PGS

1 Charakteristiky programových paradigmat

Paradigma = souhrn způsobů formulace problémů, prostředků řešení, metodik zpracování, ...

- **Procedurální programování**
- **Objektově orientované programování**
- **Deklarativní programování**
  - Aplikativní
    - funkcionální
    - logické
  - Programování ohraničeními (constraint)
- **Vizuální programování**
- **Souběžné programování**
  - Paralelní
  - Distribuované

2 Logické programování a jazyk Prolog

Program je souborem *klauzulí* (*faktů* a *pravidel*). Fakt je relace – jméno a uspořádané argumenty. Pravidla jsou definována jako vztahy vyplývající z faktů nebo jiných pravidel (`syn(S,O):-otec(O,S),muz(S).`). Pokud je součástí pravidla totéž pravidlo, dochází k rekurzivnímu volání, zastavení při nalezení faktu místo pravidla – musí být přítomna ukončovací podmínka. Cíle v podobě otázek na fakta, která nejsou přímo v databázi – vyvození. Elementy programu (*termy*) = konstanty, proměnné, struktury. Konstanty mohou být čísla nebo atomy. Proměnné v klauzuli obecně kvantifikovány, platnost omezena na klauzuli, nepoužívané proměnné lze značit `_`. Struktury jsou tvořeny operátory a operandy (`ma(dum, okno)`, `2+2`) nebo seznamy. Seznamy mají hlavu a tělo, alespoň hlava musí obsahovat prvek, seznamy lze vnořovat. Vytvoření seznamu: `[X,Y]=[a,b,c]` ... `X=a`, `Y=[b,c]`. Funkce pro práci se seznamy: `member(X,S)`, `last(S,X)`, `delete(S0,S,X)`, `append(S1,S2,S)`. Dále Prolog obsahuje příkazy pro vstup a výstup řetězců a znaků na konzoli nebo do souboru, řídící a testovací predikáty, příkazy pro práci s databází faktů a pravidel.

Princip rezoluce:

$$a :- a_1, a_2, \ldots, a_n \ ; \ b :- b_1, b_2, \ldots, b_m \ ; \ b_i \equiv a \ \Rightarrow \ b :- b_1, \ldots, b_{i-1}, \underline{a_1, \ldots, a_n}, b_{i+1}, \ldots, b_m$$

Princip unifikace:
Porovná-li se volná proměnná s konstantou, naváže se na tuto konstantu; porovnají-li se dvě volné (neinstalované) proměnné, stanou se synonymy; porovná-li se volná proměnná s termem, naváže se na tento term; porovnají-li se termy, které nejsou volnými proměnnými, musí být pro úspěšné porovnání stejné. Dva termy jsou úspěšně porovnány, pokud jsou totožné nebo proměnné v nich lze naplnit tak, aby byly totožné. Nelze porovnávat proměnnou a strukturu obsahující tuto proměnnou.

Plnění cíle je podobné prohledávání grafu do hloubky – prohledává se databáze od začátku, při nesplnění návrat do předchozí klauzule a nový pokus. Návratu lze zabránit použitím predikátu řezu `!` – bez ovlivnění výsledku = *zelený řez*, s vynecháním některých řešení = *červený řez*.

3 Funkcionální programování a jazyk Lisp

Na rozdíl od imperativních jazycích založených na Turingově stroji, které pracují s příkazy, funkcionální jazyky jsou založené na Churchově *lambda kalkulu* a pracují s matematickými funkcemi. Program je tvořen funkcí, jeho výstupem je funkční hodnota. Základní konstrukcí je výraz – *čistý* (nemění stavový prostor) × *s vedlejším efektem*. Čisté výrazy lze vyhodnocovat v libovolném pořadí beze změny výsledku (*Churchova-Roserova vlastnost*), lze je vyhodnocovat paralelně a vyhodnocení nezpůsobuje vedlejší efekty, operandy a výsledky zřejmé ze zápisu a nahrazení podvýrazu jeho hodnotou je nezávislé na výrazu, ve kterém je provedeno (*referenční transparentnost*).

Princip řešení: program je tvořen výrazem E, ten je redukován pomocí přepisovacích pravidel, dokud už nelze dále redukovat – pak je to *normální forma výrazu* E a je výstupem programu.

Program se skládá z definic funkcí (algoritmu) a aplikace funkcí na vstupní data. Funkce popsány aparátem – λ kalkul, ve výrazech použita *prefixová notace*. Zápis funkce pak vypadá následovně: `( (lambda(x y)(+ x y)) 5 )`. Argumenty funkce mohou být vyhodnoceny před aplikací funkce (*eager evaluation*) nebo až v okamžiku použití (*lazy evaluation*).

Jazyk *Lisp* – několik interpretů (Commonlisp, Franclisp, Scheme, Maclisp). Používá se zejména na úlohy umělé inteligence, návrh VLSI, CAD. Program i data popsány seznamem. Data jsou tvořena *atomy* (čísla, znaky, řetězce, symboly, ...) a *seznamy* (atomů, seznamů). Interpret má úrovně – při chybě se přejde o úroveň výše, `Abort` se vrátí zpět. Elementární funkce: `CAR`, `CDR`, `CONS`, `ATOM`, `EQUAL`. Další funkce: `APPEND`, `LIST`, `EQ`, `NULL`, `NUMBERP`, `SYMBOLP`, `LISTP`, `AND`, `OR`, `NOT`, podmíněné větvení – `COND` a `IF`, přiřazení – `SET`, `SETQ` a `LET`, definice funkcí – `DEFUN`, I/O operace – `LOAD`, `OPEN`, `CLOSE`, `READ`, `PRINT`, `TERPRI` (nový řádek), `FORMAT`, řídící struktury (cykly) – `WHEN`, `LOOP`, `DO`, `DOLIST`, vyhodnocení – `EVAL`, `PROG`, `GO`, `RETURN`. *Funkcionály* – funkce, které přijímají jako argument funkci nebo vracejí funkci (např. podmínky, cykly). Vytvářejí programová schémata (high order functions). Při definici se pro popis funkce v argumentu používá příkaz `FUNCTION`.

4 Charakteristické vlastnosti objektově orientovaného programování

Softwarová hlediska:

- **Znovupoužitelnost komponent**
  - Rozšiřování
  - Omezování
  - Předefinování
  - Abstrakce
  - Polymorfizace
- **Modifikovatelnost komponent**
- **Nezávislost komponent**

Kategorie OOP jazyků:

- **Imperativní s přidanými OOP vlastnostmi**
  C++, ADA95, Object Pascal, Scheme
- **OOP jazyky se základem v imperativním jazyce**
  Java, C#, Eiffel
- **Čistě OOP jazyky**
  Smalltalk

Základní vlastnosti OOP:

- **Zapouzdření**
  ADT je nazýván *třídou* a je šablonou pro *objekty*. Data a procedury společně ve struktuře – objektu (instanci třídy). Přístup k datům přes pevné rozhraní (ukrývání dat) $\Rightarrow$ nezávislost na implementaci. Komunikace objektů pomocí zpráv (adresát + selektor).
- **Dědičnost**
  Podtřída, supertřída, lze ukrývat vnořené entity, proměnné & metody instance $\times$ třídy = závislosti příbuzných tříd, komplikace. Třída musí obsahovat jen jednu logickou entitu.
- **Polymorfismus**
  Jedna zpráva může mít různé významy. Třída může předefinovat metody rodiče = dynamická vazba mezi metodou a objektem, zdržuje. Abstraktní $\times$ finální metody. Polymorfismus vede k dynamické typové kontrole – časově náročné.

5 Základní objektové konstrukce jazyků Java, C++, Objektový Pascal

- **Java**
  Všechna data kromě základních datových typů jsou objekty. Objekty dynamické, přístupné přes referenční proměnnou. Jednoduchá dědičnost. Dynamická vazba metod. Zapouzdření pomocí rozhraní, tříd a kontejnerů (package).
  - Třída
    `class`, každá třída v separátním souboru `.class`. Hlavní program = metoda `main`. Instance klíčovým slovem `new` + konstruktor. Objekty na haldě, čistí ji garbage collector – volá metodu `finalize` objektů. Nejsou třeba destruktory. Třída uvnitř jiné třídy = vnořená, nestatická = vnitřní. Vnořené třídy – přístup ke všem proměnným a metodám uzavírající třídy, uzavírající třída – přístup přes instanci.
  - Řízení přístupu
    Čtyři úrovně – `public` (přístup odkudkoliv), `protected` (přístup ze všech podtříd), implicitní (přístup z podtříd téhož balíku), `private` (přístup pouze z téže třídy).
  - Přetěžování
    Lze definovat více konstruktorů nebo metod stejného jména, musí se lišit argumenty.
  - Statické entity
    Lze je použít kdekoliv, společné pro všechny instance třídy, statické metody mohou přistupovat pouze ke statickým entitám. Přístup přes *název třídy . název entity* nebo *název objektu . název entity. Statický blok* – umožňuje inicializaci třídy, provádí se před jejím prvním použitím, deklarace pomocí `static{ příkazy }`.
  - Dědičnost
    Přístup jen k neprivátním entitám. Kontruktor – `super()`. Odkaz na nadtřídu – `super` (její `this`) – přístup k metodám nadtřídy při překrytí. Nelze zeslabit přístupová práva při překrytí. Místo dědičnosti případně kompozice. Odkaz referenční proměnnou nadtřídy, přístup jen k entitám nadtřídy, u překrytých – verze z nadtřídy. Třída `Object` - univerzální nadtřída. Vícenásobná dědičnost nahrazena rozhraními.
  - Abstraktní třída
  - Finální entity
    Metody nelze přakrýt, proměnné nelze měnit (poprvé lze přiřadit hodnotu), třídy nelze dědit, abstract + final nelze.
  - Balíky
    Sdružují třídy do větších celků (např. aplikace, třídy pro I/O operace). Lze je vnořovat pomocí tečky. Každý balík ve vlastním adresáři. Třída může být zařazena do kontejneru (balíku) deklarací `package`.
- **C++**
  - Třída
    Tři možnosti – `class`, `struct`, `union`. Instance – statická (netvoří se pomocí new, přístup k entitám přes tečku), dynamická (přes ukazatel, přístup k entitám přes `->`) – buď v haldě (pomocí new) nebo v zásobníku (při rekurzi). Těla metod buď uvnitř třídy (inlined) nebo pouze hlavičky uvnitř a těla mimo. Destruktor `~Třída`. C++ nemá GC, vymazání instance explicitně pomocí `delete` (mohou zůstat slepé odkazy!), jinak na konci metody (lokální) nebo programu (globální proměnné).
  - Řízení přístupu
    Public, protected, private. Implicitní – v class private, ve struct public, v union public bez možnosti změny. Práva lze při dědění zeslabit, nelze zesílit.
  - Dědičnost
    Násobná dědičnost pomocí `friend` – problém volání duplicitních metod – buď přes `Třída::metoda` nebo pomocí deklarace `virtual`.
  - Namespace
    Umožňuje omezit rozsah platnosti jmen tříd. Může být více tříd se stejným názvem, použije se ta, která patří do nastaveného jmenného prostoru. Podobný mechanismus jako balíky v Javě.
- **Object Pascal**
  - Třída
    Deklarována jako typ `class` v sekci `interface`, v deklaraci definovány proměnné, vlastnosti (property) a hlavičky metod. Těla metod definována zvlášť v sekci `implementation`. Metody označeny buď `procedure` (nemají výstupní hodnotu, jako void) nebo `function`, konstruktor a destruktor mohou mít libovolný název, značí se slovem `constructor` resp. `destructor`. Instance pomocí new. Není GC, mechanismus jako u C++, manuální odstranění instance pomocí `free`.

– Řízení přístupu
Pomocí private, protected, public, navíc u vlastností lze definovat přístup pouze pro čtení, pouze pro zápis nebo obecný.

– Dědičnost
Jednoduchá, pomocí argumentu `class` při deklaraci třídy. Obyčejná třída může být deklarována jako `class` nebo `class(TObject)`, zděděná třída pak `class(Rodič)`.

– Přetížení
Přetížené metody a konstruktory je nutné explicitně označit klíčovým slovem `overload`.

– Unity
Jistá forma kontejneru pro třídy. Unita představuje jeden soubor, může obsahovat i globální proměnné, a metody – imperativní základ. Zpřístupnění typů (i tříd), metod a proměnných z jiné unity pomocí deklarace `uses`.

6 Dědičnost - formy a problémy

- **Statická (brzká) vazba**
Je-li volána metoda z nadtřídy (není překrytá v podtřídě) a tato volá další metodu, která je překrytá v podtřídě, zavolá se metoda nadtřídy, nikoliv překrytá verze. Při všech voláních se metoda hledá pouze v aktuálně volající třídě a jejích předcích.

- **Dynamická (pozdní) vazba**
Volání metody v metodě nadtřídy zavolá verzi metody nejnižší třídy v hierarchii. Při všech voláních se hledá od třídy, která způsobila počáteční volání. Realizace pomocí virtuálních metod.

  Při překladu pro každou třídu vytvořena tabulka metod s pozdní vazbou (VMT, Virtual Method Table). Konstruktor pak přiřadí instanci odkaz na příslušnou VMT. Volání virtuální metody probíhá nepřímým skokem přes VMT.

9 Prostředky dekompozice a abstrakce v programovacích jazycích

Možnosti strukturování:

- **Podprogramy** ... původní forma abstrakce – abstrakce výpočtů
- **Moduly** ... kontejnery souvisejících dat a podprogramů
- **Kompilační jednotky** ... kolekce podprogramů a dat přeložitelných bez nutnosti překládat zbytek programu

Zapouzdření = seskupení souvisejících podprogramů, které lze samostatně překládat. ADT = zapouzdření datového typu a podprogramů pro operace s ním, lze tvořit proměnné (instance) – objekty, skutečná reprezentace skryta. Jazykové požadavky na ADT: syntaktická konstrukce pro zapouzdření definic typu, prostředek pro zpřístupnění podprogramů při zakrytí definice, primitivní operace (přiřazení, porovnání) v jazyce, ostatní definuje ADT sám.

- **ADA** ... package – interface + implementace; ukrytí typu – is private
- **C++** ... class, struct, union; ukrývání – private, proected, public; konstruktor & destruktor; namespace
- **Java** ... class; ukrývání; konstruktor; package
- **C#** ... class; ukrývání jako Java + internal a protected internal; balík = assembly (DLL knihovna)

11 Zpracování výjimečných situací

Pokud jazyk neobsahuje zpracování výjimečných situací, předává se řízení operačnímu systému – výpis chybové zprávy a ukončení programu. Výjimka je neobvyklá situace (ne nutně chybová) zaznamenaná softwarem nebo hardwarem a vyžadující speciální zpracování. Může být vestavěná nebo uživatelská. Je vyvolána, pokud nastane asociovaná událost a programová jednotka pro její zpracování se nazývá ovladač výjimky. Poprvé použity v jazyce PL1, implementace však byla nejednotná a nepřehledná.

Propagace (šíření) výjimky: není-li zpracování výjimky obsaženo v programové jednotce, kde vznikla, je předána ke zpracování do nadřazené jednotky (tou může být i operační systém).

- **C++**
  Příkazy `try`, `catch`, `throw`. Catch se volá s parametrem typu výjimky, prázdný parametr = zpracování všech výjimek. Lze zřetězit více catch za jedním try. Vyvolání pouze explicitní pomocí throw s parametrem obsahujícím data výjimky. Neexistuje žádná šablona pro výjimky, pro předání dat lze použít libovolný typ. Throw bez parametru pro předání výjimky o úroveň výš.

- **Java**
  Příkazy `try`, `catch`, `finally`, `throw`, `throws`. Založeno na objektovém principu, nadtřída výjimek `Throwable`, dvě podtřídy – `Error` pro výjimky interpretu (přetečení haldy, zásobníku apod.) a `Exception` pro výjimky zpracovatelné uživatelsky. Potomky Exception jsou `RuntimeException` (programové chyby) a `IOException` (chyby vstupně-výstupní). Zpracování v try-catch bloku jako C++, parametrem catch proměnná potomka třídy Throwable – určuje, který typ výjimky blok zpracovává. Lze vytvořit uživatelskou výjimku odvozením od třídy Exception nebo její podtřídy. Manuální vyvolání pomocí throw – buď s instancí výjimky, nebo s new pro vytvoření nové instance. Metoda s příkazem potenciálně vyvolávajícím výjimku, která neobsahuje její zpracování, musí být označena pomocí throws a typu předávané výjimky (případně více typů). Do throws není třeba psát výjimky odvozené z RuntimeException a Error. Konstrukce finally umožňuje provést příkazy bez ohledu na to, jestli se stala výjimka a jestli byla zpracována nebo předána, vykoná se také v případě, že je v try bloku zavolán příkaz `return`.

12 Problémy paralelního zpracování

- **Rychlostní závislost**
  Pracují-li dvě operace nad jednou datovou jednotkou, závisí výsledek jejich operací na pořadí, ve kterém jsou vykonány příkazy = stejná konstrukce může pokaždé vyústit v jiný výsledek. Je nutné zavést správu sdíleného přístupu k datům. Řešení pomocí semaforů – jednoduché, ale možnost deadlocku. Další možnost – monitory (v Javě objekty se synchronized metodami).

- **Zablokování** (deadlock)
  Přistupují-li dvě metody současně ke dvěma sdíleným prostředkům, může při nesprávném použití dojít k zablokování. Jedna metoda si rezervuje první prostředek, druhá mezitím druhý. Pokud se nyní první metoda pokusí přistoupit ke druhému prostředku, bude pozastavena, dokud druhá metoda tento neuvolní. Přístup druhé metody k prvnímu prostředku způsobí zablokování, kdy obě metody budou pozastaveny čekáním na sebe navzájem. Řešením tohoto problému je pevné uspořádání přístupu, Bankéřův algoritmus nebo použití monitorů.

13 Programové prostředky pro paralelní výpočty

Paralelní procesy jsou v programovacích jazycích obvykle označovány jako *vlákna*, případně *úkoly* (ADA). Na rozdíl od podprogramů mohou být implicitně spouštěny, spouštějící jednotka není pozastavena a po jejich skončení se nevrací řízení do místo, odkud byly spuštěny. Mohou být nekomunikující, komunikující (producent/konzument) nebo soutěžící. Komunikace může probíhat přes sdílené proměnné, parametry nebo zasílání zpráv. Při synchronizaci musí dříve skončený proces čekat na druhý. Soutěžení probíhá sdílením prostředku pomocí vzájemného vyloučení. Části programu pracující ve vzájemném vyloučení = *kritické sekce*. Vyloučení se zajišťuje pomocí semaforů, monitorů nebo zasílání zpráv. Semafor = čítač + fronta, použitelný pro soutěžící i spolupracující úkoly, operace zaber a uvolni, může nastat deadlock. Monitor obsahuje sdílená data a procedury pro práci s nimi, vždy pouze jeden proces se může nacházet v monitoru, ostatní čekají ve frontě. Zasílání zpráv – proces pošle žádost o přístup ke sdíleným datům a pozastaví se, druhý proces akceptuje zprávu, čímž umožní prvnímu vstup do kritické sekce.

V Javě jsou paralelní procesy realizovány pomocí vláken. Vlákno je třída s metodou `run`, která se při spuštění vlákna vyvolá pomocí metody `start`. Vlákno lze buď odvodit od třídy `Thread`, nebo implementovat rozhraní `Runnable`. Stav vlákna lze zjistit metodou `isAlive`. Vlákno může čekat na skončení jiného zavoláním jeho metody `join`, případně lze druhé vlákno předtím předčasně probudit pomocí metody `interrupt`. Každému vláknu lze přiřadit prioritu, s jakou bude soutěžit o procesor, příkazem `setPriority`, implicitně vlákno dědí prioritu procesu, který ho vytvořil. Přístup ke sdíleným prostředkům pomocí `synchronized` metod nebo příkazů `wait` a `notify` příp. `notifyAll`. Vlákno se může vzdát svého procesorového času pomocí `yield` nebo pozastavit voláním `sleep`.

14 Klasické výrazové prostředky procedurálních jazyků - typový systém, příkazy, struktura programu

- **Statická vazba jména s typem**
  Navázání před dobou výpočtu a pak už se nemění, deklarace typu explicitní nebo implicitní. Statická typová kontrola.
- **Dynamická vazba jména s typem**
  Nastává až během výpočtu a může při vykonávání se měnit. Specifikace typu přiřazením. Flexibilnější, ale pomalé (dynamická typová kontrola) a horší detekce chyb.

Typová kontrola – kontrola kompatibilních typů operátorů (buď je operátor podporuje, nebo jsou implicitně převeditelné na typ, který operátor podporuje). Silný typový systém = typová kontrola odhalí všechny typové chyby (Java a ADA téměř ano, ostatní ne). Problémy při konverzi – zaokrouhlování, ořezání, ztráta přesnosti. Typová kompatibilita – jmenná = proměnné jsou stejných jmen typů, strukturální = typy proměnných mají stejnou strukturu (obtížně implementovatelné).

Typ = datová kolekce a operace nad ní.

- **Primitivní typy** ... nevyužívají jiné typy × složené typy
- **Ordinální typy** ... zobrazitelné do celých čísel (int, char, boolean, výčtové typy, intervalové typy, ...)
- **Řetězce** ... pole znaků (Pascal, C), třída (Java String), primitivní typ (ADA, Basic); délka – statická (ADA, Java String), dynamická limitovaná (C, C++), dynamická neomezená (Snobol, Perl)
- **Pole** ... celočíselné indexy (Java, C), ordinální indexy (Pascal, ADA); statická , statická v zásobníku, dynamická v zásobníku, dynamická na haldě
- **Vícerozměrná pole** ... více indexů (Pascal, C, ADA) × jeden index, prvky jsou pole (Java)
- **Asociativní pole** ... (Perl, PHP) – kolekce dvojic (klíč, hodnota)
- **Záznam** ... heterogenní kompozice datových prvků, přístup k prvkům přes tečku, struct v C, record v Pascalu
- **Union** ... proměnné mohou obsahovat v různých okamžicích různé datové typy, union v C, case record v Pascalu
- **Set** ... kolekce prvků libovolného ordinálního typu
- **Ukazatel** ... dangling pointer = ukazuje do zrušené paměti

- **Logické výrazy** ... zkrácené vyhodnocování, využití v přiřazení (?:) a v řídících příkazech
- **Přepínač** ... switch (C, C++, Java), case (Pascal, ADA)
- **Cykly** ... loop – end loop, while, repeat – until × do – while, for
- **Skok** ... rozporný, znemožňuje formální verifikaci programu

Podprogram – jeden vstupní bod, volající pozastaven do skončení, potom návrat za místo volání. Lokální data podprogramu v *aktivačním záznamu* (v zásobníku). Přetěžování podprogramů a operátorů. Generické podprogramy. Předávání parametrů:

- **Hodnotou** ... číslo apod.
- **Výsledkem** ... výsledek volání podprogramu
- **Hodnotou i výsledkem**
- **Ukazatelem**
- **Jménem** ... řetězec se jménem, neefektivní
- **Vícerozměrným polem** ... někdy potřeba předat rozměry
- **Ukazatelem na funkci** ... C, C++