IB015 Neimperativní programování

Seznamy, Aritmetika, Tail rekurze v Prologu

Jiří Barnat

Seznamy a unifikace

Seznamy

Seznam v Prologu

- Konečná sekvence prvků.
- Různorodé (typově nestejné) prvky.
- Délkou seznamu se chápe počet prvků nejvyšší úrovně.

Zápis

- Hranaté závorky, prvky oddělené čárkou.

```
[marek, 2, matous, lukas(jan)]
```

- Prázdný seznam:

```
[]
```

- Zanořený seznam:

```
[ [], a, [c,[h]], [[[jo]]] ]
```

Neprázdné seznamy

Struktura

- Neprázdný seznam se dekomponuje na hlavu seznamu (head) a tělo seznamu (tail).
- Prázdný seznam nemá interní strukturu.

Operátor |

- Pro dekompozici seznamů na hlavu a tělo.

Příklad

- ```
  ?- [H|T] = [marek,matous,lukas,jan].
  H = marek,
  T = [matous, lukas, jan].
  ```
- ```
  ?- [X|Y] = [].
  false.
  ```

Neprázdné seznamy – rozšířená hlava

Hlava jako prefix seznamu

- Hlavu lze zobecnit na neprázdnou konečnou sekvenci prvků.
- Prvky v hlavě seznamu jsou oddělovány čárkou.
- V jednom nezanořeném seznamu je smysluplné pouze jedno použití operátoru `|`.

Použití

- Správné použití

```
?- [X,Y|Z] = [1,2,3,4].
X = 1,
Y = 2,
Z = [3, 4].
```

- Nesprávné použití

```
?- [X|Y|Z] = [1,2,3,4].
```

ERROR

Anonymní proměnná

Anonymní proměnná

- Označená znakem podtržítko.
- Nelze se na ni odkázat v jiném místě programu.
- Při použití v unifikačním algoritmu neklade žádné omezení na kompatibilitu přiřazení hodnot jednotlivým proměnným.

Příklady unifikace s anonymní proměnnou

- `?- f(a,X)=f(X,b).`
  `false.`

  `?- f(a,X)=f(_,b).`
  `X = b.`

  `?- f(a,_)=f(_,b).`
  `true.`

- Unifikací získejte 2. a 4. prvek seznamu `Seznam`:
  `[_,X,_,Y|_]= Seznam.`

Příklady unifikace těla seznamu.

- ```
  ?- [X,Y] = [1,2,3,4].
  false.
  ```
- ```
  ?- [X,Y|[Z]] = [1,2,3,4].
  false.
  ```
- ```
  ?- [_,_|[Z]] = [1,2,3].
  Z = 3.
  ```
- ```
  ?- [1,2|[Z,Y]] = [1,2,3,4].
  Z = 3,
  Y = 4.
  ```
- ```
  ?- [1,2|[Z,Y]] = [1,2,3,4,5].
  false.
  ```
- ```
  ?- [1,2|[Z|Y]] = [1,2,3,4,5].
  Z = 3,
  Y = [4, 5].
  ```

Od funkcí k relacím

Pozorování

- Při vytváření programu v prologu, který má něco spočítat nebo vytvořit, postupujeme dle obecného pravidla transformace funkce `f(A) = B` na predikát `r(A,B)`.

Příklad

- Vytvořte seznam, který začíná dvěma uvedenými prvky, a to v libovolném pořadí.
- Imperativní pohled, funkce vracející požadované výsledky (samostatnou otázkou je, jak reprezentovat více výsledků)

  `fStartsWith(X,Y)` $\rightsquigarrow^*$ `[X,Y|_]`

  $\rightsquigarrow^*$ `[Y,X|_]`

- V Prologu kódujeme jako relaci s o jedna větší aritou:

```
rStartsWith(X,Y,[X,Y|_]).
rStartsWith(X,Y,[Y,X|_]).
```

Příklad – rekurze na seznamech

Zadání

- Definujte predikát `a2b/2`, který transformuje seznam termů `a` na stejně dlouhý seznam termů `b`.

Řešení

- ```
  a2b([],[]).
  a2b([a|Ta],[b|Tb]) :- a2b(Ta,Tb).
  ```

Použití

- ```
  ?- a2b([a,a,a,a],X).
  X = [b,b,b,b].
  ```

  ```
  ?- a2b(X,[b,b,b,b]).
  X = [a,a,a,a].
  ```

- ```
  ?- a2b(X,Y).
  X = Y, Y = [] ;
  X = [a], Y = [b] ;
  X = [a, a], Y = [b, b] ;
  X = [a, a, a], Y = [b, b, b].
  ```
  ```

Operace nad seznamy

Délka seznamu a test na bytí seznamem

length/2

- `length(L,I)` je pravda pokud délka seznamu `L` má hodnotu `I`.
- Prázdný seznam má délku 0.
- `?- length([a,ab,abc],X).`
  `X = 3.`
- `?- length([[1,2,3]],X).`
  `X = 1.`
- `?- length(X,2).`
  `X = [_G907, _G910].`

Test na bytí seznamem

- `is_list/1` – Pravda, pokud parametr je seznam.
- `?- is_list( ['aha',[2,'b'],[],2.3] ).`
  `true.`

Operátor ., Otevřené seznamy

Operátor .

- Předřazení prvku k seznamu (aka : z Haskellu).
- Nemá infixový zápis.
- Příklady

```
?- X = .(b,[1,2,3]).
X = [b, 1, 2, 3].
```

```
?- X = .([a],[a]).
X = [[a], a].
```

Seznamy jako neúplná datová struktura

- Prolog umí pracovat i se seznamy, které nejsou kompletně definovány, tzv. **otevřené seznamy**.
- ```
  ?- Seznam = [1|Telo], Telo = [2|X].
  Seznam = [1,2|X],
  Telo = [2|X].
  ```
- ```
  ?- length([a,b|_],6).
  true.
  ```

Prvky seznamu – predikát `member/2`

member/2

- Zjištění přítomnosti prvku v seznamu.
- `member(X,L)` je pravda, pokud objekt X je prvkem seznamu L.
- Implementace:

```
member(X,[X|_]).
member(X,[_|T]) :- member(X,T).
```

Postup výpočtu:

- `?- member(lukas,[marek,matous,lukas,jan]).`

  $\rightsquigarrow$

  `?- member(lukas,[matous,lukas,jan]).`

  $\rightsquigarrow$

  `?- member(lukas,[lukas,jan]).`

  `true.`

Predikát member/2 – použití

Pozorování

- Volnou proměnnou lze použít jak v prvním, tak i v druhém argumentu. Pomocí predikátu je možné enumerovat prvky seznamu, ale také vynutit, že daný prvek je prvkem seznamu.

Příklady

- ```
  ?- member(X,[marek,matous,lukas,jan]).
  X = marek ;
  X = matous ;
  X = lukas ;
  X = jan ;
  false.
  ```
- ```
  ?- member(jan,[marek,matous,lukas,X]).
  X = jan ;
  false.
  ```

Konkatenace seznamů predikátem append

append/3

- Dotaz `append(L1,L2,L3)` se vyhodnotí na pravda, pokud seznam `L3` je zřetězením seznamů `L1` a `L2`.

Definice append

- ```
  append([],L,L).
  append([H1|T1],L2,[H1|T3]) :- append(T1,L2,T3).
  ```

Použití

- Test na to, zda $\texttt{L1} \cdot \texttt{L2} = \texttt{L3}$.
- Test na rovnost seznamů.
- Výpočet zřetězení dvou seznamů.
- Odvození prefixu, nebo sufixu v daném seznamu.
- Generování seznamů, jejichž zřetězení má daný výsledek.
  ```

Syntax se mění, myšlenka zůstává

Definice append v Prologu

- ```
  append([],L,L).
  append([H1|T1],L2,[H1|T3]) :- append(T1,L2,T3).
  ```

Definice append v Haskellu

- ```
  append [] l = l
  append (h1:t1) l2 = h1:t3 where t3 = (append t1 l2)
  ```

Predikáty nth

nth0(Index, List, Elem)

- `Index` prvku `Elem` v seznamu `List`, počítáno od 0, tj. první prvek má index 0.

nth1(Index, List, Elem)

- Totéž co `nth0/3`, ale počítáno od 1.

nth0(N, List, Elem, Rest)

- `Index` prvku `Elem` v seznamu `List`, počítáno od 0, tj. první prvek má index 0, `Rest` je seznam vzniklý ze seznamu `List` odebráním dotčeného prvku.

nth1(N, List, Elem, Rest)

- Totéž co `nth0/4`, ale počítáno od 1.

Cyklické seznamy

Pozorování

- Nedokonalost unifikačního algoritmu lze "zneužít" pro definici cyklických seznamů, tj. seznamů, které jsou periodicky nekonečné.

Příklady

- ```
  ?- unify_with_occurs_check(X,[1,2,3|X]).
  false.
  ```
- ```
  ?- X=[1,2,3|X], nth1(7,X,Z).
  X = [1, 2, 3|X],
  Z = 1.
  ```
- ```
  ?- X=[a,b|X], append([a,b,a,b,a],Z,X).
  X = [a,b|X],
  Z = [b|X].
  ```
  ```

Aritmetika

SWI Prolog - Aritmetické typy

Celá čísla - integer

- Nativní typ, využívá knihovnu GMP.
- Velikost čísel omezena pouze velikostí dostupné paměti.

Desetinná čísla - float

- Nativní typ, odpovídá typu `double` z programovacího jazyka C.

Racionální čísla - rational

- Reprezentované s využitím složeného termu `rdiv(N,M)`.
- Výsledek vrácený operátorem `is/2` je kanonizován, tj. `M` je kladné a `M` a `N` nemají společného dělitele.

Konverze a unifikace

- `rdiv(N,1)` se konvertuje na celé číslo `N`.
- Automatické konverze ve směru od celých čísel k desetinným.
- Riziko vyvolání výjimky přetečení.
- Čísla různých typů se **neunifikují.**

Aritmetické funkce a operátory

Relační operátory

`</2`	menší než
`>/2`	větší než
`=</2`	menší nebo rovno
`>=/2`	větší nebo rovno
`=:=/2`	rovno
`=\=/2`	nerovno

Bitové operace

`<</2`	bitový posun vlevo
`>>/2`	bitový posun vpravo
`\//2`	bitové OR
`/\/2`	bitové AND
`\/1`	bitová negace
`xor/2`	bitový XOR

Vybrané aritmetické funkce

`-/1`	unární mínus
`+/1`	znaménko plus
`+/2`	součet
`-/2`	rozdíl
`*/2`	součin
`//2`	dělení
`**/2`	mocnina
`///2`	celočíselné dělení
`rem/2`	zbytek po dělení //
`div/2`	dělení a zaokrouhlení dolů
`mod/2`	zbytek po dělení div
`max/2`	maximum
`min/2`	minimum
`is/2`	vyhodnocení a unifikace

Aritmetické operace v relacích

Pozorování

- Pro strukturovaný term, který dává do relace dva jiné termy, je možné nechat Prolog dohledat termy, pro které relace platí.
- ```
  rel(a,b).
  ?- rel(X,b).
  X = a.
  ```

Neplatí pro argumenty aritmetických operací

- V případě použití aritmetické operace takovéto chování vyžaduje inverzní aritmetickou funkci.
- Prolog při unifikaci a rezoluci nepočítá inverzní funkce, v okamžiku požadavku na takovouto operaci ohlásí interpret chybu (nedostatečná instanciace).
- Porovnejte:

```
?- X is 3*3.          ?- 9 is 3*X.
X = 9.                ERROR
```
```

Příklady nedostatečné instanciace

Vyzkoušejte

- `?- 9 is X + 1.`
  ERROR
- `?- X > 3.`
  ERROR
- `?- X = 2, X > 3.`
  `false.`

Vysvětlete rozdílné chování

- Korektní definice predikátu pro výpočet délky seznamu.

```
length([],0).
length([_|T],N) :- length(T,X), N is X+1.
```

- Nevhodná definice predikátu pro výpočet délky seznamu.

```
length1([],0).
length1([_|T],N) :- N is X+1, length1(T,X).
```

- Rozdílné chování při výpočtu.

```
?- length([a,b],X).
X = 2.
```

```
?- length1([a,b],X).
```
ERROR

Instanciace parametrů

Pozorování

- Předdefinované predikáty mohou vyžadovat, aby některé parametry byly povinně instanciované, tzn. na jejich místě nelze použít proměnnou.

Používaná notace v dokumentaci

- +Arg: musí být instanciovaný parametr.
- -Arg: očekává se proměnná.
- ?Arg: instanciovaný parametr nebo proměnná.
- @Arg: parametr nebude vázán unifikací.
- :Arg: parametrem je název predikátu.

Módy použití

- Je-li binární predikát použit s dvěma instaciovanými parametry, říkáme, že predikát je použit v (+,+) módu.

Příklady z dokumentace

between(+Low, +High, ?Value)

- `Low and High are integers, High >=Low. If Value is an integer, Low=< Value=< High. When Value is a variable it is successively bound to all integers between Low and High. If High is inf or infinite between/3 is true iff Value>= Low, a feature that is particularly interesting for generating integers from a certain value.`

plus(?Int1, ?Int2, ?Int3)

- `True if Int3 =Int1 +Int2.` **At least two of the three arguments must be instantiated to integers.**

sort(+List, -Sorted)

- `True if Sorted can be unified with a list holding the elements of List, sorted to the standard order of terms. Duplicates are removed. The implementation is in C, using natural merge sort.`

  `The sort/2 predicate can sort a cyclic list, returning a non-cyclic version with the same elements.`

Tail rekurze

Rekurze a efektivita

Pozorování

- Uvažme následující definici predikátu `length`.

```
length([],0).
length([_|T],N) :- length(T,X), N is X+1.
```

- Nevýhodou této definice je, že při volání dochází k výpočtu před rekurzivním voláním (při rekurzivním sestupu) i po rekurzivním volání (při vynořování z rekurze).

Tail rekurze

- Definice, jež nevynucuje výpočet po rekurzivním volání, tj. rekurzivní cíl je jeden a je uveden jako poslední podcíl.
- Výsledek je znám při dosažení dna rekurzivního sestupu.
- Menší režie výpočtu, větší efektivita.
- Platí i ve světě imperativních programovacích jazyků.

Tail rekurze a akumulátory

Pozorování

- Tail-rekurzivní definice lze dosáhnout použitím akumulátoru.
- Akumulátor = pomocný shromažďovací parametr.
- Zavedení akumulátoru vyžaduje definici pomocného predikátu.

Predikát `length` definován tail-rekurzivně

- Realizace pomocným predikátem s akumulátorem.

```
length(Seznam,Delka) :- accLen(Seznam,0,Delka).
```

- Definice pomocného predikátu.

```
accLen([],A,A).
accLen([_|T],A,L) :- B is A+1, accLen(T,B,L).
```

- Mód použití `accLen` je `(?,+,?)`.

Iniciální hodnota akumulátorového parametru

Pozorování

- V některých případech je obtížné zvolit iniciální hodnotu akumulačního parametru.
- Uvažme následující definici pomocného predikátu `accMax/3`, který s využitím akumulátoru pro volání `accMax(List,A,Max)` počítá největší číslo v seznamu:

```
accMax([],A,A).
accMax([H|T],A,Max) :- H > A, accMax(T,H,Max).
accMax([H|T],A,Max) :- H =< A, accMax(T,A,Max).
```

Úkol

- S využitím `accMax/3` definujte predikát `max_member/2`.
- Jakou zvolit výchozí hodnotu akumulátoru?

Iniciální hodnota akumulátorového parametru

Pozorování

- V některých případech je obtížné zvolit iniciální hodnotu akumulačního parametru.
- Uvažme následující definici pomocného predikátu `accMax/3`, který s využitím akumulátoru pro volání `accMax(List,A,Max)` počítá největší číslo v seznamu:

```
accMax([],A,A).
accMax([H|T],A,Max) :- H > A, accMax(T,H,Max).
accMax([H|T],A,Max) :- H =< A, accMax(T,A,Max).
```

Úkol

- S využitím `accMax/3` definujte predikát `max_member/2`.
- Jakou zvolit výchozí hodnotu akumulátoru?

```
max_member(List,Max) :- List = [H|_], accMax(List,H,Max).
```

Řetězce, operátor syntaktické ekvivalence

Řetězce v Prologu

Pozorování

- `'Ahoj'` není totéž co `"Ahoj"` .
- Řetězce znaků uvedených v uvozovkách jsou chápány jako seznamy čísel, které odpovídají ASCII kódům jednotlivých znaků řetězce.
- `"Ahoj" == [65,104,111,106].`

Automatická konverze na seznam čísel

- ```
  ?- append("Ale ","ne!",X).
  X = [65, 108, 101, 32, 110, 101, 33].
  ```

Konverze na řetězce

- Vybrané předdefinované predikáty vynutí prezentaci ve formě "řetězce".
  ```

Syntaktická ekvivalence (identita) - operátor ==

Syntaktická rovnost

- ```
  ?- 'pepa' == "Pepa".
  false.
  ```

  ```
  ?- 'mouka' == 'mouka'.
  true.
  ```

  ```
  ?- 4 == 3+1.
  false.
  ```

  ```
  ?- 3+1 == 3+1.
  true.
  ```

Pozor na syntaktické ekvivalenty

- ```
  ?- 'pepa' == pepa.
  true.
  ```

  ```
  ?- [97] == "a".
  true.
  ```

Konverze řetězců a jejich délka

string_to_atom(?String, ?Atom)

- Logical conversion between a string and an atom. At least one of the two arguments must be instantiated. Atom can also be an integer or floating point number.

string_to_list(?String, ?List)

- Logical conversion between a string and a list of character code characters. At least one of the two arguments must be instantiated.

string_length(+String, -Length)

- Unify Length with the number of characters in String. This predicate is functionally equivalent to atom_length/2 and also accepts atoms, integers and floats as its first argument.

Zřetězení a detekce podřetězců

string_concat(?String1, ?String2, ?String3)

- Similar to atom_concat/3, but the unbound argument will be unified with a string object rather than an atom. Also, if both String1 and String2 are unbound and String3 is bound to text, it breaks String3, unifying the start with String1 and the end with String2 as append does with lists.

sub_string(+String, ?Start, ?Length, ?After, ?Sub)

- Sub is a substring of String starting at Start, with length Length, and String has After characters left after the match. See also sub_atom/5.

Příklady

- ```
  ?- sub_string('Nenene!',X,Y,Z,'ne').
  X = Y, Y = 2, Z = 3 ;
  X = 4, Y = 2, Z = 1 ;
  false.
  ```
- ```
  ?- sub_string('Nenene!',4,2,1,X).
  X = "ne".
  ```

Příklady

Příklad: násobky daného čísla

Zadání

- Definujte predikát `nasobky(+Cislo,+Pocet,?Seznam)`, který je pravdivý, pokud `Seznam` je prvních `Pocet` násobků čísla `Cislo`.
- `?- nasobky(3,6,[3,6,9,12,15,18]).`
  `true.`

Řešení

- ```
  nasobky(N,P,L) :- aNas(N,P,[],L).
  aNas(_,0,Acc,Acc).
  aNas(N,P,A,L) :- P>0,                    /* prevent infinite recursion */
                   P1 is P-1, K is N*P,
                   aNas(N,P1,[K|A],L).
  ```
  ```

Příklad: packList

Zadání

- V prologu naprogramujte predikát `packList/2`, který realizuje vynechání sousedních identických termů v seznamu, predikát předpokládá mód použití `(+,?)`.
- ```
  ?- packList([a,a,a,1,1,c,c,c,[a],[a]],X).
  X = [a,1,c,[a]].
  ```

Řešení

- ```
  packList(L,P) :- aPL(L,RP,L,[]), reverse(RP,P).
  aPL([],P,_,P).
  aPL([H|T],P,L,Acc) :- H \== L, aPL(T,P,H,[H|Acc]);
                        H == L, aPL(T,P,L,Acc).
  ```

Domácí úloha

Zadání

- V Prologu naprogramujte predikáty `encodeRLE/2` a `decodeRLE/2`, které budou realizovat RLE kódování seznamů.
- V RLE kódování je každá `n`-tice stejných po sobě jdoucích prvků `k` v seznamu nahrazena uspořádanou dvojicí `(n,k)`.

Příklad použití

- ```
  ?- encodeRLE([a,a,a,b,b,c,d,d,e],X).
  X = [(3,a),(2,b),(1,c),(2,d),(1,e)]
  ```
- ```
  ?- decodeRLE([(5,1),(1,2),(3,3)],[1,1,1,1,1,2,3,3,3]).
  true.
  ```