

Učební texty k státní bakalářské zkoušce
Obecná informatika
Programovací jazyky

študenti MFF

15. augusta 2008

6 Programovací jazyky

Požadavky

- Principy implementace procedurálních programovacích jazyků, oddělený překlad, sestavení.
- Objektově orientované programování.
- Neprocedurální programování, logické programování.
- Generické programování.

6.1 Principy implementace procedurálních programovacích jazyků, oddělený překlad, sestavení

TODO: všechno

6.2 Objektově orientované programování

TODO: všechno

6.3 Neprocedurální programování, logické programování

Neprocedurální programování

Deklarativní programování je postaveno na paradigmatu, podle něhož je program založen na tom, co se počítá a ne jak se to počítá. Je zde deklarován vstup a výstup a celý program je chápán jako funkce vyhodnocující vstupy podávající jediný výstup. Například i webovské stránky jsou deklarativní protože popisují, jak by stránka měla vypadat – titulek, font, text a obrázky – ale nepopisují, jak konkrétně zobrazit stránky na obrazovce.

Logické programování a *funkcionální programování* jsou poddruhy deklarativního programování. Logické programování využívá programování založené na vyhodnocování vzorů - tvrzení a cílů. Klasickým zástupcem jazyka pro podporu tohoto stylu je Prolog.

Tento přístup patří pod deklarativní programování stejně jako funkcionální programování, neboť deklaruje, co je vstupem a co výstupem, a nezabývá se jak výpočet probíhá. Naopak program jako posloupnost příkazů je paradigma imperativní.

Funkcionální programování patří mezi deklarativní programovací principy.

Alonzo Church vytvořil formální výpočtový model nazvaný λ -kalkul. Tento model slouží jako základ pro funkcionální jazyky. Funkcionální jazyky dělíme na:

- typované - Haskell
- netypované - Lisp, Scheme

Výpočtem funkcionálního programu je posloupnost vzájemně ekvivalentních výrazů, které se postupně zjednodušují. Výsledkem výpočtu je výraz v normální formě, tedy dále nezjednodušitelný. Program je chápán jako jedna funkce obsahující vstupní parametry mající jediný výstup. Tato funkce pak může být dále rozložitelná na podfunkce.

Prolog

Prolog je logický programovací jazyk. Název Prolog pochází z francouzského *programmation en logique* („logické programování“). Byl vytvořen Alainem Colmerauerem v roce 1972 jako pokus vytvořit programovací jazyk, který by umožňoval vyjadřování v logice místo psaní počítačových instrukcí. Prolog patří mezi tzv. deklarativní programovací jazyky, ve kterých programátor popisuje pouze cíl výpočtu, přičemž přesný postup, jakým se k výsledku program dostane, je ponechán na libovůli systému.

Prolog je využíván především v oboru umělé inteligence a v počítačové lingvistice (obzvláště zpracování přirozeného jazyka, pro nějž byl původně navržen). Syntaxe jazyka je velice jednoduchá a snadno použitelná právě proto, že byl původně určen pro počítačově nepřiliš gramotné lingvisty.

Prolog je založen na *predikátové logice prvního řádu* (konkrétně se omezuje na Hornovy klauzule). Běh programu je pak představován aplikací dokazovacích technik na zadané klauzule. Základními využívanými přístupy jsou *unifikace*, *rekurze* a *backtracking*.

Interpret Prologu se snaží nalézt nejobecnější substituci, která splní daný cíl - tzn. nesubstituuje zbytečně, pokud nemusí (použití interních proměnných – 123 atd.). Za dvě proměnné může být substituována jedna interní proměnná (např. při hledání svíslé úsečky – konstantní X souřadnice) – tomu se říká *unifikace* proměnných. Pro proměnnou, jejíž hodnota může být libovolná, se v prologu užívá znak „-“.

Datové typy v prologu se nazývají *termy*. Základním datovým typem jsou *atomy* (začínají malým písmenem, nebo se skládají ze speciálních znaků (+ - * / ... , nebo jsou to znakové řetězce ('text')). Dále „jsou“ v prologu čísla (v komerčních implementacích i reálná), proměnné (velké písmeno) a struktury (definované rekurzivně - pomocí funktoru dané arity a příslušným počtem termů, které jsou jeho argumenty – `okamzik(datum(1,1,1999),cas(10,10))`). Posledním typem proměnných jsou seznamy, které jsou probírány později.

Základní principy:

Programování v Prologu se výrazně liší od programování v běžných procedurálních jazycích jako např. C. Program v prologu je databáze faktů a pravidel (dohromady se faktům a pravidlům paradoxně říká *procedury*), nad kterými je možno klást dotazy formou tvrzení, u kterých Prolog zhodnocuje jejich pravdivost (dokazatelnost z údajů obsažených v databázi).

Například lze do databáze uložit fakt, že Monika je dívka:

```
dívka(monika).
```

Poté lze dokazatelnost tohoto faktu prověřit otázkou, na kterou Prolog odpoví `yes` (ano):

```
?- dívka(monika).  
yes.
```

Také se lze zeptat na všechny objekty, o kterých je známo, že jsou dívky (středníkem požadujeme další výsledky):

```
?- dívka(X).
   X = monika;
   no.
```

Pravidla (závislosti) se zapisují pomocí implikací, např.

```
syn(A,B) :- rodič(B,A), muž(A).
```

Tedy: pokud B je rodičem A a zároveň je A muž, pak A je synem B. První části pravidla (tj. důsledku) se říká hlava a všemu co následuje za symbolem :- (tedy podmínkám, nutným pro splnění hlavy) se říká tělo. Podmínky ke splnění mohou být odděleny buď čárkou (pak jde o konjunkci, musejí být splněny všechny), nebo středníkem (disjunkce), přičemž čárky mají větší prioritu.

Příklad:

Typickou ukázkou základů programování v Prologu jsou rodinné vztahy.

```
sourozenec(X,Y) :- rodič(Z,X), rodič(Z,Y).
rodič(X,Y) :- otec(X,Y).
rodič(X,Y) :- matka(X,Y).
muž(X) :- otec(X,_).
žena(X) :- matka(X,_).
matka(marie,monika).
otec(jiří,monika).
otec(jiří,marek).
otec(michal,tomáš).
```

Prázdný seznam je označen atomem [], neprázdný se tvoří pomocí funktoru ' . ' (tečka) - . (Hlava, Tělo). V praxi se to (naštěstí ;-)) takhle složitě rekurzivně zapisovat nemusí, stačí napsat [a,b,c...], resp [Začátek | Tělo], kde začátek je výčet prvků (ne seznam) stojících na začátku definovaného seznamu, a tělo je (rekurzivně) seznam (např. [a,b,c|[]]).

Aritmetické výrazy se samy o sobě nevyhodnocují, dokud jim to někdo nepřikáže. Takže např. predikát `5*1 = 5` by selhal. Vyhodnocení se vynucuje pomocí operátoru `is` (pomocí `=` by došlo jen k unifikaci) - není to ale ekvivalent „=" z jiných jazyků. Tento operátor se musí použít na nějakou volnou proměnnou a aritmetický výraz, s jehož hodnotou bude tato proměnná dále svázaná (jako např. `X is 5*1, X=5` uspěje).

Důležitý je i *operátor řezu* (značíme vykřičníkem). Tento predikát okamžitě uspěje, ale při tom zakáže backtrackování přes sebe zpět. (`prvek1(X, [X|L]) :- ! . prvek1(X, [_|L]) :- prvek1(X,L)`). - je-li prvek nalezen, je zakázán návrat = najde jen první výskyt prvku). Dále je důležitá negace (`not(P) :- P, !, fail`). `not(P)` - uspěje, pokud se nepodaří cíl P splnit). Řez tedy umožňuje ovlivňovat efektivitu prologovských programů, definovat vzájemně se vylučující použití jednotlivých klauzulí procedury, definovat negaci atd.

Haskell

Haskell je standardizovaný funkcionální programovací jazyk používající zkrácené vyhodnocování, pojmenovaný na počest logika Haskellu Curryho. Byl vytvořen v 80.

letech 20. století. Posledním polooficiálním standardem je Haskell 98, který definuje minimální a přenositelnou verzi jazyka využitelnou k výuce nebo jako základ dalších rozšíření. Jazyk se rychle vyvíjí, především díky svým implementacím Hugs a GHC (viz níže).

Haskell je jazyk dodržující *referenční transparentnost*. To, zjednodušeně řečeno, znamená, že tentýž (pod)výraz má na jakémkoliv místě v programu stejnou hodnotu. Mezi další výhody tohoto jazyka patří přísné *typování proměnných*, které programátorovi může usnadnit odhalování chyb v programu. Haskell plně podporuje práci se soubory i standardními vstupy a výstupy, která je ale poměrně složitá kvůli zachování referenční transparentnosti. Jako takový se Haskell hodí hlavně pro algoritmicky náročné úlohy minimalizující interakci s uživatelem.

Příklady:

Definice funkce faktoriálu:

```
fac 0 = 1
fac n = n * fac (n - 1)
```

Jiná definice faktoriálu (používá funkci `product` ze standardní knihovny Haskellu):

```
fac n = product [1..n]
```

Naivní implementace funkce vracující n-tý prvek Fibonacciho posloupnosti:

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

Elegantní zápis řadícího algoritmu quicksort:

```
qsort [] = []
qsort (pivot:tail) =
  qsort left ++ [pivot] ++ qsort right
  where
    left = [y | y <- tail, y < pivot]
    right = [y | y <- tail, y >= pivot]
```

TODO: popsat stráže (případy, otherwise), seznamy, řetězení, pattern matching u parametrů funkcí, lok. definice (where, let) – patří to sem?

Lisp

Lisp je funkcionální programovací jazyk s dlouhou historií. Jeho název je zkratka pro List processing (zpracování seznamů). Dnes se stále používá v oboru umělé inteligence. Nic ale nebrání ho použít i pro jiné účely. Používá ho například textový editor Emacs, GIMP či konstrukční program AutoCAD.

Další jazyky od něj odvozené jsou například Tcl, Smalltalk nebo Scheme.

Syntaxe: Nejzákladnějším zápisem v Lispu je seznam. Zapisujeme ho jako:

```
(1 2 "ahoj" 13.2)
```

Tento seznam obsahuje čtyři prvky:

- celé číslo 1
- celé číslo 2
- text „ahoj“
- reálné číslo 13,2

Jde tedy o uspořádanou čtveřici. Všimněte si, že závorky nefungují tak jako v matematice, ale pouze označují začátek a konec seznamu. Seznamy jsou v Lispu implementovány jako binární strom degenerovaný na jednosměrně vázaný seznam. Co se seznamem Lisp udělá, záleží na okolnostech.

Příkazy: Příkazy píšeme také jako seznam, první prvek seznamu je však název příkazu. Například sčítání provádíme příkazem +, což interpreteru zadáme takto:

```
(+ 1 2 3)
```

Interpreter odpoví 6.

Ukázka kódu: Program hello world lze zapsat několika způsoby. Nejjednodušší vypadá takto:

```
(format t "Hello, World!")
```

Funkce se v Lispu definují pomocí klíčového slova defun:

```
(defun hello ()  
  (format t "Hello, World!")  
)  
(hello)
```

Na prvních dvou řádcích je definice funkce hello, na třetím řádku je tato funkce svým jménem zavolána. Funkcím lze předávat i argumenty. V následujícím příkladu je ukázka funkce fact, která vypočítá faktoriál zadaného čísla:

```
(defun fact (n)  
  (if (= n 0)  
      1  
      (* n (fact (- n 1))))  
)
```

Pro výpočet faktoriálu čísla 6 předáme tuto hodnotu jako argument funkci fact:

```
(fact 6)
```

Návratovou hodnotou funkce bude hodnota 720.

Logické programování

TODO (není součástí otázek pro obor Programování)

6.4 Generické programování a knihovny

Základní myšlenkou, která se skrývá za pojmem generické programování, je rozdělení kódu programu na algoritmus a datové typy takovým způsobem, aby bylo možné zápis kódu algoritmu chápat jako obecný, bez ohledu na to, nad jakými datovými typy pracuje. Konkrétní kód algoritmu se z něj stává dosazením datového typu.

U kompilovaných jazyků dochází k rozvinutí kódu v době překladu. Typickým příkladem jazyka, který podporuje tuto formu generického programování, je jazyk C++. Mechanismem, který zde generické programování umožňuje, jsou takzvané šablony (templates).

Definice (*Generické programování*)

Generic programming is a style of computer programming where algorithms are written in an extended grammar and are made adaptable by specifying variable parts that are then somehow instantiated later by the compiler with respect to the base grammar. Specifically, the extended grammar raises a non-variable element or implicit construct in the base grammar to a variable or constant and allows generic code to be used, usually implementing common software patterns that are already expressible in the base language.

Poznámka (*Metaprogramování*)

It differs from normal programming in that it somehow invokes within the language a metaprogramming facility. Because it happens as an extension of the language, new semantics are introduced and the language is enriched in this process. It is closely related to metaprogramming, but does not involve the generation of source code (none, at least, that is visible to the user of the language). It is different from programming with macros as well, as the latter refers to textual search-and-replace and is not part of the grammar of the language but implemented by a pre-processor. One exception to this is the macro facility in Common Lisp, in which macros operate on parse trees rather than text.

Příklad

Třída parametrizovaná typem (kontejner) — As an example of the benefits of generic programming, when creating containers of objects it is common to write specific implementations for each datatype contained, even if the code is virtually identical except for different datatypes. Instead, a possible declaration using generic programming could be to define a template class (using the C++ idiom):

```
template<typename T>
class List
{
    /* class contents */
};
```

```
List<Animal> list_of_animals;
List<Car> list_of_cars;
```

Above T represents the type to be instantiated. The list generated is then treated as a list of whichever type is specified. These "containers-of-type-T", commonly

called generics, are a generic programming technique that allows the defining of a class that takes and contains different datatypes (not to be confused with polymorphism, which is the algorithmic usage of exchangeable sub-classes) as long as certain contracts such as subtypes and signature are kept. Although the example above is the most common use of generic programming, and some languages implement only this aspect of it, generic programming as a concept is not limited to generics.

Příklad

Typově nezávislá funkce — Another application is type-independent functions as in the Swap example below:

```
template<typename T>
void Swap(T & a, T & b) //"&" passes parameters by reference
{
    T temp = b;
    b = a;
    a = temp;
}

string hello = "world!", world = "Hello, ";
Swap( world, hello );
cout << hello << world << endl; //Output is "Hello, world!"
```

Použití v programovacích jazycích

The template construct of C++ used in the examples above is widely cited as the generic programming construct that popularized the notion among programmers and language designers and provides full support for all generic programming idioms. D also offers fully generic-capable templates based on the C++ precedent but with a simplified syntax. Java has provided generic programming facilities syntactically based on C++'s since the introduction of J2SE 5.0 and implements the generics, or "containers-of-type-T", subset of generic programming.

Knihovna

Knihovna (angl. library) je v programování funkční logický celek, který poskytuje služby pro programy. Většinou se jedná o sbírku procedur, funkcí a datových typů, či při objektově orientovaném přístupu o sadu tříd, uložených v jednom diskovém souboru.

Knihovna poskytuje aplikační programátorské rozhraní (zvané API), které poskytuje funkce této knihovny. Existuje mnoho knihoven pro různé účely, např. pro využívání služeb operačního systému, grafické funkce, řízení periférií, vědeckotechnické výpočty atp.

Typy knihoven: Z technického hlediska je možné rozdělit knihovny dle způsobu propojení s programem, který je bude využívat:

- statická knihovna (static linking library) - spojuje se s programem ve při překlada, většinou přípona souboru .lib
- dynamická knihovna (dynamic linking library) - spojuje se s programem až při spuštění programu, většinou spojení knihovny s programem zajišťuje operační systém

Každý typ knihovny má své výhody a nevýhody. Program spojený se statickou knihovnou je přenositelnější, není třeba zajišťovat dostupnost požadovaných dynamických knihoven. Programy spojené s dynamickou knihovnou jsou menší (ve spustitelném souboru se nachází jen výkonný kód programu), výhodou je možnost jednoduše zaměnit požadovanou dynamickou knihovnu za její novější verzi. Kód v dynamicky linkované knihovně je také za běhu sdílen mezi všemi aplikacemi které jej používají, což šetří operační paměť.

Typickou příponou souboru obsahujících dynamickou knihovnu je .dll v operačním systému Microsoft Windows a .so v různých Unixech a v Linuxu.

Příklady různých knihoven: standardní knihovna jazyka C, standardní knihovna šablon (Standard Template Library=STL) v jazyce C++, grafické knihovny jako DirectX, OpenGL, SDL apod., matematická knihovna LINPACK pro řešení soustav lineárních rovnic...