PROGRAMMING LANGUAGE THEORY

A RESOURCE

FOR

TURKISH LITERATURE

BY

MERTCAN DAVULCU

SEPTEMBER 2022

CONTENTS

CONTENTS II
INFORMATION V
BİLGİLENDİRME V
ABSTRACT VI
ÖZ VII
1. PROGRAMLAMA DİLİ TEORİSİ NEDİR ..... - 8 -
2. PROGRAMLAMA DİLİ ..... - 8 -
3. BİLGİSAYAR NEDİR VE NASIL ÇALIŞIR ..... - 9 -
4. ALGORİTMA ..... - 10 -
5. KOD TÜRLERİ ..... - 10 -
5.1 Kaynak Kod (Source Code) ..... - 10 -
5.2 Ara Gösterim (Intermeatiate Representation - IR) ..... - 11 -
5.3 Kayıt Transfer Dili (Register Transfer Language – RTL) ..... - 11 -
5.4 Obje Kodu (Object Code) ..... - 11 -
5.5 Bayt Kodu (Bytecode) ..... - 11 -
5.6 Makine Kodu (Machine Code) ..... - 11 -
6. ÇEVİRİCİ (TRANSPILER) ..... - 12 -
7. DERLEYİCİ (COMPILER) ..... - 12 -
8. YORUMLAYICI (INTERPRETER) ..... - 13 -
9. P-KOD MAKİNESİ (P-CODE MACHINE) ..... - 14 -
10. ÇALIŞMA ZAMANI (RUNTIME) ..... - 14 -
11. DERLEME ZAMANI (COMPILE TIME) ..... - 14 -
12. TİP GÜENLİĞİ (TYPE SAFETY) ..... - 14 -
12.1 Statip Tip Denetimi (Static Type Checking) ..... - 15 -
12.2. Dinamik Tip Denetimi (Dynamic Type Checking) ..... - 15 -
12.3 Jenerik Tipler (Generic Types) ..... - 15 -
12.4 Ördek Yazma (Duck Typing) ..... - 16 -
13. DERLENEN VE YORUMLANAN DİLLERİN KARŞILAŞTIRILMASI ..... - 16 -
14. KAYNAK KODUN MAKİNE KODUNA DÖNÜŞÜMÜ ..... - 19 -
15. SÖZDİZİMİ (SYNTAX) ..... - 19 -
15.1 Yürütülebilir İfade (Statement) ..... - 19 -
15.2 Değerlendirilebilir İfade (Expression) ..... - 19 -
16. DENEYSEL DİL TASARIMI ..... - 19 -
17. JETONLAR (TOKEN) ..... - 21 -

18. SÖZCÜKSEL ANALİZ (LEXING) ..... - 22 -
19. SOYUT SÖZDİZİMİ AĞACI (ABSTRACT SYNTAX TREE - AST) ..... - 27 -
20. VERİLERİN İŞLENMESİ ..... - 34 -
20.1 Atama (Assignment) İfadelerinin İşlenmesi ..... - 35 -
20.2 Değerlendirilebilir İfadelerinin (Expression) İşlenmesi ..... - 35 -
20.3 Print İfadelerinin (Print Statement) İşlenmesi ..... - 35 -
20.4 Değerlendirme (Evaluation) Algoritması ..... - 36 -
21. YORUMLAYICIYI (INTERPRETER) ANLAMAK ..... - 36 -
22. ÇEVİRİCİYİ (TRANSPILER) ANLAMAK ..... - 38 -
23. DERLEYİCİYİ (COMPILER) ANLAMAK ..... - 39 -
23.1 AOT Derlemesi ..... - 40 -
23.2 JIT Derlemesi ..... - 41 -
23.3 Derleyici Yapısı ..... - 42 -
23.3.1 Ön Uç (Compiler Front-End) ..... - 42 -
23.3.2 Arka Uç (Compiler Back-End) ..... - 42 -
24. MONTAJ DİLİ (ASSEMBLY) ..... - 43 -
25. PROGRAMLAMA DİLLERİNİN SINIFLANDIRILMASI ..... - 44 -
26. TASARIM MOTİVASYONU ..... - 44 -
26.1 Programlama Dillerinin Zorluğu ..... - 46 -
27. BELLEK YÖNETİMİ (MEMORY MANAGEMENT) ..... - 47 -
27.1 Manuel Bellek Yönetimi (Manual Memory Management) ..... - 47 -
27.2 Otomatik Bellek Yönetimi (Automatic Memory Management) ..... - 48 -
27.3 Çöp Toplama Takibi (Tracing Garbage Collection) ..... - 49 -
27.4 Referans Sayma (Reference Counting) ..... - 50 -
27.5 Sahiplik (Ownership) ..... - 51 -
28. PROGRAMLAMA PARADİGMALARI (PROGRAMMING PARADIGMS) ..... - 53 -
28.1 Zorunlu Programlama (Imperative Programming) ..... - 53 -
28.2 Bildirimsel Programlama (Declarative Programming) ..... - 54 -
28.3 Fonksiyonel Programlama (Functional Programming) ..... - 55 -
28.4 Prosedürel Programlama (Procedural Programming) ..... - 55 -
28.5 Nesneye Yönelik Programlama (Object Oriented Programming - OOP) ..... - 56 -
28.6 Yapısal Programlama (Structured Programming) ..... - 56 -
28.7 Olaya Dayalı Programlama (Event-Driven Programming) ..... - 56 -
28.8 Veriye Dayalı Programlama (Data-Driven Programming) ..... - 57 -
29. TURING BÜTÜNLÜĞÜ (TURING COMPLETENESS) ..... - 58 -
30. TEKNİK BORÇ (TECHNICAL DEBT) ..... - 58 -

31. YERLEŞİK İŞLEVLER (BUILT-IN FUNCTIONS)..... - 59 -
32. VERİ TİPİ GRUPLANDIRMASI (DATA TYPE GROUPS)..... - 60 -
32.1 İlkel Veri Tipleri (Primitive Data Types)..... - 60 -
32.2 İlkel Olmayan Veri Tipleri (Non-Primitive Data Types)..... - 60 -
33. STANDART KİTAPLIK (STANDARD LIBRARY)..... - 61 -
34. DÖNGÜSEL KARMAŞIKLIK (CYCLOMATIC COMPLEXITY)..... - 61 -
34.1 Döngüsel Karmaşıklığın Hesaplanması..... - 62 -
34.2 Risk Değerlendirmesi..... - 64 -
35. BİRLİKTE ÇALIŞABİLİRLİK (INTEROPERABILITY)..... - 64 -

INFORMATION

Although this resource was designed as a thesis, it was not written as a thesis. Therefore, the source should not be approached as a thesis.

BİLGİLENDİRME

Bu kaynak bir tez gibi tasarlanmış olsa da tez olduğu kabul edilerek yazılmamıştır. Bu nedenle kaynağa tez olarak yaklaşılmaması gerekir.

ABSTRACT

PROGRAMMING LANGUAGE THEORY

Mertcan Davulcu

September 2022, 63 pages

This resource has been written to create a resource for programming language theory in Turkish literature. It targets the branch of computer science programming language theory. The reader is assumed to have a command of basic approaches, concepts, and at least a basic knowledge of discrete mathematics in programming. Not all programming and hardware concepts will be covered from scratch. Its suitability for people with a basic knowledge of the subject is debatable.

The resource generally refers to the working principles, methods and design approaches of programming languages. It includes comparing programming languages and working methods and examining their design. The content may include proven objective judgments and subjective judgments.

Abstract syntax tree creation, lexical analysis and interpretation stages are discussed in detail and a sample implementation has been developed for the exemplified algorithms in this context. The sample implementation is written int the Python programming language.

Keywords: programming language theory, compiler, transpiler, interpreter, bytecode, p-code, p-code machine, machine code, computer, programing language, scripting language, abstract syntax tree, lexer, intermediate language, binary system, discrete mathematics, memory, paradigm.

ÖZ

PROGRAMLAMA DİLİ TEORİSİ

Mertcan Davulcu

Eylül 2022, 63 sayfa

Bu kaynak programlama dili teorisi için Türkçe literatürde kaynak oluşturmak için yazılmıştır. Bilgisayar biliminin programlama dili teorisini dalını hedef alır. Okuyucunun programlama konusunda temel yaklaşımlara, konseptlere ve en azından temel düzeyde ayrık matematik bilgisine hâkim olduğunuzu kabul edilir. Tüm programlama ve donanım kavramlarına sıfırdan değinilmeyecektir. Konu hakkında başlangıç seviyesi bilgiye sahip kişiler için uygunluğu tartışmalıdır.

Kaynak genel olarak programlama dillerin çalışma prensiplerine, yöntemlerine ve tasarım yaklaşımlarına değinmektedir. Programlama dillerinin ve çalışma yöntemlerinin karşılaştırılması ve tasarımlarının incelenmesini içerir. İçerik kanıtlı nesnel yargılara ve öznel yargılara yer verebilir.

Soyut sözdizimi ağacı oluşturma, sözcüksel analiz ve yorumlama aşamalarına detaylı olarak değinilmiş ve bu kapsamda örneklenen algoritmalar için örnek bir uygulama geliştirilmiştir. Örnek uygulama Python programlama dili kullanılarak yazılmıştır.

Anahtar kelimeler: programlama dili teorisi, derleyici, çevirici, yorumlayıcı, bayt kodu, p-kod, p-kod makinesi, makine kodu, bilgisayar, programlama dili, betik dili, soyut sözdizimi ağacı, sözcüksel analiz, ara dil, ikili sistem, ayrık matematik, bellek, paradigma.

1. PROGRAMLAMA DİLİ TEORİSİ NEDİR

Programlama dili teorisi (Programming Language Theory - PLT) programlama dili olarak adlandırılan biçimsel dillerin geliştirilmesi, karakterize edilmesi ve sınıflandırılması, tasarlanması ve analizi ile ilgilenen bir bilgisayar bilimi dalıdır.

2. PROGRAMLAMA DİLİ

Programlama bir bilgisayar donanımına nasıl davranması gerektiğinin programlanmasına verilen isimdir. Çeşitli sayılar, aritmetik işlemler, bitsel ve benzeri işlemler kullanılarak gerçekleştirilen bir eylemdir.

Programlama kodlama değildir. Bilgisayar biliminde kodlamak eyleminin özel bir anlamı yoktur. Yani kodlama aslında programlama anlamına gelmez. Eğer bir anlam yüklemek gerekiyorsa da kodlamanın kelime anlamına bakıldığında zannımca bu programlama olmamalıdır. İşlemci komut seti (instruction set) içerisindeki komutları kullanarak her bir görevi adım adım gösterme işlemini kodlama olarak adlandırmak daha doğru olacaktır.

Programlama dili biçimsel bir dilidir. İnsanların programlama yapabilmesi için geliştirilmiş olan araçlardır. Programlamanın gerektirdiği eforun azaltılması, akılda kalması ve öğrenmesi kolay, daha geniş kitlelerce anlaşılabilmesi amaçlarıyla biçimsel bir dil olarak ortaya çıkmıştır.

Ortaya çıktığı ilk andan itibaren konuşma dilleri gibi etkileşim sonucu birbirlerinden etkilenerek çeşitli sorunları çözmek, farklı yaklaşımlara uygun yazılım geliştirme yapabilmeyi mümkün kılmak, daha modern olmak gibi amaçlarla değişime uğramış ve gelişmiş dillerdir.

Programlama dilleri doğrudan bilgisayarlar tarafından anlaşılabilen diller değildir. Daha insana yönelik olan dillerdir. Bu nedenle programlama dillerinin bilgisayarın anlayabileceği tek dil olan makine diline belirli aşamalardan geçerek uygun bir şekilde tercüme edilmesi gerekmektedir.

3. BİLGİSAYAR NEDİR VE NASIL ÇALIŞIR

Teknik olarak açıklamak gerekirse, bilgisayar önceden tanımlanmış bir dizi karmaşık aritmetik ve mantıksal işlemlerle birçok işi çözebilen ve çok kısa süre içerisinde gerçekleştirebilen herhangi bir cihazdır. Teknik açıklamadan da anlaşılabileceği üzere, bilgisayar işlemciye sahip olan herhangi bir cihaza verilen isimdir.

Bir bilgisayar deterministiktir. Determinizm yani belirlenimcilik ya da gerekircilik, her bir olayın gerekli ve kaçınılmaz olduğunu, hepsinin bir nedeni olduğunu, bunların belirlendiğini ve değişemeyeceğini söyleyen bir öğretidir. Zannımca zaman zaman "Delilik aynı şeyleri yapıp farklı sonuçlar beklemektir" (Anonim) cümlesinde determinizm öğretisine atıfta bulunulduğunu düşünmüşümdür. Cümleden de anlaşılabileceği üzere determinizm yalın tanımıyla rastgelelik olmadığını söylemektedir.

Bir bilgisayarın deterministik olmasının nedeni yapabileceklerinin önceden belirlenmiş olması ve bunun dışına çıkamayacak olmasıdır. Bir işlemci mimarisi (Ör. ARM, I386, AMD64) kendi içerisinde komut seti barındırır. Bu komut seti işlemcinin yapabileceği talimatların bir izlencesidir. İşlemci bu komutlar dışında farklı bir komut gerçekleştiremez. Bu komut seti içerisinde yer alan her bir komut tek bir sonuç verir ve bu sonuç hiçbir zaman değişmez. Yani bir bilgisayar işlemcisine aynı komutu verip farklı sonuçlar beklemek deliliktir. Özellikle programlamaya yeni başlayanlarda görülen "ben X yaptım ama bu Y yapıyor" sorunsalı yaygındır. Eğer kodunuz makine koduna doğru bir şekilde çevriliyorsa, sorun kesinlikle algoritmadadır zira işlemcinin kendi isteğine göre hareket etme olasılığı yoktur.

İşlemcinin içerisinde transistörler bulunur. Bu transistörlerin bizi ilgilendiren yeteneği anahtarlama olarak kullanılabiliyor olmalarıdır. Bu anahtarlama yeteneği sayesinde bazı mantıksal ve aritmetik işlemleri elektrik akımını kullanarak temsil edebilmek mümkün hale gelir. Bunu alfabeye benzetmek mümkün. Bir dilin kullandığı alfabede tek başına anlam ifade pek bir etmeyen harfler bulunur, bu harfler belirli sıralarda dizilir ve anlam ifade eden bir kelime kodlanmış olur. Bilgisayarda da akımın yaptığı şey tam olarak budur. Belirli transistörlerin anahtarlama yeteneği kullanılarak 1 ve 0 durumlarıyla kodlama yapılması sonucu komut seti içerisinde yer alan bir izlence temsil edilmiş olur.

Bir transistörün akım alması yani anahtarın açık olması 1, almaması yani anahtarın kapalı olması 0 olarak temsil edilir. Bu 0 ve 1 gösterimi konu dahilinde negatiflik ve pozitiflik temsil eder lakin genel anlamda elma ve armut, var ve yok, evet ve hayır, reçel ve pekmez gibi iki farklı durumu temsil etmek için kullanılabilir.

Bahsi geçen 0 ve 1 aynı zamanda matematikte ikili sayı (binary) sisteminin sahip olduğu rakamlardır. Bu nedenle bilgisayarın ana dilinin ikili sistem olduğunu söylemek mümkündür. Anlatıldığı üzere transistörlerin anahtar durumunun temsil edilmesi ya da bir verinin temsil edilmesi gibi ikili sistemde pek çok şey yapılabilir.

Bu kodlamanın tam olarak nasıl olduğuna dair örneklendirme yapmak gerekirse, yapılan kelime kodlama benzetmesinin doğrudan bir karşılaştırmasını vermek doğru bir örneklemedir:

Kodlanan örnek kelime: "Armut"

Alfabe ile kodlama: Armut

İkili sistem ile kodlama: 01000001 01110010 01101101 01110101 01110100

Buradaki ikili sistem metne çevrildiğinde "Armut" kelimesini verecektir. Bu ikili sistem ile veri temsiline bir örnektir. İşlemci komut seti içerisindeki bir izlencenin temsili de bundan çok farklı değildir.

4. ALGORİTMA

Algoritmalar bir amacı gerçekleştirmek için uygulanan adımlar bütününe verilen addır. Bir başlangıcı ve sonu olan sonlu işlemler kümeleridir. Genellikle bilgisayar programlamada kullanılır ve programlamada kaçınılmaz olarak yapılan bir eylemdir.

5. KOD TÜRLERİ

5.1 Kaynak Kod (Source Code)

Kaynak kod bir programlama dili kullanılarak yazılmış olan kodlar bütünüdür. Genellikle düz metin olarak yazılırlar. Kaynak kodların neredeyse hepsi aslında birer metin dosyalarıdır yani bir tek başına bir anlamları yoktur. Programlama yapmak için geliştiricilerin kullandığı daha anlaşılabilir bir format olarak ortaya çıkmıştır. Makine kodu ile yazılmış kodun okunması daha zor olduğundan, insan tarafından okunabilen bir programlama dili ile yazılmış metin dosyaları olarak öne çıkarlar.

5.2 Ara Gösterim (Intermeatiate Representation - IR)

Ara gösterim, bir kaynak kodunu temsil etmek için kullanılan ara bir dil ya da gösterimdir. Dahili bir dildir, bir derleyici ya da sanal makine tarafından kullanılır. Genellikle düşük seviye gösterime sahip bir dildir. Bir sanal makineyi ya da P-Code makinesini hedef alan her tür derleme bir ara dil olarak nitelendirilebilir.

5.3 Kayıt Transfer Dili (Register Transfer Language – RTL)

Kayıt transfer dili donanım bağımsız ancak makine diline çok yakın olan bir ara dil temsili olarak ortaya çıkmaktadır. Montaj diline çok yakındır. Bir mimarinin kayıt-aktarım seviyesindeki veri akışını tanımlamak için kullanılır.

5.4 Obje Kodu (Object Code)

Obje kodu bir derleyici tarafından üretilen çıktıya verilen isimdir. Bu çıktı genellikle makine kodu ya da bir ara dil şeklinde olur.

5.5 Bayt Kodu (Bytecode)

Bytecode ya da P-Code (Portable Code) tam olarak sanal bir işlemcinin sanal bir komut setini temsil eder. Bu komut seti platform bağımsız bir komut setidir. Nihayetinde sanal bir işlemciye aittir. Bu nedenle taşınabilir kod olarak tanımlanabilir. Hem bir yorumlayıcı tarafından çalıştırılabilir hem de derleyici tarafından derlenebilir bir formdadır. İnsanın okuyabilmesi bir amaç olmadığından, yalnızca bir yazılım tarafından işleneceğinden oldukça karmaşık, okunaksız bir şekilde üretilmiş olabilir. Verimli bir şekilde yürütülmek üzere tasarlanmıştır. Bayt kodları işletim sistemi ve donanım bağımlılığını azaltmak için kullanılabilir.

5.6 Makine Kodu (Machine Code)

Makine kodu tam olarak bilgisayarın komutlarının nasıl çalıştığıdır. Makine kodu işlemci komut seti kullanılarak yazılmış olan en alt seviye temsili olarak kabul edilebilir. Doğrudan komut setinden belirli izlenceleri gerçekleştiren ve bir amacı yerine getiren koddur. Doğrudan makine kodu yazmak mümkündür lakin bu noktada programlama dillerinin ortaya çıkış nedenleri hatırlanmalıdır. Makine kodu yazmak zahmetli, hataya açık ve maliyetli bir eylemdir. Elbette makine kodu halen belirli zamanlarda belirli nedenlerle kullanılabilir, kimsenin artık makine kodu yazmadığını söylemek doğru değildir.

Bir makine kodu donanıma bağımlıdır. Bunun nedenini anlamak için makine kodunun komut seti kullanarak yazılmış olan en alt temsil olarak kabul edilebileceği ve işlemcilerin komut setinin işlemcinin mimarisine göre değişiklik gösterdiği bilgisinin hatırlanması ilişkiyi kurmak için yeterlidir.

Bir makine kodu belirli bir komut setine göre yazıldığında, farklı bir mimarinin komut seti ile uyumsuz (yüksek olasılıkla) olabilir ya da farklı izlencelerin temsil edilmesi sonucu yanlış algoritmalar oluşabilir. Bu nedenle makine kodunun yazılımın çalışmasının istendiği her bir mimari için ayrı olarak yazılması gerekmektedir. Hedef mimariler arasında bir görevin izlencedeki temsili farklı bir kodlama olabilir ya da aynı komut doğrudan mevcut olmadığından komutun görevinin var olan talimatlarla yazılması gerekebilir. Tam da bu nedenden makine kodu donanıma bağımlı olan bir dildir.

İşlemci komut setlerinin içerisinde temelde toplama, çıkarma, çarpma, bölme ve bazı bit seviyesi işlemler bulunsa da günümüzde çoğu modern işlemci çok daha fazlasını yapabilmektedir. Yine de temelinde yatanlar bundan farklı değildir.

6. ÇEVİRİCİ (TRANSPILER)

Çevirici ya da aktarıcı bir kaynak kodun farklı bir programlama dilinde yazılmış olan kaynak kod formuna çevrilmesi için geliştirilmiş olan bir yazılımdır. X programlama dilinde yazılmış olan bir kaynak kod bu yazılımlar yardımıyla Y programlama diline çevrilebilir.

Çevirmenlikten çok farklı değildir. Bir kitabın X dilinden Y diline çevirisini yapmaktan farkı yoktur. Burada bahsi geçen kitap kaynak koddur.

7. DERLEYİCİ (COMPILER)

Derleyiciler kaynak kodları makine koduna derleme görevine hizmet eden yazılımlardır. Bir programlama dilinin derleyicisi, o programlama dili kullanılarak yazılmış olan kodu hedef işlemci mimarisine göre uygun şekilde makine koduna derler ve genellikle çıktı olarak yürütülebilir (executable) dosyasının elde edilmesini sağlar. Bu eyleme derleme denir.

Bilgi işlemde yürütülebilir dosya doğrudan işlemci tarafından çalıştırılabilecek bir dosyayı ifade eder. Dosya makine kodu ile yazılmış komut seti talimatları bütününden oluşur.

Genellikle yürütülebilir dosya elde edilmesi şeklinde tanımlanmasının nedeni ise bir derleyicinin her zaman tam anlamıyla yürütülebilir dosya oluşturmamasından kaynaklanmaktadır. Burada derleyicilerin çalışma prensipleri ortaya çıkmaktadır lakin önce bazı terimlere daha değinilmesi gerekmektedir. Bu konu ilerleyen konularda ele alınacaktır.

8. YORUMLAYICI (INTERPRETER)

Yorumlayıcılar nihayetinde makine kodu tarafından yürütülen bir yazılımdır. Bir programlama dilinin yorumlayıcısı, kaynak kodu derlemek yerine anlık olarak adım adım ifadeleri yorumlar ve ne yapması gerektiğini anlayarak ilgili eylemleri gerçekleştirir. Bir yorumlayıcı doğrudan kaynak kodu yorumlayabilir ya da bytecode ve benzeri oluşturup onu da yorumlayabilir. Yani aslında içerisinde bytecode çeviricisi ve yorumlayıcısı barındıran yorumlayıcılar bulunabilir. Buna Python programlama dili örnek gösterilebilir.

Bir dilin yorumlayıcısı kullanılarak, yeni bir dil ya da mevcut olan bir dil için yeni bir yorumlayıcı geliştirilebilir. Yani bir yorumlayıcı ile farklı bir yorumlayıcı geliştirip çalıştırmak mümkündür. Ancak her zaman eninde sonunda çalışan şey, ilk çalışan yorumlayıcının makine kodlarıdır.

Örneğin X dili yorumlanan bir dildir. Siz Y dilini geliştirdiniz ve geliştirmiş olduğunuz Y dili de bir yorumlayıcı tarafından yürütülmektedir. Bu yorumlayıcıyı X dili ile geliştirdiniz. Bu durumda Y diliniz için geliştirmiş olduğunuz yorumlayıcıyı çalıştırmak için, öncelikle onu geliştirmiş olduğunuz X dilinin yorumlayıcısını X dili ile yazılmış kaynak kodunuzu yorumlaması için çalıştırmalısınız. X kaynak kodunu yorumlayan yorumlayıcı, kaynak kodunuzdan yola çıkarak Y dili için bir yorumlayıcı olarak çalışacaktır.

Geliştirdiğiniz Y dili yorumlayıcınız verilen Y dili kaynak kodlarını yorumlayarak talimatları X dili içerisinde gerçekleştirmeye çalıştığında, X dili yorumlayıcısı ilgili adımları kendi yorumlayıcısında gerçekleştirecektir. Bu durumda makine kodu yürütülmüş olacaktır zira X dilinin yorumlayıcısı makine kodu formundadır. Bu zincir ne kadar uzarsa uzasın, ulaşacağı en temel nokta her zaman makine kodudur. Bundan kaçış yoktur.

9. P-KOD MAKİNESİ (P-CODE MACHINE)

P-Kod (Portable Code) makinesi taşınabilir bir kodu yorumlamak için tasarlanmış olan bir sanal makinedir. Amacı ilgili kodları üzerinde çalıştığı mimariye uygun bir şekilde yorumlamak ve yürütmektir. Yani P-Code'u çalıştığı mimariye uygun bir şekilde makine koduna çeviren bir tür yorumlayıcıdır.

10. ÇALIŞMA ZAMANI (RUNTIME)

Çalışma zamanı, bir programın çalıştığı süredeki anı temsil eder.

11. DERLEME ZAMANI (COMPILE TIME)

Derleme zamanı, bir programın derlenme sürecini temsil eder.

12. TİP GÜENLİĞİ (TYPE SAFETY)

Tip güvenliği bir programlama dilinin tip güvenliğini ifade etmektedir. Tip, veri tipi anlamına da gelir. Her bir veri yapısı aynı zamanda bir veri tipidir denebilir. Modern programlama dilleri yaygın olarak tip güvenliğini benimser. Birbiri ile uyumlu olan ve uyumlu olmayan tipler bulunur. Bu tiplerin uyumsuzluk durumlarında yapılan geri bildirimler sonucu tip güvenliği sağlanmış olur. Uyumlu tiplerin birbiri ile kullanılabileceği kabul edilir.

Tip güvenliği yazılım geliştirmede hata ayıklama konusunda genellikle en önemli özellik olarak öne çıkar. En yaygın iki türden biri olan string yani metin ve integer yani tamsayı tiplerinden örnek vermek gerekirse, bir tamsayı ve metin tipi uyumlu değildir zira metin her zaman tamsayı ve tamsayı da hiçbir zaman bir metin değildir. Bu tiplerin uyumsuz bir şekilde kullanılması, mantıksal olarak hataya neden olmalıdır.

Tip güvenliği konusunda iki tür yaklaşım statik ve dinamik olarak ortaya çıkmaktadır. Açıklamalara geçmeden önce bu ifadelerin net bir teknik tanımının olmadığını belirtmekte fayda var. Bu konuda halen bazı anlaşmazlıklar mevcuttur.

12.1 Statip Tip Denetimi (Static Type Checking)

Statik tip denetimi, tip güvenliği için her bir tanımın bir tipi olduğu sistemdir. Bu tip statiktir, yani hiçbir zaman değişemez. Genellikle derlenen programlama dillerinde kullanılırlar zira derlenmiş olan kodun çalışma zamanında tip güvenliği gerçekleştirmesi bir maliyettir. Bu nedenle genel olarak bu tip güvenliği, statik olduğundan açıkça derleme esnasında kontrol edilir ve uyumsuzluklar bildirilir.

Luca Cardelli'nin Typeful Programming makalesinde statik tip denetimi için "denetlenmeyen bir çalışma zamanı tür hatası olasılığının olmadığı bir sistem" olarak bahsedilmektedir.

12.2. Dinamik Tip Denetimi (Dynamic Type Checking)

Dinamik tip bir tanımın değişken veri tipine sahip olabilmesi anlamına gelir. Statik tipin aksine çalışma zamanında tip sorunu yaşanana kadar bir sorun yoktur. Yani bir tanımın depoladığı verinin türü, o veriye göre dinamik olarak değişebilir. Bu değişimlere rağmen ilgili veri ile uyumsuz bir davranış sergilenmeye çalışılmadığı sürece herhangi bir hata meydana gelmez. Genellikle yorumlanan programlama dillerinde ortaya çıkmaktadır.

12.3 Jenerik Tipler (Generic Types)

Jenerik tipler statik tip denetimli programlama dillerinde ortaya çıkan bir tasarım yaklaşımıdır zira dinamik tiplerde buna gerek yoktur. Bu tasarımda veri tipi gerektiği anda somutlaştırılır. Kod içerisinde tekrarlamayı azaltmakta etkili bir yaklaşımdır.

Fonksiyonları ve jenerik tipleri destekleyen bir programlama dili kullandığınızı kabul edersek. Yalnızca argüman olarak alınan bir sayısal değeri π sayısı ile bölen ve bunu komut satırına yazan bir fonksiyonunuz olduğunu düşünelim. Bir geliştirici olarak bu fonksiyonu tüm sayısal veri tipleri ile çağırmak istiyorsunuz ancak kullandığınız programlama dili statik tipli ve parametreniz tek bir tipe sahip olabilir. Bu durumda bunu yapmanız mümkün değildir. Jenerik tipler burada fark yaratırlar.

Gerektiği anlarda somutlaştırılır bilgisini hatırlayın. Fonksiyonu kullandığınızda jenerik için bir veri türü vermiş olursunuz, bu veri türü o an somutlaştırılır ve statik analiz gerçekleştirilir. Bir sorun yoksa, sorun yoktur. İlgili veri türüne uygun olmayan bir işlem varsa tip denetimi güvenliği için bu konuda muhtemelen sıkı bir derleyici hatası alacaksınız.

12.4 Ördek Yazma (Duck Typing)

Duck Typing kabaca bir nesnenin kullanıldığı şekilde kullanılmaya uygun olup olmadığını belirlemek için ördek testinden faydalanılması işlemidir. Ördek testi ise "Ördek gibi yürüyor ve ördek gibi vaklıyorsa, o zaman bir ördek olmalı" cümlesine dayanır.

Örneğin metotlara sahip olan bir programlama dilinde Y değişkeninde X metodunu çağırıyorsunuz. Eğer X metodu Y değişkeninde var ise, sorun yoktur, yok ise sorun vardır. Bu yaklaşımdan dolayı aslında pek çok kişi bunun dinamik tip denetimi kullanan programlama dillerinde yapılabilecek bir şey olduğunu düşünse de Duck Typing statik tipli programlama dillerinde de yapılabilir. Jenerikleri uygun şekilde destekleyen statik tip denetimine sahip bir programlama dili rahatlıkla geliştiricinin Duck Typing yapmasına izin verebilir.

Kod örneği (Python):

```
class A:
      def lock(self):
            print("A is locked")

class B:
      def lock(self):
            print("B is locked")

def lock(obj):
      obj.lock()

lock(A())
lock(B())
```

13. DERLENEN VE YORUMLANAN DİLLERİN KARŞILAŞTIRILMASI

Kaynak kodun tek başına bilgisayar için bir anlam ifade etmediği bilgisini hatırlayın. Peki bir kaynak kod nasıl bilgisayar için anlamlı bir hale gelir? Bir yorumlayıcı ya da derleyici ile. Bir kaynak kodun yorumlayıcı ve derleyici arasında anlam kazanırken ciddi farklar meydana gelir. Anlam kazanma yönteminden etkilenen baş faktörler performans ve verimliliktir. Bu nedenle seçilen teknolojinin amaçlara uygun olduğuna emin olunmalıdır.

Bir dilin yorumlanan ya da derlenen bir dil olduğunu söylemek için orijinal olarak yorumlanan ya da derlenen bir dil olarak tasarlanmış olması gerekir. Bir dilin bir yorumlayıcı ile çalıştığını söyleyebilmek için ise o dilin o an yorumlayıcı ile çalıştırılması yeterlidir. Bu ayrımın yapılmalıdır zira derlenmiş bir programlama dili için daha sonradan üçüncü parti ya

da resmi olarak ana tasarımdan ayrı bir yorumlayıcı da geliştirilmiş olabilir. Bu sayede aslında derlenmiş bir dil olarak tasarlanmış olsa da yorumlayıcı ile yürütmek mümkün hale gelebilir.

Tam tersi olarak bir yorumlanan dil için daha sonradan bir derleyici de geliştirilmiş olabilir ve bu sayede dili derlemek mümkün hale gelebilir. Ancak bunlar o dilin ana tasarımını değiştirmez, bu nedenle dilin orijinal tasarımına bakmak gerekir. Eğer bir dil ana tasarımı içerisinde birden fazla yürütme tekniği barındırıyorsa ilgili tekniklerin hepsine sahip olduğu söylenebilir. Dart programlama dili buna iyi bir örnektir.

Bu yöntemlerin birbirinden farkı nedir ve ne gibi artı eksi yönleri vardır? Kelimenin tam anlamıyla derlenen bir dil ile yapılabilecek her şeyi yapmak mümkündür, bir kısıtlaması yoktur. Lakin burada kaçırılmaması gereken konu derlenen her programlama dilinin her şeyi yapabileceğinden bahsedilmiyor oluşudur. Bir programlama dilinin yetenekleri, geliştiricileri tarafından belirlenir. Eğer ki dilde X yeteneği yok ise ve bu yeteneği taklit etmenizi sağlayacak yetenekleri de yok ise, bunu yapamazsınız. Her şeyi yapmak mümkündür ifadesi aslında daha çok geliştiricilere yönelik bir ifadedir zira eğer bir programlama dili geliştiricisi derlenmiş bir dil geliştiriyor ise, eninde sonunda makine kodu seviyesine inmesi gerektiğinden makine kodu seviyesinde mümkün olan istediği her şeyi gerçekleştirme imkanına sahiptir. Ancak o programlama dilini kullanan geliştiriciler, o programlama dilinin yapabilecekleri ile sınırlıdır.

Yorumlanan bir dilde her şeyi yapmak mümkün değildir. Aynı açıdan bakıldığında, bir programlama dili geliştiricisi için bu doğru bir söylemdir zira yorumlanan bir dilde makine kodu seviyesine inmesi o dilin yeteneğine kalmış bir konudur. Ki çoğu yorumlanan dil buna izin vermez zira genel olarak yorumlanan dillerin tasarım amacına uygun değildir. Bu nedenle bir programlama dili geliştiricisi için, yorumlanan bir dil kullandığında her şeyi yapabileceğinin mümkün olmadığı söylenebilir.

Yorumlanan diller genel olarak sistem programlama amacına hizmet etmezken, derlenen diller genel olarak bunu mümkün kılan tasarımlara sahiptir. Elbette her iki türde de aynı yetenekler sunulabilir. Zira yorumlayıcının da eninde sonunda makine kodu seviyesinde olduğunu unutmamak gerekir. Bir şekilde aynı yeteneklere sahip yorumlanan ve derlenen iki farklı dil geliştirilmesi mümkündür. Ancak burada bahsi geçen yetenekler dilin yetenekleridir, tasarımının değil. Bu ne demek? Örneğin dilin ne yapabileceği bu yetenekler dahilindedir lakin nasıl anlam kazandığı bunun dışındadır. Bu karşılaştırmada aksi halde hiçbir anlam

olmayacaktır zira ikisinin birebir aynı yeteneklerde olmayan diller olarak kabul edilmesi gerekir.

Yukarda bahsedildiği gibi, iki türde de birebir aynı yeteneklere sahip yorumlanan ve derlenen iki farklı programlama dili olduğu varsayımında dahi aralarında yine de farklar olur muydu? Evet. Bu yöntem farklılığından en çok etkilenen iki faktörün verimlilik ve performans olduğu bilgisini tekrarlayalım. Peki bu fark neden var? Derlenmiş bir programlama dilinin daha performanslı olmasının nedeni zaten pek çok işlemin ortadan kalkmış olmasıdır. Örneğin derlenen pek çok dil statik tipli olduğundan ve statik tip güvenliği denetimi sağladığından çalışma zamanında bu tip güvenliği için bir maliyete sahip değilken, yorumlanan diller genel olarak dinamik tip denetimine sahip olduğundan çalışma zamanında her seferinde tip güvenliği denetimi gerçekleştirmek zorunda kalırlar. Bu da çalışma zamanında bir maliyet olarak yansır ve performansın derlenen dillerden daha yavaş olmasındaki nedenlerden biri olur.

Öte yandan bellek verimliliği ise derlenmiş olan programlama dillerinde daha yüksektir. Örneğin bir değişken için derlenmiş olan programlama dilleri bellekten yalnızca gerekli olan alanı ayırır ve bellek konumunun adresini işler. Derlenmiş olan programlama dillerinde bir değişken tanımlayıcısı bellek adresi için bir nimoniktir. Yorumlanan programlama dillerinde ise bir değişkenin verisinin yanı sıra tanımlayıcısı da çalışma zamanında bellekte tutulmalıdır, dolayısıyla değişkenin tanımlayıcısı bellekte bir ek yüktür. Bu ve bunun gibi gerekli olan ek veriler çalışma zamanında yorumlanan dillerde daha fazla bellek ayak izi oluşmasına neden olur.

Bir yorumlanan dil ile yapılması mantıklı olmayan şeyler vardır. Bunlar işletim sistemi geliştirmek, programlama dili geliştirmek ya da gömülü sistem programlama gibi amaçlardır. Bunlar ve benzeri amaçlarda performans ve verimlilik önem arz eder, yorumlanan dillerin ise neden bu iki kriterde olumsuz etkiye sahip olduğu kabaca açıklandı. Ek olarak, bu tür bir amacınız varsa gerçekleştirebilmek için muhtemelen sistem programlama yapmanız gerekecektir ve yorumlanan dillerin tasarım amaçları gereği genellikle sistem programlama yetilerinin olmadığına değinilmişti.

Ancak yorumlanan diller direkt olarak yorumlanarak çalıştırıldığından, belirli amaçlara derlenen dillerden çok daha iyi şekilde hizmet ederler. Örneğin scripting dili olarak ya da bazı performans kritik olmayan amaçlar için kullanıldıklarında. Örneğin Python ve R programlama dilleri ile veri bilimi alanında çalışmak iyi bir tercihken, birebir aynı kütüphaneler olsa bile C

programlama dilini veri bilimi için kullanmak rasyonel bir tercih değildir. Yorumlanan bir dili bu tip amaçlarda doğrudan yürütebilmek mümkünken derlenen bir dil için derlenme adımının tamamlanması gerekir. Bu en azından bir adımlık fark anlamına gelir ve bazı durumlarda epey uzun bir zaman farkı oluşmasına neden olabilir.

14. KAYNAK KODUN MAKİNE KODUNA DÖNÜŞÜMÜ

Kaynak kodun bir derleyici veya yorumlayıcı sayesinde anlam kazandığından bahsedildi. Peki bu anlam kazanma aşamaları neler, bir programlama dili kullanarak yazdığınız kaynak kodu nasıl makine koduna evriliyor? Elbette bunu yapmanın pek çok paradigması var lakin bu kaynakta genel olarak örnek uygulamada (implementation) ve yaygın olarak tercih edilen genel paradigmalardan bahsedilecektir. Tüm bunların nihai hali ve genel olarak nasıl bir yaklaşım benimsendiğine değinilecek.

15. SÖZDİZİMİ (SYNTAX)

Sözdizimi bir dildeki doğru kullanım ve kombinasyonları, doğru yapılandırılmış ifadeleri temsil eden kurallar bütünüdür.

15.1 Yürütülebilir İfade (Statement)

Statement bir görevi, işlevi gerçekleştiren ifadeye verilen isimdir. Örneğin: atama (assignment), dönüş ifadeleri (return statement).

15.2 Değerlendirilebilir İfade (Expression)

Expression değerlendirilebilir herhangi bir ifadeye verilen isimdir. Örneğin: iki sayısının toplamı, sayısal değer taşıyan bir değişkenin x'e bölünmesi.

16. DENEYSEL DİL TASARIMI

Bu kaynakta basit bir programlama dili yapılışı anlatılacağından, bu anlatımın gerçekleştirilebilmesi için kaynağa özgü deneysel bir programlama dili tasarımı oluşturulmalıdır.

Bahsi geçen deneysel dilin pek bir yeteneğinin olması gerekmemektedir. Yalnızca aşamaları anlamlandırabilmek için yeterli örneklendirme sunması yeterlidir. Bu koşullarda yorumlanan

ve dinamik tip denetimine sahip olan bir programlama dili tasarımı iyi bir tercihtir. Zira uygulanması derlenen bir programlama diline göre daha basit ve yalın olacaktır. Dil Unicode desteği barındırmamakta, yalnızca ASCII standardı ile uyumludur.

Bu deneysel dilden bahsetmek için adının X olduğunu kabul edelim.

X dilinin tasarımı bu şekildedir:

Operatörler (Operators)	
`+`	Toplama / Pozitif / Birleştirme
`-`	Çıkarma / Negatif
`*`	Çarpma
`/`	Bölme
`^`	Üs Alma
`%`	Modulo (Kalan)
`=`	Atama (Assignment)

Anahtar kelimeler (Keywords)	
`print`	Çözüm Adımları İle Komut Satırına Yaz

Operatör Önceliği (Operator Precedence)	
`^ %`	1
`/ *`	2
`+ -`	3

Dilin amacı matematiksel hesaplama yapmak ve sonuçları hızlı bir şekilde elde etmektir. Yalnızca değişkenler ve matematiksel ifadeler desteklenmekte, yalnızca tamsayı ve ondalıklı sayı veri tiplerini barındırmaktadır. Boşluklara duyarlı değildir ve sözdizimi gereği satır satır yazılması gereklidir. Dilin sahip olduğu parantezler yalnızca "`(`" ve "`)`" parantezleridir.

Matematiksel bir dil olduğundan işlem önceliğinde parantezlerin önce gelmesi gerekmektedir. Bu nedenle parantezler içten dışa doğru değerlendirilmeli ve daha sonra operatör önceliği gözetilmelidir.

X dilinin yorumlayıcısının çalışma paradigması sırasıyla kaynak kodu okuma (sözde), sözcüksel analiz, ayrıştırma (AST oluşturma) ve yorumlanma aşamalarından oluşmaktadır.

Yarıçapı 5,5 olan bir dairenin alan hesaplaması üzerinde örnek bir temsil:

```
PI = 3.14
```

```
r = 5.5
area = PI * r * r
print area
area
```

X dilinin tasarımına göre tanımlayıcılar sayı ile başlayamaz. Yalnızca alt çizgi ve alfabetik karakter ile başlayabilirler. İlk karakterden sonrası ondalık sayı sistemine ait rakamları barındırabilir. İlgili tanımlayıcıda var olmayan değişkenler atama işlemlerinde otomatik olarak oluşturulurlar. Her bir anahtar kelime ayrılmış (reserved) bir tanımlayıcıdır, yani tanımlayıcı olarak kullanılamazlar. Tekli (unary) operatörler desteklenmez. Pozitif ve negatiflik belirtimi için aritmetik değerlerde + ve − operatörleri geçerli kabul edilir, değişkenlerle kullanıldıklarında geçersiz sözdizimidir.

Bu deneysel dilin örnek bir uygulaması, daha kolay ve geniş kitlelerce anlaşılabilmesi için Python programlama dili ile geliştirilmiştir. GitHub deposunda kaynak kodlar halka açık bir şekilde erişilebilir durumdadır.

17. JETONLAR (TOKEN)

Bir jeton tanımı şuna benzemektedir:

```
Class Token:
      column: int
      row: int
      kind: str
      identity: int
```

Jetonların kimliklerinin temsili, sürdürülebilir ve okunabilir bir yazılım geliştirme adına sihirli sayıları (magic number) önlemek için her bir kimlik bir tanımlayıcı ile edilmelidir.

Örnek algoritma tasarımında anahtar kelimeler kendileri için doğrudan benzersiz bir kimliğe sahip olurken operatör ve parantezler grup kimliğine sahip olacak şekilde tasarlanmıştır. Tanımlayıcılar ise tanımlayıcı kimliğine sahiptir.

ÖRNEK KİMLİK GÖSTERİMLERİ	
IDENTIFIER	Tanımlayıcı
PARENTHESES	Parantez
EXPR	Değerlendirilebilir İfade (Expression)
OPERATOR	Operatör
PRINT	`print` Anahtar Kelimesi

18. SÖZCÜKSEL ANALİZ (LEXING)

Lexer, Türkçede direkt çevirisinin yapılabileceği doğru bir kelime olmayan terimlerden biri denebilir. Lexer aşamasından önce derleyicinin ilgili kaynak kodu metin olarak elde ettiği kabul edilmelidir. Bu metnin önce lexer tarafından işlenmesi en yaygın ve zannımca da tavsiye edilen yöntemdir.

Bu yöntem dışında daha farklı yaklaşımlar benimsemek de mümkündür. Örneğin metni satırlardan ayırıp bir liste haline getirmek ve çeşitli karakterlerden bölüp gruplandırmak ya da direkt olarak işlemek gibi lakin bunlar rasyonel yaklaşımlar değildir.

Lexer, kaynak kodu inceleyip çözümler ve jeton (Token) haline getirerek derleyici, çevirici ya da yorumlayıcı için anlamlı hale gelmesini sağlar. Bu işleme lexing denir.

Sözcüksel analizin nasıl gerçekleştirileceği elbette geliştiricinin algoritma tasarımına kalmıştır. Her bir jeton, sözcüksek analiz gerçekleştirilirken belirli bir anlam ifade etmesi için lexer tarafından oluşturulur. Bahsi geçen anlamı kazanabilmesi için her bir jetona belirli veriler atanmalıdır. En önemli veri jetonun türüdür. Bu bir tür kimliktir, her bir tür için benzersiz bir tamsayı ideal bir seçimdir. Bu tamsayı değerine göre jetonun bir anahtar kelime, operatör, tanımlayıcı ya da farklı bir şey olduğu karakterize edilir. İkinci veri ise türdür. Örneğin jeton bir operatör lakin hangi operatör, bunun için metin türünde bir tür verisi tutmak ideal bir seçimdir. Bunların yanı sıra geliştiricinin yazılım tasarımına göre satır ve sütün verileri, hangi kaynak kod dosyasından ayrıştırıldığı gibi ek bilgiler de depolanabilir. Karakterizasyon yaklaşımı olarak operatör, anahtar kelime gibi belirli bir grubu temsil etmek yerine direkt olarak her bir anahtar kelime, operatör ve diğer dil ögelerine benzersiz bir kimlik tamsayısı ataması da gerçekleştirilebilir.

Bu kaynağın örnek tasarımında benimsenen yaklaşım her ikisidir. Anahtar kelimeler için her birine özel bir kimlik atanırken, operatörler, parantezler ve diğer şeyler için gruplama tarzında bir karakterizasyon gerçekleştirerek türlerinin kontrolü metinsel veri ile sağlanır.

Lexer algoritması dilin tasarımına göre değişiklik göstermelidir. Örneğin Python gibi sözdizimsel olarak girintiye (indention) dayalı ise, her bir indention için jeton oluşturma ya da farklı bir yaklaşım benimsenebilir. Aksi halde bu tarz karakterler anlam ifade etmiyorsa, boşluk karakterlerinin atlanması gereklidir. Tanımlanamayan bir jeton hata sebebi olmalıdır.

Bu kaynağın dil tasarımı için benimsenmiş olan lexer tasarımına geçmek gerekirse:

Algoritmada Lexer tarafından işlenen kaynak kodun tutulduğu metin alanı, parantezlerin sayımında kullanılan tamsayı alanı, mevcut satır ve sütünü ifade tamsayı alanları, kaynak kod üstündeki konumu ifade eden bir ofset tamsayı alanı ve kayıtların tutulması için bir kütük listesi alanı olmalıdır. Kaynak kod metni lexing işlemi boyunca lexer tarafından kullanılacak ve analizi gerçekleştirilecektir. Örnek algoritmada bu kaynak kod hiçbir zaman değiştirilmez.

Kaynak kod üstündeki konumu ifade eden ofset tamsayı alanı ise algoritmanın çalışma paradigması gereği elde bulunması gereken bir veridir. Algoritmanın anlatımında anlam kazanacaktır.

Mevcut sütün ve satır konumunu ifade eden tamsayı alanları bir token oluşturulurken kaynak kod üstündeki token konumunu ayarlamak için kullanılacaktır.

Parantez sayacını ifade eden tamsayı alanı kapatılmamış olan ya da açılmamış olan bir parantezin kapatılmasını lexing esnasında saptamak ve diğer algoritmalarda açılan parantezlerin kapatılmış olduğunun garantisini sağlamakta kullanılacaktır.

Kütük listesi, yalnızca hatalar için kullanıldığından listenin boş olup olmama durumuna göre lexing işleminin hata saptayıp saptamadığını anlamak için kullanılabilir ve içerisindeki hata kayıtlarından yararlanılabilir.

Tipik bir lexer algoritmasında dışarıdan alınan tek veri sözcüksel analizinin gerçekleştirileceği koddur. Lexer örneğinin bu kaynak koda sahip olduğunu kabul edelim. Lexer algoritması uygulanmadan önce sözdizimsel analizde kullanılacak olan bilgilerin elde edilmesi için iki fonksiyon yazılmalıdır. Bu fonksiyonlar bir karakterin onluk sayı sistemine ait olup olmadığını söyleyen ve bir karakterin boşluk karakteri olup olmadığını söyleyen fonksiyonlardır.

Onluk sayı sisteminde 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 olmak üzere 10 adet rakam bulunur. Argüman olarak alınan karakter bu rakamlardan birini ifade ediyorsa, dönüş değeri olumlu olmalıdır, aksi halde olumsuz bir dönüş temsil edilmelidir. Bunun için boolean kullanılması ideal bir seçimdir.

Boşluk karakterizasyonu için ASCII tablosunda boşluk olarak kabul edilen karakterlerin kontrolü gerçekleştirilmelidir. Unicode desteği bir hedef olmadığından ASCII tablosu yeterlidir.

ASCII tablosundaki boşluk karakterler karakter koduna göre sırasıyla 32, 9, 11, 13 ve 10 karakterleridir. Bu karakter kodları sırasıyla boşluk, girinti (tab), dikey girinti (vertical tab), satır başı ve yeni satır karakterlerini temsil etmektedir. Bu karakterler sözdizimsel olarak bir anlam ifade etmediğinden boşluk olarak değerlendirilmelidir. Argüman alınan karakter bu karakterlerden biri ile eşleşiyorsa olumlu, eşlemiyorsa olumsuz dönüş temsil edilmelidir. Bunun için boolean kullanılması ideal seçimdir.

Ana lexing fonksiyonu, temel bir döngüsel algoritmaya dayanır ve her daim jetonları içeren bir liste döndürmelidir. Fonksiyon algoritmasının sonunda kapatılmamış parantezler için sayaç kontrol edilmelidir. Analiz işlemi sürecinde parantezlere göre sayaç artabilir ve eksilebilir. Analiz süreci sonra erdiğinde sayacın sıfır olmaması durumunda parantezlerden en az bir tanesinin kapatılmamış olduğu anlamına gelir. Bu durumda bir hata kaydı oluşturulmalıdır. Analiz kaynak kod konumunu ifade eden ofset, kaynak kod uzunluğuna ulaşana ya da geçene kadar devam etmelidir zira kaynak kodun analizi bitmemiş olacaktır. Her analiz adımında bir jeton oluşturma girişimi gerçekleştirilmelidir. Jeton oluşturma girişimi başarılı olursa geçerli bir jeton dönüşü gerçekleştirmelidir. Başarılı olan dönüşler saptanmalı ve jeton listesine eklenmelidir.

Jeton oluşturma girişimi algoritmasında ilk aşama ofset için kaldığı yerden ilk anlam kazanabilir teorik karaktere kadar ilerlemektir. Devam etme algoritması boşlukları es geçmelidir, boşluk olmayan her bir karakter bir anlam ifade ediyor olabilir. Boşluklar geçildiği esnada doğru satır ve sütün verilerinin elde edilebilmesi için yeni satır karakterlerine duyarlı olmalı, satır ve sütün verisi gerektiğinde güncellenmektedir. Her bir yeni satırda sütün 1 olurken, satır bir artmalıdır.

Devam etme sonrasında ofset ve sonraki metin jeton oluşturma girişimini ilgilendiren bölümdür. Bu nedenle algoritma ofset ve sonrasına duyarlı olarak tasarlanmalıdır. Devam algoritması sonrası ofset sona ulaşmış ise analiz tamamlanmış demektir, bu nedenle jeton girişimi gerçekleştirilmemelidir.

Devam algoritması sonucu elde edilen devam metni (kaynak kod metni) öncelikle deneme algoritmasına göre tasarlanmıştır. Dolayısıyla her deneme algoritması kendi amacına göre

metinden bir ayrıştırma yapmayı dener ve başarılı olması durumunda olumlu, aksi halde olumsuz dönüş bildirmelidir.

Sayısal değer ayrıştırma, anahtar kelime ayrıştırma, tanımlayıcı ayrıştırma, operatör ve parantez ayrıştırma denemesi olmak üzere 5 adet ayrıştırma deneme algoritması bulunmalıdır.

Öncelikle sürdürülebilir bir geliştirme için operatör ve anahtar kelimeler için bir harita oluşturulmalıdır. Bu harita türler ve ilgili türün sahip olması gereken token kimliğini barındırmalıdır. Örneğin operatörler listesinde yer alan `=` türünün kimliği `OPERATOR`, anahtar kelime listesinde yer alan `print` türünün kimliği ise `PRINT` kimliğine eşlenmelidir. Bu haritalara duyarlı algoritma tasarımı gerçekleştirerek ilerleyen aşamalarda dile yeni bir operatör veya anahtar kelime eklemek için ilgili eşleşmenin eklenmesinin yeterli olacağı bir algoritma oluşturulmalıdır, elbette bu senaryonun gerçekleştirilemeyeceği istisnai durumlar meydana gelebilir.

Jeton deneme algoritmasında tüm ayrıştırma deneme algoritmaları sırasıyla denenmelidir. Her bir algoritma başarılı olma durumunu bildirdiğinden, olumsuz bir bildiri durumunda sonraki ayrıştırma algoritması denemesi gerçekleştirilmelidir. Hiçbir deneme algoritmasının başarılı olamaması durumu kaynak kod içerisinde geçersiz bir öge yer aldığı anlamına gelir. Bu durum hata kayıtlarında bildirilmelidir.

Ayrıştırma deneme algoritmasında, deneme sırası önemlidir. Birbirine sorun çıkarak bir deneme sırası yanlış bir sözcüksel analiz gerçekleştirilmesine neden olabilir. Örneğin Birden fazla karakterden oluşan bir `**` operatörünün olduğunu varsayalım, bu durumda bu deneme `*` operatöründen önce gelmelidir. Bunun nedeni, operatörün doğru bir şekilde ayrıştırılabilmesidir. Öncelikle `*` operatörü ayrıştırılması durumunda arka arkaya iki adet `*` operatörü elde edilir. Elbette bu sorun bu kaynakta anlatılan algoritma tasarımının bir parçasıdır. Farklı paradigmalar bunu bir sorun olarak kabul etmeyebilir.

Öncelikle sayısal ayrıştırma denemesi gerçekleştirilmelidir. Dil tasarımında kayan nokta sayıları nokta (`.`) ile temsil edilmektedir. Algoritma yalnızca onluk sistemin gösterimlerini desteklediğinden bu bağlamda algoritmada daha önceden tanımlanmış olan, bir karakter onluk sisteme ait bir rakama tekabül ediyorsa olumlu etmiyorsa olumsuz dönüş yapan fonksiyonun kullanılması yeterlidir.

Deneme algoritmasına göre sayı en azından bir adet onluk sayı sistemi rakamını ya da bir nokta ve onluk sayı sistemi rakamını bulmalıdır.

Dolayısıyla algoritmanın bunlarla eşleşmesi gerekir: `1`, `59`, `1.`, `2.`, `2.5`

Aksi halde algoritma bir eşleşme bulamamış olduğunu bildirmelidir.

Diğer denemeler sırasıyla operatör ve anahtar kelime ayrıştırması olmalıdır. Bu iki algoritma birbirine benzer şekilde uygulanabilir. İkisi de bir haritaya duyarlı olarak tasarlanacağından, harita üzerinde gerçekleştirilen bir iterasyon yardımı ile gerekli iki veri olan tür (kind) ve kimlik bilgilerine ulaşılabilir. Türe göre ilgili eşleme algoritması çalıştırılarak eşlenme durumunda jeton üstünde ilgili tür, kimlik ve diğer atamalar gerçekleştirilmelidir. Harita üstünde hiçbir eşleşme olmaması durumunda olumsuz dönüş bildirilmelidir.

Operatörlerin eşlemesi yalın bir şekilde yapılabilir. Operatörler özel karakterlerden oluşuyor ise (Ör. Ünlem, nokta, soru işareti vb.) basitçe devam metninin bu karakterle başlayıp başlamaması kontrol edilebilir.

Anahtar kelimelerde birtakım kontroller uygulanmalıdır zira anahtar kelimeler de birer tanımlayıcıdır. `FOO` tanımlayıcısına sahip olan bir anahtar kelime olduğunu kabul edelim, bu durumda anahtar kelime eşlemesi `FOO_` ile eşleştirilmemelidir zira bu bir anahtar kelime değil tanımlayıcı olmalıdır. Bunun nedeni alt çizgi (`_`) karakterinin tanımlayıcılarda kullanılabiliyor olmasıdır.

Anahtar kelime tanımlayıcısının eşleştirilmesi için tanımlayıcılarda kullanılamayan bir durumla karşılaşması gerekir. Bu durumlar da yalın bir şekilde ofsetin kaynak kod üstünde sona ulaşması, tanımlayıcıdan sonra bir boşluk karakterinin gelmesi ya da tanımlayıcıda kullanılamayacak bir karakter gelmesi gerekmektedir şeklinde özetlenebilir.

Anahtar kelimelerden sonra tanımlayıcılar ayrıştırılmalıdır. Bu sıranın nedeni yukarıdaki açıklandığı üzere ayrıştırma hatalarını önlemektir. Her bir anahtar kelime bir tanımlayıcı olduğundan ayrı olarak önce ayrıştırılmalıdır zira yalın tanımlayıcı ayrıştırma algoritması anahtar kelimelere duyarlı değildir. Bu durumda anahtar kelimeler hiçbir zaman ayrıştırılamaması söz konusu hale gelir.

Tanımlayıcı ayrıştırma algoritması da anahtar kelimelerle aynı kriterlerde ayrıştırılmalıdır.

Son olarak parantezler ayrıştırılmalıdır. Her bir parantez için ilgili sayma işlemi gerçekleştirilmelidir. Örneğin `(` için sayaç bir artırılırken `)` için bir azaltılmalıdır. Parantez kapanmalarında ek olarak sayacın sıfır olma durumu kontrol edilerek hata anında saptanabilir.

Bir parantez kapanması durumunda sayaç sıfır ise, hiçbir parantez açılmamış demektir. X dilinin tasarımına göre parantezler muhakkak açılıp kapanmalıdır. Bu durumda bu bir hatadır, bu nedenle hata kayıtlarında belgelenmelidir.

Parantez ayrıştırmalarından sonra halen bir eşleşme yoksa, tanımsız öge olduğundan ilgili karakterin hata olarak belgelenmesi gerekir.

Başarılı olunması durumunda oluşturulan jeton dönüş olarak bildirilmelidir. Ofset ve sütün ilgili ayrıştırmaya göre güncellenmelidir. Dönüş alınan jeton ana algoritmadan hatırlanabileceği üzere jeton listesine eklenecektir. Bu şekilde kaynak kod üstünde ilerlenmesi sonucu tüm kaynak kod analiz edilmiş ve jetonlaştırılmış olacaktır.

JETON ÖRNEKLERİ	
Tip	**Tür**
Anahtar Kelime (Keyword)	`while`, `for`, `if`, `switch`, `else`, `case`
Operatör (Operator)	`+`, `-`, `!`, `&`, `<`, `=`, `*`, `/`, `>`, `|`, `%`
Tanımlayıcı (Identifier)	`i`, `j`, `name`, `salary`, `pi`, `number`, `host`
Yorum (Comment)	`/* BLA BLA BLA */`
Sabit Değer (Literal)	`"PLTR"`, `3.14`, `'x'`, `1881`, `false`

19. SOYUT SÖZDİZİMİ AĞACI (ABSTRACT SYNTAX TREE - AST)

Soyut sözdizimi ağaçları ya da yalnızca sözdizimi ağacı (Syntax Tree) bilgisayar bilimlerinde yorumlama, derleme ya da çevirme işlemlerinde yardımcı olması için tasarlanmış, biçimsel bir dil kullanılarak yazılan kodun (genellikle kaynak kod) soyut bir ağaç temsilidir.

Bir AST ayrıştırıcı (Parser) tarafından oluşturulur. Bu işleme ayrıştırma (parsing) denir. Ayrıştırıcılar kodu parçalamak ve anlamlandırmak için bir temsil üretmek amacıyla kullanılan ara derleyiciler şeklinde tanımlanabilir. Çevirici, derleyici ve yorumlayıcılarda sıklıkla yer alırlar. Genellikle ayrıştırma için sözcüksel analiz sonucu elde edilmiş olan jetonlar kullanılır.

Her bir düğüm sözdizimi içerisinde yer alan her bir yapıyı temsil eder. Soyut bir ağaçtır zira her ayrıntıyı değil yalnızca yapısal ayrıntıları temsil eder. Genellikle kaynak kod analizinin bir sonucu olarak oluşturulur. Derleyici, yorumlayıcı ya da çevirici için çalışma zamanında önemli bir yardımcı araçtır. Nihai sonucun üretilmesinde büyük etkiye sahiptir.

AST yalnızca derleyici, yorumlayıcı ya da çevirici tarafından biçimsel bir dilde yazılan kodu anlamlandırmak için oluşturulmaz. Aynı zamanca bir programlama dili için geliştirilmiş olan herhangi bir araç, dil içerisinde gerçekleştireceği analizler gereği AST oluşturmayı ve bundan faydalanmayı seçebilir.

Bir AST için önemli olan yapısal ayrıntılar şunlardır:

- Yürütülebilir ifadelerin (Statement) doğru bir şekilde sıralanması
- Bir tanımın bildirildiği konum, tanımlayıcısı, statik tipli ise veri tipi
- Atamalarda (Assignment) tanımlayıcılar ve atama ifadeleri (Expression)
- İkili işlemler (Binary Operation) ve sağ ile sol ifadeleri (Expression)

AST tasarımı geliştiricilerin tercihlerine göre farklılıklara sahip olabilir. Örneğin değerlendirilebilir bir ifadenin AST temsilinde ağaç düğümleri ile genellikle operatör önceliğine göre ayrılsalar da farklı bir AST tasarımı her bir ikili işlemi düğümlere ayırmak yerine sıralı birer ana düğüm olarak temsil edebilir ya da bir AST tasarımı koşullu ifadeleri birbirleri ile ilişkili olarak temsil ederken farklı bir AST tasarımı birbirinden bağımsız olacak şekilde temsil edebilir.

Bir AST oluşturulduktan sonra bünyesindeki temsiller değişemez diye bir kısıtlama yoktur. Daha sonra AST temsillerinin işlenmesi ya da farklı nedenlerden dolayı ağaçlara ya da düğümlere yeni bilgiler ve açıklamalar eklenebilir.

Soyut sözdizimi ağaçlarının genel olarak ana kullanım amacı işlenen kodun daha anlamsal bir forma getirilmesidir. Sözcüksel analiz sonucu jetonlaştırma ile az seviyede de olsa karakterize edilmiş olan kod anlam kazanmış olsa da işlemek ve üzerinde çeşitli anlamlandırmalar yapmak için halen çok karmaşıktır. Bu nedenle sözcüksel analiz sonucu elde edilen jetonların sunmuş olduğu karakterizasyondan faydalanarak AST oluşturulur.

Örneğin işlem önceliğine sahip olan bir programlama dili içerisinde sayısal değerler kullanılarak aritmetik bir ifade (Expression) yazıldığını kabul edelim.

Bahsi geçen ifade: `5 + 9 * (2 * 6) / 2.5`

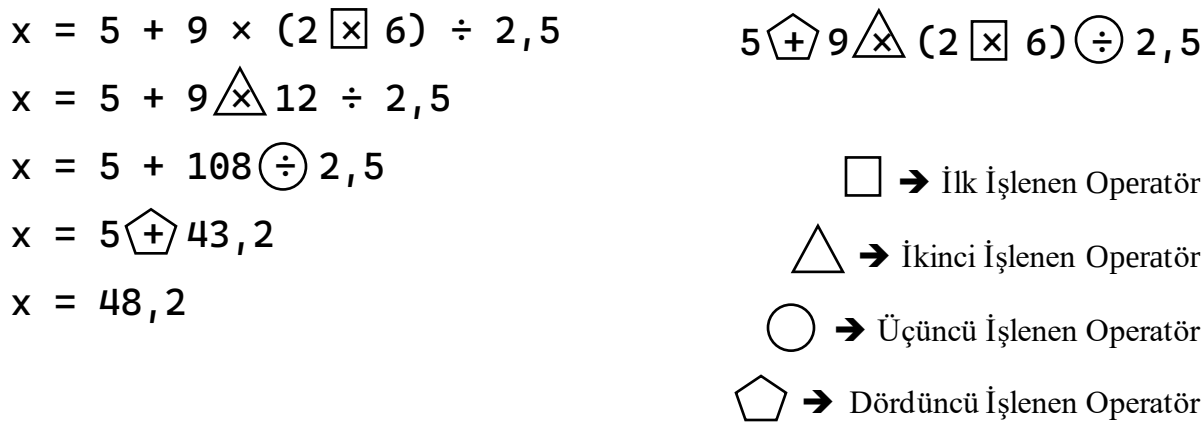
İfadenin matematiksel gösterimi: `5 + 9 × (2 × 6) ÷ 2,5`

Sözcüksel analiz sonrası elde edilen jetonlardan yararlanılarak bu ifadenin AST temsili oluşturulacaktır. Söz konusu ayrıştırma paradigması işlem önceliğini ayrıştırma esnasında değerlendirmekte ve işlem önceliğine göre bir düğüm ağacı oluşturmaktadır.

Matematikte işlem sırası yüksekten küçüğe bu şekildedir:

- Parantezler
- Üsler ve Kökler
- Çarpma ve Bölme
- Toplama ve Çıkarma

İşlem sırası ise soldan sağa olacak şekildedir. Buna göre aynı önceliğe sahip olan bir ikili işlem ile karşılaşıldığında soldan sağa doğru ilerleyen bir çözüm aşaması sergilenmelidir.

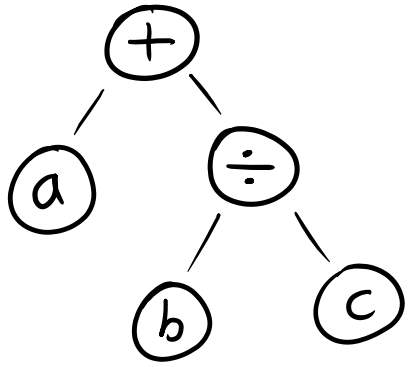


Yukarıda ifadenin matematiksel çözümünün aşamaları paylaşılmıştır ve operatör önceliği geometrik şekillerle temsil edilmiştir. Değerlendirme aşamaları yalnızca operatör önceliği bakımından AST tasarımının umurunda olmalıdır, AST içerisinde yalnızca yapısal detaylara yer verildiği açıklanmıştı. Bu durumda öncem arz eden nokta işlem önceliğine göre bir AST oluşturmaktır.

İlk işlenen operatörün kare ile temsil edilen operatör olmasının nedeni parantezlerin öncelikli olmasından kaynaklanmaktadır. İkinci işlenen operatörün üçgen ile temsil edilen operatör olmasının nedeni ise soldan sağa işlem kuralıdır. Aynı önceliğe sahip iki operatör olsa da bu durumda soldan sağa yürütme nedeniyle çarpma işlemi bölme işleminden önce gerçekleşir. Daha sonra işlem önceliği gereği bölme ve hemen ardından toplama işlemi gerçekleştirilir. Ayrıştırma sırasında operatör önceliğine göre temsil ağacı oluşturulacağından bu sıraya dikkat edilmeli ve düğümler uygun şekilde eklenmelidir.

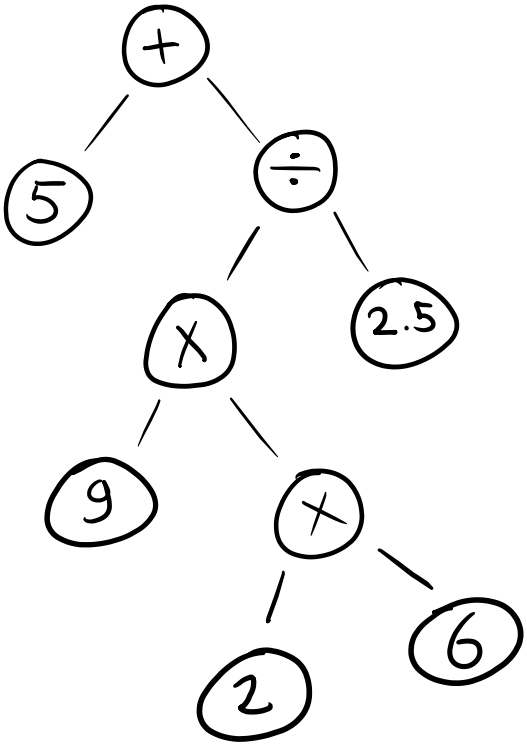
İkili işlem ağacı buna benzemektedir:

(Şekil 1)



Bu bilgilere göre örnek ifadenin potansiyel bir AST temsili buna benzemelidir:

(Şekil 2)



Şekil 1'de de gösterildiği gibi en yüksek öncelik en alt düğümden itibaren başlar. Bunun nedeni bu tasarıma sahip olan bir AST temsilinin yürütülme paradigmasıdır. İlk ikili işlemin sol işleneninin bir tanımlayıcı sağ işlenin ise farklı bir ikili ifade olduğu görülebilir. Bu ifade bir bölme işlemidir. İlk düğümün gerçekleştirilebilmesi için sağ düğümün çözümlenmesi gerektiğinden önce bölme işlemi olan düğüm değerlendirilecek ve sonuç üst düğümde kullanılacaktır.

Bu alt düğümler daha da uzayabilir. Aynı yöntemle, en alttan en üste değerlendirme olması gerekeceğinden işlem önceliği yük

Örnek ifadenin yukarıdaki yaklaşıma göre tasarlanmış temsilinde (Bkz. Şekil 2) ilk işlenen operatör anlatıldığı gibi en alt düğümde yer almaktadır. Dikkat edilmesi gereken nokta parantez gruplarının bulunmadığıdır. Bunun nedeni işlem önceliğine göre operatör ve işlemlerin düğümlere ayrılması sonucunda parantezlerin ayrıştırılarak önceliği düğümlerle temsil ediyor oluşudur. Düğümler incelendiğinde parantezlerin en alt düğümlerde yer aldığı görülmektedir. Değerlendiricinin bu temsili değerlendirme aşamaları mantıksal olarak bu şekilde olmalıdır: İlk düğüm bir toplama işlemidir. Sol işlenen 5 değeridir, sağ işlenen ise bölme işlemi yapan başka bir ikili işlemdir. Toplama işleminin sol ifadesinin elde edilebilmesi için önce bölme ikili işlemi gerçekleştirilir. Bölme ikili işleminin sağ ifadesi 2,5 olarak değerlendirilir. Sol işlenen ise çarpma işlemi yapan farklı bir ikili işlemdir. Sol ifadenin elde edilmesi için bu ikili ifade değerlendirilir. Bu ikili işlemin sol ifadesi 9'dur. Sağ ifade için başka bir ikili işlem gerçekleştirilir. Bu ikili işlemin sol ifadesi 2, sağ ifadesi 6'dır. Çözümlenen ikili işlemlerin yerini çözüm sonuçları yer alır, en son düğüme ulaştığında bu şekilde algoritma geriye doğru temsili değerlendirir ve nihai

Farklı paradigmalarda tasarım gereği parantez kümeleri ve diğer içerik ayrıntıları düğümler içerisinde yer alabilir. Değerlendirilme aşamalarında bu parantezler farklı anlamlar ifade ediyor olabileceğinden ikili işlem olması ya da bir değerlendirilebilir ifade olması durumu değerlendirilme aşamasında tespit edilir ve parantez içi ifade için farklı bir AST oluşturulup çözümlenir. Diğer durumlarda parantezler temsil ettikleri diğer amaçlara hizmet edecek şekilde değerlendirilirler. Bunun için yalnızca yüzeysel bir AST temsili oluşturulabilir.

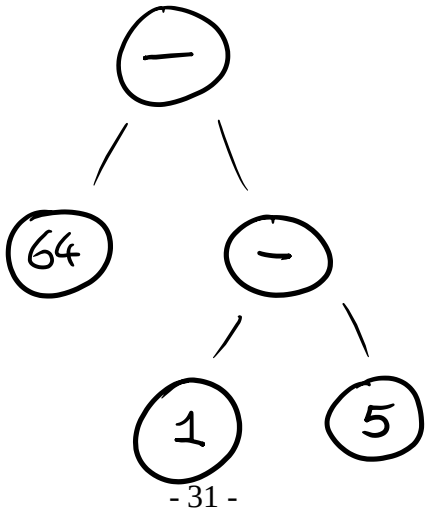
Anlatılan paradigmaya göre AST temsilinin değerlendirildiğinde doğru sonuç verebilmesi için işlenenleri sağdan sola doğru ayrıştırması gerekmektedir. Bunun nedeni matematiğin soldan sağa çözüm kuralıdır. En alt düğüm zorunluluk nedeniyle ilk değerlendirilen olduğundan AST temsilinin sağdan oluşturulması gerekmektedir. Zira sağdan alınan ikili işlemler ilk düğümler olacaktır, sağdan sola gidilmesi sonucunda en soldaki ikili işlem en alt düğüm haline geleceğinden ilk değerlendirilen ikili işlem olacaktır. Bu şekilde doğru bir değerlendirme sıralaması elde edilmiş olur ve ifade sağdan solda temsil edilmesine rağmen olması gerektiği gibi soldan sağa değerlendirilir.

Örnek ifade: `64 - 1 - 5`

Bu ifadenin çözümü: `64 - 1 = 63 - 5 = 58` şeklindedir.

AST temsilinin soldan sağa işlenerek oluşturulması durumunda temsil buna benzemelidir:

(Şekil 3)



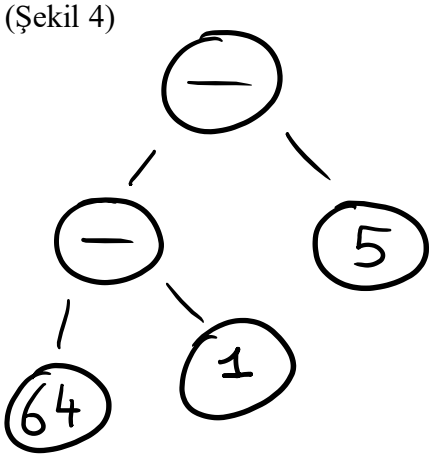
Temsilde (Bkz. Şekil 3) en soldaki ikili işlemin ilk düğüm olduğu görülmektedir. Bu temsilin değeri anlatılan algoritma ile bu şekilde olacaktır:

İlk ikili işlemin sol işleneni 64 olarak değerlendirilir. Sağ işlenen bir çıkarma ikili işlemidir. Bu ikili işlemin sol ifadesi 1, sağ ifadesi 5 olarak değerlendirilir. Bu işlemin sonucunda 4 elde edilir. İkili işlemden dönen 4 ifadesi ilk ikili işlemin çözümünde kullanılarak 64 ifadesinden çıkarılır ve işlemin sonucu 60 olarak değerlendirilir.

İfadenin çözümünün bu şekilde yanlış bir değerlendirme sonucu hatalı olacağı anlaşılmıştır.

AST temsilinin sağdan sola işlenerek oluşturulması durumunda temsil buna benzemelidir:

(Şekil 4)



Bu defa temsilde (Bkz. Şekil 4) en sağdaki ikili işlemin ilk düğüm olduğu görülmektedir. Bu temsilin değeri anlatılan algoritma ile bu şekilde olacaktır:

İlk ikili işlemin sol ifadesi bir çıkarma ikili işlemidir, sağ ifade 5 olarak değerlendirilir. Sol ifadenin elde edilmesi için sol ifadede yer alan ikili işlem değerlendirilir. Bu ikili işlemin sol ifadesi 64, sağ ifadesi ise 1 olarak değerlendirilir. İkili işlemin sonucunda 63 ifadesi elde edilir. İlk ikili işlem gerçekleştirildiğinde 63 ifadesinden 5 ifadesi çıkarılır ve 58 ifadesi elde edilir. Bu yukarıdaki anlatılan yaklaşımın doğru bir değerlendirme sergileyeceğini göstermektedir.

Yukarıda sunulan ve geçerliliği kanıtlanan algoritma iyi bir uygulamada (implementation) verimli ve hızlı bir çalışma zamanı performansı sergilemektedir. Anlatılan algoritmanın yanı

sıra genel olarak benimsenmiş belirli ayrıştırma algoritmaları bulunmaktadır. Bunlardan biri Shunting yard algoritmasıdır.

Atama (Assignment) işlemleri ve diğer işlemler genellikle düğümler şeklinde değil birçok düğümü ve açıklamayı içerisinde barındıran objeler olarak temsil edilirler. Örneğin bir atama işlemini temsil eden obje atama işleminin atama yapılacak taraf için bir ifade (Expression) ve atanan ifade tarafı için bir ifade (Expression) verisi barındırır. Bununla birlikte tasarım gereği gerekiyorsa farklı veriler de bu objenin içerisinde yer alabilir.

Örnekten anlatmak gerekirse bir atama temsili buna benzemelidir:

```
class Assignment:
    left: Expression
    right: Expression
```

Yukarıdaki temsilde bir atama ifadesini temsil eden obje tasarımı görülmektedir. `left` alanı atamayı alacak ifade ederken, `right` alanı atanan ifadeyi temsil etmektedir. Bu ifadeler genel olarak alt düğümlerden oluşan ya da tek bir ana düğümden oluşan ifade (Expression) AST temsilleridir. Bu temsillerin nasıl üretildiğine yukarıda değinilmiştir.

Bu yaygın kullanımdır zira programlama dillerinin tasarımında genel olarak sağ taraf atamayı alan, sol taraf ise atanan ifadedir. Yine de bazı tasarımlara göre her iki taraf da her zaman bir ifade AST temsili olmaz. Bazı programlamama dillerinde sağ taraf yalnızca tanımlayıcı (identifier) verisi olabilir zira programlama dili rvalue ve lvalue gibi kavramlara sahip olmayıp yalnızca değişkenlere atama yapılmasına izin veriyor olabilir. Bu durumda yalnızca tanımlayıcı verisi tutulması yeterlidir.

AST oluşturma aşamalarında genellikle jetonların kullanıldığından bahsedildi. Ayrıştırıcı algoritması bu jetonlardan yararlanarak yaygın yaklaşımda sözcüksel analizde olduğu gibi tüm içeriğin işlenmesi sonucu bir liste dönüşü gerçekleştirir. Bu liste genel olarak sırası korunmuş bir şekilde jetonlardan elde edilen soyut sözdizimi ağacının temsillerini barındırır.

Temsiller genel olarak jetonların karakterizasyonundan yararlanılarak saptanır ve AST içerisinde hangi şekilde temsil edilmesi gerektiği anlamlandırılır. Örneğin algoritma bir tanımlayıcı gördüğünde, deneysel olarak tasarlanmış X diline göre takip eden = operatörü bir atama ya da değişken oluşturma anlamına geldiğinden bunu kontrol ederek sözdizimsel kurallara göre çıkarımlar yapabilir ve jetonların gerekli kısmından faydalanarak ilgili AST temsilini oluşturabilir.

Yürütülebilir ifadelerin (Statement) saptanması da yaygın yaklaşımda sözdizimi kurallarına göre ayrıştırıcı algoritması tarafından gerçekleştirilir. Genel olarak jetonların anlamlandırılması yürütülebilir ifadelerin ayrıştırılmasından sonra her bir potansiyel yürütülebilir ifade için ayrılan jetonların analizi ile gerçekleşir.

AST tıpkı lexer yürütülürken yapıldığı gibi karşılaştığı hataları belgelemek zorundadır. Bunun için yine lexer tarafında olduğu gibi liste kullanılabilir.

20. VERİLERİN İŞLENMESİ

Örnek olarak tasarlanmış deneysel programlama dili üstünden gitmek gerekirse, X kaynak kodu halen bir anlam ifade etmemektedir. Sözcüksel analiz ve ayrıştırma sonucu elde edildikten sonra yorumlayıcının AST temsillerini anlamlandırması ve görevlerini yerine getirmesi gerekmektedir.

Örnek tasarım üstünden gitmek gerekirse:

Yorumlayıcının ek bir temsil verisine ihtiyacı olacaktır. Bu temsil verileri değişkenleri temsil etmek için bünyesinde değişkenin tanımlayıcısı (identifier) ve verisi (data) barındıran alanlara sahip olmalıdır.

Bir değişkeni temsil eden nesne buna benzemelidir:

```
class Variable:
    identifier: str
    data: int | float
```

Yukarıdaki örnekteki temsilde anlaşılabileceği üzere `data` alanı `int` ya da `float` veri tipinde verilere sahip olabilir. Bunun nedeni dil içerinde birden fazla veri tipi bulunmasıdır.

Yorumlayıcı algoritması ayrıştırıcıdan elde ettiği soyut sözdizimi ağacını kullanarak sırasıyla anlamlandırmalıdır. Örnek uygulamada işlenmesi gereken üç adet ana düğüm türü vardır, bunlar: ifade (Expression), print ifadesi (print statement) ve atamadır (assignment).

Algoritma bu üç ana düğüm türünü tanımalı, gerektiği şekilde anlamlandırmalı ve amaçlarını gerçekleştirmelidir.

20.1 Atama (Assignment) İfadelerinin İşlenmesi

Algoritma bir atama ifadesi ile karşılaştığında tasarım gereği öncelikle ifadenin (Expression) değerlendirilmesi gerekir. İfade bağımsız ve her koşulda aynı şekilde değerlendirildiğinden öncelikle değerlendirilmeli ve sonuç ifadesi elde tutulmalıdır. Sonraki aşamada atama yapılan tanımlayıcıya sahip bir değişken olup olmadığı kontrol edilmelidir. Değişken bulunması durumunda var olan değişkeni temsil eden nesne örneğinin verisi değerlendirilmiş olan ifadeye ayarlanır. Bir değişken olmaması durumunda değerlendirilen ifade ile ilgili tanımlayıcıya sahip yeni bir değişken oluşturulur.

20.2 Değerlendirilebilir İfadelerinin (Expression) İşlenmesi

Değerlendirilebilir ifadeler tek başına kullanıldığında değerlendirilip komut satırına yazdırılmalıdır. İfadenin değerlendirilmesi için bir değerlendirme (evaluation) algoritması kullanılmalıdır.

20.3 Print İfadelerinin (Print Statement) İşlenmesi

Algoritmanın karşılaştığı bir print ifadesi, verilen ifadeyi komut satırına yazmaktan çok farklı değildir. Normal bir ifade ile karşılaşıldığında bu doğrudan değerlendirilip komut satırına yazdırılmaktadır, print ifadesinin farkı işlemin çözüm adımları ile komut satırına yazdırılmasıdır.

Bunun için ek bir değerlendirme (evaluation) algoritması tasarımı çok mantıklı değildir. Zira değerlendirme algoritmasında meydana gelen bir değişiklik diğer algoritmaya da uygulamak zorunda kalacaktır. Bu kod tekrarını artıran ve kod bakımını zorlaştıran bir yaklaşımdır.

Bunun yerine geliştirilmiş olan tek bir değerlendirme algoritmasına parametre olarak çözüm adımlarının komut satırına yazdırılması durumu için bir boolean kullanılması iyi bir yaklaşımdır. Dolayısıyla algoritmanın print ifadesini işlerken tek farklı yaptığı nokta normal ifadeyi işlerken yaptığının aksine değerlendirme algoritmasından çözüm aşamalarını komut satırına yazmasını istemek olacaktır.

20.4 Değerlendirme (Evaluation) Algoritması

Değerlendirme algoritması değerlendirilebilir ifadeleri değerlendirir. Bir ifadeni iki şekilde temsil edilir: tek ifade ve ikili işlem. Tek ifadeler bir değer ifade eden düğümlerdir. Tek başlarına bir ifade bildirmeleri gereklidir. İkili işlemler ise iki ifadenin birbiri ile bir operatör ile kullanılmasını sağlar. İkili ifadenin değerlendirilmeden önce sağ ve sol işlenenlerinin değerlendirilmesi gerekir.

Tek ifadelerin değerlendirilmesinde gözetilmesi gereken üç durum vardır, bunlar: tamsayı (integer), kayan noktalı sayı (float) ve değişken (variable) olma durumlarıdır.

Bir tek ifade tanımlayıcı ise bu tanımlayıcı bir değişkene ait olmalıdır. Tanımlayıcıya sahip bir değişken henüz tanımlanmamışsa bir değerlendirme hatası alınmalıdır. Değişkenin var olduğu taktirde sahip olduğu veri direkt olarak döndürülebilir zira değişkenlerin ifadeleri hali hazırda değerlendirilmiş olacağından veri tipi kontrolü her atama esnasında yapılmış olmaktadır.

Tek ifadenin aritmetik bir değer olması durumunda ifadenin tamsayı ve kayan noktalı sayı olması fark yaratmalıdır. İfadede nokta (`.`) bulunması açıkça kayan noktalı sayı anlamına gelmektedir. Bu durumda kayan noktalı sayı olarak değerlendirilmeli ve döndürülmelidir. Aksi halde bir tamsayı dönüşü sağlanmalıdır.

İkili işlemlerin çözümünde öncelikle sağ ve sol ifadeler elde edildikten sonra operatöre göre bu ifadeler değerlendirilmelidir. Değerlendirilmenin sonunda tek bir ifade elde edilmelidir. İkili işlemin sağ ve sol ifadeleri tamsayı ise değerlendirilmiş ifade de tamsayı olmalıdır. Sağ ve sol ifadelerden herhangi biri kayan noktalı sayı ise değerlendirilmiş ifade de kayan noktalı sayı olmalıdır.

21. YORUMLAYICIYI (INTERPRETER) ANLAMAK

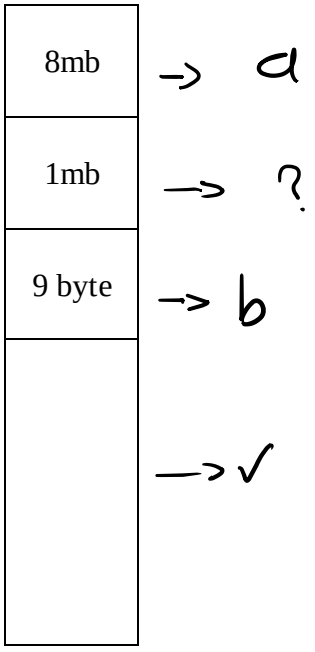
Yukarıdaki bir yorumlayıcı üzerinden temel olarak bir kaynak kodun nasıl anlamlandırıldığına değinilmiştir. Yorumlayıcının kaynak kodu makine koduna derleme davranışı sergilemediği görülmektedir. Daha önce yorumlayıcının eninde sonunda makine kodu seviyesinde çalışan bir yazılım tarafından yürütülmesi gerektiğine değinilmişti. Makine kodu ile olan iletişim tam olarak burada gerçekleşmektedir.

Yorumlayıcının kendi içerisinde bir bilgisayar gibi çalışıyor olduğunu söylemek mümkündür. Değişken tanımlarını ve dil tasarımında varsa diğer tanımları depoladığı bellek alanı yorumlayıcının kendisi için yarattığı sanal bir bilgisayar belleğinden farksızdır.

Örnek tasarımdan anlatmak gerekirse:

Değişkenler için oluşturulan liste CPython (Resmi Python Yorumlayıcısı) tarafından yönetilmektedir. CPython C programlama dili kullanılarak geliştirilmiş bir yorumlayıcıdır. Dolayısıyla liste tahsisinin yönetimi CPython yorumlayıcısının makine kodu haline getirilmiş C kodları tarafından gerçekleştirilmektedir. Python içerisinde kullanılan liste makine kodu tarafından bellekte ayrılmış olan bir alana Python içerisinde verilen bir referanstır.Makine kodu seviyesinde bir belleğin temsili bu şekildedir:

(Şekil 5)



Şekil 5'te bir bellek temsil edilmiştir. Üç farklı boyutta toplam bellek boyutu içerisinden segmentler tahsis edildiği görülmektedir. Buradaki iki tahsis bir ABC yazılımının kendisi için ayırdığı `a` ve `b` değişkenlerine aittir. Diğer tahsis edilen alanın tahsisi farklı bir kaynağa aittir. Geri kalan bellek alanının tahsis edilebilir boş alan olduğu görülmektedir.

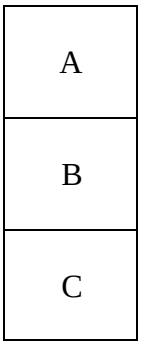
ABC yazılımı 5 bayt boyutunda yeni bir tahsis gerçekleştirdiğinde bellekteki boş alanlardan biri tahsis edilecektir. Bu bellek tahsisleri bellek üzerinde gerçekleştikleri segmentin sahip olduğu benzersiz bellek adresi ile temsil edilmekte ve ulaşılmaktadır.

Bu örnekleme bellek segmentasyonu (memory segmentation) yönteminin kullanıldığı bir sistem baz alınarak yapılmıştır.

Yukarıdaki bellek tahsisleme örneğinin bir benzeri yorumlayıcı içerinde de bulunmaktadır. Yorumlanan bir dilin değişkenleri saklamak için kullandığı bir liste olduğunu varsayımında çalışma prensibi oldukça yakındır. Her bir değişkeni temsil eden nesne örneği (instance) liste içerisinde eklenir.

Örnek bir liste üzerinde anlatım:

(Şekil 6)



Şekil 6 bir listenin temsilidir. Üç bileşen bulunmaktadır ve bu bileşenler sırasıyla A, B ve C değişkenleri için temsil verileri taşımaktadır. Yeni bir değişkenin eklenmesi bellekte yeni bir segmentin tahsis edilmesine benzer şekilde gibi listeye yeni bir bileşen eklenerek sonuçlanacaktır.

Bellek adreslerinin yerini ofset (offset) alır. Listenin bileşenlerinin bir segment olduğu benzetmesi yapılması durumunda bileşenin bellek adresinin ofset olduğu söylenebilir. Bu durumda yorumlayıcı bileşen verisine erişmek ya da atıfta bulunmak için ofset kullanabilir.

Bu şekilde listede depolama yaklaşımının bellek yönetimi ile benzerlikleri özetlenebilir.

Yukarıda açıklandığı üzere bir yorumlayıcı tanımlarını bu şekilde saklayabilir ve kullanabilir. Daha önce de yapılan bir benzetme ile bu yaklaşımın yorumlayıcının oluşturduğu bir bellek taklidi olduğu söylenebilir. Yorumlayıcı sahip olduğu tanımları bu ve buna benzer şekillerde saklayarak bir işlemci gibi davranır, işlemleri gerçekleştirmek için edindiği veri ve girdilerden yararlanarak eylem alır.

22. ÇEVİRİCİYİ (TRANSPILER) ANLAMAK

Çevirici yukarıda anlatılan süreçlerin çoğuna sahiptir. Tek fark bir çeviricinin neden çalıştığıdır. Örneğin bir yorumlayıcı kaynak kodu anlayıp yürütmek için çalışır. Çeviricinin amacı bir dilden diğerine çeviri yapmak olduğundan (tercüman benzetmesini hatırlayın) kodu yürütmek gibi yönelimleri yoktur.

Bir çevirici kodun sözcüksel analizini ve ayrıştırma aşamalarını tamamladıktan sonra, bu aşamalar algoritma tasarımına ve bazı ek kriterlere göre değişiklik gösterebilir, yapması gereken şey çeviriyi gerçekleştirmektir.

Çeviriciden elde edilen çıktı girdi olarak alınan kodun farklı bir dilde yazılmış versiyonudur. Bu çeviricinin ABC dilinden XYZ diline çeviri yapabilmesi için iki dilin aynı özelliklere, sözdizimine ya da çalışma prensibine sahip olması gerekmez. Çeviri yalnızca ortak özellikler kullanıldığında mümkün kılınabilir ya da hedef dilde var olan özellikler kullanılarak çevrilen dilin hedef dilde olmayan özellikleri ile aynı davranacak algoritmalar oluşturularak çevrilmesi gibi yaklaşımlar benimsenebilir.

23. DERLEYİCİYİ (COMPILER) ANLAMAK

Derleyiciler yürütülebilir dosyalar oluşturulmak amacıyla yaygın olarak kullanılırlar. Derleyiciler de çalışma prensiplerine göre kendi içlerinde ayrılırlar. Tipik bir derleyici yorumlayıcı ve çeviricide olduğu gibi sözcüksel analiz ve ayrıştırma aşamalarını gerçekleştirir.

Derleyicilerin ürettiği yürütülebilir dosyalar her zaman platforma özel olarak derlenmiş makine kodlarından oluşmaz. Ara gösterim dillerini hatırlayın. Bir derleyici yürütülebilir dosyayı bir ara dile derleyerek oluşturabilir. Bu ara gösterimlerin çalışma zamanında yorumlanarak platforma özel şekilde derlenmesi ve bu şekilde ara dillerin platformlar arası bir derleme gerçekleştirilebilmekte kullanılabileceğine değinilmişti.

Ara dillerin en popüler örnekleri Microsoft .NET, Java Sanal Makinesi (Java Virtual Machine - JVM) ve BEAM (Erlang Virtual Machine) şeklindedir. Bu platformlar kendilerine özgü bir ara dil kullanırlar. Bu ara dil türü örneklerin hepsinde bytecode şeklindedir. Örneğin JVM Java Bytecode olarak adlandırılan bir ara dil kullanmaktadır. Bu ara dilin çalışma zamanında yorumlanması ve ilgili platforma göre uyumlu makine kodunun oluşturulmasına daha önce değinilmişti (bkz. P-Code Machine). Bu şekilde derlenen diller zamanında (Just in Time – JIT) derleme tekniğini kullanır.

Makine üzerinde bulunan işlemcinin komut setine uygun şekilde derlenen saf makine kodlarının işlemci tarafından doğrudan yürütülebilir olduğundan en saf çalışma performansı bu şekilde derlenen dillerden elde edilebilir. Bu diller zamanından önce (Ahead of Time - AOT) derleme tekniğini kullanır.

İki ana derleyici türü vardır, bunlar: yerel (native) ve çapraz (cross) derleyicilerdir.

Yerel derleyiciler üzerinde çalıştığı platforma özgü derleme gerçekleştirirler. Dolayısıyla yerel bir derleyiciden elde edilecek çıktı yalnıza derleyicinin derleme zamanında yürütüldüğü sisteme yönelik olacaktır (Örn. Yerel bir derleyici ile C kodunun AMD64 mimarisini kullanan Linux çekirdeğinde çalışan bir makinede derlenmesi sonucu AMD64 ve Linux çekirdeğiyle uyumlu makine kodu elde edilmesi). Yerel derleyicilerin platform bağımsız düşünmeleri gerekmediğinden AOT derleme tekniği ile direkt olarak makine koduna derlenirler.

Çapraz derleyiciler üzerinde çalıştıkları platformdan bağımsız bir şekilde destekledikleri tüm platformlara özel olarak makine kodu derlemeleri üretebilirler. Buna örnek olarak ARM64 üzerinde çalışan bir derleyicinin I386 için uyumlu makine kodu derlemesi gösterilebilir.

Derleyici yalnızca kodu derlemekle ilgilenmez aynı zamanda derleme zamanında gerekli gördüğü yerlerde kodu optimize edebilir ve daha optimum bir makine kodu üretmek için çabalayabilir. Bunlara derleyici optimizasyonları denir.

23.1 AOT Derlemesi

Zamanından önce derleme tekniği bir programın yürütülmeden önce (genellikle) derlenen dilden daha düşük seviyeli başka bir dile derlenmesini temel alır. Bu yaklaşım çalışma zamanında gerekli olan iş yükünü azaltmaktadır.

Zamanından önce teriminde kastedilen programın yürütülme aşamasından öncesidir. Örneğin bir C kodunu yürütmeden önce derlenmesi ve yerel makine kodunun elde edilmesi işlemi AOT derlemeye bir örnektir. C kodunun yerel makine kodu yürütülmeden önce derlenmiştir ve derleme sonucunda çalışma zamanı ile ilgili bir yük bulunmamaktadır.

AOT derlemesi aynı zamanda ara temsil kullanan dillerin ara temsillerinin makine koduna derlenmesinde kullanılır. Örneğin Microsoft .NET platformunun dili olan MSIL (Microsoft Intermediate Language) çalışma zamanında AOT derlemesi ile yerel makine koduna derlenir.

AOT derlemesini direkt olarak hedef platformun yerel makine diline derleme gerçekleştirmek için kullanan derleyiciler genellikle gömülü sistem geliştiricileri tarafından tercih edilirler. Bunun nedeni üretilen çıktının yürütülmek için bir bağımlılığının ve derleme tekniği gereği çalışma zamanında ek bir yüke sahip olmamasıdır. Gömülü sistemlerde kaynaklar sınırlı olduğundan saf makine koduna derlemek ve yerel yürütülebilir çıktı almak önemlidir.

23.2 JIT Derlemesi

JIT derlemesi yorumlayıcı ve AOT derlemesinin birleşiminden meydana gelen bir derleme tekniğidir. Ara dile derlenen dillerin derleyicileri JIT tekniğini kullanmaktadır. JIT derlemesinde kaynak kod bir ara dile derlenir. Ara dillerin platform bağımlılığını azaltmak için kullanılabildiği bilgisini hatırlayın. Bu teknikte ara dilden o an yazılımın yürütüldüğü platforma uyumlu makine koduna olan gerçek derleme eylemi çalışma zamanında gerçekleşir.

Çalışma zamanında derlenen dil yaygın olarak bytecode formundadır. Çalışma zamanında derlenen ara dil daha sonra derlenmiş yerel makine kodu üzerinden doğrudan yürütülür. Çalışma zamanında ara dilden makine diline derleme işleminde AOT tekniği kullanılır. JIT derleme tekniğine dahil olan yorumlayıcı ise ara dilin yerel makine koduna olan çevirisinde gerçekleştirilmektedir. Tıpkı bir yorumlayıcı gibi ara dilin platforma uygun bir şekilde yorumlamasını gerçekleştirir. Uygun yorumlama ile AOT derlemesi yapılarak yerel makine kodu sonucu elde edilir. Bu işlem aslında AOT kullanan derleyicilerin kaynak kodu anlayarak makine koduna derlemesinden çok da farklı değildir, orada gerçekleşen de bir nevi çalışma zamanından önce kodun makine koduna yorumlanması ve derlenmesidir.

Bir JIT derlemesi ürünü tipik olarak çalışma zamanında sürekli olarak profilleme çıkartır ve analizler gerçekleştirir. Bu analiz ve profillemenin sebebi kodun sık kullanılan bölümlerini tespit etmek ve performans getirisi için önceden derleyerek ilgili bölümün tekrar ve tekrar derlenme gereksinimini ortadan kaldırmak gibi performans kaygılarıdır.

Çalışma zamanında derleme gerçekleştirdiğinden dinamik derleme (dynamic compilation) ve dinamik yeniden derleme (dynamic recompilation) yapabilme yeteneğine sahiptir. Dinamik derleme çalışma zamanında kodun bölümlerinin derlenmesidir, JIT tekniğinin yaptığı tam olarak budur. Dinamik yeniden derlenme ise çalışma zamanında kodun bölümlerinin yeniden derlenebilmesidir, JIT derlemesi ürünü dinamik yeniden derleme de gerçekleştirebilir ve çalışma zamanında derlenmiş olan kodları yeniden derleyebilir.

JIT derlemesi ürünleri genellikle ilk esnada yavaş bir yürütme hızına sahip olsalar da yürütmenin ilerleyen aşamalarında dinamik derleme ve yeniden dinamik derlemeler büyük oranda tamamlanmış olacağından oldukça performanslı çalışabilirler. Dinamik olarak derlenebildiğinden platformda özgü bazı optimizasyonlar gerçekleştirilebilir ve AOT derlemesinden daha iyi bir çalışma performansı elde edilmesini sağlayabilir.

JIT derlemesinin yorumlayıcı ile olan bir diğer benzerliği çalışma zamanı bağımlılığında harici bir yazılıma bağımlı olmasıdır. Yorumlanan bir dilin yürütülmesi için yorumlayıcıya ihtiyaç duyduğunu hatırlayın. JIT derlemesi kullanan bir dil de çalışma zamanında yürütülebilmesi ve derlenebilmesi için ilgili çalışma zamanı platformunun mevcut olmasını gerektirir. Bkz. Java için JDK ve Microsoft .NET için .NET ya da .NET Framework.

Yorumlanan dillerle bir diğer ortak özelliği platformlar arası taşınabilirliğinin bağımlı olduğu yazılımla sınırlı olmasıdır. Örneğin yorumlanan bir dili yorumlayıcısının yürütülebildiği her platformda yürütmek mümkün olmalıdır. Aynı şekilde JIT derlemesi olan bir dili de çalışma zamanı desteklenen tüm platformlarda yürütmek mümkün olmalıdır.

Bu bağımlılık gereği gömülü sistemler gibi sınırlı kaynaklara sahip geliştirmelerde tercih edilmez. Çoğunlukla da tercih edilemezler. Bunun nedeni gömülü programlamanın yaygın senaryoda doğrudan makine koduna ihtiyaç duymasıdır. JIT derleyicinin çalışması için gerekli olan çalışma zamanı platformuysa gömülü sistemlerde çoğunlukla kullanılabilir değildir.

Ara diller ayrıca ortak bir çalışma platformunun oluşturulmasında da kullanılır. Örneğin Microsoft .NET platformunda bulunan C#, Visual Basic.NET ve F# gibi programlama dilleri aynı çalışma zamanı platformunu (bahsi geçen sanal makine ya da sanal işlemci, konuya daha önce bytecode başlığında değinildi) kullandığından aynı ara dile (MSIL) çevrilirler. Bu durum bu dillerde geliştirilmiş olan kodların ara dile derlenmesinden kaynaklı olarak bir DLL formunda birbirleri içerisinde kullanılabilmesini sağlar zira konuştukları dil ortaktır.

23.3 Derleyici Yapısı

Bir derleyicinin yapısı hedef programlama diline göre değişiklik gösterse de genel olarak bir derleyicinin uygulanması (implementation) yaygın şekilde benzer aşamaları takip eder.

23.3.1 Ön Uç (Compiler Front-End)

Bir derleyicinin ön ucu, kaynak kodun analiz edilerek daha kolay bir şekilde anlamlandırılması ve anlamlandırılmadan önce bir takım sözdizimi gibi hataların kontrol edildiği yerdir.

Ön uçta yer alan aşamalar: Önişlemci, Sözcüksel Analiz

23.3.2 Arka Uç (Compiler Back-End)

Bir derleyicinin arka ucu işlenmiş olan ara temsilden yararlanarak hedef derleme platformuna uygun bir şekilde makine kodu oluşturulduğu yerdir. Birden fazla işlemci mimarisini

destekleyen derleyiciler her bir mimari için farklı sonuçlar üreten farklı arka uçlara sahip olabilirler.

Arka uçta yer alan aşamalar:

- **Kod oluşturma:** platforma uygun şekilde makine kodunun oluşturulmasıdır. Makine kodu içerisinde programın algoritmasının gerçekleşmesi gerekli olan talimatların yanı sıra hata ayıklama ve çalışma zamanı için özel olarak ek kodlar bulunabilir.
- **Platforma bağlı optimizasyonlar:** derleyici kod ürettiği platforma özel olarak üretilen makine kodu üstünde optimizasyonlar uygulayabilir ve girdi olarak alınan kaynak kodun teorik olarak daha iyi çalışan bir sürümünü elde edebilir.

24. MONTAJ DİLİ (ASSEMBLY)

Montaj dili makine koduna oldukça yakındır. Daha okunabilir ve nimoniklerle desteklenmiş bir formu olduğu söylenebilir. Montaj dilinin sahip olduğu nimonikler işlemci komut seti içerisindeki talimatların daha akılda kalıcı ve okunabilir olmasını sağlamak amacıyla eklenmiştir. Montaj dilinde yer alan bir nimonik muhtemelen bir anahtar kelimedir ve bu anahtar kelime komut seti üzerindeki bir izlenceyi temsil eden ikili sayı kodlamasını (binary) temsil eder.

İşlemci komut setlerinin ikili kodlamalarının mimariye göre farklılık göstermesinden dolayı montaj dilleri de tıpkı makine kodları gibi mimariye göre farklı şekillerde yazılmaları gerekir.

Montaj dilleri derleyicilere değil montajcılara (assembler) sahiptir. Bu montajcı yazılımları derleyicilerden farklıdır zira derleyiciler içlerinde optimizasyon ve benzeri farklı davranışlar sergiliyor olsa da montajcılar yaygın olarak izomorfik şekilde çalışırlar. Yani bir montajcı yaygın olarak monta kodu ile birebir aynı makine kodunu oluşturur. Montaj kodu nimonikler kullanılarak yazılmış bir makine kodu temsili olduğundan aslında anlamsal olarak makine kodundan çok da farkı yoktur.

25. PROGRAMLAMA DİLLERİNİN SINIFLANDIRILMASI

Programlama dilleri iki farklı şekilde sınıflandırılırlar, bunlar: düşük seviye ve yüksek seviyedir. Bu sınıflandırmalar ilk görüldüğünde özellikle konu hakkında bilgisi olmayanlar tarafından yaygın şekilde düşük yetenekli ve yüksek yetenekli diller şeklinde yorumlanabilse de ifade ettikleri anlamlar dilin yeteneklerinden bağımsızdır.

Düşük seviyeli bir programlama dili makine koduna yakınlığıyla, yüksek seviye programlama dili makine koduna fazla soyutlama içeren bir programlama dili anlamına gelir. Bir programlama dili makine kodundan ne kadar uzaklaşır ne kadar soyutlama yaparsa o kadar yüksek, ne kadar yakın olursa o kadar düşük seviyeli olarak sınıflandırılır.

Örneğin montaj dili düşük seviyeli bir programlama dilidir. Teknik olarak büyük oranda makine kodu temsilidir. Yalnızca nimoniklerle ve sözdizimsel tasarımla okunabilirliği desteklenmiştir.

C programlama dili yüksek seviyeli dillere örnektir. Makine koduna oldukça soyutlama sunar. Sözdizimsel olarak daha okunabilirdir. Değişkenler donanım üzerinde herhangi bir konumu belirtmeyen soyutlamalardır, nasıl depolanacaklarına derleyici karar verir. Bir dönüş ifadesinin (statement) belirtilmesi yeterlidir, derleyici nasıl bir dönüş olduğuna ve nasıl yapılması gerektiğine kendisi karar vererek derlemeyi gerçekleştirir. Bu ve buna benzer tüm soyutlamalar C dilinin derleyicinin desteklediği tüm mimarilerde herhangi bir değişiklik yapılması gerekmeden derlenebilmesini mümkün hale getirir.

Bu iki sınıflandırmanın yanı sıra çok üst düzey programlama dilleri diye farklı bir sınıflandırma da bulunmaktadır. Bu sınıflandırmada yer alan diller çok yüksek soyutlama seviyelerine sahiptirler, genel olarak betik dosyası yazmakta kullanılan ve benzeri amaca/alana yönelik dillerden oluşmakla birlikte amaçlarına göre sınırlandırılmışlardır.

Bu sınıflandırma 1990'larda genellikle Perl, Python, Visual Basic ve Ruby gibi betik dosyaları oluşturmak amacıyla tercih edilen yüksek seviyeli programlama dilleri için kullanılmıştır.

26. TASARIM MOTİVASYONU

Programlama dillerinin gelişim süreçlerinde birbirlerinden etkilenerek değişime uğradıkları belirtilmişti. Bu etkileşimler sonucu pek çok programlama dili türü meydana gelmiştir. Bir

programlama dilinin geliştirilme motivasyonu genellikle bir konuda/alanda daha iyi bir seçenek oluşturmak ve/veya bulunan programlama dillerindeki bazı sorunları çözmektir.

Modern programlama dillerinin ana kaygıları büyük oranda güvenlik ve performanstır. Programlama dilleri zaman içerisinde makine koduna daha fazla soyutlaştırma sağladığından ilk programlama dilleri doğal olarak makine koduna daha yakındı. Bununla birlikte C gibi yüksek seviye diller de bazı sorunlar içeriyordu. C programlama dilinin sorunlarını çözmek için farklı diller ortaya çıkmış olsa da C programlama dili de B programlama dilinden gelmektedir. Bu programlama dillerinin etkileşimine güzel bir örnektir.

Ana kaygıların güvenlik ve performans olması bu sorunların tam olarak istenen şekilde çözülememesine neden olduğu söylenebilir. Bunun nedeni iki gereksinim için yaygın olarak benimsenen güvenlik amaçlı yaklaşımların saf performans alınmasını zorlaştırmasıdır.

C programlama dilinden devam etmek gerekirse, C bünyesinde minimum bir çalışma zamanı barındırdığından performans konusunda çoğu zaman kayıp yoktur. Bununla birlikte C güvensi bir dildir. Bu güvensizlikler sonucu C programlarında tanımsız davranış (undefined behavior), segmentasyon hataları (segmentation fault), arabellek taşmaları (buffer overflow) ve benzeri sorunlar oldukça yaygın görülmektedir.

Bu sorunları çözmek için bellek güvenliğini sağlamak adına bir programlama dili tasarımında çeşitli yaklaşımlar benimsenebilir. Yaygın yaklaşım çalışma zamanı denetimleridir. Örneğin Go programlama dilinde bir dizinin (array) sınırları dışına çıkmak güvenlik gereği mümkün olmadığından açıkça bir çalışma zamanı hatasına neden olacaktır. C'de bu denetim olmadığından taşma meydana gelecektir.

Bu çalışma zamanı denetimleri güvenlik motivasyonuna hizmet etse de performans motivasyonu için olumsuz etkisi vardır. Örneğin C doğrudan yapılmak isteneni gerçekleştirirken Go içerisinde çalışma zamanında bazı güvenlik önlemleri gereği öncesinde bazı kontroller gerçekleştirilir. Bunlar programın yürütülmesine ek yük olarak yansır.

Tüm bunlara rağmen performans/güvenlik dengesini iyi kurmuş başarılı programlama dili örnekleri bulunduğunu belirtmekte fayda var. Aranan nokta daha fazla dengeli bir orandır.

Bu çalışma zamanı kayıplarını en aza indirmek için zamanla farklı yaklaşımlar da ortaya çıkmıştır. Örneğin Rust programlama dili bu sorunu çözmek için derleme zamanında pek çok güvensizlik sorununun olmayacağının garanti edilmesi için geliştiricisini teşvik eden bir tasarıma sahiptir. Bu yaklaşım sonucu çalışma zamanı kontrolleri diğer dillerden daha az olmakta ve çalışma zamanı üzerindeki iş yükünü azaltmaktadır.

Bellek sorunlarının yanı sıra iyileştirilmeye çalışılan diğer konulardan biri bellek yönetimidir. Bellek yönetimi yazılımcı tarafından gerçekleştirilen programlama dillerinde bellek sızıntısı (memory leak) yaygın bir sorundur. Bu sorunu çözmek için çöp toplayıcı (Garbage Collector) ve referans sayma (Reference Counting) gibi yöntemler geliştirilse de bu konuda performans/güvenlik kaygıları devam etmektedir. Çoğu geliştiricinin tercihleri kendi kriterlerine göre farklılık göstermektedir.

Tüm bunlara ek olarak okunabilirlik gibi estetik tasarım kaygıları da bulunmaktadır. Zamanla sözdizimsel olarak ortaya çıkan farklılıklar nedeniyle bu konuda çeşitli yaklaşımlar sergileyen programlama dilleri bulunmaktadır. Örneğin Go okunabilir olmayı hedefleyen bir dil olarak C dilinin sözdizimine benzerlikler içerirken bazı noktalarda sadeleştirmeye gitmiştir, Python programlama dili ise girintiye dayalı bir sözdizimi benimsemesi nedeniyle daha yalın bir kod sunabilmektedir.

26.1 Programlama Dillerinin Zorluğu

Bir programlama dili için "basit" kelimesini sığ bir açıdan kullanmak mümkün değildir. Genel anlamda bir programlama dili için bu tanımlama yapılmamalıdır. Bunun nedeni temelde dili kullanan geliştiricinin sahip olması gereken beceri seviyesi aynıdır. Bir programlama dilinin var olma amacı gereği geliştiricinin sahip olması gereken temel becerilerden biri algoritma tasarımıdır.

İki farklı dilde birebir aynı algoritmanın geliştirilmesi için gereken efor farklı olabilir. Dilin soyutlama oranı ve tasarımı gereği arada büyük bir geliştirme kolaylığı farkı bulunabilir. Bu nedenle bu açıdan bakıldığında "basit" tanımı kullanılabilir. Lakin bu algoritmanın tasarımının geliştirici tarafından gerçekleştirilmesi durumunda dilden bağımsız olarak kişinin kendi becerilerini ilgilendiren bir durum olduğundan bu açıdan programlama dili "basit" olarak nitelendirilemez.

Örnek vermek gerekirse, Python geliştirme deneyimi olarak montaj dilinden daha basittir ve gerektirdiği efor daha azdır ancak algoritma tasarımı becerisi çok düşük olan bir geliştiricinin Python kullanması durumunda sihirli - bilime aykırı - bir şekilde oldukça iyi algoritma tasarımları yapabilmesi mümkün olmadığından bu iki dil algoritma tasarlama konusunda aynı derecede zordur.

Yukarıda açıklanan perspektiften değerlendirildiğinde çıkarım yolu ile bu öznel sonuca ulaşılabilir:

Programlama dili kullanarak algoritma öğrenmeyi amaçlayan ve bilgisayar bilimi konusunda bilgisi olmayan bir öğrencinin Python kullanması durumunda daha rahat bir geliştirme deneyimine sahip olduğundan zorlanmadan hızlıca tasarladığı algoritmaları uygulamaya geçebilir. Montaj dili kullanması durumunda tasarladığı algoritmaları uygularken sahip olması gereken teknik bilgi gereksinimleri çok yüksek olduğundan yeni öğrenciler için sancılı bir süreçtir lakin montaj dilinde soyutlama çok az olduğundan öğrencinin tasarlamış olduğu algoritmayı oldukça detaylı bir şekilde uygulaması gerektiğinden kavrama ve anlama becerileriyle orantılı olarak bu konuda daha hızlı ilerleme kaydetmesi teorik olarak mümkündür.

27. BELLEK YÖNETİMİ (MEMORY MANAGEMENT)

Bellek yönetimi sistemin ana belleğinin gereksinimlere göre tahsis edilmesi, paylaştırılması ve yönetilmesidir. İşletim sistemleri üzerinde çalışan yazılımların bellek tahsislerini işletim sistemi gerçekleştirir. Bellek yönetimini gerçekleştiren işletim sistemi parçası ise bellek yöneticisi (Memory Manager) olarak adlandırılır.

Bellek tahsis etmek maliyetli bir işlemdir. Uygulanan bellek yöneticisine göre verimlilik ve performansı değişiklik gösterse de performansı önemli ölçüde etkileyebilmektedir.

27.1 Manuel Bellek Yönetimi (Manual Memory Management)

Bir yazılım çalışma zamanında bellek tahsisi (allocation) gerçekleştirmek istediğinde çekirdek (Kernel) ile iletişim halindedir. Tahsis yöntemi farklılık göstermekle birlikte işletim sisteminin bellek yöneticisi tarafından ayrılan tahsisin konumu önceden bilinmediğinden bir referans (işaretçi / pointer) ile erişilir.

Program tarafından tahsis edilen bu alanlar programın kendi içerisinde sergilediği bellek yönetimine kalmaktadır. Program tahsis ettiği alanları artık ihtiyacı kalmadığı durumlarda

serbest bırakarak (deallocation) belleği geri kazandırır ve belleğin ilgili bölümleri artık tahsis edilebilir hale gelir. Bu tahsisler geliştirici tarafından manuel bir şekilde gerçekleştirilir ve serbest bırakılır. Geliştirici ne zaman tahsis etmesi ve serbest bırakması gerektiğine kendisi karar vermelidir.

Geliştiricilerin bellek yönetimini verimli ve doğru bir şekilde gerçekleştirmemeleri çeşitli bellek sorunlarına yol açabilir. Örneğin C programlama dilinde artık kullanılmayan bu nedenle serbest bırakılması gereken tahsisler serbest bırakılmadığında bellek sızıntısına neden olabilir. Bu nedenle geliştirici tahsisleri sürekli olarak gözetmeli ve doğru konumlarda atık serbest bırakmaya özen göstermelidir.

27.2 Otomatik Bellek Yönetimi (Automatic Memory Management)

Pek çok modern programlama dilinde otomatik bellek yönetimi yaklaşımına rastlanmaktadır. Çalışma zamanında programın bellek yönetiminin kendi içerisinde otomatik bir şekilde gerçekleştirilmesine dayanır. Bu yaklaşımın yaygın örneklerinde geliştiricinin bellek yönetimini düşünmesi gerekmemektedir.

Otomatik bellek yönetimi içerisindeki tekniklerin mantığına dayanan ancak birebir aynı olmayan uygulamaları (implementation) bulunmaktadır.

Çöp toplama program tarafından tahsis edilen ve artık kullanılamayan/erişilemeyen tahsislerin serbest bırakılmasına dayanan bir otomatik bellek yönetimi biçimidir. Çöp toplama bir programın toplam çalışma süresinin önemli bir bölümünü oluşturabilir ve performansa olumsuz etki edebilir.

Çöp toplamanın yürütülmesi öngörülemez olabilir, bu da yapılan performans testlerinde çöp toplama kaynaklı değişken sonuçlar elde edilmesine yol açabilir. Bununla birlikte programda dağılmış olarak önemli duraklamalar meydana gelebilir, bu duraklamalar çöp toplayıcı yürütüldüğü esnasında programın yürütülmesinin kilitlenmesinden kaynaklanmaktadır.

Çöp toplama kullanan dillere çöp toplanmış diller (Garbage Collected Languages) denir. Bir dilin ana tasarımı birden fazla bellek yönetim biçimini barındırabilmektedir. Örneğin bazı derleyiciler çöp toplama ve manuel bellek yönetimi arasında geliştiriciye bir seçim yapma imkânı sunmaktadır.

Bir dilin çöp toplama sergiliyor olması için çöp toplamanın dilin ana tasarımına dahil olması ya da bir derleyici özelliği olması gerekmemektedir. Örneğin manuel bellek yönetimi sunan C ve C++ programlama dilleri için çeşitli çöp toplama uygulamaları (implementation) (Bkz. Boehm Garbage Collector) geliştirilmiştir.

27.3 Çöp Toplama Takibi (Tracing Garbage Collection)

Çöp toplama tekniği yaygın olarak çöp toplama takibi (Tracing Garbage Collection) ile gerçekleştirilir. Bu o kadar yaygın bir uygulamadır ki genellikle çöp toplama dendiğinde çöp toplama takibi kastedilmektedir.

Çöp toplama izleme program içerisinde gerçekleştirilen tahsislerin kullanımlarını analiz eden ve kullanımda olmayan tahsisleri, bu tahsislere çöp (garbage) denir, tespit edip serbest bırakan bir çöp toplama tekniğidir. Bu izlemenin gerçekleştirilmesi için bazı izleme verilerinin çalışma zamanında depolanması gerekmektedir, bu da bir ek yük olarak yansımaktadır. Bu izlemenin gerçekleştirilmesi için kullanılan pek çok algoritma bulunmaktadır.

Çöp toplama takibi bir derleyici özelliği olabilir ve derleyici tarafından çalışma zamanının bir parçası olarak bulunabilir. Örneğin C# ve Java programlama dillerinde çöp toplama takibi kullanılmaktadır ve bu yaklaşım çalışma zamanında tamamen otomatik bir şekilde yürütülür. Oluşturulan ilgili nesneler otomatik olarak çöp toplama takibine dahil edilir.

Bir diğer yöntemde ise çöp toplama takibine dahil edilecek tahsislerin geliştirici tarafından bildirilmesi gerekir.

Nesne sonuçlandırma (object finalization) zamanlamasında çöp toplama takibi deterministik değildir. Çöp toplamaya tabii olabilecek duruma gelen her nesne eninde sonunda serbest bırakılacaktır lakin bunun gerçekleşme durumu öngörülemez. Çöp toplamanın ne zaman gerçekleşeceğine ya da gerçekleşeceğine dair hiçbir garanti yoktur.

Çöp toplama işlemi programın asıl algoritmasından kaynaklanmayan gecikmelere ve duraklamalara neden olup yürütme zamanının oldukça değişken olmasına yol açabilir. Deterministik olmayan yapısı nedeniyle yeni bir nesne tahsis etmek bazen yalnızca nesnenin tahsis edilmesi şeklinde gerçekleşirken bazen uzun bir çöp toplama döngüsünün yürütülmesini tetikleyebilir.

27.4 Referans Sayma (Reference Counting)

Referans sayma ilgili tahsise yapılan her bir referansın sayılması ve referansların hepsinin sıfıra ulaşması durumunda tahsisin serbest bırakılmasına dayanır. Bu teknikte her referans kendisine referans eden bir sayma verisi bulundurur. Bu sayma verisi referans her kopyalandığında artırılır ve kopya her yok edildiğinde azaltılır. Referans sayma verisi sıfıra ulaştığında tahsisi kullanan bir program noktası olmadığından serbest bırakılır.

Referans sayma çöp toplama takibinde olduğu gibi doğrudan derleyici tarafından otomatik bir şekilde gerçekleştirilebilir ya da geliştirici tarafından ilgili bölümlerde referans sayma gerçekleştirilebilir.

Referans sayma deterministik bir çöp toplama performansı sergiler. Bunun nedeni program yürütülürken bir referansın oluşturulma, kopyalama ve serbest bırakılma noktaları her zaman sabit olduğundan referansın serbest bırakılma işlemi her zaman aynı algoritma adımında gerçekleşir.

Referans sayma için kullanılan sayma verisi çalışma zamanında ek bellek ayak izi anlamına gelir. Bununla birlikte referans sayma işlemi algoritmaya göre önemli miktarda çalışma zamanı yükü oluşturabilir.

Referans saymanın atomiklik (atomicity) sergilemesi gerekmektedir. Çok iş parçacıklı bir ortamda bir referansın aynı anda kopyalanması durumunda referans sayma doğru bir şekilde gerçekleşmeyebilir, bunu önlemek için referans sayma atomik bir şekilde gerçekleştirilmelidir. Bu atomiklik çok iş parçacıklı bir senaryo olmadığı durumlarda dahi küçük de olsa bir çalışma zamanı ek yükü oluşturur.

Referansın sayma verisi üstünde işlem gerektirecek herhangi bir durumun iş parçacıkları tarafından çok fazla tekrarlanması bu atomikliğin programın toplam çalışma süresinin önemli bir bölümünü oluşturmasına neden olabilir.

Diğer bir sorun döngüselliktir. Örneğin iki referansın birbirlerine referans vermesi durumunda referans sayısının asla sıfırlanmayı bir döngü oluşturulabilir. Bunu önlemek için çeşitli döngü algılama algoritmaları vardır.

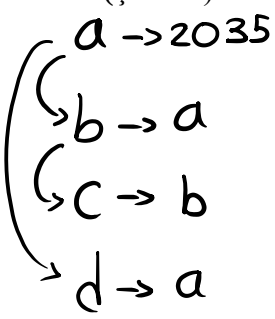
Referans sayma çeşitli şekillerde uygulanabilir ve bu uygulamaların kendilerine has dezavantajları olabilir. Bazı referans sayma uygulamalarında referans sayma verisi sıfırlandığında tahsisin direkt serbest bırakılması yerine serbest bırakılmayı bekleyen

tahsislerin bulunduğu bir bölümde listelenebilir ve periyodik olarak bu listedeki tahsisler serbest bırakılabilir.

Bir referans sayma verisinin sıfıra ulaşılması durumunda yok edilen tahsis de bir referans ise, bu sıralı bir referans sayma başlatabilir ve belirli durumlarda pek çok referansın sayma verisi sıfıra ulaştığından sıralı ve uzun serbest bırakmanın tetiklenmesine yol açabilir.

Şemalı anlatım:

(Şekil 7)



Şekil 7'de ilişkiler açıklanmıştır. a değişkeni 2035 olarak temsil edilen bir bellek adresine işaret eden tahsistir. a değişkeni aynı zamanda bir referans sayma gerçekleştirmektedir.

b ve d değişkeni a değişkeni kullanılarak oluşturulur. Bu durumda a, b ve d değişkenleri aynı alana işaret eden referanslardır.

c değişkeni b değişkeni kullanılarak oluşturulmuştur. c değişkeni de artık a değişkeninin işaret ettiği 2035 adresli tahsise işaret etmektedir. Bunun nedeni b değişkeninin a değişkenine bir referans olmasıdır.

Şekil 7'de a değişkeni 2035 olarak temsil edilen bir bellek adresine işaret eden tahsistir. a değişkeni aynı zamanda bir referans sayma gerçekleştirmektedir. a değişkeninin toplam referans sayısı kendisi de dahil olmakla birlikte 4'dür. A değişkeni ana referans olmasına rağmen yok edildiğinde tahsisi serbest bırakmaz. Bunun nedeni referans sayısının halen 3 olmasıdır. Tahsisin serbest bırakılmasının tek yolu a, b, c ve d değişkenlerinin yok edilmesi ve referans sayma verisinin sıfıra ulaşmasıdır.

27.5 Sahiplik (Ownership)

Sahiplik Rust programlama dilinin tasarımında benimsenen bir bellek yönetimi tekniğidir. Bu teknik çalışma zamanında çöp toplama amacıyla ek bellek ayak izi oluşturmaz, deterministik çalışır ve tahsisi serbest bırakmak dışında bir çalışma zamanı maliyeti yoktur. Manuel bellek yönetimi ile tamamen aynı çalışma zamanı maliyetine sahiptir.

Sahiplik kuralları derleyici tarafından kontrol edilir ve yönetilir. Bu da bellek yönetimi kurallarının ihlal edilmesi sonucunda derleyici tarafından hata belgelenmesine yol açacaktır.

Sahipliğin kuralları:

- Bir tahsisin her zaman bir sahibi olmalıdır.
- Bir tahsisin yalnızca tek bir sahibi olabilir.
- Sahibi olmayan bir tahsis kapsam dışına çıktığında serbest bırakılır.

Bu kuralları irdelemek gerekirse en önemli temel kural bir tahsisin her zaman bir sahibinin olması gerektiğidir. Bu kural sahiplik yaklaşımın kök kuralıdır. Sahibi olan bir tahsis farklı bir sahibe atandığında önceki sahip sahipliğini kaybeder zira aynı anda yalnızca tek bir sahip olma kuralı vardır.

Derleyicinin hata belgeleyeceği yerlerden birisi sahipliğin yanlış kullanımıdır. Sahipliğini kaybetmiş bir değişkeni kullanmak açıkça derleme hatasına neden olacaktır. Bu derleme zamanında gerçekleştirilen bir bellek güvenliği yaklaşımıdır.

Buraya kadar bir C programında yapılan manuel bellek yönetiminden çok farklı değildir. Bir tahsis gerçekleştirildi ve işaretçi ile bu tahsisin konumuna sahipsiniz. Bu işaretçiyi farklı bir işaretçiye atayabilirsiniz ve bu bir sorun teşkil etmez. C programının tahsisin ne zaman boşaltılacağını anlamak gibi bir amacı yoktur. Bu nedenle siz serbest bırakana kadar istediğiniz gibi kullanmakta özgürsünüz lakin sonrasında yapılan bellek yönetimi hataları sonucunda çeşitli bellek sorunları ile karşılaşabilirsiniz.

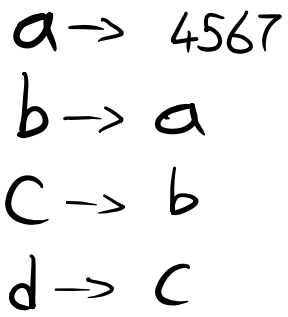
Rust, C programlama dilinin aksine derleme zamanında tahsisin nerede gerçekleşeceğini bilmek ister. Bunun için yalnızca tek bir sahip olmalıdır. O sahip izlenebilir ve yok edildiği durumda tahsisin serbest bırakılması gerektiği anlaşılmış olur. Tam olarak bu nedenden bir tahsisin tek sahibi vardır.

Sahipliği elinde tutan bir örnek yok edildiğinde, eğer halen elinde bulunduruyorsa doğal olarak başka bir noktada farklı bir sahip olamayacağından tahsisin serbest bırakılması gerçekleşir.

Bu yaklaşım manuel bellek yönetiminin derleme zamanında güvenli olması için teşvik edilen bir türevi olarak düşünülebilir.

Şemalı anlatım:

(Şekil 8)



Şekil 8'de akış açıklanmıştır. a değişkeni 4576 olarak temsil edilen bir bellek adresine işaret eden tahsistir. a değişkeni sahiplik yaklaşımını sergilemektedir.

a değişkeni b değişkenine atandığında yeni sahip b değişkeni olacaktır. Bu durumda a değişkeni kullanılamaz durumdadır, kullanılması açıkça bir derleyici hatasıdır.

b değişkeninin c değişkenine ve c değişkeninin de d değişkenine atanması yukarıda açıklanan davranışı sergiler.

Son sahip olan d, yok edildiğinde serbest de serbest bırakılır zira tek bir sahip vardır ve o da yok edildiğinden tahsis ulaşılamaz olmuştur.

Kod örneği (Rust):

```rust
fn main() {
    let a = String::from("PLTR");
    let b = a;
    let c = b;
    let d = c;
    println!("{}", d);
}
```

28. PROGRAMLAMA PARADİGMALARI (PROGRAMMING PARADIGMS)

Programlama paradigmaları programlama dillerini türlerine göre sınıflandırmanın bir yoludur. Programlama dilleri birden fazla paradigmaya sahip olabilirler. Bir paradigma temel olarak programlamanın nasıl yapıldığını tanımlar.

28.1 Zorunlu Programlama (Imperative Programming)

Zorunlu programlama paradigması en yaygın paradigmalardan birisidir ve en eski paradigma olarak kabul edilir. Programlamanın tıpkı bir algoritma gösteriminde olduğu adım adım gerçekleştirilmesine dayanır. Başka bir deyimle kodun adım adım akışının bildirilmesidir. Bunun görsel halinin akış şemaları olduğu söylenebilir.

Zorunlu programlamanın yazımı bu şekildedir:

- Komutlar sırayla yürütülmelidir
- Komutlar programın kaynak kodunda yazılmıştır

İlk yazım kriterinde görüldüğü üzere komutlar sırayla yürütülmelidir. Diğer kritere de komutların programın kaynak kodunda yazılma gerekçesidir. Bunun özellikle belirtilme nedeni eninde sonunda çalışan makine kodu doğal olarak zorunlu programlama sergileyeceğinden bu kabul edilmez. Bu kriterin sağlanması için programın geliştirici tarafından yazılan kaynak kodunun zorunlu paradigma sergilemesi gereklidir.

28.2 Bildirimsel Programlama (Declarative Programming)

Bildirimsel programlama zorunlu programlamanın aksine komutların adım adım yazılmasını gerektirmez. Zorunlu programlama amacın nasıl gerçekleştiğine odaklanırken bildirimsel programlama sonuca odaklanmaktadır. Geliştirici bir programın nasıl yürütülmesi gerektiğini değil ilgili bölümde nasıl bir sonuç elde edilmesi gerektiğini tanımlar.

Zorunlu programlamada adımların açıkça uygulanması gerekirken bildirimsel programlamada böyle bir zorunluluk yoktur. Kaynak kodun istenen sonuca ulaşması bu paradigmada yeterlidir. Bildirimsel programlamada kullanılana algoritmalar kendi içerisinde zorunlu programlamayı barındırıyor olsa da soyutlama sağlarlar. Geliştirici kaynak kodda ne yapılması gerektiğini söyler, nasıl yapılması gerektiği derleyiciye/yorumlayıcıya/çeviriciye kalmıştır.

Örneğin bir dizinin içerisindeki tamsayıların toplanması istenmektedir. Bunu "bu dizi içerisindeki tamsayıların toplamı" şeklinde sonuç odaklı bildirilmesi ile gerçekleştirmek mümkün olmalıdır. Zorunlu programlamada bu toplama algoritmasını geliştiricinin yazması gerekirken bildirimsel programlamada yalnızca ne istendiği söylendi ve nasıl yapılacağına karışılmamıştır, nasıl yapılacağı dilin derleyicisine/yorumlayıcısına/çeviricisine bırakılmıştır.

Örneğin SQL dili bildirimsel bir tasarıma sahiptir. SQL sorgusu yazılırken istenenler belirtilir lakin nasıl yapılacağı sorguyu yazan kişi tarafından umursanmaz. SQL yorumlayıcısı gördüğü ifadeleri kendi uygulama biçimine göre değerlendirir.

Yalnızca XYZ tablosunun ABC sütununa göre azalan (descending) bir şekilde sıralanmasını içeren SQL sorgusunu düşünelim. Burada XYZ tablosunda ABC sütununa göre azalan bir sıralama yapılması istenmiştir ancak nasıl yapılacağı umursanmamıştır, SQL yorumlayıcı sıralama algoritmasını kendisi gerçekleştirir ve bu herhangi bir algoritma olabilir, önemli olan istenilen sonucun elde edilmesidir.

28.3 Fonksiyonel Programlama (Functional Programming)

Fonksiyonel programlama bir bildirimsel programlama paradigmasıdır. Fonksiyonel programlamada fonksiyonlar bir değişkene atanabilir, argüman olarak kullanılabilir ve bir fonksiyondan geri döndürülebilir. Diğer bir nokta saf fonksiyon (pure function) anlayışıdır. Saf işlevler girdilere göre çalışan ve her zaman aynı girdilerle aynı sonuçlar üreten fonksiyonlardır (Bkz. Determinizm).

Ayrıca saf fonksiyonlar hiçbir yan etki sağlamazlar. Yani kendi kapsamları dışında yer alan hiçbir noktada değişiklik gerçekleştirmezler. Çıktıları yalnızca girdilerine bağlı olmalıdır.

28.4 Prosedürel Programlama (Procedural Programming)

Prosedürel programlama zorunlu programlamanın bir türevidir. Kaynak koda altyordamlar (subroutine) ekler, prosedür olarak da adlandırılır. Prosedürel programlama kod tekrarını azaltmak, modülerliği yükseltmek ve kodun bakımını kolaylaştırmak için geçiştiriciyi programı fonksiyonlara ayırmaya teşvik eder.

Fonksiyonlarda zorunlu programlama uygulanmaya devam eder. Yani bir fonksiyon bildirimsel programlama gibi davranmaz. Yalnızca belirli komut sırasını temsil eden fonksiyonlardır. Fonksiyonların içerisindeki algoritma yine zorunlu programlama ile geliştirilmek zorundadır. Herhangi bir prosedür kendisi de dahil olmak üzere (Bkz. Özyineleme / Recursion) programın herhangi bir adımında çağırılabilir.

Daha genel bir anlatımda prosedürel programlama kodun belirli amaçlara hizmet eden bölümlerinin prosedürlere ayrılmasıdır. Daha sonra bu prosedürlere çağrılar gerçekleştirilerek aynı sıralı algoritma yürütülebilir. Bu prosedürler farklı noktalarda da çağırılabileceklerinden modülerliği artırmaya fayda sağlarlar.

28.5 Nesneye Yönelik Programlama (Object Oriented Programming - OOP)

Nesneye yönelik programlama paradigması her bir amaç için nesneler oluşturmaya dayanır. Örneğin bir dikdörtgeni temsil etmek amacıyla bir nesne tasarlanabilir. Bu nesne içerisinde veriler ve kodlar barındırabilir. Örneğin dikdörtgen nesnesi uzun ve kısa kenarı veri olarak barındırabilir, alanını hesaplayıp döndüren bir fonksiyona sahip olabilir.

Bir nesne tasarımının örneği oluşturulduğunda her bir örnek farklı bir temsil anlamına gelmektedir. Örneğin iki adet dikdörtgen nesnesinin iki farklı dikdörtgeni temsil ettiği kabul edilir. Bir nesne tasarımı dikdörtgen, daire, araba gibi olguları temsil etmekte ya da belirli bir amaca hizmet eden/işi yapan aracı temsil etmekte kullanılabilirler.

OOP paradigmasını kullanan programlar genellikle nesnelerin birbirleri ile ilişkisi ve etkileşimi olacak şekilde tasarlanırlar.

Bir nesne tasarımını ifade etmek için sınıflar kullanılır. Nesneler ise sınıfların üretilmiş örnekleri (instance) olarak kabul edilir. Sınıflar kalıtım (inheritance) sergileyebilir ya da soyutlama (abstract type) için kullanılabilir.

28.6 Yapısal Programlama (Structured Programming)

Yapısal programlama paradigması kaynak kodun içerisinde kontrol yapıları, altyordamlar ve kapsamlar ile programın kalitesini, anlaşılırlığını ve geliştirme süresini iyileştirmeyi hedeflemektedir.

Her bir yapının kapsamı (Bkz. blok) sahibi olduğu söylenebilir. Bir blok olması bir kapsam olması demektir. Bloklar içerisinde o bloğa özgü talimatlar bulunabilir ve bu blok kontrol yapıları ile belirli koşullarda yürütülebilir ya da yinelenebilir.

Örneğin pek çok programlama dilinde bulunan if-else deyimleri, while, for, foreach iterasyonları yapısal programlama dahilindedir.

28.7 Olaya Dayalı Programlama (Event-Driven Programming)

Olaya dayalı programlama program akışında kullanıcı eylemleri (tuşa basma, fare tıklamaları vb.), sensör çıktıları ve diğer programlar tarafından gelen sinyallerle tetiklenen olayların tasarlandığı bir programlama algoritmasıdır.

Bu paradigma kullanıcı grafik arayüzü (Graphical User Interface - GUI) ve girdi çıktılara göre belirli olayların gerçekleştirilmesine odaklanan programlarda yaygın olarak kullanılan paradigmadır.

Paradigma dil içerisinde uygulanabilir ya da doğrudan dilin tasarımına dahil olabilir. Örneğin Microsoft .NET platformundaki diller tasarımlarında gözlemci tasarım desenini (Observer Design Pattern) dahili olarak barındırmakta ve bunu bir dil özelliği olarak sunmaktadır. Aynı tasarım deseni dile entegre olmaksızın tasarım deseninin dil içerisinde uygulanmasıyla kullanılabilir. Bu paradigmanın bir diğer örneği yaygın bir şekilde JavaScript kodlarında kullanılmaktadır. JavaScript olayları da aynı şekilde gözlemci tasarım desenini uygulamaktadır.

28.8 Veriye Dayalı Programlama (Data-Driven Programming)

Veriye dayalı programlama olaya dayalı programlamaya benzemektedir, iki paradigma da kalıp eşleştirme ve sonuç işleme olarak yapılandırılmıştır. Veriye dayalı programlamada programın akışını veri belirlemektedir, dolayısıyla programın akışı verinin kendisi tarafından kontrol edilir. Programa akışının yönlendirilmesi için farklı veri kümeleri sunularak akış yönlendirilir.

Örneğin bir mesajlaşma uygulamasında belirli yasaklı kelimeler bulunduğunu kabul edelim. Bu yasaklı kelimeler bir veri tabanında tutulmaktadır. Veri tabanına kelimeler eklenebilir ve çıkarılabilir. Bu yasaklı kelimeleri kontrol eden ve eğer girdi olarak aldığı cümlede bu kelimelerden birini tespit ederse hata bildiren bir program düşünelim.

Programın algoritması bir kez yürütülmeye başladıktan sonra davranışı veriler tarafından yönetilecektir. Örneğin bir cümle girdi olarak alınmış ve yasaklı kelime barındırmayan cümle olarak değerlendirilmiş olsun. Bundan sonra veri tabanı güncellenerek bazı yeni yasaklı kelimeler edinmiş olsun, bu eklenen kelimelerin bazıları örnek girdi cümlesinde yer alan kelimelerdir. Bu durumda aynı cümle girdi olarak alındığında veri tabanında yasaklı kelimelerle eşleştiğinden hata bildirilecektir. Programın akışı bu şekilde veriye dayalı olur, veriye göre akışı şekillenir.

Buradaki nokta kaynak kodun değiştirilmesine gerek olmamasıdır. Örneğin yasaklı kelimelerin bir listesi kaynak kodun içerisinde tutuluyor olsaydı herhangi bir yasaklı kelime kaldırılması ya da eklenmesi gerektiğinde kaynak kodun değiştirilmesi gerekirdi.

29. TURING BÜTÜNLÜĞÜ (TURING COMPLETENESS)

Hesaplama teorisinde gerçek ya da sanal bilgisayarlar, programlama dilleri ve diğer mantıksal sistemlerin evrensel bir Turing makinesi tarafından gerçekleştirebilecek hesaplama gücüne sahip olması durumunda Turing Tam (Turing Complete) olarak nitelendirilir.

Başka bir değişle bahse konu işlemci ya da programlama dili Turing Tam olarak nitelendirilebilmek için bir Turing makinesini simüle edebiliyor olmalıdır.

Turing makinesi, bir şeritteki sembolleri kurallar tablosuna göre işleyen soyut bir makineyi tanımlayan matematiksel hesaplama modelidir. Bir Turing makinesinin yapabilecekleri okuma ve yazma, bandı ileri ve geri sarma şeklindedir. Bu denli basit olmasına rağmen bir Turing makinesi herhangi bir bilgisayar algoritmasının mantığını simüle etmek için uyarlanabilir.

Turing makineleri ilk olarak Alan Turing tarafından tanımlanmıştır.

30. TEKNİK BORÇ (TECHNICAL DEBT)

Teknik borç bir yazılım geliştirme sürecinde zaman alacak lakin iyi bir yaklaşımı benimsemek yerine o an istenileni daha kısa zamanda yapabilmenin mümkün olduğu bir yaklaşımın tercih edilmesi durumunun getirdiği ek çalışma maliyeti olarak tanımlanır. Tasarım borcu veya kod borcu olarak da anılır.

Teknik borcun birikmesi ve çok fazla olması bir yazılım geliştirme sürecini fazlasıyla zorlaştırabilir, geciktirebilir veya engelleyebilir. Teknik borç mutlaka kötü bir şey değildir, bazı durumlarda (özellikle belirli bir zaman dilinde bir yazılımın geliştirilmesi gerektiğinde) geliştirme sürecini hızlandırmak için bilinçli olarak daha sonra ödenmek üzere teknik borç oluşturulabilir.

Teknik borç oluşturan bazı durumlar yazılım belgelerinin (software documentation) ertelenmesi, kod tekrarlarının giderilmemesi, sürüm yönetimi ve derleme araçları gibi teknik altyapı eksiklikleri, derleyici uyarıları ve statik kod analizi sonuçlarının değerlendirilmemesi, büyük veya aşırı karmaşık kod tasarımının iyileştirilmemesi şeklinde sıralanabilir.

Bunlara ek olarak kod sahipliği de önem teşkil etmektedir. Örneğin bir projede kod sahipliğinin olmadığı kütüphane ve benzeri araçlar çok fazla kullanıldığında programın performansı büyük oranda bu araçların performansına bağımlı hale gelebilir. Bir sorunun çözümü ya da optimizasyon için kodun sahipliği başka geliştiricilere ait olduğundan bu dışa bağımlılık ve teknik borç olarak dönecektir.

Açık kaynak projelerin genellikle katkıya açık olmasına rağmen katkıların her zaman kabul almayacak olması nedeniyle kod sahipliği eksikliği açık kaynak araçlarda da sorun teşkil edebilir. Yine de genellikle bu projenin çatallanması ve orijinali yerine çatallanan sürümünün kullanılmasıyla çözülebilecek bir problemdir lakin teknik olarak bu da bir teknik borç ödemesidir.

31. YERLEŞİK İŞLEVLER (BUILT-IN FUNCTIONS)

Yerleşik işlevler dilin tasarımına dahil olan işlevlerdir. Bu işlevler derleyici/yorumlayıcı tarafından uygulanır. Genellikle daha önceden makine koduna derlemeleri gerçekleştiğinden üretecekleri kod derleme/yorumlama zamanında belirlidir.

Yerleşik işlevler yaygın olarak satır içi derleme olarak değerlendirildiklerinden çalışma zamanında bir işlevin çağırılma maliyetlerine sahip değillerdir. Elbette bu söylem yaygın olarak derlenen bir dil tasarımı için geçerli olsa da bazı ara dile derlenen yorumlanmış diller yerleşik işlevleri de ara dile derleme davranışı sergiliyor olabilir, bu durumda satır içi derleme şeklinde ara kod üretilmesi söz konusu olabilir.

Buna ek olarak yorumlanan dillerde yerleşik işlevin gerçekleştireceği komutlar çoğu zaman yorumlayıcının kaynak kodunun bir parçası olduğundan çalışma zamanında aynı algoritmanın yorumlanmasına kıyasla büyük oranda performans sergileyebilirler. Bu nedenle yerleşik işlevlerin ara dile dahil edilen yorumlama kodlarından oluşması performansı olumsuz etkileyebilir.

Yerleşik işlevler genellikle yaygın olarak kullanılan ve performans gerektiren ya da uygulanması dil tasarımı içerisinde mümkün olmayan işlevlerin tanımlanması amacıyla dil tasarımına dahil edilirler.

32. VERİ TİPİ GRUPLANDIRMASI (DATA TYPE GROUPS)

32.1 İlkel Veri Tipleri (Primitive Data Types)

İlkel türler dil tasarımının doğrudan sahip olduğu ve derleyici/yorumlayıcı tarafından uygulandığı veri tiplerinin yer aldığı gruplandırmadır. Yerleşik veri tipleri (built-in data types) olarak da anılırlar. Bu veri tipleri belirli bir türdeki verilerin temsil edilmesinde kullanılırlar.

Karakteristik özellikleri:

- Sabit boyutludur
- Derleyici ya da yorumlayıcı tarafından tanımlanmıştır
- Bellekte tek bir alan tahsisi gerçekleştirir
- Metotlar ve benzeri sahip değildir, saf değerdir

İlkel veri türleri fizikteki bir atoma benzetilebilir. Atomlar bilinen evrendeki maddenin kimyasal ve fiziksel niteliklerini taşıyan daha fazla parçalanamayan en küçük yapı taşlarıdır. İlkel veri türleri de daha fazla parçalanamazlar, en küçük formlarındadır. Atomlar gibi ilkel veri türleri bir araya getirilerek karmaşık yapılar elde edilebilir.

Yaygın ilkel tipler: int, float, byte, char

32.2 İlkel Olmayan Veri Tipleri (Non-Primitive Data Types)

Geliştirici tarafından tanımlanırlar ve veri depolayan bellek konumlarına atıfta bulunurlar. Birden fazla ilkel tipe ev sahipliği yapabilir ya da tek bir ilken tipin veri koleksiyonunu içerebilirler.

Karakteristik özellikleri:

- Geliştirici tanımlıdır
- Değişken boyutta olabilir
- Metot ve benzeri sahip olabilirler

Yaygın ilkel olmayan tipler: diziler, karakter dizileri (string), bağlı liste, kuyruk (queue)

33. STANDART KİTAPLIK (STANDARD LIBRARY)

Standart kitaplık dil tasarımının bir parçası olarak değerlendirilebilir. Dilin pek çok yaygın sorunu çözmek için sunduğu kitaplıklar bütünüdür. Genellikle bu kitaplıklarda yer alan uygulamalar dilin kendisi ile geliştirilir. Bu nedenle standart kitaplığa sahip olan bir dilin standart kitaplığı dil yeterli olgunluğu ulaştıktan sonra geliştirilir.

Standart kitaplık bazı durumlarda tamamen yerleşik işlevlerden oluşabilir ya da bazı işlevleri yerleşik olarak tanımlanabilir. Bu genellikle yerleşik işlevlerin uygulanma motivasyonu ile aynı motivasyonda uygulanır.

Standart kitaplıklar içerisinde yer alan tanımlar bazen dilin tasarımının doğrudan parçası olan bir nokta için bağımlılık teşkil edebilir. Örneğin C# programlama dilinde yer alan LINQ sözdizimini kullanmak için `System.Linq` kitaplığı kaynak koda dahil edilmelidir. Bunun nedeni sözdizimin aslında bir sözdizimi şekeri olması ve `System.Linq` içerisindeki işlevleri kullanmasıdır. Bununla birlikte `string` veri tipi de `System.String` kitaplığına yapılan bir başvurudur.

Standart kitaplıkların kullanılması da teknik olarak bakıldığında teknik borç olarak nitelendirilebilirler. Ancak burada standart kitaplıkların bazı önemli avantajları bulunmaktadır. Ciddi programlama dili geliştiricileri tasarlamış oldukları dilde genel olarak daha önce de bahsedildiği gibi performans dengesini korumaya çalıştıklarından standart kitaplığın en iyi şekilde optimize edilmesi ve tespit edilen hatalarının çözülmesi için ciddi çaba harcayacaklardır.

Bununla birlikte standart kitaplık bir amacın dil içerisindeki standart bir uygulamasını temsil ettiğinden diğer geliştiriciler tarafından daha okunabilir ve daha standardize edilmiş bir kod geliştirilmesini sağlar. Standart kitaplığın işlevlerinin gerekmedikçe baştan yazılması bu bağlamda faydalıdır.

34. DÖNGÜSEL KARMAŞIKLIK (CYCLOMATIC COMPLEXITY)

Döngüsel karmaşıklık bir algoritmanın karmaşıklığını hesaplamak için kullanılan bir metriktir. Döngüsel karmaşıklık kabaca bir programın kaynak kodunun doğrusal olarak bağımsız yollarının sayısının nicel bir ölçüsüdür. 1976 yılında Thomas J. McCabe, Sr. tarafından geliştirilmiştir.

Kaynak kod karmaşıklığı özellikle yapısal programlama paradigması ortaya çıktıktan sonra dallanan bloklar ve kontrol akışları nedeniyle oldukça karmaşık hale gelmiştir. Bu nedenle bu metrik bir kaynak kodun döngüsel karmaşıklığını hesaplayarak fazla karmaşık kabul edilebilecek akışların tespit edilmesi amacıyla kullanılmaktadır.

Döngüsel karmaşıklığın çok yüksek olması bir sorun olarak kabul edilir. Daha düşük karmaşıklığa sahip programların anlaşılması daha kolay ve değiştirilmesi daha az risk teşkil etmektedir.

34.1 Döngüsel Karmaşıklığın Hesaplanması

Döngüsel karmaşıklığın hesaplanmasında akış kontrolü grafiğinden faydalanılır. Grafikteki düğümler bir programın en küçük komut grubunu (Bkz. altyordam) temsil eder ve içerisindeki yönlendirilmiş bir kenar, iki düğümün birbirine bağlanmasıyla temsil edilir.

Bir kaynak kodun döngüsel karmaşıklığı en az 1 olabilir, karmaşıklığın 1 olması durumunda kodda tek bir yol vardır. Kaynak kod bir kontrol akışı içeriyorsa, örneğin if koşulu, döngüsel karmaşıklık 2 olacaktır zira koşulun karşılanma ve karşılanmama durumuna göre iki farklı algoritma yolu ortaya çıkacaktır.

Döngüsel karmaşıklık matematiksel olarak bu şekilde tanımlanır:

$M = E – N + 2P$

E = kontrol akışındaki kenar sayısı (edge)

N = kontrol akışındaki düğüm sayısı (node)

P = bağlı bileşenlerin sayısı

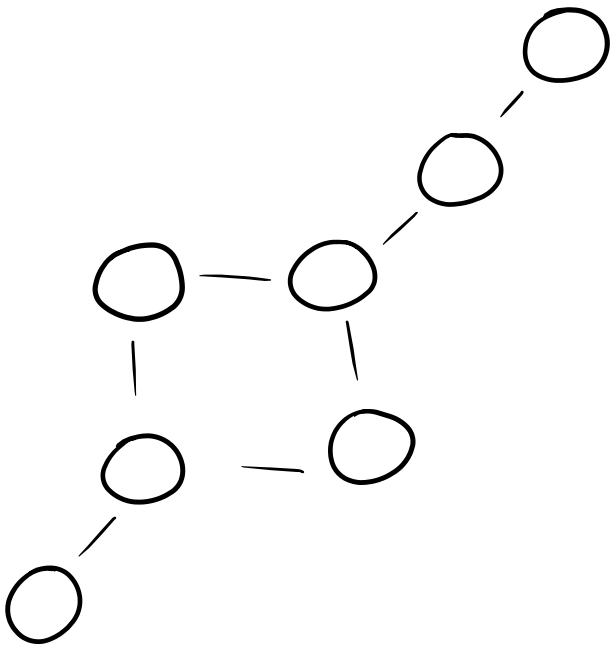
Örnek bir hesaplama:

Sözde bir kod:

```
MAX = 0
A = 2
B = 20
IF A > B THEN
     MAX = A
ELSE
     MAX = B
END
PRINT MAX
```

Sözde kodun akış grafiği:

(Şekil 8)



Şekil 8'de belirtilen akış şemasında yer alan ilk ve son düğüm programın başla ve dur adımlarıdır. Akış grafiğindeki her bir ilişki çizgisini kenar olarak, her bir daireyi ise düğüm olarak nitelendirin.

Bağlı bileşen sayısı tek bir program için her zaman 1'e eşittir. Bu durumda tek program için karmaşıklık hesaplanırken P sonuç üzerinde etkisiz olacaktır. P'nin işlem üstündeki etkisi farklı bitiş noktalarına sahip birden fazla başlangıç noktası olduğunda görülmektedir.

Sözde kodun hesaplanan döngüsel karmaşıklığı formülün uygulanmasıyla bu şekilde bulunabilir:

E = 7

N = 7

P = 1

M = 7-7+2×1 = 2

34.2 Risk Değerlendirmesi

Döngüsel karmaşıklık aralığına göre çeşitli yaygın risk değerlendirmeleri vardır.

Döngüsel Karmaşıklık	Risk Değerlendirmesi
$1 \leq M \leq 10$	Basit karmaşıklık, pek risk yok
$11 \leq M \leq 20$	Karmaşık program, ortalama risk
$21 \leq M \leq 49$	Fazla karmaşık program, yüksek risk
$M \geq 50$	Test edilemez karmaşıklık, çok yüksek risk

Bir altyordamın karmaşıklık sınırı yaygın şekilde 15 olarak kabul edilir. 15 ve üstünde bir döngüsel karmaşıklık sorun olarak nitelendirilir.

Karmaşıklığın yüksek olması her zaman kaynak kodun insan tarafından okunamaz ya da anlaşılması çok zor olduğu anlamına gelmez. Bazı durumlarda okunabilir sıralı koşul ifadelerin artarda kullanılması sonucu yüksek karmaşıklık değerleri elde edilebilir. Bu sıralı koşulların altyordamlara bölünmesi ya da daha da parçalanması mümkün değilse kabul edilebilir bir karmaşıklık olarak değerlendirilebilir.

35. BİRLİKTE ÇALIŞABİLİRLİK (INTEROPERABILITY)

Dil birlikte çalışabilirliği (language interoperability) iki farklı programlama dilinin birbirini kullanabilme yeteneğidir. Birlikte çalışabilen programlama dilleri aynı sistemin bir parçası olabilir, yerel olarak iletişime geçebilir ve aynı tür veri yapıları üzerinde çalışabilir.

Bir programlama dili tasarımında birlikte çalışabilirlik genellikle geliştirilen programlama dili farklı bir programlama dilinin mirasçısı (successor) olarak nitelendiriliyorsa kullanılır. Bununla birlikte geliştirilen programlama dilinin tasarım amacında var olan popüler ve amaca iyi hizmet eden bir dilin de birlikte çalışabilirlik sergilemesi istenebilir.

Ara dil (Bkz. Intermediate Representation) yaklaşımı da birlikte çalışabilirliğe sahip olmak için kullanılan yöntemlerden biridir. Ortak dil çalışma platformlarının birbiri ile ara dil sayesinde nasıl uyumlu bir yürütme sergileyebileceğine değinilmişti.