Deklaratív programozás

Hanák Péter
hanak@inf.bme.hu

Irányítástechnika és Informatika Tanszék

Szeredi Péter
szeredi@cs.bme.hu

Számítástudományi és Információelméleti Tanszék

KÖVETELMÉNYEK, TUDNIVALÓK

Deklaratív programozás: tudnivalók

- Honlap, levelezési lista
  - Honlap: `<http://dp.iit.bme.hu>`
  - Levlista: `<http://www.iit.bme.hu/mailman/listinfo/dp-l>`.
    A listatagoknak szóló levelet a `<dp-l@www.iit.bme.hu>` címre kell küldeni.
    Csak a feliratkozottak levele jut el moderátori jóváhagyás nélkül a listatagokhoz.
- Jegyzet
  - Szeredi Péter, Benkő Tamás: Deklaratív programozás. Bevezetés a logikai programozásba (1000 Ft)
  - Hanák D. Péter: Deklaratív programozás. Bevezetés a funkcionális programozásba (850 Ft)
  - A nyomtatott változat **KORLÁTOZOTT SZÁMBAN** megvásárolható a SZIT tanszék V2 épületbeli titkárságán a V2.104 szobában, Bazsó Lászlónénál, 10:30-12:00 (hétfő-péntek) és 13:30-15:30 (hétfő-csütörtök), **KIZÁRÓLAG 10-es kötegekben!!!!**
  - Kellő számú további igény esetén megszervezzük az újranyomtatást.

Deklaratív programozás: tudnivalók (folyt.)

Fordító- és értelmezőprogramok

- SICStus Prolog — 3.12 verzió (licensz az ETS-en keresztül kérhető)
- Moscow SML (2.0, szabad szoftver)
- Mindkettő telepítve van a `kempelen.inf.bme.hu`-n
- Mindkettő letölthető a honlapról (linux, Win95/98/NT)
- Webes gyakorló felület az ETS-ben (ld. honlap)
- Kézikönyvek HTML-, ill. PDF-változatban
- Más programok: swiProlog, gnuProlog, poly/ML, smlnj
- emacs-szövegszerkesztő SML-, ill. Prolog-módban (linux, Win95/98/NT)

Deklaratív programozás: félévközi követelmények

Nagy házi feladat (NHF)

- Programozás mindkét nyelven (Prolog, SML)
- Mindenkinek önállóan kell kódolnia (programoznia)!
- Hatékony (időlimit!), jól dokumentált („kommentezett”) programok
- A két programhoz közös, 5–10 oldalas fejlesztői dokumentáció (TXT, TeX/LaTeX, HTML, PDF, PS; de nem DOC vagy RTF)
- Kiadás a 6. héten, a honlapon, letölthető keretprogrammal
- Beadás a 12. héten; elektronikus úton (ld. honlap)
- A beadáskor és a pontozáskor külön-külön tesztsorozatot használunk (nehézségben hasonlókat, de nem azonosakat)
- A minden tesztesetet hibátlanul megoldó programok *létraversenyen* vesznek részt (hatékonyság, gyorsaság plusz pontokért)

Deklaratív programozás: félévközi követelmények (folyt.)

Nagy házi feladat (folyt.)

- Nem kötelező, de *nagyon* ajánlott!
- Beadható csak az egyik nyelvből is
- A beadási határidőig többször is beadható, csak az utolsót értékeljük
- Pontozása mindkét nyelvből:
  - helyes és időkorláton belüli futás esetén a 10 teszteset mindegyikére 0,5-0,5 pont, összesen max. 5 pont, feltéve, hogy legalább 4 teszteset sikeres
  - a dokumentációra, a kód olvashatóságára, kommentezettségére max. 2,5 pont
  - tehát nyelvenként összesen max. 7,5 pont szerezhető
- A NHF súlya az osztályzatban: 15% (a 100 pontból 15)

Deklaratív programozás: félévközi követelmények (folyt.)

Kis házi feladatok (KHF)

- 2-3 feladat Prologból is, SML-ből is
- Beadás elektronikus úton (ld. honlap)
- Nem kötelező, de *nagyon* ajánlott
- Minden feladat jó megoldásáért 1-1 jutalompont jár

Gyakorló feladatok

- Nem kötelező, de a sikeres ZH-hoz, vizsgához *elengedhetetlen*!
- Gyakorlás az ETS rendszerben (lásd honlap)

Konzultációk

- Rendszeres – számítógép melletti – konzultációs lehetőség

Deklaratív programozás: félévközi követelmények (folyt.)

Nagyzárthelyi, pótzárthelyi (NZH, PZH, PPZH)

- A zárthelyi kötelező, semmilyen jegyzet, segédlet nem használható!
- 40%-os szabály (nyelvenként a maximális részpontszám 40%-a kell az eredményességhez). Kivétel: a korábban aláírást szerzett hallgató zárthelyin szerzett pontszámát az alsó ponthatártól függetlenül beszámítjuk a félévvégi osztályzatba. A korábbi félévekben szerzett pontokat nem számítjuk be!
- Az NZH a 10. héten, a PZH az utolsó oktatási hetekben lesz
- A PPZH-ra indokolt esetben, ismétlővizsga-jelleggel a vizsgaidőszak első három hetében egyetlen alkalommal adunk lehetőséget
- Az NZH anyaga az első két blokk (nagyjából az 1.-8. hét) tananyaga
- A PZH, ill. a PPZH anyaga azonos az NZH anyagával
- A zárthelyi súlya az osztályzatban: 15% (a 100 pontból 15)
- Több zárthelyi megírása esetén a zárthelyikre kapott pontszámok közül a *legnagyobbat* vesszük figyelembe

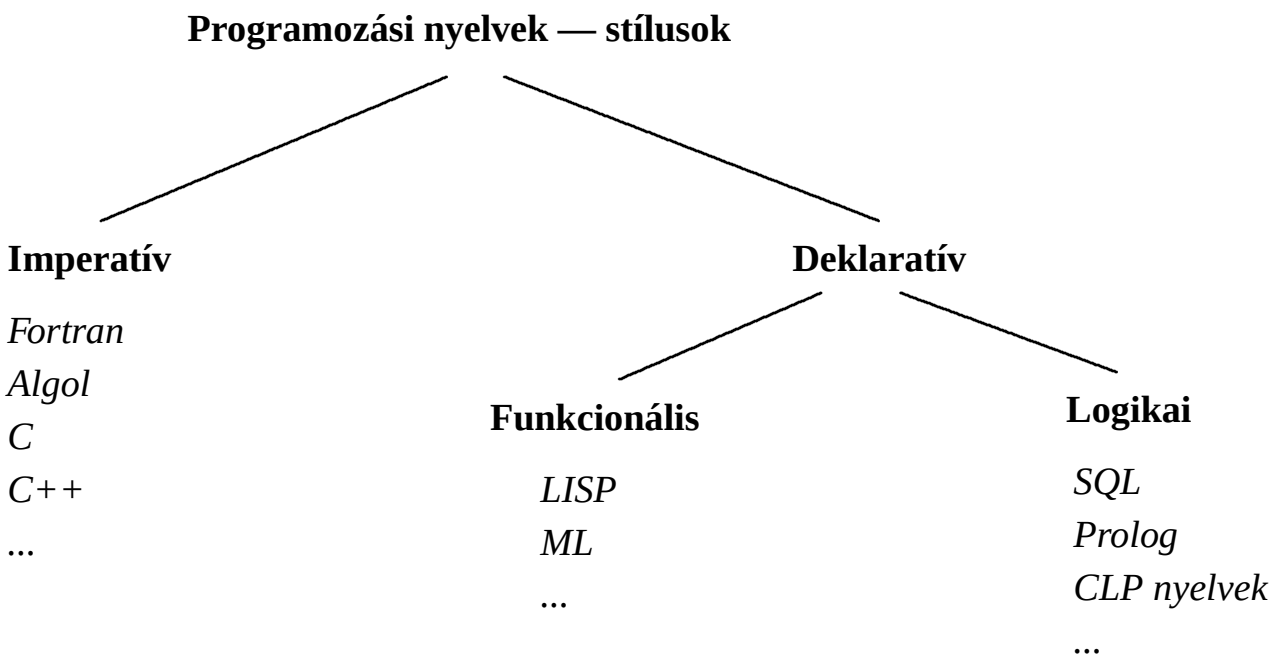
Deklaratív programozás: vizsga

Vizsga

- Vizsgára az a hallgató bocsátható, aki aláírást szerzett a jelen félévben vagy a jelen félévet megelőző négy félévben
- A vizsga szóbeli, felkészülés írásban
- Prolog, SML: több kisebb feladat (programírás, -elemzés) kétszer 35 pontért
- A vizsgán szerezhető max. 70 ponthoz adjuk hozzá a **jelen** félévben félévközi munkával szerzett pontokat: ZH: max. 15 pont, NHF: max. 15 pont, továbbá a pluszpontokat (KHF, létraverseny)
- *Korábbi* félévben szerzett pontokat *nem* számítunk be!
- A vizsgán semmilyen jegyzet, segédlet nem használható, de lehet segítséget kérni
- Ellenőrizzük a nagy házi feladat és a zárthelyi „hitelességét”
- 40%-os szabály (nyelvenként a max. részpontszám 40%-a kell az eredményességhez)
- Korábbi vizsgakérdések a honlapon találhatók

DEKLARATÍV ÉS IMPERATÍV PROGRAMOZÁS

Programozási nyelvek osztályozása



Imperatív és deklaratív programozási nyelvek

- Imperatív program
  - felszólító módú, utasításokból áll
  - változó: változtatható értékű memóriahely
  - C nyelvű példa:

```
int pow(int a, int n) {  // pow(a,n) = a ^ n
  int p = 1;             // Legyen p értéke 1!
  while (n > 0) {        // Amíg n>0 ismételd ezt:
    n = n-1;             //    Csökkentsd n-et 1-gyel!
    p = p*a;    }        //    Szorozd p-t a-val!
  return p;           }  // Add vissza p végértékét
```

- Deklaratív program
  - kijelentő módú, egyenletekből, állításokból áll
  - változó: egyetlen rögzített értékkel bír, amely a programírás idején még ismeretlen
  - SML példa:

```
fun pow(a, n) =
    if n > 0             (* Ha n > 0 *)
      then a*pow(a,n-1)  (* akkor a^n = a*a^(n-1) *)
      else 1             (* egyébként a^n = 1 *)
```

Deklaratív programozás imperatív nyelven

- Lehet pl. C-ben is deklaratívan programozni,
  - ha nem használunk: értékadó utasítást, ciklust, ugrást, stb.,
  - amit használhatunk: (rekurzív) függvények, if-then-else
- Példa (a `pow` függvény deklaratív változata a `powd`):

```
/* powd(a,n) = a^n */
int powd(int a, int n) {
  if (n > 0)                    /* Ha n > 0 */
    return a*powd(a,n-1);       /* akkor a^n = a*a^(n-1) */
  else
    return 1;                   /* egyébként a^n = 1 */
}
```

- A (fenti típusú) rekurzió költséges, nem valósítható meg konstans tárigénnyel `:-(`.

Hatékony deklaratív programozás

- A rekurziónak van egy hatékonyan megvalósítható változata
  - Példa: döntsük el, hogy egy `a` természetes szám előáll-e egy `b` szám hatványaként:

```
/* ispow(a,b) = 1 <=> exits i, such that b^i = a. Precondition: a,b
> 0 */
int ispow(int a, int b) {
                                                    /* again: */
  if (a == 1)          return 1;
  else if (a%b == 0)   return ispow(a/b, b);     /* a = a/b; goto again; */
  else                 return 0;
}
```

  - Itt a rekurzív hívás a kommentben jelzett értékadásokkal és ugrással helyettesíthető!
  - Ez azért tehető meg, mert a rekurzióból való visszatérés után *azonnal* kilépünk az adott függvényhívásból.
- Az ilyen függvényhívást **jobbrekurzió**nak vagy **terminális rekurzió**nak nevezzük
- A Gnu C fordító megfelelő optimalizálási szint mellett (`gcc -O2`) a rekurzív definícióból is a nem-rekurzív (jobboldali) kóddal azonos kódot generál!

Jobbrekurzív függvények

- Lehet-e jobbrekurzív kódot írni a hatványozási (`pow(a,n)`) feladatra?
  - A gond az, hogy a rekurzióból „kifelé jövet” már nem csinálhatunk semmit,
  - tehát a végeredménynek az utolsó hívás belsejében elő kell állnia!
  - A megoldás: segédfüggvény definiálása, amelyben további, ún. gyűjtőargumentumokat helyezünk el.
- A `pow(a,n)` jobbrekurzív megvalósítása:

```
/* Segédfüggvény: powa(a, n, p) = p*a^n */
int powa(int a, int n, int p) {
  if (n > 0)
    return powa(a, n-1, p*a);
  else
    return p;
}

int powr(int a, int n){
  return powa(a, n, 1);
}
```

Cékla: A „Cé” nyelv egy deKLAratív része

- Megszorítások:
  - Típusok: csak `int`
  - Utasítások: `if-then-else`, `return`, blokk
  - Feltétel-rész: `(` ⟨kif⟩ ⟨hasonlító-op⟩ ⟨kif⟩ `)`
    - ⟨hasonlító-op⟩: `<` | `>` | `==` | `\=` | `>=` | `<=`
  - kifejezések: változókból és számkonstansokból kétargumentumú operátorokal és függvényhívásokkal épülnek fel
    - ⟨aritmetikai-op⟩: `+` | `-` | `*` | `/` | `%` |
- Egy cékla fordító letölthető a tárgy honlapjáról.

A Cékla nyelv szintaxisa

- A szintaxist BNF jelöléssel adjuk meg, kiterjesztés:
  - ismétlés (0, 1, vagy többszöri ismétlés): «ismétlendő»... vagy [«ismétlendő»]...
- A program szintaxisa

```
<program> ::=                 <function_definition> ...
<function_definition> ::=     <head> <block>
<head> ::=                    <type> <identifier> (<formal_args>)
<type> ::=                    int
<formal_args> ::=             <formal_arg>[, <formal_arg>]... | <empty>
<formal_arg> ::=              <type> <identifier>
<block> ::=                   { <declaration>... <statement>...}
<declaration> ::=             <type> <declaration_elem> <declaration_elem>...;
<declaration_elem> ::=        <identifier> = <expression>
```

Cékla szintaxis: folytatás

- Utasítások szintaxisa

```
<statement> ::=             if <test> <statement> <optional_else_part>
                          | <block>
                          | return <expression> ;
                          | ;
<optional_else_part> ::=    else <statement> | <empty>
<test> ::=                  (<expression> <comparison_op> <expression>)
```

- Kifejezések szintaxisa

```
<expression> ::=            <term> [<additive_op> <term>]...
<term> ::=                  <factor> [<multiplicative_op> <factor>]...
<factor> ::=                <identifier>
                          | <identifier> (<actual_args>)
                          | <constant>
                          | (<expression>)
<constant> ::=              <integer>
<actual_args> ::=           <expression> [, <expression>]... | <empty>
<comparison_op> ::=         < | > | == | \= | >= | <=
<additive_op> ::=           + | -
<multiplicative_op> ::=     * | / | %
```

1. kis házi feladat

- Egy s pozitív szám n-alapú megfordítását a következőképpen definiáljuk:
  - Írjuk fel az s számot n alapú számrendszerben, a kapott (0 és $n-1$ közé eső) „számjegyeket” jelölje $a_1, \ldots, a_k$. Számeleji 0 számjegyeket nem engedünk meg, tehát, $a_1 \neq 0$.
  - Képezzük a számjegysorozat megfordítását: $a_k, \ldots, a_1$ (ez már kezdődhet 0-val).
  - A fordított számjegysorozatot n alapú számrendszerben felírt számnak tekintjük. Az így kapott számértéket nevezzük az s szám n-alapú megfordításának.
- Példák: 123 10-alapú megfordítása 321, 5 2-alapú megfordítása 5 (mert $5_{(2)}$ = 101), 8 2-alapú megfordítása 1 (mert $8_{(2)}$ = 1000)
- Megirandó egy program Cékla nyelven a következő feladat megoldására
  - A fő függvény: `forditottja(s, n) = f` jelentése: az `s` szám `n`-alapú megfordítása az `f` érték.
  - Példák:
    - `forditottja(4,2) = 1`, mert $4_{(2)}$ = 1 0 0
    - `forditottja(1023,10) = 3201`
    - `forditottja(5,3) = 7` mert $5_{(3)}$ = 1 2 és $7_{(3)}$ = 2 1

Egy kicsit bonyolultabb Cékla program

- A feladat: Egy 0 és 1023 közé eső *num* egész számot egy olyan 10-jegyű decimális számmá kell konvertálni, amely csak a 0 és 1 jegyekből áll, úgy, hogy ha ezt a 0-1 sorozatot bináris számként olvassuk ki, akkor a *num* értéket kapjuk, pl. `bin(5) = 101, bin(37) = 100101`.
- Megoldás (imperatív) C-ben és Céklában:

```
int bin(int num) {
  int bp = 512;
  int dp = 1000000000;
  int bin = 0;
  while (bp > 0) {
    if (num >= bp) {
      num = num-bp;
      bin = bin+dp;
    }
    bp = bp / 2;
    dp = dp / 10;
  }
  if (num > 0)
    return -1;
  else
    return bin;
}
```

```
int bina(int num,
         int bp,
         int dp,
         int bin) {
       if (bp > 0) {
         if (num >= bp)
           return bina(num-bp,bp/2,dp/10,bin+dp);
         else
           return bina(num,   bp/2,dp/10,bin);
         }
         if (num > 0)
           return -1;
         else
           return bin;
     }
int bind(int num) {
  return bina(num, 512, 1000000000, 0); }
```

Deklaratív programozási nyelvek — a Cékla tanulságai

- Mit vesztettünk?
  - a megváltoztatható változókat,
  - az értékadást, ciklus-utasítást stb.,
  - általánosan: a megváltoztatható állapotot
- Hogyan tudunk mégis állapotot kezelni deklaratív módon?
  - az állapotot a (segéd)függvény paraméterei tárolják,
  - az állapot változása (vagy helybenmaradása) explicit!
- Mit nyertünk?
  - Állapotmentes szemantika: egy nyelvi elem értelme nem függ a programállapottól
    - Hivatkozási átlátszóság (referential transparency) — pl. ha $f(x) = x^2$, akkor $f(a)$ **helyettesíthető** a^2-tel.
    - Egyszeres értékadás (single assignment) — párhuzamos végrehajthatóság.
  - A deklaratív programok **dekomponálhatók**:
    - A program részei egymástól **függetlenül** megírhatók, tesztelhetők, verifikálhatók.
    - A programon könnyű következtetéseket végezni, pl. helyességét bizonyítani.

Deklaratív programozási nyelvek — jelmondat

- MIT és nem HOGYAN (WHAT rather than HOW): a *megoldás módja* helyett inkább a megoldandó *feladat leírását* kell megadni
- A gyakorlatban mindkét szemponttal foglalkozni kell — kettős szemantika:
  - deklaratív szemantika — MIT (milyen feladatot) old meg a program;
  - procedurális szemantika — HOGYAN oldja meg a program a feladatot.

Deklaratív programozás — miért tanítjuk?

- Új, magasszintű programozási elemek
  - rekurzió
  - mintaillesztés
  - visszalépéses keresés
- Új gondolkodási stílus
  - dekomponálható programok: a programrészek (relációk, függvények) önálló jelentéssel bírnak
  - verifikálható programok: a kód és a jelentés összevethető
- Új alkalmazási területek
  - szimbolikus alkalmazások
  - következtetési módszerekre épülő megoldások
  - nagyfokú megbízhatóságot igénylő rendszerek
  - párhuzamosság, pl. többmagos processzorok!

Egy példa: párbeszéd egy 50 soros Prolog programmal

```
| ?- párbeszéd.
|: Magyar legény vagyok én.
Felfogtam.
|: Ki vagyok én?
Magyar legény
|: Péter kicsoda?
Nem tudom.
|: Péter tanuló.
Felfogtam.
|: Péter jó tanuló.
Felfogtam.
|: Péter kicsoda?
tanuló
jó tanuló
|: Boldog vagyok.
Felfogtam.
```

```
|: Te egy Prolog program vagy.
Felfogtam.
|: Ki vagyok én?
Magyar legény
Boldog
|: Okos vagy.
Felfogtam.
|: Te vagy a világ közepe.
Felfogtam.
|: Ki vagy  te?
egy Prolog program
Okos
a világ közepe
|: Valóban?
Nem értem.
|: Unlak.
Én is.
```

BEVEZETÉS A LOGIKAI PROGRAMOZÁSBA

A logikai programozás alapgondolata

- Logikai programozás (LP):
  - Programozás a matematikai logika segítségével
    - egy logikai program nem más mint **logikai állítások halmaza**
    - egy logikai **program futása** nem más mint **következtetési folyamat**
  - De: a logikai következtetés óriási keresési tér bejárását jelenti
    - szorítsuk meg a logika nyelvét
    - válasszunk egyszerű, ember által is követhető következtetési algoritmusokat
  - Az LP máig legelterjedtebb megvalósítása a **Prolog** = **Pro**gramozás **log**ikában (**Pro**gramming in **log**ic)
    - az elsőrendű logika egy erősen megszorított résznyelve az ún. **definit-** vagy **Horn-klóz**ok nyelve,
    - végrehajtási mechanizmusa: **mintaillesztés**es eljáráshíváson alapuló **visszalépés**es keresés.

Az előadás LP részének áttekintése

- **1. blokk:** A Prolog nyelv alapjai (6 előadás)
  - Logikai háttér
  - Szintaxis
  - Végrehajtási mechanizmus
- **2. blokk:** Prolog programozási módszerek (6 előadás)
  - A legfontosabb beépített eljárások
  - Fejlettebb nyelvi és rendszerelemek
- Kitekintés: Új irányzatok a logikai programozásban (1 előadás)

A Prolog/LP rövid történeti áttekintése

1960-as évek	Első tételbizonyító programok
1970-72	A logikai programozás elméleti alapjai (R A Kowalski)
1972	Az első Prolog interpreter (A Colmerauer)
1975	A második Prolog interpreter (Szeredi P)
1977	Az első Prolog fordítóprogram (D H D Warren)
1977–79	Számos kísérleti Prolog alkalmazás Magyarországon
1981	A japán 5. generációs projekt a logikai programozást választja
1982	A magyar MProlog az egyik első kereskedelmi forgalomba kerülő Prolog megvalósítás
1983	Egy új fordítási modell és absztrakt Prolog gép (WAM) megjelenése (D H D Warren)
1986	Prolog szabványosítás kezdete
1987–89	Új logikai programozási nyelvek (CLP, Gödel stb.)
1990–. . .	Prolog megjelenése párhuzamos számítógépeken
	Nagyhatékonyságú Prolog fordítóprogramok
	.....

Információk a logikai programozásról

- Prolog megvalósítások:
  - SWI Prolog: `http://www.swi-prolog.org/`
  - SICStus Prolog: `http://www.sics.se/sicstus`
  - GNU Prolog: `http://pauillac.inria.fr/~diaz/gnu- prolog/`
- Hálózati információforrások:
  - The WWW Virtual Library: Logic Programming: `http://www.afm.sbu.ac.uk/logic-prog`
  - CMU Prolog Repository: (a `http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/prolog/` címen belül)
    - Főlap: `0.html`
    - Prolog FAQ: `faq/prolog.faq`
    - Prolog Resource Guide: `faq/prg_1.faq`, `faq/prg_2.faq`

Magyar nyelvű Prolog irodalom

Farkas Zsuzsa, Futó Iván, Langer Tamás, Szeredi Péter:
Az MProlog programozási nyelv.
Műszaki Könyvkiadó, 1989
jó bevezetés, sajnos az MProlog beépített eljárásai nem szabványosak.

Márkusz Zsuzsa: Prologban programozni könnyű.
Novotrade, 1988
mint fent

Futó Iván (szerk.): Mesterséges intelligencia. (9.2 fejezet, Szeredi Péter)
Aula Kiadó, 1999
csak egy rövid fejezet a Prologról

Peter Flach: Logikai Programozás. Az intelligens következtetés példákon keresztül.
Panem — John Wiley & Sons, 2001
jó áttekintés, inkább elméleti érdeklődésű olvasók számára

English Textbooks on Prolog

- Logic, Programming and Prolog, 2nd Ed., by Ulf Nilsson and Jan Maluszynski, Previously published by John Wiley & Sons Ltd. (1995)

  Downloadable as a pdf file from `http://www.ida.liu.se/~ulfni/lpp`
- Prolog Programming for Artificial Intelligence, 3rd Ed., Ivan Bratko, Longman, Paperback - March 2000
- The Art of PROLOG: Advanced Programming Techniques, Leon Sterling, Ehud Shapiro, The MIT Press, Paperback - April 1994
- Programming in PROLOG: Using the ISO Standard, C.S. Mellish, W.F. Clocksin, Springer-Verlag Berlin, Paperback - July 2003

Első Prolog programunk: hatványellenőrzés

- Egy egyszerű példa Céklában és Prologban:

```
/* ispow(a,b) = 1 <=> exits i, such that b^i = a. Precondition: a,b > 0 */

int ispow(int num, int base) {
  if (num == 1)
    return 1;
  else if (num%base == 0)
    return ispow(num/base, base);
  else
    return 0;
}
```

```
ispow(Num, Base) :-
    (   Num =:= 1
    ->  true
    ;   Num rem Base =:= 0,
        Num1 is Num//Base,
        ispow(Num1, Base)
    ).
```

- `ispow` egy Prolog **predikátum**, azaz Boole értékű eljárás.
- Az eljárás egyetlen klózból áll, amely *Fej* : - *Törzs* alakú.
- Az eljárásfejben a paraméterek a `Num` és `Base` változók (**nagybetűsek**!)
- A törzs egyetlen célt tartalmaz, amely egy **feltételes szerkezet**:
  `if Felt then Akkor else Egyébként` ≡ `( Felt -> Akkor ; Egyébként )`
- A „`true`”, „`A =:= B`” és „`A is B`” szerkezetek beépített predikátumok hívásai.

Néhány beépített predikátum

- Egyesítés: `X = Y`: az `X` és `Y` **szimbolikus** kifejezések változók behelyettesítésével azonos alakra hozhatók (és el is végzi a behelyettesítéseket).
- Aritmetikai predikátumok
  - `X is Kif`: A `Kif` **aritmetikai** kifejezést kiértékeli és **értékét** egyesíti `X`-szel.
  - `Kif1<Kif2, Kif1=<Kif2, Kif1>Kif2, Kif1>=Kif2, Kif1=:=Kif2, Kif1=\=Kif2`: A `Kif1` és `Kif2` aritmetikai kifejezések értéke a megadott relációban van egymással (`=:=` jelentése: aritmetikai egyenlőség, `=\=` jelentése aritmetikai nem-egyenlőség).
  - Ha `Kif, Kif1, Kif2` valamelyike nem **tömör** (változómentes) aritmetikai kifejezés ⇒hiba.
  - Legfontosabb aritmetikai operátorok: `+, -, *, /, rem, //` (egész-osztás)
- Kiíró predikátumok
  - `write(X)`: Az `X` Prolog kifejezést kiírja.
  - `nl`: Kiír egy újsort.
- Egyéb predikátumok
  - `true, fail`: Mindig sikerül ill. mindig meghiúsul.
  - `trace, notrace`: A (teljes) nyomkövetést be- ill. kikapcsolja.

Programfejlesztési beépített predikátumok

- `consult(File)` vagy `[File]`: A `File` állományban levő programot beolvassa és értelmezendő alakban eltárolja. (`File = user` ⇒ terminálról olvas.)
- `listing` vagy `listing(Predikátum)`: Az értelmezendő alakban eltárolt összes ill. adott nevű predikátumokat kilistázza.
- `compile(File)`: A `File` állományban levő programot beolvassa, lefordítja.
- A lefordított alak gyorsabb, de nem listázható, **kicsit** kevésbe pontosan nyomkövethető.
- `halt`: A Prolog rendszer befejezi működését.

```
> sicstus
SICStus 3.11.0 (x86-linux-glibc2.3): Mon Oct 20 15:59:37 CEST 2003
| ?- consult(ispow).
% consulted /home/user/ispow.pl in module user, 0 msec 376 bytes
yes
| ?- ispow(8, 3).
no
| ?- ispow(8, 2).
yes
| ?- listing(ispow).
(...)
yes
| ?- halt.
>
```

Általános (nem Boole-értékű) függvények Prologban

- Példa: Természetes számok hatványozása Céklában és Prologban:

```
/* powd(a,n) = a^n */
int powd(int a, int n) {
  if (n > 0)
    return a*powd(a,n-1);


  else
    return 1;
}
```

```
/* powd(A, N, P): A^N = P. */
powd(A, N, P) :-
        (   N > 0
        ->  N1 is N-1,
            powd(A, N1, P1),
            P is A*P1
        ;   P = 1
        ).

| ?- powd(2, 8, P).
P = 256 ?
```

- A 2-argumentumú `powd` függvénynek a 3-argumentumú `powd` predikátum felel meg.
- A függvény két argumentumának a predikátum első két, **bemenő**, azaz behelyettesített argumentuma felel meg.
- A függvény eredménye a predikátum utolsó, kimenő argumentuma, amely általában behelyettesítetlen változó.

Predikátum definiálása több klózzal

- A feltételes szerkezet nem alap-építőeleme a Prolog nyelvnek (az első Prologokban nem is volt)
- Helyette több **egymást kizáró** klózt is lehet alkalmazni:

```
/* powd(A, N, P): A^N = P. */
powd(A, N, P) :-
        (   N > 0
        ->  N1 is N-1,
            powd(A, N1, P1),
            P is A*P1
        ;   P = 1
        ).
```

```
powd2(A, N, P) :-
        N > 0,
        N1 is N-1,
        powd2(A, N1, P1),
        P is A*P1.
powd2(A, N, 1) :-     N =< 0.
```

- Ha egy predikátum több klózból áll, a Prolog **mindegyiket** megkisérli felhasználni a futásban:
  - ha `pow2` második paramétere (`N`) pozitív, akkor az első klózt használja,
  - egyébként (azaz ha `N =< 0`) a második klózt.
- Ha `powd2` második klózaként `powd(A, 0, 1)` állna akkor negatív kitevő esetén a hívás meghiúsulna (egyik klóz sem lenne alkalmazható).
- Általában nem kell kizáró feltételeket alkalmazni, így egy kérdésre több választ is kaphatunk!

```
gyöke(A, B, C, X) :- X is (-B + sqrt(B*B-4*A*C))/(2*A).
gyöke(A, B, C, X) :- X is (-B - sqrt(B*B-4*A*C))/(2*A).
```

Több választ adó predikátumok — a családi kapcsolatok példája

- Adatok

  Egy gyerek–szülő kapcsolat, pl.

gyerek	szülő
Imre	István
Imre	Gizella
István	Géza
István	Sarolta
Gizella	Civakodó Henrik
Gizella	Burgundi Gizella

- A feladat:

  Definiálandó az unoka–nagyszülő kapcsolat, pl. keressük egy adott személy nagyszüleit.

A nagyszülő feladat — Prolog megoldás

```
% szuloje(Gy, Sz):Gy szülője Sz.
szuloje('Imre', 'István').
szuloje('Imre', 'Gizella').
szuloje('István', 'Géza').
szuloje('István', 'Sarolt').
szuloje('Gizella',
            'Civakodó Henrik').
szuloje('Gizella',
            'Burgundi Gizella').

%  Gyerek nagyszülője Nagyszulo.
nagyszuloje(Gyerek, Nagyszulo) :-
    szuloje(Gyerek, Szulo),
    szuloje(Szulo, Nagyszulo).
```

```
% Kik Imre nagyszülei?
| ?- nagyszuloje('Imre', NSz).
NSz = 'Géza' ? ;
NSz = 'Sarolt' ? ;
NSz = 'Civakodó Henrik' ? ;
NSz = 'Burgundi Gizella' ? ;
no
% Kik Géza unokái?
| ?- nagyszuloje(U, 'Géza').
U = 'Imre' ? ;
no
```

Adatstruktúrák deklaratív nyelvekben — példa

- A bináris fa adatstruktúra
  - vagy egy csomópont (`node`), amelynek két részfája van mutat (`left,right`)
  - vagy egy levél (`leaf`), amely egy egészt tartalmaz
- Binárisfa-struktúrák különböző nyelveken

```
% Struktúra deklarációk C-ben
enum treetype Node, Leaf;
struct tree {
  enum treetype type;
  union {
    struct { struct tree *left;
             struct tree *right;
           } node;
    struct { int value;
           } leaf;
  } u;
};
```

```
% Adattípus-deklaráció SML-ben

datatype Tree =
         Node of Tree * Tree
       | Leaf of int

% Adattípus-leírás Prologban

 :- type tree --->
        node(tree, tree)
      | leaf(int).
```

Bináris fák összegzése

- Egy bináris fa levélösszegének kiszámítása:
  - csomópont esetén a két részfa levélösszegének összege
  - levél esetén a levélben tárolt egész

```
% C nyelvű (deklaratív) függvény
int sum_tree(struct tree *tree)
{
  switch(tree->type) {
  case Leaf:
   return tree->u.leaf.value;
  case Node:
   return
    sum_tree(tree->u.node.left) +
    sum_tree(tree->u.node.right);
    }
}
```

```
% Prolog eljárás (predikátum)
sum_tree(leaf(Value), Value).
sum_tree(node(Left,Right), S) :-
        sum_tree(Left, S1),
        sum_tree(Right, S2),
        S is S1+S2.
```

Bináris fák összegzése

- Prolog példafutás

```
% sicstus -f
SICStus 3.10.0 (x86-linux-glibc2.1): Tue Dec 17 15:12:52 CET 2002
Licensed to BUTE DP course
| ?- consult(tree).
% consulting /home/szeredi/peldak/tree.pl...
% consulted /home/szeredi/peldak/tree.pl in module user, 0 msec 704 bytes
yes
| ?- sum_tree(node(leaf(5),
                   node(leaf(3), leaf(2))), Sum).
Sum = 10 ? ;
no
| ?- sum_tree(Tree, 10).
Tree = leaf(10) ? ;
! Instantiation error in argument 2 of is/2
! goal:  10 is _73+_74
| ?- halt.
%
```

- A hiba oka: a beépített aritmetika egyirányú: a `10 is S1+S2` hívás hibát jelez!

Peano aritmetika — összeadás

- A természetes számok halmazán az összeadást definiálhatjuk a Peano axiómákkal ha a számokat az `s(X)` „rákövetkező” függvény segítségével ábrázoljuk:
  `1 = s(0), 2 = s(s(0)), 3 = s(s(s(0))), ...` (Peano ábrázolás).

```
% plus(X, Y, Z): X és Y összege Z (X, Y, Z Peano ábrázolású).
plus(0, X, X).                      % 0+X = X.
plus(s(X), Y, s(Z)) :-
        plus(X, Y, Z).              % s(X)+Y = s(X+Y).
```

- A `plus` predikátum több irányban is használható:

```
| ?- plus(s(0), s(s(0)), Z).        Z = s(s(s(0))) ? ; no     %  1+2 = 3

| ?- plus(s(0), Y, s(s(0))).        Y = s(s(0)) ? ; no        %  3-1 = 2

| ?- plus(X, Y, s(s(0))).           X = 0, Y = s(s(0)) ? ;    %  2 = 0+2
                                    X = s(0), Y = s(0) ? ;    %  2 = 1+1
                                    X = s(s(0)), Y = 0 ? ;    %  2 = 2+0
                                    no
| ?-
```

Adott összegű fák építése

- Adott összegű fát építő eljárás Peano aritmetikával:

```
sum_tree(leaf(Value), Value).
sum_tree(node(Left, Right), S) :-
        plus(S1, S2, S),
        S1 \= 0, S2 \= 0,     % X \= Y beépített eljárás, jelentése:
                              % X  és Y nem egyesíthető
                              % A 0-t kizárjuk, mert különben ∞ sok megoldás van.
        sum_tree(Left, S1),
        sum_tree(Right, S2).
```

- Az eljárás futása:

```
| ?- sum_tree(Tree, s(s(s(0)))).
Tree = leaf(s(s(s(0)))) ? ;                                  % 3
Tree = node(leaf(s(0)),leaf(s(s(0)))) ? ;                    % (1+2)
Tree = node(leaf(s(0)),node(leaf(s(0)),leaf(s(0)))) ? ;      % (1+(1+1))
Tree = node(leaf(s(s(0))),leaf(s(0))) ? ;                    % (2+1)
Tree = node(node(leaf(s(0)),leaf(s(0))),leaf(s(0))) ? ;      % ((1+1)+1)
no
```

A Prolog adatfogalma, a Prolog kifejezés

- konstans (*atomic*)
  - számkonstans (*number*) — egész vagy lebegőpontos, pl. `1, -2.3, 3.0e10`
  - névkonstans (*atom*), pl. `'István', ispow, +, -, <, sum_tree`
- összetett- vagy struktúra-kifejezés (*compound*)
  - ún. kanonikus alak: $\langle$ struktúranév $\rangle$`(`$\langle$ arg_1 $\rangle$`, ...)`
    - a $\langle$ struktúranév $\rangle$ egy névkonstans, az $\langle$ arg_i $\rangle$ argumentumok tetszőleges Prolog kifejezések
    - példák: `leaf(1), person(william,smith,2003,1,22), <(X,Y), is(X, +(Y,1))`
  - szintaktikus „édesítőszerek”, pl. operátorok: `X is Y+1` $\equiv$ `is(X, +(Y,1))`
- változó (*var*)
  - pl. `X, Szulo, X2, _valt, _, _123`
  - a változó alaphelyzetben behelyettesítetlen, értékkel nem bír, az egyesítés (mintaillesztés) művelete során egy tetszőleges Prolog kifejezést vehet fel értékül (akár egy másik változót)

A PROLOG NYELV KÖZELÍTŐ SZINTAXISA

Predikátumok, klózok

- Példa:

```
% két klózból álló predikátum definíciója, funktora: sum_tree/2
sum_tree(leaf(Val), Val).              %                    1. klóz, tényállítás
sum_tree(node(Left,Right), S) :-       %              fej   \
        sum_tree(Left, S1),            % cél         \      |
        sum_tree(Right, S2),           % cél         | törzs | 2. klóz, szabály
        S is S1+S2.                    % cél         /      /
```

- Szintaxis:

⟨ Prolog program ⟩	::=	⟨ predikátum ⟩ . . .	
⟨ predikátum ⟩	::=	⟨ klóz ⟩ . . .	{azonos funktorú}
⟨ klóz ⟩	::=	⟨ tényállítás ⟩.␣ |	
		⟨ szabály ⟩.␣	{klóz funktora = fej funktora}
⟨ tényállítás ⟩	::=	⟨ fej ⟩	
⟨ szabály ⟩	::=	⟨ fej ⟩ :- ⟨ törzs ⟩	
⟨ törzs ⟩	::=	⟨ cél ⟩, . . .	
⟨ cél ⟩	::=	⟨ kifejezés ⟩	
⟨ fej ⟩	::=	⟨ kifejezés ⟩	

Prolog programok formázása

- Programok javasolt formázása:
  - Az egy predikátumhoz tartozó klózok legyenek egymás mellett a programban, közéjük ne tegyünk üres sort. A predikátumokat válasszuk el üres sorokkal.
  - A klózfejet írjuk sor elejére, minden célt lehetőleg külön sorba, néhány szóközzel beljebb kezdve

Prolog kifejezések

- Példa — egy klózfej mint kifejezés:

```
%     sum_tree(node(Left,Right), S)    % összetett kif., funktora sum_tree/2
%     ________ ________________  _
%        |         |             |
% struktúranév     |          argumentum, változó
%                  \- argumentum, összetett kif.
```

- Szintaxis:

⟨ kifejezés ⟩	::=	⟨ változó ⟩ |	{Nincs funktora}
		⟨ konstans ⟩ |	{Funktora: ⟨ konstans ⟩`/0`}
		⟨ összetett kifejezés ⟩ |	{Funktora: ⟨ struktúranév ⟩`/`⟨ arg.szám ⟩}
		(⟨ kifejezés ⟩ **)**	{Operátorok miatt, ld. később}
⟨ konstans ⟩	::=	⟨ névkonstans ⟩ |	
		⟨ számkonstans ⟩	
⟨ számkonstans ⟩	::=	⟨ egész szám ⟩ |	
		⟨ lebegőpontos szám ⟩	
⟨ összetett kifejezés ⟩	::=	⟨ struktúranév ⟩ (⟨ argumentum ⟩, . . .)	
⟨ struktúranév ⟩	::=	⟨ névkonstans ⟩	
⟨ argumentum ⟩	::=	⟨ kifejezés ⟩	

Lexikai elemek

- Példák:

```
% változó:            Fakt FAKT _fakt X2 _2 _
% névkonstans:        fakt ≡ 'fakt' 'István' [] ; ',' += ** \= ≡ '\\='
% számkonstans:       0 -123 10.0 -12.1e8
% nem névkonstans:    !=, Istvan
% nem számkonstans:   1e8 1.e2
```

- Szintaxis:

⟨ változó ⟩ ::= ⟨ nagybetű ⟩⟨ alfanumerikus jel ⟩... |
_ ⟨ alfanumerikus jel ⟩...

⟨ névkonstans ⟩ ::= ' ⟨ idézett karakter kar ⟩... ' |
⟨ kisbetű ⟩⟨ alfanumerikus jel ⟩... |
⟨ tapadó jel ⟩... | ! | ; | [] | {}

⟨ egész szám ⟩ ::= {előjeles vagy előjeltelen számjegysorozat}

⟨ lebegőpontos szám ⟩ ::= {belsejében tizedespontot tartalmazó számjegysorozat esetleges exponenssel}

⟨ idézett karakter ⟩ ::= {tetszőleges nem ' és nem \ karakter} | \ ⟨ escape szekvencia ⟩

⟨ alfanumerikus jel ⟩ ::= ⟨ kisbetű ⟩ | ⟨ nagybetű ⟩ | ⟨ számjegy ⟩ | _

⟨ tapadó jel ⟩ ::= + | - | * | / | \ | $ | ^ | < | > | = | ` | ~ | : | . | ? | @ | # | &

Szintaktikus édesítőszer: operátorok

- Példa:

```
% S is -S1+S2 ekvivalens az is(S, +(-(S1),S2)) kifejezéssel
```

- Operátoros kifejezések

⟨ összetett kifejezés ⟩ ::=

	⟨ struktúranév ⟩ (⟨ argumentum ⟩, . . .)	{eddig csak ez volt}
|	⟨ argumentum ⟩ ⟨ operátornév ⟩ ⟨ argumentum ⟩	{infix kifejezés}
|	⟨ operátornév ⟩ ⟨ argumentum ⟩	{prefix kifejezés}
|	⟨ argumentum ⟩ ⟨ operátornév ⟩	{posztfix kifejezés}

⟨ operátornév ⟩ ::= ⟨ struktúranév ⟩ {ha operátorként lett definiálva}

- Operátor-kezelő beépített predikátumok:
  - `op(Prioritás, Fajta, OpNév)` vagy `op(Prioritás, Fajta, [OpNév`$_1$`,OpNév`$_2$`,...])`:
    - `Prioritás`: 0–1200 közötti egész
    - `Fajta`: az `yfx`, `xfy`, `xfx`, `fy`, `fx`, `yf`, `xf` névkonstansok egyike
    - `OpNév`: tetszőleges névkonstans
    - pozitív prioritás esetén definiálja az operátor(oka)t, 0 prioritás esetén megszünteti azokat.
  - `current_op(Prioritás, Fajta, OpNév)`: felsorolja a definiált operátorokat.

Szabványos, beépített operátorok

Szabványos operátorok

```
1200  xfx  :- -->
1200   fx  :- ?-
1100  xfy  ;
1050  xfy  ->
1000  xfy  ’,’
 900   fy  \+
 700  xfx  < = \= =..
           =:= =< == \==
           =\= > >= is
           @< @=< @> @>=
 500  yfx  + - /\ \/
 400  yfx  * / // rem
           mod << >>
 200  xfx  **
 200  xfy  ^
 200   fy  - \
```

Egyéb beépített operátorok SICStus Prologban

```
1150   fx  dynamic multifile
           block meta_predicate
 900   fy  spy nospy
 550  xfy  :
 500  yfx  #
 500   fx  +
```

Operátorok jellemzői

- Egy operátort jellemez a fajtája és prioritása
- A fajta meghatározza az operátor-osztályt (írásmódot) és az asszociatívitást:

Fajta			Osztály	Értelmezés
bal-asszoc.	jobb-asszoc.	nem-asszoc.		
`yfx`	`xfy`	`xfx`	infix	`X f Y` ≡ `f(X, Y)`
	`fy`	`fx`	prefix	`f X` ≡ `f(X)`
`yf`		`xf`	posztfix	`X f` ≡ `f(X)`

- Több-operátoros kifejezésben a zárójelezést a prioritás és az asszociatívitás határozza meg, pl.
  - `a/b+c*d` ≡ `(a/b)+(c*d)` mert `/` és `*` prioritása 400, ami **kisebb** mint a `+` prioritása (500) (kisebb prioritás = **erősebb** kötés).
  - `a+b+c` ≡ `(a+b)+c` mert a `+` operátor fajtája `yfx`, azaz bal-asszociatív — balra köt, balról jobbra zárójelez (a fajtanévben az `y` betű mutatja az asszociatívitás irányát)
  - `a^b^c` ≡ `a^(b^c)` mert a `^` operátor fajtája `xfy`, azaz jobb-asszociatív (jobbra köt, jobbról balra zárójelez)
  - `a=b=c` szintaktikusan hibás, mert az `=` operátor fajtája `xfx`, azaz nem-asszociatív