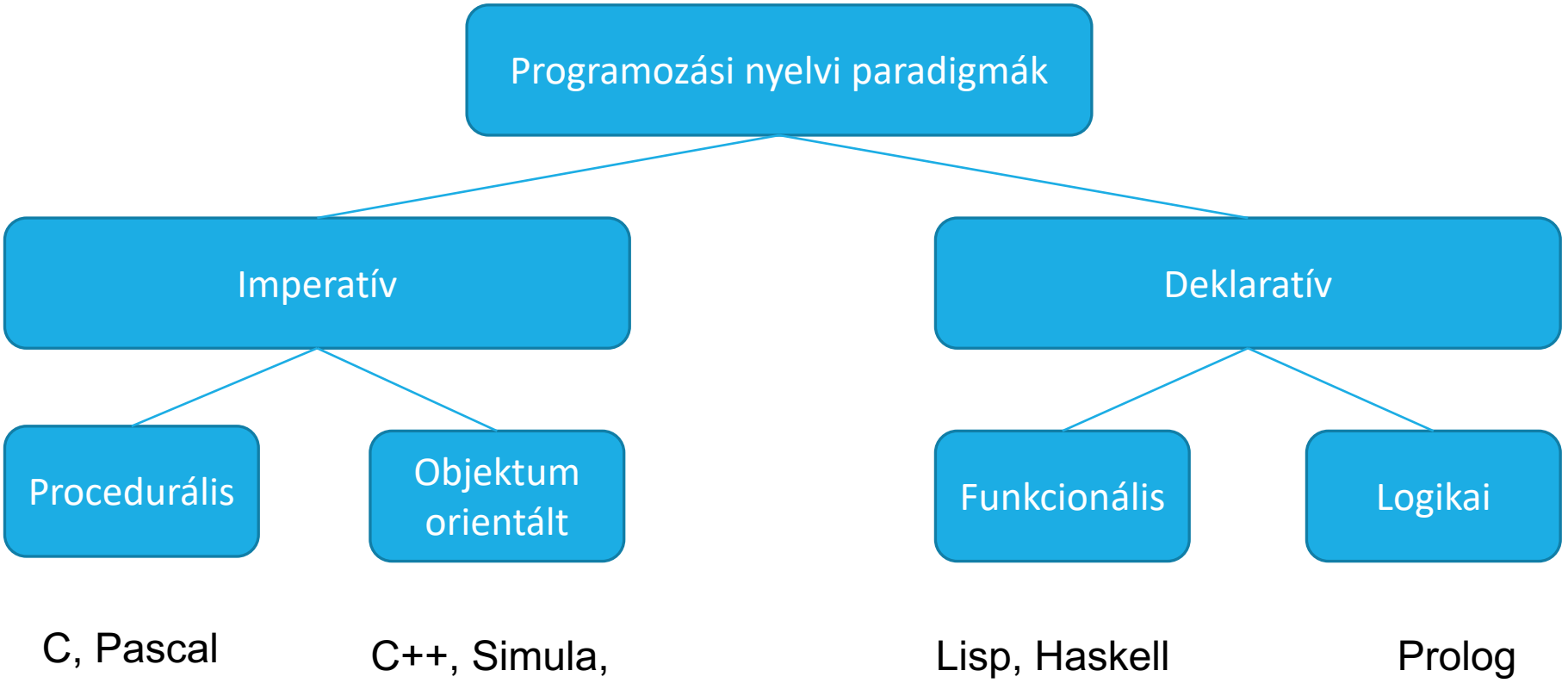

# Programozási nyelvek és módszerek

## 8. ELŐADÁS – FUNKCIONÁLIS ÉS LOGIKAI PROGRAMOZÁS

# Mai óra

- Programozási nyelvi paradigmák
- Lambda kalkulus
- Funkcionális programozás
- Logikai programozás

# Programozási nyelvi paradigmák



# Paradigmák

Az **imperatív** programozás sajátosságai

- felszólító mód
  - parancsok, utasítások
- a lényeg az algoritmus megtalálása
  - hogyan oldjuk meg a feladatot
  - pontosan követhetőek a végrehajtott lépések
- a változó egy adott memória helyen tárolt aktuális érték, mely ismételten újabb és újabb értékeket vesz fel

# Paradigmák

A **deklaratív** programozás sajátosságai

- kijelentő mód
  - állítások, egyenletek
- a lényeg
  - a specifikáció leírása
  - mit kell megoldanunk
- a végrehajtás a nyelv értelmező, fordító programjától függ
  - nem mindig követhetőek a végrehajtott lépések
- a változó a matematikából ismert fogalomnak felel meg, egy konkrét érték lehet, mely a program írásakor még ismeretlen
- Alkalmazási területek
  - Mesterséges intelligencia
  - Telekommunikáció
  - …

# Paradigmák

**Funkcionális programozási** eszközökkel minden olyan feladat megoldható, amelyik megoldható **imperatív** nyelven és viszont.

Általában funkcionális programozási stílusról beszélünk, ha a megoldó programokat mellékhatásoktól mentes összetevőkből, szabályos programkonstrukciókkal építjük fel.

„A funkcionális nyelvek matematikai számítási modellje a lambda-kalkulus.”

Mi az a lambda kalkulus??

- Egy formális rendszer, amit matematikai függvények vizsgálatára vezettek be.
- Church 1936-ban ennek segítségével bizonyította be, hogy nem létezik megoldás a Russel-paradoxonra.

# Russel paradoxon

A múlt század elején tudatosodott, hogy a matematika addig használt halmazelméleti és logikai alapozása a Russel paradoxont is tartalmazza, belső ellentmondásai vannak.

Naív halmazelméleti formában:

- Legyen
  - $R = \{X \mid X \text{ nem eleme } X\text{-nek}\}$, akkor $R \in R \Leftrightarrow (R \notin R)$

# Példák

## Borbély paradoxon

- Tegyük fel, hogy a laktanya katonai borbélya a szolgálati szabályzatnak megfelelően csak azokat a katonákat borotválja, akik maguk nem borotválkoznak, de nem borotválhatja azokat, akik maguk borotválkoznak.
- Kérdés: magát megborotválhatja-e?
  - Ha megborotválja magát, akkor olyan katonának számít, aki maga borotválja magát, ergo a szolgálati szabályzat megtiltja, hogy megborotválkozzon.
  - Ha ennek megfelelően, nem borotválkozik, akkor a szolgálati szabályzat értelmében, olyan katonának számít, akit borotválnia kell.

# Példák

Katalógus paradoxon

Karinthy-paradoxon

- „Ohó álljunk csak meg. Ön azt mondja, a rögeszmém, hogy őrült vagyok. De hiszen tényleg az vagyok, az imént mondta. De hiszen akkor ez nem rögeszme, akkor az egy logikus gondolat. Tehát nincs rögeszmém. Tehát mégse vagyok őrült. Tehát csak rögeszme, hogy őrült vagyok, tehát rögeszmém van, tehát őrült vagyok, tehát igazam van, tehát nem vagyok őrült. Mégiscsak gyönyörű dolog a tudomány!”

A lambda kalkulusban egyfajta érték van – ez a függvény (ezt hívják  lambda absztrakciónak is)

- és egyfajta művelet – a függvény alkalmazás, ez a változó behelyettesítés

Érdekesség

- bebizonyították, hogy minden algoritmus, ami Turing gépen megvalósítható, megvalósítható tisztán lambda kalkulusban is.

Vannak függvényeink, pl. :

- $x+5$
- $x^2+b*x+c$
- $4$

Minek a függvénye a kifejezés értéke?

Jelölés:

- $x \rightarrow x+5$ $\lambda x.(x+5)$ - itt az x változó „kötött”
- $x \rightarrow x^2+b*x+c$, $\lambda x.(x^2+b*x+c)$
- $4$ $\lambda x.4$

# Érdekesség:

Honnan jött a jelölés?

Jelölni akarták, ha egy változó „kötött” .

- Russell és Whitehead  kezdetben a „kalap” jelölést használta: t[â].
- Church módosította a jelölést: â.t[a], de ezt ∧a.t[a] formában nyomtatták ki
- Végül - azt mondják, nyomdai tévedésből - kialakult a végleges forma: λa.t[a].

## Művelet:  alkalmazás = változó behelyettesítés

- λx.(2*x+3) 7 azt jelenti, hogy x minden előfordulásába behelyettesítjük a 7-t
  - 2*7+3
  - 17

- λx.(2*x+3) (y-5) azt jelenti, hogy x minden előfordulásába behelyettesítjük az (y-5)-t
  - 2*(y-5)+3

Többváltozós függvények – egyváltozós függvények egymás utáni alkalmazásával

- $f(x,y) = x+y$
- $\lambda\, x.(\lambda y.(x+y))$

Ha alkalmazni akarjuk pl. $x=3$, $y=5$-re:

- $\lambda\, x.(\lambda y.(x+y))\ 3\ 5$
- $\lambda y.(3+y)$
- $3+5$
- $8$

# Magasabbrendű függvények

A függvények alkalmazhatóak más függvényekre is

- $\lambda f.(\lambda z.(f\ (f\ z)))$ – ez az f-t kétszer alkalmazza egymás után

Ha ezt alkalmazzuk a $\lambda x.(3*x + 5)$ kifejezésre, akkor a következő kifejezést kapjuk:

- $\lambda f.(\lambda z.(f\ (f\ z)))\ \lambda x.(3*x + 5)$

Behelyettesítés után

- $\lambda z.(\lambda x.(3*x + 5)\ (\lambda x.(3*x + 5)\ z))$

Ha ezt alkalmazzuk pl. 2-re (z helyébe 2-t írunk)

$\lambda z.(\lambda x.(3*x + 5)\ (\lambda x.(3*x + 5)\ z))\ 2 \Rightarrow$
$\lambda x.(3*x + 5)\ (\lambda x.(3*x + 5)\ 2) \Rightarrow$
$\lambda x.(3*x + 5)\ (3*2 + 5) \Rightarrow$
$\lambda x.(3*x + 5)\ (6 + 5) \Rightarrow$
$\lambda x.(3*x + 5)\ 11 \Rightarrow$
$3*11 + 5 \Rightarrow$
$33 + 5 \Rightarrow$
$38$

(Ez valóban az f(f(2))

Ez a redukálás normál formához vezetett, ami már tovább nem redukálható.

# Szabad és kötött változók

Informálisan, egy változó akkor kötött egy λ-kifejezésben, ha tőle balra található egy λ amelyik „kiterjed rá.”

- Például a λx.x kifejezésben nincs szabad változó, míg a λx.y kifejezésben az y szabad változó.

Csak azt a λ-absztrakciót tekintjük függvénynek, amely nem tartalmaz szabad változókat.

Azt a λ-kifejezést amelyik nem tartalmaz szabad változókat zártnak nevezzük.

- (Formális definíciók nélkül)

# Béta konverzió

A β-konverzió a függvény-alkalmazás elnevezése a λ-kalkulusban.

Amikor egy λ-absztrakciót alkalmazunk egy λ-kifejezésre, akkor a λ-kifejezést behelyettesítjük a λ-val kötött változó szabad előfordulásainak helyeire.

Speciálisan, ha mindig a „nyíl irányában” haladunk, azaz végrehajtjuk a kijelölt függvény-alkalmazásokat, így redukálva a λ-kifejezést, akkor azt mondjuk, hogy β-redukciót hajtunk végre.

Legfontosabb szerepe a λ-kifejezések szisztematikus kiértékelésében van.

# Béta konverzió

A β-redukció esetén egy λ-kifejezésben egyszerre több kifejezés is várhat kiértékelésre. Ezeket redukálható részkifejezéseknek, röviden „redex”-nek (Reducible Expression) nevezzük.

A β-redukció definíciójában nincs megkötés arra, hogy ezeket milyen sorrendben kell elvégezni, de a választott sorrendnek jelentősége van. A választott kiértékelési sorrendet kiértékelési stratégiának nevezzük.

Tekintsük az alábbi λ-kifejezést például - két redexet is tartalmaz:

- $(\lambda x.(x + 5))\ ((\lambda y.y)\ 2)$

Hogy értékeljük ki? Hol kezdjük?

- $((\lambda y.y)\ 2) + 5 \rightarrow\ 2+5 \rightarrow 7$ (lusta)
- $(\lambda x.(x + 5))\ 2 \rightarrow 2+5 \rightarrow 7$ (mohó)

Az a λ-kifejezés amelyben nincs egyetlen redukálható részkifejezés sem, normál formában van.

A λ-kifejezések kiértékelésének célja tehát a normál forma elérése, egymás utáni β-redukciókkal

Normál formában va

- $\lambda y.y$
- $42$

Még redukálható

- $(\lambda x.(\lambda y.x + y - 18))\ 2\ 9$

Nincs normál formája

- $(\lambda x.x\ x)(\lambda x.x\ x)$ $\lambda$-kifejezésnek nincs $\beta$-normál formája.

Van egy tétel, ami azt mondja ki, hogy ha egy λ-kifejezésnek van normál formája, akkor ahhoz véges redukciós lépések sorozatával eljuthatunk, ha mindig a legbaloldalibb redukálható részkifejezést redukáljuk.

# Funkcionális programok

Egy funkcionális program felépítése:

- függvény definíciók, kezdeti kifejezés, operátorok, adatszerkezetek, …
- típus definíciók

Például – néhány egyszerű függvénydefiníció:

- nulla x = 0
- id  x = x
- inc  x = x + 1
- square  x = x * x
- squareinc  x = square  ( inc x )  - kompozíció
- kezdeti kifejezés: squareinc  7

A funkcionális program végrehajtása:

- a kezdeti kifejezés értékének meghatározása, kiértékelése, redukálása
- a normál forma meghatározása, - egyértelmű, nincs lehetőség további redukciós lépésre

A program végrehajtása a kezdeti kifejezésből kiinduló redukciós vagy más néven átírási lépések sorozata.

# Néhány kezdeti kifejezés

- Cleanben
  - `Start = sqrt 5.0;`
- Haskellben
  - `main = product $ [1..10]`
- A normál formák
  - `2.236068`
  - `3628800`
  - `64`

Egy-egy redukciós lépésben egy – valamely kiértékelési stratégia szerint kiválasztott – redukálható részkifejezésben, a redex-ben szereplő függvényhívás helyettesítődik a függvény törzsében megadott kifejezéssel a formális és aktuális paraméterek megfeleltetése mellett.

Normál formájú egy kifejezés, ha további redukcióra nincs lehetőség.

- Ez az átírási lépéssorozat végeredménye.

# Kiértékelési stratégiák

## Lusta kiértékelés

- A kifejezések kiértékelésekor először a legbaloldalibb legkülső redexet helyettesíti
  - Ha a kifejezés függvény megadással kezdődik, előbb a függvény definíció lesz alkalmazva
  - Az argumentumok kiértékelését csak szükség esetén végzi el.
- A lusta kiértékelés normalizáló kiértékelési módszer
  - Mindig megtalálja a normál formát, ha az létezik
    - Pl. Clean, Haskell, Miranda

# Kiértékelési stratégiák

Mohó kiértékelés

- a legbaloldalibb, legbelső redex, az argumentumok helyettesítése történik meg először
  - hatékonyabb, mint a lusta rendszer
  - de: nem mindig ér véget a kiértékelési folyamat
    - Lisp, SML, Hope

A két stratégia kombinálható

# Példák

Mohó kiértékelés:

- `squareinc  7`
- `-> square  ( inc 7 )`
- `-> square  ( 7 + 1 )  - először az argumentum(ok)`
- `-> square 8`
- `-> 8*8`
- `-> 64`

Lusta kiértékelés:

- `squareinc  7`
- `-> square  ( inc 7 )`
- `-> ( inc 7 ) * ( inc 7 )   - először a függvény`
- `-> (7 + 1) * ( 7 + 1 )`
- `-> 8*8`
- `-> 64`

# Függvény definíciók:

Nem mindegy, hogy milyen a kiértékelés, pl.:

- `f x = f x`
- `g x y = y` – ezek függvény definíciók
- `g 1 2` ⇒ `2`
- `g (f 3) 5` ⇒

  5 csak a lusta!

Milyen kiértékelés jó?

# A modern funkcionális nyelvek fő jellemzői

## Hivatkozási átláthatóság (referential transparency)

- A kifejezések értéke hivatkozásuk előfordulási helyétől független
  - ugyanaz a kifejezés a program szövegében mindenhol ugyanazt az értéket jelöli.
- A függvények kiértékelésének nincs mellékhatása
  - egy függvény kiértékelése nem változtathatja meg egy kifejezés értékét!
- Példa:
  - $f(x) + 2*f(x) = 3* f(x)$ – biztosan!

Emlékeztető példa, hogy ez nem mindig igaz:

```
int x = 1;
int f( int a ){
  ++x;
  return a+x;
}
int main(){
  int y = f(x) + 2*f(x);
  int z = 3*f(x);
}
```

$y \neq z$

# A modern funkcionális nyelvek fő jellemzői

## Feltételek megadási lehetősége (definition by cases)

- A függvény értékét az első, igaz egyenlőséghez tartozó kifejezés adja, mely kifejezést az aktuális paraméter értéke alapján kapjuk
- Példa: az argumentum előjelének a meghatározása:

```
elojel:: Int -> Int
elojel x| x < 0 = -1
elojel x| x > 0 = 1
elojel x| x == 0 = 0
main = print (elojel (-10))
```

# A modern funkcionális nyelvek fő jellemzői

Rekurzivitás lehetősége (definition by recursion):

- A függvények hivatkozhatnak önmagukra és kölcsönösen egymásra
- Példa: két egész szám  legnagyobb közös osztója:

```
lnko :: Int -> Int -> Int
lnko a b | (a == b) = a
lnko a b | (a > b) = lnko (a-b) b
lnko a b | (a < b) = lnko a (b-a)
main = print $ lnko 18 56
```

# A modern funkcionális nyelvek fő jellemzői

A függvények ugyanolyan értékek, mint az elemi típusértékhalmazok elemei.

Magasabbrendű függvénynek nevezzük azokat a függvényeket, amelyeknek valamelyik argumentuma vagy értéke maga is függvény.

- Példa

```
Twice f x = f ( f  x )
```

# A modern funkcionális nyelvek fő jellemzői

## Mintaillesztés

- Azt vizsgálja a kiértékelő rendszer, hogy az aktuális paraméterek értéke vagy alakja megfelel-e valamelyik egyenlőség bal oldalán megadott mintának.
  - A minták sorrendje meghatározza az eredményt
  - Az első illeszkedést keresi
- Példa: faktoriális (mintaillesztés és feltétel, rekurzióval):

```
fakt  0         =  1
fakt  n | n>0   =  n * fakt (n-1)
```

# A modern funkcionális nyelvek fő jellemzői

## Szigorú, statikus típusosság

- Típusdeklarációk megadása nem kötelező, de megköveteljük, hogy a kifejezések típusa a típuslevezetési szabályok által meghatározott legyen.
- Ez azt jelenti, hogy egy adott kifejezés legáltalánosabb típusát a fordítóprogram általában még akkor is meg tudja határozni a benne szereplő részkifejezések típusa alapján, ha a programszöveg készítője nem deklarálta a kifejezés típusát.

# A modern funkcionális nyelvek fő jellemzői

Iteratív adatszerkezet elemeinek és azok sorrendjének megadására alkalmas, a matematikában halmazok megadásánál alkalmazott jelölésrendszernek megfelelő nyelvi eszköz a Zermelo-Fraenkel halmazkifejezés.

A végtelen adatszerkezetek (listák, halmazok, sorozatok, vektorok) kiértékelése lusta kiértékelési módszerrel történik.

Például:

```
negyzetek n = [x^2 | x <- [1..n]]
```

# A modern funkcionális nyelvek fő jellemzői

A függvények típusát értelmezési tartományuk és értékkészletük megadásával határozzuk meg.

- Például

```
fakt: int -> int
```

Többváltozós függvények –> minden függvénynek egyetlen argumentuma lehet, de ez esetleg egy függvény!

## Egyszerű típuskonstrukciók

- Rendezett n-esek
- Iterált – véges vagy végtelen sorozatok

## Sorozatok

- Konstruktor – egy elemből és egy sorozatból új, az adott sorozatot balról kiterjesztő sorozatot készít – mintában is használhatóak
- Első elem (fejelem, head), lista fejelem nélküli maradéka (tail),…

## Például: sorozat elemeinek összege:

```
ossz :: [Int] ->Int
ossz [] = 0
ossz (x:xs) = x + ossz xs
```

Sorozatok generálhatók is

Például: Párosak valameddig:

```
paros n= [x | x <- [1..n], even x]
```

- (az x <- [ 1 .. n] generátorral előállítjuk az összes értéket 1-től n-ig, majd az even x szűrővel kiválogatjuk a megfelelőeket)

# Példák magasabb rendű függvényekre

- Filter – adott tulajdonságot teljesítő elemek leválogatása

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs)
 | p x = x : filter p xs
 | otherwise = filter p xs


even x =  x ‘mod’ 2 == 0
evens = filter even [0..]
filter even [3,2,2,1] → [2,2]
```

map: Elemenként alkalmazza a paraméterül kapott függvényt

Példa:

```
map inc [2,1,7]→ [3;2;8]
Definíció:
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs)= f x : map f xs
```

Példa:

```
odds = map (+1) evens
```

# Példa:

```
gyorsRendezes :: [Int]->[Int]
gyorsRendezes [] = []
gyorsRendezes (x:xs) = gyorsRendezes kisebbElemek ++
    [x] ++ (gyorsRendezes nagyobbElemek)
  where
    kisebbElemek   =  filter (<x) xs
    nagyobbElemek  =  filter (>x) xs
```

# Logikai programozás

Ha adott egy rendszer logikai formulákkal definiált modellje, kiegészítendő vagy eldöntendő kérdéseket fogalmazhatunk meg bizonyítandó  állítások formájában.

Kiegészítendő kérdés esetén a megfelelő tulajdonságú objektumok, tervek stb. meghatározása számítási folyamatnak tekinthető.

Ez a logikai programozás (LP) alapgondolata.

# Logikai programozás

A logikai program egy modellre vonatkozó állítások (axiómák) egy halmaza.

- Az állítások a modell objektumainak tulajdonságait és kapcsolatait, (relációit) írják le.

Ha például adottak állítások, amelyek arra vonatkoznak, hogy ki kinek az apja, ezek együtt a apja nevű, kettő aritású relációt írják le.

Ha megmondjuk, hogy kik a férfiak, vagy kik a nők, akkor az erre vonatkozó állítások együtt a férfi, ill. nő nevű, egy aritású relációt írják le.

Tények: (a legegyszerűbb logikai programok – atomi formulák)

```
apja('Ábrahám','Izsák').
apja('Ábrahám','Ismáel').
apja('Ábrahám','Ismeretlen').
apja('Izsák','Jákob').
apja('Izsák','Ézsau').

anyja('Sára','Izsák').
anyja('Hágár','Ismáel').
anyja('Rebeka','Jákob').
anyja('Rebeka','Ézsau').

férfi('Ábrahám').
férfi('Izsák').
férfi('Ismáel').
férfi('Jákob').
férfi('Ézsau').

nő('Sára').
nő('Hágár').
nő('Rebeka').
nő('Ismeretlen').
```

# Logikai programozás

Az állítások egy adott relációt meghatározó részhalmazát predikátumnak nevezzük.

- A program futtatása minden esetben egy, az állításokból következő tétel konstruktív bizonyítása, azaz – a logikai programozásban szokásos szóhasználattal – a programnak feltett kérdés vagy más néven cél megválaszolása.
- Ennek során a predikátumok eljárásokként működnek.

A ma használatos LP nyelvekben minden, a predikátumokat alkotó állítás tény vagy szabály lehet.

Prolog: Programming in logic (1972-től)

Kérdéseket tehetek fel:

```
|?- apja('Ábrahám','Izsák').
yes
|?- anyja('Sára','Jákob').
no
- Van-e olyan X, akinek az apja Izsák?
|?- apja('Izsák',X).
X = 'Jákob'  ;    X = 'Ézsau' ;
```

Összetett célok:

- `|?- apja('Ábrahám',X), nő(X).`
- `X = 'Ismeretlen' ;`

Ilyenkor a célsorozat megoldásai a részcélok közös megoldásai.

# Horn Klózok

A logikai programok  állításait Horn klózoknak is nevezik.

Egy Horn klóz azt fejezi ki, hogy egy adott állítás igaz, ha nulla vagy több másik állítás igaz. Pl.:

- ha (if) egy személy öreg és bölcs,
  akkor (then) ez a személy boldog
- ha (if) X az apja Y-nak és  Y az apja Z-nek,
  akkor (then) X a nagyapja Z-nek

Nulla vagy több állítás lehet az if-részben, és pontosan egy következtetés a then-részben.

Egy feltétel nélküli Horn klóz egy tényt ír le.

Egy Horn klóz a következő formában írható:

- $G_0 \leftarrow G_1, G_2, \ldots, G_n$.

Ennek a jelentése:

- Ha (If) $G_1$ ... $G_n$ mind igazak, akkor (then) $G_0$ is igaz.

A szabály baloldala a feje (head); a jobboldala a törzse (body)

- $G_0$... $G_n$ a célok

A törzsben lévő célokat alcéloknak (subgoals) hívjuk

- A vesszőket logikai AND szimbólumként kell olvasni
- Megjegyzés: a $\leftarrow$ csak azt jelenti, hogy „ha” (“if”) - nem pedig azt, hogy „akkor és csak akkor” (“if and only if”) – lehetnek más szabályok is $G_0$ –ra!
  - (a $\leftarrow$ jelölése :- )
  - Speciális eset, ha $n=0$, ekkor $G_0$ egy egyszerű tény, ami mindig igaz.

A logikai programozás lehetővé teszi, hogy a programozó megadja tények és if-then szabályok egy listáját, és azután a rendszer automatikusan el tudja dönteni, hogy egy állítás igaz-e, vagy meg tudja mondani, mire igaz, stb.

Szabályok:

- `'Ábrahám lánya'(X) :- apja('Ábrahám',X), nő(X).`
- `szülője(X,Y) :- anyja(X,Y).`
- `szülője(X,Y) :- apja(X,Y).`

A szülője reláció az állítások által definiált relációk uniója.

```
fia(X,Y) :- szülője(Y,X), férfi(X).
lánya(X,Y) :- szülője(Y,X), nő(X).
```

A fia reláció a szülője és a férfi relációk metszete, és hasonlóan adódik a lánya reláció is.

```
nagyszülője(X,Y) :- szülője(X,Z), szülője(Z,Y).
```

Ez eltér az előzőektől abban, hogy a jobb oldalán új változó is előfordul. Így kétféle olvasata is van:

- Minden X,Y,Z-re nagyszülője(X,Y), ha szülője(X,Z) és szülője(Z,Y).
- Minden X,Y-ra nagyszülője(X,Y), ha van olyan Z, hogy szülője(X,Z) és szülője(Z,Y).

Ezek a deklaratív olvasatai a szabálynak. Ezzel szemben áll a procedurális olvasat, ami a logikai programok futtatásához, vagyis a konstruktív bizonyítási eljáráshoz kapcsolódik.

Minden esetben felteszünk egy kérdést.

- Ez a bizonyítandó cél(sorozat).
- Ezután a részcélok bizonyítása, és így kiejtése a feladat. Felülről lefelé érdemes …

# Keresési fa a részcélokhoz

```
?- nagyszülője('Ábrahám',X).
     szülője('Ábrahám',Z),szülője(Z,X).
        anyja('Ábrahám',Z),szülője(Z,X).  % meghiúsul
        apja('Ábrahám',Z),szülője(Z,X).
          { Z <- 'Izsák' }
            szülője('Izsák',X).
              anyja('Izsák',X).           % meghiúsul
              apja('Izsák',X).
                { X <- 'Jákób' }          % 1. megoldás
                { X <- 'Ézsau' }          % 2. megoldás
          { Z <- 'Ismáel' }
            szülője('Ismáel',X).
              anyja('Ismáel',X).          % meghiúsul
              apja('Ismáel',X).           % meghiúsul
          { Z <- 'Ismeretlen' }
            szülője('Ismeretlen',X).
              anyja('Ismeretlen',X).      % meghiúsul
              apja('Ismeretlen',X).       % meghiúsul
```

# Rekurzív szabályok

Tegyük fel, hogy azt szeretnénk leírni, X mikor őse Y-nak.

- Világos, hogy akkor őse, ha szülője, vagy valamelyik ősének a szülője.

Látható, hogy a relációnak két esete van, és az egyik eset rekurzív.

- Ezt ennek megfelelően egy nem rekurzív és egy rekurzív szabállyal fejezhetjük ki:
  - `őse(X,Y) :- szülője(X,Y).`
  - `őse(X,Y) :- szülője(X,Z), őse(Z,Y).`
  - kérdés: `?-őse('Ábrahám',X)`

A keresőfa előállításához célszerű a legbal részcélkiválasztási módszert alkalmazni

- Általában is érdemes a nem rekurzívval kezdeni…

## Felhasználási területek

- mesterséges intelligencia
- számítógépes nyelvészet
- adatbáziskezelés …