ABSZTRAKCIÓ FÜGGVÉNYEKKEL (ELJÁRÁSOKKAL)

Elágazó rekurzió

- Korábban lineáris-rekurzív, ill. lineáris-iteratív folyamatokra láttunk példákat (faktoriális kiszámítása kétféleképpen).
- Most *elágazó rekurzióra* nézzünk példát: állítsuk elő a Fibonacci-számok sorozatát.
- Egy Fibonacci-számot az előző két Fibonacci-szám összege adja:

```
0, 1, 1, 2, 3, 5, 8, 13, 21, ...
```

- A Fibonacci-számok matematikai definíciója könnyen átírható SML-függvénnyé:

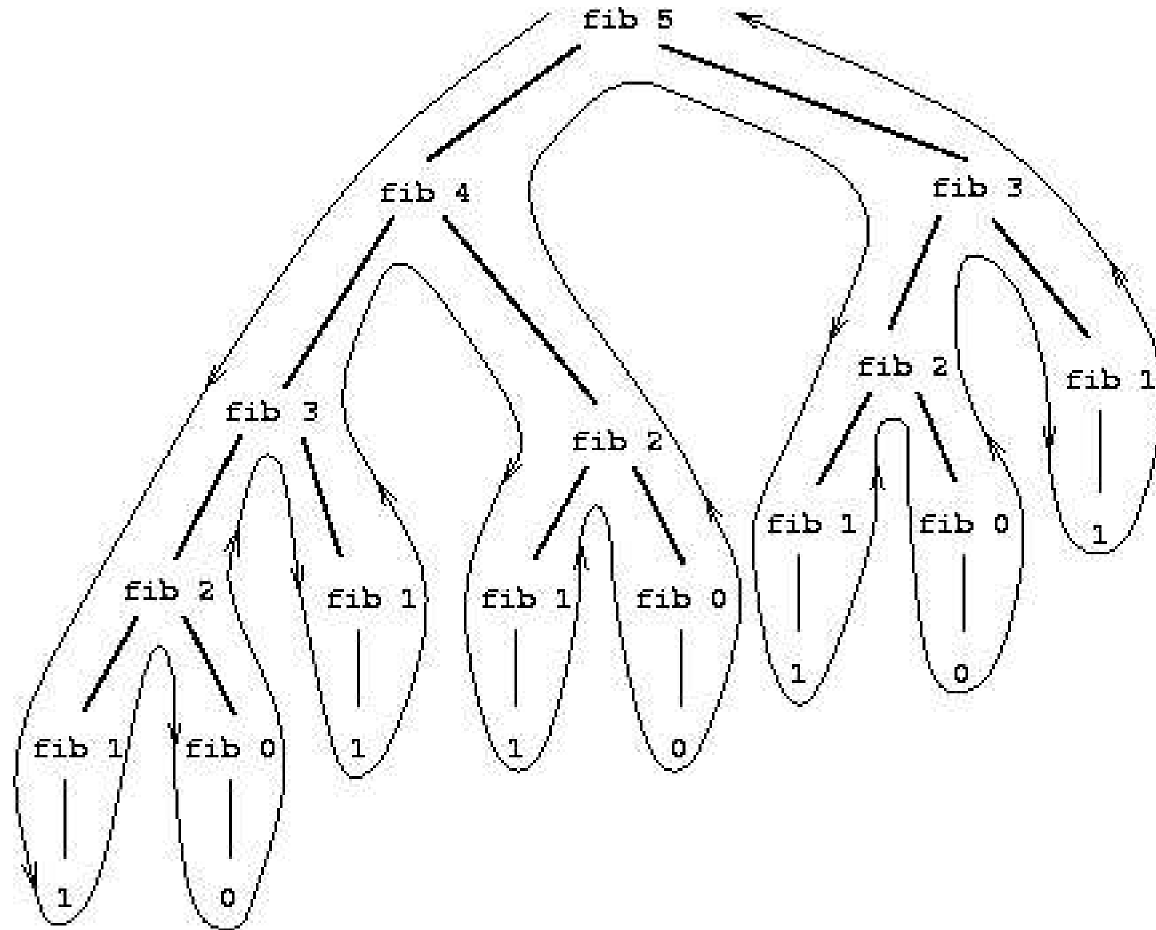
$F(0) = 0$
$F(1) = 1$
$F(n) = F(n-2) + F(n-1)$, ha $n > 1$

```
fun fib 0 = 0
  | fib 1 = 1
  | fib n = fib(n-1) + fib(n-2)
```

Emlékeztetőül: a `fib` függvény definíciójában a 3. klóznak az utolsónak kell lennie, mert az `n` minta minden argumentumra illeszkedik.

- A következő lapon látható ábra illusztrálja az elágazóan rekurzív folyamatot `fib 5` kiszámítása esetén.

Elágazó rekurzió (folyt.)



- `fib 5`-öt `fib 4` és `fib 3`, `fib 4`-et `fib 3` és `fib 2` kiszámításával stb. kapjuk.

Elágazó rekurzió (folyt.)

- A példa alkalmas az elágazó rekurzió lényegének bemutatására, de szinte alkalmatlan a Fibonacci-számok előállítására!
- Vegyük észre, hogy pl. `fib 3` kiszámítását, ami a munkának legalább a harmada, kétszer is elvégezzük.
- Könnyen belátható, hogy $F(n)$ meghatározásához pontosan $F(n+1)$ levélből álló fát kell bejárni, azaz ennyiszer kell meghatározni $F(0)$-t vagy $F(1)$-et.
- $F(n)$ exponenciálisan nő n-nel.

  Pontosabban, $F(n)$ a $\Phi^n/\sqrt{5}$-höz közel eső egész, ahol $\Phi = (1 - \sqrt{5})/2 \approx 1.61803$, az ún. *aranymetszés* arányszáma. Φ kielégíti a $\Phi^2 = \Phi + 1$ egyenletet.
- A megteendő lépések száma tehát $F(n)$-nel együtt exponenciálisan nő n-nel. Ugyanakkor a tárigény csak lineárisan nő n-nel, mert csak azt kell nyilvántartani, hogy hányadik szinten járunk a fában.
- Általában is igaz, hogy elágazó rekurzió esetén a lépések száma a fa csomópontjainak a számával, a tárigény viszont a fa maximális mélységével arányos.

Elágazó rekurzió (folyt.)

- A Fibonacci-számok lineáris-iteratív folyamattal is előállíthatók.

  Ha az a és b változók kezdőértéke rendre $F(1) = 1$ és $F(0) = 0$, és ismétlődően alkalmazzuk az $a \leftarrow a + b$, $b \leftarrow a$ transzformációkat, akkor n lépés után $a = F(n + 1)$ és $b = F(n)$ lesz. Az iteratív folyamatot létrehozó SML-függvény:

```
fun fib n = let fun fibIter (i, b, a) =
                        if i = n then b
                        else fibIter(i+1, a, a+b)
            in
                fibIter(0, 0, 1)
            end
```

- *Mintaillesztést* használhatunk, ha `i`-t nem növeljük, hanem `n`-től 0-ig csökkentjük.

  Figyelem: a klózok sorrendje, mivel nem egymást kizáróak a minták, lényeges!

```
fun fib n = let fun fibIter (0, b, a) = b
                      | fibIter (i, b, a) = fibIter(i-1, a, a+b)
            in
                fibIter(n, 0, 1)
            end
```

Elágazó rekurzió (folyt.)

- A Fibonacci-példában a lépések száma elágazó rekurziónál tehát n-nel exponenciálisan, lineáris rekurziónál n-nel arányosan nőtt, kis n-ekre is hatalmas a nyereség!
- Téves lenne azonban azt a következtetést levonni, hogy az elágazó rekurzió használhatatlan. Amikor hierarchikusan strukturált adatokon kell műveleteket végezni – pl. egy fát kell bejárni –, akkor az elágazó rekurzió (angolul: *tree recursion)* nagyon is természetes és hasznos eszköz.
- Az elágazó rekurzió numerikus számításoknál az algoritmus első megfogalmazásakor is hasznos lehet: gondoljunk csak arra, hogy milyen könnyű volt átírni a Fibonacci-számok matematikai definícióját programmá.
- Ha már értjük a feladatot, az első, rossz hatékonyságú változatot könnyebb átírni jó, hatékony programmá. Az elágazó rekurzió segíthet a feladat megértésében.

Az iteratív Fibonacci-algoritmushoz csak egy aprócska ötlet kellett. A következő feladatra azonban nem lenne könnyű iteratív algoritmust írni.

- Hányféleképpen lehet felváltani egy dollárt 50, 25, 10, 5 és 1 centesekre?
- Általánosabban: adott összeget adott érmékkel hányféleképpen lehet felváltani?

Elágazó rekurzió (folyt.): pénzváltás

Tegyük föl, hogy n darab érme áll a rendelkezésünkre valamilyen (pl. nagyság szerint csökkenő) sorrendben. Ekkor az a összeg lehetséges felváltásainak számát úgy kapjuk meg, hogy

- kiszámoljuk, hogy az a hányféleképpen váltható fel az első (d értékű) érmét kivéve a többi érmével, és ehhez
- hozzáadjuk, hogy az $a - d$ hányféleképpen váltható fel az összes érmével, az elsőt is beleértve – más szóval azt, hogy az a-t hányféleképpen tudjuk úgy felváltani, hogy a d-t legalább egyszer felhasználjuk.

A feladat tehát rekurzióval megoldható, hiszen redukálható úgy, hogy kisebb összegeket kevesebb érmével kell felváltanunk. A következő alapeseteket különböztessük meg:

- Ha $a = 0$, a felváltások száma legyen 1.
- Ha $a < 0$, a felváltások száma legyen 0.
- Ha $n = 0$, a felváltások száma legyen 0.

Gyakorló feladat. A példában a `firstDenomination`, magyarul *első címlet* függvényt felsorolással valósítottuk meg. Tömörebb és rugalmasabb lesz a megvalósítása, ha listát alkalmazunk. Írja át a függvényt `countChange` függvényt úgy, hogy a címleteket egy lista tartalmazza.

Elágazó rekurzió (folyt.): pénzváltás

```
fun countChange amount =
  let (* cC amount kindsOfCoins = az amount összes felváltásainak
                                  száma kindsOfCoins db érmével  *)
      fun cC (amount, kindsOfCoins) =
            if amount < 0 orelse kindsOfCoins = 0 then 0
            else if amount = 0 then 1
            else cC (amount, kindsOfCoins - 1) +
                 cC (amount - firstDenomination kindsOfCoins,
                                                  kindsOfCoins)
      and firstDenomination 1 = 1
        | firstDenomination 2 = 5
        | firstDenomination 3 = 10
        | firstDenomination 4 = 25
        | firstDenomination 5 = 50

  in  cC(amount, 5)  end;

countChange 10 = 4; countChange 100 = 292;
```

Hatványozás

- Az eddig látott folyamatokban a kiértékelési (végrehajtási) lépések száma az adatok n számával lineárisan, ill. exponenciálisan nőtt. Most olyan példa következik, amelyben a lépések száma az n logaritmusával arányos.
- A b szám n-edik hatványának definícióját ugyancsak könnyű átrakni SML-be.

$$b^0 = 1$$
$$b^n = b \cdot b^{n-1}$$

```
fun expt (b, 0) = 1
  | expt (b, n) = b * expt(b, n-1)
```

- A létrejövő folyamat lineáris-rekurzív. $O(n)$ lépés és $O(n)$ méretű tár kell a végrehajtásához.
- A faktoriálisszámításhoz hasonlóan könnyű felírni lineáris-iteratív változatát.

```
fun expt (b, n) =
      let fun exptIter (0, product) = product
            | exptIter (counter, product) =
                            exptIter (counter-1, b * product)
      in
         exptIter(n, 1)
      end
```

- $O(n)$ lépés és $O(1)$ – azaz konstans – méretű tár kell a végrehajtásához.

Hatványozás (folyt.)

- Kevesebb lépés is elég, ha kihasználjuk az alábbi egyenlőségeket:

$b^0 = 1$

$b^n = (b^{n/2})^2$, ha n páros

$b^n = b \cdot b^{n-1}$, ha n páratlan

```
fun expt (b, n) =
  let fun exptFast 0 = 1
        | exptFast n =
            if even n
            then square(exptFast(n div 2))
            else b * exptFast(n-1)
      and even i = i mod 2 = 0
      and square x = x * x
  in  exptFast n  end
```

- A lépések száma és a tár mérete $O(\lg n)$-nel arányos. Konstans tárigényű iteratív változata:

```
fun expt (b, 0) = 1   (* Nem hagyható el! Miért nem?  *)
  | expt (b, n) = let fun exptFast (1, r) = r
                        | exptFast (n, r) =
                          if even n then exptFast(n div 2, r*r)
                          else           exptFast(n-1, r*b)
                      and even i = i mod 2 = 0
                  in  exptFast(n, b)  end
```

Legnagyobb közös osztó

- Következő példánk a és b legnagyobb közös osztóját számolja ki az euklideszi algoritmussal.
- Az alapgondolat az, hogy ha a-t b-vel osztva r a maradék, akkor a és b közös osztói azonosak b és r közös osztóival.
- A matematikai definíciót most is pontosan követi az SML-függvény.

$$\gcd(a, 0) = a$$
$$\gcd(a, b) = \gcd(b, a \bmod b)$$

```
fun gcd (a, 0) = a
  | gcd (a, b) = gcd(b, a mod b)
```

- A *folyamat* iteratív. A lépések száma logaritmikusan nő.

  Pontosabban – Lamé tétele szerint – ha az euklideszi algoritmus egy számpár legnagyobb közös osztóját k lépésben számítja ki, akkor számpár kisebbik tagja nem lehet kisebb a k-adik Fibonacci-számnál.

  Legyen n az algoritmus kisebbik paramétere. Ha a legnagyobb közös osztó kiszámításához k lépésre van szükség, akkor $n \geq F(k) \approx \Phi^k/\sqrt{5}$. Azaz a k lépésszám az n (Φ alapú) logaritmusával arányos. (Ld. SICP, 1.2.5. szakasz.)

Prímteszt

- A következő SML-predikátum egy n szám prím voltát teszteli.
- A `findDivisor` függvény 2-től kezdve megkeresi az n szám legkisebb osztóját. Az n szám prím, ha a legkisebb osztó az n szám maga.
- Az n osztóit 2-től $\sqrt{n}$-ig kell keresni, így a lépések száma $O(\sqrt{n})$.

```
fun prime n =
  let fun smallestDivisor n = findDivisor(n, 2)
      and findDivisor (n, testDivisor) =
            if      square testDivisor > n then n
            else if divides(testDivisor, n) then testDivisor
            else    findDivisor(n, testDivisor+1)
      and square x = x * x
      and divides (a, b) = b mod a = 0
in
    n = smallestDivisor n
end
```

Prímteszt (folyt.)

- A következő SML-predikátum egy szám prím voltát *valószínűségi módszerrel* teszteli. A lépések száma $O(\lg n)$.
- Az algoritmus a kis Fermat-tételen alapul, ami azt mondja ki, hogy:

  ha n prím és $0 < a < n$, akkor a^n modulo n szerint *kongruens* a-val, azaz $a^n \bmod n = a$.

  - Két szám akkor *kongruens* modulo n szerint, ha n-nel osztva mindkettőnek ugyanaz a maradéka. Egy a szám n-nel való osztásának maradékát a modulo n szerinti maradékának, vagy röviden a modulo n-nek is nevezik.
- Ha n nem prím, akkor az $a < n$ számok nagy hányadára nem teljesül a fenti reláció.
- A prímteszt algoritmusa ezek után a következő :
  - Adott n-re véletlenszerűen válasszunk egy $a < n$ számot: ha $a^n \bmod n \neq a$, akkor n nem prím. Ellenkező esetben nagy a valószínűsége, hogy n prím.
  - Válasszunk véletlenszerűen egy másik $a < n$ számot: ha $a^n \bmod n = a$, akkor növekedett annak a valószínűsége, hogy az n prím. Újabb és újabb a értékeket választva egyre biztosabbak lehetünk abban, hogy az n prím.

Prímteszt (folyt.)

- Az `expmod` segédfüggvény a `base` syám `exp`-edik hatványának modulo `m` szerinti maradékát adja eredményül.

```
(* expmod (base, exp, m) = base exp-edik hatványa modulo m
*)
fun expmod (_, 0, _) = 1
  | expmod (b, e, m) =
      if even e then square(expmod(b, e div 2, m)) mod m
      else b * expmod(b, e-1, m) mod m
and even n = n mod 2 = 0
and square x = x * x
```

- Nagyon hasonló felépítésű `exptFast`-hoz. A lépések száma a kitevő logaritmusával arányos.
- Szükségünk van véletlenszámok előállítására. Részletek az SML alapkönyvtárából:

```
Random.range (min, max) gen = an integral random number in the
                   range [min, max).  Raises Fail if min > max.
Random.newgen () = a random number generator, taking the seed
                   from the system clock.
```

Prímteszt (folyt.)

- Betöltjük a `Random` könyvtárat:

```
loadOne "Random";
```

- `fermatTest` generál egy álvéletlen-számot, és egyszer elvégzi a vizsgálatot:

```
(* fermatTest n = false if n is not prime
*)
fun fermatTest n =
      let fun tryIt a = expmod(a, n, n) = a
      in  tryIt(Random.range (1, n) (Random.newgen()) )
      end
```

- `fastPrime` `times`-szor megismétli a vizsgálatot:

```
(* fastPrime (n, times) = true if n passes the prime test
                          times times
*)
fun fastPrime (n, 0) = true
  | fastPrime (n, t) = fermatTest n andalso fastPrime(n, t-1)
```

- Ez a megoldás csak nagy valószínűséggel, de nem teljes bizonyossággal ad választ a kérdésre.

LISTÁK

Adott számú elem egy lista elejéről és végéről (take, drop)

- Legyen $xs = [x_0, x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_{n-1}]$, akkor
  `take(xs, i)` $= [x_0, x_1, \ldots, x_{i-1}]$ és `drop(xs, i)` $= [x_i, x_{i+1}, \ldots, x_{n-1}]$.

```
(* take : 'a list * int -> 'a list
   take (xs, i) = xs, ha i < 0; az xs első i db eleméből
                  álló lista, ha i >= 0
*)
fun take (_, 0)      = []
  | take ([], _)     = []
  | take (x::xs, i) = x :: take(xs, i-1);

(* drop : 'a list * int -> 'a list
   drop(xs, i) = xs, ha i < 0; az xs első i db elemének
                 eldobásával előálló lista, ha i >= 0
*)
fun drop ([], _)     = []
  | drop (x::xs, i) = if i > 0 then drop (xs, i-1) else x::xs;
```

- Könyvtári változatuk, `List.take`, ill. `List.drop`, ha az `xs` listára alkalmazzuk, $i < 0$ vagy $i >$ `length xs` esetén `Subscript` néven kivételt jelez.

Lista redukciója kétoperandusú művelettel (`foldr`, `foldl`)

- Vissza-visszatérő feladat egy lista redukciója kétoperandusú művelettel. Közös, hogy n db értékből egyetlen értéket kell előállítani (vö. *redukció*).
- `foldr` jobbról balra, `foldl` balról jobbra haladva egy kétoperandusú műveletet (pontosabban egy *párra alkalmazható, prefix* pozíciójú *függvényt*) alkalmaz egy listára. Példák szorzat és összeg kiszámítására:

```
foldr op* 1.0 [] = 1.0;                            foldl op+ 0 [] = 0;
foldr op* 1.0 [4.0] = 4.0;                         foldl op+ 0 [4] = 4;
foldr op* 1.0 [1.0, 2.0, 3.0, 4.0] = 24.0;         foldl op+ 0 [1, 2, 3, 4] = 10;
```

- Jelöljön $\oplus$ tetszőleges kétoperandusú infix operátort. Akkor

  foldr op$\oplus$ e [x_1, x_2, ..., x_n] = (x_1 $\oplus$ (x_2 $\oplus$ ... $\oplus$ (x_n $\oplus$ e) ...))
  foldr op$\oplus$ e [] = e
  foldl op$\oplus$ e [x_1, x_2, ..., x_n] = (x_n $\oplus$ ... $\oplus$ (x_2 $\oplus$ (x_1 $\oplus$ e)) ...)
  foldl op$\oplus$ e [] = e

- Asszociatív műveleteknél `foldr` és `foldl` eredménye azonos.

Példák `foldr` és `foldl` alkalmazására

- A ⊕ művelet `e` operandusa néhány gyakori műveletben – összeadás, szorzás, konjunkció (logikai „és”), alternáció (logikai „vagy”) – a (jobb oldali) *egységelem* szerepét tölti be.
- `isum` egy egészlista elemeinek összegét, `rprod` egy valóslista elemeinek szorzatát adja eredményül.

```
val isum = foldr op+ 0;
val isum = foldl op+ 0;
```

```
val rprod = foldr op+ 1.0;
val rprod = foldl op+ 1.0;
```

- A `length` függvény is definiálható `foldl`-lel vagy `foldr`-rel. Kétoperandusú műveletként olyan segédfüggvényt (`inc`) alkalmazunk, amelyik *nem használja* az első paraméterét.

```
(* inc : 'a * int -> int
   inc (_, n) = n + 1 *)
fun inc (_, n) = n + 1;
(* lengthl, lengthr : 'a list -> int *)
val lengthl = fn ls => foldl inc 0 ls;
fun lengthr ls = foldr inc 0 ls;

lengthl (explode "tengertanc");
lengthr (explode "hajdu sogor");
```