CENG 218 Programlama Dilleri

Bölüm 11: Soyut Veri Türleri ve Modüller

Öğr.Gör. Şevket Umut Çakır

Pamukkale Üniversitesi

Hafta 13

Hedefler

- Soyut veri türlerinin cebirsel tanımını anlamak
- Soyut veri türü mekanizmalarına ve modüllerine aşina olmak
- C, C++ ad alanlarında ve Java paketlerinde ayrı derlemeyi anlamak
- Ada paketlerine aşina olmak
- ML modülleri öğrenmek
- Daha önceki dillerdeki modüller hakkında bilgi edinmek
- Soyut veri türü mekanizmalarıyla ilgili sorunları anlamak
- Soyut veri türlerinin matematiğine aşina olmak

Giriş

- Veri türü: Bir değer kümesi ve bu değerler üzerindeki belirli işlemler
- İki tür veri türü: önceden tanımlanmış ve kullanıcı tanımlı
- Önceden tanımlanmış veri türleri:
  - Kullanıcıyı makineye bağlı uygulamadan izole eder
  - Önceden tanımlanmış bir dizi işlem tarafından manipüle edilir
  - Kullanım, önceden belirlenmiş semantik tarafından tamamen belirlenir

Giriş

- Kullanıcı tanımlı veri türleri:
  - Dilin yerleşik veri türleri ve tür oluşturucuları kullanılarak veri yapılarından oluşturulmuştur
  - Dahili organizasyon kullanıcı tarafından görülebilir
  - Önceden tanımlanmış işlem yok
- Mümkün olduğunca yerleşik bir türden çok sayıda özelliğe sahip veri türleri oluşturmak için bir mekanizmaya sahip olmak arzu edilir.
- **Soyut veri türü(Abstract data type)** (veya **ADT**): kullanıcı tanımlı veri türlerini oluşturmak için bir veri türü

Giriş

- Veri türleri için önemli tasarım hedefleri arasında değiştirilebilirlik(modifiability), yeniden kullanılabilirlik(reusability) ve güvenlik(security) bulunur
- **Kapsülleme(Encapsulation)**:
  - Bir veri türü ile ilgili tüm tanımların tek bir yerde toplanması
  - Türün kullanımının o konumda tanımlanan işlemlerle kısıtlanması
- **Bilgi gizleme(Information hiding)**: uygulama ayrıntılarının veri türünün tanımından ayrılması ve gizlenmesi

Giriş

- Türleri oluşturmaya yönelik bir **mekanizma(mechanism)** ile bir türün **matematiksel kavramı(mathematical concept)** arasında bazen kafa karışıklığı olur.
- Matematiksel modeller genellikle **cebirsel tanım(algebraic specification)** olarak verilir.
- **Nesne yönelimli programlama(Object-oriented programming)**, yürütme sırasında kendi kullanımlarını kontrol etmek için varlık kavramını vurgular.
- Soyut veri türleri, gerçek nesne yönelimli programlamayı temsil eden aktif kontrol seviyesini sağlamaz.

Giriş

- Soyut bir veri türü kavramı, onu uygulamak için kullanılan dil paradigmasından bağımsızdır.
- **Modül(Module)**: veri türlerini içerebilen veya içermeyen hizmetler topluluğu

Soyut Veri Türlerinin Cebirsel Tanımı

- **Karmaşık(Complex)** veri türü: çoğu dilde yerleşik bir tür olmayan soyut bir veri türü
  - $x \pm iy$ formunun karmaşık bir sayısını temsil etmek için kullanılır; burada i, $\sqrt{-1}$ karmaşık sayısını temsil eder
  - Gerçek ve sanal(imaginary) bir kısımdan karmaşık bir sayı, artı gerçek ve sanal kısımları çıkarmak için fonksiyonlar oluşturabilmelidir.
- **Sözdizimsel tanım(Syntactic specification)**: türün adı ve işlemlerin adları, parametrelerinin tanımı ve döndürülen değerler dahil
  - Türün imzası olarak da adlandırılır

Soyut Veri Türlerinin Cebirsel Tanımı

- $f : X \rightarrow Y$ veri türünün işlemlerini belirtmek için fonksiyon gösterimi kullanılır.
- Karmaşık veri türü için imza:

type complex **imports** real

operations:

+: complex × complex → complex
-: complex × complex → complex
*: complex × complex → complex
/: complex × complex → complex
-: complex → complex
makecomplex: real × real → complex
realpart: complex → real
imaginarypart: complex → real

Soyut Veri Türlerinin Cebirsel Tanımı

- Bu belirtim, herhangi bir anlambilim kavramından veya işlemlerin gerçekten sahip olması gereken özelliklerden yoksundur.
- Matematikte, fonksiyonların anlamsal özellikleri genellikle **denklemler(denklemler)** veya **aksiyomlarla(axioms)** tanımlanır.
  - Aksiyom örnekleri: birleşme(associativity), değişme(commutative) ve dağıtım(distributive) yasaları
- Aksiyomlar, karmaşık sayıların anlamsal özelliklerini tanımlamak için kullanılabilir veya özellikler, gerçek veri türünün özelliklerinden türetilebilir.

Soyut Veri Türlerinin Cebirsel Tanımı

- Örnek: karmaşık toplama, gerçek sayıların toplamasına dayalı olabilir

$$\text{realpart}(x + y) = \text{realpart}(x) + \text{realpart}(y)$$

$$\text{imaginarypart}(x + y) = \text{imaginarypart}(x) + \text{imaginarypart}(y)$$

  - Bu, gerçek sayıların karşılık gelen özelliklerini kullanarak karmaşık sayıların aritmetik özelliklerini kanıtlamamıza olanak tanır.
- Kompleks türünün eksiksiz bir cebirsel özelliği, imzayı, değişkenleri ve denklem aksiyomlarını birleştirir
  - **Cebirsel tanım(algebraic specification)** olarak adlandırılır

Soyut Veri Türlerinin Cebirsel Tanımı

type complex **imports** real

operations:

+: complex × complex → complex
=: complex × complex → complex
*: complex × complex → complex
/: complex × complex → complex
-: complex → complex
makecomplex : real × real → complex
realpart : complex → real
imaginarypart : complex → real

variables: x,y,z: complex; r,s: real

axioms:

realpart(makecomplex(r,s)) = r
imaginarypart(makecomplex(r,s)) = s
realpart(x + y) = realpart(x) + realpart(y)
imaginarypart(x + y) = imaginarypart(x) + imaginarypart(y)
realpart(x - y) = realpart(x) - realpart(y)
imaginarypart(x - y) = imaginarypart(x) - imaginarypart(y)
. . .
(more axioms)
. . .

Soyut Veri Türlerinin Cebirsel Tanımı

- Denklemsel semantik, uygulama davranışının açık bir göstergesidir
- Bununla birlikte, uygun bir denklem seti bulmak zor olabilir
- Sözdizimsel tanımdaki **okun(arrow)** bir fonksiyonun alanını(domain) ve aralığını(range) ayırdığını, eşitliğin ise fonksiyonlar tarafından döndürülen değerlerde olduğunu unutmayın.
- Bir tanım, belirtilmemiş bir veri türü ile parametrelendirilebilir

Soyut Veri Türlerinin Cebirsel Tanımı

- `createq`: bir sabit
  - Her zaman aynı değeri döndüren parametre almayan fonksiyon olarak görülebilir - boş olarak başlatılan yeni bir kuyruğun
- **Hata aksiyomları(Error axioms)**: hata değerlerini belirten aksiyomlar
  - İşlemlerle ilgili sınırlamalar sağlar
  - Örnek: `frontq(createq) = error`
- Kuyruktan çıkarma işleminin ön elemanı geri döndürmediğine dikkat edin; sadece onu atıyor

Soyut Veri Türlerinin Cebirsel Tanımı

type queue(element) **imports** boolean

operations:

createq: queue
enqueue: queue × element → queue
dequeue: queue → queue
frontq: queue → element
emptyq: queue → boolean

variables: q: queue; x: element

axioms:

emptyq(createq) = true
emptyq(enqueue(q,x)) = false
frontq(createq) = error
frontq(enqueue(q,x)) = if emptyq(q) then x else frontq(q)
dequeue(createq) = error
dequeue(enqueue(q,x)) = if emptyq(q) then q else enqueue(dequeue(q),x)

Soyut Veri Türlerinin Cebirsel Tanımı

- İşlemlerin anlamını belirten denklemler, bir uygulamanın özelliklerinin bir tanımı olarak kullanılabilir.
- Hafıza ya da atama işleminden söz edilmiyor
  - Bu özellikler tamamen işlevsel biçimdedir
- Pratikte, soyut veri türü uygulamaları genellikle fonksiyonel davranışı eşdeğer bir zorunlu davranışla değiştirir.
- Cebirsel bir tanım için uygun bir aksiyom seti bulmak zor olabilir

Soyut Veri Türlerinin Cebirsel Tanımı

- İşlemlerin sözdizimine bakarak ihtiyaç duyulan aksiyomların türü ve sayısı hakkında bazı yargılarda bulunabilir.
- **Yapıcı(Constructor)**: veri türünde yeni bir nesne oluşturan bir işlem
- **Denetleyici(Inspector)**: önceden oluşturulmuş değerleri alan bir işlem
  - **Yüklemler(Predicates)**: Boole değerlerini döndürür
  - **Seçiciler(Selectors)**: Boole olmayan değerleri döndürür
- Genel olarak, bir denetleyicinin bir kurucu ile her kombinasyonu için bir aksiyoma ihtiyacımız var

Soyut Veri Türlerinin Cebirsel Tanımı

- Örnek:
  - Kuyruğun aksiyom kombinasyonları şunlardır:

    emptyq(createq)

    emptyq(enqueue(q,x))

    frontq(createq)

    frontq(enqueue(q,x))

    dequeue(createq)

    dequeue(enqueue(q,x))
  - Altı kurala ihtiyaç olduğunu gösterir

Soyut Veri Türü Mekanizmaları

- Soyut veri türlerini ifade etmek için bir mekanizma, ADT'nin imzasını uygulamasından ayırmanın bir yoluna sahip olmalıdır.
  - ADT tanımının dışındaki herhangi bir kodun, uygulamanın ayrıntılarını kullanamayacağını ve yalnızca sağlanan işlemler aracılığıyla tanımlanan türde bir değer üzerinde çalışması gerektiğini garanti etmelidir
- ML, `abstype` adı verilen özel bir ADT mekanizmasına sahiptir

Soyut Veri Türü Mekanizmaları

```
abstype 'element Queue = Q of 'element list
with
  val createq = Q [];
  fun enqueue (Q lis, elem) = Q (lis @ [elem]);
  fun dequeue (Q lis) = Q (tl lis);
  fun frontq (Q lis) = hd lis;
  fun emptyq (Q []) = true | emptyq (Q (h::t)) = false;
end;
```

Şekil: ML `abstype` kullanan ve ML listelerini uygulayan bir kuyruk ADT

Soyut Veri Türü Mekanizmaları

- ML tercümanı, türün imzasının bir açıklamasıyla yanıt verir:

```
type 'a Queue
val createq = - : 'a Queue
val enqueue = fn : 'a Queue * 'a -> 'a Queue
val dequeue = fn : 'a Queue -> 'a Queue
val frontq = fn : 'a Queue -> 'a
val emptyq = fn : 'a Queue -> bool
```

- ML parametrik polimorfizme sahip olduğundan, kuyruk türü kuyrukta depolanacak öğenin türüne göre parametrelendirilebilir

Soyut Veri Türü Mekanizmaları

```
abstype Complex = C of real * real
with
  fun makecomplex (x, y) = C (x, y);
  fun realpart (C (r, i)) = r;
  fun imaginarypart (C (r, i)) = i
  fun +: (C (r1, i1), C (r2, i2)) = C (r1 + r2, i1 + i2);
  infix 6 +: ;
  (* diğer işlemler *)
end;
```

Şekil: ML **abstype** kullanan bir karmaşık sayı ADT

Soyut Veri Türü Mekanizmaları

- ML, infix işlevleri adı verilen kullanıcı tanımlı operatörlere izin verir
  - Özel semboller kullanabilir
  - Standart operatör sembolleri yeniden kullanılamaz
- Örnek: karmaşık sayıdaki toplama operatörünü +: öncelik düzeyi 6 olan bir infix operatörü olarak tanımladık (yerleşik toplama operatörleriyle aynı)

Modüller

- Complex türü aşağıdaki gibi kullanılabilir

```
- val z = makecomplex (1.0, 2.0);
val z = - : Complex
- val w = makecomplex (2.0, ~1.0); (* ~, negatif anlamında *)
val w = - : Complex
- val x = z +: w;
val x = - : Complex
- realpart x;
val it = 3.0 : real
- imaginarypart x;
val it = 1.0 : real
```

Modüller

- Saf bir ADT mekanizması, ADT benzeri bir soyutlama mekanizmasının bir dilde yararlı olduğu tüm durumlara hitap etmez.
- Yakından ilişkili olan ve uygulama ayrıntılarını gizleyen bir dizi standart işlevin tanımlarını ve uygulamalarını özetlemek mantıklıdır.
  - Böyle bir paket, bir veri türü ile doğrudan ilişkili değildir ve bir ADT mekanizmasının formatına uymaz.

Modüller

- Örnek: bir derleyici, ayrı parçalardan oluşan bir kümedir

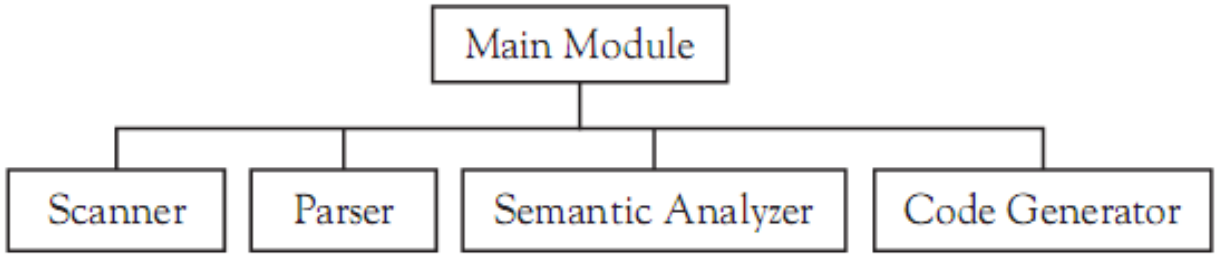


Figure 11.3 Parts of a programming language compiler

- **Modül(Module)**: ortak(public) bir arayüze ve özel(private) bir uygulamaya sahip bir program birimi
- Bir hizmet sağlayıcısı olarak modüller, herhangi bir veri türü, prosedür, değişken ve sabit karışımını dışa aktarabilir

Modüller

- Modüller **isim çoğalmasının(name proliferation)** kontrolüne yardımcı olur
  - Genellikle ek kapsam özellikleri sağlarlar
- Bir modül yalnızca arayüzünün gerektirdiği isimleri dışa aktarır ve diğerlerini gizli tutar.
- Yanlışlıkla isim çatışmalarını önlemek için isimler modül adına göre **nitelendirilir(qualified)**
  - Tipik olarak nokta gösterimi kullanılarak yapılır
- Bir modül, diğer modüllerden gelen kod kullanıldığında açık içe aktarma listeleri gerektirerek diğer modüllere olan bağımlılıkları belgeleyebilir.

C ve C++'da Ayrı Derleme

- C herhangi bir modül mekanizmasına sahip değildir
  - Modülleri simüle etmek için kullanılabilen ayrı derleme ve ad kontrol özelliklerine sahiptir
- C'de bir kuyruk veri yapısının tipik organizasyonu:
  - Bir başlık dosyasındaki tür ve işlev özellikleri `queue.h`, tür tanımlarını ve gövdeleri olmayan fonksiyon bildirimlerini (prototipler olarak adlandırılır) içerir.
  - Bu dosya, C ön işlemci yönergesi `#include` kullanılarak istemci koduna ve uygulama koduna metin olarak dahil edilerek ADT kuyruğunun bir özelliği olarak kullanılır.

C ve C++'da Ayrı Derleme

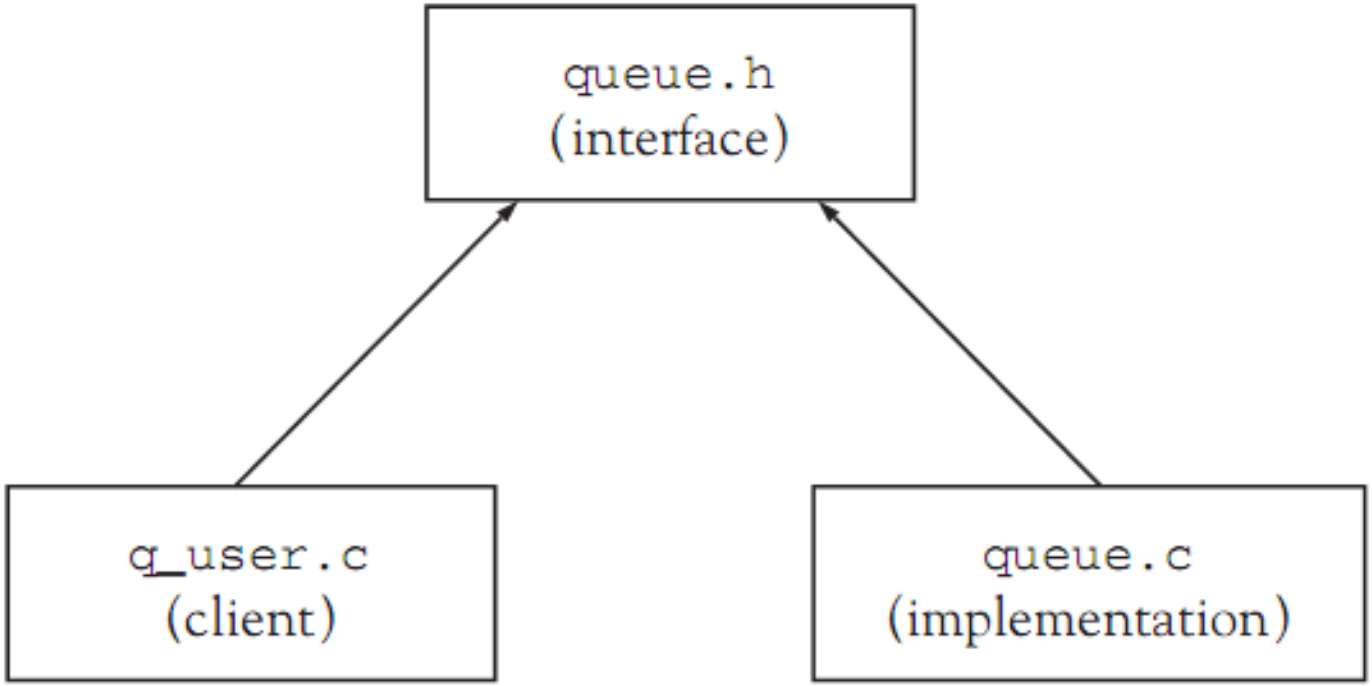


Figure 11.4 Separation of specification, implementation, and client code

Şekil: Tanımlama, uygulama ve kullanıcı kodunun ayrılması

C ve C++'da Ayrı Derleme

```
#ifndef QUEUE_H
#define QUEUE_H
struct Queuerep;
typedef struct Queuerep * Queue;
Queue createq(void);
Queue enqueue(Queue q, void* elem);
void* frontq(Queue q);
Queue dequeue(Queue q);
int emptyq(Queue q);
#endif
```

Şekil: C'de bir queue.h başlık dosyası

C ve C++'da Ayrı Derleme

- Kuyruk veri türünün tanımı, kuyruğu bir işaretçi türü olarak tanımlayarak uygulamada gizlenir
  - Gerçek kuyruk temsil yapısını **tamamlanmamış bir tür(incomplete type)** olarak bırakır
  - Başlık dosyasında tüm kuyruk yapısının bildirilmesi ihtiyacını ortadan kaldırır
- Bu mekanizmanın etkinliği yalnızca sözleşmeye bağlıdır
  - Ne derleyiciler ne de bağlayıcılar, güncel olmayan kaynak kodu için herhangi bir koruma veya denetim uygulamaz

C++ Ad Alanları ve Java Paketleri

- C++'daki `namespace` mekanizması, C'deki modüllerin simülasyonu için destek sağlar
  - Açıkça adlandırılmış bir kapsamın girişine izin verir
  - Ayrı olarak derlenmiş kitaplıklar arasında isim çatışmalarını önlemeye yardımcı olur
- Ad alanını kullanmanın üç yolu:
  - Kapsam çözümleme operatörünü kullanmak (`::`)
  - İsim alanındaki her isim için bir `using` bildirimi yazmak
  - Tek bir `using namespace` bildirimi kullanarak tüm adların niteliğini kaldırmak

C++ Ad Alanları ve Java Paketleri

```
#ifndef QUEUE_H
#define QUEUE_H
namespace MyQueue {
    struct Queuerep;
    typedef Queuerep * Queue; // C++'da struct'a ihtiyaç yok
    Queue createq();
    Queue enqueue(Queue q, void* elem);
    void* frontq(Queue q);
    Queue dequeue(Queue q);
    int emptyq(Queue q);
}
#endif
```

Şekil: C++'da bir queue.h başlık dosyası

C++ Ad Alanları ve Java Paketleri

- Java'nın **paket(package)** adı verilen ad alanına benzer bir mekanizması vardır:
  - Bir grup ilgili sınıf
- Bir paketteki bir sınıfa şu şekilde referans verebilir:
  - Sınıf adını noktalı gösterimle nitelendirme
  - Sınıf veya paketin tamamı için bir `import` bildirimi kullanma
- Java derleyicisi, arama yolu kullanılarak bulunabilen diğer tüm genel Java kodlarına erişebilir
- Derleyici, güncel olmayan kaynak dosyaları kontrol edecek ve tüm bağımlı dosyaları otomatik olarak yeniden derleyecektir.

Ada Paketleri

- Ada'nın modül mekanizması **pakettir(package)**
  - Modülleri ve parametrik polimorfizmi uygulamak için kullanılır
- Paket iki bölüme ayrılmıştır:
  - **Paket belirtimi(Package specification)**: paketin genel arabirimi ve bir ADT'nin imzasına karşılık gelir
  - **Paket gövdesi(Package body)**
- Paket belirtimi ve paket gövdeleri Ada'daki derleme birimlerini temsil eder ve ayrı olarak derlenebilir

Ada Paketleri

```
package ComplexNumbers is
type Complex is private;
function "+"(x,y: in Complex) return Complex;
function "-"(x,y: in Complex) return Complex;
function "*"(x,y: in Complex) return Complex;
function "/"(x,y: in Complex) return Complex;
function "-"(z: in Complex) return Complex;
function makeComplex (x,y in Float) return Complex;
function realPart (z: in Complex) return Float;
function imaginaryPart (z: in Complex) return Float;
private
   type Complex is
   record
      re, im: Float;
   end record;
end ComplexNumbers;
```

Şekil: Ada'da karmaşık sayılar için bir paket belirtimi

Ada Paketleri

- **private** bölümdeki herhangi bir bildirime bir kullanıcı/istemci erişemez
- Tür adları bir belirtimin genel(public) bölümünde verilebilir, ancak gerçek tür bildirimi belirtimin private bölümünde verilmelidir.
- Bu, soyut veri türü mekanizmaları için iki kriteri ihlal eder:
  - Belirtim(specification) uygulamaya bağlıdır
  - Uygulama ayrıntıları, belirtim ve uygulama arasında bölünmüştür

Ada Paketleri

- Ada'daki paketler, C++ anlamında otomatik olarak ad alanlarıdır(namespaces).
- Ada, paket adını otomatik olarak kaldıran, C++'ın `using` bildirimine benzer bir `use` bildirimine sahiptir.
- **Genel paketler(Generic packages)**: parametreli türleri uygular

Ada Paketleri

```
generic
   type T is private;
package Queues is
   type Queue is private;
   function createq return Queue;
   function enqueue(q:Queue;elem:T) return Queue;
   function frontq(q:Queue) return T;
   function dequeue(q:Queue) return Queue;
   function emptyq(q:Queue) return Boolean;
private
   type Queuerep;
   type Queue is access Queuerep;
end Queues;
```

Şekil: Ada genel paket belirtimi ile parametreli kuyruk ADT

ML'deki Modüller

- Soyut tanıma ek olarak ML, üç mekanizmadan oluşan daha genel bir modül özelliğine sahiptir:
  - **İmza(Signature)**: bir arayüz tanımı
  - **Yapı(Structure)**: imzanın bir uygulaması
  - **Fonksiyonlar(Functions)**: imzalarla verilen "türlere" sahip yapı parametreleriyle yapılardan yapılara fonksiyonlar
- İmzalar, `sig` ve `end` anahtar kelimeleri kullanılarak tanımlanır

ML'deki Modüller

```
signature QUEUE =
sig
  type 'a Queue
  val createq: 'a Queue
  val enqueue: 'a Queue * 'a -> 'a Queue
  val frontq: 'a Queue -> 'a
  val dequeue: 'a Queue -> 'a Queue
  val emptyq: 'a Queue -> bool
end;
```

Şekil: ML'de kuyruk ADT için bir QUEUE imzası

ML'deki Modüller

```
structure Queue1: QUEUE =
struct
  datatype 'a Queue = Q of 'a list
  val createq = Q [];
  fun enqueue(Q lis, elem) = Q (lis @ [elem]);
  fun frontq (Q lis) = hd lis;
  fun dequeue (Q lis) = Q (tl lis);
  fun emptyq (Q []) = true
    | emptyq (Q (h::t)) = false;
end;
```

Şekil: ML'de yerleşik listeler kullanarak QUEUE imzasını gerçekleştiren Queue1 yapısı

ML'deki Modüller

```
structure Queue2: QUEUE =
struct
  datatype 'a Queue = Createq | Enqueue of 'a Queue * 'a;
  val createq = Createq;
  fun enqueue(q, elem) = Enqueue (q, elem);
  fun frontq (Enqueue(Createq, elem)) = elem | frontq
  ↪  (Enqueue(q, elem)) = frontq q
  fun dequeue (Enqueue(Createq, elem)) = Createq | dequeue
  ↪  (Enqueue(q, elem)) = Enqueue(dequeue q, elem)
  fun emptyq Createq = true | emptyq _ = false;
end;
```

Şekil: ML'de kullanıcı tanımlı bağlı listeler kullanarak QUEUE imzasını gerçekleştiren Queue2 yapısı

ML'deki Modüller

- ML imzaları ve yapıları, soyut veri türleri için gereksinimlerin çoğunu karşılar
- Ana zorluk, istemci kodunun, modül adı açısından kullanılacak uygulamayı açıkça belirtmesi gerektiğidir.
  - Kod yalnızca imzaya bağlı olarak yazılamaz, gerçek uygulama yapısı koda harici olarak sağlanacak
  - Bunun nedeni, ML'nin açık veya örtülü ayrı bir derleme veya kod toplama mekanizmasına sahip olmamasıdır.

Daha Önceki Dillerdeki Modüller

- Tarihsel olarak, modüller ve soyut veri tipi mekanizmalar Simula67 ile başlamıştır.
- Ada ve ML'deki modül mekanizmalarına önemli ölçüde katkıda bulunan diller arasında CLU, Euclid, Modula-2, Mesa ve Cedar bulunur.

Euclid

- Euclid programlama dilinde, modüller türlerdir.
- Kullanılacak türden gerçek bir nesne bildirilmelidir
- Bir bildirimde modül türleri kullanıldığında, modül türünün bir değişkeni oluşturulur veya **somutlaştırılır(instantiated)**.
- Bir modülün aynı anda iki farklı örneğine sahip olabilir
- Bu, modüllerin türler yerine nesneler olduğu Ada veya ML'den farklıdır ve her birinin tek bir örneğini içerir.

Euclid

```
type ComplexNumbers = module
   exports(Complex, add, subtract, multiply, divide, negate, makeComplex,
   ↪  realPart, imaginaryPart)
   type Complex = record
      var re, im: real
    end Complex

    procedure add (x,y: Complex, var z: Complex) =
    begin
        z.re := x.re + y.re
        z.im := x.im + y.im
    end add

    procedure makeComplex (x,y: real, var z:Complex) =
    begin
        z.re := x
        z.im := y
    end makeComplex
    ...
end ComplexNumbers
```

Euclid

```
var C1,C2: ComplexNumbers
var x: C1.Complex
var y: C2.Complex

C1.makeComplex(1.0, 0.0, x)

C2.makeComplex(0.0, 1.0, y)
(* x ve y birlikte toplanamaz *)
```

CLU

- CLU'da modüller **küme(cluster)** mekanizması kullanılarak tanımlanır
- Veri türü doğrudan bir küme olarak tanımlanır
- Bir değişken tanımladığımızda, türü bir küme değil, **rep** bildirimi tarafından verilendir.
- CLU'daki bir küme iki farklı şeyi ifade eder:
  - Kümenin kendisi
  - Dahili temsil türü

CLU

```
Complex = cluster is add, multiply, ..., makeComplex, realPart,
↪   imaginaryPart
    rep = struct [re,im: real]
    add = proc (x,y: cvt) returns (cvt)
      return
      (rep${re: x.re+y.re, im: x.im+y.im})
    end add
    ...
    makeComplex = proc (x,y: real) returns (cvt)
      return (rep${re:x, im:y})
    realPart = proc(x: xvt) returns (real)
      return(x.re)
    end realPart
end Complex
```

CLU

- `cvt` (`convert` için) harici türden (açık bir yapı olmadan) dahili rep türüne ve tekrar geri dönüştürür

```
max(2.1,3); // which max?
```

Modula-2

- Modula-2'de, soyut bir veri türünün özellikleri ve uygulanması, bir tanım modülüne(DEFINITION MODULE) ve bir uygulama modülüne(IMPLEMENTATION MODULE) ayrılır.
- Tanım Modülü: Sadece tanımları veya bildirimleri içerir
  - Bunlar dışa aktarılan edilen tek beyanlardır (diğer modüller tarafından kullanılabilir)

  Uygulama Modülü: Uygulama kodunu içerir

Modula-2

```
DEFINITION MODULE ComplexNumbers;
TYPE Complex;
PROCEDURE Add (x,y: Complex): Complex;
PROCEDURE Subtract (x,y: Complex): Complex;
PROCEDURE Multiply (x,y: Complex): Complex;
PROCEDURE Divide (x,y: Complex): Complex;
PROCEDURE Negate (z: Complex): Complex;
PROCEDURE MakeComplex (x,y: REAL): Complex;
PROCEDURE RealPart (z: Complex) : REAL;
PROCEDURE ImaginaryPart (z: Complex) : REAL;
END ComplexNumbers.
```

Modula-2

- Bir istemci modülü, veri türünü ve işlevlerini modülünden içe aktararak kullanır.
- Modula-2, referans kaldıran FROM yüklemini kullanır
  - İçe aktarılan öğeler, IMPORT bildiriminde isme göre listelenmelidir.
  - Başka hiçbir eleman (içe aktarılan veya yerel olarak bildirilen), içe aktarılanlarla aynı isimlere sahip olmayabilir.

Soyut Veri Türü Mekanizmalarıyla İlgili Sorunlar

- Soyut veri türü mekanizmaları, koruma ve uygulama bağımsızlık gereksinimlerini karşılamak için ayrı derleme tesisleri kullanır
- ADT Mekanizması, kullanım ve uygulama tutarlılığını garanti altına almak için bir arayüz olarak kullanılır.
- Ancak, ADT mekanizmaları türler oluşturmak ve işlemleri türlerle ilişkilendirmek için kullanılırken, ayrı derleme tesisleri hizmet sağlayıcılarıdır.
  - Hizmetler değişkenler, sabitler veya diğer programlama dili varlıklarını içerebilir

Soyut Veri Türü Mekanizmalarıyla İlgili Sorunlar

- Böylece, derleme birimleri bir anlamda ADT mekanizmalarından daha geneldir
- Bir türü tanımlamak için bir derleme biriminin kullanımı, türü birim ile birlikte tanımlamadığı için daha az geneldir
  - Böylece, doğru bir tür bildirimi değil
- Ayrıca, birimler, kimliklerini yalnızca bağlamadan önce tutan statik varlıklardır.
  - Tahsis ve başlatma sorunları ile sonuçlanabilir

Soyut Veri Türü Mekanizmalarıyla İlgili Sorunlar

- Soyut veri türlerini uygulamak için ayrı derleme birimlerini kullanmak bu nedenle dil tasarımında bir uzlaşmadır.
- Bu yararlı bir uzlaşma
  - Tutarlılık kontrolü ve bağlantıdan birine olan ADT'ler için uygulama sorusunu azaltır

Modüller Tür Değildir

- C, Ada ve ML'de, bir modülün bir türü ve işlemleri dışa aktarması gerektiği için problemler ortaya çıkar.
- Bir modülü tür olarak tanımlamak faydalı olurdu
  - Uygulama ayrıntılarını eksik veya özel(private) bildirimler gibi bir geçici mekanizma ile korumak için düzenleme gereksinimini önler
- ML, hem `abstype` hem de bir modül mekanizması içeren bu ayrımı yapar
- Modül mekanizması daha geneldir, ancak bir tür dışa aktarılmalıdır.

Modüller Tür Değildir

- `abstype` bir veri türüdür, ancak uygulaması tanımından ayrılamaz
  - Uygulamanın detaylarına erişim engellenir
- `abstype` kullanıcıları dolaylı olarak uygulamaya bağlıdır

Modüller Statik Varlıklardır

- Soyut bir veri türünü uygulamak için çekici bir olasılık, bir türü hiç ortaya çıkarmamaktır.
  - Uygulama detaylarına herhangi bir şekilde bağlı olan istemci olasılığını ortadan kaldırır
  - İstemcilerin bir türü kötüye kullanmasını önler
- Ada'da, gerçek veri türünün uygulamada gömülü olduğu bir paket belirtimi oluşturabilir
  - Bu saf zorunlu(imperative) programlamadır

Modüller Statik Varlıklardır

- Normalde bu, istemcide o veri türünden yalnızca bir varlığın olabileceği anlamına gelir.
  - Aksi takdirde, kodun tamamı çoğaltılmalıdır.
- Bu, çoğu modül mekanizmasının statik doğasından kaynaklanmaktadır
- Ada'da, genel paket mekanizması, aynı genel paketin birden çok örneğini kullanarak aynı türden birkaç varlığı elde etmenin bir yolunu sunar.

Modüller Statik Varlıklardır

```
generic
   type T is private;
package Queues is
   procedure enqueue(elem:T);
   function frontq return T;
   procedure dequeue;
   function emptyq return Boolean;
end Queues;
```

Tür Dışa Aktaran Modüller Bu Türdeki Değişkenler Üzerindeki İşlemleri Yeterince Kontrol Etmez

- Verilen C ve Ada örneklerinde, soyut tipteki değişkenler, uygulamada bir prosedür çağrılarak tahsis edilmeli ve başlatılmalıdır.
  - Dışa aktarma modülü, değişken kullanılmadan önce başlatma prosedürünün çağrıldığını garanti edemez.
- Ayrıca, modülün kontrolü dışında kopyaların ve serbest bırakmaların yapılmasına izin verir
  - Kullanıcı sonuçların farkında olmadan
  - Ayrılmış belleği kullanılabilir depolama alanına döndürme yeteneği olmadan

Türleri Dışa Aktaran Modüller İşlemleri Kontrol Etmez

- Ada'da x:=y, y ile gösterilen nesneyi paylaşarak atamayı gerçekleştirir.
  - x=y, x ve y karmaşık sayılar olduğunda doğru olmayan işaretçi eşitliğini test eder
- Ada'da, atama ve eşitlik kullanımını kontrol etmek için bir **sınırlı özel türü(limited private type)** bir mekanizma olarak kullanabiliriz.
  - İstemcilerin olağan atama ve eşitlik işlemlerini kullanması engellenir
  - Paket, eşitliğin doğru bir şekilde gerçekleştirilmesini ve bu atamanın çöpleri serbest bırakmasını sağlar.

Türleri Dışa Aktaran Modüller İşlemleri Kontrol Etmez

```
package ComplexNumbers is
type Complex is limited private;
-- atama ve eşitliği içeren işlemler
...
function equal(x,y: in Complex) return Boolean;
procedure assign(x: out Complex; y: in Complex);
private
   type ComplexRec;
   type Complex is access ComplexRec;
end ComplexNumbers;
```

Türleri Dışa Aktaran Modüller İşlemleri Kontrol Etmez

- C++ atama ve eşitliğin aşırı yüklenmesine izin verir
- Nesne yönelimli diller, başlatma problemini çözmek için **yapıcıları(constructors)** kullanır
- ML, bir `abstype` veya `struct` belirtimindeki veri türünü eşitlik işlemine izin vermeyen türlerle sınırlar
  - Eşitlik testine izin veren tür parametreleri, tek bir kesme işareti `'a` yerine çift kesme işareti `''a` ile yazılmalıdır.

Türleri Dışa Aktaran Modüller İşlemleri Kontrol Etmez

- ML'de eşitliğe izin veren türler `eqtype` olarak belirtilmelidir.
  - Örnek:

```
signature QUEUE =
sig
  eqtype ''a Queue
  val createq: ''a Queue
  ...
end;
```

Modüller, İçe Aktarılan Türlere Bağımlılıklarını Her Zaman Yeterli Olarak Temsil Etmez

- Modüller genellikle tür parametreleri üzerinde belirli işlemlerin varlığına bağlıdır.
  - Modül belirtiminde varlığı açıkça belirtilmeyen işlevleri de çağırabilir
- Örnek: ikili arama ağacı, öncelik kuyruğu(priority queue) veya sıralı liste gibi veri yapılarının tümü, aritmetik küçüktür işlemi "$<$" gibi bir sıra işlemi gerektirir
- C++ şablonları, özelliklerdeki bu tür bağımlılıkları maskeler

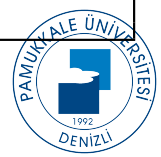
Modüller Her Zaman Bağımlılıklarını Temsil Etmez

- C++ kodunda örnek:
  - Şablon `min` fonksiyonu tanımı

```
template <typename T>
T min(T x, T y);
```

  - Gerçekleştirme, bağımlılığı gösterir

```
template <typename T>
T min(T x, T y)
//T üzerindeki < işlemine ihtiyaç duyar
{
    return x < y ? x : y;
}
```



Modüller Her Zaman Bağımlılıklarını Temsil Etmez

- Ada'da, bir paket bildiriminin genel bölümündeki ek bildirimleri kullanarak bu gereksinim belirtilebilir:

```
generic
   type Element is private;
   with function lessThan (x,y: Element) return Boolean;
package OrderedList is
...
end OrderedList;
```

- Örnekleme, `lessThan` işlevini sağlamalıdır:

```
package IntOrderedList is new
   OrderedList (Integer,"<");
```

Modüller Her Zaman Bağımlılıklarını Temsil Etmez

- Böyle bir gereksinime **kısıtlı parametreleştirme(constrained parameterization)** denir
- ML, yapıların diğer yapılar tarafından açıkça parametrelendirilmesine izin verir
  - Bu özelliğe **functor** (yapılar üzerinde bir fonksiyon) denir.

```
functor OListFUN (structure Order : ORDER):
↪   ORDERED_LIST =
struct
  ...
end
```

Modüller Her Zaman Bağımlılıklarını Temsil Etmez

- functor, yeni bir yapı oluşturmak için uygulanabilir:

```
structure IntOList = OListFUN(structure Order = IntOrder);
```

- Bu, gerekli özellikleri kapsayan ekstra bir yapının tanımlanmasını gerektirme pahasına, uygun bağımlılıkları açık hale getirir

Modüller Her Zaman Bağımlılıklarını Temsil Etmez

```
1  signature ORDER =
2  sig
3    type Elem
4    val lt: Elem * Elem -> bool
5  end;
6  signature ORDERED_LIST =
7  sig
8    type Elem
9    type OList
10   val create: OList
11   val insert: OList * Elem -> OList
12   val lookup: OList * Elem -> bool
13 end;
14 functor OListFUN (structure Order : ORDER): ORDERED_LIST =
15 struct
16   type Elem = Order.Elem;
17   type OList = Order.Elem list;
18   val create = [];
19   fun insert ([], x) = [x] | insert (h::t, x) = if Order.lt(x, h) then
     ↪  x::h::t else h:: insert(t, x);
```

Modüller Her Zaman Bağımlılıklarını Temsil Etmez

```
20    fun lookup ([], x) = false | lookup (h::t, x) = if Order.lt(x, h) then
      ↪  false else if Order.lt(h, x) then lookup (t, x) else true;
21  end;
22  structure IntOrder: ORDER =
23  struct
24    type Elem = int;
25    val lt = (op <);
26  end;
27  structure IntOList = OlistFun(structure Order = IntOrder)
```

Şekil: ML'de bir sıralı liste tanımlamak için functor kullanımı

Modül Tanımları, Sağlanan İşlemlerin Semantiğinin Belirtilmesini İçermez

- Hemen hemen tüm dillerde, soyut bir veri türünün mevcut işlemlerinin davranışının belirtilmesi gerekmez.
- Eiffel nesneye yönelik programlama dili, semantiğin belirlenmesine izin verir
  - Anlamsal özellikler ön koşullar, son koşullar ve değişmezler tarafından verilir
- Ön koşullar ve son koşullar, bir prosedürün yürütülmesinden önce ve sonra neyin doğru olması gerektiğini belirler.

Modül Tanımları, Sağlanan İşlemlerin Semantiğinin Belirtilmesini İçermez

- Değişmezler, soyut bir veri türünde verilerin iç durumu hakkında neyin doğru olması gerektiğini belirler.
- Örnek: Eiffel'deki kuyruğa ekleme işlemi:

```
enqueue (x:element) is
require
    not full
ensure
    if old empty then front = x
    else front = old front;
    not empty
end; -- enqueue
```

Modül Tanımları Semantik Spesifikasyonu İçermez

- `require` bölümü ön koşulları belirler
- `ensure` bölümü son koşulları belirler
- Bu gereksinimler cebirsel aksiyomlara karşılık gelir:

frontq(enqueue(q,x)) = if emptyq(q) then x else frontq(q)
emptyq(enqueue(q,x)) = false

Soyut Veri Türlerinin Matematiği

- Soyut bir veri türünün **varoluşsal türe(existential type)** sahip olduğu söylenir.
  - Gereksinimlerini karşılayan gerçek bir türün varlığını ileri sürer
- Gerçek bir tür, uygun biçimdeki işlemleri içeren bir kümedir.
  - Belirtimi(Specification) karşılayan bir küme ve işlemler, belirtim için bir **modeldir**.
- Hiçbir modelin olmaması veya birçok modelin olması mümkündür.

Soyut Veri Türlerinin Matematiği

- Potansiyel türlere çeşitler(sorts), potansiyel işlem kümelerine ise imza adı verilir.
  - Dolayısıyla bir çeşit, herhangi bir gerçek değer kümesiyle henüz ilişkilendirilmemiş bir türün adıdır.
  - İmza, yalnızca teoride var olan bir işlemin veya işlem kümesinin adı ve türüdür.
- O halde bir model, bir türün gerçekleşmesi ve imzasıdır ve cebir olarak adlandırılır.
- Cebirsel özellikler genellikle çeşit-imza terminolojisi kullanılarak yazılır.

Soyut Veri Türlerinin Matematiği

sort queue(element) **imports** boolean

signature:

createq: queue
enqueue: queue × element → queue
dequeue: queue → queue
frontq: queue → element
emptyq: queue → boolean

axioms:

emptyq(createq) = true
emptyq (enqueue (q, x)) = false
frontq(createq) = error
frontq(enqueue(q,x)) = if emptyq(q) then x else frontq(q)
dequeue(createq) = error
dequeue(enqueue(q,x)) = if emptyq(q) then q else enqueue(dequeue(q), x)

Soyut Veri Türlerinin Matematiği

- Türü temsil edecek spesifikasyon için benzersiz bir cebir oluşturabilmek istiyoruz.
- Bunu yapmak için standart yöntem:
  - Bir sıralama için terimlerin **serbest cebirini(free algebra)** oluşturmak
  - Denklem aksiyomları tarafından üretilen denklik ilişkisinin **bölüm cebirini(quotient algebra)** oluşturmak
- Terimlerin serbest cebiri, tüm yasal/geçerli işlem kombinasyonlarından oluşur.

Soyut Veri Türlerinin Matematiği

- Örnek: sıralama kuyruğu (tamsayı) için serbest cebir ve daha önce gösterilen imza şunları içerir:

  createq
  enqueue (createq, 2)
  enqueue (enqueue(createq, 2), 1)
  dequeue (enqueue (createq, 2))
  dequeue (enqueue(enqueue (createq, 2), -1))
  dequeue (dequeue (enqueue (createq, 3)))
  etc.

- Bir kuyruğa ilişkin aksiyomların bazı terimlerin aslında eşit olduğunu ima ettiğini unutmayın:

  dequeue (enqueue (createq, 2)) = createq

Soyut Veri Türlerinin Matematiği

- Serbest cebirde hiçbir aksiyom doğru değildir
  - Bunları doğru yapmak için (belirtimi modelleyen bir tür oluşturmak için), serbest cebirdeki farklı öğelerin sayısını azaltmak için aksiyomlar kullanmalıdır.
- Bu, aksiyomlardan bir **denklik ilişkisi(equivalence relation)** $==$ kurarak yapılabilir.
  - "$==$" simetrik, geçişli(transitive) ve dönüşlü(reflexive) ise bir denklik bağıntısıdır:

    if $x == y$ then $y == x$ (symmetry)

    if $x == y$ and $y == z$ then $x == z$ (transitivity)

    $x == x$ (reflexivity)

Soyut Veri Türlerinin Matematiği

- Bir denklik bağıntısı == ve serbest bir F cebir verildiğinde, benzersiz, iyi tanımlanmış bir F/== cebiri vardır, öyle ki, F/=='de x=y ancak ve ancak F/==de x==y ise
  - F/== cebiri, F'nin bölüm cebiri(quotient algebra) olarak adlandırılır ==
  - Her denklemin iki tarafını eşit ve dolayısıyla bölüm cebirinde eşit yapan benzersiz bir "en küçük" denklik ilişkisi vardır.
- Bölüm cebiri genellikle bir cebirsel belirtim tarafından tanımlanan veri türü olarak alınır.

Soyut Veri Türlerinin Matematiği

- Bu cebir, eşit olan tek terimin aksiyomlardan kanıtlanabilir şekilde eşit olan terimler olma özelliğine sahiptir.
- Bu cebire, belirtim tarafından temsil edilen **başlangıç cebiri(initial algebra)** denir.
  - Bunu kullanmak, **ilk anlambilim(initial semantics)** ile sonuçlanır.
- Genel olarak, aksiyom sistemleri tutarlı ve eksiksiz olmalıdır.
  - İstenen bir başka özellik de bağımsızlıktır: diğer aksiyomlar hiçbir aksiyomu ima etmez.

Soyut Veri Türlerinin Matematiği

- Uygun bir aksiyom kümesine karar vermek genellikle zor bir süreçtir.
- **Nihai cebir(Final algebra)**: Denetçi işlemleriyle ayırt edilemeyen herhangi iki veri değerinin eşit olması gerektiğini varsayan bir yaklaşım
  - İlişkili anlambilim, **nihai anlambilim(final semantics)** olarak adlandırılır.
- Nihai bir cebir de esasen benzersizdir
- Matematikte genişleme ilkesi(principle of extensionality):
  - Tüm bileşenleri eşit olduğunda iki şey tam olarak eşittir