PAPR 1 - pojmy

Lekce 1:	**Program, jeho syntax a sémantika (str. 7)**
Program Program v jazyku Scheme	- předpis s přesně daným tvarem a významem, podle kterého vzniká výpočetní proces; je napsán v programovacím jazyce - konečná posloupnost symbolických výrazů
Výpočetní proces	- proces s počátkem a koncem, probíhající v nějakých elementárních krocích, který přetváří data vstupní na data výstupní; je důsledkem vykonávání programu
Nižší programovací jazyky: 1. kód stroje 2. assembler 3. autokód 4. bajtkód	- jsou to jazyky těsně vázané na hardware počítače, neobsahují téměř žádné prostředky abstrakce 1. sekvence jedniček a nul, ve které jsou psané instrukce pro procesor - kód je vykonáván procesorem 2. vznikl jako pomůcka pro vytváření programů v kódu stroje, zápis instrukcí=zkratky 3. podobné assemblerům, poskytují programátorovi prvky, které lze považovat za rysy vyšších jazyků 4. jazyky stroje pro virtuální procesory
Vyšší programovací jazyky	- jazyky nevázané na hardware počítače (=>přenositelnost programu), s velkou mírou abstrakce, což umožňuje snadnější a rychlejší vytváření programu s menším rizikem vzniku chyb
Překladač	- načte celý program a poté provede překlad do některého nižšího jazyka (assembleru nebo bajtkódu)
Interpret	- program, který čte výrazy programu a postupně je přímo vykonává
Každý vyšší programovací jazyk dává k dispozici: 1. primitivní výrazy 2. prostředky kombinace 3. prostředky abstrakce	1. čísla, symboly 2. poskytují možnost kombinace primitivních výrazů do složitějších 3. poskytují možnost pojmenování složených výrazů a možnost s nimi manipulovat
Syntax programu Sémantika programu	- soubor přesně daných pravidel definující jak zapisujeme programy v daném programovacím jazyku - význam programu
Operace Operandy	- vyjádřuje se pomocí symbolů, např. "+"; podle pozice symbolů rozlišujeme typ notace - podvýrazy, se kterými operace provádíme
Notace: 1. infixová 2. prefixová (Scheme) 3. postfixová 4. polská (bezzávorková)	1. přirozený zápis operací, srozumitelný pro člověka, ale špatně se zpracovává strojově, př. *2 * (3 + 5)* 2. velmi dobře zpracovatelný strojově a součastně celkem čitelný pro člověka, př. *(* 2(+ 3 5))* 3. nejméně používaná, nevýhodou je složitá manipulace, př. *(2 (3 5 +) *)* 4. používá např. v PostScript, př. *2 3 5 + **
Syntaktická chyba Sémantická chyba	- chyba v zápisu programu - chyba ve významu programu
Paradigmata programování: 1. procedurální 2. funkcionální 3. objektové 4. logické 5. paralelní	- styly vytváření programů, které jsou sdíleny mezi různými programovacími jazyky 1. jedná se o paradigma imperativní, programy jsou tvořeny sekvencemi příkazů, které jsou postupně vykonávány, klíčovou roli hraje příkaz přiřazení (zápis hodnoty na místo dané v paměti) 2. programy jsou ze symbolických výrazů, výpočet je tvořen postupnou aplikací funkcí, kdy fce jsou aplikovány s hodnotami, které vznikly v předchozích krocích aplikací 3. myšlenka vzájemné interakce objektů, objekty spolu komunikují pomocí zasílání správ 4. typický je deklarativní charakter, při psaní programu je spíš popisován problém než vytváření předpisu jak problém řešit 5. souběžné vykonávání programu
Jazyk Scheme - výhody	- zápis programu je snadno pochopitelný, minimální syntaktické chyby (jednoduchá syntax), mechanická kontrola správného zápisu je přímočará
Abstraktní interpret jazyka Scheme	- množina všech elementů jazyka s pravidly jejich vyhodnocování
Symbolický výraz (S-výraz)	- skládají se z nich programy ve Scheme (základní složka všech funkcionálních jazyků) je to: a) každé číslo b) každý symbol c) jsou-li *e1...en* symbolické výrazy, pak i každý výraz ve tvaru *(e1...en)* je symbolický výraz, tzv. *seznam* - čísla a symboly (primitivní symbolické výrazy) lze kombinovat do seznamů (složené symbolické výrazy)
Element jazyka	- vznikají ze symbolických výrazů, elementem jazyka třeba je: a) interní reprezentace každého symbolického výrazu (všechny reprezentace čísel, symbolů, seznamů) b) každá primitivní procedura c) speciální formy - další definice: jsou to přípustné hodnoty, se kterými mohou (ale nemusejí) manipulovat primitivní procedury (které můžeme také považovat za jedny z elementů)
Vyhodnocování S-výrazů:	1. E je číslo - čísla označují svoji vlastní hodnotu -> vyhodnotí se samy na sebe *Eval[E] := E* 2. E je symbol - symboly jsou "jména hodnot", vyhodnotí se podle toho jestli mají/nemají vazbu F a) na svou vazbu *Eval[E] := F* b) *Chyba: Symbol E nemá vazbu* 3. E je neprázdný seznam, nejprve se vyhodnotí první prvek a zjistíme jestli je a) F1 je procedura *Eval[E] := Apply[F1...Fn]* b) F1 je speciání forma *Eval[E] := Apply[F1...En]* nebo c) F1 není ani jedno -> hlášení *Chyba* 4. E je něco jiného *Eval[E] := E*
Interní reprezentace S-výrazů	- element jazyka
Externí reprezentace elementů	- pro člověka čitelná reprezentace daného elementu; tzn. pokud je element *E* interní reprezentací symbolického výrazu *e*, tak potom externí reprezentací je právě symbolický výraz *e*

PAPR 1 - pojmy

Speciální formy	- elementy, jejichž aplikací lze generovat, příp. modifikovat výpočetní proces (řídí vyhodnocování svých argumetů), při jejich aplikaci se však nevyhodnocuje zbytek seznamu jako u procedur - jsou aplikovány s argumenty jimiž jsou elementy v jejich nevyhodnocené podobě a každá speciální forma si sama určuje jaké argumenty a v jakém pořadí (a zda-li vůbec) je bude vyhodnocovat
Cyklus REPL	- cyklus ve kterém probíhá vyhodnocování programů, které se skládají z posloupnosti symbolických výrazů 1. *Read* (načti) - pokud je prázdný vstup, činnost interpretu končí, v opačném případě reader načte první symbolický výraz, pokud je úspěšně načten, je převeden do své interní reprezentace, vzniká výsledný element *E* 2. *Eval* (vyhodnoť) - element *E* je vyhodnocen, vzniká vyhodnocený element *F* 3. *Print* (vytiksni) - provede se převod elementu *F* do externí reprezentace, výsledná reprezentace je zapsána na výstup (vytištěna) 4. *Loop* (opakuj) - vstupní symbolický výraz, který byl zpracován je odebrán a cyklus se vrací na 1. krok
1. Reader 2. Printer 3. Evaluator	1. převádí symbolické výrazy (posloupnosti znaků) na elementy jazyka (interní reprezentace symbolických výrazů) 2. vrací pro daný element (vytiskne na obrazovku) jeho externí reprezenaci 3. provádí vyhodnocení S-výrazu
Argumenty	- konkrétní elementy/hodnoty jazyka, se kterými je daná operace aplikována
Aplikace procedury	- pokud *E* je primitivní procedura a *E1...En* libovolné elementy jazyka, výsledek aplikace primitivní procedury *E* na argumenty budeme značit *Apply[E...En]*, pokud je výsledkem aplikace element F, píšeme *Apply[E…En] := F*
Definice vazeb	- prostředek abstrakce používaný kvůli zpřehlednění kódu; k zavádění nových definic používáme speciální formu *define*
Podmíněné vyhodnocování	- realizováno pomocí speciální formy *if*
Pravdivostní hodnoty	- speciální elementy, které se vyhodnocují samy na sebe; element "pravda" navázaný na symbol *#t*, element "nepravda" navázaný na symbol *#f*
Predikáty	- procedury, výsledkem jejichž aplikace jsou pravdivostní hodnoty, př. *(<= 2 3) => #t*
Nedefinovaná hodnota	- když výsledná hodnota může být jakákoliv, např. když u podmínky *if* chybí *<náhradník>*
Programovací jazyk Turingovsky úplný	- jazyk, který je zhlediska řešitelnosti problému ekvivalentní tzv. Turingovu stroji (= jeden z formálních modelů algoritmu)
Vedlejší efekt	- interakce s uživatelem
Lekce 2:	**Vytváření abstrakcí pomocí procedur (str. 41)**
Redundance kódu	- nadbytečnost kódu, projevuje se "podobnými kusy kódu", je to problém uživatelského vytváření procedur => vytvoříme novou proceduru, pojmenujeme ji a dále s ní pracujeme jako by se jednalo o primitivní proceduru (vznik UDP)
Uživatelsky definované procedury	- procedury, které lze dynamicky vytvářet během vyhodnocování programů a dále je v nich používat - vznikají vyhodnocováním λ-výrazů
λ-výraz, formální argumenty, tělo	- každý seznam ve tvaru *(lambda (p1...pn) tělo)* skládající se z formálních argumentů a těla procedury, jehož vyhodnocením vznikají nové procedury
Speciální forma lambda	- je vyvolaná vyhodnocením λ-výrazu (nic se při ní nevyhodnocuje!), vytvoří se v prostředí *P* procedura *<(<parametry>), tělo, P>*
Vázané symboly Volné symboly	- symboly, které jsou formálními argumenty λ-výrazu - symboly které se nacházejí v těle λ-výrazu
Vazby vázaných symbolů Vazby volných symbolů	- se hledají v lokálním prostředí - se hledají v počátečním prostředí
Identita Projekce Konstantní procedura Procedura bez argumentů	- procedura která pro daný argument vrací jeho hodnotu *(lambda (x) x)* - procedura, který vrací hodnotu svého i-tého argumentu *(lambda (x y z) x)* - ignoruje své argumenty a vrací nějakou konstantní hodnotu *(lambda (x) 10)* - nemá aurgument *(define no-arg (lambda () 10)) (no-arg) => 10*
Prostředí P	- tabulka vazeb mezi symboly a elementy; každé prostředí kromě Pg má vazbu na předka
Globální prostředí Pg	- počáteční prostředí, které nemá předka
Lokální prostředí Pl	- prostředí vytvořené aplikací lambdy, které má vazbu na předka (také prostředí)
Vyhodnocení elementu v prostředí	1. E je číslo - *Eval[E, P] := E* 2. E je symbol a) má vazbu *Eval[E, P] := F* b) nemá vazbu v *P*, ale pokud *P* má předka *P'*, pak *Eval[E,P] := Eval [E,P']* c) nemá vazbu v *P*, pak *"Chyba"* 3. E je neprázdný seznam a)F1 je procedura *Eval[E,P] := Apply[F1...Fn]* b)F1 je speciání forma *Eval[E] := Apply[F1...En]* c)F1 není ani jedno => "*Chyba"* 4. E je něco jiného *Eval[E,P] := E*

PAPR 1 - pojmy

Aplikace uživatelsky definované procedury	- pokud E a UDP ve tvaru *<(p1...pm), tělo, P>*, pak hodnotu *Apply[E...En]* definujeme: 1. pokud se počet *m* formálních argumentů neshoduje s *n* počtem argumentů, se kterými chceme proceduru aplikovat, aplikace končí hlášením *"Chyba"* jinak pokračujeme dalším krokem 2. vytvoří se nové prostředí Pl 3. nastaví se předek prostředí Pl na hodnotu P 4. v prostředí Pl se zavedou vazby 5. položíme *Apply[E...En] := Eval[<tělo>, Pl]*
Aplikace speciální formy	- vyhodnotí-li se první prvek seznamu na speciální formu, pak si sama zvolí, které argumenty se mají vyhodnotit a poté je s těmito argumenty aplikována
Procedura	- označení pro element jazyka, zahrnující jak primitivní tak uživatelsky definované procedury; na rozdíl od speciálních forem u nich nehraje hroli prostředí, ve kterém byly aplikovány
Primitivní procedury	- procedury, které jsou k dispozici již na začátku práce s interpretem
Procedury vyšších řádů	- procedury, které jsou aplikovány s jinými procedurami jako se svými argumenty nebo vracejí procedury jako výsledky své aplikace
Currying	- technika ve funkcionálních jazycích, která z procedury 2 argumentů vytvoří proceduru 1 argumentu, př. (*define curry+ (lambda (c) (lambda (x) (+ x c))))* - rozklad procedury dvou argumentů tímto způsobem je vhodný, když první argument je pevný a druhý může nabývat různých hodnot
Element prvního řádu	- je každý element, který může: a) být pojmenován b) být předán proceduře jako argument c) vzniknout aplikací procedury d) být obsažen v hierarchických strukturách
Pojmenované procedury	- procedury které jsou definovány se jménem (ve většině vyšších jazyků)
Anonymní procedury	- procedury nepojmenované, ve Scheme všechny => je třeba svázat jméno symbolu s procedurou pomocí *define*
Kompozice procedur Monoidální operace	- možnost naprogramovat proceduru, která provádí "složení dvou procedur": *(define compose (lambda (f g)(lambda (x)(g (f x)))))* - operace která spolu s množinou na níž jsou definované a se svým neutrálním prvkem tvoří monoid (symbol + je neutrální prvek s hodnotou 0)
Lexikálně nadřazené prostředí	- prostředí vzniku procedury
Dynamicky nadřazené prostředí	- prostředí ze kterého byla procedura volána
Rozsah platnosti	- způsob manipulace s prostředími
Lexikální rozsah platnosti	- spočívá v hledání vazeb symbolů (které nejsou nalezeny v lokálním prostředí) v prostředí vzniku procedury
Dynamický rozsah platnosti	- spočívá v hledání vazeb symbolů (které nejsou nalezeny v lokálním prostředí) v prostředí, odkud byla procedura aplikována
Lekce 3:	**Lokální vazby a definice (str. 77)**
Metoda top-down	- styl, při kterém proramátor postupně dělí problémy na menší a a menší dokud není dosaženo požadované granularity, potom teprve fyzicky programuje (procedurální jazyky)
Metoda bottom-up	- styl, ve kterém programátor se snaží obohacovat jazyk vlastními procedurami a abstrakcemi již existujících, které pak aplikuje (funkcionální jazyky), rozvrstvení do několika vrstev; snaha vytvořit bohatý jazyk pro snadné vyřešení výchozího problému
Abstrakční bariéra	- pomyslný mezník mezi dvěma vrstvami programu; "oddělovač" vrstev procedur; např. při výpočtu kořenů kvadratické rovnice -> práce s kořeny, základní operace s dvojicemi kořenů, implementace dvojice kořenů pomocí tečkových párů, a následná implementace tečkových párů
Černá skřínka	- když začneme vytvářet vyšší vrstvy, na procedury nižší vyrstvy se díváme jako na černou skříňku, tzn. neřešíme jak fungují uvnitř (jak jsou naprogramované), ale zvenčí (zajímají nás pouze *vstupní argumenty* a *výsledky aplikace*)
Lokální vazba	- vazba vytvořená v lokálním prostředí, která po ukončení běhu procedury zaniká
Let-blok	- každý seznam ve tvaru *(let((<s1><h1>)...(<sn><hn>))<tělo>)*, umožňující přehlednější zápis -> symboly vázané let-blokem jsou uvedeny hned vedle hodnot, na které jsou navázané
Let*-blok	- každý seznam ve tvaru *(let*((<s1><h1>)...(<sn><hn>))<tělo>)*, má stejnou funkci jako let, rozdíl: vznik hierarchie prostředí, postupně je vytvářeno prostředí *P* pro každou dvojici *<si><hi>* a pro každé nové prostředí je nastaven předek
Interní definice	- lokální navázání procedury na symbol
Top-level (globální) definice	- globální navázání procedury na symbol (interní definice v *Pg)*
Lekce 4:	**Tečkové páry, symbolická data a kvotování (str. 96)**
Datová abstrakce	- rozšiřování jazyka o nové datové typy
Konstruktor	- primitivní procedura, která vytváří pár nebo seznam *(cons, list, build-list, ...)*

PAPR 1 - pojmy

Selektor	- procedura umožňující přístup k prvkům páru nebo seznamu *(car, cdr, lis-ref, ...)*
Kvotování	- proces, který vrátí zadaný argument v nevyhodnoceném tvaru
Hierarchická data	- složená data, vznikají postupným vnořováním tečkových párů do sebe, hierarchická data konstruovaná z tečkových párů nazýváme *seznamy*
Symbolická data	- data, jejichž význam jsou ony sama (nevyhodnocují se, kvotují)
Tečkové páry	- element jazyka vytvořen spojením dvou libovolných elementů do uspořádané dvojice ve tvaru *(a . b)*, je zkonstruován procedurou *cons* a procedury pro přístup k jeho jednotlivým prvkům jsou navázané na symboly *car* a *cdr*
Externí reprezentace páru	- pokud má pár první prvek *a* a druhý prvek *b*, pak jeho reprezentace (notace) bude vypadat *(a . b)*
Boxová notace	- grafické znázornění tečkových párů
Syntaktický cukr	- terminus technicus užívající se pro rozšíření zápisu programu, které je dělá snadnější pro použití (apostrof ' pro *quote,* druhý apostrof ` pro *quasiquote*, etc.)
Lekce 5:	**Seznamy (str. 114)**
Seznam Prázdný seznam	- hierarchická data konstruovaná z tečkových párů - element reprezentující seznam, který neobsahuje žádný prvek, jeho externí reprezentace je () vyhodnocuje se sám na sebe
Ocas seznamu Hlava seznamu	- pokud existuje pár ve tvaru *(E.L')* kde *E* je libovolný element a *L* seznam, pak element *E* se nazývá hlava seznamu *L,* a seznam *L′* se nazývá tělo seznamu *L* (= ocas seznamu *L*)
Kvotování seznamu	- použití speciální formy *quote* na seznam zabrání vyhodnocování jejího argumentu
Mapování procedury přes seznam	- používá se když je potřeba z daného seznamu vytvořit nový seznam stejné délky, jehož prvky jsou nějakým způsobem "změněny"
Reverze seznamu Spojování seznamů	- obrácení seznamu pomocí procedury *reverse* - operace spojení dvou a více seznamů pomocí procedury *append*
Správa paměti Gargabe collector	- interpret provádí správu paměti (paměť, ve které jsou uloženy reprezentace elementů se nazývá halda) pomocí podprogramu *Gargabe collector*, který se během vyhodnocování jednou za čas spustí, projde všechny elementy v paměti a smaže ty, které již nejsou dostupné; k "úklidu" používá algoritmus *mark & sweep* ("označ a zameť")
Algoritmus mark & sweep	1. fáze - *mark* - algoritmus projde všechny elementy a to tak, že začne těmi co mají vazbu v Pg (ty jsou vždy dosažitelné), pokud ukazují na další elementy, bude dál zpracovávat i ty (např. tečkové páry, u kterých každý pár vždy odkazuje na 1. a 2. složku) 2. fáze - *sweep* - Gargabe collector projde celou haldu a pokud má element značku, smaže ji a pokračuje dalším; pokud značku nemá, odstraní celý element z paměti
Typový systém	- každý programovací jazyk lze zhlediska použitého modelu zařadit do některé ze tří kategorií: 1. podle okamžiku, kdy je možné provést kontrolu typu elementů (staticky/dynamicky typované) 2. podle "síly" typovaného systému (silně/slabě typované) 3. podle bezpečnosti (bezpěčně/nebezpečně typované)
Silně typovaný jazyky Slabě typovaný jazyky	- mají pro každou operaci přesně vymezený datový typ argumentů a pokud je operace použita s jiným typem argumentu než je dovoleno, dochází k chybě - mají definovanou sadu pravidel pro "převod" mezi datovými typy, pokud je pak operace provedena s argumenty, které ji typově neodpovídají, tak se provede konverze typů - přetypování
Staticky typovaný jazyky Dynamicky typovaný jazyky (Scheme)	- je pro ně možné udělat kontrolu typů již před interpretací nebo během překladu programu pouze na základě znalosti jejich syntaktické struktury - u nich platí že pouhá struktura nestačí ke kontrole typů a typy elementů musí být kontrolovány až za běhu programu, zároveň platí že jedno jméno (symbol nebo proměnná) může během života programu nést hodnoty různých typů
Bezpečně typovaný jazyky Nebezpečně typovaný jazyky	- se chovají korektně zhlediska provádění operací mezi různými typy elementů, tzn. že pokud je operace proveditelná, nemůže způsobit havárii programu - nejsou bezpečně typované, výsledek mezi různými typy může vést k chybě při běhu programu (např. v jazyku C - přetečení paměti, dereference ukazuje na jiné místo v paměti, etc.)
Lekce 6:	**Explicitní aplikace a vyhodnocování (str. 141)**
Implicitní aplikace procedury Explicitní aplikace procedury	- aplikace procedury, která je důsledkem vyhodnocováním seznamů; to jest když je procedura vzniklá vyhodnocením prvního prvku seznamu aplikována s argumenty (elementy) vzniklými vyhodnocením všech dalších prvků seznamu, je prováděna implicitně během vyhodnocování elementů - aplikace procedury, která probíhá na naši žádost (pomocí *apply*); to jest máme už k dispozici argumenty dané procedury ve formě seznamu a nechceme je tedy získávat vyhodnocením
Implicitní vyhodnocení elementů Explicitní vyhodnocení elementů	- veškeré vyhodnocování doposud používané - vyhodnocování pomocí primitivní procedury *eval*
Filtrace	- *filter* je procedura vyššího řádu, provádějící filtraci elementů v seznamu podle jejich vlastnosti

PAPR 1 - pojmy

Nepovinné argumenty	uvádějí se až za všemi povinnými, seznam je ve tvaru: *(<p1>...<pn>.<zbytek>)*, kde oddělujeme poslední část <*zbytek*> tečkou a na tento symbol se naváže seznam dodatečných argumentů (pokud byly použity, v opačném případě se naváže ())
Libovolné argumenty	- UDP které mají libovolný počet argumentů vznikají vyhodnocením λ-výrazů, ve kterých je místo seznamu formálních argumentů uveden jediný symbol, na tento symbol je navázán seznam všech argumentů (libovolného počtu)
Prostředí jako element prvního řádu	- myšlenka, že samotné prostředí můžeme chápat jako element jazyka Scheme, protože abychom mohli např. aplikovat *eval*, který má první argument element k vyhodnocení a druhý argument prostředí, vyhodnocuje tedy elementy vzhledem k prostředí, musí být i prostředí element prvního řádu
Lekce 7:	**Akumulace (str. 171)**
Akumulace Explicitní aplikace	- vytvoření jedné hodnoty pomocí více hodnot obsažených v seznamu (sečtení čísel, nalezení minima, atd.), má blízko k explicitní aplikaci prováděné procedurou *apply*
Zabalení směrem doprava Zabalení směrem doleva	- *foldr* (fold right) - procedura, která je aplikována tak, že její první argument je průběžný prvek seznamu a druhý argument je výsledek zabalení prvků nacházejích se v seznamu za průběžným prvkem - *foldl* (fold left) dělá to samé, ale zleva
Lekce 8:	**Rekurze a indukce (str. 191)**
Princip indukce Princip rekurze	- dokazovací princip, jímž jsme schopni dokazovat vlastnosti rekurzivně definovaných fcí a procedur - rekurzivní procedura je taková, která ve svém těle provádí aplikaci sebe sama
Rekurzivní procedura Rekurzivní aplikace procedury Rekurzivní výpočetní proces	- pokud při vyhodnocení jejího těla dochází (v některých případech) k aplikaci sebe sama - samotná aplikace “sebe sama” - proces generovaný rekurzivní procedurou
Lineární rekurze Lineární rekurzivní proces Lineární iterativní proces	- rekurze, která volá sama sebe jen jednou - rekurzivní proces obsahující fáze naviíjení a odvíjení - rekurzivní proces který využívá iteraci
Fáze naviíjejí Fáze odvíjení	- fáze, ve které dochází k postupné rekurzivní aplikaci - nastává po dosažení limitní podmínky rekurze (podmínka, po jejímž splnění je vyhodnocen výraz jež nezpůsobí další aplikaci samotné rekurzivní procedury
Lekce 9:	**Hloubková rekurze na seznamech (str. 241)**
Metody zastavení rekurze	1. pomocí speciálních forem *if* a *cond* *(define list? (lambda (l)(if (null? l) #t (and (pair? l)(list? (cdr l))))))* 2. pomocí speciálních forem *and* a *or* *(define list? (lambda (l)(or (null? l) (and (pair? l)(list? (cdr l))))))*
Y-kombinátor	- lambda výraz ve tvaru: *(lambda (y)(y y <arg1>...<argN>))* - část programu odpovědná za aplikaci rekurzivní procedury, konkrétně za aplikaci procedury spojené s předáním prvního argumentu jímž je sama procedura
Hloubková rekurze na seznamech Linearizace seznamu	- implementace rekurzivních procedur, které při zpracování seznamu aplikují samy sebe na ty prvky seznamů, které jsou opět seznamy; je-li prvkem seznamu seznam, je na něj rekurzivně aplikována tato procedura; např. spočítání atomických prvků (prvky, které nejsou seznamy) pomocí procedury *linearize* (str. 250) - linearizace je vytvoření seznamu atomických prvků
Lekce 11:	**Kvazikvotování a manipulace se symbolickými výrazy (str. 270)**
Kvazikvotování	- v porovnání se speciální formou *quote* umožňuje navíc určit podvýrazy jejího argumentu, které budou vyhodnoceny běžným způsobem, tedy projde argument a vyhledá v něm podvýrazy, které jsou jednoprvkové seznamy začínající symbolem *unquote*, a tyto podvýrazy jsou nahrazeny vyhodnocením jejich druhého prvku
Lekce 12:	**Čistě funkcionální interpret Scheme (str. 290)**
Generické procedury	- procedury, které svoji činost řídí podle typů svých argumentů, přesněji řečeno podle těch typů provádějí aplikace jiných procedur a provádějí případně dodatečnou konverzi elementů na ty s vyžadovanými typy, např. procedura sčítání
Tabulka generických procedur	- tabulka ve speciálním tvaru určujícím vzory a procedury pro aplikaci, pokud argumenty budou odpovídat danému vzoru
Koerce, Implicitní přetypování Explicitní přetypování	- automatické přetypování na elementy jiných typů, ke kterému může docházet během používání generických procedur, obecně řečeno je koerce implicitní přetypování, které provádějí automaticky některé (generické) procedury - programátorem vynucená změna typu elementu, např. převod čísla na řětězec znaků pomocí *number->string*, na programátorovu žádost byl element “převeden”

PAPR 1 - pojmy

Manifestace typů	- každý element se skládá z dvou typů 1. identifikátor typu elementu - slouží k jednoznačnému určení o jaký element se jedná (např. pomocí něj zjišťujeme, jestli daný element reprezentuje pár nebo proceduru) 2. data charakterizující element - data představující “hodnotu elementu”
Manifestovaný typ	- přítomnost identifikátoru typu “v datech”
Visačky/Tagy	- symboly označující typy, např. argument type-tag (str. 294), visačka je tedy identifikátorem typu
Metainterpret jazyka Scheme Interpret jazyka Scheme	-již existující interpret, který používáme pro vytváření dalších programů -(meta)program pro metainterpret jazyka Scheme, který provádí interpretaci jisté podmnožiny jazyka Scheme
Metajazyk Metaelement Metaprogram Metaprocedura	- jazyk interpretovaný metainterpretem - element metajazyka - program v metajazyce - v našem interpretu budeme primitvní procedury programovat jako uživatelsky definované metaprocedury => zabalíme metaproceduru, která je protějškem dané procedury, do pomocného programu, který z daných elementů vyzvedne jejich datovou část, aplikuje metaproceduru s takto získanými hodnotami a nakonec výsledek aplikace převede do interní reprezentace