Önsöz [3.5hr]

Bu kitabın 1986 edisyonundan bu yana derleyici tasarımı dünyası ciddi ölçüde değişti. Programlama dilleri gelişti ve yeni derleme problemleri ortaya çıktı. Bilgisayar mimarileri, bir derleyici tasarımcısının faydalanması gereken çeşitli kaynaklar sunuyor. Büyük ihtimalle en ilginç olanı, saygıdeğer kod optimizasyon teknolojisi derleyiciler dışında da kullanım alanlarına kavuştu. Şimdilerde yazılımlarda hataları bulan programlarda, hatta daha da önemlisi var olan kodda güvenlik problemlerini bulmak üzere kullanılıyor. Ve daha bir çok 'front end' teknoloji -gramerler, düzenli ifadeler, parser'lar ve sentaks-yönelimli dönüştürücüler- hala çokça kullanılıyor.

Dolayısı ile geçmiş edisyonlardan kalma felsefemiz değişmedi. Biliyoruz ki birkaç okuyucu başlıca bir programlama dili için derleyici tasarlayacaktır. Yine de, derleyiciler ile ilişkilendirilen modeller, teori ve algoritmalar yazılım tasarımı ve yazılım geliştirme alanlarındaki geniş bir yelpazedeki problemler için kullanılabilir. Bu yüzden dilin kaynağından veya hedef makina'dan bağımsız olarak dil işlemcisi tasarımında karşılaşılan problemlerin üstüne basıyoruz.

Kitabın Kullanımı

Kitaptaki meteryallerin çoğunu işlemek için neredeyse iki Güz dönemi gerekiyor. Yaygın olarak kitabın ilk yarısı üniversiteye girişte gerisi okulun sonlarına doğru işlenir. Genel hatlarıyla bölümleri anlatmak gerekirse;

Bölüm 1 motive edici meteryalleri, ayrıca bilgisayar mimarisi ve programlama dili prensiplerinin arka planında karşılaşılan bazı problemleri içerir.

Bölüm 2 minyatür bir derleyici geliştirip ileride daha da detaylandırılacak bir çok önemli konsepti tanıtır. Derleyicinin kendisi Ek'tedir.

Bölüm 3 lexical analiz, düzenli ifadeler, sonlu-durum makineleri ve tarayıcı-üretici araçları kapsar. Bu meteryaller her türlü metin işlemede temeldir.

Bölüm 4 başlıca *yukarıdan aşağıya* (tekrarlı-azalan, LL) ve *alttan yukarıya* çözümleme metodlarını kapsar

Bölüm 5 sentaks-yönelimli tanımlarda ve dönüştürücülerde başlıca fikirleri sunar.

Bölüm 6 Bölüm 5'in teorisini alıp bunu tipik bir programlama dili için orta seviye kod oluşturmak için nasıl kullanılacağını gösterir.

Bölüm 7 çalışma-anı ortamlarını, özellikle çalışma-anı yığınını ve atık toplamasını kapsar.

Bölüm 8 nesne-kodu üretimi üzerinedir. Temel yapıların işasını, ifadelerden ve ve temel bloklardan kod oluşturulmasını ve register-tahsisi tekniklerini kapsar.

Bölüm 9 akış grafiklerini, veri-akış framework'larını ve bu framework'ları çözmek için tekrarlı algoritmaları içeren kod optimizasyonu teknolojisini tanıtır.

Bölüm 10 komut-seviyesi optimizasyonunu kapsar. Üzerinde durulan konu, paralelizm'in küçük komut dizelerinden çıkarılması ve bunları aynı anda birden fazla işlem yapabilen işlemciler üzerinde programlanmasıdır.

Bölüm 11 geniş-ölçekli paralelizm tespiti ve bundan faydalanılması hakkında tartışır. Üzerinde durulan konu, çok boyutlu dizelere yayılan sık döngülere sahip nümerik kodlardır.

Bölüm 12 prosedürlerarası analiz üzerinedir. Pointer analizini, örtüşmeyi ve kod içinde verilen bir noktaya uzanan prosedür çağrıları dizisini de içeren veri-akış analizlerini kapsar.

Bu meteryaller ile Columbia, Harvard ve Stanford üniversitesinde dersler işlendi. Columbia'da ilk yıl programlama dilleri ve dönüştürücüler dersi sıklıkla ilk 8 bölümden meteryalleri kullandı/önerdi. Dersin amacı küçük gruplar halinde öğrencilerin kendi dillerini tasarladıkları bir Güz dönemi projesidir. Öğrenci yaratımı olan bu projeler çeşitli uygulama alanlarını -kuantum bilişim, müzik sentezi, bilgisayar grafiği, oyun vb.- kapsıyor. Öğrenciler Bölüm 2 ve 5'te bahsedilen ANTLR, Lex ve Yacc gibi derleyici-komponent üreticilerini ve ayrıca sentaks-yönelimli çevrim tekniklerini kullanarak kendi derleyicilerini tasarlayabiliyorlar. Bunu takip eden derste ağ işlemcileri ve çok işlemcili mimariler gibi modern makineler için kod oluşturma ve optimizasyonuna vurgu yapan Bölüm 9'dan 12'ye kadar olan kısma odaklanırlar.

Stanford'da çeyrek dönem giriş niteliğinde bir ders kabaca Bölüm 1'den 8'e kadar olan içerikleri kapsar, fakat Bölüm 9'dan global kod optimizasyonu konusu da işlenir. Derleyiciler üzerinde ikinci bir ders Bölüm 9'dan 12'ye kadar ve Bölüm 7'den ileri seviye atık toplayıcı içerikleri kapsar. Öğrenciler kurum içi geliştirilmiş, veri-akışı analiz algoritmalarını uygulamak için Joeq adlı Java-tabanlı bir sistem kullanırlar.

Ön Koşullar

Okuyucu ikinci sınıf programlama, veri yapıları ve ayrık matematik olmak üzere biraz bilgisayar bilimleri bilgisine sahip olmalıdır. Bir kaç farklı programlama dili bilgisi de yararlı olur.

Alıştırmalar

Kitap, neredeyse her bölümde geniş kapsamlı alıştırmalar içerir. Zorlu olan alıştırmaları veya alıştırma parçalarını ünlem işareti, daha zorlu olanları ise çift ünlem işareti ile belirteceğiz.

Teşekkür

Jon Bentley bu kitabın taslağı üzerinde birden fazla bölümde bize geniş çaplı yorumlarda bulundu. Bize yardımcı olan yorumlar ve hatalar aşağıda sayacağımız insanlar tarafından bildirildi; -

Yardımları için bu insanlara teşekkürlerimizi sunuyoruz. Hala kitapta bulunan hatalar tabiki bize aittir.

Ayrıca Monica, 18-yıllık bir eğitim için SUIF'in derleyici takımındaki iş arkadaşlarına teşekkür etmek istiyor;

İçindekiler

Bölüm 1

Giriş

Programlama dilleri, hesaplamaları insanlar ve bilgisayarlar için tanımlayan notasyonlardır. İçinde bulunduğumuz dünya programlama dillerine bağımlıdır, çünkü var olan tüm bilgisayarlar üzerinde çalışan tüm yazılımlar bir programlama dili ile yazıldı. Fakat, bir program çalışmaya başlamadan önce bilgisayarın onu anlayacağı bir forma çevrilmelidir.

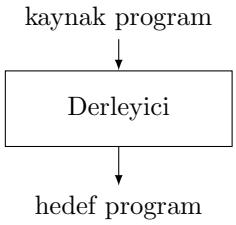
Bu çevrimi yapan yazılım sistemlerine *derleyici* denir.

Bu kitap derleyicilerin nasıl tasarlanıp uygulanacakları hakkındadır. Burada dönüştürücülerin birkaç temel fikir kullanılarak geniş çeşitlilikte diller ve makineler için oluşturulabileceğini keşfedeceğiz. Derleyicilerin yanı sıra, derleyici tasarımında kullanılan prensip ve teknikler o kadar çok alanda uygulanabilirdir ki, yüksek ihtimalle bir bilgisayar bilimcisinin/mühendisinin kariyerinin bir çok noktasında tekrar tekrar kullanılacaktır. Derleyici yazımı'nın çalışılması, programlama dilleri, makine mimarileri, dil teorisi, algoritmalar ve yazılım mühendisliği olmak üzere bir çok noktaya dokunur.

Bu ilk bölümde, dil dönüştürücülerin değişik formlarını, tipik bir derleyicinin yapısını kabaca gözden geçirecek ve derleyicileri şekillendiren programlama dilleri ve bilgisayar mimarileri trendlerinden bahsedeceğiz. Ayrıca bilgisayar bilimi ve derleyici tasarımı teorisi arasındaki ilişki ve derlemenin de ötesine geçen derleyici teknolojilerinin genel hatları üzerine gözlemleri de işleyeceğiz. Son olarak derleyici çalışmamız için gerekli olan anahtar programlama-dili konseptleri'ni özetleyerek bitireceğiz.

1.1 Dil İşlemcileri

Basitçe, derleyici, bir dilde yazılmış -*kaynak* dil- herhangi bir programı okuyup başka bir dile -*hedef* dil - eşdeğer olarak çeviren bir programdır; bkz. Şekil 1.1. Derleyicinin önemli bir rölü, dönüşüm sırasında kaynak dilde tespit ettiği hataları bildirmesidir.

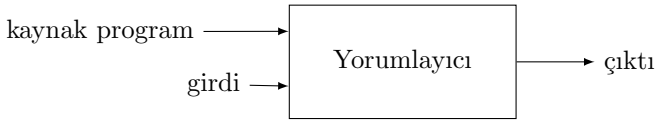


Şekil 1.1: Bir derleyici

Eğer hedef program çalıştırılabilir bir makine-dili programıysa, kullanıcı tarafından girdileri işlemesi ve çıktı üretmesi için kullanılabilir; bkz. Şekil 1.2.

Şekil 1.2: Hedef Programı Çalıştırmak

Yorumlayıcı[*] da yaygın bir dil işlemci tipidir. Şekil 1.3'te görüldüğü gibi hedef programını üretirken bunu çevrim olarak yapmaktansa, yorumlayıcı, kaynak programın içindeki komutları direkt olarak, kullanıcı tarafından sağlanan girdiler üzerinden çalıştırır.



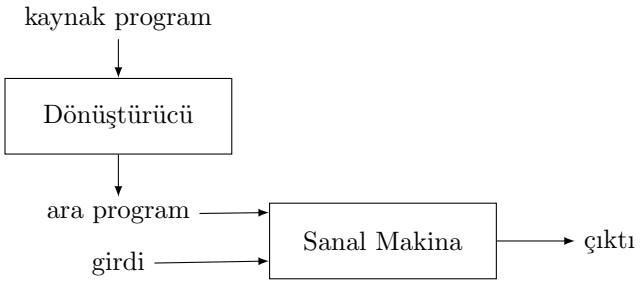
Şekil 1.3: Hedef Programı Çalıştırmak

Girdileri çıktılara eşleyen bir yorumlayıcıdansa, derleyici tarafından üretilmiş bir makine-dili hedef programı genellikle daha hızlıdır. Bununla beraber, bir yorumlayıcı, hataları genellikle bir derleyiciden daha iyi teşhis edebilir, çünkü yorumlayıcı kaynak programı satır satır çalıştırır.

Örnek 1.1: Şekil 1.4'te görüldüğü gibi Java dil işlemcileri derlemeyi ve yorumlamayı beraber kullanır. Java kaynak programı önce *bytecode* denilen bir ara form'a çevrilir. Ardından bytecode'lar bir sanal makine tarafından yorumlanır. Bu tür bir ayarlamanın avantajı, bir makinede derlenen bir kodun başka bir makinede ya da bir ağ üzerindeki makinelerde yorumlanabilmesidir.

Girdilerin çıktılara dönüştürülmesini hızlandırmak için, *tam-zamanında* derleyiciler denilen bazı Java derleyicileri, bytecode'ları ara program girdileri işlemeden önce, anlık olarak makine diline çevirir.

[*](İng.) Interpreter. (Ç.N.)



Şekil 1.4: Bir hibrid derleyici

Şekil 1.5'te gösterildiği gibi bir derleyiciye ek olarak, çalıştırılabilir bir hedef program oluşturmak için birkaç program daha gerekli olabilir. Bir kaynak program ayrı dosyalarda tutulan modüllerden oluşmuş olabilir. Bu kaynak kodu toparlamak bazen *önişlemci* denlilen başka programlara havale edilir. Önişlemci ayrıca makro denilen kısayolları kaynak kodun ifadelerine çevirebilir.

Ardından düzenlenmiş kaynak kodu derleyiciye yüklenir. Derleyici çıktısı olarak assembly-dili programı üretebilir, çünkü assembly üretimi ve hata ayıklaması kolay bir programdır. Ardından assembly dili, yeniden konumlandırılabilir makine kodu üreten ve *assembler* denilen bir programda işlenir.

Büyük programlar genelde parçalar halinde derlenir, böylece yeniden konumlandırılabilir makina kodu, makinada çalışan diğer yeniden konumlandırılabilir nesne ve kütüphane dosyaları ile bağlanabilir. *Bağlayıcı* bir dosyadaki dosyanın başka bi dosyadaki dosyaya ulaşması için harici hafıza adreslerini çözümler. *Yükleyici* sonra tüm bu çalıştırılabilir nesne dosyalarını bir araya getirip çalıştırılmak üzere hafızaya yükler.

1.1.1 Bölüm 1.1 için Alıştırmalar

Örnek 1.1.1: Bir yorumlayıcı ile derleyicinin arasındaki fark nedir?

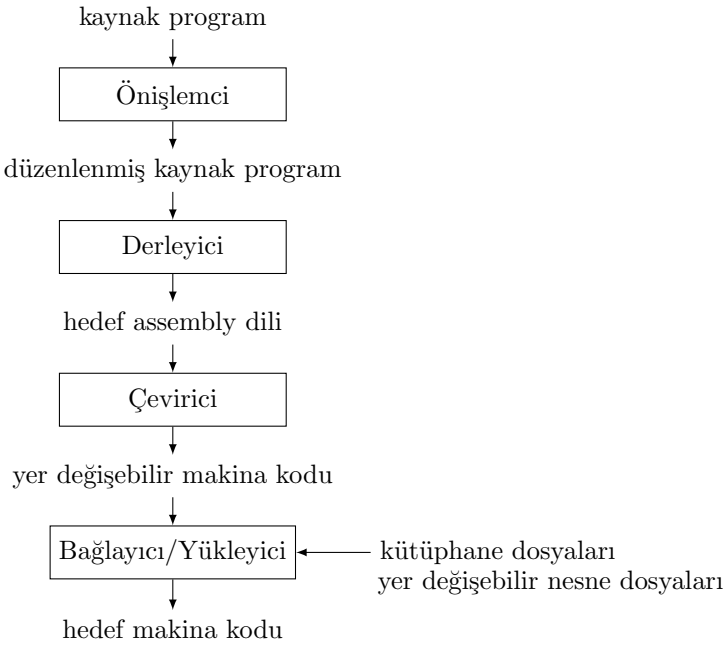
Örnek 1.1.2: a ve b maddelerin avantajları nelerdir?; a) Bir yorumlayıcı üzerine bir derleyici. b) Bir derleyici üzerine bir yorumlayıcı.

Örnek 1.1.3: Derleyicinin makina dilindense assembly dili ürettiği bir dilişleme sisteminin avantajları nelerdir?

Örnek 1.1.4: Yüksek seviyeli bir dili başka yüksek seviyeli bir dile çeviren derleyiciye *kaynaktan-kaynağa* dönüştürücü denir. C dilini bir derleyici için kaynak dil olarak kullanmanın avantajları ne olur?

Örnek 1.1.5: Bir çevirici'nin* görevlerinden birkaçını tanımlayın.

*(İng.) Assembler. (Ç.N.)



Şekil 1.5: Bir dil-işleme sistemi

1.2 Bir Derleyicinin Yapısı

Bu noktaya kadar bir derleyiciye kaynak kodu, aynı semantik ile başka bir hedef programa eşleyen bir kutucuk olarak baktık. Eğer bu kutuyu biraz aralarsak aslında bu eşleme işleminin iki kısımdan oluştuğunu görürüz: analiz ve sentez.

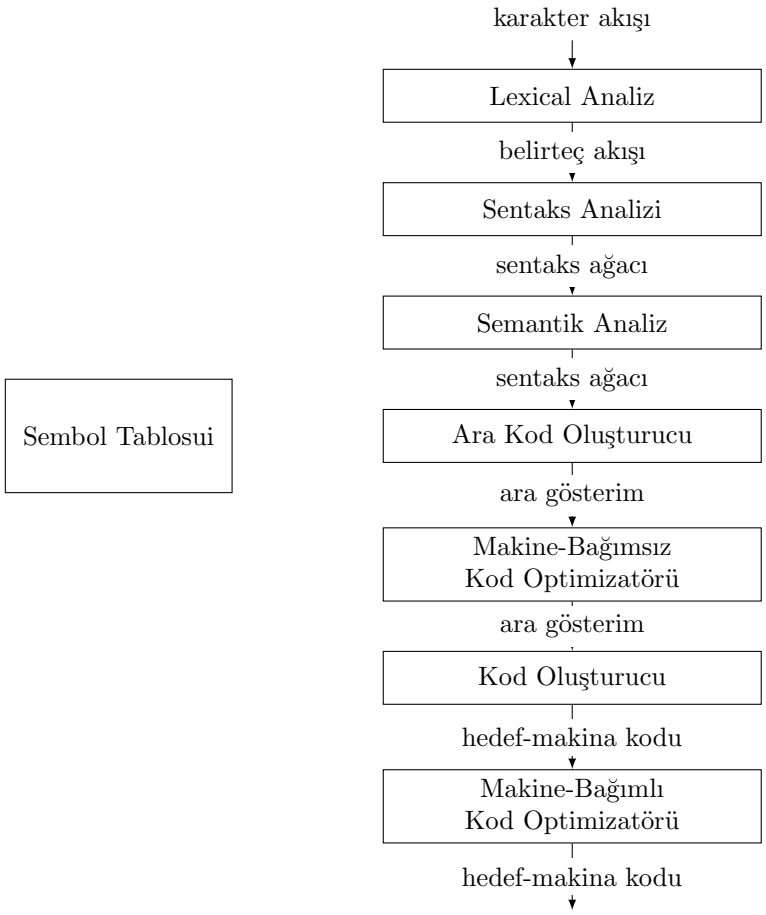
Analiz kısmı kaynak kodu bileşenlerine ayırır ve bu parçalar üzerinde bir sentaks yapısı uygular. Ardından bu yapıyı, kaynak kodun bir ara temsilini oluşturmak için kullanır. Eğer analiz kısmı bu aşamada sözdizimisel olarak yanlış kullanılmış veya semantik olarak belirsiz bir durum bulursa kullanıcıya, gerekli aksiyonları alması için bilgilendirici mesajlar yollamalıdır. Analiz kısmı ayrıca kaynak kod hakkında da bilgi toplar ve bunu *sembol tablosu* denen bir veri yapısına kaydeder, ardından ara temsili ile beraber sentez kısmına iletir.

Sentez kısmı, istelilen programı ara program temsilinden ve sembol tablosundaki bilgilerden inşa eder. Derleyicinin analiz kısmı genellikle *ön uç*[*] , sentez kısmı içine *arka uç*[†] olarak isimlendirilir.

Derleme prosesini detaylı bir şekilde incelersek, kaynak programın her bir temsilini bir diğerine çeviren bir aşamalar dizisi görürüz. Tipik bir derleyicinin aşamalar halinde ayrılmış gösterimi Şekil 1.6'da gösterilmiştir. Pratikte birkaç aşama birlikte gruplanmış olabilir ve gruplandırılmış aşamalar arasındaki ara

[*](İng.) Front end. (Ç.N.)
[†](İng.) Back end. (Ç.N.)

temsillerin açıkça belirtilmesine gerek yoktur. Tüm kaynak program hakkındaki bilgileri barındıran sembol tablosu derleyicinin tüm aşamaları tarafından kullanılır.



Şekil 1.6: Bir derleyicinin aşamaları

Bazı derleyiciler ön uç ile arka uç arasında bir makina-bağımsız kod optimizatörü barındırırlar. Bu optimizasyon, aksi takdirde optimize edilmemiş bir ara form'u çevirecek olmasından dolayı, arka ucun daha iyi bir hedef program oluşturması için kullanılan, ara form üzerindeki transformasyonları gerçekleştiren bir aşamadır. Optimizasyon opsiyonel olduğundan dolayı, Şekil 1.6'da gösterilen bir veya her iki optimizasyon aşaması bazı durumlarda orada olmayabilir.

1.2.1 Lexical Analiz [10hr (until here)]

Bir derleyicinin ilk aşaması *lexical analiz* ya da *tarama* olarak adlandırılır. Lexical analizör kaynak programı oluşturan karakter akışını okur ve onları *sözcükbirim*[*] denilen mantıklı gruplara ayırır. Her bir sözcük birim için lexical analizör kendinden sonraki sentaks analizi aşaması için çıktı olarak aşağıda görülen formda bir belirteç[†] oluşturur.

$$\langle \textit{belirteç-adı, bağlı-değer} \rangle$$

Belirteç'in içindeki ilk komponent olan *belirteç-adı,* sentaks analizi boyunca kullanılan soyut bir semboldür, ikinci komponent *bağlı-değer* ise bu belirteç için sembol tablosundaki bir girdiyi gösterir. Sembol-tablosu'ndaki bilgi girdileri, semantik analiz ve koş oluşturmak için gereklidir.

Örneğin, atama durumunu içeren bir kaynak programı düşünün

$$\text{konum} = \text{baslangic} + \text{oran} * 60 \qquad (1.1)$$

Bu atamadaki karakterler aşağıda gösterilen sözcükbirim'lere gruplanabilir ve sentaks analizine geçirilmek üzere belirteçler oluşturulabilir:

1. konum , ⟨**id**, 1⟩ şeklinde yazılabilecek bir sözcükbirimdir. **id** burada kimlik yerine geçen soyut bir semboldür, 1 ise, konum 'un sembol-tablosu'ndaki girdisini gösterir. Bir kimlik için sembol-tablosu girdisi, bu kimlik için isim ve tip gibi bilgileri tutar.

2. Atama sembolü, ⟨=⟩ şeklinde yazılabilinecek bir sözcükbirim'dir. Belirteç'in herhangi bir bağlı-değer'i olmadığı için ikinci komponenti çıkardık. Belirteç-adı için **atama** gibi herhangi bir soyut sembol de kullanabilirdik, fakat simgelemde kolaylık olması açısından, soyut sembol için direkt olarak sözcükbirim'in kendisini seçtik.

3. baslangic, ⟨**id**, 2⟩ gibi bir belirteçle yazılabilecek bir sözcükbirim'dir. 2, sembol-tablosu'ndaki baslangic girdisini gösterir.

4. +, ⟨+⟩ belirteci olarak yazılabilicek bir sözcükbirim'dir.

5. oran, ⟨**id**, 3⟩ gibi bir belirteçle yazılabilecek bir sözcükbirim'dir. 3, sembol-tablosu'ndaki oran girdisini gösterir.

6. *, ⟨*⟩ belirteci olarak yazılabilicek bir sözcükbirim'dir.

7. 60, ⟨60⟩ belirteci olarak yazılabilicek bir sözcükbirim'dir.[1]

Sözcükbirimleri arasındaki boşluklar lexical analizör tarafından çıkarılacaktır.

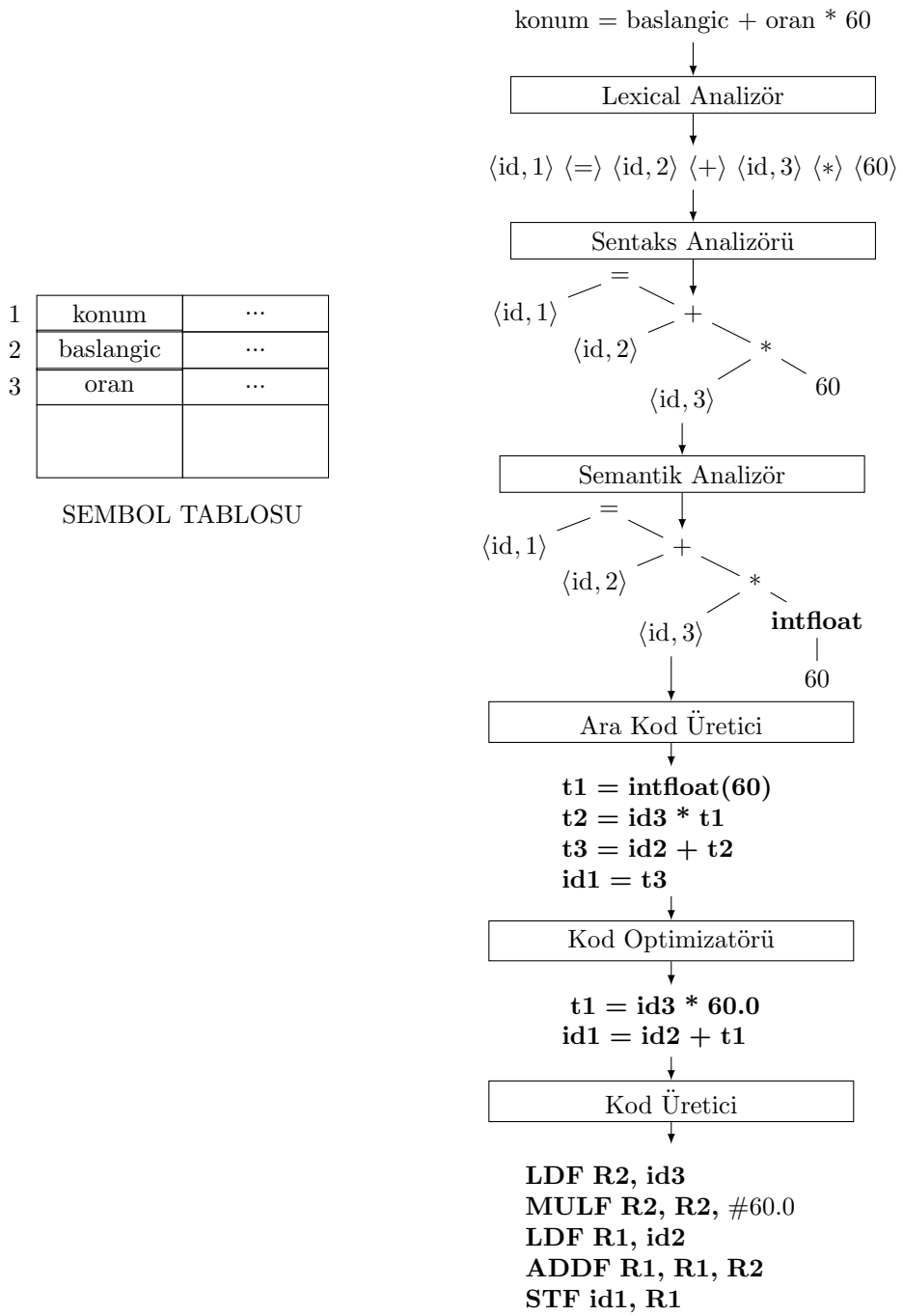
Şekil 1.7, lexical analiz'den sonra, belirteç dizisi şeklinde, (1.1)'deki atama ifadesi gösterilir.

---

[*] (İng.) Lexeme. (Ç.N.)

[†] (İng.) Token. (Ç.N.) Bir derleyici için, bir veri dizisindeki en küçük mantıklı bilgi formu.

[1] Teknik olarak, 60 sözcükbirim'i için, ⟨**id**, 4⟩ şeklinde bir token yapmamız ve 4'ün sembol tablosunda 60 tamsayısını göstermesi gerekir, fakat sayı belirteçleri konusunu Bölüm 2'ye kadar ertelememiz gerekiyor. Bölüm 3'te ise lexical analiz teknikleri işlenecek



Şekil 1.7: Bir atama işleminin çevrilmesi

$$\langle \mathrm{id}, 1\rangle \ \langle = \rangle \ \langle \mathrm{id}, 2\rangle \ \langle + \rangle \ \langle \mathrm{id}, 3\rangle \ \langle * \rangle \ \langle 60 \rangle \qquad (1.2)$$

Bu gösterimde belirteç adları, =, + ve *, atama, toplama ve çıkarma işlemleri için soyut sembollerdir.

1.2.2 Sentaks Analizi [52min]

Derleyicinin ikinci aşaması sentaks analizi'dir[†]. Sentaks analizi, lexical analizör tarafından oluşturulan belirteçlerin ilk komponentlerini, bu belirteç akışının gramatik yapısını tasvir eden ağaç yapısı türünde bir ara kod gösterimi olşturmak için kullanır. Sentaks ağacı tipik bir gösterimdir; her iç düğüm bir operasyonu, ve bu düğümlerin çocuk düğümleri ise operasyonun argümanlarını temsil eder. (1.2)'deki belirteç akışı'nın sentaks ağacı Şekil 1.7'de sentaktik analizör'ün çıktısı olarak gösterilmiştir. Bu ağaç, atamadaki operasyonların çalıştırılma sırasını gösterir.

konum = baslangic + oran * 60

Ağaç, * ile etiketlenmiş, ⟨**id**, 3⟩'ün sol çocuk, 60 tamsayısının da sağ çocuk düğümü olarak tanımlandığı bir iç düğüme sahiptir. ⟨**id**, 3⟩, **oran** tanımlayıcısını temsil eder. * ile etiketlenmiş düğüm, ilk önce **oran**'ın değeri ile 60'ı çarpmamız gerektiğini belirgin bir şekilde gösterir. + ile işaretlenmiş düğüm bu çarpım değerini **baslangic**'ın değeri ile toplamamız gerektiğini belirtir. = şeklinde etiketlenmiş, ağacın kökü olan düğüm, bu toplamın sonucunu **konum** belirtecinin olduğu lokasyona kaydetmemiz gerektiğini gösterir. Bu operasyonların sırası, geleneksel aritmetik ile tutarlıdır; çarpımın, toplamadan daha yüksek önceliği vardır, dolayısıyla çarpma operasyonu, toplamadan önce gerçekleştirilmelidir.

Bundan sonra gelen derleyici aşamaları gramatik yapıyı, hedef programı üretmek için kaynak programın analizinde yardımcı olarak kullanırlar. Bölüm 4'te bağlam-duyarsız gramerleri programlama dillerinin yapısını belirtmek için kullanacağız ve belirli sınıf ve gramerlerden verimli sentaks analizörleri'ni otomatik olarak oluşturacak algoritmaları tartışacağız. Bölüm 2 ve 5'te sentaks-yönelimli tanımlamaların, programlama dili yapılarını çevirmede yardımcı olabileceğini göreceğiz.

1.2.3 Semantik Analiz [34min]

Semantik analizör, kaynak programın dilin tanımlamalarına göre semantik tutarlılığını, sentaks ağacını ve sembol tablosundaki bilgileri kullanarak kontrol eder. Ayrıca ileride, ara kod üretiminde kullanılmak üzere, tip bilgilerini toplayarak bunu ya sentaks ağacında ya da sembol tablosunda saklar.

Semantik analiz'in önemli bir parçası, derleyicinin her bir operatör için, o operatörün terimlerinin uyuşup uyuşmadığını kontrol ettiği *tip-kontrolü*'dür. Örneğin, birçok programlama dili bir dizinin indis değerlerinin tamsayı olmasını ister; derleyici, dizi indisi belirtmek için eğer ondalık bir sayı kullanılmışsa, bunun için bir hata raporu vermelidir.

[†]Bazen *Syntax Analysis* bazen de *Parsing* olarak gösterilir (Ç.N.)

Dil spesifikasyonları, *zorlama*[‡] adı verilen bazı tip çevrimlerine izin veriyor olabilir. Örneğin bir binary aritmetik operatörü bir tamsayı çiftine ya da bir ondalıklı sayı çiftine uygulanabilir. Eğer operatör bir tamsayı ve bir ondalıklı sayıya uygulanmışsa, derleyici tamsayıyı ondalıklıya, ya da ondalıklı sayıyı tamsayıya çevirebilir veya zorlayabilir.

Böyle bir zorlama Şekil 1.7'de görünür. Farz edin ki, **konum**, **baslangic** ve **oran** ondalık sayı olarak tanımlanmış ve **60** ise kendinden dolayı bir tamsayı olsun. Şekil 1.7'deki semantik analizörün içindeki tip-kontrolcüsü * operatörünün bir ondalık sayı olan **oran**, ve bir tam sayı olan **60** için uygulandığını fark edecek. Bu durumda tamsayı, bir ondalıklı sayıya çevrilebilir. Şekil 1.7'de semantik analizörün **inttofloat** için fazladan bir düğümü olduğuna dikkat edin; ki açıkça tamsayı türündeki bir argümanı ondalıklı sayıya çevirir. Tip kontrolü ve semantik analiz Bölüm 6'da tartışılacaktır.

1.2.4 Ara Kod Üretimi [0min]

Bir kaynak programı hedef programa çevirme sürecinde, derleyici çeşitli formlarda birden fazla ara kod üretebilir. Sentaks ağaçları bir ara gösterim formlarıdır; bu formlar sentaks ve semantik analizinde yaygın olarak kullanılırlar.

Kaynak programın sentaks ve semantik analizinden sonra, birçok derleyici belirgin bir düşük-seviye ya da makine-tipi bir ara gösterim oluşturur; bunu soyut bir makina için bir program olarak düşünebiliriz. Bu ara gösterimin iki önemli özelliği olmalıdır: üretilmesi ve kaynak makineye çevrilmesi kolay olmalıdır.

Bölüm 6'da, bir dizi assembly benzeri, komut barındıran (komut başına üç terim), *üç-adress* denen bir ara formu inceleyeceğiz. Her terim bir register olarak davranabilir. Şekil 1.7'deki ara kod üreticinin çıktısı, üç-adress kod dizisi içerir.

```
t1 = intfloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

[‡](İng.) Coercion (Ç.N.)