IPP - příprava na (půl)semestrální písemku

1 První přednáška

Programování (proces tvorby programu) = taková činnost, která jistý postup (algoritmus) převádí na posloupnost elementárních úkonů nějakého stroje, počítače. Přitom dochází k uložení tohoto postupu tak, aby jej stroj mohl opakovat periodicky,

Programátor je ten, kdo proces tvorby programu realizuje.

Tvorba programu (programování není jednoduchá, její typické vlastnosti patří:

- Vysoká náročnost na čas a zdroje = vyžaduje přítomnost lidí, materiální zabezpečení po dobu vytváření a jeho života.
- Znalost programovacího jazyka jako taková není dostačující = nezaručuje, že nějaký postup správně naprogramuju
- Specifikace požadavků na cílový produkt není bezchybná = specifikace nebývá tak detailní, aby podle ní šly řešit všechny problémy. Často je měněna.
- Přeceňování/podceňování schopností/zdrojů = podceňování složitosti projektů, přeceňování schopností programátorů
- Ostré termíny = požadavky na realizaci jsou vázány ostrými termíny, ty jsou umocněny sklonem programátorů přeceňovat své schopnosti

Programovací jazyk je prostředník mezi běžnou řečí a posloupností typicky binárních číslic.

Programovací jazyk je konečná množina příkazů, která má specifikovanou syntaktickou strukturu a pevně vymezenou sémantiku.

Nevýhody přirozeného jazyka:

- Analýza přirozeného jazyka je příliš náročná, aby byla převoditelná do binární formy
- Nejednoznačnost, slovní přepisy příliš dlouhé

Nevýhody binárního jazyka:

- Příliš náročný na zapamatování

- Nevhodný pro složité problémy = kódování složitých problémů by bylo extrémně náročné
- Nevhodný pro rychlou a efektivní tvorbu programů = znovu využití kódu, udržovatelnost, rychlá orientace…

Počítačový program je zápis v programovacím jazyce, který je abstrakcí reality.

- Implementuje abstraktní model = myšlenkový postup aplikovatelný v realitě je abstrahován a přizpůsoben možnostem PC
- Abstrahuje model PC/procesoru = program zastiňuje konkrétní podobu výpočtu a jeho realizaci na cílové platformě

Programovací jazyky jsou specifikovány **syntaxí** a **sémantikou**.

Syntaxe jazyka definuje strukturu programu, tj. to, jakým způsobem je dovoleno jednotlivé konstrukce řadit za sebe.

Pomocí ní se vysvětluje i lexikální stavba jazyka. Formálně lze syntaxi jazyka popsat formálními gramatikami. Proto jsou dnes používány regulární či bezkontextové gramatiky. Někdy však existují kontextová omezení, která jsou definována slovním doprovodem.

Pro definici syntaxe pro účely popisu jazyka pro programátory se používá slovní popis (málo), syntaktické grafy, BNF, EBNF nebo gramatiky.

Sémantika je popis/definice významu jednotlivých syntaktických konstrukcí, způsobu jejich vyhodnocení, zpracování.

- **Statická** = popisuje vlastnosti, které mohou být studovány a ověřovány v době analýzy/překladu programu, např. typová kompatibilita, existence proměnných apod.
- **Dynamická** = popisuje vlastnosti, jejichž splnění lze ověřit až v době běhu programu, např. velikost indexu pole daného výrazem, velikost výsledku apod.

Pro popis sémantiky se na uživatelské úrovni nepoužívají formalismy, ale slovní spojení nebo ukázka na příkladech.

Jako formalismu lze použít např.:

- **Axiomatickou sémantiku** = pro každou syntaktickou konstrukci definuje množinu axiomů, které musí být splněny, aby byla konstrukce platná
- **Operační sémantiku** = definuje sémantiku chování programu jako posloupnost přechodů mezi danými stavy
- **Denotační sémantiku** = program je definován jako matematická funkce, která zobrazuje vstupy na výstupy

Deklarace úplně vymezuje atributy dané entity. Může být explicitní i implicitní.

Definice úplně vymezuje atributy dané entity a dále u proměnných způsob alokace paměti a u funkcí/procedur navíc tělo funkce.

Vazba spojuje vytvořenou entitu s jejími vlastnostmi/atributy.

Může vznikat v **různých časech**:

- Během definice samotného jazyka = např. množina operací nad daným typem
- Během implementace programu = přiřazení typu proměnné
- Během překladu programu = inicializační hodnota proměnné
- Během spojování přeložených modulů (linking) = určení adresy objektu cizího modulu pro přístup k němu
- Během spouštění programu = např. navázání na standardní vstup
- V době běhu programu = např. přiřazení adresy lokální proměnné

Vazby mohou být **statické** i **dynamické**. Statické jsou po vytvoření **neměnné** a dynamické jsou za běhu programu **měněny**.

Řešený příklad

Zadání: Vyjmenujte alespoň 5 druhů vazeb, které vznikají mezi entitami a vlastnostmi/atributy v tomto příkladu:

int count; ... count = count + 5;

Řešení: V tomto jednoduchém příkladu je možné pozorovat například tyto vazby:

- množina typů, jichž může proměnná count nabývat;
- přiřazení typu proměnné count;
- množina hodnot, jichž může proměnná count nabývat;

- hodnota, kterou proměnná count nabývá;
- množina významů operátoru +;
- význam operátoru + v tomto případě;
- interní reprezentace literálu 5

U každé entity můžeme studovat různé vlastnosti. Zaměříme se na **proměnnou**, jejíž role je jednoduchá.

Studované vlastnosti proměnné:

- jméno = Důležitá je **efektivní** délka, která udává počet znaků, který je skutečně zpracován. **Znaky** tvořící jméno jsou také pevně vymezené. Jsou to písmena anglické abecedy, číslice a některé speciální znaky. Některé jazyky jsou **case sensitive** (C) a některé ne (Pascal). U současných jazyků jsou množiny zakázaných identifikátorů, které jsou použity pro označení příkazů, označujeme je **klíčová slova**. U některých jazyků jsou **rezervovaná slova**.
- adresa a umístění/lokace v paměti = Máme přiřazovací příkaz L=R, kde L určuje umístění a adresu v rámci tohoto umístění, označuje se jako **L-hodnota**. R určuje hodnotu, kterou obsahuje, označuje se jako **R-hodnota**. Jedno jméno proměnné může být spojováno s více umístěními a adresami (lokální proměnné ve 2 funkcích mají stejné jméno. Ale mohou ležet na jiných místech). Nebo lokální proměnná v rekurzivním podprogramu. Globální proměnná může být umístěna do jiné paměti než globální konstantní proměnná. Naproti tomu jedna adresa může být spojována s více různými jmény. Např. ukazatel na proměnnou a proměnná nebo sdílení v důsledku optimalizací. **Vazba** mezi umístěním a adresou může být **statická** (u statických proměnných nebo globálních. Nabývá konstanty) nebo **dynamická** (proměnné umístěny na zásobníku nebo na haldě, dealokace je potom implicitní nebo explicitní).
- hodnoty, jichž může nabývat

- typ = určuje množinu hodnot, které může proměnná nabývat, dále množinu operací, které lze na ni aplikovat. Vazba mezi typem a proměnnou je často statická a vyžaduje explicitní deklaraci či definici. Implicitní definice u jazyka Fortran. Dynamická je u skriptovacích jazyků a interpretů, kde dochází dynamické typové kontrole Příklad z Pearlu: if ($p) $x = 10; else $x = "abc"; echo $x+1;)). Jazyky z hlediska typovosti rozdělujeme na **beztypové**, **netypované** a **typované** (přořazení typů explicitní nebo automaticky odvozené).
- **doba života** proměnné je časový interval, po který je pro danou proměnnou alokovaná paměť. Alokace může probíhat **staticky** (před během programu) nebo **dynamicky** (automatická = lokální proměnné, příkazem pro alokaci).
- **rozsah platnosti** proměnné určuje tu část programu, kdy je možné s proměnnou pracovat. Rozsah platnosti je spojen s **viditelností** proměnné. I když je proměnná platná, tak může být skryta jinou proměnnou stejného jména (lokální zakryje globální). Většina jazyků u rozsahu platnosti uplatňuje **statickou vazbu** (dána strukturou programu).

2 Druhá přednáška

2.1 Objektová orientace (OO)

V modulech šlo přistoupit k datům nezávisle na operacích a naopak, u objektů lze k nim přistupovat přes nějaký protokol. Jsou uzavřeny v abstrakci objektu.

Analýza modelování, návrh, implementace

Data = atributy (datové položky)

Operace = metody (členské funkce)

Naším hlavní úkolem je vytvořit **Objektový systém** = kolekce komunikujících objektů, které spolu interagují při spolupráci na řešení daného problému.

2.2 Modely OOP

Model OOP/OOJ

- Pravidla pro tvorbu objektového systému
- Jak dělat: konstrukce objektu, propojování objektů, model výpočtu (co se bude dít, když se pošle zpráva, když se invokuje metoda)

Existují 2 modely pro ty OOJ.

Statické modely (překládané) = staticky typované, zaměřené na psaní OO zdrojového textu, využívají statické VMT (tabulka virtuálních metod)

Dynamické modely (využívají VM = virtual machine, nebo jsou přímo interpretované) = požívají dynamické datové typy (netypované), jednoduchost, reflexivita

Každý OOJ má svůj model.

2.3 Model výpočtu

Model výpočtu v čistě OOJ vždy obsahuje aspoň tyto 2 sémantické konstrukce: **přiřazení** a **zasílání zprávy**.

Objekt pojmenujeme neboli přiřadíme objekt do proměnné, abychom s tím objektem mohli pracovat. Místo objektu můžeme pracovat s referencí nebo ukazatelem na objekt. Dále potřebujeme konstrukt objektu (jak objekt vypadá) a poslední je mechanismus, jak mezi sebou objekty komunikují (zaslání zprávy). Což je zasílání zprávy. Kromě identifikátoru může obsahovat parametry. Způsob reakce objektu na zprávu závisí na něm, což je koncept zapouzdření a polymorfismu. Nejčastěji je reakce na zprávy pomocí stejně pojmenované metody. Zprávy 2 typů: typu invokuj m1 (najdi metodu m1 a invokuj ji = proveď/spusť) a modifikuj m1 (na nějaké nové tělo). Objekt buď najde metodu u sebe nebo ji najde v objektu, který je v rodičovském vztahu s tímto objektem nebo ji nenajde vůbec a vyvolá výjimku (chybu).

2.4 Formální báze OOP

Matematický popis vlastností jazyků
Příklady:

- UML - používá se pro OO analýza, OO návrh, OO modelování
- Sigma-kalkul - popisuje minimální OO model

- OCaml - imperativní OOJ s formální bází

2.5 Popis syntaxe a sémantiky složitějších OOJ

Jak se popisuje formalismus u OOJ (formální základ lze rozdělit na syntaxi a sémantiku)

Syntaxe:

- Pro popis syntaxe se používá kombinace (E)BNF ((rozsířená) Backus naurova forma = vyjádření bezkontextové gramatiky, která se používá pro popis formálních jazyků), bezkontextových gramatik, slovního výkladu doplněného o příklady.

Sémantika:

- Složité ji popsat formálně, takže se používá kombinace slovního popisu, příkladů, diagramů.
- Zvýšení složitosti (protože u OO se pracuje navíc s objekty a jejich protokoly) = pár nových klíčových slov ➔ mnoho nových významových kombinací
- Víceúrovňový popis (od jednoduššího ke složitějšímu)

2.6 OO Paradigma - Koncepty (obecný popis pro třídní i beztřídní OOJ)

- Abstrakce/Objekty
- Zapouzdření
- Mnohotvárnost
- Dědičnost

To vše dohromady se snaží nám umožnit co nejefektivněji implementovat generická řešení (takové řešení, které je flexibilní a pohodlně upravitelné pro podobný problém) zadaného problému.

2.7 Objekt

(atributy + jejich hodnoty + metody, které mohou být na ně aplikovány, komunikace skrz protokol zprávou, výstup je reakce, třeba výpočet)

Je to **autonomní** (může si s obdrženou zprávou dělat co chce, omezeno sémantikou) **výpočetně úplná** (každá metoda může v sobě obsahovat kód, co dokáže cokoliv) entita. Každý objekt má **identitu**, je **nezávislá** na atributech. Používá se pro zjištění, zda 2 objekty jsou totožné (nebo jen shodné nebo ani to). Jsou základní jednotkou modularity i struktury v OO programu, umožňuje problém rozdělit na podčásti, čím se blíží realitě, díky jejich komunikaci problém řešit.

Datová povaha objektu:

- Proměnná obsahuje/odkazuje objekt
- Datový typ objektu (množina možných hodnot datových položek a množina operací nad nimi)

2.8 Abstrakce

Schopnost programu zjednodušit některé aspekty informací či vlastnosti objektů, se kterými pracuje.

Je to pohled na vybraný problém reálného světa a jeho počítačové řešení.

Vytváření abstrakce = skrývání detailů do tzv. černé skříňky, pro okolí definována jen svým protokolem.

Míra abstrakce = jak moc je vzdálená funkčnost černé skříňky od reality.

2.9 Zapouzdření

Uzavřenost vůči okolním objektům (modifikátory viditelnosti), uživatel nemůže měnit interní stav objektů libovolně, ale přes rozhraní (protokol - určuje, jak s ním mohou ostatní objekty komunikovat, co je uvnitř, je okolí skryto).

Přístup pouze přes veřejný protokol (message lookup):

- **Zaslání zprávy** = obsahuje 3 informace: **příjemce** (obecně nezná odesílatele, objektový model programu ano, ten to zařídí), **selektor** (jméno zprávy), **argumenty**
- Protokol příjemce rozhodne o reakce na zprávu:
  - **Invokace** odpovídající metody a navrácení výsledné hodnoty odesílateli
  - **Chyba** = nerozumí zprávě

Posílá-li objekt zprávu sám sobě, přistupuje přes **interní** protokol

- Zaslání zprávy
- Protokol příjemce rozhodne o reakcí na zprávu
  - **Přístup** k atributu (čtení nebo modifikace)
  - **Invokace** odpovídající metody a návrat výsledku
  - **Chyba** = nerozumí zprávě

Je potřeba proměnné self/this při posílání sama sobě. Odkazuje na příjemce zprávy. Říká se tomu self parametr.

2.10 Mnohotvárnost/Polymorfismus

Stejnou zprávu lze poslat různým objektům. A u různých objektů to vede na různou reakci (invokace různých metod).

- Některé OOJ toto omezují typovým systémem

Protokol umožňuje individuální reakce, protože kvůli zapouzdření neznám implementaci invokované metody (pro různé objekty se může lišit)

- Statické (v době překladu) určení reakce = brzká vazba (určení, jaká metoda se zavolá, určí se při překladu)
- Dynamické (za běhu) určení reakce = pozdní vazba (určení, jaká metoda se zavolá, určí se za běhu)
  - Některé OOJ umožňují druh vazby určit pro každou zprávu (v C++ implicitně brzká vazba, pozdní jen virtuální metody)
  - Např. v Javě, PHP 5 implicitní pozdní vazba (nelze změnit)

2.11 Dědičnost (obecně)

Aneb způsob, jak implementovat sdílené chování. Nové objekty mohou sdílet a rozšiřovat chování těch již existujících bez nutnosti nové implementace téhož chování.

Dědičnost = sdílení společných položek (od předků) + individuální položky (v potomcích)

- Primárně pro sdílení/znovupoužití metod

- Specializace zděděného objektu (individualizovaná část pro potomky, něco navíc):
    - Přidání nových položek
    - Modifikace metod

2.12 Vytváření nových objektů

3 přístupy:

1. Vytvoření **prázdného objektu** + **úprava** položek
2. Kopie (**klonování**) + **úprava** položek
    - **Prototyp** = klonovaný objekt
3. Vytvoření pomocí „šablony“ + **naplnění** atributů
    - Nutnost definovat „šablonu“, tzv. **třída**

2.13 Třída (Class)

- Šablona pro vytváření nových objektů tzv. **instancí**
- **Třída** = objekt/entita obsahující 2 typy informací:
    - Seznam atributů v instancích (vč. metadat)
    - Implementace metod (sdílené instancemi)
    - Je-li třída první kategorie, komunikujeme s ní též zasíláním **třídních zpráv**
- V instanci jsou pak pouze přímo hodnoty atributů, odkaz na její třídu a identita.
- **Instanciace** = proces vytvoření objektu + volání konstruktoru (akce hned po vytvoření objektu)
- **Modifikátory viditelnost** = zda jsou položky přístupné přes veřejný protokol (private - přístupný jen přes privátní, public - i přes veřejný, protected - privátní nebo když se přistupuje k potomkovi), prostě mění možnost přístupu k různým entitám jazyka.

2.14 Čistý třídní model OOJ Smalltalk (vsuvka)

- Každý objekt má svou třídu

- Třída je také objekt
- Třída třídy = Metatřída
- Když objekt neobsahuje metody, tak třída je požádána o nalezení reakce na tuto zprávu a její invokaci.

2.15 Klasifikace OOJ

- Dle **přístupu k vytváření objektů** (jestli má koncept tříd)
  - Beztřídní OOJ (prototype-based, class-less)
  - Třídní OOJ (class based)
- Dle **čistoty objektového modelu**
  - Čisté (vše je objekt)
  - Hybridní (míchání s jinými paradigmaty, doplněny objekty)
- Dle **platformy pro běh OO programů**:
  - Překládané (Výhoda: efektivita běhu, Nevýhoda: více zdroj. textu) (výsledkem binární kód)
  - Interpretované (Výhoda: velmi flexibilní, Nevýhoda: pomalé)
  - Částečně interpretované (mezikód, virtuální stroj = interpretuje mezikód)
- Dle **způsobu iniciace výpočtu**
  - Výrazem (někam se pošle zpráva, např. v sigma-kalkulu, OCaml, Smalltalk, SELF)
  - Speciální metodou/funkcí (je to častější, je tam kód, který slouží pro odstartování celého výpočtu, zastřešuje jej)
- Dle **uložení objektů**
  - Zdrojový kód + knihovny (při spuštění je ten objektový systém znovu vytvořen, a pak začne počítat, po výpočtu je to celé zahozeno)
  - Obraz objektové paměti (po dokončení výpočtu je to možné hibernovat a později ji třeba využít)
- Dle **dědičnosti** (počet přímých předků)

    - Jednoduchá (jeden **přímý** předek)
    - Vícenásobná (více **přímých** předků) - problémy s konflikty a duplikací jmen, řešení: zákaz výskytu konfliktních jmen nebo uvádění kvalifikace předka
- Dle předmětu dědění
    - Dědičnost implementace (dědíme i implementace metod)
        - Třídní dědičnost (u třídních jazyků)
        - Delegace (u beztřídní jazyků, dynamická dědičnost)
    - Dědičnost rozhraní (je tam jen informace, že nějaká metoda by měla být implementována, ale nedědíme její implementaci)
- Dle způsobů určení typů
    - **Beztypové** = jazyky bez typů, teoretické a formální jazyky
    - **Netypované** = proměnná nemá určen typ ve zdrojovém kódu, programátor tuto informaci nemá k dispozici, interpret zajišťuji implicitní typové konverze automaticky, např. BASIC, PHP
    - **Typované** = typ určen ve zdrojovém kóde u každé entity
- Dle důslednosti kontroly typů
    - **Silně typované** (zachována typová bezpečnost, nelze provést konverzi objektu na jiný typ)
    - **Slabě typované** (může docházet k implicitní konverzi)
- Dle doby kontroly typů
    - **Staticky typované** (vytvoření vazby proměnné na typ při překladu, před spuštěním, určuje množinu operací při překladu, které podporuje)
    - **Dynamicky typované** (za běhu)

2.16 Principy OO návrhu

- Identifikace objektů/tříd/instanci

- Rozdělení zodpovědností mezi objekty (co který bude mít na starost)
- Různá kritéria dobrého návrhu
    - Trasovatelnost vlastností (kde je co implementované ve zdrojovém kódu) (Ne moc velké a ne moc malé metody, Ravioli code - moc rozdrobený kód, nepřehledné)
    - Robustnost a udržovatelnost
    - Splnění (ne)funkčních požadavků
    - Kvalita služeb

2.17 UML

- **Vizuální jazyk** vhodný pro OO modelování
- Podporuje formální verifikaci a validaci
- Sada nějakých diagramů, které poskytují pohled na systém, typicky jeden ze dvou:
- Strukturální (více statický, zaměřený na data) vs. Behaviorální (zabývá se více tím, co se tam bude dít během běhu systému) **pohled**
- **Diagramy** (graf z entit a vztahů)
    - **Tříd** (když máme TOOJ) (komponent, nasazení)
    - **Objektů**, **sekvenční** (aktivit, stavový, případů použití)
- **Násobnost vztahů** = u UML lze vyjadřovat násobnost vztahů (1, *, 0…1, …)
- **Druhy vztahů** = agregace, kompozice, závislost, generalizace, …

2.18 Formální návrh v UML

- UML se nejčastěji používá na objektově orientovaný návrh/modelování
- Může fungovat jako společný jazyk mezi zákazníkem, návrhářem a programátorem
- Není výpočetně úplný

- Je rozšiřitelný (možný vstup pro generování kódu, profil s reálným časem)

2.19 Výhody/nevýhody OOP

- ✓ Jednoduché na pochopení
  - Skutečné objekty → Softwarové objekty (přímočaře)
  - Podpora dekompozice: Rozdělení komplexity na abstrakci a zapouzdření
- ✓ Praktické (nasaditelné v praxi)
- ✓ Zlepšení produktivity (jsou poté programy znovupoužitelné; propracované metodologie softwarového inženýrství)
- ✓ Stability (změny většinou lokální vůči objektu/třídě)
- Delší učící křivka
- Praxe některé výhody nepotvrdila
  - Probíhá zkoumání možností komponentních technologií
- Generuje pomalejší kód
  - Režie na objekty, režie na polymorfismus

3 Třetí přednáška (Třídní objektově-orientované jazyky a jejich implementace)

Když s objekty programujeme, tak často potřebujeme, aby nějaká množina objektů rozuměla stejným zprávám nebo měla stejnou reakci na přijaté zprávy (metody). Buď můžeme mít v každém objektu znovu přítomné ty samé metody a taky ten samý kód. To je nepraktické a zatěžující.

Řešením je zavést pojmy **třída** a **dědičnost**. A takovým jazykům se říká TOOJ. Alternativní řešení je beztřídní řešení, používá pojmy jako prototyp, rys a delegace, v jazycích založených na objektech (prototypovacích jazycích).

3.1 Třída

- Třída = objekt/entita obsahující:

    - Seznam **instančních atributů** (vč. Metadat, popsáno jméno atributu, typ atributu, pokud je to typovaný jazyk)
    - Data třídních atributů
    - Implementace instančních metod
    - Pokud je třída objektem první kategorie, tak je tam reference na její třídu (Smalltalk, Python)
    - Dále tam můžou být statické položky (metody, atributy) (Java, C++, PHP)
- **Instance** = V instanci jsou pak pouze přímo hodnoty atributů, „reference na její třídu“ a identitu.
- Třída se stará o správu protokolu objektu, směřování zpráv a obsahuje implementace některých metod

3.2 Třídní dědičnosti

- Účel: sdílení/znovupoužití položek skrz třídy, zajištění zpětné kompatibility
- (dědí od sebe třídy, nedědí od sebe instance!!!, ale např. instance třídy dědí od nějaké třídy)
- Jedna třída je při dědění **předek/rodič (nadtřída)** a druhý **potomek (podtřída)** (sdílí položky z předka, přidává své speciální záležitosti), kořenem je třída, která je předkem všech ostatních tříd.
- **Potomek (podtřída)** = odvozená třída popisující změny oproti přímé **nadtřídě**.
    - Sdílení a přidání nových položek
    - Zděděné metody mohou být **modifikovány (redefinovány/overriding)**

3.3 Hierarchie třídní dědičnosti

- **Relace dědičnosti** = částečné uspořádaní na množině tříd
- **Hierarchie třídní dědičnosti** = graf relace dědičnosti

- Reflexivní relace = třída je sama sebou, může používat své atributy, metody, její instance
- Antisymetrie = třída nemůže dědit cyklicky, C dědi od A, ale A nesmí dědit od C
- Tranzitivita = Pokud A je nad třídou C a C je nad třídou D, tak nepřímo A je nad třídou D
- C dědí A, D dědi od C a od A (nepřímo)

3.4 Definice/Deklarace

- Objekt nebo třída musí být popsána před prvním použitím
- Problém nastává, když mám vzájemné vazby mezi objekty/třídami (skrz reference/ukazatele).
    - Možné řešení: Vazba přes jméno skrz deklaraci (tj. zarezervování jména, definice bude následovat později), dopředná deklarace (v C++, jako dopředná deklarace funkce v C)
- **Jmenný prostor** = mapování jmen na objekty jazyka (instance, třídy, atributy, metody, …), nevyskytují se 2 stejná jména
- Rozsah platnosti jmenného prostoru, úrovně zanoření (aby se jména nekryly)

3.5 Operace nad instancemi

- Operace, co můžu provádět, vycházejí z obou protokolů = interního, externího
- Objekt jako chytrá datová struktura
    - Můžu přistupovat k atributům (musíme řešit pořadí atributů a jejich velikost v paměti, tohle zná třída), zasílat zprávu a nějakým způsobem na to reagovat
    - Zasílání zprávy instanci (potřeba rozšířit z pohledu TOOJ)

        - Některé metody nejsou dostupné přímo v naší třídě, ale můžou být v některé nadtřídě (super, base, parrent::)
        - Potřebuji nejdříve získat třídu instance, a pak teprve budu hledat metodu v té třídě odpovídající zprávě, popřípadě se začnu dívat do nadtříd
        - Když ji najde, tak ji invokuje, je třeba si ověřit, že se ji nesnaží invokovat modifikovaným způsobem.
    - Invokace metody
        - Používá se (skrytý) self parametr, kterým se předává do té metody odkaz příjemce - metoda jako chytrá funkce
    - No a pak návrat výsledku odesílateli, řeší běhový systém
    - To vše může zkomplikovat **Modifikátor přístupu** - omezuje to, kdo může s objektem komunikovat, kdo může jaké zprávy zasílat zprávy (private, protected, internal, public)

3.6 Operace nad třídami

Třída jako objekt první kategorie:

- Zaslání třídní zprávy objektu třídy
    - Třída je objekt = analogický postup (self parametr třídní metody je objekt třídy)

Třída jako speciální entita jazyka:

- Využívání: statické atributy a statické metody (bez self parametru) (jsou to klasické funkce uzavřené ve jmenném prostoru té třídy, neliší se to od globální funkce až na modifikátory viditelnosti)
- Statické metody mohou přistupovat pouze ke statickým atributům nebo volat jiné statické metody ze stejné či jiné třídy

3.7 Invokace metody - kontext

Aneb jaké všechny informace máme k dispozici během invokace metody. Těmto informacím budeme říkat **kontext** (Něco jako rámec volání funkce, kdy informace jsou uloženy na zásobník).

Př.:

Sender.callerMethod():

Receiver.method(arg1, arg2)

V rámci provádění metody callerMethod byla zaslána zpráva method objektu receiver (invokována metoda method, která je definována v třídě toho objektu odkazovaného proměnnou receiver), byly předány 2 argumenty.

Co bude v invokované metodě k dispozici:

- Pseudoproměnné typu **self/this**, které reprezentují adresu předanou tím self/this parametrem
- možnost dostat se ke všem **položkám příjemce** (atributy, metody)
- **lokální proměnné** a parametry prováděné metody (arg1, arg2)
- Nepřístupné:
  - **pozice v kódu metody**
  - **zásobník kontextu** (informace o invokovaných metodách, ale neprovádějí se právě, protože se provádí naše metoda)
  - **lexikální kontext**

Pozn: Nejdříve bude na zásobníku kontext callerMethod a nad ním na vrcholu zásobníku kontext metod.

3.8 Typy v OOJ: 2 přístupy

- 2 přístupy, jak je pracováno s typy
- **Čistý OOJ** = vše je objekt a existuje třída nebo objekt, od kterého jsou všechny ostatní odvozeny. (Smalltalk, SELF)
- **Hybridní OOJ** = kromě objektových typů tam mám základní/primitivní, datové typy, struktury (C++, Java)

    - Třída je speciální případ pro popis datové struktury, často se musí použít jiná syntaxe a sémantika
- Hierarchie třídní dědičnosti = některé jazyky mají univerzální kořenovou třídu (předpředpředek, který slouží jako třída, která je nejvýše v třídní hierarchii, pak je hierarchie reprezentována jedním stromem) (Java, Python, Delphi) Nebo kořenová třída neexistuje (C++, PHP5), pak vzniká graf několika nesouvislých komponent.

3.9 Implementace třídního OOJ

Jak ukládat objekty, jak invokovat metody pomocí pozdní vazby

Z hlediska konceptu objektové orientace **počítáme s objektovou pamětí, která je asociativní, heterogenní** (libovolně velké a libovolně strukturované objekty, pojmenované proměnnou) vs. **Počítačová paměť, který je lineární a homogenní** (pole bytů, s tím musím vystačit).

Model výpočtu = máme k dispozici zasílání zpráv/invokace metod, vytváření objektů, přístup k atributům vs. instrukční sada = k dispozici nástroje pro práci se zásobníkem, nějaké volání (skáče se na návěští, ukládá se na pomocný zásobník). Nepůjdeme ale tak do hloubky

3.10 Implementační problémy

Co tedy budeme diskutovat:

- Jak uložit instance v paměti (uložení atributů, metod)
    - Musíme mít možnost přistoupit i ke zděděným položkám
- Jak přistupovat atributům a metodám, aby to reflektovalo třídní hierarchii a jak tohle ovlivňují modifikátory viditelnosti a jakým způsobem implementovat pozdní vazbu mezi zprávou a invokovanou metodou (polymorfní invokace metody - virtuální metody, invokována pozdní vazbou)
- Problémy vícenásobné dědičnosti

3.11 Uložení instance v paměti: Návrh 1

- Instance bude v paměti uložena podobně jako instance datové struktury (atributy uloženy za sebou), ale ne jako variantní!

- V každé instanci třídy nebude uložen kód metod, pouze odkazy na funkce, které implementují ty metody.
- Chybí statické metody (překladač v době překladu ví, která metoda bude statická, vygeneruje kód, který skáče na jasné místo v paměti) a nevirtuální instanční metody (instanční metoda, budu tam dávat self parametr, nepotřebuju pozdní vazbu, stačí brzká vazba, zase adresa této metody může být stanovena během překladu programu, bude se navíc předávat parametr, který bude odkazovat na místo v paměti, kde je ta instance)
- Je tam odkaz na třídu, ze které ta instance vznikla (nemusí to být rodičovská třída, je to třída, ze které instance vznikla)

3.12 Přístup k položkám instance

- V těle metody bude předán self-parametr (self, this), je typicky pro čtení (pseudoproměnná, mění se pomocí běhu, neměníme jej my)
- Implementace: předává se self-parametr jako nultý parametr (standardizování pomocí ABI)

3.13 Problémy návrhu 1

- Každá instance třídy A (nebo B) nese stejnou sadu odkazů na metody
- Přístup k původním metodám při jejich redefinici
    - Potřeba vynucení hledání metody v nadtřídě
    - Proměnná: super, base

3.14 Uložení instance v paměti: Návrh 2

Vytknutí odkazů na metody do extra tabulky, pro každou třídu je jedna tabulka, která bude sdílená všemi instancemi pomocí odkazů na tu tabulku (**tabulka virtuálních metod, VMT**).

3.15 Optimalizace uložení instance

Staticky typované OOJ

- Každá třída má plochou VMT ➔ v každé tabulce virtuálních metod budou metody, které jsou dostupné v rámci toho typu.
- Jak to bude s voláním původní verze té metody (před redefinicí v instanci): řešení super: využije statickou adresu VMT předka

3.16 Vytváření instancí (Instaciace)

Pozn: **Instance** = objekt, který obsahuje instanční proměnné a odkaz na třídu, ze které vznikl

Je to vytváření objektu pomocí předpisu daného konkrétní třídou. Jednoduše vytváření instance třídy.

2 možnosti, jak se tvoří instance v TOOJ:

1. Java, C++, PHP používají klíčové slovo **new** (operátor, který bere jako parametr jméno třídy, ze které se má ta instance vytvořit). Potom je tam nějaký konstruktor, který se má zavolat na základě počtu parametrů.
2. Další variantou je, že se použije **třídní zpráva**. Objektu třídy pošlu zprávu „vytvoř mi instanci“ a rovnou v té zprávě je určena třídní metoda, která má sloužit jako konstruktor.

Vytvoření instance se skládá ze 2 částí:

1. První část je vynucena například příkazem new. **Přidělí se paměť pro tu instanci**. Většinou na haldě (new), C++ umožňuje uložit na zásobník, když se nepoužije new. Výhoda zásobníku: automaticky destruován, na haldě žije dál
2. Pak se pustí odpovídající **konstruktor**. Závisí to na počtu parametrů nebo na jménu konstruktoru. Máme už v něm k dispozici tu instanci, ne s inicializovanými instančními atributy a self parametr. Metoda pro inicializaci atributů. (Volání konstruktor dle dědičnosti = zavolá se konstruktor třídy A, který má na starost inicializaci atributů definovaných v třídě A, a pak se teprve zavolá konstruktor třídy B, který má na starost

inicializaci atributy třídy B nebo změnit atributy z A. Pozn: A je nad B)

3.17 Rušení instancí/objektů

- Implicitně = Garbage collector (vlastnost objektového modelu, zařizuje, že když nějaká instance už není potřeba v paměti, neexistuje na ni odkaz, tak ji odstraní z paměti, i shluky objektů takto odstraňuje, odkazuji vzájemně na sebe, ale programátor se na ně ztratil ukazatel)
    - Některé jazyky umožňují **finalizační metody**, které se spustí poté, co ta instance má být z paměti uvolněna. Není to spuštění garantováno.
- Explicitně = hlídá a provádí programátor, když nemáme garbage collector.
    - Pomocí speciální metody **destruktor**, pomocí něj je objekt uvolněn z paměti.

3.18 Řízení toku programu (flow control)

- U hybridních OOJ je to kombinace procedurálního řízení toku (klíčová slova, příkazy volání funkce, větvení, iterace) a OO řízení toku (instanciace, (polymorfní) invokace metody)
- U čistých OOJ to bylo jen OO řízení toku.
- Polymorfní invokace = polymorfismus (pozdní vazba)
- Nevýhody: komplikovanější syntaxe a sémantika, nevhodné pro učení OOP, Výhoda: pozvolnější přechod z modulárních na OOJ

3.19 Třída jako typ

Pokud mluvíme o třídě jako typu, tak předek je **nadtyp** svého potomka = **podtypu**.

Příklad: Mějme metodu m, která jako parametr vyžaduje objekt třídy A. Pak jako parametr lze metodě m zadat: Objekt stejného typu (např. instanci třídy A) nebo objekt třídy odvozené od třídy A (pomocí dědičnosti)

Ve většině OOJ se na třídu se dívá jako na typ, že pokud je B je podtřída třídy A a instance Beta který je instance třídy B, tak potom Beta je také instance třídy A. Instanci třídy B lze využít kdekoliv tam, kde je očekávána instance třídy A.

Této vlastnosti se říká **zahrnutí typu B do typu A**. Proto ve staticky typovaných OOJ nelze rušit položky.

4 Čtvrtá přednáška

4.1 Vývojová prostředí (vsuvka)

Čistě OOJ se liší od OO IDE (software usnadňující práci programátorům) svým přístupem

Rozlišíme 2 přístupy:

- **Zaměření na program a zdrojový kód ➔ zaměření na instance/objekty**
- Zaměření na samotnou objektovou paměť a objektový systém jako celek
- Interaktivita = možnost explorativního programování (víc než že bychom psali zdrojoví kód, tak zkoumáme ten systém a zjišťujeme, kde co jak udělat, aby to vypadalo podle našich představ)
- Vizualizace (např. diagram tříd)

4.2 Případová studie: Smalltalk

Nepoužívá se klíčové Class, vytváření tříd instaciací z metatřídy například.

Instanciace (třídní zpráva třídě: t:=Time mow.

Smalltalk je čistě OOJ kód, tudíž blok kódu je také objekt: b:=[:arg| arg+3]. Tento kód/blok bude přiřazen do proměnné až tehdy, když mu pošlu zprávu, aby byl přiřazen (b value:4.)

Ve smalltalku operátory jsou zprávy a ty se vyhodnocují levě asociativně (1+2*3=9). Všechny zprávy mají stejnou prioritu.

Ve smalltalku jsou True a False též objekty, což v třídní jazyce znamená, že mají svou třídu a jsou instance. Jsou to klíčová slova. Mají v sobě 2 metody not (u true vráti false a u false vrátí true) a

ifTrue:ifFalse: (má za úkol v případě, že příjemce je pravdivostní hodnota true, tak vykonat blok, který je zaveden jako první parametr. Když false, tak druhý parametr)

Nil zastupuje nedefinovaný objekt, nerozumí žádné zprávě až na isNill a inNotNill (pozn: smalltalk je netypovaný, neurčujeme u proměnných typ, dynamicky typovaný, zasílá zprávy bez rozmyšlení, jestli ten objekt na ně zná reakci, když nezná, vyvolá výjimku).

Selektor zprávy je objektem. Můžeme jej dát do proměnné a pomocí jiné zprávy jej poslat příjemci.

4.3 Běhová prostřední OOJ

V jakých běhových prostředích může OOJ potažmo běhový systém fungovat. Jsou 3 možnosti:

- **Kompilované/překládané jazyky**, které generují binární kód, který je spustitelný na operačním systému nebo přímo na hardwaru. Výhoda: rychlost, Nevýhoda: ústupky v návrhu toho jazyka, aby se té rychlosti dalo dosáhnout (C++).
- **Částečně interpretované jazyky** využívají virtuální stroj, což je speciální aplikace/program, který bývá platformě závislý. Potom ten Bajtkód (mezikód) může být platformě nezávislý (Java, Smalltakl, C#). Dokáží pracovat flexibilně (načíst celý balík za běhu), optimalizace pomocí JIT kompilace (kusy kódu, které jsou nejvytěžovanější, tak jsou z toho Bajtkódu přeloženy před/při spuštění na binární kód).
- **Interpretované jazyky**, kde je efektivita upozadněna, ale umožňují pohodlně se vypořádat s řadou hodně dynamických vlastností (Python, PHP), překládají se do mezikódu. Tyto jazyky mají velkou reflektivitu.

4.4 Reflektivita (jazyka)

- Je systém, který zahrnuje kauzálně propojené struktury popisující sebe sama (ten program, částečně nebo úplně)
- Jsou tam 2 úrovně:
    - Vnitřní reprezentaci pouze zkoumáme

    - Můžeme ji měnit (kauzálně) - má to potom projev, jak se to poté chová v průběhu programu
- Také rozlišujeme, jaký typ reflektivity právě používáme:
    - Strukturální - pracuje se statickými strukturami (balíčky, třídy, metody)
    - Behaviorální - pracuje se s prováděním programu (invokace metody, přiřazení, zásobník volání)

4.5 Typový systém objektově orientovaných jazyků

Typy x třídy

Třída = určuje množinu vnitřních stavů a operací nad nimi
Typ = určuje množinu validních hodnot a operací nad nimi
Třída určuje typ (argumentů, metod), ale ne naopak.

Přístupy pro určení typu

- Typ daný jménem (třídy) (C++, Java)
    - U těchto jazyků se budeme bavit o tzv. **vyžádané dědičnosti**
- Typ daný výčtem položek
    - Rozpoznávám skutečný (pod)typ (dynamicky) (u metod třeba)
- Kombinace (výčet jmen) (někde na pomezí těch dvou)
    - Např. vícenásobná dědičnost tříd (více přímých předků, to znamená vytvořit typ jako kombinaci několika typů několika nadtříd) nebo rozhraní (podobné jako u tříd, statické) nebo systémy s rolemi (dynamicky)

Typ daný jménem

Týká se to typovaných jazyků a takových, kde třídu se dá použít jako typ.

Mějme funkci f s parametrem přijímajícím instance třídy A, pak parametr může být **instance třídy A** (stejného typu) nebo **instance libovolné podtřídy A** (kompatibilního podtypu)

Vysvětlení k předchozímu odstavci: V těchto jazycích často potřebujeme pojem zahrnutí typu B do typu A. Pokud někde chci použít instance A a mám někde podtřídy třídy A, tak můžu na tom místě použít i instance těch podtříd. Tohle vše je kvůli tomu, že jazyky mají staticky typovaný systém.

Příklad: Java, C++

4.6 Vyžádaná dědičnost (druhotná úloha dědičnosti v jazycích)

Pro zajištění kompatibilního typu objektu (instance) musím použít dědičnost. Aneb když je na parametru metody vyžadována instance třídy nebo z ní zděděné třídy. Dědičnost zajišťuje existenci patřičných položek (atributů a polymorfních metod = aneb všechny podtřídy budou mít metody, které má jejich třída). Využíváno (statickou) typovou kontrolou. Potřebná u staticky typovaných jazyků, např. v Java, C++, Delphi.

4.7 Skutečné (pod)typy - duck typing

Zjištění typu prozkoumáním položek objektu, typicky mě zajímá pouze potřebný podtyp.

Jazyky musí podporovat test implementace potřebného podtypu buď staticky (potřeba staticky typovaný jazyk) nebo za běhu (jazyky s duck typing, polymorfismus nezávislý na třídní dědičnosti). Např. Smalltalk, SELF, Python. Prostě je doménou u dynamicky typovaných jazyků.

4.8 Rozdíl mezi vyžádanou dědičností a skutečnými podtypy

U vyžádané dědičnosti lze v proměnných a parametrech metod předávat pouze instance (staticky dané) třídy nebo její některé podtřídy. Tyto jazyky jsou většinou staticky typované a kontrola typu probíhá na základě jména typu/třídy. Reálné podtypy mají potom kontrolu typu prováděnou dynamicky a po položkách (není zde omezení pouze na třídní podhierarchii s danou kořenovou třídou). Vše bylo možné jako bonus završit vhodnými příklady programovacích jazyků.

4.9 Implementace skutečných podtypů

Jak najít požadovanou položku?

- Nezávislost na třídě ➔ různé pořadí položek v objektech, ale kód volání metod statický
- ➔ nutný **jednotný přístup** pro všechny objekty a nutnost řešit **za běhu (pomocí vazeb přes jméno)**!
- Interní rutiny (využívají se pro vyhledání přes jméno) běhového prostředí pro přístup k (veřejným) atributům a přístup k (veřejným; virtuálním) metodám

4.10 Důsledky pro implementaci

- Jelikož musíme vyhledávat přes jméno, tak tam musí být nějaké interní struktury datové, které toto vyhledávání podporují (třeba slovníkové struktury, třeba slovník, kde na základě jména jeho atributu najdu hodnotu nebo selektoru najdu implementaci metody). Například Python a slovník ___dict___.

4.11 Vícenásobná (třídní) dědičnost

- Zděděná třída má více jak jednu přímou nadtřídu.
- Např. C++, Python
- Je to diskutabilní vlastnost. Častý výskyt OOA (analýza) a OOD (návrh). Není často při programování nutná. OO návrhu se nám toto hodí, ale je tam hodně problémů při implementaci.
- Správně použití je ojedinělé

4.12 Problémy vícenásobné dědičnosti

- Když (přímé) nadtřídy obsahují položky stejného jména a typu
- Když (přímé) nadtřídy obsahují položky stejného jména, ale různého typu
- Inicializace instancí = pořadí volání konstruktorů při vytváření nových instancí

- Jak ukládat tyto instance v paměti = instance třídy C (C je přímá podtřída A i B, lze využít kdekoliv a očekávám instance tříd A nebo B)

4.13 Problémy metody I

- Metody stejného jména a typu v definici nové třídy, jak to řešit
  - Zakázáno (např. SELF - kontrola za běhu)
  - Skrytí (např. A::m() přístupná jen v A, ne v C)
  - První nalezený výskyt (Python) (když se ve slovníku najde první m z A, tak se provede tahle)
  - Povinná redefinice - plná kvalifikace (např. C::M {A::M}) nebo problém přístupu k instanci C jakoby to byla instance třídy B (tj. budu volat A::M)

4.14 Problémy Metody II

- Metody stejného jména a různých typů v definici nové podtřídy
  - Zakázáno
  - Povolit jejich souběžnou existenci implementací = **přetěžování metod** (koexistence několika verzí metod a volání podle počtu parametrů)

4.15 Problémy: Atributy

- Atributy stejného jména a typu
  - Zakázáno
  - Skrytí (A::d přístupný jen v A, ne C)
  - Sloučení - v paměti se to bude zdát, jako by ten atribut byl jenom jeden; v pythonu používá první výskyt v paměti; replikace atributu v paměti - bude v paměti 2x, ale když jej změním na jednom místě, tak se změní na obou místech
- Stejnojmenné atributy různých typů jsou zakázány

4.16 Inicializace instancí

- Implicitní volání konstruktorů nadtříd:
    - Využití pořadí zápisu nadtříd ve zdrojovém textu a prohledávání do hloubky (DFS) ➔ acyklické (opačné) pořadí volání konstruktorů
- Explicitní (uživatelem definované) - nebezpečí deadlocku nebo vynechání některého konstruktoru

4.17 Rozhraní

- Schéma deklarující seznam metod (jména, parametry, návratové typy)
- Objekt/třída implementující toto rozhraní musí tyto metody implementovat.
- Např Java, C#
- Teoreticky se dají deklarovat i atributy, většinou nepodporováno.
- V jazycích, kde není koncept rozhraní, bývá podobný koncept: Abstraktní třída. Je to třída, kterou nelze instanciovat, protože minimálně jedna metoda je abstraktní - nemá definované tělo.
- Výhodou oproti vícenásobné dědičnosti je, že nemá problém s kolizí různých implementací pro stejně pojmenované metody (i atributy)

4.18 Vícenásobná dědičnost rozhraní

- Hierarchie vícenásobné dědičnosti rozhraní (částečně uspořádání na rozhraních)
- Většinou kombinace s jednoduchou třídní dědičností
    - Třída implementuje libovolný počet rozhraní
        - Včetně metod deklarovaných v nadrozhraních

4.19 Definice rozhraní

Definice rozhraní obsahuje seznam implementovaných metod, není tam kód metod. Rozhraní pak od sebe mohou dědit, ale pouze tu informace typu, ale ne kód ani implementaci. Je tam prostě popsáno, které rozhraní rozšiřuje to naše, které dědí od něj.

Třídy pak implementují rozhraní, je to napsané v jejich definici (class xx implements yy)

4.20 Rozhraní a polymorfismus

- Na místě parametru vyžadujícího rozhraní lze použít instanci libovolné třídy implementující toto rozhraní
- Rozhraní jako typ proměnné: Printable sq = new Square();
- Jak zařídit, abychom našli tu správnou metodu co nejefektivněji? Hledat přes jméno (Python) a druhý přístup vytvoření struktury podobné VMT (tabulky virtuálních metod), které budou reprezentovat jednotlivá rozhraní v kombinaci s implementujícími třídami.

4.21 Implementace rozhraní (Jak najít správnou metodu co nejefektivněji)

Je to jednodušší než vícenásobné třídní dědičnosti.
Problémy k řešení

- Vazba přes jméno, nutnost slovníků (Python)
- Výběr správné VMT (Java)
- Interní rutiny

Nevýhoda proti vícenásobné třídní dědičnosti - nelze zajistit sdílení implementace jistého chování napříč hierarchií dědičnosti.

4.22 Rozhraní objektu x Protokol

- Rozhraní (koncept) = pojmenovaný seznam metod k implementaci
- Rozhraní objektu = množina zpráv, kterým se objekt zavazuje rozumět
- Protokol (obecně, jak jsme to říkali) = postup hledání reakce na zprávu
- Protokol (definice alternativně) = množina všech zpráv, kterým objekt rozumí

4.23 Systémy s rolemi OK

Vícetypový objekt = objekt má najednou více typů, tzv. role (např. více přímých předků) a dynamicky tyto role nabývá/pozbývá bez nutnosti změny své identity.

Tento objekt přetrvává v systému dlouhou dobu, pro konkrétní použití tedy objekt může přebírat konkrétní roli.

Použití v OO databázích (potřebujeme dodržet ACID vlastnosti, tak proto) a v interpretovaných systémech s dlouhou dobou života objektů (v jazycích SELF, Python).

4.24 Operace s vícetypovými objekty OK

Dodatečné požadavky na operace za běhu:

- Objekt spravuje svou identitu (role), ne třída
  - **Operace vytvoření objektu (create)**
- Pohled na objekty skrze specifickou roli (typ)
  - **Přetypování (cast) (tím pohledem)**
- Změna objektem podporovaných rolí (přidáním nových metod, atributů)
  - **Přidání role (bind)**
  - **Odebrání role (unbind)**

4.25 Perzistence objektů

Perzistentní objekt (instance) = přežívá dobu běhu aplikace; při dalším spuštění aplikace je opět k dispozici (se stejným stavem i **identitou** jako při vypnutí aplikace)

Co není perzistence - ukládání snímků celé objektové paměti, ukládání/načítání objektů na aplikační úrovni

Transientní (dynamický) objekt = neperzistentní, objekty, které nepotřebujeme, aby byly perzistentní, pracuje se s nimi rychleji, protože odpadá režie na jejich ukládání, méně náročné na zdroje

4.26 Implementace perzistence

Klíčové slovo (persistent) pro deklarativní označení perzistentních objektů.

a) Ukládají se pouze hodnoty atributů objektů, metody zůstávají někde bokem, tohle není optimální
b) Ukládáme položky objektu (případně třídu), typicky využíváme (semi)interpretovaných systému, problémy nastávají u kompilovaných systémů, je to velmi závislé na cílové platformě

4.27 Beztřídní OOJ

4.28 Charakteristika BOOJ

- Vytváří objekty klonováním prototypů (již existující objekty)
- Koncept dědičnosti implementován tak, že mám k dispozici odkaz na 1 nebo více rodičů (podle toho, zda je to vícenásobná dědičnost) a zprávy, kterým daný objekt nerozumí, jsou delegovány na toho rodiče.
- Delegace = sdílení položek mezi objekty
    - Dynamická dědičnost (možnost měnit rodiče za běhu)
    - Když se provádí kód, důsledek té delegace, je prováděn v kontextu toho skutečného příjemce, ne toho, kdo delegoval.
- Využívají se většinou interpretované systémy (SELF, JavaScript, Slate, Isaac, Lua)
    - Propojení vývojového a běhového prostřední, tzv. inkrementální interaktivní (explorativní) programování

4.29 Případová studie: SELF

Objekty v selfu se skládají z položek, kterým se říká sloty
Sloty se skládají ze jména a z odkazu na jiný objekt
Sloty mohou být datové, kdy se odkazuje na objekt, který nám dává nějaká data. Pak je tam metodový slot, který odkazuje objekt metody, je tam nějaký kód, který se dá spustit, když reagujeme na zprávu. A nakonec rodičovské sloty (hvězdička na konci parent*), delegují se na ně zprávy, které objekt nerozumí.

(Pozn.: delegace = zpráva stále s sebou nese kontext původního příjemce, další zprávy jsou stále posílány původnímu objektu, u přeposlání se další zprávy posílají objektu, kterému byla zpráva předtím přeposlána)

- Data
  - Čtení (zpráva name) ze slotu, zápis (zpráva name:) do slotu
- Metoda
  - Objekt metody s jiným kódem
  - Při invokaci klonuji objekt metody ➔ vznikne aktivační objekt, kde self parametr odkazuje na původního příjemce zprávy
  - Tímto mám zajištěný jmenný prostor a pokud nemám proměnnou ve slotech metody, tak hledám v rodiči - příjemce zprávy.

4.30 SELF „třídní“ model

Lze jej nasimulovat, bude svým způsobem dynamický, stále bude možno dynamicky měnit třídu.

„Třída“ tzv. rys = objekt pouze se sdílenými metodami a rodičovskými sloty pro delegaci.

Prototyp

- Obsahuje datové sloty pro atributy a jejich implicitní hodnoty
- Deleguje na rysy (simulující rodičovské „třídy“)

5 Přednáška šest OK

5.1 Co je programovací jazyk

I když program definuje výpočet realizovaný cílovou architekturou, tak nemusí popisovat výpočet (viz. LaTeX).

Jazyky jsou:

- Univerzální = pro obecné použití (C++, Java)
- Specializované = pro speciální použití (pro sazbu textů = LaTeX, pro popis integrovaných obvodů = VHDL a další)

Jedním z kritérií, jak lze jazyky dělit, je úroveň abstrakce a způsob abstrakce **dat** a **řízení** zpracování dat.

5.2 Abstrakce dat

Z hlediska přístupu k dělení a manipulaci s daty lze rozlišit tyto kategorie jazyků:

- **Strojové jazyky** - nejnižší úroveň, musíme si pamatovat adresy.
- **Jazyky vyšší úroveň** (Fortran, Cobol, Algol 60) - jednoduše abstrahují od stroje, nemusíme si pamatovat adresy, stačí jména.
- **Univerzální jazyky** (PL/I) - Ohromná sada abstrakcí, množina abstraktních typů se nedala změnit. (nikoliv jazyky pro obecné užití)
- Proto nastoupili **blokově strukturované jazyky**, které nedávaly tyto sady dopředu, vůbec nic. Můžu později změnit strukturu, rozšířit ji, vytvářet je.
- Na to navázaly jazyky, tyto datové struktury nám nabízejí (abstraktní datové typy) - **modulární blokově strukturované jazyky**

A poslední část jsou OO jazyky, které to spojují s kódem.

Strojové jazyky

Data jsou sekvence bytů/bitů, velikost dat vychází ze strojových instrukcí. Vyšší logická struktura dat je spravována programátorem.

Operace jsou velice jednoduché logické nebo aritmetické nebo bitové.

Podpora vyšší abstrakci dat není žádná, podpora základních typů je specifická dle cílové architektury. Programy jsou specifické pro konkrétní počítač.

První jazyky vyšší úrovně

Nabízejí jen jednoduché datové typy, na rozdíl od předchozí kategorie dochází k abstrakci od cílového systému (HW abstrakce).

Je tam velká orientace na platformu (velikost a rozsah typů kopíruje cílovou platformu). Nejsou obecného použití. Mají definovanou pevnou strukturu zdrojového textu (např. Fortran).

Jazyk PL/I

Proto vznikl jazyk PL/I, který má velkou množinu datových abstrakcí v sobě, snaží se být pro obecné použití. Definuje velké množství datových typů (jednoduché, komplexní) Ale nebylo možné vytvořit nové abstrakce, jen použít ty existující.

Blokově strukturované jazyky

Pascal, C.

Umožňují definovat složitější datové struktury pomocí jednoduchých konstrukcí. Lze definovat vhodné datové abstrakce, které můžeme spojovat i vnořovat. Je třeba kód sdílet, aby jej pochopil i někdo jiný = tudíž programy začínají být čitelnější a přehlednější.

Modulární blokově strukturované jazyky

Rozdíl oproti předchozí kategorii je v tom, že jazyky umožňují oddělit definici typu od operací, které ho manipulují. Vnitřní struktura typu je skryta, typ může uživatel použít. Datový typ pro osobu reprezentuje jen jeho jméno a modul, který jej zpracovává.

OO jazyky

Programování se stává datově centrické. Spojujeme data s kódem, uzavřeme je do virtuální krabičky a jen z venku povolíme něco, co s tím můžeme dělat.

Jiná paradigmata

- Jazyky logické = základem jsou predikáty, termy, s těmi pracujeme jako s daty
- Funkcionální = všechno je spojeno s funkcemi
- pro sazbu textu

- pro definici a manipulaci dat

5.3 Abstrakce řízení

Na nejvyšší úrovni můžeme rozlišit 2 typy programovacích jazyků.

Imperativní jazyk - (procedurální jazyky) je takový jazyk, kde programátor specifikuje 2 otázky řízení běhu programu: **co** za operace má být provedeno a **v jakém pořadí** to má být provedeno (Fortran, C, C++).

Procedurální jazyky

Sestavují program jako posloupnost příkazů, které lze vnořovat jen u některých jazyků

Strojové jazyky = nízká čitelnost, jediné větvení je v podobě skokových instrukcí/příkazů, pomalý vývoj, náchylné k chybám, možnost cyklus.

Jazyky vyšší úrovně = rozpoznáváme tyto typy abstrakcí pro řízení běhu programu: podprogramy, bloky, koprogramy, paralelní programy, odložené zpracování

- **Podprogramy** = umožňují vnořené zpracování určitých logických funkcí/operací. Volán je přes své rozhraní, předávají se mu parametry přes něj a vrací přes něj výsledek. Má jednoznačnou identifikaci = jméno.
- **Bloky** = žádné jméno, takže je nelze volat, kód uplatněn pouze v daném místě (C, Algol 60).
- **Koprogramy** = pojmenované bloky kódu, které mají vzhledem k sobě symetrický vztah. Zpracovávány současně, rozhraní stejné jako u podprogramů.
- **Paralelní zpracování** = větší logické celky, paralelní běh, abstrakce od CPU,
- **Odložené zpracování** = jazyky nezpracovávají výpočty, pokud není nutný pro zbytek výpočtů

5.4 Abstrakce řízení II

Deklarativní jazyky = programátor pouze píše, jaké operace se mají provézt, pořadí je dáno překladačem.

Je tam nějaké strategie, která dopředu určuje, jak se bude určovat to pořadí.

Datové abstrakce

Tyto jazyky jsou vysoce abstraktní a mají vysokou vyjadřovací sílu, proto zde jsou příkazy pro větvené toku řízení a opakování (ne cykly), které lze zanořovat. Nejsou tam nízko-úrovňové věci.

U těchto jazyků máme 3 strategie, jak se předává řízení z jednoho podprogramu do druhého - **volání hodnotou** (parametry se vyhodnotí před voláním podprogramu), **volání jménem** (parametry se do podprogramu předávají nevyhodnocené), **volání v případě potřeby** (až podprogram potřebuje, tak si nějaký parametr vyhodnotí).

5.5 Nestrukturované jazyky

Fortran, Basic například

V současnosti využívány ve skriptovacích jazycích.

Nestrukturované řízení toku programu = goto, smyčky řízené pomocí if goto.

Nestrukturovaná data = nedají se data strukturovat, nemůžu je shluknout k sobě.

Formální báze

Je takový formální prostředek (kalkul, algebra), který umožňuje exaktně popsat všechny konstrukce daného jazyka. (V nestrukturovaných jazycích neexistuje, zpětné vytvoření není možné a není vyžadované, mají takové určení, kde podobné úkony nejsou vyžadovány)

Nemají ji.

Syntaxe

Nestrukturovaný jazyk je jako každý jazyk doprovázen dokumentací, která vysvětluje syntaxi a sémantiku. Syntaxe je podána přirozeným jazykem s příklady ukazujícími na správný zápis. Formální zápis pomocí (E)BNF nebo pomoci gramatiky je zřídkavý. Důvodem je to, že syntaxe je

jednoduchá a často doplňována sémantikou. Má zvláštnosti, například u Fortranu se nespecifikuje pořadí u předávaných parametrů.

Sémantika

Je neformální, Vytvořeny příklady doplněné popisem, který vysvětluje dané chování. Tohle je možné, jelikož sémantika není složitá. Popis je často inkrementální (jednoduché nejdřív).

Formální verifikace a validace

U nestrukturovaných se s tímto nesetkáváme, protože to není používáno a není to realizovatelné.

Softwarové inženýrství

Aplikovat zásady softwarového inženýrství na takové jazyky je omezené. Vlastnosti těchto jazyků nejsou vhodné pro použití v týmovou práci na rozsáhlých projektech, neboť programátor lehce poruší zásady dobrého programování.

Datové a řídící abstrakce

Nestrukturované jazyky neposkytují uživateli žádné datové ani řídící abstrakce. V těchto jazycích jsou základní numerické typy, znakové řetězce a pole. Na řídící úrovni to jsou smyčka s pevným krokem a řídící proměnout (for cyklus), příkaz pro větvení (if) a skok.

Otevřené podprogramy

Nové typy nelze definovat, existující nemohou být sloučeny. Pro manipulaci s daty lze použít jen vestavěné operace. Kombinaci operací nelze na jiném místě v programu využít jinak než vytvořením kopie textu, jelikož nelze definovat podprogram. Lze to nahradit **otevřenými podprogramy**.

Pozn.: **Uzavřený podprogram** = jasně dané rozhraní, jasně dané argumenty, je jasně dán typ výsledku, je jasně dán konec, má to blok, je to jasně vymezeno, nemůže být o pár řádků dál další jeho část. Skrz rozhraní komunikujeme, dáváme argumenty, komunikujeme, dostáváme výsledek.

Otevřený podprogram = je uložen v rámci hlavního (často jediného) zdrojového textu. Nemá definované pevné rozhraní, tzn. vstupní a výstupní bod, parametry, výsledek apod. Vstup se děje skokem na příkaz, jímž má výpočet podprogramu začít, ukončení podprogramu je dáno vyvoláním příslušného příkazu (nikoliv doběhnutím výpočtu do/za určité místo). Parametry se předávají přes globální proměnné, výsledky se získávají taky přes ni, není identifikace = nemůže být rekurze, není to vidět v textu. Nemusí to ležet v jedné části textu - stačí goto a přeskočí to na jiné místo v textu.

Shrnutí - podprogramy

- Parametry se předávají jako globální proměnné, výsledky předávány pouze globálními proměnnými, neexistují lokální proměnné
- Žádná rekurze není implicitně podporovaná
- Složitá struktura programu, text celého podprogramu není jeden blok, může být roztroušen po celém textu programu.

Návrh programu

Je poměrně těžký, díky absenci uzavřených podprogramů je těžké návrh strukturovat, je jednoduché udělat chybu, týmová práce těžká, návrhové metodologie těžce aplikovatelné, data viditelné z definičních/deklaračních bodů až do konce běhu programu, pokud nejsou explicitně vymazána.

Zabudované operace a ad hoc řešení dělají programování lehčí a rychlejší, rozsáhlé a středně těžké programy z dneška jsou v tomto těžce programovatelné, nemožné použít ADT, programovací práce je těžší.

Typy a jejich zpracování

Tyto jazyky jsou netypované většinou (explicitně se neuvádí typ proměnné), za běhu se ten ty proměnné může změnit, stačí pouze přiřadit proměnnou jiného typu.

Změnám typů u takových jazyků brání **implicitní typ** = typ proměnný často rozpoznatelný ze jména proměnné (Basic, Fortran).

Typy v netypovaných jazycích = proměnné jsou typu vnitřně změnitelného. Prostě typ, jaký potřebujeme. Paměť alokována, když je proměnné poprvé přiřazena hodnota. Paměť dostatečně velká pro všechny požadované numerické reprezentace (int x float x string x list)

Tok řízení programu

Tyto jazyky nenabízejí možnost, jak blokově strukturovat skupiny příkazů, které jsou řazeny sekvenčně nebo logicky vnořené. Je to sekvence jednoduchých operací, která je provázána systémem příkazů skoku. Podprogramy jsou maximálně v podobě otevřených podprogramů. Vstup do nich je určen číslem řádku nebo textovými návěštími.

Program je tedy pak těžko čitelný a nepřehledný. Je také náročný i na překlad.

5.6 Blokově strukturované jazyky

Pascal, Algol, strukturované vlastnosti jsou v hodně jazycích, např. PHP

Snaha dát programátorovi flexibilitu a dostatečné výrazové prostředky pro tvorbu programů ze všech oborů co nejefektivněji. Je možné je využít i pro větší projekty, většinou jen jeden soubor, rovnou překládány do binárního kódu, výhodné pro výuku.

Formální báze

Nemají ji. Jazyk byl navržen bez užití formalismů.

Syntaxe

Syntaxe je podávána formální cestou. Důvodem je ukázat logiku a hierarchii jazykových konstrukcí. Bezkontextové gramatiky potom stojí v základu takových popisů. Mezi typicky užívané popisy pro sytaxi patří (E)BNF (Backus Naurova forma) a syntaktické grafy

Sémantika

Je neformální, Popis je velmi propracovaný. Není vysvětlováno na příkladech, ale obecně. Objevují se formální doplňky v místech, kde je to vhodné.

Formální verifikace a validace

Sice chybí formální báze, ale pro formální ověření vlastností algoritmů lze využít Floyd-Hoare logiku. Tato logika specifikuje pravidla pro práci se základními konstrukcemi jazyka, čísly a jejich ekvivalenty.

Floyd-Hoare logika

Specifikuje pravidla pro práci se základními konstrukcemi jazyka. Pro každý programový prvek (příkaz, posloupnost příkazů, blok příkazů) je definována podmínka splněná před a po provedení operace popsané daným prvkem.

Notace: Nechť C označuje příkaz, P označuje podmínku platící před provedením příkazu a Q podmínku platící po provedení pří- kazu, potom zápis: {P} C {Q} vyjadřuje parciální správnost (korektnost) vztahu podmínek a příkazu a zápis: [P] C [Q] úplnou (totální) správnost.

Softwarové inženýrství

Aplikovat zásady softwarového inženýrství na takové jazyky je lepší než u nestrukturovaných jazyků. Lze uplatnit zásady dekompozice problém v datové i řídící rovině. Strukturované jazyky jsou však omezovány: program je tvořen jen jedním souborem, který může být nepřehledný, lehce se udělá chyba, chybí moduly - nelze skrývat manipulaci s daty.

Datové a řídící abstrakce

Strukturované jazyky nabízejí sadu základních (atomických) typů. Tyto zahrnují potom číselné, znakové a pravdivostní reprezentace. Umožňují definovat typy odvozené. Ty vznikají z existujících typů (základních i odvozených). Vznik nových má pod kontrolou programátor.

Objevují se i ukazatele, umožňují vznik rekurzivních datových struktur. Umožňuje definovat seznam, strom atd.

Pro práci s novými typy si musí programátor implementovat vlastní funkce, jsou tam k dispozici jen složky datové struktury pro čtení a zápis. Pro základní typy už funkce jsou definovány. Jejich přístupnost na volbu zatím není možná. Řešením jsou tzv. pervasivní funkce — tyto funkce jsou přítomny pro manipulaci s daty, které jsou v jazyce přímo definovány, nicméně uživatel může vytvořit funkci stejného jména.

Rozlišují deklarace a definice, jedna z nich musí předcházet prvnímu užití. Někdy jsou na začátku programu, jindy na začátku bloku.

Návrh programu

Strukturované jazyky umožňují použít některé principy dobrého programovacího jazyka. Důležité vlastnosti:

- Vznik podprogramů - zjednodušuje rekurzi - první možnost implementace, ukrytí implementace - před ostatním kódem, nelze do podprogramu náhodně vejít, odluka od hlavního toku programu - jednodušší a bezpečnější modifikace
- Lokální proměnné v zanořených blocích - proměnné stejného jména, různého typu, různého přístupu k nim.

Tyto vlastnosti vedou k minimalizaci datových konfliktů, rozvoj týmové spolupráce.

Nevýhoda: program tvořen jen jedním souborem, lze ale vytvářet knihovny často používaných funkcí. Větší nároky na programátora.

Typy a jejich zpracování

Jsou to jazyky typované, u každé entity musí být typ uveden explicitně. Od definice/deklarace zůstává za běhu programu tentýž, ovšem jsou vestavěné funkce a operátory a případně konvence v rámci pravidel jazyka, které převod umožňují. Podle přísnosti jazyka některé převody probíhají automaticky nebo je nutné to udělat explicitně.

Novým typem je ukazatel, nese adresu libovolné paměťové buňky v cílovém systému. V úvahu je třeba brát segmentaci paměti (v PC se používaly 16bit CPU, paměť 1 MB, 16 bity maximálně adresováno 64 kB, takže se používaly 2 16bit čísel pro adresaci, jedno udávalo segment a

druhé posunutí), stránkování, oddělené datové, kódové prostory a zarovnání paměti a velikost slova.

Tok řízení programu

Tyto jazyky definují příkazy pro vytvoření bloku příkazů. Blok může obsahovat i definice lokálních proměnných. Tyto konstrukce lze vnořovat a jelikož nahrazují příkaz, lze je začlenit do ostatních programových konstrukcí.

Smyčky/cykly, podmíněné příkazy a příkaz násobného větvení lze vzájemně vnořovat a kombinovat. Snaha eliminovat goto. Pro ukončení nebo vynucení dalšího průchodu smyček je break a continue.

Zanoření je i možné pro definice funkcí. V případě vnoření na úrovni textu programu mluvíme o statickém zanoření a jeho úrovni. V době běhu programu, když se jednotlivé funkce volají, případně rekurzivně sebe potom hovoříme o dynamické úrovni zanoření.

6 Sedmá přednáška

6.1 Modulární jazyky

Je to další vývojový stupeň, rozvoj týmové spolupráce.

Představitelé

Modula-2, Ada, C, Turbo-Pascal, Free-Pascal, jazyky typu ML (mají vlastnosti jazyků funkcionálních).

Formální báze

Primárně ji nemají kromě vyjímek - jazyky vznikající v současnosti nebo blízké minulosti, například jazyk ML.

Syntaxe

Podávána formální či semiformální cestou jako u jazyků strukturovaných. Typicky užívané popisy: (E)BNF, syntaktické grafy, formální gramatiky (zejména bezkontextové).

Sémantika

Podávána neformálně. Výjimka u těch s formální bází. Díky vzniku modulů je umožněn vznik i knihoven funkcí, to vede k jejich standardizaci - vyšší využitelnost v praxi. Doplněna příklady.

Formální V&V

Tam, kde chybí formální báze, se využívá Floyd-Hoare logika. Pro rozsáhlejší důkazy je třeba modifikací pro modulární jazyky.

Softwarové inženýrství

V tomto ohledu se modulární jazyky dostaly na vrchol, kocept modularizace využíván dodnes. Tudíž umožňuje týmovou spolupráci a dekompozici problémů, ale vyžaduje pochopení a aplikaci postupů během návrhu programu, pro dekompozici problému na nezávislé části se používá metoda **shora dolů** a **zdola nahoru**.

Každý modul se tak může stát **nezávislým projektem** a mít vlastní zadání, tým, návrh, implementaci.

Modularizace přináší i nový typ chyb:

- Nemožnost spojit moduly do výsledného celku díky špatné specifikaci modulů, či nevhodné dekompozici
- Program lze sestavit, ale díky špatné logice řízení toku programu celek nepracuje správně, či ”zamrzává”
- Nevhodné užití návrhových metodologií pro modularizaci, či vazby mezi moduly jak na úrovni dat, tak na úrovni toku řízení – grafy toku řízení, konečné automaty, událostí apod.

Výhody, nevýhody

+ Dají se uplatnit tam, kde se nedá použít OOP.

+ Vhodné pro týmovou práci

+ Jsou velmi často ve formě překladače a k němu náležejícímu spojovacímu programu(linker).

+ Snadná údržba

-Extrémně velké programy ztrácejí výhody oproti OOP a je třeba využít jej

Datové a řídící abstrakce

Využívají vlastností nejčastěji z blokově strukturovaných jazyků. Jako nový typ lze brát **modul**. Z uživatelského hlediska to není typ, z formálního ano.

Role knihoven

Moduly umožňují vznik **knihoven**. Poskytují sadu funkcí pro manipulaci s určitým abstraktním typem, případně poskytují jakési zázemí pro danou výpočetní tématiku. Umožňuje skrývat implementaci před uživatelem v binární části modulu.

Skrývání dat

Modul se stará, jak o data definovaná a nabízená modulem, tak o data skrytá v modulu. Mohou vznikat chyby v případě, že neznáme zdrojové kódy modulů nebo když je jeden modul využit vrámci jedné knihovny více moduly nebo pokud modifikujeme modul, který jsme nevytvořili.. Pokud je problematika pro 1 modul široká, tak místo něj vzniká knihovna, plní podobnou funkci.

Rozhraní modulu

= popis, jak využít funkčnost modulu. O vyvážených datech, datových typech a funkcích z modulu rozhoduje rozhraní (interface). Obsahuje i deklarace pro jednotlivé konstrukce, které lze využít vně modulu.

Návrh programu

Závislosti mezi moduly

Moduly lze skládat do hierarchií, primárně se používá stromová hierarchie, někdy se jedná o acyklický orientovaný graf (DAG). U nestromových struktur vzniká problém s násobným užitím modulu, který ale podobné zacházení nepodporuje

Deklarace typů

většina jazyků nepodporuje skrývání definic typů na vhodné úrovni, často vůbec. Typy jsou vyváženy kompletně a s typy může být manipulováno vně modulu. Může být ”řešeno” pomocí ukazatelů.

Skrývání implementace

Další výhodou knihoven je možnost skrývání implementace v implementační části modulu. Doba učení je stejná jako u strukturovaných jazyků. Nevýhodou může být nemožnost sdílení jednoho modulu více dalšími, pokud na to není připraven a zdrojové texty jsou nedostupné.

Typy a jejich zpracování

Podobné paradigmatu blokově strukturovaných jazyků. Modulární jazyky nepřinášejí samy o sobě žádné typové konstrukce, abstrakce. Při překladu hraje velkou roli umístění jednotlivých datových struktur/proměnných v paměti. U modulárních jazyků ale není možné v době překladu modulu určit adresu entit, ta je známa až v době sestavení celého cílového programu (linker). Dále není známa velikost modulu, počet instancí modulu.

Tok řízení programu

Opět v mnoha věcech stejné jako jazyky, na kterých je to založeno - základní struktury pro řízení toku programu. Oproti nemodulárním jazykům však přibývá jiný fenomén — je možné předávat (díky volání funkcí/procedur) řízení z jednoho modulu do druhého. V době překladu je však obecně znám pouze jeden modul, případně jejich omezené množství.

Proto vyvstává problém, jak:

- předat parametry;
- vrátit/přijmout výsledek funkce;
- využít registry pro optimalizaci.

Důsledky jsou potom:

- Přiřazování registrů: děje se pouze v rámci funkce – pokud nejsou využity specializované registry v nějakém modulu, možnost je přiřadit globálně na závěr
- Standardizace předávání parametrů: volající i volaný dodržují určité konvence
- Standardizace předávání výsledku: volající i volaný dodržují určité konvence

Způsoby předávání parametrů:

- Hodnotou, výsledkem, hodnotou a výsledkem: vytváří se kopie, které hodnota se případně kopíruje zpět
- Odkazem: předává se ukazatel na danou proměnnou
- Jménem: předává se dvojice přístupových metod k dané entitě: pro zápis a čtení hodnoty

Překladače

Kopírují se vlastnosti, které jsou dány charakterem jazyka z hlediska typů a řídících struktur.

Některé operace nejsou možné díky modularitě:

- Některé druhy globálních optimalizací nelze provádět díky neznalosti kompletního programu
- Volání a návrat z volání funkcí musí probíhat přes standardní rámec
- Objevuje se spojovací program, který přebírá některé role překladače

Lexikální analýza

Objevuje se vlastní textový preprocesor, zprvu implementován jako separátní program spouštěný ještě před vlastním překladačem, v pozdějších implementacích součástí lexikálního analyzátoru, vysoké nároky.

Syntaktická analýza

Jazyky modulární jsou založeny na bezkontextových gramatikách, syntaktická analýza takže i zpracování probíhá standardním způsobem. Novým prvkem je označování symbolů vyvážených z modulu a dovážených do modulu.

Sémantická analýza

Generování cílového kódu probíhá do jeho relativní formy s tím, že je nutný ještě průchod spojovacího programu

Spojovací program - Linker

Linker spojí všechny moduly a knihovny do jednoho bloku, sestavuje cílový program.

Mohou se zde objevit chybová hlášení:

- Chybějící symbol - nějaký modul požaduje symbol, který ale žádný jiný modu nedefinuje.
- Vícenásobná definice - některý ze symbolů je exportován více moduly,
- Rozdílné typy symbolu - ne všechny linkery tuto chybu detekují, konflikt mezi typem souboru, který je vyvážen jedním a dovážen jiným modulem

Popis linkeru:

- Vstup linkeru: je relativní kód, který je výstupem překladu, má vazby na proměnné, funkce uskutečněny přes tabulku. Často obsažena i tabuka pro lokální symboly v modulu.
- Úloha linkeru: spojování a vzájemné provazování symbolů, které něco požadují, se symboly, které jiný modul vyváží. Během provazování se provádí kontrola na existenci, násobné nebo chybějící definice. Výstupem je relokatibilní kód nebo absolutní kód.

- Absolutní kód: linker musí kromě vzájemného propojení symbolů určit i adresu všech entit programu a zmodifikovat kód, který na ně odkazuje.

Vsuvka

Vliv modularizace na optimalizaci kódu v době jeho generování, tak vidíme, že některé optimalizace nelze uskutečnit. Běžně se překladače (a příslušné spojovací programy) tuto situaci nesnaží řešit.

Interpret

Interprety nejsou typické, pouze pro školní nebo výzkumné projekty. Častěji se vyskytují překladače do abstraktního kódu, ten je dál interpretován. Virtuální kód jenž je interprerován virtuálním strojem.

- Interprety virtuálního kódu – vylepšuje vlastnosti interpretů, protože eliminuje nutnost opakované analýzy zdrojového textu, přitom umožňuje rychlou modifikaci kódu jednotlivých funkcí, možnosti ladění programu jsou teoreticky také na vysoké urovni.

7 Osmá přednáška

Imperativní programovací jazyky vyžadují, aby programátor specifikoval, co se dá zpracovat a jak se to má zpracovat. Zpracovávání se odehrává jako postupná modifikace vnitřního stavu programu, krok za krokem. U některých jazyků je standardem dáno, kdy je dosaženo jistého selvenčního bodu výpočtu a co která proměnná obsahuje za hodnotu. Například v jazyku C je konec kroku označen středníkem.

Deklarativní paradigma přitom vyžaduje, aby programátor specifikoval jen co se má zpracovat. Interní stav výpočtu a pořadí vyhodnocování částí programu je ”neznámé” pro uživatele. Základní vyhodnocovací strategii známe. Postup vyhodnocení je určen strategií a potom překladačem a aktuálním stavem výpočtu. Tyto jazyky nemají sekvenci kroků, proto ani cyklus, takže opakování výpočtu se děje pomocí rekurze.

Zástupci deklarativních jazyků**: funkcionální jazyky, logické jazyky, jazyky pro definici a manipulaci s daty, jazyky s grafickým rozhraním.**

7.1 Čistě funkcionální jazyky

Například jazyky: Hasell, Miranda, Gofer, HP, Hope, Orwell.

Jejich **formální bází** je **lambda kalkul**, na který lze funkcionální jazyk přeložit.

Základní stavební jednotkou jak pro řízení toku vyhodnocení, tak pro data je zde funkce. Pro stejnou kombinaci parametrů vrací stejný výsledek.

Funkce: faktoriál, Fibonacciho posloupnost, Quicksort.

7.2 Logické jazyky

Například jazyky: Prolog, Parlog, Godel, CLP, KL1

Jejich **formální bází** je **podmnožina predikátové logiky**. Ne všechny její konstrukce jsou zachyceny. Výpočetní síla je přesto stejná jako u ostatních jazyků.

Základními stavebními bloky jsou entity jako klauzule, predikáty, termy apod. Z nich se sestavuje program. Na základě vyhodnocovací strategie dochází k vyhodnocení programu – logické odvození, kdy musejí být splněna stejná pravidla, jako by se odvození dělo ručně na základě pravidel predikátové logiky. Logické jazyky nejsou určeny pro početní operace. Využívá se rekurze.

7.3 Jazyky pro definici a manipulaci dat

SQL, QUEL, QBE.

Nějaký formální základ může u těchto jazyků chybět, ale stejně tak to může být relační algebra či kalkul, nebo i jiný formální prvek, který tyto nějak doplňuje, či rozšiřuje.

Tyto jazyky mají vazbu na přirozený jazyk. Příkaz, nebo častěji dotaz zapsaný v tomto jazyku potom často pasivně dekódujeme poměrně dobře.

Tyto jazyky většinou nejsou výpočetně úplné. Takže v nich nelze popsat libovolný algoritmus.

7.4 Jazyky s grafickým rozhraním

Zdrojový tvar programu je v grafické formě a textová reprezentace se buď nepoužívá nebo vůbec neexistuje. Patří sem jazyky pro popis toku dat nebo signálů. Tyto jazyky nemusejí mít jméno, nebo přejímají jméno podle vývojového prostředí.

Formálně jsou tyto systémy založeny na reaktivních systémech, komunikujících sekvenčních procesech či jiných matematicko-analytických formalismech.

Mezi typické vlastnosti patří, jak lze očekávat, manipulace s grafickými entitami, jejich kombinace a propojování liniovými entitami podobně, jako ve vektorovém grafickém editoru.

8 Devátá přednáška

8.1 Funkcionální programovací jazyky

Paradigma funkcionálních programovacích jazyků: Funkce je základním a jediným výrazovým prostředkem pro popis a definici algoritmů i dat.

Formální báze je λ-kalkul, který je v základním pojetí beztypový. Nicméně je možné zavést typovou teorii, v jednoduchém nebo i složitějším tvaru ⇒ proměnné 2. i vyšších řádů.

Tyto jazyky jsou silně typované, ale typy se neuváději explicitně, jsou odvozovány (typová interference). U funkcionálních jazyků se pro tyto účely používá jazyk core-ML, což je lambda kalkul v jiném hávu.

Syntaxe

Syntaxe je často velmi jednoduchá, zejména ve srovnání s jazyky imperativními. Není často ve formální podobě. Spíše určena pro experty.

Sémantika

U sémantiky je často popis neformální, případně s odkazem na formální bází jazyka, která dodává potřebný dovysvětlující rámec. Formalismy a mechanismy, které definují sémantiku:

- Formální systém automatické typové inference
- Formální důkaz toho, že celý jazyk a jeho zpracování a vyhodnocení je vnitřně bezesporné a správné
- Denotační sémantika

Data a jejich zpracování

Datové typy ve funkcionálních jazycích jsou zastoupeny od jednoduchých až po rekurzivní. Předdefinované typy nejsou shodně s těmi z imperativních jazyků. To, co chybí, doplní knihovnami nebo abstraktními datovými typy.

Mezi bázové typy patří klasické skalární nebo z nich přímo odvozené. Jsou to celá čísla, čísla s plovoucí desetinnou čárkou, znaky atp.

Jsou zastoupené ale i strukturované:

- Seznamy ⇒ nad libovolným typem, kvůli silné typovosti homogenní struktura
- N-tice ⇒ heterogenní datové struktury, nemají explicitní pojmenování

Vedle vestověných typů existují i Uživatelské typy:

– Výčtové typy
– Záznamy, i pojmenované
– Variantní záznamy (jméno u každé varianty)
– Rekurzivní datové typy (mohou kombinovat předchozí vlastnosti)

Vstupy a výstupy

Jsou ovládány speciálními typy, které jsou zastoupeny v knihovnách, ale často implementovány v jiném jazyce. Jsou 2 hlavní principy pro realizaci operací:

- Kontinuace: často se pojí s proudovými vstupy a výstupy, idea spočívá ve vytvoření dvou funkcí, které pokračují v závislosti na výsledku vykonání operace
- Monády: aby bylo zachováno základní paradigma funkce, kdy její výsledek je dán kombinací parametrů, nikoliv historií, akce je potom něco co nám to zaručí

Pole

Pole se díky své imperativní povaze typicky nevyskytují. U ne čistě funkcionálních jazyků se setkáváme s tzv. mutovatelnými položkami, naopak u čistě funkcionálních implementace zcela chybí a nebo se k nim přistupuje jako k monádům.

Typové proměnné

Dobrá vlastnost funkcionálních jazyků je implementace generických algoritmů díky polymorfismu na úrovni funkcí. To je umožněno možností mít typové proměnné místo konkrétních typů. Konkrétní typ se doplní za proměnnou až podle parametrů v místě aplikace.

Řídící struktury

Základními strukturami jsou funkce, které se mohou vyskytovat ve zdrojovém textu libovolně. Rekurzí se dá na funkce odvolávat pouze pokud již byly definovány. Funkce lze staticky vnořovat a vytvářet tak lokální

funkce, které pomáhají v práci globálně dosažitelným funkcím. Deklarace se spíše nevyskytují, nebo vůbec neexistují.

I ve funkcionálních jazycích se využívá **modulů**, kde je jejich rozhraní odvozeno z modulů a není tedy nutné explicitně nějaké vytvářet. Může se odvodit ze zdrojového textu nebo binární formy modulu, záleží na jazyku.

Větvení výpočtu (if then else, další větvení) musí být úplné proto, aby pro všechny kombinace vstupů bylo možno dospět k výsledku.

Návrh programu

Základním charakterem je, že zpracování dat je navrženo a implementováno zcela odděleně od implementace vstupně-výstupních operací a k vzájemnému propojení dojde zcela na závěr celého vývoje. Oproti imperativním jazykům neleží tyto operace „vespodÿ aplikace, ale spíše „nahořeÿ nad částí pro zpracování dat.

Data jsou zpracovávána postupnou manipulací a změnou dat sekvencí funkcí, které jsou postupně aplikovány na zpracovávaná data — dochází k vysokému vyžití funkcí vyššího řádu a mapovacích funkcí, kdy jsou určité operace mapovány na všechny členy nějaké rekurzivní datové struktury. Výsledný program je tak vlastně jediný výraz, který je třeba vyhodnotit, abychom dostali výsledek.

Při návrhu programu se techniky známé z imperativního programování prakticky nedají uplatnit. Například je třeba se rozloučit s tím, že

- proměnná je pojmenovaná adresa v paměti,
- operace jsou prováděny v pořadí uvedeném ve zdrojovém textu,
- je možné využít ukazatelů,
- známe stav výpočtu a ten manipulujeme. Naopak by měl návrh programu sledovat strategii vyhodnocení (viz oddíl této kapitoly 4.4), protože tak lze vytěžit z jazyka všechny jeho výhody, což se projeví na efektivitě zpracování.

Výhody nevýhody

+Funkcionální jazyky umožňují rychlý vývoj programů, které jsou ve srovnání s programy imperativními často „menší”.

+Díky existenci formální báze je možné provádět formální důkazy vlastností algoritmů

-Nízká popularita čistě funkcionálních jazyků

-Operace, pro které nejsou knihovny, je třeba implementovat např v C a potom přes rozhraní pro import modulů z jiných jazyků na ně přistupovat.

-Odpor programátorů, nemožný přechod.

Překlad, interpretace

Překlad, interpretace překlad je velmi náročný, přední část je velmi zesílená, syntaktická a sémantická analýza probíhá v podstatě několikrát, na různých úrovních jazyka. Nejsložitější úlohou překladače je převést deklarativní popis na serializovaný výpočet realizovaný na von Neumannovské architektuře dnešních počítačů.

Proč může syntaktická analýza proběhnout několikrát? Postup zpracování funkcionálního jazyka:

1. Expanze syntaktických zkratek

2. Redukce výrazových schopností jazyka, probíhá v několika krocích ⇒ převedení výrazově bohatých konstrukcí na jednodušší konstrukce, až zbyde malá množina základních

3. Automatické odvození typů, typová kontrola

Vyhodnocování strategie jsou podstatné při generování kódu, jelikož určují klíčové vlastnosti jazyka. Ovlivňují tok řízení, předávání toku řízení a tvorbu programu.

Volání hodnotou je strategie známá z imperativních jazyků. Parametry jsou vyčísleny před voláním funkce. Počet vyhodnocení nějakého výrazu je minimálně jeden, horní omezení není.

Z této strategie se vyvinulo **striktní vyhodnocení**. Sdílí se hodnoty výrazů, k vyhodnocení dochází právě jednou.

Volání v případě potřeby Strategie, kdy dochází k zavolání funkce bez toho, aby se vyhodnotily parametry ⇒ jsou pro ně vytvořeny kontextové obálky. Parametry se tedy vyhodnotí až v případě, kdy ho funkce potřebuje. Počet vyhodnocení je tedy minimálně žádný a horní mez opět není určena. Z této strategie se vyvinula tzv. **lazy evaluation**, která výrazy vyhodnocuje nejvíce jedenkrát.

Další strategie jsou významově někde mezi předchozími dvěmi. Využívají často analýzy striktnosti parametrů k tomu, aby dopředu předepsaly strategii vyhodnocení pro jednotlivé případy volání funkcí.

Generování kódu: 2 možnosti

- Přímá vazba na cílovou platformu – velmi obtížné, nutné další snižování síly jazyka, úzce spjato s knihovnami pro běh programu, některé vlastnosti jazyka mohou tak vymizet
- Speciální cílový kód určený pro funkcionální jazyky – překlad je jednoduchý, interpretace takového kódu, případně jeho semi-interpretace

Interprety funkcionálních jazyků v čisté podobě neexistují, v praxi se jedná o překlad do specializovaného virtuálního kódu určeného pro funkcionální jazyky. Takový virtuální kód může být třeba graf, reprezentace funkcionálních programů takovouto formou je přirozená a jednoduchá. Zpracování programu je potom redukce takového grafu, ale kvůli rychlosti zpracování se používá virtuální kód, který je lineárním zápisem grafu. Dva příklady virtuálního kódu:

- Three instruction code
- Spineless tag-less G-machine

Zpracování typů je ve funkcionálních jazycích zcela speciální kategorie. Jedná-li se o netypovaný jazyk, kontrola se prování až za běhu, jako u imperativních jazyků. Často jsou ale jazyky typované, využívá se tedy typového odvození a striktní typové kontroly.

Typové třídy souží k řešení problému se specializovanými operátory ohledně různých typů operandů. Sdružují funkce a operátory do skupin, kde jsou svázány jistým společným jmenovatelem.

Formální verifikace Díky vztahu k λ-kalkulu je možnost formálně verifikovat programy na výrazně vyšší úrovni.

Možnosti a vhodnosti využití

+Nabízejí řadu standardních vlastností, které jsou vhodné pro týmovou spolupráci a tvorbu rozsáhlých programů ⇒ modularita, strukturované datové typy, apod. Vhodné vastnosti pro detekc chyb v raném stádiu.

+Vhodné pro složitější díla jednoho programátora

+Pro simulace , řešení extrémních situací

-Nízká popularita oproti jiným jazykům.

-Náročné na učení a porozumění

9 Desátá přednáška

9.1 Logické programovací jazyky

Definice Paradigma logických jazyků Program v logických programovacích jazycích se sestává z predikátů případně kombinovaných s omezeními, které pracují nad atomy, termy či predikáty.

Představitelé Prolog, Parlog, Godel, CLP, KL1C

Formální báze je predikátová logika. Je vždy nějak omezená. Její přítomnost je nezbytná, protože původní určení těchto jazyků bylo automatizovat formální důkaz v predikátové logice.

Jazyk predikátové logiky se syntakticky skládá z termů, predikátů a formulí. Termy jsou konstanty, proměnné, funkční symboly.

Term v predikátové logice je:

- Každá proměnná
- Výraz f(t1, . . . , tn), je-li f n-ární funkční symbol a t1, . . . , tn jsou termy
- Každý výraz získaný konečným počtem aplikací přechozích dvou pravidel, nic jiného term není

Jelikož jsou termy definovány konečným počtem pravidel, říkáme, že to jsou konečná slova.

Základní **predikáty** jsou potom pravda (T, true) a nepravda (F, false). **Formule** potom definujeme z predikátů pomocí logických spojek a kvantifikátorů.

Formule v predikátové logice je:

- výraz p(t1, . . . , tn), je-li p n-ární predikátový symbol a t1, . . . , tn jsou termy, tento výraz je zván taktéž atomická formule
- výraz: $\neg$A, (A $\wedge$ B), (A $\vee$ B), (A $\Rightarrow$ B), (A $\Leftarrow\Rightarrow$ B), pokud A a B jsou formule
- výraz: $\forall$x : A, $\exists$x : A, pokud x je proměnná a A je formule.

Sémantika predikátové logiky není dána hned, ale nejdříve je třeba definovat interpretaci. Interpretace přiřazuje význam konstantám, funkcím a predikátům. Také je nutné volným proměnným přiřadit hodnotu. O splnitelnosti formule nejde vždy zcela rozhodnout, proto přichází částečná rozhodnost.

Částečná rozhodnutelnost znamená, že nejde vždy zcela rozhodnout, zda-li je formule splnitelná či platná.

Splnitelnost formule je za těch podmínek, že existuje taková interpretace, která je modelem. Formule je modelem tehdy, když je pravdivá pro každé ohodnocení volných proměnných.

Platnost formule znamená, že je formule pravdivá při každé interpretaci.

Syntaxe je neformální s popisy a příkladech. Můžeme se setkat i s formlní definicí syntaxe pomocí vhodné formy. Je to tak kvůli snaze prezentovat jednoduchost syntaxe jazyka. Bezkontextové gramatiky.

Sémantika není formálně vůbec. Vysvětlován jazyk na příkladech v textu nebo přímo součástí instalace pro vyzkoušení.

Data a jejich zpracování

Obsahují klasicky rozsah datových typů od jednoduchých až po složité složené struktury. Aneb celá čísla, případně znaky. Pomocí knihoven lze tento rozsah rozšířit o další typy. Například o čísla s plovoucí desetinnou čárkou.

Jako prvotní datovou abstrakci můžeme brát **termy**. Lze je pomocí implementačních triků konvertovat na predikáty a vyhodnotit. Díky této vlastnosti mohou logické jazyky provádět sestavování části programu až za jeho běhu. Termy mohou být brány jako pojmenované n-tice, např: osoba('Jan', 'Kadlec', 1969).

Termy lze vnořovat, uzavírají se nad atomy. Atomy hrají velikou roli, neboť mohou nést prakticky jakoukoliv hodnotu.

Seznamy i **literály** se zapisují podobně a práce s nimi je prakticky shodná. V logických jazycích jsou ale seznamy většinou heterogenní datová struktura.

Jsou různé techniky práce se seznamy, či termy nebo obecně daty:

- Rozpoznávání a unifikace vzorů17 v hlavičce klauzule – práce s parametry predikátu
- Převodem na jiný typ – např. převod mezi termem a sezamem
- Přímý přístup k termům

Řídící struktury Základní struktury jsou predikáty/klauzuje, které mají libovolné rozmístění po zdrojovém textu. Deklarace nejsou povoleny – nejsou potřeba. Zjednodušený typ deklarace můžeme najít pro deklaraci dynamických klauzují, pro deklaraci predikátů exportovaných z modulu. Typová kontrola nemá smysl.

Řízení výpočtu je dáno cílem výpočtu, kterého chceme dosáhnout, ten pak sestává z dílčích podcílů. Pokud dospějeme k výsledku, cíl uspěl, podobně u podcílů. V opačném případě cíle selhávají.

Vlastní tok řízení/vyhodnocení programu je ovlivněn výběrem predikátu/klauzulí na základě unifikace vzorů a na základě uspívání/selhávání těla dané klauzule. Tělo klauzuje je specifikováno řadou predikátů. Strategie vyhodnocení rozhodne v jakém pořadí budou predikáty zpracovány, typicky je to zleva doprava. Typ operátoru,

případně podmí- něných vyhodnocení18 může řídit lokální směr vyhodnocení.

Uvažujeme-li Hornovy klauzule, pak zápis v Prologu je tento:

a(. . .) :- b1(. . .), . . ., bm(. . .).

Což znamená, v jazyce predikátové logiky:

$\forall$(a(. . .) $\leftarrow$ b1(. . .) $\wedge$ . . . $\wedge$ bm(. . .))

Dále můžeme upravit:

$\forall$(a(. . .) $\vee$ $\neg$(b1(. . .) $\wedge$ . . . $\wedge$ bm(. . .))) $\forall$(a(. . .) $\vee$ $\neg$b1(. . .) $\vee$ . . . $\vee$ $\neg$bm(. . .))

Výsledný tvar je možné jednodušeji programově vyhodnotit, protože vyhodnocením každého podcíle ubude z vý- razů19 jeden člen.

Backtracking nastává v případě, že při vyhodnocování dojde k selhání. Vrací se ve výpočtu zpět a hledá jiné možnosti uspění již dřívě úspěšných podcílů. Teprve až uspějí všechny podcíle, tedy zpětné navracení se zastaví na cíli, který znovu uspěl, a další podcíle v řadě již uspějí, uspívá i celý cíl. Pokud při zpětném navracení žádný podcíl znovu neuspěje, tak selhává celý cíl.

Návrh programu a implementace může probíhat jak shora dolů, tak zdola nahoru, nebo kombinací. Díky absenci deklarací je modifikovat program jednoduché, protože je možné přidávat kamkoli do textu programu. Prolog umí měnit obsah programu za běhu, má vestavěné predikáty pro odstranění i pro přidání nových.

Výhody/nevýhody

+jsou jedinou možností automatizované podpory důkazu.
+nabízejí ve vyhodnocovací streategii zpětné navrácení.
+u Prologu je standardizace.
+používají garbage collector
-neexistuje čistá kompilace
-nižší rychlost vyhodnocení
-výjimečná existence modularity
-využitelnost pouze ve speciálních oblastech - simulace, verifikace

Překlad, interpretace

Překlad do vyšších programovacích jazyků či do mezikódu. Překlad do binární formy je nerealizovatelný. Interpretace je častější, přímá interpretace nemožná. Dochází k transformaci do interní podoby a uložení do databáze. Interprez potom nad ní pracuje a přistupuje do databíze.

SLD rezoluce je vyhodnocovací strategie pro jazyk Prolog. Vstupem jsou Hornovy klauzule. Vyhodnocení probíhá:

1. Klauzule jsou z databáze vybírány shora dolů, dle umístění ve vstupním textu.

2. Těla predikátů jsou zpracovávána zleva doprava

Jedná se tedy o prohledávání do hloubky – dokud neuspěje první podcíl, ostatní zůstávají nedotčené. Příklad:

1: a(...) :- b(...), c(...), d(...).
2: a(...) :- d(...), e(...).
3: b(...) :- c(...), d(...), c(...).
4: b(...) :- e(...).
5: c(...) :- d(...).
6: c(...) :- e(...), d(...), c(...).
7: d(...).
8: d(...) :- e(...), d(...).
9: e(...).

Řešení: Chceme vyhodnotit predikát a, vyhodnocení tedy začne bodem 1.

a(...) -->> b(...), c(...), d(...).
Dále je nutné vyhodnocení prvního podcíle b(...).
b(...) -->> c(...), d(...), c(...).
Stejný nechť je postup s podcílem c, tedy:
c(...) -->> d(...).
Tady se dostáváme do stavu, kdy je potřeba ověřit klauzuli d, která je ale bez těla (říká se jí fakt). Která ale selže:
d(...).
Musíme tedy uplatnit druhou klauzuli predíkátu d, tedy:
d(...) -->> e(...), d(...).

Průběhem tohoto postupu dochází k unifikaci mezi proměnnými a hodnotami, které se objevují na příslušných pozicích parametrů. Pokud dojde k uspění cíle, tak kromě oznámení úspěchu je vypsána substituce pro nejvyšší úroveň proměnných.

Unifikace je v logických jazycích velmi důležitý postup. Probíhá při hledání hlavičky podcíle v databázi klauzulí, kdy klauzule očekává

parametry jistého tvaru nebo je přijímá, případně v době explicitního vynucení unifikace, kdy jsou proti sobě postaveny termy/proměnné.

Dochází tak k vzájemnému provázání mezi proměnnými a dalšími entitami v celém programu. Unifikace probíhá vždy na úrovni termů a hledá se nejobecnější unifikátor.

Nejobecnější unifikátor je takový, že neexistuje žádný jiný unifikátor, který by pro nějakou část unifikace nabízel obecnější řešení, přitom dva unifikátory jsou nejobecnější, pokud se liší pouze přejmenováním volných proměnných.

Typické vlastnosti unifikace v logických jazycích:

- Kontrola výskytu: ověřuje se, zda proti sobě stojí dvě identické proměnné, neboť to může v řadě případů vést k nekonečným termům, apod.
- Hloubka unifikace: unifikace neprobíhá na libovolně rozsáhlých termech
- Zpracování cyklických a nekonečných struktur: často zakázáno a úzce souvisí s kontrolou výskytu

Robinsonův algoritmus

```
Vstup: Δ, množina literálů
Výstup: μ, mgu nebo selhání/neúspěch

μ = []                                              (prázdná substituce)
dokud v μ(Δ) existuje nesouhlasný pár
    najdi první nesouhlasný pár p v μ(Δ)
    pokud v p není žádný volná proměnná
        skonči selháním unifikace
    jinak
        nechť p = (x,t), kde x je proměnná
        pokud se x vyskytuje v t                    (kontrola výskytu)
            skonči selháním unifikace
        jinak
            nastav μ = μ ∘ [t/x]
vrať μ
```

Další strategie Jiné strategie vyhodnocení jsou například u systémů CLP, kdy vý- sledkem není seznam substitucí, ale omezení pro proměnné volné v cíli, které vymezují jejich hodnoty. Tato omezení jsou typicky reprezentována jako množina rovnic, či nerovnic, nebo dalším způsobem.

Zpracování typů se provádí až ve chvíli kdy to nějaká operace vyžaduje. Počet základních typů je nízký a jen ojediněle je může programátor obohatit o svoje. Síla termů a struktur je v tomto případě dostačující.

Formální verifikace programů v logických jazycích neprovádíme, jedná se o formule v predikátové logice, tudíž jsme omezeni samotnou predikátovou logikou. U strukturálních algoritmů je možné použít strukturální indukci.

Možnosti a vhodnosti využití Pro větší projekty se tyto jazyky mohou hodit, ale nemusí. Poměrně rozsáhlé programy mohou být vytvořeny jedním programátorem.

Při tvorbě programu je potřeba přemýšlet, jakým způsobem se vybírá klauzule pro další vyhodnocení, stejně tak to, že systém má vestavěnou strategii zpětného navracení.

Je potřeba si dávat pozor na efektivitu a optimálnost programu, protože vyhodnocovací mechanismus uplatňuje slepou strategii výpočtu, kterou řídí program, žádné zefektivnění není.

10 Jedenáctá přednáška

10.1 Ostatní deklarativní jazyky

Jazyky pro definici a manipulaci dat

Rozdělujeme je do 2 druhů: DDL — Data Definition Language a DML — Data Manipulation Language

Mají velmi často **nižší vyjadřovací sílu** než Turingovy stroje (není možné jimi pospat libovolný algoritmus).

I když existuje formální základ jejich syntaxe, tak často jsou velice blízké přirozeným jazykům, respektive angličtině. Proto je také možné to, co popisují, celkem jednoduše přečíst a pochopit.

Vytvářené akce/operace/funkce/dotazy/příkazy typicky nemají pojmenování a není možné se tedy na ně odvolávat a takto strukturovat. Často se to obchází tím, že jsou součástí skriptovacího jazyka, kde je toto nějakým způsobem umožněno.

Často jazyky obsahují velké množství klíčových slov, funkcí či operátorů. Různé databázové systémy jsou různě rozšiřovány a upravovány, aby poskytly vlastnosti vycházející z implementace SŘBD a nabídly tak nějakou výhodu oproti ostatním implementacím.

Existuje gramatika, která rozhoduje u zpracování těchto jazyků. Syntaktická analýza se pak dá zpracovat běžnými prostředky. Sémantická analýza je závislá na okamžitém stavu SŘBD, pokud se jedná o dotaz,

který čte z báze dat, tak výsledkem sémantické analýzy a přípravy vykonání dotazu je i postup v jakém se jednotlivé kroky mají vykonat.

Požadavky jsou vyhodnocovány okamžitě s tím, že požadavek je interpretován, po kompletní analýze a převodu na postup vykonání základních operací.

Distribuované databáze vyžadují perfektní přípravu dotazu, neboť přesun velkého množství dat po síti by byl pochopitelně nežadoucí, zvláště pokud by to bylo zbytečné. Kromě klasických optimalizačních postupů se nasazují postupy pro paralelní vyhodnocení.

Databáze pro speciální typ dat jsou například prostorové, temporální, multimediální, které disponují vestavě- nými operacemi pro manipulaci se specifickým typem dat.

Jazyky s grafickým rozhraním

Představiteli těchto jazyk jsou diagramy datových či signálových toků: DFD — Data Flow Diagrams a SFD — Signal Flow Diagrams.

Někdy jde do tého třídy zařadit i stavové diagramy, či konečné automaty.

Tvorba programu v těchto systémech se hodně podobá kreslení v programu pro vektorovou grafiku (CAD systémy, apod.). Často jsou manipulovány i podobné objekty, dají se otáčet, propojovat, apod.

Program může být uložen v textové formě (např. XML), ale není to vhodné pro přímý zápis.

Pro odladění programů je typickou součástí vývojového prostředí simulátor. Cílová platforma je často odlišná od té na které je program vyvíjen.

Analýza takovýchto jazyků probíhá během tvorby diagramu. Před simulací, či generováním cílového kódu však proběhne kompletní analýza, která zkontroluje korektnosti diagramu.

Srovnání deklarativních jazyků

Je možné vypozorovat některé společné a typické vlastnosti, které nalezneme u všech, jsou to zejména:

- Před vyhodnocením, či generováním kódu dochází k serializaci kódu, která je řešena kompilátorem, či interpretem

- Pokud je porovnáváme s jazyky imperativními, tak nám vždy nabízejí "něco navíc"
- Úroveň abstrakce je často mnohem vyšší (hardware je často úplně schován a není možné na něj nijak přistoupit)
- Naprostá většina těchto jazyků má jasnou a často i přímou vazbu na nějakou formální bázi na které je postavena

Mezi jednotlivými typy deklarativních jazyků však můžeme najít i řadu odlišnosti:

- Složitost analýzy je různá (Prolog vs. Haskell vs. SQL)
- Práce s typy a jejich zpracování je různé (Prolog vs. Haskell)
- Implicitní možnosti paralelizace či serializace se různí (Prolog vs. SQL)
- Výpočetní síla (SQL vs. Haskell)
- Úroveň deklarativy (Haskell vs. Prolog)
- Typická aplikační doména