Haskell ile Kategori Teori

Elis SOYLU

YÜKSEK LİSANS TEZİ
Matematik ve Bilgisayar Bilimleri Anabilim Dalı
Haziran 2013

Category Theory with Haskell

Elis SOYLU

MASTER DISSERTATION
Department of Mathematics and Computer Science
June 2013

Haskell ile Kategori Teori

Elis SOYLU

Eskişehir Osmangazi Üniversitesi

Fen Bilimleri Enstitüsü

Lisans üstü Yönetmeliği Uyarınca

Matematik ve Bilgisayar Bilimleri Anabilim Dalı

Bilgisayar Bilimleri Bilim Dalında

YÜKSEK LİSANS TEZİ

Olarak Hazırlanmıştır

Danışman: Yrd. Doç. Dr. Alper Odabaş

Haziran 2013

ONAY

Matematik ve Bilgisayar Bilimleri Anabilim Dalı yüksek lisans öğrencisi Elis SOYLU'nun YÜKSEK LİSANS TEZİ olarak hazırladığı "**Haskell ile Kategori Teori**" başlıklı bu çalışma, jürimizce lisansüstü yönetmeliğinin ilgili maddeleri uyarınca değerlendirilerek kabul edilmiştir.

Danışman : Yrd. Doç. Dr. Alper ODABAŞ

İkinci Danışman : –

Yüksek Lisans Tez Savunma Jürisi:

Üye : Yrd. Doç. Dr. Alper ODABAŞ

Üye : Prof. Dr. Zekeriya ARVASİ

Üye : Doç. Dr. Yılmaz DERELİ

Üye : Doç. Dr. Enver Önder USLU

Üye : Yrd. Doç. Dr. İlker AKÇA

Fen Bilimleri Enstitüsü Yönetim Kurulu'nun ..... tarih ve ..... sayılı kararıyla onaylanmıştır.

Prof. Dr. Nimetullah BURNAK

Enstitü Müdürü

ÖZET

Haskell ile Kategori Teorisi üzerine hazırlanan bu tez dört bölümden oluşmaktadır. İlk bölümde kategori teorinin diğer bölümlerde faydalanılacağı temel kavram ve bilgiler verilmiştir. İkinci bölümde bilgisayar programlama tarihçesi ve programlama dilleri örneklerine yer verilmiştir. Üçüncü bölümde fonksiyonel programlama dillerinden Haskell dilinin temel yapı, özellik ve örneklerine ilişkin kapsamlı bilgiler yer almıştır. Son bölümde ise Haskell dili ile kategori teorinin bazı kavramları arasındaki benzerliklere vurgu yapılmıştır. Kategori teoride öne çıkan bazı temel yapılar Haskell dili ile bilgisayar diline çevrilmiştir.

Anahtar Kelimeler: Fonksiyonel Programlama, Haskell, Kategori Teori.

SUMMARY

This thesis consist of four chapters. In the first chapter some basic notions and informations about category theory which contributed to other chapters. In the second chapter history of computer and programming and examples of programming languages is given. In the following chapter Haskell's ,one of the functional programming language, basic structures, properties and examples is given comprehensively. In the last chapter, similarity between category theory and Haskell about some notions is emphasized. Some basic prominent structures in category theory are translated to computer language via Haskell.

Keywords: Functional Programming, Haskell, Category Theory.

TEŞEKKÜR

Bu teze katkı ve yönlendirmelerinden dolayı değerli hocam, sayın;

Prof. Dr. Zekeriya ARVASİ'ye

Beni bu çalışmaya yönlendiren ve tezim boyunca her türlü bilgi ve yardımlarını esirgemeyen başta değerli danışman hocam, sayın;

Yrd. Doç. Dr. Alper ODABAŞ'a

Bu süreçte her türlü desteğini esirgemeyen sevgili arkadaşım;

Arş. Gör. Elif ILGAZ'a

Tanıştıktan sonra her anımda desteklerini esirgemeyen sevgili;

ARKADAŞLARIM'a

Ayrıca son olarak da beni bugünlere getiren,

değerli ailem Kıymet SOYLU, Enol SOYLU ve biricik kardeşim Sibel SOYLU'ya

ayrı ayrı sonsuz saygı ve teşekkürlerimi sunarım.

İÇİNDEKİLER

BÖLÜM 0. Giriş **1**

BÖLÜM 1. KATEGORİ TEORİ **3**

1.1 Giriş . . . 3

1.2 Kategoriler . . . 3

1.2.1 Küçük Kategoriler . . . 6

1.3 Özel Obje ve Morfizmler . . . 9

1.3.1 Başlangıç ve Varış Objeleri . . . 9

1.3.2 Sabitler . . . 10

1.4 Funktorlar . . . 11

1.5 Doğal Transformasyon . . . 12

BÖLÜM 2. PROGRAMLAMA DİLLERİ ve FONKSİYONEL PROGRAMLAMA **14**

2.1 Giriş . . . 14

2.2 Fonksiyonel Programlama . . . 17

2.3 Fonksiyonel Programlama ile Nesnel Programlamanın Karşılaştırılması . . . 20

BÖLÜM 3. HASKELL **24**

3.1 Giriş . . . 24

3.2 Programın yüklenmesi ve kullanımı . . . 24

3.3 Haskell Programlamada Temel Yapılar . . . 25

3.3.1 Değişkenler . . . 25

3.3.2 Fonksiyonlar . . . 27

3.3.3 Karşılaştırma Operatörleri . . . 30

3.3.4 Mantıksal Operatörler . . . 30

3.3.5 Guard Yapısı . . . 31
3.4 Tipler . . . 32
3.5 Tip Sınıfları . . . 33
3.5.1 Bazı temel tip sınıfları: . . . 34
3.6 Listeler ve Tuples . . . 39
3.6.1 Listeler . . . 39
3.6.2 Tuples . . . 42
3.7 Tekrarlama . . . 43
3.8 Liste İşlemleri . . . 45
3.8.1 Map fonksiyonu . . . 45
3.8.2 Fold Fonksiyonu . . . 46
3.8.3 Filter Fonksiyonu . . . 47
3.9 Modüller . . . 47
3.10 Input&Output . . . 49
3.11 Yüksek Dereceli Fonksiyonlar . . . 50
3.11.1 Curry Fonksiyonu . . . 51
3.12 Monad . . . 52
3.12.1 Monad Kuralları . . . 55
3.12.2 Maybe Monad Yapısı . . . 56
3.12.3 Liste Monad Yapısı . . . 57

BÖLÜM 4. KATEGORİ ve HASKELL **59**

4.1 Giriş . . . 59
4.2 Bir $\mathcal{L}$ Dilinin Kategorisi . . . 62
4.3 Programlama Dilinde Kategoriksel Yapılar . . . 68
4.3.1 Funktor . . . 71

4.3.2 Monoid 75

4.3.3 Doğal Transformasyon 78

KAYNAKLAR DİZİNİ **83**

BÖLÜM 0

Giriş

Matematiğin birçok dalı; küme ile fonksiyonlar; vektör uzayları ile lineer dönüşümler; topolojik uzaylar ile sürekli fonksiyonlar; gruplar ile homomorfizmler de olduğu gibi objeler, oklar ve bunlar arasındaki ilişkiler üzerine çalışma yapar. Özellikle farklı konuların birbiriyle bağlantısı da merak uyandıran konular arasındadır. Örneğin; cebirsel topoloji, topoloji ile grup teori arasındaki bağlantıyı kuran alanlardan biridir. Bu bağlantılar için kullanılan metotların en önemli başarısı; objelerin oklar yardımıyla somutlaştırılmasıdır.[8] Bunun en iyi örneği; izometrilerin kristal yapıların sınıflandırılmasıyla ilişkilendirilmesidir. Kategori teori ise, objeler,oklar(morfizmler) ve kompozisyonlar yardımıyla bazı şartlar sağlatılarak yeni kategoriksel yapılar elde etmeyi amaçlar. Bu yapıları geneller ve '**inşaa**' etmeyi genel anlamda karakterize eder. Böylece programlama dili teorisi ile kategori teori arasındaki sıkı bağ açığa çıkar. Bu metotsal bağlantı; her iki teorinin de '**fonksiyonlar teorisi**' üzerine kurulmuş olmasından kaynaklanır. Matematikte son 400 yıldır fonksiyonlar en temel role sahiptir. Fonksiyonlar, parametreler (input) ile sonuç (output) arasındaki bağlantıyı açıklar. Bu ise hem programlamanın hem de kategori teorinin temelini oluşturur. Bir diğer bakış açısı ise; morfizmlerin kategoriksel tanımı ile programlama dili teorisinin çeşitli konularının fonksiyonların kümeler teorisindeki genellemeyle birebir örtüşmesidir. Bunun dışında kategori teori; bilgisayar bilimlerinin Yazılım mühendisliği, Yapay zeka, Automata teorisi gibi alanlarına da katkı sağlar. Bunun gibi matematiksel katkılar sayesinde aynı yapıdaki birçok dil, dillerin kategorisi çatısı altında toplanmaya çalışılır.

İlk bilgisayar programları, çözümlenemeyen bazı problemlerin açığa çıkması amacıyla geliştirilmiştir. Alan Turing 1936'da ilk olarak Turing makinesini icat ederek makinelerin çözemediği problemler olduğunu ortaya atmıştır. Bunun üzerine genel algoritmik programlama matematikçiler ve mantıkçılar için temel bir araştırma alanı olmuştur. Böylece Alonzo Church - Lambda hesabı, Alan Turing - Turing Makineleri, Markov sistemleri, tekrarlayan fonksiyonlar gibi programlama modelleri ortaya atılmıştır. 1940 larda patlak veren II. Dünya Savaşı ile birlikte balistik hesaplamalar için dijital bilgisayarlara ihtiyaç ortaya çıkmıştır. Alan Turing; ENİGMA isimli şifre çözücü bilgisayarı icat etmiştir. 1948 yılında ilk depolayan bilgisayar tarihe geçmiştir. 1951 yılında ilk derleyici fikrinin ortaya atılmasıyla programlama dilleri altın çağına ulaşmıştır. John McCathy tarafından LISP dili ilk fonksiyonel programlama

dili olmuştur ve uzun yıllar da tek olarak kalmıştır. Bu şekilde gelişme gösteren programlama mantığı zamanla karmaşıklığa dönmüştür. Bunun için iyi bir verileme sistemine gerek duyulmuştur. Bununla birlikte esas modellemenin rolü önemi giderek artmıştır. 1980 lerde ML, Scheme, Miranda gibi bu sorunlara çözüm olabilecek diller doğmuştur. Sonuç olarak her bilim insanı kendi dilini geliştirmeye başlamıştır. Bu şekilde artan programlama dili sayısı da bilgisayar yazılımı dünyasında karmaşıklığa yol açmıştır. Gittikçe yayılan bu hareketi durdurmak için diğer birçok dilin avantajlarını birleştiren yeni bir dil oluşturulmuştur. Bu dilin 1990 lardaki ilk uygulaması Haskell fonksiyonel programlama dilidir.

Haskell gibi fonksiyonel programlama dillerinin en önemli avantajı yığılma özelliğine sahip olmasıdır. Genel olarak bir fonksiyonel programlama dili hiçbir yan etki içermez. Bir program derlenirken diğeri üzerinde etki bırakmaz. Daha da önemlisi program kendi içinde atama deyimi içermez. Böylece değişkenler programın başından sonuna kadar sabit bir değerde kalırlar. Böylece bir algoritmanın karşılaşabileceği birçok hatayı devre dışı bırakır. Bu ise programcının algoritma akışını kontrol altında tutmasını sağlar. Hataların önü kesilebildiğinden ifadeler her zaman derlenebilir. Bu özelliğinden dolayı da program referans bakımından özgür kalır. Bu özgürlük ise fonksiyonel programlamanın diğer geleneksel programlama dillerine göre matematiksel olarak işlenebilmeye açık hale gelir.

Fonksiyonel programlama dilleri içinde Haskell diğer dillerden daha önemli bir konumdadır. Örneğin; liste tipi direk olarak tiplerin ardışık dizilimi şeklindedir. Böylece 'pointer' gibi bir tanımlamaya gerek kalmaz. Ayrıca Haskell gibi fonksiyonel diller, yordamsal (imperative) dillerin düzenleyici dilleri olarak da görev alırlar.

BÖLÜM 1

KATEGORİ TEORİ

1.1 Giriş

Bu bölümde fonksiyonel programlama yapısında karşılaşılan Kategori Teori'deki bazı önemli tanımlar ve örnekler verilecektir. Daha ayrıntılı bilgiye kaynak olarak kullanılan [9] referans olarak verelim.

1.2 Kategoriler

Bir kategoride;

i) $Ob(\mathcal{C})$ bir sınıf olmak üzere bu sınıfın elemanları **obje** olarak adlandırılır.

ii) $Ob(\mathcal{C})$ sınıfından alınan her A,B objeleri için $Mor_{\mathcal{C}}(A,B)$ veya $\mathcal{C}(A,B)$ ile gösterilen A objesinden B objesine giden tüm **morfizmlerin kümesi** olarak adlandırılır.

iii) $Ob(\mathcal{C})$ sınıfındaki her A objesine karşılık gelecek şekilde

$$1_A : A \longrightarrow A$$

morfizmi **birim morfizm** olarak adlandırılır.

iv) $f : A \longrightarrow B$ ve $g : B \longrightarrow C$ morfizmleri için bir tek

$$g \circ f : A \longrightarrow C$$

morfizmi f ve g **nin kompozisyonu** olarak adlandırılır. Bu kompozisyon;

$$\begin{array}{rcl} \mathcal{C}(A,B) \times \mathcal{C}(B,C) & \longrightarrow & \mathcal{C}(A,C) \\ (f,g) & \longmapsto & \circ(f,g) = g \circ f = gf \end{array}$$

şeklinde de ifade edilir.

Bu dört yapı yardımıyla herhangi bir $\mathcal{C}$ kategorisi,

$$\mathcal{C} \sim (Ob(\mathcal{C}), Mor(\mathcal{C}), Mor(\mathcal{C}) \times Mor(\mathcal{C}) \stackrel{\circ}{\longrightarrow} Mor(\mathcal{C}); \text{Aksiyomlar})$$

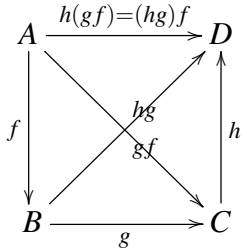
ile gösterilir.

Bu $\mathcal{C}$ sisteminin kategori özelliğini kazanabilmesi için aşağıdaki iki şartı sağlaması gerekir.

K1) $f : A \longrightarrow B, g : B \longrightarrow C$ ve $h : C \longrightarrow D$ morfizmleri için;

$$h(gf) = (hg)f$$

yani



şeklinde gösterilen diyagramında da belirtildiği gibi asosyatiflik özelliğini sağlamalıdır.

K2) $f : A \longrightarrow B$ morfizmi olmak üzere,

$$f.1_A = 1_B.f = f$$

eşitliği sağlanmalıdır.

Uyarılar:

1) Birim morfizm bir tektir.

A objesinin birimleri 1_A ve $1'_A$ olsun.

$$1'_A = 1_A.1'_A = 1_A$$

olduğundan A objesinin bir tek birimi vardır.

2) $\mathcal{C}$ kategorisi, $Mor(\mathcal{C})$ kümesi üzerinde morfizmler cebiri oluşturur. Çünkü,

$$\mathcal{C} \sim (Mor(\mathcal{C}), Mor(\mathcal{C}) \times Mor(\mathcal{C}) \longrightarrow Mor(\mathcal{C}); \text{Aksiyomlar})$$

morfizmler kümesinde bir cebir yapısı oluşturur.

Kategori Örnekleri

1) $\mathcal{C} =$ Küme; Kümeler Kategorisi

·$Ob(\mathcal{C})$, tüm kümelerin sınıfıdır.

·$Mor(\mathcal{C})$, kümeler üzerinde tanımlanan fonksiyonların kümesidir.

·Birim morfizm kümeler kategorisinde birim fonksiyondur.

·Kompozisyon, fonksiyonlar arasındaki bileşke işlemidir.

Benzer şekilde; objeleri sonlu kümeler olan yapı da sonlu kümeler kategorisini oluşturur.

2) $\mathcal{C} = Grp$; Gruplar kategorisi

·$Ob(Grp)$; tüm grupların sınıfından oluşur.

·$Mor(Grp)$; gruplar arasındaki homomorfizmlerin kümesinden oluşur.

·Kompozisyon; grup işlemlerini koruyan homomorfizmler fonksiyonlar olduğundan bileşke işlemine karşılık gelir.

Benzer şekilde, objeleri Abelyan gruplar alınarak Abelyan gruplar kategorisi oluşturulur.

3) $\mathcal{C} = Hlk$; Halkalar kategorisi

·$Ob(Hlk)$; halkaların sınıfından oluşur.

·$Mor(Hlk)$; halkalar arasındaki halka homomorfizmlerinin kümesinden oluşur.

·Kompozisyon; halka homomorfizmleri fonksiyon olduğundan bileşke işlemine karşılık gelir.

Benzer şekilde objeleri tamlık bölgesi ve cisimler olarak alınarak sırasıyla tamlık bölgesi ve cisimler kategorileri elde edilir.

4) $\mathcal{C} = Top$; Topolojik uzaylar kategorisi

·$Ob(\mathcal{C})$; topolojik uzayların sınıfından oluşur.

·$Mor(\mathcal{C})$; topolojik uzaylar arasındaki sürekli fonksiyonlardan oluşur.

·Kompozisyon; sürekli fonksiyonların bileşke işlemi ile elde edilir.

Benzer şekilde objeleri metrik uzaylar alınarak metrik uzaylar kategorisi meydana gelir.

5) Morfizmleri yalnız birim morfizmlerden oluşan kategoriye **ayrık(discrete) kategori** adı verilir.

1.2.1 Küçük Kategoriler

Tanım: Objeleri kümeler olan kategoriye **küçük kategori** adı verilir.

Küçük kategorinin objelerini O, morfizmlerini A ile gösterilirse

$$A \underset{t}{\overset{s}{\rightrightarrows}} O$$

fonksiyonları tanımlanır. Bu fonksiyonlar yardımı ile $f \in A$ için

$$s(f) \in O \text{ ve } t(f) \in O$$

elde edilir. Bu durumda $a, b \in O$ olmak üzere

$$a = s(f) \xrightarrow{f} t(f) = b$$

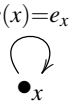
şeklinde A morfizmler kümesindeki herhangi bir fonksiyon ifade edilmiş olur. Burada s ve t fonksiyonlarına sırasıyla f fonksiyonunun **kaynak(source)** ve **hedef(target) fonksiyonları** denir.

$$O \xrightarrow{c} A$$

şeklinde tanımlanan fonksiyonunu ele alalım. Bu durumda her $x \in O$ objesi için

$$c(x) = 1_x : x \longrightarrow x$$

şeklinde birim ok mevcuttur. Yani $x = s(e_x) = t(e_x)$ olmak üzere



diyagramı ile temsil edilir.

Kompozisyon tanımı bu fonksiyonlar yardımıyla şöyle elde edilir.

$$\begin{aligned} \circ : A \times A &\longrightarrow A \\ (f,g) &\longmapsto \circ(f,g) = g \circ f = gf \end{aligned}$$

fonksiyonunun tanımlı olması için,

$$t(f) = s(g)$$

eşitliği sağlanmalıdır. Çünkü,

$$s(f) \xrightarrow{f} t(f) = s(g) \xrightarrow{g} t(g)$$

dır. Ayrıca bu kompozisyon

$$s(gf) = s(f) \xrightarrow{gf} t(g) = t(gf)$$

şeklinde de ifade edilir. O halde,

$$A \times_o A = \{(f,g) | t(f) = s(g)\} \subseteq A \times A$$

şeklinde fiber çarpım ile gösterilir. Kategori aksiyomları bu fonksiyonlar yardımıyla kolayca ifade edilir.

Küçük Kategori Örnekleri

1) Grubun kendisi tek objeli bir küçük kategori oluşturur.

$$O = \{*\} \text{ ve } A = G \sim \text{grup}$$

olmak üzere,

$$G \underset{e}{\overset{s}{\underset{t}{\rightrightarrows}}} \{*\}$$

diyagramı yardımıyla her $g \in G$ için

$$* = s(g) \xrightarrow{g} t(g) = *$$

olup grubun elemanları küçük kategorinin morfizmlerine karşılık gelir. Grubun birimi,

$$e(*) = 1_*$$

birim morfizmi olur.

$$\begin{aligned} \cdot : G \times G &\longrightarrow G \\ (g_1, g_2) &\longmapsto \cdot(g_1, g_2) = g_1 \cdot g_2 \end{aligned}$$

grup işlemi ise küçük kategorinin kompozisyonudur. Yani s ve t fonksiyonları yardımıyla,

$$* = s(g_1) \xrightarrow{g_1} t(g_1) = * = s(g_2) \xrightarrow{g_2} t(g_2) = *$$

şeklinde ifade edilir.

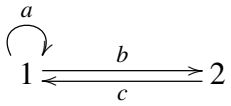
Küçük kategorinin kategori olmasını sağlayan şartlar ise grubun asosyatiflik ve birimlilik aksiyomları ile gerçeklenir.

2) Grubun sadece asosyatiflik ve birimlilik aksiyomları sağlanan cebirsel yapısına **monoid** adı verilir. Örneğin; $(\mathbb{Z}, -)$ bir monoid yapısı oluşturur. Böylece monoid yapısı tek objeli bir küçük kategoriye örnek olarak gösterilir.

3) $O = \{1,2\}$ objeleri ile $A = \{a,b,c\}$ morfizmleri ile

$$\begin{aligned} s(a) &= t(a) = s(b) = t(c) = 1 \\ t(b) &= s(c) = 2 \end{aligned}$$

olarak ele alınan yapı



şeklinde bir küçük kategori meydana getirir.

4) Obje ve morfizmleri **boş küme** olacak şekilde elde edilen kategoriye **boş kategori** denir.

$$A = \{1_0\} \text{ ve } O = \{0\}$$

morfizm ve objeleri alınırsa;



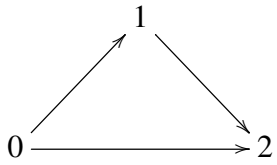
bir obje ve bir morfizmli bir küçük kategori oluşturur.

$$A = \{f, 1_0, 1_1\} \text{ ve } O = \{0,1\}$$

alınırsa;



küçük kategorisi elde edilir. Benzer şekilde



küçük kategoridir. Bu küçük kategoriler,

$$\bigtriangleup = \begin{cases} [n] = \{0 < 1 < 2 < \ldots < n-1\} \sim \text{objeler} \\ [n] \xrightarrow{f} [m]; i \leq j \Rightarrow f(i) \leq f(j) \sim \text{monoton fonksiyonlar} \end{cases}$$

şeklinde genelleştirilirse **delta kategorisi** adı verilen özel bir kategori örneği oluşmuş olur.

1.3 Özel Obje ve Morfizmler

1.3.1 Başlangıç ve Varış Objeleri

Tanım: $\mathcal{C}$ bir kategori olmak üzere her X objesi için,

$$B \xrightarrow{\exists!} X$$

bir tek morfizmi varsa B ye $\mathcal{C}$ nin **başlangıç objesi** denir ve 0 ile gösterilir.

Bu tanımın dual kavramına **varış obje** denir. Yani, her X objesi için

$$X \xrightarrow{\exists!} V$$

bir tek morfizmi varsa V objesi varış obje olur ve 1 ile gösterilir.

Örnekler

1) Kümeler kategorisinde;

$$\begin{aligned} 0 &= \{\} = \emptyset \\ 1 &= \{*\} \sim \text{tek elemanlı bir küme} \end{aligned}$$

başlangıç ve varış objeleridir. Çünkü boş kümeden herhangi bir X kümesine bir tek fonksiyon tanımlıdır. Bu fonksiyon boş fonksiyon olarak bilinir. Yani;

$$Y \xrightarrow{f} X, \;\; U = \{(y,x) | f(y) = x\} \subseteq Y \times X$$

fonksiyonu için $Y = \emptyset$ alınırsa, $U \subseteq \emptyset \times X = \emptyset$ olduğundan $U = \emptyset$ elde edilir. Buradan $\emptyset \xrightarrow{\exists!} X$ bulunur.

Diğer taraftan her X kümesi için;

$$\begin{array}{l} X \xrightarrow{\exists!} \{*\} = 1 \\ a \longmapsto * \end{array}$$

olduğu açıktır.

2) Gruplar kategorisinin başlangıç ve bitiş objeleri aynı olup

$$0 = \{e\} = 1$$

birim grubudur. Çünkü her G grubu için,

$$0 = \{e\} \xrightarrow{\exists!} G \quad \text{ve} \quad G \xrightarrow{\exists!} \{e\} = 1$$
$$e \longmapsto e \qquad\qquad g \longrightarrow e$$

bir tek grup homomorfizmleri vardır.

3) Cisimler kategorisinde başlangıç ve bitiş objeleri yoktur. Çünkü herhangi bir F cisminin toplamsal ve çarpımsal birimleri 0_F ve 1_F olmak üzere

$$E = \{0_F \neq 1_F\} \leq F$$

cismi alınırsa,

$$F \longrightarrow \{0_F, 1_F\} \quad \text{ve} \quad F \longrightarrow \{0_F, 1_F\}$$
$$x \longmapsto 0_F \qquad\qquad x \longmapsto 1_F$$

şeklinde iki homomorfizm oluşturulabilir. Benzer durum başlangıç objesi için de aynıdır.

1.3.2 Sabitler

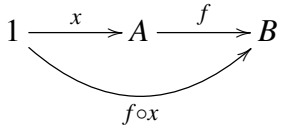
Kümeler kategorisinde, herhangi bir A kümesinin x elemanı

$$1 = \{*\} \xrightarrow{f} A$$
$$* \longmapsto f(*) = x$$

şeklindeki fonksiyonu yardımıyla gösterilir. Yani A kümesinin herhangi bir elemanı,

$$x \in A \longleftrightarrow \{1 \xrightarrow{x} A\}$$

şeklindeki fonksiyonu ile yazılabilir. Böylece $f : A \longrightarrow B$ herhangi bir fonksiyonu ve $x : 1 \longrightarrow A$ verildiğinde



bileşke fonksiyonu ile $f(x)$ elemanı gösterilir.

Kategoriksel olarak,

$$x : 1 \longrightarrow A$$

morfizmi yardımıyla A kümesinin herhangi bir elemanı **sabit** olarak düşünülür.

Örneğin; kümeler kategorisi ele alındığında bir kümenin her elemanı bir **sabit** olarak ele alınır.

1.4 Funktorlar

Bir matematiksel yapıdan diğer matematiksel yapıya tanımlanan dönüşüme **Funktor** adı verilir. Cebirsel özellikleri koruyan homomorfizm yapıları gibi funktorlar da kategoriksel özellikleri iki kategori arasında değiştirmeden taşırlar.

Tanım: $\mathcal{C}$ ve $\mathcal{D}$ iki kategori olmak üzere

$$F : Mor(\mathcal{C}) \longrightarrow Mor(\mathcal{D})$$

fonksiyonu için;

F1) A objesi $\mathcal{C}$ kategorisinin herhangi bir objesi olmak üzere

$$F(id_A) = id_{F(A)}$$

F2) $\mathcal{C}$ kategorisinin bir kompozisyonu olan $g \circ f$ için

$$F(g \circ f) = F(g) \circ F(f)$$

şartları sağlanıyorsa F fonksiyonuna $\mathcal{C}$ kategorisinden $\mathcal{D}$ kategorisine bir **Funktor** adı verilir ve

$$F : \mathcal{C} \longrightarrow \mathcal{D}$$

ile gösterilir.

Funktor Örnekleri

1) Birim Funktor: $\mathcal{C}$ kategorisinin A objesi ve f morfizmi için

$$1_{\mathcal{C}} : \mathcal{C} \longrightarrow \mathcal{C}; 1_{\mathcal{C}}(A) = A \text{ ve } 1_{\mathcal{C}}(f) = f$$

şeklinde funktor tanımlanır. Bu funktora **birim funktor** adı verilir.

2) İçine Funktor: $\mathcal{C}', \mathcal{C}$ kategorisinin bir alt kategorisi olsun. (Objeleri ve morfizmlerini $\mathcal{C}$ kategorisinden alan $\mathcal{C}'$ kategori, $\mathcal{C}$ kategorisinin alt kategorisi olarak tanımlanır.) $\mathcal{C}'$ kategorisinin A objesi ve f morfizmi için

$$F : \mathcal{C}' \longrightarrow \mathcal{C}; F(A) = A \text{ ve } F(f) = f$$

şeklinde tanımlanan funktoru **içine funktor** olarak adlandırılır.

3) Sabit Funktor: $\mathcal{C}$ ve $\mathcal{D}$ iki kategori ve $B, \mathcal{D}$ kategorisinin objesi olmak üzere;

$$F : \mathcal{C} \longrightarrow \mathcal{D}$$

funktoru, $\mathcal{C}$ kategorisinin her A objesi için

$$F(A) = B$$

şeklinde tanımlansın. $\mathcal{C}$ kategorisinin herhangi $f : A_1 \longrightarrow A_2$ morfizmi için

$$1_B = F(f) : F(A_1) = B \longrightarrow F(A_2) = B$$

elde edilir. Yani F funktoru, $\mathcal{C}$ nin herhangi morfizmini $\mathcal{D}$ nin 1_B birim morfizmine dönüştürmektedir. Bu şekildeki funktorlara **sabit funktor** adı verilir.

1.5 Doğal Transformasyon

Tanım: $\mathcal{C}$ kategorisinden $\mathcal{D}$ kategorisine F ve G funktorları ele alınsın.

DT1) $\mathcal{C}$ kategorisinin herhangi A objesi için

$$\eta_A : F(A) \longrightarrow G(A)$$

$\mathcal{D}$ kategorisinin morfizmi olmalıdır.

DT2) $f, \mathcal{C}$ kategorisinde A objesinden B objesine bir morfizm olmak üzere

$$\begin{array}{ccc} A & F(A) & \xrightarrow{\eta_A} G(A) \\ {\scriptstyle f}\downarrow & {\scriptstyle F(f)}\downarrow & \downarrow{\scriptstyle G(f)} \\ B & F(B) & \xrightarrow[\eta_B]{} G(B) \end{array}$$

diyagramı değişmeli olmalıdır.

DT1 ve DT2 şartları sağlanıyorsa

$$\eta : F \Rightarrow G$$

okuna F funktorundan G funktoruna bir **doğal transformasyon** adı verilir.

Örnekler

1) $\mathcal{C} = \mathcal{D}$ ve F, G birim funktorlar olarak alınırsa

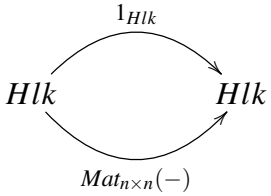
$$\eta : F \Rightarrow G$$

için

$$\eta : 1_{\mathcal{C}} \Rightarrow 1_{\mathcal{D}} (\mathcal{C} = \mathcal{D})$$

birim doğal transformasyonu elde edilir.

2) $\mathcal{C} = \mathcal{D} = Hlk$ için; F birim funktor ve G matris funktoru olarak ele alınırsa $\eta : 1_{Hlk} \longrightarrow Mat_{n\times n}(-)$ olmak üzere



diyagramı için,

DT1) H ∼halka ise

$$\eta_H : H \longrightarrow Mat_{n\times n}(H)$$

$$h \longmapsto \begin{pmatrix} h & & 0 \\ & h & \\ & \ddots & \\ 0 & & h \end{pmatrix}_{n\times n}$$

bir halka homomorfizmi elde edilir.

DT2) $f : H \longrightarrow K$ halka homomorfizmi için,

$$\begin{array}{ccc} H & H & \xrightarrow{\eta_H} Mat_{n\times n}(H) \\ \downarrow f & \downarrow f & \downarrow G(f) \\ K & K & \xrightarrow[\eta_K]{} Mat_{n\times n}(K) \end{array}$$

diyagramı değişmelidir. Çünkü

$$\eta_K \circ f = G(f) \circ \eta_H$$

eşitliği sağlanır.

BÖLÜM 2

PROGRAMLAMA DİLLERİ ve FONKSİYONEL PROGRAMLAMA

2.1 Giriş

Bu kısım programlamanın temel özellikleri ile fonksiyonel programlamanın öne çıkan özelliklerini içermektedir. Bu bölümdeki detaylar için [5],[6] ve [7] referanslarına bakınız.

Bilgisayar programlama, insan tarafından yapılacak herhangi bir işin veya hesaplamanın kısa bir sürede, enerji harcamadan gerçekleştirilmesini sağlayan bir yöntemdir. Gerek sürenin kısaltılması gerek de yazılım maliyetleri açısından çeşitli programlama dilleri ortaya çıkmıştır. Nesne tabanlı, yordamsal, bildirimsel, fonksiyonel programlama genel adları altında toplanan diller farklı yaklaşımlar ile en etkin programı yazmayı amaçlamışlardır. Hem çağın gelişen teknolojisi hem de en zor problemlerin çözüm yöntemleri açısından bilgisayarlar ortak bir çalışma ortamı olarak düşünüldüğünde büyük gelişmeler katetmiştir.Özellikle çok çekirdekli modern bilgisayarların üretilmesi ile farklı programlama mantıkları büyük önem kazanmıştır. Çok çekirdekli bilgisayarlar ile eşzamanlı ortak iş yapılması hedeflenmiştir. Dolayısıyla ortaya çıkan birçok programlama dili bu açıdan düşünüldüğünde, uyumlu veya uyumsuz birçok farklı yönteminin olduğu ortaya çıkar. Çekirdek sayısının az olduğu durumlarda etkin bir şekilde çalışabilen bir programlama dili, çekirdek sayısının artmasıyla bu etkinliğini kaybedip bazı sorunlara yol açabilir. Dolayısıyla yeni çekirdek yapıları ve eşzamanlı çalışma prensibiyle bu sistemin altyapısını oluşturabilecek uygun bir programlama dili gereklidir.

Aynı dönemde doğmuş diller bir dil kategorisi çerçevesinde toplanmaya çalışılır. Fakat bu diller belirli özellikleriyle gruplandırılabilirler. Bu gibi sınıflandırmalar da yapılırken çeşitli kriterler kullanılabilir. Bu kriterlerde benzer özellikler gizlenerek farklı yapısal özellikler ön plana çıkarılır. Genel olarak programlama dilleri iki tip sınıflandırma altında incelenirler:

1) Genel Sınıflandırma

-Yordamsal (Imperative) Programlama Dilleri

·Fortran, C, Cobol, Basic, Pascal

-Veriye Yönelik Programlama Dilleri

·Lisp, APL, Snobol, Icon

-Nesneye Yönelik (Object-Oriented) Programlama Dilleri

·Simula, C++, Ada95, Java, Visual Basic(VB)

-Fonksiyonel Programlama Dilleri

·Miranda, ML, F#, HASKELL.

2) Uygulama Alanlarına göre Sınıflandırma

-Bilim ve Mühendislik Dilleri

·Fortran, C, Pascal

-Sistem Programlama Dilleri

·C, Assembler

-Veri Tabanı Dilleri

·Dbase, Clipper

-Yapay Zeka Dilleri

·Prolog, Lisp

-Genel Amaçlı Programlama Dilleri

·C, Pascal, Basic

Programlama dillerini gösterdikleri özellikler bakımından ailelere ayırmak gerekirse, programlama dil aileleri meydana gelir.

1) Etkileşimli Diller

Bir etkileşimli dil; genel anlamda sistem içerisinde kolay program akışını ve kod çalışmasını sağlayan dillerdendir. Bir REW döngüsü içerir. Bu döngü, Read;okuma evresi, Evaluate; çalışma evresi, Write; sonuç yazma evresi içerir.

Etkileşimli dili kullanan bir program genel olarak, kısa fonksiyon ve nesne tanımlamaları içerir. Programlar genellikle ara form olarak kullanılır. Bu ara formlar daha sonra dönüştürülür. Forth ve Miranda gibi etkileşimli diller 'derleme' sözcüğünü, kaynak kodu makine diline çevirmek için kullanılırlar. APL, Basic, Forth, Lisp, Scheme, dBase ve Miranda bu tip dillerdendir.

2) Yapısal Diller

Kontrol yapıları, 'GOTO' deyimi kullanmaksızın program yazımını sağlayan dillerdir. Parametre tarafında çağırma olayı daha çok kullanılır. Pascal dili 'GOTO' deyimini kullanmasına rağmen bir yapısal dil olarak kabul edilir. Çünkü bu deyimin kullanılmasına da gerek kalmaz. Pascal, C, Forth, Lisp, T ve Scheme bu dil ailesindendir.

3) Kuvvetli Tipten Diller

Nesneler isim olarak kullanılır ve her isim bir tiptir. Herbir nesne sadece bir tipe aittir. Fortran77, Pascal, Ada, ANSI C, ML, Miranda bu tip dillere girer.

4) Nesneye Yönelik Diller

Tip dillerinin genellemesi veya genişlemesi şeklindedir. Nesneler tiplerdir ve her zaman tip özelliklerini taşırlar. Herhangi bir fonksiyon; farklı metot tanımlamaları yardımıyla çeşitli şekillerde yazılabilir.

Nesne tabanlı diller; ayrık olmayan tipler ve fonksiyon kalıtımı özelliklerini barındırır. Bir tip başka bir tipten üretilebilir. Simula, Smalltalk, T, C++ nesne tabanlı dillerdendir.

5) Yöntemsel Diller

İfadeler dizisinden oluşan bir dildir. Değişkenler yardımıyla bu ifadeler birbiriyle bağlantı kurar. Pascal, C, Fortran, Basic, Cobol bu tip dillerdendir.

6) Fonksiyonel Diller (Eski nesil)

İç içe geçmiş ifadeler ve fonksiyonlardan oluşan programlara sahiptir. Değer parametresi erişimi ile atama yapmayarak ilkel bir mekanizma oluşur.

7) Fonksiyonel Diller (Yeni Nesil)

Yeni nesil gelişen fonksiyonel diller atama dizisi, yıkıcı yapılar içermez. Değerler; programın bir parçasından fonksiyon çağırımları ve parametre bağıntıları ile sağlanır.

Eski ve yeni nesil fonksiyonel programlama arasındaki en önemli fark; işlem yaparken eski nesilin değer parametresi ile çağırması; yeni nesilin ise ihtiyaç duyuldukça (tembel dil) çağırma işlemini gerçekleştirmesidir. İhtiyaç duyuldukça çağırma; sonsuz listelerin çalışmasını sağlayan en önemli ilerlemedir. Böylece programın ihtiyacı olan parçası işleme sokulur.

Fonksiyonel diller; objeleri, fonksiyonları ve fonksiyon uygulamalarını ifade etmek için kullanılan en temiz dillerden biridir.

2.2 Fonksiyonel Programlama

Özellikle eşzamanlı çalışma yöntemi ile oluşabilecek en büyük tehlike olan yan-etki sorununu çözmesi bakımından fonksiyonel programlama dilleri aranan özelliklere en yakın dillerdendir.

Fonksiyonel programlama, kodlarını matematiksel fonksiyonlar üzerinde geliştiren bir programlama örneğidir. Program yapısının fonksiyonlara dayanması diğer programlama dillerinden özel ve eşsiz olarak ayrılmasını sağlar.

Fonksiyonel programlama dilleri:

—Programcıya algoritma ile ilgili farklı, matematiksel bir bakış açısı sunar.

—Programlar, ifadelerin biraraya uygun bir şekilde toplanması ile oluşmuştur.

—Değişken değişimi veya atama gibi değişkenlerin durumunu değiştiren ifadeler yer almadığından yan etki gözlenmez.

—Matematiksel fonksiyon yapısından kaynaklı olarak diğer dillere göre daha kısa ve anlaşılabilir kodlar yazılabilir.

Bu özellikleri ile fonksiyonel programlama dilleri birçok programcının karşılaşabileceği sıkıntıları en aza indirir.

Genel olarak fonksiyonel programlama dilleri;

- Yüksek dereceden fonksiyonlar,
- Saf fonksiyonlar,
- Tembel hesaplama,
- Polimorfizm,
- Tekrarlama yapıları,
- Tip sistemleri

gibi diğer dillerden ayrılacak özelliklere sahiptirler.

Fonksiyonel diller, 1960 öncesinde LISP dilinin oluşturulmasıyla programlama mantığına başka bir bakış açısı ortaya çıkmıştır. Bu mantık yardımıyla ML, Haskell, Scheme, F#, Erlang... gibi diller geliştirilmiştir.

1) Yüksek Dereceden Fonksiyonlar

Yüksek dereceli fonksiyonlar, parametre olarak fonksiyonların alınıp çıkış parametresinin de fonksiyon olarak geri döndüğü yapılardır. Bu sayede fonksiyonlar yardımıyla daha karmaşık yapılı fonksiyon yapıları üretilebilir. Yani nesnel dillerdeki gibi atama ifadesi yerine fonksiyonlar kullanılarak fonsiyonel dillerin yapısına uyacak şekilde karmaşık yapılar kolayca ortaya çıkar.

2) Saf Fonksiyonlar

Saf fonksiyonlar, herhangi bir yan-etki göstermeyen fonksiyonlardır. Yan-etki ise, bir işlem sonucunda dönecek değere ulaşmak için program yapısında meydana gelebilecek

değişikliklerdir. Örneğin; bir fonksiyonun herhangi bir değişkeninin değeri değişmiş ve diğer bir fonksiyonun yapısını değiştirmiş ise yan-etki ortaya çıkmış demektir. Yan-etki genel olarak, programın dış dünya ile iletişiminde, örneğin dosya sistemleri veya aynı ağdaki diğer bilgisayarlara ulaşma gibi durumlarda ortaya çıkar. Yordamsal programlama yan-etkinin en sık karşılaşıldığı programlama türüdür. Fakat fonksiyonel programlamada örneğin, ML, Scheme, Scala, Lisp gibi diller yan-etki durumunu kısıtlamazlar. Haskell gibi fonksiyonel programlama dili ise tip sisteminden dolayı yan-etkileri kısıtlar ve izin vermez. Yan-etkinin özellikle istendiği durumlar için ise monad adı verilen yapı kullanılır.

3) Tembel Hesaplama

Programlama dili teorisinde herhangi bir ifade tanımlandığında değil ihtiyaç duyulduğunda hesaplanmasını sağlayan yöntemdir. Bu yöntem ile program gereksiz tekrar hesaplamaların önüne geçer. Bu sayede hesap süresi önemli ölçüde azalır. Program performansı artış gösterir ve sadece ihtiyaç duyulan ifadelerin çalıştırılması, diğer dillere göre bir avantaj sağlar. Tembel hesaplama sayesinde, sonsuz veri yapılarının dilde yazılabilmesi sağlanır. Böylece sonsuz veri yapısına uygulanacak işlemler verinin tamamına uygulanmasa da dilde yazılabilmesine izin verilir. Ayrıca sadece gerekli ifadelerin hesaplanması hafızada yer sorununu en aza indirir.

4) Polimorfizm

Polimorfizm, farklı veri tiplerini düzgün bir sistem içinde tutmayı sağlayan önemli bir programlama dili özelliğidir. Parametrik polimorfizm ise, polimorfizmin hem veri tiplerine hem de fonksiyonlara uygulanan başka bir türüdür. Fonksiyonel programlamanın tip sistemi parametrik polimorfizm üzerine yapılandırılmıştır. Yani, bir fonksiyon veya veri tipi oluşumu sırasında belirli bir tipe bağlılık olmaz. Böylece benzer yapıları sürekli oluşturmak yerine üretici bir yapı ile istenilen tipler üzerinde benzer oluşumlar sağlanmış olur.

5) Tekrarlama Yapıları

Tekrarlama, program içerisinde adım adım aynı ifadenin kendi içinde döngü oluşturacak şekilde oluşan bir yöntemdir. Fonksiyonel programlamada nesnel programlama dillerindeki gibi if-else, while, for gibi yapılara ihtiyaç olmadığından, fonksiyonun kendi içinde tekrar tekrar çağırılacak şekilde bir tekrarlama yapısı oluşturulabilir. Birbiri ardına aynı fonksiyonun çağırıldığı açık örneklerden biri Fibonacci dizisinin elde edilmesidir. Bu dizi elde edilirken nesnel dillerde for döngüsü tercih edilir. Fakat Haskell, ML gibi fonksiyonel programlama dil-

lerinde yapının bel kemiği olan fonksiyonun tanımlanması ile istenilen dizi kolayca elde edilir. Örneğin Haskell dilinde bu dizi;

```
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

fonksiyonu ile elde edilir. İlk iki kod satırı bitiş koşullarını ifade eder. Tekrarlama yapısının asıl gerçekleşeceği kısım son satırdaki kod ifadesidir. n değeri yerine yazılacak pozitif sayıdan başlayarak 0 ve 1 sayılarına ulaşana kadar **fib** fonksiyonu sürekli çağırılır.0 ve 1 bitiş değerleri bu fonksiyonun sonlanacağı değerler olur. Bu tekrarlama yapısının kolaylıkla kurulacağı başka bir örnek de faktöriyel hesabıdır. Benzer şekilde bitiş koşulu tekrarlamayı sona erdirecek şekilde seçilir ve asıl yapılmak istenen iş fonksiyon olarak açık bir şekilde ifade edilir.

6) Tip Sistemleri

Fonksiyonel programlamanın güçlü özelliklerinden birisi de tip sistemleridir. Cebirsel veri tipleri, durum ifadeleri ve tanımlar yardımıyla yeni bir tip sistemi oluşturulabilir. Belli bir tip sisteminin asıl amacı, program derlenirken oluşabilecek hatalar konusunda programcıya yardımcı olmaktır. Algoritmadan bağımsız olarak oluşturulan programın kullanacağı uygun tipler seçilir. Bu seçim düzgün ve güvenli bir tip sistemi tarafından düzenlenir ve kontrol edilir.

2.3 Fonksiyonel Programlama ile Nesnel Programlamanın Karşılaştırılması

Fonksiyonel programlama gerek yapısı gerek işleyişi bakımından diğer programlama dillerinden oldukça farklıdır. En önemli farkların başında fonksiyonel programlamanın yan-etkileri engelleyici özelliği gelir. Yan-etki, nesnel programlama mantığında özellikle Giriş/Çıkış uygulamalarında sıkça ortaya çıkan bir durumdur. Saf fonksiyonel programlama ise yan-etki durumlarını tamamen engelleyerek, programları ortak kullanmayı, algortima yapısını doğrulatmayı sağlayacak önemli bir özellik kazanır.

Bu farklar arasında örneğin, fonksiyonel diller basit bir yazım kuralına sahiptir. Dolayısıyla bu özellik öğrenmesi ve kullanması kolay bir dil oluşturur. Nesne tabanlı diller fonksiyonel dillerin bu özelliğinin tam tersine karmaşık yazım yapısıyla öne çıkar. Bu iki zıt özellik dilleri

üretkenlik açısından iki ayrı uca iter. Bir fonksiyonel dilin ortak bir problem üzerindeki çözüm bulma hızı yordamsal bir dile göre önemli ölçüde fazladır. Bu hız fonksiyonel dillerin programları kısa bir şekilde ifade edebilmesinden kaynaklanıyor gibi gözükse de aslında fonksiyonel dil yapısının problemlere uygun cevaplar verebilmesinden kaynaklanır. Bu hıza etki edecek diğer bir faktör de programlama dili ile bilgisayarın mimarisi arasındaki uyumdur.

Genel olarak bilgisayar mimarisi tasarımı iki yaklaşım üzerinde yoğunlaşmıştır. Günümüzde tüm bilgisayarlarda kullanılan mimari yapısı, 1903-1957 yılları arasında yaşamış ünlü matematikçi, Quantum fizikçisi Jon Von Neumann tarafından geliştirilmiştir. Von Neumann mimarisinde, veri bellekten alınıp işledikten sonra tekrar belleğe gönderilmesinde çok zaman harcanır. Bu işlemler bilgisayarı yavaşlattığından, bilgisayar tasarımcılarının tabiriyle bir darboğaz oluşturmaktadır. Bu mimari yaklaşımına sahip olan bilgisayarlar günümüzde verilerin işlenmesinde, bilginin derlenmesinde ve sayısal problemlerde olduğu kadar endüstriyel denetimlerde başarılı bir şekilde kullanılmaktadır. Diğer taraftan başka bir mimari türü de Harvard Üniversitesi'nde geliştirildiği için Harvard mimarisi denen yapıdır. Harvard mimarili bilgisayar sistemlerinde, Von Neumann mimarisinden farklı olarak veri ve komutlar ayrı belleklerde tutulmaktadır. Buna göre, veri ve komut aktarımında iletişim yolları da birbirinden bağımsız yapıdadır. Standart bir bilgisayar Von Neumann mimarisine sahiptir ve belleğin bir kısmını komutlar bir kısmını da veriler için ayırır. Bazı programlarda komut sayısı fazla iken diğer programlarda veri sayısı fazla olabilir. Bu durumda belleğin tek olması ihtiyaç dahilinde komutlar için veya veriler için fazla yer ayrılmasını sağlar. Bu ana sebep nedeniyle Harvard mimarisi bilgisayar uygulamalarında uygulama alanı bulamamıştır.

Hızlı çalışan bir program oluşturabilmek için, uygun bir dil ile uyumlu donanım tasarımı yani bilgisayarın mimarisiyle bir bütünlük oluşturur. Bunun yanı sıra algoritmanın etkin kullanımı yani okunabilirlik ve kısa yapısı da programlama hızında önemli bir yere sahiptir. Örneğin C dilinde

```
int sum_cubes(int n){
    int sum = 0;
    for(int index = 1; index <= n; index++)
       sum += index * index * index;
    return sum;
}
```

şeklinde program parçası Haskell dilinde

```
sumCubes n = sum (map (^3) [1..n])
```

tek satır koduyla hızlı çalışma gösterir.

Yordamsal diller ile eşzamanlı çalışma yapmak oldukça zor bir uygulamadır. Eşzamanlı çalışabilecek parçalara ayrılabilmesi için değişmez bir bölme yöntemiyle kullanmak gerekir. Yani program 'static' değişkenler üzerine inşa edilmelidir. Fonksiyonel dillerde ise programı parçalara ayırabilmek kadar doğal bir olay yoktur. Fonksiyonların birbirinden bağımsız olması, yan-etki durumlarının ortaya çıkmasını engeller. Dolayısıyla fonksiyonel dille yazılmış herhangi bir program kolaylıkla eşzamanlı çalışabilecek şekilde parçalara ayrılabilir. Yordamsal dille yazılmış bir programda değerler büyüdükçe başlangıçta belirlenen değişken tiplerinde güncelleme gerekli olur. Aksi takdirde programın istenilen cevabı vermesi güçleşir ve hatta hata ile sonuçlanabilir. Fonksiyonel programlamada ise böyle bir güncellemeye program ne kadar uzarsa uzasın gerek duyulmaz. Bu iki mantık arasındaki farkı somutlaştırmak için C++ yordamsal dili ile Haskell fonksiyonel programlama dilinin bir program üzerindeki yazılışına bakılırsa,

```
qsort [] = []
qsort (x:xs) = (qsort elts_lt_x)++ [x] ++ (qsort elts_greq_x)
     where elts_lt_x = [y | y <- xs, y < x]
           elts_greq_x = [y | y <- xs, y >= x]
```

programı basit bir sıralama işlemini Haskell dili ile oluşturur. Burada herhangi bir atama deyimi yer almaz.

```
qsort( int a[], int lo, int hi )
{ int h, l, p, t;
 if (lo < hi) {
   l = lo; h = hi; p = a[hi];
    do {
       while ((l < h) && (a[l] <= p)) l = l+1;
       while ((h > l) && (a[h] >= p)) h = h-1;
       if (l < h) { t = a[l]; a[l] = a[h]; a[h] = t; }
     } while (l < h);
```

```
      t = a[l]; a[l] = a[hi]; a[hi] = t;
      qsort( a, lo, l-1 );
      qsort( a, l+1, hi );
     }
    }
```

programına bakılacak olursa, karmaşık, uzun ve yan-etki oluşturabilme gücü yüksek bir algoritma dikkat çeker.

Haskell dili ile oluşturulan quicksort algoritması için programın çalıştırılmasıyla

```
> qsort [2,5,3,12,-1]
 [-1, 2, 3, 5, 12]
```

sonucu elde edilir.

Yine yordamsal ve fonksiyonel diller arasındaki farkı açıkça gösterecek karşılaştırma Haskell ile Java dilleri arasındadır. Nesne tabanlı programlama dili olan Java ile fonksiyonel programlama dili Haskell arasındaki farklılıklar aşağıdaki tablo ile verilebilir.

HASKELL	JAVA
1) Fonksiyon ve etkiyi birbirinden ayırır, fonksiyonların yan-etki üretmesini engeller.	Bütün metodları yan-etkiye sahiptir. Bir çoğu bağlı olduğu objeyi değiştirir.
2) Fonksiyonlar sadece sonuçları kullanacağı zaman hesaplama yapar.	Metodlar yazıldığı andan itibaren her zaman hesaplanır.
3) Veri tipleri ve fonksiyonlar arasındaki bağ sıkı değildir.	Her bir metod bir sınıf yapısının parçasıdır.
4) Program çalıştırılırken yeni fonksiyonlar oluşturabilir.	Metodlar program için yazıldığında kullanılabilirler.
5) Fonksiyonlar fonksiyonları kullanarak çok değerli yeni fonksiyonlar üretebilirler.	Çok değerli fonksiyonlar alacağı bütün değerleri bir kere alır.

BÖLÜM 3

HASKELL

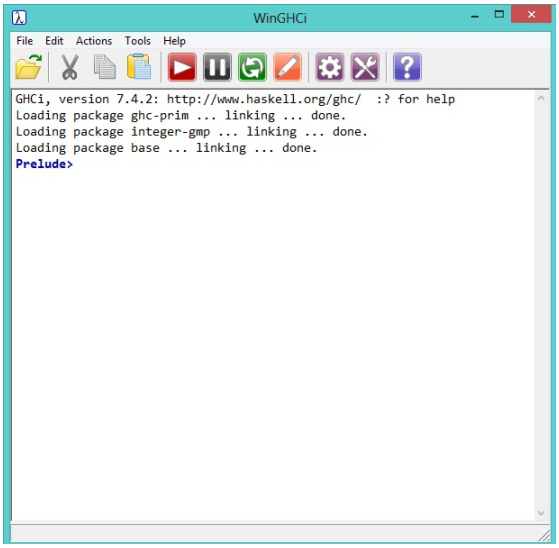
3.1 Giriş

Bu bölüm Haskell fonksiyonel programlama dilinin temel yapısal özelliklerini içermektedir. Bir matematikçi olan Haskell Curry tarafından geliştirilen bu dil matematiksel tabanı ile diğer dillerden ayrılmaktadır. Tembel değerlendirme, monadlar ve tür sınıfları gibi yapılar farklı özellikler sunmaktadır. Bu bölümdeki detaylar için [1],[4] ve [2] referanslarına bakınız.

3.2 Programın yüklenmesi ve kullanımı

Haskell, fonksiyonel programlama dilidir. Haskell üzerinde kod yazabilmek için '**Haskell Platform**' yardımıyla derleyici bilgisayara kurulur. Bu kurulum sayesinde GHCi veya WinGHCi derleyicileri kullanılabilir hale gelir. İnsanların bilgisayarı kontrol edebilmesini sağlayan kodlar derleyici yardımıyla makine diline dönüştürülür.

İlk olarak Haskell derleyicisi çalıştırıldığında gerekli paketler bu derleyicinin açılması ile yüklenmeye başlanır. Komut satırında 'Prelude>' derleyici ön ekinin görünmesiyle GHCi derleyicisi komut girmeye hazırdır. Derleyicinin görüntüsü şeklideki gibidir.



Haskell fonksiyonel programlama dili temel olarak 'Prelude' isimli çekirdek kütüphanesinden yararlanır. Haskell programı her açıldığında öncelikle varsayılan olarak bu kütüphane belirlenir ve yüklenir. Bu yükleme ile program bünyesinde temel tipleri ve sık rastlanan faydalı fonksiyonları barındırır. Haskell dilinin temel yapısını oluşturan 'Prelude' modülü ile çeşitli programlar elde edilir.

Dört işlem gibi kısa ve basit derlemeler direk olarak derleyici üzerinde yazılarak, hemen sonuç alınabilir.

Örneğin,

```
Prelude> 2+2
4
Prelude> 5*4+3
23
Prelude> 2^5
32
```

3.3 Haskell Programlamada Temel Yapılar

3.3.1 Değişkenler

Haskell dilinde, bazı ifadelerinin kullanımı için gerekli bir isimlendirmeden başka birşey değildir. Bundan dolayı 'değişken' sözcüğü bu dilde kullanıldığında aldatıcı olur. Çünkü bir Haskell programı içerisinde oluşturulan bir değişken asla değerini değiştirmez. Bunun yerine, matematiksel yaklaşıma daha yakın olacak şekilde kullanılır. İspatların matematiksel hesabı içerisinde değerlerin değişmemesi özelliği bu dildeki değişkenler için de geçerlidir. Örneğin;

```
r=5
alan=pi*r^2
r2=3
alan2=pi*r2^2
```

kodlarında $r2$ değişkeninin tanımlanmasının sebebi, yordamsal dillerden (C,C++,...) farklı olarak Haskell dilinin atama deyimine sahip olmamasıdır. Dolayısıyla Haskell, yapısında sabit

değişkenleri barındırır. Yani değişken yoktur.

Haskell programının hesaplama özelliği kısa programlar için oldukça faydalıdır. Fakat uzun programlar ve hesaplamalar için ara değerlerin tutulması gerekir. Ara değerlerin tutulması ise değişken tanımlama yardımıyla sağlanır.

Değişken, ismi ile çağırıldığında tuttuğu değeri gösteren hafıza gibi düşünülebilir. Basit olarak;

```
Prelude>3.1416 * 5^2
78.53999999999999
```

hesabı kolayca yapılabilir. Aslında yapılan hesap $A = \pi r^2$ formülü ile 5 yarıçaplı dairenin alanından başka birşey değildir. Fakat π sayısı hatırlanması zor ve kullanışsız olacak birçok sayıdan ibarettir. Burada $\pi \approx 3.1416$ biçiminde kullanılmıştır. Aslında Haskell 'pi' şeklindeki kullanım ile tekrar eden ve hafızada tutulamayacak uzunluktaki sayıyı değişken olarak belirlemiştir. Böylece

```
Prelude> pi
3.141592653589793
Prelude> pi * 5^2
78.53981633974483
```

ile daha duyarlı bir hesap yapılmış olur.

Haskell derleme dosyaları 'hs' uzantısını kullanırlar. Not defteri veya başka bir metin editörü yardımıyla oluşturulan bir 'a.hs' derlemesi GHCi komut satırına ':load a.hs' derlemesi ':l a.hs' komutunun verilmesiyle kullanılabilir hale gelir.

İsmi 'a.hs' olarak seçilen Haskell dosyasına yalnızca

```
r=5.0
```

ifadesi yazılsın.

```
Prelude> :load a.hs
Compiling Main      ( a.hs, interpreted )
Ok, modules loaded: Main.
Main>
```

Yukarıdaki komutun verilmesiyle derleme işlemi tamamlanır ve r ifadesi (değeri) kullanılabilir hale gelir.

```
Prelude> r
5
Prelude> pi*r^2
78.53981633974483
```

'a.hs' belgesi üzerinde yapılan değişiklikler tekrar **:load** komutunun kullanılmasıyla aktif olacaktır.

```
r=5
alan=pi*r^2
Prelude> alan
78.53981633974483
Prelude> alan/r
15.707963267948966
```

benzer işlem üzerinde değişiklik yapılmayacak derlemeler için 'let' komutu kullanılarak komut satırı üzerinde de yapılabilir.

```
Prelude> let alan=pi*r^2
```

Ancak bu tip basit işlemler haricinde her zaman dosyalarla çalışmak gerekir.

3.3.2 Fonksiyonlar

Matematiksel olarak, bir fonksiyon A kümesindeki bütün değerleri B kümesindeki değerlerle ilişkilendiren yapıdır. Haskell dilinde ise, istenilen tip tanımlamaları ile bu matematiksel yapı Haskell fonksiyonu özelliği kazanır. Örneğin;

```
f :: Int -> Int
f x = x * x
```

yapısına bakalım. İlk satırdaki ifade, ‘f’ ismiyle tanımlanan fonksiyonun ‘Int’ tamsayı tip özelliğine sahip değerleri alır ve ‘Int’ tipli değerlerine döndürür. Genel yapısı bu şekilde oluşturulan fonksiyonun hangi işlemi yapacağı ise ikinci satırda belirtilir. Fonksiyon isminden sonra gelen ‘x’ değişkeni ‘Int’ tipli alınan değerdir. Sonuç olarak değerin kendisiyle çarpımından oluşan ‘Int’ tipli değer ortaya çıkar. Basit şekilde tanımlanan ‘f’ fonksiyonu gibi farklı koşullar ve özellikler kazandırılarak istenilen çeşitli yapılar oluşturulabilir.

Bir fonksiyon parametre değeri (veya argümenti) alarak geriye bir sonuç döndürür. Haskell dilinde fonksiyon yapısını tanımlamak için ise;

```
alan :: Float -> Float
alan r = pi*r^2
```

şeklinde fonksiyonda değişecek parametre, fonksiyon isminin yanına boşluk bırakılarak yazılır. Böylece ‘alan’ fonksiyonunun tanımına bakılırsa;

```
Prelude> alan 5
78.53981633974483
Prelude> alan 3
28.274333882308138
Prelude> alan 17
907.9202768874502
```

hesaplamaları ile tek bir alan formülü ile farklı yarıçaplı dairelerin alanları hesaplanabilir.

Fonksiyonlar iki, üç parametre ile de tanımlanabilirler. Metin editöründe

```
ort :: Int -> Int
ort a b = (a + b)/2
```

şeklinde tanımı ile

```
Prelude> Ort 2 4
3
```

kodlarında olduğu gibi iki parametre kullanılarak ortalama fonksiyonu yazılabilir.

Fonksiyon koşullarını düzenli bir şekilde belirleyebilmek için model eşleme yöntemi en çok kullanılan yöntemlerden biridir. Bu yöntem ile parametre seçimine göre farklı sonuçlar ortaya çıkar. Örneğin;

```
bes :: Int -> Bool
bes 5 = True
bes x = False
```

fonksiyonunu göz önüne alalım. Bu fonksiyon 5 sayısını parametre olarak kullanıp kullanmadığını test eder. Aranan sayı 5 parametre olarak girilmişse fonksiyon True, diğer sayıların girildiği durumlarda ise False yanıtını verecektir. Yani 5 sayısı True cevabı ile diğer sayılar ise False cevabı ile eşlenmiş olur.

Fonksiyonlar genel olarak ‘f a b ’ şeklinde yazım ile kullanılırlar. Fakat bu yazım yerine farklı kullanımlar da Haskell dilinde mevcuttur.

```
Prelude> 2+2
Prelude> 2==True
```

gibi deyimlerde sağ ve sol taraftaki ifadeler aradaki operatör yardımıyla bir sonuca ulaşır. Bu şekilde iki ifadenin ortasında kullanılan operatörler ‘**infix operatörler**’ olarak adlandırılır. Benzer şekilde argümentlerden önce kullanılan operatörler de ‘**prefix operatörler**’ olarak bilinir. ‘prefix operatörler’i parantez içerisinde ifadenin başında gösterilir.

```
Prelude> 4+9 == 13
True
Prelude> (==) (4+9) 13
True
```

ifadeleri sırasıyla **infix** ve **prefix** operatörler yardımıyla aynı işlemlerin gösterimleridir.

3.3.3 Karşılaştırma Operatörleri

Haskell de verilen iki ifadenin eşit olup olmadığını kontrol etmek için yazılmış fonksiyonlar vardır. Bu fonksiyonlar karşılaştırma operatörleri olarak adlandırılırlar. '==' operatörü iki ifadenin eşit olup olmadığını kontrol eder.

```
Prelude> 2+3 == 5
True
```

'<, <, <=, >=' operatörleri de '==' operatörüne benzer şekilde çalışır.

```
Prelude> let alan = pi*r^2
Prelude> alan 5 < 50
False
```

3.3.4 Mantıksal Operatörler

• (&&) operatörü 've' bağlacı görevini görür. Her iki ifade doğru ise True, diğer durumlarda False yanıtını verir.

```
Prelude> (3<8) && (False == False)
True
Prelude> (&&) (6<=5) (1==1)
False
```

• (||) operatörü 'veya' bağlacı görevini görüp, ifadelerden herhangi birinin doğru olması durumunda True, diğer durumlarda False yanıtını verir.

```
Prelude> (2+2==5) || (2>0)
True
Prelude> (||) (18=17) (9>=11)
False
```

• not, operatörü verilen ifade True ise False, False ise True yanıtını verir.

```
Prelude> not (5*2==10)
False
```

3.3.5 Guard Yapısı

Guard yapısı nesne tabanlı ve imperative (buyurgan) dillerdeki if-else ve switch yapılarına benzer şekilde birden fazla durum içerisinden seçim yapılarak işlem yapmak için kullanılır. Bu yapının nasıl çalıştığını anlamak için basit anlamda **mutlak değer fonksiyonu**nu düşünelim. Eğer sayı sıfırdan küçük ise işareti değiştirilir, diğer durumlarda değişmeden sayı aynen kalır. Tanımlama,

$$|x| = \begin{cases} x, & x \geq 0 \\ -x, & x < 0 \end{cases}$$

şeklinde yapılır. Eğer sayı sıfırdan büyük eşit ise ilk ifade, küçük ise ikinci ifade işletilir. Bu yapıyı Haskell programlama diline taşıdığımızda ise

```
abs x
      | x<0       = 0-x
      | otherwise = x
```

kodları ile guard yapısı kullanılarak oluşturulur.

Guard yapısının Haskell dilindeki genel kullanımı şu şekilde olur:

• Fonksiyon ismi yazılarak fonksiyon tanımı yapılacakmış gibi parametresi boşluk bırakılarak yanına yazılır.

• '=' ifadesi yerine alt satıra geçilerek boşluk ile olası durumlar şart olarak eklenir.

• Her bir koşulun doğru olması durumunda gerçekleşecek sonuçlar, koşulun sonuna '=' sembolü ile eklenir.

• Yazılan koşulların dışında kalan son durum için 'otherwise' yani 'diğer durum'lar için şart eklenerek yapı tamamlanır.

where ifadesi guard yapıları ile kullanıldığı zaman kullanışlı olur. $ax^2 + bx + c = 0$ şeklindeki 2.dereceden denklemin reel çözümlerini incelenmeye çalışılırsa;

```
cozum a b c
       | dis >0   =2
       | dis ==0  =1
       |otherwise =0
            where dis = b^2-4*a*c
```

Haskell diline çevrilmiş olur.

3.4 Tipler

Programlama dillerinde benzer değerleri gruplandırmak için veri tipleri sıklıkla kullanılırlar. Haskell programlama dili güçlü bir tip sistemine sahip olup bu sistem hata ayıklamasının da önemli bir parçasıdır.

Derleyici tanımlanan fonksiyonun izin verilen tipler dışında kullanımını engellemeye çalışır. Örneğin; Bir karakter ile sayının toplanabilmesi mümkün değildir.

```
2+3
```

işleminin sayılar üzerinde tanımlanacağı açıktır. İsim sorulup 'Merhaba' yanıtı alınacağı bir fonksiyon için ise String'ler kullanılır. Haskell'de tüm tip isimleri mutlaka büyük harfle başlar. Farklı tipler bir araya getirilerek veritabanı gösterimi yapılabilecek yeni tipler oluşturulabilir.

Bu örneklerden tiplerin, verileri kategorileştirme yolu olduğu anlaşılır.

Herhangi bir ifadenin hangi tipte olduğunu anlamak için :t (veya :type) komutu kullanılır. Örneğin;

```
Prelude> :t True
True :: Bool
Prelude> :t (3<5)
(3<5) :: Bool
Prelude> :t 'H'
'H' :: Char
```

```
Prelude> :t "Merhaba"
"Merhaba" :: [Char]
```

Son ifadeden anlaşılacağı gibi String tipi Haskell'de karakterlerden oluşan listeye karşılık gelir. Değer tipleri yanısıra Haskell dilinde diğer dillerden farklı olarak fonksiyonların da tipleri vardır. Örneğin; '**not'** fonksiyonu Bool değerleri içinde değilleme işlemi yapar. Yani; Bool tipli değişkeni alır ve Bool tipli değişkene dönüştürür. Haskell dilindeki gösterimi;

```
not :: Bool -> Bool
```

şeklindedir. Tipler için kullanılan :t tip sorgulama komutu benzer şekilde fonksiyonlar için de kullanılır. Fonksiyonlarda tip gösterimi yanlış parametrelendirmenin önüne geçerek hata denetimi yapar.

```
Prelude> :t ort
ort :: Int -> Int -> Int
```

ortalama fonksiyonunu gözönüne alalım. 'ort' fonksiyonu için, gösterimdeki ilk iki Int tipleri bu fonksiyon için gerekli parametreleri; sonuncu 'Int' tipi ise sonuç olan değerin tipini belirtir. Böylece 'ort' fonksiyonu için eksik parametre seçimi gibi hatalar fonksiyonun tip gösterimi tarafından engellenir.

3.5 Tip Sınıfları

Tip sınıfları, Haskell dilinin tip sisteminin kuvvetini arttıran önemli bir özelliktir. Tip sınıflarının genel amacı, belirli fonksiyonlarda seçilen tipler için uygunluğunu sağlamaktır. Bir tip, tip sınıfının içinde yer alıyorsa bu tip sınıfının kurallarını sağladığı anlamına gelir.

Eşitlik kontrolü yapan == fonksiyonunun yapısına bakılırsa

```
Prelude> :t (==)
(==) :: (Eq a) => a -> a -> Bool
```

tip gösterimi ile karşılaşılır. Bu yazımda ‘Eq a’ == fonksiyonunun ait olduğu tip sınıfını belirtir. Bu tip sınıfı

```
class Eq a where
      (==) :: a-> a -> Bool
      (\=) :: a -> a -> Bool
      x == y = not (x\=y)
      x \= y = not (x==y)
```

şeklinde fonksiyonlar içerir. Dolayısıyla Haskell tip sistemindeki birçok tip için eşitlik koşulu bu tip sınıfı ile kontrol edilir.

Bir Haskell programı içerisinde kullanılması uygun olan tiplere paralel olarak tip sınıfları belirtilir. Örneğin, sayılar için ‘Num’ tip sınıfını kullanmak en uygun kullanımdır. Reel sayıların kullanıldığı bir programda ise ‘Fractional’ tip sınıfı belirtilerek fonksiyonların tip çerçevesi belirlenir. Yani kullanılması uygun olan tipler için tip sınıfları, fonksiyonların genel yapısını belirler.

3.5.1 Bazı temel tip sınıfları:

‘Eq’ tipler için eşitliklerini kontrol etmek için kullanılır. Fonksiyon temel olarak == ve /= kontrol ifadelerini kullanır.

```
Prelude> 5 == 5
True
Prelude> 5 /= 5
False
Prelude> 'a' == 'a'
True
Prelude> "Ho Ho" == "Ho Ho"
True
ghci> 3.432 == 3.432
True
```

‘Ord’ tipleri sıralamak için kullanılan bir tip sınıfıdır.

```
Prelude> :t (>)
(>) :: (Ord a) => a -> a -> Bool
```

Daha önce görülen birçok fonksiyon 'Ord' tip sınıfının bir parçasıdır. Ord fonksiyonu bütün standart $>$, $<$, $>=$ ve $<=$ karşılaştırma fonksiyonlarını içerir. Ordering 'GT', 'LT' veya 'EQ' karşılaştırmalarını içeren bir tiptir. Burada karşılaştırmalar sırasıyla büyüklük, küçüklük ve eşitlik durumlarına karşılık gelir. Örneğin

```
Prelude> "Abrakadabra" < "Zebra"
True
Prelude> "Abrakadabra" `compare` "Zebra"
LT
Prelude> 5 >= 2
True
Prelude> 5 `compare` 3
GT
```

şekline karşılaştırmalar ifade edilir. 'Show' fonksiyonunun elemanları string ler ile ifade edilir. Şimdiye kadar görülen birçok tipi içerir. En fazla kullanılan fonksiyon 'Show' tip sınıfının 'show' fonksiyonudur.

```
Prelude> show 3
"3"
Prelude> show 5.334
"5.334"
Prelude> show True
"True"
```

'Read', Show tip sınıfının bir çeşit karşıtıdır. 'read' fonksiyonu bir string değerini alır ve read fonksiyonunun bir elemanı olan tipe dönüştürür.

```
Prelude> read "True" || False
True
Prelude> read "8.2" + 3.8
```

```
12.0
Prelude> read "5" - 2
3
Prelude> read "[1,2,3,4]" ++ [3]
[1,2,3,4,3]
```

read ”4” yazımına bakılırsa;

```
Prelude> read "4"
<interactive>:1:0:
 Ambiguous type variable ‘a’ in the constraint:
  ‘Read a’ arising from a use of ‘read’ at <interactive>:1:0-7
 Probable fix: add a type signature that fixes these type variable(s)
```

Burada program ‘read’ fonksiyonunun hangi tip değişkenine değer döndüreceğini bilmediğinden hata mesajı vermiştir. ‘Read’ sınıfına bakılırsa;

```
Prelude> :t read
read :: (Read a) => String -> a
```

:: kullanılarak istediğimiz tipi belirtebiliriz:

```
Prelude> read "5" :: Int
5
Prelude> read "5" :: Float
5.0
Prelude> (read "5" :: Float) * 4
20.0
Prelude> read "[1,2,3,4]" :: [Int]
[1,2,3,4]
Prelude> read "(3, ’a’)" :: (Int, Char)
(3, ’a’)
```

‘Enum’ tip sınıfının elemanları ardışık olarak dizilmiştir. ‘Enum’ tip sınıfının en önemli özelliği liste gösterimlerini kullanabilmesidir. Ayrıca ‘succ’ ve ‘pred’ fonksiyonları kullanılarak hangi tipin önce kullanılacağını belirtebiliriz. Bu sınıftaki tipler: (), Bool, Char, Ordering, Int, Integer, Float ve Double dır.

```
Prelude> ['a'..'e']
"abcde"
Prelude> [LT .. GT]
[LT,EQ,GT]
Prelude> [3 .. 5]
[3,4,5]
Prelude> succ 'B'
'C'
```

Bounded bu sınıftaki üyelerin sınırlarını belirtir.

```
Prelude> minBound :: Int
-2147483648
Prelude> maxBound :: Char
'\1114111'
Prelude> maxBound :: Bool
True
Prelude> minBound :: Bool
False
```

‘minBound’ ve ‘maxBound’ ; (Bounded a) => a şeklinde tipler içerdiğinden çok biçimli yapılardır. Bütün ikililer bileşenleri Bounded sınıfında ise bu tip sınıfının elemanıdırlar.

```
Prelude> maxBound :: (Bool, Int, Char)
(True,2147483647,'\1114111')
```

Num sayısal bir tip sınıfıdır. Elemanları sayıların özelliklerine sahip olmalıdır.

```
Prelude> :t 20
20 :: (Num t) => t
```

Bu program ise bütün sayıların çok biçimli yapılar olduğunu gösterir.

```
Prelude> 20 :: Int
20
Prelude> 20 :: Integer
20
Prelude> 20 :: Float
20.0
Prelude> 20 :: Double
20.0
```

Bu tiplerin hepsi 'Num' tip sınıfının elemanlarıdır. *, işlemine bakılacak olursa,

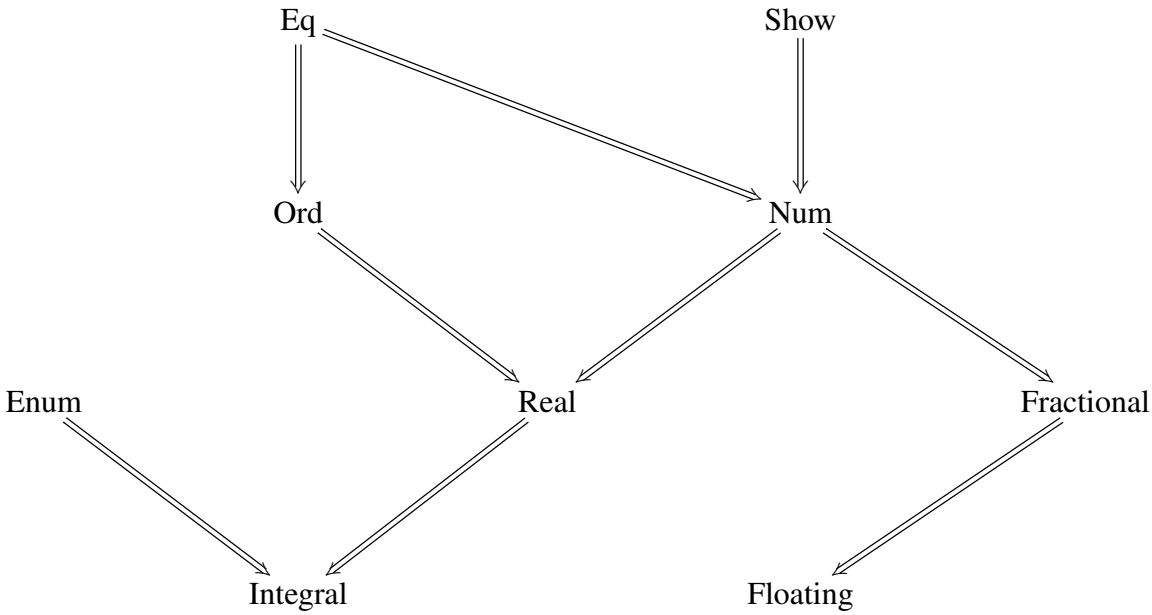
```
Prelude> :t (*)
(*) :: (Num a) => a -> a -> a
```

Aynı tipten iki sayıyı alır ve bu tipteki sayıya dönüştürür. Bu yüzden

```
(5 :: Int) * (6 :: Integer)
```

ifadesi hata verir. Fakat 5 * (6 :: Integer) düzgün bir şekilde çalışır. Çünkü 5 sayısı 'Integer' veya 'Int' olarak kullanılabilir. Integral de ayrıca bir sayısal tip sınıfıdır. Integral sadece **bütün tamsayıları** içerir. Bu tip sınıfında Int ve Integer yer alır.

Floating Float ve Double gibi ondalıklı sayıları içerir. Şekil 3.1 de Haskell'de kullanılan temel tip sınıfları ve bu tip sınıflarının ilişkisi verilmiştir.



Read
Şekil 3.1

3.6 Listeler ve Tuples

Listeler ve tuples lar birçok değerin birarada tutulduğu Haskell dili yapılarındandır.

3.6.1 Listeler

Haskell dilinde liste kavramı matematikteki küme kavramına benzer şekilde kullanılır. Kümeden farklı olarak bir eleman liste içerisinde birden fazla kez kullanılabilir. Liste içerisindeki tüm elemanlar aynı tipten olmak zorundadır. Listeler oluşturulurken elemanlar ”[ ]” kullanılarak verilirler. Boş liste oluşturmak mümkündür. Örneğin aşağıdaki kodlar farklı tiplerden oluşan listeler tanımlar.

```
Prelude> let sayi = [1,3,4,5]
Prelude> let deger = [True,False,False]
Prelude> let kelime = ["bazi ","kelimeler"]
```

Farklı tipteki elemanlar bir liste içerisinde birlikte bulundurulamaz. Farklı tipten elemanlar içeren bir liste oluşturulmaya çalışıldığında derleyici aşağıdaki gibi bir hata verecektir.

```
Prelude> let lst = [True, "dogru"]
<interactive>:1:19:
Couldn't match 'Bool' against '[Char]'
Expected type: Bool
Inferred type: [Char]
In the list element: "dogru"
In the definition of 'lst': lst = [True, "dogru"]
```

Var olan bir listeye eleman eklemek için (:) fonksiyonu kullanılabilir.

```
Prelude> let sayi = [1,3,4,5]
Prelude> sayi
[1,3,4,5]
Prelude> 2: sayi
[2,1,3,4,5]
```

(:) fonksiyonunda işlem önceliği sağdan sola doğrudur. Ayrıca, yeni eklenen eleman listenin soluna değil en başına eklenir.

```
Prelude> 1:0: sayi
[1,0,2,1,3,4,5]
```

Oluşan listenin yapısına bakacak olursak, boş liste yani [] sembolüne sayıların (:) fonksiyonu ile eklendiği anlaşılır. Yani [1,2,3] listesi ile 1:2:3:[] listesinin eş olduğu görülür.

String tipi yapısı Haskell'de karakterlerin bir listesi olarak düşünülür.

```
Prelude> "sayi" == ['s','a','y','i']
True
Prelude> "sayi" == 's':'a':'y':'i':[]
True
```

kodlarından ”sayi” stringinin aslında 4 elemanlı bir karakterler listesi olduğu görülür.

Örneklerde verilen liste yapıları haricinde iç içe kullanılan farklı listeler de mevcuttur. Liste ve tuple yapıları iç içe kullanılarak farklı özellikteki veri yapıları elde edilir.

```
Prelude> :t [[1],[1,2]]
[[1],[1,2]]:: [[t]]

Prelude> :t [(1,'2'),(1,'a')]
[(1,'2'),(1,'a')]:: [t,Char]

Prelude> :t ([1,2],[3,4])
([1,2],[3,4]):: ([t],[t1])
```

Oluşturulan bu liste ve tuplelar Haskell dilinde yeni birer tip oluşturur. Bu tür yapılar bazı karmaşık verilerin ifadesinde büyük kolaylık sağlar.

Listelerde listenin elemanları **head, tail ve last** gibi fonksiyonlar yardımıyla elde edilebilir. **head** fonksiyonu listenin ilk elemanını verirken, **tail** fonksiyonu ise listenin geri kalan kısmını gösterir. **last** ise listenin son elemanını gösterir. Bu fonksiyonların

```
Prelude> 2:[7,5,0]
[2,7,5,0]
Prelude> head [2,7,5,0]
2
Prelude> tail [2,7,5,0]
[7,5,0]
Prelude> last [2,7,5,0]
0
```

şeklinde kullanımı vardır.

3.6.2 Tuples

Tuples; matematikteki kartezyen çarpıma benzer bir yapıdadır. Farklı tipteki elemanlar liste içerisinde kullanılmamasına rağmen tuples içerisinde bulunabilir. Tuples; birçok elemanın tek bir değer olarak saklanabildiği yapılardır.

Tuples oluşturulurken ”( )” işareti kullanılır.

```
Prelude> let a = (True,1)
Prelude> let b = (4,5,"bes")
```

Tupleslar oluşturulurken tüm elemanları belirtilmelidir. Listeler gibi sonradan eleman eklemek mümkün değildir. Farklı tipten birden fazla çıkış parametresi alan fonksiyonlarda da tupleslar sıklıkla kullanılırlar. Tupleslar iç içe kullanılabileceği gibi bir liste ve bir tuple başka bir tuple ın elemanı olabilir. Örneğin;

```
Prelude> let c = ((2,"iki"),(True,3))
Prelude> let d = ([1,2,2,3],False)
```

c ve d değişkenleri tuple ve listeyi tuple içinde kullanarak yeni bir tip oluşturur.

Listelerdeki gibi ikililerde de her bir eleman bazı fonksiyonlar yardımıyla elde edilebilir. Bunlardan biri **fst,** ikili çiftindeki ilk elemanı gösteren bir fonksiyondur. Bir diğeri ise snd, ikinci elemanı ekranda gösterir.

```
Prelude> fst (8,11)
8
Prelude> snd ("Wow", False)
False
```

Haskell dili, polimorfik tip özelliği sayesinde Char, Int, Bool,... gibi tipler için ayrı ayrı fonksiyon tanımlamak yerine tek bir fonksiyon tanımlayarak programın gereksiz tekrarlarını engeller. Örneğin; **length** fonksiyonunun yapısına bakalım:

```
Prelude> :t length
length :: [a] -> Int
```

köşeli parantez içindeki **a** harfi bir tip değildir. Çünkü tip isimleri büyük harfle başlayan kelimelerle ifade edilir. Bunun yerine, o harf bir tip değişkenini sembolize eder. Haskell programında **a** sembolüyle karşılaşıldığında, bu değişken yerine herhangi bir tipin gelebileceği anlaşılır. Bu ise tamamen dilin polimorfik özelliğine bir çözüm sunar. Örneğin;

```
f :: a -> a
```

ifadesi herhangi bir tipteki değerin alınıp, yine aynı tipte bir değere görüntülendiğini ifade eder. Bu yapıdan farklı olarak,

```
f :: a -> b
```

ifadesi ise alınan ve dönen değerin aynı tipte olmak zorunda olmadığını gösterir.

Bu yaklaşım ile **fst** ve **snd** fonksiyonları düşünüldüğünde;

```
fst :: (a,b) -> a
snd :: (a,b) -> b
```

ifadeleri bu fonksiyonların genel yapısını ifade ederek kullanım kolaylığı sağlar.

3.7 Tekrarlama

Tekrarlama, Haskell dilinde önemli bir role sahiptir. Bu yapı ile fonksiyonların kendi içinde tekrar tekrar kullanılması sağlanır.

Tekrarlama öncelikle bir başlangıç veya bitiş koşulunu barındırmalıdır. Yanlış bir şekilde oluşturulmuş fonksiyonlar sonsuz döngü yapısına dönebileceğinden doğru bir biçimde tanımlanmalıdır.

Tekrarlama genel olarak iki bölümden oluşur. Öncelikle, bir veya iki temel durum ile sonlandırma koşulları belirtilmelidir. Bu koşulların sağlanması durumunda fonksiyon sonlandırılır.

En bilinen tekrarlama, faktöriyel fonksiyonudur. Bu fonksiyon negatif olmayan değeri alır ve bu değere kadar olan pozitif sayıların çarpımını hesaplar. Bu döngü genel olarak, sayının bir eksiğinin çarpıma dahil edilmesiyle ve 0 sayısı için özel bir tanımlama yapılır. Negatif sayıların çarpıma dahil olmaması için 0 değeri bu fonksiyonun sonlandırma koşulu olur. Haskell dilinde bu tanımlama

```
faktoriyel 0 = 1
faktoriyel n = n * faktoriyel (n-1)
```

kodlarıyla oluşturulur.

'faktoriyel 3' değerini, tekrarlama yapısını ayrıntılı çalıştırarak görelim:

• 3 değeri sıfırdan farklı olduğu için ilk satırdaki koda uyumluluk göstermez.

• İkinci adımda 'faktoriyel 3 = 3*faktoriyel (2)' kodu çalıştırılır.

• 'faktoriyel 2' hesabında tekrar 2 değeri sıfırdan farklı olduğundan ikinci ifade çalıştırılır ve 'faktoriyel 2 = 2* faktoriyel (1)' elde edilir.

• Son adım olarak benzer şekilde 'faktoriyel 1 = 1* faktoriyel (0)' elde edilerek 'faktoriyel 0' yerine ilk ifade çalıştırılır. Dolayısıyla,

$$\begin{aligned} faktoriyel\ 3 &= 3 * faktoriyel(2) \\ &= 3 * 2 * faktoriyel(1) \\ &= 3 * 2 * 1 * faktoriyel(0) \\ &= 3 * 2 * 1 * 1 \end{aligned}$$

hesabı 'faktoriyel' fonksiyonunun tanımı ile hesaplanmış olur.

Şimdi de listeler üzerinde sıkça kullanılan 'length' fonksiyonunu gözönüne alalım. Haskell dilindeki tanımı;

```
length :: [a] -> Int
```

```
length [] = 0
length (x:xs) = 1+ length xs
```

şeklindedir.

İlk satır fonksiyonun herhangi tipteki listenin değer olarak alınıp bir tamsayının sonuç olarak görüntülendiğini ifade eder. İkinci kod satırı; boş listenin bir sonlandırma koşulu olarak belirlendiğini gösterir. En son satır ise, genel olarak fonksiyonun ifadesinden başka birşey değildir.

3.8 Liste İşlemleri

3.8.1 Map fonksiyonu

Haskell programında listeler üzerindeki etkin yapılardan biri de **map** fonksiyonudur. Bu fonksiyon;

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = (f x) : map f xs
```

biçiminde tekrarlayan yapıyı içerir.

Bu fonksiyon ile f fonksiyonunun kuralı xs listesindeki her bir eleman üzerine tek tek uygulanır.

```
f :: a -> a
f x = 2^x
```

şeklinde tanımlanan f fonksiyonunun map ile [1,2,3] listesine uygulandığında oluşan değer;

```
Prelude> map f [1,2,3]
[2,4,8]
```

olarak elde edilir.

3.8.2 Fold Fonksiyonu

Fold fonksiyonu, map fonksiyonu ile aynı görevi yapar. Fakat map ile bir liste oluşurken, fold ile bu değerlerin fonksiyon ile değerleri bulunup toplanarak tek bir değer oluşturur. Toplam fonksiyonunu ele alalım:

```
toplam :: [Integer] -> [Integer]
toplam [] = 0
toplam (x:xs) = x + toplam xs
```

şeklinde tanımlanır. Benzer şekilde çarpım için de;

```
carpim :: [Integer] -> [Integer]
carpim [] = 1
carpim (x:xs) =x * carpim xs
```

olarak tanımlanır.

Bu fonksiyonlar, değerleri birbiri üstüne yığılarak oluşturulan yapılardır.

Standart Prelude modülünde **foldr,foldl, foldr1,foldl1** şeklinde dört farklı fold fonksiyonu türü vardır.

- **foldr**

Bu fonksiyon sağdan birleşmeli özelliğe sahip olup listenin sağından başlayarak fonksiyon kurallarını uygulayıp yığılma gerçekleştirir.

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f acc [] = acc
foldr f acc (x:xs) = f x (foldr f acc xs)
```

yapısına sahiptir.

- **foldl**

foldr fonksiyonundan farklı olarak soldan birleşme özelliğine sahiptir.

• **foldr1 ve foldl1**

foldr1 ve foldl1 fonksiyonları foldr ve foldl fonksiyonlarının yapılarıyla benzer şekilde çalışır. Sadece açık bir başlangıç değerine ihtiyaç duyulmaz. Dönüştürülecek listenin ilk (veya son) elemanını başlangıç değeri olarak görür.

3.8.3 Filter Fonksiyonu

Listeler üzerindeki en yaygın işlem filter işlemidir. Belli bir koşulu sağlayan elemanlar topluluğunu içerir. Sadece çift sayılardan oluşan bir liste düşünülürse, bu yapıyı oluşturabilmek için,

```
isEven :: Int -> Bool
isEven n = ((mod n 2)==0)

ciftlst ns= filter isEven ns
```

şeklinde kod yazımı ile yapılabilir.

3.9 Modüller

Haskell modülleri birbirine bağlı fonksiyonların tek bir pakette gruplandığı ve farklı fonksiyonların aynı isim altında toplandığı yapılardır.

Modüle isim vermek ilk yapılan işlemdir:

```
module Modul where
```

Bu yazım içerisinde modül isminin büyük harfle başlamasına ve her bir dosyanın tek bir modül yapısı içermesine özellikle dikkat edilmelidir.

'Modul' isimli modülün yer alacağı dosya da 'Modul.hs' şeklinde bir yazımda olmalıdır.

Modüller, başka modüller tarafından çağırılıp yüklenebilme özelliğine sahiptirler.

```
import Data.Char(toLower,toUpper)
import Data.List
import Modul
```

kodlarıyla yükleme yapılır.

İlk kod, Data.Char modülündeki ‘toLower’ ve ‘toUpper’ fonksiyonlarının çağırıldığını gösterir. İkinci kod, Data.List modülünün tamamının çağırılması anlamına gelir. Son kod ise, oluşturulan ‘Modul’ isimli modülün tamamen yüklendiğini ifade eder.

İki modül içerisinde aynı fonksiyon ismi geçiyorsa ve bu iki modül de dosyaya yüklenmek isteniyorsa Haskell bu durumda hangi fonksiyonun tercih edileceğine karar veremez. Bu durumda,

```
import qualified Modul
```

kodu ile fonksiyonun kısımda

```
Modul.fonk
```

şeklindeki yazım ile ‘Modul’ modülünün fonksiyonunun işlevini yapacağı anlaşılır.

Bir modül içerisindeki belli bir fonksiyon kullanılmak istenmiyorsa;

```
import Modul hiding(fonk)
```

kodu ile bu işlem gerçekleştirilir.

Modül isimlerinin uzun belirlendiği bir durumda ise yeniden isim verilerek bu yükten kolayca kurtulunabilir. Diğer bir gösterimle ,

```
import qualified Modulfonkicerir as M
```

kodu modülün ‘M’ ismiyle kullanılabileceğini gösterir.

```
module Modul(
     fonk,
     islem,
     baginti) where
```

yapısı modülün içinde tanımlanacak fonksiyon isimlerini ifade eder. Daha sonra fonksiyonların oluşturulmasıyla bu isimlerin görevleri belirlenmiş olur.

3.10 Input&Output

Fonksiyonlar yardımıyla birçok işlem Haskell dili üzerinde belirli yapılarla sağlanabilmiştir. Fakat bu fonksiyonların kullanıcıya ne yaptığı ile ilgili bir görüntü sunulması gerekir. Basit olarak input&output denilen çıktı fonksiyonlarına ihtiyaç vardır. Örneğin,

```
main = do
    putStrLn "isminizi giriniz: "
    name <- getLine
    putStrLn ("Merhaba" ++ name)
```

programı somut olarak sonuç alınabilen yapıdadır.

Input&Output fonksiyonları yan etki içerdiklerinden fonksiyonel diller içerisinde yapı olarak tercih edilmezler. Pure fonksiyonlar; aynı parametreler için aynı sonucun görüntülenmesini amaçlar. Fakat örnekte olduğu gibi ”getLine” fonksiyonunun aynı değeri döndürmesi fonksiyonel dillerin özellikle Haskell dilinin sorunlarındandır.

Input&Output kütüphanesi bir program için fonksiyonların tutulduğu yerler olarak düşünülebilir. Yani,

- Bir string değerinin ekranda görüntülenmesi,

- Girilen değerin okunması,

• Verinin dosyaya işlenmesi,

• Bir dosyadan verinin okunması

işlemleri bu kütühane yardımıyla sağlanır.

Ekrana görüntüleme işlemi, String tipinde değerin alınması ile başlar. Fakat her çıktı durumunda aynı değer döneceğinden fonksiyon yapısı bu aşamada kullanılamaz. Değerin ekrana görüntülenmesi, bir dosyadan verinin okunması gibi fonksiyonlar 'etki' olarak tanımlanabilen 'putStrLn' eylemleri ile gerçekleştirilir. Bu eyleme isim vermekten çok özel bir tip ile bu 'etki' tanımlanır.

Bu etki;

```
putStrLn :: String -> IO()
```

tipindedir. Bu fonksiyon 'etki' tanımıyla bir string değeri alıp IO() tipinin bir değerine döner.

Etkinin çalışma prensibi ise şöyledir: Derleyici main ana fonksiyonunu çalıştırırken IO() tipli değerlerle işlem görür. Bu tip sayesinde de IO etkisi değer döndüremeyecek şekilde çalışır. Fakat bu şekilde her bir etki tek tek çalıştırılamaz. Bu yüzden bir etki dizisi yaratılmalıdır.

İlk örneğe bakılacak olursa, 'do' gösterimi bir etkiler dizisini birarada toplar. '<-' sembolü etkinin değerini bir değişken yardımıyla tutma yoludur. putStrLn, getLine ve diğer PutStrLn fonksiyonlarının etkileri ile program çalışarak istenen çıktı ekrana yansıtılır.

3.11 Yüksek Dereceli Fonksiyonlar

Yüksek dereceli fonksiyonlar, fonksiyonları değer olarak alan fonksiyonlardır. **map** gibi bazı fonksiyonlar aslında yüksek dereceli fonksiyonlardır. Bu yapı, fonksiyonel programlamaya bir çeşit soyutlama özelliği kazandırır. Haskell gibi fonksiyonel programlama dillerinde fonksiyonların diğer değerlerden bir farkı yoktur.

Hızlı çalışan algoritmalardan biri olan **quickSort** programına bakılırsa, ana fikir oldukça basittir. Listeden bir eleman seçilir, bu elemana göre liste üçe ayrılır. İlk liste seçilen elemandan küçü olan elemanları, ikinci liste eşit olanları, üçüncü liste kendisinden büyük olan elemanları

içerir. Oluşturulan listelere **quickSort** algoritması bir kez daha uygulanırsa istenilen sıralama elde edilir.

```
quickSort [] = []
quickSort [x] = [x]
quickSort (x:xs) =(quickSort kucuk) ++ (x:esit) ++ (quickSort buyuk)
              where kucuk = filter (<x) xs
                    esit = filter (==x) xs
                    buyuk = flter (>x) xs
```

algoritmasıyla hızlıca sıralanmış bir liste elde edilir.

Bu fonksiyonun tipi

```
(a -> a -> Ordering) -> [a] -> [a]
```

şeklindedir. Çoğunlukla fonksiyonun tipi diğer fonksiyonların nasıl kullanılacağı hakkında bilgi verir. **quickSort** fonksiyonu için (a -> a -> Ordering) tipli fonksiyonu ve herhangi tipli listeyi ele alan dönüş değeri olarak yine parametre olarak alınan listenin elemanlarından oluşan bir liste verir. Bu yapıdan sıralama elde ederek listeye etki eden ve parametre olarak alınan fonksiyon açık olarak belirlenebilir.

```
(a -> a -> Ordering)
```

yapısı tek bir parametre olarak fonksiyonu temsil eder. Parantezler ihmal edilerek yazılırsa,

```
a -> a -> Ordering -> [a] -> [a]
```

dört parametreli bir fonksiyon halini alır. Dolayısıyla yüksek dereceli fonksiyon olmaktan çıkar.

3.11.1 Curry Fonksiyonu

Curry fonksiyonu, çok değişkenli fonksiyonun belli bir kısmına uygulanan bir tekniktir. quickSort fonksiyonu üzerinde karşılaştırma olarak kullanılan kısım curry fonksiyonu ile

değiştirilerek çeşitli quickSort yapıları elde edilebilir. Değiştirilmiş quickSort fonksiyonu ise özel karşılaştırma fonksiyonunu kullanarak liste üzerine uygular. İlk yazılan quickSort fonksiyonuna bakılırsa,

```
less = filter (<x) xs
```

tanımı ile xs listesinin x elemanından küçük elemanları filtrelemeden geçirilip 'less' listesi oluşturulur. Halbuki (<x) yazımı sadece (\y -> y<x) yapısının kısaca yazılmış gösterimini açıkça ifade eder.

Bu tür kolaylıklar '$' sembolünün kullanımında da karşılaşılır. '$' sembolünün sağındaki eleman işlem önceliğinde en son sırada yer alır. Matematiksel olarak fonksiyonların argümenti gibi görev görür.

```
map ($2) [(2*),(4*),(8*)]
```

koduna bakılırsa, fonksiyon işlemleri listenin elemanı gibi gözükür. Fakat her bir fonksiyon '$' sembolünün sağındaki 2 elemanına uygulanarak yeni bir liste elde edilir.

3.12 Monad

Monad yapısı ilk olarak Haskell dilinde input&output sorununu çözmek için oluşturulmuştur. Daha sonra, temel hesaplama yapısıyla işlem sırası önceden kestirilemeyen işlemler sorun çıkarmaya başlamıştır. Dosyayı okuma ve dosyaya yazma gibi işlemlerin sırası daha da kritik bir durum olmuştur. Fonksiyonlar dizisinin sırasını belirlemek için özelleşmiş method yapıları sunan Monadlar ihtiyacı karşılayan yapılardır.

Bu yapı sadece input&output yapısı için değil aynı zamanda herhangi yordamsal dili de bu şekilde modelleyebilir.

Bir monad 3 özellik içerir:

- Bir M tip yapıcı,
- return fonksiyonu,

- (>>=) operatörü (bağlayıcı)

şeklindedir.

Bu fonksiyon ve operatör tanımı:

```
return :: a -> M a
(>>=) :: M a -> (a -> M b) -> M b
```

şeklinde ifade edilir.

Maybe monad yapısına bakalım:

Tip yapıcı olarak M=Maybe alınır. 'return' ve (>>=) tipleri de bu yapıda mevcuttur.

```
return :: a -> Maybe a
(>>=) :: Maybe a -> (a -> Maybe a) -> Maybe b
```

Bu yapıların kuralı:

```
return x = Just x
(>>=) m g = case m of
                  Nothing -> Nothing
                  Just x -> g x
```

Maybe monad yapısı ve (>>=) fonksiyonunun faydaları için örnek üzerinde düşünelim:

Bir aile veritabanını ele alalım:

```
baba :: Person -> Maybe Person
anne :: Person -> Maybe Person
```

fonksiyonları ile veritabanında bulunan anne ve baba isimleri kontrol edilmektedir. Bu veritabanında tutulmuyor ise 'Nothing' değerini geri döndürmektedir. Örneğin; anne soyundan gelen büyükbaba durumuna bakılırken;

```
bbaba :: Person -> Maybe Person
bbaba p =
      case anne p of
            Nothing -> Nothing
            Just m -> baba m
```

kodları kullanılır. Ya da herhangi soydan gelen büyükbaba ismini belirlemek için

```
buyukbaba :: Person -> Maybe (person, Person)
buyukbaba p =
           case baba p of
                 Nothing -> Nothing
                 Just f ->
                    case baba f of
                          Nothing -> Nothing
                          Just gf ->
                             case anne p of
                                 Nothing -> Nothing
                                 Just m ->
                                    case baba m of
                                       Nothing -> nothing
                                       Just gm ->
                                           Just (gf,gm)
```

şeklindeki kodlar ile işlem yapar. Bu kodlar içerisinde kendini tekrarlayan birçok kod Maybe monad yapısı ile basitleştirilir.

```
bbaba p = anne p >>= baba
```

yazımı ile (>>=) kullanımı daha önce yazılan ‘bbaba’ fonksiyonunu sade bir şekilde tek satırda oluşturur.

‘Lambda’ ifadesi ve return fonksiyonu yardımıyla ‘buyukbaba’ fonksiyonu tekrar yazılırsa;

```
buyukbaba p =
        baba p >>=
            (\ f -> baba f >>=
                (\gf -> anne p >>=
                    (\m -> baba m >>=
                        (\gm -> return (gf,gm))))
```

elde ediilir.

Haskell dilinde Monad tip sınıfı; Monad yapısının uygulandığı, **Control.Monad** modülü ve **prelude** ana modulünün içinde tanımlanan bir sınıftır.

```
class Monad m where
        return :: a -> m a
        (>>=) :: m a -> (a -> m b) -> m b
        (>>) :: m a -> m b -> m b
        fail :: String -> m a
```

şeklinde kod yazımından oluşmaktadır. Burada farklı olarak, $(>>)$ operatörü ve fail fonksiyonu yer alır. $(>>)$ operatörü,

```
m >> n = m >> =\_ -> n
```

işlemi ile sonraki elemanı görüntüler. Bu fonksiyonlardan $(>>)$; iki monaddan doğan etkinin sıralamaya sokulması işlevini görür. Bu fonksiyon ile ikinci etki birinci etkinin sonucuna bakılmaksızın uygulanır.

fail fonksiyonu ise, 'do' yapısı ile hataların tutulmasını sağlar.

Kısaca monad yapısı ile yordamsal dilin özellikleri ifade olarak fonksiyonel programlama diline aktarılır, sanki yordamsal bir dilde yorumlanıyormuş gibi işlem yapılır.

3.12.1 Monad Kuralları

Bir Monad tip sınıfının uyması gereken 3 kural vardır:

```
m >>= return = m    (sag birim)
return x >>= f = f x   (sol birim)
(m >>= f) >>= g = m >>= (\x -> f x >>= g)    (birlesme ozelligi)
```

Bu kurallara uyan yapı bir Monad değildir. Ancak Monad tip sınıfının instance özelliği taşıyan örneği olabilir.

Monad tip sınıfının kurallarına uyabilecek tipler ancak intance yapısıyla oluşturulabilir.

Bu kuralların sağlanması ile

```
(f .g) . h = f. (g . h)
f . id = f = id . f
```

özellikleri sağlanmış olur.

3.12.2 Maybe Monad Yapısı

Maybe tipi tek bir elemanla veya hiçbir elemanla işlem görür. Yani;

```
data Maybe a = Nothing | Just a
```

şeklinde bir tip yapıcıdır. Yani değer var ise 'Just a' kalıbına sokar, yok ise 'Nothing' değerini döndürür.

Bu yapısıyla Maybe Monad'ı hiçbir değerin dönmeyebileceği programlar açısından oldukça faydalıdır. Maybe Monad'ı için;

```
return :: a -> Maybe a
return x = Just x
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
(>>=) m g = case m of
                Nothing -> Nothing
                Just x -> g x
```

monad fonksiyonları tanımlanır. Örneğin;

```
Prelude> fmap (++"!") (Just "dikkat")
Just "dikkat!"
Prelude> fmap (++"!") (Just "dikkat")
Nothing
```

kodlarına bakarsak; fmap fonksiyonundan sonra gelen ifade herhangi bir fonksiyon yapısıdır. Bu fonksiyon Maybe tipindeki değere uygulanır. Bu program monad fonksiyonları biçiminde yazılırsa;

```
>(Just "dikkat") >>= \x -> Just (++"!")
Just "dikkat!"
> Nothing >>= \x -> Just (++"1")
Nothing
```

şeklinde ifade edilir.

3.12.3 Liste Monad Yapısı

Listeler; aynı tipteki değerler topluluğudur. Liste monad yapısı bakımından, tek bir değer veya birden fazla değer döndürmesiyle Maybe monad yapısından ayrılır.

Monad yapısıyla listelere göz atarsak;

```
instance Monad [] where
      return x = [x]
      xs >>= f = concat (map f xs)
      fail _ = []
```

fonksiyonları ile karşılaşılır. 'return' fonksiyonu x elemanını liste monad yapısına aktarır. (>>=) fonksiyonu; f fonksiyonunun xs listesi üzerinde uygulanmasıdır. 'fail' fonksiyonu ise boş listenin dönebileceği değerler için geçerlidir. Fakat bu özellikleri ile liste monad yapısı keyfi değerler kümesi oluşturur. Örneğin;

```
Prelude> [3,4,5] >>= \x -> [x,-x]
[3,-3,4,-4,5,-5]
```

program parçasına bakarsak; (>>=) fonksiyonu listedeki her bir elemandan; değişmeden kendisi veya negatifi ile oluşturulmuş bir liste meydana getirir. Yani her bir değer için fonksiyon kuralı ile iki değer oluşur. Yani ilk aşamada

```
[[3,-3],[4,-4],[5,-5]]
```

listesi elde edilir. Bu liste concat fonksiyonu ile tek bir liste haline getirilerek keyfi değerler bir listede toplanmış olur.

Verilen monad yapıları şeklinde düşünülerek IO Monad yapısı, State Monad yapısı (global durum) ve diğer tip sınıfları da elde edilir.

BÖLÜM 4

KATEGORİ ve HASKELL

4.1 Giriş

Bilgisayar programları yazılırken kullanılan algoritmaların matematiksel seviyesi programın çalışma hızına ve stabilitesine doğrudan etki etmektedir. Örneğin asimetrik şifreleme sistemleri oluşturulurken büyük sayıların, büyük kuvvetlerinin mod değerlerine bakılmaktadır.

$$91518013^{20981287} \bmod 363794227$$

ifadesi hesaplanmak istendiğinde akla gelen ilk algoritma 91518013 sayısını 20981287 kez kendisiyle çarptıktan sonra elde edilen sayının mod değerine bulmak olur. Bu işlemi sembolik hesaplama alanında en güçlü programlardan birisi olan Maple ile yapılmak istenirse aşağıdaki gibi bir sonuç ortaya çıkar.

MAPLE

```
> 91518013^20981287 mod 363794227;
Error, object too large
```

Bunun anlamı şudur; kullanılan matematiksel algoritma sonucu anlamlı bir süre içerisinde hesaplayamayacağından program işleme hiç başlamaz. Bu sorunu ortadan kaldırmak için matematiksel yaklaşımımızı geliştirmek gerekir. Yeni bir algoritma aşağıdaki şekilde gösterilebilir.

$a^x \bmod n$ ifadesinde x sayısı ya çift sayı ya da tek sayıdır. Eğer x çift ise $x = 2k$ olup $(k \in \mathbb{Z})$

$$a^x = a^{2k+1} = (a^k)^2$$

elde edilir. Yani a ifadesini hesaplamak için x kez çarpım yapmak yerine $\frac{x}{2} + 1$ kez çarpım yapılmış olur.

Eğer x tek ise $x = 2k+1$ olup $(k \in \mathbb{Z})$

$$a^x = a^{2k+1} = a^{2k}.a = a^{(k)^2}.a$$

elde edilir. Yani a ifadesini hesaplamak için x kez çarpım yapmak yerine $\frac{x}{2}+2$ kez çarpım yapmış oluruz. $k \in \mathbb{Z}$ olduğundan aynı işlemler k sayısının tek sayı veya çift sayı olma durumuna göre tekrar edilirse çok daha az çarpımla sonuca ulaşılır.

Pozitif her sayının 2 nin kuvvetlerinin toplamı olarak yazılabildiğinden

$$20981287 = 16777216+4194304+8192+1024+512+32+4+2+1$$

olup

$$\begin{aligned} a^{20981287} &= a^{2^{24}+2^{22}+2^{13}+2^{10}+2^9+2^5+2^2+2^1+2^0} \\ &= a^{2^{24}}.a^{2^{22}}.a^{2^{13}}.a^{2^{10}}.a^{2^9}.a^{2^5}.a^{2^2}.a^{2^1}.a^{2^0} \end{aligned}$$

bulunur. Yani 20981287 kez çarpım yapmak yerine $9+24=33$ kez çarpım yapılarak bu ifade hesaplanabilir.

Bu algoritma kullanılarak yazılan bir program yardımıyla Maple da hesaplama yapılırsa

MAPLE

```
> 91518013 &^ 20981287 mod 363794227;
13021820
```

işlemin sonucunu 13021820 olarak bir saniye bile geçmeden bulabiliriz. Asimetrik şifrelerin, günümüzdeki uygulamalarında, kullanılan sayıların en az 156 basamaklı olduğu düşünüldüğünde yazılan algoritmaların matematiksel seviyesinin bilgisayar bilimine etkisi açıkça gözler önüne serilmektedir. Yazılan her bilgisayar programı bir matematiksel gerçekliğe dayanmalıdır. Matematiğin gelişmesiyle, yazılan programlar daha hızlı çalışacak şekilde geliştirilebilir.

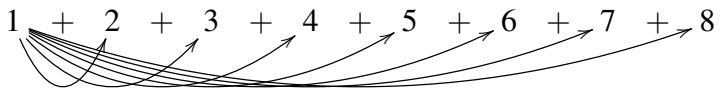
Bilgisayar bilimi üzerine çalışan bilim adamlarının güçlü algoritmalara sahip programlar yazılabilmesi amacıyla matematik üzerine yoğunlaşan ilgisi fonksiyonel programlama dillerinin yapı temelinde herhangi bir kategori ile çok benzerlikler gösterdiğini ortaya çıkarmıştır. Bu benzerliğin, yani kategori teorisinin matematiksel kuramları ile uyumlu bir programlama dilinin varlığı yukarıda açıklanan sebeple matematikten çok bilgisayar bilimine büyük bir katkı sağlamaktadır.

Peki bir programlama dilinin fonksiyonel olması ne anlama gelmektedir? Fonksiyonel dillerin en önemli özelliği yazılan programın, yaptığı hesaplamalara matematiksel bir fonksiyon

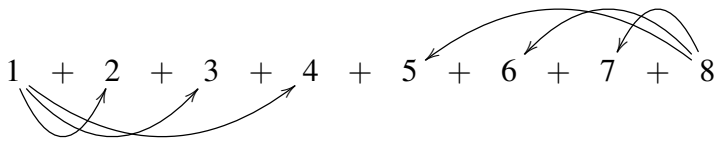
olarak bakılmasıdır. Bu bakış açısı birçok hesaplamayı kısa matematiksel tanımlamalarla yapmaya olanak sağlar.

Yordamsal diller ve nesneye dayalı diller sonuç odaklıdır. Yani değişkenler tutulup, bu değişkenler güncellenerek sonuca ulaşılır. Sonucun dışında kalan herşey **yan etki** olarak adlandırılır. Fonksiyonel diller ise olayın kendisini ifade etmek için sonuçlara değil hesaplamalara odaklanır. Diğer dillerde kullanıldığı gibi değişken değeri değiştirme ve state olayları olmadığından (özellikle istenmedikçe), fonksiyonel programlamada yan etki yoktur. Yani herhangi bir fonksiyonun veya bir değişkenin hangi durumda veya değerde olduğundan bağımsız olarak program sonucu aynı kalır ki programlar çalışırken yan etkiler sebebiyle meydana gelen hata(bug)ların oluşumu baştan engellenmiş olur.

1965 yılında Intel firmasının kurucularından olan Gordon Moore'nin ortaya attığı ve kendi adıyla anılan Moore yasası bir çip üzerindeki transistör sayısının her iki yılda bir ikiye katlanacağını söyler. Uzun yıllar geçerliliğini koruyan bu teorinin daha fazla sürdürülememesi ve işlemci üreticilerinin anlamlı hız artışlarını sağlayamamaları çok çekirdekli bilgisayarlara geçişi zorunlu hale getirmiştir. Bu ise paralel programlama kavramının doğmasına sebep olmuştur. Paralel programlama aynı görevin (parçalara bölünmüş ve uyarlanmış) sonuçları daha hızlı elde etmek için çoklu işlemcilerde eş zamanlı olarak işletilmesidir. Örneğin basit bir toplama işlemi için klasik programlamada



7 kez işlem yapılır. 2 çekirdeği birden kullanan paralel programlama ise



4 kez işlem yapılır. Basit bir işlem 1.75 kat hızlandırılmış olur. Paralel hesaplama ile performans artar ve bilimsel gerçekler paralel hesaplamaya gereksinim duyarlar.

Fonksiyonel diller state özelliği içermediğinden paralel programlama yordamsal ve nesne tabanlı dillerden daha verimlidir. Ayrıca fonksiyonel dillerin (lazy evulation) tembel öğrenme özelliğinden dolayı sonsuzluk içeren kavramları tanımlamak ve kullanmak mümkün olmuştur.

Ülkemizde bilişim firmaları hala nesne tabanlı dillere hakim yazılımcılar ararken dünyada fonksiyonel dillere hakim matematiksel düşünebilen yazılımcılar aranmaktadır. Bu ise yakın gelecekte fonksiyonel dillere yoğun bir yönelimin olacağını göstermektedir.

4.2 Bir $\mathcal{L}$ Dilinin Kategorisi

Fonksiyonel programlama yazılımı modellerken yazılımın fonksiyonlardan oluştuğunu temel alır ve veri yapılarının içeriğini değiştirmek yerine onlardan yeni veri yapıları türeten fonksiyonlar oluşturur. **Yani herşey bir fonksiyondur**. Bu cümle herşeyin fonksiyon olduğu başka bir kavramı akla getirir : kategori teori.

Bir fonksiyonel programlama dili kabaca dilde tanımlı temel tip ve fonksiyonlar ile kullanıcı tarafından tanımlanan tip ve fonksiyonlardan ibarettir. Fonksiyonel dilin elemanları;

- Dilin temel veri tipleri,
- Her bir veri tipinin sabitleri,
- Veri tipleri arasındaki fonksiyonlar,
- Veri tiplerine ve fonksiyonlara uygulanan yapıcılar,

olarak listelenebilir. Fonksiyonel dillerde yeni tip ve fonksiyonlar, **yapıcılar** (constructors) yardımıyla üretilir. Yapıcı kavramı herbir fonksiyonel programlama dili için farklılıklar gösterir. Örneğin, Haskell fonksiyonel programlama dilinde günler isimli bir veri tipi üretmek istersek

HASKELL
```
data Gunler = Pazartesi | Sali | Carsamba | Persembe | Cuma | Cumartesi | Pazar
```

kodlamasını kullanırız. Buradaki Gunler ifadesi tip yapıcı (type constructor) ve Pazartesi, Sali, ... Pazar ifadelerinin herbiri ise değer yapıcı (value constructor) olarak adlandırılır. Yeni

tip oluştururken değer yapıcılar parametre alabilir veya almayabilir. Yukarıdaki Gunler veri tipindeki değer yapıcılar parametre almamaktadır.

HASKELL
```
data Sekil = Cember Float Float Float | Dikdortgen Float Float Float Float
```

Yukarıdaki Haskell kodlamasında Sekil tip yapıcısı ve Cember, Dortgen değer yapıcıları kullanılarak yeni bir tip üretilmektedir. Buradaki değer yapıcılar, parametre **almaktadır**. Cember değer yapıcısı Float veri tipinden üç parametre almaktadır. İlk iki parametre çemberin koordinatlarını üçüncü parametre ise yarıçapı ifade etmektedir. Dikdörtgen değer yapıcısı Float veri tipinden dört parametre almaktadır. Bu parametreler dikdörtgenin koordinatlarını ifade eder. Parametre almayan değer yapıcılar fonksiyonel programlama dillerinde veri tipinin **sabitleri** olarak adlandırılırlar. Örneğin, yukarıdaki Pazartesi, Sali,... ,Pazar değer yapıcıları Gunler veri tipinin sabitleridir.

Fonksiyonel programlama dillerinde parametre olarak başka bir fonksiyonu alarak yeni bir fonksiyon oluşturan yapılar fonksiyon yapıcı (function constructor) olarak adlandırılır.

Örneğin aşağıdaki Haskell kodlamasında parametre olarak herhangi bir fonksiyon girildiğinden bu fonksiyonu iki kez işleterek yeni bir fonksiyon tanımlanmaktadır.

HASKELL
```
ikiKez :: (a -> a) -> a -> a
ikiKez f x = f (f x)
```

Benzer şekilde fonksiyonel dillerde uygun seçilmiş her iki fonksiyonu parametre olarak kabul ederek yeni bir fonksiyon oluşturulan fonksiyon yapıcılar bulunur.

HASKELL
```
(.) :: (b -> c) -> (a -> b) -> a -> c
g . f = \x -> g (f x)
```

Yukarıdaki kodlama Input ve Output veri tipleri uygun seçilmiş iki fonksiyonun bileşkesi olan fonksiyonu oluşturmaktadır. Burada Input(g) =Output(f) olacak şekilde fonksiyonlar $\cdot$

fonksiyonunda kullanılabilir. Birçok fonksiyonel dilde bileşke sırası sola duyarlı yazılmasına rağmen ($f;g$ şeklinde) Haskell bu fonksiyonu matematik uyumlu olmak için sağa duyarlı olarak $g \cdot f$ biçiminde kullanılır.

L fonksiyonel dili için $C(L)$ kategorisini oluşturmaya başlanırsa:

1) Temel veya yapıcılardan oluşturulan her bir A tipi için hiçbir işlem yapmayan kendi üzerinde bir id_A fonksiyonunun varlığı söylenebilir.

2) Herbir A tipinden 1 tipine yalnızca bir tane fonksiyon olacak şekilde 1 adı verilen bir tipi L diline eklenirse, 1 tipi herhangi bir singleton (tek elemanlı küme) olabilir. Bu tipi herhangi bir tipin sabitlerini fonksiyon olarak tanımlamak için kullanacağız.$(1 \to A)$

3) A tipinden B tipine olan bir f fonksiyonu ($f : A \to B$) ve B tipinden C tipine olan bir g fonksiyonu ($g : B \to C$) gözönüne alındığında L dilinden fonksiyon yapıcılar yardımıyla A tipinden C tipine $f;g$ ile göstereceğimiz bir fonksiyon her zaman üretilebilir.

Şimdi L fonksiyonel dili için $C(L)$ kategorisinin yapısını görülecek olursa:

L Fonksiyonel Programlama Dili	**$C(L)$ Kategorisi**
• L dili temel veri tipleri ve yeni tanımlanacak veri tipleri,	• $C(L)$ nin objeleri
• Herbir veri tipinin sabitleri ve veri tipleri tipleri arasında yazılabilen tüm fonksiyonlar.	• $C(L)$ nin morfizmleri
• L dilindeki herbir fonksiyon bir Input tipine ve bir Output tipine sahiptir.	• $Mor(C(L))$ nin her elemanının bir source objesi ve bir target objesi vardır.
• L deki herbir A veri tipi için Input'u ve Output'u A olan ve girilen parametreyi kendisine taşıyan bir fonksiyon yazılabilir.	• $C(L)$ nin herbir A objesi için $s(id_A) = A$ ve $t(id_A) = A$ olacak şekilde birim morfizm her zaman vardır.
• L dilinde Input$(f) = A$ ve Output$(f) = B$ özelliğindeki f fonksiyonu ile Input$(g) = B$ ve Output$(g) = C$ özelliğindeki herbir g fonksiyonu için Input'u A ve Output'u C olan $f;g$ bileşke fonksiyonu her zaman yazılabilir.	• $C(L)$ deki $s(f) = A$, $t(f) = B$ ve $s(g) = B$, $t(g) = C$özelliğindeki herbir f, g morfizmi için $g \circ f : A \to C$, $s(g \circ f) = A$ ve $t(g \circ f) = C$ morfizmi her zaman vardır.

Örnek : L, 3 tane temel veri tipine sahip bir fonksiyonel programlama dili olsun.

$$NAT: \text{ Doğal sayılar}$$
$$BOOLEAN = \{true, false\}$$
$$CHAR: \text{ Karakterler}$$

veri tiplerimiz olmak üzere $C(L)$ kategorisini oluşturalım:

1 bir singleton yani tek elemanlı herhangi bir küme olmak üzere

$$\text{Ob}(C(L)) = \{NAT, BOOLEAN, CHAR, 1\}$$

olsun.

$$Mor(C(L)) \underset{t}{\overset{s}{\rightrightarrows}} Ob(C(L))$$

olmak üzere $C(L)$ kategorisinin morfizmlerini belirleyelim.

$$\begin{array}{lll}
s(id_{NAT}) = t(id_{NAT}) = NAT & & \\
s(id_{CHAR}) = t(id_{CHAR}) = CHAR & & \\
s(id_{BOOLEAN}) = t(id_{BOOLEAN}) = BOOLEAN & & \\
s(id_1) = t(id_1) = 1 & & \\
s(0) = 1 & \text{ve} & t(0) = NAT \\
s(c) = 1 & \text{ve} & t(c) = CHAR \\
s(false) = 1 & \text{ve} & t(false) = BOOLEAN \\
s(true) = 1 & \text{ve} & t(true) = BOOLEAN \\
s(n) = BOOLEAN & \text{ve} & t(n) = BOOLEAN \\
s(succ) = NAT & \text{ve} & t(succ) = NAT \\
s(ord) = CHAR & \text{ve} & t(ord) = NAT \\
s(chr) = NAT & \text{ve} & t(chr) = CHAR \\
s(id_1) = 1 & \text{ve} & t(id_1) = 1 \\
s(x) = NAT & \text{ve} & t(x) = 1 \\
s(y) = CHAR & \text{ve} & t(y) = 1 \\
s(z) = BOOLEAN & \text{ve} & t(z) = 1
\end{array}$$

$$Mor(C) = \{id_1, id_{NAT}, id_{CHAR}, id_{BOOLEAN}, 0, c, false, true, n, succ, ord, chr, x, y, z\}$$

kümesinin bazı elemanlarının özellikleri şöyledir

$$0 : 1 \longrightarrow NAT$$

sabit bir fonksiyon olup $\forall x \in 1$ için $0(x) = 0 \in NAT$ sağlanır.

$$succ : NAT \longrightarrow NAT$$

fonksiyonu $\forall x \in NAT$ için $x \mapsto x+1$ özelliğindedir. Bu iki fonksiyon birlikte düşünülürse

$$1 \xrightarrow{0} \overset{succ}{\circlearrowright} NAT$$

bileşkesi tüm doğal sayıları üretebilir.

Bilgisayarların sayılarla karakter arasında ilişki kurmasına olanak sağlayan kodlama sistemlerinden birisi ASCII kodlamasıdır.

ASCII kodlaması karakterlerin 0-255 arasındaki 256 tane doğal sayıdan oluşur. Bu kodlama aşağıdaki şekilde gösterilebilir.

Regular ASCII Chart (character codes 0 - 127)

000	(nul)	016 ►	(dle)	032	sp	048	0	064	@	080	P	096	`	112	p
001 ☺	(soh)	017 ◄	(dc1)	033	!	049	1	065	A	081	Q	097	a	113	q
002 ☻	(stx)	018 ↕	(dc2)	034	"	050	2	066	B	082	R	098	b	114	r
003 ♥	(etx)	019 ‼	(dc3)	035	#	051	3	067	C	083	S	099	c	115	s
004 ♦	(eot)	020 ¶	(dc4)	036	$	052	4	068	D	084	T	100	d	116	t
005 ♣	(enq)	021 §	(nak)	037	%	053	5	069	E	085	U	101	e	117	u
006 ♠	(ack)	022 ▬	(syn)	038	&	054	6	070	F	086	V	102	f	118	v
007 •	(bel)	023 ↨	(etb)	039	'	055	7	071	G	087	W	103	g	119	w
008 ◘	(bs)	024 ↑	(can)	040	(	056	8	072	H	088	X	104	h	120	x
009 ○	(tab)	025 ↓	(em)	041	)	057	9	073	I	089	Y	105	i	121	y
010	(lf)	026 →	(eof)	042	*	058	:	074	J	090	Z	106	j	122	z
011 ♂	(vt)	027 ←	(esc)	043	+	059	;	075	K	091	[	107	k	123	{
012 ♀	(np)	028 ∟	(fs)	044	,	060	<	076	L	092	\	108	l	124	¦
013	(cr)	029 ↔	(gs)	045	-	061	=	077	M	093	]	109	m	125	}
014 ♫	(so)	030 ▲	(rs)	046	.	062	>	078	N	094	^	110	n	126	~
015 ✲	(si)	031 ▼	(us)	047	/	063	?	079	O	095	_	111	o	127	⌂

Extended ASCII Chart (character codes 128 - 255)

128	Ç	143	Å	158	×	172	¼	186	‖	200	╚	214	í	228	õ	242	≥
129	ü	144	É	159	ƒ	173	¡	187	╗	201	╔	215	Î	229	Õ	243	≤
130	é	145	æ	160	á	174	«	188	╝	202	╩	216	Ï	230	µ	244	¶
131	â	146	Æ	161	í	175	»	189	¢	203	╦	217	┘	231	þ	245	§
132	ä	147	ô	162	ó	176	░	190	¥	204	╠	218	┌	232	Þ	246	÷
133	à	148	ö	163	ú	177	▒	191	┐	205	═	219	█	233	Ú	247	¸
134	å	149	ò	164	ñ	178	▓	192	└	206	╬	220	▄	234	Û	248	°
135	ç	150	û	165	Ñ	179	│	193	┴	207	¤	221	¦	235	ù	249	¨
136	ê	151	ù	166	ª	180	┤	194	┬	208	ð	222	Ì	236	ý	250	·
137	ë	152	ÿ	167	º	181	Á	195	├	209	Ð	223	▀	237	Ý	251	¹
138	è	153	Ö	168	¿	182	Â	196	─	210	Ê	224	ó	238	¯	252	³
139	ï	154	Ü	169	®	183	À	197	┼	211	Ë	225	ß	239	´	253	²
140	î	155	ø	170	¬	184	©	198	ã	212	È	226	Ô	240	-	254	■
141	ì	156	£	171	½	185	╣	199	Ã	213	ı	227	Ò	241	±	255	
142	Ä	157	Ø														

$$chr : NAT \longrightarrow CHAR$$

fonksiyonu verilen herhangi bir doğal sayının ASCII kodlamasında karşılık geldiği karakteri verir. Bu fonksiyon 256 denklik sınıfına göre işlem yapmaktadır.

$$ord : CHAR \longrightarrow NAT$$

fonksiyonu verilen herhangi bir ASCII karakterinin karşılık geldiği 0-255 arasındaki doğal sayıyı vermektedir.

$$c : 1 \longrightarrow CHR$$

sabit bir fonksiyon olup $\forall x \in 1$ için $c(x) = c \in CHR$ sağlanır.

$$1 \xrightarrow{c} CHR \xrightarrow{ord} NAT \overset{succ}{\circlearrowleft} \xrightarrow{chr} CHR$$

bileşkesi CHR objesindeki tüm elemanları üretir.

$$true : 1 \longrightarrow BOOLEAN$$

sabit bir fonksiyon olup $\forall x \in 1$ için $true(x) = true \in BOOLEAN$,

$$false : 1 \longrightarrow BOOLEAN$$

sabit bir fonksiyon olup $\forall x \in 1$ için $false(x) = false \in BOOLEAN$ sağlanır.

$$\begin{array}{rrcl} n: & BOOLEAN & \longrightarrow & BOOLEAN \\ & true & \longmapsto & false \\ & false & \longmapsto & true \end{array}$$

şeklinde bir fonksiyon olmak üzere

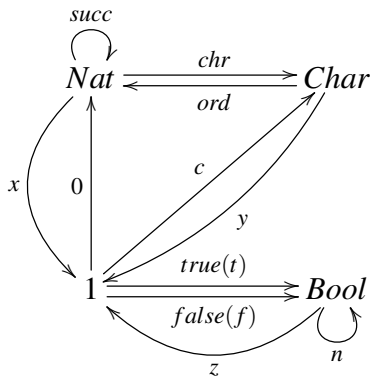
$$1 \xrightarrow{true} BOOLEAN \quad (n: BOOLEAN \to BOOLEAN)$$

veya

$$1 \xrightarrow{false} BOOLEAN \quad (n: BOOLEAN \to BOOLEAN)$$

bileşkesiyle *BOOLEAN* objesinin tüm elemanları üretilebilir.

$C(L)$ kategorisinin objelerini ve morfizmlerini diyagram olarak gösterilirse:



$$\begin{aligned} n \circ true &= false \\ n \circ false &= true \\ n \circ n &= id_{BOOLEAN} \\ chr \circ ord &= id_{CHAR} \end{aligned}$$

bileşkeleriyle birlikte kategori şartlarının sağlandığı görülür.

4.3 Programlama Dilinde Kategoriksel Yapılar

Fonksiyonel Programlama Dili'nde herhangi bir tip alındığında, bu tipin elemanlarının bir sonlu listesi yazılabilir.

Örneğin, *Int* tipi için,

$$[1,2,2,1,4]$$

bir sonlu listedir.

$List(S)$, S tipinin elemanları yardımıyla elde edilebilen tüm mümkün listelerin kümesi olsun.

Örneğin, *Int* tipi için,

$$List\ (Int) = \{\ [\ ], [1], [1,2], [1,2,2], [3,3], \ldots\}$$

şeklinde sonsuz elemanlı bir kümedir.

Hatta tek elemanlı bir $S = \{s\}$ tipi alınsa dahi,

$$List\ (S) = \{[], [s], [s,s], [s,s,s], [s,s,s,s], \ldots\}$$

şeklinde yine sonsuz elemanlı bir kümedir.

Böylece *List* ile tipler üzerinde etkisi her tipin aslında bir küme olması itibariyle dolaylı olarak,

$$List : C(\mathcal{L}) \longrightarrow C(\mathcal{L})$$

şeklinde tanımlanmış olur.

Fakat *List* in genel olarak fonksiyonel programlama mantığı altında, sadece tipler üzerinde değil, aynı zamanda aşağıda açıkça inceleneceği gibi,

$$f : S \longrightarrow S'$$

biçimdeki, iki tip arasındaki fonksiyonların üzerinde,

$$List(f) : List\ (S) \longrightarrow List\ (S')$$

şeklinde de bir etkisi mevcuttur ki bu etki basit bir örnekle,

$$List\ (f) : [a,b] \longrightarrow List\ (f)\ [a,b] = [f(a), f(b)]$$

şeklindedir.

List dönüşümünün fonksiyonlara etkisini, herhangi bir fonksiyonel programlama dilinde genel bir örnekle görelim. Herhangi bir fonksiyonel programlama dilinde f fonksiyonu,

Fonksiyonel Programlama Dilleri

```
f :: Int -> Int
f x = x*x
```

şeklinde tanımlanacak olursa,

$$List(f) : List(Int) \longrightarrow List(Int)$$

dönüşümü için,

Haskell

```
> List(f)([1,2,3])
[1,4,9]
```

sonucu elde edilir.

Bir önceki örnek özel olarak Haskell diline taşınırsa,

List dönüşümünün f fonksiyonuna etkisi burada $fmap$ fonksiyonu yardımıyla

$$List(f) = fmap\ f$$

şeklinde tanımlanmaktadır.

Bu durumda yukarıdaki fonksiyon Haskell üzerinde,

Fonksiyonel Programlama Dilleri

```
f :: Int -> Int
f x = x*x
```

biçiminde tanımlanacak, ve ayrıca

$$\text{fmap f :: [Int]} \longrightarrow \text{[Int]}$$

dönüşümü ile birlikte,

Haskell

```
> fmap f [1,2,3]
[1,4,9]
```

sonucu elde edilecektir.

4.3.1 Funktor

Haskell dilinde bir listeye eleman eklenmesi : işlemi ile sağlandığını biliyoruz.

fmap fonksiyonunun tanımına ve özelliklerine bakılacak olursa, bu fonksiyon Haskell üzerinde,

Haskell
```
fmap :: (a -> b) -> f a -> f b
fmap _ [ ] = [ ]
fmap f (x:xs) = f x : fmap f xs
```

şeklinde tanımlanabilmektedir.

Yukarıdaki tanımı basit şekilde örneklendirilecek olursa,

Haskell
```
fmap f (2 : [3,4])
>[4,9,16]

f 2 : fmap f [3,4]
>[4,9,16]

f 2 : f 3 : fmap f [4]
>[4,9,16]
```

sonucu elde edilecektir.

Ayrıca alternatif olarak *fmap* fonksiyonu,

Haskell
```
fmap :: (a -> b) -> f a -> f b
fmap _ [ ] = [ ]
fmap f [a] = [f a]
fmap f (xs ++ ys) = fmap f xs ++ fmap f ys
```

biçiminde de tanımlanabilir.

Yine bu tanımı da basit şekilde örneklendirilirse,

Haskell
```
fmap f [ ]
> [ ]

fmap f [5]
> [25]

fmap f ( [1,2] ++ [3,4] )
> [1,4,9,16]

fmap f [1,2] ++ fmap f [3,4]
> [1,4,9,16]
```

sonucu elde edilecektir.

fmap fonksiyonunun yukarıda incelediğimiz yapısı ve özellikleri göz önüne aldığında aklımıza ilk olarak, bu fonksiyonun kategoriksel olarak nasıl davrandığı,

- gf gibi bir kompozisyon üzerindeki etkisi için hangi özelliklere sahip olduğu
- Özel olarak birim fonksiyonlar için nasıl bir etkisi olduğu

soruları gelmektedir.

Bu soruların cevapları incelenecek olursa,

Haskell
```
f :: a -> b

g :: b -> c

g.f :: a -> c
```

şeklindeki fonksiyonlar ve kompozisyonlar için *fmap* fonksiyonunun,

Haskell
```
fmap f :: ( a -> b ) -> f a -> f b
fmap f [ ] = [ ]
fmap f (x:xs) = f x : fmap f xs

fmap g :: ( b -> c) -> g b -> g c
fmap g [ ] = [ ]
fmap g (y:ys) = g x : fmap g ys
```

yapısı gereği,

```Haskell
fmap g.f :: ( a -> c ) -> g.f a -> g.f c
fmap g.f [ ] = [ ]
fmap g.f (x:xs) = g.f x : fmap g.f xs
```

olup, ayrıca diğer taraftan da,

```Haskell
fmap g . fmap f ::
fmap g . fmap f [ ] = fmap g [ ] = [ ]
fmap g . fmap f (x:xs) = fmap g ( fmap f (x:xs) )
```

şeklindedir.

Bu basit bir örnekle incelenirse,

```Haskell
f :: Int -> Int
f x = x*x

g :: Int -> Int
g y = y+2
```

fonksiyonları için,

```Haskell
fmap (g.f) [1,2]
> [3,6]

( fmap g . fmap f ) [1,2]
> [3,6]
```

olduğu ve dolayısıyla da,

$$\text{fmap g.f} = \text{fmap g . fmap f} \quad \textbf{(i)}$$

şartını doğruladığı görülür.

Diğer yandan,

Haskell
```
idS :: S -> S
idS x = x
```

şeklinde tanımlı id_S ve,

Haskell
```
id(List(S)) :: [a] -> [a]
id(List(S)) y = y
```

şeklinde tanımlı $id_{List(S)}$ için,

Haskell
```
fmap idS (x:xs) = idS x : fmap idS xs
```

olmaktadır.

Bu basit bir örnekle incelenirse,

Haskell
```
fmap idS [1,2,3]
> [1,2,3]

id(List(S)) [1,2,3]
> [1,2,3]
```

olduğu ve dolayısıyla da

$$\text{fmap idS} = \text{id(List(S))} \quad \textbf{(ii)}$$

şartını doğruladığı görülür.

Sonuç olarak, *List* yapısının ve funktor tanımının sahip olduğu özellikler göz önüne alındığında,

- (i) şartına karşılık F1 şartının,

- (ii) şartına da F2 şartının

birebir aynı olduğu görülmektedir.

Bu durumda *List* in $C(\mathcal{L})$ kategorisinden $C(\mathcal{L})$ kategorisine,

$$\begin{array}{lccll} List: & C(\mathcal{L}) & \longrightarrow & C(\mathcal{L}) & \textbf{(I)} \\ & A & \rightsquigarrow & List(A) \sim A \text{ nın tüm listeleri} & \\ & f & \rightsquigarrow & List(f) = fmap\ f & \end{array}$$

şeklinde bir **funktor** olduğu sonucuna ulaşılmaktadır.

4.3.2 Monoid

Tanım: M bir küme ve $\circ$ da bu küme üzerinde bir ikili işlem olsun. Eğer bu M kümesi, üzerinde tanımlı $\circ$ işlemi altında, asosyatiflik ve birimlilik şartlarını sağlıyorsa bu durumda M ye, $\circ$ işlemine göre bir **monoid** denir.

Tanım : M ve M' birer monoid olmak üzere,

$$\phi : (M, \circ, e) \longrightarrow (M', \circ', e')$$

fonksiyonunun bir homomorfizm olabilmesi için gerek ve yeter şart, $\forall a, b \in M$ için,

* (İşlem koruması)

$$\phi(a \circ b) = \phi(a) \circ' \phi(b)$$

* (Birim koruması)

$$\phi(e) = e'$$

olmasıdır.

Kategori Teori'deki $\mathcal{C} = Monoid$ kategorisi hatırlandığında, bu kategori,

- $Ob(\mathcal{C}) =$ Monoidlerin Sınıfı
- $Mor(\mathcal{C}) =$ Monoid Homomorfizmleri
- $Komp(\mathcal{C}) =$ Bileşke İşlemi

şeklinde tanımlıdır.

Keyfi bir S tipi (kümesi) için $List(S)$ aslında, yukarıda kullandığımız bir operatör yardımıyla şu ana kadar farkında olmadığımız ekstra bir yapıya sahiptir.

Öyle ki,

- Herhangi iki listeyi uç uca birleştirme anlamındaki $++$ işlemi
- Bu işlem üzerinde [ ] boş listesinin bir etkisi olmadığı
- Bu işlemin herhangi üç liste için asosyatifliği koruduğu

bu işlemin tanımı gereği,

Haskell
```
[ ] ++ xs = xs

xs ++ [ ] = xs

xs ++ ( ys ++ zs ) = ( xs ++ ys ) ++ zs
```

şeklinde açıktır.

O halde artık S herhangi bir tip (küme) olmak üzere;

(List (S) , ++ , [ ])

yapısı,

- G1) Birimlilik
- G2) Asosyatiflik

şartlarını sağladığı için bir **monoid** yapısı oluşturmaktadır.

O halde peki $List(S)$ nin küme yapısından bir üste çıkarak, aslında bir monoid olması dolayısıyla (I) de tanımlanan funktoru da bir adım daha ilerleterek;

$$List : C(\mathcal{L}) \longrightarrow Monoid \quad \textbf{(II)}$$

şekline dönüştürülebilir mi?

Bu ifadenin gerçeklenebilmesi için, *List* in kümeleri monoidlere taşımasının yanında ayrıca, fonksiyonları da monoid homomorfizmlerine taşımalıdır.

Burada yukarıdaki hatırlatma altında herhangi iki,

$$(List(S), ++, [\,]) \quad \& \quad (List(S'), ++, [\,])$$

monoidi ve bu iki monoid arasındaki homomorfizm de,

$$f : S \longrightarrow S'$$

fonksiyonu için,

$$fmap\ f\ :\ (List(S), ++, [\,]\,) \longrightarrow (List(S'), ++, [\,])$$

şeklinde alındığı takdirde bu fonksiyonun bir homomorfizm olabilmesi için sağlaması gereken şartlar,

- fmap _ [ ] = [ ]
- fmap f (xs ++ ys) = fmap f xs ++ fmap f ys

olacak olup, bu şartların geçerli olduğu da $fmap$ fonksiyonunun yapısı ve özellikleri kısmında zaten gösterilmiştir.

Tüm bu özellikler altında *List* in $C(\mathcal{L})$ kategorisinden *Monoid* kategorisine,

$$List : C(\mathcal{L}) \longrightarrow Monoid$$

şeklinde bir **funktor** olduğu sonucuna ulaşılmaktadır.

4.3.3 Doğal Transformasyon

Doğal Transformasyon Fonksiyonel Programlama Dilinde keyfi bir S tipi (kümesi) üzerindeki listeler için, bu listeyi ters çeviren, yani elemanları sıra anlamında yansıtarak yeni bir liste oluşturan *reverse* adında bir fonksiyon bulunmaktadır.

Bu fonksiyonun yapısını Haskell için incelenecek olursa,

Haskell

```
reverse :: [a] -> [a]
reverse [ ] = [ ]
reverse ( x : xs ) = reverse xs ++ [x]
```

şeklinde tanımlı olup, örneğin herhangi bir [1,4,5] listesi için, bu listenin tersi,

Haskell
```
reverse [1,4,5]
> [5,4,1]
```

elde edilir.

Bu durumda *reverse* fonksiyonu,

$$reverse_S : List(S) \longrightarrow List(S)$$

veya

$$reverse\ (S) : List(S) \longrightarrow List(S)$$

şeklinde tanımlıdır. Hatta dikkat edilirse,

$$reverse : List \longrightarrow List$$

şeklinde bir dönüşüm olduğu görülmektedir.

O halde peki *List* yapısının bir funktor olması, ve ayrıca *reverse* dönüşümünün iki funktor arasında tanımlı olduğundan faydalanarak acaba bu *reverse* fonksiyonunun kategoriksel olarak bir doğal transformasyon olduğunu söylenebilir mi?

Yukarıdaki natural transformation tanımı için,

- $\mathcal{C}$ ve $\mathcal{D}$ olarak $C(\mathcal{L})$ kategorisi
- F ve G funktorları olarak *List* funktoru
- η morfizmi olarak da *reverse* fonksiyonu

alınsın.

Bu durumda,

$$\begin{aligned} List &: C(\mathcal{L}) \longrightarrow C(\mathcal{L}) \\ List &: C(\mathcal{L}) \longrightarrow C(\mathcal{L}) \end{aligned}$$

funktorları ve $S, S' \in Ob(Set)$ için,

$$f : S \longrightarrow S'$$

fonksiyonu tanımlı olduğunda,

(i) *Set* in her S objesi için,

$$\begin{array}{rccl} reverse: & Ob(Set) & \longrightarrow & Mor(Set) \\ & S & \longmapsto & reverse\ (S) : List\ (S) \longrightarrow List\ (S) \end{array}$$

şeklinde tanımlı olduğu *reverse* fonksiyonunun yapısı gereği açıkça görülmektedir.

(ii) $\mathcal{C}$ nin her $f : S \to S'$ morfizmi için,

$$\begin{array}{ccc} S & & List(S) & \xrightarrow{reverse(S)} & List(S) \\ \Big\downarrow{\scriptstyle f} & & \Big\downarrow{\scriptstyle fmapf} & & \Big\downarrow{\scriptstyle fmapf} \\ S' & & List(S') & \xrightarrow[reverse(S')]{} & List(S') \end{array}$$

diyagramının değişmeli yani,

$$reverse\ (S') \ .\ fmap\ f\ =\ fmap\ f\ .\ reverse\ (S)$$

olması gerekmektedir.

Örneğin,

Haskell
```
reverse ( fmap f [1,2,3] )
> [9,4,1]
```

ve ayrıca,

Haskell
```
fmap f ( reverse [1,2,3] )
> [9,4,1]
```

olup,

$$reverse\ (S')\ .\ fmap\ f\ =\ fmap\ f\ .\ reverse\ (S)$$

olduğu açıktır.

Yani *reverse* fonksiyonu,

$$reverse : List \longrightarrow List$$

şeklinde bir **doğal transformasyon**'dur.[3]

SONUÇ ve ÖNERİLER

Yapılan bu çalışma ile fonksiyonel programlama dilleri arasında önemli bir yere sahip olan Haskell, kategori teorisi bakışı altında incelenmiştir. Haskell dilinin seçilmesinin amacı, matematiksel bir mantıkla oluşturulmuş olması ve cebirsel yapıların birebir bu dile çevrilebilmesidir.

Kategori teorisi bu anlamda Haskell'in yapısıyla birebir örtüşmektedir. Cebirsel yapıların bir programlama dili ile bilgisayar diline çevrilebilmesi matematiğin bir uygulama alanını ortaya çıkarmıştır.

Bu tez, Haskell dilinde cebirsel bir paket elde etmeye yardımcı olmuştur. Haskell paketleri belli kural ve algoritmaların bütünüdür. Bu paketler sayesinde karmaşık cebirsel yapıların oluşturulması, uygunluğu ve yeni yapıların oluşturulması kolaylıkla gerçekleştirilebilir. Bu tezin bir sonucu olarak bir kategorinin Haskell dilinde yapılaştırılmış hali oluşturulmaya çalışılacaktır.

KAYNAKLAR DİZİNİ

[1] D.Hünniger, Haskell, 2012 24

[2] M.Lipovaca, Learn You a Haskell for Great Good, 2011 24

[3] D.Pitt, S. Abramsky, A. Poigne and D. Rydeheard, Category Theory and Computer Programming, U.K. Tutorial and Workshop, 1985 81

[4] G.Hutton, Programming in Haskell, Cambridge University Press, 2007 24

[5] S.P.Vimal, Functional Programming, BITS Plani, 2012 14

[6] L.Dillon, Functional vs. Imperative Programs, Lecture Notes, 2008 14

[7] G.D. Crnkovic, History of Computer Science, Mälardalen Real-Time Research Centre, 2001 14

[8] D. Pitt, Categories, U.K. Tutorial and Workshop, 1985 1

[9] Z.Arvasi, Ders Notları, 2013 3