III. KISIM - PART III

GÜNCEL TEKNOLOJİLER

CURRENT TECHNOLOGIES

16. BÖLÜM / CHAPTER 16

GÜNCEL YAZILIM TEKNOLOJİLERİ

CURRENT SOFTWARE TECHNOLOGIES

Mehmet Hakan SATMAN*, Emre AKADAL**

*Prof. Dr., İstanbul Üniversitesi, İktisat Fakültesi, Ekonometri Bölümü, İstanbul, Türkiye
E-mail: mhsatman@istanbul.edu.tr

**Dr., İstanbul Üniversitesi, Enformatik Bölümü, İstanbul, Türkiye
E-mail: emre.akadal@istanbul.edu.tr

DOI: 10.26650/B/ET07.2021.003.16

ÖZ

Sonucu matematik için zamansız olarak bilinen ifadelerin belirli bir değere kavuşturulması için geliştirilen hesap etme (calculation) ve hesaplama (computation) algoritmaları ve onların hesap makinesi, analog ve dijital bilgisayarlarda uygulanması olan yazılım, insanlık için uzunca bir serüveni oluşturur. Bu serüven boyunca hesaplama işlemleri için bulunan farklı çözüm ve yaklaşımların programlama dillerine, programlama dillerinin de farklı çözüm ve yaklaşımlara etki ettiği görülür. Gerek yazılım geliştirme araçlarının gelişimi, gerekse son kullanıcıların ihtiyaçları, bu araçların, doğrusal olmayan bir trend boyunca evrim geçirmesine sebep olduğu gözlemlenir. Bu evrim, programlama dillerinin canlı bir varlığın hayatı sürecine benzer bir süreçte sahip olmasını sağlamıştır. Kısa sayılabilecek bir süre içerisinde programlama dillerinin doğuşuna, bir topluluk tarafından kabul edilmesi durumunda gelişimine ve yerini daha etkin bir programlama diline bırakmak üzerine gerilemesine şahitlik edebiliyoruz. Bu da günün popüler yöntemlerinin yakın gelecekte “eski” sayılabilmesine sebep olabiliyor. İnsanın doğayı ve kendisini taklit eden teknolojileri geliştirmedeki arzusu günümüzü dağdık, büyük ve yapısal olmayan verinin hızlıca işlendiği, tüm mümkün durumları izlemek yerine akıl yürüten makinelerin üretildiği, birbirine bağlı ve her birinin küçük bir işi yerine getirdiği cihazların kol gezdiği, eldeki veriyi hızlıca değerlendirip insan için alınması zor kararları aldığı bir dünya haline getirmiştir. Bir insan kendisinden daha zeki bir makine yapabilir mi? İnsanın sahip olduğu bilişsel kusurlar bir taklit olarak üretilen yapay zekada da görülecek midir? Yoksa yazılım ve yapay zeka, insanlık için ne siyah ne de beyaz – hibrit – bir gelecek mi sunacaktır? Bu bölüm; matematik ve yazılım teknolojilerinin kesişimleriyle başlamakta, devamında ise fark yaratmış programlama dillerinin öne çıkan özellik ve alana kattıkları yenilikler ele alınmaktadır.

Anahtar Kelimeler: Yazılım, kodlama, güncel teknolojiler, yapay zeka

ABSTRACT

Software is an adventure, which includes calculation, computation, and application processes to determine the value of an expression known by mathematics, a time-independent language, and discipline for a long time. During the adventure, various solutions and approaches used in computation have affected programming languages and vice

versa. The improvement of programming tools and the needs of end users have caused the evolution of these tools during a non-linear trend. This evolution has enabled programming languages to generate a process similar to the vital process of a living being. In a short period of time, we can witness the birth of programming languages, their development, if they are accepted by a community, and their regression to leave their place to a more effective software language. This may cause popular methods of the day to be considered “old” in the near future. Humans’ desire to develop technologies that mimic nature and themselves have changed the world. In this world, large and unstructured data are processed quickly, decision-making processes are conducted by reasoning software that does not monitor all the possible situations, systems work by remaining interconnected and distributed, and decision-making processes are conducted quickly and automatically. Can someone build a machine that is smarter than itself? Will we face human-specific cognitive faults on artificial intelligence applications? Or will software and artificial intelligence bring a hybrid –neither white nor black–future? This section starts with the intersection of mathematics and software technologies, followed by the prominent features and innovations of programming languages that have made a difference.

Keywords: Software, programming, current technologies, artificial intelligence

1. Giriş

Biz insanlar için matematiksel ifadelerin sadeleştirilmesi veya herhangi bir değere dönüştürülmesi vakit alır. Örneğin

$$2 + 30 + 4 + 6 \quad (1)$$

ifadesinin eşitinin

$$42 \quad (2)$$

olduğunu söylemek için az da olsa bir zamana ihtiyaç vardır. Oysa ki durum matematik için

$$2 + 30 + 4 + 6 = 42 \quad (3)$$

sonucuyla zamansız bir şekilde, bir nevi, önceden bellidir. Buradaki problem insan için sonucu belli olmayan ancak matematik açısından belirli bir değeri keşfetmek olarak görülür. Belki de tahta çubuklar ve abaküslerle başlayan bu süreç, toplama yapabilen akıllı cihazların icadıyla günümüze kadar gelen teknolojik bir serüveni oluşturur.

İşin ilginç, matematiksel hesapları gerçekleştirirken hangi yolların izleneceği meselesi de başka disiplinlerin kurulmasında öncülük etmiştir. Örneğin (1) ifadesi ile verilen toplama işlemi soldan sağa değerlemeye çalışan bir hesaplayıcı öncelikle 2 ve 30 sayılarını toplayıp 32 olan sonucu aklının bir köşesinde saklayabilir. Sıradaki sayı olan 4’ü de aklında tuttuğu 32 ile toplayıp sonucu 36 olarak raporlayabilir. Son sayı olan 6 da aklında tutulan 36’ya eklenirse 42 sonucu elde edilir. Bu bakış açısıyla, her seferinde yalnızca bir sayı aklında tutarak ve bir sonraki sayıyı aklındaki sayıya ekleyerek toplama işlemi gerçekleştirilebilir. Toplamda $4 - 1 = 3$ toplama işlemi yaparak aslında n adet sayıyı $n - 1$ adımda toplayan bir algoritma kullanmış olur.

Söz konusu toplama işlemini nihai bir değere dönüştürmenin başka tanımları da yapılabilir. Örneğin n adet sayının toplamını aşağıdaki gibi tanımlayalım:

- 1) Sıfır adet sayının toplamı sıfırdır.
- 2) n adet sayının toplamı, ilk sayı ile diğer sayıların toplamının toplamına eşittir.

Burada tanımlanan haliyle $2 + 30 + 4 + 6$ ifadesinin sonucu, 2 ve $30 + 4 + 6$ ifadelerinin toplamına eşittir. $30 + 4 + 6$ ise 30 ve $4 + 6$ ifadelerinin toplamına eşit olur. Ayrıca $4 + 6$ ifadesi 4 ve 6 'nın toplamına eşittir. 6 değeri tek başına kaldığından ve sıfır adet diğer sayının toplamı sıfırdır ifadesinden de faydalanarak 6 'nın kendine değeri verilmesi sağlanır. İşlemler başa doğru değeri girilirse 42 değeri elde edilir.

Yukarıda basit bir toplama işlemi için önerilen iki farklı yoldan bahsettik. Bu yolların kaç adım işlem gerektirdiği, akılda tutulması gereken sayıların sayısı, yardımcı işlem birimlerini gerektirip gerektirmediği veya işlem süresinin ne kadar olacağı gibi meseleler, problemlerin hem hangi yollarla çözülmesinin doğru olacağı, hem de hangi programlama dilinde kodlanmasının daha uygun olacağı gibi çözülmesi gereken yeni problemlerin doğmasına sebep olur. Bilgisayar Bilimi, Bilgisayar Mühendisliği ve Yöneylem Araştırması gibi kısmen ayrık disiplinlerin birbirini tamamlayıcı ve bazen de bağımsız üretimleri, hesaplama problemlerinin nasıl ve hangi araçlarla çözülebileceği konusunda yaklaşımlar sunar.

Bilgisayarlar için akıllı toplama cihazı ibaresinin seçilmesi tesadüfi değildir. Başlangıçta doğal zekanın bir taklidi gibi çalışabilecek mekanik veya dijital aletler olarak tasarlanan bilgisayarların sonraları günlük hesaplama işlerinin yerine getirilmesi için kullanılan cihazlara dönüştüğü görülmüştür. Diğer birçok sektörün de ihtiyacına bağlı olarak şekillenen dijital bilgisayarlar ve üzerlerinde geliştirilen yazılımlar, uzunca bir süre otomasyon sistemlerinin geliştirilmesinde kullanılagelmiş ve insan çabasıyla gerçekleştirilen bilgi akışı temelli işlerin hızlandırılması amacıyla kullanılmıştır. Sonraları zeka kavramı tekrar ön plana çıkmış, uzman sistemler, makine öğrenmesi ve veri bilimi gibi yeni çalışma alanlarının altında insanın gerçekleştirdiği başka türden işlerin de bilgisayarlarca gerçekleştirilmesi sağlanmaya başlanmıştır. Evrimi ve gelişimi doğrusal bir trend izlemeyen yazılım teknolojilerinin zamana damga vurmuş araçlarından hangisinin bir diğerinin doğumuna sebep olduğunu söylemek kolay değildir. Öyle ki *LISP (LISt Processing)* gibi olgun bir fonksiyonel programlama dilinin özellikle yapay zeka çalışmalarında kullanılması, sonrasında uzunca bir süre fonksiyonel programlama dilleri dışında başka paradigmaların benimsenmesi ve yapay zekadan uzaklaşılması, sonrasında tekrar fonksiyonel dil ailesine yaklaşma ve yapay zekanın tekrar gündeme gelmesi gibi döngüler, yazılım teknolojisindeki evrimin insanın bilişsel evrimine olan ayak

uydurmaların bir yansıması olduğunun göstergesidir. Fonksiyonel paradigmaya tekrar yakınlaşan insanlık *LISP* yerine *Haskell* gibi daha modern teknolojileri benimsemiş, bu arada da *Clojure* gibi yeni *LISP* türevlerinin oluşumuna tanık olmuştur. Değişim, dönüşüm ve gelişim kavramlarının böylesine iç içe geçtiği başka bir süreç, yazılım teknolojileri dışında tanık olmak bu derece kolay değildir.

$$2 + 30 + 4 + 6 \quad (4)$$

işleminin gerçekleştirilmesi için başta sunduğumuz iki algoritmanın uygun birer dilde kodlandığını düşünelim. İlk algoritmanın uygulanması için *Python* dilini seçelim. Algoritmanın bu dildeki bir uygulaması aşağıdaki gibi tanımlanabilir:

```
numbers = [2, 30, 4, 6]
sum = 0

for num in numbers:
    sum += num

print(sum)
```

Örnek kodda *numbers* adında bir dizi tanımlanır ve bu dizinin 2, 30, 4 ve 6 elemanlarını içermesi sağlanır. *sum* değişkeni başlangıç değeri sıfır olacak şekilde tanımlanır. Sonrasında bir döngü içerisinde *sum* değişkeninin birikimli bir şekilde 2, 30, 4 ve 6 sayılarını içermesi sağlanır. En sonunda *sum* değişkeninin değeri ekranda raporlanır. Sonuç 42'dir. Bu kısa program hem bir algoritmanın basit bir uygulamasını (*implementation*) sunar hem de belirli paradigmalar ile sınırlandırılmış bir yazılım teknolojisinin özelliklerini içerir.

Söz konusu program *sum* adlı değişkenin değerini 4 kez günceller. Bu haliyle bu programın hangi dilde yazıldığını bilmesek bile dilin “değeri değiştirebilir (*mutable*) değişkenler” içerdiğini görebilirdik. Öte yandan bu programlama dili, değişkenlerin belirli bir andaki durumu (*state*) üzerine kuruludur. Her değişken her an bir cümle (*statement*) ile değiştirebileceğinden programın belirli bir andaki durumundan bahsedebilirdik.

Yine şunu söylebiliriz ki, örnek program, bir toplama işleminin her aşamasını toplama işleminin tanımlarından bağımsız olarak, belki de işlemlerinin atomlara ayrılmış parçalarının *Python* yorumlayıcısına ayrı ayrı bildirilmesiyle oluşturulmuştur. Bu bağlamda söz konusu uygulamanın buyurucu (*imperative*) türden yazıldığı söylenebilir. Böylesine bir uygulamanın, bir dizinin elemanlarının toplamının matematiksel tanımı ile onun bir bilgisayar programın-

daki uygulaması arasındaki soyutlamanın bir örneği olduğu söylenebilir. Oysa ki bazı yazılım teknolojileri, örneğin fonksiyonel programlama ve buna imkan tanıyan *Haskell* dili, problemin matematiksel tanımı ile onun bir uygulaması arasındaki soyutlamanın derecesini azaltabilir. Aradaki ilişkiyi daha net göstermek için toplama işlemini gerçekleştiren yukarıdaki programı bir *Python* fonksiyonu ile tekrar yazalım:

```
def Summation(numbers):
    mysum = 0
    for num in numbers:
        mysum += num

    return mysum

print(Summation([2, 30, 4, 6]))
```

Örnek programdaki *Summation* fonksiyonu argüman olarak aldığı *numbers* adlı herhangi bir dizinin tüm elemanlarını toplayarak elde ettiği sonucu döndürür. Toplama işlemini bir fonksiyon ile tanımlayan program hala değeri değişebilen değişkenler ve belirli bir andaki program durumuna sahip olmak gibi özelliklerle buyurucu programlamanın bir örneğidir. *Summation* adlı fonksiyon kendisine argüman olarak verilen dizinin elemanlarını toplayıp döndürmekle sorumlu gibi görünebilir ancak bu fonksiyonun toplama işlemi dışında başka işlemler gerçekleştirerek yan etkiler (*side effects*) oluşturup oluşturmadığından – dilin buna izin vermesi bağlamında – emin olamayız. Buradaki eleştiri unsurları ışığında aşağıda tanımlanmış *Haskell* fonksiyonunu inceleyelim:

```
mysum :: [Double] -> Double
mysum [] = 0.0
mysum (x:xs) = x + mysum xs
```

Örnek program, bir dizinin elemanlarının toplamını elde eden bir fonksiyonun tanımından ibarettir. Öncelikle *mysum* adlı fonksiyonun ondalıklı sayılar kümesinin elemanlarının bir dizisinden ondalıklı sayılar kümesine tanımlı olduğu bildirilir. Sonrasında, hiçbir elemanı olmayan bir dizinin elemanlarının toplamının sıfır olduğu bildirilir. Son aşamada bildirilen ise bir dizinin elemanları toplamının o dizinin ilk elemanı ile diğer elemanlarının toplamının toplamı olduğudur. Fonksiyon bu haliyle eksiksiz olarak tanımlanmıştır. Bu fonksiyon bir *Haskell* programında

```
mysum [2, 30, 4, 6]
```

şeklinde kullanılabilir. Bu haliyle program, söz konusu işlemin matematiksel tanımı ile onun bilgisayar programı tanımı arasındaki soyutlamanın derecesini azaltır. Öte yandan program, değeri değişebilen değişkenlerin varlığına izin vermez. Tüm bunların yanında programın belirli bir zamanda tanımlanmış belirli bir durumu yoktur. Ayrıca fonksiyon tanımlandığı işlem-den sorumlu olarak yan etkiler yaratmaz. Burada verilen programın fonksiyonel programlama paradigmaları ile yazıldığını söyleyebiliriz.

Elbette buraya kadar saydığımız özellikler bir uygulamanın fonksiyonel paradigmalara ile yazıldığı sonucuna ulaşmadaki kriterleri tam olarak sağlamaz. Bir fonksiyonun başka bir fonksiyonu argüman olarak alabilmesi, bir fonksiyonun başka bir fonksiyonu değer olarak döndürebilmesi, fonksiyonların aynı argümanlarla çağırılması durumunda hep aynı sonucu döndürmesi, durumu değişebilen değişkenlere izin verilmemesi gibi özellikler fonksiyonel programlama paradigmasının en önemli diğer unsurlarından birkaçını oluşturur. Yüksek dereceden fonksiyonlar ile adlandırılan bu özelliğin bir örneği aşağıdaki R kodunda verilmiştir:

```
f <- function(p) {
  return(
    function(x) {
      return (x^p)
    }
  )
}
Square <- f(2)
Cube <- f(3)

print(Square(5))
print(Cube(2))
```

Gösterilen R kodundaki f fonksiyonu *return* ile başka bir fonksiyonu döndürmektedir. Döndürülen bu fonksiyon öncesinde p ile belirtilen dereceden kuvvet alan başka bir fonksiyondur. Sonraki satırlarda görüldüğü üzere f fonksiyonu 2 argümanı ile çağırıldığında kare alan, 3 argümanı ile çağırıldığında ise küp alan bir fonksiyonu döndürür. *Square* ve *Cube* adları verilen bu fonksiyonlar, kendilerinden beklenildiği gibi 5 ve 2 argümanları ile çağırıldıklarında sırasıyla 25 ve 8 değerlerini döndürmektedir.

Dillerin zamana bağlı gelişimi ve değişimi, paradigma değişimlerine bağlı olarak keskin geçişler göstermekle birlikte, bazı diller birden fazla programlama paradigmasının kullanımına izin verir. Örneğin *Haskell* sert bir biçimde fonksiyonel programlama paradigmasının araçlarının kullanımına bağlı kalırken, *F#* ve *Scala* gibi diller hem fonksiyonel hem de buyurucu paradigmanın kullanımına izin verir. Özel olarak buyurucu paradigmanın altında incelenen nesne yönelimli paradigma (*Object-Oriented Programming*) her ne kadar temel veri

yapısı olarak sınıfları (*class*) benimsese de veri yapılarının birer duruma sahip olmaları ve fonksiyonların matematiksel fonksiyonlardan çok metod (*method*) görevini gerçekleştirmesi bağlamlarında buyurucu özellik taşırlar. *Java* gibi bazı dillerin yeni versiyonlarının, dilin orijinalinde buyurucu ve özel olarak da nesne yönelimli olmasına rağmen, fonksiyonel özellikleri de içerecek şekilde geliştirildiği görülmektedir (Urma ve diğerleri, 2014).

Dillerin ne şekilde geliştiği ve değiştiği konusundaki bir diğer etken de değişen yazılım mimarileridir. Bu yazılım mimarileri de büyük oranda donanımdaki olanakların artması ile ilgili olabilmektedir. Birden fazla ayırık işlem birimine (çekirdek - *Core*) sahip işlemcilerin (*Central Processing Unit – CPU*) yaygınlaşması ile aynı bilgisayarda tek bir veri nesnesinin durumunu aynı anda değiştirmeye çalışan iş parçacıklarının kullanım oranı artmıştır. Gerçekleştirilecek işlem, standart bir kişisel bilgisayar yardımıyla uzun zaman gerektiriyor olabilir. Bu durumda, bilgisayarların yürütmekte olduğu tüm işleri işlem birimlerine paylaşması, ancak her bir işlemin bir işlem biriminde gerçekleştiriliyor olduğu önemli bir ayrıntıdır. Yani kişisel bilgisayarda başlatılan bir hesaplama, özellikle bir ayarlama yapılmadığı müddetçe, bilgisayarın sahip olduğu işlem birimi sayısından bağımsız tek bir işlem biriminde ele alınır. Performans artışı sağlamak için, geliştirilen programın birden fazla parçaya bölünerek farklı işlem birimlerinde hesaplanmasını, sonra da sonuçların birleştirilmesini sağlamak mümkündür. Bu şekilde, program kodunun eş zamanlı olarak çok sayıda işlem biriminde çalıştırılmasına paralel programlama adı verilmektedir (Kuhail ve diğerleri, 2018). Bu yöntem daha fazla enerji harcanmasını gerektiriyor olsa da sonuçta ciddi bir performans artışı sağlamaktadır (Lorenzon ve diğerleri, 2015). Performans artışının miktarı doğrusal olmamakla birlikte işlem birimleri arasındaki senkronizasyon ve iletişim bant genişliği sınırları sebebiyle işlem birimi sayısı da ölçeklendirilememektedir (Suleman, Qureshi ve Patt, 2008). Bir duruma dayanmayan ve değeri değişmeyen veri yapılarını hesaplamanın merkezine alan fonksiyonel paradigmaya olan ihtiyacın artması, paralel hesaplamaya olan ihtiyacın artması ile bu bağlamlarda ilişkilidir.

Her ne kadar hangi problemin nasıl çözüleceği konusunda büyük farklılıklar yaratsa da bir yazılımın geliştirilmesinde kullanılan dil ve paradigma son kullanıcı açısından büyük oranda önem arz etmemektedir. Günümüzde birçok uygulama bir İnternet tarayıcısı veya mobil cihazlar üzerinde hizmet verebilmektedir. Mobil uygulamaların bazılarının da arka planda bir İnternet tarayıcısı üzerinden *HTML5* teknolojisi ile çalıştığı görülür. *HTML5* bir işaretleme dili olarak Web'in merkezinde yatarken, görsel öğelerle *DOM (Document Object Model)* teknolojisi ile haberleşen *JavaScript* dilinin de temel hesaplama ihtiyacını giderdiği görülür. Çeşitli paradigmaları benimseyen yüzlerce programlama dilinin sunucu ve masaüstünde ça-

Bu konudaki ilk girişimlerden birini teknoloji şirketi Google, *Dart* dilini üreterek gerçekleştirmiştir (Bracha, 2015). *Dart* dili modern tasarımıyla *JavaScript* ile karşılaşılan geliştirme problemlerini ortadan kaldıracak şekilde geliştirilmiştir. Ancak diğer İnternet tarayıcılarını geliştirenlerin aynı fikirde olmayışı, *Dart* dilinin bir standart olarak kabul edilmesinin önünde bir engel oluşturmuştur. *Dart* dilinde yazılan programların İnternet tarayıcılarının tümünde çalışan *JavaScript* kodu üretecek şekilde kullanılması bir çözüm olarak sunulurken *JavaScript*'e dönüştürülen yeni dillerin de ortaya çıkması dönemin ihtiyaçlarının bir sonucu olmuştur. Bir diğer teknoloji şirketi Microsoft'un *TypeScript* dili, *JavaScript*'in bir üst versiyonu olarak tip tanımlarını da içerecek şekilde geliştirilmiş ve *JavaScript*'e dönüştürülebilir bir dil olarak yerini almıştır (Bierman ve diğerleri, 2014). Günümüzde *Dart* ve *TypeScript*'in yanında *ScalaJS* (Doeraene ve diğerleri, 2016), *Kotlin* (Nozik, 2019), *ClojureScript* (McGranaghan, 2011), *ELM* (Mirzayev, 2016), *PureScript* (King ve diğerleri, 2019), *CoffeeScript* (Burnham, 2015; Lee, 2014) ve *Haxe* (Cannasse, 2008) gibi diller Web ortamında daha karmaşık projelerin yazımında tercih edilir olmuştur. Hangi dilde yazılırsa yazılsın, oluşan hedef programın *JavaScript*'e derlenmesi zorunluluğu devam eder.

İnternet tarayıcı üzerinde çalışan *JavaScript* kodu makine koduna derlenmek (*compile*) yerine yorumlanarak (*interpret*) veya çalışma anında makine koduna yakın bir dile çevrilerek veya da makine koduna çevrilerek (*just in time compiling*) çalışmaktadır. Önceden derlenmiş olmayan bir kodun gerek yorumlanması gerek de çalışma anında derlenmesi bir hız kaybına sebep olabilmektedir. Değişkenlerin önceden tanımlı bir tipe sahip olmaması ve çalışma anında bir değişkenin herhangi tipte bir değeri alabilmesi, çalışma sürelerini de uzatmaktadır. Bu eksiği gidermek için tasarlanan *WebAssembly* teknolojisi ile yüksek miktarda hesaplama hızı gerektiren işlemlerin makine koduna yakın bir şekilde derlenmesi ve bu derlenmiş kodun *JavaScript* üstünde yorumlanması sağlanmaya çalışılır (Watt, 2018). Birçok dilden *WebAssembly* koduna derleme mümkün olduğundan *C*, *C++* ve *Rust* gibi performans odaklı kod yazımında kullanılan derleyicilerden *WebAssembly* kodu üretmek mümkün hale getirilmiştir (Zakai, 2011). *WebAssembly* dili şimdilik sahip olduğu kısıtlı haliyle *JavaScript* ile birlikte uyum içinde çalıştırılabilir. Ancak yine de geriye uyumluluğu sağlamak gibi bir zorunluluk yüzünden onlarca bilimsel ilerleme ve teknik gelişmenin İnternet tarayıcılara doğrudan yansıtılamaması ve dolaylı gelişim çabaları, Web'in en büyük problemlerinden birini oluşturur. Ayrıca çok kanallı (*multi-threading*) programlamayı henüz desteklemeyişi de Web üzerine kurulan modern teknolojilerin de istenilen düzeyde başlatılamadığının da göstergesidir (Watt ve diğerleri, 2019).

Donanım ve yazılım teknolojilerinin birlikte gelişmesi çoğu zaman İnternet tarayıcılardaki dezavantajın etkisini de azaltmıştır. Şöyle ki bulut (*cloud*) ortamda çalışan ve temel işlemleri uzaktan gerçekleştirecek şekilde yazılan, yazılım mimarisinde hızla yükselen platform bağımsız bir yazılım bileşeni olan Web servisler (Web services) ile hesaplama yükünün bir kısmının sunuculara yüklenmesi sağlanmıştır (Srivastava & Koehler, 2016). Uzaktaki bir fonksiyonun çağırılması şeklinde tanımlı basitleştirilebilen Web servislerin yazıldığı dilden bağımsız olarak *JavaScript* ile birlikte kullanılması mümkündür. Uzak fonksiyonların çağırılması istemcideki (*client*) iş yükünü azaltırken, istemcide mümkün olmayan işlerin de gerçekleştirilmesi bu yollarla sağlanmış olur. Günümüzde birçok Web uygulaması mimarisinde *JavaScript* bilgiyi *HTML5* ve *CSS (Cascade Style Sheet)* ile kullanıcıya göstermek ve gerekli bilgiyi sunuculardan talep etmekle yükümlü olacak şekilde çalışır. Verinin tutulması (*store*) ve sorgulanması (*query*) gibi veritabanı işlemleri ile isteklerin yanıtlandırılması sunucu düzeyinde sağlanmış olur. İstemci ve sunucuların iletişim kurmaları başlangıçta ilkel *HTML* form etiketleriyle gerçekleştirilebilirken, *AJAX (Asynchronous JavaScript And XML)* ve sonrasında *WebSocket* teknolojileri ile bu etkileşimin kolaylaştırıldığı görülmüştür (Pimentel ve Nickerson, 2012). Günümüzde tasarlanan modern bir Web uygulaması farklı sunucularda servisler olarak tanımlanmış oturum açma işlemlerini, veri analizi, veri tabanı yönetimi ve hesaplama gereksinimlerini istemcide sunulan kullanıcı etkileşimi ile hizmet verebilmektedir (Jackson ve diğerleri, 2010).

Buraya kadar sunulan kısa bir uzaktan bakış, içerisinde onlarca bilimsel ve teknolojik gelişmeyi barındırır. Tüm bunların yanında makine öğrenmesi ve yapay zeka başlıkları altında anılan algoritmalar, verilen hizmetlerin arka planında birer yardımcı olarak çalışmaya devam eder. Bazı algoritmalar da güncel yazılım teknolojilerinin birer sonucu olarak daha kolay uygulanabilir hale gelebilmiştir.

Bir geliştiricinin belirli bir amacı sağlamak için yazdığı bir bilgisayar programını ele alalım. Burada geliştiricinin asıl zeka unsuru veya aktör olduğunu düşünebiliriz. Bu aktör belirli girdiler ile çalışan bir programın istenen özellikteki çıktıları elde etmesini sağlamaktadır. Bazen belirli bir girdinin yine belirli bir çıktının oluşmasına sebep olduğu bilinir ancak aradaki işlemlerin ne olduğu bilinmeyebilir. Diğer bir deyişle, eldeki 2, 30,4 ve 6 sayılarının bir işlemden geçirildiğini, bu işlemin sonunda 42 sayısının elde edildiğini, ancak bu işlemin ne olduğunun bilinmediğini varsayalım. Genetik Programlama (Koza, 1994) (*Genetic Programming – GP*) adıyla anılan ve makine öğrenmesi - yapay zeka başlıkları altında incelenen bir yöntemle burada verilen problemlerin benzerleri çözülebilir.

Genetik programlamada, belirli girdi ve çıktılar arasındaki ilişkiyi açıklayabildiği düşünülen bilgisayar programı veya matematiksel ifadelerin rastgele bir topluluğu oluşturulur. Bu topluluğun içinden problemi çözme konusunda daha başarılı olan iki tanesi, yeni çözümler oluşturulmak üzere seçilir (*selection*). Seçilen bu çözümler, çözümlerin gösterildiği veri yapısına da bağlı olarak çaprazlanır (*crossover*). Çaprazlama sayesinde iyi olduğu düşünülen çözümlerin parçalarının birleştirilmesi ile daha iyi çözümlerin oluşacağı beklenir. Oluşan bu yeni çözümlerin (*offspring*) bazı parçalarının düşük olasılıklarla değişmesi sağlanır (*mutation*). Tekrarlanan nesiller (*generation*) boyunca problemi çözme konusunda daha başarılı aday çözümlerin elde edilmesi ve belki de problemi çözen bir bilgisayar kodunun bulunması sağlanır (Poli ve diğerleri, 2008).

Büyük oranda genetik algoritmalar (*genetic algorithms*) adı verilen yöntemler ailesini miras alan genetik programlamada farklı olan, aday çözümlerin birer matematiksel ifade veya bilgisayar programı olmasıdır. Genetik algoritmada herhangi bir $y = f(x_1, x_2, \dots, x_p)$ fonksiyonunu en küçük veya en büyük yapan $x_1, x_2, \dots, x_p$ değerlerini aramak asıl amaç iken genetik programlamada belirli bir girdi karşısında beklenen bir çıktıyı veren bir matematiksel ifadeyi veya bilgisayar programını bulmak asıl amaç haline gelir.

Bazı hesaplama problemlerine doğayı taklit etme yoluyla çözüm aramak genetik algoritma ve genetik programlama ile sınırlı kalmamıştır. Gelişmiş canlı beynini model alarak belirli girdilerin hangi tip sınıfa sahip olduğunu aramak veya bu girdiler ile yine belirli bazı çıktılar arasındaki matematiksel ifadeyi keşfetmek için yapay sinir ağları (*artificial neural networks*) uzun süredir kullanılmaktadır. Bağımsız değişkeni başka bir lojistik fonksiyon olan lojistik fonksiyonların oluşturduğu bu bağlı yapılar, uzun süredir sınıflama ve regresyon problemlerinde kullanılmaktadır. Genetik programlamadan farklı olarak ileri beslemeli (*feed forward*) yapay sinir ağları önceden belirlenmiş bir fonksiyonel ifadenin tahmin hatasını en küçükleyecek şekilde katsayılarını aramaktadır (Buscema, 1998). Bu katsayı arama işleminde genetik algoritmaların da kullanılması iç içe geçmiş yapının en büyük örneklerinden birini oluşturur. Kaç katmanın yer alması gerektiği, katmanlardaki nöron sayıları ve nöronlardaki aktivasyon fonksiyonlarının türleri ve nöronlar arasındaki katsayıların aranması gibi meseleler birlikte ele alındığında yapay sinir ağları geniş boyutlu birer optimizasyon problemidir ve genetik algoritmaların bu problemde genel (*global*) optimumu arayan bir çözüm yöntemi olarak kullanılması, bu konudaki literatürün erken zamanlarında ele alınmıştır (Montana ve Davis, 1989; Suganuma ve diğerleri, 2017).

Kuş ve balık sürülerinin besin kaynaklarını veya göç rotalarını belirlerken izledikleri süreçlerin taklit edilmesi ile Parçacık Sürü Optimizasyonu (*Particle Swarm Optimization - PSO*)

(Poli ve diğerleri, 2007), karınca kolonilerinin besin kaynaklarını ararken takip ettikleri rotaların belirlenmesi süreçlerinin taklit edildiği Karınca Kolonisi Optimizasyonu (*Ant-colony Optimization*) (Dorigo ve diğerleri, 2006), bal arılarının davranışlarını taklit eden Bal Arısı Optimizasyonu (*Honey Bee Swarm Optimization*) (Karaboğa, 2005; Karaboğa ve Baştürk, 2007) ve alfa erkeği barındıran canlı sürülerinin çift olma davranışlarının taklit edildiği alfa erkeği genetik algoritmalar (Drezner ve Drezner, 2019; Satman ve Akadal, 2019) bahsedilen doğanın taklit edilmesine dayalı bazı optimizasyon algoritmalarıdır. Bu yöntemler evrimsel optimizasyon algoritmaları adıyla da anılmaktadır (Krishna ve Ravi, 2016).

Güncel yazılım teknolojileri meselesinden artık ayrılamayacak olan ve Alan Turing'in 1950 yılında *Mind* dergisinde yayınlanan *Computing Machinery and Intelligence* başlıklı makalesinin (Turing, 1950) neredeyse vücut bulduğu günümüzde yapay zeka ve makine öğrenmesinin beklentilerin çok altında üretimleri olduğuna da mutlaka değinmeliyiz. Amir Husain'nin 2017'de yazdığı *The sentient machine: The coming age of artificial intelligence* (Husain, 2017) başlıklı kitapta da belirtildiği gibi yapay zekanın bilgisayarların donanım bağlamında gelişimi ile sürekli ilerlemiş olduğu açıktır. Ancak bu ilerlemenin içeriği tüm mümkün durumların hızlı bir şekilde analiz edilmesine dayanması veya akıl yürütme temelli olması bakımından farklı kıvrımlara sahiptir. Google'ın DeepMind¹ projesinin bir parçası olarak geliştirilen AlphaGo² algoritmasının *Go* oyunundaki başarısının kaba kuvvet (*brute force*) gibi tüm mümkün durumların analiz edilmesine dayalı algoritmalarla sağlanamayacağı belirtilmiştir (Wang, 2017). Diğer bir taraftan, birçok yapay öğrenme algoritması, öncesinde toplanmış büyük miktarda öğrenme verisine ihtiyaç duyabilmektedir. Geçmiş veriyi bir tür genelleme veya kurallar dizisi oluşturmak için kullanan bu algoritmaların idareten iş gören ve kaba tabiriyle saçma bulunabilecek sonuçlar ortaya çıkardığı sıklıkla gözlemlenir. Girdi ve çıktı değişkenleri arasındaki ilişkiyi büyük hassasiyetle keşfeden bir yapay sinir ağının keşfettiği bu ilişki bir nevi kara kutudur (*black box*) ve bulunan ilişkinin anlamlandırılması çoğu zaman güçtür. Tüm bunlarla birlikte girdi ve çıktı değişkenleri arasındaki ilişkiyi keşfetmeye yönelik geliştirilen genetik programlama, bir insan uzmanın saçma diyebileceği alternatif bir kurallar dizisi, matematiksel ifade veya bilgisayar programı raporlayabilir. Akıl yürütme eylemi zekice mi yapılmalıdır yoksa saçma bir keşif iş görüyorsa yeterli midir? İş gören bir bulgu, iş gördüğü için değerli kabul edilebilir mi yoksa matematiksel açıdan da estetik olmalı mıdır?

Buradaki akıl yürütme değişiminin insanın karar alma sürecinde yaşadığı sürecin taklit edilmesi anlamında olduğunu belirtelim. Ayrıca gelişimi uzun bir süreye yayılmış olan canlı

1 URL: <https://deepmind.com/> Erişim Tarihi: 25.03.2020

2 URL: <https://deepmind.com/blog/article/alphago-zero-starting-scratch> Erişim Tarihi: 25.03.2020

beyninin karar alırken öncelikle canlı kalmak ve hızlı karar alıp daha iyi bir duruma ulaşmak olduğu düşünülürse, insandaki bilişsel kusurların da onu taklit eden makinelere bir nevi sirayet etmesi de doğal karşılanır bir durum haline gelebilecektir. Hata olasılığı %1 olan robot cerrahı mı yoksa hata olasılığı %2 olan insan cerrahı mı tercih edersiniz sorularına farklı yanıtlar verebiliyor oluşumuz bu bilişsel kusurların insan beyninde silinemez şekilde yer ettiğinin ve bir makinenin benzer bir gelişim süreci izlemediğinden irrasyonel karar almayacağı söylenebilir. Burada biraz da felsefeye alan açılmaktadır: Rasyonel olan nedir?

Aşağıdaki tabloyu inceleyelim. A ve B birtakım değerler almışken Y de bu değerlere bağlı olarak bazı değerler alır:

A	B	Y
1	0	1
0	1	1
1	1	0
0	0	0

XOR mantık operatörü için doğruluk tablosunu oluşturan bu değerler için tek bir kuraldan bahsedebiliriz: A ve B aynı değerleri alırsa Y sıfır değerini alır, tersi durumunda Y bir değerini alır. Ancak bir yapay zeka A , B ve Y arasındaki ilişkiyi

$$Y = |A - B| \quad (5)$$

olarak raporlayabilirdi. Şüphesiz her durum için Y , A ve B 'nin pozitif farkıdır. Bu iki kural doğallıkla aynı sonuçları üretir. Ancak anlatım ve gösterim bakımından çok farklıdır. İki bulgu da saçma olmaktan çok uzaktır. Hatta iki durumdan yalnızca birini düşünmüş olan okuyucu için diğeri heyecan verici bile gelebilir. Ancak yüzlerce akıl dışı çözümü olan bir problemde bu durum başa çıkılması zor bir probleme dönüşebilirdi. Günümüzdeki haliyle bazı yapay zeka algoritmalarının belirli bir hata fonksiyonunu en küçükleyecek şekilde yalnızca çözüm almaya odaklı çalışması ve tüm durumların çözüme dahil edilemeyişi benzer güçlüklerin oluşmasına neden olur. Bazen de insanların da içine düştüğü öğrenme problemleri yapay öğrenmede de vücut bulabilmektedir. Aşırı öğrenen bir yapay öğrenme algoritması, eldeki yapay zekanın veriyi ezberlemesi ve genellemeyi doğru gerçekleştirememesi şeklinde yorumlanır. Şüphesiz benzer bir süreci insan da gelişiminin erken zamanlarında birçok kez yaşar. Genellemeyi doğru şekilde gerçekleştirip ezberden kaçınmanın yolları nelerdir?

Hadoop, *MapReduce*, *Apache PIG* ve *NoSQL* gibi teknolojilerle yapısal olmayan ve büyük denebilecek verileri saklayabildiğimiz, sorgulayabildiğimiz ve anlamlandırmaya çalıştığımız

bir çağın içinde karar destek sistemleriyle diyabet ve onkoloji hastalarının durumlarını anlık takip edip gerekli uyarı, öneri ve müdahaleleri gerçekleştirebiliyoruz (de Bruin ve diğerleri, 2018; O'Connor ve diğerleri, 2011). Bir yandan bilgisayar bilimi gerekli donanımda iyileştirmeler yaparken; bilgisayar bilimi, yöneylem araştırması ve istatistik gibi disiplinler yeni algoritmaların geliştirilmesinde katkı sağlamakta. Bir yandan büyük veri, veri bilimi, veri eşeleme gibi disiplinler ortaya çıkarken, disiplinler arası işbirliği güncel yazılım teknolojilerinin geleceğini büyük bir hızla belirlemektedir.

İnsanlık birçok zaman aşırı kehanetlerin gerçekleşmediğine ve tabii geleceğin ara çözümler olarak ortaya çıktığına şahit oldu. İnsanları ele geçirip tüm dizginleri ele alan bir yapay zekanın mı yoksa insanla entegre çalışan, irili ufaklı çevrimiçi elektronik cihazın iletişimi ile baskınlaşmış (IoT – *Internet of Things*), yüzlerce mesleğin sonunu getirmiş ancak yüzlerce yeni mesleğin de ortaya çıkmasına sebep olmuş hibrit bir oluşumun mu daha olası olduğu düşünmeye değerdir. Gelecek henüz kısmen belirsizdir ancak daha önce de gördüğümüz gibi yapay zekanın yazılım teknolojilerini, yazılım teknolojilerinin de yapay zekayı etkileyeceğini bir süre daha göreceğimiz açıktır.

Kaynakça / References

- Bracha, G. (2015). The Dart programming language. Addison-Wesley Professional.
- Bierman, G., Abadi, M., & Torgersen, M. (2014, July). Understanding typescript. In European Conference on Object-Oriented Programming (pp. 257-281). Springer, Berlin, Heidelberg.
- Burnham, T. (2015). Coffeescript: accelerated Javascript development. Pragmatic Bookshelf.
- Buscema, M. (1998). Theory: Foundations of artificial neural networks. Substance Use & Misuse, 33(1), 17-199.
- Cannasse, N. (2008). Using haxe. In The essential guide to open source flash development (pp. 227-244). Apress.
- de Bruin, J. S., Schuh, C., Seeling, W., Luger, E., Gall, M., Hütterer, E., ... & Schindler, K. (2018). Assessing the feasibility of a mobile health-supported clinical decision support system for nutritional triage in oncology outpatients using Arden Syntax. Artificial intelligence in medicine, 92, 34-42.
- Doeraene, S., Schlatter, T., & Stucki, N. (2016, October). Semantics-driven interoperability between Scala.js and JavaScript. In Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala (pp. 85-94).
- Dorigo, M., Birattari, M., & Stutzle, T. (2006). Ant colony optimization. IEEE computational intelligence magazine, 1(4), 28-39.
- Drezner, Z., & Drezner, T. D. (2019). The alpha male genetic algorithm. IMA Journal of Management Mathematics, 30(1), 37-50.
- Githut 2.0 (2020). A small place to discover languages in github. Github.Com. https://madnight.github.io/github/#/pull_requests/2019/4 Erişim Tarihi: 24.3.2020.
- Husain, A. (2017). The sentient machine: The coming age of artificial intelligence. Simon and Schuster.
- Jackson, K. R., Ramakrishnan, L., Muriki, K., Canon, S., Cholia, S., Shalf, J., ... & Wright, N. J. (2010, November). Performance analysis of high performance computing applications on the amazon web services cloud. In 2010 IEEE second international conference on cloud computing technology and science (pp. 159-168). IEEE.
- Karaboga, D. (2005). An idea based on honey bee swarm for numerical optimization (Vol. 200, pp. 1-10). Technical report-tr06, Erciyes university, engineering faculty, computer engineering department.

- Karaboga, D., & Basturk, B. (2007, June). Artificial bee colony (ABC) optimization algorithm for solving constrained optimization problems. In *International fuzzy systems association world congress* (pp. 789-798). Springer, Berlin, Heidelberg.
- King, J., Ng, N., & Yoshida, N. (2019). Multiparty Session Type-safe Web Development with Static Linearity. *arXiv preprint arXiv:1904.01287*.
- Krishna, G. J., & Ravi, V. (2016). Evolutionary computing applied to customer relationship management: A survey. *Engineering Applications of Artificial Intelligence*, 56, 30-59.
- Koza, J. R. (1994). *Genetic programming II: Automatic discovery of reusable subprograms*. Cambridge, MA, USA, 13(8), 32.
- Kuhail, M. A., Cook, S., Neustrom, J. W., & Rao, P. (2018). Teaching parallel programming with active learning. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (pp. 369-376). IEEE.
- Lee, P. (2014). *CoffeeScript in action*. Manning Publications Co.
- Lorenzon, A. F., Sartor, A. L., Cera, M. C., & Beck, A. C. S. (2015). Optimized use of parallel programming interfaces in multithreaded embedded architectures. In *2015 IEEE Computer Society Annual Symposium on VLSI* (pp. 410-415). IEEE.
- McGranaghan, M. (2011). *Clojurescript: Functional programming for javascript platforms*. *IEEE Internet Computing*, 15(6), 97-102.
- Mirzayev, K. (2016). *Analyzing Functional Reactive Programming in ELM and JavaScript* (Doctoral dissertation, Long Island University, The Brooklyn Center).
- Montana, D. J., & Davis, L. (1989, August). Training Feedforward Neural Networks Using Genetic Algorithms. In *IJCAI* (Vol. 89, pp. 762-767).
- Nozik, A. (2019, October). Kotlin language for science and Kmath library. In *AIP Conference Proceedings* (Vol. 2163, No. 1, p. 040004). AIP Publishing LLC.
- O'Connor, P. J., Sperl-Hillen, J. M., Rush, W. A., Johnson, P. E., Amundson, G. H., Asche, S. E., ... & Gilmer, T. P. (2011). Impact of electronic health record clinical decision support on diabetes care: a randomized trial. *The Annals of Family Medicine*, 9(1), 12-21.
- Pimentel, V., & Nickerson, B. G. (2012). Communicating and displaying real-time data with websocket. *IEEE Internet Computing*, 16(4), 45-53.
- Poli, R., Kennedy, J., & Blackwell, T. (2007). Particle swarm optimization. *Swarm intelligence*, 1(1), 33-57.
- Poli, R., Langdon, W. B., McPhee, N. F., & Koza, J. R. (2008). *A field guide to genetic programming*. Lulu. Com.
- Satman, M. H., & Akadal, E. (2019). Performance Comparison of Tth Specialized Alpha Male Genetic Algorithm with some Evolutionary Algorithms. *Trakya University Journal of Social Science*, 21(1).
- Severance, C. (2012). Javascript: Designing a language in 10 days. *Computer*, 45(2), 7-8.
- Srivastava, B., & Koehler, J. (2003). Web service composition-current solutions and open problems. In *ICAPS 2003 workshop on Planning for Web Services*, 35. 28-35.
- Suganuma, M., Shirakawa, S., & Nagao, T. (2017, July). A genetic programming approach to designing convolutional neural network architectures. In *Proceedings of the Genetic and Evolutionary Computation Conference* (pp. 497-504).
- Suleman, M. A., Qureshi, M. K. & Patt Y. N. (2008). Feedback-driven threading: power-efficient and high-performance execution of multithreaded workloads on CPMs. In *Proceedings of the 13th International Conference Architectural Support for Programming Language and Operating Systems* (pp. 277-286).
- Turing, A. (1950). *Computing Machinery and Intelligence*. *Mind*. Vol. LIX,?. 236.
- Urma, R. G., Fusco, M., & Mycroft, A. (2014). *Java 8 in Action: Lambdas, Streams, and functional-style programming*. Manning Publications Co.

- Wang, F. Y., Zhang, J., Wei, Q., Zheng, X., & Li, L. (2017). PDP: Parallel dynamic programming. *IEEE/CAA Journal of Automatica Sinica*, 4(1), 1-5.
- Watt, C. (2018, January). Mechanising and verifying the WebAssembly specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs* (pp. 53-65).
- Watt, C., Rossberg, A., & Pichon-Pharabod, J. (2019). Weakening WebAssembly. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA), 1-28.
- Zakai, A. (2011, October). Emscripten: an LLVM-to-JavaScript compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion* (pp. 301-312).