hali hazırda oldukça fazla alternatif dil bulunmaktadır. Bunlardan bazıları şöyledir:

Jelly, Kawa, DynamicJava, Scala, Tcl/Java, ObjectScript, Judoscript, Yoix, Nice, Simkin, Jscheme, SICS, pnuts, Sleep, seppia, ePascal, LuaJava, fscript, Janino, Jbasic, InstantJ, Jatha

2.9.2. Kısmen Özelleştirilebilir Programlama Dili Çalışmaları

Javy Programlama Dili ve Cezve Dil Altyapısı için tam olarak muhadil özelleştirilebilir çalışmalar bulunmamakla birlikte kısmen özelleştirilebilir çalışmalar mevcuttur.

- Logix [31] bir çok programlama dili paradigmasını Python'un esnek yapısı ile aynı anda sağlamayı hedefleyen bir programlama dili ve altyapısı sunmaktadır. Bu yapı sayesinde tez çalışmamız ile en çok örtüşen literatürdeki çalışmadır.

- Frag [32] kendisini kesilip-biçilebilir (tailorable) olarak nitelendirmektedir. Projenin amacı yeni bir tasarıma veya programlama durumuna kendini dönüştüren bir yapı oluşturmaktır.

3. CEZVE VARSAYILAN İMLASI : JAVY

Şekil 16: Javy Programlama Dili

Cezve Dil Altyapısı, temelde çok imlalı bir programlama çatısı sunar. Fakat bu altyapının varsayılan bir imla üzerinde koşması gerekmektedir. Bu nedenle kolay programlanabilir ve üzerinde çalıştığı Java imlasına benzeyen bir yapıda olması uygun görülmüştür. Bunun yanında hızlı uygulama geliştirme adına programcıya kolaylık sağlayacak birçok alternatif kısayol da içerisine eklenmiştir.

Programcılar, Javy imlasını kullanarak Java imlasından çok fazla uzaklaşmadan betik programlar yazabilmektedirler. Bu imla hazırlanırken Python, Ruby, JavaScript gibi dillerin kullanıcıya sunduğu birçok avantajı da üzerinde barındıracak şekilde tasarlanmıştır.

Javy dilinin bahsi geçen şekli, sürekli güncellendiğinden en güncel sürümünü tezin eklerinin belirtildiği ilişkili proje sitesinden takip edebilirsiniz.

3.1. İmlada Doğrudan Tanımlanabilen Temel Veri Tipleri

Cezve Dil Altyapısı, Java ile paralel başvuru tipleri sunmaktadır. Bu veri tiplerine Cezve Dil Altyapısının detaylı açıklamasında değinilmiştir. Birçok temel başvuru tipi Java'daki muadillerinin metot ve özelliklerini içermekle birlikte, ek olarak Cezve Dil Altyapısının sunduğu metot ve özellikleri de barındırır. Java'nın miras alınamayan temel veri tipi sınıflarının yanında, genişlemeye imkân tanıyan yapısı ile Cezve temel tipleri, programcıya işleç etkileşimi için geniş bir yelpaze sunmaktadır.

Cezve dil altyapısı her hangi bir java sınıfını örneklendirmeye imkân tanıdığı için, doğrudan Java sınıfları veya varsa genişletilmiş Cezve muadilleri kullanılabilir. Javy sınıf örneklendirmesine ihtiyaç duymadan bazı Cezve sınıflarının doğrudan tetiklenmesine imkân tanır.

3.1.1. Boş (Null) Veri Tipi

Null, Java programlama dilinde de olduğu gibi bir başvuru (referans) tipi değildir. Diğer bir değişle bir nesneyi temsil etmez. Örneklendirilmemiş her değişken ve başvurusu bulunmayan her belirteç bu veri tipindedir. Bir değişkenin tanım tipi ne

olursa olsun bir başvurusu yok ise yani örneklendirilmiş bir sınıf veya ilkel bir tip (primitive type) değil ise, değeri her zaman null olacaktır.

```
null
```

Javy dilinde null veri tipi tanımlaması, Java'da olduğu gibi null belirteci ile tanımlanır.

3.1.2. Boolean Veri Tipi

Boolean veri tipi, temel olarak **1 - 0**, **var - yok**, **açık - kapalı**, **evet - hayır** gibi ikili önermeleri sağlayan Boolean cebirine ait bir gösterimdir. Bazı diller tam sayı 1 değerini **Boolean True** ve tam sayı 0 değerini **Boolean False** olarak kabul etmektedir. Javy Dili, bu özelliğini Java'dan miras alır ve Boolean gösterimleri Java'da olduğu gibi “true” ve “false” şeklindedir. Javy'de bu gösterim ile başlatılan bir değişken Cezve Dil Altyapısı tarafından doğrudan **java.lang.Boolean** sınıfının bir örneği olarak başlatılır. **Boolean** veri tipi, Java'daki **boolean** ilkel tipini (primitive type) örten bir başvuru tipidir.

```
true, false
```

Javy dili yukarıdaki iki önermeden biri ile değerlendirilmiş her değişkeni, Boolean sınıfının bir örneği olarak kabul edecektir.

3.1.3. Karakter (Character) Veri Tipi

Javy Dili'nde Karakter (Character) veri tipi tek karakterlik/UTF-8 kodlamaya sahip dizge belirtimi için kullanılır ve Java imlasındaki gibi tek tırnak ”’” ile ifade edilir. Javy imlası bir adet Kaçış Dizisi (Escape Sequence) karakterini de tek tırnaklar arasında kabul etmektedir. Character veri tipinin Java eş değeri, **java.lang.Character** sınıfıdır. **Character** veri tipi, Java'daki **char** ilkel tipini (primitive type) örten bir başvuru tipidir.

```
'a', 'd', 'ş', 'f', '\n'
```

gibi tek tırnak içine alınmış tek karakterlik dizgeler veya kaçış dizisi (Escape Sequence) Cezve dil altyapısı tarafından java.lang.Character sınıfının bir örneği olarak nitelendirilir.

3.1.4. Dizge/Karakter Katarı (String) Veri Tipi

Dizge/Karakter Katarı (String) veri tipi kullanımı, Java imlasındakine oldukça benzer bir yapıdadır. Java'daki gibi çift tırnaklar arasındaki dizgeler, String yani karakter katarı olarak kabul edilir. Buna ilaveten Javy Dili, tek tırnak kullanımına da olanak tanımaktadır. Birden fazla karaktere sahip tek tırnak içerisinde yazılan ifadeler, String veri tipi olarak değerlendirileceklerdir. Buradaki ayrım oldukça keskindir. Tek tırnak ile yazılmış bir ifade, tek bir karaktere karşılık geliyor

ise, Karakter (Character) veri tipidir. Eğer birden fazla karaktere karşılık geliyor ise, Dizge/Karakter Katarı veri tipi olarak nitelendirilecektir.

```
'deneme', "deneme", "merhaba dünya", "a", "?", '\n\n',
"\tmerhaba dünya"
```

gibi tek tırnaklar içerisine alınmış birden fazla karakter veya çift tırnak içerisine alınmış karakterler, String olarak değerlendirilir. Cezve bu veri tiplerini java.lang.String sınıfının örneklendirilmiş bir nesnesi olarak kabul edecektir.

Bunlara ilaveten Javy, Python'dan miras aldığı çok satırlı karakter katarlarını da destekler. Üçlü bir çift tekli tırnak veya çiftli tırnak arasına çoklu satıra sahip dizge yerleştirilir.

```
"""
Korkma, sönmez bu şafaklarda yüzen al sancak
Sönmeden yurdumun üstünde tüten en son ocak.
O benim milletimin yıldızıdır parlayacak!
O benimdir, o benim milletimindir ancak!
"""
```

Tek'li tırnak kullanımı da benzerdir.

```
'''
Âyinesi iştir kişinin lafa bakılmaz Şahsın görünür rütbe-i aklı
eserinde.
Ziya Paşa
'''
```

3.1.5. Tek Bayt Uzunluğunda Tam Sayı (Byte) Veri Tipi

Java'daki **byte** ilkel tipini (primitive type) örten **Byte** başvuru (referans) tipi, Javy imlasında doğrudan örneklendirilebilir. Javy Dili'nde b veya B son ekine sahip tam sayı ifadeler, **Byte** veri tipi olarak değerlendirilecektir. **Byte** veri tipi, 2^7 - 1 (127) azami ve -2^7 (-128) asgari değeri aralığındaki tam sayıları kapsar.

```
-41b, 23B, 0b, 30b
```

gibi b veya B son ekine sahip tam sayı sözcükler, Cezve Dil Altyapısı tarafından **java.lang.Byte** sınıfının bir örneği olarak nitelendirilirler.

3.1.6. Kısa Tam Sayı (Short) Veri Tipi

Java'daki **short** ilkel tipini (primitive type) örten **Short** başvuru (referans) tipi, Javy imlasında doğrudan örneklendirilebilir. Javy Dili'nde s veya S son ekine sahip tam sayı ifadeler, **Short** veri tipi olarak değerlendirilecektir. **Short** veri tipi, 2^{15} - 1 (32767) azami ve -2^{15} (-32768) asgari değeri aralığındaki tam sayıları kapsar.

```
-1s, 12S, 21s, 0s
```

gibi s veya S son ekine sahip tam sayı sözcükler Cezve Dil Altyapısı tarafından **java.lang.Short** sınıfının bir örneği olarak nitelendirilirler.

3.1.7. Tam Sayı (Integer) Veri Tipi

Birçok dilde en çok kullanılan numerik tip 32 bit tam sayı olan (integer) veri tipidir. Java'daki **int** ilkel tipini (primitive type) örten **Integer** başvuru (referans) tipi Javy imlasında doğrudan örneklendirilebilir. Javy Dili'nde bir son eke sahip olmayan tüm tam sayı ifadeler, **Integer** veri tipi olarak değerlendirilecektir. **Integer** veri tipi, 2^{31} - 1 (2147483647) azami ve -2^{7} (-2147483648) asgari değeri aralığındaki tam sayıları kapsar.

```
1, 134, 35, –45, 0, 422142, –10000000
```

gibi bir son eke sahip olmayan tam sayı sözcükler, Cezve Dil Altyapısı tarafından **java.lang.Integer** sınıfının bir örneği olarak nitelendirilirler.

3.1.8. Uzun Tam Sayı (Long) Veri Tipi

Java'daki **long** ilkel tipini (primitive type) örten **Long** başvuru (referans) tipi, Javy imlasında doğrudan örneklendirilebilir. Javy Dili'nde l veya L son ekine sahip tam sayı ifadeler, **Long** veri tipi olarak değerlendirilecektir. **Long** veri tipi, 2^{63} - 1 (9223372036854775807) azami ve -2^{63} (-9223372036854775808) asgari değeri aralığındaki tam sayıları kapsar.

```
–12131231, 12L, 45l, 0L
```

gibi l veya L son ekine sahip tam sayı sözcükler, Cezve Dil Altyapısı tarafından **java.lang.Long** sınıfının bir örneği olarak nitelendirilirler.

3.1.9. Ondalık Sayı (Float) Veri Tipi

Float Veri tipi, Java ve Javy dillerinin ön tanımlı ondalık sayı veri tipidir. Java'daki **float** ilkel tipini (primitive type) örten **Float** başvuru (referans) tipi, Javy imlasında doğrudan örneklendirilebilir. Javy Dili'nde f veya F son ekine sahip ondalık ve tam sayı ifadeler ve hiç bir ön eke sahip olmayan ondalık sayı ifadeler, **Float** veri tipi olarak değerlendirilecektir. **Float** veri tipi, $(2-2^{-23}).2^{127}$ azami ve 2^{-149} asgari değeri aralığındaki ondalık sayıları kapsar.

```
-81.34, 12.0, 45.312, 0.0, 34f, 23.4f, -45f, -123213.33f
```

gibi f veya F son ekine sahip tam sayı ve ondalık sözcükler ile hiç bir son eke sahip olmayan ondalık sayı sözcükleri, Cezve Dil Altyapısı tarafından **java.lang.Float** sınıfının bir örneği olarak nitelendirilirler.

3.1.10. Uzun Ondalık Sayı (Double) Veri Tipi

Java'daki **double** ilkel tipini (primitive type) örten **Double** başvuru (referans) tipi Javy imlasında doğrudan örneklendirilebilir. Javy Dili'nde d veya D son ekine sahip ondalık ve tam sayı ifadeler, **Double** veri tipi olarak

değerlendirilecektir. **Double** veri tipi, $(2-2^{-52}).2^{1023}$ azami ve 2^{-1074} asgari değeri aralığındaki ondalık sayıları kapsar.

```
-12312.312314641321d, -13123D, 34d, 41.41221d
```

gibi d veya D son ekine sahip tam sayı ve ondalık sözcükler, Cezve Dil Altyapısı tarafından **java.lang.Double** sınıfının bir örneği olarak nitelendirilirler.

3.1.11. Dizi (Array) Veri Tipi

Java ve Java Sanal Makinesi, her nesnenin ve ilkel tipin (primitive type) bir dizisini oluşturabilme yetisine sahiptir. Bu diziler ise Java'nın bütünleşik nesneleridir. Java temel kütüphanesinde bir sınıfa karşılık gelmezler. Java örneklendirmeyi bir Yapıcı metot ile değil doğrudan imla tetiklemesi ile yapmaktadır.

```
String[] dizgeler = new String[10];
```

ifadesi 10 elemana sahip bir karakter katarı (String) dizisi oluşturacaktır. Java imlası dizi elemanları atamak için de bir kısa yol sağlamaktadır.

```
String[] meyveler = new String[] {"elma", "portakal", "muz"};
```

Bu sayede, 3 elemanlı bir dizi tanımlanmış, elemanları da sırası ile atanmış olmaktadır.

Javy Dili, bu konuda Python listelerinin imlasını tercih etmektedir. Python'da listeler Köşeli parantez içinde ve virgülle ayrılmış elemanlar şeklinde ifade edilir.

```
meyveler = ["elma", "portakal", "muz"]
```

Javy yaklaşımı da benzer şekildedir. Aşağıda boş bir nesne dizisi gösterilmektedir:

```
[]
```

Farklı nesne gruplarını içeren bir başka dizi:

```
['a', 2, -45]
```

3.1.12. Dizi Liste (ArrayList) Veri Tipi

java.util kütüphanesi içerisinde gelen Dizi Liste (ArrayList), temel nesne dizisi tipinin genişletilmiş bir sınıfıdır. Java, java.util.Collection arabirimi ile ArrayList gibi bir çok liste sınıfının temel şablonunu belirtmiştir. Iterable arabirimi ile de tüm liste sınıflarının kolayca döngü ifadeleri ile işlenebilir şekilde olabilmesi için bir şablon sunmaktadır. java.util paketinin en çok kullanılan sınıflarından biri olan java.util.ArrayList Java imlasında doğrudan örneklemezken, Javy imlası bu yetiyi kendi üzerinde barındırmaktadır.

Javy Dili, bu gösterim için Python tuples (Değişken Kümesi) yaklaşımını kullanmaktadır. Python tuple veri tipi, parantezler içerisine alınmış virgülle ayrılmış nesnelerden oluşmaktadır.

```
meyveler = ("elma", "portakal", "muz")
```

Javy benzer yaklaşımı, ArrayList tanımlaması için benimsemiştir.

```
("elma", "portakal", "muz")
```

İfade, Cezve Dil Altyapısı tarafından doğrudan java.util.ArrayList sınıfının örneği olarak nitelendirilir.

3.1.13. Sözlük/Hash Tablosu (Hashtable) Veri Tipi

Hash Tablosu veya Sözlük Veri tipi bir anahtara karşılık bir değer, beklenen durumlarda kullanım bulan bir veri tipidir. Java Collections çatısı altında Hashtable sınıfını sunmaktadır. Fakat diğer java.util paketi içerisindeki sınıflar gibi doğrudan imla tarafından örneklendirilememektedir. Örneklendirme için ilgili nesnelerin Yapıcı Metodları (Constructor) çağırılmalıdır.

Javy, bu konuda da Python yaklaşımını kullanmaktadır. Python Dictionary veri tipi ile anahtar değer eşleşmelerini parantezler arasında virgülle ayrılmış eşleşmeler olarak kabul etmektedir.

```
baskentler = {'turkiye': 'ankara', 'fransa':'paris'}
```

Javy yaklaşımı da benzer şekildedir.

```
{"turkiye":"ankara", "fransa":"paris", "ingiltere":"londra"}
```

Javy yukarıdaki gibi bir gösterimi, Cezve Dil Altyapısına ilişkili ayrıştırıcı sözcük ve parametreleri ile aktaracaktır. Cezve, ifadeyi java.util.Hashtable sınıfının örneği olarak nitelendirecektir.

3.2. Paketler ve Sınıflar

Nesne Yönelimli (Object Oriented) Programlama dilerinin birçoğu beraberinde programcılara geniş bir nesne kütüphanesi desteği sunar. Bu kütüphaneler, dizi işlemlerinden soket bağlantılara kadar birçok işlevi içerisinde barındırır. Programcılar ihtiyacı olan kütüphaneyi derleyici ve yorumlayıcıya bağlayıp bu nesnelere de erişim sağlarlar.

Kütüphane yaklaşımı, beraberinde bir ad karmaşasını da getirir. Sınıflara verilecek isimler daha önceden kullanılmamış olmalıdır. Yoksa nesneler çakışıp kullanılamayacaklardır.

Bu karmaşanın önüne geçmek için, Ad Uzayı (Namespace) kavramı ortaya atılmıştır. Sınıflar, Ad Uzaylarına ve Alt Ad Uzaylarına dâhil olarak aynı isime sahip olurlar. Bu gösterim, işletim sistemlerindeki klasör-dosya ilişkisine benzemektedir.

Hatta benzemekle kalmayıp çoğu programlama dili, dizinleri Ad Uzayı olarak kullanır.

Java, Ad Uzayı yerine paket (package) ifadesini tercih eder. Her nesne dosyası bir dizine (paket) dâhildir. Bu sayede kullanılamak istenen paket içerisindeki sınıf, program içerisine dâhil edilir. Daha sonra dâhil edilen sınıfın metodlarına ve alanlarına erişim sağlanabilir.

Örneğin **java** paketinin alt paketi olan **util** paketi **Date** nesnesine sahiptir. **java** paketinin bir diğer alt paketi olan **sql** paketi de farklı bir **Date** nesnesine sahiptir. Bu sınıflardan kullanılmak istenen paket, program içerisine ithal edilir.

```
import java.util.Date;
```

Yukarıdaki ifade ile **Date** sınıfı kullanılabilir hale gelir. Buna ilaveten Java bir paket veya alt paket içerisindeki tüm sınıfları program içerisine ithal edip kullanılabilir yapmak için * işlecini kullanır.

```
import java.util.*;
```

Bu ifade, `java.util` paketi içerisindeki ArrayList, Hashtable, Date gibi tüm sınıfları kullanılabilir yapar. Programcıları, paket isimlerini teker teker girme zahmetinden kurtarır.

Javy, tüm Java import ifadelerini desteklemekle birlikte bunlara ilaveten kendi ek import ifadelerini de kullanıma açar. Örnek olarak virgülle ayrılmış import ifadeleri kullanılabilir.

```
import java.util.Date, java.lang.*;
```

Yukarıdaki import gösterimi, birden çok import ifadesini tek bir import anahtar kelimesi altında birleştirir. Bunlara ilaveten sınıflara takma isimler verilmesine de olanak sağlamaktadır.

```
import java.util.Date as Tarih;
```

as anahtar kelimesi ile sınıf bir yandan program içerisine alınırken bir yandanda ek bir takma isim edinir. Artık program içerisinde **Date** kullanılacak yerlerde pekâlâ **Tarih** kullanılabilecektir.

3.3. İşleçler

3.3.1. Cebir ve Dizge İşleçleri

Toplama, Pozitif Değer ve Dizge Birleştirme İşleci :

Javy dilindeki toplama işlemi, birçok dildeki benzer kullanıma sahiptir. ”+” işleci sayesinde iki sayısal ifadenin cebirsel değerleri toplanabilir.

```
1 + 34
23 + -56
```

```
41 + 34.4
-12.56 + 31.0
12b + 321b
23 + 45l
```

Bu dilin sahip olduğu veri tipleri, Cezve altyapısının sunduğu temel veri tipleri arasındaki dönüşümler neticesinde oluşur. Bu tip dönüşümlerine ileriki bölümlerde değinilecektir.

”+” işleci ek olarak dizge değerleri birleştirmeye yaramaktadır.

```
"merhaba" + " " + 'dünya'
"Savaşan Şahin F" + 16
```

+ operatöründe argümanlardan biri dizge ise, diğer veri tipi de dizgeye dönüştürülecektir.

”+” işleci, bu kullanımlarının yanında tekil (unary) kullanıldığında pozitif değer anlamını taşımaktadır.

```
+1 +56 +5.45 +12.45b
```

Çıkarma, Negatif Değer Dizge Tersleme İşleci :

Çıkarma işlemi, toplama işleminde olduğu gibi diğer birçok dil ile benzerlik gösterir. ”-” işleci ile numerik veri tipleri birbirlerinden çıkartılabilir.

```
12 - 7
34.5 - -56
56 + 23
```

”-” işleci, yukarıda da kullanım bulduğu gibi tekil (unary) gösterimlerde bir sayının negatif halini göstermektedir. Cezve dil altyapısında programcılara kolaylık sağlamak açısından ”-” işleci, karakter katarı gibi nesneler üzerinde de kullanım bulur. Ön tanımlı olarak bu gibi metodlar, Java Sanal Makinesindeki varsayılan kütüphanelerin üzerine yazılmıştır.

Örneğin:

```
-"merhaba dünya"
```

'nın sonucu aşağıdaki şekildedir.

```
"aynüd abahrem"
```

Çarpma, Dizge Çoğullama İşleci :

Çarpma işlemi de standart biçemdedir. Yıldız “*” karakteri, Java dâhil birçok programlama dilinde aritmetik çarpma işlemini gerçekleştirir.

```
13 * 45
5 * 10
45b * 23l
12.3 * 45 * 56
```

Javy dilindeki “*” işleci, Cezve altyapısı sayesinde, Python'un aynı işleç ile yaptığı dizge çoğullama işlemini gerçekleştirebilmektedir. Örnek olarak:

```
"merhaba" * 5
```

Çıktısı:

```
"merhabamerhabamerhabamerhabamerhaba"
```

şeklindedir.

Bölme, Dizge Ayrıştırma İşleci :

Bölme işlemi de standart biçimdedir. '/' karakteri, iki yanındaki numerik ifadelerden sol numerik ifadeyi, sağ numerik ifadeye bölüp değerini döndürmektedir.

```
34b / 7s
12 / 4
45 / 15l
```

Bölme '/' işleci, yalnızca Javy'de hayata geçirilen hızlı dizge ayrıştırma işlemini de gerçekleştirir.

```
"ali,veli,ahmet" / ","
```

Yukarıdaki örnek, virgülle ayrılmış değerler içeren bir dizgeyi ”,” virgül ile bölme manasına gelmektedir. Sonuç ise karakter katarı dizisi halinde verilir.

```
["ali", "veli", "ahmet"]
```

Bölüm İşleci :

Bölüm İşleci, Python dilinin aynı görevi yapan (Truncating division) işleci olan '//' karakterinden esinlenmiştir. Javy'deki ters bölü (back slash) karakteri, bir bölme işlemi sonucundaki bölüm değerini döndürür.

```
10 \ 3
```

işleminin sonucu 3 değeridir.

Üs/Kuvvet İşleci :

Üs/Kuvvet işleci, doğrudan Python imlasından alınmıştır. Javy imlasında, ”**” Çift yıldız karakterlerinin solundaki numerik ifade taban, sağındaki numerik ifade ise üs olarak nitelendirilir.

```
2 ** 8
10 ** 3
```

Mod İşleci :

Mod işlecinin Java'daki kullanımının aynısı, Javy imlasında da bulunmaktadır. % karakteri, mod işlemini tanımlar.

```
15 % 4
```

işleminin sonucu 3'dür. 15'in 4'e bölümünden kalan değer, sonuç değeri olarak döndürülür.

Dizge Birleştirme İşleci :

Php'nin sağladığı ”.” birleştirme işlecinin benzeri Javy içinde sağlanmıştır. “_” alt çizgi karakteri, Javy imlasında dizge (String) bazlı karakter birleştirme işlemlerini gerçekleştirir.

```
235 _ 4512
```

İşleminin sonucu “2354512” dizge veri tipidir. “_” işleci, argümanlar hangi veri tipinde olursa olsun, nesneleri String veri tipine dönüştürür ve dizge birleştirme işlemine tabi tutar.

Arttırma İşleçleri :

Arttırma işleci, bir cebir işleci olmasının yanında tekil (unary) bir işleçtir. Java, C, C++, PHP, Perl gibi benzer imlaya sahip dillerin hemen hepsinde bu işleç yer alır. Yalnızca değişken ve benzeri alan isimleri üzerinde etki sahibidir. Bu işleç, aynı zamanda bir atama operatörüdür. Değişkenin arttırılan değeri yine kendisine atanmaktadır.

```
sayi++, ++sayi
```

Örnekteki şeklinde değişkenin önüne veya arkasına gelerek değişken değerini arttırır. İşleç, değişken++ şeklinde kullanım bulduğunda, işleç ilk önce değişkenin değerini döndürür, daha sonra ise arttırılmış değeri kendine atar. + +değişken şeklindeki kullanımlarda ise, önce arttırılmış değeri kendine atar, daha sonra ise değişkenin yeni değerini döndürür.

Azaltma İşleci :

Azaltma işleci, bir cebir işleci olmasının yanında tekil (unary) bir işleçtir. Java, C, C++, PHP, Perl gibi benzer imlaya sahip dillerin hemen hepsinde bu işleç yer alır. Yalnızca değişken ve benzeri alan isimleri üzerinde etki sahibidir. Bu işleç aynı zamanda bir atama operatörüdür. Değişkenin azaltılan değeri, yine kendisine atanmaktadır.

```
sayi--, --sayi
```

Örnekteki gösterimde görüldüğü üzere; değişkenin önüne veya arkasına gelerek değişken değerini azaltır. İşleç, değişken– şeklinde kullanım bulduğunda, işleç ilk önce değişkenin değerini döndürür, daha sonra ise azaltılmış değeri kendine atar. –değişken şeklindeki kullanımlarda ise, önce azaltılmış değeri kendine atar, daha sonra da değişkenin yeni değerini döndürür.

3.3.2. Koşullu, Boolean ve Bit Bazlı İşleçler

Boolean cebiri, ikili önerme matematiğidir. Ve günümüz bilgisayar sistemlerinin temelini teşkil eder. VE, VEYA, DEĞİL, ÖZEL VEYA gibi mantıksal işleçler, algoritma geliştirmenin temel yapı taşlarındandır. Javy'nin bu operatörleri çeşitli farklılıklarla koşullu, bit bazlı veya doğrudan boolean matematiği ile ilişkilidir.

Koşullu VE İşleçleri :

'&&' çift “ve” (Ampersand) karakteri Java, C gibi aynı dil ailesinden gelen çoğu programlama dilinde koşullu VE işleci olarak kabul edilir. İşleç iki yanındaki değeri mantıksal VE işlemine sokar.

```
false && false  : false
false && true   : false
true  && false  : false
true  && true   : true
```

Yukarıdaki tablo, Koşullu VE işleminin doğruluk tablosudur. Koşullu VE işleminin, Boolean VE işleminden farkı: eğer soldaki değer false ise, sağdaki değer işlenmeden ifadeden çıkılır. Bu, hem algoritma işlem süresini kısaltmaya, hem de sağdaki ifadenin bir Exception (istisna) üretmesini engellemeye yöneliktir.

”&&” işlecine ek olarak Javy dili, PHP ve Python'daki gibi “and” ifadesini de sunmaktadır. Aynı işlemi gerçekleştiren iki işleçten herhangi biri kullanılabilir.

```
true and false
```

Koşullu VEYA İşleçleri :

'||' çift boru karakteri, Java ve C gibi aynı dil ailesinden gelen çoğu programlama dilinde koşullu VEYA işleci olarak kabul edilir. İşleç, iki yanındaki değeri, mantıksal VEYA işlemine sokar.

```
false || false  : false
false || true   : true
true  || false  : true
true  || true   : true
```

Yukarıdaki tablo, Koşullu VEYA işleminin doğruluk tablosudur. Koşullu VEYA işleminin, Boolean VEYA işleminden farkı, eğer soldaki değer false ise, sağdaki değer işlenmeden ifadeden çıkılır. Bu hem algoritma işlem süresini kısaltmaya, hem de sağdaki ifadenin bir Exception (istisna) üretmesini engellemeye yöneliktir.

"||" işlecine ek olarak Javy dili, PHP ve Python'daki gibi "or" ifadesini de sunmaktadır. Aynı işlemi gerçekleştiren iki işleçten herhangi biri kullanılabilir.

```
true or false
```

Bit Bazlı/Boolean VE İşleci :

'&' "ve" (Ampersand) karakteri, Java ve C gibi aynı dil ailesinden gelen çoğu programlama dilinde, bit tabanlı VE işleci olarak kabul edilir. İşleç iki yanındaki değeri, mantıksal VE işlemine sokar.

```
false & false  : false
false & true   : false
true  & false  : false
true  & true   : true
```

Yukarıdaki tablo, Bit bazlı VE işleminin doğruluk tablosudur. Bit bazlı VE işleminin, Koşullu VE işleminden farkı, soldaki ve sağdaki değerler beraber işlenip ifadeden çıkılır.

Bit Bazlı/Boolean VEYA İşleci :

'|' “boru” karakteri, Java ve C gibi aynı dil ailesinden gelen çoğu programlama dilinde, bit tabanlı VEYA işleci olarak kabul edilir. İşleç iki yanındaki değeri, mantıksal VEYA işlemine sokar.

```
false | false  : false
false | true   : false
true  | false  : false
true  | true   : true
```

Yukarıdaki tablo, Bit bazlı VEYA işleminin doğruluk tablosudur. Bit bazlı VEYA işleminin, Koşullu VEYA işleminden farkı, soldaki ve sağdaki değerler beraber işlenip ifadeden çıkılır.

Bit Bazlı/Boolean ÖZEL VEYA İşleci :

'^' “şapka” karakteri, Java ve C gibi aynı dil ailesinden gelen çoğu programlama dilinde, bit tabanlı ÖZEL VEYA işleci olarak kabul edilir. İşleç iki yanındaki değeri, mantıksal ÖZEL VEYA işlemine sokar.

```
false ^ false  : false
false ^ true   : false
true  ^ false  : false
true  ^ true   : true
```

Yukarıdaki tablo, Bit bazlı ÖZEL VEYA işleminin doğruluk tablosudur. “^” işlecine ek olarak Javy dili, PHP ve Python'daki gibi “xor” ifadesini de sunmaktadır. Aynı işlemi gerçekleştiren iki işleçten herhangi biri kullanılabilir.

```
true xor true  : true
```

Boolean DEĞİL İşleci :

'!' “ünlem” karakteri, Java ve C gibi aynı dil ailesinden gelen çoğu programlama dilinde, Boolean DEĞİL işleci olarak kabul edilir. İşleç, önüne geldiği değeri mantıksal DEĞİL işlemine sokar.

```
!false   : true
!true    : false
```

Yukarıdaki tablo, Boolean DEĞİL işleminin doğruluk tablosudur.

Bit Tabanlı TÜMLEME İşleci :

'~' “tilda” karakteri, Java ve C gibi aynı dil ailesinden gelen çoğu programlama dilinde, Bit Tabanlı TÜMLEME işleci olarak kabul edilir. İşleç, önüne geldiği değeri, Bit Tabanlı TÜMLEME işlemine sokar. TÜMLEME işleci, genelde sayısal veri tipleri üzerinde kullanım bulur. Bazı Integer veri tipleri için işlem sonuçları şu şekildedir.

```
~1         : -2
~100       : -101
~(-30001)  :  30000
```

3.3.3. Karşılaştırma İşleçleri

Javy dilinde nesne eşitlik doğrulama işleçleri, sayılar arası büyüklük karşılaştırma işleçleri ve nesne üyelik doğrulama işleci bu gruba girmektedir.

Eşitlik Doğrulama İşleçleri :

'==' çift “eşittir” karakteri, Java ve C gibi aynı dil ailesinden gelen çoğu programlama dilinde, EŞİTLİK DOĞRULAMA işleci olarak kabul edilir. İşleç, iki yanındaki değerin birbirlerine değer bakımından eşit olup olmadığını denetler. Java'da bu işlem equals() metodunun çağrılması ile olur. Java'daki ”==” işleci ile Javy'deki ”==” işleci aynı işleçler değildir. Javy'deki EŞİTLİK DOĞRULAMA işleci, PHP, Python, JavaScript gibi dillerdeki kullanım ile aynıdır.

Bu işleç sonucunda, Boolean true veya false ikili önermesinden biri döndürülür.

```
"arı" == "ayı"  : true
"arı" == "ayı"  : false
```

'!=' “ünlem” ve “eşittir” karakterleri ise; ”==” işlecinin mantıksal terslemesi olarak çalışır. Kısaca “eşit değildir” işleci denilebilir.

```
"arı" != "ayı"  : false
"arı" != "ayı"  : true
```

Denklik Doğrulama İşleçleri :

Java'daki '==' işleci, Javy'nin denklik işleci ile örtüşür. Fakat Java yorumlayıcısı en iyileştirme (optimizasyon) yaklaşımına sahip olduğundan sonuçlarda farklılık bulunabilir. Javy Denklik Doğrulama İşleci, ”===” üç adet

“eşittir” karakteri kullanılarak gerçekleştirilir. Javy ve Java'daki bir nesne örneğine karşılık gelen tüm değişkenler aslında birer nesne referansıdır. Denklik işleci, bu referansları karşılaştırır ve iki nesnenin aynı nesne olup olmadığını da denetler.

```
"kapı" === "kapı"  : false
```

Yukarıdaki örnekte, iki kapı kelimesini içeren dizgenin birbirlerine eşit olmadığı görülmektedir. Bunun nedeni; Cezve tarafından işletilirken, iki dizgenin de aynı değere sahip, farklı iki nesne örneği olarak oluşturulmasıdır.

```
true === true      : true
```

Bu örnekte; iki boolean değerde, Java Sanal Makinesi optimizasyonundan geçmiştir. Nesnelerin örttüğü kendi ilkel tiplerinin denklik karşılaştırması yapıldığından dolayı, nesneler birbilerine denk bulunmuşlardır.

Javy, denklik karşılaştırması için Python'daki “is” işlecini de kendi bünyesine katmıştır. ”===” ile tamamen aynı olan “is” işleci de benzer işlemler için kullanılabilir.

```
"kapı" is "kapı"  : false
```

”==” işlecinde olduğu gibi, Denklik Doğrulama İşlecinin de mantıksal dersleme yapılmış bir hali mevcuttur. Bu Python'daki “is not” işlecinin yerine geçen “isnot” işlecidir.

```
"kapı" isnot "kapı"  : true
```

Sayısal Büyüklük Karşılaştırma İşleçleri :

Javy ve C, Java dil ailesindeki sayısal büyüklük karşılaştırma işleçlerinin tümünü miras alır. ”<”, ”>”, ”⇐”, ”>=” işleçleri sırası ile; küçüktür, büyüktür, küçük eşit ve büyük eşit manalarına gelir. Bu işleçlere ek olarak, ”==” eşitlik doğrulama işleci de sayısal büyüklük karşılaştırması yapabilir.

```
12 < 24   : true
12 > 24   : false
12 <= 24  : true
12 >= 24  : false
1  >  1   : false
1  <  1   : false
1  >= 1   : true
1  <= 1   : true
1  == 1   : true
```

Üçlü Eğer ... Değilse İşleci :

Java, C, C++, C# gibi dillerde ternary işleci olarak bilinen bu operatör, basit bir if .. else gösterimidir.

```
<test ifadesi> ? <doğru ise dönen> : <yanlış ise dönen> ;
```

Test ifadesi boolean true değerini döndürür ise tüm ifadenin dönen değeri ? ile : karakterleri arasında kalan gösterim ifadesi olacaktır. Eğer test ifadesinin sonucu boolean false değeri ise tüm ifadenin son kısmındaki gösterim sonuç olarak döndürülür.

```
1 == 1 ? "1 sayısı 1 sayısına eşittir." : "eşit değildir."
```

Yukarıdaki örnek kullanımda 1 == 1 işlemi true sonuç verecektir. İfade sonucu ise `"1 sayısı 1 sayısına eşittir." dizgesi olacaktır.`

3.3.4. Dizi İşlem İşleçleri

Javy dili, Java dilinden farklı olarak diziler üzerinde işlem yapan işleçlere sahiptir.

Aitlik Denetleme İşleçleri :

'in' dizgesi, Javy dilinde dizi aitliği işleci olarak kullanılır ve Python dilinden alınmıştır.

```
"elma" in ["elma", "armut", muz]       : true
```

elma dizgesi, dizi içerisinde bulunduğundan sonuç **true** olarak dönecektir.

'in' işlecinin yanında Python dili, “not in” işleci de sunmaktadır. Bu temelde “in” operatör sonucunun terslenmesi ile bulunabilir. Javy dili de “not in” işlecinin muhadili “notin” operatörünü sunar.

```
"elma" notin ["elma", "armut", muz]       : false
```

Aralık Oluşturma İşleci :

Javy, ”..” aralık oluşturma işlecini Perl dilinden almıştır. Fakat aralık işleci konusunda Perl dili ile birebir örten bir kullanıma, henüz sahip değildir. İşlecin örnek kullanımı şu şekildedir:

```
1 .. 15  : [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

İki dizi işlecini beraber kullandığımızda, daha kullanışlı bir hal alacaktır.

```
4 in 1 .. 15    : true
```

Yukarıdaki ifadede 1 .. 15 kullanımı ile oluşan dizi içerisinde in operatörü ile aitlik testi yapılmıştır. Sonuç Boolean true değeri olacaktır.

3.3.5. Atama İşleçleri

Bir dilin en işlevsel araçları hiç şüphesiz değişken tanımlamalarıdır. Javy Dili, Cezve Dil Altyapısı ile beraber bu konuda oldukça esnek bir sistem sunmaktadır. Değişkenlere Java'da olduğu gibi nesne başvurularını aktarabilirsiniz. Hatta sınıf metodları, sınıf üyeleri ve paket isimlerine dahi yeni başvuru değişkenleri atayabilirsiniz. Cezve dil altyapısı, referans değişken tiplerine imkân tanımaktadır ve ilkel tipleri (primitive type) doğrudan kendisini örtebilen nesne örneklerine çevirir. Java Sanal Makinesi özyinelemeli (otomatik) olarak ilkel tipleri Java nesnelerine dönüştürebilmektedir. Cezve dil altyapısı da Java'nı bu özelliğini miras alır.

Doğrudan Değişken Atama İşleci :

Javy doğrudan değişken atama konusunda durağan tip denetimine sahip dillerin aksine, değişkenin tipini otomatik tanımayı seçmektedir. Cezve dil altyapısının sağladığı bu sistem, devingen tip denetimine sahip dillerin yaklaşımına benzemektedir. JavaScript, Python, PHP gibi betik dillerde olduğu gibi değişken bir tip belirtilmeden doğrudan tanımlanmaktadır. Javy tek satırlık değişken atama işlemini, noktalı virgülle biten basit bir ifade olarak nitelendirecektir. Daha ilerdeki bölümlerde, bu ayrıştırmanın ve Cezve dil altyapısı ile ilişkilendirilmenin nasıl olduğu konusuna değinilecektir.

Javy'nin geçerli saydığı değişken isimleri, rakam ile başlayamaz. Büyük harf, küçük harf veya alt çizgi karakteri ile başlayan isimleri, bunlara ilaveten rakamsal değerler takip eder.

```
rakam, isim, veriTabani, Sinif, _sayi_degeri, ornek_kume,
sayac01
```

yukarıdakiler doğru değişken tanımlaması olarak kabul edilecektir. Bunların yanında:

```
4Bina, ali#veli, sayi?degiskeni
```

gibi isimler ise yanlış değişken tanımlamalarıdır.

Integer tipine sahip bir değişkeni basit olarak bir değişkene şu şekilde atayabiliriz.

```
sayi = 45;
```

45 numerik ifadesi hiçbir son eke sahip olmadığı için ve ondalık bir biçeme sahip olmadığı için Javy ayrıştırıcısı tarafından Cezve dil altyapısının sunduğu Integer sözcüğü ile eşleştirilir. Ve sonuç itibari ile değişken bir Integer nesnesi başvurusu (referans) içerir.

Aşağıda doğrudan örnekleme yapacağımız Javy tiplerine ilişkin örnekler bulunmaktadır.

Karakter (Character) değişken tanımı örnekleri:

```
aHarfi = 'a';
satir_sonu = '\n';
_sekmeKarakteri01 = '\t';
```

Dizge (String) değişken tanımı örnekleri:

```
merhaba = 'merhaba Javy';
satir_sonu = '\n';
_sekmeKarakteri01 = '\t';
cokluSatirDizge = '''
Birinci satır
İkinci satır
''';

Diğer numerik tip örnekleri:
ornekByteVeriTipi = 12b;
ornek_short_veri_tipi = -34s;
ondalik_sayi = 41.34;
doubleVeriTipi = 3.14d;
```

Nesne Dizisi (Object Array) tanımı:

```
meyveler = ['elma', 'armut', 'muz'];
```

Dizi listesi (ArrayList) tanımı:

```
meyveler = ('elma', 'armut', 'muz');
```

Sözlük/Hash tablosu (Hashtable) tanımı:

```
baskentler = ('turkiye':'ankara', 'fransa':'paris',
'ingiltere':'londra');
```

Bileşik Atama İşleçleri :

Bileşik atama işleçleri, atama işleminden önce ilişkili işlemi uygular. Ve daha sonra atama yapar. Javy'nin kullandığı atama işleçleri şu şekildedir:

```
'+='         : Ekleme/Dizge Birleştirme ve atama
'-='         : Çıkarma ve atama
'*='         : Çarpma/Dizge Çoğulama ve atama
'/='         : Bölme/Dizge bölme ve atama
'\\='        : Bölüm ve atama
'**='        : Kuvvet/Üs ve atama
'%='         : Mod ve atama
'_='         : Dizge birleştirme ve atama
'&='         : Bit tabanlı VE ve atama
'|='         : Bit tabanlı VEYA ve atama
'^='         : Bit tabanlı ÖZEL VEYA ve atama
'<<='        : Bit tabanlı sola kaydırma ve atama
'>>='        : Bit tabanlı işaretsiz sağa kaydırma ve atama
'>>>='       : Bit tabanlı işaretli sağa kaydırma ve atama
```

Aşağıda bazı örnek atamalar bulumaktadır.

```
a  = 45;
a += 2;
```

sonuçta 'a' değişkeninin değeri, 47 olacaktır.

3.3.6. Nesne Aitliği ve Parça Alma İşleçleri

Nesne Aitliği İşleci :

Javy, nesnelerin alt metot ve alanlarına erişim için Java'nın da kullandığı "." (nokta) işlecini kullanır. Nokta işleci, solundaki nesne içerisinde sağdaki argümanı arayacaktır. Argüman bir alan olabileceği gibi bir metot da olabilir.

```
Integer.MAX_VALUE
```

Nokta (.) işleci ile Integer sınıfının içerisinde durağan bir alan olan MAX_VALUE'ya erişilebildi. Bu işleç sayesinde paketler, alt paketler ve bunlara ait sınıflara erişim de mümkündür.

```
java.lang.String
```

Yukarıdaki ifade, **java** paketinin içerisindeki **lang** alt paketindeki **String** sınıfına erişim sağlar. Nesne hiyerarşisi olabildiğince alt dallara ayrılabilir. Paketler, alt paketlere, alt paketler sınıflara, sınıflar alt sınıflara, alt sınıflar metot ve alanlara, bunlar da yine kendi aralarında sınıf, alt sınıf, metot ve alanlara ayrılabilir.

```
java.lang.System.out.println("merhaba javy");
```

Yukarıdaki örnekte, **java** paketinin alt paketi **lang**'ın alt ögesi olan **System** sınıfının alanı **out**'a ait bir metot olan **println** metodu çağırılmaktadır. **println** metoduna parametre olarak **String** veri tipinde **"merhaba javy"** cümlesi gönderilir.

Sınıfa Aitlik Test İşleci :

Javy Dili, Java'da kullanım bulan **instanceof** işlecini aynı şekilde kullanabilmektedir. instanceof işleci ile soldaki nesnenin sağdaki sınıfın bir nesne örneği olup olmadığı test edilir. Sonuç tahmin edileceği üzere, bir Boolean ifadesidir.

```
48 instanceof Integer
```

Yukarıdaki ifade, true değerini döndürecektir. Cezve dil altyapısında ilkel tip yoktur. Bu nedenle 48, bir Integer sınıfı örneğidir. Test sonucu, boolean doğru değeri döner. instanceof işleci ile üst sınıflarıda denetime ekleyebiliriz.

```
48 instanceof Object
```

Sonuç yine boolean true değeri olacaktır. Çünkü tüm sınıfların olduğu gibi, Integer sınıfı da Object sınıfının bir alt sınıfıdır.

Parça Alma İşleci :

Parça Alma (Slicing) işleci, dizilerde ve ilişkili Koleksiyon (java.util.Collections) nesnelerinde kullanılabilir. Java yalnızca dizi değişkenler üzerinde bu işleci kullanabilirken Javy, diziler, ArrayList, Hashtable, String gibi veri tipleri üzerinde kullanabilmektedir. Buna ilaveten parça alma işleci, Cezve dil altyapısında bahsedilecek olan operatör metot eşleşmesi sayesinde ilişkili bir metoda

bağlanabilir. Bu sayede ilişkili metoda sahip olan her nesne bu işleçten faydalanabilecektir.

```
("elma", "armut", "muz")[0]
```

Yukarıdaki ifade, `("elma", "armut", "muz")` ile 3 elemanlı bir Dizi Listesi (ArrayList) oluşturur. Parça Alma işleci ise bu Dizi Listesi'nin ilk elemanını döndürecektir. Tüm ifadenin döndürdüğü değer `"elma"` olacaktır.

```
{"elma":"meyve", "pirasa":"sebze"}["elma"]
```

Parça alma işleci, Sözlük/Hash Tablosu veri tiplerine de uygulanabilmektedir. Yukarıdaki ifadeden dönen sonuç `"meyve"` olacaktır.

3.3.7. İşleç Öncelikleri :

Parantezler ile ayrılmamış faklı operatörler içeren işlemler için işleç önceliği gerekmektedir. Bazı işlemler, diğerlerine göre daha önce yapılmalıdır. Örnek olarak Javy'nin dizi aitlik ve aralık oluşturma işleçleri arasında bir öncelik belirlemesi yapılması gerekmektedir.

```
1 in 1 .. 4
```

Bu ifadede “in” ve “..” işleçleri bulunmakta ve “..” işleci önceden bahsedildiği gibi 1 ile 4 arasında bir Tam Sayı dizisi oluşturmaktadır. in ise sol

taraftaki değerin sağ taraftaki dizide olup olmadığını denetleyen bir işleçtir. Eğer işleç önceliğimiz olmasaydı ayrıştırma işlemi şu şekilde olacaktı:

```
((1 in 1) .. 4)
```

in işleci, Integer değeri yine bir Integer değerde aramaya çalışacaktır. Fakat ayrıştırıcımızda tanımlandığı üzere, aralık işlemi aidiyet işleminden önce gerçekleşir. Bir başka değişle, ayrıştırma şu şekilde olur.

```
(1 in (1 .. 4))
```

Önce 1 .. 4 işlemi yapılacağından işlem sonucu olan tam sayı dizisi, tam sayıya argüman olarak gönderilebilir.

Javy işleç öncelikleri 18 seviyeye ayrılmıştır. Bu seviyelerden en yüksek olanı, en önce işletilecektir.

Tablo 1: İşleç Öncelikleri

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
=	?:	||	&&	|	^	&	in	..	==	<	<<	+	*	**	+
+=		or	and		xor		notin		!=	<=	>>	-	/		-
-=									===	>	>>>	_	\		~
*=									is	>=			%		!
/=									isnot	instanceof					()
\=															
**=														17	18
%=														++	.
_=														--	[]
&=															
|=															
^=															
<<=															
>>=															
>>>=															

Öncelik Değiştirme :

Birçok programlama dilinde olduğu gibi, Javy dili de öncelik değiştirmek için parantezler arasına alınmış ifadelere izin verir. Bu sayede önce yapılması istenen işlem, parantez içerisine alınır.

```
1 + 4 * 4
```

işlemi, 17 tam sayı sonucunu verecektir. 14. seviyede olan * işleci, önce yapılıp, daha sonra 13. seviyede olan + işleci yapılacaktır. Önce + işlecinin uygulanması için, ifadenin parantez içerisine alınması yeterli olacaktır.

```
(1 + 4) * 4
```

İşlem sonucu 5 * 4 = 20 olacaktır.

3.4. İfadeler

Değişken atama, nesne örneklendirme, nesne metot ve alan erişimi, paket sınıflarına erişim, bileşik koşul ifadeleri, bileşik döngü ifadeleri, bir Üst Düzey Programlama Dili'nin olmazsa olmazlarındandır. Javy temel ifadelerin kullanımında Java ve JavaScript benzeri C tarzı bir imla biçimine sahiptir.

3.4.1. Basit İfadeler

Tüm işleç işlemleri, hatta veri tiplerinin kendisi Javy dilinde basit ifade olarak anılır. Tüm basit ifadeler "**;**" ile bitmek durumundadırlar.

```
3;
1 + 5;
"merhaba"
1 in 1 .. 15;
```

gibi veri tiplerinin kendisi de birer basit ifadedir. Bunlara ilaveten değişken isimleri, paket ve sınıf isimleri de basit ifadelerdir.

```
ornekDegisken;
java;
Integer;
```

Tüm basit veya bileşik değişken atama işlemleri birer basit ifadedir.

```
sayi = 45;
dizge = "selam";
```

```
kisaTamsayi = 12S;
sonuc = a + 45;
islem += 42;
```

Sınıf içeri aktarma biçimi olan import gösterimleri de basit ifadelerdir.

```
import java.util.ArrayList as AL, java.lang.Integer,
java.lang.reflect.*;
import javax.swing.JFrame;
```

Nesne aitlik işleçleri ve hiyerarşik alt öge erişimlerinin hepsi, birer basit ifade olabilir.

```
meyveler["elma"];
"elma" in ("elma", "armut");
java.lang.System.out.println("merhaba uzay");
```

3.4.2. Bileşik İfadeler

Javy, mevcut sürümü itibari ile iki adet bileşik ifade kullanmaya imkan tanır. Koşullu bileşik ifade kullanım numunesi olarak Java'da da kullanılan if .. else bileşik ifadesi ve döngü kullanımı içinde, while bileşik ifadesi kullanılabilir.

if ... else Bileşik ifadesi :

if ... else ifadesi, Java, C#, JavaScript gibi benzer imlaya sahip programlama dillerinin hepsinde, aynı veya birbirlerine çok yakındır. Javy'nin if ... else bileşik ifadesi de diğerlerinde olduğu gibi, bir boolean karşılaştırma değerine ihtiyaç duyar.

```
if (<boolean karşılaştırma değeri>) {
  // karşılaştırma değerinin sonucu true ise
  // çalıştırılacak kısım
} else {
```

```
  // karşılaştırma değerinin sonucu false ise
  // çalıştırılacak kısım
}
```

Yukarıdaki gösterimde, **else** alt ifadesi seçimlidir. (opsiyonel) Programcı isteğe bağlı şekilde kullanabilir. Yine Java ve C'de olduğu gibi, if ifadesindeki mütakip satır veya ifade tek ise, blok parantez kullanımı da seçimlidir.

```
if (true)
  println("değer : true");
```

Bu ifadede de else seçimlidir. Arzu edilirse kullanılabilir.

```
if (1 == 2)
  println("1 eşit 2'dir");
else
  println("1 eşit 2 değildir");
```

Yukarıdaki kullanım bize, else kullanımını else if şeklinde kullanma seçeneği tanır.

```
if (sayi = 1) {
  println("sayı : 1");
} else if (sayi == 2) {
  println("sayı : 2");
}
```

else if kullanımı, aslında **else**'den sonra gelen bir **if** gösterimidir. Eğer sistemde blok ifade girilme zorunluluğu bulunsaydı. Java ve C'nin kullandığı yukarıdaki benzeri algoritmalar kullanılamayacaktı. İfade aslında şu şekildedir.

```
if (sayi = 1) {
  println("sayı : 1");
} else {
  if (sayi == 2) {
    println("sayı : 2");
```

```
  }
}
```

while Döngü İfadesi :

while döngüsü de, yine aynı dil ailesinden gelmiş bulunan dillerdeki kullanım ile aynıdır. Koşul ifadesi boolean true olduğu sürece, döngü devam edecektir.

```
while (<boolean karşılaştırma değeri>) {
  // karşılaştırma değerinin sonucu true ise
  // çalıştırılacak kısım.
}
```

Test sonucunun true olması durumunda, blok ifadedeki algoritma çalıştırılacaktır. Bu çalıştırma işlemi, test sonucu false olana kadar devam eder.

```
sayac = 0;
while (sayac < 5) {
  println(sayac);
  sayac += 1;
}
```

while döngüsünde de if .. else bileşik ifadesinde olduğu gibi, müteakip ifade tekil ise blok kullanılmayabilir.

```
while (false)
  println("bu dizge görünmeyecektir.");
```

Yukarıdaki ifadede test sonucu **false** değerine sahip olduğu için doğrudan döngüden çıkılacaktır.

3.4.3. Yorum İfadeleri

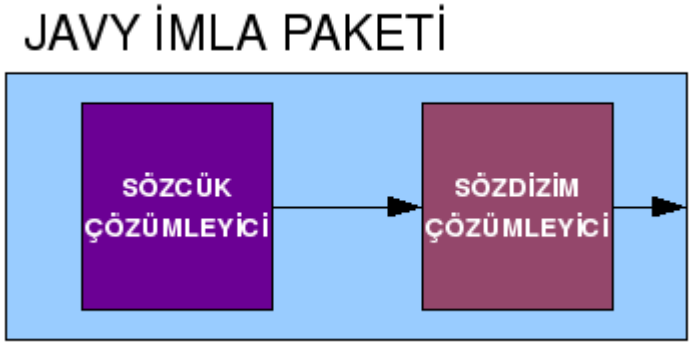
Yorum ifadeleri yorumlayıcı tarafından dikkate alınmayan ifade bloklarıdır. Yorum ifadeleri algoritma açıklama, güncel dosya hakkında bilgi verme gibi amaçlar için sıkça kullanılır. Javy, Java'nın ve C imlasına sahip dillerin kullandığı çoklu satır ve tekli satır yorum ifadelerini işleyebilir.

```
/* Çoklu Yorum Satırı Denemesi
   Satır 1
   Satır 2
 */

// Bu da tekli yorum satırı
/* Çoklu yorum satırını da tek satır için kullanabiliriz. */
```

Javy, bu yorum ifadelerine ilaveten *nix kabuğunun ve Perl, PHP, Python gibi dillerin kullandığı # ile başlayan yorum ifadelerine de olanak tanır. Bu sayede program başına kabuk tarafından anlamlı yorum satırı girildiği takdirde, doğrudan *nix kabuğu tarafından ilişkili yorumlayıcı bulunarak çalıştırılabilir.

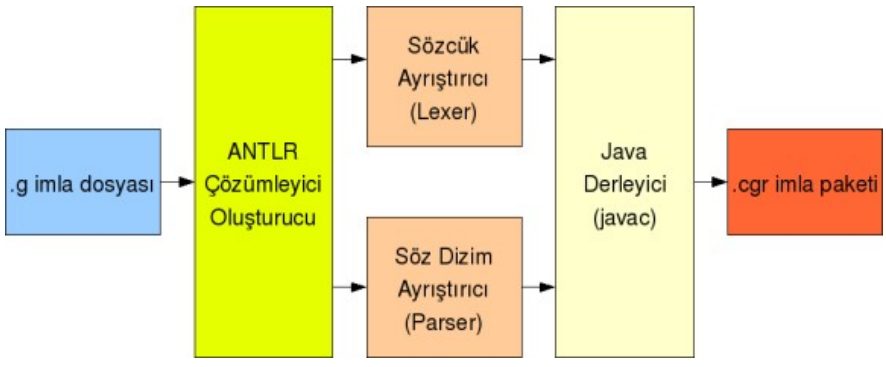
```
#!/usr/bin/cezve
# Program içerisinde yorum ifadelerini bu şekilde de
# oluşturabiliriz.
```

4. JAVY İMLASI ÇÖZÜMLEME AŞAMALARI



Şekil 17: Javy İmla Paketi

Javy Diline ait imla tanımlanması sırasında ANTLR'ın sahip olduğu EBNF imla tanımlama imlası kullanılmıştır. Girilen imla kuralları bir programlama dilinin söz dizim yapısını belirler. Bu bölümde, Sözcük Çözümleyici (Lexer) ve Söz Dizim Çözümleyici (Parser) oluşturmak için gerekli olan imla kurallarının tasarım aşamalarına değinilecektir. Bir başka değişle, bu kurallar ile Cezve dil altyapısına bir Çözümleyici modül yazılabilmektedir.

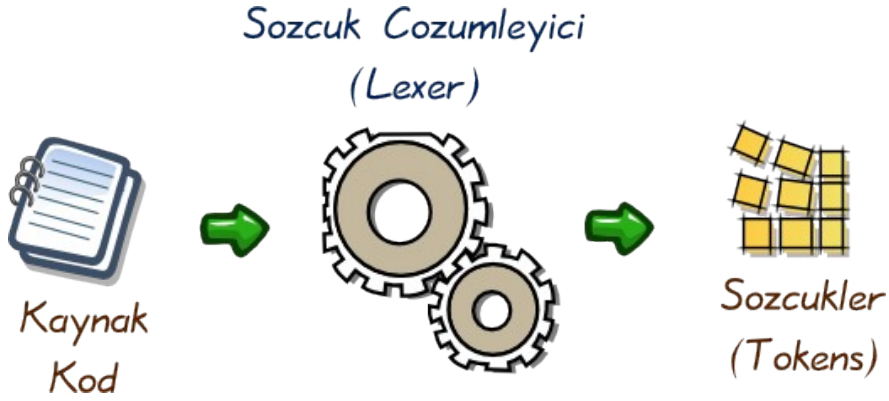


Şekil 18: Cgr Paketi Oluşum Aşamaları

İmla kurallar tanımlandıktan sonra, .g uzantılı tek bir dosya haline getirilir. Cezve dil altyapısı, standart Cezve sözcüklerine sahip olan bu imla dosyasını bütünleşik ANTLR Çözümleyici Oluşturucuya gönderecektir. Sonuç olarak, Sözcük Çözümleyici ve Söz Dizim Çözümleyici modüllerin kaynak kodları oluşur. Bu kaynak kodlar, Cezve Dil Altyapısı tarafından Java Derleyicisine gönderilir. Oluşan derlenmiş sınıf dosyaları, CGR (Cezve Grammar) uzantılı olarak bir pakete dönüştürülür ve kendi bünyesine alır.

Yeni modül etkinleştirildiği takdirde, Cezve dil altyapısı, çözümleme işlemlerinin ilk safhasını, cgr paketinin barındırdığı bu ayrıştırıcılara bırakacaktır.

4.1. Sözcük Çözümleme İmlası



Şekil 19: Sözcük Çözümleme

Sözcük çözümleme işleminde, imla kuralları ile tanımlanıp eşleşen karakter ve karakter grupları belirlenir. Bu kurallar, çözümleme hiyerarşisinin ilk basamağıdır. Hangi karakterlerin yok sayılacağı veya hangi karakter gruplarının nasıl

isimlendirileceğini sözcük çözümleyici belirler. Sözcük Çözümleme imlası ile ANTLR Çözümleyici oluşturucu sisteminin Cezve dil altyapısı için bir çözümleyici üretmesi sağlanacaktır. Girilen ANTLR tarzı BNF kuralları, bu çözümleyici davranışını tanımlar.

4.1.1. Çözümleyici Kuralları

Anahtar Sözcükler :

Javy temel imlası, şu anda desteklenen miktarda basit ve bileşik ifadeye ilişkin anahtar sözcük barındırır. Bu sözcükler, söz dizim çözümleyicide anlamlı ifadeler halini alacaktır.

```
IMPORT                          :   'import'    ;
AS                              :   'as'        ;
IF                              :   'if'        ;
ELSE                            :   'else'      ;
WHILE                           :   'while'     ;
```

import ve as ifadesi, Javy İmlası Bölümünde de açıklandığı gibi, paket ve sınıf aktarımı için kullanılacak olan sözcüklerdir. Diğer sözcükler ise, if .. else, while gibi bileşik ifadelerde kullanılır.

İşleç Sözcükleri :

Javy imlasında operatör işlemleri için de birçok sözcük tanımlanması gereklidir. Javy imlasında tanımlandığı gibi; atama, toplama, çıkarma, çarpma, bölme, dizge birleştirme gibi daha birçok işlemin temel yapı taşı, bu sözcüklerdir.

```
PLUS                                : '+'          ;
MINUS                               : '-'          ;
UNDERSCORE                          : '_'          ;
ASTERISK                            : '*'          ;
SLASH                               : '/'          ;
BACKSLASH                           : '\\'         ;
PERCENT                             : '%'          ;
DOUBLE_ASTERISK                     : '**'         ;
DOUBLE_PLUS                         : '++'         ;
DOUBLE_MINUS                        : '--'         ;
EXCLAMATION_MARK                    : '!'          ;
AMPERSAND                           : '&'          ;
CARET                               : '^'          ;
XOR                                 : 'xor'        ;
PIPE                                : '|'          ;
DOUBLE_AMPERSAND                    : '&&'         ;
AND                                 : 'and'        ;
DOUBLE_PIPE                         : '||'         ;
OR                                  : 'or'         ;
QUESTION_MARK                       : '?'          ;
COLON                               : ':'          ;
DOUBLE_LEFT_ANGLE_BRACKET           : '<<'         ;
DOUBLE_RIGHT_ANGLE_BRACKET          : '>>'         ;
TRIPLE_RIGHT_ANGLE_BRACKET          : '>>>'        ;
TILDE                               : '~'          ;
LEFT_ANGLE_BRACKET                  : '<'          ;
RIGHT_ANGLE_BRACKET                 : '>'          ;
LEFT_ANGLE_BRACKET_AND_EQUALS       : '<='         ;
RIGHT_ANGLE_BRACKET_AND_EQUALS      : '>='         ;
INSTANCEOF                          : 'instanceof' ;
TRIPLE_EQUALS                       : '==='        ;
IS                                  : 'is'         ;
ISNOT                               : 'isnot'      ;
EXCLAMATION_MARK_AND_EQUALS         : '!='         ;
DOUBLE_EQUALS                       : '=='         ;
EQUALS                              : '='          ;
PLUS_AND_EQUALS                     : '+='         ;
MINUS_AND_EQUALS                    : '-='         ;
ASTERISK_AND_EQUALS                 : '*='         ;
SLASH_AND_EQUALS                    : '/='         ;
BACKSLASH_AND_EQUALS                : '\\='        ;
DOUBLE_ASTERISK_AND_EQUALS          : '**='        ;
PERCENT_AND_EQUALS                  : '%='         ;
```

```
UNDERSCORE_AND_EQUALS                  :  '_='         ;
AMPERSAND_AND_EQUALS                   :  '&='         ;
PIPE_AND_EQUALS                        :  '|='         ;
CARET_AND_EQUALS                       :  '^='         ;
DOUBLE_LEFT_ANGLE_BRACKET_AND_EQUALS   :  '<<='        ;
DOUBLE_RIGHT_ANGLE_BRACKET_AND_EQUALS  :  '>>='        ;
TRIPLE_RIGHT_ANGLE_BRACKET_AND_EQUALS  :  '>>>='       ;
DOUBLE_DOT                             :  '..'         ;
IN                                     :  'in'         ;
NOTIN                                  :  'notin'      ;
COMMA                                  :  ','          ;
DOT                                    :  '.'          ;
LEFT_CURLY_BRACKET                     :  '{'          ;
LEFT_PARENTHESIS                       :  '('          ;
LEFT_SQUARE_BRACKET                    :  '['          ;
RIGHT_CURLY_BRACKET                    :  '}'          ;
RIGHT_PARENTHESIS                      :  ')'          ;
RIGHT_SQUARE_BRACKET                   :  ']'          ;
SEMICOLON                              :  ';'          ;
```

Yoksayılan Sözcükler :

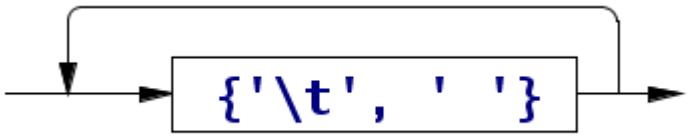
Yoksayılan Sözcükler, bir bilgisayar diline ait sözcük çözümleme sisteminin temel tanımlamalarındandır. Örneğin boşluk karakterleri yoksayılan sözcük kategorisine alınmazsa, tüm ifadelerin hiç bir boşluk bırakılmadan ardarda yazılması gerekir ki, bu da okunulabilirliği azaltır.

```
WHITESPACE   : (' ' | '\t')+     {$channel = HIDDEN;} ;
COMMENT      : SLASH_AND_ASTERISK (.)* ASTERISK_AND_SLASH
{$channel = HIDDEN;};
LINE_COMMENT : (DOUBLE_SLASH | SHARP) ~('\u000D' | '\u000A')*
{$channel = HIDDEN;};
EOL          :   ('\u000D' | '\u000A'){$channel = HIDDEN;};
```

Sözcük Çözümleyici tanımlamasında, yapılan programlama dilinde dikkate alınmayacak olan ögeleri belirtmemiz gerekmektedir.

```
WHITESPACE: (' ' | '\t')+  {$channel = HIDDEN;};
```

WHITESPACE sözcük çözümleyici kuralı, ' ' (boşluk) **VEYA** '\t' sekme (tab) işaretlerini **Beyaz Boşluk** olarak nitelenirecek ve gizli kanal üzerinden yürütülmesini sağlayacaktır. Gizli kanal üzerinden yürütülmesi ise sözcüklerin yoksayılması manasına gelecektir. Yoksayılan Sözcükler, bir sonraki aşama olan Sözdizim ayrıştırma aşamasına geçmezler.



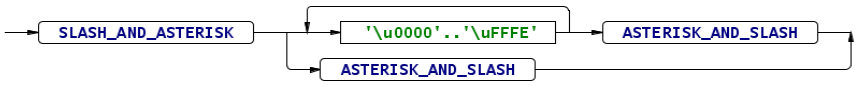
Şekil 20: WHITESPACE Sözcük Çözümleyici Kuralı

WHITESPACE kuralı, grafikten de anlaşılacağı üzere ' ' veya '\t' önermesinin bir ve birden fazla tekrarı durumu ile eşleşmektedir. Benzer eşleşmeler şu şekilde olabilir.

```
  * <boşluk>
  * <sekme>
  * <boşluk> <sekme> <sekme>
  * <sekme>  <boşluk> <sekme>
  * <boşluk> <boşluk> <boşluk>
  * ....

COMMENT: SLASH_AND_ASTERISK (.)* ASTERISK_AND_SLASH  {$channel
= HIDDEN;};
```

COMMENT kuralı, çoklu yorum satırlarını ifade etmektedir. Burada Javy'nin varsayılan olarak kullandığı C, C++, Java, C#, PHP, Perl, JavaScript, J#,

ActionScript gibi dillerin standart olarak kullandığı ”/*” ile başlayıp “*/” ile biten çoklu yorum satırlarıdır.

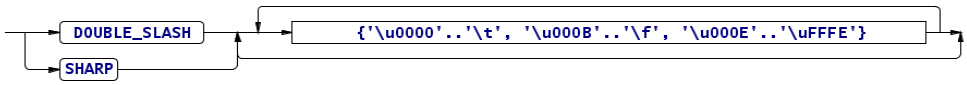


Şekil 21: COMMENT Sözcük Çözümleyici Kuralı

Tanımlama ”/*” ve “*/” sözcüklerinin arasına ya hiç ya da herhangi bir veri gelmesi durumunda işlem görür.

```
LINE_COMMENT: (DOUBLE_SLASH | SHARP) ~('\u000D' | '\u000A')*
{$channel = HIDDEN;};
```

LINE_COMMENT ise tek satırlık yorum ifadeleridir. C, C++, Java, C#, PHP, Perl, JavaScript, J#, ActionScript gibi dillerin çoğu standart olarak // ile başlayan tek satırlık yorum ifadesini kullanır. Javy, imlasında buna ek olarak *nix sistem kabuklarınca çokca ihtiyaç duyulan ve PHP, Perl, Python, Bash gibi dillerin de kullandığı **”#“** karakterini kullanmıştır. Bu sayede, dil ile yazılan betik program, *nix kabuğundan doğrudan çalıştırılabilecektir.



Şekil 22: LINE_COMMENT Sözcük Çözümleyici Kuralı

LINE_COMMENT ifadesi, ”//” ya da ”#” ile başlayıp, satır sonu harici veriler ile devam eden şekilde tanımlanmaktadır.

Yorum satırlarının örnek kullanımı şu şekildedir;

```
#!/usr/bin/javy

// Yukarıdaki #! ifadesi sayesinde Bash kabuğuna
// yorumlayıcı bildirilmiş olur. Böylece betik doğrudan
// çalıştırılabilir hale gelir.

// Bu, tek satırlık yorum ifadesidir. "//" ile
// başlayan ifadeler tek satırlık yorum ifadesidir.

# Bu da tek satırlık yorum ifadesidir.

/*
  Bu ise çok
  satırdan oluşan bir yorum ifadesidir.
*/
```

Bunlara ek olarak varsayılan Javy imlasında **Satır Sonları** da işlenmeyen sözcükler arasına katılmıştır.

```
EOL  : ('\u000D' | '\u000A') {$channel = HIDDEN;};
```

Şekil 23: EOL Sözcük Çözümleyici Kuralı

EOL (End Of Line) karakterleri onaltılık 0D veya 0A olabilir. 0xOA onaltılık verisi, LF (Line Feed) yani "Satır Besleme" olarak adlandırılır. 0xOD onaltılık verisi ise CR (Carriage Return) yani "İmleç Dönümü" olarak ifade edilebilir.

Windows metin dosyalarının satır sonlarında <CR> ve <LF> karakterlerinin ikisi de bulunur.

```
Ornek bir metin dosyasi.     <CR><LF>
                             <CR><LF>
```

Onaltılık görünümü ise şu şekildedir.

```
00000000  4f 72 6e 65 6b 20 62 69  72 20 6d 65 74 69 6e 20  |
Ornek bir metin |
00000010  64 6f 73 79 61 73 69 2e  0d 0a 0d 0a              |
dosyasi.....|
```

Son bölümdeki **0d 0a 0d 0a** kısmı, **CR LF CR LF** verisini gösterir.

Unix, Linux, BSD, Solaris gibi sistemlerde ise, satır sonları yalnızca <LF> karakteridir.

```
00000000  4f 72 6e 65 6b 20 62 69  72 20 6d 65 74 69 6e 20  |
Ornek bir metin |
00000010  64 6f 73 79 61 73 69 2e  0a 0a                    |
dosyasi...|
```

Son bölümdeki **0a 0a** kısmı, **LF LF** verisini gösterir.

Apple Macintosh ve uyumlu sistemlerde ise, satır sonları yalnızca <CR> karakteridir.

```
00000000  4f 72 6e 65 6b 20 62 69  72 20 6d 65 74 69 6e 20  |
Ornek bir metin |
00000010  64 6f 73 79 61 73 69 2e  0d 0d                    |
dosyasi...|
```

Son bölümdeki **0d 0d** kısmı, **CR CR** verisini gösterir.

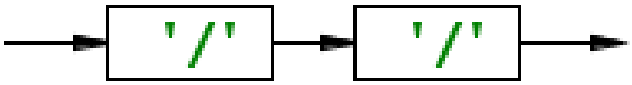
Parça Sözcükler :

Temel tip kuralları oluşturulurken, diğer birçok kuraldan veya Parça Sözcüklerden faydalanılır. **Parça Sözcükler** (Fragment Tokens), sözcük

çözümleyicinin tanımlanmasında doğrudan kullanılmaz. Kuralları tanımlamak için yardımcı alt kurallardır. Ve bunlara ait sözcük numaraları yoktur.

Javy'de kullanılan **Parça Sözcükler** şöyledir:

```
fragment DOUBLE_SLASH                : '//'          ;
fragment SHARP                       : '#'           ;
fragment SLASH_AND_ASTERISK          : '/*'          ;
fragment ASTERISK_AND_SLASH          : '*/'          ;
fragment ESCAPE_SEQUENCE             : '\\' .        ;
fragment TRUE                        : 'true'        ;
fragment FALSE                       : 'false'       ;
fragment DIGIT                       : '0' .. '9'    ;
fragment APOSTROPHE                  : '\''          ;
fragment TRIPLE_APOSTROPHE           : '\'\'\''      ;
fragment QUOTATION_MARK              : '"'           ;
fragment TRIPLE_QUOTATION_MARK       : '"""'         ;

fragment LETTER                      :
                                       '\u0024'
                                     | '\u005F'
                                     | '\u0041' .. '\u005A'
                                     | '\u0061' .. '\u007A'
                                     | '\u00C0' .. '\u00D6'
                                     | '\u00D8' .. '\u00F6'
                                     | '\u00F8' .. '\u00FF'
                                     | '\u0100' .. '\u1FFF'
                                     | '\u3040' .. '\u318F'
                                     | '\u3300' .. '\u337F'
                                     | '\u3400' .. '\u3D2D'
                                     | '\u4E00' .. '\u9FFF'
                                     | '\uF900' .. '\uFAFF'
                                     ;
```



Şekil 24: DOUBLE_SLASH Sözcük Çözümleyici Parça Kuralı

DOUBLE_SLASH Parça Sözcüğü, C tarzı tek satırlık yorum ifadelerini belirtmek için kullanılır.

```
fragment DOUBLE_SLASH  :  '//'
```

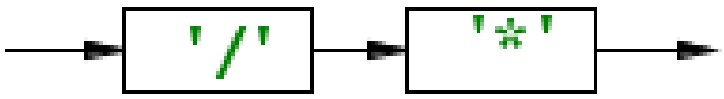
Peşpeşe iki adet bölü karakteri DOUBLE_SLASH olarak algılanacaktır. Fakat çözümleyici sözcüğe bu ismi atamaz. Çünkü oluşturulan bir parça sözcüktür. Daha büyük bir Sözcük Çözümleyici kuralının alt parçasıdır.

```
LINE_COMMENT : (DOUBLE_SLASH | SHARP) ~('\u000D' | '\u000A')*
{$channel = HIDDEN;} ;
```

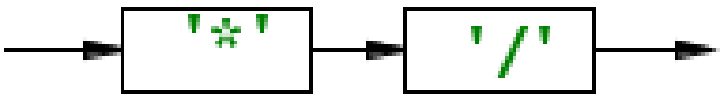
Yukarıdaki tek satırlık yorum ifadesini belirten kuralda da görüldüğü gibi, parça sözcüğümüz bu kısımda kullanılmıştır. Aynı kural içerisinde, SHARP yani (#) parça sözcüğü de kullanım bulur.

```
fragment SHARP  :  '#'          ;
```

SLASH_AND_ASTERISK ve **ASTERISK_AND_SLASH** diye isimlendirdiğimiz sözcükler de çoklu yorum ifadesinin alt parçalarıdır.



Şekil 25: SLASH_AND_ASTERISK Sözcük Çözümleyici Parça Kuralı

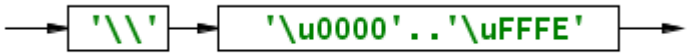


Şekil 26: ASTERISK_AND_SLASH Sözcük Çözümleyici Parça Kuralı

```
fragment SLASH_AND_ASTERISK           :   '/*';
fragment ASTERISK_AND_SLASH           :   '*/';
```

/* sözcüğü, çoklu satır yorum ifadesinin başladığını belirtirken, */ sözcüğü de bitişini belirtmektedir. Parça kuralların kullanıldığı çoklu yorum ifadesini belirten tüm kural şu şekildedir.

```
COMMENT: SLASH_AND_ASTERISK (.)* ASTERISK_AND_SLASH  {$channel
= HIDDEN;};
```



Şekil 27: ESCAPE_SEQUENCE Sözcük Çözümleyici Parça Kuralı

ESCAPE_SEQUENCE, programlama yaparken kullanıcılara "\n", "\t" gibi karakter karşılığının girilmesi mümkün olmayan veya ifadeyi farklı bir kurala çevirebilecek karakterleri tanımlamaya imkan tanır. Kaçış Dizisi (Escape Sequence) String veri tipinde kullanılan bir parça sözcüktür:

```
fragment ESCAPE_SEQUENCE :   '\\' . ;
```

Yukarıdaki EBNF tanımlamasına göre, '\' karakterinden sonra herhangi bir karakter gelir ise, bu **"Kaçış Dizisi"** olarak nitelendirilir. Yani karakter katarı, normal formundan kurtulup, başka bir tanımlamayı çağırmış olur. Herhangi bir karakter ifadesini EBNF dilindeki '.' karakteri tanımlamaktadır. '\\' karakteri ise aslında "Kaçış Düzeni" teriminin kendi kendini tanımlayan bir yaklaşımıdır. İfade,

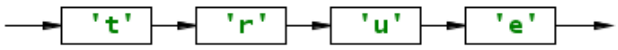
bir Kaçış Düzenidir. (Escape Sequence). Yani '\\' ifadesi ile yalnız '\' karakteri tanımlanmak istenmektedir.

Kaçış düzenine diğer örnekler:

```
\\      ters bölü (backslash)
\b      geri silme (backspace)
\n      yeni satır (new line)
\t      sekme (yatay sekme)
```

Boolean cebiri tanımlamalarında kullanılan doğru yanlış ifadelerini de **Parçacık Sözcük** olarak tanımlamamız doğru olabilir.

```
fragment TRUE  : 'true'  ;
fragment FALSE : 'false' ;
```



Şekil 28: TRUE Sözcük Çözümleyici Parça Kuralı

Şekil 29: FALSE Sözcük Çözümleyici Parça Kuralı

Genel rakamsal tanımlamalarını da parçacık olarak tanımlayıp, tip ifadelerini tanımlamada daha sonra kullanabiliriz.

```
fragment DIGIT   : '0' .. '9' ;
```

DIGIT (RAKAM), alt kuralı 0 ile 9 arasındaki tüm rakamları tanımlamaktadır. Bu kuralın bir diğer yazılışı ise şu şekilde olabilir:

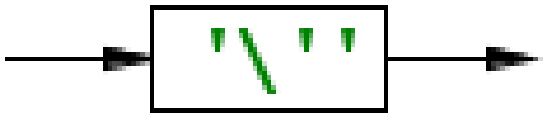
```
fragment DIGIT   : '0' | '1' | '2' | '3' | '4' | '5' | '6' |
'7' | '8' | '9' ;
```

Şekil 30: DIGIT Sözcük Çözümleyici Parça Kuralı

iki ifade de aynıdır. (0 veya 1 veya 2 veya 3 veya 4 veya 5 veya 6 veya 7 veya 8 veya 9) manasına gelir.

Çoklu ve tekli karakter katarı (string) tanımlamaları için ise Tırnak veya Üçlü Tırnak kullanılmaktadır. Bunları da parça kural olarak tanımlamamız, yapacağımız kuralların okunulabilirliğini arttıracaktır.

```
fragment APOSTROPHE  :  '\''   ;
```

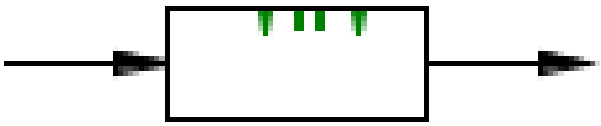


Şekil 31: APOSTROPHE Sözcük Çözümleyici Parça Kuralı

```
fragment TRIPLE_APOSTROPHE : APOSTROPHE APOSTROPHE APOSTROPHE;
```

Şekil 32: TRIPLE_APOSTROPHE Sözcük Çözümleyici Parça Kuralı

```
fragment QUOTATION_MARK   :   '"'     ;
```



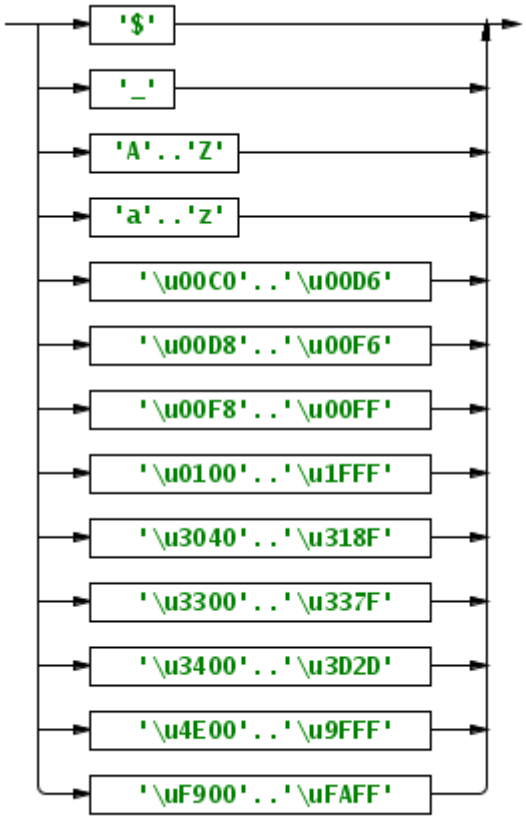
Şekil 33: QUOTATION_MARK Sözcük Çözümleyici Parça Kuralı

```
fragment TRIPLE_QUOTATION_MARK  : '"""'  ;
```

Şekil 34: TRIPLE_QUOTATION_MARK Sözcük Çözümleyici Parça Kuralı

Parça ifadeleri, kendileri içerisinde birbirlerini tanımlamada da kullanabilmekteyiz. Yukarıdaki ifade, tip ifadelerinde daha ayrıntılı olarak ele alınacaktır. Üçlü tırnak yaklaşımı, Python'a özel çok satırlı karakter katarları tanımlamamızı sağlayan bir yaklaşımdır.

```
fragment LETTER  :
                     '\u0024'
                 |   '\u005F'
                 |   '\u0041' .. '\u005A'
                 |   '\u0061' .. '\u007A'
                 |   '\u00C0' .. '\u00D6'
                 |   '\u00D8' .. '\u00F6'
                 |   '\u00F8' .. '\u00FF'
                 |   '\u0100' .. '\u1FFF'
                 |   '\u3040' .. '\u318F'
                 |   '\u3300' .. '\u337F'
                 |   '\u3400' .. '\u3D2D'
                 |   '\u4E00' .. '\u9FFF'
                 |   '\uF900' .. '\uFAFF'
                 ;
```



Şekil 35: LETTER Sözcük Çözümleyici Parça Kuralı

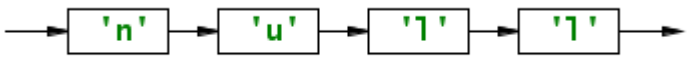
LETTER (HARF) Parça Kuralı ise Unicode aralığı dahil, tüm yazılabilecek harfleri içerir. Daha sonra String veri tipinde ve name, identifier gibi çeşitli belirteç tanımlarında kullanım bulacaktır.

Tip Tanımları :

Tip tanımları, anlamsal çözümlemede nesnelere dönüştürülecek olan sözcüklerdir. Bu kuralların bir kısmı çözümleme kolaylığı açısından söz dizim çözümleyiciye taşınmıştır. Fakat geneli sözcük çözümleyicide bulunmaktadır.

```
CEZVE_LITERAL_NULL      :  'null'                          ;
```

```
CEZVE_LITERAL_BOOLEAN   :  TRUE | FALSE                      ;
CEZVE_LITERAL_BYTE      :  CEZVE_LITERAL_NUMBER ('B' | 'b')  ;
CEZVE_LITERAL_SHORT     :  CEZVE_LITERAL_NUMBER ('S' | 's')  ;
CEZVE_LITERAL_LONG      :  CEZVE_LITERAL_NUMBER ('L' | 'l')  ;
CEZVE_LITERAL_NUMBER    :  DIGIT+                            ;
CEZVE_LITERAL_CHARACTER :
  APOSTROPHE
  (ESCAPE_SEQUENCE | ~('\\'|'\u000D'|'\u000A'|'\''))
  APOSTROPHE
  ;
CEZVE_LITERAL_STRING    :
  (
       TRIPLE_APOSTROPHE
       (options {greedy=false;}:.)* TRIPLE_APOSTROPHE
       {
         String st = getText();
         setText(st.substring(2, st.length() - 2));
       }
    |  TRIPLE_QUOTATION_MARK
       (options {greedy=false;}:.)* TRIPLE_QUOTATION_MARK
       {
         String st = getText();
         setText(st.substring(2, st.length() - 2));
       }
    |  QUOTATION_MARK
       (ESCAPE_SEQUENCE | ~('\\'|'\u000D'|'\u000A'|'"' ))*
       QUOTATION_MARK
    |  APOSTROPHE
       (ESCAPE_SEQUENCE | ~('\\'|'\u000D'|'\u000A'|'\''))+
       APOSTROPHE
  )
  ;
```



Şekil 36: CEZVE_LITERAL_NULL Sözcük Çözümleyici Kuralı

null ifadesi, Java ve Javy'de herhangi bir nesne referansına karşılık gelmeyen değerler için kullanılır.

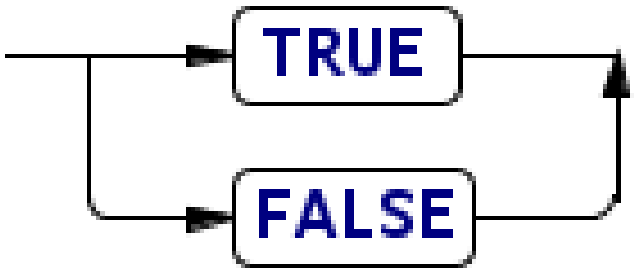
```
CEZVE_LITERAL_NULL  :  'null'     ;
```

Peş peşe gelen 4 adet karakterin null olması durumunda, sözcük çözümleyici tarafından CEZVE_LITERAL_NULL adı ile nitelendirilecektir.

```
CEZVE_LITERAL_BOOLEAN  :  TRUE | FALSE ;
```

Boolean aritmetiğinde, 1 veya 0 yaklaşımını sağlamak için Boolean adında anlamsal bir sözcük tanımlamamız gereklidir. Birçok programlama dilinde, “Doğru” ve “Yanlış” ifadeleri benzer şekilde tanımlanmıştır.

Boolean kuralı, parça kurallar kullanılarak tanımlanmaktadır. Kısaca kuralı özetlemek gerekirse, CEZVE_LITERAL_BOOLEAN diye nitelendirilen sözcük ya TRUE parça kuralında tanımlanılan gibi olmalıdır ya da FALSE parça kuralında tanımlanılan gibi olmalıdır. TRUE parça kuralı 'true', FALSE parça kuralı ise 'false' karakter katarı ifadelerini içermektedir.

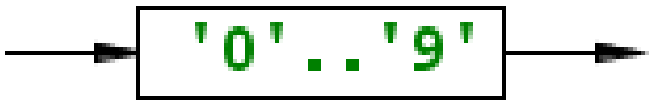


Şekil 37: CEZVE_LITERAL_BOOLEAN Sözcük Çözümleyici Kuralı

Temelde bu yaklaşım bir DFA yani Belirgin Sonlu Otomattır. TRUE parça kuralı 't' harfini işler, sonraki durum ise 'r' olmalıdır. Daha sonra, sırası ile 'u' ve 'e' işlenmelidir.

Boolean sözcüğü daha sonra anlamsal hale getirilip, java.lang.Boolean sınıfının bir örneği olacaktır.

```
CEZVE_LITERAL_NUMBER  :  DIGIT+ ;
```



Şekil 38: CEZVE_LITERAL_NUMBER Sözcük Çözümleyici Kuralı

CEZVE_LITERAL_NUMBER (SAYI) kuralı, birden fazla rakamın (DIGIT) bir araya gelmesi ile oluşmaktadır. Örneğin; `1312` bir sayıdır. `03123` bir sayıdır. `2` ise hem bir rakam hem de bir sayıdır. Rakam kuralı, bir parçacık sözcük olduğundan çözümlemeye dahil edilmez. Bu nedenle `2` gibi bir rakam da DIGIT kuralı ile doğrudan örtüşmesine rağmen, yalnızca CEZVE_LITRAL_NUMBER sözcüğü ile eşleşecektir.

```
CEZVE_LITERAL_BYTE :  CEZVE_LITERAL_NUMBER ('B' | 'b')  ;
```

Şekil 39: CEZVE_LITERAL_BYTE Sözcük Çözümleyici Kuralı

BYTE veri tipi sözcüğü, Parça Sözcük olan CEZVE_LITERAL_NUMBER kuralının sonunda 'b' veya 'B' son eki eklenerek ifade edilir. Java'nın temel referans

tiplerinden biri olan java.lang.Byte sınıfı ile ilişkili olan Byte sözcüğü, sözdizim çözümleyicide anlamsal hale getirilir.

```
CEZVE_LITERAL_SHORT  :  CEZVE_LITERAL_NUMBER ('S' | 's')  ;
```

Şekil 40: CEZVE_LITERAL_SHORT Sözcük Çözümleyici Kuralı

SHORT veri tipi sözcüğü, Parça Sözcük olan CEZVE_LITERAL_NUMBER kuralının sonunda, 's' veya 'S' son eki eklenerek ifade edilir. Java'nın temel referans tiplerinden biri olan java.lang.Short sınıfı ile ilişkili olan SHORT sözcüğü, sözdizim çözümleyicide anlamsal hale getirilir.

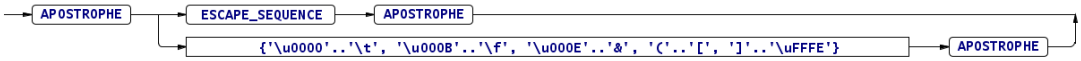
```
CEZVE_LITERAL_LONG  :  CEZVE_LITERAL_NUMBER ('L' | 'l')  ;
```

Şekil 41: CEZVE_LITERAL_LONG Sözcük Çözümleyici Kuralı

LONG veri tipi sözcüğü, Parça Sözcük olan CEZVE_LITERAL_NUMBER kuralının sonunda 'l' veya 'L' son eki eklenerek ifade edilir. Java'nın temel referans tiplerinden biri olan java.lang.Long sınıfı ile ilişkili olan LONG sözcüğü, sözdizim çözümleyicide anlamsal hale getirilir.

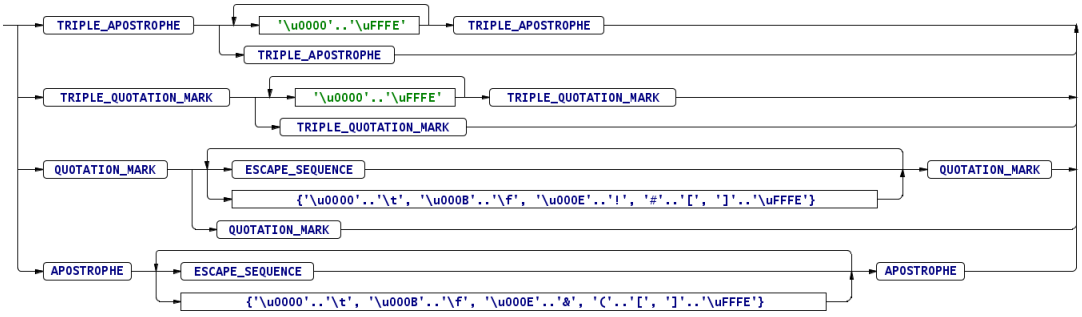
```
CEZVE_LITERAL_CHARACTER :
  APOSTROPHE
  (ESCAPE_SEQUENCE | ~('\\'|'\u000D'|'\u000A'|'\''))
  APOSTROPHE
```

```
    ;
```



Şekil 42: CEZVE_LITERAL_CHARACTER Sözcük Çözümleyici Kuralı

```
CEZVE_LITERAL_STRING    :
  (
       TRIPLE_APOSTROPHE
       (options {greedy=false;}:.)* TRIPLE_APOSTROPHE
       {
         String st = getText();
         setText(st.substring(2, st.length() - 2));
       }
     |  TRIPLE_QUOTATION_MARK
       (options {greedy=false;}:.)* TRIPLE_QUOTATION_MARK
       {
         String st = getText();
         setText(st.substring(2, st.length() - 2));
       }
     |  QUOTATION_MARK
       (ESCAPE_SEQUENCE | ~('\\'|'\u000D'|'\u000A'|'"' ))*
       QUOTATION_MARK
     |  APOSTROPHE
       (ESCAPE_SEQUENCE | ~('\\'|'\u000D'|'\u000A'|'\''))+
       APOSTROPHE
  )
  ;
```



Şekil 43: CEZVE_LITERAL_STRING Sözcük Çözümleyici Kuralı

Javy'de Karakter Katarları tek satırlık karakter katarları ve çok satır ile ifade edilebilen karakter katarları şeklinde çeşitlenmiştir. Bu yaklaşım, Python ve C tarzı imlaya sahip dillerin bir karışımı şeklindedir.

Karakter katarları, iki tane üçerli tek tırnak veya çift tırnak arasına alınabilir. Bu, çok satırlı karakter katarı ifadesini oluşturur. Tek tırnak veya çift tırnak arasına alınan veriler ise, tek satırlık karakter katarı gösterimleridir. Bunlara ilaveten, çözümleyici kuralına ek Java kodları eklenmiştir. Bu, 3 tırnak ile oluşturulan çoklu dizgelerin fazladan tırnaklarının bir üst çözümleyiciye aktarılmamasına yöneliktir.

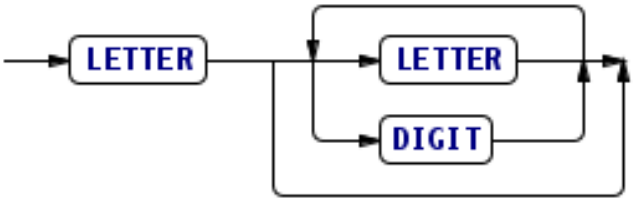
String sözcüğü söz dizim anlamlaştırılmasında, java.lang.String ile ilişkilendirilir.

Belirteç Tanımlamaları :

```
CEZVE_ATOM_NAME  :  LETTER+  ;
```

Şekil 44: CEZVE_ATOM_NAME Sözcük Çözümleyici Kuralı

```
CEZVE_ATOM_IDENTIFIER  :  LETTER (LETTER | DIGIT)* ;
```



Şekil 45: CEZVE_ATOM_IDENTIFIER Çözümleyici Kuralı

Belirteçler, iki farklı şekilde tanımlanmıştır. Birisi İSİM (NAME), diğeri ise BELİRTEÇ (IDENTIFIER) şeklindedir. CEZVE_ATOM_NAME sözcüğü, genelde Java paket tanımlamaları için kullanılırken, IDENTIFIER sözcüğü ise değişken adları için kullanılır. NAME sözcükleri yalnızca harf değerlerden oluşmalıdır. IDENTIFIER sözcüğü ise alfasayısal olabilir.

Belirteç tanımlaması, belirsiz (nondeterministic) şekilde tanımlanmıştır. Sözcük çözümleyici oluşturucu, bu belirsizliği yukarıdan aşağı kuralları işleterek giderir. Ayrıştırma sırasında ayrıştırılan kelimenin önce NAME kuralına uyup uyulmadığı araştırılır. Eğer veri kurala uymuyor ise IDENTIFIER kuralı yürütülür.

Cezve Anlamsal Sözcükleri :

Çözümleyici Oluşturucular, Söz dizim analizinde veya Ağaç Çözümlemede kullanılmak üzere, Kavramsal Sözcükler kabul edebilir. Bunlar, Parça Sözcüklerden farklı olarak gerçekten bir sözcük numarasına sahiptirler. Fakat çoğu karşılık olarak bir karakter katarı ile eşleşmezler. Bir sonraki bölümde anlatılacak olan Sözdizim Çözümleme esnasında kurallar ile eşleştirilebilirler. Bu sözcükler ağaç sisteminin anlaşılabilirliğini arttıracaktır. Çoklu imla sisteminin temel iletişim kuralını bu anlamsal sözcükler oluşturur.

Cezve çoklu imla sisteminin yapı taşı bu kavramsal tanımlamalardır. Tümü kendi arasında bölümlendirilmiştir. Ayrıca kavramsal olmayan diğer sözcükler (token) ile karışmaması için *CEZVE_* ön eki kullanılmıştır. Kullanılan bu sözcüklerin gerçekten bir karakter karşılığı varken bazıları da tamamen kavramsaldır.

Örneğin; CEZVE_LITERAL_NULL bir karakter katarına karşılık gelmelidir. Fakat CEZVE_LITERAL_INTEGER bir karakter katarına karşılık gelmeyebilir. Cezve sisteminde kullanılan anlamsal sözcükler şu şekildedir:

```
CEZVE_SCRIPT
CEZVE_STATEMENT_ASSG
CEZVE_STATEMENT_ASSG_ADDITION
CEZVE_STATEMENT_ASSG_AND
CEZVE_STATEMENT_ASSG_BITWISE_SL
CEZVE_STATEMENT_ASSG_BITWISE_SRS
CEZVE_STATEMENT_ASSG_BITWISE_SRU
CEZVE_STATEMENT_ASSG_CONCATENATION
CEZVE_STATEMENT_ASSG_DIVISION
CEZVE_STATEMENT_ASSG_MODULO
CEZVE_STATEMENT_ASSG_MULTIPLICATION
CEZVE_STATEMENT_ASSG_OR
CEZVE_STATEMENT_ASSG_POWER
CEZVE_STATEMENT_ASSG_SUBTRACTION
CEZVE_STATEMENT_ASSG_T_DIVISION
CEZVE_STATEMENT_ASSG_XOR
CEZVE_STATEMENT_EXPRESSION
CEZVE_STATEMENT_IF
CEZVE_STATEMENT_IF_END
CEZVE_STATEMENT_IMPORT
CEZVE_STATEMENT_WHILE
CEZVE_STATEMENT_WHILE_END
CEZVE_SUB_STATEMENT_ELSE
CEZVE_SUB_STATEMENT_ELSE_END
CEZVE_SUB_STATEMENT_IF
CEZVE_SUB_STATEMENT_IF_END
CEZVE_OPERATION_ADDITION
CEZVE_OPERATION_AND
CEZVE_OPERATION_BITWISE_AND
CEZVE_OPERATION_BITWISE_COMPLEMENT
CEZVE_OPERATION_BITWISE_OR
```

```
CEZVE_OPERATION_BITWISE_SL
CEZVE_OPERATION_BITWISE_SRS
CEZVE_OPERATION_BITWISE_SRU
CEZVE_OPERATION_BITWISE_XOR
CEZVE_OPERATION_COMPARISION_EQU
CEZVE_OPERATION_COMPARISION_GT
CEZVE_OPERATION_COMPARISION_GTE
CEZVE_OPERATION_COMPARISION_IDE
CEZVE_OPERATION_COMPARISION_INSTANCEOF
CEZVE_OPERATION_COMPARISION_LT
CEZVE_OPERATION_COMPARISION_LTE
CEZVE_OPERATION_COMPARISION_NIDE
CEZVE_OPERATION_COMPARISION_NEQ
CEZVE_OPERATION_CONCATENATION
CEZVE_OPERATION_DIVISION
CEZVE_OPERATION_IN
CEZVE_OPERATION_IN_PARENS
CEZVE_OPERATION_MODULO
CEZVE_OPERATION_MULTIPLICATION
CEZVE_OPERATION_NEGATIVE
CEZVE_OPERATION_NOT
CEZVE_OPERATION_NOTIN
CEZVE_OPERATION_OR
CEZVE_OPERATION_POSITIVE
CEZVE_OPERATION_POST_DECREMENT
CEZVE_OPERATION_POST_INCREMENT
CEZVE_OPERATION_POWER
CEZVE_OPERATION_PRE_DECREMENT
CEZVE_OPERATION_PRE_INCREMENT
CEZVE_OPERATION_RANGE
CEZVE_OPERATION_SUBTRACTION
CEZVE_OPERATION_TRUNCATING_DIVISION
CEZVE_OPERATION_TERNARY
CEZVE_ATOM_IDENTIFIER
CEZVE_ATOM_NAME
CEZVE_LITERAL_ARRAY
CEZVE_LITERAL_ARRAY_LIST
CEZVE_LITERAL_BOOLEAN
CEZVE_LITERAL_BYTE
CEZVE_LITERAL_CHARACTER
CEZVE_LITERAL_DOUBLE
CEZVE_LITERAL_FLOAT
CEZVE_LITERAL_HASHTABLE
CEZVE_LITERAL_INTEGER
CEZVE_LITERAL_LONG
CEZVE_LITERAL_NULL
CEZVE_LITERAL_SHORT
CEZVE_LITERAL_STRING
CEZVE_FRAGMENT_ATOM
CEZVE_FRAGMENT_EXPRESSION_LIST
CEZVE_FRAGMENT_FIELD
CEZVE_FRAGMENT_HASHTABLE_PAIR
CEZVE_FRAGMENT_METHOD
```

```
CEZVE_FRAGMENT_METHOD_ARGUMENTS
CEZVE_FRAGMENT_PACK_ALIAS
CEZVE_FRAGMENT_PACK_ALL
CEZVE_FRAGMENT_PACK_SINGLE
CEZVE_FRAGMENT_SLICER
CEZVE_FRAGMENT_SLICER_START
CEZVE_FRAGMENT_SLICER_STEP
CEZVE_FRAGMENT_SLICER_STOP
```

4.1.2. Çözümleme Örnekleri

Javy Sözcük Çözümleyici (Lexer), girilen karakter katarını sözcük (token) yığınına dönüştürür. Örnek olarak aşağıdaki bileşik if ifadesi ele alınabilir. BNF kuralları ve antlr yardımı ile oluşturduğumuz Sözcük çözümleyici karakter katarını ayrıştırır. Sözcük çözümleyici ayrıştırma yaparken, boşluk karakterlerini, sekme (tab) karakterini, satır sonu ve yorum dizgelerini yok sayar.

```
if (true) {
  println("true");
} else {
  println("false");
}
```

Bu kod parçasının muhtemel sözcük ayrıştırılması şu şekilde olacaktır.

```
'if'       : IF
' '        : WHITESPACE
'('        : LEFT_PARENTHESIS
'true'     : CEZVE_LITERAL_BOOLEAN
')'        : RIGHT_PARENTHESIS
' '        : WHITESPACE
'{'        : LEFT_CURLY_BRACKET
'\n'       : EOL
'  '       : WHITESPACE
'println'  : CEZVE_ATOM_NAME
'('        : LEFT_PARENTHESIS
'"true"'   : CEZVE_LITERAL_STRING
')'        : RIGHT_PARENTHESIS
';'        : SEMICOLON
'\n'       : EOL
```

```
'}'         : RIGHT_CURLY_BRACKET
' '         : WHITESPACE
'else'      : ELSE
' '         : WHITESPACE
'{'         : LEFT_CURLY_BRACKET
'\n'        : EOL
'  '        : WHITESPACE
'println'   : CEZVE_ATOM_NAME
'('         : LEFT_PARENTHESIS
'"false"'   : CEZVE_LITERAL_STRING
')'         : RIGHT_PARENTHESIS
';'         : SEMICOLON
'\n'        : EOL
'}'         : RIGHT_CURLY_BRACKET
'\n'        : EOL
```

Aritmetik bir işlemi ayrıştırma işlemine tabi tutalım.

```
println(1 + 5 - 56);
```

Oluşacak muhtemel sözcük listesi şu şekilde olacaktır.

```
'println'    : CEZVE_ATOM_NAME
'('          : LEFT_PARENTHESIS
'1'          : CEZVE_LITERAL_NUMBER
' '          : WHITESPACE
'+'          : PLUS
' '          : WHITESPACE
'5'          : CEZVE_LITERAL_NUMBER
' '          : WHITESPACE
'-'          : MINUS
' '          : WHITESPACE
'56'         : CEZVE_LITERAL_NUMBER
')'          : RIGHT_PARENTHESIS
';'          : SEMICOLON
'\n'         : EOL
```

4.2. Sözdizim Çözümleme İmlası

Sözcük Çözümleme işleminden sonra işlenen sözcükler, başka bir Sonlu Durum Makinesi içerisine sokulur. Bu çözümleme sisteminde sözcüklerin sırası denetlenir.

4.2.1. Çözümleyici Kuralları

Javy, C ve Java benzeri bir söz dizime sahiptir. Bunun çözümlenmesi, bir derleyici veya yorumlayıcı ile olur. ANTLR, bu derleyici veya yorumlayıcı sisteminin temelini oluşturmaya yardımcı olup, BNF kurallarından temel kodu üretecektir. Javy'de kullandığımız temel söz dizim kuralları şöyledir:

`startParser` Söz Dizim Kuralı :

startParser, ayrıştırıcı sistemin temel kuralıdır. Söz dizim ayrıştırma işlemi, sözcük ayrıştırmadan farklı olarak bir başlangıç kuralı ile başlar. Sözcük ayrıştırıcıda ise daha önceden bahsettiğimiz gibi kurallar yukarıdan aşağı işletilir.

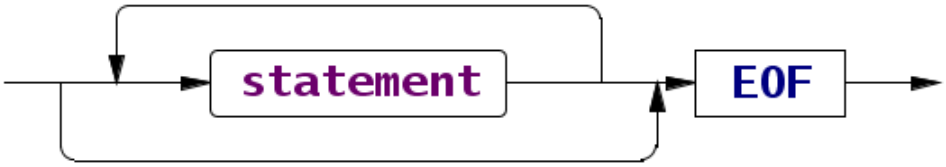
```
startParser
  :  statement* EOF -> ^(CEZVE_SCRIPT statement*)
  ;
```

Javy Sözdizim başlangıç kuralı temel olarak bu şekildedir. Fakat bu kural ağaç ayrıştırıcı için ek tanımlamalar içermektedir. Kural, yeniden yazım (rewrite) ifadelerinden bağımsız olarak şu şekilde özetlenebilir.

```
startParser: statement* EOF;
```

ifade edilebilir. startParser başlangıç kuralı, *ya hiç ya da bir çok "statement" kuralından* oluşur. Fakat ne olursa olsun, dizi bir dosya sonu ile

sonlandırılmalıdır. EOF (End Of File) sözcüğü (token), ANTLR sisteminin entegre sözcüklerinden biridir.



Şekil 46: startParser Söz Dizim Çözümleyici Kuralı

```
-> ^(CEZVE_SCRIPT statement*)
```

Opsiyonel kısmı, ağaç yapısı oluşturmamızı sağlamaktadır. JAVY_SCRIPT Kavramsal Sözcüğü ile beraber ifade parametrelerinin ağaç çözümleyiciye aktarılması sağlanır. Örneğin işlevsel (LISP tarzı) şekilde yazarsak, aktarılan veriler şu kombinasyonda olabilir.

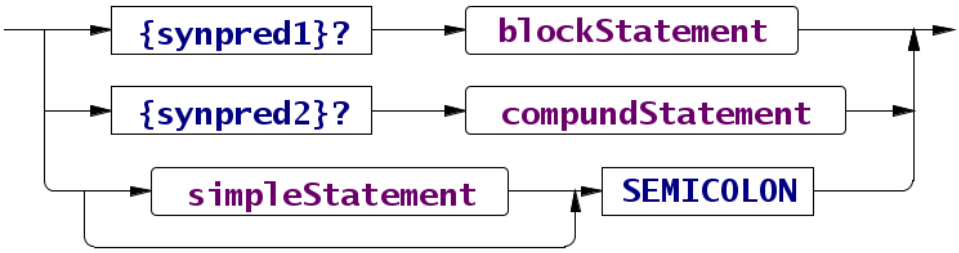
- CEZVE_SCRIPT
- CEZVE_SCRIPT ifade1
- CEZVE_SCRIPT ifade1 ifade2
- CEZVE_SCRIPT ifade1 ifade2 ifade3
- ….

`statement` **Söz Dizim Kuralı :**

Bir programlama dilinin üst düzey yapı taşları, ifadelerdir. Döngü, koşul, seçim gibi birçok ifade, programlama diline ait algoritmanın şekillenmesini sağlar. Javy programlama dili; tek satırlık basit ifadeler, bileşik ifadeler ve blok ifadeler adlı 3 temel ifade kullanmaktadır.

```
statement
options {backtrack = true;}
  :  blockStatement
  |  compoundStatement
  |  simpleStatement? SEMICOLON!
  ;
```

ifade kuralına ait temel gösterimdir. Kuraldaki (**options {backtrack = true;}**) özellik tanımı, ANTLR'a özgü "geri dönüş" özellik bildirimidir. Eğer bu bayrak ayarlanmış ise herhangi bir belirsizlik (nonterminism) durumunda, Sözdizim Çözümleyici, dallandığı alt kurallardan tekrar bu kurala kadar geri dönüp, **"veya"** ile birleştirilmiş alt kurallardan diğerini deneyebilecektir. Bu işlem nispeten çözümleme süresini yavaşlatır, fakat belirsiz bir sistemi kısmen belirliye dönüştürebilir.



Şekil 47: statement Söz Dizim Çözümleyici Kuralı

Yukarıdaki imlada alt kurallarda bahsi geçeceği gibi, blockStatement ifadesi de simpleStatement ifadesi de kıvrık parantez **"{"** ile başlayabilecektir. Bu bir belirsizlik yaratacağından imla tahmini (syntactic predicate) bir başka değişle, geri dönüm (backtrack) kullanılmıştır.

statement kuralı, blockStatement, compoundStatement, simpleStatement kurallarından uygun olanına dallanacaktır. Dikkat edilmesi gereken noktalardan biri, simpleStatement kuralına uygun basit ifade, noktalı virgül **";"** ile sonlandırılmalıdır. Noktalı virgül, sözcük çözümleyici tanımlarında SEMICOLON olarak anlamsallaştırılmıştır.

SEMICOLON sözcüğü ünlem işareti ile bitmektedir. Bu tanım, SEMICOLON sözcüğünün ağaç ayrıştırıcıya aktarılmasına gerek olmadığını belirtmektedir.

`blockStatement` **Söz Dizim Kuralı :**

Javy varsayılan imlasında blockStatement ifadesi, bir alanı izole etmek için kullanılmaktadır. Tek başına bir ifade olabileceği gibi, compoundStatement'a ait bir alt ifade de olabilmektedir. Java'da her blockStatement, kendi özel değişken uzayına sahiptir. Javy şimdilik her bloğa özgü isim uzayını desteklememektedir.

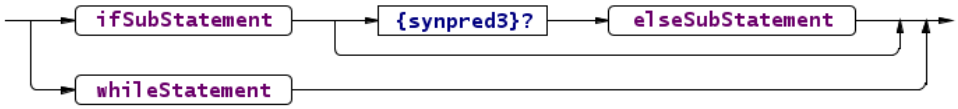
```
blockStatement
    :   LEFT_CURLY_BRACKET! statement+ RIGHT_CURLY_BRACKET!
    ;
```

Kıvırcık parantezler (CURLY BRACKET) "{ … }" arasında kalan tüm ifadeler, Blok İfade olarak adlandırılmaktadır. Blok İfadenin Hash Tablosu oluşturan

tip ifadesi ile belirsizliğe girmemesi için kural; *"Bir ve birden fazla ifadeyi barındırır."* şeklinde tanımlanmıştır.

Şekil 48: blockStatement Söz Dizim Çözümleyici Kuralı

`compoundStatement` Söz Dizim Kuralı :



Şekil 49: compoundStatement Söz Dizim Çözümleyici Kuralı

Javy, svn R41 sürümünden itibaren bileşik ifade olarak if ve while ifadelerini desteklemektedir. compoundStatement ifadesi, ayrıştırma ağacında if ve while ifadelerinin yol ayrımını teşkil eder. if, while ve daha sonraki sürümlere ait diğer bileşik ifadeler bu kuraldan dallanacaklardır.

```
compoundStatement
  :  ifSubStatement (options {backtrack = true;} :
elseSubStatement)?
     -> ^(CEZVE_STATEMENT_IF ifSubStatement elseSubStatement?
CEZVE_STATEMENT_IF_END)
  |  whileStatement
  ;
```

Kural, kabaca if ifadesi ya da while ifadesi şeklindedir. if ifadesi, iki adet alt kurala bölünmüştür. Ek olarak, seçimli (opsiyonel) bir adet else ifadesi bulunmaktadır.

if alt kuralı,

```
ifSubStatement
    ifSubStatement
        elseSubStatement
```

ile

```
ifSubStatement
    ifSubStatement
elseSubStatement
```

arasındaki ayrımı belirleyemez bu nedenle burada imla tahmini bir başka değişle geri dönüş kullanılmıştır.

compoundStatement ifadesi yalnızca if ifadesi için Ağaç Ayrıştırma Sözcükleri kullanmaktadır. While ifadesine ait ağaç ayrıştırma sözcüğü, ait olduğu kural tanımlanırken belirtilecektir. Bir if ifadesinin ağaç gösterimi, CEZVE_STATEMENT_IF kavramsal sözcüğü ile CEZVE_STATEMENT_IF_END kavramsal sözcüğü arasında tanımlanmaktadır.

`ifSubStatement` Söz Dizim Kuralı :

Şekil 50: ifSubStatement Söz Dizim Çözümleyici Kuralı

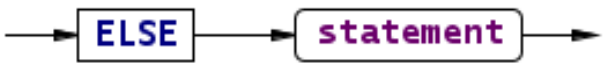
ifSubStatement cümleciği, C ve Java tarzı programlama dillerinde bulunan şartlı ifadeleri sunmaktadır. compoundStatement ifadesinde tüm bir if ifadesi, iki parçaya bölünmüştü. Bu iki alt parçadan zorunlu olanı, if alt ifadesidir.

```
ifSubStatement
  :  IF LEFT_PARENTHESIS expression RIGHT_PARENTHESIS statement
     -> ^(CEZVE_SUB_STATEMENT_IF expression statement
CEZVE_SUB_STATEMENT_IF_END)
  ;
```

ifSubStatement kuralı, sırası ile if ile başlayıp parantez içerisinde koşul ifadesi ve koşul gerçekleştiğinde çalışacak olan ifade şeklinde devam eder.

kural, dallandığı compundStatement ifadesinin alt parçası olan if ifadesinin ağaç gösterimindeki gibi başlangıç ve bitiş sözcüğünden oluşur. Koşul ifadesi ve çalıştırılacak olan ifade, bu iki sözcük arasında ağaç ayrıştırıcıya sunulur.

`elseSubStatement` Söz Dizim Kuralı :



Şekil 51: elseSubStatement Söz Dizim Çözümleyici Kuralı

elseSubStatement cümleciği, ifSubStatement gibi if ifadesine ait alt kurallardan biridir. Fakat çoğu programlama dilinde olduğu gibi seçimlidir (opsiyonel). İfade, ifSubStatement kuralında olduğu gibi bir koşul gösterimi içermez. ifSubStatement koşulu gerçeklenmez ise, varsayılan olarak bu alt ifade çalıştırılacaktır.

```
elseSubStatement
  :  ELSE statement -> ^(CEZVE_SUB_STATEMENT_ELSE statement
CEZVE_SUB_STATEMENT_ELSE_END)
  ;
```

Kurala ait ağaç gösterim,i ifSubStatement ile benzerlik göstermektedir. Kuralın başladığını bildiren CEZVE_SUB_STATEMENT_ELSE sözcüğü ve bittiğini gösteren CEZVE_SUB_STATEMENT_ELSE_END sözcüğü arasına çalıştırılacak olan ifade yazılmaktadır.

`whileStatement` Söz Dizim Kuralı :

Şekil 52: whileStatement Söz Dizim Çözümleyici Kuralı

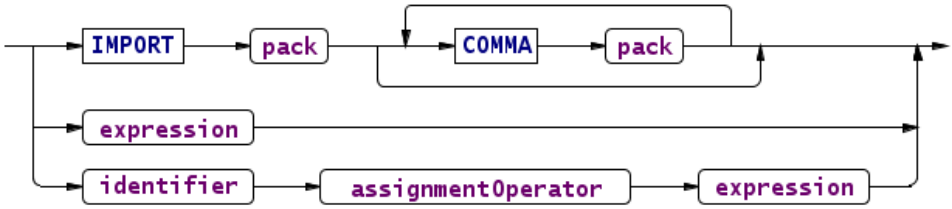
while ifadesi, bir döngü ifadesidir. C, Java tarzı programlama dillerindeki benzer kullanıma sahiptir. Koşul doğru olana kadar ifade işletilir.

```
whileStatement
  :  WHILE LEFT_PARENTHESIS expression RIGHT_PARENTHESIS
statement
     -> ^(CEZVE_STATEMENT_WHILE expression statement
CEZVE_STATEMENT_WHILE_END)
  ;
```

whileStatement kuralında, while ifadesinden sonra parantezler arasında koşul tanımlanır. Kuralın sonunda ise koşulun doğru olması durumunda, çalışacak olan ifade yer alır.

Ağaç Çözümleyiciye aktarılan veri ifSubStatement ile oldukça benzerdir. Yalnızca kavramsal sözcükler değişmiştir.

`simpleStatement` Söz Dizim Kuralı :



Şekil 53: simpleStatement Söz Dizim Çözümleyici Kuralı

simpleStatement BNF kuralı, tek satırlık ifadeleri kapsar. Standart çıktı aygıtına yazı yazdırma, paket aktarımı ve atama işlemlerini üzerinde barındırır.

```
simpleStatement
  :  IMPORT pack (COMMA pack)*
     -> ^(CEZVE_STATEMENT_IMPORT pack+)
  |  expression -> ^(CEZVE_STATEMENT_EXPRESSION  expression)
  |  identifier assignmentOperator^ expression
  ;
```

Kural, bir çok alt kural “ya da” işlemi ile birleştirilmiştir. Kural, bir sınıf aktarım ifadesi olan import veya tekil bir gösterim olabileceği gibi, bir atama işlemi de olabilir. IMPORT alt kuralı, virgülle ayrılmış paket kurallarını barındırır. En az bir paket adı ve sonra girilen her paket için virgül kullanılmalıdır.

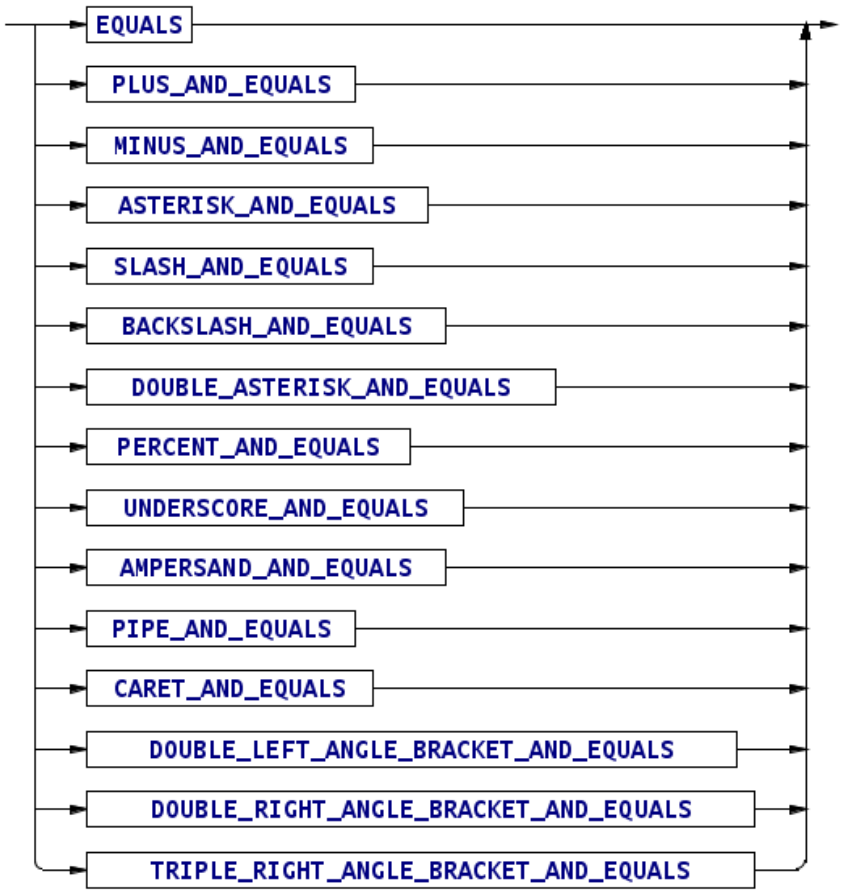
```
import paket1, paket2, paket3;
```

simpleStatement kuralı, sadece işleç değerlendirme ifadesi (expression) çalıştırmaya da olanak tanır. Bu sayede kabuk üzerinden hesaplamalar veya işleç işlemleri bir ifade içerisine alınmadan gerçeklenmiş olacaktır.

Atama işlemini yapan alt kural, bir belirteçten sonra assignmentOperator kuralını barındırır. Son olarak atama yapan değeri göstereceğimiz işleç değerlendirme ifadesi (expression) bulunur.

Kuralların Ağaç gösterimi ise kendilerine ait CEZVE kavramsal sözcükleri ile başlayıp alt parametrelerinin belirtilmesi ile tanımlanmaktadır.

`assignmentOperator` **Söz Dizim Kuralı :**



Şekil 54: assignmentOperator Söz Dizim Çözümleyici Kuralı

Javy, Cezve dil altyapısı ile doğrudan atama ve bileşik atama işlemlerini desteklemektedir. Bunların birçoğu Java'daki kullanımları ile benzeşirken bazıları da Javy Diline özel kullanılmaktadır. Atama işleçleri, Javy İşleç önceliği tablosuna göre 1. seviye işleçlerdir. Yani ifade içerisinde yer alması durumunda, en son işlem görürler. Öncelik seviyesi en düşük olduğundan, diğer bütün işleç işlemlerinin sonucu atama işleçleri kullanılarak bir değişkene atanır.

```
assignmentOperator
  :  EQUALS -> CEZVE_STATEMENT_ASSG
  |  PLUS_AND_EQUALS -> CEZVE_STATEMENT_ASSG_ADDITION
  |  MINUS_AND_EQUALS -> CEZVE_STATEMENT_ASSG_SUBTRACTION
```

```
|  ASTERISK_AND_EQUALS -> CEZVE_STATEMENT_ASSG_MULTIPLICATION
|  SLASH_AND_EQUALS -> CEZVE_STATEMENT_ASSG_DIVISION
|  BACKSLASH_AND_EQUALS -> CEZVE_STATEMENT_ASSG_T_DIVISION
|  DOUBLE_ASTERISK_AND_EQUALS -> CEZVE_STATEMENT_ASSG_POWER
|  PERCENT_AND_EQUALS -> CEZVE_STATEMENT_ASSG_MODULO
|  UNDERSCORE_AND_EQUALS
   -> CEZVE_STATEMENT_ASSG_CONCATENATION
|  AMPERSAND_AND_EQUALS -> CEZVE_STATEMENT_ASSG_AND
|  PIPE_AND_EQUALS -> CEZVE_STATEMENT_ASSG_OR
|  CARET_AND_EQUALS -> CEZVE_STATEMENT_ASSG_XOR
|  DOUBLE_LEFT_ANGLE_BRACKET_AND_EQUALS
   -> CEZVE_STATEMENT_ASSG_BITWISE_SL
|  DOUBLE_RIGHT_ANGLE_BRACKET_AND_EQUALS
   -> CEZVE_STATEMENT_ASSG_BITWISE_SRS
|  TRIPLE_RIGHT_ANGLE_BRACKET_AND_EQUALS
   -> CEZVE_STATEMENT_ASSG_BITWISE_SRU
;
```

Atama işleç sözcükleri, normalde simpleStatement kuralındaki atama kuralı içerisine eklenebilir. Ayrı bir söz dizim kuralı olarak tanımlanmasındaki temel neden, Cezve Anlamsal Sözcükleri ile kuralın yeniden yazılmasını mümkün kılmaktır.

`expressionList` Söz Dizim Kuralı :

Şekil 55: expressionList Söz Dizim Çözümleyici Kuralı

expressionList kuralı virgülle ayrılmış birçok gösterim ifadesini barındırır. Metod argümanları, liste verileri gibi kurallar tarafından çağrılmaktadır.

```
expressionList
  :  expression (COMMA expression)* COMMA? ->
^(CEZVE_FRAGMENT_EXPRESSION_LIST expression+)
  ;
```

Sonuç olarak Cezve Dil Altyapısındaki bütünleşik Ağaç Çözümleyiciye CEZVE_FRAGMENT_EXPRESSION_LIST tanım sözcüğü ile beraber birçok gösterim ifadesi gönderir. Kural, işleç değerlendirme ifadelerinin virgülle ayırılması şeklinde tanımlanabilir. Seçimli (opsiyonel) olarak ifade sonuna da virgül konulabilir.

`expression` Söz Dizim Kuralı :

Şekil 56: expression Söz Dizim Çözümleyici Kuralı

expression, işleç (operatör) değerlendirme kuralıdır. Zincirleme olarak alt kurallara dallanır. Bu dallanma, operatör önceliklerini belirlemektedir.

```
expression
  :  conditionalOrOperation
     (QUESTION_MARK expression COLON expression)?
     -> ^(
          CEZVE_OPERATION_TERNARY
          conditionalOrOperation
          (expression expression)?
        )
  ;
```

Diğer alt işleç değerlendirme alt kurallarına kök teşkil eden bu kural aslında if .. else ifade tanımıdır.

```
a = true ? "ifade dogru" : "ifade yanlis";
```

İfadede true değerinin bulunduğu alan, bir boolean true değeri veya bu değeri veren bir gösterim ifadesi ise, ? ve : arasında kalan gösterim ifadesinin sonucu çıktı olacaktır. Aksi bir durumda ise, ifadenin son kısmı sonuç olarak döndürülecektir.

Diğer dillerde ternary işleci olarak geçen bu gösterim kuralının seçimli (opsiyonel) kısmı, özyinelemeli (recursive) bir yapı oluşturur. Seçimli kısmında expression kuralı tekrar kendini çağırmaktadır. ANTLR sistemi, kuralın sağındaki değerlerin özyineleme döngüsüne girmesine izin verir. Kuralın ilk elemanı böyle bir döngüye (LEFT RECURSION) giremez. Böyle bir belirsiz durum ile karşılaşılması halinde, imla tahminine başvurulur (Syntactic Predication).

expression kuralı, Javy dilinde 2. seviye işleç işlemi olarak tanımlanır. Yani önceliği neredeyse en düşük işlemdir.

`conditionalOrOperation` **Söz Dizim Kuralı :**

Şekil 57: conditionalOrOperation Söz Dizim Çözümleyici Kuralı

conditionalOrOperation kuralı koşullu VEYA işlemini belirtmek için kullanılır. Kural birden fazla işleç seçeneği sunduğundan, bu işleçler ayrı bir kural olarak belirtilmiştir.

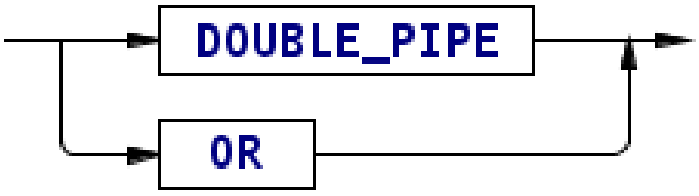
```
conditionalOrOperation
  :  conditionalAndOperation
     (conditionalOrOperator^ conditionalAndOperation)*
  ;
```

Kuralın seçimli opsiyonel kısmı, hemen hemen bütün iki parametre gerektiren işleç tanımlama kurallarında olduğu gibi seçimlidir.

- <koşullu VE işlemi>
- <koşullu VE işlemi> <VEYA İşleci> <koşullu VE işlemi>
- <koşullu VE işlemi> <VEYA İşleci> <koşullu VE işlemi> <VEYA İşleci> <koşullu VE işlemi>
- ....

Kuralın seçimli kısmı 0,1 veya 1'den fazla tekrara olanak sağlamaktadır. Kural, ayrıca 3. seviyeden önceliğe sahip bir işleç kuralıdır.

`conditionalOrOperator` **Söz Dizim Kuralı :**



Şekil 58: conditionalOrOperator Söz Dizim Çözümleyici Kuralı

4. seviyede önceliğe sahip conditionalOrOperation (Koşullu VEYA) işlemi Javy dilinde iki adet eş operatör ile sağlanabilir. İki işleçten herhangi biri gereken yerde kullanılabilir.

```
conditionalOrOperator
```

```
:  DOUBLE_PIPE -> CEZVE_OPERATION_OR
|  OR          -> CEZVE_OPERATION_OR
;
```

"or" ve "||" işleçlerinin her ikisi de, CEZVE_OPERATION_OR anlamsal sözcüğüne denkleştirilmiştir. Bu da işleçlerin birbirlerinin aynısı olduğunu gösterir.

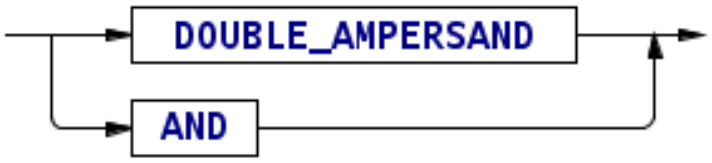
`conditionalAndOperation` Söz Dizim Kuralı :

Şekil 59: conditionalAndOperation Söz Dizim Çözümleyici Kuralı

Temel koşullu VE işlemlerini tanımlayan bu söz dizim kuralı, diğer iki argümana sahip işleç işlemi kurallarındaki benzer yapıya sahiptir. Bir üst seviye kurala dallanmayı sağlayan bir alt kural bulunur. Daha sonra, seçimli olarak kullanılan bir adet işleç ve yine seçimli olarak kullanılan üst seviye kural bağlantısı bulunur.

```
conditionalAndOperation
  :  bitwiseOrOperation
     (conditionalAndOperator^ bitwiseOrOperation)*
  ;
```

`conditionalAndOperator` Söz Dizim Kuralı :



Şekil 60: conditionalAndOperator Söz Dizim Çözümleyici Kuralı

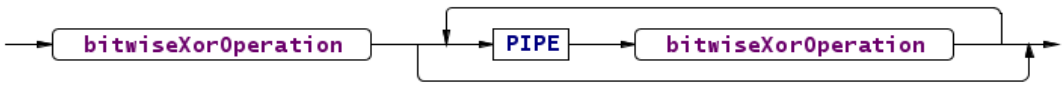
Koşullu VE işleminin sahip olduğu kuralların CEZVE standart sözcükleri ile eşleşmesi için, ek bir conditionalAndOperator kuralı tanımlanır. Bu kural AND ve DOUBLE_AMPERSAND sözcüklerini içermektedir.

```
conditionalAndOperator
  :  DOUBLE_AMPERSAND -> CEZVE_OPERATION_AND
  |  AND              -> CEZVE_OPERATION_AND
  ;
```

İki işleç te CEZVE_OPERATION_AND anlamsal sözcüğü ile eşleşir. Bu da işleçlerin birbirleri yerine kullanılabileceği manasına gelir.

```
<bitwiseOrOperation> && <bitwiseOrOperation>
<bitwiseOrOperation> and <bitwiseOrOperation>
```

`bitwiseOrOperation` **Söz Dizim Kuralı :**

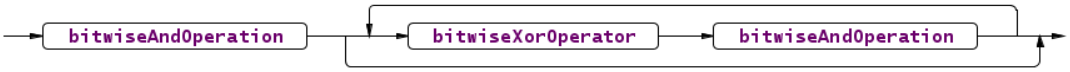


Şekil 61: bitwiseOrOperation Söz Dizim Çözümleyici Kuralı

bitwiseOrOperation, bit bazlı VEYA işleminin tanımlanmasını sağlamaktadır. VEYA işleci, Javy dilinde, Java'da olduğu gibi tek bir işleç ile kullanıldığından ek bir söz dizim kuralı ile tanımlanmamıştır.

```
bitwiseOrOperation
  :  bitwiseXorOperation (PIPE bitwiseXorOperation)*
     -> ^(CEZVE_OPERATION_BITWISE_OR
          bitwiseXorOperation
          bitwiseXorOperation?)
  ;
```

`bitwiseXorOperation` Söz Dizim Kuralı :

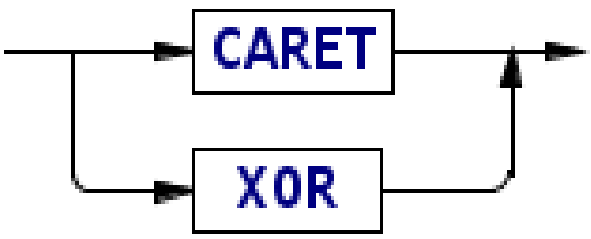


Şekil 62: bitwiseXorOperation Söz Dizim Çözümleyici Kuralı

bitwiseOrOperation, bit bazlı ÖZEL VEYA işleminin tanımlanmasını sağlamaktadır. Javy dilinde Java'dan farklı olarak iki eş işleç ile ifade edilebilir. Bu işleçler, bitwiseXorOperation adlı farklı bir söz dizim kuralında tanımlanmışlardır.

```
bitwiseXorOperation
  :  bitwiseAndOperation
     (bitwiseXorOperator^ bitwiseAndOperation)*
  ;
```

`bitwiseXorOperator` Söz Dizim Kuralı :



Şekil 63: bitwiseXorOperator Söz Dizim Çözümleyici Kuralı

bitwiseXorOperator söz dizim kuralı, bitwiseXorOperation kuralı tarafından çağrılır. CARET ve XOR sözcüklerinden herhangi birinin bulunması, bu kuralı sağlayacaktır. Java dilinden farklı olarak Javy, xor işlecini sunmaktadır.

```
bitwiseXorOperator
  :  CARET -> CEZVE_OPERATION_BITWISE_XOR
  |  XOR   -> CEZVE_OPERATION_BITWISE_XOR
  ;
```

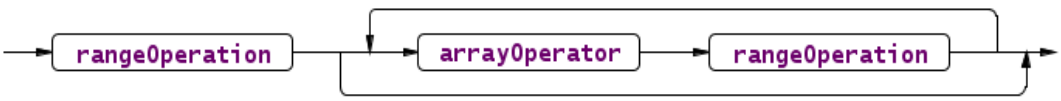
`bitwiseAndOperation` Söz Dizim Kuralı :

Şekil 64: bitwiseAndOperation Söz Dizim Çözümleyici Kuralı

bitwiseAndOperation, bit bazlı VE işleminin tanımlanmasını sağlamaktadır. Java dilindeki kullanım ile aynıdır. Yalnızca bir adet işleç, bu kural tarafından çağrılacağı için işleçler ek bir kurala bölünmemişlerdir.

```
bitwiseAndOperation
  :  arrayOperation (AMPERSAND arrayOperation)*
     -> ^(CEZVE_OPERATION_BITWISE_AND
        arrayOperation arrayOperation?)
  ;
```

`arrayOperation` Söz Dizim Kuralı :

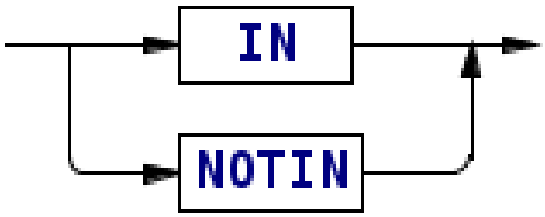


Şekil 65: arrayOperation Söz Dizim Çözümleyici Kuralı

Javy, Java'dan farklı olarak diziler üzerinde işlem yapabilen işleçler de sunmaktadır. Bu öncelik seviyesine ait iki farklı işleç bulunmaktadır. Bu işleçlerin CEZVE standart sözcüklerine dönüşümü, diğer işleç işlemi kurallarında olduğu gibi farklı bir söz dizim kuralı ile yapılır.

```
arrayOperation
  :  rangeOperation (arrayOperator^ rangeOperation)*
  ;
```

`arrayOperator` **Söz Dizim Kuralı :**



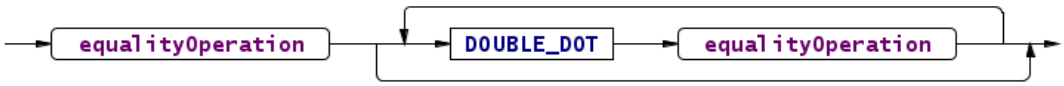
Şekil 66: arrayOperator Söz Dizim Çözümleyici Kuralı

arrayOperator söz dizim kuralı, arrayOperation kuralı tarafından çağrılan bir işleç eşleme kuralıdır. İki farklı işleç seçeneği sunan bu kural, dizi aitliğini test eden IN ve NOTIN sözcüklerini barındırır. IN işleci, Python programlama dilinde bir nesne örneğinin bir dizi içerisinde olup olmadığını test etmemizi sağlayan bir operatör olarak karşımıza çıkar. Bu işleç, sonuç olarak bir boolean değerini döndürür.

NOTIN sözcüğü ise IN sözcüğünün bağlı olduğu işleç işlemini uygulayıp, sonucu tersleyip döndürecektir. Tüm bu işlemlerin hepsi, Cezve Dil Altyapısında meydana gelir. Bu kuralın yaptığı ise CEZVE_OPERATION_IN ve CEZVE_OPERATION_NOTIN anlamsal sözcüklerini çağrılan kurala döndürmektir.

```
arrayOperator
  :  IN          -> CEZVE_OPERATION_IN
  |  NOTIN       -> CEZVE_OPERATION_NOTIN
  ;
```

`rangeOperation` **Söz Dizim Kuralı :**



Şekil 67: rangeOperation Söz Dizim Çözümleyici Kuralı

Dizi işlemlerinin bir üst seviyesinde yer alan rangeOperation kuralı, temelde bir dizi işlemini sağlar. Belirtilen aralıkta bir dizi oluşturmayı sağlayan bu kural, Perl programlama dilindeki kullanımına oldukça benzemektedir. Kural, yalnızca bir adet işleç barındırdığından, işleçlerin toplandığı ek bir alt kurala ihtiyaç duymaz.

```
rangeOperation
  :  equalityOperation (DOUBLE_DOT equalityOperation)*
     -> ^(CEZVE_OPERATION_RANGE
        equalityOperation equalityOperation?)
  ;
```

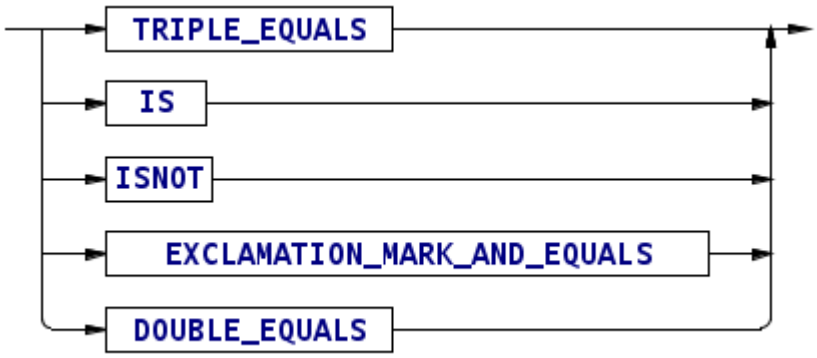
`equalityOperation` **Söz Dizim Kuralı :**

Şekil 68: equalityOperation Söz Dizim Çözümleyici Kuralı

equalityOperation kuralı, eşitlik test işleçlerini barındıran Javy söz dizim kuralıdır. Birden fazla eşitlik doğrulama işleci sunan kural, işleçler için ek bir alt kurala ayrılmıştır.

```
equalityOperation
  :  comparisionOperation (equalityOperator^
comparisionOperation)*
  ;
```

`equalityOperator` **Söz Dizim Kuralı :**



Şekil 69: equalityOperator Söz Dizim Çözümleyici Kuralı

equalityOperator kuralı, equalityOperation kuralı tarafından çağrılan bir söz dizim kuralıdır. Bir çok nesne, eşitlik testini gerçekleştirecek işleç içerir. TRIPLE_EQUALS, IS, ISNOT, EXCLAMATION_MARK_AND_EQUALS, DOUBLE_EQUALS sözcüklerini barındırır. Bu sözcükler, Cezve Dil Altyapısındaki

karşılıkları ile eşleştirilir. Bu sözcüklerin kullanımına Javy imlası bölümünde değinilmiştir.

```
equalityOperator
  :  TRIPLE_EQUALS -> CEZVE_OPERATION_COMPARISION_IDE
  |  IS -> CEZVE_OPERATION_COMPARISION_IDE
  |  ISNOT -> CEZVE_OPERATION_COMPARISION_NIDE
  |  EXCLAMATION_MARK_AND_EQUALS
     -> CEZVE_OPERATION_COMPARISION_NEQ
  |  DOUBLE_EQUALS -> CEZVE_OPERATION_COMPARISION_EQU
  ;
```

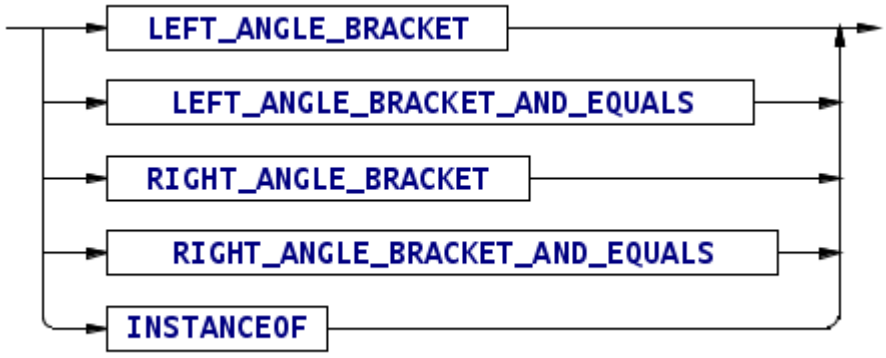
`comparisionOperation` **Söz Dizim Kuralı :**

Şekil 70: comparisionOperation Söz Dizim Çözümleyici Kuralı

Javy'de sayısal büyüklük karşılaştırma işleçlerini bu kural çağırmaktadır. Kural birden çok işlece ev sahipliği yapacağından, işleçler diğer kurallarda olduğu gibi ek bir söz dizim kuralına ayrılmıştır.

```
comparisionOperation
  :  shiftOperation (comparisionOperator^ shiftOperation)*
  ;
```

`comparisionOperator` **Söz Dizim Kuralı :**

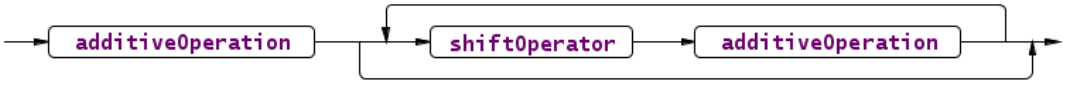


Şekil 71: comparisionOperator Söz Dizim Çözümleyici Kuralı

comparisionOperator kuralı, comparisionOperation kuralı tarafından çağırılır. Kural içeriği, diğer benzer işleç barındıran kurallar ile temelde aynıdır. LEFT_ANGLE_BRACKET, LEFT_ANGLE_BRACKET_AND_EQUALS, RIGHT_ANGLE_BRACKET, RIGHT_ANGLE_BRACKET_AND_EQUALS sayısal büyüklük karşılaştırma sözcüklerini içerir. Bunların yanında, instanceof nesne aidiyet karşılaştırması işlecini de içermektedir.

```
comparisionOperator
  :  LEFT_ANGLE_BRACKET -> CEZVE_OPERATION_COMPARISION_LT
  |  LEFT_ANGLE_BRACKET_AND_EQUALS
     -> CEZVE_OPERATION_COMPARISION_LTE
  |  RIGHT_ANGLE_BRACKET
     -> CEZVE_OPERATION_COMPARISION_GT
  |  RIGHT_ANGLE_BRACKET_AND_EQUALS
     -> CEZVE_OPERATION_COMPARISION_GTE
  |  INSTANCEOF -> CEZVE_OPERATION_COMPARISION_INSTANCEOF
  ;
```

`shiftOperation` **Söz Dizim Kuralı :**



Şekil 72: shiftOperation Söz Dizim Çözümleyici Kuralı

Javy dili, Java'da olduğu gibi bit tabanlı kaldırma işleçlerine sahiptir. Bu işleçlerin çağrıldığı ana kural, shiftOperation söz dizim kuralıdır. Kuralda, işleçler ayrı bir alt kural çağrılarak eklenir.

```
shiftOperation
  :  additiveOperation (shiftOperator^ additiveOperation)*
  ;
```

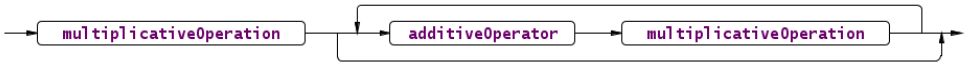
`shiftOperator` **Söz Dizim Kuralı :**

Şekil 73: shiftOperator Söz Dizim Çözümleyici Kuralı

shiftOperator kuralı, shiftOperation söz dizim kuralı tarafından çağrılır. Temel bit tabanlı kaydırma işleçlerine ait sözcükleri barındırır. DOUBLE_LEFT_ANGLE_BRACKET, DOUBLE_RIGHT_ANGLE_BRACKET, TRIPLE_RIGHT_ANGLE_BRACKET sözcüklerini Cezve dil altyapısı tarafından yorumlanabilecek standart sözcüklere dönüştürür.

```
shiftOperator
  :  DOUBLE_LEFT_ANGLE_BRACKET  -> CEZVE_OPERATION_BITWISE_SL
  |  DOUBLE_RIGHT_ANGLE_BRACKET -> CEZVE_OPERATION_BITWISE_SRS
  |  TRIPLE_RIGHT_ANGLE_BRACKET -> CEZVE_OPERATION_BITWISE_SRU
  ;
```

`additiveOperation` **Söz Dizim Kuralı :**

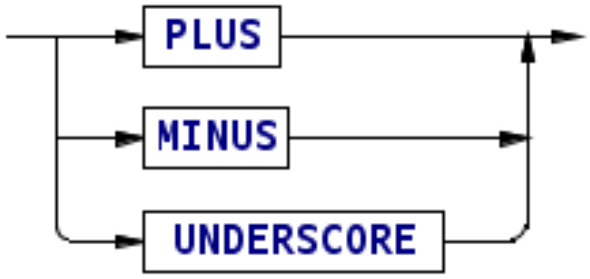


Şekil 74: additiveOperation Söz Dizim Çözümleyici Kuralı

additiveOperation, temel toplama, çıkarma ve dizge birleştirme işleçlerinin dallanma kuralıdır. Bu kural, işleçlerin barındığı alt kuralı çağırıp, sonuç ağacını Cezve dil altyapısına gönderir.

```
additiveOperation
  :  multiplicativeOperation
     (additiveOperator^ multiplicativeOperation)*
  ;
```

`additiveOperator` Söz Dizim Kuralı :



Şekil 75: additiveOperator Söz Dizim Çözümleyici Kuralı

additiveOperator kuralı, additiveOperation kuralı tarafından çağrılmaktadır. Temel toplama, çıkarma ve dizge birleştirme işleçleri olan PLUS, MINUS, UNDERSCORE sözcüklerini barındırır.

```
additiveOperator
  :  PLUS       -> CEZVE_OPERATION_ADDITION
  |  MINUS      -> CEZVE_OPERATION_SUBTRACTION
  |  UNDERSCORE -> CEZVE_OPERATION_CONCATENATION
```

```
;
```

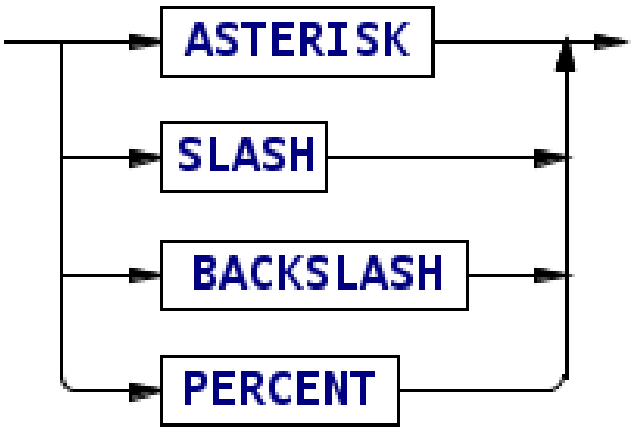
`multiplicativeOperation` **Söz Dizim Kuralı :**

Şekil 76: multiplicativeOperation Söz Dizim Çözümleyici Kuralı

Çarpma ve Çoğullama işleçlerinin dallanma kuralı olan multiplicativeOperation söz dizim kuralı, gelen standart cezve sözcüğü merkezinde bir ağaç oluşturacaktır.

```
multiplicativeOperation
  :  powerOperation (multiplicativeOperator^ powerOperation)*
  ;
```

`multiplicativeOperator` **Söz Dizim Kuralı :**



Şekil 77: multiplicativeOperator Söz Dizim Çözümleyici Kuralı

multiplicativeOperator kuralı, multiplicativeOperation söz dizim kuralı tarafından çağrılmaktadır. Temel çarpma, bölme, kalansız bölüm alma ve mod işlemlerini gerçekleştirecek ASTERISK, SLASH, BACKSLASH, PERCENT sözcüklerini içerir. Bu sözcükler, diğer işleç kurallarında olduğu gibi, Cezve Dil altyapısına ait standart sözcükler ile eşleşir.

```
multiplicativeOperator
  :  ASTERISK  -> CEZVE_OPERATION_MULTIPLICATION
  |  SLASH     -> CEZVE_OPERATION_DIVISION
  |  BACKSLASH -> CEZVE_OPERATION_TRUNCATING_DIVISION
  |  PERCENT   -> CEZVE_OPERATION_MODULO
  ;
```

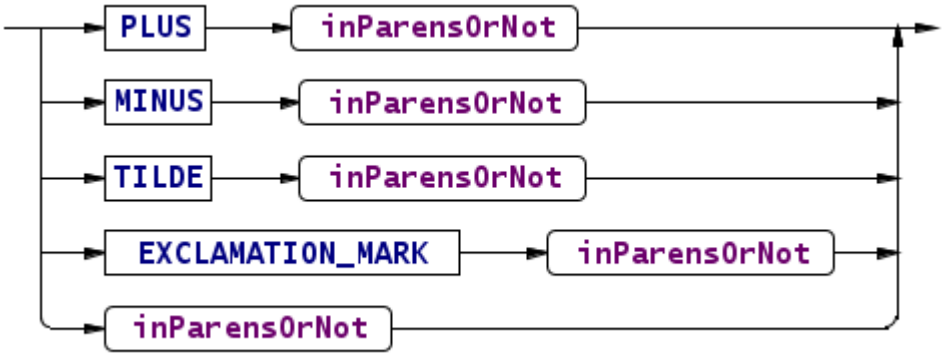
`powerOperation` **Söz Dizim Kuralı :**

Şekil 78: powerOperation Söz Dizim Çözümleyici Kuralı

powerOperation, tek bir işleç barındıran bir dallanma kuralıdır. İşleç bir adet olduğu için, ek bir alt kural içerisinde tanımlanmamıştır. powerOperation kuralı, Python programlama dilinde de kullanılan “**” : DOUBLE_ASTERISK sözcüğünü barındırır.

```
powerOperation
  :  unaryOperation (DOUBLE_ASTERISK unaryOperation)*
     -> ^(CEZVE_OPERATION_POWER unaryOperation unaryOperation?)
  ;
```

`unaryOperation` **Söz Dizim Kuralı :**

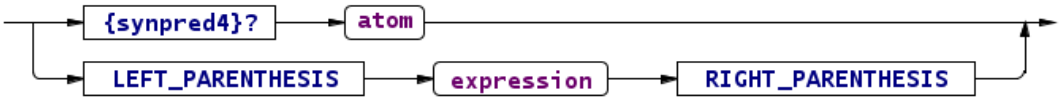


Şekil 79: unaryOperation Söz Dizim Çözümleyici Kuralı

Bundan önceki işleçlerin hemen hepsi, iki argüman üzerinde çalışmaktaydı. Bu söz dizim kuralı ise tekil işleçleri tanımlamaktadır. Kural, pozitif ve negatif belirteci olan + (PLUS) ve – (MINUS) sözcüklerini ve bit tabanlı tümleme işleci olan ~ (TILDE) ile Boolean tersleme işleci olan (!) EXCLAMATION_MARK sözcüklerini barındırır. unaryOperation kuralı, bir alt kuralı olan inParensOrNot kuralına doğrudan erişim veya önüne bir tekil işlecin getirildiği durumları doğrular.

```
unaryOperation
  :  PLUS inParensOrNot
     -> ^(CEZVE_OPERATION_POSITIVE inParensOrNot)
  |  MINUS inParensOrNot
     -> ^(CEZVE_OPERATION_NEGATIVE inParensOrNot)
  |  TILDE inParensOrNot
     -> ^(CEZVE_OPERATION_BITWISE_COMPLEMENT inParensOrNot)
  |  EXCLAMATION_MARK inParensOrNot
     -> ^(CEZVE_OPERATION_NOT inParensOrNot)
  |  inParensOrNot
  ;
```

`inParensOrNot` **Söz Dizim Kuralı :**



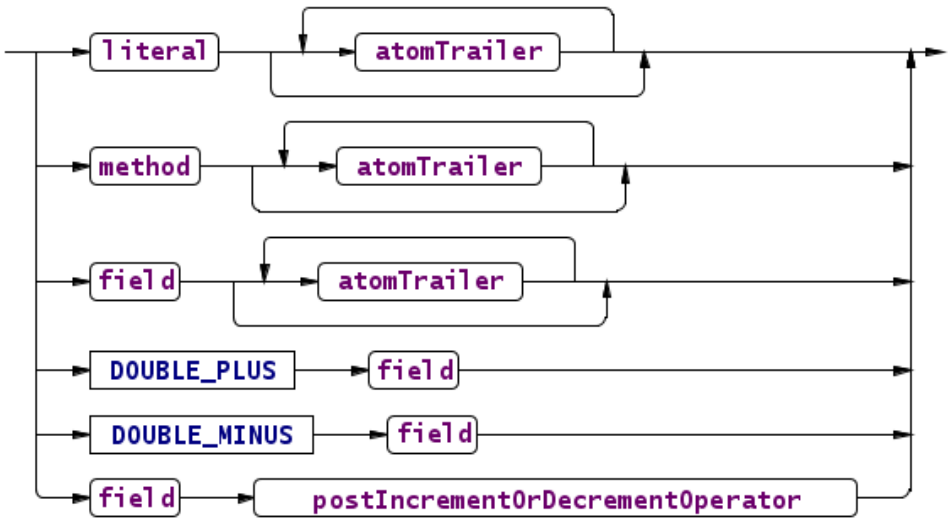
Şekil 80: inParensOrNot Söz Dizim Çözümleyici Kuralı

inParensOrNot kuralı, bir gösterim ifadesinin parantez içine alınıp alınmadığını denetleyecektir. Ayrıştırma işlemi, ifadeler parantez içerisine alınmamış ise atom alt kuralı ile devam edecektir. Eğer parantez içerisine alınan bir gösterim mevcut ise, parantez içine alınan değer için expression kuralına geri dönülür. Ve öz yinelemeli olarak, iç içe geçen parantezler miktarınca işlem devam eder.

Javy imlasında belirtildiği gibi, parantez işlemi aynı zamanda işleç önceliklerini özelleştirebilmemizi de sağlamaktadır.

```
inParensOrNot
options {backtrack = true;}
  :  atom
  |  LEFT_PARENTHESIS expression RIGHT_PARENTHESIS ->
^(CEZVE_OPERATION_IN_PARENS expression)
  ;
```

`atom` Söz Dizim Kuralı :



Şekil 81: atom Söz Dizim Çözümleyici Kuralı

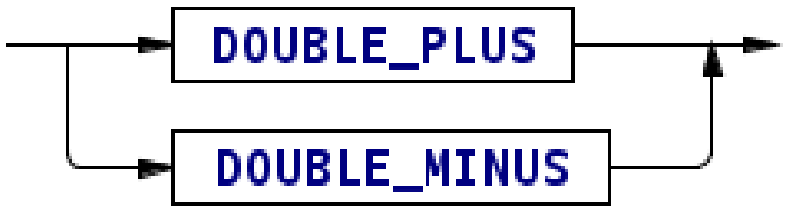
atom söz dizim kuralı, söz dizim çözümleyicinin en temel alt kurallarından biridir. Literal, method, field alt kurallarından herhangi birinden sonra, bir veya birden fazla atomTrailer kuralı gelebilir.

Buna ilaveten, herhangi bir alandan önce, DOUBLE_PLUS ve DOUBLE_MINUS arttırma ve azaltma işleçleri gelebilir. Bu işleçler, bir field'dan sonra da gelebilir, fakat Cezve sistemine bu işleçlerin her biri farklı bir sözcük ile ulaştırılacaktır.

field kuralından önce gelen arttırma işleci, CEZVE_OPERATION_PRE_INCREMENT olurken, azaltma işlemi de PRE_DECREMENT olacaktır. field kuralından sonra gelen işleçler ise aşağıdaki kuralda göründüğü gibi, POST dizgesi ile ifade edilmektedir.

```
atom
  :  literal atomTrailer*
     -> ^(CEZVE_FRAGMENT_ATOM literal atomTrailer*)
  |  method atomTrailer*
     -> ^(CEZVE_FRAGMENT_ATOM method  atomTrailer*)
  |  field atomTrailer*
     -> ^(CEZVE_FRAGMENT_ATOM field   atomTrailer*)
  |  DOUBLE_PLUS field
      -> ^(CEZVE_FRAGMENT_ATOM
           CEZVE_OPERATION_PRE_INCREMENT field)
  |  DOUBLE_MINUS field
     -> ^(CEZVE_FRAGMENT_ATOM
          CEZVE_OPERATION_PRE_DECREMENT field)
  |  field postIncrementOrDecrementOperator
     -> ^(CEZVE_FRAGMENT_ATOM
          postIncrementOrDecrementOperator field)
  ;
```

`postIncrementOrDecrementOperator` **Söz Dizim Kuralı :**

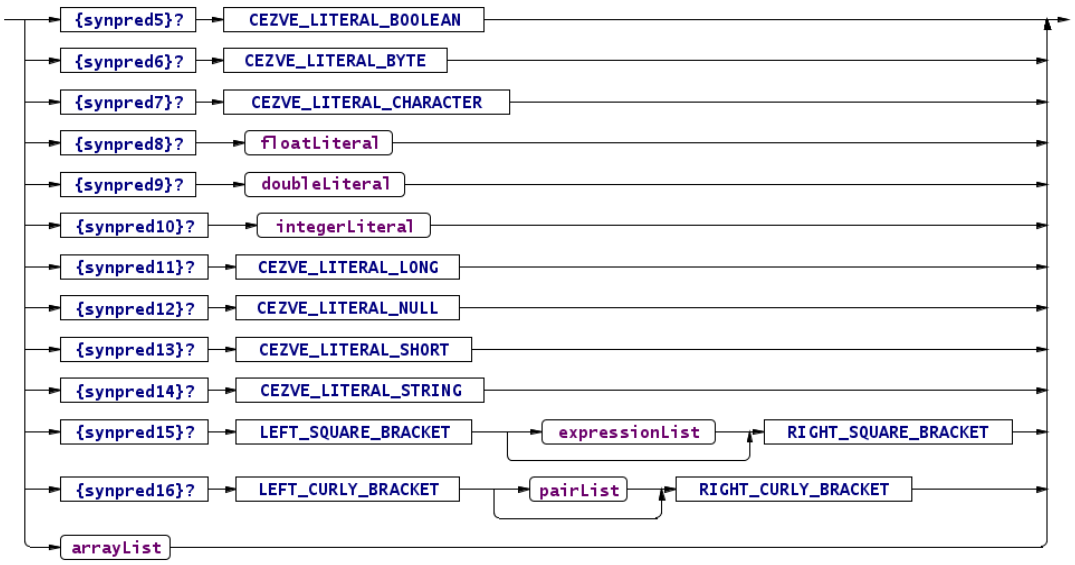


Şekil 82: postIncrementOrDecrementOperator Söz Dizim Çözümleyici Kuralı

postIncrementOrDecrementOperator kuralı, arkasından geldiği field üzerine etkiyerek değerin arttırılmasını veya azaltılmasını sağlar. field sonrası gelen arttırma (DOUBLE_PLUS) ve azaltma (DOUBLE_MINUS) operatörlerini barındıran postIncrementOrDecrementOperator kuralı, atom kuralı tarafından çağrılacaktır.

```
postIncrementOrDecrementOperator
  :  DOUBLE_PLUS  -> CEZVE_OPERATION_POST_INCREMENT
  |  DOUBLE_MINUS -> CEZVE_OPERATION_POST_DECREMENT
  ;
```

`literal` Söz Dizim Kuralı :



Şekil 83: literal Söz Dizim Çözümleyici Kuralı

literal kuralı, bir atom altı söz dizim kuralıdır ve doğrudan tanımlanabilen temel veri tiplerini belirtmemizi sağlar. Birçok seçeneği, sözcük çözümleyici (Lexer) kurallarıdır. Ve bu kurallar, Sözcük Çözümleyici imlasında tanımlanmıştır. Bu seçeneklere ilaveten, dizi tanımlamaya yarayan köşeli parantez ifadesi, Sözlük/Hash tablosu gibi koleksiyon tipleri de burada tanımlanır.

Bir "[" (LEFT_SQUARE_BRACKET) sözcüğünden sonra seçimli (opsiyonel) ifade listesi ve kapanış olarak ta "]" (RIGHT_SQUARE_BRACKET) kullanıldığında, kural bu sözcükleri dizi olarak tanımlayacaktır. Benzer şekilde, "{" (LEFT_CURLY_BRACKET), eşleşme listesi (pair list) ve kapanış olarak ta

“}” (RIGHT_CURLY_BRACKET) kullanıldığında, kural sözcükleri bir Sözlük/Hashtable olarak tanımlayacaktır.

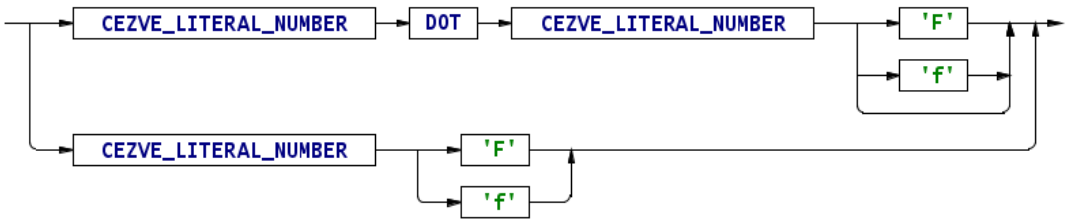
Temel sayısal veri tiplerinden floatLiteral, doubleLiteral ve integerLiteral hariç, diğer hepsi sözcük çözümleyici kuralı olarak tanımlanmışken, bu sayısal veri tipi kuralları, söz dizim kuralı olarak tanımlanmıştır. Bunun temel nedeni, float ve double veri tiplerinin sahip olduğu ondalık bölüm ayracı olan noktanın kullanılmasıdır. Benzer bir işleç, Javy'de nesne aitliği işleci olarak kullanılır. Bu nedenle, belirsizliği önlemek için bu kurallar söz dizim kuralı olarak tanımlanmıştır. Tanımlanan kuralların ikisi de sayısal veri ile başladığından dolayı farklı bir belirsizlik (non-determism) ortaya çıkmaktadır. Bu belirsizlikte, imla tahmini (syntactic predication) ile çözümlenir. Bu nedenle, kuralın en başına {backtrack=true;} özelliği konulmuştur. Bu özelliğe ilaveten, kuralda sözcük çözümleyicinin ileri bakış (lookahead) miktarı 2'den 4'e çıkarılmıştır.

Kural seçeneklerine ek olarak, dizi listesi de ek bir alt kural ile tanımlanmaktadır. arrayList adlı alt söz dizim kuralı bu tanımlamayı yapmaktadır.

```
literal
options {backtrack = true; k = 4;}
  :  CEZVE_LITERAL_BOOLEAN
  |  CEZVE_LITERAL_BYTE
  |  CEZVE_LITERAL_CHARACTER
  |  floatLiteral
  |  doubleLiteral
  |  integerLiteral
  |  CEZVE_LITERAL_LONG
```

```
|  CEZVE_LITERAL_NULL
|  CEZVE_LITERAL_SHORT
|  CEZVE_LITERAL_STRING
|  LEFT_SQUARE_BRACKET expressionList? RIGHT_SQUARE_BRACKET
   -> ^(CEZVE_LITERAL_ARRAY expressionList?)
|  LEFT_CURLY_BRACKET pairList? RIGHT_CURLY_BRACKET
   -> ^(CEZVE_LITERAL_HASHTABLE pairList?)
|  arrayList
;
```

`floatLiteral` Söz Dizim Kuralı :

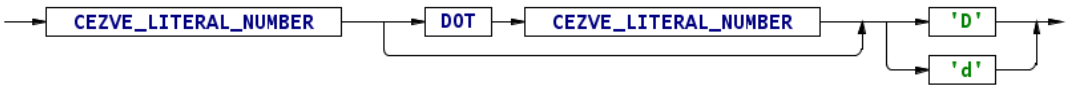


Şekil 84: floatLiteral Söz Dizim Çözümleyici Kuralı

Temel ondalık sayıları tanımlayan bu kural şu şekildedir. Mutlaka nümerik sözcüğün ondalık kısmı, bir nokta ile ayırt edilmelidir ve ifade sonuna seçimli olarak F veya f son eki getirilebilir. Bu varsayılan ondalık veri tipi olduğundan, nümerik veri ondalık biçimde ise bir son eke ihtiyaç duyulmadan belirtilebilir. Eğer ondalık kısmı olmayan bir sayı, float veri tipinde tanımlanmak isteniyor ise mutlaka F veya f son ekine sahip olmalıdır.

```
floatLiteral
  :  CEZVE_LITERAL_NUMBER DOT CEZVE_LITERAL_NUMBER ('F' | 'f')?
     -> ^(CEZVE_LITERAL_FLOAT
          CEZVE_LITERAL_NUMBER CEZVE_LITERAL_NUMBER)
  |  CEZVE_LITERAL_NUMBER ('F' | 'f')
     -> ^(CEZVE_LITERAL_FLOAT CEZVE_LITERAL_NUMBER)
  ;
```

`doubleLiteral` Söz Dizim Kuralı :



Şekil 85: doubleLiteral Söz Dizim Çözümleyici Kuralı

doubleLiteral söz dizim kuralı, double veri tipindeki sözcükleri tanımlamak için kullanılmaktadır. Kuralda bir nümerik ifade ondalık kısmı olsun olmasın mutlaka D veya d son ekine sahip olmalıdır. Bu biçimi sağlayan bir nümerik veri, Cezve Dil Altyapısına CEZVE_LITERAL_DOUBLE belirteci ile gönderilir.

```
doubleLiteral
  :  CEZVE_LITERAL_NUMBER
     (DOT CEZVE_LITERAL_NUMBER)? ('D' | 'd')
     -> ^(CEZVE_LITERAL_DOUBLE
          CEZVE_LITERAL_NUMBER CEZVE_LITERAL_NUMBER?)
  ;
```

`integerLiteral` Söz Dizim Kuralı :

Şekil 86: integerLiteral Söz Dizim Çözümleyici Kuralı

integerLiteral kuralı, CEZVE_LITERAL_NUMBER sözcük kuralının söz dizim kuralının eşidir. Kural ek olarak sözcüğün belirtecini değiştirip, CEZVE_LITERAL_NUMBER yapacaktır.

```
integerLiteral
  :  CEZVE_LITERAL_NUMBER
     -> ^(CEZVE_LITERAL_INTEGER CEZVE_LITERAL_NUMBER)
  ;
```

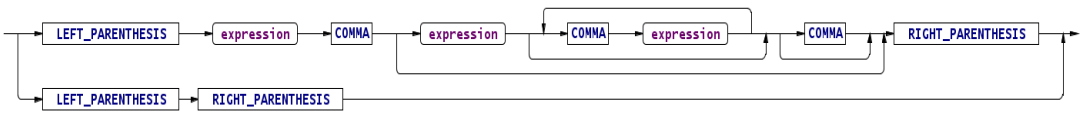
`identifier` Söz Dizim Kuralı :

Şekil 87: identifier Söz Dizim Çözümleyici Kuralı

identifer söz dizim kuralı, paket isimleri, değişken, sınıf, metot ve alan isimlerinin tümünü kapsayabilen bir kuraldır. Kural, yalnızca harflerden oluşabilen CEZVE_ATOM_IDENTIFIER veya sayısal değerler de içerebilen CEZVE_ATOM_NAME sözcük çözümleme kurallarından birinin eşleşmesi ile doğru sonuç döndürmektedir.

```
identifier
  :  CEZVE_ATOM_IDENTIFIER
  |  CEZVE_ATOM_NAME
  ;
```

`arrayList` Söz Dizim Kuralı :



Şekil 88: arrayList Söz Dizim Çözümleyici Kuralı

arrayList sözdizim kuralı, dizi listesi tanımlamamız için kullandığımız temel veri tipi kurallarından biridir. Kural, boş bir dizi tanımını da içerebileceği gibi, virgülle ayrılmış gösterim ifadelerini içeren alt ögelerden oluşmuş bir dizi tanımı da içerebilir.

```
arrayList
  :  LEFT_PARENTHESIS expression COMMA
```

```
    (expression (COMMA expression)* COMMA?)? RIGHT_PARENTHESIS
    -> ^(CEZVE_LITERAL_ARRAY_LIST expression+)
|   LEFT_PARENTHESIS RIGHT_PARENTHESIS
    -> CEZVE_LITERAL_ARRAY_LIST
;
```

`pairList` Söz Dizim Kuralı :

Şekil 89: pairList Söz Dizim Çözümleyici Kuralı

pairList söz dizim kural, sözlük/hashtable tanımlamasının alt kurallarından biridir. Hashtable veri tipinin tanımlandığı, literal kuralı tarafından çağırılır. Virgülle ayrılmış eşleşmelerden oluşmaktadır. Ve listede en az bir eşleşme bulunması gerekmektedir.

```
pairList
  :  pair (COMMA pair)* COMMA? -> pair+
  ;
```

`pair` Söz Dizim Kuralı :

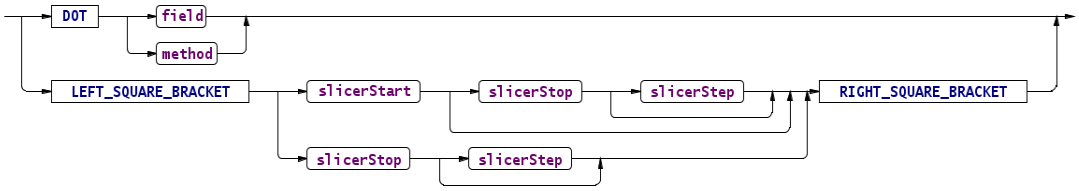
Şekil 90: pair Söz Dizim Çözümleyici Kuralı

pair söz dizim kuralı, literal kuralında tanımlanan hashtable veri tipi ile başlayan pairList kuralı ile devam eden kurallar zincirinin bir halkasıdır. Sözlük/hash

tablosu veri tipi, tanımlanırken virgülle ayrılmış eşleşme listesine ve ":" karakteri ile ayrılmış eşleşmelere ihtiyaç duyulur. pair kuralı, bu eşleşmeleri tanımlamaktadır.

```
pair
  :  expression COLON expression
     -> ^(CEZVE_FRAGMENT_HASHTABLE_PAIR expression+)
  ;
```

`atomTrailer` Söz Dizim Kuralı :



Şekil 91: atomTrailer Söz Dizim Çözümleyici Kuralı

atom kuralı, nesnelerin alt özelliklerine erişim sağlamak için kullanılan bir kuraldır. Bu kural ile nesnelerin metotlarına, alanlarına veya nesne dizilerinin istenen eleman ve elamanlarına ulaşılabilir. Paket, sınıf ve sınıf örneklerinin alt ögelerine erişim için, "." işleci kullanırız ve bu yapı, hiyerarşik bir şekilde devam eder. Nesnenin sonuna eklenen zincir belirteçleri, bu kuralın ilk seçeneğini sağlamaktadır.

Kuralın ilk seçeneği, bir nokta sözcüğünden sonra field veya method kuralının gelebileceğini söyler. Kuralın ikinci seçeneği ise köşeli parantezler içerisine alınmış dizi eleman erişim ifadesini tanımlamaktadır. Bu dizi elemanı erişim yaklaşımı, python programlama dilindekine benzer bir şekilde tasarlanmıştır.

```
atomTrailer
  :  DOT! (field | method)
  |  LEFT_SQUARE_BRACKET
     (slicerStart (slicerStop slicerStep?)? | slicerStop
slicerStep?)
     RIGHT_SQUARE_BRACKET
     -> ^(CEZVE_FRAGMENT_SLICER slicerStart? slicerStop?
slicerStep?)
  ;
```

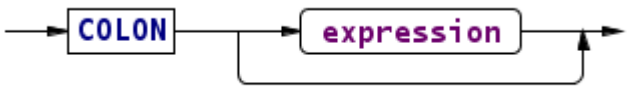
`slicerStart` Söz Dizim Kuralı :

Şekil 92: slicerStart Söz Dizim Çözümleyici Kuralı

Bir dizi ve karakter katarının belirli bir elemanına erişimi sağlayan trailer kuralının ikinci seçeneği, tek bir elemanın yanında belirli bir indeks aralığındaki elemanlara da erişim sağlar. Bu bağlamda slicerStart kuralı, bu aralığın ilk değerini belirtir.

```
slicerStart
  :  expression -> ^(CEZVE_FRAGMENT_SLICER_START expression)
  ;
```

`slicerStop` Söz Dizim Kuralı :

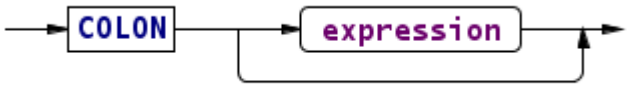


Şekil 93: slicerStop Söz Dizim Çözümleyici Kuralı

slicerStop kuralı, slicerStart kuralı ile belirtilen bir aralık başlangıç değerine karşılık bir son değer girilmesine olanak tanımaktadır.

```
slicerStop
  :  COLON expression? -> ^(CEZVE_FRAGMENT_SLICER_STOP
expression?)
  ;
```

`slicerStep` Söz Dizim Kuralı :



Şekil 94: slicerStep Söz Dizim Çözümleyici Kuralı

slicerStep kuralı, slicerStart ve slicerStop kuralları ile belirtilen dizi aralığındaki değerler seçilirken gerek duyulan artım miktarını belirtir. Ön tanımlı artım miktarı 1'dir. Bir dizinin 4 ile 8 nolu indeks değerleri arasındaki elemanları seçtiğimizde, ön tanımlı dönecek indeks değerleri: 4,5,6,7,8 olacaktır. Fakat slicerStep ile basamak miktarını 2 yaptığımızda, dönecek değerler: 4,6,8 olacaktır.

```
slicerStep
  :  COLON expression? -> ^(CEZVE_FRAGMENT_SLICER_STEP
expression?)
  ;
```

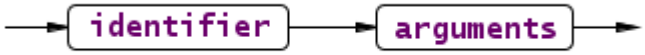
`field` Söz Dizim Kuralı :

Şekil 95: field Söz Dizim Çözümleyici Kuralı

field söz dizim kuralı, sınıf, paket, sınıf ögeleri, değişkenler gibi temel birçok belirtecin söz dizim çözümleyiciye tanıtıldığı kuraldır. field kuralı, identifier alt kuralını çağırdığından, alfa sayısal harf ve nümerik değerler kombinasyonlarını kapsayabilir.

```
field
  :  identifier -> ^(CEZVE_FRAGMENT_FIELD identifier)
  ;
```

`method` Söz Dizim Kuralı :



Şekil 96: method Söz Dizim Çözümleyici Kuralı

method kuralı, field kuralından farklı olarak sonuna argüman listesi alır. Sınıf metodları, yapıcı metotlar ve Cezve dil altyapısı ön tanımlı metotlarının belirtildiği temel kuraldır.

```
method
  :  identifier arguments
     -> ^(CEZVE_FRAGMENT_METHOD identifier arguments)
  ;
```

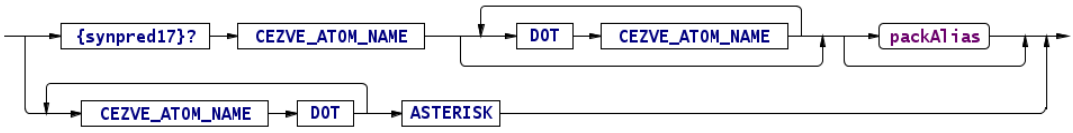
`arguments` Söz Dizim Kuralı :

Şekil 97: arguments Söz Dizim Çözümleyici Kuralı

arguments kuralı, method kuralı tarafından çağırılır. Bir metot, yapıcı metot veya Cezve Dil Altyapısına ait bütünleşik metodun parametreleri olabilir. Bu parametrelere karşılık gönderilen argümanlar, arguments kuralı tarafından tanımlanır.

```
arguments
  :  LEFT_PARENTHESIS expressionList? RIGHT_PARENTHESIS
     -> ^(CEZVE_FRAGMENT_METHOD_ARGUMENTS expressionList?)
  ;
```

`pack` Söz Dizim Kuralı :



Şekil 98: pack Söz Dizim Çözümleyici Kuralı

pack kuralı, import sınıf aktarım ifadesi tarafından çağırılır. Bir pack tanımı, tek bir sınıfı kapsayacak şekilde nokta ile ayrılmış harf katarı olan CEZVE_ATOM_NAME sözcüklerini kapsayabileceği gibi, birçok sınıfı kapsayacak şekilde nokta ile ayrılmış belirteç değerlerinin en sonuna "*" karakteri de alabilir.

```
pack
options {backtrack = true;}
  :  CEZVE_ATOM_NAME (DOT CEZVE_ATOM_NAME)* packAlias?
     -> ^(CEZVE_FRAGMENT_PACK_SINGLE CEZVE_ATOM_NAME+
packAlias?)
  |  (CEZVE_ATOM_NAME DOT)+ ASTERISK ->
^(CEZVE_FRAGMENT_PACK_ALL CEZVE_ATOM_NAME+)
  ;
```

Bu tanımlamalara ilaveten, tekli paket tanımlamalarında sınıf takma ismi verilebilir. Bunun için packAlias metodu çağırılır.

`packAlias` **Söz Dizim Kuralı :**

Şekil 99: packAlias Söz Dizim Çözümleyici Kuralı

packAlias metodu, pack kuralı tarafından çağrılır. Basit bir şekilde, "as" sözcüğünden sonra gelen bir belirteç, bu kuralı doğrulamış olur. Bu kural sayesinde, Cezve Dil Altyapısına gönderilen belirteç verisi, sınıf takma ismine dönüştürülür.

```
packAlias
  :  AS identifier -> ^(CEZVE_FRAGMENT_PACK_ALIAS identifier)
  ;
```

4.2.2. Çözümleme Örnekleri

Sözcük çözümleyici, girilen betiği sözcük yığınlarına dönüştürür. Sözcük çözümleyicinin kuralları, bu yığınların oluşumunu şekillendirir. Javy sözcük çözümleme imlasındaki örneği, bu kısma taşıyalım:

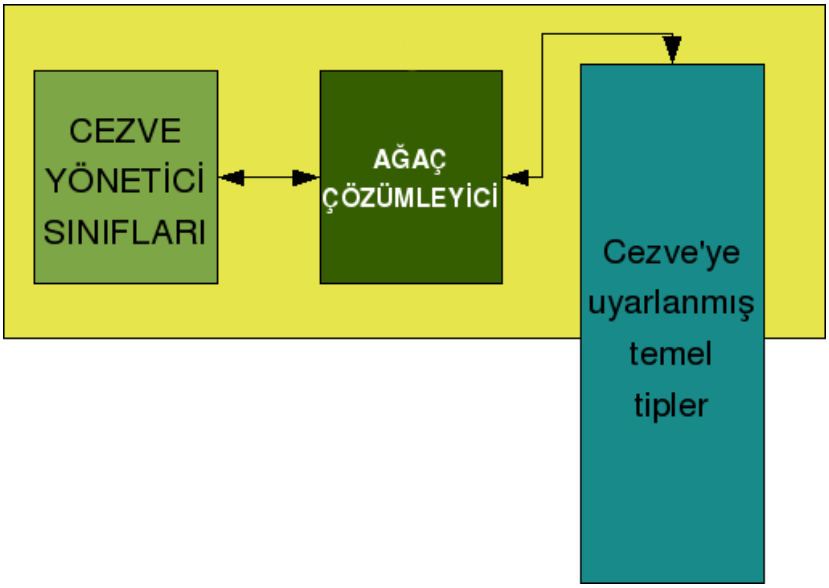
```
if (true) {
  println("true");
} else {
  println("false");
}
```

Bu kod parçasının muhtemel sözcük ayrıştırılması, şu şekilde olacaktır.

```
'if'       : IF
' '        : WHITESPACE
'('        : LEFT_PARENTHESIS
'true'     : CEZVE_LITERAL_BOOLEAN
')'        : RIGHT_PARENTHESIS
' '        : WHITESPACE
'{'        : LEFT_CURLY_BRACKET
'\n'       : EOL
'  '       : WHITESPACE
'println'  : CEZVE_ATOM_NAME
'('        : LEFT_PARENTHESIS
'"true"'   : CEZVE_LITERAL_STRING
')'        : RIGHT_PARENTHESIS
';'        : SEMICOLON
'\n'       : EOL
'}'        : RIGHT_CURLY_BRACKET
' '        : WHITESPACE
'else'     : ELSE
' '        : WHITESPACE
'{'        : LEFT_CURLY_BRACKET
'\n'       : EOL
'  '       : WHITESPACE
'println'  : CEZVE_ATOM_NAME
'('        : LEFT_PARENTHESIS
'"false"'  : CEZVE_LITERAL_STRING
')'        : RIGHT_PARENTHESIS
';'        : SEMICOLON
'\n'       : EOL
'}'        : RIGHT_CURLY_BRACKET
'\n'       : EOL
```

Mevcut örneği söz dizim ayrıştırma işlemine tabi tuttuğumuzda, söz dizim ayrıştırma ağacı, EK-A'daki gibi olacaktır.

5. CEZVE DİL ALTYAPISI VE İMLA ENTEGRASYONU



Şekil 100: Cezve Dil Altyapısı İç Mimarisi

Cezve Dil Altyapısı, bütünleşik ağaç ayrıştırıcısı ve ağaç ayrıştırıcısındaki verileri çözümlemek için birçok yönetici sınıf barındırır. Bunlara ilaveten, Java Sanal Makinesinin sağladığı temel veri tiplerinin bazılarına müdahalede bulunur.

Cezve Dil Altyapısı, bir çok imla paketini modül olarak bünyesine alabilecek bir sisteme sahiptir. Çekirdek yapısı içerisinde bulunan yönetici sınıflarına ek olarak, komut istemi yorumlayıcısı, servlet alt sınıfı ve istisna yönetimi paketine de sahiptir.

Cezve dil altyapısının bir imla paketini karşıladığı ilk yer, ağaç ayrıştırıcısıdır. Ağaç ayrıştırıcının imla paketi ile standart bir iletişim kuralı

üzerinden haberleşmesi gerekir. Bunu sağlayan komut seti ise ağaç ayrıştırıcı sözcüklerinin kendisidir.

AST (Abstract Syntax Tree – Özet İmla Ağacı) olarak nitelendirilen bu sözcükler, kendileri ile beraber gönderilen argümanlar ile anlamlıdırlar. Bu AST kuralları, sözcük çözümleyici ve söz dizim çözümleyicide olduğu gibi, bir imla ile tanımlanmıştır. Bu tanımlamaların arasına Javy yönetici sınıflarının metotları eklenir. Bu sayede ağaç ayrıştırıcı kodu oluşturulup derlendiğinde, sistem ile bütünleşik çalışabilir.

Ağaç ayrıştırıcı örneği oluşturulduğu andan itibaren bazı küresel public değişken tanımlamaları yapılır.

```
Hashtable variableTable = null;
Hashtable importTable = null;
AssignmentManager assignmentManager = null;
StatementManager statementManager = null;
BuiltIn builtIn = null;
ClassPathManager classPathManager = null;
AtomManager atomManager = null;
ExpressionManager expressionManager = new ExpressionManager();
TypeManager typeManager = new TypeManager();
Object outputWriter = null;
```

variableTable küresel değişkeni, sistem içerisindeki tanımlanacak olan değişkenlerin tutulacağı havuzdur. Küresel değişkenin adı ile değeri eşleştirilir.

importTable küresel değişkeni, sistem içerisine aktarılmış sınıf isimlerini barındırmaktadır. Bu sınıflar doğrudan örneklendirilebilirler.

assignmentManager küresel değişkeni, atama işlemlerini yöneten bir sınıf örneğidir.

builtIn küresel değişkeni ise Cezve dil altyapısının bütünleşik işlevlerini barındıran sınıf örneğidir.

classPathManager, paket yükleme ve yüklü paketlerin aktarımını sağlayan yönetici sınıf örneğidir.

atomManager, temel veri tiplerini, örneklendirilmiş nesne alanlarını, gösterim ifadelerini, metotları çalıştıran ve yöneten sınıf örneğidir.

expressionManager, gösterim ifadelerinin yönetilmesini sağlayan yönetici sınıf örneğidir.

typeManager küresel değişkeni ise temel veri tiplerinin yönetilmesini sağlayan TypeManager sınıfının bir örneğidir.

outputWriter, kullanılan sistem çıktı birimi tutan değişkendir.

5.1.1. `startTreeParser` Ağaç Ayrıştırıcısı Kuralı

```
startTreeParser
  :  ^(CEZVE_SCRIPT statement*)
  ;
```

startTreeParser kuralı, kök kural olarak CEZVE_SCRIPT ifadesini zorunlu tutar. Müteakip kuralı, seçimli olarak bir veya birden fazla statement kuralı takip edebilir. Bu kural çalıştığı anda, birçok yönetici sınıfın örneğini oluşturup küresel değişkenlere atayacaktır.

```
assignmentManager = new AssignmentManager(
  variableTable, expressionManager
);
statementManager = new StatementManager(
  importTable, classPathManager, outputWriter
);
builtIn = new BuiltIn(importTable);
atomManager = new AtomManager(
  importTable,
  variableTable,
  builtIn,
  classPathManager,
  expressionManager
);
```

5.1.2. `statement` Ağaç Ayrıştırcısı Kuralı

```
statement
  :  ^(CEZVE_STATEMENT_EXPRESSION expression)
  |  ^(CEZVE_STATEMENT_IMPORT     pack+)
  |  ^(CEZVE_STATEMENT_IF
       ifSubStatement elseSubStatement? CEZVE_STATEMENT_IF_END)
  |  ^(CEZVE_STATEMENT_WHILE      whileStatement)
  |  ^(CEZVE_STATEMENT_ASSG
       identifier expression)
  |  ^(CEZVE_STATEMENT_ASSG_ADDITION
       identifier expression)
  |  ^(CEZVE_STATEMENT_ASSG_SUBTRACTION
       identifier expression)
  |  ^(CEZVE_STATEMENT_ASSG_MULTIPLICATION
       identifier expression)
  |  ^(CEZVE_STATEMENT_ASSG_DIVISION
       identifier expression)
  |  ^(CEZVE_STATEMENT_ASSG_T_DIVISION
       identifier expression)
  |  ^(CEZVE_STATEMENT_ASSG_POWER
```

```
      identifier expression)
|   ^(CEZVE_STATEMENT_ASSG_MODULO
      identifier expression)
|   ^(CEZVE_STATEMENT_ASSG_CONCATENATION
      identifier expression)
|   ^(CEZVE_STATEMENT_ASSG_AND
      identifier expression)
|   ^(CEZVE_STATEMENT_ASSG_OR
      identifier expression)
|   ^(CEZVE_STATEMENT_ASSG_XOR
      identifier expression)
|   ^(CEZVE_STATEMENT_ASSG_BITWISE_SL
      identifier expression)
|   ^(CEZVE_STATEMENT_ASSG_BITWISE_SRS
      identifier expression)
|   ^(CEZVE_STATEMENT_ASSG_BITWISE_SRU
      identifier expression)
;
```

Statement kuralı, birçok seçeneği beraberinde sunmaktadır. CEZVE_STATEMENT_ASSG ile belirtilen atama işlemleri, CEZVE_STATEMENT_IF ve CEZVE_STATEMENT_WHILE ile belirtilen bileşik ifadeler, sınıf aktarım ifadesi olan CEZVE_STATEMENT_IMPORT ve doğrudan gösterim ifadesi çalıştırmak için CEZVE_STATEMENT_EXPRESSION sözcüklerini içerir.

Bu kural altında yalnızca atama işlevleri, çekirdek sistem ile iletişime geçer.

```
|   ^(CEZVE_STATEMENT_ASSG i = identifier e = expression)
    {assignmentManager.assign((String)i, e);}
```

Kural içerisinde tanımlanan i = identifier ve e = expression değerleri, bu kurallardan dönecek olan verilerin tutulacağı değişkenlerdir. Alınan değişkenler, assignmentManager atama yöneticisi sınıf örneğinin assign metoduna argüman

olarak gönderilir. assignmentManager ise belirteç değerini anahtar-değer ikilisi olarak variableTable hash tablosuna yazacaktır.

Diğer bileşik atama işlevlerinde ise assignmentManager sınıfının postAssign metodu çağırılır.

```
|  ^(CEZVE_STATEMENT_ASSG_ADDITION
     i = identifier e = expression)
   {assignmentManager.postAssign((String)i, e, "add");}
```

postAssign metodunda, assign metodundan farklı olarak bir parametre daha bulunur. Bu parametre, atama yapılmadan önce kullanılacak metodu göndermemizi sağlayacaktır. Cezve dil altyapısı, Python programlama dilinde kullanılan işleç aşırı yükleme sistemine sahiptir. Bu maksatla, işleçler metotlar ile ilişkilendirilmiştir.

```
1 + 5
```

gibi bir matematiksel işlem, aslında 1 sayısının add metodunun çağrılması şeklinde özetlenebilmektedir.

```
1.add(5);
```

Bu maksatla, CEZVE_STATEMENT_ASSG_ADDITION işlemi için argüman olarak add verisi gönderilmektedir. Öncelikle belirteç değişken tablosunda aranıp, daha sonra bulunan değerin add metodu yeni gösterim ifadesi (expression) argümanı ile çalıştırılacaktır. Atama işlemi, bu metot çağırma işleminden sonra yapılır.

5.1.3. `expressionList` Ağaç Ayrıştırıcısı Kuralı

```
expressionList returns [ArrayList ret]
@init {ret = new ArrayList();}
  :  ^(CEZVE_FRAGMENT_EXPRESSION_LIST
      (e = expression {ret.add(e);})+)
  |
  ;
```

expressionList kuralı, birçok gösterim ifadesinin dizi listesi şeklinde istendiği durumlarda kullanılır. Kural başlangıcında, bir ArrayList nesnesi oluşturulur. Ve her gösterim ifadesi bu listeye eklenip, sonuç olarak döndürülür.

5.1.4. `expression` Ağaç Ayrıştırıcısı Kuralı

Gösterim ifadelerini (expression) bir kaç grupta inceleyebiliriz.

ternaryOperation işlemi başlı başına bir grup olarak nitelendirilir. Parametre olarak, 3 adet expression gönderilebilir. Bunlardan ikisi seçime bağlıdır (opsiyonel).

```
:  ^(CEZVE_OPERATION_TERNARY
     e = expression (l = expression r = expression)?)
   {ret = expressionManager.ternaryOperation(e, l, r);}
```

Kural, expressionManager yönetici sınıf örneğinin ternaryOperation metodunu çağıracaktır.

Bir diğer gösterim ifadesi grubu da, çift parametre alan ifadelerdir. Bunlarda expressionManager yönetici sınıfının ilişkili metoduna gönderilir.

```
|  ^(CEZVE_OPERATION_ADDITION
     l = expression (r = expression)?)
```

```
        {ret = expressionManager.operation(l, "add", r);}
```

Tekil işlemlerde bu kural içerisinde yer alır. Bu işlemler, etkide bıraktığı nesne örneğini ve çağıracağı metot adını argüman olarak gönderir.

```
|   ^(CEZVE_OPERATION_NEGATIVE e = expression)
    {ret = expressionManager.unaryOperation(e, "negative");}
```

Tanımlanan bu işlemler dışında temel veri tipleri, değişken isimleri, metot adları, sınıf adları gibi bir çok belirteç ve gösterimi kapsayan ATOM alt seçeneği bulunur.

```
|   ^(CEZVE_FRAGMENT_ATOM
      l = literal {ret = l;}
      (
        t = atomTrailer[ret]
        {ret = atomManager.trailer(ret, t);}
       )*
    )
```

Bu atom seçeneği, bir literal ile başlayıp, sonra birçok alt metot ve ögeye erişim sağlayabilir. literal kuralından dönen veri, l değişkenine atanacaktır. Bu l değişkeni de expression kuralının dönüş değeri olan ret değişkenine atanır. Ret değeri, daha sonra takip eden atomTrailer döngüsüne argüman olarak verilir. atomTrailer kuralı ise t değişkenini döndürmektedir. Son olarak eldeki tüm değişkenler, atomManager yönetici sınıf örneğinin trailer metoduna gönderilir. Sonuç, expression kuralının dönen değeri olacaktır.

```
|   ^(CEZVE_FRAGMENT_ATOM
      a = anyIdentifier[0, null] {ret = a;}
      (
        t = atomTrailer[ret]
```

```
        {ret = atomManager.trailer(ret, t);}
      )*
  )
```

Bir diğer atom seçeneği ise bir belirteç ile başlamaktadır. Bu belirteç bir değişken, paket ismi, sınıf ismi, builtIn metot ismi veya bunlara ait bir alternatif isim olabilir.

```
| ^(CEZVE_FRAGMENT_ATOM
    m = methodOrConstructor {ret = m;}
    (
      t = atomTrailer[ret]
      {ret = atomManager.trailer(ret, t);}
    )*
  )
```

Metot, yapıcı metot veya bütünleşik metot ile başlayan bir diğer atom seçeneği, literal veya identifier ile başlayan diğer seçenekleri gibi, atomTrailer alt kuralı döngüsü ile devam eder.

```
| ^(CEZVE_FRAGMENT_ATOM CEZVE_OPERATION_PRE_INCREMENT
    a = anyIdentifier[1, "increase"]) {ret = a;}
| ^(CEZVE_FRAGMENT_ATOM CEZVE_OPERATION_PRE_DECREMENT
    a = anyIdentifier[1, "decrease"]) {ret = a;}
| ^(CEZVE_FRAGMENT_ATOM CEZVE_OPERATION_POST_INCREMENT
    a = anyIdentifier[2, "increase"]) {ret = a;}
| ^(CEZVE_FRAGMENT_ATOM CEZVE_OPERATION_POST_DECREMENT
    a = anyIdentifier[2, "decrease"]) {ret = a;}
```

Son olarak, nitelediği elemanın değerini arttırıp azaltan tekil işleçler de, **expression** kuralının bir alt seçeneğidir. Bunlar da kendilerini **anyIdentifier** kuralına 2 argümanı ile tanıtır ve dönen değer, **expression** kuralının dönecek değeri olur.

5.1.5. `atomTrailer` Ağaç Ayrıştırıcısı Kuralı

```
atomTrailer[Object previousObject] returns [Object ret]
options {backtrack = true;}
  :  f = trailingField   {ret = f;}
  |  m = trailingMethodOrConstructor[previousObject] {ret = m;}
  |  ^(CEZVE_FRAGMENT_SLICER start = slicerStart)
      {ret = atomManager.newSlicer(start);}
  |  ^(CEZVE_FRAGMENT_SLICER
       (start = slicerStart)?
       (stop  = slicerStop)
       (step  = slicerStep)?
     )
    {ret = atomManager.newSlicer(start, stop, step);}
  ;
```

atomTrailer kuralı, nesnelerin aitlik hiyerarşisini sağlayan bir kuraldır. Dizi erişimleri için, atomManager yönetici sınıfı newSlicer metodunu sağlar. Bu metot uygun argümanlara göre yeni bir Slicer nesnesi oluşturacaktır. Bir nesnenin müteakip ögesi; bir alan ise trailingField kuralı, eğer öge, bir metot ise trailingMethodOrConstructor kuralı çağırılacaktır.

5.1.6. `slicerStart` Ağaç Ayrıştırıcısı Kuralı

```
slicerStart returns [Object ret]
  :  ^(CEZVE_FRAGMENT_SLICER_START e = expression)   {ret = e;}
  ;
```

slicerStart, oluşturulan Slicer nesnesine gönderilecek olan başlangıç parametresini döndürecektir.

5.1.7. `slicerStop` Ağaç Ayrıştırıcısı Kuralı

```
slicerStop returns [Object ret]
  : ^(CEZVE_FRAGMENT_SLICER_STOP (e = expression)?) {ret = e;}
  ;
```

slicerStop, oluşturulan Slicer nesnesine gönderilecek olan sonlandırma parametresini döndürecektir.

5.1.8. `slicerStep` Ağaç Ayrıştırıcısı Kuralı

```
slicerStep returns [Object ret]
  :  ^(CEZVE_FRAGMENT_SLICER_STEP (e = expression)?) {ret = e;}
  ;
```

slicerSteop, oluşturulan Slicer nesnesine gönderilecek olan adım değerleri parametresini döndürecektir.

5.1.9. `trailingField` Ağaç Ayrıştırıcısı Kuralı

```
trailingField returns [cezve.core.Field ret]
  :  ^(CEZVE_FRAGMENT_FIELD id = identifier)
      {ret = new cezve.core.Field(id);}
  ;
```

Bir literal, field veya metot kuralını takiben bir alan geliyor ise, bu kural çalıştırılacaktır. atomTrailer tarafından çalıştırılan bu kural, yeni bir cezve.core.Field nesnesi döndürecektir.

5.1.10. `trailingMethodOrConstructor` Ağaç Ayrıştırıcısı Kuralı

```
trailingMethodOrConstructor[Object previousObject]
returns [Object ret]
  :  ^(CEZVE_FRAGMENT_METHOD id = identifier arg = arguments)
{ret = atomManager.trailingMethodOrConstructor(
  previousObject, id, arg
 );}
  ;
```

Bir literal, field veya metot kuralını takiben bir metot geliyor ise, bu kural çalıştırılacaktır. atomTrailer tarafından çalıştırılan bu kural,

atomManager.trailingMethodOrConstructor işlevinin sonucunu döndürecektir. Bu metot sonucu, bir java.lang.reflect.Method nesnesi olabileceği gibi, bir Constructor nesnesi veya bu nesnelerin dizeleri olabilir.

5.1.11. `methodOrConstructor` Ağaç Ayrıştırıcısı Kuralı

```
methodOrConstructor returns [Object ret]
  :  ^(CEZVE_FRAGMENT_METHOD
     name = identifier args = arguments)
     {ret = atomManager.methodOrConstructor(name, args);}
  ;
```

Bu kural bir atom ifadesi, metot, yapıcı metot veya bütünleşik metot ile başlıyor ise çağırılmaktadır. identifier ve arguments kurallarından gelen argümanlar, atomManager yönetici sınıf örneğinin methodOrConstructor metoduna aktarılır.

5.1.12. `anyIdentifier` Ağaç Ayrıştırıcısı Kuralı

```
anyIdentifier[int methodInvokationType, String methodName]
returns [Object ret]
  :  ^(CEZVE_FRAGMENT_FIELD variableName = identifier)
     {
       ret = atomManager.anyIdentifier(variableName);

       if (methodInvokationType != 0) {// not normal invokation

        Object methodResult = (methodName == null)
          ? ret
          : expressionManager.operation(ret, methodName, null);

        if (methodInvokationType == 1)
          ret = methodResult;

        assignmentManager.assign(variableName, methodResult);
        // pre invokation + post invokation
       }
     }
  ;
```

Bu kural, bir atom ifadesi, alan, değişken, paket ismi, sınıf ismi veya bunlara ait alternatif isimler ile başlıyor ise çağırılmaktadır. identifier kuralından gelen argüman, atomManager yönetici sınıf örneğinin anyIdentifier metoduna aktarılır.

Bunlara ilaveten, bu kurala aktarılan argümanlar ile arttırma ve azaltma tekil işleç işlemleri devreye alınır.

5.1.13. `arguments` Ağaç Ayrıştırıcısı Kuralı

```
arguments returns [ArrayList ret]
  :  ^(CEZVE_FRAGMENT_METHOD_ARGUMENTS  e = expressionList)
{ret = e;}
  ;
```

arguments kuralı, bir metot, yapıcı metot veya bütünleşik metot tarafından çağrılır. arguments kuralı, temelde expressionList kuralı ile aynı yapıdadır. Ve dönen sonuçta, argümanları barındıran bir ArrayList olacaktır.

5.1.14. `literal` Ağaç Ayrıştırıcısı Kuralı

```
literal returns [Object ret]
  :  tkn = CEZVE_LITERAL_NULL
  |  tkn = CEZVE_LITERAL_BOOLEAN
     {ret = typeManager.newBoolean(tkn.getText());}
  |  tkn = CEZVE_LITERAL_BYTE
     {ret = typeManager.newByte(tkn.getText());}
  |  tkn = CEZVE_LITERAL_CHARACTER
     {ret = typeManager.newCharacter(tkn.getText());}
  |  tkn = CEZVE_LITERAL_LONG
     {ret = typeManager.newLong(tkn.getText());}
  |  tkn = CEZVE_LITERAL_SHORT
     {ret = typeManager.newShort(tkn.getText());}
  |  tkn = CEZVE_LITERAL_STRING
     {ret = typeManager.newString(tkn.getText());}
```

```
|  ^(CEZVE_LITERAL_FLOAT a = CEZVE_LITERAL_NUMBER)
    {ret = typeManager.newFloat(a.getText() + ".0");}
|  ^(CEZVE_LITERAL_FLOAT a = CEZVE_LITERAL_NUMBER
     b = CEZVE_LITERAL_NUMBER)
    {ret = typeManager.newFloat(a.getText()
     + "." + b.getText());}
|  ^(CEZVE_LITERAL_DOUBLE   a = CEZVE_LITERAL_NUMBER)
    {ret = typeManager.newDouble(a.getText() + ".0");}
|  ^(CEZVE_LITERAL_DOUBLE   a = CEZVE_LITERAL_NUMBER
     b = CEZVE_LITERAL_NUMBER)
    {ret = typeManager.newFloat(a.getText() + "." +
     b.getText());}
    {ret = typeManager.newDouble(a.getText() + "." +
     (b != null ? b.getText() : "0"));}
|  ^(CEZVE_LITERAL_INTEGER  a = CEZVE_LITERAL_NUMBER)
    {ret = typeManager.newInteger(a.getText());}
|  ^(CEZVE_LITERAL_ARRAY_LIST  ae = arrayElements)
    {ret = typeManager.newArrayList(ae);}
|  ^(CEZVE_LITERAL_ARRAY       el = expressionList)
    {ret = typeManager.newArray(el);}
|  ^(CEZVE_LITERAL_HASHTABLE   pl = pairList)
    {ret = typeManager.newHashtable(pl);}
;
```

Cezve dil altyapısı, imla dosyaları tarafından doğrudan bir çok veri tipinin örneklendirilmesine izin verir. Bu işlemleri yapan yönetici sınıf ise, TypeManager sınıfıdır. Gelen argümanlar doğrudan ilişkili metoda yönlendirilir.

```
|  tkn = CEZVE_LITERAL_STRING
   {ret = typeManager.newString(tkn.getText());}
```

Örnek olarak; bir karakter katarı oluşturmak için gönderilen argüman alınıp, typeManager sınıfının newString metoduna gönderilir. Sonuç, yeni bir java.lang.String nesnesi olacaktır.

5.1.15. `identifier` Ağaç Ayrıştırıcısı Kuralı

```
identifier returns [String ret]
  :  id = CEZVE_ATOM_IDENTIFIER {ret = id.getText();}
  |  id = CEZVE_ATOM_NAME       {ret = id.getText();}
  ;
```

Belirteçler, sınıf isimleri, paket isimleri, değişken, metot ve yapıcı isimleri gibi hemen hemen tüm isimlendirme gereken yerlerde kullanılmaktadır. identifier kuralı, bu isimleri gelen sözcüklerden ayrıştırıp döndürür.

5.1.16. `arrayElements` Ağaç Ayrıştırıcısı Kuralı

```
arrayElements returns [ArrayList ret]
@init {ret = new ArrayList();}
  :  (e = expression {ret.add(e);})*
  ;
```

literal kuralı ile tanımlanan dizilere ilişkin elemanlar başka bir alt kural olan arrayElements kuralı çağırılarak oluşturulur. Bu kural, her gösterim (expression) çevriminde diziye yeni bir eleman ekler ve sonuç olarak, tüm dizi elemanlarını bir ArrayList içerisinde talep eden üst kurala aktarır.

5.1.17. `pairList` Ağaç Ayrıştırıcısı Kuralı

```
pairList returns [Hashtable ret]
@init {ret = new Hashtable();}
  :  (p = pair {ret.put(p.get(0), p.get(1));})*
  ;
```

pairList kuralı, sözlük/Hash tablosu veri tipi için gerekli olan eşleşme listesini sağlamaktadır. Bu eşleşmeler, bir java.util.Hashtable nesnesi olarak döndürülecektir.

5.1.18. `pair` Ağaç Ayrıştırıcısı Kuralı

```
pair returns [ArrayList returnValue]
  :  ^(CEZVE_FRAGMENT_HASHTABLE_PAIR
     leftExpression = expression rightExpression = expression)
```

```
      {returnValue = typeManager.newPair(
    leftExpression, rightExpression);}
  ;
```

pair kuralı, pairList tarafından talep edilen eşleşmeleri iki elemanlı bir ArrayList nesnesi olarak döndürmektedir. Bu işlemi, typeManager yönetici sınıf örneğinin newPair metodu gerçekleştirir.

5.1.19. `pack` Ağaç Ayrıştırıcısı Kuralı

```
pack
@init {ArrayList classList = new ArrayList();}
  :  ^(CEZVE_FRAGMENT_PACK_SINGLE (n = CEZVE_ATOM_NAME
{classList.add(n.getText());})+ (a = packAlias)?)
     {statementManager.importClass(classList, a);}
  |  ^(CEZVE_FRAGMENT_PACK_ALL    (n = CEZVE_ATOM_NAME
{classList.add(n.getText());})+)
     {statementManager.importClasses(classList);}
  ;
```

pack kuralı, import ifadesi tarafından doğrudan talep edilen paket isimlerini döndürür. Bu işlemi yaparken statementManager sınıfının importClass ve importClasses metodlarını kullanır.

5.1.20. `packAlias` Ağaç Ayrıştırıcısı Kuralı

```
packAlias returns [String ret]
  :  ^(CEZVE_FRAGMENT_PACK_ALIAS id = identifier) {ret = id;}
  ;
```

packAlias kuralı, paketler için gerekli olabilecek takma isimleri String nesnesi olarak pack kuralına iletir.

5.1.21. `ifSubStatement` Ağaç Ayrıştırıcısı Kuralı

```
ifSubStatement
  :  ^(CEZVE_SUB_STATEMENT_IF e = expression
       {if (!((Boolean)e).booleanValue())
         consumeUntil(input, CEZVE_SUB_STATEMENT_IF_END);}
       statement* CEZVE_SUB_STATEMENT_IF_END
     )
     {if (((Boolean)e).booleanValue())
       consumeUntil(input, CEZVE_STATEMENT_IF_END);}
  ;
```

if bileşik ifadesi, iki alt ifadeden oluşur. Bileşik ifadeler diğer ağaç ayrıştırma kurallarından farklı olarak sonlandırma sözcüklerine sahiptir. ifSubStatement kuralına gönderilen expression parametresi, eğer boolean true değeri değil ise, sonlandırma sözcüğüne kadar olan tüm sözcükler işlenmeden geçilir.

5.1.22. `elseSubStatement` Ağaç Ayrıştırıcısı Kuralı

```
elseSubStatement
  :  ^(CEZVE_SUB_STATEMENT_ELSE
       statement* CEZVE_SUB_STATEMENT_ELSE_END)
  ;
```

if bileşik ifadesinin ikinci alt kuralı, elseSubStatement kuralıdır. Eğer ifSubStatement kuralında bulunan test gösterim ifadesi yanlış ise bu kural çağırılacaktır.

5.1.23. `whileStatement` Ağaç Ayrıştırıcısı Kuralı

```
whileStatement
@init {
  int      bookmark  = input.index();
  boolean  condition = false;
}
  :  e = expression
     {
```

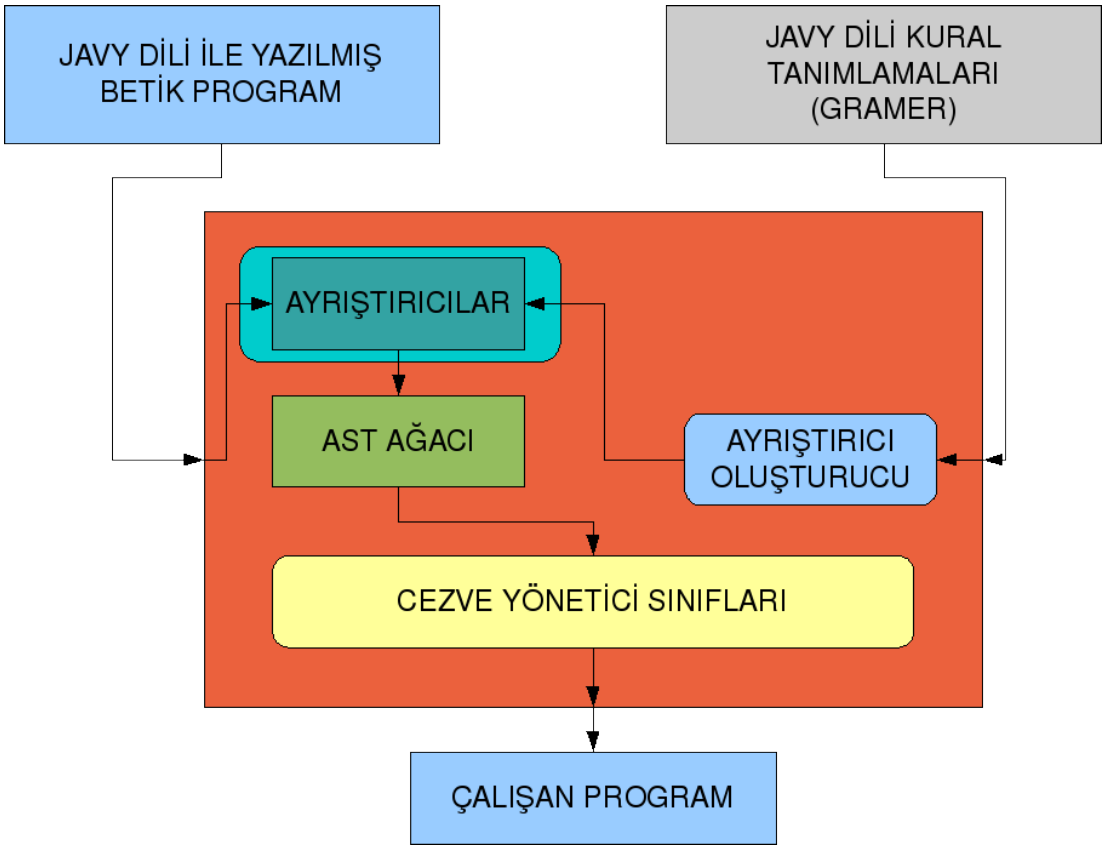
```
      condition = ((Boolean)e).booleanValue();
      if (!condition) consumeUntil(input,
CEZVE_STATEMENT_WHILE_END);
    }
    statement* CEZVE_STATEMENT_WHILE_END
    {
      if (condition) {
        input.rewind(bookmark);
        whileStatement();
      }
    }
  ;
```

while bileşik ifadesi e, expression argumanı tarafından sağlanan değerin true olması durumunda, içerdiği kuralları tekrar ve tekrar işler. Döngüden çıkılması için e ifadesinin boolean false değeri alması gerekir.

6. DİL ÖZELLEŞTİRİLMESİ

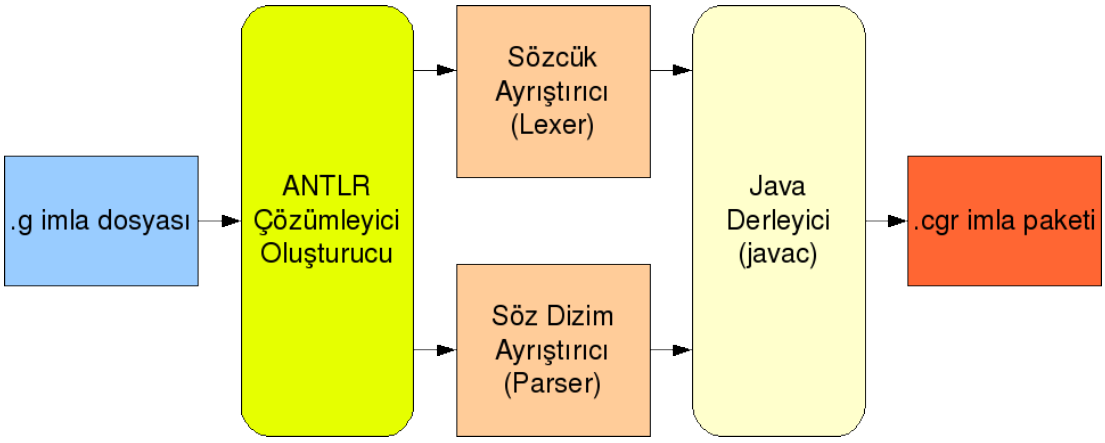
Cezve dil altyapısının sunduğu en önemli özellik özelleştirilebilme sistemidir. Özelleştirilebilme kavramı altında Cezve, dil imlasının kolayca değiştirilebilmesini ve işleç aşırı yükleme ve tip denetimi paradigmalarının seçimli olarak kullanılmasını sağlamaktadır.

Javy dilinin temel çalışma sistematiği Şekil 101'deki gibidir. Kural tanımlamaları ile çalışma zamanında ayrıştırıcılar oluşturulur ve ayrıştırıcıya ait dil ile yazılan betik programlar kolayca işlenip sonuç bütünleşik ağaç ayrıştırıcısına gönderilir. Kural tanımlamaları .g ANTLR gramer dosyası ile yapılır. Bu standart bir ANTLR v3 imla dosyasıdır. Tek farkı CEZVE anlamsal sözcüklerinin yeniden yazılma (rewrite) kuralları içerisinde kullanılmış olmasıdır.



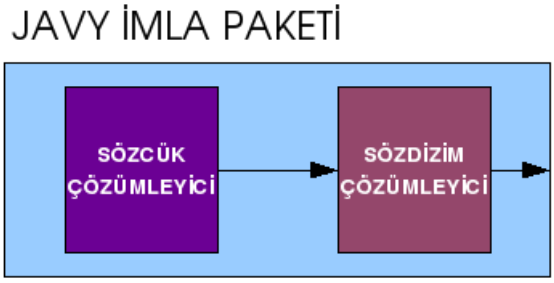
Şekil 101: Dil Tanımlaması ve Tanımlanan Dilin Kullanımı

Cezve dil altyapısı .g imla dosyası ile tanımlanmış dosyalardan ayrıştırıcı sınıfları oluşturur. Ayrıştırıcı tanımlamaları ANTLR kütüphanesi ile kaynak kodlara dönüştürülür. Cezve, çalışma zamanında Java derleyicisini kullanarak kaynak kodlardan sınıflar meydana getirir. Daha sonra bu sınıfların çalışma zamanında dil altyapısı tarafından kolayca yüklenebilmesi için özelleştirilmiş java arşivi oluşturulup sınıf dosyaları arşiv içine eklenir. Bu özelleştirilmiş java arşivleri veya bir başka değişle CGR (Cezve Grammar) paketleri Cezve Dil Altyapısının kolayca imla değiştirebilmesini sağlar.



Şekil 102:CGR Paketi oluşturma

İmla paketlerinin içerisinde Sözcük çözümleyici ve Söz dizim çözümleyici bulunur. Bunlar bir betik programı karşılayan ilk ayrıştırıcılardır. Dil Altyapısının özelleştirilebilir kısmı olan imla paketleri sonuç olarak ağaç ayrıştırıcı sistemine parametreler gönderir. Bu parametreler ise dil altyapısı için standartlaştırılmış anlamsal sözcüklerdir.



Şekil 103: Örnek İmla Paketi

6.1. Kabuk (Shell) Kullanılarak Özelleştirme

Cezve dil altypısı 3 farklı çalışma kipi sunar:

1. Dil Altyapısına gönderilen dosya adı parametresi
2. Dil Altyapısına gönderilen betik içeriği parametresi
3. Dil Altyapısı tarafından sağlanılan kabuk

Dil altyapısı hem bir tanı kipi olarak hem de kolayca denetim ve test için her özelleştirilebilir dile göre bir kabuk sunar.Öntanımlı özelleştirilebilir dil Javy olduğu için kabuk başka bir dil tanımlanmamış ise bu dil ile açılır ve buna uygun bir kabuk işaretçisi (prompt) görüntüler.

```
javy>
```

Bir cgr paketi oluşturmak için yaptığımız .g imla dosyasını kabuk ile çağırmamız yeterlidir. **:gr** veya **:grammar** kabuk komutları ile bir imla dosyası yolu gösterebiliriz.

```
javy> :gr yeniDil /tmp/Javy.g
GRAMMAR GENERATED!
```

Artık CGR paketimiz oluşturulup Cezve'nin yerel dil havuzuna eklenmiştir. Artık yapmamız gereken **yeniDil** adını verdiğimiz bu dili **:sg** veya **:set** ile etkin imla olarak tanımlamaktır.

```
javy> :sg yeniDil
ACTIVE GRAMMAR : yeniDil
```

Artık bu dil ile yazılan kodlar çalıştırabilir. Kabuk üzerinden girilen kodların çalıştırılması için ise **:evaluate**, **:ev** veya **:run** komutlarından herhangi biri kullanılabilir.

7. JAVY PROGRAM ÖRNEKLERİ

Cezve dil altyapısı için yapılan konsept programlama dili Javy dil altyapısı ile yapılabilecekler için bir örnek teşkil etmektedir. Temelde Java kütüphanelerini kullandığından Java ile yapılan uygulamalar dil altyapısının el verdiği ölçüde Javy dili ile de yapılmaktadır.

7.1. Merhaba Dünya

Javy ile standart girdi çıktı işlemleri Java'ya nazaran oldukça kolaydır.

```
println("Merhaba Dünya");
```

Javy bu tarz bütünleşik metodlarının yanında Java'nın standart yaklaşımlarını da kullanabilir.

```
System.out.println("Merhaba Dünya");
```

Ya da daha uzun ifadesi ile

```
java.lang.System.out.println("Merhaba Dünya");
```

7.2. Uçbirim uygulama Örneği

Dil altyapsı standart kabuğu gibi uç birimde çalışacak uygulamalar Javy kullanılarak kolaylıkla yazılabilir.1'den 10'a kadar olan sayıları toplayan Javy ile yazılmış örnek bir uç birim uygulaması şu şekildedir.

```
a = true;

if (a) {

  sayac  = 1;
  toplam = 0;

  while (sayac < 10) {
    toplam += sayac;
    sayac++;
  }

  println(toplam);
}
```

Program bir while döngüsü içerisinde sayac değerlerini toplam değişkenine eklemetedir. Ek olarak örnek bir if ifadesi de programa eklenmiştir.

7.3. KGA Tabanlı Uygulama

Javy, uçbirim uygulamalarının yanında Kullanıcı Grafik Arabirimi (KGA), içeren programların gerçekleştirilmesine de olanak tanır. Aşağıdaki program örnek bir pencere uygulaması yapmaktadır.

```
import javax.swing.*;
import java.awt.Color;
import java.awt.Font;

pencere = JFrame("merhaba");
pencere.setLocation(100, 100);
pencere.setSize(300, 300);
pencere.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

panel = JPanel();
panel.setBackground(Color.RED);
pencere.add(panel);

etiket = JLabel("merhaba Javy");
etiket.setForeground(Color.YELLOW);
etiket.setFont(etiket.getFont().deriveFont(20));
panel.add(etiket);
```

```
pencere.setVisible(true);
```

Programın çıktısı şu şekildedir:

Şekil 104: Örnek KGA Programı

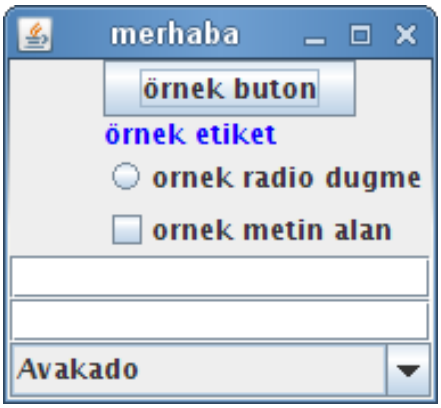
7.4. Swing Nesneleri Kullanımı

Javy ile Java'nın modern GUI kütüphanesi olan Swing'in tüm öğeleri kolayca kullanılabilir. Yaklaşım Java'da kullanılan ile örtüşmektedir. Örnek Java nesnelerinin kullanımı şu şekildedir.

```
import javax.swing.*;
import java.awt.Color;
import java.awt.Font;

pencere = JFrame("merhaba");
pencere.setLocation(100, 100);
pencere.setSize(300, 300);
pencere.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
yerlesim = BoxLayout(pencere.getContentPane(),
BoxLayout.Y_AXIS);
pencere.getContentPane().setLayout(yerlesim);
```

```
dugme = JButton("örnek buton");
pencere.add(dugme);

etiket = JLabel("örnek etiket");
etiket.setForeground(Color.BLUE);
pencere.add(etiket);

radioDugme = JRadioButton();
radioDugme.setText("ornek radio dugme");
pencere.add(radioDugme);

denetimKutusu = JCheckBox();
denetimKutusu.setText("ornek denetim kutusu");
pencere.add(denetimKutusu);

metinAlan = JTextField();
denetimKutusu.setText("ornek metin alan");
pencere.add(metinAlan);

parolaAlani = JPasswordField();
pencere.add(parolaAlani);

kombo = JComboBox();
kombo.setModel(DefaultComboBoxModel(["Avakado", "Armut",
"Muz"]));
pencere.add(kombo);

pencere.pack();

pencere.setVisible(true);
```

Programın çıktısı Şekil 105'te olduğu gibidir:



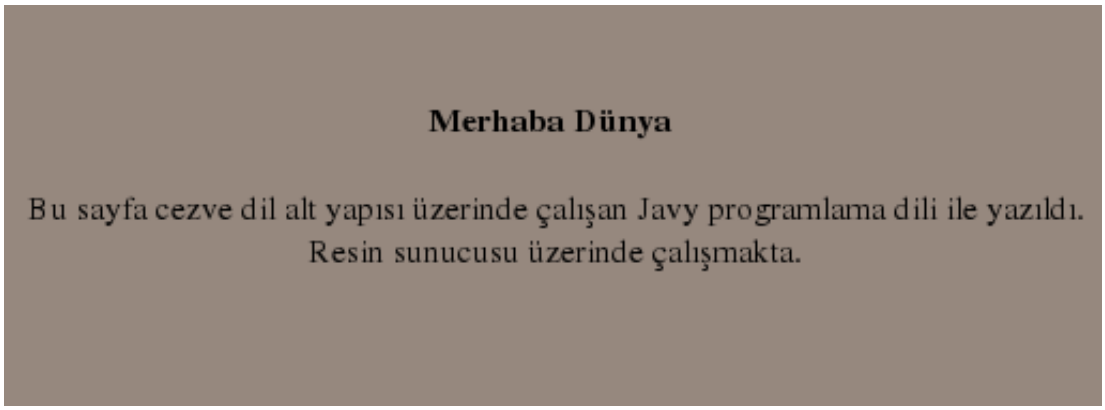
Şekil 105: Örnek Swing GUI kullanımı

7.5. Web Tabanlı Uygulama Örneği

Cezve dil altyapısı bütünleşik Servlet Container'ı sayesinde web tabanlı uygulamalar içinde bir platform oluşturmaktadır. Aşağıda biasit bir HTML çıktısını ekrana yazan .javy dosyası bulunmaktadır.

```
println(
"""
<html>
 <head>
  <title>Merhaba Javy</title>
 </head>
 <body bgcolor="#96887E" color="#635247">
  <div align="center">
   <br /><br /><br />
   <b>Merhaba Dünya</b>
   <br /><br />
   Bu sayfa cezve dil altyapısı üzerinde çalışan Javy
programlama dili ile yazıldı.<br />
   Resin sunucusu üzerinde çalışmakta.
  </div>
 </body>
</html>
"""
);
```

println metodu uç birim örneğinde standar çıktı aygıtı olan konsol ile etkileşim kurarken şimdiki çıktı aygıtı ise bir http cevap nesnesidir. Bu ayarlamayı yapan Cezve'nin bütünleşik ServletContainer'ıdır. Örnek çıktı ise Şekil 106'da olduğu gibidir.



Şekil 106: Örnek Web Tabanlı Uygulama Çıktısı

8. SONUÇ

Yapılan bu çalışmada yalnızca bir adet imla dosyası kullanılarak özelleştirilmiş programlama dilleri yapmaya olanak sağlayan bir dil altyapısı geliştirilmiştir. Geliştirilen bu dil altyapısı farklı programlama dili geçmişine sahip programcıların kendi programlama dillerini oluşturarak aynı projede çalışmalarına imkan tanımaktadır. Yine bu dil altyapısı üzerinde çalışan Javy adında kavramsal bir programlama dili yapılmıştır.

Cezve Dil Altyapısı için yapılan bu kavramsal dil, birçok üst düzey programlama dilinin programcılar tarafından beğenilen özellikleri kullanılarak yapılmıştır. Oluşturulan dil, Java Sanal Makinesi üzerinde çalışmasına rağmen, Java dilinin desteklemediği işleç aşırı yükleme (Operator Overloading) gibi özellikleri de bünyesinde barındırmaktadır.

Bunu yaparken Java'nın varsayılan çalışma zamanı kütüphanesini değiştirmektedir. Yaptığımız testlerde Çok Kanallı (Multi Thread) programlamada uyarlanan yeni veri tiplerinin güvenle çalıştığı görülmüştür. Fakat GPL lisansı gereği yazılım, bu konuda bir garanti vermez. Çalışmanın ele aldındığı sürüm Programlama Dilinin test öncesi (Pre Alpha) sürümüdür.

Oluşturulan programlama dili, Java, JavaScript, Python, Ruby, Perl gibi dillerin bazı özelliklerini almıştır. İmla bakımından en çok JavaScript diline benzemektedir.

Oluşturulan dil ve dil altyapısı açık kaynak öğeler içerdiğinden içerdiği öğelerin lisansı nedeni ile projenin kendisi de GPL lisansına sahip olmak durumundadır. Projenin benzer çalışmalara ön ayak olması için biz de bu yaklaşımı isteyerek benimsemekteyiz.

Yapılan çalışmadaki programlama dili ve dil altyapısı, masaüstü, uç birim tabanlı ve web tabanlı programlar yapılmasına imkân tanımaktadır. Fakat kurumsal çaplı programlar geliştirmek için halen kararlı bir sürüme ve hafıza yönetimine sahip değildir.

Dil altyapısı üzerinde çalışan programlama dili, yapılan testlerde Java programlama dilinden yaklaşık 20 kat daha yavaş çalışmaktadır. Bunun temel nedeni dil altyapısının özelleştirilebilir bir yapı sunmasıdır.

Özelleştirilebilir yapıya ek olarak birçok sözcükten oluşmuş ve imla tahminleri kullanan bir ağaç ayrıştırıcı da önemli negatif performans etmenlerindendir.

Proje temelde ANTLR çözümleyici oluşturucu, ANT inşa sistemi, JLine konsol sistemini kullanmaktadır. Bunlara ilaveten sistem Servlet barındırdığından J2EE kütüphanesine de bağımlıdır.

sourceforge.net açık kaynak proje yönetim sistemi, barınma, sürümlendirme, sorun yönetimi, konusunda yüz binlerce açık kaynak projeye destekleyici olduğu gibi bu çalışmaya da destek vermektedir.

8.1. Gelecek Çalışmalar

Proje neticesinde oluşan dil altyapısı ANTLR ayrıştırıcı oluşturucu yazılımına doğrudan bağımlıdır. İleride bu bağımlılık ortadan kaldırılarak dil altyapısı için özel olarak tasarlanmış bir devingen ayrıştırıcı oluşturucu eklenecektir. Bu hem performans artımını hem de çalışma zamanında dil özelleştirilmesini sağlayacaktır.

Ek olarak dil altyapsısı ve yorumlayıcı alt sistemine çok kanallı (multithread) imla işleme özelliği kazandırılarak CPU yükünün dengeli bir şekilde kullanımı sağlanacaktır. Bu sayede altyapı çok daha hızlı imla işleyip sonuç oluşturabilecektir.

Projenin temel performans eksikliğini ise çekirdek kısmının CNI veya JNI kütüphaneleri ile yazılarak yapılması hedeflenmektedir. Ortaya çıkacak derlemiş kod

işletim sistemi tarafından doğrudan anlaşılabilir temel kod (native code) niteliğinde olacaktır. Böylece proje doğrudan Java Sanal Makinesine ihtiyaç duymayacaktır.

Ayrıca yakın gelecekte Cezve tarafından çalışma zamanında Java Sanal Makinesine yapılan müdahalenin seçimli hale gelmesi hedeflenmektedir. Bu sayede sınırlandırılmış platformlarda da dil altyapısı kolayca çalıştırılabilecektir.

Dil altyapısı ile daha esnek programlama dili özelleştirilebilme yelpazesi sağlanması için ağaç ayrıştırıcıya aktarılan parametrelerin ilkel tip veya çok az bellek tüketen, optimizasyonu yapılmış nesneler olması hedeflenmektedir. Buna ilaveten dil altyapısının desteklediği sözcük yelpazesi de genişletilerek daha esnek bir yapıya dönüştürülmesi hedeflenmektedir.

Mevcut Javy sürümünün imla eksiklikleri tamamlanıp kurumsal seviyede bir dil olması için Java Community Process standardının kullanılması ve Java 1.6 sürümünün getirdiği scripting yaklaşımı kullanılması düşünülmektedir. Bu sayede dil ve dilaltyapısının küresel standartlar ile uyumu sağlanacaktır.

KAYNAKÇA

1: O'reilly Corp., *O'reilly Language Poster*,

www.oreilly.com/news/graphics/prog_lang_poster.pdf, 2008

2: Robert W. Sebesta, *Concept of Programming Languages*, 2005

3: Apple Corp., *APPLE WORLDWIDE DEVELOPPER CONFERENCE*, 2006

4: Fırat Küçük, *Javy Programlama Dili ve Cezve Dil Altyapısı*, http://www.javy.org/,

2008

5: Netscape Communication Coprporation, *JavaScript Programlama Dili*,

http://developer.mozilla.org/en/docs/JavaScript, 2008

6: Guido Van Rossum, *Python Programlama Dili*, http://www.python.org/, 2008

7: Larry Wall, *Perl Programlama Dili*, http://www.perl.org/, 2008

8: Rasmus Lerdorf, *PHP Programlama Dili*, http://www.php.net/, 2008

9: Yukihiro “matz” Matsumoto, *Ruby Programlama Dili*, http://www.ruby-lang.org/,

2008

10: Anonim, *Zuse*, http://en.wikipedia.org/wiki/Zuse, 2008

11: Anonim, *Plankalkül*, http://en.wikipedia.org/wiki/Plankalkül, 2007

12: Herbert SCHILDT, *C++ Temel Öğrenim Kılavuzu*, 1999

13: Anonim, *SIMULA Programming Language*, http://en.wikipedia.org/wiki/Simula,

14: Anonim, *ALGOL*, http://en.wikipedia.org/wiki/ALGOL, 2007

15: Anonim, *ADA Programlama Dili*,

http://tr.wikipedia.org/wiki/Ada_programlama_dili, 2008

16: Anonim, *VHDL*, http://en.wikipedia.org/wiki/VHDL, 2008

17: José de Oliveira Guimarães, *The Green language*, 2005

18: Brian A. Malloy, James F. Power, John T. Waldron, *Applying Software*

Engineering Techniques to Parser Design: C# Parser, 2002

19: Dick Grune, Ceriel Jacobs, *Parsing Techniques - A Practical Guide*, 1998

20: John E. Hopcroft, Rajeev Motwani, Jefrey D. Ullman, *Introduction to Automata*

Theory, Languages and Computation, 2001

21: International Standarts Organisation, ISO/IEC 14977 Extended BNF, 1996

22: Kenneth Slonneger, Barry L. Kurtz, *Formal Syntax and Semantics of Programming Languages*, 1995

23: Terrence Parr, *ANTLR Parser Generator*, http://www.antlr.org/, 2008

24: James Gosling, Bill Joy, Guy Steele, Gilad Bracha, The JavaTMLanguage Specification, 2005

25: Sun Microsystems, Inc., JSR 202: JavaTM Class File Specification Update, 2006

26: Anonim, *Java ByteCode*, http://en.wikipedia.org/wiki/Java ByteCode, 2007

27: Guillaume Laforge, *Groovy Programlama Dili*, http://groovy.codehaus.org/, 2008

28: Jim Hugunin, Barry Warsaw, Samuele Pedroni, *Jython Programlama Dili*, http://www.jython.org/, 2008

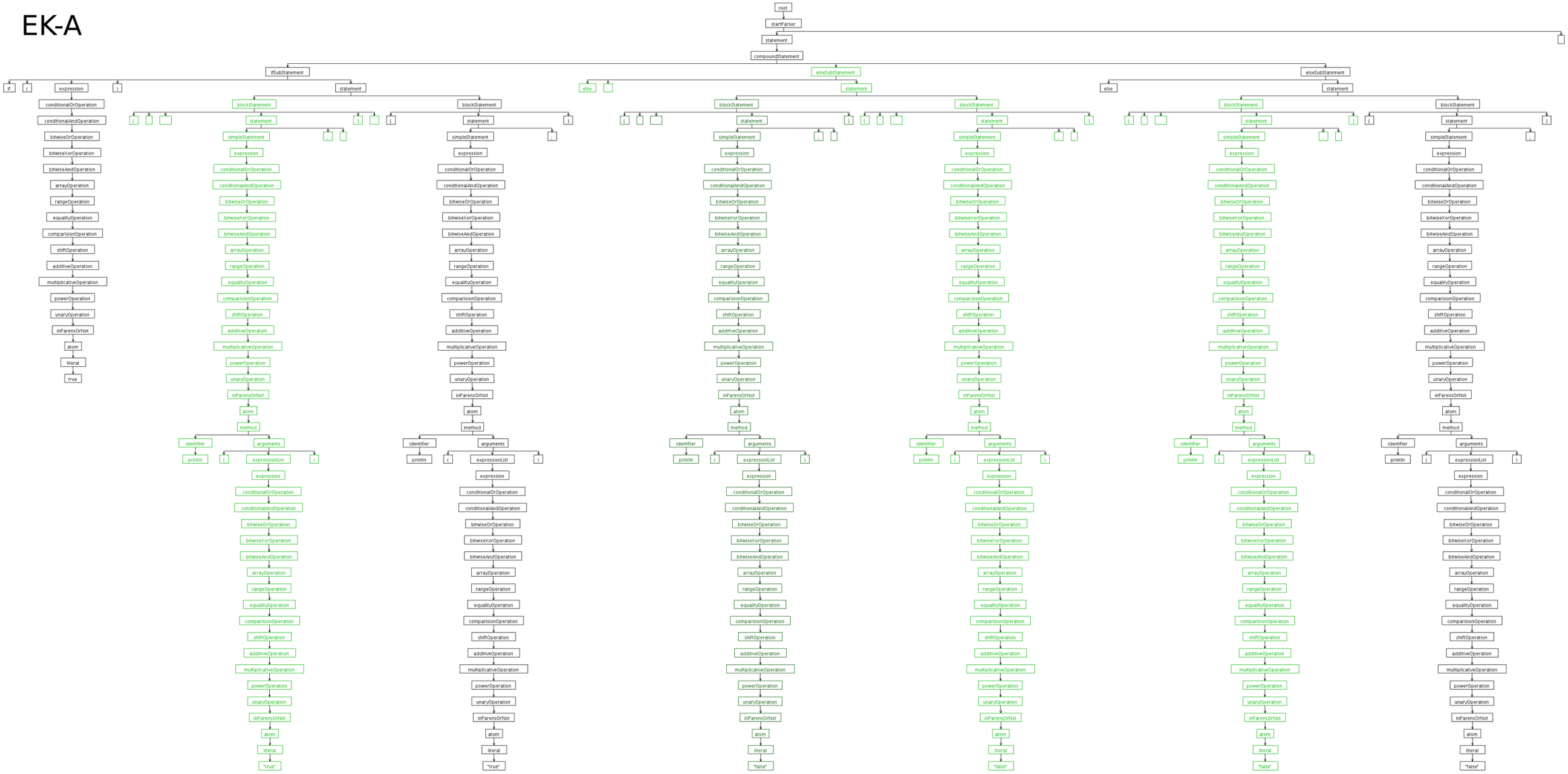
29: Patrick Niemeyer, *BeanShell Programlama Dili*, http://www.beanshell.org/, 2008

30: Charles Nutter, Thomas Enebo, Ola Bini, Nick Sieger, *JRuby Programlama Dili*, http://jruby.codehaus.org/,

31: LiveLogix, *Logix Programlama Dili*, http://www.livelogix.net/logix/, 2008

32: Uwe Zdun, *Tailorable language for behavioral composition and configuration of sofware*, 2005

EK-A AYRIŞTIRMA AĞACI



EK-B JAVY SÖZCÜK VE SÖZDİZİM ÇÖZÜMLEYİCİ KURALLARI

```
grammar Javy;

options {
  output       = AST;
  ASTLabelType = CommonTree;
  tokenVocab   = Cezve;
  backtrack    = false;
  k            = 2;
}



// ### [TR] SOZDIZIM COZUMLEYICI KURALLARI ###########################################################
// ### [EN] PARSER RULES #############################################################################
// ###################################################################################################



// ### startParser ###################################################################################

startParser
  :  statement* EOF -> ^(CEZVE_SCRIPT statement*)
  ;



// ### statement #####################################################################################

statement
options {backtrack = true;}
  :  blockStatement
  |  compoundStatement
  |  simpleStatement? SEMICOLON!
  ;



// ### blockStatement ################################################################################

blockStatement
  :  LEFT_CURLY_BRACKET! statement+ RIGHT_CURLY_BRACKET!
  ;



// ### compoundStatement #############################################################################

compoundStatement
  :  ifSubStatement (options {backtrack = true;} : elseSubStatement)?
     -> ^(CEZVE_STATEMENT_IF ifSubStatement elseSubStatement? CEZVE_STATEMENT_IF_END)
  |  whileStatement
  ;



// ### ifSubStatement ################################################################################

ifSubStatement
  :  IF LEFT_PARENTHESIS expression RIGHT_PARENTHESIS statement
     -> ^(CEZVE_SUB_STATEMENT_IF expression statement CEZVE_SUB_STATEMENT_IF_END)
  ;



// ### elseSubStatement ##############################################################################

elseSubStatement
  :  ELSE statement -> ^(CEZVE_SUB_STATEMENT_ELSE statement CEZVE_SUB_STATEMENT_ELSE_END)
  ;
```

```
// ### whileStatement ###############################################################################

whileStatement
  :  WHILE LEFT_PARENTHESIS expression RIGHT_PARENTHESIS statement
     -> ^(CEZVE_STATEMENT_WHILE expression statement CEZVE_STATEMENT_WHILE_END)
  ;


// ### simpleStatement ##############################################################################

simpleStatement
  :  IMPORT  pack (COMMA pack)* -> ^(CEZVE_STATEMENT_IMPORT      pack+)
  |  expression                 -> ^(CEZVE_STATEMENT_EXPRESSION  expression)
  |  identifier assignmentOperator^ expression
  ;


// ### assignmentOperator ###########################################################################

assignmentOperator
  :  EQUALS                                -> CEZVE_STATEMENT_ASSG
  |  PLUS_AND_EQUALS                       -> CEZVE_STATEMENT_ASSG_ADDITION
  |  MINUS_AND_EQUALS                      -> CEZVE_STATEMENT_ASSG_SUBTRACTION
  |  ASTERISK_AND_EQUALS                   -> CEZVE_STATEMENT_ASSG_MULTIPLICATION
  |  SLASH_AND_EQUALS                      -> CEZVE_STATEMENT_ASSG_DIVISION
  |  BACKSLASH_AND_EQUALS                  -> CEZVE_STATEMENT_ASSG_T_DIVISION
  |  DOUBLE_ASTERISK_AND_EQUALS            -> CEZVE_STATEMENT_ASSG_POWER
  |  PERCENT_AND_EQUALS                    -> CEZVE_STATEMENT_ASSG_MODULO
  |  UNDERSCORE_AND_EQUALS                 -> CEZVE_STATEMENT_ASSG_CONCATENATION
  |  AMPERSAND_AND_EQUALS                  -> CEZVE_STATEMENT_ASSG_AND
  |  PIPE_AND_EQUALS                       -> CEZVE_STATEMENT_ASSG_OR
  |  CARET_AND_EQUALS                      -> CEZVE_STATEMENT_ASSG_XOR
  |  DOUBLE_LEFT_ANGLE_BRACKET_AND_EQUALS  -> CEZVE_STATEMENT_ASSG_BITWISE_SL
  |  DOUBLE_RIGHT_ANGLE_BRACKET_AND_EQUALS -> CEZVE_STATEMENT_ASSG_BITWISE_SRS
  |  TRIPLE_RIGHT_ANGLE_BRACKET_AND_EQUALS -> CEZVE_STATEMENT_ASSG_BITWISE_SRU
  ;



// ### expressionList ###############################################################################

expressionList
  :  expression (COMMA expression)* COMMA? -> ^(CEZVE_FRAGMENT_EXPRESSION_LIST expression+)
  ;



// ### expression ###################################################################################

expression
  :  conditionalOrOperation (QUESTION_MARK expression COLON expression)?
     -> ^(CEZVE_OPERATION_TERNARY conditionalOrOperation (expression expression)?)
  ;



// ### conditionalOrOperation #######################################################################

conditionalOrOperation
  :  conditionalAndOperation (conditionalOrOperator^ conditionalAndOperation)*
  ;



// ### conditionalOrOperator ########################################################################

conditionalOrOperator
  :  DOUBLE_PIPE -> CEZVE_OPERATION_OR
  |  OR          -> CEZVE_OPERATION_OR
  ;



// ### conditionalAndOperation #######################################################################

conditionalAndOperation
  :  bitwiseOrOperation (conditionalAndOperator^ bitwiseOrOperation)*
  ;
```

```
// ### conditionalAndOperator ###############################################################################

conditionalAndOperator
  :  DOUBLE_AMPERSAND -> CEZVE_OPERATION_AND
  |  AND              -> CEZVE_OPERATION_AND
  ;



// ### bitwiseOrOperation ###################################################################################

bitwiseOrOperation
  :  bitwiseXorOperation (PIPE bitwiseXorOperation)*
     -> ^(CEZVE_OPERATION_BITWISE_OR  bitwiseXorOperation bitwiseXorOperation?)
  ;



// ### bitwiseXorOperation ##################################################################################

bitwiseXorOperation
  :  bitwiseAndOperation (bitwiseXorOperator^ bitwiseAndOperation)*
  ;



// ### bitwiseXorOperator ###################################################################################

bitwiseXorOperator
  :  CARET -> CEZVE_OPERATION_BITWISE_XOR
  |  XOR   -> CEZVE_OPERATION_BITWISE_XOR
  ;



// ### bitwiseAndOperation ##################################################################################

bitwiseAndOperation
  :  arrayOperation (AMPERSAND arrayOperation)*
     -> ^(CEZVE_OPERATION_BITWISE_AND arrayOperation arrayOperation?)
  ;



// ### arrayOperation #######################################################################################

arrayOperation
  :  rangeOperation (arrayOperator^ rangeOperation)*
  ;



// ### arrayOperator ########################################################################################

arrayOperator
  :  IN         -> CEZVE_OPERATION_IN
  |  NOTIN      -> CEZVE_OPERATION_NOTIN
  ;



// ### rangeOperation #######################################################################################

rangeOperation
  :  equalityOperation (DOUBLE_DOT equalityOperation)*
     -> ^(CEZVE_OPERATION_RANGE equalityOperation equalityOperation?)
  ;



// ### equalityOperation ####################################################################################

equalityOperation
  :  comparisionOperation (equalityOperator^ comparisionOperation)*
  ;
```

```
// ### equalityOperator ##############################################################################

equalityOperator
  :  TRIPLE_EQUALS                -> CEZVE_OPERATION_COMPARISION_IDE
  |  IS                           -> CEZVE_OPERATION_COMPARISION_IDE
  |  ISNOT                        -> CEZVE_OPERATION_COMPARISION_NIDE
  |  EXCLAMATION_MARK_AND_EQUALS  -> CEZVE_OPERATION_COMPARISION_NEQ
  |  DOUBLE_EQUALS                -> CEZVE_OPERATION_COMPARISION_EQU
  ;



// ### comparisionOperation ##########################################################################

comparisionOperation
  :  shiftOperation (comparisionOperator^ shiftOperation)*
  ;



// ### comparisionOperator ###########################################################################

comparisionOperator
  :  LEFT_ANGLE_BRACKET              -> CEZVE_OPERATION_COMPARISION_LT
  |  LEFT_ANGLE_BRACKET_AND_EQUALS   -> CEZVE_OPERATION_COMPARISION_LTE
  |  RIGHT_ANGLE_BRACKET             -> CEZVE_OPERATION_COMPARISION_GT
  |  RIGHT_ANGLE_BRACKET_AND_EQUALS  -> CEZVE_OPERATION_COMPARISION_GTE
  |  INSTANCEOF                      -> CEZVE_OPERATION_COMPARISION_INSTANCEOF
  ;



// ### shiftOperation ################################################################################

shiftOperation
  :  additiveOperation (shiftOperator^ additiveOperation)*
  ;



// ### shiftOperator #################################################################################

shiftOperator
  :  DOUBLE_LEFT_ANGLE_BRACKET  -> CEZVE_OPERATION_BITWISE_SL
  |  DOUBLE_RIGHT_ANGLE_BRACKET -> CEZVE_OPERATION_BITWISE_SRS
  |  TRIPLE_RIGHT_ANGLE_BRACKET -> CEZVE_OPERATION_BITWISE_SRU
  ;



// ### additiveOperation #############################################################################

additiveOperation
  :  multiplicativeOperation (additiveOperator^ multiplicativeOperation)*
  ;



// ### additionOrSubtractionOperator #################################################################

additiveOperator
  :  PLUS       -> CEZVE_OPERATION_ADDITION
  |  MINUS      -> CEZVE_OPERATION_SUBTRACTION
  |  UNDERSCORE -> CEZVE_OPERATION_CONCATENATION
  ;



// ### multiplicativeOperation #######################################################################

multiplicativeOperation
  :  powerOperation (multiplicativeOperator^ powerOperation)*
  ;



// ### multiplicativeOperator ########################################################################
```

```
multiplicativeOperator
  :  ASTERISK  -> CEZVE_OPERATION_MULTIPLICATION
  |  SLASH     -> CEZVE_OPERATION_DIVISION
  |  BACKSLASH -> CEZVE_OPERATION_TRUNCATING_DIVISION
  |  PERCENT   -> CEZVE_OPERATION_MODULO
  ;



// ### powerOperation ###########################################################################

powerOperation
  :  unaryOperation (DOUBLE_ASTERISK unaryOperation)*
     -> ^(CEZVE_OPERATION_POWER unaryOperation unaryOperation?)
  ;



// ### unaryOperation ###########################################################################

unaryOperation
  :  PLUS             inParensOrNot -> ^(CEZVE_OPERATION_POSITIVE           inParensOrNot)
  |  MINUS            inParensOrNot -> ^(CEZVE_OPERATION_NEGATIVE           inParensOrNot)
  |  TILDE            inParensOrNot -> ^(CEZVE_OPERATION_BITWISE_COMPLEMENT inParensOrNot)
  |  EXCLAMATION_MARK inParensOrNot -> ^(CEZVE_OPERATION_NOT                inParensOrNot)
  |  inParensOrNot
  ;



// ### inParensOrNot ############################################################################

inParensOrNot
options {backtrack = true;}
  :  atom
  |  LEFT_PARENTHESIS expression RIGHT_PARENTHESIS -> ^(CEZVE_OPERATION_IN_PARENS expression)
  ;



// ### atom #####################################################################################

atom
  :  literal atomTrailer* -> ^(CEZVE_FRAGMENT_ATOM literal atomTrailer*)
  |  method  atomTrailer* -> ^(CEZVE_FRAGMENT_ATOM method  atomTrailer*)
  |  field   atomTrailer* -> ^(CEZVE_FRAGMENT_ATOM field   atomTrailer*)
  |  DOUBLE_PLUS  field   -> ^(CEZVE_FRAGMENT_ATOM CEZVE_OPERATION_PRE_INCREMENT field)
  |  DOUBLE_MINUS field   -> ^(CEZVE_FRAGMENT_ATOM CEZVE_OPERATION_PRE_DECREMENT field)
  |  field postIncrementOrDecrementOperator
     -> ^(CEZVE_FRAGMENT_ATOM postIncrementOrDecrementOperator field)
  ;



// ### postIncrementOrDecrementOperator #########################################################

postIncrementOrDecrementOperator
  :  DOUBLE_PLUS  -> CEZVE_OPERATION_POST_INCREMENT
  |  DOUBLE_MINUS -> CEZVE_OPERATION_POST_DECREMENT
  ;



// ### literal ##################################################################################

literal
options {backtrack = true; k = 4;}
  :  CEZVE_LITERAL_BOOLEAN
  |  CEZVE_LITERAL_BYTE
  |  CEZVE_LITERAL_CHARACTER
  |  floatLiteral
  |  doubleLiteral
  |  integerLiteral
  |  CEZVE_LITERAL_LONG
  |  CEZVE_LITERAL_NULL
  |  CEZVE_LITERAL_SHORT
  |  CEZVE_LITERAL_STRING
  |  LEFT_SQUARE_BRACKET expressionList? RIGHT_SQUARE_BRACKET
     -> ^(CEZVE_LITERAL_ARRAY expressionList?)
```

```
  |  LEFT_CURLY_BRACKET pairList? RIGHT_CURLY_BRACKET
     -> ^(CEZVE_LITERAL_HASHTABLE pairList?)
  |  arrayList
  ;



// ### floatLiteral ##############################################################################

floatLiteral
  :  CEZVE_LITERAL_NUMBER DOT CEZVE_LITERAL_NUMBER ('F' | 'f')?
     -> ^(CEZVE_LITERAL_FLOAT CEZVE_LITERAL_NUMBER CEZVE_LITERAL_NUMBER)
  |  CEZVE_LITERAL_NUMBER ('F' | 'f')
     -> ^(CEZVE_LITERAL_FLOAT CEZVE_LITERAL_NUMBER)
  ;



// ### doubleLiteral #############################################################################

doubleLiteral
  :  CEZVE_LITERAL_NUMBER (DOT CEZVE_LITERAL_NUMBER)? ('D' | 'd')
     -> ^(CEZVE_LITERAL_DOUBLE CEZVE_LITERAL_NUMBER CEZVE_LITERAL_NUMBER?)
  ;



// ### integerLiteral ############################################################################

integerLiteral
  :  CEZVE_LITERAL_NUMBER -> ^(CEZVE_LITERAL_INTEGER CEZVE_LITERAL_NUMBER)
  ;



// ### identifier ################################################################################

identifier
  :  CEZVE_ATOM_IDENTIFIER
  |  CEZVE_ATOM_NAME
  ;



// ### arrayList #################################################################################

arrayList
  :  LEFT_PARENTHESIS expression COMMA (expression (COMMA expression)* COMMA?)? RIGHT_PARENTHESIS
     -> ^(CEZVE_LITERAL_ARRAY_LIST expression+)
  |  LEFT_PARENTHESIS RIGHT_PARENTHESIS -> CEZVE_LITERAL_ARRAY_LIST
  ;



// ### pairList ##################################################################################

pairList
  :  pair (COMMA pair)* COMMA? -> pair+
  ;



// ### pair ######################################################################################

pair
  :  expression COLON expression -> ^(CEZVE_FRAGMENT_HASHTABLE_PAIR expression+)
  ;



// ### atomTrailer ###############################################################################

atomTrailer
  :  DOT! (field | method)
  |  LEFT_SQUARE_BRACKET
     (slicerStart (slicerStop slicerStep?)? | slicerStop  slicerStep?)
     RIGHT_SQUARE_BRACKET
     -> ^(CEZVE_FRAGMENT_SLICER slicerStart? slicerStop? slicerStep?)
  ;
```

```
// ### slicerStart ##############################################################################

slicerStart
  :  expression -> ^(CEZVE_FRAGMENT_SLICER_START expression)
  ;



// ### slicerStop ###############################################################################

slicerStop
  :  COLON expression? -> ^(CEZVE_FRAGMENT_SLICER_STOP expression?)
  ;



// ### slicerStep ###############################################################################

slicerStep
  :  COLON expression? -> ^(CEZVE_FRAGMENT_SLICER_STEP expression?)
  ;



// ### field ####################################################################################

field
  :  identifier -> ^(CEZVE_FRAGMENT_FIELD identifier)
  ;



// ### method ###################################################################################

method
  :  identifier arguments -> ^(CEZVE_FRAGMENT_METHOD identifier arguments)
  ;



// ### arguments ################################################################################

arguments
  :  LEFT_PARENTHESIS expressionList? RIGHT_PARENTHESIS
     -> ^(CEZVE_FRAGMENT_METHOD_ARGUMENTS expressionList?)
  ;



// ### pack #####################################################################################
pack
options {backtrack = true;}
  :  CEZVE_ATOM_NAME (DOT CEZVE_ATOM_NAME)* packAlias?
     -> ^(CEZVE_FRAGMENT_PACK_SINGLE CEZVE_ATOM_NAME+ packAlias?)
  |  (CEZVE_ATOM_NAME DOT)+ ASTERISK -> ^(CEZVE_FRAGMENT_PACK_ALL CEZVE_ATOM_NAME+)
  ;



// ### packAlias ################################################################################

packAlias
  :  AS identifier -> ^(CEZVE_FRAGMENT_PACK_ALIAS identifier)
  ;



// ### [TR] SOZCUK COZUMLEYICI KURALLARI ########################################################
// ### [EN] LEXER RULES #########################################################################
// ############################################################################################

// [TR] ANAHTAR KELIMELER ---------------------------------------------------------------------
// [EN] KEYWORDS ------------------------------------------------------------------------------

IMPORT                          :   'import'    ;
AS                              :   'as'        ;
```

```
IF                                  :   'if'        ;
ELSE                                :   'else'      ;
WHILE                               :   'while'     ;

// [TR] REZERVE SOZCUKLER ---------------------------------------------------------------------------
// [EN] RESERVED TOKENS -----------------------------------------------------------------------------

// # TIMES                           :   'times'     ;
// # TRY                             :   'try'       ;
// # UNLESS                          :   'unless'    ;
// # UNTIL                           :   'until'     ;
// # ABSTRACT                        :   'abstract'  ;
// # BREAK                           :   'break'     ;
// # CASE                            :   'case'      ;
// # CLASS                           :   'class'     ;
// # CONTINUE                        :   'continue'  ;
// # DEFAULT                         :   'default'   ;
// # DEF                             :   'def'       ;
// # DO                              :   'do'        ;
// # FINAL                           :   'final'     ;
// # FINALLY                         :   'finally'   ;
// # FUNC                            :   'func'      ;
// # GET                             :   'get'       ;
// # EACH                            :   'each'      ;
// # ENUM                            :   'enum'      ;
// # IFACE                           :   'iface'     ;
// # PACKAGE                         :   'package'   ;
// # PRIVATE                         :   'private'   ;
// # PROTECTED                       :   'protected' ;
// # PUBLIC                          :   'public'    ;
// # RAISE                           :   'raise'     ;
// # RETURN                          :   'return'    ;
// # STATIC                          :   'static'    ;
// # SWITCH                          :   'switch'    ;
// # GRAVE_ACCENT                    :   '`'         ;

// [TR] ISLECLER ------------------------------------------------------------------------------------
// [EN] PERATORS ------------------------------------------------------------------------------------

// P: Precedence

PLUS                                :  '+'          ; // [P12] Addition, Concat. [P15] positive
MINUS                               :  '-'          ; // [P12] Subtraction, [P15] negative
UNDERSCORE                          :  '_'          ; // [P12] String Concatenation (PHP .)
ASTERISK                            :  '*'          ; // [P13] Multiplication
SLASH                               :  '/'          ; // [P13] Division
BACKSLASH                           :  '\\'         ; // [P13] Truncating division (Python //)
PERCENT                             :  '%'          ; // [P13] Modulo
DOUBLE_ASTERISK                     :  '**'         ; // [P14] Power (Python **)
DOUBLE_PLUS                         :  '++'         ; // Unary Increment
DOUBLE_MINUS                        :  '--'         ; // Unary Decrement

EXCLAMATION_MARK                    :  '!'          ; // [P15] Boolean NOT
AMPERSAND                           :  '&'          ; // [P07] Bitwise AND
CARET                               :  '^'          ; // [P06] Bitwise XOR
XOR                                 :  'xor'        ; // [P06] Bitwise XOR (Javy xor)
PIPE                                :  '|'          ; // [P05] Bitwise OR
DOUBLE_AMPERSAND                    :  '&&'         ; // [P04] Conditional AND
AND                                 :  'and'        ; // [P04] Conditional AND (Python,PHP and)
DOUBLE_PIPE                         :  '||'         ; // [P03] Conditional OR
OR                                  :  'or'         ; // [P03] Conditional OR (Python, PHP or)
QUESTION_MARK                       :  '?'          ; // [P02] Conditional Ternary if
COLON                               :  ':'          ; // [P02] Conditional Ternary "else", HT delimiter
DOUBLE_LEFT_ANGLE_BRACKET           :  '<<'         ; // [P11] Bitwise shift left
DOUBLE_RIGHT_ANGLE_BRACKET          :  '>>'         ; // [P11] Bitwise shift right - signed
TRIPLE_RIGHT_ANGLE_BRACKET          :  '>>>'        ; // [P11] Bitwise shift right - unsigned
TILDE                               :  '~'          ; // [P15] Bitwise Unary Complement

LEFT_ANGLE_BRACKET                  :  '<'          ; // [P10] Less Than
RIGHT_ANGLE_BRACKET                 :  '>'          ; // [P10] Greater Than
LEFT_ANGLE_BRACKET_AND_EQUALS       :  '<='         ; // [P10] Less Than or equal to
RIGHT_ANGLE_BRACKET_AND_EQUALS      :  '>='         ; // [P10] Greater Than or equal to
INSTANCEOF                          :  'instanceof' ; // [P10] Greater Than or equal to
TRIPLE_EQUALS                       :  '==='        ; // [P09] Identical
IS                                  :  'is'         ; // [P09] Identical (Python is)
ISNOT                               :  'isnot'      ; // [P09] Not identical (Python is not)
EXCLAMATION_MARK_AND_EQUALS         :  '!='         ; // [P09] Not equal to
DOUBLE_EQUALS                       :  '=='         ; // [P09] Equal to
```

```
EQUALS                                :  '='          ; // [P01] Assignment
PLUS_AND_EQUALS                       :  '+='         ; // [P01] Addition and assignment,
                                                        //       Concatenation and assignment
MINUS_AND_EQUALS                      :  '-='         ; // [P01] Subtraction and assignment
ASTERISK_AND_EQUALS                   :  '*='         ; // [P01] Multiplication and assignment
SLASH_AND_EQUALS                      :  '/='         ; // [P01] Division and assignment
BACKSLASH_AND_EQUALS                  :  '\\='        ; // [P01] Truncating division and assignment
DOUBLE_ASTERISK_AND_EQUALS            :  '**='        ; // [P01] Power and assignment
PERCENT_AND_EQUALS                    :  '%='         ; // [P01] Modulo and assignment
UNDERSCORE_AND_EQUALS                 :  '_='         ; // [P01] Concatenation and assignment
AMPERSAND_AND_EQUALS                  :  '&='         ; // [P01] Bitwise AND and assignment
PIPE_AND_EQUALS                       :  '|='         ; // [P01] Bitwise OR and assignment
CARET_AND_EQUALS                      :  '^='         ; // [P01] Bitwise XOR and assignment
DOUBLE_LEFT_ANGLE_BRACKET_AND_EQUALS  :  '<<='        ; // [P01] Bitwise shift left and assignment
DOUBLE_RIGHT_ANGLE_BRACKET_AND_EQUALS :  '>>='        ; // [P01] Bitwise shift right and assignment
TRIPLE_RIGHT_ANGLE_BRACKET_AND_EQUALS :  '>>>='       ; // [P01] Bitwise shift right and assignment

DOUBLE_DOT                            :  '..'         ; // [P08] range (Perl ..)
IN                                    :  'in'         ; // [P08] is in array (Python in)
NOTIN                                 :  'notin'      ; // [P08] is not in array (Python not in)

COMMA                                 :  ','          ; // Method params, Array, ArrayList,
                                                        // Hashtable delimiter
DOT                                   :  '.'          ; // Member access

LEFT_CURLY_BRACKET                    :  '{'          ; // Start of block or hashtable
LEFT_PARENTHESIS                      :  '('          ; // Start of in parens expressions or list
LEFT_SQUARE_BRACKET                   :  '['          ; // Start of arraylists or slicers
RIGHT_CURLY_BRACKET                   :  '}'          ; // End of block or hashtable
RIGHT_PARENTHESIS                     :  ')'          ; // End of in parens expressions or list
RIGHT_SQUARE_BRACKET                  :  ']'          ; // End of arraylists or slicers
SEMICOLON                             :  ';'          ; // End of single line statements

// [TR] YOKSAYILAN KURALLAR ------------------------------------------------------------------------
// [EN] IGNORE RULES -------------------------------------------------------------------------------

WHITESPACE                            :  (' ' | '\t')+                                  {$channel = HIDDEN;} ;
COMMENT                               :  SLASH_AND_ASTERISK (.)* ASTERISK_AND_SLASH     {$channel = HIDDEN;} ;
LINE_COMMENT                          :  (DOUBLE_SLASH | SHARP) ~('\u000D' | '\u000A')* {$channel = HIDDEN;} ;
EOL                                   :  ('\u000D' | '\u000A')                          {$channel = HIDDEN;} ;

// [TR]TEMEL VERI IFADELERI ------------------------------------------------------------------------
// [EN]FUNDAMENTAL DATA LITERALS -------------------------------------------------------------------

CEZVE_LITERAL_NULL                    :  'null'                          ;
CEZVE_LITERAL_BOOLEAN                 :  TRUE | FALSE                    ;
CEZVE_LITERAL_BYTE                    :  CEZVE_LITERAL_NUMBER ('B' | 'b')  ;
CEZVE_LITERAL_SHORT                   :  CEZVE_LITERAL_NUMBER ('S' | 's')  ;
CEZVE_LITERAL_LONG                    :  CEZVE_LITERAL_NUMBER ('L' | 'l')  ;
CEZVE_LITERAL_NUMBER                  :  DIGIT+        ;
CEZVE_LITERAL_CHARACTER               :  APOSTROPHE
                                         (ESCAPE_SEQUENCE | ~('\\' | '\u000D' | '\u000A' | '\''))
                                         APOSTROPHE
                                      ;
CEZVE_LITERAL_STRING
                                      :
                                      (
                                          TRIPLE_APOSTROPHE     (options {greedy=false;}:.)* TRIPLE_APOSTROPHE
                                          {String st = getText();setText(st.substring(2, st.length() - 2));}
                                        |  TRIPLE_QUOTATION_MARK (options {greedy=false;}:.)*
TRIPLE_QUOTATION_MARK
                                          {String st = getText();setText(st.substring(2, st.length() - 2));}
                                        |  QUOTATION_MARK  (ESCAPE_SEQUENCE | ~('\\' | '\u000D' | '\u000A' |
'"' ))* QUOTATION_MARK
                                        |  APOSTROPHE      (ESCAPE_SEQUENCE | ~('\\' | '\u000D' | '\u000A' |
'\''))+ APOSTROPHE
                                      )
                                           ;

// [TR] BELIRTECLER --------------------------------------------------------------------------------
// [EN] IDENTIFIERS --------------------------------------------------------------------------------

CEZVE_ATOM_NAME                       :  LETTER+                 ;
CEZVE_ATOM_IDENTIFIER                 :  LETTER (LETTER | DIGIT)* ;

// [TR] PARCA SOZCUKLER ----------------------------------------------------------------------------
// [EN] FRAGMENT TOKENS ----------------------------------------------------------------------------

fragment DOUBLE_SLASH                 :  '//'         ; // C style single line comment
```

```
fragment SHARP                          :  '#'          ; // Bash style single line comment
fragment SLASH_AND_ASTERISK             :  '/*'         ; // Start of C style multi line comment
fragment ASTERISK_AND_SLASH             :  '*/'         ; // End of C style multi line comment
fragment ESCAPE_SEQUENCE                :  '\\' .       ;
fragment TRUE                           :  'true'       ;
fragment FALSE                          :  'false'      ;
fragment DIGIT                          :  '0' .. '9'   ;
fragment APOSTROPHE                     :  '\''         ;
fragment TRIPLE_APOSTROPHE              :  '\'\'\''     ;
fragment QUOTATION_MARK                 :  '"'          ;
fragment TRIPLE_QUOTATION_MARK          :  '"""'        ;

fragment LETTER                         :
                                           '\u0024'
                                        |  '\u005F'
                                        |  '\u0041' .. '\u005A'
                                        |  '\u0061' .. '\u007A'
                                        |  '\u00C0' .. '\u00D6'
                                        |  '\u00D8' .. '\u00F6'
                                        |  '\u00F8' .. '\u00FF'
                                        |  '\u0100' .. '\u1FFF'
                                        |  '\u3040' .. '\u318F'
                                        |  '\u3300' .. '\u337F'
                                        |  '\u3400' .. '\u3D2D'
                                        |  '\u4E00' .. '\u9FFF'
                                        |  '\uF900' .. '\uFAFF'
                                        ;
```