Obsah

1 Úvodní poznámky

1.1 Slovo autora

Zde jsou obsaženy přednášky dr. Matysky „Logické programování“ ze zimního semestru 3. ročníku (říjen 1992 – leden 1993). Přednášky opatřil Dejva (David Krásenský, kapitoly 1 až 9), zčásti od Hanči Tesařové (kapitoly 10 až 12). TEX-oval a komentáři opatřil Dejva.

1.2 Historie Prologu

Základy logického programování položil Robinson tím, že formuloval unifikaci. V roce 1973 prof. Colmerauer v Marseilles dostal nápad vytvořit logický programovací jazyk. Vznikl tak Prolog (PROGramation à LOGic), jehož první interpreter byl napsán ve Fortranu. Původní myšlenka byla vlastně analýza přirozeného jazyka (francouzštiny).

Současně na podobné problematice pracoval v Edinburghu prof. Kowalski.

V roce 1978 vytvořil v Edinburghu D. H. D. Warren kompilátor Prologu pro DEC–10, který byl na svou dobu velmi efektivní; od stejného autora pochází model WAM – Warren Abstract Machine[1] z roku 1983.

V polovině osmdesátých let, kdy Japonci ohlašují brzký nástup 5. generace počítačů, na kterých chtějí pro jádra operačních systémů používat jazyky logického programování, nabývá Prolog na významu. Začínají se o něj zajímat Američané, kteří připravují v současné době ANSI standard.

Další vývoj Prologu směřuje zejména k paralelním systémům (viz část 12) a k systémům s omezujícími podmínkami (viz část 10).

2 Prolog od základů

2.1 Základní pojmy

Prolog vychází z reprezentace formulí pomocí Hornových klausulí. Zápis v Prologu je poněkud odlišný a shrnuje jej tabulka 1.

Tabulka 1: Hornovy klausule v Prologu

Typ klausule	Množinový zápis	Prologovský zápis
Pravidlo	$\{ A, \neg B, \neg C, \neg D \}$	A :– B,C,D.
Fakt	$\{ A \}$	A.
Dotaz (cíl)	$\{ \neg B, \neg C, \neg D \}$	?–B,C,D.

Nyní si uvedeme základní prvky jakyku Prolog a jak je zapisujeme:

Predikáty píšeme zpravidla malými písmeny. Musí začínat malým písmenem, dále jsou dovolena písmena, číslice a podtrhy[2].

Atom je nulární predikát, tedy predikát bez argumentů.

Proměnné musí začínat velkým písmenem, popř. podtrhem.

Predikát má tvar `atom(`$\mathtt{arg}_1$`,...,`$\mathtt{arg}_n$`)`.

Funkční symboly jsou syntakticky totéž, co predikáty, avšak mají odlišné použití: používáme je na místě argumentů predikátů a funkčních symbolů.

Program v Prologu je pak konečná množina Hornových klausulí, a to klausulí programových (tedy faktů a pravidel).

Jako příklad si uvedeme následující jednoduchý program:

```
otec(jan,pavel).
otec(pavel,jana).
děda(X,Y):-otec(X,Z),otec(Z,Y).
```

Fakta v databázi přitom interpretujeme tak, že Jan je otcem Pavla a Pavel je otcem Jany.
Na dotazy, které napíšeme na prompt „?“ (otazník) můžeme dostat tyto odpovědi:

[1] neboli Warrenova abstraktní mašina
[2] V originále „podtrženítka“.

Dotaz	*Odpověď*
`?-otec(jan,pavel)`	yes (unifikace uspěje)
`?-otec(jan,jana)`	no (neuspěje)
`?-děda(jan,jana)`	yes
`?-otec(X,Y)`	X=jan, Y=pavel; odpovíme-li dále „;“ (středník), Prolog se pokusí uspět znovu a vydá odpověď: X=pavel, Y=jana; odpovíme-li znovu „;“, Prolog již neuspěje a odpoví *no*

2.2 Peanova aritmetika

Zavedeme si přirozená čísla jako následníka nuly, tedy *0,s(0),s(s(0)),* atd. Na takto definovaných číslech pak můžeme zavést operaci sčítání následujícím způsobem:

```
add(0,Y,Y).
add(s(X),Y,s(Z)):-add(X,Y,Z).
```

Zajisté nás nepřekvapí toto „sčítání“, jako spíše to, že takto definovanou proceduru[3] `add` můžeme stejně dobře použít pro odčítání: položíme-li dotaz `?-add(X,s(0),s(s(s(0))))`, tedy vlastně rovnici $x+1=3$, dá Prolog správnou odpověď `X=s(s(0))`.

Odtud plyne následující poučení: v Prologu neexistuje rozlišení vstupních a výstupních parametrů procedur, záleží na nás, za které dosadíme hodnotu (jakoby vstupní parametr) a za které volnou proměnnou (výstupní).

Ještě můžeme na tomto jednoduchém příkladu ilustrovat tuto obecnou zásadu programování v Prologu: vždy je vhodné definovat nějaké rekurzívní pravidlo (pro dekompozici problému) a okrajové podmínky[4].

2.3 Predikát řezu

Řez (cut, !) je jedním z „nelogických“ rysů Prologu. Nejedná se tedy již o „čistý“ Prolog.

Predikát řezu funguje tím způsobem, že pokud Prolog narazí při splňování jednotlivých cílů na řez, „odřízne“ všechny varianty výpočtu, takže se při opětovném výpočtu (pokud některý z predikátů za řezem neuspěje, anebo při pokusu o novou instanciaci proměnných) již nevrací před řez.

Uvažujme např. následující proceduru:

```
a:-b,c,!,d,e.
a:-f,g.
a:-h.
```

Jestliže položíme dotaz `?-a`, a predikáty `b,c` jsou splněny, nebudou se díky řezu `b,c` prohledávat znovu, ani když `d,e` neuspěje, a ani se neuplatní druhé a třetí pravidlo.

I když se doporučuje nepoužívat řezy pokud možno vůbec, může mít smysl použít řezů i několik. Tak v proceduře :-

```
a(a,X):-!,c(X),d(X),!.
a(b,X):- ...
a(c,X):- ...
```

používám první řez k tomu, aby se Prolog snažil splnit dotaz `a(a,_)`[5] pouze jednou, a druhým řezem dávám najevo, že mne nezajímají ani další alternativy pod `c(X),d(X)`.

2.4 Predikát `fail`, negace

Pro vyjádření neúspěchu má Prolog definován speciální predikát `fail`.

Funguje zde přitom „negace jako neúspěch“, to znamená, že `not(A)`[6] selže, právě když `A` uspěje.

[3] Procedurou v Prologu nazýváme jednotku, kterou tvoří všechny predikáty se stejnou hlavou, tedy se stejným názvem a aritou (počtem argumentů).

[4] Na tomto místě si všimněme, že nebylo třeba definovat `add(X,0,X)`, okrajová podmínka `add(0,Y,Y)` je dostačující.

[5] Samotné podtržení označuje tzv. anonymní proměnnou.

[6] Většinou se dá psát bez závorek – `not A`.

Na tomto místě je třeba upozornit na jednu záludnost, a sice že v Prologu obecně neplatí poučka známá z matematické logiky, že $\neg\neg A \equiv A$. S tímto „paradoxem“ se setkáme, pokud A není uzavřený term. Ukážeme si to na příkladu:

```
a(1).
a(2).
```

Tento „program“ pak dá následující výsledky:

Dotaz	*Odpověď*
`?-a(X)`	X=1; X=2
`?-a(3)`	no
`?-not(not(a(3)))`	no (což je „v pořádku“)
`?-not(a(X))`	no (poněvadž `a(X)` uspěje)
`?-not(not(a(X)))`	yes (ale nedostaneme žádnou hodnotu X, neboť po neúspěchu je proměnná volná)
`?-a(X),X=3`	no
`?-not(not(a(X))),X=3`	yes !

Negace (stejně jako většina ostatních vestavěných predikátů Prologu) (predikát `not`) se přitom dá napsat v Prologu, a to takto:

```
not(X):-call(X),!,fail.
not(X).
```

Druhý predikát se zde uplatní jen v případě, že první (a tedy volání `call(X)`) neuspěje.

Posloupnost `!,fail` je přitom „dost běžná“ v tom smyslu, že se dá dost dobře použít v případech, kdy na základě nějakých podmínek snadno vydedukujeme neúspěch.

2.5 Operátory

Následující poznámka se týká operátorů v Prologu, jejich možných zápisů, priority a asociativity.

U operátorů známe tři druhy notací:

prefixová – operátor je zapsán před svými argumenty. Typicky u „obvyklých“ funkcí (píšeme $f(x, y, z)$, jak jsme zvyklí), lze ale také psát $+(1, 2)$.

infixová – operátor mezi argumenty, tak jak to známe z aritmetických zápisů typu $14 + 4$.

postfixová – spíše na doplnění; postfixovou notaci, jak známo, používaly některé kalkulačky, bylo tedy nutné psát $14\ 4\ +$.

V Prologu se přitom dají prefixová a infixová notace používat způsobem, který pro nás není obvyklý: je-li funktor (operátor)[7] zapsán jen pomocí nealfanumerických znaků, můžeme infixovou notaci použít bez apostrofů: napíšeme tedy $1 <> 2$, ale $1\,'<\,r\,>'\,2$.

Pro operátory má Prolog vestavěný predikát `op/3`.[8] Používá se :-

```
op(Priorita,Asociativita,Funktor)
```

Priorita leží v intervalu $\langle 1, 1024\rangle$ (zaručeně v $\langle 1, 255\rangle$); čím je toto číslo vyšší, tím vyšší prioritu funktor má.

Asociativita se vyjadřuje posloupností písmen, v níž :-

`x, y` nahrazují argumenty,

`f` vyjadřuje polohu funktoru.

[7] Oba pojmy znamenají totéž; u operátoru nemusíme psát závorky.

[8] Tímto zápisem vyjadřujeme, že predikát op má tři argumenty.

Tak například `fx` vyjadřuje prefixový, a např. `xfx` infixový zápis.

Přitom písmena `x` a `y` vyjadřují asociativitu operátorů se stejnou prioritou:[9] `y` vyjadřuje, že tento argument musí být uzávorkován, tedy např. pro operátor obyčejného odečítání platí :-

```
op(_,yfx,'-').
```

Poznámka. Pozor na rozdíl mezi těmito dvěma zápisy:

```
jméno(X,Y,Z)
jméno␣(X,Y,Z)
```

Zatímco v prvém případě jde o *funktor* se třemi argumenty, v případě druhém se jedná o *operátor* s argumentem jediným.

2.6 Unifikace

Princip unifikace je vlastně základem jazyka Prolog. V Prologu se vyjadřuje rovnítkem, přičemž opět lze psát `=(A,B)` místo obvyklejšího `A=B`. Tato prologovská unifikace ovšem z důvodu snazší implementace neprovádí test výskytů proměnných na levé a pravé straně, a proto např. dotaz `?-f(X)=X` uspěje.

Napíšeme-li `A=B`, je `A, B` dále identické. Provede se totiž korektní unifikace včetně sdílení proměnných.

Pro nerovnost je v Prologu predikát `\=/2`, který pouze provádí test:

```
A\=B :- not(A=B).
```

S unifikací souvisí rovněž operátory `==/2` a `\==/2`, které vyjadřují „slabou“ formu unifikace – test na identitu. Shrňme si proto rozdíly mezi oběma typy unifikace:

`?-A=B`	uspěje, jsou-li A, B *unifikovatelné*
`?-A==B`	uspěje, jsou-li A, B *identické* (tedy selže, jsou-li různé)
`?-A=B,A==B`	uspěje, poněvadž v první fázi se provede unifikace, a ve fázi druhé testuje totožnost (a po unifikaci termy totožné jsou)

3 Vestavěné predikáty

3.1 Vstupy a výstupy

Prolog má operace pro vstup a výstup implementovány pomocí tzv. *proudů* (streams). Základní predikáty pro operace s proudy jsou shrnuty v tabulce 2.

Tabulka 2: Predikáty pro operace vstupu a výstupu

Vstup	*Výstup*	*Komentář*
`see/1`	`tell/1`	Aktivuje proud (standardní je klávesnice, resp. obrazovka). Další predikáty pak pracují s tímto aktivním proudem.
`seen/0`	`told/0`	Nulární predikáty, které uzavřou aktivní proud.
`seeing/1`	`telling/1`	Tyto predikáty svůj argument zunifikují se jménem příslušného aktivního proudu.
`get/1`	`put/1`	Přečte, resp. vypíše jeden znak, zadaný ASCII kódem.
`read/1`	`write/1`	Přečte, resp. vypíše term (viz text).

Některé z predikátů v tabulce, jakož i některé v ní neuvedené, si zaslouží další komentář:

[9] MYSLÍM, ŽE V UMĚLÉ INTELIGENCI JE TO POŘÁDNĚ.

`read/1` přečte kompletní prologovský term včetně tečky na konci a unifikuje jej s proměnnou, která je jeho argumentem.

Pokud tedy zadáme `?-read(I)`, a dále `|a(X,Y,z(1,2)).`[10] (přičemž Prolog načte navíc ještě další znak – mezeru nebo oddělovač řádků), dostaneme `I=a(_1,_2,z(1,2))` (jména proměnných se při čtení „ztratí“).

`write/1` vypíše term (bez tečky na konci). Bohužel nelze obecně zaručit, že tento výstup bude čitelný `read`-em.

`get/1` přečte první tištitelný znak a vrátí ASCII kód.

Zajímavý výsledek dostaneme, pokud parametr instanciujeme: tak např. `?-get(65)` uspěje jen zadáme-li „A“, jinak ohlásí *Instantiation error*.

`skip/1` je třeba spouštět *vždy* s instanciovaným argumentem. Přeskočí vstup až po znak, zadaný argumentem.

`display/1` je výstupní predikát, který použije vždy funktorovou notaci, tedy nikoliv infix.

3.2 Metalogické predikáty a manipulace s programem

3.2.1 Metalogické predikáty

Tato skupina predikátů umí určit typ argumentu:

`var/1` uspěje, je-li její argument volná proměnná

`nonvar/1` uspěje, není-li volná proměnná

`atom/1` testuje, zda je atom (viz str. 3)

`integer/1` určí, jestli je celočíselný literál

`atomic/1` je definován: `atomic(X):-atom(X);integer(X).`[11]

3.2.2 Načítání databáze, predikáty skupiny `consult`

Predikáty této skupiny slouží k načtení dříve připravené databáze. Jedná se o tyto dva predikáty, mezi nimiž je určitý rozdíl:

`consult(jméno)` místo kterého lze rovněž použít zkratku `[jméno]`, načte databázi. Pokud se místo jména souboru zadá `user`, načítá z klávesnice.

Narazí-li na predikát, který již v databázi obsažen je, ponechá se tak, jak je, tedy i vícekrát.

`reconsult(jméno)` se od `consult`-u liší právě v tom, že odstraňuje tyto duplicity. Přesněji řečeno, provádí cosi navíc: pokud načte libovolný funktor s určitou hlavou (tedy s urč. jménem a aritou), smaže staré klausule (starou proceduru) se stejnou hlavou.

3.2.3 Zásahy do databáze – predikáty pro práci s klausulemi

V Prologu se dá měnit program během výpočtu nejen použitím `consult`-u v programu, ale také mnohem důmyslnějšími, rafinovanějšími a mocnějšími prostředky. Jedná se o predikáty `clause, assert, retract` a příbuzné. Nyní si postupně vyložíme jejich funkci:

`clause/2` se volá `clause(H,B)`; hledá v databázi klausuli tvaru `H:-B`. Přitom `H` musí být instanciované tak, aby `clause` bylo schopno najít hlavní funktor a jeho argumenty.

Tak například `?-clause(add(X,Y),B)` bude hledat tělo funktoru `add/2`.

Pokud je nalezená klausule faktem (tedy vlastně nemá tělo), potom se `B` unifikuje s `true`.

`assert/1` uloží do databáze klausuli. Argumentem musí být pochopitelně kompletní klausule včetně znaménka „`:-`“. (Zpravidla vkládá na konec databáze.)

[10] Znak „|“ je nápovědou (promptem) příkazu read.
[11] Středník „;“ značí disjunkci.

`asserta/1` se chová naprosto stejně jako `assert` s tím, že klausuli vkládá vždy na *začátek* databáze.

`assertz/1` podobně vkládá na *konec* databáze.

`retract/1` smaže klausuli z databáze. Je potřeba zadat alespoň částečně instanciovanou klausuli[12], aby `clause` byl schopen v ní rozeznat dvě komponenty – hlavu a tělo. (Tedy nepsat `retract(H)`, ale alespoň něco jako `retract(add(X,Y):-B)` apod.)

`deny/2` má argumenty jako `clause` a účinek jako `retract`.

`retractall/1` smaže celou proceduru, tedy všechny příbuzné klausule. Jako argument se zadává pouze hlava, tedy např. `retractall(add(X,Y))`.

3.3 Složené termy

3.3.1 Seznamy

Poměrně zřejmým příkladem je použití seznamů. K tomu je třeba podotknout, že v Prologu je seznam *dvojicí* sestávající z prvního prvku a „zbytku“ seznamu. Pro konstrukci seznamu pak je vestavěný predikát `./2` (tečka), který se zapisuje ve tvaru `.(Hlava,Ocas)`. Seznam čtyř prvků `a, b, c, d` pak zapíšeme `.(a,.(b,.(c,.(d,nil))))`[13].

Pro snazší zápis seznamů jsou v Prologu hranaté závorky, tedy poslední příklad lze názorněji zapsat `[a,b,c,d]`. Symbol „|“ pak štěpí seznam na hlavu a ocas, tedy `[a,b,c,d]=[a|[b,c,d]]`; pokud je `[H|T]=[a,b,c,d]`, dostaneme `H=a, T=[b,c,d]`. Prázdný seznam pak zcela přirozeně zapisujeme `[]`.

Můžeme jít dále a konstruovat seznamy seznamů, např. `[[a,b,c],[],[a]]` je tříprvkový seznam, jehož prvním prvkem je seznam `[a,b,c]`, druhým prázdný seznam, a třetím jednoprvkový seznam `[a]`.

3.3.2 Funktorové složené termy

Nejsnáze začneme tímto příkladem: máme za úkol implementovat systém pro knihovnu. Zavedeme tedy predikát

```
kniha(Ident,Půjčeno,Vráceno)
```

kde ovšem `Ident` bude složeným termem:

```
Ident='id(ID,autor(_),název(_))
```

Pro tyto složené termy je v Prologu nadefinováno několik vestavěných predikátů:

`=../2` nazývaný též predikát *univ*, rozčlení term na hlavní funktor a argumenty, uspořádané do jednoho seznamu. Tedy

```
a(X,b(c),d)=..[a,X,b(c),d].
```

Pomocí *univ*-u lze pak konstruovat libovolné termy, a to i v programu.

Přitom alespoň jeden z argumentů musí být instanciován (nelze tedy zadat dvě volné proměnné); prvním prvkem seznamu na pravé straně musí být atom. (Přesto však lze psát `a(X,c)=..[T,Q,P]`: pak totiž zcela přirozeně `T=a`.

Následující dva predikáty jsou *univ*-u ekvivalentní:

`arg/3` volání `arg(T,N,A)` nám „řekne“, že n-tý argument termu `T` je `A`; korektně řečeno, `arg` zunifikuje n-tý argument termu `T` s `A`.

`functor/3` volání `functor(T,N,F)` vrátí hlavní funktor `F` termu `T` a jeho aritu `N`. Tak např. `?-functor(T,2,x)` dá `T=x(_,_)`[14].

Poznámka. Predikát `arg` je vhodný pro reprezentaci polí, se kterým jsou jinak v Prologu problémy. Podrobněji o polích pojednává část 5.4.

[12] jinak by zřejmě smazal, co ho napadne

[13] `nil` vyjadřuje prázdný seznam.

[14] Každý podtrh představuje novou anonymní proměnnou.

3.4 Interní databáze

Vedle programové databáze máme k dispozici ještě tzv. *interní* databázi, s níž pracují následující predikáty:

`record/3` uloží položku do databáze. Volá se `record(Key,Arg,Ref)`, kde

`Key` musí být atom (jedná se o analogii arity a funktoru hlavy v programové databázi),

`Arg` je hodnota (term), která se má uložit, a konečně

`Ref` (reference) je speciální objekt umožňující rychlý přístup k hodnotě (něco jako pointer).

`recorded/3` má stejné argumenty jako `record`. Pokud nainstanciuji `Ref`, určí klíč `Key` a argument `Arg`; pokud naopak nainstanciuji klíč, vrátí referenci. Přitom, je-li pod stejným klíčem uloženo více položek, vybírají se postupně a pomocí backtrackingu můžeme dostat všechny hodnoty.

`erase/1` smaže položku určenou referencí (píšeme `erase(Ref)`).

`abolish/1` vyvoláním `abolish(Key)` smažeme všechny položky asiciované daným klíčem. (Tento příkaz není vždy k dispozici.)

3.5 Aritmetika

Zde je nejdůležitějším predikátem `is/2` (přiřazení). Ten se zpravidla píše v podobě `A is expr` a znamená: vyhodnoť výraz `expr` a ulož ho do `A`. (Je to tedy něco zcela jiného než unifikace – o té viz 2.6.)

Můžeme tedy psát např:

```
E=A+B
A=3,B=C*D
C=A,D=7
E=3+3*7
```

Zajisté nás nyní nepřekvapí, že pokud po provedení posledně jmenovaného přiřazení napíšeme `X is E`, objeví se v proměnné `X` hodnota 24.

Ke skupině aritmetických operací patří ještě relační operátory:

`=:=/2` je obyčejný číselný test na rovnost,

`==/2` je test na nerovnost,

`>/2, </2, >=/2, =</2` jsou relace ostré a neostré nerovnosti.

3.6 Nalezení všech řešení

Tyto predikáty se snaží najít všechna možná řešení určitého dotazu a uspořádají je do seznamu. Jedná se o predikáty `findall/3`, `bagof/3` a `setof/3`. Všechny tři mají stejně používané argumenty: `Template`, tedy *vzor*, podle kterého se bude hledat tvar výsledků, `Enumerator`, představující dotaz, jehož řešení hledáme, a `Solutions`, tedy seznam všech řešení. Ukážeme si např. činnost predikátu `findall`:

```
a(1).
a(2).

findall(X,a(X),S)                 S=[1,2]
findall(a(X),a(X),S)              S=[a(1),a(2)]
findall(a(0),a(X),S)              S=[a(0),a(0)]
findall(a(Y),(a(X),Y is X+1),S)   S=[a(2),a(3)]
```

Predikát `findall` je z celé skupiny nejjednodušší, ale zároveň nejméně korektní: nemá např. dáno pořadí vyhledávání a má problémy s volnou proměnnou.

Pro enumerátor rozlišujeme z hlediska logiky tyto proměnné:

- proměnné použité též v `template`,
- proměnné, které si přejeme vázat,
- proměnné, jejichž hodnotu nechceme znát (tedy zřejmě asi zůstanou volné).

K dispozici máme proto zápis `Vars^Goal`;[15] proměnné zde uvedené jsou známy jen pro cíle `Goal`, jinak zachovávají svoje původní hodnoty. Píšeme tedy např. `Z^A^a(X,Z,A)`.

Predikát `bagof` se chová poněkud odlišně:

```
a(1,a).
a(2,b).

bagof(X,a(X,Y),S)       S=[1],Y=a;  S=[2],Y=b
bagof(X,Y^a(X,Y),S)     S=[1,2]
```

Predikát `setof` se chová podobně, avšak hledá pouze *různá* řešení, tedy neobsahuje duplikáty.

Neexistuje-li řešení, `findall` naváže prázdný seznam, zatímco `bagof` a `setof` neuspěje.

Ještě se zmíníme o jednom predikátu, který budem citovat později v 7.3.2, a sice predikát `once`, který provede svůj argument jen jednou:

```
once(G):-call(G),!.
```

Pokud chceme navíc šetřit pamětí a zrušit všechno, co se (zbytečně) během výpočtu nainstanciovalo, píšeme

```
once_go(G):-findall(G,(G,!),[G]).
```

4 Ladění

Jak se dá očekávat, v Prologu je nejen samotný průběh výpočtu, ale tím spíše i ladění, odlišné od „klasických“ programovacích jazyků. Proto se nejprve seznámíme s modelem programu, typickým právě pro Prolog:

4.1 Krabičkový model

(UDĚLEJ OBRÁZEK!!!)

Na každou proceduru v prologovském prologu se můžeme dívat jako na jakousi krabičku, která má čtyři vstupně–výstupní brány:

call vyjadřuje vstup do procedury při normálním vyvolání,

exit je výstup po úspěchu,

re-do slouží pro opětovný vstup poté, co byl někde dále během výpočtu indikován neúspěch (tedy snažíme se najít jiné řešení),

fail je výstup s neúspěchem – predikát končí, jestliže nenalezneme žádné řešení.

Zatímco brány *call* a *exit* jsou normální i pro běžné jazyky, *fail* a *re-do* jsou typické právě pro logické programování.

Jak souvisí krabičkový model s laděním: namísto breakpiontů jsou v Prologu prostředky pro zastavení výpočtu na každé bráně a vypsání příslušných hodnot argumentů. K tomu používáme :-

[15] Predikát `findall` s tímto ale *neumí* pracovat.

4.2 Predikáty `trace` a `notrace`

Oba predikáty tvoří dvojici – `trace` zapíná trasování, zatímco `notrace` je vypíná.

Po zadání `trace` se Prolog uvede do jiného stavu, v němž zastavuje na každé bráně každého predikátu. Tak například po spuštění

```
?-add(s(0),s(s(0)),X).
```

dostáváme postupně ladící výpisy v této podobě:

```
[Call 1] add(s(0),s(s(0)),_1) ?
[Call 2] add(0,s(s(0)),_2) ?
[Exit 2] add(0,s(s(0)),s(s(0))) ?
[Exit 1] add(s(0),s(s(0)),s(s(s(0)))) ?
```

Samozřejmě můžeme dále ovlivnit průběh trasování, a to odklepnutím některého písmene (příkazu):

c – continue, pokračuje výpočet i trasování v nezměněné podobě

s – skip, pokračuj a zastav se na nějaké bráně téže krabičky

r – retry, vrať se na bránu *call*; to je užitečné zejména v případě, kdy výpočet např. zfailuje, ale vzhledem k předchozímu příkazu `skip` nevíme, proč[16]

a – abort, zruš celý vypočet

h – help, nápověda.

4.3 Ostatní predikáty pro ladění

Pokud chceme zabránit přílišné záplavě ladících výpisů, které způsobí přesný opak toho, co jsme si přáli, čili totální chaos, můžeme použít některý z následujících predikátů:

`spy/1` „špión“: argumentem je atom/arita nebo seznam; nastaví se „špionážní“ (sledovací) body do příslušných krabiček

`nospy/1` zruší špióna (pro argument platí to stejné)

`nodebug/0` má stejný účinek, jako `nospy` na všechno

`debugging/0` vypíše predikáty, na kterých sedí špión.

Prolog přitom počítá s tím, že výpisy příliš dlouhých argumentů by mohly být nepřehledné, a proto dává programátorovi možnost výpis předefinovat. Při ladění se totiž termy nevypisují predikátem `write`, ale `portray`, který je definován následovně:

```
portray(X):-print(X),!.
portray(X):-write(X).
```

Predikát `print` přitom definován *není*; v základní podobě tedy selže a uplatní se predikát `write`. Pokud však nadefinujeme `print` (např. jako výpis pouze hlavního funktoru a arity – anebo čehokoliv jiného), uplatní se, a predikát `write` se nezavolá.

Jiné ladící prostředky v klasickém Prologu nejsou.

[16] Příkaz samozřejmě nemá smysl, pokud mezitím došlo k nevratným změnám – např. operace se soubory, zásahy do databáze apod.

5 Rekurzívní datové struktury

5.1 Seznamy

Úvodní poznámky k prologovským seznamům a jejich zavedení jako dvojic[17] obsahovala část 3.3.1. Nyní si ukážeme některé základní operace se seznamy, a posléze některé speciální druhy seznamů.

Hledání prvku v seznamu – `member(X,List)`.

```
member(X,[]):-!,fail. – je zde zcela zbytečné
member(H,[H|_]).[18]
member(X,[_|T]):-member(X,T).
```

Takto nadefinovaný predikát `member` však můžeme použít nejen ke zjištění, zda je určitá hodnota prvkem seznamu, ale také např. tímto kuriózním způsobem: zadejme

```
?-member(a,L).
```

Dotaz kupodivu uspěje; v proměnné `L` se nám pak objeví všechny seznamy obsahující `a`, tedy:

```
L=[a|_];
L=[_,a|_];
L=[_,_,a|_]; ...
```

Spojení seznamů – `append(L1,L2,L)`, kde `L` je výsledný spojený seznam.

```
append([],L,L).
append([H|T],L,[H|LL]):-append(T,L,LL).
```

To odpovídá obvyklému přístupu, kdy první seznam rozebereme na jednotlivé prvky, na poslední místo (přesněji: místo posledního `nil`) přidáme druhý seznam, a zrekonstruujeme výsledek.

Takováto kompozice seznamů je však – stejně jako přidávání na konec, provedené podobným způsobem – velmi „drahá“ (tedy časově náročná) operace: seznam je nutno rozebrat a znovu sestavit. Naštěstí můžeme tuto nepříjemnost obejít pomocí tzv. rozdílových seznamů:

5.1.1 Rozdílové seznamy

Představme si, že místo seznamu jednoho vytvoříme seznamy dva:

```
D=[x1,...,xn]
=[x1,...,xn,xn+1,...,xn+m]-[xn+1,...,xn+m]
```

tedy jako bychom náš seznam `D` zkonstruovali jako jakýsi „rozdíl“ dvou seznamů, přičemž tím druhým, „odečítaným“, seznamem je doplňkový fiktivní seznam, na jehož složení nezáleží. (Prázdný seznam zapíšeme v notaci rozdílových seznamů zcela přirozeně `L-L`.)

Takovéto seznamy se spojují mnohem snáze:

```
append(X-Y,Y-Z,X-Z).
```

Přidání prvku na konec je rovněž snadné: máme-li seznam `D=[a,b,c|R]-R`, a máme přidat prvek `d`, vytvoříme seznam `DD=[d|RR]-RR`, proměnná `R` se nainstanciuje na `[d|RR]`, a tak – podobně jako při spojování – dostaneme výsledek `D1=[a,b,c,d|RR]-RR`.

Přidávání na začátek zůstává snadné – `D1=[a|D]`.

Poznámka. V samotném zápise `[a,b,c|R]-R` je `R` volnou proměnnou.

[17] Tedy seznam `[Hlava|Ocas]` jako `.(Hlava,Ocas)`.

[18] Všimněme si zde zcela přirozeného použití anonymní proměnné.

5.1.2 Uspořádané seznamy

Uvedeme si nejprve proceduru, která zjistí, zdali je seznam uspořádaný či nikoliv:

```
uspořádaný([]).
uspořádaný([_]).
uspořádaný([A,B|R]):-A<B,uspořádaný([B|R]).
```

Zde nejprve prohlásíme prázdný a jednoprvkový seznam za uspořádané, a pak definujeme rekurzívní pravidlo pro dvou- a víceprvkové seznamy. Všimněme si, že musíme požadovat uspořádanost *celého zbytku* `[B|R]`, nikoli pouze `R`, jinak by stačilo mít seznam „uspořádaný po dvojicích“!

A teď jak seznam pomocí tohoto testu setřídit:

```
třídit(L,S):-perm(L,S),uspořádaný(S).
```

Tento přístup vypadá – a také kupodivu i je – logicky v pořádku; je však naprosto nejhorší z toho, co by se dalo napsat: vezmeme totiž *libovolnou* permutaci a zkoušíme, jestli je náhodou uspořádaná.

Permutaci můžeme přitom definovat např. tímto způsobem:

```
perm([],[]).
perm([H|T],P):-perm(T,W),insert(H,W,P).
insert(X,Yz,Yxz):-append(Y,Z,Yz)[19],append(Y,[X|Z],Yxz).
```

Zde tedy permutujeme seznam tak, že první prvek vložíme na libovolné místo zpermutovaného zbytku.

Jiný způsob naopak vybere libovolný prvek na první místo permutace, a pak teprve zpermutuje zbytek:

```
perm([],[]).
perm([H|T],P):-select(H,P,R),perm(T,R).
select(X,[X|R],R).
select(X,[H|T],[H|R]):-select(X,T,R).
```

Nabízí se ještě uvést proceduru pro spojení (merge) dvou setříděných seznamů, která toto uspořádání zachovává.

```
merge([],L,L).
merge([H|L1],[X|L2],[X|M]):-X<H,!,merge([H|L1],L2,M).
merge([X|L1],L2,[X|M]):-merge(L1,L2,M).
```

Program je opět poměrně průhledný, takže jej ponechávám bez komentáře.

5.2 Čítače

Přestože v Prologu jakožto logickém programovacím jazyce nejsou prostředky pro psaní cyklů v klasickém smyslu, není obtížné vytvořit proceduru, která pracuje s čítačem. Ukážeme si to na následujícím jednoduchém příkladu, a sice určení délky seznamu:

```
length(L,N):-length(L,0,N).
length([],N,N).
length([_|R],A,N):-A1 is A+1,length(R,A1,N).
```

K tomuto příkladu následující komentář: program zde obsahuje *dvě* procedury, a sice `length/2` a `length/3`. První z nich je přitom ta, kterou budeme vyvolávat; všimněme si, že pouze „předhodí“ seznam `L` druhé, čítač (druhý argument) inicializuje na 0. Dojdeme-li až k prázdnému seznamu, jsme u konce, a tedy vrátíme délku (korektněji: unifikujeme hodnotu čítače – 2. argumentu se třetím argumentem). V opačném případě pokračujeme s hodnotou čítače zvětšenou o jedničku.

Podobným způsobem se dají zkonstruovat i další příklady používající čítače.

[19] Zde používáme append pro dekompozici.

5.3 Slovník

5.3.1 Implementace pomocí databáze

Uvažujme o následujícím problému: máme implementovat česko-anglický slovník a vyhledávání v něm. První možnost, která nás napadne, je vložit slova do programové databáze:

```
slovo(pes,dog).
slovo(krysa,rat).
slovo(kocovina,hangover).
```

Takto ovšem nezískáme možnost rychlého přístupu do slovníku. Ani seznam –

```
[[pes,dog],[krysa,rat],[kocovina,hangover]]
```

nám tuto možnost nedává – tato implementace je naopak vysoce neefektivní. Proto budeme slovník reprezentovat jinak:

5.3.2 Implementace datovou strukturou

Zavedeme pro uložení slov klasický binární strom:

```
tree(Key,Value,Left,Right),
```

kde klíč `Key` představuje české slovo, a `Value` anglický ekvivalent. Napíšeme nyní predikát `lookup(C,T,A)`, jehož argumenty jsou postupně: české slovo, binární strom, a konečně anglický ekvivalent:

```
lookup(C,tree(Key,Value,Left,Right),A):-C=Key,A=Value.
```
nebo stručněji:
```
lookup(C,tree(C,A,_,_),A).
lookup(C,tree(Key,_,L,_),A):-C@<Key[20],lookup(C,L,A).
lookup(C,tree(Key,_,_,R),A):-C@>Key,lookup(C,R,A).
```

Tento predikát se ovšem dá použít jen pro *jednosměrný* překlad: totiž, strom je setříděn podle českých slov, a tato se také testují jako klíče.

Na druhé straně se ale dá takto nadefinované vyhledávání použít i pro uložení dalších slovíček: stačí zadat např. `lookup(krígl,T,jug)`.

Takto definovaný strom je ale neúplný, neboť nemá definovány konce. V listech se totiž objevují volné proměnné; proto doplníme proceduru `lookup` o následující klausuli:

```
lookup(C,T,A):-var(T),!,fail.
```

Druhá možnost je strom „uzemnit“, tedy volné proměnné v listech nahradíme atomem – třeba `konec`:

```
ground(T):-var(T),T=konec.
ground(tree(K,V,L,R)):-var(L),L=konec.
ground(tree(K,V,L,R)):-nonvar(L),ground(L),ground(R).
```

To se dá opět zestručnit:

```
ground(konec).
ground(tree(_,_,L,R)):-ground(L),ground(R).
```

5.3.3 Výpis slovníku

Pokusíme se nyní takovýto slovník vypsat. Nadefinujeme tedy predikát `list`:

```
list(tree(K,V,L,R)):-var(L),write(K),write(V),list(R).
list(tree(K,V,L,R)):-nonvar(L),
                      list(L),write(K),write(V),list(R).
```

[20] Symboly `@<` a `@>` vyjadřují lexikografické uspořádání.

Toto nám totiž nestačí: máme-li totiž „neuzemněný“ strom, vytvářel by se cyklicky „šlahoun“ a program by se zasukoval; musíme tedy dát na začátek

```
list(T):-var(T).
```

Tato klausule nemá účinek, ale uspěje. A to je právě to, co potřebujeme.

5.3.4 Transformace stromu na seznam

Pro úplnost ještě uvedeme prdeikát `tree_to_list(T,L)`, který vytváří seznam, který jsme prapůvodně zavrhli:

```
tree_to_list(T,L):-tree_to_list(T,[],L).
tree_to_list(T,L,L):-var(T).
tree_to_list(tree(K,V,L,R),List0,List):-
             tree_to_list(L,List0,L1),
             tree_to_list(R,[[K,V]|L1],List).
```

Prostřední argument predikátu `tree_to_list/3` je přitom akumulátor, díky kterému – a díky pěknému triku, kdy seznam vytvořený z jedné části stromu „předhodíme“ jako akumulátor do části druhé – jsme se zbavili nepříjemného `append`-ování. S použitím `append` by poslední klausule vypadala takto:

```
tree_to_list(tree(K,V,L,P),List0,List):-
             tree_to_list(L,[],L1),tree_to_list(R,[],L2),
             append(L1,[[K,V]],L3),append(L3,L2,List).
```

což zdaleka není tak elegantní.

5.4 Pole

Vedle jiných prvků obvyklých v běžných jazycích, na které jsme již narazili, Prolog rovněž postrádá *pole* v obvyklém smyslu. Uvedeme si proto některé metody, pomocí nichž můžeme pole reprezentovat.

5.4.1 Pole jako seznam nebo binární strom

První, co nás asi napadne, je reprezentace pole seznamem. Zatímco jindy je použití seznamu docela přirozené a má třebas i některé příjemné vlastnosti, tady je použití seznamu zcela nevhodné – přístup k jednotlivým prvkům je příliš obtížný.

Vhodnější je proto např. vytvořit *binární strom*, který je obzvláště vhodný pro tzv. asociativní pole (k jehož prvkům se přistupuje nikoli pomocí indexu, nýbrž pomocí klíče), a také pole dynamická, tedy nejen dynamicky alokovaná, ale i pravá dynamická pole (která svůj rozměr mění za výpočtu).

5.4.2 Pole konstantní délky

Má-li mít pole konstantní délku, nabízí se poměrně snadno možnost využít operátoru `functor` (viz část 3.3.2, kde jsou funktory popsány podrobněji), a vytvořit pole jako term, jehož jednotlivé argumenty budou zastupovat prvky pole. Napíšeme tedy

```
functor(T,pole,N)
```

a dostaneme tak term následujícího tvaru:

```
pole(_,_,...,_)
```

Zde je na místě malé upozornění, že n – počet argumentů funktoru (prvků pole) bývá omezen, typicky např. na 255.

Přístup k prvkům pole se pak provede jednoduše pomocí `arg(T,N,A)`. Příjemná vlastnost těchto polí – tak typická pro Prolog – pak je ta, že prvky pole (tedy argumenty funktoru) *nemusí být stejného typu.*

5.5 Fronta

Fronta je poměrně důležitá datová struktura. V Prologu ji samozřejmě budeme definovat jako seznam. Nadefinujeme postupně predikáty `empty`, který signalizuje, je-li fronta prázdná, `head`, který odebere z fronty čelo (první prvek), a `last`, který přidá prvek na konec:

```
empty([]).
head(H,[H|Q],Q).
last(X,Q0,Q):-append(Q,[X],Q0).
```

Tato definice je poměrně přirozená, ale zajisté si všimneme použití predikátu `append` pro přidávání na konec, o kterém víme, že je pomalý. Proto náš příklad „vylepšíme“ použitím rozdílových seznamů:

```
empty(Q-Q).
head(H,[H|Q]-B,Q-B).
last(X,Q-[X|B],Q-B).
```

Tento zápis je nejen stručnější a přehlednější, ale hlavně – o což nám šlo – rychlejší.

6 Abstraktní interpret

V této části se budeme zabývat tím, jak se vykonává program v Prologu. Nejprve se proto seznámíme se základní kostrou, podle níž výpočet probíhá, a posléze „nakoukneme pod pokličku“ na uložení dat a na možné optimalizace činnosti programu.

6.1 Schéma výpočtu

Základní schéma vykonávání programu shrneme do následujících bodů:

1. Inicializujeme množinu $S = G$, tedy na cíl.
2. Dokud $S \neq \emptyset$, dělej:
3. Vyber $A \in S$, klausuli $A' : -B_1, ..., B_n$ a najdi $unify(A, A')$ σ tedy $A\sigma = A'\sigma$
4. $S = S - \{A\} \cup \{B_1, ..., B_n\}$
5. aplikuj σ na S i G (abychom dostali odpověď)
6. Konec cyklu
7. $S = \emptyset \Rightarrow G$ vydej jako výsledek

Při bližším prozkoumání tohoto schématu ovšem narazíme na následující nedostatek: není řečeno, jak v bodě 3 vybrat literál A a klausuli $A' : -B_1, ..., B_n$. (Tento abstraktní interpret je totiž „geniální“ – vždy vybere klausuli, která vede k cíli, a nestará se o neúspěšné větve.) Prakticky se to provádí tak, že se vybere nejlevější literál, a stejně i $B_1, ..., B_n$ se přidávají do S doleva.

Klausule jsou v programu uspořádány, a jejich pořadí se vždy dodržuje. Problémem však zůstává otázka, zda SLD-strom prohledávat do hloubky nebo do šířky: obecně to totiž nemusí být ekvivalentní, a hlavně to má různé paměťové nároky.

Nyní můžeme říct, že *program v Prologu* je posloupnost volání procedur řízených unifikací.

6.2 Funkce zásobníku

Narozdíl od imperativních programovacích jazyků vyvstává v Prologu problém: je třeba rozlišit dva návraty z procedur, a sice:

- po úspěchu – budeme pokračovat v rezoluci,
- po neúspěchu – budeme zkoušet alternativy.

Musíme tedy nějakým způsobem uložit bod návratu (kde se má pokračovat), lokální proměnné, a co dělat, jestliže se neuspěje. Proto definujeme:

okolí procedury – bod návratu a lokální proměnné (odpovídá úspěchu)

bod volby – odpovídá neúspěchu, obsahuje proto ukazatel do databáze na následující klausuli a argumenty (aby se daly obnovit).

Navíc potřebujeme informace o proměnných, které získávají hodnotu (unifikují se): totiž, pokud v některé větvi výpočtového stromu neuspějeme a jdeme do větve druhé, musíme zrušit všechno, co jsme přiřadili ve větvi, kterou opouštíme.

Pokud se zamyslíme, zjistíme, že jsou jen dva možné stavy proměnných: volná, anebo přiřazená. Při výpočtu se o proměnných udržuje tzv. *stopa* (trail).

TADY BUDOU OBRÁZKY (ze str. 16 mých poznámek)

6.3 Optimalizace základního modelu

Uvažujeme-li dále o výše popsaných funkcích zásobníku, může nás napadnout, že bod volby stačí zakládat jen pro procedury, které *nejsou deterministické* (tedy mohou uspět více způsoby); unifikace tuto vlastnost nemá.

Zřejmě jsou tedy nedeterministické ty procedury, které mají alespoň dvě klausule. Např.

```
a(1):-... .
a(2):-...,... .
```

Zde ovšem např. volání `?-a(1)` je deterministické; avšak `?-a(X)` již může uspět vícekrát.

Toto řeší Prolog *indexací* na prvním argumentu. Mějme např. proceduru, která je tvořena následujícími klausulemi:

klausule	*první argument je*
`a(1):- ...`	konstanta (*const*)
`a(a):- ...`	konstanta
`a([X|Y]):-...`	seznam (*list*)
`a(f(X,a)):-...`	struktura (*struct*)

Vytvoříme tedy rozskokovou tabulku (analogie `select-case`) a při volání uplatníme klausuli odpovídající příslušnému argumentu. Je-li přitom skutečným argumentem volná proměnná, musí se postupně vyzkoušet všechny alternativy.

Dalším pozitivním důsledkem vedle úspory paměti je i toto: jestliže procedura uspěje, systém se „podívá“ na zásobník, a pokud nejsou žádné body volby, zruší se okolí.

Řez přitom odřízne všechny body volby nad ním, ovlivní se tedy nejen směr výpočtu, ale i paměťová náročnost. Např. klausule

```
a(..):-..,.., ... ,!.
```

je deterministická; na koci se „zařízne“ zásobník.

6.4 Vyjádření iterace rekurzí

Vezměme „klasickou“ definici predikátu `member` pro zjištění přítomnosti prvku v seznamu:

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

Uvažujme nyní o následujícím vylepšení optimalizace: založíme na zásobníku nejprve bod volby, potom okolí.

Nechť nyní vytvoříme bod volby, a nechť např. první klausule neuspěje. Pak „zařízneme“ bod volby (poněvadž jsme se dostali k poslední klausuli), založíme okolí; to se však „odřízne“ už před posledním podcílem, čili (v našem případě) hned na začátku. Zkopírují se tedy proměnné, zruší se okolí, a procedura se zavolá *skokem*.

Takto nadefinovaný predikát `member` má tedy paměťovou náročnost *konstantní* – nezáleží na délce řetězce (seznamu).

Máme-li pak klausuli tvořenou nejvýše jedním podcílem, není třeba okolí zakládat vůbec – všechno jsou argumenty nebo na haldě.

Tato operace se nazývá **TRO** – *Tail Recursion Optimization.* Může fungovat jak v interpreteru, tak v kompilátoru Prologu.

Poznámka. Některé systémy umožňují programátorovi zvolit si argument (tedy číslo argumentu), na kterém má provádět indexaci (popsanou v 6.3); je možné přitom zadat i více různých pozic.

7 Kompilátor Prologu

7.1 Schéma WAM

Prof. Warren navrhl schéma kompilátoru, které se po něm nazývá Warren Abstract Machine. Tato abstrakce obsahuje:

- struktury pro organizaci paměti (*heap, trail, stack, push-down list*), OBRÁZEK!
- registry:

  A-registry čili argumentové: jsou globální, ukládají se do nich argumenty každé klausule: první argument do A1, druhý do A2, atd.

  Y-registry představují abstraktní znázornění lokálních proměnných. Tak např. v klausuli

  ```
  a(X):-b(X,Y),c(Y).
  ```

  se do proměnné Y1 uloží proměnná Y, protože musí „přežít“ volání `b`.

- instrukce:
  - ⋆ řízení (*call, goto*)
  - ⋆ přebírání argumentů
  - ⋆ příprava argumentů
  - ⋆ indexační instrukce.

Většina kompilátorů přitom funguje právě podle schématu WAM: buď se instrukce WAM dále přeloží, anebo se již vykonávají – emulují (interpretují).

7.2 Práce se seznamy

Tato poznámka se týká další optimalizace paměťové náročnosti, kterou si ukážeme na příkladu spojování setříděných seznamů – predikát `merge`, který jsme již tvořili v části 5.1.2 (tam jsme probírali setříděné seznamy vůbec):

```
merge([H1|T1],[X|L2],[X|M1]):-X@<H1,!,merge([H1|T1],L2,M1).
merge([X|L1],L2,[X|M1]):-merge(L1,L2,M1).
merge([],L,L).
```

Podíváme-li se na první klausuli podrobněji, zjistíme, že se zde seznam zbytečně dekomponuje (na hlavu a ocas) a znovu vytváří, a toto všechno se dostává na haldu. Proto raději tuto klausuli přepíšeme na

```
merge(L1,[X|L2],[X|M1]):-L1=[H1|_],X@<H1,!,merge(L1,L2,M1).
```

přičemž nemáme-li „inteligentní“ kompilátor, je vhodnější dále psát namísto `L1=[H1|_]` něco jako `decomp(L1,H1)`, kde `decomp` je definováno:

```
decomp([X|_],X).
```

7.3 Vyjádření cyklů a `if-then-else`

7.3.1 Cyklus

Nejobvyklejší je „klasické“ vyjádření cyklu rekurzí. Ukážeme si to na příkladu zjišťování délky seznamu:

```
len([],0).
len([_|Ocas],M):-len(Ocas,N),M is N+1.
```

Toto je neefektivní, poněvadž zde se okolí vytvořit musí, a nedá se tak použít TRO.

Zavedeme si tedy čítač (akumulátor – viz též 5.2):

```
len(List,Len):-len(List,0,Len).
len([],N,N).
len([_|Ocas],L0,L):-L1 is L0+1,len(Ocas,L1,L).
```

Podařilo se nám tedy přesunout rekurzívní volání na konec, což je díky možnosti použití TRO efektivnější – zabírá $O(1)$ paměti namísto $O(n)$.

7.3.2 Cyklus `repeat`

Další možnost je vyjádřit cyklus backtrackingem (pomocí neúspěchu), čímž dostáváme obvyklé cykly `repeat` (můžeme tedy `repeat-until` – „opakuj, dokud není splněna podmínka“ chápat jako „opakuj, dokud není úspěch“). Vestavěný predikát `repeat` je přitom definován

```
repeat.
repeat:-repeat.
```

Takže: `repeat` uspěje, při neúspěchu (čehokoli dále v „cyklu“) se tedy vrátíme na `repeat`, který znovu uspěje (podle druhé alternativy) a nastartuje znovu tělo cyklu. (Všimněme si, že pokud použijeme `repeat`, backtracking se před něj nikdy nedostane.)

Obecné schéma procedury využívající cyklus (predikát) `repeat` pak bude:

```
p(Args):-initialize(Args),
         repeat,
             get_next_value(D),
             process(D),
             last(D),
         !,shut_down.
```

Zde inicializace představuje část před cyklem, volání predikátů `get_next_value` a `process` pak tělo cyklu, `last` je výstupní podmínka (není-li `D` poslední, neuspěje, a následuje návrat na `repeat`); řez před ukončením (`shut_down`) je nezbytný proto, abychom se nevrátili zpět na cyklus.

Poznámka. Pomocí cyklu `repeat` můžeme snadno vyzkoušet, zdali náš Prolog používá optimalizaci TRO či nikoliv: zadáme-li totiž

```
?-repeat,fail.
```

pak Prolog bez TRO skončí s přetečením paměti, zatímco TRO „způsobí“ zacyklení a výpočet neskončí.

7.3.3 Jak psát tělo cyklu

V cyklu `repeat` by se měly používat zásadně *deterministické* predikáty, přesněji řečeno bez bodů volby, popřípadě alespoň do nich zapsat jako poslední podcíl řez, aby se výpočet do nich nevracel.

Druhá možnost je, napsat v cyklu na příslušných místech

```
once(get_next_value(D)),
once(process(D)),
```

kde predikát `once` provede svůj argument právě jednou (nejvýše jednou), a je buď vestavěný, anebo si ho můžeme nadefinovat jako

```
once(G):-call(G),!.
```

Jako další metodickou poznámku uvedeme následující příklad:

```
p(Args,D):-initialize(Args),
           repeat,
               get_next_value(D),
               (last(D),
                  !,
                  shut_down,fail
               ;true
               ).
```

Tato technika psaní cyklu je *špatná*, neboť napíšeme-li nyní

```
p(File,T),!,write(T),fail.
```

pak řez (mezi voláním `p` a `write`) způsobí, že se nikdy neprovede `shut_down`.

7.3.4 Definice `if-then-else`

Nadefinujeme podmínku jako ternární operátor:

```
If->Then;Else :- call(If),!,call(Then).
If->Then;Else :- call(Else).
```

Zde je podstatné to, že takto definovaný operátor hledá pouze *první* řešení podmínky `If`.

Proto např. máme-li databázi

```
a(1).
a(2).
b(2).
e(1).
```

a zadáme

```
?-a(X):-b(X),e(X).
```

pak pokud `X=1`, dotaz zcela přirozeně selže (`If` je splněn, ale `Then` nikoliv). Je-li ale X volná proměnná, opět neuspěje, protože se provede pouze první instanciace (`X=1`), zatímco řešení pro `X=2` se nenajde.

Pomocí takto definovaného `if-then-else` se dá také nadefinovat sekvence `if-elseif`:

```
Test1->WhenTrue1;(
Test2->WhenTrue2;(
...(
TestN->WhenTrueN;
WhenAllElseFail)..))
```

Závorky zde vyjadřují, že se vlastně jedná o bloky `if-then` vnořené do sebe.

7.4 Datovod (pipe, roura)

Datovod (unixovskou „trubku“ – pipe) můžeme v Prologu nadefinovat pomocí cyklu `repeat`:

```
read_and_print(Fajl):-seeing(OldF),
                      see(Fajl),
                      repeat,
                          read(T),
                          write(T),nl,
                          eof(T),
                      !,seen,see(OldF).
```

Příklad je opět jednoduchý, bez komentáře; všimněme si pouze, že po vypsání souboru opět přesměruje výstupní proud do původního `OldF`.

7.5 Predikáty skupiny `retract`

Připomeňme si (podrobněji viz 3.2.3), že predikát `retract` (který voláme – jak jsme uvedli dříve – `retract(H:-B)`) zajistí vyřazení klausule z programové databáze. Uveďme nyní následující definici:

1. `retractall(H):-functor(H,F,A),functor(H1,F,A),retractall1(H1).`
2. `retract(H:-B),fail.`
3. `retractall(_).`

K tomu komentáře:

- v bodě 1 vytvoříme term `H1` tak, že nezáleží na instanciaci argumentů termu `H` (argumenty termu `H1` jsou volné proměnné)
- predikát `fail` v bodě 2 funguje tak, že se vracíme na bod volby; až `retract` neuspěje (už nic neexistuje), skočíme na bod 3, ten uspěje a skončíme.

Procedura `retractall1` má logické čtení `true`, a jako svůj „boční efekt“ zruší všechny klausule dané hlavy.
Poznámka. Všimněme si, že zde jsme napsali cyklus bez použití rekurze a bez ukládání výsledků na haldu.

8 Vyjádření gramatik v Prologu

Uvažme klasickou bezkontextovou gramatiku $G = (N, T, P, S)$, kde N, P, T, S jsou postupně množina neterminálních a terminálních symbolů, množina přepisovacích pravidel a počáteční symbol gramatiky (kořen). Protože máme gramatiku bezkontextovou, je $P \subseteq N \times (N \cup T)^*$; prázdné slovo budeme v souladu s běžnými konvencemi značit jako ε.

Jak tyto úvahy mohou souviset s logickým programováním, si ukážeme na příkladu gramatiky přirozeného jazyka:[21]

<*věta*> → <*podmět*><*přísudek*>
<*podmět*> → `the` <*podst.jméno*>
<*podst.jméno*> → `rabbit`
<*podst.jméno*> → `engine`
<*přísudek*> → `runs`
<*přísudek*> → `eats`

To silně připomíná logický program tak, jak jsme na něj zvyklí; snadno tedy odhalíme přepis jako

```
věta(L):-append(Podm,Přís,L),podmět(Podm),přísudek(Přís).
podmět([the|X]):-podst_jméno(X).
podst_jméno(rabbit).
podst_jméno(engine).
přísudek(runs).
přísudek(eats).
```

Poznámka. Přepis bezkontextové gramatiky do Prologu zřejmě odpovídá syntaktické analýze shora dolů.

8.1 Rozdílové seznamy

Jestliže reprezentujeme věty a jejich části jako seznamy, zajisté nás napadne možnost použití rozdílových seznamů:

```
věta(S0-S2):-podmět(S0-S1),přísudek(S1-S2).
podmět([the|S1]-S2):-podst_jméno(S1-S2).
 nebo:
podmět(S0-S2):-diff(S0-S1,[the]),podst_jméno(S1-S2).
diff([X|Y]-Y,[X]).
```

[21] Úvahy se budou týkat angličtiny (proto se zde objeví např. určitý člen).

8.2 Kontextová gramatika

Analýza kontextové gramatiky již tak přímočará není, i když opět lze provést snadným způsobem. Mějme např. gramatiku popisující jazyk, který obsahuje stejný počet znaků a, b, c:

$$L = \{a^n b^n c^n | n \geq 0\}$$

Analýzu provedeme pomocí dalšího argumentu, který vyjadřuje počet jednotlivých znaků:

```
abc(S0-S3):-a(N,S0-S1),
            b(N,S1-S2),
            c(N,S2-S3).
a(0,S-S).
a(s(N),[a|S1]-S2):-a(N,S1-S2).
b(0,S-S).
b(s(N),[b|S1]-S2):-b(N,S1-S2).
c(0,S-S).
c(s(N),[c|S1]-S2):-c(N,S1-S2).
```

Chceme-li jazyk dále specifikovat, např. pouze pro n sudé – tedy

$$L = \{a^n b^n c^n | n \geq 0 \wedge 2|n\}$$

stačí doplnit

```
abce(S0-S3):-a(...),b(...),c(...),even(N).
```

8.3 DCG – Definite Clause Grammars

Jsou to gramatiky Hornových klausulí. Pro jejich zápis jsou čtyři základní pravidla:

- terminály píšeme vždy v seznamu (např. `[the]`)
- určité funktory nesmíme použít jako neterminál (např. `./2`)
- zavádí se speciální funktor `-->/2`, který odděluje levou a pravou stranu pravidla (`A-->B1,...,Bn`)
- ε reprezentujeme prázdným seznamem, čili ε-pravidlo $A \to \varepsilon$ zapíšeme `A-->[]`
- (páté, doplňkové) pravidla můžeme doplnit o volání prologovských podcílů; zapisujeme je do složených závorek (`{even(N)}`).

Můžeme proto psát např.:[22]

```
abc-->a(N),b(N),c(N).
a(0)-->[].
a(s(N))-->[a],a(N).
abce-->a(N),b(N),c(N),{even(N)}.
```

Zde tedy vidíme příklad volání predikátu podle doplňkového pravidla.

Ještě malá poznámka k použití řezu: přestože řez se nebere v zásadě jako neterminál, je vhodnější psát vždy `{!}`.

9 Predikátový počet

9.1 Predikátová logika 1. řádu

Abecedou predikátového počtu budeme rozumět písmena, číslice, podtrženítka, dále vlastní symboly, logické spojky, kvantifikátory.

Formule v predikátové logice 1. řádu jsou pak predikáty a funkční symboly; *dobře utvořená formule* označuje pravdivostní hodnoty.

[22] Predikáty `b,c` napíšeme analogicky – jako v předcházejícím příkladě.

9.2 Herbrandova interpretace

Definice.

Herbrandovo univerzum U(P) je množina všech uzavřených termů, které mohou být tvořeny predikáty a funkčními symboly z P.

Herbrandova báze B(P) je množina všech atomických formulí nad prvky U(P).

Herbrandova interpretace přiřazuje

- proměnným prvky Herbrandova univerza
- a konstantám sebe sama.

Věta. Množina P klausulí je splnitelná, právě když existuje Herbrandův model.
Dále si připomeňme definice unifikace, unifikátoru a nejobecnějšího unifikátoru (viz Zlatuškova logika).

9.3 Rezoluce v logice 1. řádu

Opět pouze připomenutí: je-li ϱ přejmenování proměnných a je-li σ nejobecnější unifikátor $A_1\varrho, ..., A_n\varrho, B_1, ..., B_n$, pak

$$P \cup \{A_1, ..., A_n\}, \{\neg B_1, ..., \neg B_n\} \cup Q$$

rezolvujeme na $P\varrho\sigma \cup Q\sigma$.
Dále pak $\neg A \vee B, A$ zjednodušíme na B (toto se nazývá *modus ponens*, též pravidlo řezu).

9.3.1 Varianty rezoluční metody

Jako variatnty rezoluce, jejíž princip jsme právě popsali, si uvedeme zejména následující:

Sémantická rezoluce – zvolíme interpretaci a při rezoluci použijeme jen takové klausule, z nichž aspoň jedna je v této interpretaci nepravdivá.

Použití oporných množin – zde rozlišujeme mezi axiomy a dokazovaným teorémem. *Opornou množinou* T množiny klausulí S přitom nazýváme takovou $T \subseteq S$, že množina $S - T$ je bezesporná; při rezoluci pak používáme jen klausule z T a/nebo z předchozí rezolventy.

P_1–rezoluce – jedna z klausulí, které se účastní rezoluce, musí být pozitivní.

W_1–rezoluce – jedna z klausulí, které se účastní rezoluce, musí být negativní.

Lineární rezoluce – v každém kroku (kromě prvního) můžeme použít bezprostředně předchozí rezolventu a k tomu buď některé klausule ze vstupní množiny S nebo některé z předchozích rezolvent.

9.3.2 Neúplné varianty rezoluce

Sem patří:

jednotková rezoluce – aspoň jedna klausule použitá při rezoluci je jednoprvková

vstupní rezoluce – aspoň jedna klausule použitá při rezoluci pochází z výchozí množiny S.

10 Prolog s omezujícími podmínkami

[23] Logické programování s omezujícími podmínkami (tzv. Constraint Logical Programming – CLP) se vyznačuje zejména:

Explicitní reprezentace objektů. Např. píšeme $\{1, .., 1000\}$. Obtíže jsou např. u kružnice.

[23] Odtud to není z mých poznámek, ale opisuju to od Hanči a nechápu. Takže by se možná slušelo trochu to doplnit a opravit.

Omezení na Herbrandovo univerzum. Neexistuje interpretace s naším konkrétním oborem hodnot.

Problémy s negací. Po vyčerpání všech možností nastává neúspěch (finite failure).

Unifikace a kanonická reprezentace termů. Vezměme např. Peanovu aritmetiku – $0, s(0)$ atd. Doplníme nyní `fact(X)`

```
s(0),fact(0),fact(1)
```

pak ale unifikace `s(0)=fact(0)` *selže.*

Tzv. *sémantická unifikace* rozšiřuje unifikaci o systém rovnic, tzv. *přepisovací pravidla.* Zavedeme tedy např. `s(0)=fact(0)`; tyto rovnice se pak při unifikaci používají jako přepisovací systém. Je tak dána „ekvivalence" mezi objekty Herbrandova univerza. Není to však dokonalé – pro daný systém rovnic se těžko rozhoduje (má exponenciální složitost, neboť se jedná o NP-úplný problém).

Strategické vyhodnocování klausulí neboli vyhodnocování zleva. Tak např.

```
?-X=a,var(X).  neuspěje, zatímco
?-var(X),X=a.  uspěje.
```

To se řeší tzv. *pozdržením výpočtu* (přepisovací systém v Prologu II), a sice zavedením predikátu `freeze(X,Goal)`, který funguje takto: je-li `X` instanciováno, proveď `call(Goal)`; jinak také jakoby uspěje, avšak `Goal` se provede až po instanciaci `X`.

Tak např. `?-freeze(X,var(X)),X=a` vždy selže.

10.1 NU–Prolog

Podobně funguje v NU-Prologu deklarace `when`: zápis

```
when X:var(X)
```

způsobí, že `var(X)` se spustí, až když je `X` instanciováno.

Místo toho, abychom testovali něco vygenerovaného (princip *generate and test*), tak přímo generujeme něco blíže specifikovaného (*constraint and generate*), což je pochopitelně efektivnější.

10.2 Náhrada Herbrandova univerza

Herbrandovo univerzum nahradíme algebraickou strukturou $\mathcal{A}$. Budeme uvažovat o *unifikaci řešitelností* (dokazovat řešitelnost a hledat řešení).[24]

$\forall d \in \mathcal{A}, \lceil = \cap\mathcal{C}$	klasický Prolog	$(HB, =)$
$\forall C, \bar{C} = \cup C$	semantický Prolog	$(HB\|E, =)$

Příklad. Mějme množinu reálných čísel s obvyklými operacemi. Pak např.

$X + 3 > Y * 3$ je omezující podmínka pro X, Y, zatímco
$a + 3 > 7$ je *nekorektní*, neboť a není objekt $\mathcal{R}$.

K omezujícím podmínkám existuje přístup

- *imperativní*: $Var = Exp$
- *lokální*: $Expr1\ rel\ Expr2$. Tak např. $X > 4, X < -4$ neselže.
- *globální*: snažím se dokázat řešitelnost větších celků. Máme tedy $H : -c_i, b_j, \forall i \geq 0, j \geq 0$, kde b_j jsou podcíle a c_i omezující podmínky. Zkoumáme tedy, zda je $\{c_i\}$ splnitelné, pak řešíme (unifikujeme) podcíle.

[24] V dalším textu bude C značit omezující podmínku.

11 Prohledávání stavového prostoru

Obvykle se v Prologu používá

- generuj a testuj,
- standardní backtracking.

Obě tyto metody v sobě obsahují vzpamatování se po neúspěchu.

Consistency techniques – constraint satisfaction:

- dopřená kontrola – forward checking
- pohled dopředu – looking ahead (je složitější).

Tyto metody se naopak snaží o *předvídání* neúspěchu.

11.1 Nevýhody backtrackingu

- opakovaný výpočet již nalezených faktů
- pozdní rozeznání neúspěchu
- špatné body návratu.

11.2 Apriorní kontrola

Jsou definovány *obory hodnot* D proměnných.[25] Unifikace se chová krapet odlišně:

- unifikace d_i s konstantou uspěje, je-li $c \in D_i$
- unifikace d_i s d_j (nechť $D_k = D_i \wedge D_j$) uspěje
  $d_i = d_k$, $d_j = d_k$, je-li $card(D_k) > 1$;
  $d_i = d_j = c$, je-li $card(D_k) = 1$ a $D_k = \{c\}$;
  neuspěje, je-li $D_k = \emptyset$.

Dopředná deklarace se pak zapíše

$$domain\ p(a_1, ..., a_n)$$

kde a_i je H (Herbrandovo univerzum) nebo D pro $1 \leq i \leq n$

$$forward\ p(a_1, ..., a_n)$$

kde a_i je g (ground term) nebo d (doménová proměnná) pro $1 \leq i \leq n$.

Forward checking se spustí jen je-li jedna proměnná doménová (volná); ta podmínka omezí doménu a už se nemusí opakovat.

11.3 Specializace

Nadefinujeme např. $\neq, forward\ d \neq d; X \neq Y :- not(X = Y)$. Např. máme `X=[1,2,3]`, `Y=[1,2]`; `X≠Y, Y=1`, pak `X=[2,3]`.

Můžeme použít k řešení problému osmi dam[26].

Systém CHIP má i deklarace oboru hodnot, forward deklarace, look ahead deklarace. Dále uveďme systém Charme and Bull (asi jsem to blbě opsal) a Prolog III (Kolmerauer).

Použití:

- rozvrhy
- plánování (kritické cesty).

[25] Budeme značit D_i obor hodnot proměnné d_i (doufám, že ne naopak).

[26] Známá úloha – jak postavit na šachovnici osm dam tak, aby se nesežraly.

12 Paralelní logické programování

Budeme zde hovořit o transparentní paralelizaci (kompilátor) a explicitní paralelizaci (změněná sémantika) v Parlogu, tedy parelelním Prologu. Rovněž se zmíníme o Concurrent Prologu a o tzv. strážených Hornových klausulích (guarded Horn clauses – GHC).

12.1 Transparentní paralelismus

Při tomto přístupu se snažíme vyhodnotit *každou* klausuli.

12.1.1 `and`–paralelismus

znamená, že jednotlivé podcíle dané klausule řešíme paralelně. Nastávají však tyto problémy se společnými proměnnými. Řešení:

- zavedeme proto operátor `&`, který znamená (jako náhrada čárky), že tyto dva cíle mohou běžet paralelně (ale nepoužívá se to).
- konzervativní odhad – pustíme paralelně jen ty podcíle, kde by se proměnné neměly překrývat, avšak je to za cenu nízké míry paralelismu
- anotace proměnných – u proměnné říkáme, zda se proměnná v cíli smí instanciovat. Příslušné spuštění cíle se pak pozdrží do doby, než proměnná získá jednoznačnou hodnotu. Pozor, snadno může dojít k deadlocku.
- slabá (weak) unifikace – používá se pro práci s anotovanými proměnnými

12.1.2 `or`–paralelismus

Je nutno zajistit duplikace všech proměnných, které jsou přes argumenty dosažitelné. Dále musíme zachovat uspořádání (řešení v první větvi musím dostat vždy dříve než ve druhé). Proto také např. řez (cut) musíme uplatnit až když jsou všechny klausule před řezem provedeny; do té doby se výpočet pozdrží.

Výhoda je, že i když je např. první větev nekonečná, můžeme dostat řešení z větve druhé (určíme, že chceme okamžitě všechna řešení); nevýhodou jsou pak neúměrné požadavky na paměť.

12.2 Explicitní paralelismus

Transformační (uzavřené) programy obecně končí, tj. dostanou vstupní údaje, počítají a nakonec poskytnou výsledek. Oproti tomu *reaktivní* (otevřené) programy nekončí (obecně), jsou ve stálé interakci s okolím, reagují na podněty, avšak neposkytují výsledek samy o sobě (např. databáze, operační systémy).

Paralelní systémy jsou konkurentní systémy implementující transformační programy; avšak každá složka paralelního systému je de facto reaktivním programem.

12.2.1 Procesová sémantika

Shrňme si přehledně význam jednotlivých odpovídajících pojmů tak, jak je známe z nauky o paralelním programování, a z programování logického:

proces	atomický cíl
síť procesů	konjunkce cílů
instrukce procesu	klausule
komunikační kanál	logická proměnná
komunikace, přiřazení	obecná unifikace
synchronizace, předání parametrů	vstupní unifikace (forma slabé unifikace)

12.2.2 Don't Know nedeterminismus

znamená, že se programátor nestará o to, která z možností je správná (neví to) – program sám při vykonávání vybere (dříve nebo později) správnou cestu. Výsledky neúspěšných cest nehrají roli (nejsou pozorovatelné).

12.2.3 Don't Care nedeterminismus

oproti tomu znamená, že svou roli hrají i neúspěšné cesty; pokud je vybrána nějaká cesta, ostatní cesty se již nevyužijí (zapomenou se).

12.2.4 Stráže

Program je množina *stážených (guarded)* klausulí tvaru

$$H :- G_1, ..., G_m | B_1, ..., B_n;\ n, m \geq 0,$$

kde H je hlava, G_i jsou stráže, | je operátor upnutí, a B_j tvoří tělo klausule. Deklarativně lze klausuli číst: H platí, platí-li G_i a B_j.

Z hlediska chování je takto definována síť procesů, kde každé B_j je proces.

Jakmile se v jedné cestě dojde k operátoru upnutí, ostatní cesty se zapomenou. *Stráže* představují zpravidla jednoduché testy; používá se v nich slabá unifikace. Operátor upnutí funguje vlastně místo řezu.

1. *Stav výpočtu* je dvojice (S, θ), kde

   S je multimnožina atomických formulí (jeden prvek může být obsažen vícekrát)

   θ je substituce.

2. *Transformace stavů výpočtu* jsou zobrazení mezi jednotlivými stavy. Redukce pak je

   $$((p_1, ..., p_i, ..., p_n), \theta) \rightarrow ((p_1\vartheta, ..., B\vartheta, ..., p_n\vartheta), \vartheta \circ \theta),$$

   kde $H :- B$ je klausule a ϑ největší společný unifikátor $mgu(H, p_i)$. Přitom

   $$spch(p, \theta) \rightarrow (spch, \theta)$$

   $$nespch(p, \theta) \rightarrow (nespch, \theta)$$

3. *Výpočet* je posloupnost transformací stavů začínající cílem a prázdnou substitucí, a končící v úspěchu nebo neúspěchu (v posledně jmenovaných stavech).

12.2.5 Chování procesů

`A:-true.`	ukončení procesu
`A:-B.`	změna stavu z `A` na `B`
`A:-`B_1`,...,`B_n`.`	rozštěpení `A` na procesy $B_1, .., B_n$

12.3 Omezený `or`–paralelismus

Paralelně jsou spuštěny všechny větve, ale jakmile jedna z nich provede operátor upnutí, jsou ostatní `or`-paralelní větve násilně přerušeny.

Stráž obecně může být cokoli; všechny operace ale nemohou být provedeny na úrovni stráží (dochází k problémům jako v `or`-paralelismu – extralogické operace).

12.4 Ploché (flat) jazyky

Stráže jsou tvořeny vestavěnými predikáty. Vstupní unifikace probíhá jako obvyklá unifikace, avšak je-li v průběhu unifikace nutno dosadit hodnotu vstupní proměnné, výpočet je pozastaven.