Alkalmazott Modul III

1. előadás

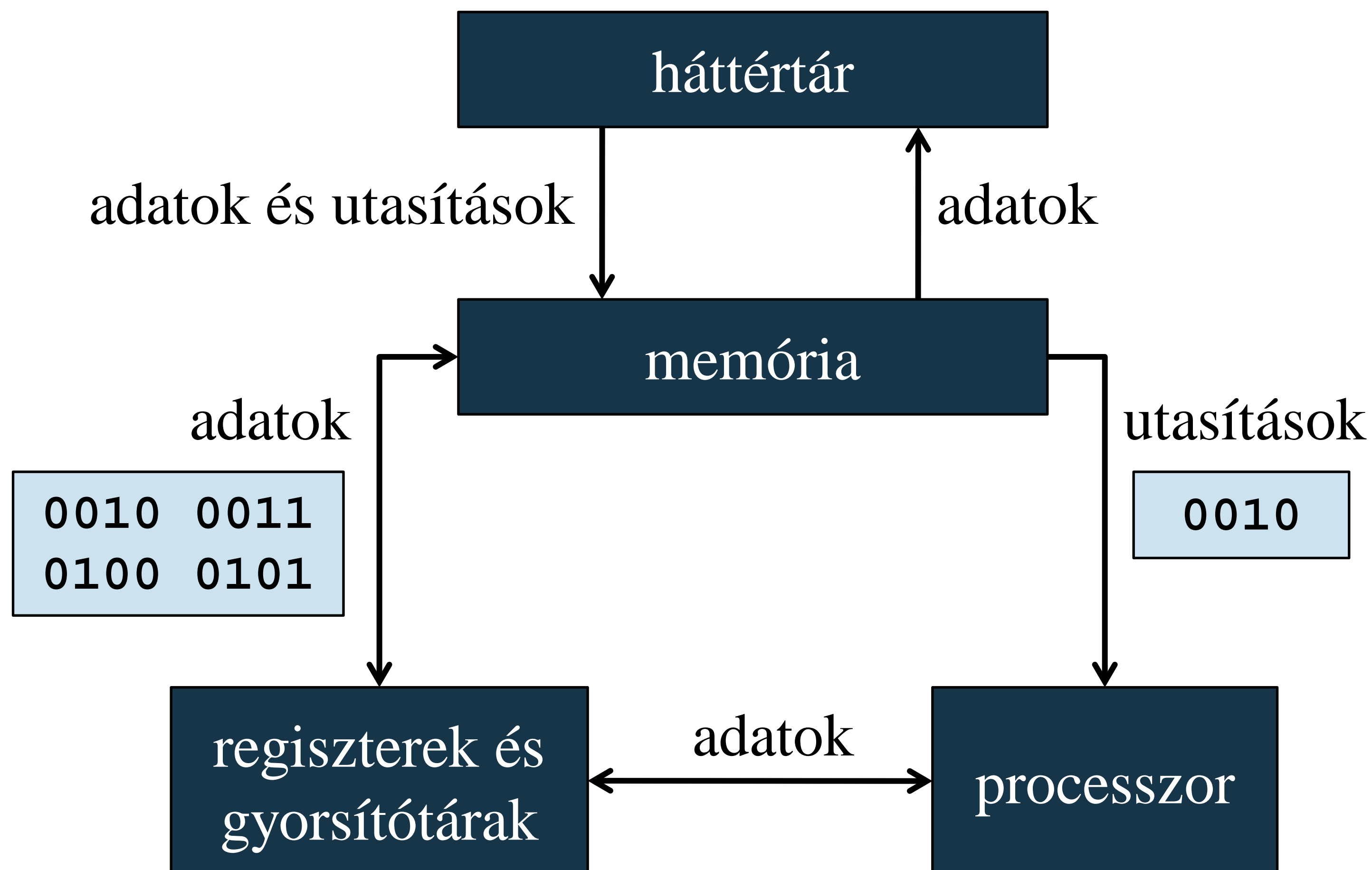
Szoftverfejlesztés, programozási paradigmák

© 2011.09.19. Giachetta Roberto
groberto@inf.elte.hu
http://people.inf.elte.hu/groberto

Szoftverfejlesztés

A program

- A *program*
  - *matematikailag*: állapotterek (értékek direktszorzata) felett értelmezett reláció
  - *informatikailag*: utasítások sorozata, amelyek műveleteket hajtanak végre a megadott értékekkel, az *adat*okkal
- A programban foglalt utasítássorozatot, vagy *programkód*ot a *processzor* (CPU, GPU, …) hajtja végre
  - a processzor korlátozott utasításkészlettel rendelkezik, ezért összetett utasításokat nem képes véghezvinni
  - a végrehajtáshoz segédtárakat (regiszterek, gyorsítótárak) használ, és kommunikál a *memóriá*val
  - az utasítások és adatok binárisan vannak eltárolva



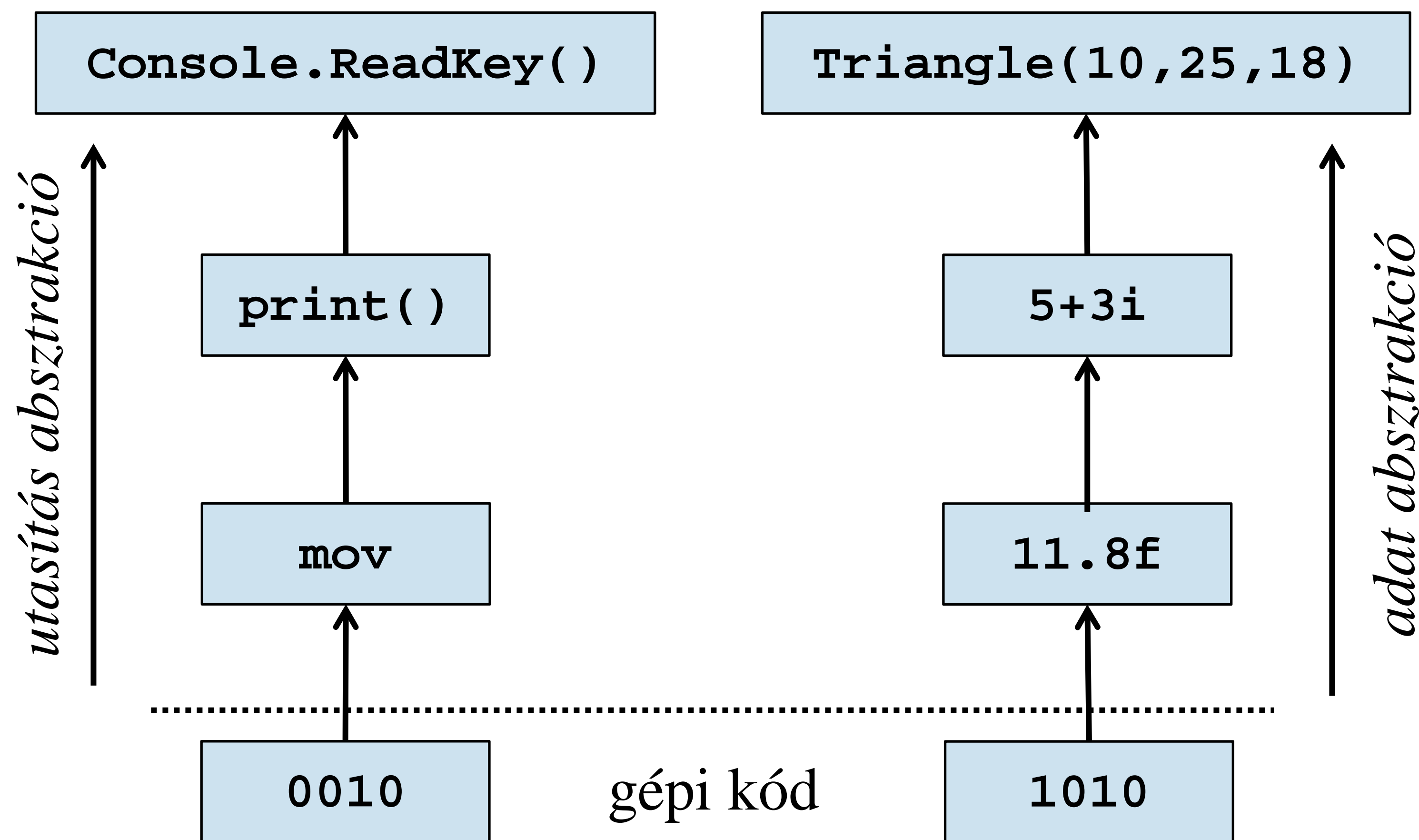
háttértár
adatok és utasítások
adatok
memória
adatok
0010 0011
0100 0101
utasítások
0010
regiszterek és
gyorsítótárak
adatok
processzor

Absztrakció a programkészítésben

- A processzor által értelmezhető utasításkészletet és adathalmazt nevezzük *gépi kód*nak (*object code*)
- Mivel a programokat nem tudjuk közvetlenül a processzor feldolgozási szintjén elkészíteni, szükségünk van a működés és az adatkezelés absztrakciójára:
  - az *utasításabsztrakció* (*control abstraction*) biztosítja, hogy a processzor egyszerű, egymást követő utasításai (összeadás, mozgatás, összehasonlítás) mellett összetett parancsokat és vezérlési módszert fogalmazzunk meg
  - az *adatabsztrakció* (*data abstraction*) lehetővé teszi, hogy különböző típusokba soroljuk adatainkat, amelyek meghatározzák az értéktartományt, és a végezhető műveleteket

Szoftverfejlesztés

Absztrakció a programkészítésben



- Az adatabsztrakcióhoz az adatok kétféle fajtáját tarjuk nyilván:
  - *konstans* (*constant*): értéke és típusa rögzített, nem változhat a program futása során
  - *változó* (*variable*): értéke (esetleg típusa is) változtatható a program futása során, mindig a memóriában tároljuk, és megfelelő *azonosító*val (változónév) látjuk el
- A memória tekinthető egy byte-sorozatnak, ahol minden byte-nak sorszáma van
  - ez a *memóriacím*, amelyhez rendelhetünk egy alkalmas azonosítót, amivel a program futása során hivatkozhatunk rá
  - minden futó program önálló memóriaterületet (*szegmens*t) kap, amelyen csak az ő adatai helyezkednek el

- A típus meghatározza a memóriában elfoglalt terület mértékét, az értékhalmazt, valamint az alkalmazható műveletek halmazát
  - a mindenhol megtalálható, egyszerű típusokat nevezzük *elemi*, vagy *primitív típusok*nak (pl. egész, valós, logikai, karakter, …)
  - a programozó alkothat saját, *összetett típusok*at a *típuskonstrukció*k (direktszorzat, iterált, unió) segítségével már létező típusokból kiindulva, így tetszőlegesen összetett adatokat tárolhat el (pl. szöveg, komplex, …)
  - a létrehozott típusokhoz különböző műveletek társíthatóak, akár többféle módon, így különböző típusokat létrehozva (pl. a tömb használható veremként, sorként, kupacként, …)

- Az utasításabsztrakció teszi lehetővé a processzor által értelmezhetőnél bonyolultabb utasítások megfogalmazását
  - a bonyolultabb utasítások egy elég nagy halmaza eleve adott, amit a fejlesztő tetszőleges mértékben bővíthet
  - lényegében a létrehozott utasítások maguk is programok, amelyek felépítéséhez a *programkonstrukció*kat használjuk: *szekvencia*, *elágazás*, *ciklus*
  - a programkonstrukciók is részei az absztrakciónak, hiszen a processzor csak utasítások szekvenciáját tudja végrehajtani, ezért elágazás és ciklus esetén a megfelelő sorrendben kell az utasításokat átadni, ami alacsonyabb szinten a programkódban történő *ugrás*ok (*goto*) segítségével történik meg

A programozási nyelv

- Az absztrakciót megvalósító eszközt nevezzük *programozási nyelv*nek
  - egy adott programozási nyelven megírt programkódot nevezünk a program *forráskód*jának (*source code*)
  - a programozási nyelv meghatározza az absztrakció szintjét, a használható típusok és utasítások halmazát, amely egy adott nyelvre rögzített, ám a programozó által általában kiterjeszthető
  - a nyelvet meghatározza a célja, vagyis milyen feladatkörre alkalmazható, továbbá a nyelv rendelkezik egy *kifejezőerő*vel, azaz milyen összetett számításokat képes kifejezni

- A programozási nyelvek osztályozása:
  - *alacsony szintű* (assembly): a gépi kódot egyszerűsíti szövegszerűre, de nem biztosít utasításabsztrakciót, pl.:

```
data segment ; adatok
   number dw -5 ; változó létrehozása
data ends
code segment ; utasítások
   …
   mov ax, number ; regiszterbe helyezése
   cmp ax, 0 ; regiszterérték összehasonlítása
   jge label1 ; ugrás, amennyiben nem negatív
   mov cx, 0
   sub cx, ax ; pozitívvá alakítás kivonással
   …
```

- *magas szintű*: a gépi architektúrától független utasításkészlettel rendelkezik, tovább egyszerűsíti az assembly kódot, és további lehetőségeket biztosít a programozó számára, pl.:

```
int main(){
   int number = -5; // változó létrehozása
   if (number < 0) // ha negatív
      number = -number; // ellentettre váltás
   …
}
```

- Fontos szempont a nyelveknél a *Turing-teljesség*, azaz adható-e a programkóddal ekvivalens Turing-gép, amely utasítások sorozatával kiszámolja bármilyen művelet eredményét

Programkód átalakítás

- A programozási nyelven megírt kódot át kell alakítani a processzor számára értelmezhetővé, erre általában két módszert alkalmazhatunk:
  - *fordítás, vagy szerkesztés* (*compilation*): a kódot előzetesen lefordítjuk gépi kódra, és a futtatható állományt indítjuk el, így az gyorsan futtatható, de gépfüggő lesz, ezt a műveletet a *fordítóprogram* (*compiler*) végzi
  - *értelmezés* (*interpretation*): a kódot a futtatás közben alakítjuk át gépi kóddá, ezért a program teljesen hordozhatóvá válik, ugyanakkor a futtatás lassú, ezt a műveletet az *értelmező* (*interpreter*) végzi
  - lehetséges a két módszer bizonyos szintű kombinálása, amelyet *futásidejű fordítás*nak nevezünk

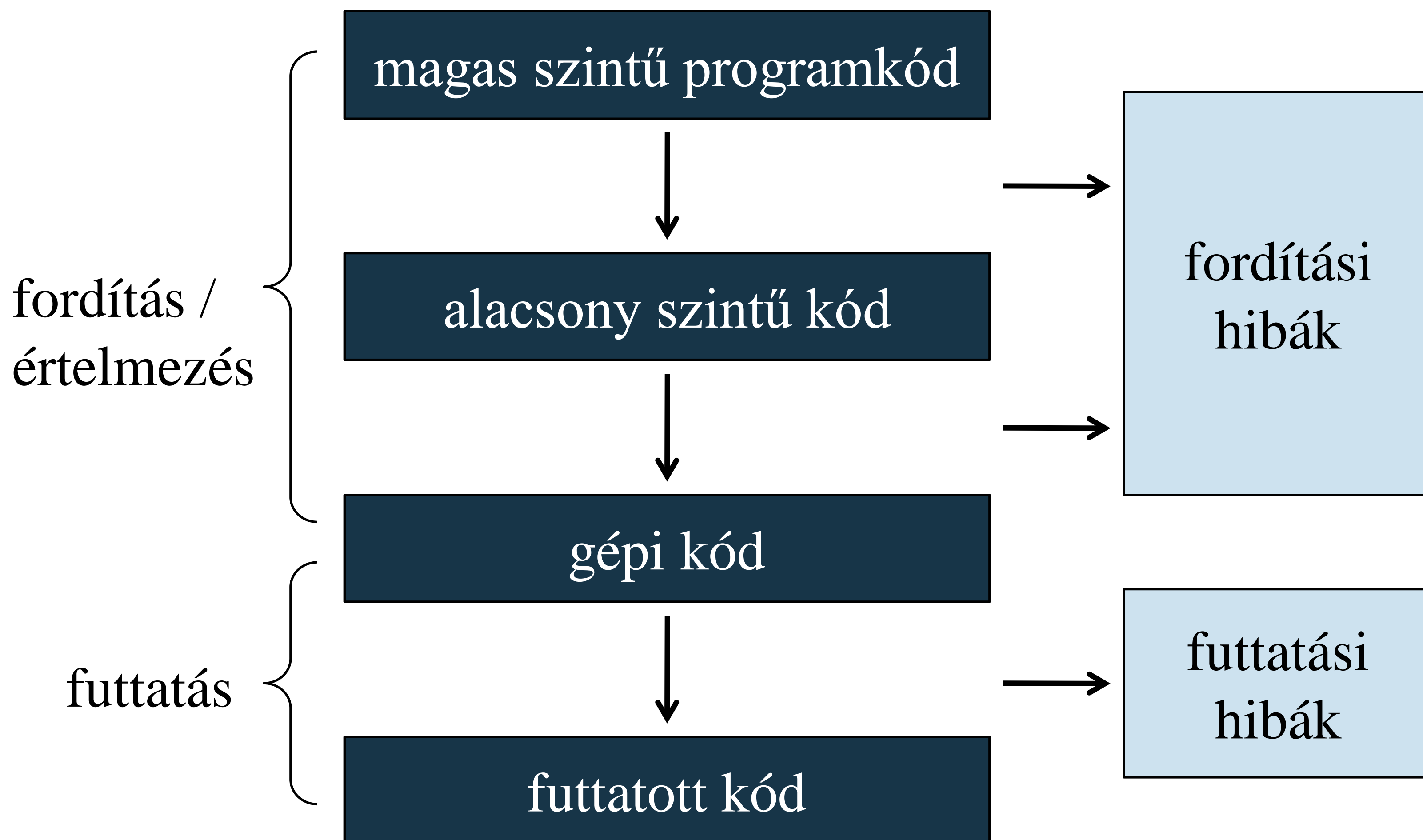
Programkód átalakítás

- A programkód átalakítása rendszerint több lépésben történik, magasabb szintű nyelv esetén először alacsonyabb szintű kód, majd abból gépi kód készül
- A programkód tartalmazhat hibákat, amelyeket két kategóriába sorolunk:
  - *szintaktikai*, *vagy elemzési hibák* (*syntax error*): a programkód szerkezete helytelen, pl. hibás utasításnév, hivatkozás, zárójelezés, …
  - *szemantikai*, *vagy értelmezési hibák* (*semantic error*): az érték változásával, a műveletek végrehajtásával bekövetkező hibák, pl. 0-val történő osztás, hibás memóriacím hivatkozás, …

Programkód átalakítás

- A programhibákról az átalakítás során a lehető legpontosabb információt kapjuk (lehetséges ok és hely megadásával)
  - a fordítás során az összes szintaktikus hiba, és a szemantikus hibák egy része ellenőrizhető, a többi hibára csak a program futtatása során derül fény
  - értelmezés esetén valamennyi hibával csak a futtatáskor szembesülünk
- A további programhibák ellenőrzését *tesztelés*sel végezhetjük
  - a *statikus tesztelés* során a programkódot vizsgáljuk át
  - a *dinamikus tesztelés* során futás közben keressük a hibákat
- A programfejlesztői környezetek megadják a *nyomkövetés* (*debug*) lehetőségét (futás közben végigkövethetjük a kódot)

Programkód transzformáció

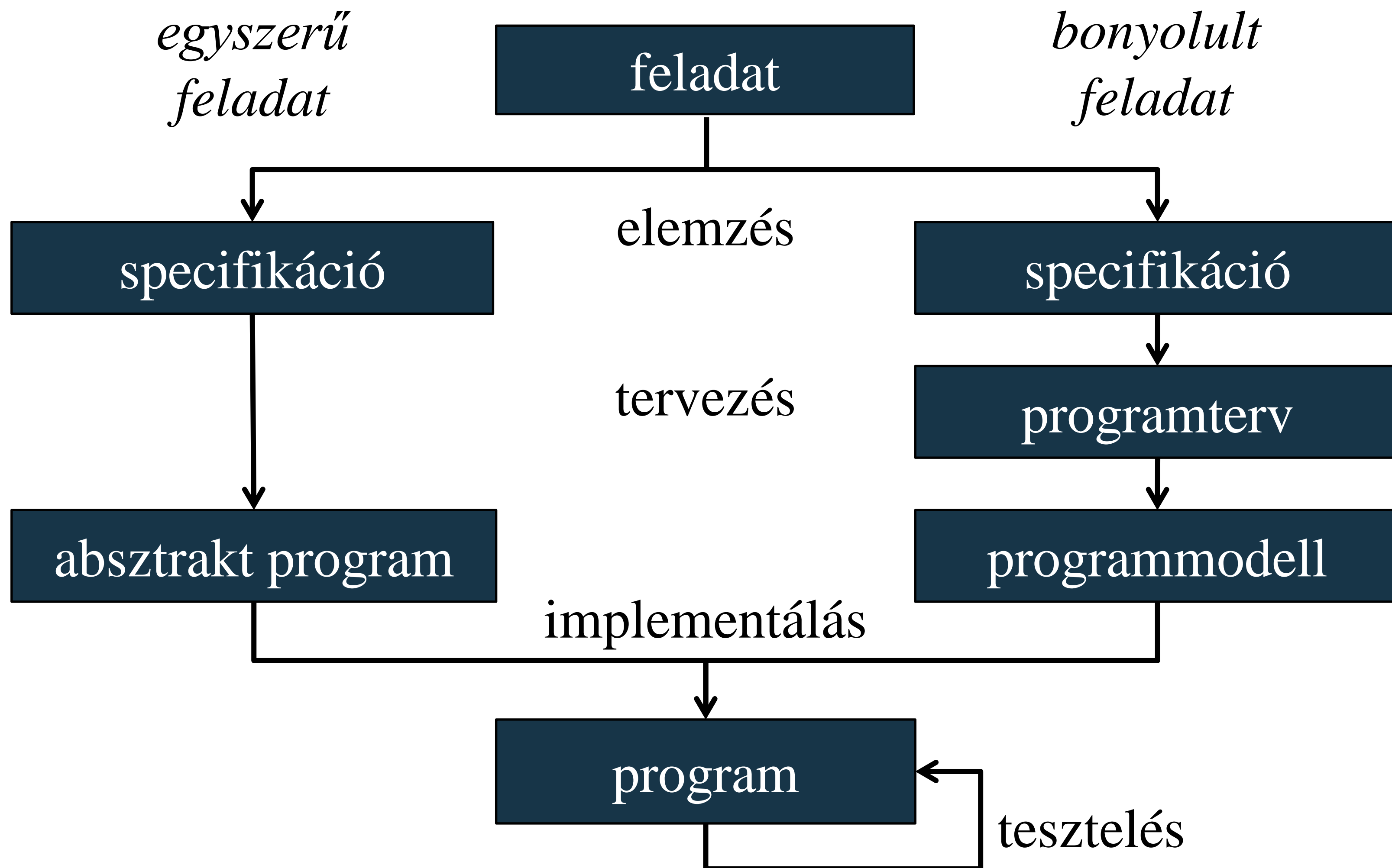


A szoftverfejlesztés folyamata

- A szoftverfejlesztés a kódoláson túl több lépésből áll, amely függ a feladat bonyolultságától is:
  1. A feladatot elemezni kell, és megadni a formális megfelelőjét, vagyis a *specifikáció*t
  2. A specifikációt alapján megtervezhető a program, amely egyszerű feladatnál az *absztrakt program*, míg bonyolult feladatnál a *programterv* elkészítésével jár, amelyből elállítható a *programmodell* (egyszerűsített élprogram)
  3. A tervet implementáljuk a megfelelő programozási nyelven
  4. Az implementált programot, illetve a programkódot *tesztelés*nek vetjük alá, ami módosításokat eredményezhet az implementációban (vagy a korábbi fázisokban)

Szoftverfejlesztés

A szoftverfejlesztés folyamata



A feladat elemzése

- Önmagában a feladat elemzése is nagyban meghatározza a programfejlesztés folyamatát, a lehetséges elemzési lehetőségek:
  - *felülről lefelé* (*top-down*): a főfeladatot részfeladatokra, majd azokat további részfeladatokra bontjuk, míg azonosítani nem tudjuk a részfeladatokat az utasításokkal és adatokkal
  - *alulról felfelé* (*bottom-up*): a feladatban szereplő entitásokat, egységeket határozzuk meg, majd azokat kombináljuk össze olyan módon, hogy megadják a főfeladat megoldását
  - általában az alulról felfelé módszer az alsóbb szinteken felülről lefelé stratégiát igényel

- A tervezés során egy nyelv-független vázát kell elkészítenünk a szoftvernek, amely megadja annak közeli működését
  - a tervezés valamilyen modell alapján történik, amely lehet formális, matematikai, vagy kevésbé formális
    - kevésbé formális megközelítés megkönnyíti a tervezést, és növeli az átláthatóságot, de kevésbé garantálja a program helyességét
    - bizonyos tervezési modellek támogatják a kódgenerálást, és verifikálást, könnyítve ezzel az implementáción
  - fontos szempont a program be- és kimenetének meghatározása, amit *specifikáció*nak nevezünk
  - külön kell megtervezi a program szerkezetének felépítését, illetve futásának menetét, illetve felhasználási eseteit

Szoftverfejlesztés

A szoftverfejlesztés optimalizálása

- A szoftverfejlesztés során a legfőbb cél, hogy a kész programrendszer megfeleljen a *funkcionális* és *minőségi követelményekn*ek
  - ennek érdekében folyamatosan kell vezetni a fejlesztői dokumentációt
- Emellett, a fejlesztők számára fontos, hogy a kész szoftver fejlesztése a lehető legoptimálisabb legyen
  - a szoftverfejlesztési modell meghatározza a fejlesztés módját és menetét
  - azt nem, hogy milyen legyen a fejlesztés stílusa, milyen absztrakciós szinten történjen a megvalósítás, hiszen ezek a tényezők nem határozzák meg a szoftvert önmagát, csak a fejlesztés folyamatát

Programozási paradigmák

A paradigma jelentősége

- A szoftverek tervezésének és programozásának módszerét nevezzük *programozási paradigmá*nak
  - meghatározza a programozási stílust, az absztrakciós szintet
  - vannak általános célú, és szűk körben alkalmazott paradigmák, minden feladatra megtalálható a legjobban alkalmazható paradigma, vagy paradigmák köre
- A fejlesztés során az elemzési szakasz végén célszerű meghatározni az alkalmazni kívánt paradigmákat
  - a paradigma meghatározza az alkalmazható programozási nyelvek körét is, és fordítva
  - sok programozási nyelv több paradigmát is támogatnak, ezek a *multiparadigma* nyelvek

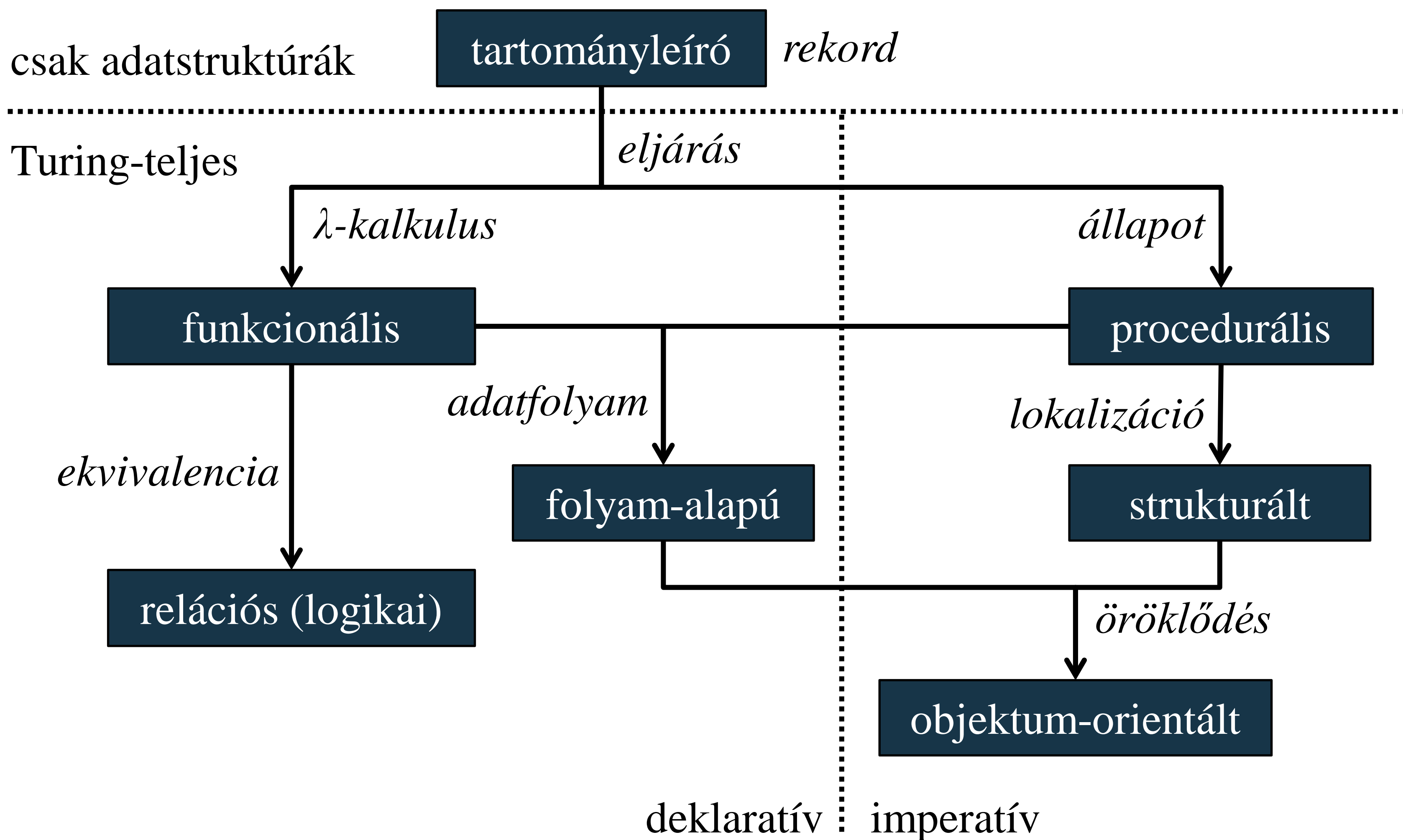
Programozási paradigmák

A paradigmák csoportosítása

- A programozási paradigmákat két csoportba soroljuk:
  - *imperatív*: a program állapotváltozások sorozata, ahol egy állapotot a programban lévő összes változó együttes értéke adja fel, az utasításokat szekvenciálisan hajtja végre
  - *deklaratív*: a program a tartalmát, megjelenését írja le, nem pedig a funkció módját, nem alkalmaz változókat, a program működéséhez csak konstans értékeket használ fel, az utasításokat nem szekvenciálisan hajtja végre
- Magához a gépi kódhoz az imperatív programozás áll közel, a deklaratív nagyobb absztrakciót igényel (mivel nem a hardvernek megfelelő működési elveken alapszik), ugyanakkor nagyobb kifejezőerővel is rendelkezik

Programozási paradigmák

A jelentősebb paradigmák



- *Tartományleíró* (*DSL*, *Domain-Specific Language*)
  - a legdeklaratívabb nyelvek, amely egy adott feladatkör eredményleírását végzi el
  - nyelvek: *XML*, *XSLT*, *HTML*, *CSS*, *SQL*
  - általában nem alkalmas teljes értékű programok írására, mert nem Turing-teljes, rendszerint kiegészítésként szolgál más paradigma használata mellett
  - megszokott alkalmazási területe konfigurációs fájlok, interfész felületek programozására
  - egyes funkcionális nyelvek is támogatják tartományleíró részek írását (pl. *LISP*, *Erlang*, *Prolog*), illetve imperatív nyelvekben is előfordulhatnak (pl. .NET-ben a *LINQ*)

- pl. (*XAML*, szövegcímke a felhasználói felület):

```
<UserControl x:Class="Silverlight.X.Page"
 Width="400" Height="300">
  <Grid x:Name="LayoutRoot" Background="White">
      <!-- rács definiálása -->
      <TextBlock x:Name="text_box"/>
  </Grid> <!-- rács vége -->
</UserControl>
```

- pl. (*SQL*, adott oktatónál lévő hallgatók lekérdezése):

```
select student_name, student_id, start_year
from Seminars -- forrás
where teacher='Roberto' -- feltétel
order by name; -- rendezés
```

- *Funkcionális* (*Functional*):
  - a programfutás matematikai függvények kiértékelésének sorozata, alapja a lambda-kalkulus
  - nincsenek benne állapotok és változó adatok, gyakori a rekurzió, könnyű a párhuzamosítás
  - a nagyobb absztrakciós szint miatt magasabb az erőforrásigény, de rövidebb a programkód
  - nyelvek: *LISP*, *Erlang*, *Clean*, *Haskell*
  - egyes imperatív nyelvekben lehet szimulálni függvénymutatók és lambda-függvények segítségével (pl. *C++*, *C#*, *Mathematica*, *Maple*, *Matlab*)

Programozási paradigmák

Az elsődleges paradigmák

- pl. (*LISP*, lista hossza):

```
(defun length(x) ; függvény definiálás
   (cond ( (null x) 0 ) ; elágazás
         ( (atom x) 1 )
         ( (listp x) (+ 1 (length (cdr x) )))
   )                                ; rekurzív hívás
)
```

- pl. (*Clean*, beszúró rendezés):

```
InsertSort :: a [a] -> [a] | Ord a //deklaráció
InsertSort e [] = [e] // szabályok
InsertSort e [x:xs]
  | e<=x       = [e,x:xs] // elágazás
  | otherwise = [x: InsertSort e xs]
```

- *Relációs, logikai:*
  - a program logikai kiértékelések sorozata, általában implikációkon keresztül, a programozó feladat, hogy a program „igaz legyen”, és megfelelően hatékony
  - nyelvek: *Prolog*, *Oz*, *Castor for C++*
  - pl. (*Prolog*, rokonsági kapcsolatok definiálása):

```
mother_child(trude, sally). % axiómák
father_child(tom, sally).
father_child(mike, tom).
sibling(X, Y) :- parent_child(Z, X), % szabályok
                 parent_child(Z, Y).
parent_child(X, Y) :- father_child(X, Y).
parent_child(X, Y) :- mother_child(X, Y).
```

- *Folyam-alapú* (*FBP*, *Flow-Based Programming*):
    - a program fekete doboz jellegű alkalmazások hálózata, amelyek üzenetküldéssel kommunikálnak egymással, az alkalmazások működése kívülről nem ismert
    - az egyes folyamatok párhuzamosan, aszinkron módon futnak, és információs csomagokat fogadnak, illetve küldenek, az információ áramlása folyamszerű
    - leginkább csak elv maradt, nincs olyan konkrét nyelv, amely megvalósítani, bizonyos rendszerekben előfordulhatnak folyam-alapú részek (pl. Linda), továbbá fontos alapja az objektum-orientált szemléletnek

- *Procedurális* (*Procedural*):
  - a programot *alprogram*okra (*subroutine*) bontja, és minden alprogram meghatározott részfeladatot végez el, ezek lehetnek:
    - *eljárás* (*procedure*): valamilyen utasítássorozatot futtat, végeredmény nélkül
    - *függvény* (*function*): valamilyen matematikai számítást végez el, és megadja annak eredményét
  - az alprogramok a vezérlési szerkezetek segítségével épülnek fel, meghívhatnak más alprogramokat, és kommunikálhatnak velük (azaz adatokat adhatnak át, paraméterek és visszatérési értékek segítségével)

- a vezérlést a főprogram szolgáltatja, az kezeli a teljes programban jelen lévő adatokat
- nyelvek: *Fortran*, *C*, *BASIC*, *Pascal*
- pl. (*C*++, vektor összegzése függvénnyel):

```
int Sum(vector<int> values){
   // a függvény paraméterben megkapja a vektort
   int sum = 0;
   for (int i = 0; i < value.size(); i++)
      sum += values[i];
   // ciklussal minden elemet hozzáveszünk az
   // összeghez
   return sum; // visszatérési érték az összeg
}
```

- pl. (*Maple*, szám faktoriálisa):

```
myfac := proc(n::nonnegint)
    local out, i;
    out := 1;
    for i from 2 to n do
        out := out * i
    end do;
    out
end proc;
```

ugyanez lambda-kifejezés segítségével (funkcionálisan):

```
myfac := n -> product( i, i = 1..n );
```

Programozási paradigmák

Az elsődleges paradigmák

- pl. (*Fortran*, tömb átlaga):

```
implicit none
integer :: number_of_points real,
dimension(:), allocatable :: points
real :: average_points=0.

read (*,*) number_of_points
allocate (points(number_of_points))
read (*,*) points
if (number_of_points > 0)
   average_points = sum(points)/number_of_points
deallocate (points)
write (*,'(''Average = '', 1g12.4)')
   average_points
```

Programozási paradigmák

Az elsődleges paradigmák

- *Strukturált* (*Structured*):
  - a program részegységekre (csomagokra, vagy blokkokra) tagolódik, minden egység rendelkezik egy belépési ponttal, és egy kilépési ponttal
  - a programban használt adatstruktúrák a programegységeknek megfelelően strukturálódnak
  - támogatja a kivételkezelést (a program felmerülő problémák lekezelése a programfutás károsodása nélkül), tiltja a programkódban történő ugrálást (goto)
  - nyelvek: *Pascal*, *ADA*

- pl. (*Ada*, verem csomag):

```
package STACK_T is
 type Stack is limited private;
 procedure Push (v: in out Stack; e: Value);
 procedure Pop (v: in out Stack);
 function Top (v: Stack) return Value;
private
 type Node; type Pointer is access Node;
 type Node is record
    data: Value; next: Pointer := null;
 end record;
 type Stack is record top: Pointer := null;
 end record;
end STACK_T;
```

Programozási paradigmák

Az elsődleges paradigmák

- *Objektum-orientált* (*Object-oriented*):
  - a feladat megoldásában az alulról-felfelé megközelítést alkalmazza, két alapgondolata az egységbe zárás és az öröklődés
  - a programegységet egymással kommunikáló objektumok összessége adja, amelyek valamilyen relációban állnak egymással
  - manapság a legnépszerűbb programozási paradigma, a programozási nyelvek jelentős része támogatja
  - objektumorientált támogatással rendelkező nyelvek: *C++*, *Object-C*, *Common LISP*, *Matlab*, *PHP*, *Python*, *Perl*, …
  - objektumorientált nyelvek: *Smalltalk*, *JAVA*, *C#*, *Ruby*, …

- pl. (*C*++, verem osztály):

```
class Stack {
private: // rejtett rész, implementáció
    int* values; // attribútumok
    int top;
public: // látható rész, interfész
    Stack(){ values = new int[10]; top = 0; }
    // konstruktor
    ~Stack() { delete[] values; } // destruktor
    void Push(int v) { // metódus
        if (top < 10) { values[top] = v; top++; }
    }
    // további metódusok ...
};
```

- pl. (*Java*, mobil felhasználói felület):

```
public class HelloWorld extends MIDlet
       // a MIDlet osztály leszármazottja
       implements CommandListener {
   private Form form; // implementáció
   private Command exitCommand;

   public HelloWorld() { // konstruktor
      form = new Form("Welcome!", new Item[]
         { new StringItem("","Hello World!") });
      form.addCommand(exitCommand);
      form.setCommandListener(this);
   }
}
```

Programozási paradigmák

A másodlagos paradigmák

- Vannak úgynevezett másodlagos programozási paradigmák, amelyek a programozást stílusát nagyban befolyásolják, pl.:
  - *Eseményvezérelt* (*Event-driven programming*): a programfutás mentetét az interfészeken keresztül érkező események vezérlik, az eseményeket több tényező válthatja ki, majd azokat megfelelő eseménykezelők feldolgozzák
  - *Sablonprogramozás* (*Generic programming*): az algoritmusokat sablonos formában állítja elő, és az alkalmazáskor helyettesíti be a konkrét tényezőket
  - *Metaprogramozás* (*Metaprogramming*): a folyamatot a program metainformációs (programmanipulációs) szinten vezérli, ezáltal a programkód szerkeszthetővé válik futás alatt